Is Parallel Programming Hard, And, If So, What Can You Do About It?

Edited by:

Paul E. McKenney
Linux Technology Center
IBM Beaverton
paulmck@linux.vnet.ibm.com

December 16, 2011
Legal Statement

This work represents the views of the authors and does not necessarily represent the view of their employers. IBM, zSeries, and PowerPC are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. Linux is a registered trademark of Linus Torvalds. i386 is a trademarks of Intel Corporation or its subsidiaries in the United States, other countries, or both. Other company, product, and service names may be trademarks or service marks of such companies.

The non-source-code text and images in this document are provided under the terms of the Creative Commons Attribution-Share Alike 3.0 United States license (http://creativecommons.org/licenses/by-sa/3.0/us/). In brief, you may use the contents of this document for any purpose, personal, commercial, or otherwise, so long as attribution to the authors is maintained. Likewise, the document may be modified, and derivative works and translations made available, so long as such modifications and derivations are offered to the public on equal terms as the non-source-code text and images in the original document.

Source code is covered by various versions of the GPL (http://www.gnu.org/licenses/gpl-2.0.html). Some of this code is GPLv2-only, as it derives from the Linux kernel, while other code is GPLv2-or-later. See the CodeSamples directory in the git archive (git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git) for the exact licenses, which are included in comment headers in each file. If you are unsure of the license for a given code fragment, you should assume GPLv2-only.

Combined work © 2005-2011 by Paul E. McKenney.
# Contents

1 Introduction 1
   1.1 Historic Parallel Programming Difficulties .......................... 1
   1.2 Parallel Programming Goals ............................................. 2
      1.2.1 Performance ....................................................... 2
      1.2.2 Productivity ...................................................... 3
      1.2.3 Generality .......................................................... 4
   1.3 Alternatives to Parallel Programming .................................. 5
      1.3.1 Multiple Instances of a Sequential Application ............... 6
      1.3.2 Make Use of Existing Parallel Software ....................... 6
      1.3.3 Performance Optimization ....................................... 6
   1.4 What Makes Parallel Programming Hard? ................................. 7
      1.4.1 Work Partitioning ................................................ 7
      1.4.2 Parallel Access Control ......................................... 8
      1.4.3 Resource Partitioning and Replication ........................ 8
      1.4.4 Interacting With Hardware ...................................... 8
      1.4.5 Composite Capabilities ......................................... 8
      1.4.6 How Do Languages and Environments Assist With These Tasks? 9
   1.5 Guide to This Book ...................................................... 9
      1.5.1 Quick Quizzes ..................................................... 9
      1.5.2 Sample Source Code ............................................. 9

2 Hardware and its Habits 11
   2.1 Overview ................................................................. 11
      2.1.1 Pipelined CPUs ..................................................... 11
      2.1.2 Memory References ............................................... 12
      2.1.3 Atomic Operations ................................................. 12
      2.1.4 Memory Barriers ................................................ 13
      2.1.5 Cache Misses ..................................................... 13
      2.1.6 I/O Operations .................................................... 13
   2.2 Overheads ............................................................... 14
      2.2.1 Hardware System Architecture .................................. 14
      2.2.2 Costs of Operations ............................................. 15
   2.3 Hardware Free Lunch? ................................................... 16
      2.3.1 3D Integration .................................................... 17
      2.3.2 Novel Materials and Processes .................................. 17
      2.3.3 Special-Purpose Accelerators ................................... 17
      2.3.4 Existing Parallel Software ....................................... 18
   2.4 Software Design Implications .......................................... 18
### 3 Tools of the Trade

#### 3.1 Scripting Languages

#### 3.2 POSIX Multiprocessing

- **3.2.1 POSIX Process Creation and Destruction**
- **3.2.2 POSIX Thread Creation and Destruction**
- **3.2.3 POSIX Locking**
- **3.2.4 POSIX Reader-Writer Locking**

#### 3.3 Atomic Operations

#### 3.4 Linux-Kernel Equivalents to POSIX Operations

#### 3.5 The Right Tool for the Job: How to Choose?

### 4 Counting

#### 4.1 Why Isn’t Concurrent Counting Trivial?

#### 4.2 Statistical Counters

- **4.2.1 Design**
- **4.2.2 Array-Based Implementation**
- **4.2.3 Eventually Consistent Implementation**
- **4.2.4 Per-Thread-Variable-Based Implementation**
- **4.2.5 Discussion**

#### 4.3 Approximate Limit Counters

- **4.3.1 Design**
- **4.3.2 Simple Limit Counter Implementation**
- **4.3.3 Simple Limit Counter Discussion**
- **4.3.4 Approximate Limit Counter Implementation**
- **4.3.5 Approximate Limit Counter Discussion**

#### 4.4 Exact Limit Counters

- **4.4.1 Atomic Limit Counter Implementation**
- **4.4.2 Atomic Limit Counter Discussion**
- **4.4.3 Signal-Theft Limit Counter Design**
- **4.4.4 Signal-Theft Limit Counter Implementation**
- **4.4.5 Signal-Theft Limit Counter Discussion**

#### 4.5 Applying Specialized Parallel Counters

#### 4.6 Parallel Counting Discussion

### 5 Partitioning and Synchronization Design

#### 5.1 Partitioning Exercises

- **5.1.1 Dining Philosophers Problem**
- **5.1.2 Double-Ended Queue**
- **5.1.3 Partitioning Example Discussion**

#### 5.2 Design Criteria

#### 5.3 Synchronization Granularity

- **5.3.1 Sequential Program**
- **5.3.2 Code Locking**
- **5.3.3 Data Locking**
- **5.3.4 Data Ownership**
- **5.3.5 Locking Granularity and Performance**

#### 5.4 Parallel Fastpath

- **5.4.1 Reader/Writer Locking**
- **5.4.2 Hierarchical Locking**
- **5.4.3 Resource Allocator Caches**
CONTENTS

5.5 Performance Summary .......................................................... 69

6 Locking ...................................................................................... 71
6.1 Staying Alive .......................................................................... 71
   6.1.1 Deadlock .......................................................................... 71
   6.1.2 Livelock and Starvation ..................................................... 76
   6.1.3 Unfairness ......................................................................... 78
   6.1.4 Inefficiency ....................................................................... 78
6.2 Types of Locks ......................................................................... 78
   6.2.1 Exclusive Locks ............................................................... 78
   6.2.2 Reader-Writer Locks ......................................................... 78
   6.2.3 Beyond Reader-Writer Locks ........................................... 79
6.3 Locking Implementation Issues .............................................. 80
   6.3.1 Sample Exclusive-Locking Implementation Based on Atomic Exchange .................................................. 80
   6.3.2 Other Exclusive-Locking Implementations ....................... 80
6.4 Lock-Based Existence Guarantees ......................................... 81
6.5 Locking: Hero or Villain? ........................................................ 82

7 Data Ownership .......................................................................... 83
7.1 Multiple Processes ................................................................. 83
7.2 Partial Data Ownership and pthreads .................................... 83
7.3 Function Shipping ................................................................. 84
7.4 Designated Thread ............................................................... 84
7.5 Privatization .......................................................................... 84
7.6 Other Uses of Data Ownership .............................................. 84

8 Deferred Processing .................................................................. 87
8.1 Reference Counting .................................................................. 87
   8.1.1 Implementation of Reference-Counting Categories .......... 88
   8.1.2 Linux Primitives Supporting Reference Counting .......... 91
   8.1.3 Counter Optimizations ..................................................... 92
8.2 Sequence Locks ....................................................................... 92
8.3 Read-Copy Update (RCU) ....................................................... 94
   8.3.1 Introduction to RCU .......................................................... 94
   8.3.2 RCU Fundamentals .......................................................... 96
   8.3.3 RCU Usage ..................................................................... 102
   8.3.4 RCU Linux-Kernel API ................................................... 110
   8.3.5 “Toy” RCU Implementations ........................................ 115
   8.3.6 RCU Exercises ............................................................... 127

9 Applying RCU ............................................................................ 129
9.1 RCU and Per-Thread-Variable-Based Statistical Counters ........ 129
   9.1.1 Design ........................................................................... 129
   9.1.2 Implementation ............................................................. 129
   9.1.3 Discussion ..................................................................... 130
9.2 RCU and Counters for Removable I/O Devices ...................... 131
10 Validation
10.1 Required Mindset .................................................. 133
10.2 Tracing ................................................................. 133
10.3 Assertions ............................................................ 134
10.4 Static Analysis ....................................................... 134
10.5 Probability and Heisenbugs ........................................ 134
  10.5.1 Statistics for Discrete Testing ................................. 134
  10.5.2 Abusing Statistics for Discrete Testing ....................... 136
  10.5.3 Statistics for Continuous Testing ............................. 136
  10.5.4 Heisenbugs and Creating Anti-Heisenbugs .................... 136
10.6 Profiling .............................................................. 136
10.7 Differential Profiling ............................................... 136
10.8 Performance Estimation ............................................ 136

11 Data Structures
11.1 Lists ........................................................................ 137
11.2 Computational Complexity and Performance ....................... 137
11.3 Design Tradeoffs ..................................................... 137
11.4 Protection .............................................................. 137
11.5 Bits and Bytes .......................................................... 137
11.6 Hardware Considerations ........................................... 137

12 Advanced Synchronization
12.1 Avoiding Locks .......................................................... 139
12.2 Memory Barriers ....................................................... 139
  12.2.1 Memory Ordering and Memory Barriers ....................... 139
  12.2.2 If B Follows A, and C Follows B, Why Doesn’t C Follow A? 140
  12.2.3 Variables Can Have More Than One Value .................... 141
  12.2.4 What Can You Trust? .............................................. 141
  12.2.5 Review of Locking Implementations ............................ 145
  12.2.6 A Few Simple Rules ............................................... 145
  12.2.7 Abstract Memory Access Model ................................. 146
  12.2.8 Device Operations ............................................... 147
  12.2.9 Guarantees .......................................................... 147
  12.2.10 What Are Memory Barriers? .................................... 148
  12.2.11 Locking Constraints ............................................. 148
  12.2.12 Memory-Barrier Examples ..................................... 154
  12.2.13 The Effects of the CPU Cache ................................. 157
  12.2.14 Where Are Memory Barriers Needed? ......................... 157
12.3 Non-Blocking Synchronization .................................... 158
  12.3.1 Simple NBS ......................................................... 158
  12.3.2 Hazard Pointers .................................................... 158
  12.3.3 Atomic Data Structures ......................................... 158
  12.3.4 “Macho” NBS ....................................................... 159

13 Ease of Use
13.1 Rusty Scale for API Design ......................................... 161
13.2 Shaving the Mandelbrot Set ........................................ 161

14 Time Management .................................................... 165
15 Conflicting Visions of the Future .................................................. 167
  15.1 The Future of CPU Technology Ain’t What it Used to Be .......... 167
    15.1.1 Uniprocessor Über Alles ........................................... 167
    15.1.2 Multithreaded Mania .............................................. 168
    15.1.3 More of the Same .................................................. 169
    15.1.4 Crash Dummies Slamming into the Memory Wall .............. 169
  15.2 Transactional Memory ..................................................... 170
    15.2.1 I/O Operations ................................................... 171
    15.2.2 RPC Operations ................................................ 171
    15.2.3 Memory-Mapping Operations .................................... 172
    15.2.4 Multithreaded Transactions .................................... 173
    15.2.5 Extra-Transactional Accesses ................................... 173
    15.2.6 Time Delays .................................................... 174
    15.2.7 Locking .......................................................... 174
    15.2.8 Reader-Writer Locking ......................................... 175
    15.2.9 Persistence ..................................................... 175
    15.2.10 Dynamic Linking and Loading .................................. 176
    15.2.11 Debugging ..................................................... 177
    15.2.12 The exec() System Call ....................................... 177
    15.2.13 RCU .......................................................... 178
    15.2.14 Discussion ..................................................... 179
  15.3 Shared-Memory Parallel Functional Programming .................. 179
  15.4 Process-Based Parallel Functional Programming ................... 179

A Important Questions .......................................................... 181
  A.1 What Does “After” Mean? .............................................. 181

B Synchronization Primitives .................................................... 185
  B.1 Organization and Initialization ....................................... 185
    B.1.1 smp_init(): .................................................... 185
  B.2 Thread Creation, Destruction, and Control ......................... 185
    B.2.1 create_thread() ................................................ 186
    B.2.2 smp_thread_id() .............................................. 186
    B.2.3 for_each_thread() .............................................. 186
    B.2.4 for_each_running_thread() ................................... 186
    B.2.5 wait_thread() ................................................ 186
    B.2.6 wait_all_threads() ............................................ 186
    B.2.7 Example Usage ................................................ 186
  B.3 Locking ........................................................................ 187
    B.3.1 spin_lock_init() ................................................ 187
    B.3.2 spin_lock() ...................................................... 187
    B.3.3 spin_trylock() .................................................. 187
    B.3.4 spin_unlock() .................................................. 187
    B.3.5 Example Usage ................................................ 187
  B.4 Per-Thread Variables ...................................................... 187
    B.4.1 DEFINE_PER_THREAD() ......................................... 188
    B.4.2 DECLARE_PER_THREAD() ....................................... 188
    B.4.3 per_thread() ..................................................... 188
    B.4.4 __get_thread_var() ............................................ 188
    B.4.5 init_per_thread() ................................................ 188
D.2.7 Use Cases .......................................................... 222
D.2.8 Testing ............................................................. 225
D.2.9 Conclusion ......................................................... 228
D.3 Hierarchical RCU Code Walkthrough ........................................ 228
D.3.1 Data Structures and Kernel Parameters .................................. 229
D.3.2 External Interfaces .................................................. 234
D.3.3 Initialization ......................................................... 239
D.3.4 CPU Hotplug ........................................................ 242
D.3.5 Miscellaneous Functions .............................................. 245
D.3.6 Grace-Period-Detection Functions .................................... 245
D.3.7 Dyntick-Idle Functions .............................................. 251
D.3.8 Forcing Quiescent States ............................................ 255
D.3.9 CPU-Stall Detection .................................................. 259
D.3.10 Possible Flaws and Changes ........................................ 261
D.4 Preemptible RCU ......................................................... 261
D.4.1 Conceptual RCU ..................................................... 261
D.4.2 Overview of Preemptible RCU Algorithm .................................. 262
D.4.3 Validation of Preemptible RCU ....................................... 271
E Read-Copy Update in Linux .................................................. 275
E.1 RCU Usage Within Linux ................................................ 275
E.2 RCU Evolution .......................................................... 275
E.2.1 2.6.27 Linux Kernel ............................................... 275
E.2.2 2.6.28 Linux Kernel ............................................... 276
E.2.3 2.6.29 Linux Kernel ............................................... 276
E.2.4 2.6.31 Linux Kernel ............................................... 277
E.2.5 2.6.32 Linux Kernel ............................................... 277
E.2.6 2.6.33 Linux Kernel ............................................... 277
E.2.7 2.6.34 Linux Kernel ............................................... 277
E.2.8 2.6.35 Linux Kernel ............................................... 277
E.2.9 2.6.36 Linux Kernel ............................................... 278
E.2.10 2.6.37 Linux Kernel ............................................... 278
E.2.11 2.6.38 Linux Kernel ............................................... 278
E.2.12 2.6.39 Linux Kernel ............................................... 278
E.2.13 What Comes After 2.6.39? ......................................... 278
F Formal Verification ......................................................... 281
F.1 What are Promela and Spin? .............................................. 281
F.2 Promela Example: Non-Atomic Increment .................................... 281
F.3 Promela Example: Atomic Increment ..................................... 284
F.3.1 Combinatorial Explosion ............................................. 284
F.4 How to Use Promela ..................................................... 284
F.4.1 Promela Peculiarities .............................................. 285
F.4.2 Promela Coding Tricks ............................................. 286
F.5 Promela Example: Locking .............................................. 286
F.6 Promela Example: QRCU ............................................... 287
F.6.1 Running the QRCU Example ....................................... 290
F.6.2 How Many Readers and Updaters Are Really Needed? .................. 290
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>F.6.5 Alternative Approach: Divide and Conquer</td>
<td>291</td>
</tr>
<tr>
<td>F.7 Promela Parable: dynticks and Preemptible RCU</td>
<td>291</td>
</tr>
<tr>
<td>F.7.1 Introduction to Preemptible RCU and dynticks</td>
<td>292</td>
</tr>
<tr>
<td>F.7.2 Validating Preemptible RCU and dynticks</td>
<td>294</td>
</tr>
<tr>
<td>F.7.3 Lessons (Re)Learned</td>
<td>303</td>
</tr>
<tr>
<td>F.8 Simplicity Avoids Formal Verification</td>
<td>303</td>
</tr>
<tr>
<td>F.8.1 State Variables for Simplified Dynticks Interface</td>
<td>304</td>
</tr>
<tr>
<td>F.8.2 Entering and Leaving Dynticks-Idle Mode</td>
<td>304</td>
</tr>
<tr>
<td>F.8.3 NMI From Dynticks-Idle Mode</td>
<td>305</td>
</tr>
<tr>
<td>F.8.4 Interrupts From Dynticks-Idle Mode</td>
<td>305</td>
</tr>
<tr>
<td>F.8.5 Checking For Dynticks Quiescent States</td>
<td>306</td>
</tr>
<tr>
<td>F.8.6 Discussion</td>
<td>306</td>
</tr>
<tr>
<td>F.9 Summary</td>
<td>307</td>
</tr>
<tr>
<td>G Answers to Quick Quizzes</td>
<td>309</td>
</tr>
<tr>
<td>G.1 Introduction</td>
<td>309</td>
</tr>
<tr>
<td>G.2 Hardware and its Habits</td>
<td>313</td>
</tr>
<tr>
<td>G.3 Tools of the Trade</td>
<td>315</td>
</tr>
<tr>
<td>G.4 Counting</td>
<td>319</td>
</tr>
<tr>
<td>G.5 Partitioning and Synchronization Design</td>
<td>329</td>
</tr>
<tr>
<td>G.6 Locking</td>
<td>332</td>
</tr>
<tr>
<td>G.7 Data Ownership</td>
<td>336</td>
</tr>
<tr>
<td>G.8 Deferred Processing</td>
<td>338</td>
</tr>
<tr>
<td>G.9 Applying RCU</td>
<td>352</td>
</tr>
<tr>
<td>G.10 Validation</td>
<td>353</td>
</tr>
<tr>
<td>G.11 Advanced Synchronization</td>
<td>355</td>
</tr>
<tr>
<td>G.12 Ease of Use</td>
<td>357</td>
</tr>
<tr>
<td>G.13 Conflicting Visions of the Future</td>
<td>357</td>
</tr>
<tr>
<td>G.14 Important Questions</td>
<td>358</td>
</tr>
<tr>
<td>G.15 Synchronization Primitives</td>
<td>358</td>
</tr>
<tr>
<td>G.16 Why Memory Barriers?</td>
<td>359</td>
</tr>
<tr>
<td>G.17 Read-Copy Update Implementations</td>
<td>361</td>
</tr>
<tr>
<td>G.18 Formal Verification</td>
<td>373</td>
</tr>
<tr>
<td>H Glossary</td>
<td>379</td>
</tr>
<tr>
<td>I Credits</td>
<td>399</td>
</tr>
<tr>
<td>I.1 Authors</td>
<td>399</td>
</tr>
<tr>
<td>I.2 Reviewers</td>
<td>399</td>
</tr>
<tr>
<td>I.3 Machine Owners</td>
<td>399</td>
</tr>
<tr>
<td>I.4 Original Publications</td>
<td>400</td>
</tr>
<tr>
<td>I.5 Figure Credits</td>
<td>400</td>
</tr>
<tr>
<td>I.6 Other Support</td>
<td>401</td>
</tr>
</tbody>
</table>
Preface

The purpose of this book is to help you understand how to program shared-memory parallel machines without risking your sanity. By describing the algorithms and designs that have worked well in the past, we hope to help you avoid at least some of the pitfalls that have beset parallel projects. But you should think of this book as a foundation on which to build, rather than as a completed cathedral. Your mission, if you choose to accept, is to help make further progress in the exciting field of parallel programming, progress that should in time render this book obsolete. Parallel programming is not as hard as it is reputed, and it is hoped that this book makes it even easier for you.

This book follows a watershed shift in the parallel-programming field, from being primarily the domain of science, research, and grand-challenge projects to being primarily an engineering discipline. In presenting this engineering discipline, this book will examine the specific development tasks peculiar to parallel programming, and describe how they may be most effectively handled, and, in some surprisingly common special cases, automated.

This book is written in the hope that presenting the engineering discipline underlying successful parallel-programming projects will free a new generation of parallel hackers from the need to slowly and painstakingly reinvent old wheels, instead focusing their energy and creativity on new frontiers. Although the book is intended primarily for self-study, it is likely to be more generally useful. It is hoped that this book will be useful to you, and that the experience of parallel programming will bring you as much fun, excitement, and challenge as it has provided the authors over the years.

---

1 Or, perhaps more accurately, without much greater risk to your sanity than that incurred by non-parallel programming. Which, come to think of it, might not be saying all that much. Either way, Appendix A discusses some important questions whose answers are less intuitive in parallel programs than in sequential programs.
Chapter 1

Introduction

Parallel programming has earned a reputation as one of the most difficult areas a hacker can tackle. Papers and textbooks warn of the perils of deadlock, livelock, race conditions, non-determinism, Amdahl’s-Law limits to scaling, and excessive realtime latencies. And these perils are quite real; we authors have accumulated uncounted years of experience dealing with them, and all of the emotional scars, grey hairs, and hair loss that go with such an experience.

However, new technologies have always been difficult to use at introduction, but have invariably become easier over time. For example, there was a time when the ability to drive a car was a rare skill, but in many developed countries, this skill is now commonplace. This dramatic change came about for two basic reasons: (1) cars became cheaper and more readily available, so that more people had the opportunity to learn to drive, and (2) cars became simpler to operate, due to automatic transmissions, automatic chokes, automatic starters, greatly improved reliability, and a host of other technological improvements.

The same is true of a host of other technologies, including computers. It is no longer necessary to operate a keypunch in order to program. Spreadsheets allow most non-programmers to get results from their computers that would have required a team of specialists a few decades ago. Perhaps the most compelling example is web-surfing and content creation, which since the early 2000s has been easily done by untrained, uneducated people using various now-commonplace social-networking tools. As recently as 1968, such content creation was a far-out research project [Eng68], described at the time as “like a UFO landing on the White House lawn”[Gri00].

Therefore, if you wish to argue that parallel programming will remain as difficult as it is currently perceived by many to be, it is you who bears the burden of proof, keeping in mind the many centuries of counter-examples in a variety of fields of endeavor.

1.1 Historic Parallel Programming Difficulties

As indicated by its title, this book takes a different approach. Rather than complain about the difficulty of parallel programming, it instead examines the reasons why parallel programming is difficult, and then works to help the reader to overcome these difficulties. As will be seen, these difficulties have fallen into several categories, including:

1. The historic high cost and relative rarity of parallel systems.
2. The typical researcher’s and practitioner’s lack of experience with parallel systems.
3. The paucity of publicly accessible parallel code.
4. The lack of a widely understood engineering discipline of parallel programming.
5. The high cost of communication relative to that of processing, even in tightly coupled shared-memory computers.

Many of these historic difficulties are well on the way to being overcome. First, over the past few decades, the cost of parallel systems has decreased from many multiples of that of a house to a fraction of that of a used car, thanks to the advent of multicore systems. Papers calling out the advantages of multicore CPUs were published as early as 1996 [ONH‘96], IBM introduced simultaneous multi-threading into its high-end POWER family in 2000, and

Second, the advent of low-cost and readily available multicore systems means that the once-rare experience of parallel programming is now available to almost all researchers and practitioners. In fact, parallel systems are now well within the budget of students and hobbyists. We can therefore expect greatly increased levels of invention and innovation surrounding parallel systems, and that increased familiarity will over time make once-forbidding field of parallel programming much more friendly and commonplace.

Third, where in the 20th century, large systems of highly parallel software were almost always closely guarded proprietary secrets, the 21st century has seen numerous open-source (and thus publicly available) parallel software projects, including the Linux kernel [Tor03c], database systems [Pos08, MS08], and message-passing systems [The08, UoC08]. This book will draw primarily from the Linux kernel, but will provide much material suitable for user-level applications.

Fourth, even though the large-scale parallel-programming projects of the 1980s and 1990s were almost all proprietary projects, these projects have seeded the community with a cadre of developers who understand the engineering discipline required to develop production-quality parallel code. A major purpose of this book is to present this engineering discipline.

Unfortunately, the fifth difficulty, the high cost of communication relative to that of processing, remains largely in force. Although this difficulty has been receiving increasing attention during the new millennium, according to Stephen Hawking, the finite speed of light and the atomic nature of matter is likely to limit progress in this area [Gar07, Moo03]. Fortunately, this difficulty has been in force since the late 1980s, so that the aforementioned engineering discipline has evolved practical and effective strategies for handling it. In addition, hardware designers are increasingly aware of these issues, so perhaps future hardware will be more friendly to parallel software as discussed in Section 2.3.

Quick Quiz 1.1: Come on now!!! Parallel programming has been known to be exceedingly hard for many decades. You seem to be hinting that it is not so hard. What sort of game are you playing?

However, even though parallel programming might not be as hard as is commonly advertised, it is often more work than is sequential programming.

Quick Quiz 1.2: How could parallel programming ever be as easy as sequential programming?

It therefore makes sense to consider alternatives to parallel programming. However, it is not possible to reasonably consider parallel-programming alternatives without understanding parallel-programming goals. This topic is addressed in the next section.

1.2 Parallel Programming Goals

The three major goals of parallel programming (over and above those of sequential programming) are as follows:

1. Performance.
2. Productivity.

Quick Quiz 1.3: Oh, really?? What about correctness, maintainability, robustness, and so on?

Quick Quiz 1.4: And if correctness, maintainability, and robustness don’t make the list, why do productivity and generality?

Quick Quiz 1.5: Given that parallel programs are much harder to prove correct than are sequential programs, again, shouldn’t correctness really be on the list?

Quick Quiz 1.6: What about just having fun?

Each of these goals is elaborated upon in the following sections.

1.2.1 Performance

Performance is the primary goal behind most parallel-programming effort. After all, if performance is not a concern, why not do yourself a favor, just write sequential code, and be happy? It will very likely be easier, and you will probably get done much more quickly.

Quick Quiz 1.7: Are there no cases where parallel programming is about something other than performance?

Note that “performance” is interpreted quite broadly here, including scalability (performance per CPU) and efficiency (for example, performance per watt).
That said, the focus of performance has shifted from hardware to parallel software. This change in focus is due to the fact that although Moore’s Law continues to deliver increases in transistor density, it has ceased to provide the traditional single-threaded performance increases, as can be seen in Figure 1.1.\footnote{This plot shows clock frequencies for newer CPUs theoretically capable of retiring one or more instructions per clock, and MIPS for older CPUs requiring multiple clocks to execute even the simplest instruction. The reason for taking this approach is that the newer CPUs’ ability to retire multiple instructions per clock is typically limited by memory-system performance.} This means that writing single-threaded code and simply waiting a year or two for the CPUs to catch up may no longer be an option. Given the recent trends on the part of all major manufacturers towards multicore/multithreaded systems, parallelism is the way to go for those wanting the avail themselves of the full performance of their systems.

Even so, the first goal is performance rather than scalability, especially given that the easiest way to attain linear scalability is to reduce the performance of each CPU [Tor01]. Given a four-CPU system, which would you prefer? A program that provides 100 transactions per second on a single CPU, but does not scale at all? Or a program that provides 10 transactions per second on a single CPU, but scales perfectly? The first program seems like a better bet, though the answer might change if you happened to be one of the lucky few with access to a 32-CPU system.

That said, just because you have multiple CPUs is not necessarily in and of itself a reason to use them all, especially given the recent decreases in price of multi-CPU systems. The key point to understand is that parallel programming is primarily a performance optimization, and, as such, it is one potential optimization of many. If your program is fast enough as currently written, there is no reason to optimize, either by parallelizing it or by applying any of a number of potential sequential optimizations.\footnote{Of course, if you are a hobbyist whose primary interest is writing parallel software, that is more than enough reason to parallelize whatever software you are interested in.} By the same token, if you are looking to apply parallelism as an optimization to a sequential program, then you will need to compare parallel algorithms to the best sequential algorithms. This may require some care, as far too many publications ignore the sequential case when analyzing the performance of parallel algorithms.

1.2.2 Productivity

Quick Quiz 1.8: Why all this prattling on about non-technical issues??? And not just any non-technical issue, but productivity of all things? Who cares? ■

Productivity has been becoming increasingly important through the decades. To see this, consider that early computers cost millions of dollars at a time when engineering salaries were a few thousand dollars a year. If dedicating a team of ten engineers to such a machine would improve its performance by 10%, their salaries would be repaid many times over.

One such machine was the CSIRAC, the oldest still-intact stored-program computer, put in operation in 1949 [Mus04, Mel06]. Given that the machine had but 768 words of RAM, it is safe to say that the productivity issues that arise in large-scale software projects were not an issue for this machine. Because this machine was built before the transistor era, it was constructed of 2,000 vacuum tubes, ran with a clock frequency of 1kHz, consumed 30kW of power, and weighed more than three metric tons. It would be difficult to purchase a machine with this little compute power roughly sixty years later (2008), with the closest equivalents being 8-bit embedded microprocessors exemplified by the venerable Z80 [Wik08]. This CPU had 8,500 transistors, and can still be purchased in 2008 for less than $2 US per unit in 1,000-unit quantities. In stark contrast to the CSIRAC, software-development costs are anything but insignificant for the Z80.

The CSIRAC and the Z80 are two points in a long-term
CHAPTER 1. INTRODUCTION

0.1

1

10

100

1000

10000

100000

1975

1980

1985

1990

1995

2000

2005

2010

2015

MIPS per Die

Year

Figure 1.2: MIPS per Die for Intel CPUs

trend, as can be seen in Figure 1.2. This figure plots an approximation to computational power per die over the past three decades, showing a consistent four-order-of-magnitude increase. Note that the advent of multicore CPUs has permitted this increase to continue unabated despite the clock-frequency wall encountered in 2003.

One of the inescapable consequences of the rapid decrease in the cost of hardware is that software productivity grows increasingly important. It is no longer sufficient merely to make efficient use of the hardware, it is now also necessary to make extremely efficient use of software developers. This has long been the case for sequential hardware, but only recently has parallel hardware become a low-cost commodity. Therefore, the need for high productivity in creating parallel software has only recently become hugely important.

Quick Quiz 1.9: Given how cheap parallel hardware has become, how can anyone afford to pay people to program it?

Perhaps at one time, the sole purpose of parallel software was performance. Now, however, productivity is increasingly important.

1.2.3 Generality

One way to justify the high cost of developing parallel software is to strive for maximal generality. All else being equal, the cost of a more-general software artifact can be spread over more users than can a less-general artifact.

Unfortunately, generality often comes at the cost of performance, productivity, or both. To see this, consider the following popular parallel programming environments:

C/C++ “Locking Plus Threads”: This category, which includes POSIX Threads (pthreads) [Ope97], Windows Threads, and numerous operating-system kernel environments, offers excellent performance (at least within the confines of a single SMP system) and also offers good generality. Pity about the relatively low productivity.

Java: This programming environment, which is inherently multithreaded, is widely believed to be much more productive than C or C++, courtesy of the automatic garbage collector and the rich set of class libraries, and is reasonably general purpose. However, its performance, though greatly improved over the past ten years, is generally considered to be less than that of C and C++.

MPI: This Message Passing Interface [MPI08] powers the largest scientific and technical computing clusters in the world, so offers unparalleled performance and scalability. It is in theory general purpose, but has generally been used for scientific and technical computing. Its productivity is believed by many to be even less than that of C/C++ “locking plus threads” environments.

OpenMP: This set of compiler directives can be used to parallelize loops. It is thus quite specific to this task, and this specificity often limits its performance. It is, however, much easier to use than MPI or parallel C/C++.

SQL: Structured Query Language [Int92] is extremely specific, applying only to relational database queries. However, its performance is quite good, doing quite well in Transaction Processing Performance Council (TPC) benchmarks [Tra01]. Productivity is excellent, in fact, this parallel programming environment permits people who know almost nothing about parallel programming to make good use of a large parallel machine.

The nirvana of parallel programming environments, one that offers world-class performance, productivity, and generality, simply does not yet exist. Until such a nirvana appears, it will be necessary to make engineering tradeoffs among performance, productivity, and generality. One such tradeoff is shown in Figure 1.3, which
shows how productivity becomes increasingly important at the upper layers of the system stack, while performance and generality become increasingly important at the lower layers of the system stack. The huge development costs incurred near the bottom of the stack must be spread over equally huge numbers of users on the one hand (hence the importance of generality), and performance lost near the bottom of the stack cannot easily be recovered further up the stack. Near the top of the stack, there might be very few users for a given specific application, in which case productivity concerns are paramount. This explains the tendency towards “bloatware” further up the stack: extra hardware is often cheaper than would be the extra developers. This book is intended primarily for developers working near the bottom of the stack, where performance and generality are paramount concerns.

It is important to note that a tradeoff between productivity and generality has existed for centuries in many fields. For but one example, a nailgun is far more productive than is a hammer, but in contrast to the nailgun, a hammer can be used for many things besides driving nails. It should therefore be absolutely no surprise to see similar tradeoffs appear in the field of parallel computing. This tradeoff is shown schematically in Figure 1.4. Here, users 1, 2, 3, and 4 have specific jobs that they need the computer to help them with. The most productive possible language or environment for a given user is one that simply does that user’s job, without requiring any programming, configuration, or other setup.

Quick Quiz 1.10: This is a ridiculously unachievable ideal! Why not focus on something that is achievable in practice?

Unfortunately, a system that does the job required by user 1 is unlikely to do user 2’s job. In other words, the most productive languages and environments are domain-specific, and thus by definition lacking generality.

Another option is to tailor a given programming language or environment to the hardware system (for example, low-level languages such as assembly, C, C++, or Java) or to some abstraction (for example, Haskell, Prolog, or Snobol), as is shown by the circular region near the center of Figure 1.4. These languages can be considered to be general in the sense that they are equally ill-suited to the jobs required by users 1, 2, 3, and 4. In other words, their generality is purchased at the expense of decreased productivity when compared to domain-specific languages and environments.

With the three often-conflicting parallel-programming goals of performance, productivity, and generality in mind, it is now time to look into avoiding these conflicts by considering alternatives to parallel programming.

1.3 Alternatives to Parallel Programming

In order to properly consider alternatives to parallel programming, you must first have thought through what you expect the parallelism to do for you. As seen in Section 1.2, the primary goals of parallel programming are performance, productivity, and generality.

Although historically most parallel developers might
be most concerned with the first goal, one advantage of the other goals is that they relieve you of the need to justify using parallelism. The remainder of this section is concerned only performance improvement.

It is important to keep in mind that parallelism is but one way to improve performance. Other well-known approaches include the following, in roughly increasing order of difficulty:

1. Run multiple instances of a sequential application.
2. Construct the application to make use of existing parallel software.
3. Apply performance optimization to the serial application.

### 1.3.1 Multiple Instances of a Sequential Application

Running multiple instances of a sequential application can allow you to do parallel programming without actually doing parallel programming. There are a large number of ways to approach this, depending on the structure of the application.

If your program is analyzing a large number of different scenarios, or is analyzing a large number of independent data sets, one easy and effective approach is to create a single sequential program that carries out a single analysis, then use any of a number of scripting environments (for example the 
bash shell) to run a number of instances of this sequential program in parallel. In some cases, this approach can be easily extended to a cluster of machines.

This approach may seem like cheating, and in fact some denigrate such programs as “embarrassingly parallel”. And in fact, this approach does have some potential disadvantages, including increased memory consumption, waste of CPU cycles recomputing common intermediate results, and increased copying of data. However, it is often extremely effective, garnering extreme performance gains with little or no added effort.

### 1.3.2 Make Use of Existing Parallel Software

There is no longer any shortage of parallel software environments that can present a single-threaded programming environment, including relational databases, web-application servers, and map-reduce environments. For example, a common design provides a separate program for each user, each of which generates SQL that is run concurrently against a common relational database. The per-user programs are responsible only for the user interface, with the relational database taking full responsibility for the difficult issues surrounding parallelism and persistence.

Taking this approach often sacrifices some performance, at least when compared to carefully hand-coding a fully parallel application. However, such sacrifice is often justified given the great reduction in development effort required.

### 1.3.3 Performance Optimization

Up through the early 2000s, CPU performance was doubling every 18 months. In such an environment, it is often much more important to create new functionality than to do careful performance optimization. Now that Moore’s Law is “only” increasing transistor density instead of increasing both transistor density and per-transistor performance, it might be a good time to rethink the importance of performance optimization.

After all, performance optimization can reduce power consumption as well as increasing performance.

From this viewpoint, parallel programming is but another performance optimization, albeit one that is becoming much more attractive as parallel systems become cheaper and more readily available. However, it is wise to keep in mind that the speedup available from parallelism is limited to roughly the number of CPUs, while the speedup potentially available from straight software optimization can be multiple orders of magnitude.

Furthermore, different programs might have different performance bottlenecks. Parallel programming will only help with some bottlenecks. For example, suppose that your program spends most of its time waiting on data from your disk drive. In this case, making your program use multiple CPUs is not likely to gain much performance. In fact, if the program was reading from a large file laid out sequentially on a rotating disk, parallelizing your program might well make it a lot slower. You should instead add more disk drives, optimize the data so that the file can be smaller (thus faster to read), or, if possible, avoid the need to read quite so much of the data.

**Quick Quiz 1.11:** What other bottlenecks might prevent additional CPUs from providing additional performance?

Parallelism can be a powerful optimization technique, but it is not the only such technique, nor is it appropriate...
for all situations. Of course, the easier it is to parallelize your program, the more attractive parallelization becomes as an optimization. Parallelization has a reputation of being quite difficult, which leads to the question “exactly what makes parallel programming so difficult?”

### 1.4 What Makes Parallel Programming Hard?

It is important to note that the difficulty of parallel programming is as much a human-factors issue as it is a set of technical properties of the parallel programming problem. This is the case because we need human beings to be able to tell parallel systems what to do, and this two-way communication between human and computer is as much a function of the human as it is of the computer. Therefore, appeals to abstractions or to mathematical analyses will necessarily be of severely limited utility.

In the Industrial Revolution, the interface between human and machine was evaluated by human-factor studies, then called time-and-motion studies. Although there have been a few human-factor studies examining parallel programming [ENS05, ES05, HCS05, SS94], these studies have been extremely narrowly focused, and hence unable to demonstrate any general results. Furthermore, given that the normal range of programmer productivity spans more than an order of magnitude, it is unrealistic to expect an affordable study to be capable of detecting (say) a 10% difference in productivity. Although the multiple-order-of-magnitude differences that such studies can reliably detect are extremely valuable, the most impressive improvements tend to be based on a long series of 10% improvements.

We must therefore take a different approach.

One such approach is to carefully consider the tasks that parallel programmers must undertake that are not required of sequential programmers. We can then evaluate how well a given programming language or environment assists the developer with these tasks. These tasks fall into the four categories shown in Figure 1.5, each of which is covered in the following sections.

#### 1.4.1 Work Partitioning

Work partitioning is absolutely required for parallel execution: if there is but one “glob” of work, then it can be executed by at most one CPU at a time, which is by definition sequential execution. However, partitioning the code

![Diagram](image-url)
might have extremely large state spaces, but nevertheless be easy to understand due to their regular structure, while poor designs can be impenetrable despite having a comparatively small state space. The best designs exploit embarrassing parallelism, or transform the problem to one having an embarrassingly parallel solution. In either case, “embarrassingly parallel” is in fact an embarrassment of riches. The current state of the art enumerates good designs; more work is required to make more general judgements on state-space size and structure.

### 1.4.2 Parallel Access Control

Given a sequential program with only a single thread, that single thread has full access to all of the program’s resources. These resources are most often in-memory data structures, but can be CPUs, memory (including caches), I/O devices, computational accelerators, files, and much else besides.

The first parallel-access-control issue is whether the form of the access to a given resource depends on that resource’s location. For example, in many message-passing environments, local-variable access is via expressions and assignments, while remote-variable access uses an entirely different syntax, usually involving messaging. The POSIX Threads environment [Ope97], Structured Query Language (SQL) [Int92], and partitioned global address-space (PGAS) environments such as Universal Parallel C (UPC) [EGCD03] offer implicit access, while Message Passing Interface (MPI) [MPI08] offers explicit access because access to remote data requires explicit messaging.

The other parallel-access-control issue is how threads coordinate access to the resources. This coordination is carried out by the very large number of synchronization mechanisms provided by various parallel languages and environments, including message passing, locking, transactions, reference counting, explicit timing, shared atomic variables, and data ownership. Many traditional parallel-programming concerns such as deadlock, livelock, and transaction rollback stem from this coordination. This framework can be elaborated to include comparisons of these synchronization mechanisms, for example locking vs. transactional memory [MMW07], but such elaboration is beyond the scope of this section.

### 1.4.3 Resource Partitioning and Replication

The most effective parallel algorithms and systems exploit resource parallelism, so much so that it is usually wise to begin parallelization by partitioning your write-intensive resources and replicating frequently accessed read-mostly resources. The resource in question is most frequently data, which might be partitioned over computer systems, mass-storage devices, NUMA nodes, CPU cores (or dies or hardware threads), pages, cache lines, instances of synchronization primitives, or critical sections of code. For example, partitioning over locking primitives is termed “data locking” [BK85].

Resource partitioning is frequently application dependent, for example, numerical applications frequently partition matrices by row, column, or sub-matrix, while commercial applications frequently partition write-intensive data structures and replicate read-mostly data structures. For example, a commercial application might assign the data for a given customer to a given few computer systems out of a large cluster. An application might statically partition data, or dynamically change the partitioning over time.

Resource partitioning is extremely effective, but it can be quite challenging for complex multilinked data structures.

### 1.4.4 Interacting With Hardware

Hardware interaction is normally the domain of the operating system, the compiler, libraries, or other software-environment infrastructure. However, developers working with novel hardware features and components will often need to work directly with such hardware. In addition, direct access to the hardware can be required when squeezing the last drop of performance out of a given system. In this case, the developer may need to tailor or configure the application to the cache geometry, system topology, or interconnect protocol of the target hardware.

In some cases, hardware may be considered to be a resource which may be subject to partitioning or access control, as described in the previous sections.

### 1.4.5 Composite Capabilities

Although these four capabilities are fundamental, good engineering practice uses composites of these capabilities. For example, the data-parallel approach first partitions the data so as to minimize the need for inter-partition
1.5 Guide to This Book

This book is not a collection of optimal algorithms with tiny areas of applicability; instead, it is a handbook of widely applicable and heavily used techniques. We of course could not resist the urge to include some of our favorites that have not (yet!) passed the test of time (what author could?), but we have nonetheless gritted our teeth and banished our darlings to appendices. Perhaps in time, some of them will see enough use that we can promote them into the main body of the text.

1.5.1 Quick Quizzes

“Quick quizzes” appear throughout this book. Some of these quizzes are based on material in which that quick quiz appears, but others require you to think beyond that section, and, in some cases, beyond the entire book. As with most endeavors, what you get out of this book is largely determined by what you are willing to put into it. Therefore, readers who invest some time into these quizzes will find their effort repaid handsomely with increased understanding of parallel programming.

Answers to the quizzes may be found in Appendix G starting on page 309.

Quick Quiz 1.14: Where are the answers to the Quick Quizzes found? ■

Quick Quiz 1.15: Some of the Quick Quiz questions seem to be from the viewpoint of the reader rather than the author. Is that really the intent? ■

Quick Quiz 1.16: These Quick Quizzes just are not my cup of tea. What do you recommend? ■

1.5.2 Sample Source Code

This book discusses its fair share of source code, and in many cases this source code may be found in the CodeSamples directory of this book’s git tree. For example, on UNIX systems, you should be able to type:

```
fund CodeSamples -name rcu_rcpls.c -print
```

to locate the file rcu_rcpls.c, which is called out in Section 8.3.5. Other types of systems have well-known ways of locating files by filename.

The source to this book may be found in the git archive at git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git, and git itself is available as part of most mainstream Linux distributions. PDFs of this book are sporadically posted at http://kernel.org/pub/
linux/kernel/people/paulmck/perfbook/perfbook.html.
Chapter 2

Hardware and its Habits

Most people have an intuitive understanding that passing messages between systems is considerably more expensive than performing simple calculations within the confines of a single system. However, it is not always so clear that communicating among threads within the confines of a single shared-memory system can also be quite expensive. This chapter therefore looks at the cost of synchronization and communication within a shared-memory system. This chapter merely scratches the surface of shared-memory parallel hardware design; readers desiring more detail would do well to start with a recent edition of Hennessy and Patterson’s classic text [HP95].

Quick Quiz 2.1: Why should parallel programmers bother learning low-level properties of the hardware? Wouldn’t it be easier, better, and more general to remain at a higher level of abstraction?

2.1 Overview

Careless reading of computer-system specification sheets might lead one to believe that CPU performance is a footrace on a clear track, as illustrated in Figure 2.1, where the race always goes to the swiftest.

Although there are a few CPU-bound benchmarks that approach the ideal shown in Figure 2.1, the typical program more closely resembles an obstacle course than a race track. This is because the internal architecture of CPUs has changed dramatically over the past few decades, courtesy of Moore’s Law. These changes are described in the following sections.

2.1.1 Pipelined CPUs

In the early 1980s, the typical microprocessor fetched an instruction, decoded it, and executed it, typically taking at least three clock cycles to complete one instruction before proceeding to the next. In contrast, the CPU of the late 1990s and early 2000s will be executing many instructions simultaneously, using a deep “pipeline” to control the flow of instructions internally to the CPU, this difference being illustrated by Figure 2.2.

Achieving full performance with a CPU having a long pipeline requires highly predictable control flow through the program. Suitable control flow can be provided by a program that executes primarily in tight loops, for example, programs doing arithmetic on large matrices or vectors. The CPU can then correctly predict that the branch at the end of the loop will be taken in almost all cases. In such programs, the pipeline can be kept full and...
CHAPTER 2. HARDWARE AND ITS HABITS

If, on the other hand, the program has many loops with small loop counts, or if the program is object oriented with many virtual objects that can reference many different real objects, all with different implementations for frequently invoked member functions, then it is difficult or even impossible for the CPU to predict where a given branch might lead. The CPU must then either stall waiting for execution to proceed far enough to know for certain where the branch will lead, or guess — and, in the face of programs with unpredictable control flow, frequently guess wrong. In either case, the pipeline will empty and have to be refilled, leading to stalls that can drastically reduce performance, as fancifully depicted in Figure 2.3.

Unfortunately, pipeline flushes are not the only hazards in the obstacle course that modern CPUs must run. The next section covers the hazards of referencing memory.

2.1.2 Memory References

In the 1980s, it often took less time for a microprocessor to load a value from memory than it did to execute an instruction. In 2006, a microprocessor might be capable of executing hundreds or even thousands of instructions in the time required to access memory. This disparity is due to the fact that Moore’s Law has increased CPU performance at a much greater rate than it has increased memory performance, in part due to the rate at which memory sizes have grown. For example, a typical 1970s minicomputer might have 4KB (yes, kilobytes, not megabytes, let alone gigabytes) of main memory, with single-cycle access. In 2008, CPU designers still can construct a 4KB memory with single-cycle access, even on systems with multi-GHz clock frequencies. And in fact they frequently do construct such memories, but they now call them “level-0 caches”.

Although the large caches found on modern microprocessors can do quite a bit to help combat memory-access latencies, these caches require highly predictable data-access patterns to successfully hide memory latencies. Unfortunately, common operations, such as traversing a linked list, have extremely unpredictable memory-access patterns — after all, if the pattern was predictable, us software types would not bother with the pointers, right?

Therefore, as shown in Figure 2.4, memory references are often severe obstacles for modern CPUs.

Thus far, we have only been considering obstacles that can arise during a given CPU’s execution of single-threaded code. Multi-threading presents additional obstacles to the CPU, as described in the following sections.

2.1.3 Atomic Operations

One such obstacle is atomic operations. The whole idea of an atomic operation in some sense conflicts with the piece-at-a-time assembly-line operation of a CPU pipeline. To hardware designers’ credit, modern CPUs use a number of extremely clever tricks to make such operations look atomic even though they are in fact being executed piece-at-a-time, but even so, there are cases where the pipeline
must be delayed or even flushed in order to permit a given atomic operation to complete correctly.

The resulting effect on performance is depicted in Figure 2.5.

Unfortunately, atomic operations usually apply only to single elements of data. Because many parallel algorithms require that ordering constraints be maintained between updates of multiple data elements, most CPUs provide memory barriers. These memory barriers also serve as performance-sapping obstacles, as described in the next section.

Quick Quiz 2.2: What types of machines would allow atomic operations on multiple data elements? ■

2.1.4 Memory Barriers

Memory barriers will be considered in more detail in Section 12.2 and Appendix C. In the meantime, consider the following simple lock-based critical section:

1 spin_lock(&mylock);
2 a = a + 1;
3 spin_unlock(&mylock);

If the CPU were not constrained to execute these statements in the order shown, the effect would be that the variable “a” would be incremented without the protection of “mylock”, which would certainly defeat the purpose of acquiring it. To prevent such destructive reordering, locking primitives contain either explicit or implicit memory barriers. Because the whole purpose of these memory barriers is to prevent reorderings that the CPU would otherwise undertake in order to increase performance, memory barriers almost always reduce performance, as depicted in Figure 2.6.

2.1.5 Cache Misses

An additional multi-threading obstacle to CPU performance is the “cache miss”. As noted earlier, modern CPUs sport large caches in order to reduce the performance penalty that would otherwise be incurred due to high memory latencies. However, these caches are actually counter-productive for variables that are frequently shared among CPUs. This is because when a given CPU wishes to modify the variable, it is most likely the case that some other CPU has modified it recently. In this case, the variable will be in that other CPU’s cache, but not in this CPU’s cache, which will therefore incur an expensive cache miss (see Section C.1 for more detail). Such cache misses form a major obstacle to CPU performance, as shown in Figure 2.7.

2.1.6 I/O Operations

A cache miss can be thought of as a CPU-to-CPU I/O operation, and as such is one of the cheapest I/O oper-
I/O operations involving networking, mass storage, or (worse yet) human beings pose much greater obstacles than the internal obstacles called out in the prior sections, as illustrated by Figure 2.8.

This is one of the differences between shared-memory and distributed-system parallelism: shared-memory parallel programs must normally deal with no obstacle worse than a cache miss, while a distributed parallel program will typically incur the larger network communication latencies. In both cases, the relevant latencies can be thought of as a cost of communication—a cost that would be absent in a sequential program. Therefore, the ratio between the overhead of the communication to that of the actual work being performed is a key design parameter. A major goal of parallel design is to reduce this ratio as needed to achieve the relevant performance and scalability goals.

Of course, it is one thing to say that a given operation is an obstacle, and quite another to show that the operation is a significant obstacle. This distinction is discussed in the following sections.

### 2.2 Overheads

This section presents actual overheads of the obstacles to performance listed out in the previous section. However, it is first necessary to get a rough view of hardware system architecture, which is the subject of the next section.

#### 2.2.1 Hardware System Architecture

Figure 2.9 shows a rough schematic of an eight-core computer system. Each die has a pair of CPU cores, each with its cache, as well as an interconnect allowing the pair of CPUs to communicate with each other. The system interconnect in the middle of the diagram allows the four dies to communicate, and also connects them to main memory.

Data moves through this system in units of “cache lines”, which are power-of-two fixed-size aligned blocks of memory, usually ranging from 32 to 256 bytes in size. When a CPU loads a variable from memory to one of its registers, it must first load the cacheline containing that variable into its cache. Similarly, when a CPU stores a
value from one of its registers into memory, it must also load the cacheline containing that variable into its cache, but must also ensure that no other CPU has a copy of that cacheline.

For example, if CPU 0 were to perform a compare-and-swap (CAS) operation on a variable whose cacheline resided in CPU 7’s cache, the following over-simplified sequence of events might ensue:

1. CPU 0 checks its local cache, and does not find the cacheline.
2. The request is forwarded to CPU 0’s and 1’s interconnect, which checks CPU 1’s local cache, and does not find the cacheline.
3. The request is forwarded to the system interconnect, which checks with the other three dies, learning that the cacheline is held by the die containing CPU 6 and 7.
4. The request is forwarded to CPU 6’s and 7’s interconnect, which checks both CPUs’ caches, finding the value in CPU 7’s cache.
5. CPU 7 forwards the cacheline to its interconnect, and also flushes the cacheline from its cache.
6. CPU 6’s and 7’s interconnect forwards the cacheline to the system interconnect.
7. The system interconnect forwards the cacheline to CPU 0’s and 1’s interconnect.
8. CPU 0’s and 1’s interconnect forwards the cacheline to CPU 0’s cache.
9. CPU 0 can now perform the CAS operation on the value in its cache.

Quick Quiz 2.3: This is a simplified sequence of events? How could it possibly be any more complex?

Quick Quiz 2.4: Why is it necessary to flush the cacheline from CPU 7’s cache?

2.2.2 Costs of Operations

The overheads of some common operations important to parallel programs are displayed in Table 2.1. This system’s clock period rounds to 0.6ns. Although it is not unusual for modern microprocessors to be able to retire multiple instructions per clock period, the operations will be normalized to a full clock period in the third column, labeled “Ratio”. The first thing to note about this table is the large values of many of the ratios.

The best-case CAS operation consumes almost forty nanoseconds, a duration more than sixty times that of the clock period. Here, “best case” means that the same CPU...
CHAPTER 2. HARDWARE AND ITS HABITS

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost (ns)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock period</td>
<td>0.6</td>
<td>1.0</td>
</tr>
<tr>
<td>Best-case CAS</td>
<td>37.9</td>
<td>63.2</td>
</tr>
<tr>
<td>Best-case lock</td>
<td>65.6</td>
<td>109.3</td>
</tr>
<tr>
<td>Single cache miss</td>
<td>139.5</td>
<td>232.5</td>
</tr>
<tr>
<td>CAS cache miss</td>
<td>306.0</td>
<td>510.0</td>
</tr>
<tr>
<td>Comms Fabric</td>
<td>3,000</td>
<td>5,000</td>
</tr>
<tr>
<td>Global Comms</td>
<td>130,000,000</td>
<td>216,000,000</td>
</tr>
</tbody>
</table>

Table 2.1: Performance of Synchronization Mechanisms on 4-CPU 1.8GHz AMD Opteron 844 System

now performing the CAS operation on a given variable was the last CPU to operate on this variable, so that the corresponding cache line is already held in that CPU’s cache. Similarly, the best-case lock operation (a “round trip” pair consisting of a lock acquisition followed by a lock release) consumes more than sixty nanoseconds, or more than one hundred clock cycles. Again, “best case” means that the data structure representing the lock is already in the cache belonging to the CPU acquiring and releasing the lock. The lock operation is more expensive than CAS because it requires two atomic operations on the lock data structure.

An operation that misses the cache consumes almost one hundred and forty nanoseconds, or more than two hundred clock cycles. A CAS operation, which must look at the old value of the variable as well as store a new value, consumes over three hundred nanoseconds, or more than five hundred clock cycles. Think about this a bit. In the time required to do one CAS operation, the CPU could have executed more than five hundred normal instructions. This should demonstrate the limitations of fine-grained locking.

Quick Quiz 2.5: Surely the hardware designers could be persuaded to improve this situation! Why have they been content with such abysmal performance for these single-instruction operations?

I/O operations are even more expensive. A high performance (and expensive!) communications fabric, such as InfiniBand or any number of proprietary interconnects, has a latency of roughly three microseconds, during which time five thousand instructions might have been executed. Standards-based communications networks often require some sort of protocol processing, which further increases the latency. Of course, geographic distance also increases latency, with the theoretical speed-of-light latency around the world coming to roughly 130 milliseconds, or more than 200 million clock cycles.

Quick Quiz 2.6: These numbers are insanely large! How can I possibly get my head around them?

2.3 Hardware Free Lunch?

The major reason that concurrency has been receiving so much focus over the past few years is the end of Moore’s Law induced single-threaded performance increases (or “free lunch” [Sut08]), as shown in Figure 1.1 on page 3. This section briefly surveys a few ways that hardware designers might be able to bring back some form of the “free lunch”.

However, the preceding section presented some substantial hardware obstacles to exploiting concurrency. One severe physical limitation that hardware designers face is the finite speed of light. As noted in Figure 2.9 on page 15, light can travel only about an 8-centimeters round trip in a vacuum during the duration of a 1.8 GHz clock period. This distance drops to about 3 centimeters for a 5 GHz clock. Both of these distances are relatively small compared to the size of a modern computer system.

To make matters even worse, electrons in silicon move from three to thirty times more slowly than does light in a vacuum, and common clocked logic constructs run still more slowly, for example, a memory reference may need to wait for a local cache lookup to complete before the request may be passed on to the rest of the system. Furthermore, relatively low speed and high power drivers are required to move electrical signals from one silicon die to another, for example, to communicate between a CPU and main memory.

There are nevertheless some technologies (both hardware and software) that might help improve matters:

1. 3D integration,
2. Novel materials and processes,
3. Substituting light for electrons,
4. Special-purpose accelerators, and
5. Existing parallel software.

Each of these is described in one of the following sections.
2.3. HARDWARE FREE LUNCH?

2.3.1 3D Integration

3-dimensional integration (3DI) is the practice of bonding very thin silicon dies to each other in a vertical stack. This practice provides potential benefits, but also poses significant fabrication challenges [Kni08].

Perhaps the most important benefit of 3DI is decreased path length through the system, as shown in Figure 2.10. A 3-centimeter silicon die is replaced with a stack of four 1.5-centimeter dies, in theory decreasing the maximum path through the system by a factor of two, keeping in mind that each layer is quite thin. In addition, given proper attention to design and placement, long horizontal electrical connections (which are both slow and power hungry) can be replaced by short vertical electrical connections, which are both faster and more power efficient.

However, delays due to levels of clocked logic will not be decreased by 3D integration, and significant manufacturing, testing, power-supply, and heat-dissipation problems must be solved for 3D integration to reach production while still delivering on its promise. The heat-dissipation problems might be solved using semiconductors based on diamond, which is a good conductor for heat, but an electrical insulator. That said, it remains difficult to grow large single diamond crystals, to say nothing of slicing them into wafers. In addition, it seems unlikely that any of these technologies will be able to deliver the exponential increases to which some people have become accustomed. That said, they may be necessary steps on the path to the late Jim Gray’s “smoking hairy golf balls” [Gra02].

2.3.2 Novel Materials and Processes

Stephen Hawking is said to have claimed that semiconductor manufacturers have but two fundamental problems: (1) the finite speed of light and (2) the atomic nature of matter [Gar07]. It is possible that semiconductor manufacturers are approaching these limits, but there are nevertheless a few avenues of research and development focused on working around these fundamental limits.

One workaround for the atomic nature of matter are so-called “high-K dielectric” materials, which allow larger devices to mimic the electrical properties of infeasibly small devices. These materials pose some severe fabrication challenges, but nevertheless may help push the frontiers out a bit farther. Another more-exotic workaround stores multiple bits in a single electron, relying on the fact that a given electron can exist at a number of energy levels. It remains to be seen if this particular approach can be made to work reliably in production semiconductor devices.

Another proposed workaround is the “quantum dot” approach that allows much smaller device sizes, but which is still in the research stage.

Although the speed of light would be a hard limit, the fact is that semiconductor devices are limited by the speed of electrons rather than that of light, given that electrons in semiconductor materials move at between 3% and 30% of the speed of light in a vacuum. The use of copper connections on silicon devices is one way to increase the speed of electrons, and it is quite possible that additional advances will push closer still to the actual speed of light. In addition, there have been some experiments with tiny optical fibers as interconnects within and between chips, based on the fact that the speed of light in glass is more than 60% of the speed of light in a vacuum. One obstacle to such optical fibers is the inefficiency conversion between electricity and light and vice versa, resulting in both power-consumption and heat-dissipation problems.

That said, absent some fundamental advances in the field of physics, any exponential increases in the speed of data flow will be sharply limited by the actual speed of light in a vacuum.

2.3.3 Special-Purpose Accelerators

A general-purpose CPU working on a specialized problem is often spending significant time and energy doing work that is only tangentially related to the problem at hand. For example, when taking the dot product of a pair of vectors, a general-purpose CPU will normally use a loop (possibly unrolled) with a loop counter. Decoding the instructions, incrementing the loop counter, testing this counter, and branching back to the top of the loop are in some sense wasted effort: the real goal is instead to multi-
ply corresponding elements of the two vectors. Therefore, a specialized piece of hardware designed specifically to multiply vectors could get the job done more quickly and with less energy consumed.

This is in fact the motivation for the vector instructions present in many commodity microprocessors. Because these instructions operate on multiple data items simultaneously, they would permit a dot product to be computed with less instruction-decode and loop overhead.

Similarly, specialized hardware can more efficiently encrypt and decrypt, compress and decompress, encode and decode, and many other tasks besides. Unfortunately, this efficiency does not come for free. A computer system incorporating this specialized hardware will contain more transistors, which will consume some power even when not in use. Software must be modified to take advantage of this specialized hardware, and this specialized hardware must be sufficiently generally useful that the high up-front hardware-design costs can be spread over enough users to make the specialized hardware affordable. In part due to these sorts of economic considerations, specialized hardware has thus far appeared only for a few application areas, including graphics processing (GPUs), vector processors (MMX, SSE, and VMX instructions), and, to a lesser extent, encryption.

Nevertheless, given the end of Moore’s-Law-induced single-threaded performance increases, it seems safe to predict that there will be an increasing variety of special-purpose hardware going forward.

2.3.4 Existing Parallel Software

Although multicore CPUs seem to have taken the computing industry by surprise, the fact remains that shared-memory parallel computer systems have been commercially available for more than a quarter century. This is more than enough time for significant parallel software to make its appearance, and it indeed has. Parallel operating systems are quite commonplace, as are parallel threading libraries, parallel relational database management systems, and parallel numerical software. Using existing parallel software goes a long ways towards solving any parallel-software crisis we might encounter.

Perhaps the most common example is the parallel relational database management system. It is not unusual for single-threaded programs, often written in high-level scripting languages, to access a central relational database concurrently. In the resulting highly parallel system, only the database need actually deal directly with parallelism.

A very nice trick when it works!

2.4 Software Design Implications

The values of the ratios in Table 2.1 are critically important, as they limit the efficiency of a given parallel application. To see this, suppose that the parallel application uses CAS operations to communicate among threads. These CAS operations will typically involve a cache miss, that is, assuming that the threads are communicating primarily with each other rather than with themselves. Suppose further that the unit of work corresponding to each CAS communication operation takes 300ns, which is sufficient time to compute several floating-point transcendental functions. Then about half of the execution time will be consumed by the CAS communication operations! This in turn means that a two-CPU system running such a parallel program would run no faster than one a sequential implementation running on a single CPU.

The situation is even worse in the distributed-system case, where the latency of a single communications operation might take as long as thousands or even millions of floating-point operations. This illustrates how important it is for communications operations to be extremely infrequent and to enable very large quantities of processing.

Quick Quiz 2.7: Given that distributed-systems communication is so horribly expensive, why does anyone bother with them?

The lesson should be quite clear: parallel algorithms must be explicitly designed to run nearly independent threads. The less frequently the threads communicate, whether by atomic operations, locks, or explicit messages, the better the application’s performance and scalability will be. In short, achieving excellent parallel performance and scalability means striving for embarrassingly parallel algorithms and implementations, whether by careful choice of data structures and algorithms, use of existing parallel applications and environments, or transforming the problem into one for which an embarrassingly parallel solution exists.

Chapter 5 will discuss design disciplines that promote performance and scalability.
Chapter 3
Tools of the Trade

This chapter provides a brief introduction to some basic tools of the parallel-programming trade, focusing mainly on those available to user applications running on operating systems similar to Linux. Section 3.1 begins with scripting languages, Section 3.2 describes the multi-process parallelism supported by the POSIX API, Section 3.2 touches on POSIX threads, and finally, Section 3.3 describes atomic operations.

Please note that this chapter provides but a brief introduction. More detail is available from the references cited, and more information on how best to use these tools will be provided in later chapters.

3.1 Scripting Languages

The Linux shell scripting languages provide simple but effective ways of managing parallelism. For example, suppose that you had a program compute_it that you needed to run twice with two different sets of arguments. This can be accomplished as follows:

```bash
1 compute_it 1 > compute_it.1.out &
2 compute_it 2 > compute_it.2.out &
3 wait
4 cat compute_it.1.out
5 cat compute_it.2.out
```

Lines 1 and 2 launch two instances of this program, redirecting their output to two separate files, with the & character directing the shell to run the two instances of the program in the background. Line 3 waits for both instances to complete, and lines 4 and 5 display their output. The resulting execution is as shown in Figure 3.1: the two instances of compute_it execute in parallel, wait completes after both of them do, and then the two instances of cat execute sequentially.

Quick Quiz 3.1: But this silly shell script isn’t a real parallel program! Why bother with such trivia???

Quick Quiz 3.2: Is there a simpler way to create a parallel shell script? If so, how? If not, why not?

For another example, the make software-build scripting language provides a \(-j\) option that specifies how much parallelism should be introduced into the build process. For example, typing `make -j4` when building a Linux kernel specifies that up to four parallel compiles be carried out concurrently.

It is hoped that these simple examples convince you that parallel programming need not always be complex or difficult.

Quick Quiz 3.3: But if script-based parallel programming is so easy, why bother with anything else?
CHAPTER 3. TOOLS OF THE TRADE

3.2 POSIX Multiprocessing

This section scratches the surface of the POSIX environment, including pthreads [Ope97], as this environment is readily available and widely implemented. Section 3.2.1 provides a glimpse of the POSIX fork() and related primitives, Section 3.2.2 touches on thread creation and destruction, Section 3.2.3 gives a brief overview of POSIX locking, and, finally, Section 3.4 presents the analogous operations within the Linux kernel.

3.2.1 POSIX Process Creation and Destruction

Processes are created using the fork() primitive, they may be destroyed using the kill() primitive, they may destroy themselves using the exit() primitive. A process executing a fork() primitive is said to be the “parent” of the newly created process. A parent may wait on its children using the wait() primitive.

Please note that the examples in this section are quite simple. Real-world applications using these primitives might need to manipulate signals, file descriptors, shared memory segments, and any number of other resources. In addition, some applications need to take specific actions if a given child terminates, and might also need to be concerned with the reason that the child terminated. These concerns can of course add substantial complexity to the code. For more information, see any of a number of textbooks on the subject [Ste92].

If fork() succeeds, it returns twice, once for the parent and again for the child. The value returned from fork() allows the caller to tell the difference, as shown in Figure 3.2 (forkjoin.c). Line 1 executes the fork() primitive, and saves its return value in local variable pid. Line 2 checks to see if pid is zero, in which case, this is the child, which continues on to execute line 3. As noted earlier, the child may terminate via the exit() primitive. Otherwise, this is the parent, which checks for an error return from the fork() primitive on line 4, and prints an error and exits on lines 5-7 if so. Otherwise, the fork() has executed successfully, and the parent therefore executes line 9 with the variable pid containing the process ID of the child.

The parent process may use the wait() primitive to wait for its children to complete. However, use of this primitive is a bit more complicated than its shell-script counterpart, as each invocation of wait() waits for but one child process. It is therefore customary to wrap wait() into a function similar to the waitall() function shown in Figure 3.3 (api-pthread.h), this waitall() function having semantics similar to the shell-script wait command. Each pass through the loop spanning lines 6-15 waits on one child process. Line 7 invokes the wait() primitive, which blocks until a child process exits, and returns that child’s process ID. If the process ID is instead -1, this indicates that the wait() primitive was unable to wait on a child. If so, line 9 checks for the ECHILD errno, which indicates that there are no more child processes, so that line 10 exits the loop. Otherwise, lines 11 and 12 print an error and exit.

Quick Quiz 3.4: Why does this wait() primitive need to be so complicated? Why not just make it work like the shell-script wait does?

It is critically important to note that the parent and child do not share memory. This is illustrated by the program shown in Figure 3.4 (forkjoinvar.c), in which the child sets a global variable x to 1 on line 6, prints a message on line 7, and exits on line 8. The parent continues at line 14, where it waits on the child, and on line 15 finds that its copy of the variable x is still zero. The output is thus as follows:

```c
Child process set x=1
Parent process sees x=0
```
3.2. POSIX MULTIPROCESSING

Figure 3.4: Processes Created Via fork() Do Not Share Memory

```c
1 int x = 0;
2 int pid;
3
4 pid = fork();
5 if (pid == 0) { /* child */
6   x = 1;
7   printf("Child process set x=1\n");
8   exit(0);
9 }
10 if (pid < 0) { /* parent, upon error */
11   perror("fork");
12   exit(-1);
13 }
14 waitall();
15 printf("Parent process sees x=%d\n", x);
```

Quick Quiz 3.5: Isn’t there a lot more to fork() and wait() than discussed here? ■

The finest-grained parallelism requires shared memory, and this is covered in Section 3.2.2. That said, shared-memory parallelism can be significantly more complex than fork-join parallelism.

3.2.2 POSIX Thread Creation and Destruction

To create a thread within an existing process, invoke the `pthread_create()` primitive, for example, as shown on lines 15 and 16 of Figure 3.5 (pcreate.c). The first argument is a pointer to a `pthread_t` in which to store the ID of the thread to be created, the second NULL argument is a pointer to an optional `pthread_attr_t`, the third argument is the function (in this case, `mythread()`) that is to be invoked by the new thread, and the last NULL argument is the argument that will be passed to `mythread`.

In this example, `mythread()` simply returns, but it could instead call `pthread_exit()`.

Quick Quiz 3.6: If the `mythread()` function in Figure 3.5 can simply return, why bother with `pthread_exit()`? ■

The `pthread_join()` primitive, shown on line 20, is analogous to the fork-join `wait()` primitive. It blocks until the thread specified by the `tid` variable completes execution, either by invoking `pthread_exit()` or by returning from the thread’s top-level function. The thread’s exit value will be stored through the pointer passed as the second argument to `pthread_join()`. The thread’s exit value is either the value passed to `pthread_exit()` or the value returned by the thread’s top-level function, depending on how the thread in question exits.

The program shown in Figure 3.5 produces output as follows, demonstrating that memory is in fact shared between the two threads:

```
Child process set x=1
Parent process sees x=1
```

Quick Quiz 3.7: If the C language makes no guarantees in presence of a data race, then why does the Linux kernel have so many data races? Are you trying to tell me that the Linux kernel is completely broken???

3.2.3 POSIX Locking

The POSIX standard allows the programmer to avoid data races via “POSIX locking”. POSIX locking features a number of primitives, the most fundamental of which are
pthread_mutex_lock() and pthread_mutex_unlock(). These primitives operate on locks, which are of type pthread_mutex_t. These locks may be declared statically and initialized with PTHREAD_MUTEX_INITIALIZER, or they may be allocated dynamically and initialized using the pthread_mutex_init() primitive. The demonstration code in this section will take the former course.

The pthread_mutex_lock() primitive “acquires” the specified lock, and the pthread_mutex_unlock() “releases” the specified lock. Because these are “exclusive” locking primitives, only one thread at a time may “hold” a given lock at a given time. For example, if a pair of threads attempt to acquire the same lock concurrently, one of the pair will be “granted” the lock first, and the other will wait until the first thread releases the lock.

Quick Quiz 3.8: What if I want several threads to hold the same lock at the same time? ■

This exclusive-locking property is demonstrated using the code shown in Figure 3.6 (lock.c). Line 1 defines and initializes a POSIX lock named lock_a, while line 2 similarly defines and initializes a lock named lock_b. Line 3 defines and initializes a shared variable x.

Lines 5-28 defines a function lock_reader() which repeatedly reads the shared variable x while holding the lock specified by arg. Line 10 casts arg to a pointer to a pthread_mutex_t, as required by the pthread_mutex_lock() and pthread_mutex_unlock() primitives.

Quick Quiz 3.9: Why not simply make the argument to lock_reader() on line 5 of Figure 3.6 be a pointer to a pthread_mutex_t? ■

Lines 12-15 acquire the specified pthread_mutex_t, checking for errors and exiting the program if any occur. Lines 16-23 repeatedly check the value of x, printing the new value each time that it changes. Line 22 sleeps for one millisecond, which allows this demonstration to run nicely on a uniprocessor machine. Line 24-27 release the pthread_mutex_t, again checking for errors and exiting the program if any occur. Finally, line 28 returns NULL, again to match the function type required by pthread_create().

Quick Quiz 3.10: Writing four lines of code for each acquisition and release of a pthread_mutex_t sure seems painful! Isn’t there a better way? ■

Lines 31-49 of Figure 3.6 shows lock_writer(), which periodically update the shared variable x while holding the specified pthread_mutex_t. As with

```
1 pthread_mutex_t lock_a = PTHREAD_MUTEX_INITIALIZER;
2 pthread_mutex_t lock_b = PTHREAD_MUTEX_INITIALIZER;
3 int x = 0;
4
5 void *lock_reader(void *arg)
6 {
7   int i;
8   int newx = -1;
9   int oldx = -1;
10  pthread_mutex_t *pmlp = (pthread_mutex_t *)arg;
11  if (pthread_mutex_lock(pmlp) != 0) {
12     perror("lock_reader:pthread_mutex_lock");
13     exit(-1);
14  }
15  for (i = 0; i < 100; i++) {
16     newx = ACCESS_ONCE(x);
17     if (newx != oldx) {
18       printf("lock_reader(): x = %d\n", newx);
19     }
20   }
21   oldx = newx;
22   poll(NULL, 0, 1);
23  }
24  if (pthread_mutex_unlock(pmlp) != 0) {
25     perror("lock_reader:pthread_mutex_unlock");
26     exit(-1);
27  }
28  return NULL;
29 }
30
31 void *lock_writer(void *arg)
32 {
33   int i;
34  pthread_mutex_t *pmlp = (pthread_mutex_t *)arg;
35  if (pthread_mutex_lock(pmlp) != 0) {
36    perror("lock_reader:pthread_mutex_lock");
37    exit(-1);
38  }
39  for (i = 0; i < 3; i++) {
40    ACCESS_ONCE(x)++;
41    poll(NULL, 0, 5);
42  }
43  if (pthread_mutex_unlock(pmlp) != 0) {
44    perror("lock_reader:pthread_mutex_unlock");
45    exit(-1);
46  }
47  return NULL;
48 }
```

Figure 3.6: Demonstration of Exclusive Locks
3.2. POSIX MULTIPROCESSING

```c
printf("Creating two threads using same lock:\n");
if (pthread_create(&tid1, NULL,
 lock_reader, &lock_a) != 0) {
 perror("pthread_create");
 exit(-1);
}
if (pthread_create(&tid2, NULL,
 lock_writer, &lock_a) != 0) {
 perror("pthread_create");
 exit(-1);
}
if (pthread_join(tid1, &vp) != 0) {
 perror("pthread_join");
 exit(-1);
}
if (pthread_join(tid2, &vp) != 0) {
 perror("pthread_join");
 exit(-1);
}
```

Figure 3.7: Demonstration of Same Exclusive Lock

lock_reader(), line 34 casts arg to a pointer to pthread_mutex_t, lines 36-39 acquires the specified lock, and lines 44-47 releases it. While holding the lock, lines 40-48 increment the shared variable x, sleeping for five milliseconds between each increment.

Figure 3.7 shows a code fragment that runs lock_reader() and lock_writer() as thread using the same lock, namely, lock_a. Lines 2-6 create a thread running lock_reader(), and then Lines 7-11 create a thread running lock_writer(). Lines 12-19 wait for both threads to complete. The output of this code fragment is as follows:

Creating two threads using same lock:
lock_reader(): x = 0

Because both threads are using the same lock, the lock_reader() thread cannot see any of the intermediate values of x produced by lock_writer() while holding the lock.

Quick Quiz 3.11: Is “x = 0” the only possible output from the code fragment shown in Figure 3.7? If so, why? If not, what other output could appear, and why?

Figure 3.8 shows a similar code fragment, but this time using different locks: lock_a for lock_reader() and lock_b for lock_writer(). The output of this code fragment is as follows:

Creating two threads w/different locks:
lock_reader(): x = 0
lock_reader(): x = 1
lock_reader(): x = 2
lock_reader(): x = 3

Because the two threads are using different locks, they do not exclude each other, and can run concurrently. The lock_reader() function can therefore see the intermediate values of x stored by lock_writer().

Quick Quiz 3.12: Using different locks could cause quite a bit of confusion, what with threads seeing each others’ intermediate states. So should well-written parallel programs restrict themselves to using a single lock in order to avoid this kind of confusion?

Quick Quiz 3.13: In the code shown in Figure 3.8, is lock_reader() guaranteed to see all the values produced by lock_writer()? Why or why not?

Quick Quiz 3.14: Wait a minute here!!! Figure 3.7 didn’t initialize shared variable x, so why does it need to be initialized in Figure 3.8?

Although there is quite a bit more to POSIX exclusive locking, these primitives provide a good start and are in fact sufficient in a great many situations. The next section takes a brief look at POSIX reader-writer locking.

3.2.4 POSIX Reader-Writer Locking

The POSIX API provides a reader-writer lock, which is represented by a pthread_rwlock_t. As with pthread_mutex_t, pthread_rwlock_t may be statically initialized via PTHREAD_RWLOCK_INITIALIZER or dynamically initialized via the pthread_rwlock_init() primitive. The pthread_rwlock_rdlock() primitive read-acquires the specified pthread_rwlock_t, the pthread_rwlock_rwlock() primitive write-acquires it, and the pthread_rwlock_unlock() primitive releases it. Only a single thread may write-hold a given pthread_rwlock_t at any given time, but
multiple threads may read-hold a given `pthread_rwlock_t`, at least while there is no thread currently write-holding it.

As you might expect, reader-writer locks are designed for read-mostly situations. In these situations, a reader-writer lock can provide greater scalability than can an exclusive lock because the exclusive lock is by definition limited to a single thread holding the lock at any given time, while the reader-writer lock permits an arbitrarily large number of readers to concurrently hold the lock. However, in practice, we need to know how much additional scalability is provided by reader-writer locks.

Figure 3.9 (rwlockscale.c) shows one way of measuring reader-writer lock scalability. Line 1 shows the definition and initialization of the reader-writer lock, line 2 shows the `holdtime` argument controlling the time each thread holds the reader-writer lock, line 3 shows the `thinktime` argument controlling the time between the release of the reader-writer lock and the next acquisition, line 4 defines the `readcounts[]` array into which each reader thread places the number of times it acquired the lock, and line 5 defines the `nreadersrunning` variable, which determines when all reader threads have started running.

Lines 7-10 define `goflag`, which synchronizes the start and the end of the test. This variable is initially set to `GOFLAG_INIT`, then set to `GOFLAG_RUN` after all the reader threads have started, and finally set to `GOFLAG_STOP` to terminate the test run.

Lines 12-41 define `reader()`, which is the reader thread. Line 18 atomically increments the `nreadersrunning` variable to indicate that this thread is now running, and lines 19-21 wait for the test to start. The `ACCESS_ONCE()` primitive forces the compiler to fetch `goflag` on each pass through the loop—the compiler would otherwise be within its rights to assume that the value of `goflag` would never change.

The loop spanning lines 22-38 carries out the performance test. Lines 23-26 acquire the lock, lines 27-29 hold the lock for the specified duration (and the `barrier()` directive prevents the compiler from optimizing the loop out of existence), lines 30-33 release the lock, and lines 34-36 wait for the specified duration before re-acquiring the lock. Line 37 counts this lock acquisition.

Line 38 moves the lock-acquisition count to this thread's element of the `readcounts[]` array, and line 40 returns, terminating this thread.

Figure 3.10 shows the results of running this test on a 64-core Power-5 system with two hardware threads per core for a total of 128 software-visible CPUs. The `thinktime` parameter was zero for all these tests, and the `holdtime` parameter set to values ranging from one thousand ("1K" on the graph) to 100 million ("100M" on the graph). The actual value plotted is:

\[
\frac{L_N}{NL_1}
\]  

(3.1)

where \( N \) is the number of threads, \( LN \) is the number of lock acquisitions by \( N \) threads, and \( L1 \) is the number of lock acquisitions by a single thread. Given ideal hardware and software scalability, this value will always be 1.0.

As can be seen in the figure, reader-writer locking scalability is decidedly non-ideal, especially for smaller sizes of critical sections. To see why read-acquisition can be so slow, consider that all the acquiring threads must update the `pthread_rwlock_t` data structure.
Therefore, if all 128 executing threads attempt to read-acquire the reader-writer lock concurrently, they must update this underlying pthread_rwlock_t one at a time. One lucky thread might do so almost immediately, but the least-lucky thread must wait for all the other 127 threads to do their updates. This situation will only get worse as you add CPUs.

**Quick Quiz 3.15:** Isn’t comparing against single-CPU throughput a bit harsh? ■

**Quick Quiz 3.16:** But 1,000 instructions is not a particularly small size for a critical section. What do I do if I need a much smaller critical section, for example, one containing only a few tens of instructions? ■

**Quick Quiz 3.17:** In Figure 3.10, all of the traces other than the 100M trace deviate gently from the ideal line. In contrast, the 100M trace breaks sharply from the ideal line at 64 CPUs. In addition, the spacing between the 100M trace and the 10M trace is much smaller than that between the 10M trace and the 1M trace. Why does the 100M trace behave so much differently than the other traces? ■

**Quick Quiz 3.18:** Power-5 is several years old, and new hardware should be faster. So why should anyone worry about reader-writer locks being slow? ■

Despite these limitations, reader-writer locking is quite useful in many cases, for example when the readers must do high-latency file or network I/O. There are alternatives, some of which will be presented in Chapters 4 and 8.

### 3.3 Atomic Operations

Given that Figure 3.10 shows that the overhead of reader-writer locking is most severe for the smallest critical sections, it would be nice to have some other way to protect the tiniest of critical sections. One such way are atomic operations. We have seen one atomic operations already, in the form of the `__sync_fetch_and_add()` primitive on line 18 of Figure 3.9. This primitive atomically adds the value of its second argument to the value referenced by its first argument, returning the old value (which was ignored in this case). If a pair of threads concurrently execute `__sync_fetch_and_add()` on the same variable, the resulting value of the variable will include the result of both additions.

The gcc compiler offers a number of additional atomic operations, including `__sync_fetch_and_sub()`, `__sync_fetch_and_or()`, `__sync_fetch_and_and()`, `__sync_fetch_and_xor()`, and `__sync_fetch_and_nand()`, all of which return the old value. If you instead need the new value, you can instead use the `__sync_add_and_fetch()`, `__sync_sub_and_fetch()`, `__sync_or_and_fetch()`, `__sync_and_and_fetch()`, `__sync_xor_and_fetch()`, and `__sync_nand_and_fetch()` primitives.

**Quick Quiz 3.19:** Is it really necessary to have both sets of primitives? ■

The classic compare-and-swap operation is provided by a pair of primitives, `__sync_bool_compare_and_swap()` and `__sync_val_compare_and_swap()`. Both of these primitive atomically update a location to a new value, but only if its prior value was equal to the specified old value. The first variant returns 1 if the operation succeeded and 0 if it failed, for example, if the prior value was not equal to the specified old value. The second variant returns the prior value of the location, which, if equal to the specified old value, indicates that the operation succeeded. Either of the compare-and-swap operation is “universal” in the sense that any atomic operation on a single location can be implemented in terms of compare-and-swap, though the earlier operations are often more efficient where they apply. The compare-and-swap operation is also capable of serving as the basis for a wider set of atomic operations, though the more elaborate of these often suffer from complexity, scalability, and performance problems [Her90].

The `__sync_synchronize()` primitive issues a “memory barrier”, which constrains both the compiler’s
26

CHAPTER 3. TOOLS OF THE TRADE

and the CPU’s ability to reorder operations, as discussed
in Section 12.2. In some cases, it is sufficient to constrain
the compiler’s ability to reorder operations, while allowing the CPU free rein, in which case the barrier()
primitive may be used, as it in fact was on line 28 of
Figure 3.9. In some cases, it is only necessary to ensure
that the compiler avoids optimizing away a given memory
access, in which case the ACCESS_ONCE() primitive
may be used, as it was on line 17 of Figure 3.6. These last
two primitives are not provided directly by gcc, but may
be implemented straightforwardly as follows:
#define ACCESS_ONCE(x) (*(volatile typeof(x) *)&(x))
#define barrier() __asm__ __volatile__("": : :"memory")

Quick Quiz 3.20: Given that these atomic operations
will often be able to generate single atomic instructions
that are directly supported by the underlying instruction
set, shouldn’t they be the fastest possible way to get things
done?

3.4

Linux-Kernel Equivalents to
POSIX Operations

Unfortunately, threading operations, locking primitives,
and atomic operations were in reasonably wide use long
before the various standards committees got around to
them. As a result, there is considerable variation in how
these operations are supported. It is still quite common to
find these operations implemented in assembly language,
either for historical reasons or to obtain better performance in specialized circumstances. For example, the
gcc __sync_ family of primitives all provide memoryordering semantics, motivating many developers to create
their own implementations for situations where the memory ordering semantics are not required.
Therefore, Table 3.1 on page 27 provides a rough mapping between the POSIX and gcc primitives to those used
in the Linux kernel. Exact mappings are not always available, for example, the Linux kernel has a wide variety of
locking primitives, while gcc has a number of atomic operations that are not directly available in the Linux kernel.
Of course, on the one hand, user-level code does not need
the Linux kernel’s wide array of locking primitives, while
on the other hand, gcc’s atomic operations can be emulated reasonably straightforwardly using cmpxchg().
Quick Quiz 3.21: What happened to the Linux-kernel
equivalents to fork() and join()?

3.5

The Right Tool for the Job:
How to Choose?

As a rough rule of thumb, use the simplest tool that will
get the job done. If you can, simply program sequentially.
If that is insufficient, try using a shell script to mediate parallelism. If the resulting shell-script fork()/exec()
overhead (about 480 microseconds for a minimal C program on an Intel Core Duo laptop) is too large, try using
the C-language fork() and wait() primitives. If the
overhead of these primitives (about 80 microseconds for
a minimal child process) is still too large, then you might
need to use the POSIX threading primitives, choosing the
appropriate locking and/or atomic-operation primitives. If
the overhead of the POSIX threading primitives (typically
sub-microsecond) is too great, then the primitives introduced in Chapter 8 may be required. Always remember
that inter-process communication and message-passing
can be good alternatives to shared-memory multithreaded
execution.
Of course, the actual overheads will depend not only
on your hardware, but most critically on the manner in
which you use the primitives. Therefore, it is necessary
to make the right design choices as well as the correct
choice of individual primitives, as is discussed at length
in subsequent chapters.


3.5. THE RIGHT TOOL FOR THE JOB: HOW TO CHOOSE?

<table>
<thead>
<tr>
<th>Category</th>
<th>POSIX</th>
<th>Linux Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread Management</td>
<td>pthread_t</td>
<td>struct task_struct</td>
</tr>
<tr>
<td></td>
<td>pthread_create()</td>
<td>kthread_create</td>
</tr>
<tr>
<td></td>
<td>pthread_exit()</td>
<td>kthread_should_stop()</td>
</tr>
<tr>
<td></td>
<td>pthread_join()</td>
<td>kthread_stop(); (rough)</td>
</tr>
<tr>
<td></td>
<td>poll(NULL, 0, 5)</td>
<td>schedule_timeout_interruptible();</td>
</tr>
<tr>
<td>POSIX Locking</td>
<td>pthread_mutex_t</td>
<td>spinlock_t (rough)</td>
</tr>
<tr>
<td></td>
<td>PTHREAD_MUTEX_INITIALIZER</td>
<td>struct mutex</td>
</tr>
<tr>
<td></td>
<td>pthread_mutex_lock()</td>
<td>spin_lock() (and friends)</td>
</tr>
<tr>
<td></td>
<td>pthread_mutex_unlock()</td>
<td>spin_unlock() (and friends)</td>
</tr>
<tr>
<td>POSIX Reader-Writer Locking</td>
<td>pthread_rwlock_t</td>
<td>rwlock_t (rough)</td>
</tr>
<tr>
<td></td>
<td>PTHREAD_RWLOCK_INITIALIZER</td>
<td>struct_rwlock</td>
</tr>
<tr>
<td></td>
<td>pthread_rwlock_rdlock()</td>
<td>read_lock() (and friends)</td>
</tr>
<tr>
<td></td>
<td>pthread_rwlock_unlock()</td>
<td>read_unlock() (and friends)</td>
</tr>
<tr>
<td></td>
<td>pthread_rwlock_wrlock()</td>
<td>write_lock() (and friends)</td>
</tr>
<tr>
<td></td>
<td>pthread_rwlock_unlock()</td>
<td>write_unlock() (and friends)</td>
</tr>
<tr>
<td>Atomic Operations</td>
<td>C Scalar Types</td>
<td>atomic_t</td>
</tr>
<tr>
<td></td>
<td>atomic64_t</td>
<td>atomic64_add()</td>
</tr>
<tr>
<td></td>
<td>__sync_fetch_and_add()</td>
<td>atomic64_add_return()</td>
</tr>
<tr>
<td></td>
<td>__sync_fetch_and_sub()</td>
<td>atomic64_sub()</td>
</tr>
<tr>
<td></td>
<td>__sync_val_compare_and_swap()</td>
<td>atomic64_sub_return();</td>
</tr>
<tr>
<td></td>
<td>__sync_lock_test_and_set()</td>
<td>cmpxchg()</td>
</tr>
<tr>
<td></td>
<td>__sync_synchronize()</td>
<td>xchg() (rough)</td>
</tr>
<tr>
<td></td>
<td>smp_mb()</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Mapping from POSIX to Linux-Kernel Primitives
Counting is perhaps the simplest and most natural for a computer to do. However, counting efficiently and scalably on a large shared-memory multiprocessor can be quite challenging. Furthermore, the simplicity of the underlying concept of counting allows us to explore the fundamental issues of concurrency without the distractions of elaborate data structures or complex synchronization primitives. Counting therefore provides an excellent introduction to parallel programming.

This chapter covers a number of special cases for which there are simple, fast, and scalable counting algorithms. But first, let us find out how much you already know about concurrent counting.

Quick Quiz 4.1: Why on earth should efficient and scalable counting be hard? After all, computers have special hardware for the sole purpose of doing counting, addition, subtraction, and lots more besides, don’t they? ■

Quick Quiz 4.2: Network-packet counting problem. Suppose that you need to collect statistics on the number of networking packets (or total number of bytes) transmitted and/or received. Packets might be transmitted or received by any CPU on the system. Suppose further that this large machine is capable of handling a million packets per second, and that there is a systems-monitoring package that reads out the count every five seconds. How would you implement this statistical counter? ■

Quick Quiz 4.3: Approximate structure-allocation limit problem. Suppose that you need to maintain a count of the number of structures allocated in order to fail any allocations once the number of structures in use exceeds a limit (say, 10,000). Suppose further that these structures are short-lived, and that the limit is rarely exceeded, that there is almost always at least one structure in use, and suppose further still that it is necessary to know exactly when this counter reaches zero, for example, in order to free up some memory that is not required unless there is at least one structure in use. ■

Quick Quiz 4.4: Exact structure-allocation limit problem. Suppose that you need to maintain a count of the number of structures allocated in order to fail any allocations once the number of structures in use exceeds an exact limit (say, 10,000). Suppose further that these structures are short-lived, and that the limit is rarely exceeded, that there is almost always at least one structure in use, and suppose further still that it is necessary to know exactly when this counter reaches zero, for example, in order to free up some memory that is not required unless there is at least one structure in use. ■

Quick Quiz 4.5: Removable I/O device access-count problem. Suppose that you need to maintain a reference count on a heavily used removable mass-storage device, so that you can tell the user when it is safe to remove the device. This device follows the usual removal procedure where the user indicates a desire to remove the device, and the system tells the user when it is safe to do so. ■

The remainder of this chapter will develop answers to these questions.

```c
long counter = 0;
void inc_count(void)
{
    counter++;
}
long read_count(void)
{
    return counter;
}
```

Figure 4.1: Just Count!
CHAPTER 4. COUNTING

4.1 Why Isn’t Concurrent Counting Trivial?

Let’s start with something simple, for example, the straightforward use of arithmetic shown in Figure 4.1 (count nonatomic.c). Here, we have a counter on line 1, we increment it on line 5, and we read out its value on line 10. What could be simpler?

This approach has the additional advantage of being blazingly fast if you are doing lots of reading and almost no incrementing, and on small systems, the performance is excellent.

There is just one large fly in the ointment: this approach can lose counts. On my dual-core laptop, a short run invoked inc_count() 100,014,000 times, but the final value of the counter was only 52,909,118. Although it is true that approximate values have their place in computing, it is almost always necessary to do better than this.

Quick Quiz 4.6: But doesn’t the ++ operator produce an x86 add-to-memory instruction? And won’t the CPU cache cause this to be atomic? ■

Quick Quiz 4.7: The 8-figure accuracy on the number of failures indicates that you really did test this. Why would it be necessary to test such a trivial program, especially when the bug is easily seen by inspection? ■

The straightforward way to count accurately is to use atomic operations, as shown in Figure 4.2 (count_atomic.c). Line 1 defines an atomic variable, line 5 atomically increments it, and line 10 reads it out. Because this is atomic, it keeps perfect count. However, it is slower: on a Intel Core Duo laptop, it is about six times slower than non-atomic increment when a single thread is incrementing, and more than ten times slower if two threads are incrementing.

This poor performance should not be a surprise, given the discussion in Chapter 2, nor should it be a surprise that the performance of atomic increment gets slower as the number of CPUs and threads increase, as shown in Figure 4.3. In this figure, the horizontal dashed line resting on the x axis is the ideal performance that would be achieved by a perfectly scalable algorithm: with such an algorithm, a given increment would incur the same overhead that it would in a single-threaded program. Atomic increment of a single global variable is clearly decidedly non-ideal, and gets worse as you add CPUs.

Quick Quiz 4.8: Why doesn’t the dashed line on the x axis meet the diagonal line at y = 1? ■

Quick Quiz 4.9: But atomic increment is still pretty fast. And incrementing a single variable in a tight loop sounds pretty unrealistic to me, after all, most of the program’s execution should be devoted to actually doing work, not accounting for the work it has done! Why should I care about making this go faster? ■

For another perspective on global atomic increment, consider Figure 4.4. In order for each CPU to get a chance to increment a given global variable, the cache line containing that variable must circulate among all the CPUs, as shown by the red arrows. Such circulation will take significant time, resulting in the poor performance seen in Figure 4.3.

The following sections discuss high-performance counting, which avoids the delays inherent in such circulation.

Quick Quiz 4.10: But why can’t CPU designers sim-
4.2 Statistical Counters

This section covers the common special case of statistical counters, where the count is updated extremely frequently and the value is read out rarely, if ever. These will be used to solve the network-packet counting problem from the Quick Quiz on page 29.

4.2.1 Design

Statistical counting is typically handled by providing a counter per thread (or CPU, when running in the kernel), so that each thread updates its own counter. The aggregate value of the counters is read out by simply summing up all of the threads’ counters, relying on the commutative and associative properties of addition. This is an example of the Data Ownership pattern that will be introduced in Section 5.3.4.

Quick Quiz 4.11: But doesn’t the fact that C’s “integers” are limited in size complicate things?

4.2.2 Array-Based Implementation

One way to provide per-thread variables is to allocate an array with one element per thread (presumably cache aligned and padded to avoid false sharing).

Quick Quiz 4.12: An array??? But doesn’t that limit the number of threads?

Such an array can be wrapped into per-thread primitives, as shown in Figure 4.5 (count_stat.c). Line 1 defines an array containing a set of per-thread counters of type long named, creatively enough, counter.

Lines 3-6 show a function that increments the counters, using the __get_thread_var() primitive to locate the currently running thread’s element of the counter array. Because this element is modified only by the corresponding thread, non-atomic increment suffices.

Lines 8-16 show a function that reads out the aggregate value of the counter, using the for_each_thread() primitive to iterate over the list of currently running threads, and using the per_thread() primitive to fetch the specified thread’s counter. Because the hardware can fetch and store a properly aligned long atomically, and because gcc is kind enough to make use of this capability, normal loads suffice, and no special atomic instructions are required.

Quick Quiz 4.13: What other choice does gcc have, anyway???

Quick Quiz 4.14: How does the per-thread counter variable in Figure 4.5 get initialized?

Quick Quiz 4.15: How is the code in Figure 4.5 supposed to permit more than one counter?

This approach scales linearly with increasing number of updater threads invoking inc_count(). As is shown by the green arrows in Figure 4.6, the reason for this is that each CPU can make rapid progress incrementing its thread’s variable, with no expensive communication required crossing the full diameter of the computer system. However, this excellent update-side scalability comes at great read-side expense for large numbers of threads. The next section shows one way to reduce read-side expense while still retaining the update-side scalability.
4.2.3 Eventually Consistent Implementation

One way to retain update-side scalability while greatly improving read-side performance is to weaken consistency requirements. While the counting algorithm in the previous section is guaranteed to return a value between the value that an ideal counter would have taken on near the beginning of read_count()’s execution and that near the end of read_count()’s execution. 

Eventually consistency [Vog09] provides a weaker guarantee: in absence of calls to inc_count(), calls to read_count() will eventually return the correct answer.

We exploit eventual consistency by maintaining a global counter. However, updaters only manipulate their per-thread counters. A separate thread is provided to transfer counts from the per-thread counters to the global counter. Readers simply access the value of the global counter. If updaters are active, the value used by the readers will be out of date, however, once updates cease, the global counter will eventually converge on the true value—hence this approach qualifies as eventually consistent.

The implementation is shown in Figure 4.7 (count_stat_eventual.c). Lines 1-2 show the per-thread variable and the global variable that track the counter’s value, and line three shows stopflag which is used to coordinate termination (for the case where we want to terminate the program with an accurate counter value). The inc_count() function shown on lines 5-8 is identical to its counterpart in Figure 4.5. The read_count() function shown on lines 10-13 simply returns the value of the global_count variable.

However, the count_init() function on lines 34-

```c
1 DEFINE_PER_THREAD(atomic_t, counter);
2 atomic_t global_count;
3 int stopflag;
4
5 void inc_count(void)
6 {
7    atomic_inc(&__get_thread_var(counter));
8 }
9
10 unsigned long read_count(void)
11 {
12    return atomic_read(&global_count);
13 }
14
15 void *eventual(void *arg)
16 {
17    int t;
18    int sum;
19
20    while (stopflag < 3) {
21        sum = 0;
22        for_each_thread(t)
23            sum += atomic_xchg(&per_thread(counter, t), 0);
24        atomic_add(sum, &global_count);
25        poll(NULL, 0, 1);
26        if (stopflag) {
27            smp_mb();
28            stopflag++;
29        }
30    }
31    return NULL;
32 }
33
34 void count_init(void)
35 {
36    thread_id_t tid;
37
38    if (pthread_create(&tid, NULL, eventual, NULL) != 0) {
39        perror("count_init:pthread_create");
40        exit(-1);
41    }
42 }
43
44 void count_cleanup(void)
45 {
46    stopflag = 1;
47    while (stopflag < 3)
48        poll(NULL, 0, 1);
49    smp_mb();
50 }
```

Figure 4.7: Array-Based Per-Thread Eventually Consistent Counters
4.2. STATISTICAL COUNTERS

42 creates the eventual() thread shown on lines 15-32, which cycles through all the threads, using the atomic_xchg() function to remove count from each thread’s local counter, adding the sum to the global count variable. The eventual() thread waits an arbitrarily chosen one millisecond between passes. The count_cleanup() function on lines 44-50 coordinates termination.

This approach gives extremely fast counter read-out while still supporting linear counter-update performance. However, this excellent read-side performance and update-side scalability comes at the cost of high update-side overhead, due to both the atomic operations and the array indexing hidden in the __get_thread_var() primitive, which can be quite expensive on some CPUs with deep pipelines.

Quick Quiz 4.16: Why does inc_count() in Figure 4.7 need to use atomic instructions? ■

Quick Quiz 4.17: Won’t the single global thread in the function eventual() of Figure 4.7 be just as severe a bottleneck as a global lock would be? ■

Quick Quiz 4.18: Won’t the estimate returned by read_count() in Figure 4.7 become increasingly inaccurate as the number of threads rises? ■

4.2.4 Per-Thread-Variable-Based Implementation

Fortunately, gcc provides an __thread storage class that provides per-thread storage. This can be used as shown in Figure 4.8 (count_end.c) to implement a statistical counter that not only scales, but that also incurs little or no performance penalty to incrementers compared to simple non-atomic increment.

Lines 1-4 define needed variables: counter is the per-thread counter variable, the counterp[] array allows threads to access each others’ counters, finalcount accumulates the total as individual threads exit, and final_mutex coordinates between threads accumulating the total value of the counter and exiting threads.

Quick Quiz 4.19: Why do we need an explicit array to find the other threads’ counters? Why doesn’t gcc provide a per_thread() interface, similar to the Linux kernel’s per_cpu() primitive, to allow threads to more easily access each others’ per-thread variables? ■

The inc_count() function used by updaters is quite simple, as can be seen on lines 6-9.

The read_count() function used by readers is a bit more complex. Line 16 acquires a lock to exclude

Figure 4.8: Per-Thread Statistical Counters
CHAPTER 4. COUNTING

4.2.5 Discussion

These two implementations show that it is possible to obtain uniprocessor performance for statistical counters, despite running on a parallel machine.

**Quick Quiz 4.23:** What fundamental difference is there between counting packets and counting the total number of bytes in the packets, given that the packets vary in size?

**Quick Quiz 4.24:** Given that the reader must sum all the threads’ counters, this could take a long time given large numbers of threads. Is there any way that the increment operation can remain fast and scalable while allowing readers to also enjoy reasonable performance and scalability?

Given what has been presented in this section, you should now be able to answer the Quick Quiz about statistical counters for networking near the beginning of this chapter.

4.3 Approximate Limit Counters

Another special case of counting involves limit-checking. For example, as noted in the approximate structure-allocation limit problem in the Quick Quiz on page 29, suppose that you need to maintain a count of the number of structures allocated in order to fail any allocations once the number of structures in use exceeds a limit, in this case, 10,000. Suppose further that these structures are short-lived, and that this limit is rarely exceeded.

4.3.1 Design

One possible design for limit counters is to divide the limit of 10,000 by the number of threads, and give each thread a fixed pool of structures. For example, given 100 threads, each thread would manage its own pool of 100 structures. This approach is simple, and in some cases works well, but it does not handle the common case where a given structure is allocated by one thread and freed by another [MS93]. On the one hand, if a given thread takes credit for any structures it frees, then the thread doing most of the allocating runs out of structures, while the threads doing most of the freeing have lots of credits that they cannot use. On the other hand, if freed structures are credited to the CPU that allocated them, it will be necessary for CPUs to manipulate each others’ counters, which will require lots of expensive atomic instructions. Furthermore, because structures come in different
sizes, rather than supporting `inc_count()` and `dec_count()` interfaces, we implement `add_count()` and `sub_count()` to allow variable-sized structures to be properly accounted for.

In short, for many important workloads, we cannot fully partition the counter. However, we can partially partition the counter, so that in the common case, each thread need only manipulate its own private state, while still allowing counts to flow between threads as needed. The statistical counting scheme discussed in Section 4.2.4 provides an interesting starting point, in that it maintains a global counter as well as per-thread counters, with the aggregate value being the sum of all of these counters, global along with per-thread. The key change is to pull each thread’s counter into the global sum while that thread is still running, rather than waiting for thread exit. Clearly, we want threads to pull in their own counts, as cross-thread accesses are expensive and scale poorly.

This leaves open the question of exactly when a given thread’s counter should be pulled into the global counter. In the initial implementation, we will start by maintaining a limit on the value of the per-thread counter. When this limit would be exceeded, the thread pulls its counter into the global counter. Of course, we cannot simply add to the counter when a structure is allocated: we must also subtract from the counter when a structure is freed. We must therefore make use of the global counter when a subtraction would otherwise reduce the value of the per-thread counter below zero. However, if the limit is reasonably large, almost all of the addition and subtraction operations should be handled by the per-thread counter, which should give us good performance and scalability.

This design is an example of “parallel fastpath”, which is an important design pattern in which the common case executes with no expensive instructions and no interactions between threads, but where occasional use is also made of a more conservatively designed global algorithm.

4.3.2 Simple Limit Counter Implementation

Figure 4.9 shows both the per-thread and global variables used by this implementation. The per-thread `counter` and `countermax` variables are the corresponding thread’s local counter and the upper bound on that counter, respectively. The `globalcountmax` variable on line 3 contains the upper bound for the aggregate counter, and the `globalcount` variable on line 4 is the global counter. The sum of `globalcount` and each thread’s `counter` gives the aggregate value of the overall counter. The `globalreserve` variable on line 5 is the sum of all of the per-thread `countermax` variables. The relationship among these variables is shown by Figure 4.10:

1. The sum of `globalcount` and `globalreserve` must be less than or equal to `globalcountmax`.
2. The sum of all threads’ `countermax` values must be less than or equal to `globalreserve`.
3. Each thread’s `counter` must be less than or equal to that thread’s `countermax`.

Each element of the `counterp[]` array references the corresponding thread’s `counter` variable, and, fi-
```c
int add_count(unsigned long delta) {
    if (countermax - counter >= delta) {
        counter += delta;
        return 1;
    }
    spin_lock(&gblcnt_mutex);
    globalize_count();
    if (globalcountmax - globalcount - globalreserve < delta) {
        spin_unlock(&gblcnt_mutex);
        return 0;
    }
    globalcount += delta;
    balance_count();
    spin_unlock(&gblcnt_mutex);
    return 1;
}

int sub_count(unsigned long delta) {
    if (counter >= delta) {
        counter -= delta;
        return 1;
    }
    spin_lock(&gblcnt_mutex);
    globalize_count();
    if (globalcount < delta) {
        spin_unlock(&gblcnt_mutex);
        return 0;
    }
    globalcount -= delta;
    balance_count();
    spin_unlock(&gblcnt_mutex);
    return 1;
}

unsigned long read_count(void) {
    int t;
    unsigned long sum;
    spin_lock(&gblcnt_mutex);
    sum = globalcount;
    for_each_thread(t)
        if (counterp[t] != NULL)
            sum += *counterp[t];
    spin_unlock(&gblcnt_mutex);
    return sum;
}
```

Figure 4.11 shows the `add_count()`, `sub_count()`, and `read_count()` functions (count_lim.c).

Lines 1-18 show `add_count()`, which adds the specified value `delta` to the counter. Line 3 checks to see if there is room for `delta` on this thread's `counter`, and, if so, line 4 adds it and line 6 returns success. This is the `add_counter()` fastpath, and it does no atomic operations, references only per-thread variables, and should not incur any cache misses.

**Quick Quiz 4.25:** What is with the strange form of the condition on line 3 of Figure 4.11? Why not the following more intuitive form of the fastpath?

```c
3 if (counter + delta <= countermax) {
    counter += delta;
    return 1;
}
```

If the test on line 3 fails, we must access global variables, and thus must acquire `gblcnt_mutex` on line 7, which we release on line 11 in the failure case or on line 16 in the success case. Line 8 invokes `globalize_count()`, shown in Figure 4.12, which clears the thread-local variables, adjusting the global variables as needed, thus simplifying global processing. (But don't take my word for it, try coding it yourself!) Lines 9 and 10 check to see if addition of `delta` can be accommodated, with the meaning of the expression preceding the less-than sign shown in Figure 4.10 as the difference in height of the two red bars. If the addition of `delta` cannot be accommodated, then line 11 (as noted earlier) releases `gblcnt_mutex` and line 12 returns indicating failure.

Otherwise, line 14 subtracts `delta` from `globalcount`, line 15 invokes `balance_count()` (shown in Figure 4.12) in order to update both the global and the per-thread variables (hopefully setting this thread's `countermax` to re-enable the fastpath), if appropriate, to re-enable fastpath processing, line 16 release `gblcnt_mutex` (again, as noted earlier), and, finally, line 17 returns indicating success.

**Quick Quiz 4.26:** Why does `globalize_count()` zero the per-thread variables, only to later call `balance_count()` to refill them in Figure 4.11? Why not just leave the per-thread variables non-zero?

Lines 20-36 show `sub_count()`, which subtracts the specified `delta` from the counter. Line 22 checks to see if the per-thread counter can accommodate this subtraction, and, if so, line 23 does the subtraction and line 24 returns success. These lines form `sub_count()`'s fastpath, and, as with `add_count()`, this fastpath executes no costly operations.

If the fastpath cannot accommodate subtraction of `delta`, execution proceeds to the slowpath on lines 26-35. Because the slowpath must access global state, line 26 acquires `gblcnt_mutex`, which is release either by line 29 (in case of failure) or by line 34 (in case of suc-
 APPROXIMATE LIMIT COUNTERS

4.3. APPROXIMATE LIMIT COUNTERS

Line 27 invokes `globalize_count()` shown in Figure 4.12, which again clears the thread-local variables, adjusting the global variables as needed. Line 28 checks to see if the counter can accommodate subtracting `delta`, and, if not, line 29 releases `gblcnt_mutex` (as noted earlier) and line 30 returns failure.

**Quick Quiz 4.27:** Given that `globalreserve` counted against us in `add_count()`, why doesn’t it count for us in `sub_count()` in Figure 4.11?

If, on the other hand, line 28 finds that the counter can accommodate subtracting `delta`, then line 32 does the subtraction, line 33 invokes `balance_count()` (shown in Figure 4.12) in order to update both global and per-thread variables (hopefully re-enabling the fastpath), line 34 releases `gblcnt_mutex`, and line 35 returns success.

**Quick Quiz 4.28:** Why have both `add_count()` and `sub_count()` in Figure 4.11? Why not simply pass a negative number to `add_count()`?

Lines 38-50 show `read_count()`, which returns the aggregate value of the counter. It acquires `gblcnt_mutex` on line 43 and releases it on line 48, excluding global operations from `add_count()` and `sub_count()`, and, as we will see, also excluding thread creation and exit. Line 44 initializes local variable `sum` to the value of `globalcount`, and then the loop spanning lines 45-47 sums the per-thread `counter` variables. Line 49 then returns the sum.

Figure 4.12 shows a number of utility functions that support the `add_count()` `sub_count()`, and `read_count()` primitives shown in Figure 4.11.

Lines 1-7 show `globalize_count()`, which zeros the current thread’s per-thread counters, adjusting the global variables appropriately. It is important to note that this function does not change the aggregate value of the counter, but instead changes how the counter’s current value is represented. Line 3 adds the thread’s `counter` variable to `globalcount`, and line 4 zeros `counter`. Similarly, line 5 subtracts the per-thread `countermax` from `globalreserve`, and line 6 zeros `countermax`. It is helpful to refer to Figure 4.10 when reading both this function and `balance_count()`, which is next.

Lines 9-19 show `balance_count()`, which is, roughly speaking the inverse of `globalize_count()`. This function sets the current thread’s `counter` and `countermax` variables (with corresponding adjustments to `globalcount` and `globalreserve`) in an attempt to promote use of `add_count()`’s and

```c
1 static void globalize_count(void)
2 {
3   globalcount += counter;
4   counter = 0;
5   globalreserve -= countermax;
6   countermax = 0;
7 }
8
9 static void balance_count(void)
10 {
11   countermax = globalcountmax -
12     globalcount - globalreserve;
13   countermax /= num_online_threads();
14   globalreserve += countermax;
15   counter = countermax / 2;
16   if (counter > globalcount)
17     counter = globalcount;
18   globalcount -= counter;
19 }
20
21 void count_register_thread(void)
22 {
23   int idx = smp_thread_id();
24   spin_lock(&gblcnt_mutex);
25   counterp[idx] = &counter;
26   spin_unlock(&gblcnt_mutex);
27 }
28
29 void count_unregister_thread(int nthreadsexpected)
30 {
31   int idx = smp_thread_id();
32   spin_lock(&gblcnt_mutex);
33   counterp[idx] = NULL;
34   spin_unlock(&gblcnt_mutex);
35   
36   
37   
38   
```

Figure 4.12: Simple Limit Counter Utility Functions
sub_count()'s fastpaths. As with globalize_count(), balance_count() does not change the aggregate value of the counter. Lines 11-13 compute this thread’s share of that portion of globalcountmax that is not already covered by either globalcount or globalreserve, and assign the computed quantity to this thread’s countermax. Line 14 makes the corresponding adjustment to globalreserve. Line 15 sets this thread’s counter to the middle of the range from zero to countermax. Line 16 checks to see whether globalcount can in fact accommodate this value of counter, and, if not, line 17 decreases counter accordingly. Finally, in either case, line 18 makes the corresponding adjustment to globalcount.

Lines 21-28 show count_register_thread(), which sets up state for newly created threads. This function simply installs a pointer to the newly created thread’s counter variable into the corresponding entry of the counterp[] array under the protection of gblcnt_mutex.

Finally, lines 30-38 show count_unregister_thread(), which tears down state for a soon-to-be exiting thread. Line 34 acquires gblcnt_mutex and line 37 releases it. Line 35 invokes globalize_count() to clear out this thread’s counter state, and line 36 clears this thread’s entry in the counterp[] array.

### 4.3.3 Simple Limit Counter Discussion

This type of counter is quite fast when aggregate values are near zero, with some overhead due to the comparison and branch in both add_count()'s and sub_count()'s fastpaths. However, the use of a per-thread countermax reserve means that add_count() can fail even when the aggregate value of the counter is nowhere near globalcountmax. Similarly, sub_count() can fail even when the aggregate value of the counter is nowhere near zero.

In many cases, this is unacceptable. Even if the globalcountmax is intended to be an approximate limit, there is usually a limit to exactly how much approximation can be tolerated. One way to limit the degree of approximation is to impose an upper limit on the value of the per-thread countermax instances. This task is undertaken in the next section.

```c
1 unsigned long __thread counter = 0;
2 unsigned long __thread countermax = 0;
3 unsigned long globalcountmax = 10000;
4 unsigned long globalcount = 0;
5 unsigned long globalreserve = 0;
6 unsigned long *counterp[NR_THREADS] = { NULL;}
7 DEFINE_SPINLOCK(gblcnt_mutex);
8 #define MAX_COUNTERMAX 100
```

Figure 4.13: Approximate Limit Counter Variables

```c
1 static void balance_count(void)
2 {
3     countermax = globalcountmax -
4         globalcount - globalreserve;
5     countermax /= num_online_threads();
6     if (countermax > MAX_COUNTERMAX)
7         countermax = MAX_COUNTERMAX;
8     globalreserve += countermax;
9     counter = countermax / 2;
10    if (counter > globalcount)
11        counter = globalcount;
12    globalcount -= counter;
13 }
```

Figure 4.14: Approximate Limit Counter Balancing

### 4.3.4 Approximate Limit Counter Implementation

Because this implementation (count_lim_app.c) is quite similar to that in the previous section (Figures 4.9, 4.11, and 4.12), only the changes are shown here. Figure 4.13 is identical to Figure 4.9, with the addition of MAX_COUNTERMAX, which sets the maximum permissible value of the per-thread countermax variable.

Similarly, Figure 4.14 is identical to the balance_count() function in Figure 4.12, with the addition of lines 6 and 7, which enforce the MAX_COUNTERMAX limit on the per-thread countermax variable.

### 4.3.5 Approximate Limit Counter Discussion

These changes greatly reduce the limit inaccuracy seen in the previous version, but present another problem: any given value of MAX_COUNTERMAX will cause a workload-dependent fraction of accesses to fall off the fastpath. As the number of threads increase, non-fastpath execution will become both a performance and a scalability problem. However, we will defer this problem and turn instead to counters with exact limits.
4.4 EXACT LIMIT COUNTERS

To solve the exact structure-allocation limit problem noted in the Quick Quiz on page 29, we need a limit counter that can tell exactly when its limits are exceeded. One way of implementing such a limit counter is to cause threads that have reserved counts to give them up. One way to do this is to use atomic instructions. Of course, atomic instructions will slow down the fastpath, but on the other hand, it would be silly not to at least give them a try.

4.4.1 Atomic Limit Counter Implementation

Unfortunately, when causing a given thread to give up its count, it is necessary to atomically manipulate both that thread’s counter and countermax variables. The usual way to do this is to combine these two variables into a single variable, for example, given a 32-bit variable, using the high-order 16 bits to represent counter and the low-order 16 bits to represent countermax.

The variables and access functions for a simple atomic limit counter are shown in Figure 4.15 (count_lim_atom.c). The counter and countermax variables in earlier algorithms are combined into the single variable ctrandmax shown on line 1, with counter in the upper half and countermax in the lower half. This variable is of type atomic_t, which has an underlying representation of int.

Lines 2-6 show the definitions for globalcountmax, globalcount, globalreserve, counterp, and gblcnt_mutex, all of which take on roles similar to their counterparts in Figure 4.13. Line 7 defines CM_BITS, which gives the number of bits in each half of ctrandmax, and line 8 defines MAX_COUNTERMAX, which gives the maximum value that may be held in either half of ctrandmax.

Quick Quiz 4.29: In what way does line 7 of Figure 4.15 violate the C standard?

Lines 10-15 show the split_ctrandmax_int() function, which, when given the underlying int from the atomic_t ctrandmax variable, splits the current thread’s ctrandmax variable into its counter (in c) and countermax (in cm) respectively.

Quick Quiz 4.30: Given that there is only one ctrandmax variable, why bother passing in a pointer to it on line 18 of Figure 4.15?

Lines 17-25 show the split_ctrandmax() function, which picks up the underlying int from the specified variable on line 21, stores it as specified by the old argument on line 23, and then invokes split_ctrandmax_int() to split it on line 24.

Quick Quiz 4.31: Why does merge_ctrandmax() in Figure 4.15 return an int rather than storing directly into an atomic_t?

Figure 4.16 shows the add_count(), sub_count(), and read_count() functions.

Lines 1-32 show add_count(), whose fastpath spans lines 8-15, with the remainder of the function being the slowpath. Lines 8-14 of the fastpath form a compare-and-swap (CAS) loop, with the atomic_cmpxchg() primitives on lines 13-14 performing the actual CAS. Line 9 splits the current thread’s ctrandmax variable into its counter (in c) and countermax (in cm) com-
```c
1 int add_count(unsigned long delta)
2 {
3     int c;
4     int cm;
5     int old;
6     int new;
7     do {
8         split_ctrandmax(&ctrandmax, &old, &c, &cm);
9         if (delta > MAX_COUNTERMAX || c + delta > cm)
10            goto slowpath;
11         new = merge_ctrandmax(c + delta, cm);
12     } while (atomic_cmpxchg(&ctrandmax, old, new) != old);
13     return 1;
14 }
15
16 int sub_count(unsigned long delta)
17 {
18     int c;
19     int cm;
20     int old;
21     int new;
22     do {
23         split_ctrandmax(&ctrandmax, &old, &c, &cm);
24         if (delta > c)
25            goto slowpath;
26         new = merge_ctrandmax(c - delta, cm);
27     } while (atomic_cmpxchg(&ctrandmax, old, new) != old);
28     return 1;
29 }
30
31 int read_count()
32 {
33     int c;
34     int cm;
35     int old;
36     do {
37         split_ctrandmax(&ctrandmax, &old, &c, &cm);
38         if (delta > c)
39            goto slowpath;
40         new = merge_ctrandmax(c - delta, cm);
41     } while (atomic_cmpxchg(&ctrandmax, old, new) != old);
42     return 1;
43 }
44
45 int sub_count(long delta)
46 {
47     int c;
48     int cm;
49     int old;
50     int new;
51     do {
52         split_ctrandmax(&ctrandmax, &old, &c, &cm);
53         if (delta > c)
54            goto slowpath;
55         new = merge_ctrandmax(c - delta, cm);
56     } while (atomic_cmpxchg(&ctrandmax, old, new) != old);
57     return 1;
58 }
59
60 int read_count()
61 {
62     int c;
63     int cm;
64     int old;
65     int new;
66     do {
67         split_ctrandmax(&ctrandmax, &old, &c, &cm);
68         if (delta > c)
69            goto slowpath;
70         new = merge_ctrandmax(c - delta, cm);
71     } while (atomic_cmpxchg(&ctrandmax, old, new) != old);
72     return 1;
73 }
74
75 CHAPTER 4. COUNTING
76
77 Figure 4.16: Atomic Limit Counter Add and Subtract
78```
4.4. EXACT LIMIT COUNTERS

```
unsigned long read_count(void)
{
    int c;
    int cm;
    int old;
    int t;
    unsigned long sum;

    spin_lock(&gblcnt_mutex);
    sum = globalcount;
    for_each_thread(t)
        if (counterp[t] != NULL) {
            splitCtrandMax(counterp[t], &old, &c, &cm);
            sum += c;
        }
    spin_unlock(&gblcnt_mutex);
    return sum;
}
```

Figure 4.17: Atomic Limit Counter Read

The code for `globalize_count()` is shown on lines 1-12, and it is similar to that of previous algorithms, with the addition of line 7, which is now required to split out `counter` and `countermax` from `ctrandmax`.

The code for `flush_local_count()`, which moves all threads' local counter state to the global counter, is shown on lines 14-32. Line 22 checks to see if the value of `globalreserve` permits any per-thread counts, and, if not, line 23 returns. Otherwise, line 24 initializes local variable `zero` to a combined zeroed `counter` and `countermax`. The loop spanning lines 25-31 sequences through each thread. Line 26 checks to see if the current thread has counter state, and, if so, lines 27-30 move that state to the global counters. Line 27 atomically fetches the current thread's state while replacing it with zero. Line 28 splits this state into its `counter` and `countermax` components. Line 29 adds this thread's `counter` to `globalcount`, while line 30 subtracts this thread's `countermax` from `globalreserve`.

Quick Quiz 4.34: What stops a thread from simply refilling its `ctrandmax` variable immediately after `flush_local_count()` on line 14 of Figure 4.18 empties it?

Quick Quiz 4.35: What prevents concurrent execution of the fastpath of either `atomic_add()` or `atomic_sub()` from interfering with the `ctrandmax` variable while `flush_local_count()` is accessing it on line 27 of Figure 4.18 empties it?

Lines 34-54 show the code for `balance_count()`, which refills the calling thread's local `ctrandmax` vari-

```
static void globalize_count(void)
{
    int c;
    int cm;
    int old;
    splitCtrandMax(&ctrandmax, &old, &c, &cm);
    globalcount += c;
    globalreserve -= cm;
    old = mergeCtrandMax(0, 0);
    atomic_set(&ctrandmax, old);
}

static void flush_local_count(void)
{
    int c;
    int cm;
    int old;
    int t;
    int zero;
    if (globalreserve == 0)
        return;
    zero = mergeCtrandMax(0, 0);
    for_each_thread(t)
        if (counterp[t] != NULL) {
            old = atomic_xchg(counterp[t], zero);
            splitCtrandMax(old, &c, &cm);
            globalcount += c;
            globalreserve -= cm;
        }
}

static void balance_count(void)
{
    int c;
    int cm;
    int old;
    unsigned long limit;
    limit = globalcountmax - globalcount - globalreserve;
    limit /= num_online_threads();
    if (limit > MAX_COUNTERMAX)
        cm = MAX_COUNTERMAX;
    else
        cm = limit;
    globalreserve += cm;
    c = cm / 2;
    if (c > globalcount)
        c = globalcount;
    globalcount -= c;
    old = mergeCtrandMax(c, cm);
    atomic_set(&ctrandmax, old);
}

void count_register_thread(void)
{
    int idx = smp_thread_id();
    spin_lock(&gblcnt_mutex);
    counterp[idx] = &ctrandmax;
    spin_unlock(&gblcnt_mutex);
}

void count_unregister_thread(int nthreadsexpected)
{
    int idx = smp_thread_id();
    spin_lock(&gblcnt_mutex);
    globalize_count();
    counterp[idx] = NULL;
    spin_unlock(&gblcnt_mutex);
}
```

Figure 4.18: Atomic Limit Counter Utility Functions
able. This function is quite similar to that of the preceding algorithms, with changes required to handle the merged ctrandmax variable. Detailed analysis of the code is left as an exercise for the reader, as it is with the count_register_thread() function starting on line 56 and the count_unregister_thread() function starting on line 65.

Quick Quiz 4.36: Given that the atomic_set() primitive does a simple store to the specified atomic_t, how can line 53 of balance_count() in Figure 4.18 work correctly in face of concurrent flush_local_count() updates to this variable?

4.4.2 Atomic Limit Counter Discussion

This is the first implementation that actually allows the counter to be run all the way to either of its limits, but it does so at the expense of adding atomic operations to the fastpaths, which slow down the fastpaths significantly. Although some workloads might tolerate this slowdown, it is worthwhile looking for algorithms with better read-side performance. One such algorithm uses a signal handler to steal counts from other threads. Because signal handlers run in the context of the signaled thread, atomic operations are not necessary, as shown in the next section.

Quick Quiz 4.37: But signal handlers can be migrated to some other CPU while running. Doesn’t this possibility require that atomic instructions and memory barriers are required to reliably communicate between a thread and a signal handler that interrupts that thread?

4.4.3 Signal-Theft Limit Counter Design

Figure 4.19 shows the state diagram. The state machine starts out in the IDLE state, and when add_count() or sub_count() find that the combination of the local thread’s count and the global count cannot accommodate the request, the corresponding slowpath sets each thread’s theft state to REQ (unless that thread has no count, in which case it transitions directly to READY). Only the slowpath, which holds the gblcnt_mutex lock, is permitted to transition from the IDLE state, as indicated by the green color. The slowpath then sends a signal to each thread, and the corresponding signal handler checks the corresponding thread’s theft and counting variables. If the theft state is not REQ, then the signal handler is not permitted to change the state, and therefore simply returns. Otherwise, if the counting variable is set, indicating that the current thread’s fastpath is in progress, the signal handler sets the theft state to ACK, otherwise to READY.

If the theft state is ACK, only the fastpath is permitted to change the theft state, as indicated by the blue color. When the fastpath completes, it sets the theft state to READY.

Once the slowpath sees a thread’s theft state is READY, the slowpath is permitted to steal that thread’s count. The slowpath then sets that thread’s theft state to IDLE.

Quick Quiz 4.38: In Figure 4.19, why is the REQ theft state colored blue?

Quick Quiz 4.39: In Figure 4.19, what is the point of having separate REQ and ACK theft states? Why not simplify the state machine by collapsing them into a single state? Then whichever of the signal handler or the fastpath gets there first could set the state to READY.

4.4.4 Signal-Theft Limit Counter Implementation

Figure 4.20 (count_lim_sig.c) shows the data structures used by the signal-theft based counter implementation. Lines 1-7 define the states and values for the per-thread theft state machine described in the preceding section. Lines 8-17 are similar to earlier implementations, with the addition of lines 14 and 15 to allow remote ac-
4.4. EXACT LIMIT COUNTERS

1 #define THEFT_IDLE 0
2 #define THEFT_REQ 1
3 #define THEFT_ACK 2
4 #define THEFT_READY 3

5 int __thread theft = THEFT_IDLE;
6 int __thread counting = 0;
7 unsigned long __thread counter = 0;
8 unsigned long __thread countermax = 0;
9 unsigned long globalcountmax = 10000;
10 unsigned long globalcount = 0;
11 unsigned long globalreserve = 0;
12 unsigned long *counterp[NR_THREADS] = { NULL };
13 unsigned long *countermaxp[NR_THREADS] = { NULL };
14 int *theftp[NR_THREADS] = { NULL };
15 DEFINE_SPINLOCK(gblcnt_mutex);
16 #define MAX_COUNTERMAX 100

Figure 4.20: Signal-Theft Limit Counter Data

cess to a thread's countermax and theft variables, respectively.

Figure 4.21 shows the functions responsible for migrating counts between per-thread variables and the global variables. Lines 1-7 shows global_count(), which is identical to earlier implementations. Lines 9-19 shows flush_local_count_sig(), which is the signal handler used in the theft process. Lines 11 and 12 check to see if the theft state is REQ, and, if not returns without change. Line 13 executes a memory barrier to ensure that the sampling of the theft variable happens before any change to that variable. Line 14 sets the theft state to ACK, and, if line 15 sees that this thread’s fastpaths are not running, line 16 sets the theft state to READY.

Quick Quiz 4.40: In Figure 4.21 function flush_local_count_sig(), why are there ACCESS_ONCE() wrappers around the uses of the theft per-thread variable?

Lines 21-49 shows flush_local_count(), which is called from the slowpath to flush all threads’ local counts. The loop spanning lines 26-34 advances the theft state for each thread that has local count, and also sends that thread a signal. Line 27 skips any non-existent threads. Otherwise, line 28 checks to see if the current thread holds any local count, and, if not, line 29 sets the thread’s theft state to READY and line 28 skips to the next thread. Otherwise, line 32 sets the thread’s theft state to REQ and line 29 sends the thread a signal.

Quick Quiz 4.41: In Figure 4.21, why is it safe for line 28 to directly access the other thread’s countermax variable?

Quick Quiz 4.42: In Figure 4.21, why doesn’t line 33 check for the current thread sending itself a signal?

Quick Quiz 4.43: The code in Figure 4.21, works with

1 static void globalize_count(void)
2 {
3 globalcount += counter;
4 counter = 0;
5 globalreserve -= countermax;
6 countermax = 0;
7 }
8
9 static void flush_local_count_sig(int unused)
10 {
11 if (ACCESS_ONCE(theft) != THEFT_REQ)
12 return;
13 smp_mb();
14 ACCESS_ONCE(theft) = THEFT_ACK;
15 if (!counting) {
16 ACCESS_ONCE(theft) = THEFT_READY;
17 }
18 smp_mb();
19 }
20
21 static void flush_local_count(void)
22 {
23 int t;
24 thread_id_t tid;
25
26 for_each_tid(t, tid)
27 if (theftp[t] != NULL) {
28 if (*countermaxp[t] == 0) {
29 ACCESS_ONCE(*theftp[t]) = THEFT_READY;
30 continue;
31 }
32 ACCESS_ONCE(*theftp[t]) = THEFT_REQ;
33 pthread_kill(tid, SIGUSR1);
34 }
35 for_each_tid(t, tid) {
36 if (theftp[t] == NULL)
37 continue;
38 while (ACCESS_ONCE(*theftp[t]) != THEFT_READY) {
39 poll(NULL, 0, 1);
40 if (ACCESS_ONCE(*theftp[t]) == THEFT_REQ)
41 pthread_kill(tid, SIGUSR1);
42 }
43 globalcount += *counterp[t];
44 *counterp[t] = 0;
45 globalreserve -= *countermaxp[t];
46 *countermaxp[t] = 0;
47 ACCESS_ONCE(*theftp[t]) = THEFT_IDLE;
48 }
49 }
50 }
51
52 static void balance_count(void)
53 {
54 countermax = globalcountmax -
55 globalcount - globalreserve;
56 countermax /= num_online_threads();
57 if (countermax > MAX_COUNTERMAX)
58 countermax = MAX_COUNTERMAX;
59 globalreserve += countermax;
60 counter = countermax / 2;
61 if (counter > globalcount)
62 globalcount -= counter;
63 }

Figure 4.21: Signal-Theft Limit Counter Value-Migration Functions
gcc and POSIX. What would be required to make it also conform to the ISO C standard?

The loop spanning lines 35-48 waits until each thread reaches READY state, then steals that thread’s count. Lines 36-37 skip any non-existent threads, and the loop spanning lines 38-42 wait until the current thread’s theft state becomes READY. Line 39 blocks for a millisecond to avoid priority-inversion problems, and if line 40 determines that the thread’s signal has not yet arrived, line 41 resends the signal. Execution reaches line 43 when the thread’s theft state becomes READY, so lines 43-46 do the thieving. Line 47 then sets the thread’s theft state back to IDLE.

Quick Quiz 4.44: In Figure 4.21, why does line 41 resend the signal?

Lines 51-63 show balance_count(), which is similar to that of earlier examples.

Lines 1-36 of Figure 4.22 shows the add_count() function. The fastpath spans lines 5-20, and the slowpath lines 21-35. Line 5 sets the per-thread counting variable to 1 so that any subsequent signal handlers interrupting this thread will set the theft state to ACK rather than READY, allowing this fastpath to complete properly. Line 6 prevents the compiler from reordering any of the fastpath body to precede the setting of counting. Lines 7 and 8 check to see if the per-thread data can accommodate the add_count() and if there is no ongoing theft in progress, and if so line 9 does the fastpath addition and line 10 notes that the fastpath was taken.

In either case, line 12 prevents the compiler from reordering the fastpath body to follow line 13, which permits any subsequent signal handlers to undertake theft. Line 14 again disables compiler reordering, and then line 15 checks to see if the signal handler deferred the theft state-change to READY, and, if so, line 16 executes a memory barrier to ensure that any CPU that sees line 17 setting state to READY also sees the effects of line 9. If the fastpath addition at line 9 was executed, then line 20 returns success.

Otherwise, we fall through to the slowpath starting at line 21. The structure of the slowpath is similar to those of earlier examples, so its analysis is left as an exercise to the reader. Similarly, the structure of sub_count() on lines 38-71 is the same as that of add_count(), so the analysis of sub_count() is also left as an exercise for the reader, as is the analysis of read_count() in Figure 4.23.

Lines 1-12 of Figure 4.24 show count_init(), which set up flush_local_count_sig() as the

```c
1 int add_count(unsigned long delta) 2 { 3 int fastpath = 0; 4 5 counting = 1; 6 barrier(); 7 if (countermax - counter >= delta && 8 ACCESS_ONCE(theft) <= THEFT_REQ) { 9 counter += delta; 10 fastpath = 1; 11 } 12 barrier(); 13 counting = 0; 14 barrier(); 15 if (ACCESS_ONCE(theft) == THEFT_ACK) { 16 smp_mb(); 17 ACCESS_ONCE(theft) = THEFT_READY; 18 } 19 if (fastpath) 20 return 1; 21 spin_lock(&gblcnt_mutex); 22 globalize_count(); 23 if (globalcountmax - globalcount - 24 globalreserve < delta) { 25 flush_local_count(); 26 if (globalcountmax - globalcount - 27 globalreserve < delta) { 28 spin_unlock(&gblcnt_mutex); 29 return 0; 30 } 31 } 32 globalcount += delta; 33 balance_count(); 34 spin_unlock(&gblcnt_mutex); 35 return 1; 36 } 37
38 int sub_count(unsigned long delta) 39 { 40 int fastpath = 0; 41 42 counting = 1; 43 barrier(); 44 if (counter >= delta && 45 ACCESS_ONCE(theft) <= THEFT_REQ) { 46 counter -= delta; 47 fastpath = 1; 48 } 49 barrier(); 50 counting = 0; 51 barrier(); 52 if (ACCESS_ONCE(theft) == THEFT_ACK) { 53 smp_mb(); 54 ACCESS_ONCE(theft) = THEFT_READY; 55 } 56 if (fastpath) 57 return 1; 58 spin_lock(&gblcnt_mutex); 59 globalize_count(); 60 if (globalcount < delta) { 61 flush_local_count(); 62 if (globalcount < delta) { 63 spin_unlock(&gblcnt_mutex); 64 return 0; 65 } 66 } 67 globalcount -= delta; 68 balance_count(); 69 spin_unlock(&gblcnt_mutex); 70 return 1; 71 }
```

Figure 4.22: Signal-Theft Limit Counter Add and Subtract Functions
4.5. APPLYING SPECIALIZED PARALLEL COUNTERS

```c
1    unsigned long read_count(void)
2    {           
3       int t;  
4     unsigned long sum;  
5     spin_lock(&gblcnt_mutex);  
6     sum = globalcount;  
7     for_each_thread(t)
8       if (counterp[t] != NULL)
9          sum += *counterp[t];
10    spin_unlock(&gblcnt_mutex);  
11    return sum;
12 }  
```

Figure 4.23: Signal-Theft Limit Counter Read Function

```c
1    void count_init(void)
2    {                            
3       struct sigaction sa;       
4       sa.sa_handler = flush_local_count_sig;
5       sigemptyset(&sa.sa_mask);
6       sa.sa_flags = 0;
7       if (sigaction(SIGUSR1, &sa, NULL) != 0) {
8          perror("sigaction");
9          exit(-1);
10     }
11 }  
```

Figure 4.24: Signal-Theft Limit Counter Initialization Functions

The signal-theft implementation runs more than twice as fast as the atomic implementation on my Intel Core Duo laptop. Is it always preferable? The signal-theft implementation would be vastly preferable on Pentium-4 systems, given their slow atomic instructions, but the old 80386-based Sequent Symmetry systems would do much better with the shorter path length of the atomic implementation. If ultimate performance is of the essence, you will need to measure them both on the system that your application is to be deployed on.

This is but one reason why high-quality APIs are so important: they permit implementations to be changed as required by ever-changing hardware performance characteristics.

Quick Quiz 4.45: What if you want an exact limit counter to be exact only for its lower limit? ■

4.5 Applying Specialized Parallel Counters

Although the exact limit counter implementations in Section 4.4 can be very useful, they are not much help if the counter’s value remains near zero at all times, as it might when counting the number of outstanding accesses to an I/O device. The high overhead of such near-zero counting is especially painful given that we normally don’t care how many references there are. As noted in the removable I/O device access-count problem on page 29, the number of accesses is irrelevant except in those rare cases when someone is actually trying to remove the device.

One simple solution to this problem is to add a large “bias” (for example, one billion) to the counter in order to ensure that the value is far enough from zero that the counter can operate efficiently. When someone wants to remove the device, this bias is subtracted from the counter value. Counting the last few accesses will be quite inefficient, but the important point is that the many prior accesses will have been counted at full speed.

Quick Quiz 4.46: What else had you better have done
when using a biased counter?

Although a biased counter can be quite helpful and useful, it is only a partial solution to the removable I/O device access-count problem called out on page 29. When attempting to remove a device, we must not only know the precise number of current I/O accesses, we also need to prevent any future accesses from starting. One way to accomplish this is to read-acquire a reader-writer lock when updating the counter, and to write-acquire that same reader-writer lock when checking the counter. Code for doing I/O might be as follows:

```c
1 read_lock(&mylock);
2 if (removing) {
3     read_unlock(&mylock);
4     cancel_io();
5 } else {
6     add_count(1);
7     read_unlock(&mylock);
8     do_io();
9     sub_count(1);
10 }
```

Line 1 read-acquires the lock, and either line 3 or 7 releases it. Line 2 checks to see if the device is being removed, and, if so, line 3 releases the lock and line 4 cancels the I/O, or takes whatever action is appropriate given that the device is to be removed. Otherwise, line 6 increments the access count, line 7 releases the lock, line 8 performs the I/O, and line 9 decrements the access count.

Quick Quiz 4.47: This is ridiculous! We are read-acquiring a reader-writer lock to update the counter? What are you playing at???

The code to remove the device might be as follows:

```c
1 write_lock(&mylock);
2 removing = 1;
3 sub_count(mybias);
4 write_unlock(&mylock);
5 while (read_count() != 0) {
6     poll(NULL, 0, 1);
7 }
8 remove_device();
```

Line 1 write-acquires the lock and line 4 releases it. Line 2 notes that the device is being removed, and the loop spanning lines 5-7 wait for any I/O operations to complete. Finally, line 8 does any additional processing needed to prepare for device removal.

Quick Quiz 4.48: What other issues would need to be accounted for in a real system?

### 4.6 Parallel Counting Discussion

This chapter has presented the reliability, performance, and scalability problems with traditional counting primitives. The C-language ++ operator is not guaranteed to function reliably in multithreaded code, and atomic operations to a single variable neither perform nor scale well. This chapter has also presented a number of counting algorithms that perform and scale extremely well in certain special cases.

Table 4.1 shows the performance of the three parallel statistical counting algorithms. All three algorithms provide perfect linear scalability for updates. The per-thread-variable implementation is significantly faster on updates than the array-based implementation, but is slower at reads, and suffers severe lock contention when there are many parallel readers. This contention can be addressed using techniques introduced in Chapter 8, as shown on the last row of Table 4.1.

Quick Quiz 4.49: On the `count_stat.c` row of Table 4.1, we see that the update side scales linearly with the number of threads. How is that possible given that the more threads there are, the more per-thread counters must be summed up?

Quick Quiz 4.50: Even on the last row of Table 4.1, the read-side performance of these statistical counter implementations is pretty horrible. So why bother with them?

Figure 4.2 shows the performance of the parallel limit-counting algorithms. Exact enforcement of the limits incurs a substantial performance penalty, although on the Power-5 system this penalty can be reduced by substituting read-side signals for update-side atomic operations. All of these implementations suffer from read-side lock contention in the face of concurrent readers.

Quick Quiz 4.51: Given the performance data shown in Table 4.2, we should always prefer update-side signals over read-side atomic operations, right?

Quick Quiz 4.52: Can advanced techniques be applied to address the lock contention for readers seen in Table 4.2?

The fact that these algorithms only work well in their respective special cases might be considered a major problem with parallel programming in general. After all, the C-language ++ operator works just fine in single-threaded code, and not just for special cases, but in general, right?

This line of reasoning does contain a grain of truth, but is in essence misguided. The problem is not parallelism as such, but rather scalability. To understand this, first
4.6. PARALLEL COUNTING DISCUSSION

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Section</th>
<th>Exact?</th>
<th>Updates</th>
<th>Reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>count_stat.c</td>
<td>4.2.2</td>
<td></td>
<td>40.4 ns</td>
<td>220 ns</td>
</tr>
<tr>
<td>count_end.c</td>
<td>4.2.4</td>
<td></td>
<td>6.7 ns</td>
<td>521 ns</td>
</tr>
<tr>
<td>count_end_rcu.c</td>
<td>9.1</td>
<td></td>
<td>6.7 ns</td>
<td>481 ns</td>
</tr>
</tbody>
</table>

Table 4.1: Statistical Counter Performance on Power-5

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Section</th>
<th>Exact?</th>
<th>Updates</th>
<th>Reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>count_lim.c</td>
<td>4.9</td>
<td>N</td>
<td>9.7 ns</td>
<td>517 ns</td>
</tr>
<tr>
<td>count_lim_app.c</td>
<td>4.3.4</td>
<td>N</td>
<td>6.6 ns</td>
<td>520 ns</td>
</tr>
<tr>
<td>count_lim_atomic.c</td>
<td>4.4.1</td>
<td>Y</td>
<td>56.1 ns</td>
<td>606 ns</td>
</tr>
<tr>
<td>count_lim_sig.c</td>
<td>4.4.4</td>
<td>Y</td>
<td>17.5 ns</td>
<td>520 ns</td>
</tr>
</tbody>
</table>

Table 4.2: Limit Counter Performance on Power-5

consider the C-language ++ operator. The fact is that it does not work in general, only for a restricted range of numbers. If you need to deal with 1,000-digit decimal numbers, the C-language ++ operator will not work for you.

Quick Quiz 4.53: The ++ operator works just fine for 1,000-digit numbers! Haven’t you heard of operator overloading???

This problem is not specific to arithmetic. Suppose you need to store and query data. Should you use an ASCII file, XML, a relational database, a linked list, a dense array, a B-tree, a radix tree, or any of the plethora of other data structures and environments that permit data to be stored and queried? It depends on what you need to do, how fast you need it done, and how large your data set is.

Similarly, if you need to count, your solution will depend on how large of numbers you need to work with, how many CPUs need to be manipulating a given number concurrently, how the number is to be used, and what level of performance and scalability you will need.

Nor is this problem specific to software. The design for a bridge meant to allow people to walk across a small brook might be a simple as a plank thrown across the brook. But this solution of using a plank does not scale. You would probably not use a plank to span the kilometers-wide mouth of the Columbia River, nor would such a design be advisable for bridges carrying concrete trucks. In short, just as bridge design must change with increasing span and load, so must software design change as the number of CPUs increases.

The examples in this chapter have shown that an important tool permitting large numbers of CPUs to be brought to bear is partitioning. Whether fully partitioned, as in the statistical counters discussed in Section 4.2, or partially partitioned as in the limit counters discussed in Sections 4.3 and 4.4. Partitioning will be considered in far greater depth in the next chapter.

Quick Quiz 4.54: But if we are going to have to partition everything, why bother with shared-memory multithreading? Why not just partition the problem completely and run as multiple processes, each in its own address space?
Chapter 5

Partitioning and Synchronization Design

This chapter describes how to design software to take advantage of the multiple CPUs that are increasingly appearing in commodity systems. It does this by presenting a number of idioms, or “design patterns” that can help you balance performance, scalability, and response time. As noted in earlier chapters, the most important decision you will make when creating parallel software is how to carry out the partitioning. Correctly partitioned problems lead to simple, scalable, and high-performance solutions, while poorly partitioned problems result in slow and complex solutions.

@@@ roadmap @@@

5.1 Partitioning Exercises

This section uses a pair of exercises (the classic Dining Philosophers problem and a double-ended queue) to demonstrate the value of partitioning.

5.1.1 Dining Philosophers Problem

Figure 5.1 shows a diagram of the classic Dining Philosophers problem [Dij71]. This problem features five philosophers who do nothing but think and eat a “very difficult kind of spaghetti” which requires two forks to eat. A given philosopher is permitted to use only the forks to his or her immediate right and left, and once a philosopher picks up a fork, he or she will not put it down until sated.1

The object is to construct an algorithm that, quite literally, prevents starvation. One starvation scenario would be if all of the philosophers picked up their leftmost forks simultaneously. Because none of them would put down their fork until after they ate, and because none of them may pick up their second fork until at least one has finished eating, they all starve.

Dijkstra’s solution used a global semaphore, which works fine assuming negligible communications delays, an assumption that has become invalid in the ensuing decades. Therefore, recent solutions number the forks as shown in Figure 5.2. Each philosopher picks up the lowest-numbered fork next to his or her plate, then picks up the highest-numbered fork. The philosopher sitting in the uppermost position in the diagram thus picks up the leftmost fork first, then the rightmost fork, while the rest of the philosophers instead pick up their rightmost fork first. Because two of the philosophers will attempt to pick up fork 1 first, and because only one of those two philosophers will succeed, there will be five forks available to four philosophers. At least one of these four will be guaranteed to have two forks, and thus be able to

---

1 Readers who have difficulty imagining a food that requires two forks are invited to instead think in terms of chopsticks.
Figure 5.2: Dining Philosophers Problem, Textbook Solution

This general technique of numbering resources and acquiring them in numerical order is heavily used as a deadlock-prevention technique. However, it is easy to imagine a sequence of events that will result in only one philosopher eating at a time even though all are hungry:

1. P2 picks up fork 1, preventing P1 from taking a fork.
2. P3 picks up fork 2.
3. P4 picks up fork 3.
4. P5 picks up fork 4.
5. P5 picks up fork 5 and eats.
6. P5 puts down forks 4 and 5.
7. P4 picks up fork 4 and eats.

Please think about ways of partitioning the Dining Philosophers Problem before reading further.
5.1. PARTITIONING EXERCISES

5.1. PARTITIONING EXERCISES

5.1. PARTITIONING EXERCISES

Figure 5.3: Dining Philosophers Problem, Partitioned

One approach is shown in Figure 5.3, which includes four philosophers rather than five to better illustrate the partition technique. Here the upper and rightmost philosophers share a pair of forks, while the lower and leftmost philosophers share another pair of forks. If all philosophers are simultaneously hungry, at least two will be able to eat concurrently. In addition, as shown in the figure, the forks can now be bundled so that the pair are picked up and put down simultaneously, simplifying the acquisition and release algorithms.

Quick Quiz 5.1: Is there a better solution to the Dining Philosophers Problem? ■

This is an example of “horizontal parallelism” [Inm85] or “data parallelism”, so named because there is no dependency among the philosophers. In a data-processing system, a given item of data would pass through only one of a replicated set of software components.

Quick Quiz 5.2: And in just what sense can this “horizontal parallelism” be said to be “horizontal”? ■

5.1.2 Double-Ended Queue

A double-ended queue is a data structure containing a list of elements that may be inserted or removed from either end [Knu73]. It has been claimed that a lock-based implementation permitting concurrent operations on both ends of the double-ended queue is difficult [Gro07]. This section shows how a partitioning design strategy can result in a reasonably simple implementation, looking at three general approaches in the following sections.

5.1.2.1 Left- and Right-Hand Locks

One seemingly straightforward approach would be to have a left-hand lock for left-hand-end enqueue and dequeue operations along with a right-hand lock for right-hand-end operations, as shown in Figure 5.4. However, the problem with this approach is that the two locks’ domains must overlap when there are fewer than four elements on the list. This overlap is due to the fact that removing any given element affects not only that element, but also its left- and right-hand neighbors. These domains are indicated by color in the figure, with blue indicating the domain of the left-hand lock, red indicating the domain of the right-hand lock, and purple indicating overlapping domains. Although it is possible to create an algorithm that works this way, the fact that it has no fewer than five special cases should raise a big red flag, especially given that concurrent activity at the other end of the list can shift the queue from one special case to another at any time. It is far better to consider other designs.
5.1.2.2 Compound Double-Ended Queue

One way of forcing non-overlapping lock domains is shown in Figure 5.5. Two separate double-ended queues are run in tandem, each protected by its own lock. This means that elements must occasionally be shuttled from one of the double-ended queues to the other, in which case both locks must be held. A simple lock hierarchy may be used to avoid deadlock, for example, always acquiring the left-hand lock before acquiring the right-hand lock. This will be much simpler than applying two locks to the same double-ended queue, as we can unconditionally left-enqueue elements to the left-hand queue and right-enqueue elements to the right-hand queue. The main complication arises when dequeuing from an empty queue, in which case it is necessary to:

1. If holding the right-hand lock, release it and acquire the left-hand lock.
2. Acquire the right-hand lock.
3. Rebalance the elements across the two queues.
4. Remove the required element if there is one.
5. Release both locks.

Quick Quiz 5.3: In this compound double-ended queue implementation, what should be done if the queue has become non-empty while releasing and reacquiring the lock?

The rebalancing operation might well shuttle a given element back and forth between the two queues, wasting time and possibly requiring workload-dependent heuristics to obtain optimal performance. Although this might well be the best approach in some cases, it is interesting to try for an algorithm with greater determinism.

5.1.2.3 Hashed Double-Ended Queue

One of the simplest and most effective ways to deterministically partition a data structure is to hash it. It is possible to trivially hash a double-ended queue by assigning each element a sequence number based on its position in the list, so that the first element left-enqueued into an empty queue is numbered zero and the first element right-enqueued into an empty queue is numbered one. A series of elements left-enqueued into an otherwise-idle queue would be assigned decreasing numbers (-1, -2, -3, ...), while a series of elements right-enqueued into an otherwise-idle queue would be assigned increasing numbers (2, 3, 4, ...). A key point is that it is not necessary to actually represent a given element’s number, as this number will be implied by its position in the queue.

Given this approach, we assign one lock to guard the left-hand index, one to guard the right-hand index, and one lock for each hash chain. Figure 5.6 shows the resulting data structure given four hash chains. Note that the lock domains do not overlap, and that deadlock is avoided by acquiring the index locks before the chain locks, and by never acquiring more than one lock of each type (index or chain) at a time.

Each hash chain is itself a double-ended queue, and in this example, each holds every fourth element. The uppermost portion of Figure 5.7 shows the state after a single element (“R1”) has been right-enqueued, with the right-hand index having been incremented to reference hash chain 2. The middle portion of this same figure shows the state after three more elements have been right-enqueued. As you can see, the indexes are back to their initial states, however, each hash chain is now non-empty. The lower portion of this figure shows the state after three additional elements have been left-enqueued and an additional element has been right-enqueued.

From the last state shown in Figure 5.7, a left-dequeue operation would return element “L-2” and left the left-hand index referencing hash chain 2, which would then contain only a single element (“R2”). In this state, a left-enqueue running concurrently with a right-enqueue would result in lock contention, but the probability of such contention can be arbitrarily reduced by using a larger
Figure 5.7: Hashed Double-Ended Queue After Insertions

Figure 5.8: Hashed Double-Ended Queue With 12 Elements

Figure 5.9: Lock-Based Parallel Double-Ended Queue Data Structure

of course use padding or special alignment directives to avoid false sharing.

Figure 5.10 shows the implementation of the enqueue and dequeue functions.2 Discussion will focus on the left-hand operations, as the right-hand operations are trivially derived from them.

Lines 1-13 show pdeq_dequeue_l(), which left-dequeues and returns an element if possible, returning NULL otherwise. Line 6 acquires the left-hand spinlock, and line 7 computes the index to be dequeued from. Line 8 dequeues the element, and, if line 9 finds the result to be non-NULL, line 10 records the new left-hand index. Either way, line 11 releases the lock, and, finally, line 12 returns the element if there was one, or NULL otherwise.

Lines 15-24 shows pdeq_enqueue_l(), which left-enqueues the specified element. Line 19 acquires the left-hand lock, and line 20 picks up the left-hand index. Line 21 left-enqueues the specified element onto the double-ended queue indexed by the left-hand index. Line 22 updates the left-hand index, and finally line 23 releases the lock.

As noted earlier, the right-hand operations are completely analogous to their left-handed counterparts.

Quick Quiz 5.4: Is the hashed double-ended queue a good solution? Why or why not? ■

2 One could easily create a polymorphic implementation in any number of languages, but doing so is left as an exercise for the reader.
struct element *pdeq_dequeue_l(struct pdeq *d)
{
  struct element *e;
  int i;

  spin_lock(&d->llock);
  i = moveright(d->lidx);
  e = deq_dequeue_l(&d->bkt[i]);
  if (e != NULL)
    d->lidx = i;
  spin_unlock(&d->llock);
  return e;
}

void pdeq_enqueue_l(struct element *e, struct pdeq *d)
{
  int i;

  spin_lock(&d->llock);
  i = d->lidx;
  deq_enqueue_l(e, &d->bkt[i]);
  d->lidx = moveleft(d->lidx);
  spin_unlock(&d->llock);
}

struct element *pdeq_dequeue_r(struct pdeq *d)
{
  struct element *e;
  int i;

  spin_lock(&d->rlock);
  i = moveleft(d->ridx);
  e = deq_dequeue_r(&d->bkt[i]);
  if (e != NULL)
    d->ridx = i;
  spin_unlock(&d->rlock);
  return e;
}

void pdeq_enqueue_r(struct element *e, struct pdeq *d)
{
  int i;

  spin_lock(&d->rlock);
  i = d->ridx;
  deq_enqueue_r(e, &d->bkt[i]);
  d->ridx = moveright(d->lidx);
  spin_unlock(&d->rlock);
}

Figure 5.10: Lock-Based Parallel Double-Ended Queue Implementation
5.1. PARTITIONING EXERCISES

5.1.2.4 Compound Double-Ended Queue Revisited

This section revisits the compound double-ended queue, using a trivial rebalancing scheme that moves all the elements from the non-empty queue to the now-empty queue.

Quick Quiz 5.5: Move all the elements to the queue that became empty? In what possible universe is this braindead solution in any way optimal???

In contrast to the hashed implementation presented in the previous section, the compound implementation will build on a sequential implementation of a double-ended queue that uses neither locks nor atomic operations.

Figure 5.11 shows the implementation. Unlike the hashed implementation, this compound implementation is asymmetric, so that we must consider the pdeq_dequeue_l() and pdeq_dequeue_r() implementations separately.

Quick Quiz 5.6: Why can’t the compound parallel double-ended queue implementation be symmetric??

The pdeq_dequeue_l() implementation is shown on lines 1-16 of the figure. Line 6 acquires the left-hand lock, which line 14 releases. Line 7 attempts to left-dequeue an element from the left-hand underlying double-ended queue, and, if successful, skips lines 8-13 to simply return this element. Otherwise, line 9 acquires the right-hand lock, line 10 left-dequeues an element from the right-hand queue, and line 11 moves any remaining elements on the right-hand queue to the left-hand queue, and line 12 releases the right-hand lock. The element, if any, that was dequeued on line 10 will be returned.

The pdeq_dequeue_r() implementation is shown on lines 18-38 of the figure. As before, line 23 acquires the right-hand lock (and line 36 releases it), and line 24 attempts to right-dequeue an element from the right-hand queue, and, if successful, skips lines 24-35 to simply return this element. However, if line 25 determines that there was no element to dequeue, line 26 releases the right-hand lock and lines 27-28 acquire both locks in the proper order. Line 29 then attempts to right-dequeue an element from the right-hand list again, and if line 30 determines that this second attempt has failed, line 31 right-dequeues an element from the left-hand queue (if there is one available) and line 32 moves any remaining elements from the left-hand queue to the right-hand queue. Either way, line 34 releases the left-hand lock.

Quick Quiz 5.7: Why is it necessary to retry the right-dequeue operation on line 29 of Figure 5.11??

Quick Quiz 5.8: Surely the left-hand lock must sometimes be available!!! So why is it necessary that line 26 of Figure 5.11 unconditionally release the right-hand lock?

The pdeq_enqueue_l() implementation is shown on lines 40-47 of Figure 5.11. Line 44 acquires the left-hand spinlock, line 45 left-enqueues the element onto the left-hand queue, and finally line 46 releases the lock. The pdeq_enqueue_r() implementation (shown on lines 49-56) is quite similar.

5.1.2.5 Double-Ended Queue Discussion

The compound implementation is somewhat more complex than the hashed variant presented in Section 5.1.2.3, but is still reasonably simple. Of course, a more intelligent rebalancing scheme could be arbitrarily complex, but the simple scheme shown here has been shown to perform well compared to software alternatives [DCW+11] and even compared to algorithms using hardware assist [DLM+10]. Nevertheless, the best we can hope for from such a scheme is 2x scalability, as at most two threads can be holding the dequeue’s locks concurrently.

The key point is that there can be significant overhead enqueuing to or dequeuing from a shared queue.

5.1.3 Partitioning Example Discussion

The optimal solution to the dining philosophers problem given in the answer to the Quick Quiz in Section 5.1.1 is an excellent example of “horizontal parallelism” or “data parallelism”. The synchronization overhead in this case is nearly (or even exactly) zero. In contrast, the double-ended queue implementations are examples of “vertical parallelism” or “pipelining”, given that data moves from one thread to another. The tighter coordination required for pipelining in turn requires larger units of work to obtain a given level of efficiency.

Quick Quiz 5.9: The tandem double-ended queue runs about twice as fast as the hashed double-ended queue, even when I increase the size of the hash table to an insanely large number. Why is that??

Quick Quiz 5.10: Is there a significantly better way of handling concurrency for double-ended queues??

These two examples show just how powerful partitioning can be in devising parallel algorithms. However, these example beg for more and better design criteria for parallel programs, a topic taken up in the next section.
Figure 5.11: Compound Parallel Double-Ended Queue Implementation
5.2 Design Criteria

Section 1.2 called out the three parallel-programming goals of performance, productivity, and generality. However, more detailed design criteria are required to actually produce a real-world design, a task taken up in this section. This being the real world, these criteria often conflict to a greater or lesser degree, requiring that the designer carefully balance the resulting tradeoffs.

As such, these criteria may be thought of as the “forces” acting on the design, with particularly good tradeoffs between these forces being called “design patterns” [Ale79, GHJV95].

The design criteria for attaining the three parallel-programming goals are speedup, contention, overhead, read-to-write ratio, and complexity:

**Speedup:** As noted in Section 1.2, increased performance is the major reason to go to all of the time and trouble required to parallelize it. Speedup is defined to be the ratio of the time required to run a sequential version of the program to the time required to run a parallel version.

**Contention:** If more CPUs are applied to a parallel program than can be kept busy by that program, the excess CPUs are prevented from doing useful work by contention. This may be lock contention, memory contention, or a host of other performance killers.

**Work-to-Synchronization Ratio:** A uniprocessor, single-threaded, non-preemptible, and non-interruptible version of a given parallel program would not need any synchronization primitives. Therefore, any time consumed by these primitives (including communication cache misses as well as message latency, locking primitives, atomic instructions, and memory barriers) is overhead that does not contribute directly to the useful work that the program is intended to accomplish. Note that the important measure is the relationship between the synchronization overhead and the overhead of the code in the critical section, with larger critical sections able to tolerate greater synchronization overhead. The work-to-synchronization ratio is related to the notion of synchronization efficiency.

**Read-to-Write Ratio:** A data structure that is rarely updated may often be replicated rather than partitioned, and furthermore may be protected with asymmetric synchronization primitives that reduce readers’ synchronization overhead at the expense of that of writers, thereby reducing overall synchronization overhead. Corresponding optimizations are possible for frequently updated data structures, as discussed in Chapter 4.

**Complexity:** A parallel program is more complex than an equivalent sequential program because the parallel program has a much larger state space than does the sequential program, although these larger state spaces can in some cases be easily understood given sufficient regularity and structure. A parallel programmer must consider synchronization primitives, messaging, locking design, critical-section identification, and deadlock in the context of this larger state space.

This greater complexity often translates to higher development and maintenance costs. Therefore, budgetary constraints can limit the number and types of modifications made to an existing program, since a given degree of speedup is worth only so much time and trouble. Furthermore, there may be potential sequential optimizations that are cheaper and more effective than parallelization. As noted in Section 1.2.1, parallelization is but one performance optimization of many, and is furthermore an optimization that applies most readily to CPU-based bottlenecks.

These criteria will act together to enforce a maximum speedup. The first three criteria are deeply interrelated, so the remainder of this section analyzes these interrelationships.\(^4\)

Note that these criteria may also appear as part of the requirements specification. For example, speedup may act as a desideratum (“the faster, the better”) or as an absolute requirement of the workload, or “context” (“the system must support at least 1,000,000 web hits per second”).

An understanding of the relationships between these design criteria can be very helpful when identifying appropriate design tradeoffs for a parallel program.

1. The less time a program spends in critical sections, the greater the potential speedup. This is a consequence of Amdahl’s Law [Amd67] and of the fact

\(^3\) Either by masking interrupts or by being oblivious to them.

\(^4\) A real-world parallel system will be subject to many additional design criteria, such as data-structure layout, memory size, memory-hierarchy latencies, and bandwidth limitations.
that only one CPU may execute within a given critical section at a given time.

2. The fraction of time that the program spends in a given exclusive critical section must be much less than the reciprocal of the number of CPUs for the actual speedup to approach the number of CPUs. For example, a program running on 10 CPUs must spend much less than one tenth of its time in the most-restrictive critical section if it is to scale at all well.

3. Contention effects will consume the excess CPU and/or wallclock time should the actual speedup be less than the number of available CPUs. The larger the gap between the number of CPUs and the actual speedup, the less efficiently the CPUs will be used. Similarly, the greater the desired efficiency, the smaller the achievable speedup.

4. If the available synchronization primitives have high overhead compared to the critical sections that they guard, the best way to improve speedup is to reduce the number of times that the primitives are invoked (perhaps by batching critical sections, using data ownership, using RCU, or by moving toward a more coarse-grained design such as code locking).

5. If the critical sections have high overhead compared to the primitives guarding them, the best way to improve speedup is to increase parallelism by moving to reader/writer locking, data locking, RCU, or data ownership.

6. If the critical sections have high overhead compared to the primitives guarding them and the data structure being guarded is read much more often than modified, the best way to increase parallelism is to move to reader/writer locking or RCU.

7. Many changes that improve SMP performance, for example, reducing lock contention, also improve response times.

5.3 Synchronization Granularity

Figure 5.12 gives a pictorial view of different levels of synchronization granularity, each of which is described in one of the following sections. These sections focus primarily on locking, but similar granularity issues arise with all forms of synchronization.

5.3.1 Sequential Program

If the program runs fast enough on a single processor, and has no interactions with other processes, threads, or interrupt handlers, you should remove the synchronization primitives and spare yourself their overhead and complexity. Some years back, there were those who would argue that Moore’s Law would eventually force all programs into this category. However, given the cessation in rate of CPU MIPS and clock-frequency growth in Intel CPUs since the year 2003, as can be seen in Figure 5.13 increasing performance will increasingly require parallelism. The debate as to whether this new trend will result in single chips with thousands of CPUs will not be settled soon, but given that Paul is typing this sentence on a dual-core laptop, the age of SMP does seem to be upon us. It is also important to note that Ethernet bandwidth is continuing to grow, as shown in Figure 5.14. This growth will motivate multithreaded servers in order to handle the communications load.

Please note that this does not mean that you should code each and every program in a multi-threaded manner. Again, if a program runs quickly enough on a single CPU.

\[^5\] This plot shows clock frequencies for newer CPUs theoretically capable of retiring one or more instructions per clock, and MIPS for older CPUs requiring multiple clocks to execute even the simplest instruction. The reason for taking this approach is that the newer CPUs’ ability to retire multiple instructions per clock is typically limited by memory-system performance.
5.3. SYNCHRONIZATION GRANULARITY

5.3.2 Code Locking

Code locking is the simplest locking design, using only global locks. It is especially easy to retrofit an existing program to use code locking in order to run it on a multiprocessor. If the program has only a single shared resource, code locking will even give optimal performance. However, many of the larger and more complex programs require much of the execution to occur in critical sections, which in turn causes code locking to sharply limit their scalability.

Therefore, you should use code locking on programs that spend only a small fraction of their execution time in critical sections or from which only modest scaling is required. In these cases, code locking will provide a relatively simple program that is very similar to its sequential counterpart, as can be seen in Figure 5.16. However, not that the simple return of the comparison in hash_search() in Figure 5.15 has now become three statements due to the need to release the lock before returning.

However, code locking is particularly prone to “lock contention”, where multiple CPUs need to acquire the lock concurrently. SMP programmers who have taken care of groups of small children (or of older people who are acting like children) will immediately recognize the danger of having only one of something, as illustrated in Figure 5.17.

One solution to this problem, named “data locking”, is described in the next section.

5.3.3 Data Locking

Many data structures may be partitioned, with each partition of the data structure having its own lock. Then the critical sections for each part of the data structure can execute in parallel, although only one instance of the critical section for a given part could be executing at a given time. Use data locking when contention must be reduced, and where synchronization overhead is not limiting speedups. Data locking reduces contention by distributing the instances of the overly-large critical section into multiple critical sections, for example, maintaining per-hash-bucket critical sections in a hash table, as shown.

---

6 The examples in this section are taken from Hart et al. [HMB06], adapted for clarity by gathering code related code from multiple files.
7 If your program instead has locks in data structures, or, in the case of Java, uses classes with synchronized instances, you are instead using “data locking”, described in Section 5.3.3.
CHAPTER 5. PARTITIONING AND SYNCHRONIZATION DESIGN

Figure 5.15: Sequential-Program Hash Table Search

```c
struct hash_table {
    long nbuckets;
    struct node **buckets;
};

typedef struct node {
    unsigned long key;
    struct node *next;
} node_t;

int hash_search(struct hash_table *h, long key) {
    struct node *cur;
    cur = h->buckets[key % h->nbuckets];
    while (cur != NULL) {
        if (cur->key >= key) {
            return (cur->key == key);
        }
        cur = cur->next;
    }
    return 0;
}
```

Figure 5.16: Code-Locking Hash Table Search

```c
spinlock_t hash_lock;

struct hash_table {
    long nbuckets;
    struct node **buckets;
};

typedef struct node {
    unsigned long key;
    struct node *next;
} node_t;

int hash_search(struct hash_table *h, long key) {
    struct node *cur;
    int retval;
    spin_lock(&hash_lock);
    cur = h->buckets[key % h->nbuckets];
    while (cur != NULL) {
        if (cur->key >= key) {
            return (cur->key == key);
        }
        cur = cur->next;
    }
    spin_unlock(&hash_lock);
    return 0;
}
```

Figure 5.17: Lock Contention

in Figure 5.18. The increased scalability again results in increased complexity in the form of an additional data structure, the `struct bucket`.

In contrast with the contentious situation shown in Figure 5.17, data locking helps promote harmony, as illustrated by Figure 5.19 — and in parallel programs, this *almost* always translates into increased performance and scalability. For this reason, data locking was heavily used by Sequent in both its DYNIX and DYNIX/ptx operating systems [BK85, Inm85, Gar90, Dov90, MD92, MG92, MS93].

However, as those how have taken care of small children can again attest, even providing enough to go around is no guarantee of tranquillity. The analogous situation can arise in SMP programs. For example, the Linux kernel maintains a cache of files and directories (called "dcache"). Each entry in this cache has its own lock, but the entries corresponding to the root directory and its direct descendants are much more likely to be traversed than are more obscure entries. This can result in many CPUs contending for the locks of these popular entries, resulting in a situation not unlike that shown in Figure 5.20.

In many cases, algorithms can be designed to reduce the instance of data skew, and in some cases eliminate it entirely (as appears to be possible with the Linux kernel’s dcache [MSS04]). Data locking is often used for partitionable data structures such as hash tables, as well as in situations where multiple entities are each represented
5.3. SYNCHRONIZATION GRANULARITY

Figure 5.18: Data-Locking Hash Table Search

by an instance of a given data structure. The task list in version 2.6.17 of the Linux kernel is an example of the latter, each task structure having its own proc_lock.

A key challenge with data locking on dynamically allocated structures is ensuring that the structure remains in existence while the lock is being acquired. The code in Figure 5.18 finesses this challenge by placing the locks in the statically allocated hash buckets, which are never freed. However, this trick would not work if the hash table were resizeable, so that the locks were now dynamically allocated. In this case, there would need to be some means to prevent the hash bucket from being freed during the time that its lock was being acquired.

Quick Quiz 5.11: What are some ways of preventing a structure from being freed while its lock is being acquired?

5.3.4 Data Ownership

Data ownership partitions a given data structure over the threads or CPUs, so that each thread/CPU accesses its subset of the data structure without any synchronization overhead whatsoever. However, if one thread wishes to access some other thread’s data, the first thread is unable to do so directly. Instead, the first thread must communicate with the second thread, so that the second thread
CHAPTER 5. PARTITIONING AND SYNCHRONIZATION DESIGN

Data ownership might seem arcane, but it is used very frequently:

1. Any variables accessible by only one CPU or thread (such as `auto` variables in C and C++) are owned by that CPU or process.

2. An instance of a user interface owns the corresponding user’s context. It is very common for applications interacting with parallel database engines to be written as if they were entirely sequential programs. Such applications own the user interface and his current action. Explicit parallelism is thus confined to the database engine itself.

3. Parametric simulations are often trivially parallelized by granting each thread ownership of a particular region of the parameter space.

If there is significant sharing, communication between the threads or CPUs can result in significant complexity and overhead. Furthermore, if the most-heavily used data happens to be that owned by a single CPU, that CPU will be a “hot spot”, sometimes with results resembling that shown in Figure 5.20. However, in situations where no sharing is required, data ownership achieves ideal performance, and with code that can be as simple as the sequential-program case shown in Figure 5.15. Such situations are often referred to as “embarrassingly parallel”, and, in the best case, resemble the situation previously shown in Figure 5.19.

Another important instance of data ownership occurs when the data is read-only, in which case, all threads can “own” it via replication.

Data ownership will be presented in more detail in Chapter 7.

5.3.5 Locking Granularity and Performance

This section looks at locking granularity and performance from a mathematical synchronization-efficiency viewpoint. Readers who are uninspired by mathematics might choose to skip this section.

The approach is to use a crude queueing model for the efficiency of synchronization mechanism that operate on a single shared global variable, based on an M/M/1 queue. M/M/1 queueing models are based on an exponentially distributed “inter-arrival rate” $\lambda$ and an exponentially distributed “service rate” $\mu$. The inter-arrival rate $\lambda$ can be thought of as the average number of synchronization operations per second that the system would process if the synchronization were free, in other words, $\lambda$ is an inverse measure of the overhead of each non-synchronization unit of work. For example, if each unit of work was a transaction, if each transaction took one millisecond to process, not counting synchronization overhead, then $\lambda$ would be 1,000 transactions per second.

The service rate $\mu$ is defined similarly, but for the average number of synchronization operations per second that the system would process if the overhead of each transaction was zero, and ignoring the fact that CPUs must wait on each other to complete their increment operations, in other words, $\mu$ can be roughly thought of as the synchronization overhead in absence of contention. For example, some recent computer systems are able to do an atomic increment every 25 nanoseconds or so if all CPUs are doing atomic increments in a tight loop. The value of $\mu$ is therefore about 40,000,000 atomic increments per second.

Of course, the value of $\lambda$ increases with increasing numbers of CPUs, as each CPU is capable of processing
transactions independently (again, ignoring synchronization):

$$\lambda = n\lambda_0$$  \hspace{1cm} (5.1)

where \( n \) is the number of CPUs and \( \lambda_0 \) is the transaction-processing capability of a single CPU. Note that the expected time for a single CPU to execute a single transaction is \( 1/\lambda_0 \).

Because the CPUs have to “wait in line” behind each other to get their chance to increment the single shared variable, we can use the M/M/1 queueing-model expression for the expected total waiting time:

$$T = \frac{1}{\mu - \lambda}$$  \hspace{1cm} (5.2)

Substituting the above value of \( \lambda \):

$$T = \frac{1}{\mu - n\lambda_0}$$  \hspace{1cm} (5.3)

Now, the efficiency is just the ratio of the time required to process a transaction in absence of synchronization to the time required including synchronization:

$$e = \frac{1/\lambda_0}{T + 1/\lambda_0}$$  \hspace{1cm} (5.4)

Substituting the above value for \( T \) and simplifying:

$$e = \frac{\mu}{\lambda_0 (n - 1)} - \frac{n}{\lambda_0 (n - 1)}$$  \hspace{1cm} (5.5)

But the value of \( \mu/\lambda_0 \) is just the ratio of the time required to process the transaction (absent synchronization overhead) to that of the synchronization overhead itself (absent contention). If we call this ratio \( f \), we have:

$$e = \frac{f - n}{f - (n - 1)}$$  \hspace{1cm} (5.6)

Figure 5.21 plots the synchronization efficiency \( e \) as a function of the number of CPUs/threads \( n \) for a few values of the overhead ratio \( f \). For example, again using the 25-nanosecond atomic increment, the \( f = 10 \) line corresponds to each CPU attempting an atomic increment every 250 nanoseconds, and the \( f = 100 \) line corresponds to each CPU attempting an atomic increment every 2.5 microseconds, which in turn corresponds to several thousand instructions. Given that each trace drops off sharply with increasing numbers of CPUs or threads, we can conclude that synchronization mechanisms based on atomic manipulation of a single global shared variable will not scale well if used heavily on current commodity hardware. This is a mathematical depiction of the forces leading to the parallel counting algorithms that were discussed in Chapter 4.

The concept of efficiency is useful even in cases having little or no formal synchronization. Consider for example a matrix multiply, in which the columns of one matrix are multiplied (via “dot product”) by the rows of another, resulting in an entry in a third matrix. Because none of these operations conflict, it is possible to partition the columns of the first matrix among a group of threads, with each thread computing the corresponding columns of the result matrix. The threads can therefore operate entirely independently, with no synchronization overhead whatsoever, as is done in matmul.c. One might therefore expect a parallel matrix multiply to have a perfect efficiency of 1.0.

However, Figure 5.22 tells a different story, especially for a 64-by-64 matrix multiply, which never gets above an efficiency of about 0.7, even when running single-threaded. The 512-by-512 matrix multiply’s efficiency is measurably less than 1.0 on as few as 10 threads, and even the 1024-by-1024 matrix multiply deviates noticeably from perfection at a few tens of threads.

**Quick Quiz 5.12:** How can a single-threaded 64-by-64 matrix multiple possibly have an efficiency of less than 1.0? Shouldn’t all of the traces in Figure 5.22 have
efficiency of exactly 1.0 when running on only one thread?

Given these inefficiencies, it is worthwhile to look into more-scalable approaches such as the data locking described in Section 5.3.3 or the parallel-fastpath approach discussed in the next section.

**Quick Quiz 5.13:** How are data-parallel techniques going to help with matrix multiply? It is *already* data parallel!!!

### 5.4 Parallel Fastpath

Fine-grained (and therefore *usually* higher-performance) designs are typically more complex than are coarser-grained designs. In many cases, most of the overhead is incurred by a small fraction of the code [Knu73]. So why not focus effort on that small fraction?

This is the idea behind the parallel-fastpath design pattern, to aggressively parallelize the common-case code path without incurring the complexity that would be required to aggressively parallelize the entire algorithm. You must understand not only the specific algorithm you wish to parallelize, but also the workload that the algorithm will be subjected to. Great creativity and design effort is often required to construct a parallel fastpath.

Parallel fastpath combines different patterns (one for the fastpath, one elsewhere) and is therefore a template pattern. The following instances of parallel fastpath occur often enough to warrant their own patterns, as depicted in Figure 5.23:

1. Reader/Writer Locking (described below in Section 5.4.1).
2. Read-copy update (RCU), which may be used as a high-performance replacement for reader/writer locking, is introduced in Section 8.3, and will not be discussed further in this chapter.
3. Hierarchical Locking ([McK96a]), which is touched upon in Section 5.4.2.
4. Resource Allocator Caches ([McK96a, MS93]). See Section 5.4.3 for more detail.

#### 5.4.1 Reader/Writer Locking

If synchronization overhead is negligible (for example, if the program uses coarse-grained parallelism), and if only a small fraction of the critical sections modify data, then allowing multiple readers to proceed in parallel can greatly increase scalability. Writers exclude both readers and each other. Figure 5.24 shows how the hash search might be implemented using reader-writer locking.

Reader/writer locking is a simple instance of asymmetric locking. Snaman [ST87] describes a more ornate six-mode asymmetric locking design used in several clustered
5.4. PARALLEL FASTPATH

1 rwlock_t hash_lock;
2
3 struct hash_table
4 {
5   long nbuckets;
6   struct node **buckets;
7 ;}
8
9 typedef struct node {
10   unsigned long key;
11   struct node *next;
12 } node_t;
13
14 int hash_search(struct hash_table *h, long key)
15 {
16   struct node *cur;
17   int retval;
18
19   read_lock(&hash_lock);
20   cur = h->buckets[key % h->nbuckets];
21   while (cur != NULL) {
22     if (cur->key >= key) {
23       spin_lock(&cur->node_lock);
24       spin_unlock(&hash_lock);
25       retval = (cur->key == key);
26       spin_unlock(&cur->node_lock);
27       return retval;
28     }
29     cur = cur->next;
30   }
31   spin_unlock(&hash_lock);
32   return 0;
33 }

Figure 5.24: Reader-Writer-Locking Hash Table Search

systems. Locking in general and reader-writer locking in particular is described extensively in Chapter 6.

5.4.2 Hierarchical Locking

The idea behind hierarchical locking is to have a coarse-grained lock that is held only long enough to work out which fine-grained lock to acquire. Figure 5.25 shows how our hash-table search might be adapted to do hierarchical locking, but also shows the great weakness of this approach: we have paid the overhead of acquiring a second lock, but we only hold it for a short time. In this case, the simpler data-locking approach would be simpler and likely perform better.

Quick Quiz 5.14: In what situation would hierarchical locking work well? ■

5.4.3 Resource Allocator Caches

This section presents a simplified schematic of a parallel fixed-block-size memory allocator. More detailed descriptions may be found in the literature [MG92, MS93, BA01, MSK01] or in the Linux kernel [Tor03c].
5.4.3.1 Parallel Resource Allocation Problem

The basic problem facing a parallel memory allocator is the tension between the need to provide extremely fast memory allocation and freeing in the common case and the need to efficiently distribute memory in face of unfavorable allocation and freeing patterns.

To see this tension, consider a straightforward application of data ownership to this problem — simply carve up memory so that each CPU owns its share. For example, suppose that a system with two CPUs has two gigabytes of memory (such as the one that I am typing on right now). We could simply assign each CPU one gigabyte of memory, and allow each CPU to access its own private chunk of memory, without the need for locking and its complexities and overheads. Unfortunately, this simple scheme breaks down if an algorithm happens to have CPU 0 allocate all of the memory and CPU 1 the free it, as would happen in a simple producer-consumer workload.

The other extreme, code locking, suffers from excessive lock contention and overhead [MS93].

5.4.3.2 Parallel Fastpath for Resource Allocation

The commonly used solution uses parallel fastpath with each CPU owning a modest cache of blocks, and with a large code-locked shared pool for additional blocks. To prevent any given CPU from monopolizing the memory blocks, we place a limit on the number of blocks that can be in each CPU’s cache. In a two-CPU system, the flow of memory blocks will be as shown in Figure 5.26: when a given CPU is trying to free a block when its pool is full, it sends blocks to the global pool, and, similarly, when that CPU is trying to allocate a block when its pool is empty, it retrieves blocks from the global pool.

5.4.3.3 Data Structures

The actual data structures for a “toy” implementation of allocator caches are shown in Figure 5.27. The “Global Pool” of Figure 5.26 is implemented by globalmem of type struct globalmempool, and the two CPU pools by the per-CPU variable percpumem of type percpumempool. Both of these data structures have arrays of pointers to blocks in their pool fields, which are filled from index zero upwards. Thus, if globalmem.pool[3] is NULL, then the remainder of the array from index 4 up must also be NULL. The cur fields contain the index of the highest-numbered full element of the pool array, or -1 if all elements are empty.

All elements from globalmem.pool[0] through globalmem.pool[globalmem.cur] must be full, and all the rest must be empty.9

```c
#define TARGET_POOL_SIZE 3
#define GLOBAL_POOL_SIZE 40

struct globalmempool {
    spinlock_t mutex;
    int cur;
    struct memblock *pool[GLOBAL_POOL_SIZE];
} globalmem;

struct percpumempool {
    int cur;
    struct memblock *pool[2 * TARGET_POOL_SIZE];
} percpumem;

DEFINE_PER_THREAD(struct percpumempool, percpumem);
```

Figure 5.26: Allocator Cache Schematic

Figure 5.27: Allocator-Cache Data Structures

The operation of the pool data structures is illustrated by Figure 5.28, with the six boxes representing the array of pointers making up the pool field, and the number preceding them representing the cur field. The shaded boxes represent non-NULL pointers, while the empty boxes represent NULL pointers. An important, though potentially confusing, invariant of this data structure is that the cur field is always one smaller than the number of non-NULL

---

9 Both pool sizes (TARGET_POOL_SIZE and GLOBAL_POOL_SIZE) are unrealistically small, but this small size makes it easier to single-step the program in order to get a feel for its operation.
5.4. PARALLEL FASTPATH

5.4.3.4 Allocation Function

The allocation function `memblock_alloc()` may be seen in Figure 5.29. Line 7 picks up the current thread’s per-thread pool, and line 8 check to see if it is empty.

If so, lines 9-16 attempt to refill it from the global pool under the spinlock acquired on line 9 and released on line 16. Lines 10-14 move blocks from the global to the per-thread pool until either the local pool reaches its target size (half full) or the global pool is exhausted, and line 15 sets the per-thread pool’s count to the proper value.

In either case, line 18 checks for the per-thread pool still being empty, and if not, lines 19-21 remove a block and return it. Otherwise, line 23 tells the sad tale of memory exhaustion.

5.4.3.5 Free Function

Figure 5.30 shows the memory-block free function. Line 6 gets a pointer to this thread’s pool, and line 7 checks to see if this per-thread pool is full.

If so, lines 8-15 empty half of the per-thread pool into the global pool, with lines 8 and 14 acquiring and releasing the spinlock. Lines 9-12 implement the loop moving blocks from the local to the global pool, and line 13 sets the per-thread pool’s count to the proper value.

In either case, line 16 then places the newly freed block into the per-thread pool.
5.4.3.6 Performance

Rough performance results\(^{10}\) are shown in Figure 5.31, running on a dual-core Intel x86 running at 1GHz (4300 bogomips per CPU) with at most six blocks allowed in each CPU’s cache. In this micro-benchmark, each thread repeatedly allocates a group of blocks and then frees it, with the size of the group being the “allocation run length” displayed on the x-axis. The y-axis shows the number of successful allocation/free pairs per microsecond — failed allocations are not counted. The “X”s are from a two-thread run, while the “+”s are from a single-threaded run.

![Figure 5.31: Allocator Cache Performance](image)

Note that run lengths up to six scale linearly and give excellent performance, while run lengths greater than six show poor performance and almost always also show negative scaling. It is therefore quite important to size `TARGET_POOL_SIZE` sufficiently large, which fortunately is usually quite easy to do in actual practice [MSK01], especially given today’s large memories. For example, in most systems, it is quite reasonable to set `TARGET_POOL_SIZE` to 100, in which case allocations and frees are guaranteed to be confined to per-thread pools at least 99% of the time.

Quick Quiz 5.15: In Figure 5.31, there is a pattern of performance rising with increasing run length in groups of three samples, for example, for run lengths 10, 11, and 12. Why?

Quick Quiz 5.16: Allocation failures were observed in the two-thread tests at run lengths of 19 and greater. Given the global-pool size of 40 and the per-CPU target pool size of three, what is the smallest allocation run length at which failures can occur?

5.4.3.7 Real-World Design

The toy parallel resource allocator was quite simple, but real-world designs expand on this approach in a number of ways.

First, real-world allocators are required to handle a wide range of allocation sizes, as opposed to the single size shown in this toy example. One popular way to do this is to offer a fixed set of sizes, spaced so as to balance external and internal fragmentation, such as in the late-1980s BSD memory allocator [MK88]. Doing this would mean that the “globalmem” variable would need to be replicated on a per-size basis, and that the associated lock would similarly be replicated, resulting in data locking rather than the toy program’s code locking.

Second, production-quality systems must be able to repurpose memory, meaning that they must be able to coalesce blocks into larger structures, such as pages [MS93]. This coalescing will also need to be protected by a lock, which again could be replicated on a per-size basis.

Third, coalesced memory must be returned to the underlying memory system, and pages of memory must also be allocated from the underlying memory system. The locking required at this level will depend on that of the underlying memory system, but could well be code locking. Code locking can often be tolerated at this level, because this level is so infrequently reached in well-designed systems [MSK01].

Despite this real-world design’s greater complexity, the underlying idea is the same — repeated application of parallel fastpath, as shown in Table 5.1.

---

\(^{10}\) This data was not collected in a statistically meaningful way, and therefore should be viewed with great skepticism and suspicion. Good data-collection and -reduction practice is discussed in Chapter 333. That said, repeated runs gave similar results, and these results match more careful evaluations of similar algorithms.
5.5. PERFORMANCE SUMMARY

<table>
<thead>
<tr>
<th>Level</th>
<th>Locking</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Per-thread pool</td>
<td>Data ownership</td>
<td>High-speed allocation</td>
</tr>
<tr>
<td>Global block pool</td>
<td>Data locking</td>
<td>Distributing blocks among threads</td>
</tr>
<tr>
<td>Coalescing</td>
<td>Data locking</td>
<td>Combining blocks into pages</td>
</tr>
<tr>
<td>System memory</td>
<td>Code locking</td>
<td>Memory from/to system</td>
</tr>
</tbody>
</table>

Table 5.1: Schematic of Real-World Parallel Allocator

5.5 Performance Summary

@@@ summarize performance of the various options.
Forward-reference to the RCU/NBS section.
Chapter 6

Locking

The role of villain in much of the past few decades’ concurrency research literature is played by locking, which stands accused of promoting deadlocks, convoying, starvation, unfairness, data races, and all manner of other concurrency sins. Interestingly enough, the role of workhorse in shared-memory parallel software is played by, you guessed it, locking.

There are a number of reasons behind this dichotomy:

1. Many of locking’s sins have pragmatic design solutions that work well in most cases, for example:
   (a) Lock hierarchies to avoid deadlock.
   (b) Deadlock-detection tools, for example, the Linux kernel’s lockdep facility [Cor06a].
   (c) Locking-friendly data structures, such as arrays, hash tables, and radix trees, which will be covered in Chapter 11.

2. Some of locking’s sins are problems only at high levels of contention, levels reached only by poorly designed programs.

3. Some of locking’s sins are avoided by using other synchronization mechanisms in concert with locking. These other mechanisms include reference counters, statistical counters, simple non-blocking data structures, and RCU.

4. Until quite recently, almost all large shared-memory parallel programs were developed in secret, so that it was difficult for most researchers to learn of these pragmatic solutions.

5. All good stories need a villain, and locking has a long and honorable history serving as a research-paper whipping boy.

This chapter will give an overview of a number of ways to avoid locking’s more serious sins.

6.1 Staying Alive

Given that locking stands accused of deadlock and starvation, one important concern for shared-memory parallel developers is simply staying alive. The following sections therefore cover deadlock, livelock, starvation, unfairness, and inefficiency.

6.1.1 Deadlock

Deadlock occurs when each of a group of threads is holding at least one lock while at the same time waiting on a
Without some sort of external intervention, deadlock is forever. No thread can acquire the lock it is waiting on until that lock is released by the thread holding it, but the thread holding it cannot release it until the holding thread acquires the lock that it is waiting on.

We can create a directed-graph representation of a deadlock scenario with nodes for threads and locks, as shown in Figure 6.3. An arrow from a lock to a thread indicates that the thread holds the lock, for example, Thread B holds Locks 2 and 4. An arrow from a thread to a lock indicates that the thread is waiting on the lock, for example, Thread B is waiting on Lock 3.

A deadlock scenario will always contain at least one deadlock cycle. In Figure 6.3, this cycle is Thread B, Lock 3, Thread C, Lock 4, and back to Thread B.

Quick Quiz 6.1: But the definition of deadlock only said that each thread was holding at least one lock and waiting on another lock that was held by some thread. How do you know that there is a cycle?

Although there are some software environments such as database systems that can repair an existing deadlock, this approach requires either that one of the threads be killed or that a lock be forcibly stolen from one of the threads. This killing and forcible stealing can be inappropriate for transactions, but is often problematic for kernel and application-level use of locking: dealing with the resulting partially updated structures can be extremely complex, hazardous, and error-prone.

Kernels and applications therefore work to avoid deadlocks rather than to recover from them. There are a number of deadlock-avoidance strategies, including locking hierarchies (Section 6.1.1.1), local locking hierarchies (Section 6.1.1.2), layered locking hierarchies (Section 6.1.1.3), strategies for dealing with APIs containing pointers to locks (Section 6.1.1.4), conditional locking (Section 6.1.1.5), acquiring all needed locks first (Section 6.1.1.6), single-lock-at-a-time designs (Section 6.1.1.7), and strategies for signal/interrupt handlers (Section 6.1.1.8). Although there is no deadlock-avoidance strategy that works perfectly for all situations, there is a good selection of deadlock-avoidance tools to choose from.

6.1.1.1 Locking Hierarchies

Locking hierarchies order the locks and prohibit acquiring locks out of order. In Figure 6.3, we might order the locks numerically, so that a thread was forbidden from

![Figure 6.3: Deadlock Cycle](image)
acquiring a given lock if it already held a lock with the same or a higher number. Thread B has violated this hierarchy because it is attempting to acquire Lock 3 while holding Lock 4, which permitted the deadlock to occur.

Again, to apply a locking hierarchy, order the locks and prohibit out-of-order lock acquisition. In large programs, it is wise to use tools to enforce your locking hierarchy [Cor06a].

6.1.1.2 Local Locking Hierarchies

However, the global nature of locking hierarchies makes them difficult to apply to library functions. After all, the program using a given library function has not even been written yet, so how can the poor library-function implementor possibly hope to adhere to the yet-to-be-written program’s locking hierarchy?

One special case that is fortunately the common case is when the library function does not invoke any of the caller’s code. In this case, the caller’s locks will never be acquired while holding any of the library’s locks, so that there cannot be a deadlock cycle containing locks from both the library and the caller.

Quick Quiz 6.2: Are there any exceptions to this rule, so that there really could be a deadlock cycle containing locks from both the library and the caller, even given that the library code never invokes any of the caller’s functions?

But suppose that a library function does invoke the caller’s code. For example, the qsort() function invokes a caller-provided comparison function. A concurrent implementation of qsort() likely uses locking, which might result in deadlock in the perhaps-unlikely case where the comparison function is a complicated function involving locking. How can the library function avoid deadlock?

The golden rule in this case is “release all locks before invoking unknown code.” To follow this rule, the qsort() function must release all locks before invoking the comparison function.

Quick Quiz 6.3: But if qsort() releases all its locks before invoking the comparison function, how can it protect against races with other qsort() threads?

To see the benefits of local locking hierarchies, compare Figures 6.5 and 6.6. In both figures, application functions foo() and bar() invoke qsort() while holding locks A and B, respectively. Because this is a parallel implementation of qsort(), it acquires lock C. Function foo() passes function cmp() to qsort(), and cmp() acquires lock B. Function bar() passes a simple integer-comparison function (not shown) to qsort(), and this simple function does not acquire any locks.

Now, if qsort() holds Lock C while calling cmp()
in violation of the golden release-all-locks rule above, as shown in Figure 6.5, deadlock can occur. To see this, suppose that one thread invokes `foo()` while a second thread concurrently invokes `bar()`. The first thread will acquire lock A and the second thread will acquire lock B. If the first thread’s call to `qsort()` acquires lock C, then it will be unable to acquire lock B when it calls `cmp()`. But the first thread holds lock C, so the second thread’s call to `qsort()` will be unable to acquire it, and thus unable to release lock B, resulting in deadlock.

In contrast, if `qsort()` releases lock C before invoking the comparison function (which is unknown code from `qsort()`’s perspective), then deadlock is avoided as shown in Figure 6.6.

If each module releases all locks before invoking unknown code, then deadlock is avoided if each module separately avoids deadlock. This rule therefore greatly simplifies deadlock analysis and greatly improves modularity.

### 6.1.1.3 Layered Locking Hierarchies

Unfortunately, it might not be possible for `qsort()` to release all of its locks before invoking the comparison function. In this case, we cannot construct a local locking hierarchy by releasing all locks before invoking unknown code. However, we can instead construct a layered locking hierarchy, as shown in Figure 6.7. Here, the `cmp()` function uses a new lock D that is acquired after all of locks A, B, and C, avoiding deadlock. We therefore have three layers to the global deadlock hierarchy, the first containing locks A and B, the second containing lock C, and the third containing lock D.

For another example where releasing all locks before invoking unknown code is impractical, imagine an iterator over a linked list, as shown in Figure 6.8 (`locked_list.c`). The `list_start()` function acquires a lock on the list and returns the first element (if there is one), and `list_next()` either returns a pointer to the next element in the list or releases the lock and returns `NULL` if the end of the list has been reached.

Figure 6.9 shows how this list iterator may be used. Lines 1-4 define the `list ints` element containing a single integer, and lines 6-17 show how to iterate over the list. Line 11 locks the list and fetches a pointer to the first element, line 13 provides a pointer to our enclosing `list ints` structure, line 14 prints the corresponding integer, and line 15 moves to the next element. This is quite simple, and hides all of the locking.

That is, the locking remains hidden as long as the code processing each list element does not itself acquire a lock that is held across some other call to `list_start()` or `list_next()`, which results in deadlock. We can avoid the deadlock by layering the locking hierarchy to
6.1. STAYING ALIVE

75

```c
struct list_ints {
    struct list_head n;
    int a;
};

void list_print(struct locked_list *lp) {
    struct list_head *np;
    struct list_ints *ip;

    np = list_start(lp);
    while (np != NULL) {
        ip = list_entry(np, struct list_ints, n);
        printf("\t%d\n", ip->a);
        np = list_next(lp, np);
    }
}
```

Figure 6.9: Concurrent List Iterator Usage

take the list-iterator locking into account.

This layered approach can be extended to an arbitrarily large number of layers, but each added layer increases the complexity of the locking design. Such increases in complexity are particularly inconvenient for some types of object-oriented designs, in which control passes back and forth among a large group of objects in an undisciplined manner. This mismatch between the habits of object-oriented design and the need to avoid deadlock is an important reason why parallel programming is perceived by some to be so difficult.

Some alternatives to highly layered locking hierarchies are covered in Chapter 8.

6.1.1.4 Locking Hierarchies and Pointers to Locks

Although there are some exceptions, an external API containing a pointer to a lock is very often a misdesigned API. Handing an internal lock to some other software component is after all the antithesis of information hiding, which is in turn a key design principle.

Quick Quiz 6.4: Name one common exception where it is perfectly reasonable to pass a pointer to a lock into a function.

One exception is functions that hand off some entity, where the caller’s lock must be held until the handoff is complete, but where the lock must be released before the function returns. One example of such a function is the POSIX `pthread_cond_wait()` function, where passing an pointer to a `pthread_mutex_t` prevents hangs due to lost wakeups.

Quick Quiz 6.5: Doesn’t the fact that `pthread_cond_wait()` first releases the mutex and then re-acquires it eliminate the possibility of deadlock?

```c
retry:
    spin_lock(&lock2);
    layer_2_processing(pkt);
    nextlayer = layer_1(pkt);
    if (!spin_trylock(&nextlayer->lock1)) {
        spin_unlock(&lock2);
        spin_lock(&nextlayer->lock1);
        spin_lock(&lock2);
        if (layer_1(pkt) != nextlayer) {
            spin_unlock(&nextlayer->lock1);
            spin_unlock(&lock2);
            goto retry;
        }
    }
    layer_1_processing(pkt);
    spin_unlock(&lock2);
    spin_unlock(&nextlayer->lock1);
```

Figure 6.10: Avoiding Deadlock Via Conditional Locking

In short, if you find yourself exporting an API with a pointer to a lock as an argument or the return value, do yourself a favor and carefully reconsider your API design. It might well be the right thing to do, but experience indicates that this is unlikely.

6.1.1.5 Conditional Locking

But suppose that there is no reasonable locking hierarchy. This can happen in real life, for example, in layered network protocol stacks where packets flow in both directions. In the networking case, it might be necessary to hold the locks from both layers when passing a packet from one layer to another. Given that packets travel both up and down the protocol stack, this is an excellent recipe for deadlock, as illustrated in Figure 6.4. Here, a packet moving down the stack towards the wire must acquire the next layer’s lock out of order. Given that packets moving up the stack away from the wire are acquiring the locks in order, the lock acquisition in line 4 of the figure can result in deadlock.

One way to avoid deadlocks in this case is to impose a locking hierarchy, but when it is necessary to acquire a lock out of order, acquire it conditionally, as shown in Figure 6.10. Instead of unconditionally acquiring the layer-1 lock, line 5 conditionally acquires the lock using the `spin_trylock()` primitive. This primitive acquires the lock immediately if the lock is available (returning non-zero), and otherwise returns zero without acquiring the lock.

If `spin_trylock()` was successful, line 15 does the needed layer-1 processing. Otherwise, line 6 releases the lock, and lines 7 and 8 acquire them in the correct order. Unfortunately, there might be multiple networking
devices on the system (e.g., Ethernet and WiFi), so that the `layer_1()` function must make a routing decision. This decision might change at any time, especially if the system is mobile.\(^1\) Therefore, line 9 must recheck the decision, and if it has changed, must release the locks and start over.

**Quick Quiz 6.6:** Can the transformation from Figure 6.4 to Figure 6.10 be applied universally? ■

**Quick Quiz 6.7:** But the complexity in Figure 6.10 is well worthwhile given that it avoids deadlock, right? ■

### 6.1.1.6 Acquire Needed Locks First

In an important special case of conditional locking all needed locks are acquired before any processing is carried out. In this case, processing need not be idempotent: if it turns out to be impossible to acquire a given lock without first releasing one that was already acquired, just release all the locks and try again. Only once all needed locks are held will any processing be carried out.

However, this procedure can result in *livelock*, which will be discussed in Section 6.1.2.

### 6.1.1.7 Single-Lock-at-a-Time Designs

In some cases, it is possible to avoid nesting locks, thus avoiding deadlock. For example, if a problem is perfectly partitionable, a single lock may be assigned to each partition. Then a thread working on a given partition need only acquire the one corresponding lock. Because no thread ever holds more than one lock at a time, deadlock is impossible.

However, there must be some mechanism to ensure that the needed data structures remain in existence during the time that neither lock is held. One such mechanism is discussed in Section 6.4 and several others are presented in Chapter 8.

### 6.1.1.8 Signal/Interrupt Handlers

Deadlocks involving signal handlers are often quickly dismissed by noting that it is not legal to invoke `pthread_mutex_lock()` from within a signal handler [Ope97]. However, it is possible (though almost always unwise) to hand-craft locking primitives that can be invoked from signal handlers. Besides which, almost all operating-system kernels permit locks to be acquired from within interrupt handlers, which are the kernel analog to signal handlers.

The trick is to block signals (or disable interrupts, as the case may be) when acquiring any lock that might be acquired within an interrupt handler. Furthermore, if holding such a lock, it is illegal to attempt to acquire any lock that is ever acquired outside of a signal handler without blocking signals.

**Quick Quiz 6.8:** Why is it illegal to acquire a Lock A that is acquired outside of a signal handler without blocking signals while holding a Lock B that is acquired within a signal handler? ■

If a lock is acquired by the handlers for several signals, then each and every one of these signals must be blocked whenever that lock is acquired, even when that lock is acquired within a signal handler.

**Quick Quiz 6.9:** How can you legally block signals within a signal handler? ■

Unfortunately, blocking and unblocking signals can be expensive in some operating systems, notably including Linux, so performance concerns often mean that locks acquired in signal handlers are only acquired in signal handlers, and that lockless synchronization mechanisms are used to communicate between application code and signal handlers.

Or that signal handlers are avoided completely except for handling fatal errors.

### 6.1.1.9 Discussion

There are a large number of deadlock-avoidance strategies available to the shared-memory parallel programmer, but there are sequential programs for which none of them is a good fit. This is one of the reasons that expert programmers have more than one tool in their toolbox: locking is a powerful concurrency tool, but there are jobs better addressed with other tools.

**Quick Quiz 6.10:** Given an object-oriented application that passes control freely among a group of objects such that there is no reasonable locking hierarchy, layered or otherwise, how can this application be parallelized? ■

Nevertheless, the strategies described in this section have proven quite useful in many settings.

### 6.1.2 Livelock and Starvation

Although conditional locking can be an effective deadlock-avoidance mechanism, it can be abused. Consider for example the beautifully symmetric example shown in Figure 6.11. This example’s beauty hides an ugly livelock. To see this, consider the following sequence of events:

\(^1\) And, in contrast to the 1900s, mobility is the common case.
Figure 6.11: Abusing Conditional Locking

1. Thread 1 acquires lock1 on line 4, then invokes do_one_thing().
2. Thread 2 acquires lock2 on line 18, then invokes do_a_third_thing().
3. Thread 1 attempts to acquire lock2, but fails because Thread 2 holds it.
4. Thread 2 attempts to acquire lock1, but fails because Thread 1 holds it.
5. Thread 1 releases lock1, and jumps to retry.
6. Thread 2 releases lock2, and jumps to retry.
7. The livelock dance repeats from the beginning.

Quick Quiz 6.11: How can the livelock shown in Figure 6.11 be avoided? ■

Starvation is very similar to livelock. Put another way, livelock is an extreme form of starvation where a group of threads starve, rather than just one of them.2

Livelock and starvation are serious issues in software transactional memory implementations, and so the concept of contention manager has been introduced to encapsulate these issues. In the case of locking, simple

---

2 Try not to get too hung up on the exact definitions of terms like livelock, starvation, and unfairness. Anything that causes a group of threads to fail to make good forward progress is a problem that needs to be fixed, regardless of what name you choose for it.
CHAPTER 6. LOCKING

6.1.3 Unfairness
Unfairness can be thought of as a less-severe form of starvation, where a subset of threads contending for a given lock are granted the lion’s share of the acquisitions. This can happen on machines with shared caches or NUMA characteristics, for example, as shown in Figure 6.13. If CPU 0 releases a lock that all the other CPUs are attempting to acquire, the interconnect shared between CPUs 0 and 1 means that CPU 1 will have an advantage over CPUs 2-7. Therefore CPU 1 will likely acquire the lock. If CPU 1 hold the lock long enough for CPU 0 to be requesting the lock by the time CPU 1 releases it and vice versa, the lock can shuttle between CPUs 1 and 2, bypassing CPUs 2-7.

Quick Quiz 6.13: Wouldn’t it be better just to use a good parallel design so that lock contention was low enough to avoid unfairness?

6.1.4 Inefficiency
Locks are implemented using atomic instructions and memory barriers, and often involve cache misses. As we saw in Chapter 2, these instructions are quite expensive, roughly two orders of magnitude greater overhead than simple instructions. This can be a serious problem for locking: If you protect a single instruction with a lock, you will increase the overhead by a factor of one hundred. Even assuming perfect scalability, one hundred CPUs would be required to keep up with a single CPU executing the same code without locking.

This situation underscores the synchronization-granularity tradeoff discussed in Section 5.3, especially Figure 5.21: Too coarse a granularity will limit scalability, while too fine a granularity will result in excessive synchronization overhead.

That said, once a lock is held, the data protected by that lock can be accessed by the lock holder without interference. Acquiring a lock might be expensive, but once held, the CPU’s caches are an effective performance booster, at least for large critical sections.

Quick Quiz 6.14: How might the lock holder be interfered with?

6.2 Types of Locks
There are a surprising number of types of locks, more than this short chapter can possibly do justice to. The following sections discuss exclusive locks (Section 6.2.1), reader-writer locks (Section 6.2.2), and multi-role locks (Section 6.2.3).

6.2.1 Exclusive Locks
Exclusive locks are what they say they are: only one thread may hold the lock at a time. The holder of such a lock thus has exclusive access to all data protected by that lock, hence the name.

Of course, this all assumes that this lock is held across all accesses to data purportedly protected by the lock. Although there are some tools that can help, the ultimate responsibility for ensuring that the lock is acquired in all necessary code paths rests with the developer.

6.2.2 Reader-Writer Locks
Reader-writer locks [CHP71] permit any number of readers to hold the lock concurrently on the one hand or a single writer to hold the lock on the other. In theory, then, reader-writer locks should allow excellent scalability for data that is read often and written rarely. In practice, the scalability will depend on the reader-writer lock implementation.
6.2. TYPES OF LOCKS

The classic reader-writer lock implementation involves a set of counters and flags that are manipulated atomically. This type of implementation suffers from the same problem as does exclusive locking for short critical sections: The overhead of acquiring and releasing the lock is about two orders of magnitude greater than the overhead of a simple instruction. Of course, if the critical section is long enough, the overhead of acquiring and releasing the lock becomes negligible. However, because only one thread at a time can be manipulating the lock, the required critical-section size increases with the number of CPUs.

It is possible to design a reader-writer lock that is much more favorable to readers through use of per-thread exclusive locks [HW92]. To read, a thread acquires only its own lock. To write, a thread acquires all locks. In the absence of writers, each reader incurs only atomic-instruction and memory-barrier overhead, with no cache misses, which is quite good for a locking primitive. Unfortunately, writers must incur cache misses as well as atomic-instruction and memory-barrier overhead—multiplied by the number of threads.

In short, reader-writer locks can be quite useful in a number of situations, but each type of implementation does have its drawbacks.

6.2.3 Beyond Reader-Writer Locks

Reader-writer locks and exclusive locks differ in their admission policy: exclusive locks allow at most one holder, while reader-writer locks permit an arbitrary number of read-holders (but only one write-holder). There is a very large number of possible admission policies, one of the more elaborate being that of the VAX/VMS distributed lock manager (DLM) [ST87], which is shown in Table 6.1. Blank cells indicate compatible modes, while cells containing “X” indicate incompatible modes.

The VAX/VMS DLM uses six modes. For purposes of comparison, exclusive locks use two modes (not held and held), while reader-writer locks use three modes (not held, read held, and write held).

The first mode is null, or not held. This mode is compatible with all other modes, which is to be expected: If a thread is not holding a lock, it should not prevent any other thread from acquiring that lock.

The second mode is concurrent read, which is compatible with every other mode except for exclusive. The concurrent-read mode might be used to accumulate approximate statistics on a data structure, while permitting updates to proceed concurrently.

The third mode is concurrent write, which is compatible with null, concurrent read, and concurrent write. The concurrent-write mode might be used to update approximate statistics, while still permitting reads and concurrent updates to proceed concurrently.

The fourth mode is protected read, which is compatible with null, concurrent read, and protected read. The protected-read mode might be used to obtain a consistent snapshot of the data structure, while permitting reads but not updates to proceed concurrently.

The fifth mode is protected write, which is compatible with null and protected read. The protected-write mode might be used to carry out updates to a data structure that could interfere with protected readers but which could be tolerated by concurrent readers.

The sixth and final mode is exclusive, which is compatible only with null. The exclusive mode is used when it is necessary to exclude all other accesses.

It is interesting to note that exclusive locks and reader-writer locks can be emulated by the VAX/VMS DLM. Exclusive locks would use only the null and exclusive modes, while reader-writer locks might use the null, protected-read, and protected-write modes.

Quick Quiz 6.15: Is there any other way for the VAX/VMS DLM to emulate a reader-writer lock?

Although the VAX/VMS DLM policy has seen widespread production use for distributed databases, it does not appear to be used much in shared-memory applications. One possible reason for this is that the greater communication overheads of distributed databases can hide the greater overhead of the VAX/VMS DLM’s more-complex admission policy.

Nevertheless, the VAX/VMS DLM is an interesting il-

Table 6.1: VAX/VMS Distributed Lock Manager Policy

<table>
<thead>
<tr>
<th></th>
<th>Null (Not Held)</th>
<th>Concurrent Read</th>
<th>Concurrent Write</th>
<th>Protected Read</th>
<th>Protected Write</th>
<th>Exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null (Not Held)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Concurrent Read</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Concurrent Write</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Protected Read</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Protected Write</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Exclusive</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
typedef int xchglock_t;
#define DEFINE_XCHG_LOCK(n) xchglock_t n = 0

void xchg_lock(xchglock_t *xp) {
    while (xchg(xp, 1) == 1) {
        while (*xp == 1) continue;
    }
}

void xchg_unlock(xchglock_t *xp) {
    (void)xchg(xp, 0);
}

Figure 6.14: Sample Lock Based on Atomic Exchange

Illustration of just how flexible the concepts behind locking can be.

6.3 Locking Implementation Issues

Developers are almost always best-served by using whatever locking primitives are provided by the system, for example, the POSIX pthread mutex locks [Ope97, But97]. Nevertheless, studying sample implementations can be helpful, as can considering the challenges posed by extreme workloads and environments.

6.3.1 Sample Exclusive-Locking Implementation Based on Atomic Exchange

This section reviews the implementation shown in Figure 6.14. The data structure for this lock is just an int, as shown on line 1, but could be any integral type. The initial value of this lock is zero, meaning “unlocked”, as shown on line 2.

Lock acquisition is carried out by the xchg_lock() function shown on lines 4-9. This function uses a nested loop, with the outer loop repeatedly atomically exchanging the value of the lock with the value one (meaning “locked”). If the old value was already the value one (in other words, someone else already holds the lock), then the inner loop (lines 7-8) spins until the lock is available, at which point the outer loop makes another attempt to acquire the lock.

Quick Quiz 6.16: Why bother with the inner loop on lines 7-8 of Figure 6.14? Why not simply repeatedly do the atomic exchange operation on line 6? ■

Lock release is carried out by the xchg_unlock() function shown on lines 12-15. Line 14 atomically exchanges the value zero (“unlocked”) into the lock, thus marking it as having been released.

Quick Quiz 6.17: Why not simply store zero into the lock word on line 14 of Figure 6.14? ■

This lock is a simple example of a test-and-set lock [SR84], but very similar mechanisms have been used extensively as pure spinlocks in production.

6.3.2 Other Exclusive-Locking Implementations

There are a great many other possible implementations of locking based on atomic instructions, many of which are reviewed by Mellor-Crummey and Scott [MCS91]. These implementations represent different points in a multi-dimensional design tradeoff [McK96b]. For example, the atomic-exchange-based test-and-set lock presented in the previous section works well when contention is low and has the advantage of small memory footprint. It avoids giving the lock to threads that cannot use it, but as a result can suffer from unfairness or even starvation at high contention levels.

In contrast, ticket lock [MCS91], which is used in the Linux kernel, avoids unfairness at high contention levels, but as a consequence of its first-in-first-out discipline can grant the lock to a thread that is currently unable to use it, for example, due to being preempted, interrupted, or otherwise out of action.

All locking implementations where waiters spin on a single memory location, including both test-and-set locks and ticket locks, suffer from performance problems at high contention levels. The problem is that the thread releasing the lock must update the value of the corresponding memory location. At low contention, this is not a problem: The corresponding cache line is very likely still local to and writeable by the thread holding the lock. In contrast, at high levels of contention, each thread attempting to acquire the lock will have a read-only copy of the cache line, and the lock holder will need to invalidate all such copies before it can carry out the update that releases the lock. In general, the more CPUs and threads there are, the greater the overhead incurred when releasing the lock under conditions of high contention.

This negative scalability has motivated a number of different queued-lock implementations [And90, GT90, MCS91, WK94, Cra94, MLH94, TS93], which assign different queue elements to each of the threads attempting to acquire the lock, thus reducing the lock’s memory
contention.

More recent queued-lock implementations also take the system’s architecture into account, preferentially granting locks locally, while also taking steps to avoid starvation [SSVM02, RH03, RH02, JMRR02, MCM02]. Many of these can be thought of as analogous to the elevator algorithms traditionally used in scheduling disk I/O.

Unfortunately, the same scheduling logic that improves the efficiency of queued locks at high contention also increases their overhead at low contention. Beng-hong Lim and Anant Agarwal therefore combined a simple test-and-set lock with a queued lock, using the test-and-set lock at low levels of contention and switching to the queued lock at high levels of contention [LA94], thus getting low overhead at low levels of contention and getting fairness and high throughput at high levels of contention. Browning et al. took a similar approach, but avoided the use of a separate flag, so that the test-and-set fast path uses the same sequence of instructions that would be used in a simple test-and-set lock [BMMM05]. This approach has been used in production.

Another issue that arises at high levels of contention is when the lock holder is delayed, especially when the delay is due to preemption, which can result in priority inversion, where a low-priority thread holds a lock, but is preempted by a medium priority CPU-bound thread, which results in a high-priority process blocking while attempting to acquire the lock. The result is that the CPU-bound medium-priority process is preventing the high-priority process from running. One solution is priority inheritance [LR80], which has been widely used for real-time computing [SRL90, Cor06b], despite some lingering controversy over this practice [Yod04, Loc02].

Another way to avoid priority inversion is to prevent preemption while a lock is held. Because preventing preemption while locks are held also improves throughput, most proprietary UNIX kernels offer some form of scheduler-conscious synchronization mechanism [KWS97], largely due to the efforts of a large database vendor. These mechanisms usually take the form of a hint that preemption would be inappropriate. These hints frequently take the form of a bit set in a particular machine register, which enables extremely low per-lock-acquisition overhead for these mechanisms. In contrast, Linux avoids these hints, instead getting similar results from a mechanism called futexes [FRK02, Mol06, Ros06].

Interestingly enough, atomic instructions are not strictly needed to implement locks [Dij65, Lam74].

```c
1 int delete(int key)
2 {
3   int b;
4   struct element *p;
5   b = hashfunction(key);
6   p = hashtable[b];
7   if (p == NULL || p->key != key)
8      return 0;
9   spin_lock(&p->lock);
10  hashtable[b] = NULL;
11  spin_unlock(&p->lock);
12  kfree(p);
13  return 1;
14 }
```

Figure 6.15: Per-Element Locking Without Existence Guarantees

An excellent exposition of the issues surrounding locking implementations based on simple loads and stores may be found in Herlihy’s and Shavit’s textbook [HS08]. The main point echoed here is that such implementations currently have little practical application, although a careful study of them can be both entertaining and enlightening. Nevertheless, such study is left as an exercise for the reader.

6.4 Lock-Based Existence Guarantees

A key challenge in parallel programming is to provide existence guarantees [GKAS99], so that attempts to access a given object can rely on that object being in existence throughout a given access attempt. In some cases, existence guarantees are implicit:

1. Global variables and static local variables in the base module will exist as long as the application is running.
2. Global variables and static local variables in a loaded module will exist as long as that module remains loaded.
3. A module will remain loaded as long as at least one of its functions has an active instance.
4. A given function instance’s on-stack variables will exist until that instance returns.
5. If you are executing within a given function or have been called from that function, then the given function has an active instance.
CHAPTER 6. LOCKING

82

1 int delete(int key)
2 {
3     int b;
4     struct element *p;
5     spinlock_t *sp;
6     b = hashfunction(key);
7     sp = &locktable[b];
8     spin_lock(sp);
9     p = hashtable[b];
10    if (p == NULL || p->key != key) {
11        spin_unlock(sp);
12        return 0;
13    }
14    hashtable[b] = NULL;
15    spin_unlock(sp);
16    kfree(p);
17    return 1;
18 }

Figure 6.16: Per-Element Locking With Lock-Based Existence Guarantees

These implicit existence guarantees are straightforward, though bugs involving implicit existence guarantees really can happen.

Quick Quiz 6.18: How can relying on implicit existence guarantees result in a bug?

But the more interesting—and troublesome—guarantee involves heap memory: A dynamically allocated data structure will exist until it is freed. The problem to be solved is to synchronize the freeing of the structure with concurrent accesses to that same structure. One way to do this is with explicit guarantees, such as locking. If a given structure may only be freed while holding a given lock, then holding that lock guarantees that structure’s existence.

But this guarantee depends on the existence of the lock itself. One straightforward way to guarantee the lock’s existence is to place the lock in a global variable, but global locking has the disadvantage of limiting scalability. One way of providing scalability that improves as the size of the data structure increases is to place a lock in each element of the structure. Unfortunately, putting the lock that is to protect a data element in the data element itself is subject to subtle race conditions, as shown in Figure 6.15.

Quick Quiz 6.19: What if the element we need to delete is not the first element of the list on line 8 of Figure 6.15?

Quick Quiz 6.20: What race condition can occur in Figure 6.15?

One way to fix this example is to use a hashed set of global locks, so that each hash bucket has its own lock, as shown in Figure 6.16. This approach allows acquiring the proper lock (on line 9) before gaining a pointer to the data element (on line 10). Although this approach works quite well for elements contained in a single partitionable data structure such as the hash table shown in the figure, it can be problematic if a given data element can be a member of multiple hash tables or given more-complex data structures such as trees or graphs. These problems can be solved, in fact, such solutions form the basis of lock-based software transactional memory implementations [ST95, DSS06]. However, Chapter 8 describes simpler—and faster—ways of providing existence guarantees.

6.5 Locking: Hero or Villain?

As is often the case in real life, locking can be either hero or villain, depending on how it is used and on the problem at hand. Locking is perhaps the most widely used and most generally useful tool, but it should not be the only tool in your parallel-programming toolbox.

The next few chapters will discuss other tools, and how they can best be used in concert with locking and with each other.
Chapter 7

Data Ownership

One of the simplest ways to avoid the synchronization overhead that comes with locking is to parcel the data out among the threads (or, in the case of kernels, CPUs) so that a given piece of data is accessed and modified by only one of the threads. This approach is used extremely heavily, in fact, it is one usage pattern that even novices use almost instinctively. In fact, it is used so heavily that this chapter will not introduce any new examples, but will instead recycle examples from previous chapters.

Quick Quiz 7.1: What form of data ownership that is extremely difficult to avoid using when creating shared-memory parallel programs (for example, using pthreads) in C or C++?

There are a number of approaches to data ownership. Section 7.1 presents the logical extreme in data ownership, where each thread has its own private address space. Section 7.2 looks at the opposite extreme, where the data is shared, but different threads own different access rights to the data. Section 7.3 describes function shipping, which is a way of allowing other threads to have indirect access to data owned by a particular thread. Section 7.4 describes how designated threads can be assigned ownership of a specified function and the related data. Section 7.5 discusses improving performance by transforming algorithms with shared data to instead use data ownership. Finally, Section 7.6 lists a few software environments that feature data ownership as a first-class citizen.

7.1 Multiple Processes

Section 3.1 introduced the following example:

```
1 compute_it 1 > compute_it.1.out &
2 compute_it 2 > compute_it.2.out &
3 wait
4 cat compute_it.1.out
5 cat compute_it.2.out
```

This example runs two instances of the `compute_it` program in parallel, as separate processes that do not share memory. Therefore, all data in a given process is owned by that process, so that almost the entirety of data in the above example is owned. This approach almost entirely eliminates synchronization overhead. The resulting combination of extreme simplicity and optimal performance is obviously quite attractive.

Quick Quiz 7.2: What synchronization remains in the example shown in Section 7.1?

Quick Quiz 7.3: Is there any shared data in the example shown in Section 7.1?

This same pattern can be written in C as well as in sh, as illustrated by Figures 3.2 and 3.3.

The next section discusses use of data ownership in shared-memory parallel programs.

7.2 Partial Data Ownership and pthreads

Chapter 4 makes heavy use of data ownership, but adds a twist. Threads are not allowed to modify data owned by other threads, but they are permitted to read it. In short, the use of shared memory allows more nuanced notions of ownership and access rights.

For example, consider the per-thread statistical counter implementation shown in Figure 4.8 on page 33. Here, `inc_count()` updates only the corresponding thread’s instance of `counter`, while `read_count()` accesses, but does not modify, all threads’ instances of `counter`.

Quick Quiz 7.4: Does it ever make sense to have partial data ownership where each thread reads only its own instance of a per-thread variable, but writes to other threads’ instances?
Pure data ownership is also both common and useful, for example, the per-thread memory-allocator caches discussed in Section 5.4.3 starting on page 65. In this algorithm, each thread’s cache is completely private to that thread.

**CHAPTER 7. DATA OWNERSHIP**

**7.3 Function Shipping**

The previous section described a weak form of data ownership where threads reached out to other threads’ data. This can be thought of as bringing the data to the functions that need it. An alternative approach is to send the functions to the data.

Such an approach is illustrated in Section 4.4.3 beginning on page 42, in particular the `flush_local_count_sig()` and `flush_local_count()` functions in Figure 4.21 on page 43.

The `flush_local_count_sig()` function is a signal handler that acts as the shipped function. The `pthread_kill()` function in `flush_local_count()` sends the signal—shipping the function—and then waits until the shipped function executes. This shipped function has the not-unusual added complication of needing to interact with any concurrently executing `add_count()` or `sub_count()` functions (see Figure 4.22 on page 44).

Quick Quiz 7.5: What mechanisms other than POSIX signals may be used to ship functions?

**7.4 Designated Thread**

The earlier sections describe ways of allowing each thread to keep its own copy or its own portion of the data. In contrast, this section describes a functional-decomposition approach, where a special designated thread that owns the rights to the data that is required to do its job. The eventually consistent counter implementation described in Section 4.2.3. This implementation has a designated thread that runs the `eventual()` function shown on lines 15-32 of Figure 4.7. This `eventual()` thread periodically pulls the per-thread counts into the global counter, so that accesses to the global counter will, as the name says, eventually converge on the actual value.

Quick Quiz 7.6: But none of the data in the `eventual()` function shown on lines 15-32 of Figure 4.7 is actually owned by the `eventual()` thread! In just what way is this data ownership???

**7.5 Privatization**

One way of improving the performance and scalability of a shared-memory parallel program is to transform it so as to convert shared data to private data that is owned by a particular thread.

An excellent example of this is shown in the answer to one of the Quick Quizzes in Section 5.1.1, which uses privatization to produce a solution to the Dining Philosophers problem with much better performance and scalability than that of the standard textbook solution. The original problem has five philosophers sitting around the table with one fork between each adjacent pair of philosophers, which permits at most two philosophers to eat concurrently.

We can trivially privatize this problem by providing an additional five forks, so that each philosopher has his or her own private pair of forks. This allows all five philosophers to eat concurrently, and also offers a considerable reduction in the spread of certain types of disease.

In other cases, privatization imposes costs. For example, consider the simple limit counter shown in Figure 4.11 on page 41. This is an example of an algorithm were threads can read each others’ data, but are only permitted to update their own data. A quick review of the algorithm shows that the only cross-thread accesses are in the summation loop in `read_count()`. If this loop is eliminated, we move to the more-efficient pure data ownership, but at the cost of a less-accurate result from `read_count()`.

Quick Quiz 7.7: Is it possible to obtain greater accuracy while still maintaining full privacy of the per-thread data?

In short, privatization is a powerful tool in the parallel programmer’s toolbox, but it must nevertheless be used with care. Just like every other synchronization primitive, it has the potential to increase complexity while decreasing performance and scalability.

**7.6 Other Uses of Data Ownership**

Data ownership works best when the data can be partitioned so that there is little or no need for cross thread access or update. Fortunately, this situation is reasonably common, and in a wide variety of parallel-programming environments.

Examples of data ownership include:

1. All message-passing environments, such as MPI,
PVM, and BOINC.

2. Client-server systems, including RPC, web services, and pretty much any system with a back-end database server.

3. Shared-nothing database systems.

4. Fork-join systems with separate per-process address spaces.

5. Process-based parallelism, such as the Erlang language.

Data ownership is perhaps the most underappreciated synchronization mechanism in existence. When used properly, it delivers unrivaled simplicity, performance, and scalability. Perhaps its simplicity costs it the respect that it deserves. Hopefully a greater appreciation for the subtlety and power of data ownership will lead to greater level of respect.
Chapter 8
Deferred Processing

The strategy of deferring work probably predates mankind, but only in the last few decades have workers recognized this strategy’s value in simplifying parallel algorithms [KL80, Mas92]. General approaches to work deferral in parallel programming include queuing, reference counting, and RCU.

8.1 Reference Counting

Reference counting tracks the number of references to a given object in order to prevent that object from being prematurely freed. Although this is a conceptually simple technique, many devils hide in the details. After all, if the object was not subject to being prematurely freed, there would be no need for the reference counter. But if the object is subject to being prematurely freed, what prevents that object from being freed during the reference-acquisition process itself?

There are a number of possible answers to this question, including:

1. A lock residing outside of the object must be held while manipulating the reference count. Note that there are a wide variety of types of locks, however, pretty much any type will suffice.

2. The object is created with a non-zero reference count, and new references may be acquired only when the current value of the reference counter is non-zero. Once acquired, a reference may be handed off to some other entity.

3. An existence guarantee is provided for the object, so that it cannot be freed during any time interval when some entity might be attempting to acquire a reference. Existence guarantees are often provided by automatic garbage collectors, and, as will be seen in Section 8.3, they can also be provided by RCU.

4. A type-safety guarantee is provided for the object, and there is in addition some identity check that can be performed once the reference is acquired. Typesafety guarantees can be provided by special-purpose memory allocators, and can also be provided by the SLAB_DESTROY_BY_RCU feature within the Linux kernel, again, as will be seen in Section 8.3.

Of course, any mechanism that provides existence guarantees by definition also provides type-safety guarantees. This section will therefore group the last two answers together under the rubric of RCU, leaving us with three general categories of reference-acquisition protection, namely, locking, reference counting, and RCU.

Quick Quiz 8.1: Why not implement reference-acquisition using a simple compare-and-swap operation that only acquires a reference if the reference counter is non-zero?

Given that the key reference-counting issue is synchronization between acquisition of a reference and freeing of the object, we have nine possible combinations of mechanisms, as shown in Table 8.1. This table divides
reference-counting mechanisms into the following broad categories:

1. Simple counting with neither atomic operations, memory barriers, nor alignment constraints ("-").
2. Atomic counting without memory barriers ("A").
3. Atomic counting, with memory barriers required only on release ("AM").
4. Atomic counting with a check combined with the atomic acquisition operation, and with memory barriers required only on release ("CAM").
5. Atomic counting with a check combined with the atomic acquisition operation ("CA").
6. Atomic counting with a check combined with the atomic acquisition operation, and with memory barriers also required on acquisition ("MCA").

However, because all Linux-kernel atomic operations that return a value are defined to contain memory barriers, all release operations contain memory barriers, and all checked acquisition operations also contain memory barriers. Therefore, cases “CA” and “MCA” are equivalent to “CAM”, so that there are sections below for only the first four cases: “-”, “A”, “AM”, and “CAM”. The Linux primitives that support reference counting are presented in Section 8.1.2. Later sections cite optimizations that can improve performance if reference acquisition and release is very frequent, and the reference count need be checked for zero only very rarely.

### 8.1.1 Implementation of Reference-Counting Categories

Simple counting protected by locking (“-”) is described in Section 8.1.1.1, atomic counting with no memory barriers (“A”) is described in Section 8.1.1.2 atomic counting with acquisition memory barrier (“AM”) is described in Section 8.1.1.3, and atomic counting with check and release memory barrier (“CAM”) is described in Section 8.1.1.4.

#### 8.1.1.1 Simple Counting

Simple counting, with neither atomic operations nor memory barriers, can be used when the reference-counter acquisition and release are both protected by the same lock. In this case, it should be clear that the reference count itself may be manipulated non-atomically, because the lock provides any necessary exclusion, memory barriers, atomic instructions, and disabling of compiler optimizations. This is the method of choice when the lock is required to protect other operations in addition to the reference count, but where a reference to the object must be held after the lock is released. Figure 8.1 shows a simple API that might be used to implement simple non-atomic reference counting – although simple reference counting is almost always open-coded instead.

```c
1 struct sref {
2    int refcount;
3 };
4
5 void sref_init(struct sref *sref)
6 {
7    sref->refcount = 1;
8 }
9
10 void sref_get(struct sref *sref)
11 {
12    sref->refcount++;
13 }
14
15 int sref_put(struct sref *sref,
16              void (*release)(struct sref *))
17 {
18    WARN_ON(release == NULL);
19    WARN_ON(release == (void (*)(struct sref *))kfree);
20
21    if (--sref->refcount == 0) {
22        release(sref);
23        return 1;
24    }
25    return 0;
26 }
```

Figure 8.1: Simple Reference-Count API

#### 8.1.1.2 Atomic Counting

Simple atomic counting may be used in cases where any CPU acquiring a reference must already hold a reference. This style is used when a single CPU creates an object for its own private use, but must allow other CPU, tasks, timer handlers, or I/O completion handlers that it later spawns to also access this object. Any CPU that hands the object off must first acquire a new reference on behalf of the recipient object. In the Linux kernel, the kref primitives are used to implement this style of reference counting, as shown in Figure 8.2.

Atomic counting is required because locking is not used to protect all reference-count operations, which means that it is possible for two different CPUs to concurrently manipulate the reference count. If normal increment and decrement were used, a pair of CPUs might both fetch the reference count concurrently, perhaps both obtaining
the value “3”. If both of them increment their value, they will both obtain “4”, and both will store this value back into the counter. Since the new value of the counter should instead be “5”, one of the two increments has been lost. Therefore, atomic operations must be used both for counter increments and for counter decrements.

If releases are guarded by locking or RCU, memory barriers are not required, but for different reasons. In the case of locking, the locks provide any needed memory barriers (and disabling of compiler optimizations), and the locks also prevent a pair of releases from running concurrently. In the case of RCU, cleanup must be deferred until all currently executing RCU read-side critical sections have completed, and any needed memory barriers or disabling of compiler optimizations will be provided by the RCU infrastructure. Therefore, if two CPUs release the final two references concurrently, the actual cleanup will be deferred until both CPUs exit their RCU read-side critical sections.

Quick Quiz 8.2: Why isn’t it necessary to guard against cases where one CPU acquires a reference just after another CPU releases the last reference?

```c
1 struct kref {
2 atomic_t refcount;
3 }; 4
5 void kref_init(struct kref *kref) {
6 atomic_set(&kref->refcount, 1);
7 }
8
9 void kref_get(struct kref *kref) {
10 atomic_inc(&kref->refcount);
11 }
12
13 int kref_put(struct kref *kref, void (*release)(struct kref *)) {
14 if ((atomic_read(&kref->refcount) == 1) ||
15 atomic_dec_and_test(&kref->refcount)) {
16 release(kref);
17 return 1;
18 }
19 WARN_ON(release == NULL);
20 WARN_ON(release == (void (*)(struct kref *))kfree);}
21
```

Figure 8.2: Linux Kernel kref API

The `kref` structure itself, consisting of a single atomic data item, is shown in lines 1-3 of Figure 8.2. The `kref_init()` function on lines 5-8 initializes the counter to the value “1”. Note that the `atomic_set()` primitive is a simple assignment, the name stems from the data type of `atomic_t` rather than from the operation. The `kref_init()` function must be invoked during object creation, before the object has been made available to any other CPU.

The `kref_get()` function on lines 10-14 unconditionally atomically increments the counter. The `atomic_inc()` primitive does not necessarily explicitly disable compiler optimizations on all platforms, but the fact that the `kref` primitives are in a separate module and that the Linux kernel build process does no cross-module optimizations has the same effect.

The `kref_put()` function on lines 16-28 checks for the counter having the value “1” on line 22 (in which case no concurrent `kref_get()` is permitted), or if atomically decrementing the counter results in zero on line 23. In either of these two cases, `kref_put()` invokes the specified `release` function and returns “1”, telling the caller that cleanup was performed. Otherwise, `kref_put()` returns “0”.

Quick Quiz 8.3: If the check on line 22 of Figure 8.2 fails, how could the check on line 23 possibly succeed?

Quick Quiz 8.4: How can it possibly be safe to non-atomically check for equality with “1” on line 22 of Figure 8.2?

8.1.3 Atomic Counting With Release Memory Barrier

This style of reference is used in the Linux kernel’s networking layer to track the destination caches that are used in packet routing. The actual implementation is quite a bit more involved; this section focuses on the aspects of struct dst_entry reference-count handling that matches this use case, shown in Figure 8.3.

```c
1 static inline
2 struct dst_entry * dst_clone(struct dst_entry * dst) {
3 if (dst) atomic_inc(&dst->__refcnt);
4 atomic_inc(&dst->__refcnt); 15
5 if ((atomic_read(&dst->__refcnt) == 1) ||
6 atomic_dec_and_test(&dst->__refcnt)) {
7 return 1;
8 }
9 return 0;
10 }
11 }
12 if (dst) {
13 WARN_ON(atomic_read(&dst->__refcnt) < 1);
14 smp_mb_before_atomic_dec();
15 atomic_dec(&dst->__refcnt); 17 }
```

Figure 8.3: Linux Kernel dst_clone API
The `dst_clone()` primitive may be used if the caller already has a reference to the specified `dst_entry`, in which case it obtains another reference that may be handed off to some other entity within the kernel. Because a reference is already held by the caller, `dst_clone()` need not execute any memory barriers. The act of handing the `dst_entry` to some other entity might or might not require a memory barrier, but if such a memory barrier is required, it will be embedded in the mechanism used to hand the `dst_entry` off.

The `dst_release()` primitive may be invoked from any environment, and the caller might well reference elements of the `dst_entry` structure immediately prior to the call to `dst_release()`. The `dst_release()` primitive therefore contains a memory barrier on line 14 preventing both the compiler and the CPU from misordering accesses.

Please note that the programmer making use of `dst_clone()` and `dst_release()` need not be aware of the memory barriers, only of the rules for using these two primitives.

### 8.1.1.4 Atomic Counting With Check and Release Memory Barrier

The fact that reference-count acquisition can run concurrently with reference-count release adds further complications. Suppose that a reference-count release finds that the new value of the reference count is zero, signalling that it is now safe to clean up the reference-counted object. We clearly cannot allow a reference-count acquisition to start after such clean-up has commenced, so the acquisition must include a check for a zero reference count. This check must be part of the atomic increment operation, as shown below.

**Quick Quiz 8.5:** Why can’t the check for a zero reference count be made in a simple “if” statement with an atomic increment in its “then” clause?

The Linux kernel’s `fget()` and `fput()` primitives use this style of reference counting. Simplified versions of these functions are shown in Figure 8.4.

Line 4 of `fget()` fetches the pointer to the current process’s file-descriptor table, which might well be shared with other processes. Line 6 invokes `rcu_read_lock()`, which enters an RCU read-side critical section. The callback function from any subsequent `call_rcu()` primitive will be deferred until a matching `rcu_read_unlock()` is reached (line 10 or 14 in this example). Line 7 looks up the file structure corresponding to the file descriptor specified by the `fd` argument, as will

```c
1 struct file *fget(unsigned int fd)
2 {
3     struct file *file;
4     struct files_struct *files = current->files;
5     rcu_read_lock();
6     file = fcheck_files(files, fd);
7     if (file) {
8         if (!atomic_inc_not_zero(&file->f_count)) {
9             rcu_read_unlock();
10             return NULL;
11         }
12     }
13     rcu_read_unlock();
14     return file;
15 }
16
17 struct file *
18 fcheck_files(struct files_struct *files, unsigned int fd)
19 {
20     struct file * file = NULL;
21     struct fdtable *fdt = rcu_dereference((files)->fdt);
22     if (fd < fdt->max_fds)
23         file = rcu_dereference(fdt->fd[fd]);
24     return file;
25 }
26
27 void fput(struct file *file)
28 {
29     if (atomic_dec_and_test(&file->f_count))
30         call_rcu(&file->f_u.fu_rcuhead, file_free_rcu);
31 }
32
33 static void file_free_rcu(struct rcu_head *head)
34 {
35     struct file *f;
36     f = container_of(head, struct file, f_u.fu_rcuhead);
37     kmem_cache_free(filp_cachep, f);
38 }
39
Figure 8.4: Linux Kernel fget/fput API
be described later. If there is an open file corresponding to the specified file descriptor, then line 9 attempts to atomically acquire a reference count. If it fails to do so, lines 10-11 exit the RCU read-side critical section and report failure. Otherwise, if the attempt is successful, lines 14-15 exit the read-side critical section and return a pointer to the file structure.

The fcheck_files() primitive is a helper function for fget(). It uses the rcu_dereference() primitive to safely fetch an RCU-protected pointer for later dereferencing (this emits a memory barrier on CPUs such as DEC Alpha in which data dependencies do not enforce memory ordering). Line 22 uses rcu_dereference() to fetch a pointer to this task’s current file-descriptor table, and line 24 checks to see if the specified file descriptor is in range. If so, line 25 fetches the pointer to the file structure, again using the rcu_dereference() primitive. Line 26 then returns a pointer to the file structure or NULL in case of failure.

The fput() primitive releases a reference to a file structure. Line 31 atomically decrements the reference count, and, if the result was zero, line 32 invokes the call_rcu() primitives in order to free up the file structure (via the file_free_rcu() function specified in call_rcu()’s second argument), but only after all currently-executing RCU read-side critical sections complete. The time period required for all currently-executing RCU read-side critical sections to complete is termed a “grace period”. Note that the atomic_dec_and_test() primitive contains a memory barrier. This memory barrier is not necessary in this example, since the structure cannot be destroyed until the RCU read-side critical section completes, but in Linux, all atomic operations that return a result must by definition contain memory barriers.

Once the grace period completes, the file_free_rcu() function obtains a pointer to the file structure on line 39, and frees it on line 40.

This approach is also used by Linux’s virtual-memory system, see get_page_unless_zero() and put_page_testzero() for page structures as well as try_to_unuse() and mmput() for memory-map structures.

8.1.2 Linux Primitives Supporting Reference Counting

The Linux-kernel primitives used in the above examples are summarized in the following list.

- **atomic_t** Type definition for 32-bit quantity to be manipulated atomically.
- **void atomic_dec(atomic_t *var);** Atomically decrements the referenced variable without necessarily issuing a memory barrier or disabling compiler optimizations.
- **int atomic_dec_and_test(atomic_t *var);** Atomically decrements the referenced variable, returning true if the result is zero. Issues a memory barrier and disables compiler optimizations that might otherwise move memory references across this primitive.
- **void atomic_inc(atomic_t *var);** Atomically increments the referenced variable without necessarily issuing a memory barrier or disabling compiler optimizations.
- **int atomic_inc_not_zero(atomic_t *var);** Atomically increments the referenced variable, but only if the value is non-zero, and returning true if the increment occurred. Issues a memory barrier and disables compiler optimizations that might otherwise move memory references across this primitive.
- **void atomic_read(atomic_t *var);** Returns the integer value of the referenced variable. This is not an atomic operation, and it neither issues memory barriers nor disables compiler optimizations.
- **int atomic_set(atomic_t *var, int val);** Sets the value of the referenced atomic variable to “val”. This is not an atomic operation, and it neither issues memory barriers nor disables compiler optimizations.
- **void call_rcu(struct rcu_head *head, void (*func)(struct rcu_head *head));** Invokes func(head) some time after all currently executing RCU read-side critical sections complete, however, the call_rcu() primitive returns immediately. Note that head is normally a field within an RCU-protected data structure, and that func is normally a function that frees up this data structure. The time interval between the invocation of call_rcu() and the invocation of func is termed a “grace period”. Any interval of time containing a grace period is itself a grace period.
• type *container_of(p, type, f);
  Given a pointer “p” to a field “f” within a structure
  of the specified type, return a pointer to the structure.

• void rcu_read_lock(void); Marks the be-
  beginning of an RCU read-side critical section.

• void rcu_read_unlock(void); Marks the
  end of an RCU read-side critical section. RCU read-
  side critical sections may be nested.

• void smp_mb__before_atomic__
# comment
  dec(void); Issues a memory barrier and
disables code-motion compiler optimizations only if
the platform’s atomic_dec() primitive does not
already do so.

• struct rcu_head A data structure used by the
  RCU infrastructure to track objects awaiting a grace
  period. This is normally included as a field within
  an RCU-protected data structure.

8.1.3 Counter Optimizations

In some cases where increments and decrements are com-
mon, but checks for zero are rare, it makes sense to main-
tain per-CPU or per-task counters, as was discussed in
Chapter 4. See Appendix D.1 for an example of this
technique applied to RCU. This approach eliminates the
need for atomic instructions or memory barriers on the
increment and decrement primitives, but still requires that
code-motion compiler optimizations be disabled. In ad-
dition, the primitives such as synchronizesrcu() that check for the aggregate reference count reaching zero
can be quite slow. This underscores the fact that these
techniques are designed for situations where the refer-
ences are frequently acquired and released, but where it
is rarely necessary to check for a zero reference count.

However, it is often the case that use of reference counts
requires writing (often atomically) to a data structure that
is otherwise read only. In this case, reference counts are
imposing expensive cache misses on readers. It is there-
fore worthwhile to look into synchronization mechanisms
that do not require readers to do writes. One such syn-
chronization mechanism, sequence locks, is covered in
the next section.

8.2 Sequence Locks

Sequence locks are used in the Linux kernel for read-
mostly data that must be seen in a consistent state by
readers. However, unlike reader-writer locking, readers
do not exclude writers. Instead, sequence-lock readers
retry an operation if they detect activity from a concurrent
writer.

Quick Quiz 8.6: Why isn’t this sequence-lock discus-
sion in Chapter 6, you know, the one on locking? ■

The key component of sequence locking is the sequence
number, which has an even value in the absence of writers
and an odd value if there is an update in progress. Readers
can then snapshot the value before and after each access.
If either snapshot has an odd value, or if the two snap-
shots differ, there has been a concurrent update, and the
reader must discard the results of the access and then retry
it. Readers use the read_seqbegin() and read_se-
qretry() functions, as shown in Figure 8.5, when
accessing data protected by a sequence lock. Writers must
increment the value before and after each update, and
only one writer is permitted at a given time. Writers use
the write_seqlock() and write_sequnlock() functions, as shown in Figure 8.6, when updating data
protected by a sequence lock.

Sequence-lock-protected data can have an arbitrarily
large number of concurrent readers, but only one writer
at a time. Sequence locking is used in the Linux kernel
to protect calibration quantities used for timekeeping. It
is also used in pathname traversal to detect concurrent
rename operations.

Quick Quiz 8.7: Can you use sequence locks as the
only synchronization mechanism protecting a linked list
supporting concurrent addition, deletion, and search? ■

A simple implementation of sequence locks is shown in
Figure 8.7 (seqlock.h). The seqlock_t data struc-
ture is shown on lines 1-4, and contains the sequence
number along with a lock to serialize writers. Lines 6-10
show seqlock_init(), which, as the name indicates,
initializes a seqlock_t.

Lines 12-22 show read_seqbegin(), which be-
1 do |
2  seq = read_seqbegin(&test_seqlock); |
3  /* read-side access. */ |
4  while (read_seqretry(&test_seqlock, seq)); |

Figure 8.5: Sequence-Locking Reader

1 write_seqlock(&test_seqlock); |
2  /* Update */ |
3  write_sequnlock(&test_seqlock); |

Figure 8.6: Sequence-Locking Writer
8.2. **SEQUENCE LOCKS**

typedef struct {
  unsigned long seq;
  spinlock_t lock;
} seqlock_t;

static void seqlock_init(seqlock_t *slp)
{
  slp->seq = 0;
  spin_lock_init(&slp->lock);
}

static unsigned long read_seqbegin(seqlock_t *slp)
{
  unsigned long s;

  repeat:
  s = ACCESS_ONCE(slp->seq);
  smp_mb();
  if (unlikely(s & 1))
    goto repeat;
  return s;
}

static int read_seqretry(seqlock_t *slp,
unsigned long oldseq)
{
  unsigned long s;

  smp_mb();
  s = ACCESS_ONCE(slp->seq);
  return s != oldseq;
}

static void write_seqlock(seqlock_t *slp)
{
  spin_lock(&slp->lock);
  ++slp->seq;
  smp_mb();
}

static void write_sequnlock(seqlock_t *slp)
{
  smp_mb();
  ++slp->seq;
  spin_unlock(&slp->lock);
}

Figure 8.7: Sequence-Locking Implementation

gins a sequence-lock read-side critical section. Line 17
takes a snapshot of the sequence counter, and line 18 or-
ders this snapshot operation before the caller’s critical
section. Line 19 checks to see if the snapshot is odd, indi-
cating that there is a concurrent writer, and, if so, line 20
jumps back to the beginning. Otherwise, line 21 returns
the value of the snapshot, which the caller will pass to a
later call to read_seqretry().

Quick Quiz 8.8: Why bother with the check on line 19
of read_seqbegin() in Figure 8.7? Given that a new
writer could begin at any time, why not simply incorporate
the check into line 31 of read_seqretry()?

Lines 24-32 show read_seqretry(), which
returns true if there were no writers present since the time of
the corresponding call to read_seqbegin(). Line 29
orders the caller’s prior critical section before line 30’s
fetch of the new snapshot of the sequence counter. Finally,
line 30 checks that the sequence counter has not changed,
in other words, that there has been no writer, and returns
true if so.

Quick Quiz 8.9: What prevents sequence-locking up-
daters from starving readers?

Lines 34-39 show write_seqlock(), which sim-
ply acquires the lock, increments the sequence number,
and executes a memory barrier to ensure that this in-
crement is ordered before the caller’s critical section.
Lines 41-46 show write_sequnlock(), which ex-
ecutes a memory barrier to ensure that the caller’s critical
section is ordered before the increment of the sequence
number on line 44, then releases the lock.

Quick Quiz 8.10: What if something else serializes
writers, so that the lock is not needed?

Quick Quiz 8.11: Why isn’t seq on line 2 of Fig-
ure 8.7 unsigned rather than unsigned long? Af-
after all, if unsigned is good enough for the Linux kernel,
shouldn’t it be good enough for everyone?

Both the read-side and write-side critical sections of
a sequence lock can be thought of as transactions, and
sequence locking therefore can be thought of as a limited
form of transactional memory, which will be discussed in
Section 15.2.

Sequence locks allow writers to defer readers, but not
vice versa. This can result in unfairness and even starva-
tion in writer-heavy workloads. On the other hand, in the
absence of writers, sequence-lock readers are reasonably
fast and scale linearly. It is only human to want the best
of both worlds: fast readers without the possibility of
starvation. In addition, it would also be nice to overcome
sequence locking’s limitations with pointers. The follow-
ing section presents a synchronization mechanism with exactly these properties.

# 8.3 Read-Copy Update (RCU)

This section covers RCU from a number of different perspectives. Section 8.3.1 provides the classic introduction to RCU, Section 8.3.2 covers fundamental RCU concepts, Section 8.3.3 introduces some common uses of RCU, Section 8.3.4 presents the Linux-kernel API, Section 8.3.5 covers a sequence of “toy” implementations of user-level RCU, and finally Section 8.3.6 provides some RCU exercises.

## 8.3.1 Introduction to RCU

Suppose that you are writing a parallel real-time program that needs to access data that is subject to gradual change, perhaps due to changes in temperature, humidity, and barometric pressure. The real-time response constraints on this program are so severe that it is not permissible to spin or block, thus ruling out locking, nor is it permissible to use a retry loop, thus ruling out sequence locks. Fortunately, the temperature and pressure are normally controlled, so that a default hard-coded set of data is usually sufficient.

However, the temperature, humidity, and pressure occasionally deviate too far from the defaults, and in such situations it is necessary to provide data that replaces the defaults. Because the temperature, humidity, and pressure change gradually, providing the updated values is not a matter of urgency, though it must happen within a few minutes. The program is to use a global pointer imaginatively named \textit{gptr} that is normally NULL, which indicates that the default values are to be used. Otherwise, \textit{gptr} points to a structure providing values imaginatively named \textit{a}, \textit{b}, and \textit{c} that are to be used in the real-time calculations.

How can we safely provide updated values when needed without impeding real-time readers?

A classic approach is shown in Figure 8.8. The first row shows the default state, with \textit{gptr} equal to NULL. In the second row, we have allocated a structure which is uninitialized, as indicated by the question marks. In the third row, we have initialized the structure. Next, we assign \textit{gptr} to reference this new element.\footnote{On many computer systems, simple assignment is insufficient due to interference from both the compiler and the CPU. These issues will be covered in Section 8.3.2.} On modern general-purpose systems, this assignment is atomic in the sense that concurrent readers will see either a NULL pointer or a pointer to the new structure \textit{p}, but not some mash-up containing bits from both values. Each reader is therefore guaranteed to either get the default value of NULL or to get the newly installed non-default values, but either way each reader will see a consistent result. Even better, readers need not use any expensive synchronization primitives, so this approach is quite suitable for real-time use.\footnote{Again, on many computer systems, additional work is required to prevent interference from the compiler, and, on DEC Alpha systems, the CPU as well. This will be covered in Section 8.3.2.}

But sooner or later, it will be necessary to remove data that is being referenced by concurrent readers. Let us move to a more complex example where we are removing an element from a linked list, as shown in Figure 8.9. This list initially contains elements \textit{A}, \textit{B}, and \textit{C}, and we need to remove element \textit{B}. First, we use \texttt{list_del()}.
One constraint that fits well with some types of real-time operating systems (as well as some operating-system kernels) is to consider the case where threads are not subject to preemption. In such non-preemptible environments, each thread runs until it explicitly and voluntarily blocks. This means that an infinite loop without blocking will render a CPU useless for any other purpose from the start of the infinite loop onwards. Non-preemptibility also requires that threads be prohibited from blocking while holding spinlocks. Without this prohibition, all CPUs might be consumed by threads spinning attempting to acquire a spinlock held by a blocked thread. The spinning threads will not relinquish their CPUs until they acquire the lock, but the thread holding the lock cannot possibly release it until one of the spinning threads relinquishes a CPU. This is a classic deadlock situation.

But how can we tell when the readers are finished? It is tempting to consider a reference-counting scheme, but Figure 4.3 in Chapter 4 shows that this can also result in long delays, just as can the locking and sequence-locking approaches that we already rejected.

Let’s consider the logical extreme where the readers do absolutely nothing to announce their presence. This approach clearly allows optimal performance for readers (after all, free is a very good price), but leaves open the question of how the updater can possibly determine when all the old readers are done. We clearly need some additional constraints if we are to provide a reasonable answer to this question.

3 And yet again, this approximates reality, which will be expanded on in Section 8.3.2.

Non-preemptibility also requires that threads be prohibited from blocking while holding spinlocks. Without this prohibition, all CPUs might be consumed by threads spinning attempting to acquire a spinlock held by a blocked thread. The spinning threads will not relinquish their CPUs until they acquire the lock, but the thread holding the lock cannot possibly release it until one of the spinning threads relinquishes a CPU. This is a classic deadlock situation.

A schematic of this approach is shown in Figure 8.10, with time advancing from the top of the figure to the bottom.

Although production-quality implementations of this approach can be quite complex, a toy implementatoin is exceedingly simple:

```c
1 for_each_cpu(cpu)
2    run_on(cpu);
```

The `for_online_cpu()` primitive iterates over all CPUs, and the `run_on()` function causes the current thread to execute on the specified CPU, which forces the destination CPU to execute a context switch. Therefore, once the `for_online_cpu()` has completed, each CPU has executed a context switch, which in turn guarantees that all pre-existing reader threads have completed.

4 In contrast, an infinite loop in a preemtible environment might be preempted. This infinite loop might still waste considerable CPU time, but the CPU in question would nevertheless be able to do other work.
Figure 8.10: Waiting for Pre-Existing Readers

Please note that this approach is not production quality. Correct handling of a number of corner cases and the need for a number of powerful optimizations mean that production-quality implementations have significant additional complexity. In addition, RCU implementations for preemptible environments require that readers actually do something. However, this simple non-preemptible approach is conceptually complete, and forms a good initial basis for understanding the RCU fundamentals covered in the following section.

8.3.2 RCU Fundamentals

Authors: Paul E. McKenney and Jonathan Walpole

Read-copy update (RCU) is a synchronization mechanism that was added to the Linux kernel in October of 2002. RCU achieves scalability improvements by allowing reads to occur concurrently with updates. In contrast with conventional locking primitives that ensure mutual exclusion among concurrent threads regardless of whether they be readers or updaters, or with reader-writer locks that allow concurrent reads but not in the presence of updates, RCU supports concurrency between a single updater and multiple readers. RCU ensures that reads are coherent by maintaining multiple versions of objects and ensuring that they are not freed up until all pre-existing read-side critical sections complete. RCU defines and uses efficient and scalable mechanisms for publishing and reading new versions of an object, and also for deferring the collection of old versions. These mechanisms distribute the work among read and update paths in such a way as to make read paths extremely fast. In some cases (non-preemptible kernels), RCU’s read-side primitives have zero overhead.

Quick Quiz 8.12: But doesn’t Section 8.2’s seqlock also permit readers and updaters to get work done concurrently?

This leads to the question “what exactly is RCU?”, and perhaps also to the question “how can RCU possibly work?” (or, not infrequently, the assertion that RCU cannot possibly work). This document addresses these questions from a fundamental viewpoint; later installments look at them from usage and from API viewpoints. This last installment also includes a list of references.

RCU is made up of three fundamental mechanisms, the first being used for insertion, the second being used for deletion, and the third being used to allow readers to tolerate concurrent insertions and deletions. Section 8.3.2.1 describes the publish-subscribe mechanism used for insertion, Section 8.3.2.2 describes how waiting for pre-existing RCU readers enabled deletion, and Section 8.3.2.3 discusses how maintaining multiple versions of recently updated objects permits concurrent insertions and deletions. Finally, Section 8.3.2.4 summarizes RCU fundamentals.

8.3.2.1 Publish-Subscribe Mechanism

One key attribute of RCU is the ability to safely scan data, even though that data is being modified concurrently. To provide this ability for concurrent insertion, RCU uses what can be thought of as a publish-subscribe mechanism.

```c
1 struct foo {
2    int a;
3    int b;
4    int c;
5  };
6 struct foo *gp = NULL;
7 /* . . . */
8 p = kmalloc(sizeof(*p), GFP_KERNEL);
9 p->a = 1;
10 p->b = 2;
11 p->c = 3;
12 gp = p;
```

Figure 8.11: Data Structure Publication (Unsafe)
8.3. READ-COPY UPDATE (RCU)

For example, consider an initially NULL global pointer gp that is to be modified to point to a newly allocated and initialized data structure. The code fragment shown in Figure 8.11 (with the addition of appropriate locking) might be used for this purpose.

Unfortunately, there is nothing forcing the compiler and CPU to execute the last four assignment statements in order. If the assignment to gp happens before the initialization of p fields, then concurrent readers could see the uninitialized values. Memory barriers are required to keep things ordered, but memory barriers are notoriously difficult to use. We therefore encapsulate them into a primitive rcu_assign_pointer() that has publication semantics. The last four lines would then be as follows:

```c
1 p->a = 1;
2 p->b = 2;
3 p->c = 3;
4 rcu_assign_pointer(gp, p);
```

The rcu_assign_pointer() would publish the new structure, forcing both the compiler and the CPU to execute the assignment to gp after the assignments to the fields referenced by p.

However, it is not sufficient to only enforce ordering at the updater, as the reader must enforce proper ordering as well. Consider for example the following code fragment:

```c
1 p = gp;
2 if (p != NULL) {
3 do_something_with(p->a, p->b, p->c);
4 }
```

Although this code fragment might well seem immune to misordering, unfortunately, the DEC Alpha CPU [McK05a, McK05b] and value-speculation compiler optimizations can, believe it or not, cause the values of p->a, p->b, and p->c to be fetched before the value of p. This is perhaps easiest to see in the case of value-speculation compiler optimizations, where the compiler guesses the value of p fetches p->a, p->b, and p->c then fetches the actual value of p in order to check whether its guess was correct. This sort of optimization is quite aggressive, perhaps insanely so, but does actually occur in the context of profile-driven optimization.

Clearly, we need to prevent this sort of skullduggery on the part of both the compiler and the CPU. The rcu_dereference() primitive uses whatever memory-barrier instructions and compiler directives are required for this purpose:

```c
1 rcu_read_lock();
2 p = rcu_dereference(gp);
3 if (p != NULL) {
4 do_something_with(p->a, p->b, p->c);
5 }
6 rcu_read_unlock();
```

The rcu_dereference() primitive can thus be thought of as subscribing to a given value of the specified pointer, guaranteeing that subsequent dereference operations will see any initialization that occurred before the corresponding rcu_assign_pointer() operation that published that pointer. The rcu_read_lock() and rcu_read_unlock() calls are absolutely required: they define the extent of the RCU reads-side critical section. Their purpose is explained in Section 8.3.2.2, however, they never spin or block, nor do they prevent the list_add_rcu() from executing concurrently. In fact, in non-CONFIG_PREEMPT kernels, they generate absolutely no code.

Although rcu_assign_pointer() and rcu_dereference() can in theory be used to construct any conceivable RCU-protected data structure, in practice it is often better to use higher-level constructs. Therefore, the rcu_assign_pointer() and rcu_dereference() primitives have been embedded in special RCU variants of Linux’s list-manipulation API. Linux has two variants of doubly linked list, the circular struct list_head and the linear struct hlist_head/struct hlist_node pair. The former is laid out as shown in Figure 8.12, where the green boxes represent the list header and the blue boxes represent the elements in the list. This notation is cumbersome, and will therefore be abbreviated as shown in Figure 8.13.

Adapting the pointer-publish example for the linked list results in the code shown in Figure 8.14.
CHAPTER 8. DEFERRED PROCESSING

1 struct foo {
2     struct list_head *list;
3     int a;
4     int b;
5     int c;
6 };
7 LIST_HEAD(head);
8 /
9 /* . . . */
10 p = kmalloc(sizeof(*p), GFP_KERNEL);
11 p->a = 1;
12 p->b = 2;
13 p->c = 3;
14 list_add_rcu(&p->list, &head);

Figure 8.14: RCU Data Structure Publication

Line 15 must be protected by some synchronization mechanism (most commonly some sort of lock) to prevent multiple list_add() instances from executing concurrently. However, such synchronization does not prevent this list_add() instance from executing concurrently with RCU readers.

Subscribing to an RCU-protected list is straightforward:

1 rcu_read_lock();
2 list_for_each_entry_rcu(p, head, list) {
3     do_something_with(p->a, p->b, p->c);
4 };
5 rcu_read_unlock();

The list_add_rcu() primitive publishes an entry into the specified list, guaranteeing that the corresponding list_for_each_entry_rcu() invocation will properly subscribe to this same entry.

Quick Quiz 8.13: What prevents the list_for_each_entry_rcu() from getting a segfault if it happens to execute at exactly the same time as the list_add_rcu()?

Linux’s other doubly linked list, the hlist, is a linear list, which means that it needs only one pointer for the header rather than the two required for the circular list, as shown in Figure 8.15. Thus, use of hlist can halve the memory consumption for the hash-bucket arrays of large hash tables. As before, this notation is cumbersome, so hlists will be abbreviated in the same way lists are, as shown in Figure 8.16.

Publishing a new element to an RCU-protected hlist is quite similar to doing so for the circular list, as shown in Figure 8.16.

As before, line 15 must be protected by some sort of synchronization mechanism, for example, a lock.

Subscribing to an RCU-protected hlist is also similar to the circular list:

1 rcu_read_lock();
2 hlist_for_each_entry_rcu(p, q, head, list) {
3     do_something_with(p->a, p->b, p->c);
4 };
5 rcu_read_unlock();

Quick Quiz 8.14: Why do we need to pass two pointers into hlist_for_each_entry_rcu() when only one is needed for list_for_each_entry_rcu()?

The set of RCU publish and subscribe primitives are shown in Table 8.2, along with additional primitives to “unpublish”, or retract.

Note that the list_replace_rcu(), list_del_rcu(), hlist_replace_rcu(), and hlist_del_rcu() APIs add a complication. When is it safe to free up the data element that was replaced or removed? In particular, how can we possibly know when all the readers have released their references to that data element?

These questions are addressed in the following section.

8.3.2.2 Wait For Pre-Existing RCU Readers to Complete

In its most basic form, RCU is a way of waiting for things to finish. Of course, there are a great many other ways of waiting for things to finish, including reference counts,
8.3. READ-COPY UPDATE (RCU)

<table>
<thead>
<tr>
<th>Category</th>
<th>Publish</th>
<th>Retract</th>
<th>Subscribe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointers</td>
<td>rcu_assign_pointer()</td>
<td>rcu_assign_pointer(..., NULL)</td>
<td>rcu_decrefence()</td>
</tr>
<tr>
<td>Lists</td>
<td>list_add_tail_rcu()</td>
<td>list_del_rcu()</td>
<td>list_for_each_entry_rcu()</td>
</tr>
<tr>
<td></td>
<td>list_replace_rcu()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>hlist_add_after_rcu()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>hlist_add_before_rcu()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>hlist_add_head_rcu()</td>
<td>hlist_del_rcu()</td>
<td>hlist_for_each_entry_rcu()</td>
</tr>
<tr>
<td></td>
<td>hlist_replace_rcu()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8.2: RCU Publish and Subscribe Primitives

```c
1 struct foo {
2     struct list_head *list;
3     int a;
4     int b;
5     int c;
6 };
7 LIST_HEAD(head);
8
9 /* . . . */
10
11 p = search(head, key);
12 if (p == NULL) {
13     /* Take appropriate action, unlock, & return. */
14 }
15 q = kmalloc(sizeof(*p), GFP_KERNEL);
16     q = *p;
17     q->b = 2;
18     q->c = 3;
19     list_replace_rcu(&p->list, &q->list);
20     synchronize_rcu();
21     kfree(p);
```

Figure 8.18: Canonical RCU Replacement Example

reader-writer locks, events, and so on. The great advantage of RCU is that it can wait for each of (say) 20,000 different things without having to explicitly track each and every one of them, and without having to worry about the performance degradation, scalability limitations, complex deadlock scenarios, and memory-leak hazards that are inherent in schemes using explicit tracking.

In RCU’s case, the things waited on are called “RCU read-side critical sections”. An RCU read-side critical section starts with an `rcu_read_lock()` primitive, and ends with a corresponding `rcu_readunlock()` primitive. RCU read-side critical sections can be nested, and may contain pretty much any code, as long as that code does not explicitly block or sleep (although a special form of RCU called SRCU [McK06b] does permit general sleeping in SRCU read-side critical sections). If you abide by these conventions, you can use RCU to wait for any desired piece of code to complete.

RCU accomplishes this feat by indirectly determining when these other things have finished [McK07g, McK07a], as is described in detail in Appendix D.

In particular, as shown in Figure 8.17, RCU is a way of waiting for pre-existing RCU read-side critical sections to completely finish, including memory operations executed by those critical sections. However, note that RCU read-side critical sections that begin after the beginning of a given grace period can and will extend beyond the end of that grace period.

The following pseudocode shows the basic form of algorithms that use RCU to wait for readers:

1. Make a change, for example, replace an element in a linked list.
2. Wait for all pre-existing RCU read-side critical sections to completely finish (for example, by using the `synchronize_rcu()` primitive). The key observation here is that subsequent RCU read-side critical sections have no way to gain a reference to the newly removed element.
3. Clean up, for example, free the element that was replaced above.

The code fragment shown in Figure 8.18, adapted from those in Section 8.3.2.1, demonstrates this process, with field `a` being the search key.
Lines 19, 20, and 21 implement the three steps called out above. Lines 16-19 give RCU (“read-copy update”) its name: while permitting concurrent reads, line 16 copies and lines 17-19 do an update.

As discussed in Section 8.3.1, the synchronize_rcu() primitive can be quite simple (see Section 8.3.5 for additional “toy” RCU implementations). However, production-quality implementations must deal with difficult corner cases and also incorporate powerful optimizations, both of which result in significant complexity. Although it is good to know that there is a simple conceptual implementation of synchronize_rcu(), other questions remain. For example, what exactly do RCU readers see when traversing a concurrently updated list? This question is addressed in the following section.

### 8.3.2.3 Maintain Multiple Versions of Recently Updated Objects

This section demonstrates how RCU maintains multiple versions of lists to accommodate synchronization-free readers. Two examples are presented showing how an element that might be referenced by a given reader must remain intact while that reader remains in its RCU read-side critical section. The first example demonstrates deletion of a list element, and the second example demonstrates replacement of an element.

**Example 1: Maintaining Multiple Versions During Deletion**

We can now revisit the deletion example from Section 8.3.1, but now with the benefit of a firm understanding of the fundamental concepts underlying RCU. To begin this new version of the deletion example, we will modify lines 11-21 in Figure 8.18 to read as follows:

```c
1 p = search(head, key);
2 if (p != NULL) {
3  list_del_rcu(&p->list);
4  synchronize_rcu();
5  kfree(p);
6 }
```

This code will update the list as shown in Figure 8.19. The triples in each element represent the values of fields a, b, and c, respectively. The red-shaded elements indicate that RCU readers might be holding references to them. Please note that we have omitted the backwards pointers and the link from the tail of the list to the head for clarity.

After the list_del_rcu() on line 3 has completed, the 5, 6, 7 element has been removed from the list, as shown in the second row of Figure 8.19. Since readers do not synchronize directly with updaters, readers might be concurrently scanning this list. These concurrent readers might or might not see the newly removed element, depending on timing. However, readers that were delayed (e.g., due to interrupts, ECC memory errors, or, in CONFIG_PREEMPT_RT kernels, preemption) just after fetching a pointer to the newly removed element might see the old version of the list for quite some time after the removal. Therefore, we now have two versions of the list, one with element 5, 6, 7 and one without. The 5, 6, 7 element is shaded yellow, indicating that old readers might still be referencing it, but that new readers cannot obtain a reference to it.

Please note that readers are not permitted to maintain references to element 5, 6, 7 after exiting from their RCU read-side critical sections. Therefore, once the synchronize_rcu() on line 4 completes, so that all pre-existing readers are guaranteed to have completed, there can be no more readers referencing this element, as indicated by its green shading on the third row of Figure 8.19. We are thus back to a single version of the list.

At this point, the 5, 6, 7 element may safely be freed, as shown on the final row of Figure 8.19. At this point, we have completed the deletion of element 5, 6, 7. The
The following section covers replacement.

Example 2: Maintaining Multiple Versions During Replacement  To start the replacement example, here are the last few lines of the example shown in Figure 8.18:

```c
1 q = kmalloc(sizeof(*p), GFP_KERNEL);
2 *q = *p;
3 q->b = 2;
4 q->c = 3;
5 list_replace_rcu(&p->list, &q->list);
6 synchronize_rcu();
7 kfree(p);
```

The initial state of the list, including the pointer \( p \), is the same as for the deletion example, as shown on the first row of Figure 8.20.

As before, the triples in each element represent the values of fields \( a \), \( b \), and \( c \), respectively. The red-shaded elements might be referenced by readers, and because readers do not synchronize directly with updaters, readers might run concurrently with this entire replacement process. Please note that we again omit the backwards pointers and the link from the tail of the list to the head for clarity.

The following text describes how to replace the \( 5, 6, 7 \) element with \( 5, 2, 3 \) in such a way that any given reader sees one of these two values.

Line 1 \texttt{kmalloc()} is a replacement element, as follows, resulting in the state as shown in the second row of Figure 8.20. At this point, no reader can hold a reference to the newly allocated element (as indicated by its green shading), and it is uninitialized (as indicated by the question marks).

Line 2 copies the old element to the new one, resulting in the state as shown in the third row of Figure 8.20. The newly allocated element still cannot be referenced by readers, but it is now initialized.

Line 3 updates \( q->b \) to the value “2”, and line 4 updates \( q->c \) to the value “3”, as shown on the fourth row of Figure 8.20.

Now, line 5 does the replacement, so that the new element is finally visible to readers, and hence is shaded red, as shown on the fifth row of Figure 8.20. At this point, as shown below, we have two versions of the list. Pre-existing readers might see the \( 5, 6, 7 \) element (which is therefore now shaded yellow), but new readers will instead see the \( 5, 2, 3 \) element. But any given reader is guaranteed to see some well-defined list.

After the \texttt{synchronize_rcu()} on line 6 returns, a grace period will have elapsed, and so all reads that started before the \texttt{list_replace_rcu()} will have
completed. In particular, any readers that might have been holding references to the 5, 6, 7 element are guaranteed to have exited their RCU read-side critical sections, and are thus prohibited from continuing to hold a reference. Therefore, there can no longer be any readers holding references to the old element, as indicated its green shading in the sixth row of Figure 8.20. As far as the readers are concerned, we are back to having a single version of the list, but with the new element in place of the old.

After the `kfree()` on line 7 completes, the list will appear as shown on the final row of Figure 8.20.

Despite the fact that RCU was named after the replacement case, the vast majority of RCU usage within the Linux kernel relies on the simple deletion case shown in Section 8.3.2.3.

Discussion These examples assumed that a mutex was held across the entire update operation, which would mean that there could be at most two versions of the list active at a given time.

Quick Quiz 8.15: How would you modify the deletion example to permit more than two versions of the list to be active? ■

Quick Quiz 8.16: How many RCU versions of a given list can be active at any given time? ■

This sequence of events shows how RCU updates use multiple versions to safely carry out changes in presence of concurrent readers. Of course, some algorithms cannot gracefully handle multiple versions. There are techniques for adapting such algorithms to RCU [McK04], but these are beyond the scope of this section.

8.3.2.4 Summary of RCU Fundamentals

This section has described the three fundamental components of RCU-based algorithms:

1. a publish-subscribe mechanism for adding new data,
2. a way of waiting for pre-existing RCU readers to finish, and
3. a discipline of maintaining multiple versions to permit change without harming or unduly delaying concurrent RCU readers.

Quick Quiz 8.17: How can RCU updaters possibly delay RCU readers, given that the `rcu_read_lock()` and `rcu_read_unlock()` primitives neither spin nor block? ■

These three RCU components allow data to be updated in face of concurrent readers, and can be combined in different ways to implement a surprising variety of different types of RCU-based algorithms, some of which are described in the following section.

8.3.3 RCU Usage

This section answers the question "what is RCU?" from the viewpoint of the uses to which RCU can be put. Because RCU is most frequently used to replace some existing mechanism, we look at it primarily in terms of its relationship to such mechanisms, as listed in Table 8.3. Following the sections listed in this table, Section 8.3.3.8 provides a summary.

8.3.3.1 RCU is a Reader-Writer Lock Replacement

Perhaps the most common use of RCU within the Linux kernel is as a replacement for reader-writer locking in read-intensive situations. Nevertheless, this use of RCU was not immediately apparent to me at the outset, in fact, I chose to implement something similar to brlock before implementing a general-purpose RCU implementation back in the early 1990s. Each and every one of the uses I envisioned for the proto-brlock primitive was instead implemented using RCU. In fact, it was more than three years before the proto-brlock primitive saw its first use. Boy, did I feel foolish!

The key similarity between RCU and reader-writer locking is that both have read-side critical sections that can execute in parallel. In fact, in some cases, it is possible to mechanically substitute RCU API members for the corresponding reader-writer lock API members. But first, why bother?

Advantages of RCU include performance, deadlock immunity, and real-time latency. There are, of course, limitations to RCU, including the fact that readers and updaters run concurrently, that low-priority RCU readers can block high-priority threads waiting for a grace period.
to elapse, and that grace-period latencies can extend for many milliseconds. These advantages and limitations are discussed in the following sections.

**Performance** The read-side performance advantages of RCU over reader-writer locking are shown in Figure 8.21.

Quick Quiz 8.18: WTF? How the heck do you expect me to believe that RCU has a 100-femtosecond overhead when the clock period at 3GHz is more than 300 picoseconds?

Note that reader-writer locking is orders of magnitude slower than RCU on a single CPU, and is almost two additional orders of magnitude slower on 16 CPUs. In contrast, RCU scales quite well. In both cases, the error bars span a single standard deviation in either direction.

A more moderate view may be obtained from a CONFIG_PREEMPT kernel, though RCU still beats reader-writer locking by between one and three orders of magnitude, as shown in Figure 8.22. Note the high variability of reader-writer locking at larger numbers of CPUs. The error bars span a single standard deviation in either direction.

Of course, the low performance of reader-writer locking in Figure 8.22 is exaggerated by the unrealistic zero-length critical sections. The performance advantages of RCU become less significant as the overhead of the critical section increases, as shown in Figure 8.23 for a 16-CPU system, in which the y-axis represents the sum of the overhead of the read-side primitives and that of the critical section.

Quick Quiz 8.19: Why does both the variability and overhead of rwlock decrease as the critical-section overhead increases?

However, this observation must be tempered by the fact that a number of system calls (and thus any RCU read-side critical sections that they contain) can complete within a few microseconds.

In addition, as is discussed in the next section, RCU read-side primitives are almost entirely deadlock-immune.

**Deadlock Immunity** Although RCU offers significant performance advantages for read-mostly workloads, one of the primary reasons for creating RCU in the first place was in fact its immunity to read-side deadlocks. This immunity stems from the fact that RCU read-side primitives do not block, spin, or even do backwards branches, so that their execution time is deterministic. It is therefore impossible for them to participate in a deadlock cycle.

Quick Quiz 8.20: Is there an exception to this deadlock immunity, and if so, what sequence of events could lead to deadlock?

An interesting consequence of RCU’s read-side deadlock immunity is that it is possible to unconditionally upgrade an RCU reader to an RCU updater. Attempting to do such an upgrade with reader-writer locking results
in deadlock. A sample code fragment that does an RCU read-to-update upgrade follows:

```c
rcu_read_lock();
list_for_each_entry_rcu(p, &head, list_field) {
  do_something_with(p);
  if (need_update(p)) {
    spin_lock(my_lock);
    do_update(p);
    spin_unlock(&my_lock);
  }
}
rcu_read_unlock();
```

Note that `do_update()` is executed under the protection of the lock and under RCU read-side protection.

Another interesting consequence of RCU’s deadlock immunity is its immunity to a large class of priority inversion problems. For example, low-priority RCU readers cannot prevent a high-priority RCU updater from acquiring the update-side lock. Similarly, a low-priority RCU updater cannot prevent high-priority RCU readers from entering an RCU read-side critical section.

**Realtime Latency** Because RCU read-side primitives neither spin nor block, they offer excellent realtime latencies. In addition, as noted earlier, this means that they are immune to priority inversion involving the RCU read-side primitives and locks.

However, RCU is susceptible to more subtle priority-inversion scenarios, for example, a high-priority process blocked waiting for an RCU grace period to elapse can be blocked by low-priority RCU readers in -rt kernels. This can be solved by using RCU priority boosting [McK07d, GMTW08].

**RCU Readers and Updaters Run Concurrently** Because RCU readers never spin nor block, and because updaters are not subject to any sort of rollback or abort semantics, RCU readers and updaters must necessarily run concurrently. This means that RCU readers might access stale data, and might even see inconsistencies, either of which can render conversion from reader-writer locking to RCU non-trivial.

However, in a surprisingly large number of situations, inconsistencies and stale data are not problems. The classic example is the networking routing table. Because routing updates can take considerable time to reach a given system (seconds or even minutes), the system will have been sending packets the wrong way for quite some time when the update arrives. It is usually not a problem to continue sending updates the wrong way for a few additional milliseconds. Furthermore, because RCU updaters can make changes without waiting for RCU readers to finish, the RCU readers might well see the change more quickly than would batch-fair reader-writer-locking readers, as shown in Figure 8.24.

Once the update is received, the rwlock writer cannot proceed until the last reader completes, and subsequent readers cannot proceed until the writer completes. However, these subsequent readers are guaranteed to see the new value, as indicated by the green background. In contrast, RCU readers and updaters do not block each other,
which permits the RCU readers to see the updated values sooner. Of course, because their execution overlaps that of the RCU updater, all of the RCU readers might well see updated values, including the three readers that started before the update. Nevertheless only the RCU readers with green backgrounds are guaranteed to see the updated values, again, as indicated by the green background.

Reader-writer locking and RCU simply provide different guarantees. With reader-writer locking, any reader that begins after the writer begins is guaranteed to see new values, and any reader that attempts to begin while the writer is spinning might or might not see new values, depending on the reader/writer preference of the_rwlock implementation in question. In contrast, with RCU, any reader that begins after the updater completes is guaranteed to see new values, and any reader that completes after the updater begins might or might not see new values, depending on timing.

The key point here is that, although reader-writer locking does indeed guarantee consistency within the confines of the computer system, there are situations where this consistency comes at the price of increased inconsistency with the outside world. In other words, reader-writer locking obtains internal consistency at the price of silently stale data with respect to the outside world.

Nevertheless, there are situations where inconsistency and stale data within the confines of the system cannot be tolerated. Fortunately, there are a number of approaches that avoid inconsistency and stale data [McK04, ACMS03], and some methods based on reference counting are discussed in Section 8.1.

**Low-Priority RCU Readers Can Block High-Priority Reclaimers** In Realtime RCU [GMTW08] (see Section D.4), SRCU [McK06b] (see Section D.1, or QRCU [McK07f] (see Section F.6, each of which is described in the final installment of this series, a preempted reader will prevent a grace period from completing, even if a high-priority task is blocked waiting for that grace period to complete. Realtime RCU can avoid this problem by substituting call_rcu() for synchronize_rcu() or by using RCU priority boosting [McK07d, GMTW08], which is still in experimental status as of early 2008. It might become necessary to augment SRCU and QRCU with priority boosting, but not before a clear real-world need is demonstrated.

**RCU Grace Periods Extend for Many Milliseconds** With the exception of QRCU and several of the “toy” RCU implementations described in Section 8.3.5, RCU grace periods extend for multiple milliseconds. Although there are a number of techniques to render such long delays harmless, including use of the asynchronous interfaces where available (call_rcu() and call_rcu_bh()), this situation is a major reason for the rule of thumb that RCU be used in read-mostly situations.

**Comparison of Reader-Writer Locking and RCU Code** In the best case, the conversion from reader-writer locking to RCU is quite simple, as shown in Figures 8.25, 8.26, and 8.27, all taken from Wikipedia [MPA+06].

More elaborately cases of replacing reader-writer locking with RCU are beyond the scope of this document.

**8.3.3.2 RCU is a Restricted Reference-Counting Mechanism** Because grace periods are not allowed to complete while there is an RCU read-side critical section in progress, the RCU read-side primitives may be used as a restricted reference-counting mechanism. For example, consider the following code fragment:

```c
1 rcu_read_lock(); /* acquire reference. */
2 p = rcu_dereference(head);
3 /* do something with p. */
4 rcu_read_unlock(); /* release reference. */
```

The rcu_read_lock() primitive can be thought of as acquiring a reference to p, because a grace period starting after the rcu_dereference() assigns to p cannot possibly end until after we reach the matching rcu_read_unlock(). This reference-counting scheme is restricted in that we are not allowed to block in RCU read-side critical sections, nor are we permitted to hand off an RCU read-side critical section from one task to another.

Regardless of these restrictions, the following code can safely delete p:

```c
1 spin_lock(&mylock);
2 p = head;
3 rcu_assign_pointer(head, NULL);
4 spin_unlock(&mylock);
5 /* Wait for all references to be released. */
6 synchronize_rcu();
7 kfree(p);
```

The assignment to head prevents any future references to p from being acquired, and the synchronize_rcu() waits for any previously acquired references to be released.

**Quick Quiz 8.21:** But wait! This is exactly the same code that might be used when thinking of RCU as a replacement for reader-writer locking! What gives?
1 struct el {
2   struct list_head lp;
3   long key;
4   spinlock_t mutex;
5   int data;
6   /* Other data fields */
7 }; 8
8 DEFINE_RWLOCK(listmutex);
9 LIST_HEAD(head);

Figure 8.25: Converting Reader-Writer Locking to RCU: Data

1 int search(long key, int *result) 1 int search(long key, int *result)
2 { 2 {
3    struct el *p; 3    struct el *p;
4    read_lock(&listmutex); 4    rcu_read_lock();
5    list_for_each_entry(p, &head, lp) { 5    list_for_each_entry_rcu(p, &head, lp) {
6       if (p->key == key) { 6       if (p->key == key) {
7          *result = p->data; 7          *result = p->data;
8         read_unlock(&listmutex); 8         rcu_read_unlock();
9       } 9       return 1;
10      } 10     }
11     read_unlock(&listmutex); 11     rcu_read_unlock();
12     return 0; 12     return 0;
13 } 13 }

Figure 8.26: Converting Reader-Writer Locking to RCU: Search

1 int delete(long key) 1 int delete(long key)
2 { 2 {
3    struct el *p; 3    struct el *p;
4    write_lock(&listmutex); 4    spin_lock(&listmutex);
5    list_for_each_entry(p, &head, lp) { 5    list_for_each_entry(p, &head, lp) {
6       if (p->key == key) 6       if (p->key == key) {
7          list_del(&p->lp); 7          list_del_rcu(&p->lp);
8          write_unlock(&listmutex); 8          spin_unlock(&listmutex);
9          kfree(p); 9          synchronize_rcu();
10         kfree(p); 10         spin_unlock(&listmutex);
11         return 1; 11         return 1;
12      } 12     }
13     } 13     }
14     return 0; 14     return 0;
15 } 15 }

Figure 8.27: Converting Reader-Writer Locking to RCU: Deletion
Of course, RCU can also be combined with traditional reference counting, as has been discussed on LKML and as summarized in Section 8.1.

But why bother? Again, part of the answer is performance, as shown in Figure 8.28, again showing data taken on a 16-CPU 3GHz Intel x86 system.

Quick Quiz 8.22: Why the dip in refcnt overhead near 6 CPUs?

And, as with reader-writer locking, the performance advantages of RCU are most pronounced for short-duration critical sections, as shown Figure 8.29 for a 16-CPU system. In addition, as with reader-writer locking, many system calls (and thus any RCU read-side critical sections that they contain) complete in a few microseconds.

However, the restrictions that go with RCU can be quite onerous. For example, in many cases, the prohibition against sleeping while in an RCU read-side critical section would defeat the entire purpose. The next section looks at ways of addressing this problem, while also reducing the complexity of traditional reference counting, at least in some cases.

8.3.3.3 RCU is a Bulk Reference-Counting Mechanism

As noted in the preceding section, traditional reference counters are usually associated with a specific data structure, or perhaps a specific group of data structures. However, maintaining a single global reference counter for a large variety of data structures typically results in bouncing the cache line containing the reference count. Such cache-line bouncing can severely degrade performance.

In contrast, RCU’s light-weight read-side primitives permit extremely frequent read-side usage with negligible performance degradation, permitting RCU to be used as a "bulk reference-counting" mechanism with little or no performance penalty. Situations where a reference must be held by a single task across a section of code that blocks may be accommodated with Sleepable RCU (SRCU) [McK06b]. This fails to cover the not-uncommon situation where a reference is "passed" from one task to another, for example, when a reference is acquired when starting an I/O and released in the corresponding completion interrupt handler. (In principle, this could be handled by the SRCU implementation, but in practice, it is not yet clear whether this is a good tradeoff.)

Of course, SRCU brings restrictions of its own, namely that the return value from srcu_read_lock() be passed into the corresponding srcu_read_unlock(), and that no SRCU primitives be invoked from hardware irq handlers or from NMI/SMI handlers. The jury is still out as to how much of a problem is presented by these restrictions, and as to how they can best be handled.
8.3.3.4 RCU is a Poor Man’s Garbage Collector

A not-uncommon exclamation made by people first learning about RCU is "RCU is sort of like a garbage collector!". This exclamation has a large grain of truth, but it can also be misleading.

Perhaps the best way to think of the relationship between RCU and automatic garbage collectors (GCs) is that RCU resembles a GC in that the timing of collection is automatically determined, but that RCU differs from a GC in that: (1) the programmer must manually indicate when a given data structure is eligible to be collected, and (2) the programmer must manually mark the RCU read-side critical sections where references might legitimately be held.

Despite these differences, the resemblance does go quite deep, and has appeared in at least one theoretical analysis of RCU. Furthermore, the first RCU-like mechanism I am aware of used a garbage collector to handle the grace periods. Nevertheless, a better way of thinking of RCU is described in the following section.

8.3.3.5 RCU is a Way of Providing Existence Guarantees

Gamsa et al. [GKAS99] discuss existence guarantees and describe how a mechanism resembling RCU can be used to provide these existence guarantees (see section 5 on page 7 of the PDF), and Section 6.4 discusses how to guarantee existence via locking, along with the ensuing disadvantages of doing so. The effect is that if any RCU-protected data element is accessed within an RCU read-side critical section, that data element is guaranteed to remain in existence for the duration of that RCU read-side critical section.

Figure 8.30 demonstrates how RCU-based existence guarantees can enable per-element locking via a function that deletes an element from a hash table. Line 6 computes a hash function, and line 7 enters an RCU read-side critical section. If line 9 finds that the corresponding bucket of the hash table is empty or that the element present is not the one we wish to delete, then line 10 exits the RCU read-side critical section and line 11 indicates failure.

Quick Quiz 8.23: What if the element we need to delete is not the first element of the list on line 9 of Figure 8.30?

Otherwise, line 13 acquires the update-side spinlock, and line 14 then checks that the element is still the one that we want. If so, line 15 leaves the RCU read-side critical section, line 16 removes it from the table, line 17 releases the lock, line 18 waits for all pre-existing RCU read-side critical sections to complete, line 19 frees the newly removed element, and line 20 indicates success. If the element is no longer the one we want, line 22 releases the lock, line 23 leaves the RCU read-side critical section, and line 24 indicates failure to delete the specified key.

Quick Quiz 8.24: Why is it OK to exit the RCU read-side critical section on line 15 of Figure 8.30 before releasing the lock on line 17?

Quick Quiz 8.25: Why not exit the RCU read-side critical section on line 23 of Figure 8.30 before releasing the lock on line 22?

Alert readers will recognize this as only a slight variation on the original "RCU is a way of waiting for things to finish" theme, which is addressed in Section 8.3.3.7. They might also note the deadlock-immunity advantages over the lock-based existence guarantees discussed in Section 6.4.

8.3.3.6 RCU is a Way of Providing Type-Safe Memory

A number of lockless algorithms do not require that a given data element keep the same identity through a given RCU read-side critical section referencing it—but only if that data element retains the same type. In other words, these lockless algorithms can tolerate a given data element being freed and reallocated as the same type of structure

```c
1 int delete(int key)
2 {
3     struct element *p;
4     int b;
5     b = hashfunction(key);
6     rcu_read_lock();
7     p = rcu_dereference(hashtable[b]);
8     if (p == NULL || p->key != key) {
9         rcu_read_unlock();
10         return 0;
11     }
12     spin_lock(&p->lock);
13     if (hashtable[b] == p && p->key == key) {
14         rcu_read_unlock();
15         hashtable[b] = NULL;
16         spin_unlock(&p->lock);
17         synchronize_rcu();
18         kfree(p);
19         return 1;
20     } else {
21         spin_unlock(&p->lock);
22         rcu_read_unlock();
23         return 0;
24     }
```
while they are referencing it, but must prohibit a change in type. This guarantee, called “type-safe memory” in academic literature [GC96], is weaker than the existence guarantees in the previous section, and is therefore quite a bit harder to work with. Type-safe memory algorithms in the Linux kernel make use of slab caches, specially marking these caches with SLAB_DESTROY_BY_RCU so that RCU is used when returning a freed-up slab to system memory. This use of RCU guarantees that any in-use element of such a slab will remain in that slab, thus retaining its type, for the duration of any pre-existing RCU read-side critical sections.

Quick Quiz 8.26: But what if there is an arbitrarily long series of RCU read-side critical sections in multiple threads, so that at any point in time there is at least one thread in the system executing in an RCU read-side critical section? Wouldn’t that prevent any data from a SLAB_DESTROY_BY_RCU slab ever being returned to the system, possibly resulting in OOM events?

These algorithms typically use a validation step that checks to make sure that the newly referenced data structure really is the one that was requested [LS86, Section 2.5]. These validation checks require that portions of the data structure remain untouched by the free-reallocate process. Such validation checks are usually very hard to get right, and can hide subtle and difficult bugs.

Therefore, although type-safety-based lockless algorithms can be extremely helpful in a very few difficult situations, you should instead use existence guarantees where possible. Simpler is after all almost always better!

8.3.3.7 RCU is a Way of Waiting for Things to Finish

As noted in Section 8.3.2 an important component of RCU is a way of waiting for RCU readers to finish. One of RCU’s great strengths is that it allows you to wait for each of thousands of different things to finish without having to explicitly track each and every one of them, and without having to worry about the performance degradation, scalability limitations, complex deadlock scenarios, and memory-leak hazards that are inherent in schemes that use explicit tracking.

In this section, we will show how synchronize_sched()’s read-side counterparts (which include anything that disables preemption, along with hardware operations and primitives that disable irq) permit you to implement interactions with non-maskable interrupt (NMI) handlers that would be quite difficult if using locking.

```c
1 struct profile_buffer {
2   long size;
3   atomic_t entry[0];
4 });
5 static struct profile_buffer *buf = NULL;
6
7 void nmi_profile(unsigned long pcvalue)
8 { struct profile_buffer *p = rcu_dereference(buf);
9   if (p == NULL)
10      return;
11   if (pcvalue >= p->size)
12      return;
13   atomic_inc(&p->entry[pcvalue]);
14 }
15
16 void nmi_stop(void)
17 { struct profile_buffer *p = buf;
18   if (p == NULL)
19      return;
20   rcu_assign_pointer(buf, NULL);
21   synchronize_sched();
22   kfree(p);
23 }
```

Figure 8.31: Using RCU to Wait for NMIs to Finish

This approach has been called "Pure RCU" [McK04], and it is used in a number of places in the Linux kernel.

The basic form of such "Pure RCU" designs is as follows:

1. Make a change, for example, to the way that the OS reacts to an NMI.
2. Wait for all pre-existing read-side critical sections to completely finish (for example, by using the synchronize_sched() primitive). The key observation here is that subsequent RCU read-side critical sections are guaranteed to see whatever change was made.
3. Clean up, for example, return status indicating that the change was successfully made.

The remainder of this section presents example code adapted from the Linux kernel. In this example, the timer_stop function uses synchronize_sched() to ensure that all in-flight NMI notifications have completed before freeing the associated resources. A simplified version of this code is shown Figure 8.31.

Lines 1-4 define a profile_buffer structure, containing a size and an indefinite array of entries. Line 5 defines a pointer to a profile buffer, which is presumably initialized elsewhere to point to a dynamically allocated region of memory.
Lines 7–16 define the `nmi_profile()` function, which is called from within an NMI handler. As such, it cannot be preempted, nor can it be interrupted by a normal irq handler, however, it is still subject to delays due to cache misses, ECC errors, and cycle stealing by other hardware threads within the same core. Line 9 gets a local pointer to the profile buffer using the `rcu_dereference()` primitive to ensure memory ordering on DEC Alpha, and lines 11 and 12 exit from this function if there is no profile buffer currently allocated, while lines 13 and 14 exit from this function if the `pcvalue` argument is out of range. Otherwise, line 15 increments the profile-buffer entry indexed by the `pcvalue` argument. Note that storing the size with the buffer guarantees that the range check matches the buffer, even if a large buffer is suddenly replaced by a smaller one.

Lines 18–27 define the `nmi_stop()` function, where the caller is responsible for mutual exclusion (for example, holding the correct lock). Line 20 fetches a pointer to the profile buffer, and lines 22 and 23 exit the function if there is no buffer. Otherwise, line 24 NULLs out the profile-buffer pointer (using the `rcu_assign_pointer()` primitive to maintain memory ordering on weakly ordered machines), and line 25 waits for an RCU Sched grace period to elapse, in particular, waiting for all non-preemptible regions of code, including NMI handlers, to complete. Once execution continues at line 26, we are guaranteed that any instance of `nmi_profile()` that obtained a pointer to the old buffer has returned. It is therefore safe to free the buffer, in this case using the `kfree()` primitive.

Quick Quiz 8.27: Suppose that the `nmi_profile()` function was preemptible. What would need to change to make this example work correctly?

In short, RCU makes it easy to dynamically switch among profile buffers (you just try doing this efficiently with atomic operations, or at all with locking?). However, RCU is normally used at a higher level of abstraction, as was shown in the previous sections.

8.3.3.8 RCU Usage Summary

At its core, RCU is nothing more nor less than an API that provides:

1. a publish-subscribe mechanism for adding new data,
2. a way of waiting for pre-existing RCU readers to finish, and
3. a discipline of maintaining multiple versions to permit change without harming or unduly delaying concurrent RCU readers.

That said, it is possible to build higher-level constructs on top of RCU, including the reader-writer-locking, reference-counting, and existence-guarantee constructs listed in the earlier sections. Furthermore, I have no doubt that the Linux community will continue to find interesting new uses for RCU, as well as for any of a number of other synchronization primitives.

8.3.4 RCU Linux-Kernel API

This section looks at RCU from the viewpoint of its Linux-kernel API. Section 8.3.4.1 presents RCU’s wait-to-finish APIs, and Section 8.3.4.2 presents RCU’s publish-subscribe and version-maintenance APIs. Finally, Section 8.3.4.4 presents concluding remarks.

8.3.4.1 RCU has a Family of Wait-to-Finish APIs

The most straightforward answer to “what is RCU” is that RCU is an API used in the Linux kernel, as summarized by Tables 8.4 and 8.5, which shows the wait-for-RCU-readers portions of the non-sleepable and sleepable APIs, respectively, and by Table 8.6, which shows the publish-subscribe portions of the API.

If you are new to RCU, you might consider focusing on just one of the columns in Table 8.4, each of which summarizes one member of the Linux kernel’s RCU API family. For example, if you are primarily interested in understanding how RCU is used in the Linux kernel, “RCU Classic” would be the place to start, as it is used most frequently. On the other hand, if you want to understand RCU for its own sake, “SRCU” has the simplest API. You can always come back for the other columns later.

If you are already familiar with RCU, these tables can serve as a useful reference.

Quick Quiz 8.28: Why do some of the cells in Table 8.4 have exclamation marks (“!”)?

The “RCU Classic” column corresponds to the original RCU implementation, in which RCU read-side critical sections are delimited by `rcu_read_lock()` and `rcu_read_unlock()`, which may be nested. The corresponding synchronous update-side primitives, `synchronize_rcu()`, along with its synonym `synchronize_net()`, wait for any currently executing RCU read-side critical sections to complete. The
## 8.3. READ-COPY UPDATE (RCU)

<table>
<thead>
<tr>
<th>Attribute</th>
<th>RCU Classic</th>
<th>RCU BH</th>
<th>RCU Sched</th>
<th>Realtime RCU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Original</td>
<td>Prevent DDoS attacks</td>
<td>Wait for preempt-disable regions, hardirqs, &amp; NMI</td>
<td>Realtime response</td>
</tr>
<tr>
<td>Availability</td>
<td>2.6.43</td>
<td>2.6.9</td>
<td>2.6.12</td>
<td>2.6.26</td>
</tr>
<tr>
<td>Read-side primitives</td>
<td><code>rcu_read_lock()</code></td>
<td><code>rcu_read_bh_lock()</code></td>
<td><code>preempt_disable()</code></td>
<td><code>rcu_read_lock()</code></td>
</tr>
<tr>
<td></td>
<td><code>rcu_read_unlock()</code></td>
<td><code>rcu_read_unlock_bh()</code></td>
<td><code>preempt_enable()</code></td>
<td><code>rcu_read_unlock()</code></td>
</tr>
<tr>
<td>Update-side primitives</td>
<td><code>synchronize_rcu()</code></td>
<td><code>synchronize_net()</code></td>
<td><code>synchronize_sched()</code></td>
<td><code>synchronize_rcu()</code></td>
</tr>
<tr>
<td>(synchronous)</td>
<td><code>call_rcu()</code></td>
<td><code>call_rcu_bh()</code></td>
<td><code>call_rcu_sched()</code></td>
<td><code>call_rcu()</code></td>
</tr>
<tr>
<td>Update-side primitives</td>
<td><code>rcu_barrier()</code></td>
<td><code>rcu_barrier_bh()</code></td>
<td><code>rcu_barrier_sched()</code></td>
<td><code>rcu_barrier()</code></td>
</tr>
<tr>
<td>(wait for callbacks)</td>
<td><code>synchronize_rcu()</code></td>
<td><code>synchronize_net()</code></td>
<td><code>synchronize_sched()</code></td>
<td><code>synchronize_rcu()</code></td>
</tr>
<tr>
<td>Type-safe memory</td>
<td><code>SLAB_DESTROY_BY_RCU</code></td>
<td><code>SLAB_DESTROY_BY_RCU</code></td>
<td><code>SLAB_DESTROY_BY_RCU</code></td>
<td><code>SLAB_DESTROY_BY_RCU</code></td>
</tr>
<tr>
<td>Read side constraints</td>
<td>No blocking</td>
<td>No irq enabling</td>
<td>No blocking</td>
<td>Only preemption and lock acquisition</td>
</tr>
<tr>
<td>Read side overhead</td>
<td>Preempt disable/enable (free on non-PREEMPT)</td>
<td>BH disable/enable</td>
<td>Preempt disable/enable (free on non-PREEMPT)</td>
<td>Simple instructions, irq disable/enable</td>
</tr>
<tr>
<td>Asynchronous update-side overhead</td>
<td>sub-microsecond</td>
<td>sub-microsecond</td>
<td>sub-microsecond</td>
<td>sub-microsecond</td>
</tr>
<tr>
<td>Grace-period latency</td>
<td>10s of milliseconds</td>
<td>10s of milliseconds</td>
<td>10s of milliseconds</td>
<td>10s of milliseconds</td>
</tr>
<tr>
<td>Non-PREEMPT_RT implement</td>
<td>RCU Classic</td>
<td>RCU BH</td>
<td>RCU Classic</td>
<td>Preemptible RCU</td>
</tr>
<tr>
<td>PREEMPT_RT implement</td>
<td>Preemptible RCU</td>
<td>Realtime RCU</td>
<td>Forced Schedule on all CPUs</td>
<td>Realtime RCU</td>
</tr>
</tbody>
</table>

### Table 8.4: RCU Wait-to-Finish APIs

<table>
<thead>
<tr>
<th>Attribute</th>
<th>SRCU</th>
<th>QRCU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Sleeping readers</td>
<td>Sleeping readers and fast grace periods</td>
</tr>
<tr>
<td>Availability</td>
<td>2.6.19</td>
<td></td>
</tr>
<tr>
<td>Read-side primitives</td>
<td><code>srcu_read_lock()</code></td>
<td><code>grcu_read_lock()</code></td>
</tr>
<tr>
<td></td>
<td><code>srcu_read_unlock()</code></td>
<td><code>grcu_read_unlock()</code></td>
</tr>
<tr>
<td>Update-side primitives</td>
<td><code>synchronize_srcu()</code></td>
<td><code>synchronize_qrcu()</code></td>
</tr>
<tr>
<td>(synchronous)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Update-side primitives</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>(asynchronous/callback)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Update-side primitives</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>(wait for callbacks)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Type-safe memory</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Read side constraints</td>
<td>No <code>synchronize_srcu()</code></td>
<td>No <code>synchronize_qrcu()</code></td>
</tr>
<tr>
<td>Read side overhead</td>
<td>Simple instructions, preempt disable/enable</td>
<td>Atomic increment and decrement of shared variable</td>
</tr>
<tr>
<td>Asynchronous update-side overhead</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Grace-period latency</td>
<td>10s of milliseconds</td>
<td>10s of nanoseconds in absence of readers</td>
</tr>
<tr>
<td>Non-PREEMPT_RT implement</td>
<td>SRCU</td>
<td>N/A</td>
</tr>
<tr>
<td>PREEMPT_RT implement</td>
<td>SRCU</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Table 8.5: Sleepable RCU Wait-to-Finish APIs
length of this wait is known as a “grace period”. The asynchronous update-side primitive, call_rcu(), invokes a specified function with a specified argument after a subsequent grace period. For example, call_rcu(p, f); will result in the “RCU callback” \( f(p) \) being invoked after a subsequent grace period. There are situations, such as when unloading a Linux-kernel module that uses call_rcu(), when it is necessary to wait for all outstanding RCU callbacks to complete [McK07e]. The rcu_barrier() primitive does this job. Note that the more recent hierarchical RCU [McK08a] implementation described in Sections D.2 and D.3 also adheres to “RCU Classic” semantics.

Finally, RCU may be used to provide type-safe memory [GC96], as described in Section 8.3.3.6. In the context of RCU, type-safe memory guarantees that a given data element will not change type during any RCU read-side critical section that accesses it. To make use of RCU-based type-safe memory, pass SLAB_DESTROY_BY_RCU to kmem_cache_create(). It is important to note that SLAB_DESTROY_BY_RCU will in no way prevent kmem_cache_alloc() from immediately re-allocating memory that was just now freed via kmem_cache_free(). In fact, the SLAB_DESTROY_BY_RCU-protected data structure just returned by rcu_dereference might be freed and reallocated an arbitrarily large number of times, even when under the protection of rcu_read_lock(). Instead, SLAB_DESTROY_BY_RCU operates by preventing kmem_cache_free() from returning a completely freed-up slab of data structures to the system until after an RCU grace period elapses. In short, although the data element might be freed and reallocated arbitrarily often, at least its type will remain the same.

**Quick Quiz 8.29:** How do you prevent a huge number of RCU read-side critical sections from indefinitely blocking a synchronize_rcu() invocation? ■

**Quick Quiz 8.30:** The synchronize_rcu() API waits for all pre-existing interrupt handlers to complete, right? ■

In the “RCU BH” column, rcu_read_lock_bh() and rcu_read_unlock_bh() delimit RCU read-side critical sections, and call_rcu_bh() invokes the specified function and argument after a subsequent grace period. Note that RCU BH does not have a synchronous synchronize_rcu_bh() interface, though one could easily be added if required.

**Quick Quiz 8.31:** What happens if you mix and match? For example, suppose you use rcu_read_lock() and rcu_read_unlock() to delimit RCU read-side critical sections, but then use call_rcu_bh() to post an RCU callback? ■

**Quick Quiz 8.32:** Hardware interrupt handlers can be thought of as being under the protection of an implicit rcu_read_lock_bh(), right? ■

In the “RCU Sched” column, anything that disables preemption acts as an RCU read-side critical section, and synchronize_sched() waits for the corresponding RCU grace period. This RCU API family was added in the 2.6.12 kernel, which split the old synchronize_kernel() API into the current synchronize_rcu() (for RCU Classic) and synchronize_sched() (for RCU Sched). Note that RCU Sched did not originally have an asynchronous call_rcu_sched() interface, but one was added in 2.6.26. In accordance with the quasi-minimalist philosophy of the Linux community, APIs are added on an as-needed basis.

**Quick Quiz 8.33:** What happens if you mix and match RCU Classic and RCU Sched? ■

**Quick Quiz 8.34:** In general, you cannot rely on synchronize_sched() to wait for all pre-existing interrupt handlers, right? ■

The “Realtime RCU” column has the same API as does RCU Classic, the only difference being that RCU read-side critical sections may be preempted and may block while acquiring spinlocks. The design of Realtime RCU is described elsewhere [McK07a].

**Quick Quiz 8.35:** Why do both SRCU and QRCU lack asynchronous call_srcu() or call_qrcu() interfaces? ■

The “SRCU” column in Table 8.5 displays a specialized RCU API that permits general sleeping in RCU read-side critical sections (see Appendix D.1 for more details). Of course, use of synchronize_srcu() in an SRCU read-side critical section may result in self-deadlock, so should be avoided. SRCU differs from earlier RCU implementations in that the caller allocates an srcu_struct for each distinct SRCU usage. This approach prevents SRCU read-side critical sections from blocking unrelated synchronize_srcu() invocations. In addition, in this variant of RCU, srcu_read_lock() returns a value that must be passed into the corresponding srcu_read_unlock().

The “QRCU” column presents an RCU implementation with the same API structure as SRCU, but optimized for extremely low-latency grace periods in absence of readers, as described elsewhere [McK07f]. As with SRCU, use of
8.3. READ-COPY UPDATE (RCU)

synchronize_qrcu() in a QRCU read-side critical section can result in self-deadlock, so should be avoided. Although QRCU has not yet been accepted into the Linux kernel, it is worth mentioning given that it is the only kernel-level RCU implementation that can boast deep sub-microsecond grace-period latencies.

Quick Quiz 8.36: Under what conditions can synchronize_srcu() be safely used within an SRCU read-side critical section? ■

The Linux kernel currently has a surprising number of RCU APIs and implementations. There is some hope of reducing this number, evidenced by the fact that a given build of the Linux kernel currently has at most three implementations behind four APIs (given that RCU Classic and Realtime RCU share the same API). However, careful inspection and analysis will be required, just as would be required in order to eliminate one of the many locking APIs.

The various RCU APIs are distinguished by the forward-progress guarantees that their RCU read-side critical sections must provide, and also by their scope, as follows:

1. RCU BH: read-side critical sections must guarantee forward progress against everything except for NMI and irq handlers, but not including softirq handlers. RCU BH is global in scope.

2. RCU Sched: read-side critical sections must guarantee forward progress against everything except for NMI and irq handlers, including softirq handlers. RCU Sched is global in scope.

3. RCU (both classic and real-time): read-side critical sections must guarantee forward progress against everything except for NMI handlers, irq handlers, softirq handlers, and (in the real-time case) higher-priority real-time tasks. RCU is global in scope.

4. SRCU and QRCU: read-side critical sections need not guarantee forward progress unless some other task is waiting for the corresponding grace period to complete, in which case these read-side critical sections should complete in no more than a few seconds (and preferably much more quickly). SRCU’s and QRCU’s scope is defined by the use of the corresponding srcu_struct or qrcu_struct, respectively.

In other words, SRCU and QRCU compensate for their extremely weak forward-progress guarantees by permitting the developer to restrict their scope.

8.3.4.2 RCU has Publish-Subscribe and Version-Maintenance APIs

Fortunately, the RCU publish-subscribe and version-maintenance primitives shown in the following table apply to all of the variants of RCU discussed above. This commonality can in some cases allow more code to be shared, which certainly reduces the API proliferation that would otherwise occur. The original purpose of the RCU publish-subscribe APIs was to bury memory barriers into these APIs, so that Linux kernel programmers could use RCU without needing to become expert on the memory-ordering models of each of the 20+ CPU families that Linux supports [Spr01].

The first pair of categories operate on Linux struct list_head lists, which are circular, doubly-linked lists. The list_for_each_entry_rcu() primitive traverses an RCU-protected list in a type-safe manner, while also enforcing memory ordering for situations where a new list element is inserted into the list concurrently with traversal. On non-Alpha platforms, this primitive incurs little or no performance penalty compared to list_for_each_entry(). The list_add_rcu(), list_add_tail_rcu(), and list_replace_rcu() primitives are analogous to their non-RCU counterparts, but incur the overhead of an additional memory barrier on weakly-ordered machines. The list_del_rcu() primitive is also analogous to its non-RCU counterpart, but oddly enough is very slightly faster due to the fact that it poisons only the prev pointer rather than both the prev and next pointers as list_del() must do. Finally, the list_splice_init_rcu() primitive is similar to its non-RCU counterpart, but incurs a full grace-period latency. The purpose of this grace period is to allow RCU readers to finish their traversal of the source list before completely disconnecting it from the list header – failure to do this could prevent such readers from ever terminating their traversal.

Quick Quiz 8.37: Why doesn’t list_del_rcu() poison both the next and prev pointers? ■

The second pair of categories operate on Linux’s struct hlist_head, which is a linear linked list. One advantage of struct hlist_head over struct list_head is that the former requires only a single-pointer list header, which can save significant memory in large hash tables. The struct hlist_head

5 Thanks to James Bottomley for urging me to this formulation, as opposed to simply saying that there are no forward-progress guarantees.
<table>
<thead>
<tr>
<th>Category</th>
<th>Primitives</th>
<th>Availability</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>List traversal</td>
<td>list_for_each_entry_rcu()</td>
<td>2.5.59</td>
<td>Simple instructions (memory barrier on Alpha)</td>
</tr>
<tr>
<td></td>
<td>list_add_rcu()</td>
<td>2.5.44</td>
<td>Memory barrier</td>
</tr>
<tr>
<td></td>
<td>list_add_tail_rcu()</td>
<td>2.5.44</td>
<td>Memory barrier</td>
</tr>
<tr>
<td></td>
<td>list_del_rcu()</td>
<td>2.5.44</td>
<td>Simple instructions</td>
</tr>
<tr>
<td></td>
<td>list_replace_rcu()</td>
<td>2.6.9</td>
<td>Memory barrier</td>
</tr>
<tr>
<td></td>
<td>list_splice_init_rcu()</td>
<td>2.6.21</td>
<td>Grace-period latency</td>
</tr>
<tr>
<td>List update</td>
<td>list_add_tail_rcu()</td>
<td>2.5.44</td>
<td>Memory barrier</td>
</tr>
<tr>
<td></td>
<td>list_del_rcu()</td>
<td>2.5.44</td>
<td>Memory barrier</td>
</tr>
<tr>
<td></td>
<td>list_replace_rcu()</td>
<td>2.6.9</td>
<td>Memory barrier</td>
</tr>
<tr>
<td></td>
<td>list_splice_init_rcu()</td>
<td>2.6.21</td>
<td>Grace-period latency</td>
</tr>
<tr>
<td>Hlist traversal</td>
<td>hlist_for_each_entry_rcu()</td>
<td>2.6.8</td>
<td>Simple instructions (memory barrier on Alpha)</td>
</tr>
<tr>
<td></td>
<td>hlist_add_after_rcu()</td>
<td>2.6.14</td>
<td>Memory barrier</td>
</tr>
<tr>
<td></td>
<td>hlist_add_before_rcu()</td>
<td>2.6.14</td>
<td>Memory barrier</td>
</tr>
<tr>
<td></td>
<td>hlist_add_head_rcu()</td>
<td>2.5.64</td>
<td>Memory barrier</td>
</tr>
<tr>
<td></td>
<td>hlist_del_rcu()</td>
<td>2.5.64</td>
<td>Simple instructions</td>
</tr>
<tr>
<td></td>
<td>hlist_replace_rcu()</td>
<td>2.6.15</td>
<td>Memory barrier</td>
</tr>
<tr>
<td>Pointer traversal</td>
<td>rcu_dereference()</td>
<td>2.6.9</td>
<td>Simple instructions (memory barrier on Alpha)</td>
</tr>
<tr>
<td>Pointer update</td>
<td>rcu_assign_pointer()</td>
<td>2.6.10</td>
<td>Memory barrier</td>
</tr>
</tbody>
</table>

Table 8.6: RCU Publish-Subscribe and Version Maintenance APIs

The final pair of categories operate directly on pointers, and are useful for creating RCU-protected non-list data structures, such as RCU-protected arrays and trees. The `rcu_assign_pointer()` primitive ensures that any prior initialization remains ordered before the assignment to the pointer on weakly ordered machines. Similarly, the `rcu_dereference()` primitive ensures that subsequent code dereferencing the pointer will see the effects of initialization code prior to the corresponding `rcu_assign_pointer()` on Alpha CPUs. On non-Alpha CPUs, `rcu_dereference()` documents which pointer dereferences are protected by RCU.

**Quick Quiz 8.38:** Normally, any pointer subject to `rcu_dereference()` must always be updated using `rcu_assign_pointer()`. What is an exception to this rule? ■

**Quick Quiz 8.39:** Are there any downsides to the fact that these traversal and update primitives can be used with any of the RCU API family members? ■

8.3.4.3 Where Can RCU’s APIs Be Used?

Figure 8.32 shows which APIs may be used in which in-kernel environments. The RCU read-side primitives may be used in any environment, including NMI, the RCU mutation and asynchronous grace-period primitives may be used in any environment other than NMI, and, fi-
nally, the RCU synchronous grace-period primitives may be used only in process context. The RCU list-traversal primitives include `list_for_each_entry_rcu()`, `hlist_for_each_entry_rcu()`, etc. Similarly, the RCU list-mutation primitives include `list_add_rcu()`, `hlist_del_rcu()`, etc.

Note that primitives from other families of RCU may be substituted, for example, `srcu_read_lock()` may be used in any context in which `rcu_read_lock()` may be used.

8.3.4.4 So, What is RCU Really?

At its core, RCU is nothing more nor less than an API that supports publication and subscription for insertions, waiting for all RCU readers to complete, and maintenance of multiple versions. That said, it is possible to build higher-level constructs on top of RCU, including the reader-writer-locking, reference-counting, and existence-guarantee constructs listed in the companion article. Furthermore, I have no doubt that the Linux community will continue to find interesting new uses for RCU, just as they do for any of a number of synchronization primitives throughout the kernel.

Of course, a more-complete view of RCU would also include all of the things you can do with these APIs.

However, for many people, a complete view of RCU must include sample RCU implementations. The next section therefore presents a series of “toy” RCU implementations of increasing complexity and capability.

8.3.5 “Toy” RCU Implementations

The toy RCU implementations in this section are designed not for high performance, practicality, or any kind of production use, but rather for clarity. Nevertheless, you will need a thorough understanding of Chapters 1, 2, 3, 5, and 8 for even these toy RCU implementations to be easily understandable.

This section provides a series of RCU implementations in order of increasing sophistication, from the viewpoint of solving the existence-guarantee problem. Section 8.3.5.1 presents a rudimentary RCU implementation based on simple locking, while Section 8.3.5.3 through 8.3.5.9 present a series of simple RCU implementations based on locking, reference counters, and free-running counters. Finally, Section 8.3.5.10 provides a summary and a list of desirable RCU properties.

8.3.5.1 Lock-Based RCU

Perhaps the simplest RCU implementation leverages locking, as shown in Figure 8.33 (`rcu_lock.h` and `rcu_lock.c`). In this implementation, `rcu_read_lock()` acquires a global spinlock, `rcu_read_unlock()` releases it, and `synchronize_rcu()` acquires it then immediately releases it.

Because `synchronize_rcu()` does not return until it has acquired (and released) the lock, it cannot return until all prior RCU read-side critical sections have completed, thus faithfully implementing RCU semantics. Of course, only one RCU reader may be in its read-side critical section at a time, which almost entirely defeats the purpose of RCU. In addition, the lock operations in `rcu_read_lock()` and `rcu_read_unlock()` are extremely heavyweight, with read-side overhead ranging from about 100 nanoseconds on a single Power5 CPU up to more than 17 microseconds on a 64-CPU system. Worse yet, these same lock operations permit `rcu_read_lock()` to participate in deadlock cycles. Furthermore, in absence of recursive locks, RCU read-side critical sections cannot be nested, and, finally, although concurrent RCU updates could in principle be satisfied by a common grace period, this implementation serializes grace periods, preventing grace-period sharing.

Quick Quiz 8.40: Why wouldn’t any deadlock in the RCU implementation in Figure 8.33 also be a deadlock in any other RCU implementation? ■

Quick Quiz 8.41: Why not simply use reader-writer locks in the RCU implementation in Figure 8.33 in order to allow RCU readers to proceed in parallel? ■

It is hard to imagine this implementation being useful in a production setting, though it does have the virtue of being implementable in almost any user-level application.

6 However, production-quality user-level RCU implementations are available [Des09].

---

Figure 8.33: Lock-Based RCU Implementation
Furthermore, similar implementations having one lock per CPU or using reader-writer locks have been used in production in the 2.4 Linux kernel.

A modified version of this one-lock-per-CPU approach, but instead using one lock per thread, is described in the next section.

### 8.3.5.2 Per-Thread Lock-Based RCU

Figure 8.34 (rcu_lock_percpu.h and rcu_lock_percpu.c) shows an implementation based on one lock per thread. The rcu_read_lock() and rcu_read_unlock() functions acquire and release, respectively, the current thread’s lock. The synchronize_rcu() function acquires and releases each thread’s lock in turn. Therefore, all RCU read-side critical sections running when synchronize_rcu() starts must have completed before synchronize_rcu() can return.

This implementation does have the virtue of permitting concurrent RCU readers, and does avoid the deadlock condition that can arise with a single global lock. Furthermore, the read-side overhead, though high at roughly 140 nanoseconds, remains at about 140 nanoseconds regardless of the number of CPUs. However, the update-side overhead ranges from about 600 nanoseconds on a single Power5 CPU up to more than 100 microseconds on 64 CPUs.

**Quick Quiz 8.42:** Wouldn’t it be cleaner to acquire all the locks, and then release them all in the loop from lines 15-18 of Figure 8.34? After all, with this change, there would be a point in time when there were no readers, simplifying things greatly.

**Quick Quiz 8.43:** Is the implementation shown in Figure 8.34 free from deadlocks? Why or why not?

**Quick Quiz 8.44:** Isn’t one advantage of the RCU algorithm shown in Figure 8.34 that it uses only primitives that are widely available, for example, in POSIX pthreads?

This approach could be useful in some situations, given that a similar approach was used in the Linux 2.4 kernel [MM00].

The counter-based RCU implementation described next overcomes some of the shortcomings of the lock-based implementation.

### 8.3.5.3 Simple Counter-Based RCU

A slightly more sophisticated RCU implementation is shown in Figure 8.35 (rcu_rcg.h and rcu_rcg.c). This implementation makes use of a global reference

```
1 atomic_t rcu_refcnt;
2
3 static void rcu_read_lock(void)
4 {
5    atomic_inc(&rcu_refcnt);
6    smp_mb();
7 }
8
9 static void rcu_read_unlock(void)
10 {
11    smp_mb();
12    atomic_dec(&rcu_refcnt);
13 }
14
15 void synchronize_rcu(void)
16 {
17    smp_mb();
18    while (atomic_read(&rcu_refcnt) != 0) {
19        poll(NULL, 0, 10);
20    }
21    smp_mb();
22 }
```

Figure 8.35: RCU Implementation Using Single Global Reference Counter
counter rcu_refcnt defined on line 1. The rcu_read_lock() primitive atomically increments this counter, then executes a memory barrier to ensure that the RCU read-side critical section is ordered after the atomic increment. Similarly, rcu_read_unlock() executes a memory barrier to confine the RCU read-side critical section, then atomically decrements the counter. The synchronize_rcu() primitive spins waiting for the reference counter to reach zero, surrounded by memory barriers. The poll() on line 19 merely provides pure delay, and from a pure RCU-semantics point of view could be omitted. Again, once synchronize_rcu() returns, all prior RCU read-side critical sections are guaranteed to have completed.

In happy contrast to the lock-based implementation shown in Section 8.3.5.1, this implementation allows parallel execution of RCU read-side critical sections. In happy contrast to the per-thread lock-based implementation shown in Section 8.3.5.2, it also allows them to be nested. In addition, the rcu_read_lock() primitive cannot possibly participate in deadlock cycles, as it never spins nor blocks.

Quick Quiz 8.45: But what if you hold a lock across a call to synchronize_rcu(), and then acquire that same lock within an RCU read-side critical section? ■

However, these implementations still have some serious shortcomings. First, the atomic operations in rcu_read_lock() and rcu_read_unlock() are still quite heavyweight, with read-side overhead ranging from about 100 nanoseconds on a single Power5 CPU up to almost 40 microseconds on a 64-CPU system. This means that the RCU read-side critical sections have to be extremely long in order to get any real read-side parallelism. On the other hand, in the absence of readers, grace periods elapse in about 40 nanoseconds, many orders of magnitude faster than production-quality implementations in the Linux kernel.

Quick Quiz 8.46: How can the grace period possibly elapse in 40 nanoseconds when synchronize_rcu() contains a 10-millisecond delay? ■

Second, if there are many concurrent rcu_read_lock() and rcu_read_unlock() operations, there will be extreme memory contention on rcu_refcnt, resulting in expensive cache misses. Both of these first two shortcomings largely defeat a major purpose of RCU, namely to provide low-overhead read-side synchronization primitives.

Finally, a large number of RCU readers with long read-side critical sections could prevent synchronize_rcu() from ever completing, as the global counter might never reach zero. This could result in starvation of RCU updates, which is of course unacceptable in production settings.

Quick Quiz 8.47: Why not simply make rcu_read_lock() wait when a concurrent synchronize_rcu() has been waiting too long in the RCU implementation in Figure 8.35? Wouldn’t that prevent synchronize_rcu() from starving? ■

Therefore, it is still hard to imagine this implementation being useful in a production setting, though it has a bit more potential than the lock-based mechanism, for example, as an RCU implementation suitable for a high-stress debugging environment. The next section describes a variation on the reference-counting scheme that is more favorable to writers.

```c
1 static void rcu_read_lock(void) {
2     int i;
3     int n;
4
5     n = __get_thread_var(rcu_nesting);
6     if (n == 0) {
7         i = atomic_read(&rcu_idx);
8         __get_thread_var(rcu_read_idx) = i;
9         atomic_inc(&rcu_refcnt[i]);
10         __get_thread_var(rcu_nesting) = n + 1;
11         smp_mb();
12     }
13     __get_thread_var(rcu_nesting) = n - 1;
14     __get_thread_var(rcu_read_idx) = i;
15     smp_mb();
16 }
```

Figure 8.37: RCU Read-Side Using Global Reference-Count Pair

```c
1 static void rcu_read_unlock(void) {
2     int i;
3     int n;
4
5     smp_mb();
6     n = __get_thread_var(rcu_nesting);
7     if (n == 1) {
8         i = __get_thread_var(rcu_read_idx);
9         atomic_dec(&rcu_refcnt[i]);
10         __get_thread_var(rcu_nesting) = n - 1;
11         __get_thread_var(rcu_read_idx) = i;
12     }
```

Figure 8.36: RCU Global Reference-Count Pair Data
8.3.5.4 Starvation-Free Counter-Based RCU

Figure 8.37 (rcu_rcgp.h) shows the read-side primitives of an RCU implementation that uses a pair of reference counters (rcu_refcnt[]), along with a global index that selects one counter out of the pair (rcu_idx), a per-thread nesting counter rcu_nesting, a per-thread snapshot of the global index (rcu_read_idx), and a global lock (rcu_gp_lock), which are themselves shown in Figure 8.36.

The rcu_read_lock() primitive atomically increments the member of the rcu_refcnt[] pair indexed by rcu_idx, and keeps a snapshot of this index in the per-thread variable rcu_read_idx. The rcu_read_unlock() primitive then atomically decrements whichever counter of the pair that the corresponding rcu_read_lock() incremented. However, because only one value of rcu_idx is remembered per thread, additional measures must be taken to permit nesting. These additional measures use the per-thread rcu_nesting variable to track nesting.

To make all this work, line 6 of rcu_read_lock() in Figure 8.37 picks up the current thread’s instance of rcu_nesting, and if line 7 finds that this is the outermost rcu_read_lock(), then lines 8-10 pick up the current value of rcu_idx, save it in this thread’s instance of rcu_read_idx, and atomically increment the selected element of rcu_refcnt. Regardless of the value of rcu_nesting, line 12 increments it. Line 13 executes a memory barrier to ensure that the RCU read-side critical section does not bleed out before the rcu_read_lock() code.

Similarly, the rcu_read_unlock() function executes a memory barrier at line 21 to ensure that the RCU read-side critical section does not bleed out after the rcu_read_unlock() code. Line 22 picks up this thread’s instance of rcu_nesting, and if line 23 finds that this is the outermost rcu_read_unlock(), then lines 24 and 25 pick up this thread’s instance of rcu_read_idx (saved by the outermost rcu_read_lock()) and atomically decrements the selected element of rcu_refcnt. Regardless of the nesting level, line 27 decrements this thread’s instance of rcu_nesting.

Figure 8.38 (rcu_rcpg.c) shows the corresponding synchronize_rcu() implementation. Lines 6 and 19 acquire and release rcu_gp_lock in order to prevent more than one concurrent instance of synchronize_rcu(). Lines 7-8 pick up the value of rcu_idx and complement it, respectively, so that subsequent instances of rcu_read_lock() will use a different element of

```c
1 void synchronize_rcu(void) {
2    int i;
3    smp_mb();
4    spin_lock(&rcu_gp_lock);
5    i = atomic_read(&rcu_idx);
6    atomic_set(&rcu_idx, !i);
7    smp_mb();
8    while (atomic_read(&rcu_refcnt[i]) != 0) {  // Quick Quiz 8.48
9        poll(NULL, 0, 10);
10       smp_mb();
11      }  // Quick Quiz 8.49
12     atomic_set(&rcu_idx, !i);
13     smp_mb();
14     while (atomic_read(&rcu_refcnt[!i]) != 0) {  // Quick Quiz 8.49
15         poll(NULL, 0, 10);
16          smp_mb();
17      }  // Quick Quiz 8.49
18     spin_unlock(&rcu_gp_lock);
19    smp_mb();
20   }
21 }
```

Figure 8.38: RCU Update Using Global Reference-Count Pair

rcu_idx that did preceding instances. Lines 10-12 then wait for the prior element of rcu_idx to reach zero, with the memory barrier on line 9 ensuring that the check of rcu_idx is not reordered to precede the complementing of rcu_idx. Lines 13-18 repeat this process, and line 20 ensures that any subsequent reclamation operations are not reordered to precede the checking of rcu_refcnt.

Quick Quiz 8.48: Why the memory barrier on line 5 of synchronize_rcu() in Figure 8.38 given that there is a spin-lock acquisition immediately after? ■

Quick Quiz 8.49: Why is the counter flipped twice in Figure 8.38? Shouldn’t a single flip-and-wait cycle be sufficient? ■

This implementation avoids the update-starvation issues that could occur in the single-counter implementation shown in Figure 8.35.

There are still some serious shortcomings. First, the atomic operations in rcu_read_lock() and rcu_read_unlock() are still quite heavyweight. In fact, they are more complex than those of the single-counter variant shown in Figure 8.35, with the read-side primitives consuming about 150 nanoseconds on a single Power5 CPU and almost 40 microseconds on a 64-CPU system. The updates-side synchronize_rcu() primitive is more costly as well, ranging from about 200 nanoseconds on a single Power5 CPU to more than 40 microseconds on a 64-CPU system. This means that the RCU read-side critical sections have to be extremely long in order to get any real read-side parallelism.

Second, if there are many concurrent rcu_read_
8.3. READ-COPY UPDATE (RCU)

8.3. READ-COPY UPDATE (RCU)

1 DEFINE_SPINLOCK(rcu_gp_lock);
2 DEFINE_PER_THREAD(int [2], rcu_refcnt);
3 atomic_t rcu_idx;
4 DEFINE_PER_THREAD(int, rcu_nesting);
5 DEFINE_PER_THREAD(int, rcu_read_idx);

Figure 8.39: RCU Per-Thread Reference-Count Pair Data

lock() and rcu_read_unlock() operations, there
will be extreme memory contention on the rcu_refcnt
elements, resulting in expensive cache misses. This fur-
ther extends the RCU read-side critical-section duration
required to provide parallel read-side access. These first
two shortcomings defeat the purpose of RCU in most
situations.

Third, the need to flip rcu_idx twice imposes sub-
stantial overhead on updates, especially if there are large
numbers of threads.

Finally, despite the fact that concurrent RCU updates
could in principle be satisfied by a common grace period,
this implementation serializes grace periods, preventing
grace-period sharing.

Quick Quiz 8.50: Given that atomic increment and
decrement are so expensive, why not just use non-atomic
increment on line 10 and a non-atomic decrement on
line 25 of Figure 8.37?

Despite these shortcomings, one could imagine this
variant of RCU being used on small tightly coupled multi-
processors, perhaps as a memory-conserving implementa-
tion that maintains API compatibility with more complex
implementations. However, it would not not likely scale
well beyond a few CPUs.

The next section describes yet another variation on the
reference-counting scheme that provides greatly improved
read-side performance and scalability.

8.3.5.5 Scalable Counter-Based RCU

Figure 8.40 (rcu_rcpl.h) shows the read-side primi-
tives of an RCU implementation that uses per-thread
pairs of reference counters. This implementation is quite
similar to that shown in Figure 8.37, the only difference
being that rcu_refcnt is now a per-thread variable (as
shown in Figure 8.39), so the rcu_read_lock() and
rcu_read_unlock() primitives no longer perform
atomic operations.

Quick Quiz 8.51: Come off it! We can see the
atomic_read() primitive in rcu_read_-
lock()!!! So why are you trying to pretend that rcu_-
read_lock() contains no atomic operations???

Figure 8.41: RCU Update Using Per-Thread Reference-
Count Pair
CHAPTER 8. DEFERRED PROCESSING

Figure 8.41 (rcu_rcpl.c) shows the implementation of synchronize_rcu(), along with a helper function named flip_counter_and_wait(). The synchronize_rcu() function resembles that shown in Figure 8.38, except that the repeated counter flip is replaced by a pair of calls on lines 22 and 23 to the new helper function.

The new flip_counter_and_wait() function updates the rcu_idx variable on line 5, executes a memory barrier on line 6, then lines 7-11 spin on each thread’s prior rcu_refcnt element, waiting for it to go to zero. Once all such elements have gone to zero, it executes another memory barrier on line 12 and returns.

This RCU implementation imposes important new requirements on its software environment, namely, (1) that it be possible to declare per-thread variables, (2) that these per-thread variables be accessible from other threads, and (3) that it is possible to enumerate all threads. These requirements can be met in almost all software environments, but often result in fixed upper bounds on the number of threads. More-complex implementations might avoid such bounds, for example, by using expandable hash tables. Such implementations might dynamically track threads, for example, by adding them on their first call to rcu_read_lock().

Quick Quiz 8.52: Great, if we have \( N \) threads, we can have \( 2^N \) ten-millisecond waits (one set per flip_counter_and_wait() invocation, and even that assumes that we wait only once for each thread. Don’t we need the grace period to complete *much* more quickly?

This implementation still has several shortcomings. First, the need to flip rcu_idx twice imposes substantial overhead on updates, especially if there are large numbers of threads.

Second, synchronize_rcu() must now examine a number of variables that increases linearly with the number of threads, imposing substantial overhead on applications with large numbers of threads.

Third, as before, although concurrent RCU updates could in principle be satisfied by a common grace period, this implementation serializes grace periods, preventing grace-period sharing.

Finally, as noted in the text, the need for per-thread variables and for enumerating threads may be problematic in some software environments.

That said, the read-side primitives scale very nicely, requiring about 115 nanoseconds regardless of whether running on a single-CPU or a 64-CPU Power5 system. As noted above, the synchronize_rcu() primitive does not scale, ranging in overhead from almost a microsecond on a single Power5 CPU up to almost 200 microseconds on a 64-CPU system. This implementation could conceivably form the basis for a production-quality user-level RCU implementation.

The next section describes an algorithm permitting more efficient concurrent RCU updates.

8.3.5.6 Scalable Counter-Based RCU With Shared Grace Periods

Figure 8.43 (rcu_rcpls.h) shows the read-side primitives for an RCU implementation using per-thread reference-count pairs, as before, but permitting updates to share grace periods. The main difference from the earlier implementation shown in Figure 8.40 is that rcu_idx is now a long that counts freely, so that line 8 of Figure 8.43 must mask off the low-order bit. We also switched from
8.3. READ-COPY UPDATE (RCU)

static void flip_counter_and_wait(int ctr)
{
    int i;
    int t;
    ACCESS_ONCE(rcu_idx) = ctr + 1;
    i = ctr & 0x1;
    smp_mb();
    for_each_thread(t) {
        while (per_thread(rcu_refcnt, t)[i] != 0) {
            poll(NULL, 0, 10);
        }
    }
    smp_mb();
}

void synchronize_rcu(void)
{
    int ctr;
    int oldctr;
    smp_mb();
    oldctr = ACCESS_ONCE(rcu_idx);
    spin_lock(&rcu_gp_lock);
    ctr = ACCESS_ONCE(rcu_idx);
    if (ctr - oldctr >= 3) {
        spin_unlock(&rcu_gp_lock);
        smp_mb();
        return;
    }
    flip_counter_and_wait(ctr);
    if (ctr - oldctr < 2)
        flip_counter_and_wait(ctr + 1);
    spin_unlock(&rcu_gp_lock);
    smp_mb();
}

Figure 8.44: RCU Shared Update Using Per-Thread Reference-Count Pair

using atomic_read() and atomic_set() to using ACCESS_ONCE(). The data is also quite similar, as shown in Figure 8.42, with rcu_idx now being a lock instead of an atomic_t.

Figure 8.44 (rcu_rclps.c) shows the implementation of synchronize_rcu() and its helper function flip_counter_and_wait(). These are similar to those in Figure 8.41. The differences in flip_counter_and_wait() include:

1. Line 6 uses ACCESS_ONCE() instead of atomic_set(), and increments rather than complementing.
2. A new line 7 masks the counter down to its bottom bit.

The changes to synchronize_rcu() are more pervasive:

1. There is a new oldctr local variable that captures the pre-lock-acquisition value of rcu_idx on line 23.
2. Line 26 uses ACCESS_ONCE() instead of atomic_read().
3. Lines 27-30 check to see if at least three counter flips were performed by other threads while the lock was being acquired, and, if so, releases the lock, does a memory barrier, and returns. In this case, there were two full waits for the counters to go to zero, so those other threads already did all the required work.
4. At lines 33-34, flip_counter_and_wait() is only invoked a second time if there were fewer than two counter flips while the lock was being acquired. On the other hand, if there were two counter flips, some other thread did one full wait for all the counters to go to zero, so only one more is required.

With this approach, if an arbitrarily large number of threads invoke synchronize_rcu() concurrently, with one CPU for each thread, there will be a total of only three waits for counters to go to zero.

Despite the improvements, this implementation of RCU still has a few shortcomings. First, as before, the need to flip rcu_idx twice imposes substantial overhead on updates, especially if there are large numbers of threads.

Second, each updater still acquires rcu_gp_lock, even if there is no work to be done. This can result in a severe scalability limitation if there are large numbers of concurrent updates. Section D.4 shows one way to avoid this in a production-quality real-time implementation of RCU for the Linux kernel.

Third, this implementation requires per-thread variables and the ability to enumerate threads, which again can be problematic in some software environments.

Finally, on 32-bit machines, a given update thread might be preempted long enough for the rcu_idx counter to overflow. This could cause such a thread to force an unnecessary pair of counter flips. However, even if each grace period took only one microsecond, the offending thread would need to be preempted for more than an hour, in which case an extra pair of counter flips is likely the least of your worries.

As with the implementation described in Section 8.3.5.3, the read-side primitives scale extremely well, incurring roughly 115 nanoseconds of overhead regardless of the number of CPUs. The synchronize_rcu() primitives is still expensive, ranging from about one microsecond up to about 16 microseconds. This is
CHAPTER 8. DEFERRED PROCESSING

1. DEFINE_SPINLOCK(rcu_gp_lock);
2. long rcu_gp_ctr = 0;
3. DEFINE_PER_THREAD(long, rcu_reader_gp);
4. DEFINE_PER_THREAD(long, rcu_reader_gp_snap);

Figure 8.45: Data for Free-Running Counter Using RCU

nevertheless much cheaper than the roughly 200 microseconds incurred by the implementation in Section 8.3.5.5. So, despite its shortcomings, one could imagine this RCU implementation being used in production in real-life applications.

Quick Quiz 8.53: All of these toy RCU implementations have either atomic operations in rcu_read_lock() and rcu_read_unlock(), or synchronize_rcu() overhead that increases linearly with the number of threads. Under what circumstances could an RCU implementation enjoy light-weight implementations for all three of these primitives, all having deterministic $O(1)$ overheads and latencies?

Referring back to Figure 8.43, we see that there is one global-variable access and no fewer than four accesses to thread-local variables. Given the relatively high cost of thread-local accesses on systems implementing POSIX threads, it is tempting to collapse the three thread-local variables into a single structure, permitting rcu_read_lock() and rcu_read_unlock() to access their thread-local data with a single thread-local-storage access. However, an even better approach would be to reduce the number of thread-local accesses to one, as is done in the next section.

8.3.5.7 RCU Based on Free-Running Counter

Figure 8.46 (rcu.h and rcu.c) show an RCU implementation based on a single global free-running counter that takes on only even-numbered values, with data shown in Figure 8.45. The resulting rcu_read_lock() implementation is extremely straightforward. Line 3 simply adds one to the global free-running rcu_gp_ctr variable and stores the resulting odd-numbered value into the rcu_reader_gp per-thread variable. Line 4 executes a memory barrier to prevent the content of the subsequent RCU read-side critical section from “leaking out”.

The rcu_read_unlock() implementation is similar. Line 9 executes a memory barrier, again to prevent the prior RCU read-side critical section from “leaking out”. Line 10 then copies the rcu_gp_ctr global variable to the rcu_reader_gp per-thread variable, leaving this per-thread variable with an even-numbered value so that a concurrent instance of synchronize_rcu() will know to ignore it.

Quick Quiz 8.54: If any even value is sufficient to tell synchronize_rcu() to ignore a given task, why doesn’t line 10 of Figure 8.46 simply assign zero to rcu_reader_gp?

Thus, synchronize_rcu() could wait for all of the per-thread rcu_reader_gp variables to take on even-numbered values. However, it is possible to do much better than that because synchronize_rcu() need only wait on pre-existing RCU read-side critical sections. Line 17 executes a memory barrier to prevent prior manipulations of RCU-protected data structures from being reordered (by either the CPU or the compiler) to follow the increment on line 17. Line 18 acquires the rcu_gp_lock (and line 28 releases it) in order to prevent multiple synchronize_rcu() instances from running concurrently. Line 19 then increments the global rcu_gp_ctr variable by two, so that all pre-existing RCU read-side critical sections will have corresponding per-thread rcu_reader_gp variables with values less than that of rcu_gp_ctr, modulo the machine’s word size. Recall also that threads with even-numbered values of rcu_reader_gp are not in an RCU read-side critical section,
so that lines 21-27 scan the rcu_reader_gp values until they all are either even (line 22) or are greater than the global rcu_gp_ctr (lines 23-24). Line 25 blocks for a short period of time to wait for a pre-existing RCU read-side critical section, but this can be replaced with a spin-loop if grace-period latency is of the essence. Finally, the memory barrier at line 29 ensures that any subsequent destruction will not be reordered into the preceding loop.

Quick Quiz 8.55: Why are the memory barriers on lines 17 and 29 of Figure 8.46 needed? Aren’t the memory barriers inherent in the locking primitives on lines 18 and 28 sufficient?

This approach achieves much better read-side performance, incurring roughly 63 nanoseconds of overhead regardless of the number of Power5 CPUs. Updates incur more overhead, ranging from about 500 nanoseconds on a single Power5 CPU to more than 100 microseconds on 64 such CPUs.

Quick Quiz 8.56: Couldn’t the update-side optimization described in Section 8.3.5.6 be applied to the implementation shown in Figure 8.46?

This implementation suffers from some serious shortcomings in addition to the high update-side overhead noted earlier. First, it is no longer permissible to nest RCU read-side critical sections, a topic that is taken up in the next section. Second, if a reader is preempted at line 3 of Figure 8.46 after fetching from rcu_gp_ctr but before storing to rcu_reader_gp, and if the rcu_gp_ctr counter then runs through more than half but less than all of its possible values, then synchronize_rcu() will ignore the subsequent RCU read-side critical section. Third and finally, this implementation requires that the enclosing software environment be able to enumerate threads and maintain per-thread variables.

Quick Quiz 8.57: Is the possibility of readers being preempted in line 3 of Figure 8.46 a real problem, in other words, is there a real sequence of events that could lead to failure? If not, why not? If so, what is the sequence of events, and how can the failure be addressed?

8.3.5.8 Nestable RCU Based on Free-Running Counter

Figure 8.48 (rcu_nest.h and rcu_nest.c) show an RCU implementation based on a single global free-running counter, but that permits nesting of RCU read-side critical sections. This nestability is accomplished by reserving the low-order bits of the global rcu_gp_ctr to count nesting, using the definitions shown in Figure 8.47. This is a generalization of the scheme

```c
1 define_spinlock(rcu_gp_lock);
2 define RCU_GP_CTR_SHIFT 7
3 define RCU_GP_CTR_BOTTOM_BIT (1 << RCU_GP_CTR_SHIFT)
4 define RCU_GP_CTR_NEST_MASK (RCU_GP_CTR BOTTOM_BIT - 1)
5 long rcu_gp_ctr = 0;
6 define_per_thread(long, rcu_reader_gp);
```

Figure 8.47: Data for Nestable RCU Using a Free-Running Counter

```c
1 static void rcu_read_lock(void)
2 { long tmp;
3 long *rrgp;
4 6 rrgp = &__get_thread_var(rcu_reader_gp);
5 tmp = *rrgp;
6 if ((tmp & RCU_GP_CTR_NEST_MASK) == 0)
7 tmp = rcu_gp_ctr;
8 tmp++;
9 *rrgp = tmp;
10 smp_mb();
13 }
14
15 static void rcu_read_unlock(void)
16 { long tmp;
17 smp_mb();
18 __get_thread_var(rcu_reader_gp)--;}
21 
22 void synchronize_rcu(void)
24 { int t;
25 smp_mb();
28 spin_lock(&rcu_gp_lock);
29 rcu_gp_ctr += RCU_GP_CTR BOTTOM_BIT;
30 smp_mb();
31 for_each_thread(t) {
32 while (rcu_gp_ongoing(t) &&
33 ((per_thread(rcu_reader_gp, t) -
34 rcu_gp_ctr) < 0)) {
35 poll(NULL, 0, 10);
36 }
37 } spin_unlock(&rcu_gp_lock);
39 smp_mb();
40 }
```

Figure 8.48: Nestable RCU Using a Free-Running Counter
in Section 8.3.5.7, which can be thought of as having a single low-order bit reserved for counting nesting depth. Two C-preprocessor macros are used to arrange this, RCU_GP_CTR_NEST_MASK and RCU_GP_CTR_BOTTOM_BIT. These are related: RCU_GP_CTR_NEST_MASK=RCU_GP_CTR_BOTTOM_BIT-1. The RCU_GP_CTR_BOTTOM_BIT macro contains a single bit that is positioned just above the bits reserved for counting nesting, and the RCU_GP_CTR_NEST_MASK has all one bits covering the region of rcu_gp_ctr used to count nesting. Obviously, these two C-preprocessor macros must reserve enough of the low-order bits of the counter to permit the maximum required nesting of RCU read-side critical sections, and this implementation reserves seven bits, for a maximum RCU read-side critical-section nesting depth of 127, which should be well in excess of that needed by most applications.

The resulting rcu_read_lock() implementation is still reasonably straightforward. Line 6 places a pointer to this thread’s instance of rcu_reader_gp into the local variable rrgp, minimizing the number of expensive calls to the pthreads thread-local-state API. Line 7 records the current value of rcu_reader_gp into another local variable tmp, and line 8 checks to see if the low-order bits are zero, which would indicate that this is the outermost rcu_read_lock(). If so, line 9 places the global rcu_gp_ctr into tmp because the current value previously fetched by line 7 is likely to be obsolete. In either case, line 10 increments the nesting depth, which you will recall is stored in the seven low-order bits of the counter. Line 11 stores the updated counter back into this thread’s instance of rcu_reader_gp, and, finally, line 12 executes a memory barrier to prevent the RCU read-side critical section from bleeding out into the code preceding the call to rcu_read_lock().

In other words, this implementation of rcu_read_lock() picks up a copy of the global rcu_gp_ctr unless the current invocation of rcu_read_lock() is nested within an RCU read-side critical section, in which case it instead fetches the contents of the current thread’s instance of rcu_reader_gp. Either way, it increments whatever value it fetched in order to record an additional nesting level, and stores the result in the current thread’s instance of rcu_reader_gp.

Interestingly enough, the implementation of rcu_read_unlock() is identical to that shown in Section 8.3.5.7. Line 19 executes a memory barrier in order to prevent the RCU read-side critical section from bleeding out into code following the call to rcu_read_unlock(), and line 20 decrements this thread’s instance of rcu_reader_gp, which has the effect of decrementing the nesting count contained in rcu_reader_gp’s low-order bits. Debugging versions of this primitive would check (before decrementing!) that these low-order bits were non-zero.

The implementation of synchronize_rcu() is quite similar to that shown in Section 8.3.5.7. There are two differences. The first is that line 29 adds RCU_GP_CTR_BOTTOM_BIT to the global rcu_gp_ctr instead of adding the constant “2”, and the second is that the comparison on line 32 has been abstracted out to a separate function, where it checks the bit indicated by RCU_GP_CTR_BOTTOM_BIT instead of unconditionally checking the low-order bit.

This approach achieves read-side performance almost equal to that shown in Section 8.3.5.7, incurring roughly 65 nanoseconds of overhead regardless of the number of Power5 CPUs. Updates again incur more overhead, ranging from about 600 nanoseconds on a single Power5 CPU to more than 100 microseconds on 64 such CPUs.

Quick Quiz 8.58: Why not simply maintain a separate per-thread nesting-level variable, as was done in previous section, rather than having all this complicated bit manipulation?

This implementation suffers from the same shortcomings as does that of Section 8.3.5.7, except that nesting of RCU read-side critical sections is now permitted. In addition, on 32-bit systems, this approach shortens the time required to overflow the global rcu_gp_ctr variable. The following section shows one way to greatly increase the time required for overflow to occur, while greatly reducing read-side overhead.

Quick Quiz 8.59: Given the algorithm shown in Figure 8.48, how could you double the time required to overflow the global rcu_gp_ctr?

Quick Quiz 8.60: Again, given the algorithm shown in Figure 8.48, is counter overflow fatal? Why or why not? If it is fatal, what can be done to fix it?
8.3. READ-COPY UPDATE (RCU)

1 static void rcu_read_lock(void)
2 {
3 }
4
5 static void rcu_read_unlock(void)
6 {
7 }
8
9 rcu_quiescent_state(void)
10 {
11 smp_mb();
12 __get_thread_var(rcu_reader_qs_gp) = ACCESS_ONCE(rcu_gp_ctr) + 1;
13 smp_mb();
14 }
15
16 static void rcu_thread_offline(void)
17 {
18 smp_mb();
19 __get_thread_var(rcu_reader_qs_gp) = ACCESS_ONCE(rcu_gp_ctr);
20 smp_mb();
21 }
22
23 static void rcu_thread_online(void)
24 {
25 rcu_quiescent_state();
26 }
27
28

Figure 8.50: Quiescent-State-Based RCU Read Side

8.3.5.9 RCU Based on Quiescent States

Figure 8.50 (rcu_qs.h) shows the read-side primitives used to construct a user-level implementation of RCU based on quiescent states, with the data shown in Figure 8.49. As can be seen from lines 1-7 in the figure, the rcu_read_lock() and rcu_read_unlock() primitives do nothing, and can in fact be expected to be inlined and optimized away, as they are in server builds of the Linux kernel. This is due to the fact that quiescent-state-based RCU implementations approximate the extents of RCU read-side critical sections using the aforementioned quiescent states, which contains calls to rcu_quiescent_state(), shown from lines 9-15 in the figure. Threads entering extended quiescent states (for example, when blocking) may instead use the thread_offline() and thread_online() APIs to mark the beginning and the end, respectively, of such an extended quiescent state. As such, thread_online() is analogous to rcu_read_lock() and thread_offline() is analogous to rcu_read_unlock(). These two functions are shown on lines 17-28 in the figure. In either case, it is illegal for a quiescent state to appear within an RCU read-side critical section.

In rcu_quiescent_state(), line 11 executes a memory barrier to prevent any code prior to the quiescent state from being reordered into the quiescent state. Lines 12-13 pick up a copy of the global rcu_gp_ctr, using ACCESS_ONCE() to ensure that the compiler does not employ any optimizations that would result in rcu_gp_ctr being fetched more than once, and then adds one to the value fetched and stores it into the per-thread rcu_reader_qs_gp variable, so that any concurrent instance of synchronize_rcu() will see an odd-numbered value, thus becoming aware that a new RCU read-side critical section has started. Instances of synchronize_rcu() that are waiting on older RCU read-side critical sections will thus know to ignore this new one. Finally, line 14 executes a memory barrier.

Quick Quiz 8.61: Doesn’t the additional memory barrier shown on line 14 of Figure 8.50, greatly increase the overhead of rcu_quiescent_state? ■

Some applications might use RCU only occasionally, but use it very heavily when they do use it. Such applications might choose to use rcu_thread_offline() when starting to use RCU and rcu_thread_online() when no longer using RCU. The time between a call to rcu_thread_offline() and a subsequent call to rcu_thread_online() is an extended quiescent state, so that RCU will not expect explicit quiescent states to be registered during this time.

The rcu_thread_offline() function simply sets the per-thread rcu_reader_qs_gp variable to the current value of rcu_gp_ctr, which has an even-numbered value. Any concurrent instances of synchronize_rcu() will thus know to ignore this thread.

Quick Quiz 8.62: Why are the two memory barriers on lines 19 and 22 of Figure 8.50 needed? ■

The rcu_thread_online() function simply invokes rcu_quiescent_state(), thus marking the end of the extended quiescent state.

Figure 8.51 (rcu_qs.c) shows the implementation of synchronize_rcu(), which is quite similar to that of the preceding sections.

This implementation has blazingly fast read-side primitives, with an rcu_read_lock().rcu_read_unlock() round trip incurring an overhead of roughly 50 picoseconds. The synchronize_rcu() overhead ranges from about 600 nanoseconds on a single-CPU Power5 system up to more than 100 microseconds on a 64-CPU system.

Quick Quiz 8.63: To be sure, the clock frequencies of ca-2008 Power systems were quite high, but even a 5GHz clock frequency is insufficient to allow loops to be executed in 50 picoseconds! What is going on here? ■
```c
void synchronize_rcu(void) {
    int t;
smp_mb();
    spin_lock(&rcu_gp_lock);
    rcu_gp_ctr += 2;
smp_mb();
    for_each_thread(t) {
        while (rcu_gp_ongoing(t) &&
            ((per_thread(rcu_reader_qs_gp, t) -
            rcu_gp_ctr) < 0)) {
            poll(NULL, 0, 10);
        }
    }
    spin_unlock(&rcu_gp_lock);
smp_mb();
}
```

Figure 8.51: RCU Update Side Using Quiescent States

However, this implementation requires that each thread either invoke `rcu_quiescent_state()` periodically or to invoke `rcu_thread_offline()` for extended quiescent states. The need to invoke these functions periodically can make this implementation difficult to use in some situations, such as for certain types of library functions.

**Quick Quiz 8.64:** Why would the fact that the code is in a library make any difference for how easy it is to use the RCU implementation shown in Figures 8.50 and 8.51?

**Quick Quiz 8.65:** But what if you hold a lock across a call to `synchronize_rcu()`, and then acquire that same lock within an RCU read-side critical section? This should be a deadlock, but how can a primitive that generates absolutely no code possibly participate in a deadlock cycle?

In addition, this implementation does not permit concurrent calls to `synchronize_rcu()` to share grace periods. That said, one could easily imagine a production-quality RCU implementation based on this version of RCU.

### 8.3.5.10 Summary of Toy RCU Implementations

If you made it this far, congratulations! You should now have a much clearer understanding not only of RCU itself, but also of the requirements of enclosing software environments and applications. Those wishing an even deeper understanding are invited to read Appendix D, which presents some RCU implementations that have seen extensive use in production.

The preceding sections listed some desirable properties of the various RCU primitives. The following list is provided for easy reference for those wishing to create a new RCU implementation.

1. There must be read-side primitives (such as `rcu_read_lock()` and `rcu_read_unlock()`) and grace-period primitives (such as `synchronize_rcu()` and `call_rcu()`), such that any RCU read-side critical section in existence at the start of a grace period has completed by the end of the grace period.

2. RCU read-side primitives should have minimal overhead. In particular, expensive operations such as cache misses, atomic instructions, memory barriers, and branches should be avoided.

3. RCU read-side primitives should have $O(1)$ computational complexity to enable real-time use. (This implies that readers run concurrently with updaters.)

4. RCU read-side primitives should be usable in all contexts (in the Linux kernel, they are permitted everywhere except in the idle loop). An important special case is that RCU read-side primitives be usable within an RCU read-side critical section, in other words, that it be possible to nest RCU read-side critical sections.

5. RCU read-side primitives should be unconditional, with no failure returns. This property is extremely important, as failure checking increases complexity and complicates testing and validation.

6. Any operation other than a quiescent state (and thus a grace period) should be permitted in an RCU read-side critical section. This includes non-idempotent operations such as I/O.

7. It should be possible to update an RCU-protected data structure while executing within an RCU read-side critical section.

8. Both RCU read-side and update-side primitives should be independent of memory allocator design and implementation, in other words, the same RCU implementation should be able to protect a given data structure regardless of how the data elements are allocated and freed.

9. RCU grace periods should not be blocked by threads that halt outside of RCU read-side critical sections. (But note that most quiescent-state-based implementations violate this desideratum.)
Quick Quiz 8.66: Given that grace periods are prohibited within RCU read-side critical sections, how can an RCU data structure possibly be updated while in an RCU read-side critical section?

8.3.6 RCU Exercises

This section is organized as a series of Quick Quizzes that invite you to apply RCU to a number of examples earlier in this book. The answer to each Quick Quiz gives some hints, and also contains a pointer to a later section where the solution is explained at length. The `rcu_read_lock()`, `rcu_read_unlock()`, `rcu_dereference()`, `rcu_assign_pointer()`, and `synchronize_rcu()` primitives should suffice for most of these exercises.

Quick Quiz 8.67: The statistical-counter implementation shown in Figure 4.8 (`count_end.c`) used a global lock to guard the summation in `read_count()`, which resulted in poor performance and negative scalability. How could you use RCU to provide `read_count()` with excellent performance and good scalability. (Keep in mind that `read_count()`'s scalability will necessarily be limited by its need to scan all threads' counters.)

Quick Quiz 8.68: Section 4.5 showed a fanciful pair of code fragments that dealt with counting I/O accesses to removable devices. These code fragments suffered from high overhead on the fastpath (starting an I/O) due to the need to acquire a reader-writer lock. How would you use RCU to provide excellent performance and scalability? (Keep in mind that the performance of the common-case first code fragment that does I/O accesses is much more important than that of the device-removal code fragment.)
Chapter 9
Applying RCU

This chapter shows how to apply RCU to some examples discussed earlier in this book. In some cases, RCU provides simpler code, in other cases better performance and scalability, and in still other cases, both.

9.1 RCU and Per-Thread-Variable-Based Statistical Counters

Section 4.2.4 described an implementation of statistical counters that provided excellent performance, roughly that of simple increment (as in the C++ `operator++`), and linear scalability — but only for incrementing via `inc_count()`. Unfortunately, threads needing to read out the value via `read_count()` were required to acquire a global lock, and thus incurred high overhead and suffered poor scalability. The code for the lock-based implementation is shown in Figure 4.8 on Page 33.

Quick Quiz 9.1: Why on earth did we need that global lock in the first place?

9.1.1 Design

The hope is to use RCU rather than `final_mutex` to protect the thread traversal in `read_count()` in order to obtain excellent performance and scalability from `read_count()`, rather than just from `inc_count()`. However, we do not want to give up any accuracy in the computed sum. In particular, when a given thread exits, we absolutely cannot lose the exiting thread’s count, nor can we double-count it. Such an error could result in inaccuracies equal to the full precision of the result, in other words, such an error would make the result completely useless. And in fact, one of the purposes of `final_mutex` is to ensure that threads do not come and go in the middle of `read_count()` execution.

Quick Quiz 9.2: Just what is the accuracy of `read_count()`, anyway?

Therefore, if we are to dispense with `final_mutex`, we will need to come up with some other method for ensuring consistency. One approach is to place the total count for all previously exited threads and the array of pointers to the per-thread counters into a single structure. Such a structure, once made available to `read_count()`, is held constant, ensuring that `read_count()` sees consistent data.

9.1.2 Implementation

Lines 1-4 of Figure 9.1 show the `countarray` structure, which contains a `->total` field for the count from previously exited threads, and a `counterp[]` array of pointers to the per-thread counter for each currently running thread. This structure allows a given execution of `read_count()` to see a total that is consistent with the indicated set of running threads.

Quick Quiz 9.1: Why on earth did we need that global lock in the first place?

Lines 6-8 contain the definition of the per-thread counter variable, the global pointer `countarrayp` referencing the current `countarray` structure, and the `final_mutex` spinlock.

Lines 10-13 show `inc_count()`, which is unchanged from Figure 4.8.

Lines 15-29 show `read_count()`, which has changed significantly. Lines 21 and 27 substitute `rcu_read_lock()` and `rcu_read_unlock()` for acquisition and release of `final_mutex`. Line 22 uses `rcu_dereference()` to snapshot the current `countarray` structure into local variable `cap`. Proper use of RCU will guarantee that this `countarray` structure will remain with us through at least the end of the current RCU read-side critical section at line 27. Line 23
struct countarray {
    unsigned long total;
    unsigned long *counterp[NR_THREADS];
};

long __thread counter = 0;
struct countarray *countarrayp = NULL;
DEFINE_SPINLOCK(final_mutex);

void inc_count(void)
{
    counter++;
}

long read_count(void)
{
    struct countarray *cap;
    unsigned long sum;
    int t;
    rcu_read_lock();
    cap = rcu_dereference(countarrayp);
    sum = cap->total;
    for_each_thread(t)
        if (cap->counterp[t] != NULL)
            sum += *cap->counterp[t];
    rcu_read_unlock();
    return sum;
}

void count_init(void)
{
    countarrayp = malloc(sizeof(*countarrayp));
    if (countarrayp == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(-1);
    }
    memset(countarrayp, '0', sizeof(*countarrayp));
}

void count_register_thread(void)
{
    int idx = smp_thread_id();
    spin_lock(&final_mutex);
    countarrayp->counterp[idx] = &counter;
    spin_unlock(&final_mutex);
}

void count_unregister_thread(int nthreadsexpected)
{
    struct countarray *cap;
    struct countarray *capold;
    int idx = smp_thread_id();
    cap = malloc(sizeof(*countarrayp));
    if (cap == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(-1);
    }
    memset(countarrayp, '\0', sizeof(*countarrayp));
    spin_lock(&final_mutex);
    *cap = *countarrayp;
    cap->total += counter;
    cap->counterp[idx] = NULL;
    capold = countarrayp;
    rcu_assign_pointer(countarrayp, cap);
    spin_unlock(&final_mutex);
    synchronize_rcu();
    free(capold);
}

void count_init(void)
{
    int idx = smp_thread_id();
    spin_lock(&final_mutex);
    countarrayp = malloc(sizeof(*countarrayp));
    if (countarrayp == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(-1);
    }
    memset(countarrayp, '0', sizeof(*countarrayp));
    spin_unlock(&final_mutex);
    rcu_read_lock();
    cap = rcu_dereference(countarrayp);
    sum = cap->total;
    for_each_thread(t)
        if (cap->counterp[t] != NULL)
            sum += *cap->counterp[t];
    rcu_read_unlock();
    return sum;
}

Quick Quiz 9.3: Hey!!! Line 45 of Figure 9.1 modifies a value in a pre-existing countarray structure! Didn’t you say that this structure, once made available to read_count(), remained constant??!

Lines 50-70 shows count_unregister_thread(), which is invoked by each thread just before it exits. Lines 56-60 allocate a new countarray structure, line 61 acquires final_mutex and line 67 releases it. Line 62 copies the contents of the current countarray into the newly allocated version, line 63 adds the exiting thread’s counter to new structure’s total, and line 64 NULLs the exiting thread’s counterp[] array element. Line 65 then retains a pointer to the current (soon to be old) countarray structure, and line 66 uses rcu_assign_pointer() to install the new version of the countarray structure. Line 68 waits for a grace period to elapse, so that any threads that might be concurrently executing in read_count, and thus might have references to the old countarray structure, will be allowed to exit their RCU read-side critical sections, thus dropping any such references. Line 69 can then safely free the old countarray structure.

9.1.3 Discussion

Quick Quiz 9.4: Wow! Figure 9.1 contains 69 lines of code, compared to only 42 in Figure 4.8. Is this extra complexity really worth it? Use of RCU enables exiting threads to wait until other threads are guaranteed to be done using the exiting threads’ __thread variables. This allows the read_count() function to dispense with locking, thereby providing excellent performance and scalability for both the
inc_count() and read_count() functions. However, this performance and scalability come at the cost of some increase in code complexity. It is hoped that compiler and library writers employ user-level RCU [Des09] to provide safe cross-thread access to __thread variables, greatly reducing the complexity seen by users of __thread variables.

9.2 RCU and Counters for Removable I/O Devices

Section 4.5 showed a fanciful pair of code fragments for dealing with counting I/O accesses to removable devices. These code fragments suffered from high overhead on the fastpath (starting an I/O) due to the need to acquire a reader-writer lock.

This section shows how RCU may be used to avoid this overhead.

The code for performing an I/O is quite similar to the original, with an RCU read-side critical section be substituted for the reader-writer lock read-side critical section in the original:

```c
1   rcu_read_lock();
2   if (removing) {
3     rcu_read_unlock();
4     cancel_io();
5   } else {
6     add_count(1);
7     rcu_read_unlock();
8     do_io();
9     sub_count(1);
10  }
```

The RCU read-side primitives have minimal overhead, thus speeding up the fastpath, as desired.

The updated code fragment removing a device is as follows:

```c
1   spin_lock(&mylock);
2   removing = 1;
3   sub_count(mybias);
4   spin_unlock(&mylock);
5   synchronize_rcu();
6   while (read_count() != 0) {
7       poll(NULL, 0, 1);
8   }
9   remove_device();
```

Because of the synchronize_rcu(), once we reach line 6, we know that all remaining I/Os have been accounted for.

Of course, the overhead of synchronize_rcu() can be large, but given that device removal is quite rare, this is usually a good tradeoff.
Chapter 10

Validation

I have had a few parallel programs work the first time, but that is only because I have written so many parallel programs over the past two decades. And I have had far more parallel programs that fooled me into thinking that they were working correctly the first time than actually were working the first time.

I have therefore had great need of validation for my parallel programs. The basic trick behind parallel validation, as with other software validation, is to realize that the computer knows what is wrong. It is therefore your job to force it to tell you. This chapter can therefore be thought of as a short course in machine interrogation.\footnote{But you can leave the thumbscrews and waterboards at home. This chapter covers much more sophisticated and effective methods, especially when you consider that most computer systems neither feel pain nor fear drowning.}

A longer course may be found in many recent books on validation, as well as at least one rather old but quite worthwhile one [Myc79]. Validation is an extremely important topic that cuts across all forms of software, and is therefore worth intensive study in its own right. However, this book is primarily about concurrency, so this chapter will necessarily do little more than scratch the surface of this critically important topic.

@@@ roadmap

10.1 Required Mindset

When carrying out any validation effort, you should keep the following definitions in mind:

1. The only bug-free programs are trivial programs.
2. A reliable program has no known bugs.

From these definitions, it logically follows that any reliable non-trivial program contains at least one bug that you do not know about. Therefore, any validation effort undertaken on a non-trivial program that fails to find any bugs is itself a failure. A good validation is therefore an exercise in destruction. It helps if you deeply enjoy breaking things.

Quick Quiz 10.1: Suppose that you are writing a script that processes the output of the \texttt{time} command, which looks as follows:

\begin{verbatim}
real 0m0.132s
user 0m0.040s
sys 0m0.008s
\end{verbatim}

The script is required to check its input for errors, and to give appropriate diagnostics if fed erroneous \texttt{time} output. What test inputs should you provide to this program to test it for use with \texttt{time} output generated by single-threaded programs? ■

10.2 Tracing

When all else fails, add a \texttt{printf()}! Or a \texttt{printf()}, if you are working with user-mode applications.

The rationale is simple: If you cannot figure out how execution reached a given point in the code, sprinkle print statements earlier in the code to work out what happened. You can get a similar effect, and with more convenience and flexibility, by using a debugger such as \texttt{gdb} (for user applications) or \texttt{kgdb} (for debugging Linux kernels). Much more sophisticated tools exist, with some of the more recent offering the ability to rewind backwards in time from the point of failure.

These brute-force testing tools are all valuable, and much has been written about them, so this chapter will add little more.

However, they all have a serious shortcoming when the job at hand is to convince a the fastpath of a high-performance parallel algorithm to tell you what is go-
ing wrong, namely, they often have excessive overheads. There are special tracing technologies for this purpose, which typically leverage data ownership techniques (see Chapter 7) to minimize the overhead of runtime data collection. One example within the Linux kernel is “trace events” [Ros10b, Ros10c, Ros10d, Ros10a]. Another example that handles userspace (but has not been accepted into the Linux kernel) is LTTng [DD09]. Each of these uses per-CPU buffers to allow data to be collected with extremely low overhead.

Problems with brute-force debugging. The machine knows all, which is almost always more than your head can hold. Post-processing with scripts. But beware – scripts won’t necessarily notice surprising things.

10.3 Assertions

straight assertions (improvement over comments), lock-dep.

10.4 Static Analysis

Compiler warnings.

The sparse static analyzer.

10.5 Probability and Heisenbugs

So your parallel program fails. Sometimes.

But you figured out the problem and now have a fix in place! Congratulations!!!

But now how much testing do you have to do in order to be certain that you actually fixed the bug, as opposed to just reducing the probability of it occurring on the one hand or having fixed only one of several related bugs on the other?

Unfortunately, the honest answer is that an infinite amount of testing is required to attain absolute certainty.

Quick Quiz 10.2: Suppose that you had a very large number of systems at your disposal. For example, at current cloud prices, you can purchase a huge amount of CPU time at a reasonably low cost. Why not use this approach to get close enough to certainty for all practical purposes?

But suppose that we are willing to give up absolute certainty in favor of high probability. Then we can bring powerful statistical tools to bear on this problem. However, this section will focus on simple statistical tools.

These tools are extremely helpful, but readers should not make the mistake of assuming that reading this section is in any way a substitute for taking a good set of statistics classes.\(^2\)

For our start with simple statistical tools, we need to decide whether we are doing discrete or continuous testing. Discrete testing features well-defined individual test runs. For example, a boot-up test of a Linux kernel patch is an example of a discrete test. You boot the kernel, and it either comes up or it does not. Although you might spend an hour boot-testing your kernel, the number of times you attempted to boot the kernel and the number of times the boot-up succeeded would often be of more interest than the length of time you spent testing. Functional tests tend to be discrete.

On the other hand, if my patch involved RCU, I would probably run rcutorture, which is a kernel module that, strangely enough, tests RCU. Unlike booting the kernel, where the appearance of a login prompt signals the successful end of a discrete test, rcutorture will happily continue torturing RCU until either the kernel crashes or until you tell it to stop. The duration of the rcutorture test is therefore (usually) of more interest than the number of times you started and stopped it. Therefore, rcutorture is an example of a continuous test, a category that includes many stress tests.

The statistics governing discrete and continuous tests differs somewhat. However, the statistics for discrete tests is simpler and more familiar than that for continuous tests, and furthermore the statistics for discrete tests can often be pressed into service (with some loss of accuracy) for continuous tests, we start with discrete tests.

10.5.1 Statistics for Discrete Testing

Suppose that the bug had a 10% chance of occurring in a given run and that we do five runs. How do we compute that probability of at least one run failing? One way is as follows:

1. Compute the probability of a given run succeeding, which is 90%.

2. Compute the probability of all five runs succeeding, which is 0.9 raised to the fifth power, or about 59%.

3. There are only two possibilities: either all five runs succeed, or at least one fails. Therefore, the proba-

\(^2\) Which I most highly recommend. The few statistics courses I have taken have provided value way out of proportion to the time I spent studying for them.
bility of at least one failure is 59% taken away from
100%, or 41%.

However, many people find it easier to work with a
formula than a series of steps, although if you prefer the
above series of steps, have at it! For those who like for-
mulas, call the probability of a single failure \( f \). The prob-
bility of a single success is then \( 1 - f \) and the probability
that all of \( n \) tests will succeed is then:

\[
S_n = (1 - f)^n \quad (10.1)
\]

The probability of failure is \( 1 - S_n \), or:

\[
F_n = 1 - (1 - f)^n \quad (10.2)
\]

Quick Quiz 10.3: Say what??? When I plug the earlier
example of five tests each with a 10% failure rate into
the formula, I get 59,050% and that just doesn’t make
sense!!

So suppose that a given test has been failing 10% of
the time. How many times do you have to run the test to
be 99% sure that your supposed fix has actually improved
matters?

Another way to ask this question is “how many times
would we need to run the test to cause the probability of
failure to rise above 99%?” After all, if we were to run
the test enough times that the probability of seeing at least
one failure becomes 99%, if there are no failures, there
is only 1% probability of this being due to dumb luck.
And if we plug \( f = 0.1 \) into Equation 10.2 and vary \( n \),
we find that 43 runs gives us a 98.92% chance of at least
one test failing given the original 10% per-test failure rate,
while 44 runs gives us a 99.03% chance of at least one
test failing. So if we run the test on our fix 44 times and
see no failures, there is a 99% probability that our fix was
actually a real improvement.

But repeatedly plugging numbers into Equation 10.2
can get tedious, so let’s solve for \( n \):

\[
F_n = 1 - (1 - f)^n \quad (10.3)
\]

1 - \( F_n \) = \( (1 - f)^n \) \quad (10.4)

\[
\log (1 - F_n) = n \log (1 - f) \quad (10.5)
\]

Finally the number of tests required is given by:

\[
n = \frac{\log (1 - F_n)}{\log (1 - f)} \quad (10.7)
\]

Quick Quiz 10.4: In Equation 10.7, are the logarithms
base-10, base-2, or base-\( e \)?

Figure 10.1 shows a plot of this function. Not surpris-
ingly, the less frequently each test run fails, the more test
runs are required to be 99% confident that the bug has
been fixed. If the bug caused the test to fail only 1% of
the time, then a mind-boggling 458 test runs are required.

The moral of this story is that when you have found a
rarely occurring bug, your testing job will be much easier
if you can come up with a carefully targeted test with a
much higher failure rate. For example, if your targeted test
raised the failure rate from 1% to 30%, then the number
of runs required for 99% confidence would drop from a
mind-boggling 458 test runs to a mere thirteen test runs.

But these thirteen test runs would only give you 99% confidence that your fix had produced “some improve-
mation”. Suppose you instead want to have 99% confidence that your fix reduced the failure rate by an order of mag-
nitude. How many failure-free test runs are required?

An order of magnitude improvement from a 30% fail-
ure rate would be a 3% failure rate. Plugging these num-
bers into Equation 10.7 yields:

\[
n = \frac{\log (1 - 0.99)}{\log (1 - 0.03)} = 151.2 \quad (10.8)
\]

So our order of magnitude improvement requires
roughly an order of magnitude more testing. Certainty
is impossible, and high probabilities are quite expensive.
Clearly making tests run more quickly and making failures more probable are essential skills in the development of highly reliable software. These skills will be covered in a later section.

10.5.2 Abusing Statistics for Discrete Testing

But suppose that you have a continuous test that fails about three times every ten hours, and that you fix the bug that you believe was causing the failure. How long do you have to run this test without failure to be 99% certain that you reduced the probability of failure?

Without doing excessive violence to statistics, we could simply redefine a one-hour run to be a discrete test that has a 30% probability of failure. Then the results of in the previous section tell us that if the test runs for 13 hours without failure, there is a 99% probability that our fix actually improved the program’s reliability.

A dogmatic statistician might not approve of this approach, but the sad fact is that the errors introduced by this sort of abuse of statistical methodology are usually way down in the noise compared to the errors inherent in your measurements of your program’s failure rates. Nevertheless, the next section describes a slightly less dodgy approach.

10.5.3 Statistics for Continuous Testing

@@@ continuous formulation for time-based tests leads to Poisson distribution. This can be handled using maxima with quantile_poisson() and cdf_poisson(), after a load(distrib).

10.5.4 Heisenbugs and Creating Anti-Heisenbugs

This line of thought also leads to an understanding of heisenbugs: adding tracing and assertions can easily reduce the probability of a bug appearing. And this is why extremely lightweight tracing and assertion mechanism are so critically important.

Measure bug probability as a function of configuration parameters, input, intensity of load, number of CPUs, etc. Set up experiments.

10.6 Profiling

10.7 Differential Profiling


10.8 Performance Estimation

Chapter 11

Data Structures

11.1 Lists
Lists, double lists, hlists, hashes, trees, rbtrees, radix trees.

11.2 Computational Complexity and Performance
Complexity, performance, O(N).

11.3 Design Tradeoffs
Trade-offs between memory consumption, performance, complexity.

11.4 Protection
Compiler (e.g., const) and hardware.

11.5 Bits and Bytes
Bit fields, endianness, packing.

11.6 Hardware Considerations
CPU word alignment, cache alignment.

@@ pull in material from Orran Kreiger’s 1995 paper (permission granted).
Chapter 12

Advanced Synchronization

12.1 Avoiding Locks

List the ways: RCU, non-blocking synchronization (notably simpler forms), memory barriers, deferred processing.

@@@ Pull deferral stuff back to this section?

12.2 Memory Barriers

Author: David Howells and Paul McKenney.

Causality and sequencing are deeply intuitive, and hackers often tend to have a much stronger grasp of these concepts than does the general population. These intuitions can be extremely powerful tools when writing, analyzing, and debugging both sequential code and parallel code that makes use of standard mutual-exclusion mechanisms, such as locking and RCU.

Unfortunately, these intuitions break down completely in face of code that makes direct use of explicit memory barriers for data structures in shared memory (driver writers making use of MMIO registers can place greater trust in their intuition, but more on this @@ later). The following sections show exactly where this intuition breaks down, and then puts forward a mental model of memory barriers that can help you avoid these pitfalls.

Section 12.2.1 gives a brief overview of memory ordering and memory barriers. Once this background is in place, the next step is to get you to admit that your intuition has a problem. This painful task is taken up by Section 12.2.2, which shows an intuitively correct code fragment that fails miserably on real hardware, and by Section 12.2.3, which presents some code demonstrating that scalar variables can take on multiple values simultaneously. Once your intuition has made it through the grieving process, Section 12.2.4 provides the basic rules that memory barriers follow, rules that we will build upon.

@@@ roadmap...

12.2.1 Memory Ordering and Memory Barriers

But why are memory barriers needed in the first place? Can’t CPUs keep track of ordering on their own? Isn’t that why we have computers in the first place, to keep track of things?

Many people do indeed expect their computers to keep track of things, but many also insist that they keep track of things quickly. One difficulty that modern computer-system vendors face is that the main memory cannot keep up with the CPU – modern CPUs can execute hundreds of instructions in time required to fetch a single variable from memory. CPUs therefore sport increasingly large caches, as shown in Figure 12.1. Variables that are heavily used by a given CPU will tend to remain in that CPU’s cache, allowing high-speed access to the corresponding data.

![Figure 12.1: Modern Computer System Cache Structure](image)

Figure 12.1: Modern Computer System Cache Structure
Unfortunately, when a CPU accesses data that is not yet in its cache will result in an expensive “cache miss”, requiring the data to be fetched from main memory. Doubly unfortunately, running typical code results in a significant number of cache misses. To limit the resulting performance degradation, CPUs have been designed to execute other instructions and memory references while waiting for a cache miss to fetch data from memory. This clearly causes instructions and memory references to execute out of order, which could cause serious confusion, as illustrated in Figure 12.2. Compilers and synchronization primitives (such as locking and RCU) are responsible for maintaining the illusion of ordering through use of “memory barriers” (for example, `smp_mb()` in the Linux kernel). These memory barriers can be explicit instructions, as they are on ARM, POWER, Itanium, and Alpha, or they can be implied by other instructions, as they are on x86.

![Figure 12.2: CPUs Can Do Things Out of Order](image)

Since the standard synchronization primitives preserve the illusion of ordering, your path of least resistance is to stop reading this section and simply use these primitives.

However, if you need to implement the synchronization primitives themselves, or if you are simply interested in understanding how memory ordering and memory barriers work, read on!

The next sections present counter-intuitive scenarios that you might encounter when using explicit memory barriers.

12.2.2 If B Follows A, and C Follows B, Why Doesn’t C Follow A?

Memory ordering and memory barriers can be extremely counter-intuitive. For example, consider the functions shown in Figure 12.3 executing in parallel where variables A, B, and C are initially zero:

```c
1 thread0(void)  
2 {  
3     A = 1;  
4     smp_mb();  
5     B = 1;  
6 }  
7  
8 thread1(void)  
9 {  
10     while (B != 1) continue;  
11     barrier();  
12     C = 1;  
13 }  
14  
15 thread2(void)  
16 {  
17     while (C != 1) continue;  
18     smp_mb();  
19     assert(A != 0);  
20 }
```

Figure 12.3: Parallel Hardware is Non-Causal

Intuitively, `thread0()` assigns to B after it assigns to A, `thread1()` waits until `thread0()` has assigned to B before assigning to C, and `thread2()` waits until `thread1()` has assigned to C before referencing A. Therefore, again intuitively, the assertion on line 21 cannot possibly fire.

This line of reasoning, intuitively obvious though it may be, is completely and utterly incorrect. Please note that this is not a theoretical assertion: actually running this code on real-world weakly-ordered hardware (a 1.5GHz 16-CPU POWER 5 system) resulted in the assertion firing 16 times out of 10 million runs. Clearly, anyone who produces code with explicit memory barriers should do some extreme testing – although a proof of correctness might be helpful, the strongly counter-intuitive nature of the behavior of memory barriers should in turn strongly limit one’s trust in such proofs. The requirement for extreme testing should not be taken lightly, given that a number of dirty hardware-dependent tricks were used to greatly increase the probability of failure in this run.

Quick Quiz 12.1: How on earth could the assertion on line 21 of the code in Figure 12.3 on page 140 possibly fail? ■

Quick Quiz 12.2: Great... So how do I fix it? ■
So what should you do? Your best strategy, if possible, is to use existing primitives that incorporate any needed memory barriers, so that you can simply ignore the rest of this chapter.

Of course, if you are implementing synchronization primitives, you don’t have this luxury. The following discussion of memory ordering and memory barriers is for you.

12.2.3 Variables Can Have More Than One Value

It is natural to think of a variable as taking on a well-defined sequence of values in a well-defined, global order. Unfortunately, it is time to say “goodbye” to this sort of comforting fiction.

To see this, consider the program fragment shown in Figure 12.4. This code fragment is executed in parallel by several CPUs. Line 1 sets a shared variable to the current CPU’s ID, line 2 initializes several variables from a gettb() function that delivers the value of fine-grained hardware “timebase” counter that is synchronized among all CPUs (not available from all CPU architectures, unfortunately!), and the loop from lines 3-8 records the length of time that the variable retains the value that this CPU assigned to it. Of course, one of the CPUs will “win”, and would thus never exit the loop if not for the check on lines 7-8.

Quick Quiz 12.3: What assumption is the code fragment in Figure 12.4 making that might not be valid on real hardware?

Each horizontal bar represents the observations of a given CPU over time, with the black regions to the left indicating the time before the corresponding CPU’s first measurement. During the first 5ns, only CPU 3 has an opinion about the value of the variable. During the next 10ns, CPUs 2 and 3 disagree on the value of the variable, but thereafter agree that the value is “2”, which is in fact the final agreed-upon value. However, CPU 1 believes that the value is “1” for almost 300ns, and CPU 4 believes that the value is “4” for almost 500ns.

Quick Quiz 12.4: How could CPUs possibly have different views of the value of a single variable at the same time?

Quick Quiz 12.5: Why do CPUs 2 and 3 come to agreement so quickly, when it takes so long for CPUs 1 and 4 to come to the party?

We have entered a regime where we must bade a fond farewell to comforting intuitions about values of variables and the passage of time. This is the regime where memory barriers are needed.

12.2.4 What Can You Trust?

You most definitely cannot trust your intuition.

What can you trust?

It turns out that there are a few reasonably simple rules that allow you to make good use of memory barriers. This section derives those rules, for those who wish to get to the bottom of the memory-barrier story, at least from the viewpoint of portable code. If you just want to be told what the rules are rather than suffering through the actual derivation, please feel free to skip to Section 12.2.6.

Figure 12.5: A Variable With Multiple Simultaneous Values
The exact semantics of memory barriers vary wildly from one CPU to another, so portable code must rely only on the least-common-denominator semantics of memory barriers.

Fortunately, all CPUs impose the following rules:

1. All accesses by a given CPU will appear to that CPU to have occurred in program order.

2. All CPUs’ accesses to a single variable will be consistent with some global ordering of stores to that variable.

3. Memory barriers will operate in a pair-wise fashion.

4. Operations will be provided from which exclusive locking primitives may be constructed.

Therefore, if you need to use memory barriers in portable code, you can rely on all of these properties. Each of these properties is described in the following sections.

12.2.4.1 Self-References Are Ordered

A given CPU will see its own accesses as occurring in “program order”, as if the CPU was executing only one instruction at a time with no reordering or speculation. For older CPUs, this restriction is necessary for binary compatibility, and only secondarily for the sanity of us software types. There have been a few CPUs that violate this rule to a limited extent, but in those cases, the compiler has been responsible for ensuring that ordering is explicitly enforced as needed.

Either way, from the programmer’s viewpoint, the CPU sees its own accesses in program order.

12.2.4.2 Single-Variable Memory Consistency

If a group of CPUs all do concurrent stores to a single variable, the series of values seen by all CPUs will be consistent with at least one global ordering. For example, in the series of accesses shown in Figure 12.5, CPU 1 sees the sequence \(1, 2\), CPU 2 sees the sequence \(2\), CPU 3 sees the sequence \(3, 2\), and CPU 4 sees the sequence \(4, 2\). This is consistent with the global sequence \(3, 1, 4, 2\), but also with all five of the other sequence of these four numbers that end in “2”.

Had the CPUs used atomic operations (such as the Linux kernel’s `atomic_inc_return()` primitive) rather than simple stores of unique values, their observations would be guaranteed to determine a single globally consistent sequence of values.

12.2.4.3 Pair-Wise Memory Barriers

Pair-wise memory barriers provide conditional ordering semantics. For example, in the following set of operations, CPU 1’s access to A does not unconditionally precede its access to B from the viewpoint of an external logic analyzer (see Appendix C for examples). However, if CPU 2’s access to B sees the result of CPU 1’s access to B, then CPU 2’s access to A is guaranteed to see the result of CPU 1’s access to A. Although some CPUs’ memory barriers do in fact provide stronger, unconditional ordering guarantees, portable code may rely only on this weaker if-then conditional ordering guarantee.

```
CPU 1  CPU 2
access(A); access(B);
smp_mb(); smp_mb();
access(B); access(A);
```

Quick Quiz 12.6: But if the memory barriers do not unconditionally force ordering, how the heck can a device driver reliably execute sequences of loads and stores to MMIO registers? ■

Of course, accesses must be either loads or stores, and these do have different properties. Table 12.1 shows all possible combinations of loads and stores from a pair of CPUs. Of course, to enforce conditional ordering, there must be a memory barrier between each CPU’s pair of operations.

12.2.4.4 Pair-Wise Memory Barriers: Portable Combinations

The following pairings from Table 12.1, enumerate all the combinations of memory-barrier pairings that portable software may depend on.

Pairing 1. In this pairing, one CPU executes a pair of loads separated by a memory barrier, while a second CPU executes a pair of stores also separated by a memory barrier, as follows (both A and B are initially equal to zero):

```
CPU 1  CPU 2
A=1;    Y=B;
smp_mb(); smp_mb();
B=1;    X=A;
```

After both CPUs have completed executing these code sequences, if \(Y==1\), then we must also have \(X==1\). In
12.2. MEMORY BARRIERS

<table>
<thead>
<tr>
<th>CPU 1</th>
<th>CPU 2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>load(A) load(B) load(B) load(A)</td>
<td>Ears to ears.</td>
</tr>
<tr>
<td>1</td>
<td>load(A) load(B) load(B) store(A)</td>
<td>Only one store.</td>
</tr>
<tr>
<td>2</td>
<td>load(A) load(B) store(B) load(A)</td>
<td>Only one store.</td>
</tr>
<tr>
<td>3</td>
<td>load(A) load(B) store(B) store(A)</td>
<td>Pairing 1.</td>
</tr>
<tr>
<td>4</td>
<td>load(A) store(B) load(B) load(A)</td>
<td>Only one store.</td>
</tr>
<tr>
<td>5</td>
<td>load(A) store(B) load(B) store(A)</td>
<td>Pairing 2.</td>
</tr>
<tr>
<td>6</td>
<td>load(A) store(B) store(B) load(A)</td>
<td>Mouth to mouth, ear to ear.</td>
</tr>
<tr>
<td>7</td>
<td>load(A) store(B) store(B) store(A)</td>
<td>Pairing 3.</td>
</tr>
<tr>
<td>8</td>
<td>store(A) load(B) load(B) load(A)</td>
<td>Only one store.</td>
</tr>
<tr>
<td>9</td>
<td>store(A) load(B) load(B) store(A)</td>
<td>Mouth to mouth, ear to ear.</td>
</tr>
<tr>
<td>A</td>
<td>store(A) load(B) store(B) load(A)</td>
<td>Ears to mouths.</td>
</tr>
<tr>
<td>B</td>
<td>store(A) load(B) store(B) store(A)</td>
<td>Stores “pass in the night”.</td>
</tr>
<tr>
<td>C</td>
<td>store(A) store(B) load(B) load(A)</td>
<td>Pairing 1.</td>
</tr>
<tr>
<td>D</td>
<td>store(A) store(B) load(B) store(A)</td>
<td>Pairing 3.</td>
</tr>
<tr>
<td>E</td>
<td>store(A) store(B) store(B) load(A)</td>
<td>Stores “pass in the night”.</td>
</tr>
<tr>
<td>F</td>
<td>store(A) store(B) store(B) store(A)</td>
<td>Stores “pass in the night”.</td>
</tr>
</tbody>
</table>

Table 12.1: Memory-Barrier Combinations

In this case, the fact that $Y==1$ means that CPU 2’s load prior to its memory barrier has seen the store following CPU 1’s memory barrier. Due to the pairwise nature of memory barriers, CPU 2’s load following its memory barrier must therefore see the store that precedes CPU 1’s memory barrier, so that $Y==1$.

On the other hand, if $Y==0$, the memory-barrier condition does not hold, and so in this case, X could be either 0 or 1.

**Pairing 2.** In this pairing, each CPU executes a load followed by a memory barrier followed by a store, as follows (both A and B are initially equal to zero):

```
CPU 1         CPU 2
X=A;         Y=B;
smp_mb();    smp_mb();
B=1;         A=1;
```

After both CPUs have completed executing these code sequences, if $X==1$, then we must also have $Y==0$. In this case, the fact that $X==1$ means that CPU 1’s load prior to its memory barrier has seen the store following CPU 2’s memory barrier. Due to the pairwise nature of memory barriers, CPU 1’s store following its memory barrier must therefore see the results of CPU 2’s load preceding its memory barrier, so that $Y==0$.

On the other hand, if $X==0$, the memory-barrier condition does not hold, and so in this case, Y could be either 0 or 1.

The two CPUs’ code sequences are symmetric, so if $Y==1$ after both CPUs have finished executing these code sequences, then we must have $X==0$.

**Pairing 3.** In this pairing, one CPU executes a load followed by a memory barrier followed by a store, while the other CPU executes a pair of stores separated by a memory barrier, as follows (both A and B are initially equal to zero):

```
CPU 1         CPU 2
X=A;         B=2;
smp_mb();    smp_mb();
B=1;         A=1;
```

After both CPUs have completed executing these code sequences, if $X==1$, then we must also have $B==1$. In this case, the fact that $X==1$ means that CPU 1’s load prior to its memory barrier has seen the store following CPU 2’s memory barrier. Due to the pairwise nature of memory barriers, CPU 1’s store following its memory barrier must therefore see the results of CPU 2’s store preceding its memory barrier. This means that CPU 1’s store to B will overwrite CPU 2’s store to B, resulting in $B==1$.

On the other hand, if $X==0$, the memory-barrier condition does not hold, and so in this case, B could be either 1 or 2.
12.2.4.5 Pair-Wise Memory Barriers: Semi-Portable Combinations

The following pairings from Table 12.1 can be used on modern hardware, but might fail on some systems that were produced in the 1990s. However, these can safely be used on all mainstream hardware introduced since the year 2000.

**Ears to Mouths.** Since the stores cannot see the results of the loads (again, ignoring MMIO registers for the moment), it is not always possible to determine whether the memory-barrier condition has been met. However, recent hardware would guarantee that at least one of the loads saw the value stored by the corresponding store (or some later value for that same variable).

**Stores “Pass in the Night”.** In the following example, after both CPUs have finished executing their code sequences, it is quite tempting to conclude that the result \( \{A=1, B=2\} \) cannot happen.\(^2\)

```
CPU 1  CPU 2
A=1;   B=2;
smp_mb(); smp_mb();
B=1;   A=2;
```

Unfortunately, such a conclusion does not necessarily hold on all 20th-century systems. Suppose that the cache line containing \( A \) is initially owned by CPU 2, and that containing \( B \) is initially owned by CPU 1. Then, in systems that have invalidation queues and store buffers, it is possible for the first assignments to “pass in the night”, so that the second assignments actually happen first. This strange (but quite common) effect is explained in Appendix C.

This same effect can happen in any memory-barrier pairing where each CPU’s memory barrier is preceded by a store, including the “ears to mouths” pairing.

However, 21st-century hardware does accommodate ordering intuitions, and do permit this combination to be used safely.

12.2.4.6 Pair-Wise Memory Barriers: Non-Portable Combinations

In the following pairings from Table 12.1, the memory barriers have no effect that portable code can safely depend on.

**Ears to Ears.** Since loads do not change the state of memory (ignoring MMIO registers for the moment), it is not possible for one of the loads to see the results of the other load.

**Mouth to Mouth, Ear to Ear.** One of the variables is only loaded from, and the other is only stored to. Because (once again, ignoring MMIO registers) it is not possible for one load to see the results of the other, it is not possible to detect the conditional ordering provided by the memory barrier. (Yes, it is possible to determine which store happened last, but this does not depend on the memory barrier.)

**Only One Store.** Because there is only one store, only one of the variables permits one CPU to see the results of the other CPU’s access. Therefore, there is no way to detect the conditional ordering provided by the memory barriers. (Yes, it is possible to determine whether or not the load saw the result of the corresponding store, but this does not depend on the memory barrier.)

12.2.4.7 Semantics Sufficient to Implement Locking

Suppose we have an exclusive lock \((\text{spinlock}_t \text{ in the Linux kernel, pthread_mutex}_t \text{ in pthreads code})\)

that guards a number of variables (in other words, these variables are not accessed except from the lock’s critical sections). The following properties must then hold true:

1. A given CPU or thread must see all of its own loads and stores as if they had occurred in program order.

2. The lock acquisitions and releases must appear to have executed in a single global order.\(^2\)

3. Suppose a given variable has not yet been stored to in a critical section that is currently executing. Then any load from a given variable performed in that critical section must see the last store to that variable from the last previous critical section that stored to it.

The difference between the last two properties is a bit subtle: the second requires that the lock acquisitions and releases occur in a well-defined order, while the third requires that the critical sections not “bleed out” far enough to cause difficulties for other critical section.

\(^2\) Of course, this order might be different from one run to the next. On any given run, however, all CPUs and threads must have a consistent view of the order of critical sections for a given exclusive lock.
12.2. MEMORY BARRIERS

Why are these properties necessary?
Suppose the first property did not hold. Then the assertion in the following code might well fail!

```c
a = 1;
b = 1 + a;
assert(b == 2);
```

Quick Quiz 12.7: How could the assertion `b==2` on page 145 possibly fail?
Suppose that the second property did not hold. Then the following code might leak memory!

```c
spin_lock(&mylock);
if (p == NULL)
p = kmalloc(sizeof(*p), GFP_KERNEL);
spin_unlock(&mylock);
```

Quick Quiz 12.8: How could the code on page 145 possibly leak memory?
Suppose that the third property did not hold. Then the counter shown in the following code might well count backwards. This third property is crucial, as it cannot be strictly with pairwise memory barriers.

```c
spin_lock(&mylock);
ctr = ctr + 1;
spin_unlock(&mylock);
```

Quick Quiz 12.9: How could the code on page 145 possibly count backwards?
If you are convinced that these rules are necessary, let’s look at how they interact with a typical locking implementation.

12.2.5 Review of Locking Implementations

Naive pseudocode for simple lock and unlock operations are shown below. Note that the `atomic_xchg()` primitive implies a memory barrier both before and after the atomic exchange operation, which eliminates the need for an explicit memory barrier in `spin_lock()`. Note also that, despite the names, `atomic_read()` and `atomic_set()` do not execute any atomic instructions, instead, it merely executes a simple load and store, respectively. This pseudocode follows a number of Linux implementations for the unlock operation, which is a simple non-atomic store following a memory barrier. These minimal implementations must possess all the locking properties laid out in Section 12.2.4.

```c
1 void spin_lock(spinlock_t *lck)
2 {
3   while (atomic_xchg(&lck->a, 1) != 0)
4     while (atomic_read(&lck->a) != 0)
5       continue;
6 }
7
8 void spin_unlock(spinlock_t lck)
9 {
10   smp_mb();
11   atomic_set(&lck->a, 0);
12 }
```

The `spin_lock()` primitive cannot proceed until the preceding `spin_unlock()` primitive completes. If CPU 1 is releasing a lock that CPU 2 is attempting to acquire, the sequence of operations might be as follows:

<table>
<thead>
<tr>
<th>CPU 1</th>
<th>CPU 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(critical section)</td>
<td><code>atomic_xchg(&amp;lck-&gt;a, 1)</code>-&gt;1</td>
</tr>
<tr>
<td><code>smp_mb();</code></td>
<td><code>lck-&gt;a</code>-&gt;1</td>
</tr>
<tr>
<td><code>lck-&gt;a</code>-&gt;0</td>
<td><code>lck-&gt;a</code>-&gt;1</td>
</tr>
<tr>
<td>(implicit <code>smp_mb();</code>)</td>
<td><code>lck-&gt;a</code>-&gt;0</td>
</tr>
<tr>
<td><code>atomic_xchg(&amp;lck-&gt;a, 1)</code>-&gt;0</td>
<td>(implicit <code>smp_mb();</code>)</td>
</tr>
<tr>
<td>(critical section)</td>
<td></td>
</tr>
</tbody>
</table>

In this particular case, pairwise memory barriers suffice to keep the two critical sections in place. CPU 2’s `atomic_xchg(&lck->a, 1)` has seen CPU 1’s `lck->a`->0, so therefore everything in CPU 2’s following critical section must see everything that CPU 1’s preceding critical section did. Conversely, CPU 1’s critical section cannot see anything that CPU 2’s critical section will do.

@@@

12.2.6 A Few Simple Rules

@@@

Probably the easiest way to understand memory barriers is to understand a few simple rules:

1. Each CPU sees its own accesses in order.

2. If a single shared variable is loaded and stored by multiple CPUs, then the series of values seen by a given CPU will be consistent with the series seen by the other CPUs, and there will be at least one sequence consisting of all values stored to that variable with which each CPUs series will be consistent.\(^3\)

3. If one CPU does ordered stores to variables A and

\(^3\) A given CPU’s series may of course be incomplete, for example, if a given CPU never loaded or stored the shared variable, then it can have no opinion about that variable’s value.
B, and if a second CPU does ordered loads from B and A, then if the second CPU’s load from B gives the value stored by the first CPU, then the second CPU’s load from A must give the value stored by the first CPU.

4. If one CPU does a load from A ordered before a store to B, and if a second CPU does a load from B ordered before a store from A, and if the second CPU’s load from B gives the value stored by the first CPU, then the first CPU’s load from A must not give the value stored by the second CPU.

5. If one CPU does a load from A ordered before a store to B, and if a second CPU does a store to B ordered before a store from A, and if the first CPU’s load from A gives the value stored by the second CPU, then the second CPU’s store to B must happen after the second CPU’s store to B, hence the value stored by the first CPU persists.

6. So what exactly

12.2.7 Abstract Memory Access Model

Consider the abstract model of the system shown in Figure 12.6.

![Figure 12.6: Abstract Memory Access Model](image)

Each CPU executes a program that generates memory access operations. In the abstract CPU, memory operation ordering is very relaxed, and a CPU may actually perform the memory operations in any order it likes, provided program causality appears to be maintained. Similarly, the compiler may also arrange the instructions it emits in any order it likes, provided it doesn’t affect the apparent operation of the program.

So in the above diagram, the effects of the memory operations performed by a CPU are perceived by the rest of the system as the operations cross the interface between the CPU and rest of the system (the dotted lines).

For example, consider the following sequence of events given the initial values \( \{A = 1, \ B = 2\} \):

<table>
<thead>
<tr>
<th>CPU 1</th>
<th>CPU 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = 3;</td>
<td>x = A2</td>
</tr>
<tr>
<td>B = 4;</td>
<td>y = B4</td>
</tr>
</tbody>
</table>

The set of accesses as seen by the memory system in the middle can be arranged in 24 different combinations, with loads denoted by “ld” and stores denoted by “st”:

- \( st \ A=3, \ st \ B=4, \ x=ld \ A \rightarrow 3, \ y=ld \ B \rightarrow 4 \)
- \( st \ A=3, \ st \ B=4, \ y=ld \ B \rightarrow 4, \ x=ld \ A \rightarrow 3 \)
- \( st \ A=3, \ x=ld \ A \rightarrow 3, \ st \ B=4, \ y=ld \ B \rightarrow 2, \ st \ B=4 \)
- \( st \ A=3, \ y=ld \ B \rightarrow 2, \ st \ B=4, \ x=ld \ A \rightarrow 3 \)
- \( st \ A=3, \ x=ld \ A \rightarrow 3, \ y=ld \ B \rightarrow 2, \ st \ B=4 \)
- \( st \ A=3, \ y=ld \ B \rightarrow 2, \ st \ B=4, \ x=ld \ A \rightarrow 3 \)
- \( st \ B=4, \ st \ A=3, \ x=ld \ A \rightarrow 3, \ y=ld \ B \rightarrow 4 \)
- \( st \ B=4, \ st \ A=3, \ y=ld \ B \rightarrow 2, \ x=ld \ A \rightarrow 3 \)

and can thus result in four different combinations of values:

- \( x == 1, \ y == 2 \)
- \( x == 1, \ y == 4 \)
- \( x == 3, \ y == 2 \)
- \( x == 3, \ y == 4 \)

Furthermore, the stores committed by a CPU to the memory system may not be perceived by the loads made by another CPU in the same order as the stores were committed.

As a further example, consider this sequence of events given the initial values \( \{A = 1, \ B = 2, \ C = 3, \ P = &A, \ Q = &C\} \):

<table>
<thead>
<tr>
<th>CPU 1</th>
<th>CPU 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>B = 4;</td>
<td>P = P1</td>
</tr>
<tr>
<td>P = &amp;B</td>
<td>D = *Q1</td>
</tr>
</tbody>
</table>

There is an obvious data dependency here, as the value loaded into \( D \) depends on the address retrieved from \( P \) by CPU 2. At the end of the sequence, any of the following results are possible:

- \( Q == &A \) and \( (D == 1) \)
- \( Q == &B \) and \( (D == 2) \)
- \( Q == &B \) and \( (D == 4) \)

Note that CPU 2 will never try and load \( C \) into \( D \).
because the CPU will load P into Q before issuing the load of *Q.

12.2.8 Device Operations

Some devices present their control interfaces as collections of memory locations, but the order in which the control registers are accessed is very important. For instance, imagine an Ethernet card with a set of internal registers that are accessed through an address port register (A) and a data port register (D). To read internal register 5, the following code might then be used:

\[
\begin{align*}
\text{A} &= 5; \\
\text{x} &= *\text{D}; \\
\end{align*}
\]

but this might show up as either of the following two sequences:

\[
\begin{align*}
\text{STORE } \text{A} &= 5, \text{x} = \text{LOAD} \text{D} \\
\text{x} &= \text{LOAD} \text{D}, \text{STORE } \text{A} = 5 \\
\end{align*}
\]

the second of which will almost certainly result in a malfunction, since it set the address after attempting to read the register.

12.2.9 Guarantees

There are some minimal guarantees that may be expected of a CPU:

1. On any given CPU, dependent memory accesses will be issued in order, with respect to itself. This means that for:

\[
\begin{align*}
\text{Q} &= \text{P}; \text{D} = *\text{Q}; \\
\end{align*}
\]

the CPU will issue the following memory operations:

\[
\begin{align*}
\text{Q} &= \text{LOAD P, D = LOAD *Q} \\
\end{align*}
\]

and always in that order.

2. Overlapping loads and stores within a particular CPU will appear to be ordered within that CPU. This means that for:

\[
\begin{align*}
\text{a} &= *\text{X}; \text{X} = \text{b}; \\
\end{align*}
\]

the CPU will only issue the following sequence of memory operations:

\[
\begin{align*}
\text{a} &= \text{LOAD } \text{X}, \text{STORE } \text{X} = \text{b}; \\
\end{align*}
\]

And for:

\[
\begin{align*}
\text{X} &= \text{c}; \text{d} = *\text{X}; \\
\end{align*}
\]

the CPU will only issue:

\[
\begin{align*}
\text{STORE } \text{X} &= \text{c, d = LOAD } \text{X} \\
\end{align*}
\]

(Loads and stores overlap if they are targeted at overlapping pieces of memory).

3. A series of stores to a single variable will appear to all CPUs to have occurred in a single order, thought this order might not be predictable from the code, and in fact the order might vary from one run to another.

And there are a number of things that must or must not be assumed:

1. It must not be assumed that independent loads and stores will be issued in the order given. This means that for:

\[
\begin{align*}
\text{x} &= \text{A}; \text{y} = \text{B}; \text{D} = \text{Z}; \\
\end{align*}
\]

we may get any of the following sequences:

\[
\begin{align*}
\text{X} &= \text{LOAD } \text{A}, \text{Y} = \text{LOAD } \text{B}, \text{STORE } \text{D} = \text{Z} \\
\text{X} &= \text{LOAD } \text{A}, \text{STORE } \text{D} = \text{Z}, \text{Y} = \text{LOAD } \text{B} \\
\text{Y} &= \text{LOAD } \text{B}, \text{X} = \text{LOAD } \text{A}, \text{STORE } \text{D} = \text{Z} \\
\text{Y} &= \text{LOAD } \text{B}, \text{STORE } \text{D} = \text{Z}, \text{X} = \text{LOAD } \text{A} \\
\text{STORE } \text{D} &= \text{Z}, \text{X} &= \text{LOAD } \text{A}, \text{Y} = \text{LOAD } \text{B} \\
\text{STORE } \text{D} &= \text{Z}, \text{Y} &= \text{LOAD } \text{B}, \text{X} &= \text{LOAD } \text{A} \\
\end{align*}
\]

2. It must be assumed that overlapping memory accesses may be merged or discarded. This means that for:

\[
\begin{align*}
\text{x} &= \text{A}; \text{y} = \text{(A + 4);} \\
\end{align*}
\]

we may get any one of the following sequences:

\[
\begin{align*}
\text{X} &= \text{LOAD } \text{A}; \text{Y} = \text{LOAD } \text{(A + 4);} \\
\text{Y} &= \text{LOAD } \text{(A + 4);} \text{X} &= \text{LOAD } \text{A}; \\
\{\text{X}, \text{Y}\} &= \text{LOAD } \{\text{A}, \text{(A + 4)}\}; \\
\end{align*}
\]

And for:
we may get either of:

\[
\text{STORE } *A = X; Y = \text{LOAD } *A; \\
\text{STORE } *A = Y = X;
\]

12.2.10 What Are Memory Barriers?

As can be seen above, independent memory operations are effectively performed in random order, but this can be a problem for CPU-CPU interaction and for I/O. What is required is some way of intervening to instruct the compiler and the CPU to restrict the order.

Memory barriers are such interventions. They impose a perceived partial ordering over the memory operations on either side of the barrier.

Such enforcement is important because the CPUs and other devices in a system can use a variety of tricks to improve performance - including reordering, deferral and combination of memory operations; speculative loads; speculative branch prediction and various types of caching. Memory barriers are used to override or suppress these tricks, allowing the code to sanely control the interaction of multiple CPUs and/or devices.

12.2.10.1 Explicit Memory Barriers

Memory barriers come in four basic varieties:

1. Write (or store) memory barriers,
2. Data dependency barriers,
3. Read (or load) memory barriers, and
4. General memory barriers.

Each variety is described below.

**Write Memory Barriers** A write memory barrier gives a guarantee that all the STORE operations specified before the barrier will appear to happen before all the STORE operations specified after the barrier with respect to the other components of the system.

A write barrier is a partial ordering on stores only; it is not required to have any effect on loads.

A CPU can be viewed as committing a sequence of store operations to the memory system as time progresses.

All stores before a write barrier will occur in the sequence before all the stores after the write barrier.

Note that write barriers should normally be paired with read or data dependency barriers; see the "SMP barrier pairing" subsection.

**Data Dependency Barriers** A data dependency barrier is a weaker form of read barrier. In the case where two loads are performed such that the second depends on the result of the first (e.g., the first load retrieves the address to which the second load will be directed), a data dependency barrier would be required to make sure that the target of the second load is updated before the address obtained by the first load is accessed.

A data dependency barrier is a partial ordering on interdependent loads only; it is not required to have any effect on stores, independent loads or overlapping loads.

As mentioned for write memory barriers, the other CPUs in the system can be viewed as committing sequences of stores to the memory system that the CPU being considered can then perceive. A data dependency barrier issued by the CPU under consideration guarantees that for any load preceding it, if that load touches one of a sequence of stores from another CPU, then by the time the barrier completes, the effects of all the stores prior to that touched by the load will be perceptible to any loads issued after the data dependency barrier.

See the "Examples of memory barrier sequences" subsection for diagrams showing the ordering constraints.

Note that the first load really has to have a data dependency and not a control dependency. If the address for the second load is dependent on the first load, but the dependency is through a conditional rather than actually loading the address itself, then it’s a control dependency and a full read barrier or better is required. See the "Control dependencies" subsection for more information.

Note that data dependency barriers should normally be paired with write barriers; see the "SMP barrier pairing" subsection.

**Read Memory Barriers** A read barrier is a data dependency barrier plus a guarantee that all the LOAD operations specified before the barrier will appear to happen before all the LOAD operations specified after the barrier with respect to the other components of the system.

A read barrier is a partial ordering on loads only; it is not required to have any effect on stores.

Read memory barriers imply data dependency barriers, and so can substitute for them.
† Note that read barriers should normally be paired with write barriers; see the "SMP barrier pairing" subsection.

**General Memory Barriers**
A general memory barrier gives a guarantee that all the LOAD and STORE operations specified before the barrier will appear to happen before all the LOAD and STORE operations specified after the barrier with respect to the other components of the system.

A general memory barrier is a partial ordering over both loads and stores.

General memory barriers imply both read and write memory barriers, and so can substitute for either.

**12.2.10.2 Implicit Memory Barriers**
There are a couple of types of implicit memory barriers, so called because they are embedded into locking primitives:

1. LOCK operations and
2. UNLOCK operations.

**LOCK Operations**
A lock operation acts as a one-way permeable barrier. It guarantees that all memory operations after the LOCK operation will appear to happen after the LOCK operation with respect to the other components of the system.

Memory operations that occur before a LOCK operation may appear to happen after it completes.

A LOCK operation should almost always be paired with an UNLOCK operation.

**UNLOCK Operations**
Unlock operations also act as a one-way permeable barrier. It guarantees that all memory operations before the UNLOCK operation will appear to happen before the UNLOCK operation with respect to the other components of the system.

Memory operations that occur after an UNLOCK operation may appear to happen before it completes.

LOCK and UNLOCK operations are guaranteed to appear with respect to each other strictly in the order specified.

The use of LOCK and UNLOCK operations generally precludes the need for other sorts of memory barrier (but note the exceptions mentioned in the subsection "MMIO write barrier").

**Quick Quiz 12.10:** What effect does the following sequence have on the order of stores to variables “a” and “b”?

```c
a = 1;
b = 1;
<write barrier>
```

**12.2.10.3 What May Not Be Assumed About Memory Barriers?**
There are certain things that memory barriers cannot guarantee outside of the confines of a given architecture:

1. There is no guarantee that any of the memory accesses specified before a memory barrier will be complete by the completion of a memory barrier instruction; the barrier can be considered to draw a line in that CPU’s access queue that accesses of the appropriate type may not cross.

2. There is no guarantee that issuing a memory barrier on one CPU will have any direct effect on another CPU or any other hardware in the system. The indirect effect will be the order in which the second CPU sees the effects of the first CPU’s accesses occur, but see the next point.

3. There is no guarantee that a CPU will see the correct order of effects from a second CPU’s accesses, even if the second CPU uses a memory barrier, unless the first CPU also uses a matching memory barrier (see the subsection on “SMP Barrier Pairing”).

4. There is no guarantee that some intervening piece of off-the-CPU hardware will not reorder the memory accesses. CPU cache coherency mechanisms should propagate the indirect effects of a memory barrier between CPUs, but might not do so in order.

**12.2.10.4 Data Dependency Barriers**
The usage requirements of data dependency barriers are a little subtle, and it’s not always obvious that they’re needed. To illustrate, consider the following sequence of events, with initial values \( \{ A = 1, B = 2, C = 3, P = &A, Q = &C \} \):

```
CPU 1
B = 4;
<write barrier>
P = &B;
Q = P;
D = *Q;
```

```
CPU 2
```

**7** This is of concern primarily in operating-system kernels. For more information on hardware operations and memory ordering, see the files pci.txt, DMA-API-HOWTO.txt, and DMA-API.txt in the Documentation directory in the Linux source tree [Tor03c].
There’s a clear data dependency here, and it would seem intuitively obvious that by the end of the sequence, \( Q \) must be either \&A or \&B, and that:

- \((Q == \&A) \implies (D == 1)\)
- \((Q == \&B) \implies (D == 4)\)

Counter-intuitive though it might be, it is quite possible that CPU 2’s perception of \( P \) might be updated before its perception of \( B \), thus leading to the following situation:

\((Q == \&B) \text{ and } (D == 2) ????

Whilst this may seem like a failure of coherency or causality maintenance, it isn’t, and this behaviour can be observed on certain real CPUs (such as the DEC Alpha).

To deal with this, a data dependency barrier must be inserted between the address load and the data load (again with initial values of \([A = 1, B = 2, C = 3, P = \&A, Q = \&C]\)):

```
CPU 1:  B = 4;
        <write barrier>
        P = \&B;
        Q = P;
        <data dependency barrier>
        D = *Q;
CPU 2:  
```

This enforces the occurrence of one of the two implications, and prevents the third possibility from arising.

Note that this extremely counterintuitive situation arises most easily on machines with split caches, so that, for example, one cache bank processes even-numbered cache lines and the other bank processes odd-numbered cache lines. The pointer \( P \) might be stored in an odd-numbered cache line, and the variable \( B \) might be stored in an even-numbered cache line. Then, if the even-numbered bank of the reading CPU’s cache is extremely busy while the odd-numbered bank is idle, one can see the new value of the pointer \( P \) (which is \&B), but the old value of the variable \( B \) (which is 1).

Another example of where data dependency barriers might by required is where a number is read from memory and then used to calculate the index for an array access:

```
CPU 1:  M[1] = 4;
        <write barrier>
        P = 1;
        Q = P;
        <data dependency barrier>
        D = M[Q];
CPU 2:  
```

The data dependency barrier is very important to the Linux kernel’s RCU system, for example, see `rcu_dereference()` in `include/linux/rcupdate.h`. This permits the current target of an RCU’d pointer to be replaced with a new modified target, without the replacement target appearing to be incompletely initialised.

See also the subsection on "Cache Coherency" for a more thorough example.

12.2.10.5 Control Dependencies

A control dependency requires a full read memory barrier, not simply a data dependency barrier to make it work correctly. Consider the following bit of code:

```
1 q = \&a;
2 if (p)
3  q = \&b;
4 <data dependency barrier>
5  x = *q;
```

This will not have the desired effect because there is no actual data dependency, but rather a control dependency that the CPU may short-circuit by attempting to predict the outcome in advance. In such a case what’s actually required is:

```
1 q = \&a;
2 if (p)
3  q = \&b;
4 <read barrier>
5  x = *q;
```

12.2.10.6 SMP Barrier Pairing

When dealing with CPU-CPU interactions, certain types of memory barrier should always be paired. A lack of appropriate pairing is almost certainly an error.

A write barrier should always be paired with a data dependency barrier or read barrier, though a general barrier would also be viable. Similarly a read barrier or a data dependency barrier should always be paired with at least an write barrier, though, again, a general barrier is viable:

```
CPU 1:  A = 1;
        <write barrier>
        B = 2;
        <read barrier>
        X = B;
        Y = A;
CPU 2:  
```

Or:

```
CPU 1:  A = 1;
        <write barrier>
        B = \&A;
        <data dependency barrier>
        X = B;
        Y = *X;
CPU 2:  
```
12.2. MEMORY BARRIERS

One way or another, the read barrier must always be present, even though it might be of a weaker type.\(^8\)

Note that the stores before the write barrier would normally be expected to match the loads after the read barrier or data dependency barrier, and vice versa:

\[
\begin{align*}
\text{CPU 1} & \quad \text{CPU 2} \\
a = 1; & \quad y = c; \\
b = 2; \quad w = d; & \quad \text{<read barrier>} \\
\text{<write barrier> } & \quad \text{<read barrier>} \\
c = 3; & \quad x = a; \\
d = 4; & \quad y = b; \\
\end{align*}
\]

12.2.10.7 Examples of Memory Barrier Pairings

Firstly, write barriers act as a partial orderings on store operations. Consider the following sequence of events:

\[
\begin{align*}
\text{STORE A} & = 1 \\
\text{STORE B} & = 2 \\
\text{STORE C} & = 3 \\
\text{<write barrier>} \\
\text{STORE D} & = 4 \\
\text{STORE E} & = 5 \\
\end{align*}
\]

This sequence of events is committed to the memory coherence system in an order that the rest of the system might perceive as the unordered set of \(\{A=1, B=2, C=3\}\) all occurring before the unordered set of \(\{D=4, E=5\}\), as shown in Figure 12.7.

Secondly, data dependency barriers act as a partial orderings on data-dependent loads. Consider the following sequence of events with initial values \(\{B = 7, X = 9, Y = 8, C = &Y\}\):

\[
\begin{align*}
\text{CPU 1} & \quad \text{CPU 2} \\
A = 1; & \quad \text{LOAD X} \\
B = 2; & \quad \text{LOAD C (gets &B)} \\
\text{<write barrier>} & \quad \text{<data dependency barrier>} \\
C = &B; & \quad \text{LOAD *C (reads B)} \\
D = 4; & \quad \text{LOAD A} \\
\end{align*}
\]

Without intervention, CPU 2 may then choose to perceive the events on CPU 1 in some effectively random order, despite the write barrier issued by CPU 1, as shown in Figure 12.8.

If, however, a read barrier were to be placed between the load of \(B\) and the load of \(*C\) on CPU 2, again with initial values of \(\{A = 0, B = 9\}\):

\[
\begin{align*}
\text{CPU 1} & \quad \text{CPU 2} \\
A = 1; & \quad \text{LOAD B} \\
\text{<write barrier>} & \quad \text{<read barrier>} \\
B = 2; & \quad \text{LOAD A} \\
\end{align*}
\]

then ordering will be as intuitively expected, as shown in Figure 12.9.

And thirdly, a read barrier acts as a partial order on loads. Consider the following sequence of events, with initial values \(\{A = 0, B = 9\}\):

\[
\begin{align*}
\text{CPU 1} & \quad \text{CPU 2} \\
A = 1; & \quad \text{LOAD B} \\
\text{<write barrier>} & \quad \text{LOAD A} \\
B = 2; & \quad \text{LOAD A (1st)} \\
\text{<read barrier>} & \quad \text{LOAD A (2nd)} \\
\end{align*}
\]

To illustrate this more completely, consider what could happen if the code contained a load of \(A\) either side of the read barrier, once again with the same initial values of \(\{A = 0, B = 9\}\):

\[
\begin{align*}
\text{CPU 1} & \quad \text{CPU 2} \\
A = 1; & \quad \text{LOAD B} \\
\text{<write barrier>} & \quad \text{<read barrier>} \\
B = 2; & \quad \text{LOAD A} \\
\end{align*}
\]

Even though the two loads of \(A\) both occur after the load of \(B\), they may both come up with different values, as shown in Figure 12.12.

Of course, it may well be that CPU 1’s update to \(A\) becomes perceptible to CPU 2 before the read barrier completes, as shown in Figure 12.13.

The guarantee is that the second load will always come

---

\(^8\) By “weaker”, we mean “makes fewer ordering guarantees”. A weaker barrier is usually also lower-overhead than is a stronger barrier.
12.2.10.8 Read Memory Barriers vs. Load Speculation

Many CPUs speculate with loads: that is, they see that they will need to load an item from memory, and they find a time where they’re not using the bus for any other loads, and then do the load in advance — even though they haven’t actually got to that point in the instruction execution flow yet. Later on, this potentially permits the actual load instruction to complete immediately because the CPU already has the value on hand.

It may turn out that the CPU didn’t actually need the value (perhaps because a branch circumvented the load) in which case it can discard the value or just cache it for later use. For example, consider the following:

<table>
<thead>
<tr>
<th>CPU 1</th>
<th>CPU 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD B</td>
<td>DIVIDE</td>
</tr>
<tr>
<td>DIVIDE</td>
<td>LOAD A</td>
</tr>
</tbody>
</table>

On some CPUs, divide instructions can take a long time to complete, which means that CPU 2’s bus might go idle during that time. CPU 2 might therefore speculatively load A before the divides complete. In the (hopefully) unlikely event of an exception from one of the dividees, this speculative load will have been wasted, but in the (again, hopefully) common case, overlapping the load
with the divides will permit the load to complete more quickly, as illustrated by Figure 12.14.

Placing a read barrier or a data dependency barrier just before the second load:

```
CPU 1    CPU 2
LOAD B   <read barrier>
DIVIDE   LOAD A
```

will force any value speculatively obtained to be reconsidered to an extent dependent on the type of barrier used. If there was no change made to the speculated memory location, then the speculated value will just be used, as shown in Figure 12.15. On the other hand, if there was an update or invalidation to \( \text{A} \) from some other CPU, then the speculation will be cancelled and the value of \( \text{A} \) will be reloaded, as shown in Figure 12.16.

### 12.2.11 Locking Constraints

As noted earlier, locking primitives contain implicit memory barriers. These implicit memory barriers provide the following guarantees:

1. **LOCK operation guarantee:**
   - Memory operations issued after the LOCK will be completed after the LOCK operation has completed.
   - Memory operations issued before the LOCK may be completed after the LOCK operation has completed.

2. **UNLOCK operation guarantee:**
   - Memory operations issued before the UNLOCK will be completed before the UNLOCK
At this point the read barrier causes all effects prior to the storage of B to be perceptible to CPU 2.

Figure 12.11: Read Barrier Supplied

At this point the read barrier causes all effects prior to the storage of B to be perceptible to CPU 2.

Figure 12.12: Read Barrier Supplied, Double Load

operation has completed.

- Memory operations issued after the UNLOCK may be completed before the UNLOCK operation has completed.

3. LOCK vs LOCK guarantee:

- All LOCK operations issued before another LOCK operation will be completed before that LOCK operation.

4. LOCK vs UNLOCK guarantee:

- All LOCK operations issued before an UNLOCK operation will be completed before the UNLOCK operation.
- All UNLOCK operations issued before a LOCK operation will be completed before the LOCK operation.

5. Failed conditional LOCK guarantee:

- Certain variants of the LOCK operation may fail, either due to being unable to get the lock immediately, or due to receiving an unblocked signal or exception whilst asleep waiting for the lock to become available. Failed locks do not imply any sort of barrier.

12.2.12 Memory-Barrier Examples

12.2.12.1 Locking Examples

**LOCK Followed by UNLOCK:** A LOCK followed by an UNLOCK may not be assumed to be a full memory barrier because it is possible for an access preceding the LOCK to happen after the LOCK, and an access following the UNLOCK to happen before the UNLOCK, and the two accesses can themselves then cross. For example, the following:
12.2 MEMORY BARRIERS

Figure 12.13: Read Barrier Supplied, Take Two

Once the divisions are complete the CPU can then perform the LOAD with immediate effect

DIVIDE
CPU 2
CPU 1

Figure 12.14: Speculative Load

might well execute in the following order:

LOCK
UNLOCK
might well execute in the following order:

1 *A = a;
2 LOCK
3 *B = b;
4 UNLOCK

Again, always remember that both LOCK and UNLOCK are permitted to let preceding operations “bleed in” to the critical section.

Quick Quiz 12.11: What sequence of LOCK-UNLOCK operations would act as a full memory barrier?

Quick Quiz 12.12: What (if any) CPUs have memory-barrier instructions from which these semi-permeable locking primitives might be constructed?

LOCK-Based Critical Sections: Although a LOCK-UNLOCK pair does not act as a full memory barrier, these operations do affect memory ordering.

Consider the following code:

1 *A = a;
2 *B = b;
3 LOCK
4 *C = c;
5 *D = d;
6 UNLOCK
7 *E = e;
8 *F = f;

This could legitimately execute in the following order, where pairs of operations on the same line indicate that the CPU executed those operations concurrently:

LOCK
1 *A = a;
2 *B = b;
3 LOCK
4 *C = c;
5 *D = d;
6 UNLOCK
7 *E = e;
8 *F = f;

Quick Quiz 12.13: Given that operations grouped in curly braces are executed concurrently, which of the rows of Table 12.2 are legitimate reorderings of the assignments to variables “A” through “F” and the LOCK/UNLOCK
CHAPTER 12. ADVANCED SYNCHRONIZATION

The CPU being busy doing a division speculates on the LOAD of A

Figure 12.15: Speculative Load and Barrier

The CPU being busy doing a division speculates on the LOAD of A

Figure 12.16: Speculative Load Cancelled by Barrier

<table>
<thead>
<tr>
<th>#</th>
<th>Ordering: legitimate or not?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>*A; *B; LOCK; *C; *D; UNLOCK; *E; *F;</td>
</tr>
<tr>
<td>2</td>
<td>*A; /*B; LOCK; */C; *D; UNLOCK; *E; *F;</td>
</tr>
<tr>
<td>3</td>
<td>*F; *A; *B; LOCK; *C; *D; UNLOCK; *E; *F;</td>
</tr>
<tr>
<td>4</td>
<td>*B; /*C; *D; *A; UNLOCK; *E; *F;</td>
</tr>
<tr>
<td>5</td>
<td>*A; /*B; LOCK; */C; *D; UNLOCK; *E; *F;</td>
</tr>
<tr>
<td>6</td>
<td>*A; /*B; LOCK; */C; *D; UNLOCK; *E; *F;</td>
</tr>
<tr>
<td>7</td>
<td>*A; /*B; LOCK; */C; *D; UNLOCK; *E; *F;</td>
</tr>
<tr>
<td>8</td>
<td>*B; /*A; LOCK; */C; *D; UNLOCK; *F; /*E;</td>
</tr>
<tr>
<td>9</td>
<td>*F; /*C; *D; /*A; UNLOCK; *F;</td>
</tr>
</tbody>
</table>

Table 12.2: Lock-Based Critical Sections

Ordering with Multiple Locks: Code containing multiple locks still sees ordering constraints from those locks, but one must be careful to keep track of which lock is which. For example, consider the code shown in Table 12.3, which uses a pair of locks named “M” and “Q”.

In this example, there are no guarantees as to what order the assignments to variables “A” through “H” will appear in, other than the constraints imposed by the locks themselves, as described in the previous section.

Quick Quiz 12.14: What are the constraints for Table 12.3? ■

Ordering with Multiple CPUs on One Lock: Suppose, instead of the two different locks as shown in Table 12.3, both CPUs acquire the same lock, as shown in Table 12.4?

In this case, either CPU 1 acquires M before CPU 2 does, or vice versa. In the first case, the assignments to A, B, and C must precede those to F, G, and H. On the other hand, if CPU 2 acquires the lock first, then the assignments to E, F, and G must precede those to B, C,
12.2. MEMORY BARRIERS

<table>
<thead>
<tr>
<th>CPU 1</th>
<th>CPU 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = a;</td>
<td>E = e;</td>
</tr>
<tr>
<td>LOCK M;</td>
<td>LOCK M;</td>
</tr>
<tr>
<td>B = b;</td>
<td>F = f;</td>
</tr>
<tr>
<td>C = c;</td>
<td>G = g;</td>
</tr>
<tr>
<td>UNLOCK M;</td>
<td>UNLOCK M;</td>
</tr>
<tr>
<td>D = d;</td>
<td>H = h;</td>
</tr>
</tbody>
</table>

Table 12.4: Ordering With Multiple CPUs on One Lock

12.2.13 The Effects of the CPU Cache

The perceived ordering of memory operations is affected by the caches that lie between the CPUs and memory, as well as by the cache coherence protocol that maintains memory consistency and ordering. From a software viewpoint, these caches are for all intents and purposes part of memory. Memory barriers can be thought of as acting on the vertical dotted line in Figure 12.17, ensuring that the CPU presents its values to memory in the proper order, as well as ensuring that it sees changes made by other CPUs in the proper order.

Although the caches can “hide” a given CPU’s memory accesses from the rest of the system, the cache-coherence protocol ensures that all other CPUs see any effects of these hidden accesses, migrating and invalidating cache-lines as required. Furthermore, the CPU core may execute instructions in any order, restricted only by the requirement that program causality and memory ordering appear to be maintained. Some of these instructions may generate memory accesses that must be queued in the CPU’s memory access queue, but execution may nonetheless continue until the CPU either fills up its internal resources or until it must wait for some queued memory access to complete.

12.2.13.1 Cache Coherency

Although cache-coherence protocols guarantee that a given CPU sees its own accesses in order, and that all CPUs agree on the order of modifications to a single variable contained within a single cache line, there is no guarantee that modifications to different variables will be seen in the same order by all CPUs — although some computer systems do make some such guarantees, portable software cannot rely on them.

To see why reordering can occur, consider the two-CPU system shown in Figure 12.18, in which each CPU has a split cache. This system has the following properties:

1. An odd-numbered cache line may be in cache A, cache C, in memory, or some combination of the above.
2. An even-numbered cache line may be in cache B, cache D, in memory, or some combination of the above.
3. While the CPU core is interrogating one of its caches, its other cache is not necessarily quiescent. This other cache may instead be responding to an invalidation request, writing back a dirty cache line, processing elements in the CPU’s memory-access queue, and so on.
4. Each cache has queues of operations that need to be applied to that cache in order to maintain the required coherence and ordering properties.
5. These queues are not necessarily flushed by loads from or stores to cache lines affected by entries in those queues.

In short, if cache A is busy, but cache B is idle, then CPU 1’s stores to odd-numbered cache lines may be delayed compared to CPU 2’s stores to even-numbered cache lines. In not-so-extreme cases, CPU 2 may see CPU 1’s operations out of order.

Much more detail on memory ordering in hardware and software may be found in Appendix C.

12.2.14 Where Are Memory Barriers Needed?

Memory barriers are only required where there’s a possibility of interaction between two CPUs or between a CPU and a device. If it can be guaranteed that there won’t be any such interaction in any particular piece of code, then memory barriers are unnecessary in that piece of code.

Note that these are the minimum guarantees. Different architectures may give more substantial guarantees, as discussed in Appendix C, but they may not be relied upon outside of code specifically designed to run only on the corresponding architecture.

However, primitives that implement atomic operations, such as locking primitives and atomic data-structure manipulation and traversal primitives, will normally include any needed memory barriers in their definitions. However,

---

9 But note that in “superscalar” systems, the CPU might well be accessing both halves of its cache at once, and might in fact be performing multiple concurrent accesses to each of the halves.
there are some exceptions, such as `atomic_inc()` in the Linux kernel, so be sure to review the documentation, and, if possible, the actual implementations, for your software environment.

One final word of advice: use of raw memory-barrier primitives should be a last resort. It is almost always better to use an existing primitive that takes care of memory barriers.

12.3 Non-Blocking Synchronization

12.3.1 Simple NBS

12.3.2 Hazard Pointers

`@@@ combination of hazard pointers and RCU to eliminate memory barriers?`

12.3.3 Atomic Data Structures

Queues and stacks — avoiding full-race non-blocking properties often yields great simplifications.
12.3.4 “Macho” NBS

Cite Herlihy and his crowd.
- Describe constraints (X-freedom, linearizability, ...)
and show examples breaking them.
“Creating a perfect API is like committing the perfect crime. There are at least fifty things that can go wrong, and if you are a genius, you might be able to anticipate twenty-five of them.”

13.1 Rusty Scale for API Design

1. It is impossible to get wrong. dwim()
2. The compiler or linker won’t let you get it wrong.
3. The compiler or linker will warn you if you get it wrong.
4. The simplest use is the correct one.
5. The name tells you how to use it.
6. Do it right or it will always break at runtime.
7. Follow common convention and you will get it right. malloc()
8. Read the documentation and you will get it right.
9. Read the implementation and you will get it right.
10. Read the right mailing-list archive and you will get it right.
11. Read the right mailing-list archive and you will get it wrong.
12. Read the implementation and you will get it wrong. The non-CONFIG_PREEMPT implementation of rcu_read_lock().
13. Read the documentation and you will get it wrong. DEC Alpha wmb instruction.
14. Follow common convention and you will get it wrong. printf() (failing to check for error return).
15. Do it right and it will break at runtime.
16. The name tells you how not to use it.
17. The obvious use is wrong. smp_mb().
18. The compiler or linker will warn you if you get it right.
19. The compiler or linker won’t let you get it right.
20. It is impossible to get right. gets().

13.2 Shaving the Mandelbrot Set

The set of useful programs resembles the Mandelbrot set (shown in Figure 13.1) in that it does not have a clear-cut smooth boundary — if it did, the halting problem would be solvable. But we need APIs that real people can use, not ones that require a Ph.D. dissertation be completed for each and every potential use. So, we “shave the Mandelbrot set”,\(^1\) restricting the use of the API to an easily described subset of the full set of potential uses.

Such shaving may seem counterproductive. After all, if an algorithm works, why shouldn’t it be used?

To see why at least some shaving is absolutely necessary, consider a locking design that avoids deadlock, but in perhaps the worst possible way. This design uses a circular doubly linked list, which contains one element for each thread in the system along with a header element. When a new thread is spawned, the parent thread must insert a new element into this list, which requires some sort of synchronization.

\(^1\) Due to Josh Triplett.
One way to protect the list is to use a global lock. However, this might be a bottleneck if threads were being created and deleted frequently. Another approach would be to use a hash table and to lock the individual hash buckets, but this can perform poorly when scanning the list in order.

A third approach is to lock the individual list elements, and to require the locks for both the predecessor and successor to be held during the insertion. Since both locks must be acquired, we need to decide which order to acquire them in. Two conventional approaches would be to acquire the locks in address order, or to acquire them in the order that they appear in the list, so that the header is always acquired first when it is one of the two elements being locked. However, both of these methods require special checks and branches.

The to-be-shaven solution is to unconditionally acquire the locks in list order. But what about deadlock?

Deadlock cannot occur.

To see this, number the elements in the list starting with zero for the header up to \( N \) for the last element in the list (the one preceding the header, given that the list is circular). Similarly, number the threads from zero to \( N - 1 \). If each thread attempts to lock some consecutive pair of elements, at least one of the threads is guaranteed to be able to acquire both locks.

Why?

Because there are not enough threads to reach all the way around the list. Suppose thread 0 acquires element 0’s lock. To be blocked, some other thread must have already acquired element 1’s lock, so let us assume that thread 1 has done so. Similarly, for thread 1 to be blocked, some other thread must have acquired element 2’s lock, and so on, up through thread \( N - 1 \), who acquires element \( N - 1 \)’s lock. For thread \( N - 1 \) to be blocked, some other thread must have acquired element \( N \)’s lock. But there are no more threads, and so thread \( N - 1 \) cannot be blocked. Therefore, deadlock cannot occur.

So why should we prohibit use of this delightful little algorithm?

The fact is that if you really want to use it, we cannot stop you. We can, however, recommend against such code being included in any project that we care about.

But, before you use this algorithm, please think through the following Quick Quiz.

**Quick Quiz 13.1:** Can a similar algorithm be used when deleting elements?

The fact is that this algorithm is extremely specialized (it only works on certain sized lists), and also quite fragile. Any bug that accidentally failed to add a node to the list could result in deadlock. In fact, simply adding the node a bit too late could result in deadlock.

In addition, the other algorithms described above are “good and sufficient”. For example, simply acquiring the locks in address order is fairly simple and quick, while allowing the use of lists of any size. Just be careful of the special cases presented by empty lists and lists containing only one element!

**Quick Quiz 13.2:** Yetch! What ever possessed someone to come up with an algorithm that deserves to be shaved as much as this one does???

In summary, we do not use algorithms simply because they happen to work. We instead restrict ourselves to algorithms that are useful enough to make it worthwhile learning about them. The more difficult and complex the algorithm, the more generally useful it must be in order for the pain of learning it and fixing its bugs to be worthwhile.

**Quick Quiz 13.3:** Give an exception to this rule.

Exceptions aside, we must continue to shave the software “Mandelbrot set” so that our programs remain maintainable, as shown in Figure 13.2.

---

\( ^2 \) Those of you with strong operating-system backgrounds, please suspend disbelief. If you are unable to suspend disbelief, send us a better example.
13.2. SHAVING THE MANDELBROT SET

Figure 13.2: Shaving the Mandelbrot Set
Chapter 14

Time Management

Scheduling ticks
Tickless operation
Timers
Current time, monotonic operation
The many ways in which time can appear to go backwards
Causality, the only real time in SMP (or distributed) systems
Chapter 15

Conflicting Visions of the Future

This chapter presents some conflicting visions of the future of parallel programming. It is not clear which of these will come to pass, in fact, it is not clear that any of them will. They are nevertheless important because each vision has its devoted adherents, and if enough people believe in something fervently enough, you will need to deal with at least the shadow of that thing’s existence in the form of its influence on the thoughts, words, and deeds of its adherents. Besides which, it is entirely possible that one or more of these visions will actually come to pass. But most are bogus. Tell which is which and you’ll be rich [Spi77]!

Therefore, the following sections give an overview of transactional memory, shared-memory parallel functional programming, and process-based parallel functional programming. But first, a cautionary tale on prognostication taken from the early 2000s.

15.1 The Future of CPU Technology Ain’t What it Used to Be

Years past always seem so simple and innocent when viewed through the lens of many years of experience. And the early 2000s were for the most part innocent of the impending failure of Moore’s Law to continue delivering the then-traditional increases in CPU clock frequency. Oh, there were the occasional warnings about the limits of technology, but such warnings had been sounded for decades. With that in mind, consider the following scenarios:

1. Uniprocessor Über Alles (Figure 15.1),
2. Multithreaded Mania (Figure 15.2),
3. More of the Same (Figure 15.3), and
4. Crash Dummies Slamming into the Memory Wall (Figure 15.4).

Each of these scenarios are covered in the following sections, first with a quote from a 2004 source [McK04].

15.1.1 Uniprocessor Über Alles

In this scenario, the combination of Moore’s-Law increases in CPU clock rate and continued progress in horizontally scaled computing render SMMP systems irrelevant. This scenario is therefore dubbed “Uniprocessor Über Alles”, literally, uniprocessors above all else.

These uniprocessor systems would be subject only to instruction overhead, since memory bar-
riers, cache thrashing, and contention do not affect single-CPU systems. In this scenario, RCU is useful only for niche applications, such as interacting with NMIs. It is not clear that an operating system lacking RCU would see the need to adopt it, although operating systems that already implement RCU might continue to do so.

However, recent progress with multithreaded CPUs seems to indicate that this scenario is quite unlikely.

However, recent progress with multithreaded CPUs seems to indicate that this scenario is quite unlikely.

Unlikely indeed! But the larger software community was reluctant to accept the fact that they would need to embrace parallelism, and so it was some time before this community concluded that the “free lunch” of Moore’s-Law-induced CPU core-clock frequency increases was well and truly finished. Never forget: belief is an emotion, not necessarily the result of a rational technical thought process!

### 15.1.2 Multithreaded Mania

A less-extreme variant of Uniprocessor Über Alles features uniprocessors with hardware multithreading, and in fact multithreaded CPUs are now standard for many desktop and laptop computer systems. The most aggressively multithreaded CPUs share all levels of cache hierarchy, thereby eliminating CPU-to-CPU memory latency, in turn greatly reducing the performance penalty for traditional synchronization mechanisms. However, a multithreaded CPU would still incur overhead due to contention and to pipeline stalls caused by memory barriers. Furthermore, because all hardware threads share all levels of cache, the cache available to a given hardware thread is a fraction of what it would be on an equivalent single-threaded CPU, which can degrade performance for applications with large cache footprints. There is also some possibility that the restricted amount of cache available will cause RCU-based algo-
15.1. THE FUTURE OF CPU TECHNOLOGY AIN’T WHAT IT USED TO BE

Algorithms to incur performance penalties due to their grace-period-induced additional memory consumption. Investigating this possibility is future work.

However, in order to avoid such performance degradation, a number of multithreaded CPUs and multi-CPU chips partition at least some of the levels of cache on a per-hardware-thread basis. This increases the amount of cache available to each hardware thread, but re-introduces memory latency for cachelines that are passed from one hardware thread to another.

And we all know how this story has played out, with multiple multi-threaded cores on a single die plugged into a single socket. The question then becomes whether or not future shared-memory systems will always fit into a single socket.

15.1.3 More of the Same

The More-of-the-Same scenario assumes that the memory-latency ratios will remain roughly where they are today.

This scenario actually represents a change, since to have more of the same, interconnect performance must begin keeping up with the Moore’s-Law increases in core CPU performance. In this scenario, overhead due to pipeline stalls, memory latency, and contention remains significant, and RCU retains the high level of applicability that it enjoys today.

And the change has been the ever-increasing levels of integration that Moore’s Law is still providing. But longer term, which will it be? More CPUs per die? Or more I/O, cache, and memory?

Servers seem to be choosing the former, while embedded systems on a chip (SoCs) continue choosing the latter.

15.1.4 Crash Dummies Slamming into the Memory Wall

If the memory-latency trends shown in Figure 15.5 continue, then memory latency will continue to grow relative to instruction-execution overhead. Systems such as Linux that have significant use of RCU will find additional
use of RCU to be profitable, as shown in Figure 15.6. As can be seen in this figure, if RCU is heavily used, increasing memory-latency ratios give RCU an increasing advantage over other synchronization mechanisms. In contrast, systems with minor use of RCU will require increasingly high degrees of read intensity for use of RCU to pay off, as shown in Figure 15.7. As can be seen in this figure, if RCU is lightly used, increasing memory-latency ratios put RCU at an increasing disadvantage compared to other synchronization mechanisms. Since Linux has been observed with over 1,600 callbacks per grace period under heavy load [SM04], it seems safe to say that Linux falls into the former category.

On the one hand, this passage failed to anticipate the cache-warmth issues that RCU can suffer from in workloads with significant update intensity, in part because it seemed unlikely that RCU would really be used in such cases. In the event, the SLAB_DESTROY_BY_RCU has been pressed into service in a number of instances where these cache-warmth issues would otherwise be problematic, as has sequence locking. On the other hand, this passage also failed to anticipate that RCU would be used to reduce scheduling latency or for security.

In short, beware of prognostications, including those in the remainder of this chapter.

15.2 Transactional Memory

The idea of using transactions outside of databases goes back many decades [Lom77], with the key difference between database and non-database transactions being that non-database transactions drop the “D” in the “ACID” properties defining database transactions. The idea of supporting memory-based transactions, or “transactional memory” (TM), in hardware is more recent [HM93], but unfortunately, support for such transactions in commodity hardware was not immediately forthcoming, despite other somewhat similar proposals being put forward [SSHT93]. Not long after, Shavit and Touitou proposed a software-only implementation of transactional memory (STM) that was capable of running on commodity hardware, give or take memory-ordering issues. This proposal languished for many years, perhaps due to the fact that the research community’s attention was absorbed by non-blocking synchronization (see Section 12.3).

But by the turn of the century, TM started receiving more attention [MT01, RG01], and by the middle of the decade, the level of interest can only be termed “incandescent” [Her05, Gro07], despite a few voices of caution [BLM05, MMW07].

The basic idea behind TM is to execute a section of code atomically, so that other threads see no intermediate state. As such, the semantics of TM could be implemented by simply replacing each transaction with a recursively acquirable global lock acquisition and release, albeit with abysmal performance and scalability. Much of the complexity inherent in TM implementations, whether hardware or software, is efficiently detecting when concurrent transactions can safely run in parallel. Because this detection is done dynamically, conflicting transactions can be aborted or “rolled back”, and in some implementations, this failure mode is visible to the programmer.

Because transaction roll-back is increasingly unlikely as transaction size decreases, TM might become quite attractive for small memory-based operations, such as linked-list manipulations used for stacks, queues, hash tables, and search trees. However, it is currently much more difficult to make the case for large transactions, particularly those containing non-memory operations such as I/O and process creation. The following sections look at current challenges to the grand vision of “Transactional Memory Everywhere” [McK09d].
15.2. TRANSACTIONAL MEMORY

15.2.1 I/O Operations

One can execute I/O operations within a lock-based critical section, and, at least in principle, from within an RCU read-side critical section. What happens when you attempt to execute an I/O operation from within a transaction?

The underlying problem is that transactions may be rolled back, for example, due to conflicts. Roughly speaking, this requires that all operations within any given transaction be idempotent, so that executing the operation twice has the same effect as executing it once. Unfortunately, I/O is in general the prototypical non-idempotent operation, making it difficult to include general I/O operations in transactions.

Here are some options for handling of I/O within transactions:

1. Restrict I/O within transactions to buffered I/O with in-memory buffers. These buffers may then be included in the transaction in the same way that any other memory location might be included. This seems to be the mechanism of choice, and it does work well in many common cases of situations such as stream I/O and mass-storage I/O. However, special handling is required in cases where multiple record-oriented output streams are merged onto a single file from multiple processes, as might be done using the ”+” option to fopen() or the O_APPEND flag to open(). In addition, as will be seen in the next section, common networking operations cannot be handled via buffering.

2. Prohibit I/O within transactions, so that any attempt to execute an I/O operation aborts the enclosing transaction (and perhaps multiple nested transactions). This approach seems to be the conventional TM approach for unbuffered I/O, but requires that TM interoperate with other synchronization primitives that do tolerate I/O.

3. Prohibit I/O within transactions, but enlist the compiler’s aid in enforcing this prohibition.

4. Permit only one special “inevitable” transaction [SMS08] to proceed at any given time, thus allowing inevitable transactions to contain I/O operations. This works in general, but severely limits the scalability and performance of I/O operations. Given that scalability and performance is a first-class goal of parallelism, this approach’s generality seems a bit self-limiting. Worse yet, use of inevitability to tolerate I/O operations seems to prohibit use of manual transaction-abort operations.\footnote{This difficulty was pointed out by Michael Factor.}

5. Create new hardware and protocols such that I/O operations can be pulled into the transactional substrate. In the case of input operations, the hardware would need to correctly predict the result of the operation, and to abort the transaction if the prediction failed.

I/O operations are a well-known weakness of TM, and it is not clear that the problem of supporting I/O in transactions has a reasonable general solution, at least if “reasonable” is to include usable performance and scalability. Nevertheless, continued time and attention to this problem will likely produce additional progress.

15.2.2 RPC Operations

One can execute RPCs within a lock-based critical section, as well as from within an RCU read-side critical section. What happens when you attempt to execute an RPC from within a transaction?

If both the RPC request and its response are to be contained within the transaction, and if some part of the transaction depends on the result returned by the response, then it is not possible to use the memory-buffer tricks that can be used in the case of buffered I/O. Any attempt to take this buffering approach would deadlock the transaction, as the request could not be transmitted until the transaction was guaranteed to succeed, but the transaction’s success might not be knowable until after the response is received, as is the case in the following example:

```c
1 begin_trans();
2 rpc_request();
3 i = rpc_response();
4 a[i]++;
5 end_trans();
```

The transaction’s memory footprint cannot be determined until after the RPC response is received, and until the transaction’s memory footprint can be determined, it is impossible to determine whether the transaction can be allowed to commit. The only action consistent with transactional semantics is therefore to unconditionally abort the transaction, which is, to say the least, unhelpful.

Here are some options available to TM:
CHAPTER 15. CONFLICTING VISIONS OF THE FUTURE

1. Prohibit RPC within transactions, so that any attempt to execute an RPC operation aborts the enclosing transaction (and perhaps multiple nested transactions). Alternatively, enlist the compiler to enforce RPC-free transactions. This approach does works, but will require TM to interact with other synchronization primitives.

2. Permit only one special “inevitable” transaction [SMS08] to proceed at any given time, thus allowing inevitable transactions to contain RPC operations. This works in general, but severely limits the scalability and performance of RPC operations. Given that scalability and performance is a first-class goal of parallelism, this approach’s generality seems a bit self-limiting. Furthermore, use of inevitable transactions to permit RPC operations rules out manual transaction-abort operations once the RPC operation has started.

3. Identify special cases where the success of the transaction may be determined before the RPC response is received, and automatically convert these to inevitable transactions immediately before sending the RPC request. Of course, if several concurrent transactions attempt RPC calls in this manner, it might be necessary to roll all but one of them back, with consequent degradation of performance and scalability. This approach nevertheless might be valuable given long-running transactions ending with an RPC. This approach still has problems with manual transaction-abort operations.

4. Identify special cases where the RPC response may be moved out of the transaction, and then proceed using techniques similar to those used for buffered I/O.

5. Extend the transactional substrate to include the RPC server as well as its client. This is in theory possible, as has been demonstrated by distributed databases. However, it is unclear whether the requisite performance and scalability requirements can be met by distributed-database techniques, given that memory-based TM cannot hide such latencies behind those of slow disk drives. Of course, given the advent of solid-state disks, it is also unclear how much longer databases will be permitted to hide their latencies behind those of disks drives.

As noted in the prior section, I/O is a known weakness of TM, and RPC is simply an especially problematic case of I/O.

15.2.3 Memory-Mapping Operations

It is perfectly legal to execute memory-mapping operations (including `mmap()`, `shmat()` and `munmap()` [Gro01]) within a lock-based critical section, and, at least in principle, from within an RCU read-side critical section. What happens when you attempt to execute such an operation from within a transaction? More to the point, what happens if the memory region being remapped contains some variables participating in the current thread’s transaction? And what if this memory region contains variables participating in some other thread’s transaction?

It should not be necessary to consider cases where the TM system’s metadata is remapped, given that most locking primitives do not define the outcome of remapping their lock variables.

Here are some memory-mapping options available to TM:

1. Memory remapping is illegal within a transaction, and will result in all enclosing transactions being aborted. This does simplify things somewhat, but also requires that TM interoperate with synchronization primitives that do tolerate remapping from within their critical sections.

2. Memory remapping is illegal within a transaction, and the compiler is enlisted to enforce this prohibition.

3. Memory mapping is legal within a transaction, but aborts all other transactions having variables in the region mapped over.

4. Memory mapping is legal within a transaction, but the mapping operation will fail if the region being mapped overlaps with the current transaction’s footprint.

5. All memory-mapping operations, whether within or outside a transaction, check the region being mapped against the memory footprint of all transactions in the system. If there is overlap, then the memory-mapping operation fails.

6. The effect of memory-mapping operations that overlap the memory footprint of any transaction in the system is determined by the TM conflict manager,
which might dynamically determine whether to fail
the memory-mapping operation or abort any conflict-
ing transactions.

It is interesting to note that munmap() leaves the rel-

vant region of memory unmapped, which could have
additional interesting implications.\(^2\)

### 15.2.4 Multithreaded Transactions

It is perfectly legal to create processes and threads while
holding a lock or, for that matter, from within an RCU
read-side critical section. Not only is it legal, but it is quite
simple, as can be seen from the following code fragment:

```c
1   pthread_mutex_lock(...);
2   for (i = 0; i < ncpus; i++)
3       pthread_create(&tid[i], ...);
4   for (i = 0; i < ncpus; i++)
5       pthread_join(tid[i], ...);
6   pthread_mutex_unlock(...);
```

This pseudo-code fragment uses `pthread_create()` to spawn one thread per CPU, then uses `pthread_join()` to wait for each to complete, all un-
der the protection of `pthread_mutex_lock()`. The

effect is to execute a lock-based critical section in parallel,
and one could obtain a similar effect using `fork()` and `wait()`. Of course, the critical section would need to
be quite large to justify the thread-spawning overhead,
but there are many examples of large critical sections in
production software.

What might TM do about thread spawning within a
transaction?

1. Declare `pthread_create()` to be illegal within
transactions, resulting in transaction abort (preferred)
or undefined behavior. Alternatively, enlist the compi-
ler to enforce `pthread_create()`-free trans-
actions.

2. Permit `pthread_create()` to be executed
within a transaction, but only the parent thread will
be considered to be part of the transaction. This
approach seems to be reasonably compatible with
existing and posited TM implementations, but seems
to be a trap for the unwary. This approach raises
further questions, such as how to handle conflicting
child-thread accesses.

3. Convert the `pthread_create()`s to function
calls. This approach is also an attractive nuisance, as
it does not handle the not-uncommon cases where
the child threads communicate with one another. In
addition, it does not permit parallel execution of the
body of the transaction.

4. Extend the transaction to cover the parent and all
child threads. This approach raises interesting ques-
tions about the nature of conflicting accesses, given
that the parent and children are presumably permit-
ted to conflict with each other, but not with other
threads. It also raises interesting questions as to
what should happen if the parent thread does not
wait for its children before committing the transac-
tion. Even more interesting, what happens if the
parent conditionally executes `pthread_join()` based on the values of variables participating in the
transaction? The answers to these questions are rea-
sonably straightforward in the case of locking. The
answers for TM are left as an exercise for the reader.

Given that parallel execution of transactions is com-
monplace in the database world, it is perhaps surprising
that current TM proposals do not provide for it. On the
other hand, the example above is a fairly sophisticated
use of locking that is not normally found in simple text-
book examples, so perhaps its omission is to be expected.
That said, there are rumors that some TM researchers
are investigating fork/join parallelism within transactions,
so perhaps this topic will soon be addressed more thor-
oughly.

### 15.2.5 Extra-Transactional Accesses

Within a lock-based critical section, it is perfectly legal
to manipulate variables that are concurrently accessed or
even modified outside that lock’s critical section, with one
common example being statistical counters. The same
thing is possible within RCU read-side critical sections,
and is in fact the common case.

Given mechanisms such as the so-called “dirty reads”
that are prevalent in production database systems, it is not
surprising that extra-transactional accesses have received
serious attention from the proponents of TM, with the
concepts of weak and strong atomicity [BLM06] being
but one case in point.

Here are some extra-transactional options available to
TM:

\(^2\) This difference between mapping and unmapping was noted by Josh Triplett.
1. Conflicts due to extra-transactional accesses always abort transactions. This is strong atomicity.

2. Conflicts due to extra-transactional accesses are ignored, so only conflicts among transactions can abort transactions. This is weak atomicity.

3. Transactions are permitted to carry out non-transactional operations in special cases, such as when allocating memory or interacting with lock-based critical sections.

4. Produce hardware extensions that permit some operations (for example, addition) to be carried out concurrently on a single variable by multiple transactions.

It appears that transactions were conceived as standing alone, with no interaction required with any other synchronization mechanism. If so, it is no surprise that much confusion and complexity arises when combining transactions with non-transactional accesses. But unless transactions are to be confined to small updates to isolated data structures, or alternatively to be confined to new programs that do not interact with the huge body of existing parallel code, then transactions absolutely must be so combined if they are to have large-scale practical impact in the near term.

15.2.6 Time Delays

An important special case of interaction with extra-transactional accesses involves explicit time delays within a transaction. Of course, the idea of a time delay within a transaction flies in the face of TM’s atomicity property, but one can argue that this sort of thing is what weak atomicity is all about. Furthermore, correct interaction with memory-mapped I/O sometimes requires carefully controlled timing, and applications often use time delays for varied purposes.

So, what can TM do about time delays within transactions?

1. Ignore time delays within transactions. This has an appearance of elegance, but like too many other “elegant” solutions, fails to survive first contact with legacy code. Such code, which might well have important time delays in critical sections, would fail upon being transactionalized.

2. Abort transactions upon encountering a time-delay operation. This is attractive, but it is unfortunately not always possible to automatically detect a time-delay operation. Is that tight loop computing something important, or is it instead waiting for time to elapse?

3. Enlist the compiler to prohibit time delays within transactions.

4. Let the time delays execute normally. Unfortunately, some TM implementations publish modifications only at commit time, which would in many cases defeat the purpose of the time delay.

It is not clear that there is a single correct answer. TM implementations featuring weak atomicity that publish changes immediately within the transaction (rolling these changes back upon abort) might be reasonably well served by the last alternative. Even in this case, the code at the other end of the transaction may require a substantial redesign to tolerate aborted transactions.

15.2.7 Locking

It is commonplace to acquire locks while holding other locks, which works quite well, at least as long as the usual well-known software-engineering techniques are employed to avoid deadlock. It is not unusual to acquire locks from within RCU read-side critical sections, which eases deadlock concerns because RCU read-side primitives cannot participated in lock-based deadlock cycles. But happens when you attempt to acquire a lock from within a transaction? In theory, the answer is trivial: simply manipulate the data structure representing the lock as part of the transaction, and everything works out perfectly. In practice, a number of non-obvious complications [VGS08] can arise, depending on implementation details of the TM system. These complications can be resolved, but at the cost of a 45% increase in overhead for locks acquired outside of transactions and a 300% increase in overhead for locks acquired within transactions. Although these overheads might be acceptable for transactional programs containing small amounts of locking, they are often completely unacceptable for production-quality lock-based programs wishing to use the occasional transaction.

1. Use only locking-friendly TM implementations. Unfortunately, the locking-unfriendly implementations have some attractive properties, including low overhead for successful transactions and the ability to accommodate extremely large transactions.
2. Use TM only “in the small” when introducing TM to lock-based programs, thereby accommodating the limitations of locking-friendly TM implementations.

3. Set aside locking-based legacy systems entirely, re-implementing everything in terms of transactions. This approach has no shortage of advocates, but this requires that all the issues described in this series be resolved. During the time it takes to resolve these issues, competing synchronization mechanisms will of course also have the opportunity to improve.

4. Use TM strictly as an optimization in lock-based systems, as was done by the TxLinux [RHP+07] group. This approach seems sound, but leaves the locking design constraints (such as the need to avoid deadlock) firmly in place.

5. Strive to reduce the overhead imposed on locking primitives.

The fact that there could possibly a problem interfacing TM and locking came as a surprise to many, which underscores the need to try out new mechanisms and primitives in real-world production software. Fortunately, the advent of open source means that a huge quantity of such software is now freely available to everyone, including researchers.

### 15.2.8 Reader-Writer Locking

It is commonplace to read-acquire reader-writer locks while holding other locks, just as long as the usual well-known software-engineering techniques are employed to avoid deadlock. Read-acquiring reader-writer locks from within RCU read-side critical sections also works, and doing so eases deadlock concerns because RCU read-side primitives cannot participate in lock-based deadlock cycles. But what happens when you attempt to read-acquire a reader-writer lock from within a transaction?

Unfortunately, the straightforward approach to read-acquiring the traditional counter-based reader-writer lock within a transaction defeats the purpose of the reader-writer lock. To see this, consider a pair of transactions concurrently attempting to read-acquire the same reader-writer lock. Because read-acquisition involves modifying the reader-writer lock’s data structures, a conflict will result, which will roll back one of the two transactions. This behavior is completely inconsistent with the reader-writer lock’s goal of allowing concurrent readers.

Here are some options available to TM:

1. Use per-CPU or per-thread reader-writer locking [HW92], which allows a given CPU (or thread, respectively) to manipulate only local data when read-acquiring the lock. This would avoid the conflict between the two transactions concurrently read-acquiring the lock, permitting both to proceed, as intended. Unfortunately, (1) the write-acquisition overhead of per-CPU/thread locking can be extremely high, (2) the memory overhead of per-CPU/thread locking can be prohibitive, and (3) this transformation is available only when you have access to the source code in question. Other more-recent scalable reader-writer locks [LLO09] might avoid some or all of these problems.

2. Use TM only “in the small” when introducing TM to lock-based programs, thereby avoiding read-acquiring reader-writer locks from within transactions.

3. Set aside locking-based legacy systems entirely, re-implementing everything in terms of transactions. This approach has no shortage of advocates, but this requires that all the issues described in this series be resolved. During the time it takes to resolve these issues, competing synchronization mechanisms will of course also have the opportunity to improve.

4. Use TM strictly as an optimization in lock-based systems, as was done by the TxLinux [RHP+07] group. This approach sounds sound, but leaves the locking design constraints (such as the need to avoid deadlock) firmly in place. Furthermore, this approach can result in unnecessary transaction rollbacks when multiple transactions attempt to read-acquire the same lock.

Of course, there might well be other non-obvious issues surrounding combining TM with reader-writer locking, as there in fact were with exclusive locking.

### 15.2.9 Persistence

There are many different types of locking primitives. One interesting distinction is persistence, in other words, whether the lock can exist independently of the address space of the process using the lock.

Non-persistent locks include `pthread_mutex_lock()`, `pthread_rwlock_rdlock()`, and most
kernel-level locking primitives. If the memory locations
instantiating a non-persistent lock’s data structures dis-
appear, so does the lock. For typical use of pthread-
mutex_lock(), this means that when the process exits,
all of its locks vanish. This property can be exploited in
order to trivialize lock cleanup at program shutdown time,
but makes it more difficult for unrelated applications to
share locks, as such sharing requires the applications to
share memory.

Persistent locks help avoid the need to share memory
among unrelated applications. Persistent locking APIs in-
clude the flock family, lockf(), System V semaphores,
or the O_CREAT flag to open(). These persistent APIs
can be used to protect large-scale operations spanning
runs of multiple applications, and, in the case of O_CREAT
even surviving operating-system reboot. If need be, locks can span multiple computer systems via dis-
tributed lock managers.

Persistent locks can be used by any application, in-
cluding applications written using multiple languages and
software environments. In fact, a persistent lock might
well be acquired by an application written in C and re-
leased by an application written in Python.

How could a similar persistent functionality be pro-
vided for TM?

1. Restrict persistent transactions to special-purpose
environments designed to support them, for example,
SQL. This clearly works, given the decades-long
history of database systems, but does not provide
the same degree of flexibility provided by persistent
locks.

2. Use snapshot facilities provided by some storage de-
vices and/or filesystems. Unfortunately, this does not
handle network communication, nor does it handle
I/O to devices that do not provide snapshot capabili-
ties, for example, memory sticks.

3. Build a time machine.

Of course, the fact that it is called transactional memory
should give us pause, as the name itself conflicts with
the concept of a persistent transaction. It is nevertheless
worthwhile to consider this possibility as an important
test case probing the inherent limitations of transactional
memory.

15.2.10 Dynamic Linking and Loading

Both lock-based critical sections and RCU read-side criti-
cal sections can legitimately contain code that invokes dy-
namically linked and loaded functions, including C/C++
shared libraries and Java class libraries. Of course, the
code contained in these libraries is by definition unknow-
able at compile time. So, what happens if a dynamically
loaded function is invoked within a transaction?

This question has two parts: (a) how do you dynam-
ically link and load a function within a transaction and
(b) what do you do about the unknowable nature of the
code within this function? To be fair, item (b) poses some
challenges for locking and RCU as well, at least in the-
ory. For example, the dynamically linked function might
introduce a deadlock for locking or might (erroneously)
introduce a quiescent state into an RCU read-side critical
section. The difference is that while the class of opera-
tions permitted in locking and RCU critical sections is
well-understood, there appears to still be considerable
uncertainty in the case of TM. In fact, different implemen-
tations of TM seem to have different restrictions.

So what can TM do about dynamically linked and
loaded library functions? Options for part (a), the ac-
tual loading of the code, include the following:

1. Treat the dynamic linking and loading in a manner
similar to a page fault, so that the function is loaded
and linked, possibly aborting the transaction in the
process. If the transaction is aborted, the retry will
find the function already present, and the transaction
can thus be expected to proceed normally.

2. Disallow dynamic linking and loading of functions
from within transactions.

Options for part (b), the inability to detect TM-
unfriendly operations in a not-yet-loaded function, possi-
bilities include the following:

1. Just execute the code: if there are any TM-unfriendly
operations in the function, simply abort the transac-
tion. Unfortunately, this approach makes it impos-
sible for the compiler to determine whether a given
group of transactions may be safely composed. One
way to permit composability regardless is inevitable
transactions, however, current implementations per-
mit only a single inevitable transaction to proceed
at any given time, which can severely limit perfor-
mance and scalability. Inevitable transactions also
seem to rule out use of manual transaction-abort
operations.

2. Decorate the function declarations indicating which
functions are TM-friendly. These decorations can
then be enforced by the compiler’s type system. Of course, for many languages, this requires language extensions to be proposed, standardized, and implemented, with the corresponding time delays. That said, the standardization effort is already in progress [ATS09].

3. As above, disallow dynamic linking and loading of functions from within transactions.

I/O operations are of course a known weakness of TM, and dynamic linking and loading can be thought of as yet another special case of I/O. Nevertheless, the proponents of TM must either solve this problem, or resign themselves to a world where TM is but one tool of several in the parallel programmer’s toolbox. (To be fair, a number of TM proponents have long since resigned themselves to a world containing more than just TM.)

15.2.11 Debugging

The usual debugging operations such as breakpoints work normally within lock-based critical sections and from RCU read-side critical sections. However, in initial transactional-memory hardware implementations [DLMN09] an exception within a transaction will abort that transaction, which in turn means that breakpoints abort all enclosing transactions.

So how can transactions be debugged?

1. Use software emulation techniques within transactions containing breakpoints. Of course, it might be necessary to emulate all transactions any time a breakpoint is set within the scope of any transaction. If the runtime system is unable to determine whether or not a given breakpoint is within the scope of a transaction, then it might be necessary to emulate all transactions just to be on the safe side. However, this approach might impose significant overhead, which might in turn obscure the bug being pursued.

2. Use only hardware TM implementations that are capable of handling breakpoint exceptions. Unfortunately, as of this writing (September 2008), all such implementations are strictly research prototypes.

3. Use only software TM implementations, which are (very roughly speaking) more tolerant of exceptions than are the simpler of the hardware TM implementations. Of course, software TM tends to have higher overhead than hardware TM, so this approach may not be acceptable in all situations.

4. Program more carefully, so as to avoid having bugs in the transactions in the first place. As soon as you figure out how to do this, please do let everyone know the secret!

There is some reason to believe that transactional memory will deliver productivity improvements compared to other synchronization mechanisms, but it does seem quite possible that these improvements could easily be lost if traditional debugging techniques cannot be applied to transactions. This seems especially true if transactional memory is to be used by novices on large transactions. In contrast, macho “top-gun” programmers might be able to dispense with such debugging aids, especially for small transactions.

Therefore, if transactional memory is to deliver on its productivity promises to novice programmers, the debugging problem does need to be solved.

15.2.12 The exec() System Call

One can execute an exec() system call while holding a lock, and also from within an RCU read-side critical section. The exact semantics depends on the type of primitive.

In the case of non-persistent primitives (including pthread_mutex_lock(), pthread_rwlock_rdlock(), and RCU), if the exec() succeeds, the whole address space vanishes, along with any locks being held. Of course, if the exec() fails, the address space still lives, so any associated locks would also still live. A bit strange perhaps, but reasonably well defined.

On the other hand, persistent primitives (including the flock family, lockf(), System V semaphores, and the O_CREAT flag to open()) would survive regardless of whether the exec() succeeded or failed, so that the exec()ed program might well release them.

Quick Quiz 15.1: What about non-persistent primitives represented by data structures in mmap() regions of memory? What happens when their is an exec() within a critical section of such a primitive?

What happens when you attempt to execute an exec() system call from within a transaction?

1. Disallow exec() within transactions, so that the enclosing transactions abort upon encountering the exec(). This is well defined, but clearly requires non-TM synchronization primitives for use in conjunction with exec().
2. Disallow \texttt{exec()} within transactions, with the compiler enforcing this prohibition. There is a draft specification for TM in C++ that takes this approach, allowing functions to be decorated with the \texttt{transaction\_safe} and \texttt{transaction\_unsafe} attributes. This approach has some advantages over aborting the transaction at runtime, but again requires non-TM synchronization primitives for use in conjunction with \texttt{exec()}.

3. Treat the transaction in a manner similar to non-persistent Locking primitives, so that the transaction survives if \texttt{exec()} fails, and silently commits if the \texttt{exec()} succeeds. The case were some of the variables affected by the transaction reside in \texttt{mmap()}ed memory (and thus could survive a successful \texttt{exec()} system call) is left as an exercise for the reader.

4. Abort the transaction (and the \texttt{exec()} system call) if the \texttt{exec()} system call would have succeeded, but allow the transaction to continue if the \texttt{exec()} system call would fail. This is in some sense the “correct” approach, but it would require considerable work for a rather unsatisfying result.

The \texttt{exec()} system call is perhaps the strangest example of an obstacle to universal TM applicability, as it is not completely clear what approach makes sense, and some might argue that this is merely a reflection of the perils of interacting with execls in real life. That said, the two options prohibiting \texttt{exec()} within transactions are perhaps the most logical of the group.

15.2.13 RCU

Because read-copy update (RCU) finds its main use in the Linux kernel, one might be forgiven for assuming that there had been no academic work on combining RCU and TM. However, the TxLinux group from the University of Texas at Austin had no choice [RHP+07]. The fact that they applied TM to the Linux 2.6 kernel, which uses RCU, forced them to integrate TM and RCU, with TM taking the place of locking for RCU updates. Unfortunately, although the paper does state that the RCU implementation’s locks (e.g., \texttt{rcu\_ctrlblk.lock}) were converted to transactions, it is silent about what happened to locks used in RCU-based updates (e.g., \texttt{dcache\_lock}).

It is important to note that RCU permits readers and updaters to run concurrently, further permitting RCU readers to access data that is in the act of being updated. Of course, this property of RCU, whatever its performance, scalability, and real-time-response benefits might be, flies in the face of the underlying atomicity properties of TM.

So how should TM-based updates interact with concurrent RCU readers? Some possibilities are as follows:

1. RCU readers abort concurrent conflicting TM updates. This is in fact the approach taken by the TxLinux project. This approach does preserve RCU semantics, and also preserves RCU’s read-side performance, scalability, and real-time-response properties, but it does have the unfortunate side-effect of unnecessarily aborting conflicting updates. In the worst case, a long sequence of RCU readers could potentially starve all updaters, which could in theory result in system hangs. In addition, not all TM implementations offer the strong atomicity required to implement this approach.

2. RCU readers that run concurrently with conflicting TM updates get old (pre-transaction) values from any conflicting RCU loads. This preserves RCU semantics and performance, and also prevents RCU-update starvation. However, not all TM implementations can provide timely access to old values of variables that have been tentatively updated by an in-flight transaction. In particular, log-based TM implementations that maintain old values in the log (thus making for excellent TM commit performance) are not likely to be happy with this approach. Perhaps the \texttt{rcu\_dereference()} primitive can be leveraged to permit RCU to access the old values within a greater range of TM implementations, though performance might still be an issue.

3. If an RCU reader executes an access that conflicts with an in-flight transaction, then that RCU access is delayed until the conflicting transaction either commits or aborts. This approach preserves RCU semantics, but not RCU’s performance or real-time response, particularly in presence of long-running transactions. In addition, not all TM implementations are capable of delaying conflicting accesses. That said, this approach seems eminently reasonable for hardware TM implementations that support only small transactions.

4. RCU readers are converted to transactions. This ap-
proach pretty much guarantees that RCU is compatible with any TM implementation, but it also imposes TM’s rollbacks on RCU read-side critical sections, destroying RCU’s real-time response guarantees, and also degrading RCU’s read-side performance. Furthermore, this approach is infeasible in cases where any of the RCU read-side critical sections contains operations that the TM implementation in question is incapable of handling.

5. Many update-side uses of RCU modify a single pointer to publish a new data structure. In some these cases, RCU can safely be permitted to see a transactional pointer update that is subsequently rolled back, as long as the transaction respects memory ordering and as long as the roll-back process uses call_rcu() to free up the corresponding structure. Unfortunately, not all TM implementations respect memory barriers within a transaction. Apparently, the thought is that because transactions are supposed to be atomic, the ordering of the accesses within the transaction is not supposed to matter.

6. Prohibit use of TM in RCU updates. This is guaranteed to work, but seems a bit restrictive.

It seems likely that additional approaches will be uncovered, especially given the advent of user-level RCU implementations.⁴

15.2.14 Discussion

The obstacles to universal TM adoption lead to the following conclusions:

1. One interesting property of TM is the fact that transactions are subject to rollback and retry. This property underlies TM’s difficulties with irreversible operations, including unbuffered I/O, RPCs, memory-mapping operations, time delays, and the exec() system call. This property also has the unfortunate consequence of introducing all the complexities inherent in the possibility of failure into synchronization primitives, often in a developer-visible manner.

2. Another interesting property of TM, noted by Shpeisman et al. [SATG⁺09], is that TM intertwines the synchronization with the data it protects. This property underlies TM’s issues with I/O, memory-mapping operations, extra-transactional accesses, and debugging breakpoints. In contrast, conventional synchronization primitives, including locking and RCU, maintain a clear separation between the synchronization primitives and the data that they protect.

3. One of the stated goals of many workers in the TM area is to ease parallelization of large sequential programs. As such, individual transactions are commonly expected to execute serially, which might do much to explain TM’s issues with multithreaded transactions.

What should TM researchers and developers do about all of this?

One approach is to focus on TM in the small, focusing on situations where hardware assist potentially provides substantial advantages over other synchronization primitives. This is in fact the approach Sun took with its Rock research CPU [DLMN09]. Some TM researchers seem to agree with this approach, while others have much higher hopes for TM.

Of course, it is quite possible that TM will be able to take on larger problems, and this section lists a few of the issues that must be resolved if TM is to achieve this lofty goal.

Of course, everyone involved should treat this as a learning experience. It would seem that TM researchers have great deal to learn from practitioners who have successfully built large software systems using traditional synchronization primitives.

And vice versa.

15.3 Shared-Memory Parallel Functional Programming

15.4 Process-Based Parallel Functional Programming
Appendix A

Important Questions

The following sections discuss some important questions relating to SMP programming. Each section also shows how to avoid having to worry about the corresponding question, which can be extremely important if your goal is to simply get your SMP code working as quickly and painlessly as possible — which is an excellent goal, by the way!

Although the answers to these questions are often quite a bit less intuitive than they would be in a single-threaded setting, with a bit of work, they are not that difficult to understand. If you managed to master recursion, there is nothing in here that should pose an overwhelming challenge.

A.1 What Does “After” Mean?

“After” is an intuitive, but surprisingly difficult concept. An important non-intuitive issue is that code can be delayed at any point for any amount of time. Consider a producing and a consuming thread that communicate using a global struct with a timestamp “t” and integer fields “a”, “b”, and “c”. The producer loops recording the current time (in seconds since 1970 in decimal), then updating the values of “a”, “b”, and “c”, as shown in Figure A.1. The consumer code loops, also recording the current time, but also copying the producer’s timestamp along with the fields “a”, “b”, and “c”, as shown in Figure A.2. At the end of the run, the consumer outputs a list of anomalous recordings, e.g., where time has appeared to go backwards.

Quick Quiz A.1: What SMP coding errors can you see in these examples? See time.c for full code.

One might intuitively expect that the difference between the producer and consumer timestamps would be quite small, as it should not take much time for the producer to record the timestamps or the values. An excerpt of some sample output on a dual-core 1GHz x86 is shown in Table A.1. Here, the “seq” column is the number of times through the loop, the “time” column is the time of the anomaly in seconds, the “delta” column is the number of seconds the consumer’s timestamp follows that of the producer (where a negative value indicates that the consumer has collected its timestamp before the producer did), and the columns labelled “a”, “b”, and “c” show the amount that these variables increased since the prior snapshot collected by the consumer.

```
/* WARNING: BUGGY CODE. */
void *producer(void *ignored)
{
  int i = 0;
  producer_ready = 1;
  while (!goflag)
    sched_yield();
  while (goflag) {
    ss.t = dgettimeofday();
    ss.a = ss.c + 1;
    ss.b = ss.a + 1;
    ss.c = ss.b + 1;
    i++;
    printf("producer exiting: %d samples\n", i);
    producer_done = 1;
    return (NULL);
  }
}
```

```c
Figure A.1: “After” Producer Function
```

```
<table>
<thead>
<tr>
<th>seq</th>
<th>time (seconds)</th>
<th>delta</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>17563:</td>
<td>1152396.251585</td>
<td>(-16.928)</td>
<td>27</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>18004:</td>
<td>1152396.252581</td>
<td>(-12.875)</td>
<td>24</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>18163:</td>
<td>1152396.252955</td>
<td>(-19.073)</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>18765:</td>
<td>1152396.254449</td>
<td>(-148.773)</td>
<td>216</td>
<td>216</td>
<td>216</td>
</tr>
<tr>
<td>19863:</td>
<td>1152396.256960</td>
<td>(-6.914)</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>21644:</td>
<td>1152396.260959</td>
<td>(-5.960)</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>23408:</td>
<td>1152396.264957</td>
<td>(-20.027)</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>
```

Table A.1: “After” Program Sample Output
APPENDIX A. IMPORTANT QUESTIONS

Why is time going backwards? The number in parentheses is the difference in microseconds, with a large number exceeding 10 microseconds, and one exceeding even 100 microseconds! Please note that this CPU can potentially execute about more than 100,000 instructions in that time.

One possible reason is given by the following sequence of events:

1. Consumer obtains timestamp (Figure A.2, line 13).
2. Consumer is preempted.
3. An arbitrary amount of time passes.
4. Producer obtains timestamp (Figure A.1, line 10).
5. Consumer starts running again, and picks up the producer’s timestamp (Figure A.2, line 14).

In this scenario, the producer’s timestamp might be an arbitrary amount of time after the consumer’s timestamp.

How do you avoid agonizing over the meaning of “after” in your SMP code?

Simply use SMP primitives as designed.

In this example, the easiest fix is to use locking, for example, acquire a lock in the producer before line 10 in Figure A.1 and in the consumer before line 13 in Figure A.2. This lock must also be released after line 13 in Figure A.1 and after line 17 in Figure A.2. These locks cause the code segments in line 10-13 of Figure A.1 and in line 13-17 of Figure A.2 to exclude each other, in other words, to run atomically with respect to each other. This is represented in Figure A.3: the locking prevents any of the boxes of code from overlapping in time, so that the consumer’s timestamp must be collected after the prior producer’s timestamp. The segments of code in each box in this figure are termed “critical sections”; only one such critical section may be executing at a given time.

This addition of locking results in output as shown in Figure A.2. Here there are no instances of time going backwards, instead, there are only cases with more than 1,000 counts different between consecutive reads by the consumer.

Quick Quiz A.2: How could there be such a large gap between successive consumer reads? See timelocked. c for full code.

<table>
<thead>
<tr>
<th>seq</th>
<th>time (seconds)</th>
<th>delta</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>58597</td>
<td>1156521.556296</td>
<td>3.815</td>
<td>1485</td>
<td>1485</td>
<td>1485</td>
</tr>
<tr>
<td>403927</td>
<td>1156523.446636</td>
<td>2.146</td>
<td>2583</td>
<td>2583</td>
<td>2583</td>
</tr>
</tbody>
</table>

Table A.2: Locked “After” Program Sample Output
A.1. WHAT DOES “AFTER” MEAN?

In summary, if you acquire an exclusive lock, you know that anything you do while holding that lock will appear to happen after anything done by any prior holder of that lock. No need to worry about which CPU did or did not execute a memory barrier, no need to worry about the CPU or compiler reordering operations – life is simple. Of course, the fact that this locking prevents these two pieces of code from running concurrently might limit the program’s ability to gain increased performance on multiprocessors, possibly resulting in a “safe but slow” situation. Chapter 5 describes ways of gaining performance and scalability in many situations.

However, in most cases, if you find yourself worrying about what happens before or after a given piece of code, you should take this as a hint to make better use of the standard primitives. Let these primitives do the worrying for you.

Figure A.3: Effect of Locking on Snapshot Collection
Appendix B

Synchronization Primitives

All but the simplest parallel programs require synchronization primitives. This appendix gives a quick overview of a set of primitives based loosely on those in the Linux kernel.

Why Linux? Because it is one of the well-known, largest, and easily obtained bodies of parallel code available. We believe that reading code is, if anything, more important to learning than is writing code, so by using examples similar to real code in the Linux kernel, we are enabling you to use Linux to continue your learning as you progress beyond the confines of this book.

Why based loosely rather than following the Linux kernel API exactly? First, the Linux API changes with time, so any attempt to track it exactly would likely end in total frustration for all involved. Second, many of the members of the Linux kernel API are specialized for use in a production-quality operating-system kernel. This specialization introduces complexities that, though absolutely necessary in the Linux kernel itself, are often more trouble than they are worth in the “toy” programs that we will be using to demonstrate SMP and realtime design principles and practices. For example, properly checking for error conditions such as memory exhaustion is a “must” in the Linux kernel, however, in “toy” programs it is perfectly acceptable to simply abort() the program, correct the problem, and rerun.

Finally, it should be possible to implement a trivial mapping layer between this API and most production-level APIs. A pthreads implementation is available (CodeSamples/api-pthreads/api-pthreads.h), and a Linux-kernel-module API would not be difficult to create.

Quick Quiz B.1: Give an example of a parallel program that could be written without synchronization primitives.

The following sections describe commonly used classes of synchronization primitives. More esoteric primitives will be introduced in later revision.

Section B.1 covers organization/initialization primitives; Section B.2 presents thread creation, destruction, and control primitives; Section B.3 presents locking primitives; Section B.4 presents per-thread and per-CPU variable primitives; and Section B.5 gives an overview of the relative performance of the various primitives.

B.1 Organization and Initialization

Currently include ../api.h, and there is only pthreads. Expand and complete once the CodeSamples structure settles down.

B.1.1 smp_init():
You must invoke smp_init() before invoking any other primitives.

B.2 Thread Creation, Destruction, and Control

This API focuses on “threads”, which are a locus of control. Each such thread has an identifier of type thread_id_t, and no two threads running at a given time will have the same identifier. Threads share everything except for per-thread local state, which includes program counter and stack.

The thread API is shown in Figure B.1, and members are described in the following sections.

---

1 There are many other names for similar software constructs, including “process”, “task”, “fiber”, “event”, and so on. Similar design principles apply to all of them.

2 How is that for a circular definition?


APPENDIX B. SYNCHRONIZATION PRIMITIVES

186

Figure B.1: Thread API

B.2.1 create_thread()

The create_thread() primitive creates a new thread, starting the new thread’s execution at the function func specified by create_thread()’s first argument, and passing it the argument specified by create_thread()’s second argument. This newly created thread will terminate when it returns from the starting function specified by func. The create_thread() primitive returns the thread_id_t corresponding to the newly created child thread.

This primitive will abort the program if more than NR_THREADS threads are created, counting the one implicitly created by running the program. NR_THREADS is a compile-time constant that may be modified, though some systems may have an upper bound for the allowable number of threads.

B.2.2 smp_thread_id()

Because the thread_id_t returned from create_thread() is system-dependent, the smp_thread_id() primitive returns a thread index corresponding to the thread making the request. This index is guaranteed to be less than the maximum number of threads that have been in existence since the program started, and is therefore useful for bitmasks, array indices, and the like.

B.2.3 for_each_thread()

The for_each_thread() macro loops through all threads that exist, including all threads that would exist if created. This macro is useful for handling per-thread variables as will be seen in Section B.4.

B.2.4 for_each_running_thread()

The for_each_running_thread() macro loops through only those threads that currently exist. It is the caller’s responsibility to synchronize with thread creation and deletion if required.

B.2.5 wait_thread()

The wait_thread() primitive waits for completion of the thread specified by the thread_id_t passed to it. This in no way interferes with the execution of the specified thread; instead, it merely waits for it. Note that wait_thread() returns the value that was returned by the corresponding thread.

B.2.6 wait_all_threads()

The wait_all_threads() primitive waits for completion of all currently running threads. It is the caller’s responsibility to synchronize with thread creation and deletion if required. However, this primitive is normally used to clean up and end a run, so such synchronization is normally not needed.

B.2.7 Example Usage

Figure B.2 shows an example hello-world-like child thread. As noted earlier, each thread is allocated its own stack, so each thread has its own private arg argument and myarg variable. Each child simply prints its argument and its smp_thread_id() before exiting. Note that the return statement on line 7 terminates the thread, returning a NULL to whoever invokes wait_thread() on this thread.

```c
1 void *thread_test(void *arg)
2 {
3     int myarg = (int)arg;
4     printf("child thread \%d: smp_thread_id() = \%d\n", myarg, smp_thread_id());
5     return NULL;
6 }
```

Figure B.2: Example Child Thread

The parent program is shown in Figure B.3. It invokes smp_init() to initialize the threading system on line 6, parses arguments on lines 7-14, and announces its presence on line 15. It creates the specified number of child
threads on lines 16-17, and waits for them to complete on line 18. Note that `wait_all_threads()` discards the threads return values, as in this case they are all NULL, which is not very interesting.

```c
1 int main(int argc, char *argv[]) {
2    int i;
3 int nkids = 1;
4   smp_init();
5   if (argc > 1) {
6       int nkids = strtoul(argv[1], NULL, 0);
7       if (nkids > NR_THREADS) {
8           fprintf(stderr, "nkids=%d too big, max=%d\n",
9                   n
10                      , NR_THREADS);
11           usage(argv[0]);
12       }
13   }
14   printf("Parent spawning %d threads.\n", nkids);
15   for (i = 0; i < nkids; i++)
16       create_thread(thread_test, (void *)i);
17   wait_all_threads();
18   printf("All threads completed.\n", nkids);
19   exit(0);
20 }
```

Figure B.3: Example Parent Thread

### B.3 Locking

The locking API is shown in Figure B.4, each API element being described in the following sections.

```c
void spin_lock_init(spinlock_t *sp);
void spin_lock(spinlock_t *sp);
int spin_trylock(spinlock_t *sp);
void spin_unlock(spinlock_t *sp);
```

Figure B.4: Locking API

#### B.3.1 spin_lock_init()

The `spin_lock_init()` primitive initializes the specified spinlock_t variable, and must be invoked before this variable is passed to any other spinlock primitive.

#### B.3.2 spin_lock()

The `spin_lock()` primitive acquires the specified spinlock, if necessary, waiting until the spinlock becomes available. In some environments, such as pthreads, this waiting will involve “spinning”, while in others, such as the Linux kernel, it will involve blocking.

The key point is that only one thread may hold a spinlock at any given time.

#### B.3.3 spin_trylock()

The `spin_trylock()` primitive acquires the specified spinlock, but only if it is immediately available. It returns true if it was able to acquire the spinlock and false otherwise.

#### B.3.4 spin_unlock()

The `spin_unlock()` primitive releases the specified spinlock, allowing other threads to acquire it.

@@@ likely need to add reader-writer locking.

#### B.3.5 Example Usage

A spinlock named `mutex` may be used to protect a variable `counter` as follows:

```c
spin_lock(&mutex);
counter++;
spin_unlock(&mutex);
```

Quick Quiz B.2: What problems could occur if the variable `counter` were incremented without the protection of `mutex`? ■

However, the `spin_lock()` and `spin_unlock()` primitives do have performance consequences, as will be seen in Section B.5.

### B.4 Per-Thread Variables

Figure B.5 shows the per-thread-variable API. This API provides the per-thread equivalent of global variables. Although this API is, strictly speaking, not necessary, it can greatly simply coding.

```c
DEFINE_PER_THREAD(type, name)
DECLARE_PER_THREAD(type, name)
per_thread(name, thread)
__get_thread_var(name)
init_per_thread(name, v)
```

Figure B.5: Per-Thread-Variable API

Quick Quiz B.3: How could you work around the lack of a per-thread-variable API on systems that do not provide it? ■
B.4.1 DEFINE_PER_THREAD()

The DEFINE_PER_THREAD() primitive defines a per-thread variable. Unfortunately, it is not possible to provide an initializer in the way permitted by the Linux kernel’s DEFINE_PER_THREAD() primitive, but there is an init_per_thread() primitive that permits easy runtime initialization.

B.4.2 DECLARE_PER_THREAD()

The DECLARE_PER_THREAD() primitive is a declaration in the C sense, as opposed to a definition. Thus, a DECLARE_PER_THREAD() primitive may be used to access a per-thread variable defined in some other file.

B.4.3 per_thread()

The per_thread() primitive accesses the specified thread’s variable.

B.4.4 __get_thread_var()

The __get_thread_var() primitive accesses the current thread’s variable.

B.4.5 init_per_thread()

The init_per_thread() primitive sets all threads’ instances of the specified variable to the specified value.

B.4.6 Usage Example

Suppose that we have a counter that is incremented very frequently but read out quite rarely. As will become clear in Section B.5, it is helpful to implement such a counter using a per-CPU variable. Such a variable can be defined as follows:

```c
DEFINE_PER_THREAD(int, counter);
```

The counter must be initialized as follows:

```c
init_per_thread(counter, 0);
```

A thread can increment its instance of this counter as follows:

```c
__get_thread_var(counter)++;
```

The value of the counter is then the sum of its instances. A snapshot of the value of the counter can thus be collected as follows:

```c
for_each_thread(i)
    sum += per_thread(counter, i);
```

Again, it is possible to gain a similar effect using other mechanisms, but per-thread variables combine convenience and high performance.

B.5 Performance

It is instructive to compare the performance of the locked increment shown in Section B.3 to that of per-thread variables (see Section B.4), as well as to conventional increment (as in “counter++”).

@@@ need parable on cache thrashing.
@@@ more here using performance results from a modest multiprocessor.
@@@ Also work in something about critical-section size? Or put later?

The difference in performance is quite large, to put it mildly. The purpose of this book is to help you write SMP programs, perhaps with realtime response, while avoiding such performance pitfalls. The next section starts this process by describing some of the reasons for this performance shortfall.
Appendix C

Why Memory Barriers?

So what possessed CPU designers to cause them to inflict memory barriers on poor unsuspecting SMP software designers?

In short, because reordering memory references allows much better performance, and so memory barriers are needed to force ordering in things like synchronization primitives whose correct operation depends on ordered memory references.

Getting a more detailed answer to this question requires a good understanding of how CPU caches work, and especially what is required to make caches really work well. The following sections:

1. present the structure of a cache,
2. describe how cache-coherency protocols ensure that CPUs agree on the value of each location in memory, and, finally,
3. outline how store buffers and invalidate queues help caches and cache-coherency protocols achieve high performance.

We will see that memory barriers are a necessary evil that is required to enable good performance and scalability, an evil that stems from the fact that CPUs are orders of magnitude faster than are both the interconnects between them and the memory they are attempting to access.

C.1 Cache Structure

Modern CPUs are much faster than are modern memory systems. A 2006 CPU might be capable of executing ten instructions per nanosecond, but will require many tens of nanoseconds to fetch a data item from main memory. This disparity in speed — more than two orders of magnitude — has resulted in the multi-megabyte caches found on modern CPUs. These caches are associated with the CPUs as shown in Figure C.1, and can typically be accessed in a few cycles.\footnote{It is standard practice to use multiple levels of cache, with a small level-one cache close to the CPU with single-cycle access time, and a larger level-two cache with a longer access time, perhaps roughly ten clock cycles. Higher-performance CPUs often have three or even four levels of cache.}

![Figure C.1: Modern Computer System Cache Structure](image)

Data flows among the CPUs’ caches and memory in fixed-length blocks called “cache lines”, which are normally a power of two in size, ranging from 16 to 256 bytes. When a given data item is first accessed by a given CPU, it will be absent from that CPU’s cache, meaning that a “cache miss” (or, more specifically, a “startup” or “warmup” cache miss) has occurred. The cache miss means that the CPU will have to wait (or be “stalled”) for hundreds of cycles while the item is fetched from memory. However, the item will be loaded into that CPU’s cache, so that subsequent accesses will find it in the cache and therefore run at full speed.
APPENDIX C. WHY MEMORY BARRIERS?

After some time, the CPU’s cache will fill, and subsequent misses will likely need to eject an item from the cache in order to make room for the newly fetched item. Such a cache miss is termed a “capacity miss”, because it is caused by the cache’s limited capacity. However, most caches can be forced to eject an old item to make room for a new item even when they are not yet full. This is due to the fact that large caches are implemented as hardware hash tables with fixed-size hash buckets (or “sets”, as CPU designers call them) and no chaining, as shown in Figure C.2.

This cache has sixteen “sets” and two “ways” for a total of 32 “lines”, each entry containing a single 256-byte “cache line”, which is a 256-byte-aligned block of memory. This cache line size is a little on the large side, but makes the hexadecimal arithmetic much simpler. In hardware parlance, this is a two-way set-associative cache, and is analogous to a software hash table with sixteen buckets, where each bucket’s hash chain is limited to at most two elements. The size (32 cache lines in this case) and the associativity (two in this case) are collectively called the cache’s “geometry”. Since this cache is implemented in hardware, the hash function is extremely simple: extract four bits from the memory address.

In Figure C.2, each box corresponds to a cache entry, which can contain a 256-byte cache line. However, a cache entry can be empty, as indicated by the empty boxes in the figure. The rest of the boxes are flagged with the memory address of the cache line that they contain. Since the cache lines must be 256-byte aligned, the low eight bits of each address are zero, and the choice of hardware hash function means that the next-higher four bits match the hash line number.

The situation depicted in the figure might arise if the program’s code were located at address 0x43210E00 through 0x43210EFF, and this program accessed data sequentially from 0x12345000 through 0x12345EFF. Suppose that the program were now to access location 0x1234F00. This location hashes to line 0xF, and both ways of this line are empty, so the corresponding 256-byte line can be accommodated. If the program were to access location 0x1233000, which hashes to line 0x0, the corresponding 256-byte cache line can be accommodated in way 1. However, if the program were to access location 0x1233E00, which hashes to line 0xE, one of the existing lines must be ejected from the cache to make room for the new cache line. If this ejected line were accessed later, a cache miss would result. Such a cache miss is termed an “associativity miss”.

Thus far, we have been considering only cases where a CPU reads a data item. What happens when it does a write? Because it is important that all CPUs agree on the value of a given data item, before a given CPU writes to that data item, it must first cause it to be removed, or “invalidated”, from other CPUs’ caches. Once this invalidation has completed, the CPU may safely modify the data item. If the data item was present in this CPU’s cache, but was read-only, this process is termed a “write miss”. Once a given CPU has completed invalidating a given data item from other CPUs’ caches, that CPU may repeatedly write (and read) that data item.

Later, if one of the other CPUs attempts to access the data item, it will incur a cache miss, this time because the first CPU invalidated the item in order to write to it. This type of cache miss is termed a “communication miss”, since it is usually due to several CPUs using the data items to communicate (for example, a lock is a data item that is used to communicate among CPUs using a mutual-exclusion algorithm).

Clearly, much care must be taken to ensure that all CPUs maintain a coherent view of the data. With all this fetching, invalidating, and writing, it is easy to imagine data being lost or (perhaps worse) different CPUs having conflicting values for the same data item in their respective caches. These problems are prevented by “cache-coherency protocols”, described in the next section.
C.2 Cache-Coherence Protocols

Cache-coherency protocols manage cache-line states so as to prevent inconsistent or lost data. These protocols can be quite complex, with many tens of states, but for our purposes we need only concern ourselves with the four-state MESI cache-coherence protocol.

C.2.1 MESI States

MESI stands for “modified”, “exclusive”, “shared”, and “invalid”, the four states a given cache line can take on using this protocol. Caches using this protocol therefore maintain a two-bit state “tag” on each cache line in addition to that line’s physical address and data.

A line in the “modified” state has been subject to a recent memory store from the corresponding CPU, and the corresponding memory is guaranteed not to appear in any other CPU’s cache. Cache lines in the “modified” state can thus be said to be “owned” by the CPU. Because this cache holds the only up-to-date copy of the data, this cache is ultimately responsible for either writing it back to memory or handing it off to some other cache, and must do so before reusing this line to hold other data.

The “exclusive” state is very similar to the “modified” state, the single exception being that the cache line has not yet been modified by the corresponding CPU, which in turn means that the copy of the cache line’s data that resides in memory is up-to-date. However, since the CPU can store to this line at any time, without consulting other CPUs, a line in the “exclusive” state can still be said to be owned by the corresponding CPU. That said, because the corresponding value in memory is up to date, this cache can discard this data without writing it back to memory or handing it off to some other CPU.

A line in the “shared” state might be replicated in at least one other CPU’s cache, so that this CPU is not permitted to store to the line without first consulting with other CPUs. As with the “exclusive” state, because the corresponding value in memory is up to date, this cache can discard this data without writing it back to memory or handing it off to some other CPU.

A line in the “invalid” state is empty, in other words, it holds no data. When new data enters the cache, it is placed into a cache line that was in the “invalid” state if possible. This approach is preferred because replacing a line in any other state could result in an expensive cache miss should the replaced line be referenced in the future.

Since all CPUs must maintain a coherent view of the data carried in the cache lines, the cache-coherence protocol provides messages that coordinate the movement of cache lines through the system.

C.2.2 MESI Protocol Messages

Many of the transitions described in the previous section require communication among the CPUs. If the CPUs are on a single shared bus, the following messages suffice:

- Read: The “read” message contains the physical address of the cache line to be read.
- Read Response: The “read response” message contains the data requested by an earlier “read” message. This “read response” message might be supplied either by memory or by one of the other caches. For example, if one of the caches has the desired data in “modified” state, that cache must supply the “read response” message.
- Invalidate: The “invalidate” message contains the physical address of the cache line to be invalidated. All other caches must remove the corresponding data from their caches and respond.
- Invalidate Acknowledge: A CPU receiving an “invalidate” message must respond with an “invalidate acknowledge” message after removing the specified data from its cache.
- Read Invalidate: The “read invalidate” message contains the physical address of the cache line to be read, while at the same time directing other caches to remove the data. Hence, it is a combination of a “read” and an “invalidate”, as indicated by its name. A “read invalidate” message requires both a “read response” and a set of “invalidate acknowledge” messages in reply.
- Writeback: The “writeback” message contains both the address and the data to be written back to memory (and perhaps “snooped” into other CPUs’ caches along the way). This message permits caches to eject lines in the “modified” state as needed to make room for other data.

Interestingly enough, a shared-memory multiprocessor system really is a message-passing computer under the

\[ \text{See Culler et al. [CSG99] pages 670 and 671 for the nine-state and 26-state diagrams for SGI Origin2000 and Sequent (now IBM) NUMA-Q, respectively. Both diagrams are significantly simpler than real life.} \]
covers. This means that clusters of SMP machines that use distributed shared memory are using message passing to implement shared memory at two different levels of the system architecture.

Quick Quiz C.1: What happens if two CPUs attempt to invalidate the same cache line concurrently?

Quick Quiz C.2: When an “invalidate” message appears in a large multiprocessor, every CPU must give an “invalidate acknowledge” response. Wouldn’t the resulting “storm” of “invalidate acknowledge” responses totally saturate the system bus?

Quick Quiz C.3: If SMP machines are really using message passing anyway, why bother with SMP at all?

C.2.3 MESI State Diagram

A given cache line’s state changes as protocol messages are sent and received, as shown in Figure C.3.

![Figure C.3: MESI Cache-Coherency State Diagram](image)

The transition arcs in this figure are as follows:

- Transition (a): A cache line is written back to memory, but the CPU retains it in its cache and further retains the right to modify it. This transition requires a “writeback” message.

- Transition (b): The CPU writes to the cache line that it already had exclusive access to. This transition does not require any messages to be sent or received.

- Transition (c): The CPU receives a “read invalidate” message for a cache line that it has modified. The CPU must invalidate its local copy, then respond with both a “read response” and an “invalidate acknowledge” message, both sending the data to the requesting CPU and indicating that it no longer has a local copy.

- Transition (d): The CPU does a read-modify-write operation on a data item that was not present in its cache. It transmits a “read invalidate”, receiving the data via a “read response”. The CPU can complete the transition once it has also received a full set of “invalidate acknowledge” responses.

- Transition (e): The CPU does a read-modify-write operation on a data item that was previously read-only in its cache. It must transmit “invalidate” messages, and must wait for a full set of “invalidate acknowledge” responses before completing the transition.

- Transition (f): Some other CPU reads the cache line, and it is supplied from this CPU’s cache, which retains a read-only copy, possibly also writing it back to memory. This transition is initiated by the reception of a “read” message, and this CPU responds with a “read response” message containing the requested data.

- Transition (g): Some other CPU reads a data item in this cache line, and it is supplied either from this CPU’s cache or from memory. In either case, the CPU retains a read-only copy. This transition is initiated by the reception of a “read” message, and this CPU responds with a “read response” message containing the requested data.

- Transition (h): This CPU realizes that it will soon need to write to some data item in this cache line, and thus transmits an “invalidate” message. The CPU cannot complete the transition until it receives a full set of “invalidate acknowledge” responses. Alternatively, all other CPUs eject this cache line from their caches via “writeback” messages (presumably to make room for other cache lines), so that this CPU is the last CPU caching it.

- Transition (i): Some other CPU does an atomic read-modify-write operation on a data item in a cache line held only in this CPU’s cache, so this CPU invalidates it from its cache. This transition is initiated by the reception of a “read invalidate” message, and this CPU responds with both a “read response” and an “invalidate acknowledge” message.
• Transition (j): This CPU does a store to a data item in a cache line that was not in its cache, and thus transmits a “read invalidate” message. The CPU cannot complete the transition until it receives the “read response” and a full set of “invalidate acknowledge” messages. The cache line will presumably transition to “modified” state via transition (b) as soon as the actual store completes.

• Transition (k): This CPU loads a data item in a cache line that was not in its cache. The CPU transmits a “read” message, and completes the transition upon receiving the corresponding “read response”.

• Transition (l): Some other CPU does a store to a data item in this cache line, but holds this cache line in read-only state due to its being held in other CPUs’ caches (such as the current CPU’s cache). This transition is initiated by the reception of an “invalidate” message, and this CPU responds with an “invalidate acknowledge” message.

Quick Quiz C.4: How does the hardware handle the delayed transitions described above?

C.2.4 MESI Protocol Example

Let’s now look at this from the perspective of a cache line’s worth of data, initially residing in memory at address 0, as it travels through the various single-line direct-mapped caches in a four-CPU system. Table C.1 shows this flow of data, with the first column showing the sequence of operations, the second the CPU performing the operation, the third the operation being performed, the next four the state of each CPU’s cache line (memory address followed by MESI state), and the final two columns whether the corresponding memory contents are up to date (“V”) or not (“I”).

Initially, the CPU cache lines in which the data would reside are in the “invalid” state, and the data is valid in memory. When CPU 0 loads the data at address 0, it enters the “shared” state in CPU 0’s cache, and is still valid in memory. CPU 3 also loads the data at address 0, so that it is in the “shared” state in both CPUs’ caches, and is still valid in memory. Next CPU 0 loads some other cache line (at address 8), which forces the data at address 0 out of its cache via an invalidation, replacing it with the data at address 8. CPU 2 now does a load from address 0, but this CPU realizes that it will soon need to store to it, and so it uses a “read invalidate” message in order to gain an exclusive copy, invalidating it from CPU 3’s cache (though the copy in memory remains up to date). Next CPU 2 does its anticipated store, changing the state to “modified”. The copy of the data in memory is now out of date. CPU 1 does an atomic increment, using a “read invalidate” to snoop the data from CPU 2’s cache and invalidate it, so that the copy in CPU 1’s cache is in the “modified” state (and the copy in memory remains out of date). Finally, CPU 1 reads the cache line at address 8, which uses a “writeback” message to push address 0’s data back out to memory.

Note that we end with data in some of the CPU’s caches.

Quick Quiz C.5: What sequence of operations would put the CPUs’ caches all back into the “invalid” state?

C.3 Stores Result in Unnecessary Stalls

Although the cache structure shown in Figure C.1 provides good performance for repeated reads and writes from a given CPU to a given item of data, its performance for the first write to a given cache line is quite poor. To see this, consider Figure C.4, which shows a timeline of a write by CPU 0 to a cacheline held in CPU 1’s cache. Since CPU 0 must wait for the cache line to arrive before it can write to it, CPU 0 must stall for an extended period of time.\(^3\)

But there is no real reason to force CPU 0 to stall for so long — after all, regardless of what data happens to be in the cache line that CPU 1 sends it, CPU 0 is going to unconditionally overwrite it.

C.3.1 Store Buffers

One way to prevent this unnecessary stalling of writes is to add “store buffers” between each CPU and its cache, as shown in Figure C.5. With the addition of these store buffers, CPU 0 can simply record its write in its store buffer and continue executing. When the cache line does finally make its way from CPU 1 to CPU 0, the data will be moved from the store buffer to the cache line.

However, there are complications that must be addressed, which are covered in the next two sections.

\(^3\) The time required to transfer a cache line from one CPU’s cache to another’s is typically a few orders of magnitude more than that required to execute a simple register-to-register instruction.
APPENDIX C. WHY MEMORY BARRIERS?

### APPENDIX C.  WHY MEMORY BARRIERS?

<table>
<thead>
<tr>
<th>Sequence #</th>
<th>CPU #</th>
<th>Operation</th>
<th>CPU Cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Initial State</td>
<td>-/I -/I</td>
<td>V V</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Load 0/S</td>
<td>-/I -/I</td>
<td>V V</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>Load 0/S</td>
<td>-/I -/I</td>
<td>0/S V V</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>Invalidation</td>
<td>8/S -/I</td>
<td>0/S V V</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>RMW 8/S</td>
<td>-/I 0/E</td>
<td>V V</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>Store 8/S</td>
<td>-/I -/I</td>
<td>I V</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>Atomic Inc 8/S 0/M</td>
<td>-/I -/I</td>
<td>I V</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>Writeback 8/S 8/S</td>
<td>-/I -/I</td>
<td>V V</td>
</tr>
</tbody>
</table>

Table C.1: Cache Coherence Example

#### C.3.2 Store Forwarding

To see the first complication, a violation of self-consistency, consider the following code with variables “a" and “b" both initially zero, and with the cache line containing variable “a” initially owned by CPU 1 and that containing “b” initially owned by CPU 0:

```plaintext
1   a = 1;
2   b = a + 1;
3   assert(b == 2);
```

One would not expect the assertion to fail. However, if one were foolish enough to use the very simple architecture shown in Figure C.5, one would be surprised. Such a system could potentially see the following sequence of events:

1. CPU 0 starts executing the `a = 1`.
2. CPU 0 looks “a” up in the cache, and finds that it is missing.
3. CPU 0 therefore sends a “read invalidate” message in order to get exclusive ownership of the cache line containing “a”.
4. CPU 0 records the store to “a” in its store buffer.
5. CPU 1 receives the “read invalidate” message, and responds by transmitting the cache line and removing that cacheline from its cache.
6. CPU 0 starts executing the `b = a + 1`.
7. CPU 0 receives the cache line from CPU 1, which still has a value of zero for “a”.

![Figure C.4: Writes See Unnecessary Stalls](image)

![Figure C.5: Caches With Store Buffers](image)
C.3. STORES RESULT IN UNNECESSARY STALLS

8. CPU 0 loads “a” from its cache, finding the value zero.

9. CPU 0 applies the entry from its store queue to the newly arrived cache line, setting the value of “a” in its cache to one.

10. CPU 0 adds one to the value zero loaded for “a” above, and stores it into the cache line containing “b” (which we will assume is already owned by CPU 0).

11. CPU 0 executes `assert(b == 2)`, which fails.

The problem is that we have two copies of “a”, one in the cache and the other in the store buffer.

This example breaks a very important guarantee, namely that each CPU will always see its own operations as if they happened in program order. Breaking this guarantee is violently counter-intuitive to software types, so much so that the hardware guys took pity and implemented “store forwarding”, where each CPU refers to (or “snoops”) its store buffer as well as its cache when performing loads, as shown in Figure C.6. In other words, a given CPU’s stores are directly forwarded to its subsequent loads, without having to pass through the cache.

C.3.3 Store Buffers and Memory Barriers

To see the second complication, a violation of global memory ordering, consider the following code sequences with variables “a” and “b” initially zero:

    1 void foo(void)
    2 {
    3    a = 1;
    4    b = 1;
    5 }
    6
    7 void bar(void)
    8 {
    9    while (b == 0) continue;
    10    assert(a == 1);
    11 }

Suppose CPU 0 executes `foo()` and CPU 1 executes `bar()`. Suppose further that the cache line containing “a” resides only in CPU 1’s cache, and that the cache line containing “b” is owned by CPU 0. Then the sequence of operations might be as follows:

1. CPU 0 executes `a = 1`. The cache line is not in CPU 0’s cache, so CPU 0 places the new value of “a” in its store buffer and transmits a “read invalidate” message.

2. CPU 1 executes `while (b == 0) continue`, but the cache line containing “b” is not in its cache. It therefore transmits a “read” message.

3. CPU 0 executes `b = 1`. It already owns this cache line (in other words, the cache line is already in either the “modified” or the “exclusive” state), so it stores the new value of “b” in its cache line.

4. CPU 0 receives the “read” message, and transmits the cache line containing the now-updated value of “b” to CPU 1, also marking the line as “shared” in its own cache.

5. CPU 1 receives the cache line containing “b” and installs it in its cache.

6. CPU 1 can now finish executing `while (b == 0) continue`, and since it finds that the value of “b” is 1, it proceeds to the next statement.

7. CPU 1 executes the `assert(a == 1)`, and, since CPU 1 is working with the old value of “a”, this assertion fails.

Figure C.6: Caches With Store Forwarding

With store forwarding in place, item 8 in the above sequence would have found the correct value of 1 for “a” in the store buffer, so that the final value of “b” would have been 2, as one would hope.
APPENDIX C. WHY MEMORY BARRIERS?

8. CPU 1 receives the “read invalidate” message, and transmits the cache line containing “a” to CPU 0 and invalidates this cache line from its own cache. But it is too late.

9. CPU 0 receives the cache line containing “a” and applies the buffered store just in time to fall victim to CPU 1’s failed assertion.

Quick Quiz C.6: In step 1 above, why does CPU 0 need to issue a “read invalidate” rather than a simple “invalidate”? The hardware designers cannot help directly here, since the CPUs have no idea which variables are related, let alone how they might be related. Therefore, the hardware designers provide memory-barrier instructions to allow the software to tell the CPU about such relations. The program fragment must be updated to contain the memory barrier:

```c
void foo(void)
{
  a = 1;
  smp_mb();
  b = 1;
}

void bar(void)
{
  while (b == 0) continue;
  assert(a == 1);
}
```

The memory barrier `smp_mb()` will cause the CPU to flush its store buffer before applying each subsequent store to its variable’s cache line. The CPU could either simply stall until the store buffer was empty before proceeding, or it could use the store buffer to hold subsequent stores until all of the prior entries in the store buffer had been applied.

With this latter approach the sequence of operations might be as follows:

1. CPU 0 executes `a = 1`. The cache line is not in CPU 0’s cache, so CPU 0 places the new value of “a” in its store buffer and transmits a “read invalidate” message.

2. CPU 1 executes `while (b == 0) continue`, but the cache line containing “b” is not in its cache. It therefore transmits a “read” message.

3. CPU 0 executes `smp_mb()`, and marks all current store-buffer entries (namely, the `a = 1`).

4. CPU 0 executes `b = 1`. It already owns this cache line (in other words, the cache line is already in either the “modified” or the “exclusive” state), but there is a marked entry in the store buffer. Therefore, rather than store the new value of “b” in the cache line, it instead places it in the store buffer (but in an unmarked entry).

5. CPU 0 receives the “read” message, and transmits the cache line containing the original value of “b” to CPU 1. It also marks its own copy of this cache line as “shared”.

6. CPU 1 receives the cache line containing “b” and installs it in its cache.

7. CPU 1 can now load the value of “b”, but since it finds that the value of “b” is still 0, it repeats the `while` statement. The new value of “b” is safely hidden in CPU 0’s store buffer.

8. CPU 1 receives the “read invalidate” message, and transmits the cache line containing “a” to CPU 0 and invalidates this cache line from its own cache.

9. CPU 0 receives the cache line containing “a” and applies the buffered store, placing this line into the “modified” state.

10. Since the store to “a” was the only entry in the store buffer that was marked by the `smp_mb()`, CPU 0 can also store the new value of “b” — except for the fact that the cache line containing “b” is now in “shared” state.

11. CPU 0 therefore sends an “invalidate” message to CPU 1.

12. CPU 1 receives the “invalidate” message, invalidates the cache line containing “b” from its cache, and sends an “acknowledgement” message to CPU 0.

13. CPU 1 executes `while (b == 0) continue`, but the cache line containing “b” is not in its cache. It therefore transmits a “read” message to CPU 0.

14. CPU 0 receives the “acknowledgement” message, and puts the cache line containing “b” into the “exclusive” state. CPU 0 now stores the new value of “b” into the cache line.

15. CPU 0 receives the “read” message, and transmits the cache line containing the new value of “b” to
C.4. STORE SEQUENCES RESULT IN UNNECESSARY STALLS

Unfortunately, each store buffer must be relatively small, which means that a CPU executing a modest sequence of stores can fill its store buffer (for example, if all of them result in cache misses). At that point, the CPU must once again wait for invalidations to complete in order to drain its store buffer before it can continue executing. This same situation can arise immediately after a memory barrier, when all subsequent store instructions must wait for invalidations to complete, regardless of whether or not these stores result in cache misses.

This situation can be improved by making invalidate acknowledge messages arrive more quickly. One way of accomplishing this is to use per-CPU queues of invalidate messages, or “invalidate queues”.

C.4.1 Invalidate Queues

One reason that invalidate acknowledge messages can take so long is that they must ensure that the corresponding cache line is actually invalidated, and this invalidation can be delayed if the cache is busy, for example, if the CPU is intensively loading and storing data, all of which resides in the cache. In addition, if a large number of invalidate messages arrive in a short time period, a given CPU might fall behind in processing them, thus possibly stalling all the other CPUs.

However, the CPU need not actually invalidate the cache line before sending the acknowledgement. It could instead queue the invalidate message with the understanding that the message will be processed before the CPU sends any further messages regarding that cache line.

C.4.2 Invalidate Queues and Invalidate Acknowledge

Figure C.7 shows a system with invalidate queues. A CPU with an invalidate queue may acknowledge an invalidate message as soon as it is placed in the queue, instead of having to wait until the corresponding line is actually invalidated. Of course, the CPU must refer to its invalidate queue when preparing to transmit invalidation messages — if an entry for the corresponding cache line is in the invalidate queue, the CPU cannot immediately transmit the invalidate message; it must instead wait until the invalidate-queue entry has been processed.

As you can see, this process involves no small amount of bookkeeping. Even something intuitively simple, like “load the value of a” can involve lots of complex steps in silicon.

CPU 0

CPU 1

Store Buffer

Cache

Invalidate Queue

Interconnect

Memory

Figure C.7: Caches With Invalidate Queues

Placing an entry into the invalidate queue is essentially a promise by the CPU to process that entry before transmitting any MESI protocol messages regarding that cache line. As long as the corresponding data structures are not highly contended, the CPU will rarely be inconvenienced by such a promise.
However, the fact that invalidate messages can be buffered in the invalidate queue provides additional opportunity for memory-misordering, as discussed in the next section.

C.4.3 Invalidate Queues and Memory Barriers

Let us suppose that CPUs queue invalidation requests, but respond to them immediately. This approach minimizes the cache-invalidation latency seen by CPUs doing stores, but can defeat memory barriers, as seen in the following example.

Suppose the values of “a” and “b” are initially zero, that “a” is replicated read-only (MESI “shared” state), and that “b” is owned by CPU 0 (MESI “exclusive” or “modified” state). Then suppose that CPU 0 executes foo() while CPU 1 executes function bar() in the following code fragment:

```c
1 void foo(void)  
2 {  
3   a = 1;  
4   smp_mb();  
5   b = 1;  
6 }  
7 
8 void bar(void)  
9 {  
10  while (b == 0) continue;  
11  assert(a == 1);  
12 }
```

Then the sequence of operations might be as follows:

1. CPU 0 executes a = 1. The corresponding cache line is read-only in CPU 0’s cache, so CPU 0 places the new value of “a” in its store buffer and transmits an “invalidate” message in order to flush the corresponding cache line from CPU 1’s cache.

2. CPU 1 executes while (b == 0) continue, but the cache line containing “b” is not in its cache. It therefore transmits a “read” message.

3. CPU 1 receives CPU 0’s “invalidate” message, queues it, and immediately responds to it.

4. CPU 0 receives the response from CPU 1, and is therefore free to proceed past the smp_mb() on line 4 above, moving the value of “a” from its store buffer to its cache line.

5. CPU 0 executes b = 1. It already owns this cache line (in other words, the cache line is already in either the “modified” or the “exclusive” state), so it stores the new value of “b” in its cache line.

6. CPU 0 receives the “read” message, and transmits the cache line containing the now-updated value of “b” to CPU 1, also marking the line as “shared” in its own cache.

7. CPU 1 receives the cache line containing “b” and installs it in its cache.

8. CPU 1 can now finish executing while (b == 0) continue, and since it finds that the value of “b” is 1, it proceeds to the next statement.

9. CPU 1 executes the assert(a == 1), and, since the old value of “a” is still in CPU 1’s cache, this assertion fails.

10. Despite the assertion failure, CPU 1 processes the queued “invalidate” message, and (tardily) invalidates the cache line containing “a” from its own cache.

Quick Quiz C.7: In step 1 of the first scenario in Section C.4.3, why is an “invalidate” sent instead of a “read invalidate” message? Doesn’t CPU 0 need the values of the other variables that share this cache line with “a”? ■

There is clearly not much point in accelerating invalidation responses if doing so causes memory barriers to effectively be ignored. However, the memory-barrier instructions can interact with the invalidate queue, so that when a given CPU executes a memory barrier, it marks all the entries currently in its invalidate queue, and forces any subsequent load to wait until all marked entries have been applied to the CPU’s cache. Therefore, we can add a memory barrier to function bar as follows:

```c
1 void foo(void)  
2 {  
3   a = 1;  
4   smp_mb();  
5   b = 1;  
6 }  
7 
8 void bar(void)  
9 {  
10  while (b == 0) continue;  
11  smp_mb();  
12  assert(a == 1);  
13 }
```
Quick Quiz C.8: Say what??? Why do we need a memory barrier here, given that the CPU cannot possibly execute the `assert()` until after the `while` loop completes?

With this change, the sequence of operations might be as follows:

1. CPU 0 executes `a = 1`. The corresponding cache line is read-only in CPU 0’s cache, so CPU 0 places the new value of “a” in its store buffer and transmits an “invalidate” message in order to flush the corresponding cache line from CPU 1’s cache.

2. CPU 1 executes `while (b == 0) continue`, but the cache line containing “b” is not in its cache. It therefore transmits a “read” message.

3. CPU 1 receives CPU 0’s “invalidate” message, queues it, and immediately responds to it.

4. CPU 0 receives the response from CPU 1, and is therefore free to proceed past the `smp_mb()` on line 4 above, moving the value of “a” from its store buffer to its cache line.

5. CPU 0 executes `b = 1`. It already owns this cache line (in other words, the cache line is already in either the “modified” or the “exclusive” state), so it stores the new value of “b” in its cache line.

6. CPU 0 receives the “read” message, and transmits the cache line containing the now-updated value of “b” to CPU 1, also marking the line as “shared” in its own cache.

7. CPU 1 receives the cache line containing “b” and installs it in its cache.

8. CPU 1 can now finish executing `while (b == 0) continue`, and since it finds that the value of “b” is 1, it proceeds to the next statement, which is now a memory barrier.

9. CPU 1 must now stall until it processes all pre-existing messages in its invalidation queue.

10. CPU 1 now processes the queued “invalidate” message, and invalidates the cache line containing “a” from its own cache.

11. CPU 1 executes the `assert(a == 1)`, and, since the cache line containing “a” is no longer in CPU 1’s cache, it transmits a “read” message.

12. CPU 0 responds to this “read” message with the cache line containing the new value of “a”.

13. CPU 1 receives this cache line, which contains a value of 1 for “a”, so that the assertion does not trigger.

With much passing of MESI messages, the CPUs arrive at the correct answer. This section illustrates why CPU designers must be extremely careful with their cache-coherence optimizations.

C.5 Read and Write Memory Barriers

In the previous section, memory barriers were used to mark entries in both the store buffer and the invalidate queue. But in our code fragment, `foo()` had no reason to do anything with the invalidate queue, and `bar()` similarly had no reason to do anything with the store queue.

Many CPU architectures therefore provide weaker memory-barrier instructions that do only one or the other of these two. Roughly speaking, a “read memory barrier” marks only the invalidate queue and a “write memory barrier” marks only the store buffer, while a full-fledged memory barrier does both.

The effect of this is that a read memory barrier orders only loads on the CPU that executes it, so that all loads preceding the read memory barrier will appear to have completed before any load following the read memory barrier. Similarly, a write memory barrier orders only stores, again on the CPU that executes it, and again so that all stores preceding the write memory barrier will appear to have completed before any store following the write memory barrier. A full-fledged memory barrier orders both loads and stores, but again only on the CPU executing the memory barrier.

If we update `foo` and `bar` to use read and write memory barriers, they appear as follows:
APPENDIX C. WHY MEMORY BARRIERS?

1 void foo(void)
2 {
3   a = 1;
4   smp_wmb();
5   b = 1;
6 }
7
8 void bar(void)
9 {
10   while (b == 0) continue;
11   smp_rmb();
12   assert(a == 1);
13 }

Some computers have even more flavors of memory barriers, but understanding these three variants will provide a good introduction to memory barriers in general.

C.6 Example Memory-Barrier Sequences

This section presents some seductive but subtly broken uses of memory barriers. Although many of them will work most of the time, and some will work all the time on some specific CPUs, these uses must be avoided if the goal is to produce code that works reliably on all CPUs. To help us better see the subtle breakage, we first need to focus on an ordering-hostile architecture.

C.6.1 Ordering-Hostile Architecture

Paul has come across a number of ordering-hostile computer systems, but the nature of the hostility has always been extremely subtle, and understanding it has required detailed knowledge of the specific hardware. Rather than picking on a specific hardware vendor, and as a presumably attractive alternative to dragging the reader through detailed technical specifications, let us instead design a mythical but maximally memory-ordering-hostile computer architecture.4

This hardware must obey the following ordering constraints [McK05a, McK05b]:

1. Each CPU will always perceive its own memory accesses as occurring in program order.

2. CPUs will reorder a given operation with a store only if the two operations are referencing different locations.

3. All of a given CPU’s loads preceding a read memory barrier (smp_rmb()) will be perceived by all CPUs to precede any loads following that read memory barrier.

4. All of a given CPU’s stores preceding a write memory barrier (smp_wmb()) will be perceived by all CPUs to precede any stores following that write memory barrier.

5. All of a given CPU’s accesses (loads and stores) preceding a full memory barrier (smp_mb()) will be perceived by all CPUs to precede any accesses following that memory barrier.

Quick Quiz C.9: Does the guarantee that each CPU sees its own memory accesses in order also guarantee that each user-level thread will see its own memory accesses in order? Why or why not? ■

Imagine a large non-uniform cache architecture (NUCA) system that, in order to provide fair allocation of interconnect bandwidth to CPUs in a given node, provided per-CPU queues in each node’s interconnect interface, as shown in Figure C.8. Although a given CPU’s accesses are ordered as specified by memory barriers executed by that CPU, however, the relative order of a given pair of CPUs’ accesses could be severely reordered, as we will see.5

C.6.2 Example 1

Table C.2 shows three code fragments, executed concurrently by CPUs 0, 1, and 2. Each of “a”, “b”, and “c” are initially zero.

Suppose CPU 0 recently experienced many cache misses, so that its message queue is full, but that CPU 1 has been running exclusively within the cache, so that its message queue is empty. Then CPU 0’s assignment to “a” and “b” will appear in Node 0’s cache immediately (and thus be visible to CPU 1), but will be blocked behind CPU 0’s prior traffic. In contrast, CPU 1’s assignment to “c” will sail through CPU 1’s previously empty queue.

Quick Quiz C.9: Does the guarantee that each CPU sees its own memory accesses in order also guarantee that each user-level thread will see its own memory accesses in order? Why or why not? ■

Imagine a large non-uniform cache architecture (NUCA) system that, in order to provide fair allocation of interconnect bandwidth to CPUs in a given node, provided per-CPU queues in each node’s interconnect interface, as shown in Figure C.8. Although a given CPU’s accesses are ordered as specified by memory barriers executed by that CPU, however, the relative order of a given pair of CPUs’ accesses could be severely reordered, as we will see.

Quick Quiz C.9: Does the guarantee that each CPU sees its own memory accesses in order also guarantee that each user-level thread will see its own memory accesses in order? Why or why not? ■

Any real hardware architect or designer will no doubt be loudly calling for Ralph on the porcelain intercom, as they just might be just a bit upset about the prospect of working out which queue should handle a message involving a cache line that both CPUs accessed, to say nothing of the many races that this example poses. All I can say is “Give me a better example”.

4 Readers preferring a detailed look at real hardware architectures are encouraged to consult CPU vendors’ manuals [SW95, Adv02, Int02b, IBM94, LSH02, SPA94, Int04b, Int04a, Int04c], Gharachorloo’s dissertation [Gha95], or Peter Sewell’s work [Sew].

5 Any real hardware architect or designer will no doubt be loudly calling for Ralph on the porcelain intercom, as they just might be just a bit upset about the prospect of working out which queue should handle a message involving a cache line that both CPUs accessed, to say nothing of the many races that this example poses. All I can say is “Give me a better example”.

Quick Quiz C.9: Does the guarantee that each CPU sees its own memory accesses in order also guarantee that each user-level thread will see its own memory accesses in order? Why or why not? ■

Imagine a large non-uniform cache architecture (NUCA) system that, in order to provide fair allocation of interconnect bandwidth to CPUs in a given node, provided per-CPU queues in each node’s interconnect interface, as shown in Figure C.8. Although a given CPU’s accesses are ordered as specified by memory barriers executed by that CPU, however, the relative order of a given pair of CPUs’ accesses could be severely reordered, as we will see.5

Quick Quiz C.9: Does the guarantee that each CPU sees its own memory accesses in order also guarantee that each user-level thread will see its own memory accesses in order? Why or why not? ■

Any real hardware architect or designer will no doubt be loudly calling for Ralph on the porcelain intercom, as they just might be just a bit upset about the prospect of working out which queue should handle a message involving a cache line that both CPUs accessed, to say nothing of the many races that this example poses. All I can say is “Give me a better example”.
C.6. EXAMPLE MEMORY-BARRIER SEQUENCES

C.6.3 Example 2

Table C.3 shows three code fragments, executed concurrently by CPUs 0, 1, and 2. All variables are initially zero.

<table>
<thead>
<tr>
<th>CPU 0</th>
<th>CPU 1</th>
<th>CPU 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 1; smp_wmb(); b = 1;</td>
<td>while (b == 0); c = 1;</td>
<td>z = c; smp_rmb(); x = a; assert(z == 0</td>
</tr>
</tbody>
</table>

Table C.2: Memory Barrier Example 1

Figure C.8: Example Ordering-Hostile Architecture

Therefore, CPU 2 might well see CPU 1’s assignment to “c” before it sees CPU 0’s assignment to “a”, causing the assertion to fire, despite the memory barriers.

In theory, portable code cannot rely on this example code sequence, however, in practice it actually does work on all mainstream computer systems.

Quick Quiz C.10: Could this code be fixed by inserting a memory barrier between CPU 1’s “while” and assignment to “c”? Why or why not?

C.6.4 Example 3

Table C.4 shows three code fragments, executed concurrently by CPUs 0, 1, and 2. All variables are initially zero.

<table>
<thead>
<tr>
<th>CPU 0</th>
<th>CPU 1</th>
<th>CPU 2</th>
<th>CPU 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Queue</td>
<td>Message Queue</td>
<td>Message Queue</td>
<td>Message Queue</td>
</tr>
</tbody>
</table>

Note that neither CPU 1 nor CPU 2 can proceed to line 5 until they see CPU 0’s assignment to “b” on line 3. Once CPU 1 and 2 have executed their memory barriers on line 4, they are both guaranteed to see all assignments by CPU 0 preceding its memory barrier on line 2. Similarly, CPU 0’s memory barrier on line 8 pairs with those of CPUs 1 and 2 on line 4, so that CPU 0 will not execute the assignment to “e” on line 9 until after its assignment to “a” is visible to both of the other CPUs. Therefore, CPU 2’s assertion on line 9 is guaranteed not to fire.

Quick Quiz C.11: Suppose that lines 3-5 for CPUs 1 and 2 in Table C.4 are in an interrupt handler, and that the CPU 2’s line 9 is run at process level. What changes, if any, are required to enable the code to work correctly, in other words, to prevent the assertion from firing?

Quick Quiz C.12: If CPU 2 executed an assert(e==0||c==1) in the example in Table C.4, would this assert ever trigger?

The Linux kernel’s synchronize_rcu() primitive uses an algorithm similar to that shown in this example.
APPENDIX C. WHY MEMORY BARRIERS?

<table>
<thead>
<tr>
<th>CPU 0</th>
<th>CPU 1</th>
<th>CPU 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 1;</td>
<td>while (a == 0);</td>
<td>y = b;</td>
</tr>
<tr>
<td></td>
<td>smp_mb();</td>
<td></td>
</tr>
<tr>
<td>b = 1;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>x = a;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
assert(y == 0 || x == 1);
``` 

Table C.3: Memory Barrier Example 2

<table>
<thead>
<tr>
<th>CPU 0</th>
<th>CPU 1</th>
<th>CPU 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 1;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>smp_wmb();</td>
<td></td>
</tr>
<tr>
<td>b = 1;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>while (c == 0);</td>
<td>while (b == 0);</td>
<td>while (b == 0);</td>
</tr>
<tr>
<td>while (d == 0);</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>smp_mb();</td>
<td></td>
</tr>
<tr>
<td>e = 1;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>assert(e == 0</td>
</tr>
</tbody>
</table>

Table C.4: Memory Barrier Example 3

C.7 Memory-Barrier Instructions

For Specific CPUs

Each CPU has its own peculiar memory-barrier instructions, which can make portability a challenge, as indicated by Table C.5. In fact, many software environments, including pthreads and Java, simply prohibit direct use of memory barriers, restricting the programmer to mutual-exclusion primitives that incorporate them to the extent that they are required. In the table, the first four columns indicate whether a given CPU allows the four possible combinations of loads and stores to be reordered. The next two columns indicate whether a given CPU allows loads and stores to be reordered with atomic instructions.

The seventh column, data-dependent reads reordered, requires some explanation, which is undertaken in the following section covering Alpha CPUs. The short version is that Alpha requires memory barriers for readers as well as updaters of linked data structures. Yes, this does mean that Alpha can in effect fetch the data pointed to before it fetches the pointer itself, strange but true. Please see: http://www.openvms.compaq.com/wizard/wiz_2637.html if you think that I am just making this up. The benefit of this extremely weak memory model is that Alpha can use simpler cache hardware, which in turn permitted higher clock frequency in Alpha’s heyday.

The last column indicates whether a given CPU has an incoherent instruction cache and pipeline. Such CPUs require special instructions be executed for self-modifying code.

Parenthesized CPU names indicate modes that are architecturally allowed, but rarely used in practice.

The common “just say no” approach to memory barriers can be eminently reasonable where it applies, but there are environments, such as the Linux kernel, where direct use of memory barriers is required. Therefore, Linux provides a carefully chosen least-common-denominator set of memory-barrier primitives, which are as follows:

- `smp_mb()` : “memory barrier” that orders both loads and stores. This means that loads and stores preceding the memory barrier will be committed to memory before any loads and stores following the memory barrier.

- `smp_rmb()` : “read memory barrier” that orders only loads.

- `smp_wmb()` : “write memory barrier” that orders only stores.

- `smp_read_barrier_depends()` that forces subsequent operations that depend on prior operations to be ordered. This primitive is a no-op on all platforms except Alpha.

- `mmiowb()` that forces ordering on MMIO writes that are guarded by global spinlocks. This primitive is a no-op on all platforms on which the memory barriers in spinlocks already enforce MMIO ordering.
C.7. MEMORY-BARRIER INSTRUCTIONS FOR SPECIFIC CPUS

Table C.5: Summary of Memory Ordering

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>AMD64</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>ARMv7-A/R</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>IA64</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>(PA-RISC)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>PA-RISC CPUs</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>POWER™</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>(SPARC RMO)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>(SPARC PSO)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>SPARC TSO</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>x86</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>(x86 OOSStore)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>zSeries®</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

The platforms with a non-no-op `mniowb()` definition include some (but not all) IA64, FRV, MIPS, and SH systems. This primitive is relatively new, so relatively few drivers take advantage of it.

The `smp_mb()`, `smp_rmb()`, and `smp_wmb()` primitives also force the compiler to eschew any optimizations that would have the effect of reordering memory optimizations across the barriers. The `smp_read_barrier_depends()` primitive has a similar effect, but only on Alpha CPUs. See Section 12.2 for more information on use of these primitives. These primitives generate code only in SMP kernels, however, each also has a UP version (`mb()`, `rmb()`, `wmb()`, and `read_barrier_depends()`, respectively) that generate a memory barrier even in UP kernels. The `smp_` versions should be used in most cases. However, these latter primitives are useful when writing drivers, because MMIO accesses must remain ordered even in UP kernels. In absence of memory-barrier instructions, both CPUs and compilers would happily rearrange these accesses, which at best would make the device act strangely, and could crash your kernel or, in some cases, even damage your hardware.

So most kernel programmers need not worry about the memory-barrier peculiarities of each and every CPU, as long as they stick to these interfaces. If you are working deep in a given CPU’s architecture-specific code, of course, all bets are off.

Furthermore, all of Linux’s locking primitives (spinlocks, reader-writer locks, semaphores, RCU, ...) include any needed barrier primitives. So if you are working with code that uses these primitives, you don’t even need to worry about Linux’s memory-ordering primitives.

That said, deep knowledge of each CPU’s memory-consistency model can be very helpful when debugging, to say nothing of when writing architecture-specific code or synchronization primitives.

Besides, they say that a little knowledge is a very dangerous thing. Just imagine the damage you could do with a lot of knowledge! For those who wish to understand more about individual CPUs’ memory consistency models, the next sections describes those of the most popular and prominent CPUs. Although nothing can replace actually reading a given CPU’s documentation, these sections give a good overview.
APPENDIX C. WHY MEMORY BARRIERS?

```c
struct el *insert(long key, long data)
{
    struct el *p;
    p = kmalloc(sizeof(*p), GFP_ATOMIC);
    spin_lock(&mutex);
    p->next = head.next;
    p->key = key;
    p->data = data;
    smp_wmb();
    head.next = p;
    spin_unlock(&mutex);
}

struct el *search(long key)
{
    struct el *p;
    p = head.next;
    while (p != &head) {
        /* BUG ON ALPHA!!! */
        if (p->key == key) {
            return (p);
        }
        p = p->next;
    }
    return (NULL);
}
```

Figure C.9: Insert and Lock-Free Search

C.7.1 Alpha

It may seem strange to say much of anything about a CPU whose end of life has been announced, but Alpha is interesting because, with the weakest memory ordering model, it reorders memory operations the most aggressively. It therefore has defined the Linux-kernel memory-ordering primitives, which must work on all CPUs, including Alpha. Understanding Alpha is therefore surprisingly important to the Linux kernel hacker.

The difference between Alpha and the other CPUs is illustrated by the code shown in Figure C.9. This `smp_wmb()` on line 9 of this figure guarantees that the element initialization in lines 6-8 is executed before the element is added to the list on line 10, so that the lock-free search will work correctly. That is, it makes this guarantee on all CPUs except Alpha.

Alpha has extremely weak memory ordering such that the code on line 20 of Figure C.9 could see the old garbage values that were present before the initialization on lines 6-8. However, this imposes unneeded overhead on systems (such as i386, IA64, PPC, and SPARC) that respect data dependencies on the read side. A `smp_read_barrier_depends()` primitive has been added to the Linux 2.6 kernel to eliminate overhead on these systems. This primitive may be used as shown on line 19 of Figure C.11.

One could place an `smp_rmb()` primitive between the pointer fetch and dereference. However, this imposes overhead on systems that respect data dependencies on the read side. A `smp_read_barrier_depends()` primitive has been added to the Linux 2.6 kernel to eliminate overhead on these systems. This primitive may be used as shown on line 19 of Figure C.11.

It is also possible to implement a software barrier that could be in place of `smp_wmb()`, which would force all reading CPUs to see the writing CPU's writes in order. However, this approach was deemed by the Linux community to impose excessive overhead on extremely weakly ordered CPUs such as Alpha. This software barrier could be implemented by sending inter-processor interrupts (IPIs) to all other CPUs. Upon receipt of such an IPI, a CPU would execute a memory-barrier instruction.

Figure C.10 shows how this can happen on an aggressively parallel machine with partitioned caches, so that alternating caches lines are processed by the different partitions of the caches. Assume that the list header `head` will be processed by cache bank 0, and that the new element will be processed by cache bank 1. On Alpha, the `smp_wmb()` will guarantee that the cache invalidates performed by lines 6-8 of Figure C.9 will reach the interconnect before that of line 10 does, but makes absolutely no guarantee about the order in which the new values will reach the reading CPU's core. For example, it is possible that the reading CPU's cache bank 1 is very busy, but cache bank 0 is idle. This could result in the cache invalidates for the new element being delayed, so that the reading CPU gets the new value for the pointer, but sees the old cached values for the new element. See the Web site called out earlier for more information, or, again, if you think that I am just making all this up.6

Figure C.10: Why `smp_read_barrier_depends()` is Required

---

6 Of course, the astute reader will have already recognized that Alpha is nowhere near as mean and nasty as it could be, the (thankfully) mythical architecture in Section C.6.1 being a case in point.
C.7. MEMORY-BARRIER INSTRUCTIONS FOR SPECIFIC CPUs

implementing a memory-barrier shootdown. Additional logic is required to avoid deadlocks. Of course, CPUs that respect data dependencies would define such a barrier to simply be `smp_wmb()`. Perhaps this decision should be revisited in the future as Alpha fades off into the sunset.

The Linux memory-barrier primitives took their names from the Alpha instructions, so `smp_mb()` is `mb`, `smp_rmb()` is `rmb`, and `smp_wmb()` is `wmb`. Alpha is the only CPU where `smp_read_barrier_depends()` is an `smp_mb()` rather than a no-op.

Quick Quiz C.13: Why is Alpha’s `smp_read_barrier_depends()` an `smp_mb()` rather than `smp_rmb()`?

For more detail on Alpha, see the reference manual [SW95].

C.7.2 AMD64

AMD64 is compatible with x86, and has recently updated its memory model [Adv07] to enforce the tighter ordering that actual implementations have provided for some time. The AMD64 implementation of the Linux `smp_mb()` primitive is `mfence`, `smp_rmb()` is `lfence`, and `smp_wmb()` is `sfence`. In theory, these might be relaxed, but any such relaxation must take SSE and 3DNow! instructions into account.

C.7.3 ARMv7-A/R

The ARM family of CPUs is extremely popular in embedded applications, particularly for power-constrained applications such as cellphones. There have nevertheless been multiprocessor implementations of ARM for more than five years. Its memory model is similar to that of Power (see Section C.7.6, but ARM uses a different set of memory-barrier instructions [ARM10]:

1. `DMB` (data memory barrier) causes the specified type of operations to appear to have completed before any subsequent operations of the same type. The “type” of operations can be all operations or can be restricted to only writes (similar to the Alpha `wmb` and the POWER `eieio` instructions). In addition, ARM allows cache coherence to have one of three scopes: single processor, a subset of the processors (“inner”) and global (“outer”).

2. `DSB` (data synchronization barrier) causes the specified type of operations to actually complete before any subsequent operations (of any type) are executed. The “type” of operations is the same as that of `DMB`. The `DSB` instruction was called `DWB` (drain write buffer or data write barrier, your choice) in early versions of the ARM architecture.

3. `ISB` (instruction synchronization barrier) flushes the CPU pipeline, so that all instructions following the `ISB` are fetched only after the `ISB` completes. For example, if you are writing a self-modifying program (such as a JIT), you should execute an `ISB` after between generating the code and executing it.

None of these instructions exactly match the semantics of Linux’s `rmb()` primitive, which must therefore be implemented as a full `DMB`. The `DMB` and `DSB` instructions have a recursive definition of accesses ordered before and after the barrier, which has an effect similar to that of POWER’s cumulativity.

ARM also implements control dependencies, so that if a conditional branch depends on a load, then any store executed after that conditional branch will be ordered after the load. However, loads following the conditional branch will not be guaranteed to be ordered unless there is an `ISB` instruction between the branch and the load. Consider the following example:
206

APPENDIX C. WHY MEMORY BARRIERS?

1 r1 = x;
2 if (r1 == 0)
3   nop();
4 y = 1;
5 r2 = z;
6 ISB();
7 r3 = z;

In this example, load-store control dependency ordering causes the load from x on line 1 to be ordered before the store to y on line 4. However, ARM does not respect load-load control dependencies, so that the load on line 1 might well happen after the load on line 5. On the other hand, the combination of the conditional branch on line 2 and the ISB instruction on line 6 ensures that the load on line 7 happens after the load on line 1. Note that inserting an additional ISB instruction somewhere between lines 3 and 4 would enforce ordering between lines 1 and 5.

C.7.4 IA64

IA64 offers a weak consistency model, so that in absence of explicit memory-barrier instructions, IA64 is within its rights to arbitrarily reorder memory references [Int02b]. IA64 has a memory-fence instruction named mf, but also has “half-memory fence” modifiers to loads, stores, and to some of its atomic instructions [Int02a]. The acq modifier prevents subsequent memory-reference instructions from being reordered before the acq, but permits prior memory-reference instructions to be reordered after the acq, as fancifully illustrated by Figure C.12. Similarly, the rel modifier prevents prior memory-reference instructions from being reordered after the rel, but allows subsequent memory-reference instructions to be reordered before the rel.

These half-memory fences are useful for critical sections, since it is safe to push operations into a critical section, but can be fatal to allow them to bleed out. However, as one of the only CPUs with this property, IA64 defines Linux’s semantics of memory ordering associated with lock acquisition and release.

The IA64 mf instruction is used for the smp_rmb(), smp_mb(), and smp_wmb() primitives in the Linux kernel. Oh, and despite rumors to the contrary, the “mf” mnemonic really does stand for “memory fence”.

Finally, IA64 offers a global total order for “release” operations, including the “mf” instruction. This provides the notion of transitivity, where if a given code fragment sees a given access as having happened, any later code fragment will also see that earlier access as having happened. Assuming, that is, that all the code fragments involved correctly use memory barriers.

C.7.5 PA-RISC

Although the PA-RISC architecture permits full reordering of loads and stores, actual CPUs run fully ordered [Kan96]. This means that the Linux kernel’s memory-ordering primitives generate no code, however, they do use the gcc memory attribute to disable compiler optimizations that would reorder code across the memory barrier.

C.7.6 POWER / PowerPC

The POWER and PowerPC® CPU families have a wide variety of memory-barrier instructions [IBM94, LSH02]:

1. sync causes all preceding operations to appear to have completed before any subsequent operations are started. This instruction is therefore quite expensive.

2. lwsync (light-weight sync) orders loads with respect to subsequent loads and stores, and also orders stores. However, it does not order stores with respect to subsequent loads. Interestingly enough, the lwsync instruction enforces the same ordering as does zSeries, and coincidentally, SPARC TSO.

3. eieio (enforce in-order execution of I/O, in case you were wondering) causes all preceding cacheable stores to appear to have completed before all subsequent stores. However, stores to cacheable memory

Figure C.12: Half Memory Barrier
are ordered separately from stores to non-cacheable memory, which means that eieio will not force an MMIO store to precede a spinlock release.

4. isync forces all preceding instructions to appear to have completed before any subsequent instructions start execution. This means that the preceding instructions must have progressed far enough that any traps they might generate have either happened or are guaranteed not to happen, and that any side-effects of these instructions (for example, page-table changes) are seen by the subsequent instructions.

Unfortunately, none of these instructions line up exactly with Linux’s wmb() primitive, which requires all stores to be ordered, but does not require the other high-overhead actions of the sync instruction. But there is no choice: ppc64 versions of wmb() and mb() are defined to be the heavyweight sync instruction. However, Linux’s smp_wmb() instruction is never used for MMIO (since a driver must carefully order MMIOs in UP as well as SMP kernels, after all), so it is defined to be the lighter weight eieio instruction. This instruction may well be unique in having a five-vowel mnemonic. The smp_mb() instruction is also defined to be the sync instruction, but both smp_rmb() and rmb() are defined to be the lighter-weight lwsync instruction.

Power features “cumulativity”, which can be used to obtain transitivity. When used properly, any code seeing the results of an earlier code fragment will also see the accesses that this earlier code fragment itself saw. Much more detail is available from McKenney and Silvera [MS09].

Power respects control dependencies in much the same way that ARM does, with the exception that the Power isync instruction is substituted for the ARM ISB instruction.

Many members of the POWER architecture have incoherent instruction caches, so that a store to memory will not necessarily be reflected in the instruction cache. Thankfully, few people write self-modifying code these days, but JITs and compilers do it all the time. Furthermore, recompiling a recently run program looks just like self-modifying code from the CPU’s viewpoint. The icbi instruction (instruction cache block invalidate) invalidates a specified cache line from the instruction cache, and may be used in these situations.

C.7.7 SPARC RMO, PSO, and TSO

Solaris on SPARC uses TSO (total-store order), as does Linux when built for the “sparc” 32-bit architecture. However, a 64-bit Linux kernel (the “sparc64” architecture) runs SPARC in RMO (relaxed-memory order) mode [SPA94]. The SPARC architecture also offers an intermediate PSO (partial store order). Any program that runs in RMO will also run in either PSO or TSO, and similarly, a program that runs in PSO will also run in TSO. Moving a shared-memory parallel program in the other direction may require careful insertion of memory barriers, although, as noted earlier, programs that make standard use of synchronization primitives need not worry about memory barriers.

SPARC has a very flexible memory-barrier instruction [SPA94] that permits fine-grained control of ordering:

- StoreStore: order preceding stores before subsequent stores. (This option is used by the Linux smp_wmb() primitive.)
- LoadStore: order preceding loads before subsequent stores.
- StoreLoad: order preceding stores before subsequent loads.
- LoadLoad: order preceding loads before subsequent loads. (This option is used by the Linux smp_rmb() primitive.)
- Sync: fully complete all preceding operations before starting any subsequent operations.
- MemIssue: complete preceding memory operations before subsequent memory operations, important for some instances of memory-mapped I/O.
- Lookaside: same as MemIssue, but only applies to preceding stores and subsequent loads, and even then only for stores and loads that access the same memory location.

The Linux smp_mb() primitive uses the first four options together, as in membar #LoadLoad | #LoadStore | #StoreStore | #StoreLoad, thus fully ordering memory operations.

So, why is membar #MemIssue needed? Because a membar #StoreLoad could permit a subsequent load to get its value from a write buffer, which would be disastrous if the write was to an MMIO register that induced
side effects on the value to be read. In contrast, membar #MemIssue would wait until the write buffers were flushed before permitting the loads to execute, thereby ensuring that the load actually gets its value from the MMIO register. Drivers could instead use membar #Sync, but the lighter-weight membar #MemIssue is preferred in cases where the additional function of the more-expensive membar #Sync are not required.

The membar #Lookaside is a lighter-weight version of membar #MemIssue, which is useful when writing to a given MMIO register affects the value that will next be read from that register. However, the heavier-weight membar #MemIssue must be used when a write to a given MMIO register affects the value that will next be read from some other MMIO register.

It is not clear why SPARC does not define wmb() to be membar #MemIssue and smb_wmb() to be membar #StoreStore, as the current definitions seem vulnerable to bugs in some drivers. It is quite possible that all the SPARC CPUs that Linux runs on implement a more conservative memory-ordering model than the architecture would permit.

SPARC requires a flush instruction be used between the time that an instruction is stored and executed [SPA94]. This is needed to flush any prior value for that location from the SPARC’s instruction cache. Note that flush takes an address, and will flush only that address from the instruction cache. On SMP systems, all CPUs’ caches are flushed, but there is no convenient way to determine when the off-CPU flushes complete, though there is a reference to an implementation note.

C.7.8 x86

Since the x86 CPUs provide “process ordering” so that all CPUs agree on the order of a given CPU’s writes to memory, the smp_wmb() primitive is a no-op for the CPU [Int04b]. However, a compiler directive is required to prevent the compiler from performing optimizations that would result in reordering across the smp_wmb() primitive.

On the other hand, x86 CPUs have traditionally given no ordering guarantees for loads, so the smp_mb() and smp_rmb() primitives expand to lock;add1. This atomic instruction acts as a barrier to both loads and stores.

More recently, Intel has published a memory model for x86 [Int07]. It turns out that Intel’s actual CPUs enforced tighter ordering than was claimed in the previous specifications, so this model is in effect simply mandating the earlier de-facto behavior. Even more recently, Intel published an updated memory model for x86 [Int11, Section 8.2], which mandates a total global order for stores, although individual CPUs are still permitted to see their own stores as having happened earlier than this total global order would indicate. This exception to the total ordering is needed to allow important hardware optimizations involving store buffers. In addition, memory ordering obeys causality, so that if CPU 0 sees a store by CPU 1, then CPU 0 is guaranteed to see all stores that CPU 1 saw prior to its store. Software may use atomic operations to override these hardware optimizations, which is one reason that atomic operations tend to be more expensive than their non-atomic counterparts. This total store order is not guaranteed on older processors.

It is also important to note that atomic instructions operating on a given memory location should all be of the same size [Int11, Section 8.1.2.2]. For example, if you write a program where one CPU atomically increments a byte while another CPU executes a 4-byte atomic increment on that same location, you are on your own.

However, note that some SSE instructions are weakly ordered (clflush and non-temporal move instructions [Int04a]). CPUs that have SSE can use mfence for smp_mb(), lfence for smp_rmb(), and sfence for smp_wmb().

A few versions of the x86 CPU have a mode bit that enables out-of-order stores, and for these CPUs, smp_wmb() must also be defined to be lock;add1.

Although many older x86 implementations accommodated self-modifying code without the need for any special instructions, newer revisions of the x86 architecture no longer requires x86 CPUs to be so accommodating. Interestingly enough, this relaxation comes just in time to inconvenience JIT implementors.

C.7.9 zSeries

The zSeries machines make up the IBM™ mainframe family, previously known as the 360, 370, and 390 [Int04c]. Parallelism came late to zSeries, but given that these mainframes first shipped in the mid 1960s, this is not saying much. The bcr 15,0 instruction is used for the Linux smp_mb(), smp_rmb(), and smp_wmb() primitives. It also has comparatively strong memory-ordering semantics, as shown in Table C.5, which should allow the smp_wmb() primitive to be a nop (and by the time you read this, this change may well have happened). The table
C.9. ADVICE TO HARDWARE DESIGNERS

actually understates the situation, as the zSeries memory model is otherwise sequentially consistent, meaning that all CPUs will agree on the order of unrelated stores from different CPUs.

As with most CPUs, the zSeries architecture does not guarantee a cache-coherent instruction stream, hence, self-modifying code must execute a serializing instruction between updating the instructions and executing them. That said, many actual zSeries machines do in fact accommodate self-modifying code without serializing instructions. The zSeries instruction set provides a large set of serializing instructions, including compare-and-swap, some types of branches (for example, the aforementioned \texttt{bcr 15,0} instruction), and test-and-set, among others.

C.8 Are Memory Barriers Forever?

There have been a number of recent systems that are significantly less aggressive about out-of-order execution in general and re-ordering memory references in particular. Will this trend continue to the point where memory barriers are a thing of the past?

The argument in favor would cite proposed massively multi-threaded hardware architectures, so that each thread would wait until memory was ready, with tens, hundreds, or even thousands of other threads making progress in the meantime. In such an architecture, there would be no need for memory barriers, because a given thread would simply wait for all outstanding operations to complete before proceeding to the next instruction. Because there would be potentially thousands of other threads, the CPU would be completely utilized, so no CPU time would be wasted.

The argument against would cite the extremely limited number of applications capable of scaling up to a thousand threads, as well as increasingly severe realtime requirements, which are in the tens of microseconds for some applications. The realtime-response requirements are difficult enough to meet as is, and would be even more difficult to meet given the extremely low single-threaded throughput implied by the massive multi-threaded scenarios.

Another argument in favor would cite increasingly sophisticated latency-hiding hardware implementation techniques that might well allow the CPU to provide the illusion of fully sequentially consistent execution while still providing almost all of the performance advantages of out-of-order execution. A counter-argument would cite the increasingly severe power-efficiency requirements presented both by battery-operated devices and by environmental responsibility.

Who is right? We have no clue, so are preparing to live with either scenario.

C.9 Advice to Hardware Designers

There are any number of things that hardware designers can do to make the lives of software people difficult. Here is a list of a few such things that we have encountered in the past, presented here in the hope that it might help prevent future such problems:

1. I/O devices that ignore cache coherence.

   This charming misfeature can result in DMAs from memory missing recent changes to the output buffer, or, just as bad, cause input buffers to be overwritten by the contents of CPU caches just after the DMA completes. To make your system work in face of such misbehavior, you must carefully flush the CPU caches of any location in any DMA buffer before presenting that buffer to the I/O device. And even then, you need to be very careful to avoid pointer bugs, as even a misplaced read to an input buffer can result in corrupting the data input!

2. External busses that fail to transmit cache-coherence data.

   This is an even more painful variant of the above problem, but causes groups of devices—and even memory itself—to fail to respect cache coherence. It is my painful duty to inform you that as embedded systems move to multicore architectures, we will no doubt see a fair number of such problems arise. Hopefully these problems will clear up by the year 2015.

3. Device interrupts that ignore cache coherence.

   This might sound innocent enough — after all, interrupts aren’t memory references, are they? But imagine a CPU with a split cache, one bank of which is extremely busy, therefore holding onto the last cacheline of the input buffer. If the corresponding I/O-complete interrupt reaches this CPU, then that CPU’s memory reference to the last cache line of the buffer could return old data, again resulting in data corruption, but in a form that will be invisible
in a later crash dump. By the time the system gets around to dumping the offending input buffer, the DMA will most likely have completed.

4. Inter-processor interrupts (IPIs) that ignore cache coherence.
   This can be problematic if the IPI reaches its destination before all of the cache lines in the corresponding message buffer have been committed to memory.

5. Context switches that get ahead of cache coherence.
   If memory accesses can complete too wildly out of order, then context switches can be quite harrowing. If the task flits from one CPU to another before all the memory accesses visible to the source CPU make it to the destination CPU, then the task could easily see the corresponding variables revert to prior values, which can fatally confuse most algorithms.

6. Overly kind simulators and emulators.
   It is difficult to write simulators or emulators that force memory re-ordering, so software that runs just fine in these environments can get a nasty surprise when it first runs on the real hardware. Unfortunately, it is still the rule that the hardware is more devious than are the simulators and emulators, but we hope that this situation changes.

   Again, we encourage hardware designers to avoid these practices!
Appendix D

Read-Copy Update Implementations

This appendix describes several fully functional production-quality RCU implementations. Understanding of these implementations requires a thorough understanding of the material in Chapters 1 and 8, as well as a reasonably good understanding of the Linux kernel, the latter of which may be found in several textbooks and websites [BC05, CRKH05, Cor08, Lov05].

If you are new to RCU implementations, you should start with the simpler “toy” RCU implementations that may be found in Section 8.3.5.

Section D.1 presents “Sleepable RCU”, or SRCU, which allows SRCU readers to sleep arbitrarily. This is a simple implementation, as production-quality RCU implementations go, and a good place to start learning about such implementations.

Section D.2 gives an overview of a highly scalable implementation of Classic RCU, designed for SMP systems sporting thousands of CPUs. Section D.3 takes the reader on a code walkthrough of this same implementation (as of late 2008).

Finally, Section D.4 provides a detailed view of the preemptible RCU implementation used in real-time systems.

D.1 Sleepable RCU Implementation

Classic RCU requires that read-side critical sections obey the same rules obeyed by the critical sections of pure spinlocks: blocking or sleeping of any sort is strictly prohibited. This has frequently been an obstacle to the use of RCU, and Paul has received numerous requests for a “sleepable RCU” (SRCU) that permits arbitrary sleeping (or blocking) within RCU read-side critical sections. Paul had previously rejected all such requests as unworkable, since arbitrary sleeping in RCU read-side could indefinitely extend grace periods, which in turn could result in arbitrarily large amounts of memory awaiting the end of a grace period, which finally would result in disaster, as fancifully depicted in Figure D.1, with the most likely disaster being hangs due to memory exhaustion. After all, any concurrency-control primitive that could result in system hangs — even when used correctly — does not deserve to exist.

However, the realtime kernels that require spinlock critical sections be preemptible [Mol05] also require that RCU read-side critical sections be preemptible [MS05]. Preemptible critical sections in turn require that lock-acquisition primitives block in order to avoid deadlock, which in turns means that both RCU’s and spinlocks’ critical sections be able to block awaiting a lock. However, these two forms of sleeping have the special property that priority boosting and priority inheritance may be used to awaken the sleeping tasks in short order.

Nevertheless, use of RCU in realtime kernels was the...
first crack in the tablets of stone on which were inscribed “RCU read-side critical sections can never sleep”. That said, indefinite sleeping, such as blocking waiting for an incoming TCP connection, is strictly verboten even in realtime kernels.

Quick Quiz D.1: Why is sleeping prohibited within Classic RCU read-side critical sections? ■

Quick Quiz D.2: Why not permit sleeping in Classic RCU read-side critical sections by eliminating context switch as a quiescent state, leaving user-mode execution and idle loop as the remaining quiescent states? ■

D.1.1 SRCU Implementation Strategy

The primary challenge in designing an SRCU is to prevent any given task sleeping in an RCU read-side critical section from blocking an unbounded number of RCU callbacks. SRCU uses two strategies to achieve this goal:

1. refusing to provide asynchronous grace-period interfaces, such as the Classic RCU’s call_rcu() API, and

2. isolating grace-period detection within each subsystem using SRCU.

The rationale for these strategies are discussed in the following sections.

D.1.1.1 Abolish Asynchronous Grace-Period APIs

The problem with the call_rcu() API is that a single thread can generate an arbitrarily large number of blocks of memory awaiting a grace period, as illustrated by the following:

```c
1 while (p = kmalloc(sizeof(*p), GFP_ATOMIC))
2   call_rcu(&p->rcu, f);
```

In contrast, the analogous code using synchronize_rcu() can have at most a single block of memory per thread awaiting a grace period:

```c
1 while (p = kmalloc(sizeof(*p), GFP_ATOMIC)) {
2   synchronize_rcu();
3   kfree(&p->rcu, f);
}
```

Therefore, SRCU provides an equivalent to synchronize_rcu(), but not to call_rcu().

D.1.1.2 Isolate Grace-Period Detection

In Classic RCU, a single read-side critical section could indefinitely delay all RCU callbacks, for example, as follows:

```c
1 /* BUGGY: Do not use!! */
2   rcu_read_lock();
3   schedule_timeout_interruptible(longdelay);
4   rcu_read_unlock();
```

This sort of behavior might be tolerated if RCU were used only within a single subsystem that was carefully designed to withstand long-term delay of grace periods. It is the fact that a single RCU read-side bug in one isolated subsystem can delay all users of RCU that forced these long-term RCU read-side delays to be abolished.

One way around this issue is for grace-period detection to be performed on a subsystem-by-subsystem basis, so that a lethargic RCU reader will delay grace periods only within that reader’s subsystem. Since each subsystem can have only a bounded number of memory blocks awaiting a grace period, and since the number of subsystems is also presumably bounded, the total amount of memory awaiting a grace period will also be bounded. The designer of a given subsystem is responsible for: (1) ensuring that SRCU read-side sleeping is bounded and (2) limiting the amount of memory waiting for synchronize_srcu().

This is precisely the approach that SRCU takes, as described in the following section.

D.1.2 SRCU API and Usage

The SRCU API is shown in Figure D.2. The following sections describe how to use it.

```c
int init_srcu_struct(struct srcu_struct *sp);
void cleanup_srcu_struct(struct srcu_struct *sp);
int srcu_read_lock(struct srcu_struct *sp);
void srcu_read_unlock(struct srcu_struct *sp, int idx);
void synchronize_srcu(struct srcu_struct *sp);
long srcu_batches_completed(struct srcu_struct *sp);
```

Figure D.2: SRCU API

D.1.2.1 Initialization and Cleanup

Each subsystem using SRCU must create an struct srcu_struct, either by declaring a variable of this type:

```c
struct srcu_struct srcu;
```
D.1. SLEEPABLE RCU IMPLEMENTATION

213

type or by dynamically allocating the memory, for example, via kmalloc(). Once this structure is in place, it must be initialized via init_srcu_struct(), which returns zero for success or an error code for failure (for example, upon memory exhaustion).

If the struct srcu_struct is dynamically allocated, then cleanup_srcu_struct() must be called before it is freed. Similarly, if the struct srcu_struct is a variable declared within a Linux kernel module, then cleanup_srcu_struct() must be called before the module is unloaded. Either way, the caller must take care to ensure that all SRCU read-side critical sections have completed (and that no more will commence) before calling cleanup_srcu_struct(). One way to accomplish this is described in Section D.1.2.4.

D.1.2.2 Read-Side Primitives

The read-side srcu_read_lock() and srcu_read_unlock() primitives are used as shown:

```c
1 idx = srcu_read_lock(&ss);
2 /* read-side critical section. */
3 srcu_read_unlock(&ss, idx);
```

The ss variable is the struct srcu_struct whose initialization was described in Section D.1.2.1, and the idx variable is an integer that in effect tells srcu_read_unlock() the grace period during which the corresponding srcu_read_lock() started.

This carrying of an index is a departure from the RCU API, which, when required, stores the equivalent information in the task structure. However, since a given task could potentially occupy an arbitrarily large number of nested SRCU read-side critical sections, SRCU cannot reasonably store this index in the task structure.

D.1.2.3 Update-Side Primitives

The synchronize_srcu() primitives may be used as shown below:

```c
1 list_del_rcu(p);
2 synchronize_srcu(&ss);
3 kfree(p);
```

As one might expect by analogy with Classic RCU, this primitive blocks until until after the completion of all SRCU read-side critical sections that started before the synchronize_srcu() started, as shown in Table D.1. Here, CPU 1 need only wait for the completion of CPU 0’s SRCU read-side critical section. It need not wait for the completion of CPU 2’s SRCU read-side critical section, because CPU 2 did not start this critical section until after CPU 1 began executing synchronize_srcu(). Finally, CPU 1’s synchronize_srcu() need not wait for CPU 3’s SRCU read-side critical section, because CPU 3 is using s2 rather than s1 as its struct srcu_struct. CPU 3’s SRCU read-side critical section is thus related to a different set of grace periods than those of CPUs 0 and 2.

The srcu_batches_completed() primitive may be used to monitor the progress of a given struct srcu_struct’s grace periods. This primitive is used in “torture tests” that validate SRCU’s operation.

D.1.2.4 Cleaning Up Safely

Cleaning up SRCU safely can be a challenge, but fortunately many uses need not do so. For example, uses in operating-system kernels that are initialized at boot time need not be cleaned up. However, uses within loadable modules must clean up if the corresponding module is to be safely unloaded.

In some cases, such as the RCU torture module, only a small known set of threads are using the SRCU read-side primitives against a particular struct srcu_struct. In these cases, the module-exit code need only kill that set of threads, wait for them to exit, and then clean up.

In other cases, for example, for device drivers, any thread in the system might be using the SRCU read-side primitives. Although one could apply the method of the previous paragraph, this ends up being equivalent to a full reboot, which can be unattractive. Figure D.3 shows one way that cleanup could be accomplished without a reboot.

The readside() function overlaps an RCU and an SRCU read-side critical section, with the former running from lines 5-11 and the latter running from lines 10-13. The RCU read-side critical section uses Pure RCU [McK04] to guard the value of the nomoresrcu variable. If this variable is set, we are cleaning up, and therefore must not enter the SRCU read-side critical section, so we return -EINVAL instead. On the other hand, if we are not yet cleaning up, we proceed into the SRCU read-side critical section.

The cleanup() function first sets the nomoresrcu variable on line 19, but then must wait for all currently executing RCU read-side critical sections to complete via the synchronize_rcu() primitive on line 20. Once the cleanup() function reaches line 21, all calls to readside() that could possibly have seen nomoresrcu equal to zero must have already reached
### APPENDIX D. READ-COPY UPDATE IMPLEMENTATIONS

#### Table D.1: SRCU Update and Read-Side Critical Sections

<table>
<thead>
<tr>
<th></th>
<th>CPU 0</th>
<th>CPU 1</th>
<th>CPU 2</th>
<th>CPU 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>i0 = srcu_read_lock(&amp;s1)</code></td>
<td><code>synchronize_srcu(&amp;s1)</code></td>
<td><code>i3 = srcu_read_lock(&amp;s2)</code></td>
<td><code>synchronize_srcu(&amp;s2)</code></td>
</tr>
<tr>
<td>2</td>
<td><code>synchronize_srcu(&amp;s1)</code></td>
<td><code>enter</code></td>
<td><code>i2 = srcu_read_lock(&amp;s1)</code></td>
<td><code>exit</code></td>
</tr>
<tr>
<td>3</td>
<td><code>srcu_read_unlock(&amp;s1, i0)</code></td>
<td><code>synchronize_srcu(&amp;s1)</code></td>
<td><code>srcu_read_unlock(&amp;s1, i2)</code></td>
<td></td>
</tr>
</tbody>
</table>

#### Figure D.3: SRCU Safe Cleanup

```c
int readside(void)
{
  int idx;
  rcu_read_lock();
  if (nomoresrcu) {
    rcu_read_unlock();
    return -EINVAL;
  }
  idx = srcu_read_lock(&ss);
  rcu_read_unlock();
  /* SRCU read-side critical section. */
  srcu_read_unlock(&ss, idx);
  return 0;
}
void cleanup(void)
{
  nomoresrcu = 1;
  synchronize_rcu();
  synchronize_srcu(&ss);
  cleanup_srcu_struct(&ss);
}
```

struct srcu was initialized, and as shown in the diagram, its low-order bit is used to index the struct srcu_struct_array. The per_cpu_ref field points to the array, and the mutex field is used to permit but one synchronize_srcu() at a time to proceed.

```c
struct srcu_struct_array {
  int c[2];
};
struct srcu_struct {
  int completed;
  struct srcu_struct_array *per_cpu_ref;
  struct mutex mutex;
};
```

#### Figure D.4: SRCU Data Structures

#### Figure D.5: SRCU Data-Structure Diagram

D.1.3 Implementation

This section describes SRCU’s data structures, initialization and cleanup primitives, read-side primitives, and update-side primitives.

D.1.3.1 Data Structures

SRCU’s data structures are shown in Figure D.4, and are depicted schematically in Figure D.5. The completed field is a count of the number of grace periods since the struct srcu was initialized, and as shown in the diagram, its low-order bit is used to index the struct srcu_struct_array. The per_cpu_ref field points to the array, and the mutex field is used to permit but one synchronize_srcu() at a time to proceed.

```c
struct srcu_struct_array {
  int c[2];
};
struct srcu_struct {
  int completed;
  struct srcu_struct_array *per_cpu_ref;
  struct mutex mutex;
};
```

D.1.3.2 Initialization Implementation

SRCU’s initialization function, init_srcu_struct(), is shown in Figure D.6. This function simply initializes the fields in the struct srcu_struct, returning zero if initialization succeeds or -ENOMEM otherwise.
D.1. SLEEPABLE RCU IMPLEMENTATION

1 int init_srcu_struct(struct srcu_struct *sp)
2 {
3     sp->completed = 0;
4     mutex_init(&sp->mutex);
5     sp->per_cpu_ref =
6         alloc_percpu(struct srcu_struct_array);
7     return (sp->per_cpu_ref ? 0 : -ENOMEM);
8 }

Figure D.6: SRCU Initialization

SRCU’s cleanup functions are shown in Figure D.7. The main cleanup function, cleanup_srcu_struct(), is shown on lines 19-29 of this figure, however, it immediately invokes srcu_readers_active(), shown on lines 13-17 of this figure, to verify that there are no readers currently using this struct srcu_struct.

The srcu_readers_active() function simply returns the sum of srcu_readers_active_idx() on both possible indexes, while srcu_readers_active_idx(), as shown on lines 1-11, sums up the per-CPU counters corresponding to the specified index, returning the result.

If the value returned from srcu_readers_active() is non-zero, then cleanup_srcu_struct() issues a warning on line 24 and simply returns on lines 25 and 26, declining to destroy a struct srcu_struct that is still in use. Such a warning always indicates a bug, and given that the bug has been reported, it is better to allow the system to continue with a modest memory leak than to introduce possible memory corruption.

Otherwise, cleanup_srcu_struct() frees the array of per-CPU counters and NULLs the pointer on lines 27 and 28.

D.1.3.3 Read-Side Implementation

The code implementing srcu_read_lock() is shown in Figure D.8. This function has been carefully constructed to avoid the need for memory barriers and atomic instructions.

Lines 5 and 11 disable and re-enable preemption, in order to force the sequence of code to execute unpreempted on a single CPU. Line 6 picks up the bottom bit of the grace-period counter, which will be used to select which rank of per-CPU counters is to be used for this SRCU read-side critical section. The barrier() call on line 7 is a directive to the compiler that ensures that the index is fetched but once, so that the index used on line 9 is the same one returned on line 12. Lines 8-9 increment the selected counter for the current CPU.3 Line 10 forces subsequent execution to occur after lines 8-9, in order to prevent misordering of any code in a non-CONFIG_PREEMPT build, but only from the perspective of an intervening interrupt handler. However, in a CONFIG_PREEMPT kernel, the required barrier() call is embedded in the preempt_enable() on line 11, so the srcu_barrier() is a no-op in that case. Finally, line 12 returns the index so that it may be passed in to the corresponding srcu_read_unlock().

The code for srcu_read_unlock() is shown in Figure D.9. Again, lines 3 and 7 disable and re-enable preemption so that the whole code sequence executes unpreempted on a single CPU. In CONFIG_PREEMPT kernels, the preempt_disable() on line 3 contains a barrier() primitive, otherwise, the barrier() is supplied by line 4. Again, this directive forces the barrier() has absolutely no effect on the CPU’s ability to reorder execution of both code and of memory accesses.

2 Please note that, despite the name, barrier() has absolutely no effect on the CPU’s ability to reorder execution of both code and of memory accesses.

3 It is important to note that the smp_processor_id() primitive has long-term meaning only if preemption is disabled. In absence of preemption disabling, a potential preemption immediately following execution of this primitive could cause the subsequent code to execute on some other CPU.
int srcu_read_lock(struct srcu_struct *sp) {
    int idx;
    preempt_disable();
    idx = sp->completed & 0x1;
    barrier();
    per_cpu_ptr(sp->per_cpu_ref, smp_processor_id())->c[idx]++;
    srcu_barrier();
    preempt_enable();
    return idx;
}

void srcu_read_unlock(struct srcu_struct *sp, int idx) {
    preempt_disable();
    srcu_barrier();
    per_cpu_ptr(sp->per_cpu_ref, smp_processor_id())->c[idx]--;
    preempt_enable();
}

Figure D.8: SRCU Read-Side Acquisition

The key point is that a given CPU’s counters can be observed by other CPUs only in cooperation with that CPU’s interrupt handlers. These interrupt handlers are responsible for ensuring that any needed memory barriers are executed prior to observing the counters.

D.1.3.4 Update-Side Implementation

The key point behind SRCU is that synchronize_sched() blocks until all currently-executing preempt-disabled regions of code complete. The synchronize_srcu() primitive makes heavy use of this effect, as can be seen in Figure D.10.

Line 5 takes a snapshot of the grace-period counter. Line 6 acquires the mutex, and lines 7-10 check to see whether at least two grace periods have elapsed since the snapshot, and, if so, releases the lock and returns — in this case, someone else has done our work for us. Otherwise, line 11 guarantees that any other CPU that sees the incremented value of the grace period counter in srcu_read_lock() also sees any changes made by this CPU prior to entering synchronize_srcu().

This guarantee is required to make sure that any SRCU read-side critical sections not blocking the next grace period have seen any prior changes.

Line 12 fetches the bottom bit of the grace-period counter for later use as an index into the per-CPU counter arrays, and then line 13 increments the grace-period counter. Line 14 then waits for any currently-executing srcu_read_lock() to complete, so that by the time that we reach line 15, all extant instances of srcu_read_lock() will be using the updated value from sp->completed. Therefore, the counters sampled in by srcu_readers_active_idx() on line 15 are guaranteed to be monotonically decreasing, so that once their sum reaches zero, it is guaranteed to stay there.

However, there are no memory barriers in the srcu_read_unlock() primitive, so the CPU is within its rights to reorder the counter decrement up into the SRCU critical section, so that references to an SRCU-protected data structure could in effect “bleed out” of the SRCU critical section. This scenario is addressed by the synchronize_sched() on line 17, which blocks until all other CPUs executing in preempt_disable() code sequences (such as that in srcu_read_unlock()) complete these sequences. Because completion of a given preempt_disable() code sequence is observed from the CPU executing that sequence, completion of the sequence implies completion of any prior SRCU read-side critical section. Any required memory barriers are supplied by the code making the observation.

At this point, it is therefore safe to release the mutex as shown on line 18 and return to the caller, who can now be assured that all SRCU read-side critical sections sharing the same struct srcu_struct will observe any update made prior to the call to synchronize_srcu().

Quick Quiz D.3: Why is it OK to assume that updates separated by synchronize_sched() will be performed in order? ■

Quick Quiz D.4: Why must line 17 in synchronize_srcu() (Figure D.10) precede the release of the mutex on line 18? What would have to change to permit these two lines to be interchanged? Would such a change be worthwhile? Why or why not? ■

D.1.4 SRCU Summary

SRCU provides an RCU-like set of primitives that permit general sleeping in the SRCU read-side critical sections.
1 void synchronize_srcu(struct srcu_struct *sp)
2 {
3     int idx;
4
5     idx = sp->completed;
6     mutex_lock(&sp->mutex);
7     if ((sp->completed - idx) >= 2) {
8         mutex_unlock(&sp->mutex);
9         return;
10     }
11     synchronize_sched();
12     idx = sp->completed & 0x1;
13     sp->completed++;
14     synchronize_sched();
15     while (srcu_readers_active_idx(sp, idx))
16         schedule_timeout_interruptible(1);
17     synchronize_sched();
18     mutex_unlock(&sp->mutex);
19 }

Figure D.10: SRCU Update-Side Implementation

However, it is important to note that SRCU has been used only in prototype code, though it has passed the RCU torture test. It will be very interesting to see what use, if any, SRCU sees in the future.

D.2 Hierarchical RCU Overview

Although Classic RCU’s read-side primitives enjoy excellent performance and scalability, the update-side primitives, which determine when pre-existing read-side critical sections have finished, were designed with only a few tens of CPUs in mind. Their scalability is limited by a global lock that must be acquired by each CPU at least once during each grace period. Although Classic RCU actually scales to a couple of hundred CPUs, and can be tweaked to scale to roughly a thousand CPUs (but at the expense of extending grace periods), emerging multicore systems will require it to scale better.

In addition, Classic RCU has a sub-optimal dynticks interface, with the result that Classic RCU will wake up every CPU at least once per grace period. To see the problem with this, consider a 16-CPU system that is sufficiently lightly loaded that it is keeping only four CPUs busy. In a perfect world, the remaining twelve CPUs could be put into deep sleep mode in order to conserve energy. Unfortunately, if the four busy CPUs are frequently performing RCU updates, those twelve idle CPUs will be awakened frequently, wasting significant energy. Thus, any major change to Classic RCU should also leave sleeping CPUs lie.

Both the classic and the hierarchical implementations have Classic RCU semantics and identical APIs, however, the old implementation will be called “classic RCU” and the new implementation will be called “hierarchical RCU”.

@ @ @ roadmap @ @ @

D.2.1 Review of RCU Fundamentals

In its most basic form, RCU is a way of waiting for things to finish. Of course, there are a great many other ways of waiting for things to finish, including reference counts, reader-writer locks, events, and so on. The great advantage of RCU is that it can wait for each of (say) 20,000 different things without having to explicitly track each and every one of them, and without having to worry about the performance degradation, scalability limitations, complex deadlock scenarios, and memory-leak hazards that are inherent in schemes using explicit tracking.

In RCU’s case, the things waited on are called “RCU read-side critical sections”. An RCU read-side critical section starts with an rcu_read_lock() primitive, and ends with a corresponding rcu_read_unlock() primitive. RCU read-side critical sections can be nested, and may contain pretty much any code, as long as that code does not explicitly block or sleep (although a special form of RCU called SRCU, described in Section D.1 does permit general sleeping in SRCU read-side critical sections). If you abide by these conventions, you can use RCU to wait for any desired piece of code to complete.

RCU accomplishes this feat by indirectly determining when these other things have finished, as has been described elsewhere [MS98] for classic RCU and Section D.4 for preemptible RCU.

In particular, as shown in the Figure 8.17 on page 8.17, RCU is a way of waiting for pre-existing RCU read-side critical sections to completely finish, also including the memory operations executed by those critical sections. However, note that RCU read-side critical sections that begin after the beginning of a given grace period can and will extend beyond the end of that grace period.

The following section gives a very high-level view of how the Classic RCU implementation operates.

D.2.2 Brief Overview of Classic RCU Implementation

The key concept behind the Classic RCU implementation is that Classic RCU read-side critical sections are confined to kernel code and are not permitted to block. This means that any time a given CPU is seen either blocking, in the
idle loop, or exiting the kernel, we know that all RCU
read-side critical sections that were previously running
on that CPU must have completed. Such states are called
“quiescent states”, and after each CPU has passed through
at least one quiescent state, the RCU grace period ends.

2. Reliable, so that RCU supports 24x7 operation for
years at a time.

3. Callable from irq handlers.

4. Contained memory footprint, so that mechanisms
exist to expedite grace periods if there are too many
callbacks. (This is weakened from the LCA2005
list.)

5. Independent of memory blocks, so that RCU can
work with any conceivable memory allocator.

6. Synchronization-free read side, so that only normal
non-atomic instructions operating on CPU- or task-
local memory are permitted. (This is strengthened
from the LCA2005 list.)

7. Unconditional read-to-write upgrade, which is used
in several places in the Linux kernel where the
update-side lock is acquired within the RCU read-
side critical section.

8. Compatible API.

9. Because this is not to be a real-time RCU, the require-
ment for preemptible RCU read-side critical sections
can be dropped. However, we need to add the follow-
ing new requirements to account for changes over
the past few years.

10. Scalability with extremely low internal-to-RCU lock
contention. RCU must support at least 1,024 CPUs
gracefully, and preferably at least 4,096.

11. Energy conservation: RCU must be able to avoid
awakening low-power-state dynticks-idle CPUs, but
still determine when the current grace period ends.
This has been implemented in real-time RCU, but
needs serious simplification.

12. RCU read-side critical sections must be permitted
in NMI handlers as well as irq handlers. Note that
preemptible RCU was able to avoid this requirement
due to a separately implemented synchronize_
sched().

13. RCU must operate gracefully in face of repeated
CPU-hotplug operations. This is simply carrying
forward a requirement met by both classic and real-
time.

14. It must be possible to wait for all previously reg-
istered RCU callbacks to complete, though this is
already provided in the form of rcu_barrier().

Classic RCU’s most important data structure is
the rcu_ctrlblk structure, which contains the
->cpumask field, which contains one bit per CPU, as
shown in Figure D.11. Each CPU’s bit is set to one at the
beginning of each grace period, and each CPU must clear
its bit after it passes through a quiescent state. Because
multiple CPUs might want to clear their bits concurrently,
which would corrupt the ->cpumask field, a ->lock
spinlock is used to protect ->cpumask, preventing any
such corruption. Unfortunately, this spinlock can also
suffer extreme contention if there are more than a few
hundred CPUs, which might soon become quite common
if multicore trends continue. Worse yet, the fact that all
CPUs must clear their own bit means that CPUs are not
permitted to sleep through a grace period, which limits
Linux’s ability to conserve power.

The next section lays out what we need from a new
non-real-time RCU implementation.

D.2.3 RCU Desiderata

The list of real-time RCU desiderata [MS05] is a very
good start:

1. Deferred destruction, so that an RCU grace period
cannot end until all pre-existing RCU read-side criti-
cal sections have completed.
15. Detecting CPUs that are failing to respond is desirable, to assist diagnosis both of RCU and of various infinite loop bugs and hardware failures that can prevent RCU grace periods from ending.

16. Extreme expediting of RCU grace periods is desirable, so that an RCU grace period can be forced to complete within a few hundred microseconds of the last relevant RCU read-side critical second completing. However, such an operation would be expected to incur severe CPU overhead, and would be primarily useful when carrying out a long sequence of operations that each needed to wait for an RCU grace period.

The most pressing of the new requirements is the first one, scalability. The next section therefore describes how to make order-of-magnitude reductions in contention on RCU’s internal locks.

D.2.4 Towards a More Scalable RCU Implementation

![Hierarchical RCU State Diagram](image)

The tree of `rcu_node` structures is embedded into a linear array in the `rcu_state` structure, with the root of the tree in element zero, as shown in Figure D.13 for an eight-CPU system with a three-level hierarchy. Each arrow links a given `rcu_node` structure to its parent, representing the `rcu_node`'s `->parent` field. Each `rcu_node` indicates the range of CPUs covered, so that the root node covers all of the CPUs, each node in the second level covers half of the CPUs, and each node in the leaf level covering a pair of CPUs. This array is allocated statically at compile time based on the value of `NR_CPUS`.

The sequence of diagrams in Figure D.14 shows how grace periods are detected. In the first figure, no CPU has yet passed through a quiescent state, as indicated by the red rectangles. Suppose that all six CPUs simultaneously try to tell RCU that they have passed through a quiescent state. Only one of each pair will be able to acquire the lock on the corresponding lower `rcu_node`, and so the second figure shows the result if the lucky CPUs are numbers 0, 3, and 5, as indicated by the green rectangles. Once these lucky CPUs have finished, then the other CPUs will acquire the lock, as shown in the third figure. Each of these CPUs will see that they are the last in their group, and therefore all three will attempt to move to the upper `rcu_node`. Only one at a time can acquire the upper `rcu_node`’s lock, and the fourth, fifth, and sixth figures show the sequence of states assuming that CPU 1, CPU 2, and CPU 4 acquire the lock in that order. The sixth and final figure in the group shows that
all CPUs have passed through a quiescent state, so that the grace period has ended.

In the above sequence, there were never more than three CPUs contending for any one lock, in happy contrast to Classic RCU, where all six CPUs might contend. However, even more dramatic reductions in lock contention are possible with larger numbers of CPUs. Consider a hierarchy of \texttt{rcu_node} structures, with 64 lower structures and \(64 \times 64 = 4,096\) CPUs, as shown in Figure D.15.

Here each of the lower \texttt{rcu_node} structures’ locks are acquired by 64 CPUs, a 64-times reduction from the 4,096 CPUs that would acquire Classic RCU’s single global lock. Similarly, during a given grace period, only one CPU from each of the lower \texttt{rcu_node} structures will acquire the upper \texttt{rcu_node} structure’s lock, which is again a 64x reduction from the contention level that would be experienced by Classic RCU running on a 4,096-CPU system.

\textbf{Quick Quiz D.5:} Wait a minute! With all those new locks, how do you avoid deadlock? 

\textbf{Quick Quiz D.6:} Why stop at a 64-times reduction? Why not go for a few orders of magnitude instead? 

\textbf{Quick Quiz D.7:} But I don’t care about McKenney’s lame excuses in the answer to Quick Quiz 2!!! I want to get the number of CPUs contending on a single lock down to something reasonable, like sixteen or so!!!

The implementation maintains some per-CPU data, such as lists of RCU callbacks, organized into \texttt{rcu_data} structures. In addition, \texttt{rcu} (as in \texttt{call_rcu()}) and \texttt{rcu_bh} (as in \texttt{call_rcu_bh()}) each maintain their own hierarchy, as shown in Figure D.16.

\textbf{Quick Quiz D.8:} OK, so what is the story with the colors?
The next section discusses energy conservation.

**D.2.5 Towards a Greener RCU Implementation**

As noted earlier, an important goal of this effort is to leave sleeping CPUs lie in order to promote energy conservation. In contrast, classic RCU will happily awaken each and every sleeping CPU at least once per grace period in some cases, which is suboptimal in the case where a small number of CPUs are busy doing RCU updates and the majority of the CPUs are mostly idle. This situation occurs frequently in systems sized for peak loads, and we need to be able to accommodate it gracefully. Furthermore, we need to fix a long-standing bug in Classic RCU where a dynticks-idle CPU servicing an interrupt containing a long-running RCU read-side critical section will fail to prevent an RCU grace period from ending.

**Quick Quiz D.9:** Given such an egregious bug, why does Linux run at all?

This is accomplished by requiring that all CPUs manipulate counters located in a per-CPU `rcu_dynticks` structure. Loosely speaking, these counters have even-numbered values when the corresponding CPU is in dynticks idle mode, and have odd-numbered values otherwise. RCU thus needs to wait for quiescent states only for those CPUs whose `rcu_dynticks` counters are odd, and need not wake up sleeping CPUs, whose counters will be even. As shown in Figure D.17, each per-CPU `rcu_dynticks` structure is shared by the "rcu" and "rcu_bh" implementations.

The following section presents a high-level view of the RCU state machine.

**D.2.6 State Machine**

At a sufficiently high level, Linux-kernel RCU implementations can be thought of as high-level state machines as shown in Figure D.18. The common-case path through this state machine on a busy system goes through the two uppermost loops, initializing at the beginning of each grace period (GP), waiting for quiescent states (QS), and noting when each CPU passes through its first quiescent state for a given grace period. On such a system, quiescent states will occur on each context switch, or, for CPUs that are either idle or executing user-mode code, each scheduling-clock interrupt. CPU-hotplug events will take the state machine through the "CPU Offline" box, while the presence of "holdout" CPUs that fail to pass through quiescent states quickly enough will exercise the path through the "Send resched IPIs to Holdout CPUs" box. RCU implementations that avoid unnecessarily awakening dyntick-idle CPUs will mark those CPUs as being in an extended quiescent state, taking the "Y" branch out of the "CPUs in dyntick-idle Mode?" decision diamond (but note that CPUs in dyntick-idle mode will not be sent resched IPIs). Finally, if `CONFIG_RCU_CPU_STALL_DETECTOR` is enabled, truly excessive delays in reaching quiescent states will exercise the "Complain About Holdout CPUs" path.

**Quick Quiz D.10:** But doesn’t this state diagram indicate that dyntick-idle CPUs will get hit with reschedule IPIs? Won’t that wake them up?

The events in the above state schematic interact with different data structures, as shown in Figure D.19. However, the state schematic does not directly translate into C code for any of the RCU implementations. Instead,
these implementations are coded as an event-driven system within the kernel. Therefore, the following section describes some “use cases”, or ways in which the RCU algorithm traverses the above state schematic as well as the relevant data structures.

D.2.7 Use Cases

This section gives an overview of several “use cases” within the RCU implementation, listing the data structures touched and the functions invoked. The use cases are as follows:

1. Start a New Grace Period (Section D.2.7.1)
2. Pass Through a Quiescent State (Section D.2.7.2)
3. Announce a Quiescent State to RCU (Section D.2.7.3)

4. Enter and Leave Dynticks Idle Mode (Section D.2.7.4)

5. Interrupt from Dynticks Idle Mode (Section D.2.7.5)

6. NMI from Dynticks Idle Mode (Section D.2.7.6)

7. Note That a CPU is in Dynticks Idle Mode (Section D.2.7.7)

8. Offline a CPU (Section D.2.7.8)

9. Online a CPU (Section D.2.7.9)

10. Detect a Too-Long Grace Period (Section D.2.7.10)

Each of these use cases is described in the following sections.

D.2.7.1 Start a New Grace Period

The `rcu_start_gp()` function starts a new grace period. This function is invoked when a CPU having callbacks waiting for a grace period notices that no grace period is in progress.

The `rcu_start_gp()` function updates state in the `rcu_state` and `rcu_data` structures to note the newly started grace period, acquires the `->onoff` lock (and disables irqs) to exclude any concurrent CPU-hotplug operations, sets the bits in all of the `rcu_node` structures to indicate that all CPUs (including this one) must pass through a quiescent state, and finally releases the `->onoff` lock.

The bit-setting operation is carried out in two phases. First, the non-leaf `rcu_node` structures’ bits are set without holding any additional locks, and then finally each leaf `rcu_node` structure’s bits are set in turn while holding that structure’s `->lock`.

**Quick Quiz D.11:** But what happens if a CPU tries to report going through a quiescent state (by clearing its bit) before the bit-setting CPU has finished? ■

**Quick Quiz D.12:** And what happens if all CPUs try to report going through a quiescent state before the bit-setting CPU has finished, thus ending the new grace period before it starts? ■

D.2.7.2 Pass Through a Quiescent State

The rcu and rcu_bh flavors of RCU have different sets of quiescent states. Quiescent states for rcu are context switch, idle (either dynticks or the idle loop), and user-mode execution, while quiescent states for rcu_bh are any code outside of softirq with interrupts enabled. Note that an quiescent state for rcu is also a quiescent state for rcu_bh. Quiescent states for rcu are recorded by invoking `rcu_qsctr_inc()`, while quiescent states for rcu_bh are recorded by invoking `rcu_bh_qsctr_inc()`. These two functions record their state in the current CPU's `rcu_data` structure.

These functions are invoked from the scheduler, from `__do_softirq()`, and from `rcu_check_callbacks()`. This latter function is invoked from the scheduling-clock interrupt, and analyzes state to determine whether this interrupt occurred within a quiescent state, invoking `rcu_qsctr_inc()` and/or `rcu_bh_qsctr_inc()`, as appropriate. It also raises RCU_SOFTIRQ, which results in `rcu_process_callbacks()` being invoked on the current CPU at some later time from softirq context.

D.2.7.3 Announce a Quiescent State to RCU

The afore-mentioned `rcu_process_callbacks()` function has several duties:

1. Determining when to take measures to end an over-long grace period (via `force_quiescent_state()`).

2. Taking appropriate action when some other CPU detected the end of a grace period (via `rcu_process_gp_end()`). “Appropriate action” includes advancing this CPU’s callbacks and recording the new grace period. This same function updates state in response to some other CPU starting a new grace period.

3. Reporting the current CPU’s quiescent states to the core RCU mechanism (via `rcu_check_quiescent_state()`, which in turn invokes `cpu_quiet()`). This of course might mark the end of the current grace period.

4. Starting a new grace period if there is no grace period in progress and this CPU has RCU callbacks still waiting for a grace period (via `cpu_needs_another_gp()` and `rcu_start_gp()`).
5. Invoking any of this CPU’s callbacks whose grace period has ended (via rcu_do_batch()).

These interactions are carefully orchestrated in order to avoid buggy behavior such as reporting a quiescent state from the previous grace period against the current grace period.

D.2.7.4 Enter and Leave Dynticks Idle Mode

The scheduler invokes rcu_enter_nohz() to enter dynticks-idle mode, and invokes rcu_exit_nohz() to exit it. The rcu_enter_nohz() function increments a per-CPU dynticks_nesting variable and also a per-CPU dynticks counter, the latter of which must then have an even-numbered value. The rcu_exit_nohz() function decrements this same per-CPU dynticks_nesting variable, and again increments the per-CPU dynticks counter, the latter of which must then have an odd-numbered value.

The dynticks counter can be sampled by other CPUs. If the value is even, the first CPU is in an extended quiescent state. Similarly, if the counter value changes during a given grace period, the first CPU must have been in an extended quiescent state at some point during the grace period. However, there is another dynticks_nmi per-CPU variable that must also be sampled, as will be discussed below.

D.2.7.5 Interrupt from Dynticks Idle Mode

Interrupts from dynticks idle mode are handled by rcu_irq_enter() and rcu_irq_exit(). The rcu_irq_enter() function increments the per-CPU dynticks_nesting variable, and, if the prior value was zero, also increments the dynticks per-CPU variable (which must then have an odd-numbered value).

The rcu_irq_exit() function decrements the per-CPU dynticks_nesting variable, and, if the new value is zero, also increments the dynticks per-CPU variable (which must then have an even-numbered value).

Note that entering an irq handler exits dynticks idle mode and vice versa. This enter/exit anti-correspondence can cause much confusion. You have been warned.

D.2.7.6 NMI from Dynticks Idle Mode

NMIs from dynticks idle mode are handled by rcu_nmi_enter() and rcu_nmi_exit(). These functions both increment the dynticks_nmi counter, but only if the aforementioned dynticks counter is even. In other words, NMI’s refrain from manipulating the dynticks_nmi counter if the NMI occurred in non-dynticks-idle mode or within an interrupt handler.

The only difference between these two functions is the error checks, as rcu_nmi_enter() must leave the dynticks_nmi counter with an odd value, and rcu_nmi_exit() must leave this counter with an even value.

D.2.7.7 Note That a CPU is in Dynticks Idle Mode

The force_quiescent_state() function implements a three-phase state machine. The first phase (RCU_INITIALIZING) waits for rcu_start_gp() to complete grace-period initialization. This state is not exited by force_quiescent_state(), but rather by rcu_start_gp().

In the second phase (RCU_SAVE_DYNTICK), the dyntick_save_progress_counter() function scans the CPUs that have not yet reported a quiescent state, recording their per-CPU dynticks and dynticks_nmi counters. If these counters both have even-numbered values, then the corresponding CPU is in dynticks-idle state, which is therefore noted as an extended quiescent state (reported via cpu_quiet_msk()).

In the third phase (RCU_FORCE_QS), the rcu_implicit_dynticks_qs() function again scans the CPUs that have not yet reported a quiescent state (either explicitly or implicitly during the RCU_SAVE_DYNTICK phase), again checking the per-CPU dynticks and dynticks_nmi counters. If each of these has either changed in value or is now even, then the corresponding CPU has either passed through or is now in dynticks idle, which as before is noted as an extended quiescent state.

If rcu_implicit_dynticks_qs() finds that a given CPU has neither been in dynticks idle mode nor reported a quiescent state (either explicitly or implicitly during the RCU_SAVE_DYNTICK phase), it invokes rcu_implicit_offline_qs(), which checks to see if that CPU is offline, which is also reported as an extended quiescent state. If the CPU is online, then rcu_implicit_offline_qs() sends it a reschedule IPI in an attempt to remind it of its duty to report a quiescent state to RCU.

Note that force_quiescent_state() does not directly invoke either dyntick_save_progress_counter() or rcu_implicit_dynticks_qs(), instead passing these functions to an intervening rcu_process_dyntick() function that abstracts out the
D.2. HIERARCHICAL RCU OVERVIEW

common code involved in scanning the CPUs and reporting extended quiescent states.

Quick Quiz D.13: And what happens if one CPU comes out of dyntick-idle mode and then passed through a quiescent state just as another CPU notices that the first CPU was in dyntick-idle mode? Couldn’t they both attempt to report a quiescent state at the same time, resulting in confusion?

Quick Quiz D.14: But what if all the CPUs end up in dyntick-idle mode? Wouldn’t that prevent the current RCU grace period from ever ending?

Quick Quiz D.15: Given that force_quiescent_state() is a three-phase state machine, don’t we have triple the scheduling latency due to scanning all the CPUs?

D.2.7.8 Offline a CPU

CPU-offline events cause rcu_cpu_notify() to invoke rcu_offline_cpu(), which in turn invokes rcu_offline_cpu() on both the rcu and the rcu_bh instances of the data structures. This function clears the outgoing CPU's bits so that future grace periods will not expect this CPU to announce quiescent states, and further invokes cpu_quiet() in order to announce the offline-induced extended quiescent state. This work is performed with the global ->onofflock held in order to prevent interference with concurrent grace-period initialization.

Quick Quiz D.16: But the other reason to hold ->onofflock is to prevent multiple concurrent online/offline operations, right?

D.2.7.9 Online a CPU

CPU-online events cause rcu_cpu_notify() to invoke rcu_online_cpu(), which initializes the incoming CPU's dynticks state, and then invokes rcu_init_percpu_data() to initialize the incoming CPU's rcu_data structure, and also to set this CPU's bits (again protected by the global ->onofflock) so that future grace periods will wait for a quiescent state from this CPU. Finally, rcu_online_cpu() sets up the RCU softirq vector for this CPU.

Quick Quiz D.17: Given all these acquisitions of the global ->onofflock, won't there be horrible lock contention when running with thousands of CPUs?

Quick Quiz D.18: Why not simplify the code by merging the detection of dyntick-idle CPUs with that of offline CPUs?

D.2.7.10 Detect a Too-Long Grace Period

When the CONFIG_RCU_CPUSTALL_DETECTOR kernel parameter is specified, the record_gp_stall_check_time() function records the time and also a timestamp set three seconds into the future. If the current grace period still has not ended by that time, the check_cpu_stall() function will check for the culprit, invoking print_cpu_stall() if the current CPU is the holdout, or print_other_cpu_stall() if it is some other CPU. A two-jiffies offset helps ensure that CPUs report on themselves when possible, taking advantage of the fact that a CPU can normally do a better job of tracing its own stack than it can tracing some other CPU's stack.

D.2.8 Testing

RCU is fundamental synchronization code, so any failure of RCU results in random, difficult-to-debug memory corruption. It is therefore extremely important that RCU be highly reliable. Some of this reliability stems from careful design, but at the end of the day we must also rely on heavy stress testing, otherwise known as torture.

Fortunately, although there has been some debate as to exactly what populations are covered by the provisions of the Geneva Convention it is still the case that it does not apply to software. Therefore, it is still legal to torture your software. In fact, it is strongly encouraged, because if you don’t torture your software, it will end up torturing you by crashing at the most inconvenient times imaginable.

Therefore, we torture RCU quite vigorously using the rcutorture module.

However, it is not sufficient to torture the common-case uses of RCU. It is also necessary to torture it in unusual situations, for example, when concurrently onlining and offlining CPUs and when CPUs are concurrently entering and exiting dynticks idle mode. I use a script move to CodeSamples, ref @@@ and use the test_no_idle_hz module parameter to rcutorture to stress-test dynticks idle mode. Just to be fully paranoid, I sometimes run a kernbench workload in parallel as well. Ten hours of this sort of torture on a 128-way machine seems sufficient to shake out most bugs.

Even this is not the complete story. As Alexey Dobriyan and Nick Piggin demonstrated in early 2008, it is also necessary to torture RCU with all relevant combinations of kernel parameters. The relevant kernel parameters may be identified using yet another script move to CodeSamples, ref @@@
1. `CONFIG_CLASSIC_RCU`: Classic RCU.

2. `CONFIG_PREEMPT_RCU`: Preemptible (real-time) RCU.

3. `CONFIG_TREE_RCU`: Classic RCU for huge SMP systems.

4. `CONFIG_RCU_FANOUT`: Number of children for each `rcu_node`.

5. `CONFIG_RCU_FANOUT_EXACT`: Balance the `rcu_node` tree.

6. `CONFIG_HOTPLUG_CPU`: Allow CPUs to be offline and online.

7. `CONFIG_NO_HZ`: Enable dtyick-idle mode.

8. `CONFIG_SMP`: Enable multi-CPU operation.

9. `CONFIG_RCU_CPU_STALL_DETECTOR`: Enable RCU to detect when CPUs go on extended quiescent-state vacations.

10. `CONFIG_RCU_TRACE`: Generate RCU trace files in debugfs.

We ignore the `CONFIG_DEBUG_LOCK_ALLOC` configuration variable under the perhaps-naive assumption that hierarchical RCU could not have broken lockdep. There are still 10 configuration variables, which would result in 1,024 combinations if they were independent boolean variables. Fortunately the first three are mutually exclusive, which reduces the number of combinations down to 384, but `CONFIG_RCU_FANOUT` can take on values from 2 to 64, increasing the number of combinations to 12,096. This is an infeasible number of combinations.

One key observation is that only `CONFIG_NO_HZ` and `CONFIG_PREEMPT` can be expected to have changed behavior if either `CONFIG_CLASSIC_RCU` or `CONFIG_PREEMPT_RCU` are in effect, as only these portions of the two pre-existing RCU implementations were changed during this effort. This cuts out almost two thirds of the possible combinations.

Furthermore, not all of the possible values of `CONFIG_RCU_FANOUT` produce significantly different results, in fact only a few cases really need to be tested separately:

1. Single-node “tree”.
2. Two-level balanced tree.
3. Three-level balanced tree.
4. Autobalanced tree, where `CONFIG_RCU_FANOUT` specifies an unbalanced tree, but such that it is autobalanced in absence of `CONFIG_RCU_FANOUT_EXACT`.
5. Unbalanced tree.

Looking further, `CONFIG_HOTPLUG_CPU` makes sense only given `CONFIG_SMP`, and `CONFIG_RCU_CPU_STALL_DETECTOR` is independent, and really only needs to be tested once (though someone even more paranoid than am I might decide to test it both with and without `CONFIG_SMP`). Similarly, `CONFIG_RCU_TRACE` need only be tested once, but the truly paranoid (such as myself) will choose to run it both with and without `CONFIG_NO_HZ`.

This allows us to obtain excellent coverage of RCU with only 15 test cases. All test cases specify the following configuration parameters in order to run `rcutorture` and so that `CONFIG_HOTPLUG_CPU=n` actually takes effect:

```
CONFIG_RCU_TORTURE_TEST=n
CONFIG_MODULE_UNLOAD=y
CONFIG_SUSPEND=n
CONFIG_HIBERNATION=n
```

The 15 test cases are as follows:

1. Force single-node “tree” for small systems:
   ```
   CONFIG_NR_CPUS=8
   CONFIG_RCU_FANOUT=8
   CONFIG_RCU_FANOUT_EXACT=n
   CONFIG_RCU_TRACE=y
   CONFIG_PREEMPT_RCU=n
   CONFIG_CLASSIC_RCU=n
   CONFIG_TREE_RCU=y
   ```

2. Force two-level tree for large systems:
   ```
   CONFIG_NR_CPUS=8
   CONFIG_RCU_FANOUT=4
   CONFIG_RCU_FANOUT_EXACT=n
   CONFIG_RCU_TRACE=y
   CONFIG_PREEMPT_RCU=n
   CONFIG_CLASSIC_RCU=n
   CONFIG_TREE_RCU=y
   ```

3. Force three-level tree for huge systems:
   ```
   CONFIG_NR_CPUS=8
   CONFIG_RCU_FANOUT=2
   CONFIG_RCU_FANOUT_EXACT=n
   CONFIG_RCU_TRACE=y
   CONFIG_PREEMPT_RCU=n
   CONFIG_CLASSIC_RCU=n
   CONFIG_TREE_RCU=y
   ```
4. Test autobalancing to a balanced tree:

```c
CONFIG_NR_CPUS=8
CONFIG_RCU_FANOUT=6
CONFIG_RCU_FANOUT_EXACT=n
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=n
CONFIG_CLASSIC_RCU=n
CONFIG_TREE_RCU=y
```

5. Test unbalanced tree:

```c
CONFIG_NR_CPUS=8
CONFIG_RCU_FANOUT=6
CONFIG_RCU_FANOUT_EXACT=y
CONFIG_RCU_CPUSTALL_DETECTOR=y
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=n
CONFIG_CLASSIC_RCU=n
CONFIG_TREE_RCU=y
```

6. Disable CPU-stall detection:

```c
CONFIG_SMP=y
CONFIG_NO_HZ=y
CONFIG_RCU_CPUSTALL_DETECTOR=n
CONFIG_HOTPLUG_CPU=y
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=n
CONFIG_CLASSIC_RCU=y
CONFIG_TREE_RCU=y
```

7. Disable CPU-stall detection and dyntick idle mode:

```c
CONFIG_SMP=y
CONFIG_NO_HZ=n
CONFIG_RCU_CPUSTALL_DETECTOR=n
CONFIG_HOTPLUG_CPU=y
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=n
CONFIG_CLASSIC_RCU=n
CONFIG_TREE_RCU=y
```

8. Disable CPU-stall detection and CPU hotplug:

```c
CONFIG_SMP=y
CONFIG_NO_HZ=y
CONFIG_RCU_CPUSTALL_DETECTOR=n
CONFIG_HOTPLUG_CPU=n
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=n
CONFIG_CLASSIC_RCU=n
CONFIG_TREE_RCU=y
```

9. Disable CPU-stall detection, dyntick idle mode, and CPU hotplug:

```c
CONFIG_SMP=y
CONFIG_NO_HZ=n
CONFIG_RCU_CPUSTALL_DETECTOR=n
CONFIG_HOTPLUG_CPU=n
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=n
CONFIG_CLASSIC_RCU=n
CONFIG_TREE_RCU=y
```

10. Disable SMP, CPU-stall detection, dyntick idle mode, and CPU hotplug:

```c
CONFIG_SMP=n
CONFIG_NO_HZ=n
CONFIG_RCU_CPUSTALL_DETECTOR=n
CONFIG_HOTPLUG_CPU=n
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=n
CONFIG_CLASSIC_RCU=n
CONFIG_TREE_RCU=y
```

This combination located a number of compiler warnings.

11. Disable SMP and CPU hotplug:

```c
CONFIG_SMP=n
CONFIG_NO_HZ=n
CONFIG_RCU_CPUSTALL_DETECTOR=n
CONFIG_HOTPLUG_CPU=n
CONFIG_RCU_TRACE=n
CONFIG_PREEMPT_RCU=n
CONFIG_CLASSIC_RCU=n
CONFIG_TREE_RCU=y
```

12. Test Classic RCU with dynticks idle but without preemption:

```c
CONFIG_NO_HZ=y
CONFIG_PREEMPT=n
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=n
CONFIG_CLASSIC_RCU=y
CONFIG_TREE_RCU=n
```

13. Test Classic RCU with preemption but without dynticks idle:

```c
CONFIG_NO_HZ=n
CONFIG_PREEMPT=y
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=n
CONFIG_CLASSIC_RCU=y
CONFIG_TREE_RCU=n
```

14. Test Preemptible RCU with dynticks idle:

```c
CONFIG_NO_HZ=y
CONFIG_PREEMPT=y
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=y
CONFIG_CLASSIC_RCU=n
CONFIG_TREE_RCU=n
```

15. Test Preemptible RCU without dynticks idle:
For a large change that affects RCU core code, one should run rcutorture for each of the above combinations, and concurrently with CPU offlining and onlining for cases with CONFIG_HOTPLUG_CPU. For small changes, it may suffice to run kernbench in each case. Of course, if the change is confined to a particular subset of the configuration parameters, it may be possible to reduce the number of test cases.

Torturing software: the Geneva Convention does not (yet) prohibit it, and I strongly recommend it!

D.2.9 Conclusion

This hierarchical implementation of RCU reduces lock contention, avoids unnecessarily awakening dyntick-idle sleeping CPUs, while helping to debug Linux’s hotplug-CPU code paths. This implementation is designed to handle single systems with thousands of CPUs, and on 64-bit systems has an architectural limitation of a quarter million CPUs, a limit I expect to be sufficient for at least the next few years.

This RCU implementation of course has some limitations:

1. The force_quiescent_state() can scan the full set of CPUs with irqs disabled. This would be fatal in a real-time implementation of RCU, so if hierarchy ever needs to be introduced to preemptible RCU, some other approach will be required. It is possible that it will be problematic on 4,096-CPU systems, but actual testing on such systems is required to prove this one way or the other.

On busy systems, the force_quiescent_state() scan would not be expected to happen, as CPUs should pass through quiescent states within three jiffies of the start of a quiescent state. On semi-busy systems, only the CPUs in dynticks-idle mode throughout would need to be scanned. In some cases, for example when a dynticks-idle CPU is handling an interrupt during a scan, subsequent scans are required. However, each such scan is performed separately, so scheduling latency is degraded by the overhead of only one such scan.

If this scan proves problematic, one straightforward solution would be to do the scan incrementally. This would increase code complexity slightly and would also increase the time required to end a grace period, but would nonetheless be a likely solution.

2. The rcu_node hierarchy is created at compile time, and is therefore sized for the worst-case NR_CPUS number of CPUs. However, even for 4,096 CPUs, the rcu_node hierarchy consumes only 65 cache lines on a 64-bit machine (and just you try accommodating 4,096 CPUs on a 32-bit machine!). Of course, a kernel built with NR_CPUS=4096 running on a 16-CPU machine would use a two-level tree when a single-node tree would work just fine. Although this configuration would incur added locking overhead, this does not affect hot-path read-side code, so should not be a problem in practice.

3. This patch does increase kernel text and data somewhat: the old Classic RCU implementation consumes 1,757 bytes of kernel text and 456 bytes of kernel data for a total of 2,213 bytes, while the new hierarchical RCU implementation consumes 4,006 bytes of kernel text and 624 bytes of kernel data for a total of 4,630 bytes on a NR_CPUS=4 system. This is a non-problem even for most embedded systems, which often come with hundreds of megabytes of main memory. However, if this is a problem for tiny embedded systems, it may be necessary to provide both “scale up” and “scale down” implementations of RCU.

This hierarchical RCU implementation should nevertheless be a vast improvement over Classic RCU for machines with hundreds of CPUs. After all, Classic RCU was designed for systems with only 16-32 CPUs.

At some point, it may be necessary to also apply hierarchy to the preemptible RCU implementation. This will be challenging due to the modular arithmetic used on the per-CPU counter pairs, but should be doable.

D.3 Hierarchical RCU Code Walk-through

This section walks through selected sections of the Linux-kernel hierarchical RCU code. As such, this section is intended for hard-core hackers who wish to understand hierarchical RCU at a very low level, and such hackers
should first read Section D.2. Hard-core masochists might also be interested in reading this section. Of course really hard-core masochists will read this section before reading Section D.2.

Section D.3.1 describes data structures and kernel parameters, Section D.3.2 covers external function interfaces, Section D.3.3 presents the initialization process, Section D.3.4 explains the CPU-hotplug interface, Section D.3.5 covers miscellaneous utility functions, Section D.3.6 describes the mechanics of grace-period detection, Section D.3.7 presents the dynticks-idle interface, Section D.3.8 covers the functions that handle holdout CPUs (including offline and dynticks-idle CPUs), and Section D.3.9 presents functions that report on stalled CPUs, namely those spinning in kernel mode for many seconds. Finally, Section D.3.10 reports on possible design flaws and fixes.

D.3.1 Data Structures and Kernel Parameters

A full understanding of the Hierarchical RCU data structures is critically important to understanding the algorithms. To this end, Section D.3.1.1 describes the data structures used to track each CPU’s dyntick-idle state, Section D.3.1.2 describes the fields in the per-node data structure making up the rcu_node hierarchy, Section D.3.1.3 describes per-CPU rcu_data structure, Section D.3.1.4 describes the field in the global rcu_state structure, and Section D.3.1.5 describes the kernel parameters that control Hierarchical RCU’s operation.

Figure D.17 on Page 221 and Figure D.26 on Page 239 can be very helpful in keeping one’s place through the following detailed data-structure descriptions.

D.3.1.1 Tracking Dyntick State

The per-CPU rcu_dynticks structure tracks dynticks state using the following fields:

- **dynticks_nesting**: This int counts the number of reasons that the corresponding CPU should be monitored for RCU read-side critical sections. If the CPU is in dynticks-idle mode, then this counts the irq nesting level, otherwise it is one greater than the irq nesting level.

- **dynticks**: This int counter’s value is even if the corresponding CPU is in dynticks-idle mode and there are no irq handlers currently running on that CPU, otherwise the counter’s value is odd. In other words, if this counter’s value is odd, then the corresponding CPU might be in an RCU read-side critical section.

- **dynticks_nmi**: This int counter’s value is odd if the corresponding CPU is in an NMI handler, but only if the NMI arrived while this CPU was in dyntick-idle mode with no irq handlers running. Otherwise, the counter’s value will be even.

This state is shared between the rcu and rcu_bh implementations.

D.3.1.2 Nodes in the Hierarchy

As noted earlier, the rcu_node hierarchy is flattened into the rcu_state structure as shown in Figure D.13 on page 219. Each rcu_node in this hierarchy has fields as follows:

- **lock**: This spinlock guards the non-constant fields in this structure. This lock is acquired from softirq context, so must disable irqs.

Quick Quiz D.19: Why not simply disable bottom halves (softirq) when acquiring the rcu_data structure’s lock? Wouldn’t this be faster?

The lock field of the root rcu_node has additional responsibilities:

1. Serializes CPU-stall checking, so that a given stall is reported by only one CPU. This can be important on systems with thousands of CPUs!
2. Serializes starting a new grace period, so that multiple CPUs don’t start conflicting grace periods concurrently.
3. Prevents new grace periods from starting in code that needs to run within the confines of a single grace period.
4. Serializes the state machine forcing quiescent states (in force_quiescent_state()) in order to keep the number of reschedule IPIs down to a dull roar.

- **qsmask**: This bitmask tracks which CPUs (for leaf rcu_node structures) or groups of CPUs (for non-leaf rcu_node structures) still need to pass through a quiescent state in order for the current grace period to end.
• qsmaskinit: This bitmask tracks which CPUs or groups of CPUs will need to pass through a quiescent state for subsequent grace periods to end. The online/offline code manipulates the qsmaskinit fields, which are copied to the corresponding qsmask fields at the beginning of each grace period. This copy operation is one reason why grace period initialization must exclude online/offline operations.

• grpmask: This bitmask has a single bit set, and that is the bit corresponding to the this rcu_node structure's position in the parent rcu_node structure's qsmask and qsmaskinit fields. Use of this field simplifies quiescent-state processing, as suggested by Manfred Spraul.

Quick Quiz D.20: How about the qsmask and qsmaskinit fields for the leaf rcu_node structures? Doesn’t there have to be some way to work out which of the bits in these fields corresponds to each CPU covered by the rcu_node structure in question?

• grplo: This field contains the number of the lowest-numbered CPU covered by this rcu_node structure.

• grphi: This field contains the number of the highest-numbered CPU covered by this rcu_node structure.

• grpnum: This field contains the bit number in the parent rcu_node structure’s qsmask and qsmaskinit fields that this rcu_node structure corresponds to. In other words, given a pointer rnp to a given rcu_node structure, it will always be the case that 1UL << rnp->grpnum == rnp->grpmask. The grpnum field is used only for tracing output.

• level: This field contains zero for the root rcu_node structure, one for the rcu_node structures that are children of the root, and so on down the hierarchy.

• parent: This field is a pointer to the parent rcu_node structure, or NULL for the root rcu_node structure.

D.3.1.3 Per-CPU Data

The rcu_data structure contains RCU’s per-CPU state. It contains control variables governing grace periods and quiescent states (completed, gpnum, passed_quiesc_completed, passed_quiesc, qs_pending, beenonline, mynode, and grpmask). The rcu_data structure also contains control variables pertaining to RCU callbacks (nxtlist, nxttail, qlen, and blimit). Kernels with dynticks enabled will have relevant control variables in the rcu_data structure (dynticks, dynticks_snap, and dynticks_nmi_snap). The rcu_data structure contains event counters used by tracing (dynticks_fqs given dynticks, offline_fqs, and resched_ipi). Finally, a pair of fields count calls to rcu_pending() in order to determine when to force quiescent states (n_rcu_pending and n_rcu_pendingsforce_qs), and a cpu field indicates which CPU to which a given rcu_data structure corresponds.

Each of these fields is described below.

• completed: This field contains the number of the most recent grace period that this CPU is aware of having completed.

• gpnum: This field contains the number of the most recent grace period that this CPU is aware of having started.

• passed_quiesc_completed: This field contains the number of the grace period that had most recently completed when this CPU last passed through a quiescent state. The "most recently completed" will be from the viewpoint of the CPU passing through the quiescent state: if the CPU is not yet aware that grace period (say) 42 has completed, it will still record the old value of 41. This is OK, because the only way that the grace period can complete is if this CPU has already passed through a quiescent state. This field is initialized to a (possibly mythical) past grace period number to avoid race conditions when booting and when onlining a CPU.

• passed_quiesc: This field indicates whether this CPU has passed through a quiescent state since the grace period number stored in passed_quiesc_completed completed. This field is cleared each time the corresponding CPU becomes aware of the start of a new grace period.
• qs_pending: This field indicates that this CPU is aware that the core RCU mechanism is waiting for it to pass through a quiescent state. This field is set to one when the CPU detects a new grace period or when a CPU is coming online.

Quick Quiz D.21: But why bother setting qs_pending to one when a CPU is coming online, given that being offline is an extended quiescent state that should cover any ongoing grace period?

Quick Quiz D.22: Why record the last completed grace period number in passed_quiesc_completed? Doesn’t that cause this RCU implementation to be vulnerable to quiescent states seen while no grace period was in progress being incorrectly applied to the next grace period that starts?

• beenonline: This field, initially zero, is set to one whenever the corresponding CPU comes online. This is used to avoid producing useless tracing output for CPUs that never have been online, which is useful in kernels where NR_CPUS greatly exceeds the actual number of CPUs.

Quick Quiz D.23: What is the point of running a system with NR_CPUS way bigger than the actual number of CPUs?

• mynode: This field is a pointer to the leaf rcu_node structure that handles the corresponding CPU.

• grpmask: This field is a bitmask that has the single bit set that indicates which bit in mynode->qsmask signifies the corresponding CPU.

• nxtlist: This field is a pointer to the oldest RCU callback (rcu_head structure) residing on this CPU, or NULL if this CPU currently has no such callbacks. Additional callbacks may be chained via their next pointers.

• nxttail: This field is an array of double-indirect tail pointers into the nxtlist callback list. If nxtlist is empty, then all of the nxttail pointers directly reference the nxtlist field. Each element of the nxttail array has meaning as follows:
  - RCU_DONE_TAIL=0: This element references the ->next field of the last callback that has passed through its grace period and is ready to invoke, or references the nxtlist field if there is no such callback.
  - RCU_WAIT_TAIL=1: This element references the next field of the last callback that is waiting for the current grace period to end, or is equal to the RCU_DONE_TAIL element if there is no such callback.
  - RCU_NEXT_READY_TAIL=2: This element references the next field of the last callback that is ready to wait for the next grace period, or is equal to the RCU_WAIT_TAIL element if there is no such callback.
  - RCU_NEXT_TAIL=3: This element references the next field of the last callback in the list, or references the nxtlist field if the list is empty.

Quick Quiz D.24: Why not simply have multiple lists rather than this funny multi-tailed list?

• qlen: This field contains the number of callbacks queued on nxtlist.

• blimit: This field contains the maximum number of callbacks that may be invoked at a time. This limitation improves system responsiveness under heavy load.

• dynticks: This field references the rcu_dynticks structure for the corresponding CPU, which is described in Section D.3.1.1.

• dynticks_snap: This field contains a past value of dynticks->dynticks, which is used to detect when a CPU passes through a dynticks idle state when this CPU happens to be in an irq handler each time that force_quiescent_state() checks it.

• dynticks_nmi_snap: This field contains a past value of dynticks->dynticks_nmi, which is used to detect when a CPU passes through a dynticks idle state when this CPU happens to be in an NMI handler each time that force_quiescent_state() checks it.

• dynticks_fqs: This field counts the number of times that some other CPU noted a quiescent state on behalf of the CPU corresponding to this rcu_data structure due to its being in dynticks-idle mode.
The \texttt{rcu_state} structure also contains a number of fields coordinating various details of the current grace period and its interaction with other mechanisms (\texttt{signaled}, \texttt{gnum}, \texttt{completed}, \texttt{onofflock}, \texttt{fqslock}, \texttt{jiffies\_force\_qs}, \texttt{n\_force\_qs}, \texttt{n\_force\_qs\_lh}, \texttt{n\_force\_qs\_ngp}, \texttt{gp\_start}, \texttt{jiffies\_stall}, and \texttt{dynticks\_completed}). Each of these fields are described below.

- \texttt{offline\_fqs}: This field counts the number of times that some other CPU noted a quiescent state on behalf of the CPU corresponding to this \texttt{rcu\_data} structure due to its being offline.

- \texttt{resched\_ipi}: This field counts the number of times that a reschedule IPI is sent to the corresponding CPU. Such IPIs are sent to CPUs that fail to report passing through a quiescent states in a timely manner, but are neither offline nor in dynticks idle state.

- \texttt{n\_rcu\_pending}: This field counts the number of calls to \texttt{rcu\_pending()}, which is called once per jiffy on non-dynticks-idle CPUs.

- \texttt{n\_rcu\_pending\_force\_qs}: This field holds a threshold value for \texttt{n\_rcu\_pending}. If \texttt{n\_rcu\_pending} reaches this threshold, that indicates that the current grace period has extended too long, so \texttt{force\_quiescent\_state()} is invoked to expedite it.

- \texttt{level\_timeout}: This field is an array of pointers into the \texttt{node} array. The root node of the hierarchy is referenced by \texttt{->level[0]}, the first node of the second level of the hierarchy (if there is one) by \texttt{->level[1]}, and so on. The first leaf node is referenced by \texttt{->level[NUM\_RCU\_LVLS\_1]}, and the size of the level array is thus specified by \texttt{NUM\_RCU\_LVLS}, which is computed as described in Section D.3.1.5. The \texttt{->level} field is often used in combination with \texttt{->node} to scan a level of the \texttt{rcu\_node} hierarchy, for example, all of the leaf nodes. The elements of \texttt{->level} are filled in by the boot-time \texttt{rcu\_init\_one()} function.

- \texttt{levelcnt}: This field is an array containing the number of \texttt{rcu\_node} structures in each level of the hierarchy, including the number of \texttt{rcu\_data} structures referencing the leaf \texttt{rcu\_node} structures, so that this array has one more element than does the \texttt{->level} array. Note that \texttt{->levelcnt[0]} will always contain a value of one, corresponding to the single root \texttt{rcu\_node} at the top of the hierarchy. This array is initialized with the values \texttt{NUM\_RCU\_LVLS\_0}, \texttt{NUM\_RCU\_LVLS\_1}, \texttt{NUM\_RCU\_LVLS\_2}, and \texttt{NUM\_RCU\_LVLS\_3}, which are C-preprocessor macros computed as described in Section D.3.1.5. The \texttt{->levelcnt} field is used to initialize other parts of the hierarchy and for debugging purposes.

- \texttt{levelspread}: Each element of this field contains the desired number of children for the corresponding level of the \texttt{rcu\_node} hierarchy. This array’s element’s values are computed at runtime by one of the two \texttt{rcu\_init\_levelspread()} functions, selected by the \texttt{CONFIG\_RCU\_FANOUT\_EXACT} kernel parameter.

- \texttt{rda}: Each element of this field contains a pointer to the corresponding CPU’s \texttt{rcu\_data} structure.
This array is initialized at boot time by the \texttt{RCU\_DATA\_PTR\_INIT()} macro.

- **signaled**: This field is used to maintain state used by the \texttt{force\_quiescent\_state()} function, as described in Section D.3.8. This field takes on values as follows:
  - \texttt{RCU\_GP\_INIT}: This value indicates that the current grace period is still in the process of being initialized, so that \texttt{force\_quiescent\_state()} should take no action. Of course, grace-period initialization would need to stretch out for three jiffies before this race could arise, but if you have a very large number of CPUs, this race could in fact occur. Once grace-period initialization is complete, this value is set to either \texttt{RCU\_SAVE\_DYNTICK} (if \texttt{CONFIG\_NO\_HZ}) or \texttt{RCU\_FORCE\_QS} otherwise.
  - \texttt{RCU\_SAVE\_DYNTICK}: This value indicates that \texttt{force\_quiescent\_state()} should check the dynticks state of any CPUs that have not yet reported quiescent states for the current grace period. Quiescent states will be reported on behalf of any CPUs that are in dynticks-idle mode.
  - \texttt{RCU\_FORCE\_QS}: This value indicates that \texttt{force\_quiescent\_state()} should recheck dynticks state along with the online/offline state of any CPUs that have not yet reported quiescent states for the current grace period. The rechecking of dynticks states allows the implementation to handle cases where a given CPU might be in dynticks-idle state, but have been in an irq or NMI handler both times it was checked. If all else fails, a reschedule IPI will be sent to the laggard CPU.

This field is guarded by the root \texttt{rcu\_node} structure’s lock.

**Quick Quiz D.26**: So what guards the earlier fields in this structure? ■

- **gpnum**: This field contains the number of the current grace period, or that of the last grace period if no grace period is currently in effect. This field is guarded by the root \texttt{rcu\_node} structure’s lock, but is frequently accessed (but never modified) without holding this lock.

- **completed**: This field contains the number of the last completed grace period. As such, it is equal to \texttt{->gpnum} when there is no grace period in progress, or one less than \texttt{->gpnum} when there is a grace period in progress. In principle, one could replace this pair of fields with a single boolean, as is done in Classic RCU in some versions of Linux, but in practice race resolution is much simpler given the pair of numbers. This field is guarded by the root \texttt{rcu\_node} structure’s lock, but is frequently accessed (but never modified) without holding this lock.

- **onofflock**: This field prevents online/offline processing from running concurrently with grace-period initialization. There is one exception to this: if the \texttt{rcu\_node} hierarchy consists of but a single structure, then that single structure’s \texttt{->lock} field will instead take on this job.

- **fqslock**: This field prevents more than one task from forcing quiescent states with \texttt{force\_quiescent\_state()}.

- **jiffies\_force\_qs**: This field contains the time, in jiffies, when \texttt{force\_quiescent\_state()} should be invoked in order to force CPUs into quiescent states and/or report extended quiescent states. This field is guarded by the root \texttt{rcu\_node} structure’s lock, but is frequently accessed (but never modified) without holding this lock.

- **n\_force\_qs**: This field counts the number of calls to \texttt{force\_quiescent\_state()} that actually do work, as opposed to leaving early due to the grace period having already completed, some other CPU currently running \texttt{force\_quiescent\_state()}, or \texttt{force\_quiescent\_state()} having run too recently. This field is used for tracing and debugging, and is guarded by \texttt{->fqslock}.

- **n\_force\_qs\_lh**: This field holds an approximate count of the number of times that \texttt{force\_quiescent\_state()} returned early due to the \texttt{->fqslock} being held by some other CPU. This field is used for tracing and debugging, and is not guarded by any lock, hence its approximate nature.

- **n\_force\_qs\_ngp**: This field counts the number of times that \texttt{force\_quiescent\_state()} that successfully acquire \texttt{->fqslock}, but then find that there is no grace period in progress. This field is
used for tracing and debugging, and is guarded by 
\(-fqslock\).

- **gp_start**: This field records the time at which the most recent grace period began, in jiffies. This is used to detect stalled CPUs, but only when the CONFIG_RCU_CPU_STALL_DETECTOR kernel parameter is selected. This field is guarded by the root rcu_node's \(->lock\), but is sometimes accessed (but not modified) outside of this lock.

- **jiffies_stall**: This field holds the time, in jiffies, at which the current grace period will have extended for so long that it will be appropriate to check for CPU stalls. As with \(->gp_start\), this field exists only when the CONFIG_RCU_CPU_STALL_DETECTOR kernel parameter is selected. This field is guarded by the root rcu_node's \(->lock\), but is sometimes accessed (but not modified) outside of this lock.

- **dynticks_completed**: This field records the value of \(->completed\) at the time when force_quiescent_state() snapshots dyntick state, but is also initialized to an earlier grace period at the beginning of each grace period. This field is used to prevent dyntick-idle quiescent states from a prior grace period from being applied to the current grace period. As such, this field exists only when the CONFIG_NO_HZ kernel parameter is selected. This field is guarded by the root rcu_node's \(->lock\), but is sometimes accessed (but not modified) outside of this lock.

D.3.1.5 Kernel Parameters

The following kernel parameters affect this variant of RCU:

- **NR_CPUS**, the maximum number of CPUs in the system.
- **CONFIG_RCU_FANOUT**, the desired number of children for each node in the rcu_node hierarchy.
- **CONFIG_RCU_FANOUT_EXACT**, a boolean preventing rebalancing of the rcu_node hierarchy.
- **CONFIG_HOTPLUG_CPU**, permitting CPUs to come online and go offline.
- **CONFIG_NO_HZ**, indicating that dynticks-idle mode is supported.
- **CONFIG_SMP**, indicating that multiple CPUs may be present.
- **CONFIG_RCU_CPU_STALL_DETECTOR**, indicating that RCU should check for stalled CPUs when RCU grace periods extend too long.
- **CONFIG_RCU_TRACE**, indicating that RCU should provide tracing information in debugfs.

The CONFIG_RCU_FANOUT and NR_CPUS parameters are used to determine the shape of the rcu_node hierarchy at compile time, as shown in Figure D.20. Line 1 defines the maximum depth of the rcu_node hierarchy, currently three. Note that increasing the maximum permitted depth requires changes elsewhere, for example, adding another leg to the #if statement running from lines 6-26. Lines 2-4 compute the fanout, the square of the fanout, and the cube of the fanout, respectively. Then these values are compared to NR_CPUS to determine the required depth of the rcu_node hierarchy, which is placed into NUM_RCU_LVLS, which is used to size a number of arrays in the rcu_state structure. There is always one node at the root level, and there are always NUM_CPUS number of rcu_data structures below the leaf level. If there is more than just the root level, the number of nodes at the leaf level is computed by dividing NR_CPUS by RCU_FANOUT, rounding up. The number of nodes at other levels is computed in a similar manner, but using (for example) RCU_FANOUT_SQ instead of RCU_FANOUT.

Line 28 then sums up all of the levels, resulting in the number of rcu_node structures plus the number of rcu_data structures. Finally, line 29 subtracts NR_CPUS (which is the number of rcu_data structures) from the sum, resulting in the number of rcu_node structures, which is retained in NUM_RCU_NODES. This value is then used to size the \(->nodes\) array in the rcu_state structure.

D.3.2 External Interfaces

RCU's external interfaces include not just the standard RCU API, but also the internal interfaces to the rest of the kernel that are required for the RCU implementation itself. The interfaces are

```
rcu_read_lock()
rcu_read_unlock()
rcu_read_lock_bh()
rcu_read_unlock_bh()
call_rcu()
call_rcu_bh()
rcu_check_callbacks()
```


D.3. HIERARCHICAL RCU CODE WALKTHROUGH

```c
1 #define MAX_RCU_LVLS 3
2 #define RCU_FANOUT (CONFIG_RCU_FANOUT)
3 #define RCU_FANOUT_SQ (RCU_FANOUT * RCU_FANOUT)
4 #define RCU_FANOUT_CUBE (RCU_FANOUT_SQ * RCU_FANOUT)
5
6 #if NR_CPUS <= RCU_FANOUT
7 # define NUM_RCU_LVLS 1
8 # define NUM_RCU_LVL_0 1
9 # define NUM_RCU_LVL_1 (NR_CPUS)
10 # define NUM_RCU_LVL_2 0
11 # define NUM_RCU_LVL_3 0
12 #elif NR_CPUS <= RCU_FANOUT_SQ
13 # define NUM_RCU_LVLS 2
14 # define NUM_RCU_LVL_0 1
15 # define NUM_RCU_LVL_1 (((NR_CPUS) + RCU_FANOUT - 1) / RCU_FANOUT)
16 # define NUM_RCU_LVL_2 (NR_CPUS)
17 # define NUM_RCU_LVL_3 0
18 #elif NR_CPUS <= RCU_FANOUT_CUBE
19 # define NUM_RCU_LVLS 3
20 # define NUM_RCU_LVL_0 1
21 # define NUM_RCU_LVL_1 (((NR_CPUS) + RCU_FANOUT_SQ - 1) / RCU_FANOUT_SQ)
22 # define NUM_RCU_LVL_2 (((NR_CPUS) + (RCU_FANOUT) - 1) / (RCU_FANOUT))
23 # define NUM_RCU_LVL_3 (RCU_FANOUT)
24 #else
25 # define RCU_FANOUT_CUBE (NR_CPUS)
26 #endif /* #if (NR_CPUS) <= RCU_FANOUT */
27
28 #define RCU_SUM (NUM_RCU_LVL_0 + NUM_RCU_LVL_1 + NUM_RCU_LVL_2 + NUM_RCU_LVL_3)
29 #define NUM_RCU_NODES (RCU_SUM - NR_CPUS)
```

Figure D.20: Determining Shape of RCU Hierarchy

rcu_process_callbacks() (which is a wrapper around __rcu_process_callbacks(), rcu_pending() (which is a wrapper around __rcu_pending()), rcu_needs_cpu(), rcu_cpu_notify(), and __rcu_init()). Note that synchronize_rcu() and rcu_barrier() are common to all RCU implementations, and are defined in terms of call_rcu(). Similarly, rcu_barrier_bh() is common to all RCU implementations and is defined in terms of call_rcu_bh().

These external APIs are each described in the following sections.

D.3.2.1 Read-Side Critical Sections

Figure D.21 shows the functions that demark RCU read-side critical sections. Lines 1-6 show __rcu_read_lock(), which begins an “rcu” read-side critical section. line 3 disables preemption, line 4 is a sparse marker noting the beginning of an RCU read-side critical section, and line 5 updates lockdep state. Lines 8-13 show __rcu_read_unlock(), which is the inverse of __rcu_read_lock(). Lines 15-20 show __rcu_read_lock_bh() and lines 22-27 show __rcu_read_unlock_bh(), which are analogous to the previous two functions, but disable and enable bottom-

```
1 void __rcu_read_lock(void)
2 {
3 preempt_disable();
4 __acquire(RCU);
5 rcu_read_acquire();
6 }
7
8 void __rcu_read_unlock(void)
9 {
10 rcu_read_release();
11 __release(RCU);
12 preempt_enable();
13 }
14
15 void __rcu_read_lock_bh(void)
16 {
17 local_bh_disable();
18 __acquire(RCU_BH);
19 rcu_read_acquire();
20 }
21
22 void __rcu_read_unlock_bh(void)
23 {
24 rcu_read_release();
25 __release(RCU_BH);
26 local_bh_enable();
27 }
```
APPENDIX D. READ-COPY UPDATE IMPLEMENTATIONS

```c
1 static void __call_rcu(struct rcu_head *head,
2     void (*func)(struct rcu_head *rcu),
3     struct rcu_state *rsp)
4 {
5     unsigned long flags;
6     struct rcu_data *rdp;
7
8     head->func = func;
9     head->next = NULL;
10     smp_mb();
11     local_irq_save(flags);
12     rdp = rsp->rda[smp_processor_id()];
13     rcu_process_gp_end(rsp, rdp);
14     check_for_new_grace_period(rsp, rdp);
15     *rdp->nxttail[RCU_NEXT_TAIL] = head;
16     rdp->nxttail[RCU_NEXT_TAIL] = &head->next;
17     if (ACCESS_ONCE(rsp->completed) ==
18         ACCESS_ONCE(rsp->gpnum)) {
19         ACCESS_ONCE(rsp->completed) =
20         ACCESS_ONCE(rsp->gpnum);}
21     unsigned long nestflag;
22     struct rcu_node *rnp_root = rcu_get_root(rsp);
23     spin_lock_irqsave(&rnp_root->lock, nestflag);
24     rcu_start_gp(rsp, nestflag);
25 }
26     if (unlikely(++rdp->qlen > qhimark)) {
27         rdp->blimit = LONG_MAX;
28         force_quiescent_state(rsp, 0);
29     } else if ((long)(ACCESS_ONCE(rsp->jiffies_force_qs) -
30         jiffies) < 0 ||
31         (rdp->n_rcu_pending_force_qs -
32         rdp->n_rcu_pending) < 0)
33         force_quiescent_state(rsp, 1);
34     local_irq_restore(flags);
35 }
36
37 void call_rcu(struct rcu_head *head,
38     void (*func)(struct rcu_head *rcu))
39 {
40     __call_rcu(head, func, &rcu_state);
41 }
42
43 void call_rcu_bh(struct rcu_head *head,
44     void (*func)(struct rcu_head *rcu))
45 {
46     __call_rcu(head, func, &rcu_bh_state);
47 }
```

Figure D.22: `call_rcu()` Code

half processing rather than preemption.

Quick Quiz D.27: I thought that RCU read-side processing was supposed to be fast! The functions shown in Figure D.21 have so much junk in them that they just have to be slow! What gives here?

D.3.2.2 `call_rcu()`

Figure D.22 shows the code for `__call_rcu()`, `call_rcu()`, and `call_rcu_bh()`. Note that `call_rcu()` and `call_rcu_bh()` are simple wrappers for `call_rcu()`, and thus will not be considered further here.

Turning attention to `__call_rcu()`, lines 9-10 initialize the specified `rcu_head`, and line 11 ensures that updates to RCU-protected data structures carried out prior to invoking `__call_rcu()` are seen prior to callback registry. Lines 12 and 34 disable and re-enable interrupts to prevent destructive interference by any calls to `__call_rcu()` from an interrupt handler. Line 13 obtains a reference to the current CPU’s `rcu_data` structure, line 14 invokes `rcu_process_gp_end()` in order to advance callbacks if the current grace period has now ended, while line 15 invokes `check_for_new_grace_period()` to record state if a new grace period has started.

Quick Quiz D.28: Why not simply use `__get_cpu_var()` to pick up a reference to the current CPU’s `rcu_data` structure on line 13 in Figure D.22?

Lines 16 and 17 enqueue the new callback. Lines 18 and 19 check to see there is a grace period in progress, and, if not, line 23 acquires the root `rcu_node` structure’s lock and line 24 invokes `rcu_start_gp()` to start a new grace period (and also to release the lock).

Line 26 checks to see if too many RCU callbacks are waiting on this CPU, and, if so, line 27 increases `->blimit` in order to increase the rate at which callbacks are processed, while line 28 invokes `force_quiescent_state()` urgently in order to try to convince holdout CPUs to pass through quiescent states. Otherwise, lines 29-32 check to see if it has been too long since the grace period started (or since the last call to `force_quiescent_state()`, as the case may be), and, if so, line 33 invokes `force_quiescent_state()` non-urgently, again to convince holdout CPUs to pass through quiescent states.

D.3.2.3 `rcu_check_callbacks()`

Figure D.23 shows the code that is called from the scheduling-clock interrupt handler once per jiffy from each CPU. The `rcu_pending()` function (which is a wrapper for `__rcu_pending()`) is invoked, and if it returns non-zero, then `rcu_check_callbacks()` is invoked. (Note that there is some thought being given to merging `rcu_pending()` into `rcu_check_callbacks()`.)

Starting with `__rcu_pending()`, line 4 counts this call to `rcu_pending()` for use in deciding when to force quiescent states. Line 6 invokes `check_cpu_stall()` in order to report on CPUs that are spinning in the kernel, or perhaps that have hardware problems, if `CONFIG_RCU_CPU_STALL_DETECTOR` is selected. Lines 7-23 perform a series of checks, returning non-zero
D.3. HIERARCHICAL RCU CODE WALKTHROUGH

```c
static int __rcu_pending(struct rcu_state *rsp,
struct rcu_data *rdp)
{
  rdp->n_rcu_pending++;
  if (rdp->qs_pending)
    return 1;
  if (cpu_has_callbacks_ready_to_invoke(rdp))
    return 1;
  if (cpu_needs_another_gp(rsp, rdp))
    return 1;
  if (ACCESS_ONCE(rsp->completed) != rdp->completed)
    return 1;
  if (ACCESS_ONCE(rsp->gpnum) != rdp->gpnum)
    return 1;
  if (ACCESS_ONCE(rsp->completed) !=
      ACCESS_ONCE(rsp->gpnum) &&
      ((long)(ACCESS_ONCE(rsp->jiffies_force_qs) -
        jiffies) < 0 ||
       (rdp->n_rcu_pending_force_qs -
        rdp->n_rcu_pending) < 0))
    return 1;
  return 0;
}

int rcu_pending(int cpu)
{
  return __rcu_pending(&rcu_state,
                       &per_cpu(rcu_data, cpu)) ||
        __rcu_pending(&rcu_bh_state,
                      &per_cpu(rcu_bh_data, cpu));
}

void rcu_check_callbacks(int cpu, int user)
{
  if (user ||
      (!cpu->idle_cpu() && !in_softirq()) &&
      hardirq_count() <= (1 << HARDIRQ_SHIFT)) {
    rcu_qscgr_inc(cpu);
    rcu_bh_qscgr_inc(cpu);
  } else if (!in_softirq()) {
    rcu_bh_qscgr_inc(cpu);
  }
  raise_softirq(RCU_SOFTIRQ);
}
```

Figure D.23: rcu_check_callbacks() Code

if RCU needs the current CPU to do something. Line 7 checks to see if the current CPU owes RCU a quiescent state for the current grace period, line 9 invokes cpu_has_callbacks_ready_to_invoke() to see if the current CPU has callbacks whose grace period has ended, thus being ready to invoke, line 11 invokes cpu_needs_another_gp() to see if the current CPU has callbacks that need another RCU grace period to elapse, line 13 checks to see if the current grace period has ended, line 15 checks to see if a new grace period has started, and, finally, lines 17-22 check to see if it is time to attempt to force holdout CPUs to pass through a quiescent state. This latter check breaks down as follows: (1) lines 17-18 check to see if there is a grace period in progress, and, if so, lines 19-22 check to see if sufficient jiffies (lines 19-20) or calls to rcu_pending() (lines 21-22) have elapsed that force_quiescent_state() should be invoked. If none of the checks in the series triggers, then line 24 returns zero, indicating that rcu_check_callbacks() need not be invoked.

Lines 27-33 show rcu_pending(), which simply invokes __rcu_pending() twice, once for “rcu” and again for “rcu_bh”.

Quick Quiz D.29: Given that rcu_pending() is always called twice on lines 29-32 of Figure D.23, shouldn’t there be some way to combine the checks of the two structures?

Lines 35-48 show rcu_check_callbacks(), which checks to see if the scheduling-clock interrupt interrupted an extended quiescent state, and then initiates RCU’s softirq processing (rcu_process_callbacks()). Lines 37-41 perform this check for “rcu”, while lines 42-43 perform the check for “rcu_bh”.

Lines 37-39 check to see if the scheduling clock interrupt came from user-mode execution (line 37) or directly from the idle loop (line 38’s idle_cpu() invocation) with no intervening levels of interrupt (the remainder of line 38 and all of line 39). If this check succeeds, so that the scheduling clock interrupt did come from an extended quiescent state, then because any quiescent state for “rcu” is also a quiescent state for “rcu_bh”, lines 40 and 41 report the quiescent state for both flavors of RCU.

Similarly for “rcu_bh”, line 42 checks to see if the scheduling-clock interrupt came from a region of code with softirqs enabled, and, if so line 43 reports the quiescent state for “rcu_bh” only.

Quick Quiz D.30: Shouldn’t line 42 of Figure D.23 also check for in_hardirq()?

In either case, line 45 invokes an RCU softirq, which
static void __rcu_process_callbacks(struct rcu_state *rsp, struct rcu_data *rdp) {

unsigned long flags;

if ((long)(ACCESS_ONCE(rsp->jiffies_force_qs) - jiffies) < 0 ||
    (rdp->n_rcu_pending_force_qs - rdp->n_rcu_pending) < 0) {
force_quiescent_state(rsp, 1);
rcu_process_gp_end(rsp, rdp);
rcu_check_quiescent_state(rsp, rdp);
if (cpu_needs_another_gp(rsp, rdp)) {
    spin_lock_irqsave(&rcu_get_root(rsp)->lock, flags);
    rcu_start_gp(rsp, flags);
}
}
rcu_do_batch(rdp);
}

static void rcu_process_callbacks(struct softirq_action *unused) {
smp_mb();
__rcu_process_callbacks(&rcu_state, 
&__get_cpu_var(rcu_data));
__rcu_process_callbacks(&rcu_bh_state, 
&__get_cpu_var(rcu_bh_data));
smp_mb();
}

rcu_process_callbacks() Code

Figure D.24: rcu_process_callbacks() Code

will result in rcu_process_callbacks() being called on this CPU at some future time (like when interrupts are re-enabled after exiting the scheduler-clock interrupt).

D.3.2.4 rcu_process_callbacks()

Figure D.24 shows the code for rcu_process_callbacks(), which is a wrapper around __rcu_process_callbacks(). These functions are invoked as a result of a call to raise_softirq(RCU_SOFTIRQ), for example, line 47 of Figure D.23, which is normally done if there is reason to believe that the RCU core needs this CPU to do something.

Lines 7-10 check to see if it has been awhile since the current grace period started, and, if so, line 11 invokes force_quiescent_state() in order to try to convince holdout CPUs to pass through a quiescent state for this grace period.

Quick Quiz D.31: But don’t we also need to check that a grace period is actually in progress in __rcu_process_callbacks in Figure D.24? ■

In any case, line 12 invokes rcu_process_gp_end(), which checks to see if some other CPU ended the last grace period that this CPU was aware of, and, if so, notes the end of the grace period and advances this CPU’s RCU callbacks accordingly. Line 13 invokes rcu_check_quiescent_state(), which checks to see if some other CPU has started a new grace period, and also whether the current CPU has passed through a quiescent state for the current grace period, updating state appropriately if so. Line 14 checks to see if there is no grace period in progress and whether the current CPU has callbacks that need another grace period. If so, line 15 acquires the root rcu_node structure’s lock, and line 17 invokes rcu_start_gp(), which starts a new grace period (and also releases the root rcu_node structure’s lock). In either case, line 18 invokes rcu_do_batch(), which invokes any of this CPU’s callbacks whose grace period has completed.

Quick Quiz D.32: What happens if two CPUs attempt to start a new grace period concurrently in Figure D.24? ■

Lines 21-30 are rcu_process_callbacks(), which is again a wrapper for __rcu_process_callbacks(). Line 24 executes a memory barrier to ensure that any prior RCU read-side critical sections are seen to have ended before any subsequent RCU processing. Lines 25-26 and 27-28 invoke __rcu_process_callbacks() for “rcu” and “rcu_bh”, respectively, and, finally, line 29 executes a memory barrier to ensure that any RCU processing carried out by __rcu_process_callbacks() is seen prior to any subsequent RCU read-side critical sections.

D.3.2.5 rcu_needs_cpu() and rcu_cpu_notify()

Figure D.25 shows the code for rcu_needs_cpu() and rcu_cpu_notify(), which are invoked by the Linux kernel to check on switching to dynticks-idle mode and to handle CPU hotplug, respectively.

Lines 1-5 show rcu_needs_cpu(), which simply checks if the specified CPU has either “rcu” (line 3) or “rcu_bh” (line 4) callbacks.

Lines 7-28 show rcu_cpu_notify(), which is a very typical CPU-hotplug notifier function with the typical switch statement. Line 16 invokes rcu_online_cpu() if the specified CPU is going to be coming online, and line 22 invokes rcu_offline_cpu() if the specified CPU has gone to be going offline. It is important to note that CPU-hotplug operations are not atomic, but rather happen in stages that can extend for multiple grace periods. RCU must therefore gracefully handle CPUs that are in the process of coming or going.
D.3. HIERARCHICAL RCU CODE WALKTHROUGH

This section walks through the initialization code, which links the main data structures together as shown in Figure D.26. The yellow region represents fields in the `rcu_state` data structure, including the `->node` array, individual elements of which are shown in pink, matching the convention used in Section D.2. The blue boxes each represent one `rcu_data` structure, and the group of blue boxes makes up a set of per-CPU `rcu_data` structures.

The `->levelcnt[]` array is initialized at compile time, as is `->level[0]`, but the rest of the values and pointers are filled in by the functions described in the following sections. The figure shows a two-level hierarchy, but one-level and three-level hierarchies are possible as well. Each element of the `->levelsread[]` array gives the number of children per node at the corresponding level of the hierarchy. In the figure, therefore, the root node has two children and the nodes at the leaf level each have three children. Each element of the `levelcnt[]` array indicates how many nodes there are on the corresponding level of the hierarchy: 1 at the root level, 2 at the leaf level, and 6 at the `rcu_data` level—and any extra elements are unused and left as zero. Each element of the `->level[]` array references the first node of the cor-

---

```
1 int rcu_needs_cpu(int cpu)
2 {
3     return per_cpu(rcu_data, cpu).nxtlist ||
4         per_cpu(rcu_bh_data, cpu).nxtlist;
5 }

7 static int __cpuinit
8 rcu_cpu_notify(struct notifier_block *self,
9         unsigned long action, void *hcpu)
10  {
11     long cpu = (long)hcpu;
12     switch (action) {
13     case CPU_UP_PREPARE:
14     case CPU_UP_PREPARE_FROZEN:
15         rcu_online_cpu(cpu);
16         break;
17     case CPU_ONLINE:
18     case CPU_DEAD:
19     case CPU_DEAD_FROZEN:
20         rcu_offline_cpu(cpu);
21     case CPU_UP_CANCELED:
22         rcu_cancelled_cpu(cpu);
23         break;
24     default:
25         break;
26     }
27     return NOTIFY_OK;
28 }
```

---

Figure D.25: `rcu_needs_cpu()` and `rcu_cpu_notify` Code

Figure D.26: Initialized RCU Data Layout

D.3.3 Initialization
D.3.3.1 rcu_init_levelspread()

Figure D.27 shows the code for the rcu_init_levelspread() function, which controls the fanout, or the number of children per parent, in the rcu_node hierarchy. There are two versions of this function, one shown on lines 2-9 that enforces the exact fanout (specified by CONFIG_RCU_FANOUT_EXACT), and the other on lines 11-25 that determines the number of child nodes based indirectly on the specified fanout, but then balances the tree. The CONFIG_RCU_FANOUT_EXACT kernel parameter selects which version to use for a given kernel build.

The exact-fanout version simply assigns all of the elements of the specified rcu_state structure’s ->levelspread array to the CONFIG_RCU_FANOUT kernel parameter, as shown by the loop on lines 7 and 8.

The hierarchy-balancing version on lines 11-24 uses a pair of local variables ccur and cprv which track the number of rcu_node structures on the current and previous levels, respectively. This function works from the leaf level up the hierarchy, so cprv is initialized by line 18 to NR_CPUS, which corresponds to the number of rcu_data structures that feed into the leaf level.

Lines 19-23 iterate from the leaf to the root. Within this loop, line 20 picking up the number of rcu_node structures for the current level into ccur. Line 21 then rounds up the ratio of the number of nodes on the previous (lower) level (be they rcu_node or rcu_data) to the number of rcu_node structures on the current level, placing the result in the specified rcu_state structure’s ->levelspread array. Line 22 then sets up for the next pass through the loop.

After a call to either function, the ->levelspread array contains the number of children for each level of the rcu_node hierarchy.

D.3.3.2 rcu_init_one()

Figure D.28 shows the code for rcu_init_one(), which does boot-time initialization for the specified rcu_state structure.

Recall from Section D.3.1.4 that the ->levelcnt[] array in the rcu_state structure is compile-time initialized to the number of nodes at each level of the hierarchy starting from the root, with an additional element in the array initialized to the maximum possible number of CPUs, NR_CPUS. In addition, the first element of the ->level[] array is compile-time initialized to reference to the root rcu_node structure, which is in turn the first element of the ->node[] array in the rcu_state structure. This array is further laid out in breadth-first order. Keeping all of this in mind, the loop at lines 8-10 initializes the rest of the ->level[] array to reference the first rcu_node structure of each level of the rcu_node hierarchy.

Line 11 then invokes rcu_init_levelspread(), which fills in the ->levelspread[] array, as was described in Section D.3.3.1. The auxiliary arrays are then fully initialized, and thus ready for the loop from lines 15-35, each pass through which initializes one level of the rcu_node hierarchy, starting from the leaves.

Line 13 computes the number of CPUs per rcu_node structure for the current level of the hierarchy, and line 14 obtains a pointer to the first rcu_node structure on the
D.3. HIERARCHICAL RCU CODE WALKTHROUGH

```
1 static void __init rcu_init_one(struct rcu_state *rsp)
2 {
3   int cpustride = 1;
4   int i;
5   int j;
6   struct rcu_node *rnp;
7   for (i = 1; i < NUM_RCU_LVLS; i++)
8     rsp->level[i] = rsp->level[i - 1] +
9                      rsp->levelcnt[i - 1];
10   rcu_init_levelspread(rsp);
11   for (i = NUM_RCU_LVLS - 1; i >= 0; i--) {
12     cpustride *= rsp->levelspread[i];
13     for (j = 0; j < rsp->levelcnt[i]; j++, rnp++) {
14       spin_lock_init(&rnp->lock);
15       rnp->qsmask = 0;
16       rnp->qsmaskinit = 0;
17       rnp->grplo = j * cpustride;
18       rnp->grphi = (j + 1) * cpustride - 1;
19       if (rnp->grphi >= NR_CPUS)
20         rnp->grphi = NR_CPUS - 1;
21       if (i == 0) {
22         rnp->grpnum = 0;
23         rnp->grpmask = 0;
24         rnp->parent = NULL;
25       } else {
26         rnp->grpnum = j % rsp->levelspread[i - 1];
27         rnp->grpmask = LVL << rnp->grpnum;
28         rnp->parent = rsp->level[i - 1] +
29            j / rsp->levelspread[i - 1];
30       }
31       rnp->level = i;
32     }
33   }
34 }
```

Figure D.28: rcu_init_one() Code

current level of the hierarchy, in preparation for the loop from lines 15-34, each pass through which initializes one rcu_node structure.

Lines 16-18 initialize the rcu_node structure's spin-lock and its CPU masks. The qsmaskinit field will have bits set as CPUs come online later in boot, and the qsmask field will have bits set when the first grace period starts. Line 19 sets the ->grplo field to the number of the this rcu_node structure's first CPU and line 20 sets the ->grphi to the number of this rcu_node structure's last CPU. If the last rcu_node structure on a given level of the hierarchy is only partially full, lines 21 and 22 set its ->grphi field to the number of the last possible CPU in the system.

Lines 24-26 initialize the ->grpnum, ->grpmask, and ->parent fields for the root rcu_node structure, which has no parent, hence the zeroes and NULL. Lines 28-31 initialize these same fields for the rest of the rcu_node structures in the hierarchy. Line 28 computes the ->grpnum field as the index of this rcu_node structure within the set having the same parent, and line 29 sets the corresponding bit in the ->grpmask field. Finally, lines 30-31 places a pointer to the parent node into the ->parent field. These three fields will used to propagate quiescent states up the hierarchy.

Finally, line 33 records the hierarchy level in ->level, which is used for tracing when traversing the full hierarchy.

D.3.3.3 __rcu_init()

Figure D.29 shows the __rcu_init() function and its RCU_DATA_PTR_INIT() helper macro. The __rcu_init() function is invoked during early boot, before the scheduler has initialized, and before more than one CPU is running.

The RCU_DATA_PTR_INIT() macro takes as arguments a pointer to an rcu_state structure and the name of a set of rcu_data per-CPU variables. This macro scans the per-CPU rcu_data structures, assigning the ->mynode pointer of each rcu_data structure to point to the corresponding leaf rcu_node structure. It also fills out the specified rcu_state structure's ->rda[] array entries to each point to the corresponding rcu_data structure. Line 3 picks up a pointer to the first leaf rcu_node structure in local variable rnp (which must be declared by the invoker of this macro), and line 4 sets local variable j to the corresponding leaf-node number of zero. Each pass through the loop spanning lines 5-10 performs initialization for the corresponding potential
APPENDIX D. READ-COPY UPDATE IMPLEMENTATIONS

```c
#define RCU_DATA_PTR_INIT(rsp, rcu_data) do { 
    rnp = (rsp)->level[NUM_RCU_LVLS - 1]; 
    j = 0; 
    for_each_possible_cpu(i) { 
        if (i > rnp[j].grphi) 
            j++; 
        per_cpu(rcu_data, i).mynode = &rnp[j]; 
        (rsp)->rda[i] = &per_cpu(rcu_data, i); 
    } 
} while (0)

void __init __rcu_init(void)
{
    int i;
    int j;
    struct rcu_node *rnp;

    rcu_init_one(&rcu_state);
    RCU_DATA_PTR_INIT(&rcu_state, rcu_data);
    rcu_init_one(&rcu_bh_state);
    RCU_DATA_PTR_INIT(&rcu_bh_state, rcu_bh_data);

    for_each_online_cpu(i)
        rcu_cpu_notify(&rcu_nb, CPU_UP_PREPARE, 
                       (void *)(long)i);
    register_cpu_notifier(&rcu_nb);
}
```

Figure D.29: __rcu_init() Code

CPU (as specified by NR_CPUS). Within this loop, line 6 checks to see if we have moved beyond the bounds of the current leaf rcu_node structure, and, if so, line 7 advances to the next structure. Then, still within the loop, line 8 sets the ->mynode pointer of the current CPU’s rcu_data structure to reference the current leaf rcu_node structure, and line 9 sets the current CPU’s ->rda[] element (within the rcu_state structure) to reference the current CPU’s rcu_data structure.

Quick Quiz D.34: C-preprocessor macros are so 1990s! Why not get with the times and convert RCU_DATA_PTR_INIT() in Figure D.29 to be a function?

The __rcu_init() function first invokes rcu_init_one() on the rcu_state structure on line 19, then invokes RCU_DATA_PTR_INIT() on the rcu_state structure and the rcu_data set of per-CPU variables. It then repeats this for rcu_bh_state and rcu_bh_data on lines 21-22. The loop spanning lines 24-26 invokes rcu_cpu_notify() for each CPU that is currently online (which should be only the boot CPU), and line 27 registers a notifier so that rcu_cpu_notify() will be invoked each time a CPU comes online, in order to inform RCU of its presence.

Quick Quiz D.35: What happens if a CPU comes online between the time that the last online CPU is notified on lines 25-26 of Figure D.29 and the time that register_cpu_notifier() is invoked on line 27?

The rcu_cpu_notify() and related functions are discussed in Section D.3.4 below.

D.3.4 CPU Hotplug

The CPU-hotplug functions described in the following sections allow RCU to track which CPUs are and are not present, but also complete initialization of each CPU’s rcu_data structure as that CPU comes online.

D.3.4.1 rcu_init_percpu_data()

Figure D.30 shows the code for rcu_init_percpu_data(), which initializes the specified CPU’s rcu_
data structure in response to booting up or to that CPU coming online. It also sets up the `rcu_node` hierarchy so that this CPU will participate in future grace periods.

Line 8 gets a pointer to this CPU’s `rcu_data` structure, based on the specified `rcu_state` structure, and places this pointer into the local variable `rdp`. Line 9 gets a pointer to the root `rcu_node` structure for the specified `rcu_state` structure, placing it in local variable `rnp`.

Lines 11-29 initialize the fields of the `rcu_data` structure under the protection of the root `rcu_node` structure’s lock in order to ensure consistent values. Line 17 is important for tracing, due to the fact that many Linux distributions set `NR_CPUS` to a very large number, which could result in excessive output when tracing `rcu_data` structures. The `->beenonline` field is used to solve this problem, as it will be set to the value one on any `rcu_data` structure corresponding to a CPU that has ever been online, and set to zero for all other `rcu_data` structures. This allows the tracing code to easily ignore irrelevant CPUs.

Lines 30-40 propagate the onlining CPU’s bit up the `rcu_node` hierarchy, proceeding until either the root `rcu_node` is reached or until the corresponding bit is already set, whichever comes first. This bit-setting is done under the protection of `->onofflock` in order to exclude initialization of a new grace period, and, in addition, each `rcu_node` structure is initialized under the protection of its lock. Line 41 then invokes `cpu_quiet()` to signal RCU that this CPU has been in an extended quiescent state, and finally, line 42 re-enables iqs.

Quick Quiz D.36: Why call `cpu_quiet()` on line 41 of Figure D.30, given that we are excluding grace periods with various locks, and given that any earlier grace periods would not have been waiting on this previously-offlined CPU?

It is important to note that `rcu_init_percpu_data()` is invoked not only at boot time, but also every time that a given CPU is brought online.

D.3.4.2 `rcu_online_cpu()`

Figure D.31 shows the code for `rcu_online_cpu()`, which informs RCU that the specified CPU is coming online.

When dynticks (CONFIG_NO_HZ) is enabled, line 6 obtains a reference to the specified CPU’s `rcu_dynticks` structure, which is shared between the “rcu” and “rcu_bh” implementations of RCU. Line 7 sets the `->dynticks_nesting` field to the value one, reflecting the fact that a newly onlined CPU is not in dynticks-idle mode (recall that the `->dynticks_nesting` field tracks the number of reasons that the corresponding CPU needs to be tracked for RCU read-side critical sections, in this case because it can run process-level code). Line 8 forces the `->dynticks` field to an odd value that is at least as large as the last value it had when previously online, again reflecting the fact that newly onlined CPUs are not in dynticks-idle mode, and line 9 forces the `->dynticks_nmi` field to an even value that is at least as large as the last value it had when previously online, reflecting the fact that this CPU is not currently executing in an NMI handler.

Lines 11-13 are executed regardless of the value of the CONFIG_NO_HZ kernel parameter. Line 11 initializes the specified CPU’s `rcu_data` structure for “rcu”, and line 12 does so for “rcu_bh”. Finally, line 13 registers the `rcu_process_callbacks()` to be invoked by subsequent `raise_softirq()` invocations on this CPU.

D.3.4.3 `rcu_offline_cpu()`

Figure D.32 shows the code for `rcu_offline_cpu()` and its wrapper function, `rcu_offline_cpu()`. The purpose of this wrapper function (shown in lines 43-47 of the figure) is simply to invoke `__rcu_offline_cpu()` twice, once for “rcu” and again for “rcu_bh”. The purpose of the `__rcu_offline_cpu()` function is to prevent future grace periods from waiting on the CPU being offlined, to note the extended quiescent state, and to find a new home for any RCU callbacks in process on this CPU.

Turning to `__rcu_offline_cpu()`, shown on lines 1-41 of the figure, line 12 acquires the speci-
static void __rcu_offline_cpu(int cpu, struct rcu_state *rsp)
{ int i;
    unsigned long flags;
    long lastcomp;
    unsigned long mask;
    struct rcu_data *rdp = rsp->rda[cpu];
    struct rcu_data *rdp_me;
    struct rcu_node *rnp;

    spin_lock_irqsave(&rsp->onofflock, flags);
    rnp = rdp->mynode;
    mask = rdp->grpmask;
    do {
        spin_lock(&rnp->lock);
        rnp->qsmaskinit &= ~mask;
        if (rnp->qsmaskinit != 0) {
            spin_unlock(&rnp->lock);
            break;
        }
        mask = rnp->grpmask;
        spin_unlock(&rnp->lock);
        rnp = rnp->parent;
    } while (rnp != NULL);
    lastcomp = rsp->completed;
    spin_unlock(&rsp->onofflock);
    cpu_quiet(cpu, rsp, rdp, lastcomp);
    rdp_me = rsp->rda[smp_processor_id()];
    if (rdp->nxtlist != NULL) {
        *rdp_me->nxttail[RCU_NEXT_TAIL] = rdp->nxtlist;
        rdp_me->nxttail[RCU_NEXT_TAIL] = rdp->nxtlist;
        for (i = 0; i < RCU_NEXT_SIZE; i++)
            rdp->nxttail[i] = &rdp->nxtlist;
        rdp_me->qlen += rdp->qlen;
        rdp->qlen = 0;
    }
    local_irq_restore(flags);
}

static void rcu_offline_cpu(int cpu)
{ __rcu_offline_cpu(cpu, &rcu_state);
    __rcu_offline_cpu(cpu, &rcu_bh_state);
}

Figure D.32: rcu_offline_cpu() Code

APPENDIX D. READ-COPY UPDATE IMPLEMENTATIONS
extended quiescent state, why does line 28 of Figure D.32 need to care which grace period it is dealing with?

Lines 29-39 move any RCU callbacks from the CPU going offline to the currently running CPU. This operation must avoid reordering the callbacks being moved, otherwise rcu_barrier() will not work correctly. Line 29 puts a pointer to the currently running CPU’s rcu_data structure into local variable rdp_me. Line 30 then checks to see if the CPU going offline has any RCU callbacks. If so, lines 31-38 move them. Line 31 splices the list of callbacks onto the end of the running CPU’s list. Lines 32-33 sets the running CPU’s callback tail pointer to that of the CPU going offline, and then lines 34-36 initialize the going-offline CPU’s list to be empty. Line 37 adds the length of the going-offline CPU’s callback list to that of the currently running CPU, and, finally, line 38 zeroes the going-offline CPU’s list length.

Quick Quiz D.41: But this list movement in Figure D.32 makes all of the going-offline CPU’s callbacks go through another grace period, even if they were ready to invoke. Isn’t that inefficient? Furthermore, couldn’t an unfortunate pattern of CPUs going offline then coming back online prevent a given callback from ever being invoked?

Finally, line 40 re-enables irqs.

### D.3.5 Miscellaneous Functions

This section describes the miscellaneous utility functions:

1. `rcu_batches_completed`
2. `rcu_batches_completed_bh`
3. `cpu_hascallbacks_ready_to_invoke`
4. `cpu_needs_another_gp`
5. `rcu_get_root`

Figure D.33 shows a number of miscellaneous functions. Lines 1-9 show `rcu_batches_completed()` and `rcu_batches_completed_bh()`, which are used by the rcutorture test suite. Lines 11-15 show `cpu_hascallbacks_ready_to_invoke()`, which indicates whether the specified `rcu_data` structure has RCU callbacks that have passed through their grace period, which is indicated by the “done” tail pointer no longer pointing to the head of the list. Lines 17-24 show `cpu_needs_another_gp()`, which indicates whether the CPU corresponding to the specified `rcu_data` structure requires an additional grace period during a time when no grace period is in progress. Note that the specified `rcu_data` structure is required to be associated with the specified `rcu_state` structure. Finally, lines 26-30 show `rcu_get_root()`, which returns the root `rcu_node` structure associated with the specified `rcu_state` structure.

### D.3.6 Grace-Period-Detection Functions

This section covers functions that are directly involved in detecting beginnings and ends of grace periods. This of course includes actually starting and ending grace periods, but also includes noting when other CPUs have started or ended grace periods.

#### D.3.6.1 Noting New Grace Periods

The main purpose of Hierarchical RCU is to detect grace periods, and the functions more directly involved in this task are described in this section. Section D.3.6.1 covers functions that allow CPUs to note that a new grace period has begun, Section D.3.6.2 covers functions that allow CPUs to note that an existing grace period has ended, Section D.3.6.3 covers `rcu_start_gp()`, which starts a new grace period, and Section D.3.6.4 covers functions...

```c
1 long rcu_batches_completed(void)
2 {
3 return rcu_state.completed;
4 }
5
6 long rcu_batches_completed_bh(void)
7 {
8 return rcu_bh_state.completed;
9 }
10
11 static int
12 cpu_hascallbacks_ready_to_invoke(struct rcu_data *rdp)
13 {
14 return &rdp->nxtlist != rdp->nxttail[RCU_DONE_TAIL];
15 }
16
17 static int
18 cpu_needs_another_gp(struct rcu_state *rsp,
19 struct rcu_data *rdp)
20 {
21 return *rdp->nxttail[RCU_DONE_TAIL] &&
22 ACCESS_ONCE(rsp->completed) ==
23 ACCESS_ONCE(rsp->gpnum);
24 }
25
26 static struct rcu_node
27 *rcu_get_root(struct rcu_state *rsp)
28 {
29 return &rsp->node[0];
30 }
```

Figure D.33: Miscellaneous Functions
static void note_new_gpnum(struct rcu_state *rsp, 
        struct rcu_data *rdp) {
    rdp->qs_pending = 1;
    rdp->passed_quiesc = 0;
    rdp->gpnum = rsp->gpnum;
    rdp->n_rcu_pending_force_qs = rdp->n_rcu_pending + 
        RCU_JIFFIES_TILL_FORCE_QS;
}

static int check_for_new_grace_period(struct rcu_state *rsp, 
        struct rcu_data *rdp) {
    unsigned long flags;
    int ret = 0;
    local_irq_save(flags);
    if (rdp->gpnum != rsp->gpnum) {
        note_new_gpnum(rsp, rdp);
        ret = 1;
    }
    local_irq_restore(flags);
    return ret;
}

Figure D.34: Noting New Grace Periods

Figure D.35: Noting End of Old Grace Periods

Figure D.36: RCU Callback List

D.3.6.2 Noting End of Old Grace Periods

Figure D.35 shows rcu_process_gp_end(), which is invoked when a CPU suspects that a grace period might have ended (possibly because the CPU in question in fact ended the grace period). If a grace period really has ended, then this function advances the current CPU’s
RCU callbacks, which are managed as a singly linked list with multiple tail pointers, as shown in Figure D.36. This multiple tail pointer layout, spearheaded by Lai Jiangshan, simplifies list handling [Jia08]. In this figure, the blue box represents one CPU’s rcu_data structure, with the six white boxes at the bottom of the diagram representing a list of six RCU callbacks (rcu_head structures). In this list, the first three callbacks have passed through their grace period and are thus waiting to be invoked, the fourth callback (the first on the second line) is waiting for the current grace period to complete, and the last two are waiting for the next grace period. The last two tail pointers reference the last element, so that the final sublist, which would comprise callbacks that had not yet been associated with a specific grace period, is empty.

Lines 8 and 19 of Figure D.35 suppress and re-enable interrupts, respectively. Line 9 picks up a snapshot of the rcu_state structure’s ->completed field, storing it in the local variable completed_snap. Line 10 checks to see if the current CPU is not yet aware of the end of a grace period, and if it is not aware, lines 11-16 advance this CPU’s RCU callbacks by manipulating the tail pointers. Line 17 then records the most recently completed grace period number in this CPU’s rcu_data structure in the ->completed field.

### D.3.6.3 Starting a Grace Period

Figure D.37 shows rcu_start_gp(), which starts a new grace period, also releasing the root rcu_node structure’s lock, which must be acquired by the caller.

Line 4 is annotation for the sparse utility, indicating that rcu_start_gp() releases the root rcu_node structure’s lock. Local variable rdp references the running CPU’s rcu_data structure, rnp references the root rcu_node structure, and rnp_cur and rnp_end are used as cursors in traversing the rcu_node hierarchy.

Line 10 invokes cpu_needs_another_gp() to see if this CPU really needs another grace period to be started, and if not, line 11 releases the root rcu_node structure’s lock and line 12 returns. This code path can be executed due to multiple CPUs concurrently attempting to start a grace period. In this case, the winner will start the grace period, and the losers will exit out via this code path.

Otherwise, line 14 increments the specified rcu_state structure’s ->gpnum field, officially marking the start of a new grace period.

**Quick Quiz D.43:** But there has been no initialization

```c
1 static void
2 rcu_start_gp(struct rcu_state *rsp, unsigned long flags)
3 __releases(rcu_get_root(rsp)->lock)
4 {
5 struct rcu_data *rdp = rsp->rda[smp_processor_id()];
6 struct rcu_node *rnp = rcu_get_root(rsp);
7 struct rcu_node *rnp_cur;
8 struct rcu_node *rnp_end;
9
10 if (!cpu_needs_another_gp(rsp, rdp)) {
11 spin_unlock_irqrestore(&rnp->lock, flags);
12 return;
13 }
14 rsp->gpnum++;
15 rsp->signaled = RCU_GP_INIT;
16 rsp->jiffies_force_qs = jiffies +
17 RCU_JIFFIES_TILL_FORCE_QS;
18 rnp->n_rcu_pending_force_qs = rdp->n_rcu_pending +
19 RCU_JIFFIES_TILL_FORCE_QS;
20 record_gp_stall_check_time(rsp);
21 dyntick_record_completed(rsp, rdp->completed - 1);
22 note_new_gpnum(rsp, rdp);
23 rdp->nxttail[RCU_NEXT_READY_TAIL] =
24 rdp->nxttail[RCU_NEXT_TAIL];
25 rdp->nxttail[RCU_WAIT_TAIL] =
26 rdp->nxttail[RCU_NEXT_TAIL];
27 if (NUM_RCU_NODES == 1) {
28 rnp->qsmask = rnp->qsmaskinit;
29 spin_unlock_irqrestore(&rnp->lock, flags);
30 return;
31 }
32 spin_unlock(&rnp->lock);
33 spin_lock(&rsp->onofflock);
34 rnp_end = rsp->level[NUM_RCU_LVLS - 1];
35 rnp_cur = &rsp->node[0];
36 for (; rnp_cur < rnp_end; rnp_cur++)
37 rnp_cur->qsmask = rnp_cur->qsmaskinit;
38 rnp_end = &rsp->node[NUM_RCU_NODES];
39 rnp_cur = rsp->level[NUM_RCU_LVLS - 1];
40 for (; rnp_cur < rnp_end; rnp_cur++)
41 spin_lock(&rnp_cur->lock);
42 rnp_cur->qsmask = rnp_cur->qsmaskinit;
43 spin_unlock(&rnp_cur->lock);
44 }
45 rnp->qsmask = rnp->qsmaskinit;
46 spin_unlock_irqrestore(&rsp->onofflock, flags);
47 }
```

Figure D.37: Starting a Grace Period
yet at line 15 of Figure D.37! What happens if a CPU notices the new grace period and immediately attempts to report a quiescent state? Won’t it get confused? ■

Line 15 sets the ->signaled field to RCU_GP_INIT in order to prevent any other CPU from attempting to force an end to the new grace period before its initialization completes. Lines 16-18 schedule the next attempt to force an end to the new grace period, first in terms of jiffies and second in terms of the number of calls to rcu_pending. Of course, if the grace period ends naturally before that time, there will be no need to attempt to force it. Line 20 invokes record_gp_stall_check_time() to schedule a longer-term progress check—if the grace period extends beyond this time, it should be considered to be an error. Line 22 invokes note_new_gpnum() in order to initialize this CPU’s rcu_data structure to account for the new grace period.

Lines 23-26 advance all of this CPU’s callbacks so that they will be eligible to be invoked at the end of this new grace period. This represents an acceleration of callbacks, as other CPUs would only be able to move the RCU_NEXT_READY_TAIL batch to be serviced by the current grace period; the RCU_NEXT_TAIL would instead need to be advanced to the RCU_NEXT_READY_TAIL batch. The reason that this CPU can accelerate the RCU_NEXT_TAIL batch is that it knows exactly when this new grace period started. In contrast, other CPUs would be unable to correctly resolve the race between the start of a new grace period and the arrival of a new RCU callback.

Line 27 checks to see if there is but one rcu_node structure in the hierarchy, and if so, line 28 sets the ->qmask bits corresponding to all online CPUs, in other words, corresponding to those CPUs that must pass through a quiescent state for the new grace period to end. Line 29 releases the root rcu_node structure’s lock and line 30 returns. In this case, gcc’s dead-code elimination is expected to dispense with lines 32-46.

Otherwise, the rcu_node hierarchy has multiple structures, requiring a more involved initialization scheme. Line 32 releases the root rcu_node structure’s lock, but keeps interrupts disabled, and then line 33 acquires the specified rcu_state structure’s ->onofflock, preventing any concurrent CPU-hotplug operations from manipulating RCU-specific state.

Line 34 sets the rnp_end local variable to reference the first leaf rcu_node structure, which also happens to be the rcu_node structure immediately following the last non-leaf rcu_node structure in the ->node array. Line 35 sets the rnp_cur local variable to reference the root rcu_node structure, which also happens to be first such structure in the ->node array. Lines 36 and 37 then traverse all of the non-leaf rcu_node structures, setting the bits corresponding to lower-level rcu_node structures that have CPUs that must pass through quiescent states in order for the new grace period to end.

Quick Quiz D.44: Hey! Shouldn’t we hold the non-leaf rcu_node structures’ locks when munging their state in line 37 of Figure D.37?? ■

Line 38 sets local variable rnp_end to one past the last leaf rcu_node structure, and line 39 sets local variable rnp_cur to the first leaf rcu_node structure, so that the loop spanning lines 40-44 traverses all leaves of the rcu_node hierarchy. During each pass through this loop, line 41 acquires the current leaf rcu_node structure’s lock, line 42 sets the bits corresponding to online CPUs (each of which must pass through a quiescent state before the new grace period can end), and line 43 releases the lock.

Quick Quiz D.45: Why can’t we merge the loop spanning lines 36-37 with the loop spanning lines 40-44 in Figure D.37?? ■

Line 45 then sets the specified rcu_state structure’s ->signaled field to permit forcing of quiescent states, and line 46 releases the ->onofflock to permit CPU-hotplug operations to manipulate RCU state.

D.3.6.4 Reporting Quiescent States

This hierarchical RCU implementation implements a layered approach to reporting quiescent states, using the following functions:

1. rcu_qsctr_inc() and rcu_bh_qsctr_inc() are invoked when a given CPU passes through a quiescent state for “rcu” and “rcu_bh”, respectively. Note that the dynticks-idle and CPU-offline quiescent states are handled specially, due to the fact that such a CPU is not executing, and thus is unable to report itself as being in a quiescent state.

2. rcu_check_quiescent_state() checks to see if the current CPU has passed through a quiescent state, invoking cpu_quiet() if so.

3. cpu_quiet() reports the specified CPU as having passed through a quiescent state by invoking cpu_quiet_mask(). The specified CPU must either be the current CPU or an offline CPU.
D.3. HIERARCHICAL RCU CODE WALKTHROUGH

1 void rcu_qsctr_inc(int cpu)
2 {
3 struct rcu_data *rdp = &per_cpu(rcu_data, cpu);
4 rdp->passed_quiesc = 1;
5 rdp->passed_quiesc_completed = rdp->completed;
6 }
7
8 void rcu_bh_qsctr_inc(int cpu)
9 {
10 struct rcu_data *rdp = &per_cpu(rcu_bh_data, cpu);
11 rdp->passed_quiesc = 1;
12 rdp->passed_quiesc_completed = rdp->completed;
13 }

Figure D.38: Code for Recording Quiescent States

1 static void
2 rcu_check_quiescent_state(struct rcu_state *rsp,
3 struct rcu_data *rdp)
4 {
5 if (check_for_new_grace_period(rsp, rdp))
6 return;
7 if (!rdp->qs_pending)
8 return;
9 if (!rdp->passed_quiesc)
10 return;
11 cpu_quiet(rdp->cpu, rsp, rdp,
12 rdp->passed_quiesc_completed);
13 }

Figure D.39: Code for rcu_check_quiescent_state()

4. cpu_quiet_msk() reports the specified vector of CPUs as having passed through a quiescent state. The CPUs in the vector need not be the current CPU, nor must they be offline.

Each of these functions is described below.

Figure D.38 shows the code for rcu_qsctr_inc() and rcu_bh_qsctr_inc(), which note the current CPU’s passage through a quiescent state.

Line 3 of rcu_qsctr_inc() obtains a pointer to the specified CPU’s rcu_data structure (which corresponds to “rcu” as opposed to “rcu_bh”). Line 4 sets the ->passed_quiesc field, recording the quiescent state. Line 5 sets the ->passed_quiesc_completed field to the number of the last completed grace period that this CPU knows of (which is stored in the ->completed field of the rcu_data structure).

The rcu_bh_qsctr_inc() function operates in the same manner, the only difference being that line 10 obtains the rcu_data pointer from the rcu_bh_data per-CPU variable rather than the rcu_data per-CPU variable.

Figure D.39 shows the code for rcu_check_quiescent_state(), which is invoked from rcu_process_callbacks() (described in Section D.3.2.4) in order to determine when other CPUs have started a new grace period and to inform RCU of recent quiescent states for this CPU.

Line 5 invokes check_for_new_grace_period() to check for a new grace period having been started by some other CPU, and also updating this CPU’s local state to account for that new grace period. If a new grace period has just started, line 6 returns. Line 7 checks to see if RCU is still expecting a quiescent state from the current CPU, and line 8 returns if not. Line 9 checks to see if this CPU has passed through a quiescent state since the start of the current grace period (in other words, if rcu_qsctr_inc() or rcu_bh_qsctr_inc() have been invoked for “rcu” and “rcu_bh”, respectively), and line 10 returns if not.

Therefore, execution reaches line 11 only if a previously noted grace period is still in effect, if this CPU needs to pass through a quiescent state in order to allow this grace period to end, and if this CPU has passed through such a quiescent state. In this case, lines 11-12 invoke cpu_quiet() in order to report this quiescent state to RCU.

Quick Quiz D.46: What prevents lines 11-12 of Figure D.39 from reporting a quiescent state from a prior grace period against the current grace period? ■

Figure D.40 shows cpu_quiet, which is used to report a quiescent state for the specified CPU. As noted
earlier, this must either be the currently running CPU or a
CPU that is guaranteed to remain offline throughout.

Line 9 picks up a pointer to the leaf rcu_node structure
responsible for this CPU. Line 10 acquires this
leaf rcu_node structure’s lock and disables interrupts.
Line 11 checks to make sure that the specified grace pe-
riod is still in effect, and, if not, line 11 clears the indica-
tion that this CPU passed through a quiescent state (since
it belongs to a defunct grace period), line 13 releases the
lock and re-enables interrupts, and line 14 returns to the
caller.

Otherwise, line 16 forms a mask with the specified
CPU’s bit set. Line 17 checks to see if this bit is still set in
the leaf rcu_node structure, and, if not, line 18 releases the
lock and re-enables interrupts.

On the other hand, if the CPU’s bit is still set, line 20
clears ->qsmask, reflecting that this CPU has
passed through its quiescent state for this grace period.
Line 21 then overwrites local variable rdp with a pointer
to the running CPU’s rcu_data structure, and lines 22-
23 updates the running CPU’s RCU callbacks so that
all those not yet associated with a specific grace period
be serviced by the next grace period. Finally, line 24
clears bits up the rcu_node hierarchy, ending the cur-
cent grace period if appropriate and perhaps even starting
a new one. Note that cpu_quiet() releases the lock and re-enables interrupts.

Quick Quiz D.47: How do lines 22-23 of Figure D.40
know that it is safe to promote the running CPU’s RCU
callbacks? ■

Figure D.41 shows cpu_quiet_msk(), which up-
dates the rcu_node hierarchy to reflect the passage of the
CPUs indicated by argument mask through their res-
spective quiescent states. Note that argument rnp is the
leaf rcu_node structure corresponding to the specified
CPUs.

Quick Quiz D.48: Given that argument mask on line
2 of Figure D.41 is an unsigned long, how can it possibly
deal with systems with more than 64 CPUs? ■

Line 4 is annotation for the sparse utility, indicating
that cpu_quiet_msk() releases the leaf rcu_node
structure’s lock.

Each pass through the loop spanning lines 6-23 does
the required processing for one level of the rcu_node
hierarchy, traversing the data structures as shown by the
blue arrow in Figure D.42.

Line 7 checks to see if all of the bits in mask have
already been cleared in the current rcu_node structure’s
->qsmask field, and, if so, line 8 releases the lock and

```c
1 static void
2 cpu_quiet_msk(unsigned long mask, struct rcu_state *rsp,
3   struct rcu_node *rnp, unsigned long flags)
4   __releases(rnp->lock)
5   { for (;;) {  6   if (!((rnp->qsmask & mask)) {  7   spin_unlock_irqrestore(&rnp->lock, flags);
8   return;
9   } 10   } 11   rnp->qsmask &= ~mask;
12   if (rnp->qsmask == 0) { 13   spin_unlock_irqrestore(&rnp->lock, flags);
14   return;
15   }
16   mask = rnp->grpmask;
17   if (rnp->parent == NULL) { 18   break;
19   }
20   spin_unlock_irqrestore(&rnp->lock, flags);
21   rnp = rnp->parent;
22   spin_lock_irqsave(&rnp->lock, flags);
23   } 24   rnp->completed = rnp->gpnun;
25   rcu_process_gp_end(rsp, rnp, rda[omp_processor_id()]);
26   rcu_start_gp(rsp, flags);
27 }
```

Figure D.41: Code for cpu_quiet_msk() re-enables interrupts, and line 9 returns to the caller. If not, line 11 clears the bits specified by mask from the current rcu_node structure’s qsmask field. Line 12 then checks to see if there are more bits remaining in ->qsmask, and, if so, line 13 releases the lock and re-enables interrupts, and line 14 returns to the caller.

Otherwise, it is necessary to advance up to the next
level of the rcu_node hierarchy. In preparation for this next level, line 16 places a mask with the single bit set corresponding to the current rcu_node structure within its parent. Line 17 checks to see if there in fact is a parent for the current rcu_node structure, and, if not, line 18 breaks from the loop. On the other hand, if there is a parent rcu_node structure, line 20 releases the current rcu_node structure’s lock, line 21 advances the
rnp local variable to the parent, and line 22 acquires the
parent’s lock. Execution then continues at the beginning
of the loop on line 7.

If line 18 breaks from the loop, we know that the cur-
cent grace period has ended, as the only way that all
bits can be cleared in the root rcu_node structure is
if all CPUs have passed through quiescent states. In
this case, line 24 updates the rcu_state structure’s
->completed field to match the number of the newly
ended grace period, indicating that the grace period has in
fact ended. Line 24 then invokes rcu_process_gp_ end() to advance the running CPU’s RCU callbacks,
D.3. HIERARCHICAL RCU CODE WALKTHROUGH

and, finally, line 26 invokes `rcu_start_gp()` in order to start a new grace period should any remaining callbacks on the currently running CPU require one.

Figure D.43 shows `rcu_do_batch()`, which invokes RCU callbacks whose grace periods have ended. Only callbacks on the running CPU will be invoked—other CPUs must invoke their own callbacks.

**Quick Quiz D.49:** How do RCU callbacks on dynticks-idle or offline CPUs get invoked? ■

Line 7 invokes `cpu_has_callbacks_ready_to_invoke()` to see if this CPU has any RCU callbacks whose grace period has completed, and, if not, line 8 returns. Lines 9 and 18 disable and re-enable interrupts, respectively. Lines 11-13 remove the ready-to-involve callbacks from `->nxtlist`, and lines 14-17 make any needed adjustments to the tail pointers.

**Quick Quiz D.50:** Why would lines 14-17 in Figure D.43 need to adjust the tail pointers? ■

Line 19 initializes local variable `count` to zero in preparation for counting the number of callbacks that will actually be invoked. Each pass through the loop spanning lines 20-27 invokes and counts a callback, with lines 25-26 exiting the loop if too many callbacks are to be invoked at a time (thus preserving responsiveness). The remainder of the function then requeues any callbacks that could not be invoked due to this limit.

Lines 28 and 41 disable and re-enable interrupts, respectively. Line 29 updates the `->qlen` field, which maintains a count of the total number of RCU callbacks for this CPU. Line 30 checks to see if there were any ready-to-involve callbacks that could not be invoked at the moment due to the limit on the number that may be invoked at a given time. If such callbacks remain, lines 30-38 requeue them, again adjusting the tail pointers as needed. Lines 39-40 restore the batch limit if it was increased due to excessive callback backlog, and lines 42-43 cause additional RCU processing to be scheduled if there are any ready-to-involve callbacks remaining.

D.3.7 Dyntick-Idle Functions

The functions in this section are defined only in `CONFIG_NO_HZ` builds of the Linux kernel, though in some cases, extended-no-op versions are present otherwise. These functions control whether or not RCU pays attention to a given CPU. CPUs in dynticks-idle mode are ignored, but only if they are not currently in an interrupt or NMI handler. The functions in this section communicate this CPU state to RCU.

This set of functions is greatly simplified from that used
APPENDIX D. READ-COPY UPDATE IMPLEMENTATIONS

1 static void rcu_do_batch(struct rcu_data *rdp)
2 {
3     unsigned long flags;
4     struct rcu_head *next, *list, **tail;
5     int count;
6
7     if (!cpu_has_callbacks_ready_to_invoke(rdp))
8         return;
9     local_irq_save(flags);
10    list = rdp->nxtlist;
11    rdp->nxtlist = *rdp->nxttail[RCU_DONE_TAIL];
12    *rdp->nxttail[RCU_DONE_TAIL] = NULL;
13    tail = rdp->nxttail[RCU_DONE_TAIL];
14    for (count = RCU_NEXT_SIZE - 1; count >= 0; count--)
15        if (rdp->nxttail[count] == rdp->nxttail[RCU_DONE_TAIL])
16            rdp->nxttail[count] = &rdp->nxtlist;
17    local_irq_restore(flags);
18    count = 0;
19    while (list) {
20        next = list->next;
21        prefetch(next);
22        list->func(list);
23        list = next;
24        if (++count >= rdp->blimit) break;
25    }
26
27    local_irq_save(flags);
28    rdp->qlen -= count;
29    if (list != NULL) {
30        *tail = rdp->nxtlist;
31        rdp->nxtlist = list;
32        for (count = 0; count < RCU_NEXT_SIZE; count++)
33            if (&rdp->nxtlist == rdp->nxttail[count])
34                rdp->nxttail[count] = tail;
35        else break;
36    }
37
38    local_irq_restore(flags);
39    if (rdp->blimit == LONG_MAX && rdp->qlen <= qlowmark)
40        rdp->blimit = blimit;
41    local_irq_restore(flags);
42    if (!cpu_has_callbacks_ready_to_invoke(rdp))
43        raise_softirq(RCU_SOFTIRQ);
44 }

Figure D.43: Code for rcu_do_batch()
1 void rcu_nmi_enter(void)
2 {
3 struct rcu_dynticks *rdtp;
4
5 rdtp = &__get_cpu_var(rcu_dynticks);
6 if (rdtp->dynticks & 0x1)
7 return;
8 rdtp->dynticks_nmi++;
9 smp_mb();
10 }
11
12 void rcu_nmi_exit(void)
13 {
14 struct rcu_dynticks *rdtp;
15
16 rdtp = &__get_cpu_var(rcu_dynticks);
17 if (rdtp->dynticks & 0x1)
18 return;
19 smp_mb();
20 rdtp->dynticks_nmi++;
21
Figure D.45: NMIs from Dyntick-Idle Mode

nally line 10 decrements the \texttt{\textasciitilde dynticks\_nesting} field (which now must be zero to indicate that there is no reason to pay attention to this CPU).

Lines 19 and 23 of \texttt{rcu\_exit\_nohz()} disable and re-enable interrupts, again to avoid interference. Line 20 obtains a pointer to this CPU’s \texttt{rcu\_dynticks} structure, line 21 increments the \texttt{\textasciitilde dynticks} field (which now must be odd in order to indicate that RCU must once again pay attention to this CPU), and line 22 increments the \texttt{\textasciitilde dynticks\_nesting} field (which now must have the value 1 to indicate that there is one reason to pay attention to this CPU).

\textbf{D.3.7.2 NMIs from Dyntick-Idle Mode}

Figure D.45 shows \texttt{rcu\_nmi\_enter()} and \texttt{rcu\_nmi\_exit()}, which handle NMI entry and exit, respectively. It is important to keep in mind that entering an NMI handler exits dyntick-idle mode and vice versa, in other words, RCU must pay attention to CPUs that claim to be in dyntick-idle mode while they are executing NMI handlers, due to the fact that NMI handlers can contain RCU read-side critical sections. This reversal of roles can be quite confusing: you have been warned.

Line 5 of \texttt{rcu\_nmi\_enter()} obtains a pointer to this CPU’s \texttt{rcu\_dynticks} structure, and line 6 checks to see if this CPU is already under scrutiny by RCU, with line 7 silently returning if so. Otherwise, line 8 increments the \texttt{\textasciitilde dynticks\_nmi} field, which must now have an odd-numbered value. Finally, line 9 executes a memory barrier to ensure that the prior increment of \texttt{\textasciitilde dynticks\_nmi} is seen by all CPUs to happen before any subsequent RCU read-side critical section.

Line 16 of \texttt{rcu\_nmi\_exit()} again fetches a pointer to this CPU’s \texttt{rcu\_dynticks} structure, and line 17 checks to see if RCU would be paying attention to this CPU even if it were not in an NMI, with line 18 silently returning if so. Otherwise, line 19 executes a memory barrier to ensure that any RCU read-side critical sections within the handler are seen by all CPUs to happen before the increment of the \texttt{\textasciitilde dynticks\_nmi} field on line 20. The new value of this field must now be even.

\textbf{Quick Quiz D.51:} But how does the code in Figure D.45 handle nested NMI's?

\textbf{D.3.7.3 Interrupts from Dyntick-Idle Mode}

Figure D.46 shows \texttt{rcu\_irq\_enter()} and \texttt{rcu\_irq\_exit()}, which handle interrupt entry and exit, respectively. As with NMIs, it is important to note that entering an interrupt handler exits dyntick-idle mode and vice versa, due to the fact that RCU read-side critical sections can appear in interrupt handlers.

Line 5 of \texttt{rcu\_irq\_enter()} once again acquires a reference to the current CPU’s \texttt{rcu\_dynticks} structure. Line 6 increments the \texttt{\textasciitilde dynticks\_nesting} field, and if the original value was already non-zero (in other words, RCU was already paying attention to this CPU), line 7 silently returns. Otherwise, line 8 increments the \texttt{\textasciitilde dynticks} field, which then must have an odd-numbered value. Finally, line 9 executes a memory barrier
APPENDIX D. READ-COPY UPDATE IMPLEMENTATIONS

so that this increment is seen by all CPUs as happening before any RCU read-side critical sections that might be in the interrupt handler.

Line 16 of `rcu_irq_exit()` does the by-now traditional acquisition of a reference to the currently running CPU’s `rcu_dynticks` structure. Line 17 decrements the `->dynticks_nesting` field, and, if the result is non-zero (in other words, RCU must still pay attention to this CPU despite exiting this interrupt handler), then line 18 silently returns. Otherwise, line 19 executes a memory barrier so that any RCU read-side critical sections that might have been in the interrupt handler are seen by all CPUs as having happened before the increment on line 20 of the `->dynticks` field (which must now have an even-numbered value). Lines 21 and 22 check to see if the interrupt handler posted any “rcu” or “rcu_bh” callbacks, and, if so, line 23 forces this CPU to reschedule, which has the side-effect of forcing it out of dynticks-idle mode, as is required to allow RCU to handle the grace period required by these callbacks.

D.3.7.4 Checking for Dyntick-Idle Mode

The `dyntick_save_progress_counter()` and `rcu_implicit_dynticks_qs()` functions are used to check whether a CPU is in dynticks-idle mode. The `dyntick_save_progress_counter()` function is invoked first, and returns non-zero if the CPU was not in dynticks-idle mode, for example, because it is currently handling an interrupt or NMI, then the `rcu_implicit_dynticks_qs()` function is called some jiffies later. This function looks at the current state in conjunction with state stored away by the earlier call to `dyntick_save_progress_counter()`, again returning non-zero if the CPU either is in dynticks-idle mode or was in dynticks-idle mode during the intervening time. The `rcu_implicit_dynticks_qs()` function may be invoked repeatedly, if need be, until is returns true.

Figure D.47 shows the code for `dyntick_save_progress_counter()`

```c
1 static int
2 dyntick_save_progress_counter(struct rcu_data *rdp)
3 {
4   int ret;
5   int snap;
6   int snap_nmi;
7   
8   snap = rdp->dynticks->dynticks;
9   snap_nmi = rdp->dynticks->dynticks_nmi;
10  smp_mb();
11  rdp->dynticks_snap = snap;
12  rdp->dynticks_nmi_snap = snap_nmi;
13  ret = ((snap & 0x1) == 0) && ((snap_nmi & 0x1) == 0);
14  if (ret)
15    rdp->dynticks_fqs++;
16  return ret;
17 }
```

Figure D.48 shows the code for `rcu_implicit_dynticks_qs()`. Lines 9-12 pick up both new values for the CPU’s `rcu_dynticks` structure’s `->dynticks` and `->dynticks_nmi` fields, as well as the snapshots taken by the last call to `dyntick_save_progress_counter()`. Line 13 then executes a memory barrier to ensure that the values are seen by other CPUs to be gathered prior to subsequent RCU processing. As with `dyntick_save_progress_counter()`, this memory barrier pairs with those in `rcu_enter_nohz()`, `rcu_exit_nohz()`, `rcu_nmi_enter()`, `rcu_nmi_exit()`, `rcu_irq_enter()`, and `rcu_irq_exit()`. Lines 14-15 then check to make sure that this CPU is either currently in dynticks-idle mode ((curr & 0x1) == 0 and (curr_nmi & 0x1) == 0) or has passed through

Quick Quiz D.52: Why isn’t there a memory barrier between lines 8 and 9 of Figure D.47? Couldn’t this cause the code to fetch even-numbered values from both the `->dynticks` and `->dynticks_nmi` fields, even though these two fields never were zero at the same time?
D.3. Hierarchical RCU Code Walkthrough

Figure D.48: Code for rcu_implicit_dynticks_qs()

```
1 static int
2 rcu_implicit_dynticks_qs(struct rcu_data *rdp)
3 {
4     long curr;
5     long curr_nmi;
6     long snap;
7     long snap_nmi;
8     curr = rdp->dynticks->dynticks;
9     snap = rdp->dynticks_snap;
10    curr_nmi = rdp->dynticks->dynticks_nmi;
11    snap_nmi = rdp->dynticks_nmi_snap;
12    smp_mb();
13    if ((curr != snap || (curr & 0x1) == 0) &&
14         (curr_nmi != snap_nmi || (curr_nmi & 0x1) == 0)) {
15        rdp->dynticks_fqs++;
16        return 1;
17    }
18    return rcu_implicit_offline_qs(rdp);
19 }
```

Figure D.49: Recording and Recalling Dynticks-Idle Grace Period

```
1 static void
2 dyntick_record_completed(struct rcu_state *rsp,
3     long comp)
4 {
5     rsp->dynticks_completed = comp;
6 }
7 static long
8 dyntick_recall_completed(struct rcu_state *rsp)
9 {
10    return rsp->dynticks_completed;
11 }
12 }
```

D.3.8 Forcing Quiescent States

Normally, CPUs pass through quiescent states which are duly recorded, so that grace periods end in a timely manner. However, any of the following three conditions can prevent CPUs from passing through quiescent states:

1. The CPU is in dynticks-idle state, and is sleeping in a low-power mode. Although such a CPU is officially in an extended quiescent state, because it is not executing instructions, it cannot do anything on its own.

2. The CPU is in the process of coming online, and RCU has been informed that it is online, but this CPU is not yet actually executing code, nor is it marked as online in cpu_online_map. The current grace period will therefore wait on it, but it cannot yet pass through quiescent states on its own.

3. The CPU is running user-level code, but has avoided entering the scheduler for an extended time period.

D.3.8.1 Recording and Recalling Dynticks-Idle Grace Period

In each of these cases, RCU needs to take action on behalf of the non-responding CPU. The following sections describe the functions that take such action. Section D.3.8.1 describes the functions that record and recall the dynticks-idle grace-period number (in order to avoid incorrectly applying a dynticks-idle quiescent state to the wrong grace period), Section D.3.8.2 describes functions that detect offline and holdout CPUs, Section D.3.8.3 covers rcu_process_dyntick(), which scans for holdout CPUs, and Section D.3.8.4 describes force_quiescent_state(), which drives the process of detecting extended quiescent states and forcing quiescent states on holdout CPUs.

D.3.8.2 Detecting Offline and Holdout CPUs

Figure D.49 shows the code for dyntick_record_completed() and dyntick_recall_completed(). These functions are defined as shown only if dynticks is enabled (in other words, the CONFIG_NO_HZ kernel parameter is selected), otherwise they are essentially no-ops. The purpose of these functions is to ensure that a given observation of a CPU in dynticks-idle mode is associated with the correct grace period in face of races between reporting this CPU in dynticks-idle mode and this CPU coming out of dynticks-idle mode and reporting a quiescent state on its own.

Lines 1-6 show dyntick_record_completed(), which stores the value specified by its comp argument into the specified rcu_state structure’s ->dynticks_completed field. Lines 8-12 show dyntick_recall_completed(), which returns the value stored by the most recent call to dyntick_record_completed() for this combination of CPU and rcu_state structure.
APPENDIX D. READ-COPY UPDATE IMPLEMENTATIONS

D.3.8.2 Handling Offline and Holdout CPUs

Figure D.50 shows the code for `rcu_implicit_offline_qs()`, which checks for offline CPUs and forcing online holdout CPUs to enter a quiescent state.

Line 3 checks to see if the specified CPU is offline, and, if so, line 4 increments statistical counter `->offline_fqs` (which is used only for tracing), and line 5 returns non-zero to indicate that the CPU is in an extended quiescent state.

Otherwise, the CPU is online, not in dynticks-idle mode (or this function would not have been called in the first place), and has not yet passed through a quiescent state for this grace period. Line 7 checks to see if the holdout CPU is the current running CPU, and, if not, line 8 sends the holdout CPU a reschedule IPI. Otherwise, line 10 sets the `TIF_NEED_RESCHED` flag for the current task, forcing the current CPU into the scheduler. In either case, the CPU should then quickly enter a quiescent state. Line 11 increments statistical counter `resched_ipi`, which is again used only for tracing. Finally, line 12 returns zero to indicate that the holdout CPU is still refusing to pass through a quiescent state.

D.3.8.3 Scanning for Holdout CPUs

Figure D.51 shows the code for `rcu_process_dyntick()`, which scans the leaf `rcu_node` structures in search of holdout CPUs, as illustrated by the blue arrow in Figure D.52. It invokes the function passed in through argument `f` on each such CPU’s `rcu_data` structure, and returns non-zero if the grace period specified by the `lastcomp` argument has ended.

Lines 13 and 14 acquire references to the first and the last leaf `rcu_node` structures, respectively. Each pass through the loop spanning lines 15-38 processes one of the leaf `rcu_node` structures.

```c
1 static int rcu_implicit_offline_qs(struct rcu_data *rdp) 2 { 3 if (cpu_is_offline(rdp->cpu)) { 4 rdp->offline_fqs++; 5 return 1; 6 } 7 if (rdp->cpu != smp_processor_id()) 8 smp_send_reschedule(rdp->cpu); 9 else 10 set_need_resched(); 11 rdp->resched_ipi++; 12 return 0; 13 }
```

Figure D.50: Handling Offline and Holdout CPUs

```c
1 static int rcu_process_dyntick(struct rcu_state *rsp, 2 long lastcomp, 3 int (*f)(struct rcu_data *)) 4 { 5 unsigned long bit; 6 int cpu; 7 unsigned long flags; 8 unsigned long mask; 9 struct rcu_node *rnp_cur; 10 struct rcu_node *rnp_end; 11 12 rnp_cur = rsp->level[NUM_RCU_LVLS - 1]; 13 rnp_end = &rsp->node[NUM_RCU_NODES]; 14 for (; rnp_cur < rnp_end; rnp_cur++) { 15 mask = 0; 16 spin_lock_irqsave(&rnp_cur->lock, flags); 17 if (rsp->completed != lastcomp) { 18 spin_unlock_irqrestore(&rnp_cur->lock, flags); 19 return 1; 20 } 21 if (rnp_cur->qsmask == 0) { 22 spin_unlock_irqrestore(&rnp_cur->lock, flags); 23 continue; 24 } 25 cpu = rnp_cur->grplo; 26 bit = 1; 27 for (; cpu <= rnp_cur->grphi; cpu++, bit <<= 1) { 28 if ((rnp_cur->qsmask & bit) != 0 && 29 f(rsp->rda[cpu]) 30 mask |= bit; 31 } 32 if (mask != 0 && rsp->completed == lastcomp) { 33 cpu_quiet_msk(mask, rsp, rnp_cur, flags); 34 continue; 35 } 36 spin_unlock_irqrestore(&rnp_cur->lock, flags); 37 return 0; 38 } 39 return 0; 40 }
```

Figure D.51: Scanning for Holdout CPUs
D.3. HIERARCHICAL RCU CODE WALKTHROUGH

Line 16 sets the local variable `mask` to zero. This variable will be used to accumulate the CPUs within the current leaf `rcu_node` structure that are in extended quiescent states, and can thus be reported as such. Line 17 acquires the current leaf `rcu_node` structure’s lock, and line 18 checks to see if the current grace period has completed, and, if so, line 19 releases the lock and line 20 returns non-zero. Otherwise, line 22 checks for holdout CPUs associated with this `rcu_node` structure, and, if there are none, line 23 releases the lock and line 24 restarts the loop from the beginning on the next lead `rcu_node` structure.

Execution reaches line 26 if there is at least one holdout CPU associated with this `rcu_node` structure. Lines 26 and 27 set local variables `cpu` and `bit` to reference the lowest-numbered CPU associated with this `rcu_node` structure. Each pass through the loop spanning lines 28-32 checks one of the CPUs associated with the current `rcu_node` structure. Line 29 checks to see if the this CPU is still holding out or if it has already passed through a quiescent state. If it is still a holdout, line 30 invokes the specified function (either `dyntick_save_progress_counter()` or `rcu_implicit_dynticks_qs()`, as specified by the caller), and if that function returns non-zero (indicating that the current CPU is in an extended quiescent state), then line 31 sets the current CPU’s bit in `mask`.

Line 33 then checks to see if any CPUs were identified as being in extended quiescent states and if the current grace period is still in force, and, if so, line 34 invokes `cpu_quiet_msk()` to report that the grace period need no longer wait for those CPUs and then line 35 restarts the loop with the next `rcu_node` structure. (Note that `cpu_quiet_msk()` releases the current `rcu_node` structure’s lock, and might well end the current grace period.) Otherwise, if all holdout CPUs really are still holding out, line 37 releases the current `rcu_node` structure’s lock.

Once all of the leaf `rcu_node` structures have been processed, the loop exits, and line 39 returns zero to indicate that the current grace period is still in full force. (Recall that line 20 returns non-zero should the current grace period come to an end.)

D.3.8.4 Code for `force_quiescent_state()`

Figure D.53 shows the code for `force_quiescent_state()` for `CONFIG_SMP`, which is invoked when

\[\text{For non-CONFIG_SMP, force_quiescent_state is a simple wrapper around set_need_resched().}\]
APPENDIX D. READ-COPY UPDATE IMPLEMENTATIONS

```c
1 static void
2 force_quiescent_state(struct rcu_state *rsp, int relaxed)
3 {
4     unsigned long flags;
5     long lastcomp;
6     struct rcu_data *rdp = rsp->rda[smp_processor_id()];
7     struct rcu_node *rnp = rcu_get_root(rsp);
8     u8 signaled;
9
10    if (ACCESS_ONCE(rsp->completed) ==
11       ACCESS_ONCE(rsp->gpnum))
12       return;
13    if (!spin_trylock_irqsave(&rsp->fqslock, flags)) {
14       rsp->n_force_qs_lh++;
15       return;
16    }
17    if (relaxed &&
18       (long)(rsp->jiffies_force_qs - jiffies) >= 0 &&
19       (rdp->n_rcu_pending_force_qs -
20       rdp->n_rcu_pending) >= 0)
21       goto unlock_ret;
22    rsp->n_force_qs++;
23    spin_lock(&rnp->lock);
24    lastcomp = rsp->completed;
25    signaled = rsp->signaled;
26    rsp->jiffies_force_qs =
27       jiffies + RCU_JIFFIES_TILL_FORCE_QS;
28    rdp->n_rcu_pending_force_qs =
29       rdp->n_rcu_pending +
30       RCU_JIFFIES_TILL_FORCE_QS;
31    if (lastcomp == rsp->completed) {
32       rsp->n_force_qs_ngp++;
33       spin_unlock(&rnp->lock);
34       goto unlock_ret;
35    }
36    spin_unlock(&rnp->lock);
37    switch (signaled) {
38       case RCU_GP_INIT:
39          break;
40       case RCU_SAVE_DYNTICK:
41       if (RCU_SIGNAL_INIT != RCU_SAVE_DYNTICK)
42          break;
43       if (rcu_process_dyntick(rsp, lastcomp,
44          dyntick_save_progress_counter))
45          goto unlock_ret;
46       spin_lock(&rnp->lock);
47       if (lastcomp == rsp->completed) {
48          rdp->n_force_qs_ngp++;
49          dyntick_record_completed(rsp, lastcomp);
50       }
51       spin_unlock(&rnp->lock);
52       break;
53       case RCU_FORCE_QS:
54       if (rcu_process_dyntick(rsp,
55          dyntick_recall_completed(rsp),
56          rcu_implicit_dynticks_qs))
57          goto unlock_ret;
58       break;
59    }
60    unlock_ret:
61    spin_unlock_irqrestore(&rsp->fqslock, flags);
62 }
```

Figure D.53: force_quiescent_state() Code

RCU feels the need to expedite the current grace period by forcing CPUs through quiescent states. RCU feels this need when either:

1. the current grace period has gone on for more than three jiffies (or as specified by the compile-time value of RCU_JIFFIES_TILL_FORCE_QS), or

2. a CPU enqueuing an RCU callback via either call_rcu() or call_rcu_bh() sees more than 10,000 callbacks enqueued (or as specified by the boot-time parameter qhimark).

Lines 10-12 check to see if there is a grace period in progress, silently exiting if not. Lines 13-16 attempt to acquire ->fqslock, which prevents concurrent attempts to expedite a grace period. The ->n_force_qs_lh counter is incremented when this lock is already held, and is visible via the fqlh= field in the rcuhier debugfs file when the CONFIG_RCU_TRACE kernel parameter is enabled. Lines 17-21 check to see if it is really necessary to expedite the current grace period, in other words, if (1) the current CPU has 10,000 RCU callbacks waiting, or (2) at least three jiffies have passed since either the beginning of the current grace period or since the last attempt to expedite the current grace period, measured either by the jiffies counter or by the number of calls to rcu_pending. Line 22 then counts the number of attempts to expedite grace periods.

Lines 23-36 are executed with the root rcu_node structure’s lock held in order to prevent confusion should the current grace period happen to end just as we try to expedite it. Lines 24 and 25 snapshot the ->completed and ->signaled fields, lines 26-30 set the soonest time that a subsequent non-relaxed force_quiescent_state() will be allowed to actually do any expediting, and lines 31-35 check to see if the grace period ended while we were acquiring the rcu_node structure’s lock, releasing this lock and returning if so.

Lines 37-59 drive the force_quiescent_state() state machine. If the grace period is still in the midst of initialization, lines 41 and 42 simply return, allowing force_quiescent_state() to be called again at a later time, presumably after initialization has completed. If dynticks are enabled (via the CONFIG_NO_HZ kernel parameter), the first post-initialization call to force_quiescent_state() in a given grace period will execute lines 40-52, and the second and subsequent calls will execute lines 53-59. On the other hand, if dynticks is not enabled, then all post-initialization
D.3. HIERARCHICAL RCU CODE WALKTHROUGH

D.3.9 CPU-Stall Detection

RCU checks for stalled CPUs when the CONFIG_RCU_CPU_STALL_DETECTOR kernel parameter is selected. “Stalled CPUs” are those spinning in the kernel with preemption disabled, which degrades response time. These checks are implemented via the record_gp_stall_check_time(), check_cpu_stall(), print_cpu_stall(), and print_other_cpu_stall() functions, each of which is described below. All of these functions are no-ops when the CONFIG_RCU_CPU_STALL_DETECTOR kernel parameter is not selected.

Figure D.54 shows the code for record_gp_stall_check_time(). Line 4 records the current time (of the start of the grace period) in jiffies, and lines 5-6 record the time at which CPU stalls should be checked for, should the grace period run on that long.

Figure D.55 shows the code for check_cpu_stall(), which checks to see if the grace period has stretched on too long, invoking either print_cpu_stall() or print_other_cpu_stall() in order to print a CPU-stall warning message if so.

Line 8 computes the number of jiffies since the time at which stall warnings should be printed, which will be negative if it is not yet time to print warnings. Line 9 obtains a pointer to the leaf rcu_node structure corresponding to the current CPU, and line 10 checks to see if the CPU has not yet passed through a quiescent state and if the grace period has extended too long (in other words, if the current CPU is stalled), with line 11 invoking print_cpu_stall() if so.

Otherwise, lines 12-13 check to see if the grace period is still in effect and if it has extended a couple of jiffies.
APPENDIX D. READ-COPY UPDATE IMPLEMENTATIONS

1 static void print_cpu_stall(struct rcu_state *rsp)
2 {
3     unsigned long flags;
4     struct rcu_node *rnp = rcu_get_root(rsp);
5     printk(KERN_ERR
6          "INFO: RCU detected CPU %d stall 
7          *(t=klj jiffies)n",*
8          smp_processor_id(),
9          jiffies - rsp->gp_start);
10     dump_stack();
11     spin_lock_irqsave(&rnp->lock, flags);
12     if ((long)(jiffies - rsp->jiffies_stall) >= 0)
13         rsp->jiffies_stall =
14         jiffies + RCU_SECONDS_TILL_STALL_RECHECK;
15     spin_unlock_irqrestore(&rnp->lock, flags);
16     set_need_resched();
17 }

Figure D.56: print_cpu_stall() Code

past the CPU-stall warning duration, with line 14 invoking
print_other_cpu_stall() if so.

Quick Quiz D.53: Why wait the extra couple jiffies on
lines 12-13 in Figure D.55?

Figure D.56 shows the code for print_cpu_stall().

Line 6-11 prints a console message and dumps the
current CPU’s stack, while lines 12-17 compute the time
to the next CPU stall warning, should the grace period
stretch on that much additional time.

Quick Quiz D.54: What prevents the grace period
from ending before the stall warning is printed in Fig-
ure D.56?

Figure D.57 shows the code for print_other_cpu_stall(),
which prints out stall warnings for
CPUs other than the currently running CPU.

Lines 10 and 11 pick up references to the first leaf
rcu_node structure and one past the last leaf rcu-
node structure, respectively. Line 12 acquires the root
rcu_node structure’s lock, and also disables interrupts.
Line 13 calculates the how long ago the CPU-stall warn-
ing time occurred (which will be negative if it has not
yet occurred), and lines 14 and 15 check to see if the
CPU-stall warning time has passed and if the grace period
has not yet ended, with line 16 releasing the lock (and
re-enabling interrupts) and line 17 returning if so.

Quick Quiz D.55: Why does print_other_cpu_stall() in Figure D.57 need to check for the grace
period ending when print_cpu_stall() did not?

Otherwise, lines 19 and 20 compute the next time
that CPU stall warnings should be printed (if the grace
period extends that long) and line 21 releases the lock
and re-enables interrupts. Lines 23-33 print a list of

1 static void print_other_cpu_stall(struct rcu_state *rsp)
2 {
3     int cpu;
4     long delta;
5     unsigned long flags;
6     struct rcu_node *rnp = rcu_get_root(rsp);
7     struct rcu_node *rnp_cur;
8     struct rcu_node *rnp_end;
9     rnp_cur = rsp->level[NUM_RCU_LVLS - 1];
10     rnp_end = &rsp->node[NUM_RCU_NODES];
11     spin_lock_irqsave(&rnp->lock, flags);
12     delta = jiffies - rsp->jiffies_stall;
13     if (delta < RCU_STALL_RAT_DELAY ||
14         (long)(jiffies - rsp->gp_start)) {*
15         rsp->jiffies_stall = jiffies +
16         RCU_SECONDS_TILL_STALL_RECHECK;
17         spin_unlock_irqrestore(&rnp->lock, flags);
18         printk(KERN_ERR *INFO: RCU detected CPU stalls:*
19            "[CPU %d] detected by %d, t=%ld jiffies\n",
20            smp_processor_id(),
21            (long)(jiffies - rsp->gp_start));
22         return;
23     } for (; rnp_cur < rnp_end; rnp_cur++) {
24         if ((rnp_cur->qsmask == 0) ||
25             continue;
26             cpu = 0;
27             for (; cpu < rnp_cur->grp * (rnp_cur->grp + 1);
28                 if (cpu++)
29                     printk("[CPU %d, t=%ld jiffies]\n",
30                         smp_processor_id(),
31                         (long)(jiffies - rsp->gp_start));
32                 force_quiescent_state(rsp, 0);
33     }*

Figure D.57: print_other_cpu_stall() Code
the stalled CPUs, and, finally, line 34 invokes `force_quiescent_state()` in order to nudge the offending CPUs into passing through a quiescent state.

### D.3.10 Possible Flaws and Changes

The biggest possible issue with Hierarchical RCU put forward as of this writing is the fact that `force_quiescent_state()` involves a potential walk through all CPUs' `rcu_data` structures. On a machine with thousands of CPUs, this could potentially represent an excessive impact on scheduling latency, given that this scan is conducted with interrupts disabled.

Should this become a problem in real life, one fix is to maintain separate `force_quiescent_state()` sequencing on a per-leaf-`rcu_node` basis as well as the current `per-rcu_state -->signaled` state variable. This would allow incremental forcing of quiescent states on a per-leaf-`rcu_node` basis, greatly reducing the worst-case degradation of scheduling latency.

In the meantime, those caring deeply about scheduling latency can limit the number of CPUs in the system or use the preemptible RCU implementation.

### D.4 Preemptible RCU

The preemptible RCU implementation is unusual in that it permits read-side critical sections to be preempted and to be blocked waiting for locks. However, it does not handle general blocking (for example, via the `wait_event()` primitive): if you need that, you should instead use SRCU, which is described in Appendix D.1. In contrast to SRCU, preemptible RCU only permits blocking within primitives that are both subject to priority inheritance and non-blocking in a non-`CONFIG_PREEMPT` kernel. This ability to acquire blocking locks and to be preempted within RCU read-side critical sections is required for the aggressive real-time capabilities provided by Ingo Molnar’s `-rt` patchset. However, the initial preemptible RCU implementation [McK05c] had some limitations, including:

1. Its read-side primitives cannot be called from within non-maskable interrupt (NMI) or system-management interrupt handlers.
2. Its read-side primitives use both atomic instructions and memory barriers, both of which have excessive overhead.
3. It does no priority boosting of RCU read-side critical sections [McK07d].

The new preemptible RCU implementation that accepted into the 2.6.26 Linux kernel removes these limitations, and this appendix describes its design, serving as an update to the LWN article [McK07a]. However, please note that this implementation was replaced with a faster and simpler implementation in the 2.6.32 Linux kernel. This description nevertheless remains to bear witness to the most complex RCU implementation ever devised.

**Quick Quiz D.56:** Why is it important that blocking primitives called from within a preemptible-RCU read-side critical section be subject to priority inheritance? ■

**Quick Quiz D.57:** Could the prohibition against using primitives that would block in a non-`CONFIG_PREEMPT` kernel be lifted, and if so, under what conditions? ■

### D.4.1 Conceptual RCU

Understanding and validating an RCU implementation is much easier given a view of RCU at the lowest possible level. This section gives a very brief overview of the most basic concurrency requirements that an RCU implementation must support. For more detail, please see Section 8.3.2.

RCU implementations must obey the following rule: if any statement in a given RCU read-side critical section precedes a grace period, then all statements in that RCU read-side critical section must complete before that grace period ends.

![Picture](image.png)

**Figure D.58:** Buggy Grace Period From Broken RCU

This is illustrated by Figure D.58, where time advances from left to right. The red "Removal" box represents...
the update-side critical section that modifies the RCU-
protected data structure, for example, via list_del_rcu(); the large yellow "Grace Period" box represents a grace period (surprise!) which might be invoked via synchronize_rcu(), and the green "Reclamation" box represents freeing the affected data element, perhaps via kfree(). The blue "Reader" boxes each represent an RCU read-side critical section, for example, beginning with rcu_read_lock() and ending with rcu_read_unlock(). The red-rimmed "Reader" box is an example of an illegal situation: any so-called RCU implementation that permits a read-side critical section to completely overlap a grace period is buggy, since the updater might free up memory that this reader is still using.

So, what is the poor RCU implementation to do in this situation?

It must extend the grace period, perhaps as shown in Figure D.59. In short, the RCU implementation must ensure that any RCU read-side critical sections in progress at the start of a given grace period have completely finished, memory operations and all, before that grace period is permitted to complete. This fact allows RCU validation to be extremely focused: simply demonstrate that any RCU read-side critical section in progress at the beginning of a grace period must terminate before that grace period ends, along with sufficient barriers to prevent either the compiler or the CPU from undoing the RCU implementation’s work.

D.4.2 Overview of Preemptible RCU Algorithm

This section focuses on a specific implementation of preemptible RCU. Many other implementations are possible, and are described elsewhere [MSMB06, MS05]. This article focuses on this specific implementation’s general approach, the data structures, the grace-period state machine, and a walk through the read-side primitives.

D.4.2.1 General Approach

![Figure D.60: Classic vs. Preemptible RCU Callback Processing](attachment:0)

Because this implementation of preemptible RCU does not require memory barriers in rcu_read_lock() and rcu_read_unlock(), a multi-stage grace-period detection algorithm is required. Instead of using a single wait queue of callbacks (which has sufficed for earlier RCU implementations), this implementation uses an array of wait queues, so that RCU callbacks are enqueued on each element of this array in turn. This difference in callback flow is shown in Figure D.60 for a preemptible RCU implementation with two waitlist stages per grace period (in contrast, the September 10 2007 patch to -rt [McK07c] uses four waitlist stages).
Given two stages per grace period, any pair of stages forms a full grace period. Similarly, in an implementation with four stages per grace period, any sequence of four stages would form a full grace period.

![Figure D.61: Preemptible RCU Counter Flip Operation](image)

To determine when a grace-period stage can end, preemptible RCU uses a per-CPU two-element `rcu_flipctr` array that tracks in-progress RCU read-side critical sections. One element of a given CPU’s `rcu_flipctr` array tracks old RCU read-side critical sections, in other words, critical sections that started before the current grace-period stage. The other element tracks new RCU read-side critical sections, namely those starting during the current grace-period stage. The array elements switch roles at the beginning of each new grace-period stage, as shown in Figure D.61.

During the first stage on the left-hand side of the above figure, `rcu_flipctr[0]` tracks the new RCU read-side critical sections, and is therefore incremented by `rcu_read_lock()` and decremented by `rcu_read_unlock()`. Similarly, `rcu_flipctr[1]` tracks the old RCU read-side critical sections (those that started during earlier stages), and is therefore decremented by `rcu_read_unlock()` and never incremented at all.

Because each CPU’s old `rcu_flipctr[1]` elements are never incremented, their sum across all CPUs must eventually go to zero, although preemption in the midst of an RCU read-side critical section might cause any individual counter to remain non-zero or even to go negative. For example, suppose that a task calls `rcu_read_lock()` on one CPU, is preempted, resumes on another CPU, and then calls `rcu_read_unlock()`. The first CPU’s counter will then be +1 and the second CPU’s counter will be -1, however, they will still sum to zero. Regardless of possible preemption, when the sum of the old counter elements goes to zero, it is safe to move to the next grace-period stage, as shown on the right-hand side of the above figure.

In this second stage, the elements of each CPU’s `rcu_flipctr` counter array switch roles. The `rcu_flipctr[0]` counter now tracks the old RCU read-side critical sections, in other words, the ones that started during grace period stage 0. Similarly, the `rcu_flipctr[1]` counter now tracks the new RCU read-side critical sections that start in grace period stage 1. Therefore, `rcu_read_lock()` now increments `rcu_flipctr[1]`, while `rcu_read_unlock()` still might decrement either counter. Specifically, if the matching `rcu_read_lock()` executed during grace-period stage 0 (the old stage at this point), then `rcu_read_unlock()` must decrement `rcu_flipctr[0]`, but if the matching `rcu_read_lock()` executed during grace-period stage 1 (the new stage), then `rcu_read_unlock()` must instead decrement `rcu_flipctr[1]`.

The critical point is that all `rcu_flipctr` elements tracking the old RCU read-side critical sections must strictly decrease. Therefore, once the sum of these old counters reaches zero, it cannot change.

The `rcu_read_lock()` primitive uses the bottom bit of the current grace-period counter (`rcu_ctrlblk.completed & 0x1`) to index the `rcu_flipctr` array, and records this index in the task structure. The matching `rcu_read_unlock()` uses this recorded value to ensure that it decrements a counter corresponding to the one that the matching `rcu_read_lock()` incremented. Of course, if the RCU read-side critical section has been preempted, `rcu_read_lock()` might be decrementing the counter belonging to a different CPU than the one whose counter was incremented by the matching `rcu_read_lock()`.

Each CPU also maintains `rcu_flip_flag` and `rcu_mb_flag` per-CPU variables. The `rcu_flip_flag` variable is used to synchronize the start of each grace-period stage: once a given CPU has responded to its `rcu_flip_flag`, it must refrain from incrementing the `rcu_flip` array element that now corresponds to the old grace-period stage. The CPU that advances the counter (`rcu_ctrlblk.completed`) changes the value of each CPU’s `rcu_mb_flag` to `rcu_flipped`, but a given `rcu_mb_flag` may be changed back to
The `rcu_mb_flag` variable is used to force each CPU to execute a memory barrier at the end of each grace-period stage. These memory barriers are required to ensure that memory accesses from RCU read-side critical sections ending in a given grace-period stage are ordered before the end of that stage. This approach gains the benefits memory barriers at the beginning and end of each RCU read-side critical section without having to actually execute all those costly barriers. The `rcu_mb_flag` is set to `rcu_mb_needed` by the CPU that detects that the sum of the old counters is zero, but a given `rcu_mb_flag` is changed back to `rcu_mb_done` only by the corresponding CPU, and even then only after executing a memory barrier.

D.4.2.2 Data Structures

This section describes preemptible RCU’s major data structures, including `rcu_ctrlblk`, `rcu_data`, `rcu_flipctr`, `rcu_try_flip_state`, `rcu_try_flip_flag`, and `rcu_mb_flag`.

`rcu_ctrlblk` The `rcu_ctrlblk` structure is global, and holds the lock that protects grace-period processing (`fliplock`) as well as holding the global grace-period counter (`completed`). The least-significant bit of `completed` is used by `rcu_read_lock()` to select which set of counters to increment.

`rcu_data` The `rcu_data` structure is a per-CPU structure, and contains the following fields:

- `lock` guards the remaining fields in this structure.
- `completed` is used to synchronize CPU-local activity with the global counter in `rcu_ctrlblk`.
- `waitlistcount` is used to maintain a count of the number of non-empty wait-lists. This field is used by `rcu_pending()` to help determine if this CPU has any RCU-related work left to be done.
- `nextlist`, `nexttail`, `waitlist`, `waittail`, `donelist`, and `donetail` form lists containing RCU callbacks that are waiting for invocation at the end of a grace period. Each list has a tail pointer, allowing $O(1)$ appends. The RCU callbacks flow through these lists as shown below.
- `rcupreempt_trace` accumulates statistics.

```
call_rcu()

nextlist nexttail
waitlist[0] waittail[0]
donelist donetail
call_rcu()
rcu_process_callbacks()
```

Figure D.62: Preemptible RCU Callback Flow

Figure D.62 shows how RCU callbacks flow through a given `rcu_data` structure’s lists, from creation by `call_rcu()` through invocation by `rcu_process_callbacks()`. Each blue arrow represents one pass by the grace-period state machine, which is described in a later section.

`rcu_flipctr` As noted earlier, the `rcu_flipctr` per-CPU array of counters contains the counter pairs that track outstanding RCU read-side critical sections. Any given counter in this array can go negative, for example, when a task is migrated to a different CPU in the middle of an RCU read-side critical section. However, the sum of the counters will still remain positive throughout the corresponding grace period, and will furthermore go to zero at the end of that grace period.

`rcu_try_flip_state` The `rcu_try_flip_state` variable tracks the current state of the grace-period state machine, as described in the next section.

`rcu_try_flip_flag` The `rcu_try_flip_flag` per-CPU variable alerts the corresponding CPU that the grace-period counter has recently been incremented, and also records that CPU’s acknowledgment.
Once a given CPU has acknowledged the counter flip, all subsequent actions taken by `rcu_read_lock()` on that CPU must account for the new value of the grace-period counter, in particular, when incrementing `rcu_flipctr` in `rcu_read_lock()`.

`rcu_mb_flag` The `rcu_mb_flag` per-CPU variable alerts the corresponding CPU that it must execute a memory barrier in order for the grace-period state machine to proceed, and also records that CPU’s acknowledgment. Once a given CPU has executed its memory barrier, the memory operations of all prior RCU read-side critical will be visible to any code sequenced after the corresponding grace period.

### D.4.2.3 Grace-Period State Machine

This section gives an overview of the states executed by the grace-period state machine, and then walks through the relevant code.

**Grace-Period State Machine Overview** The state (recorded in `rcu_try_flip_state`) can take on the following values:

- **rcu_try_flip_idle_state**: the grace-period state machine is idle due to there being no RCU grace-period activity. The `rcu_ctrlblk.completed` grace-period counter is incremented upon exit from this state, and all of the per-CPU `rcu_flip_flag` variables are set to `rcu_flipped`.

- **rcu_try_flip_waitack_state**: waiting for all CPUs to acknowledge that they have seen the previous state’s increment, which they do by setting their `rcu_flip_flag` variables to `rcu_flip_seen`. Once all CPUs have so acknowledged, we know that the old set of counters can no longer be incremented.

- **rcu_try_flip_waitzero_state**: waiting for the old counters to sum to zero. Once the counters sum to zero, all of the per-CPU `rcu_mb_flag` variables are set to `rcu_mb_needed`.

- **rcu_try_flip_waitmb_state**: waiting for all CPUs to execute a memory-barrier instruction, which they signify by setting their `rcu_mb_flag` variables to `rcu_mb_done`. Once all CPUs have done so, all CPUs are guaranteed to see the changes made by any RCU read-side critical section that started before the beginning of the corresponding grace period, even on weakly ordered machines.

![Preemptible RCU State Machine](image)

**Figure D.63: Preemptible RCU State Machine**

The grace period state machine cycles through these states sequentially, as shown in Figure D.63.

Figure D.64 shows how the state machine operates over time. The states are shown along the figure’s left-hand side and the relevant events are shown along the timeline, with time proceeding in the downward direction. We will elaborate on this figure when we validate the algorithm in a later section.

In the meantime, here are some important things to note:

1. The increment of the `rcu_ctrlblk.completed` counter might be observed at different times by different CPUs, as indicated by the blue oval. However, after a given CPU has acknowledged the increment, it is required to use the new counter. Therefore, once all CPUs have acknowledged, the old counter can only be decremented.

2. A given CPU advances its callback lists just before acknowledging the counter increment.
APPENDIX D. READ-COPY UPDATE IMPLEMENTATIONS

3. The blue oval represents the fact that memory reordering might cause different CPUs to see the increment at different times. This means that a given CPU might believe that some other CPU has jumped the gun, using the new value of the counter before the counter was actually incremented. In fact, in theory, a given CPU might see the next increment of the rcu_ctrlblk.completed counter as early as the last preceding memory barrier. (Note well that this sentence is very imprecise. If you intend to do correctness proofs involving memory barriers, please see Appendix D.4.3.3.

4. Because rcu_read_lock() does not contain any memory barriers, the corresponding RCU read-side critical sections might be reordered by the CPU to follow the rcu_read_unlock(). Therefore, the memory barriers are required to ensure that the actions of the RCU read-side critical sections have in fact completed.

5. As we will see, the fact that different CPUs can see the counter flip happening at different times means that a single trip through the state machine is not sufficient for a grace period: multiple trips are required.

Grace-Period State Machine Walkthrough  This section walks through the C code that implements the RCU grace-period state machine, which is invoked from the scheduling-clock interrupt, which invokes rcu_check_callbacks() with irqs (and thus also preemption) disabled. This function is implemented as shown in Figure D.65. Line 4 selects the rcu_data structure corresponding to the current CPU, and line 6 checks to see if this CPU needs to execute a memory barrier to advance the state machine out of the rcu_try_flip_waitmb_state state. Line 7 checks to see if this CPU is already aware of the current grace-period stage number, and line 8 attempts to advance the state machine if so. Lines 9 and 12 hold the rcu_data’s lock, and line 11 advances callbacks if appropriate. Line 10 updates RCU tracing statistics, if enabled via CONFIG_RCU_TRACE.

The rcu_check_mb() function executes a memory barrier as needed as shown in Figure D.66. Line 3 checks to see if this CPU needs to execute a memory barrier, and, if so, line 4 executes one and line 5 informs the state machine. Note that this memory barrier ensures that any CPU that sees the new value of rcu_mb_flag will also see the memory operations executed by this CPU in any prior RCU read-side critical section.

The rcu_try_flip() function implements the top level of the RCU grace-period state machine, as shown in Figure D.67. Line 6 attempts to acquire the global
D.4. PREEMPTIBLE RCU

1 static void rcu_try_flip(void)
2 {
3     unsigned long flags;
4     
5     RCU_TRACE_ME(rcupreempt_trace_try_flip entertaining);  // Replace with actual function name
6     if (!spin_trylock_irqsave(&rcu_ctrlblk.fliplock, flags)) {
7         RCU_TRACE_ME(rcupreempt_trace_try_flip_e);  // Replace with actual function name
8         return;
9     }
10    
11    switch (rcu_try_flip_state) {
12        case rcu_try_flip_idle_state:
13            if (rcu_try_flip_idle())
14                rcu_try_flip_state = rcu_try_flip_waitack_state;
15                break;
16        case rcu_try_flip_waitack_state:
17            if (rcu_try_flip_waitack())
18                rcu_try_flip_state = rcu_try_flip_waitzero_state;
19                break;
20        case rcu_try_flip_waitzero_state:
21            if (rcu_try_flip_waitzero())
22                rcu_try_flip_state = rcu_try_flip_waitmb_state;
23                break;
24        case rcu_try_flip_waitmb_state:
25            if (rcu_try_flip_waitmb())
26                rcu_try_flip_state = rcu_try_flip_idle_state;
27    }
28     spin_unlock_irqrestore(&rcu_ctrlblk.fliplock, flags);
29 }

Figure D.67: rcu_try_flip() Implementation

1 static int rcu_try_flip_idle(void)
2 {
3     int cpu;
4     
5     RCU_TRACE_ME(rcupreempt_trace_try_flip_i);  // Replace with actual function name
6     if (!rcu_pending(smp_processor_id())) {
7         RCU_TRACE_ME(rcupreempt_trace_try_flip_i_e);  // Replace with actual function name
8         return 0;
9     }
10    
11    RCU_TRACE_ME(rcupreempt_trace_task_i_g);  // Replace with actual function name
12    rcu_ctrlblk.completed++;  // Replace with actual function name
13    smp_mb();  // Replace with actual function name
14    for_each_cpu_mask(cpu, rcu_cpu_online_map)
15        per_cpu(rcu_flip_flag, cpu) = rcu_flipped;  // Replace with actual function name
16    return 1;
17 }

Figure D.68: rcu_try_flip_idle() Implementation

RCU state-machine lock, and returns if unsuccessful. 
Lines; 5 and 7 accumulate RCU-tracing statistics (again, 
if CONFIG_RCU_TRACE is enabled). Lines 10 through 
26 execute the state machine, each invoking a function 
specific to that state. Each such function returns 1 if the 
state needs to be advanced and 0 otherwise. In prin-
ciple, the next state could be executed immediately, but in 
practice we choose not to do so in order to reduce latency. 
Finally, line 27 releases the global RCU state-machine 
lock that was acquired by line 6.

The rcu_try_flip_idle() function is called

when the RCU grace-period state machine is idle, and 
is thus responsible for getting it started when needed. Its 
_code is shown in Figure D.68. Line 6 checks to see if 
there is any RCU grace-period work pending for this CPU, 
and if not, line 8 leaves, telling the top-level state machine 
to remain in the idle state. If instead there is work to do, 
line 11 increments the grace-period stage counter, line 12 
does a memory barrier to ensure that CPUs see the new 
counter before they see the request to acknowledge it, and 
lines 13 and 14 set all of the online CPUs’ rcu_flip_ 
flag. Finally, line 15 tells the top-level state machine to 
advance to the next state.

The rcu_try_flip_waitack() function, shown 
in Figure D.69, checks to see if all online CPUs have 
acknowledged the counter flip (AKA “increment”, but 
called “flip” because the bottom bit, which rcu_read_ 
lock() uses to index the rcu_flipctr array, does 
flip). If they have, it tells the top-level grace-period state 
machine to move to the next state.

Line 6 cycles through all of the online CPUs, and line 7 
checks to see if the current such CPU has acknowledged 
the last counter flip. If not, line 9 tells the top-level grace-
period state machine to remain in this state. Otherwise, if 
all online CPUs have acknowledged, then line 11 does a 
memory barrier to ensure that we don’t check for zeroes 
before the last CPU acknowledges. This may seem du-
bious, but CPU designers have sometimes done strange 
things. Finally, line 13 tells the top-level grace-period 
state machine to advance to the next state.

The rcu_try_flip_waitzero() function, 
shown in Figure D.70, checks to see if all pre-existing 
RCU read-side critical sections have completed, telling 
the state machine to advance if so. Lines 8 and 9 sum the 
counters, and line 10 checks to see if the result is zero, 
and, if not, line 12 tells the state machine to stay right

Figure D.69: rcu_try_flip_waitack() Implementation
APPENDIX D. READ-COPY UPDATE IMPLEMENTATIONS

1 static int rcu_try_flip_waitzero(void)
2 {
3     int cpu;
4     int lastidx = !(rcu_ctrlblk.completed & 0x1);
5     int sum = 0;
6
7     RCU_TRACE_ME(rcupreempt_trace_try_flip_z1);
8     for_each_possible_cpu(cpu)
9         sum += per_cpu(rcu_flipctr, cpu)[lastidx];
10     if (sum != 0) {
11         RCU_TRACE_ME(rcupreempt_trace_try_flip_ze1);
12         return 0;
13     } else {
14         smp_mb();
15         for_each_cpu_mask(cpu, rcu_cpu_online_map)
16             per_cpu(rcu_mb_flag, cpu) = rcu_mb_needed;
17         RCU_TRACE_ME(rcupreempt_trace_try_flip_z2);
18         return 1;
19     }
20 }

Figure D.70: rcu_try_flip_waitzero() Implementation

1 static int rcu_try_flip_waitmb(void)
2 {
3     int cpu;
4
5     RCU_TRACE_ME(rcupreempt_trace_try_flip_ml);
6     for_each_cpu_mask(cpu, rcu_cpu_online_map)
7     if (per_cpu(rcu_mb_flag, cpu) != rcu_mb_done) {
8         RCU_TRACE_ME(rcupreempt_trace_try_flip_mel);
9         return 0;
10     } else {
11         smp_mb();
12         RCU_TRACE_ME(rcupreempt_trace_try_flip_m2);
13         return 1;
14     }

Figure D.71: rcu_try_flip_waitmb() Implementation

where it is. Otherwise, line 14 executes a memory barrier to ensure that no CPU sees the subsequent call for a memory barrier before it has exited its last RCU read-side critical section. This possibility might seem remote, but again, CPU designers have done stranger things, and besides, this is anything but a fastpath. Lines 15 and 16 set all online CPUs’ rcu_mb_flag variables, and line 18 tells the state machine to advance to the next state.

The rcu_try_flip_waitmb() function, shown in Figure D.71, checks to see if all online CPUs have executed the requested memory barrier, telling the state machine to advance if so. Lines 6 and 7 check each online CPU to see if it has done the needed memory barrier, and if not, line 9 tells the state machine not to advance. Otherwise, if all CPUs have executed a memory barrier, line 11 executes a memory barrier to ensure that any RCU callback invocation follows all of the memory barriers, and line 13 tells the state machine to advance.

1 static void __rcu_advance_callbacks(struct rcu_data *rdp)
2 {
3     int cpu;
4     int i;
5     int wlc = 0;
6
7     if (rdp->completed != rcu_ctrlblk.completed) {
8         if (rdp->waitlist[GP_STAGES - 1] != NULL) {
9             *rdp->donetail = rdp->waitlist[GP_STAGES - 1];
10             rdp->donetail = rdp->waittail[GP_STAGES - 1];
11             RCU_TRACE_RDP(rcupreempt_trace_move2done, rdp);
12         }
13         for (i = GP_STAGES - 2; i >= 0; i--) {
14             if (rdp->waitlist[i] != NULL) {
15                 rdp->waitlist[i + 1] = rdp->waitlist[i];
16                 rdp->waittail[i + 1] = rdp->waittail[i];
17                 wlc++;
18             } else {
19                 rdp->waitlist[i + 1] = NULL;
20                 rdp->waittail[i + 1] =
21                 &rdp->waitlist[i + 1];
22             }
23         }
24         if (rdp->nextlist != NULL) {
25             rdp->waitlist[0] = rdp->nextlist;
26             rdp->waittail[0] = rdp->nexttail;
27             wlc++;
28             rdp->nextlist = NULL;
29             rdp->nexttail = &rdp->nextlist;
30             RCU_TRACE_RDP(rcupreempt_trace_move2wait, rdp);
31         } else {
32             rdp->waitlist[0] = NULL;
33             rdp->waittail[0] = &rdp->waitlist[0];
34         }
35         rdp->waitlistcount = wlc;
36         rdp->completed = rcu_ctrlblk.completed;
37     } else {
38         cpu = raw_smp_processor_id();
39         if (per_cpu(rcu_flip_flag, cpu) == rcu_flipped) {
40             smp_mb();
41             per_cpu(rcu_flip_flag, cpu) = rcu_flip_seen;
42             smp_mb();
43         }
44     }

Figure D.72: __rcu_advance_callbacks() Implementation
D.4. PREEMPTIBLE RCU

1 void __rcu_read_lock(void)
2 {
3   int idx;
4   struct task_struct *t = current;
5   int nesting;
6
7   nesting = ACCESS_ONCE(t->rcu_read_lock_nesting);
8   if (nesting != 0) {
9     t->rcu_read_lock_nesting = nesting + 1;
10   } else {
11     unsigned long flags;
12
13     local_irq_save(flags);
14     idx = ACCESS_ONCE(rcu_ctrlblk.completed) & 0x1;
15     ACCESS_ONCE(__get_cpu_var(rcu_flipctr)[idx])++;
16     ACCESS_ONCE(t->rcu_read_lock_nesting) = nesting + 1;
17     ACCESS_ONCE(t->rcu_flipctr_idx) = idx;
18     local_irq_restore(flags);
19   }
20 }

Figure D.73: __rcu_read_lock() Implementation

The __rcu_advance_callbacks() function, shown in Figure D.72, advances callbacks and acknowledges the counter flip. Line 7 checks to see if the global rcu_ctrlblk.completed counter has advanced since the last call by the current CPU to this function. If not, callbacks need not be advanced (lines 8-37). Otherwise, lines 8 through 37 advance callbacks through the lists (while maintaining a count of the number of non-empty lists in the wlc variable). In either case, lines 38 through 43 acknowledge the counter flip if needed.

Quick Quiz D.58: How is it possible for lines 38-43 of __rcu_advance_callbacks() to be executed when lines 7-37 have not? Won’t they both be executed just after a counter flip, and never at any other time?

D.4.2.4 Read-Side Primitives

This section examines the rcu_read_lock() and rcu_read_unlock() primitives, followed by a discussion of how this implementation deals with the fact that these two primitives do not contain memory barriers.

rcu_read_lock() The implementation of rcu_read_lock() is as shown in Figure D.73. Line 7 fetches this task’s RCU read-side critical-section nesting counter. If line 8 finds that this counter is non-zero, then we are already protected by an outer rcu_read_lock(), in which case line 9 simply increments this counter.

However, if this is the outermost rcu_read_lock(), then more work is required. Lines 13 and 18 suppress and restore irqs to ensure that the intervening code is neither preempted nor interrupted by a scheduling-clock interrupt (which runs the grace period state machine). Line 14 fetches the grace-period counter, line 15 increments the current counter for this CPU, line 16 increments the nesting counter, and line 17 records the old/new counter index so that rcu_read_unlock() can decrement the corresponding counter (but on whatever CPU it ends up running on).

The ACCESS_ONCE() macros force the compiler to emit the accesses in order. Although this does not prevent the CPU from reordering the accesses from the viewpoint of other CPUs, it does ensure that NMI and SMI handlers running on this CPU will see these accesses in order. This is critically important:

1. In absence of the ACCESS_ONCE() in the assignment to idx, the compiler would be within its rights to: (a) eliminate the local variable idx and (b) compile the increment on line 16 as a fetch-increment-store sequence, doing separate accesses to rcu_ctrlblk.completed for the fetch and the store. If the value of rcu_ctrlblk.completed had changed in the meantime, this would corrupt the rcu_flipctr values.

2. If the assignment to rcu_read_lock_nesting (line 17) were to be reordered to precede the increment of rcu_flipctr (line 16), and if an NMI occurred between these two events, then an rcu_read_lock() in that NMI’s handler would incorrectly conclude that it was already under the protection of rcu_read_lock().

3. If the assignment to rcu_read_lock_nesting (line 17) were to be reordered to follow the assignment to rcu_flipctr_idx (line 18), and if an NMI occurred between these two events, then an rcu_read_lock() in that NMI’s handler would clobber rcu_flipctr_idx, possibly causing the matching rcu_read_unlock() to decrement the wrong counter. This in turn could result in premature ending of a grace period, indefinite extension of a grace period, or even both.

It is not clear that the ACCESS_ONCE on the assignment to nesting (line 7) is required. It is also unclear whether the smp_read_barrier_depends() (line 15) is needed: it was added to ensure that changes to index and value remain ordered.

The reasons that irqs must be disabled from line 13 through line 19 are as follows:
1. Suppose one CPU loaded `rcu_ctrlblk.completed` (line 14), then a second CPU incremented this counter, and then the first CPU took a scheduling-clock interrupt. The first CPU would then see that it needed to acknowledge the counter flip, which it would do. This acknowledgment is a promise to avoid incrementing the newly old counter, and this CPU would break this promise. Worse yet, this CPU might be preempted immediately upon return from the scheduling-clock interrupt, and thus end up incrementing the counter at some random point in the future. Either situation could disrupt grace-period detection.

2. Disabling irqs has the side effect of disabling preemption. If this code were to be preempted between fetching `rcu_ctrlblk.completed` (line 14) and incrementing `rcu_flipctr` (line 16), it might well be migrated to some other CPU. This would result in it non-atomically incrementing the counter from that other CPU. If this CPU happened to be executing in `rcu_read_lock()` or `rcu_read_unlock()` just at that time, one of the increments or decrements might be lost, again disrupting grace-period detection. The same result could happen on RISC machines if the preemption occurred in the middle of the increment (after the fetch of the old counter but before the store of the newly incremented counter).

3. Permitting preemption in the midst of line 16, between selecting the current CPU’s copy of the `rcu_flipctr` array and the increment of the element indicated by `rcu_flipctr_idx`, can result in a similar failure. Execution might well resume on some other CPU. If this resumption happened concurrently with an `rcu_read_lock()` or `rcu_read_unlock()` running on the original CPU, an increment or decrement might be lost, resulting in either premature termination of a grace period, indefinite extension of a grace period, or even both.

4. Failing to disable preemption can also defeat RCU priority boosting, which relies on `rcu_read_lock_nesting` to determine when a given task is in an RCU read-side critical section. So, for example, if a given task is indefinitely preempted just after incrementing `rcu_flipctr`, but before updating `rcu_read_lock_nesting`, then it will stall RCU grace periods for as long as it is preempted. However, because `rcu_read_lock_nesting

```c
1 void __rcu_read_unlock(void)
2 {
3     int idx;
4     struct task_struct *t = current;
5     int nesting;
6     nesting = ACCESS_ONCE(t->rcu_read_lock_nesting);
7     if (nesting > 1) {
8         t->rcu_read_lock_nesting = nesting - 1;
9     } else {
10         unsigned long flags;
11         local_irq_save(flags);
12         idx = ACCESS_ONCE(t->rcu_flipctr_idx);
13         ACCESS_ONCE(t->rcu_read_lock_nesting) = nesting - 1;
14         ACCESS_ONCE(__get_cpu_var(rcu_flipctr)[idx])--;
15         local_irq_restore(flags);
16     }
```

Figure D.74: __rcu_read_unlock() Implementation

has not yet been incremented, the RCU priority booster has no way to tell that boosting is needed. Therefore, in the presence of CPU-bound realtime threads, the preempted task might stall grace periods indefinitely, eventually causing an OOM event.

The last three reasons could of course be addressed by disabling preemption rather than disabling of irqs, but given that the first reason requires disabling irqs in any case, there is little reason to separately disable preemption. It is entirely possible that the first reason might be tolerated by requiring an additional grace-period stage, however, it is not clear that disabling preemption is much faster than disabling interrupts on modern CPUs.

`rcu_read_unlock()` The implementation of `rcu_read_unlock()` is shown in Figure D.74. Line 7 fetches the `rcu_read_lock_nesting` counter, which line 8 checks to see if we are under the protection of an enclosing `rcu_read_lock()` primitive. If so, line 9 simply decrements the counter.

However, as with `rcu_read_lock()`, we otherwise must do more work. Lines 13 and 17 disable and restore irqs in order to prevent the scheduling-clock interrupt from invoking the grace-period state machine while in the midst of `rcu_read_unlock()` processing. Line 14 picks up the `rcu_flipctr_idx` that was saved by the matching `rcu_read_lock()`, line 15 decrements `rcu_read_lock_nesting` so that irq and NMI/SMI handlers will henceforth update `rcu_flipctr`, line 16 decrements the counter (with the same index as, but possibly on a different CPU than, that incremented by the
D.4. PREEMPTIBLE RCU

matching rcu_read_lock(). The ACCESS_ONCE() macros and irq disabling are required for similar reasons that they are in rcu_read_lock().

Quick Quiz D.59: What problems could arise if the lines containing ACCESS_ONCE() in rcu_read_unlock() were reordered by the compiler?

Quick Quiz D.60: What problems could arise if the lines containing ACCESS_ONCE() in rcu_read_unlock() were reordered by the CPU?

Quick Quiz D.61: What problems could arise in rcu_read_unlock() if irqs were not disabled?

Figure D.75: Preemptible RCU with Read-Side Memory Barriers

Memory-Barrier Considerations Note that these two primitives contains no memory barriers, so there is nothing to stop the CPU from executing the critical section before executing the rcu_read_lock() or after executing the rcu_read_unlock(). The purpose of the rcu_try_flip_waitmb_state is to account for this possible reordering, but only at the beginning or end of a grace period. To see why this approach is helpful, consider Figure D.75, which shows the wastefulness of the conventional approach of placing a memory barrier at the beginning and end of each RCU read-side critical section [MSMB06].

Figure D.76: Preemptible RCU with Grace-Period Memory Barriers

The "MB"s represent memory barriers, and only the emboldened barriers are needed, namely the first and last on a given CPU for each grace period. This preemptible RCU implementation therefore associates the memory barriers with the grace period, as shown in Figure D.76. Given that the Linux kernel can execute literally millions of RCU read-side critical sections per grace period, this latter approach can result in substantial read-side savings, due to the fact that it amortizes the cost of the memory barrier over all the read-side critical sections in a grace period.

D.4.3 Validation of Preemptible RCU

D.4.3.1 Testing

The preemptible RCU algorithm was tested with a two-stage grace period on weakly ordered POWER4 and POWER5 CPUs using rcutorture running for more than 24 hours on each machine, with 15M and 20M grace periods, respectively, and with no errors. Of course, this in no way proves that this algorithm is correct. At most, it shows either that these two machines were extremely lucky or
that any bugs remaining in preemptible RCU have an extremely low probability of occurring. We therefore required additional assurance that this algorithm works, or, alternatively, identification of remaining bugs.

This task requires a conceptual approach, which is taken in the next section.

D.4.3.2 Conceptual Validation

Because neither \texttt{rcu_read_lock()} nor \texttt{rcu_read_unlock()} contain memory barriers, the RCU read-side critical section can bleed out on weakly ordered machines. In addition, the relatively loose coupling of this RCU implementation permits CPUs to disagree on when a given grace period starts and ends. This leads to the question as to how long a given RCU read-side critical section can possibly extend relative to the grace-period state machine.

![Diagram]

Figure D.77: Preemptible RCU Worst-Case Scenario

The worst-case scenario is shown in Figure D.77. Here, CPU 0 is executing the shortest possible removal and reclamation sequence, while CPU 1 executes the longest possible RCU read-side critical section. Because the callback queues are advanced just before acknowledging a counter flip, the latest that CPU 0 can execute its \texttt{list_del_rcu()} and \texttt{call_rcu()} is just before its scheduling-clock interrupt that acknowledges the counter flip. The \texttt{call_rcu()} invocation places the callback on CPU 0’s next list, and the interrupt will move the callback from the next list to the wait[0] list. This callback will move again (from the wait[0] list to the wait[1] list) at CPU 0’s first scheduling-clock interrupt following the next counter flip. Similarly, the callback will move from the wait[1] list to the done list at CPU 0’s first scheduling-clock interrupt following the counter flip resulting in the value 3. The callback might be invoked immediately afterward.

Meanwhile, CPU 1 is executing an RCU read-side critical section. Let us assume that the \texttt{rcu_read_lock()} follows the first counter flip (the one resulting in the value 1), so that the \texttt{rcu_read_lock()} increments CPU 1’s \texttt{rcu_flipctr[1]} counter. Note that because \texttt{rcu_read_lock()} does not contain any memory barriers, the contents of the critical section might be executed early by the CPU. However, this early execution cannot precede the last memory barrier executed by CPU 1, as shown on the diagram. This is nevertheless sufficiently early that an \texttt{rcu_dereference()} could fetch a pointer to the item being deleted by CPU 0’s \texttt{list_del_rcu}().

Because the \texttt{rcu_read_lock()} incremented an index-1 counter, the corresponding \texttt{rcu_read_unlock()} must precede the "old counters zero" event for index 1. However, because \texttt{rcu_read_unlock()} contains no memory barriers, the contents of the corresponding RCU read-side critical section (possibly including a reference to the item deleted by CPU 0) can be executed late by CPU 1. However, it cannot be executed after CPU 1’s next memory barrier, as shown on the diagram. Because the latest possible reference by CPU 1 precedes the earliest possible callback invocation by CPU 0, two passes through the grace-period state machine suffice to constitute a full grace period, and hence it is safe to do:

```c
#define GP_STAGES 2
```

Quick Quiz D.62: Suppose that the irq disabling in \texttt{rcu_read_lock()} was replaced by preemption disabling. What effect would that have on \texttt{GP_STAGES}?

Quick Quiz D.63: Why can’t the \texttt{rcu_dereference()} precede the memory barrier?
D.4.3.3 Formal Validation

Formal validation of this algorithm is quite important, but remains as future work. One tool for doing this validation is described in Appendix F.

**Quick Quiz D.64:** What is a more precise way to say "CPU 0 might see CPU 1’s increment as early as CPU 1’s last previous memory barrier"? ■
Appendix E

Read-Copy Update in Linux

This chapter gives a history of RCU in the Linux kernel from mid-2008 onwards. Earlier history of RCU may be found elsewhere [McK04, MW08]. Section E.1 gives an overview of the growth of RCU usage in Linux and Section E.2 presents a detailed view of recent RCU evolution.

E.1 RCU Usage Within Linux

The Linux kernel’s usage of RCU has increased over the years, as can be seen from Figure E.1 [McK06a]. RCU has replaced other synchronization mechanisms in existing code (for example, brolck in the networking protocol stacks [MM00, Tor03a, Tor03b]), and it has also been introduced with code implementing new functionality (for example, the audit system within SELinux [Mor04]). However, RCU remains a niche technology compared to locking, as shown in Figure E.2. If locking is the hammer in the kernel hacker’s concurrency toolbox, perhaps RCU is the screwdriver. If so, it is an rapidly evolving screwdriver, as can be seen in Figure E.3.

E.2 RCU Evolution

This section presents ongoing experience with RCU since mid-2008.

E.2.1 2.6.27 Linux Kernel

This release added the call_rcu_sched(), rcu_barrier_sched(), and rcu_barrier_bh() RCU API members.
E.2.2 2.6.28 Linux Kernel

One welcome change involved an actual reduction in the size of RCU’s API with the removal of the list_for_each_rcu() primitive. This primitive is superseded by list_for_each_entry_rcu(), which has the advantage of iterating over structures rather than iterating over the pointer pairs making up a list_head structure (which, confusingly, acts as a list element as well as a list header). This change was accepted into the 2.6.28 Linux kernel.

Unfortunately, the 2.6.28 Linux kernel also added rcu_read_lock_sched() and rcu_read_unlock_sched() RCU API members. These APIs were added to promote readability. In the past, primitives to disable interrupts or preemption were used to mark the RCU read-side critical sections corresponding to synchronize_sched(). However, this practice led to bugs when developers removed the need to disable preemption or interrupts, but failed to notice the need for RCU protection. Use of rcu_read_lock_sched() will help prevent such bugs in the future.

E.2.3 2.6.29 Linux Kernel

A new more-scalable implementation, dubbed “Tree RCU”, replaces the flat bitmap with a combining tree, and was accepted into the 2.6.29 Linux kernel. This implementation was inspired by the ever-growing core counts of modern multiprocessors, and is designed for many hundreds of CPUs. Its current architectural limit is 262,144 CPUs, which the developer (perhaps naively) believes to be sufficient for quite some time. This implementation also adopts preemptible RCU’s improved dynamic-tick interface.

Mathieu Desnoyers added rcu_read_lock_sched_notrace() and rcu_read_unlock_sched_notrace(), which are required to permit the tracing code in the Linux kernel to use RCU. Without these APIs, attempts to trace RCU read-side critical sections lead to infinite recursion.

Eric Dumazet added a new type of RCU-protected list that allows single-bit markers to be stored in the list pointers. This type of list enables a number of lockless algorithms, including some reported on by Maged Michael [Mic04]. Eric’s work adds the hlist_nulls_add_head_rcu(), hlist_nulls_del_rcu(), hlist_nulls_del_init_rcu(), and hlist_nulls_for_each_entry_rcu(). It also adds a new structure named hlist_nulls_node.
Although it is strictly speaking not part of the Linux kernel, at about this same time, Mathieu Desnoyers announced his user-space RCU implementation [Des09]. This is an important first step towards a real-time user-level RCU implementation.

### E.2.4 2.6.31 Linux Kernel

Jiri Pirko added list_entry_rcu and list_first_entry_rcu() primitives that encapsulate the rcu_dereference() RCU-subscription primitive into higher-level list-access primitives, which will hopefully eliminate a class of bugs.

In addition, the "Tree RCU" implementation was upgraded from "experimental" status.

### E.2.5 2.6.32 Linux Kernel

Perhaps the largest change in this version of the Linux kernel is the removal of the old "Classic RCU" implementation. This implementation is superseded by the "Tree RCU" implementation.

This version saw a number of other changes, including:

1. The appearance of synchronize_rcu_expedited(), synchronize_sched_expedited(), and synchronize_rcu_bh_expedited() RCU API members. These primitives are equivalent to their non-expedited counterparts, except that they take measures to expedite the grace period.
2. Add preemptible-RCU functionality to the "Tree RCU" implementation, thus removing one obstacle to real-time response from large multiprocessor machines running Linux.
3. This new "Tree Preemptible RCU" implementation obsoletes the old preemptible RCU implementation, which was removed from the Linux kernel.

### E.2.6 2.6.33 Linux Kernel

Perhaps the most dramatic addition to this release was a day-one bug in Tree RCU [McK09a]. Other changes include:

1. "Tiny RCU", also known as "RCU: The Bloatwatch Edition" [McK09b].
2. Expedited SRCU in the form of synchronize_srcu_expedited().
3. A cleanup of Tree RCU synchronization prompted by the afore-mentioned bug.
4. Add expedited implementation for Tree Preemptible RCU (in earlier releases, “expedited” support had simply mapped to synchronize_rcu(), which is semantically correct if somewhat unhelpful from a performance viewpoint.)
5. Add a fourth level to Tree RCU, which improves stress testing. Therefore, if someone ever wants to run Linux on a system with 16,777,216 CPUs, RCU is ready for them! Give or take the response-time implications of scanning through 16 million per-CPU data elements...

### E.2.7 2.6.34 Linux Kernel

The most visible addition for this release was CONFIG_PROVE_RCU, which allows rcu_dereference() to check for correct locking conditions [McK10]. Other changes include:

1. Simplifying Tree RCU’s interactions between forcing an old grace period and starting a new one.
2. Rework counters so that free-running counters are unsigned. (You simply cannot imagine the glee on the faces of certain C-compiler hackers while they discussed optimizations that would break code that naively overflowed signed integers!!!)
3. Update Tree Preemptible RCU’s stall detection to print out any tasks preempted for excessive time periods while in an RCU read-side critical section.
4. Other bug fixes and improvements to Tree RCU’s CPU-stall-detection code. This code checks for CPUs being locked up, for example, in infinite loops with interrupts disabled.
5. Prototype some code to accelerate grace periods when the last CPU goes idle in battery-powered multiprocessor systems. There were people who were quite unhappy about RCU taking a few extra milliseconds to get the system in a state where all CPUs could be powered down!

### E.2.8 2.6.35 Linux Kernel

This release includes a number of bug fixes and cleanups. The major change is the first installment of Mathieu
Desnoyers’s patch to check for misuse of RCU callbacks, for example, passing a *rcu_head* structure to `call_rcu()` a second time within a single grace period.

### E.2.9 2.6.36 Linux Kernel

The core of Mathieu Desnoyers’s debugobjects work appeared in 2.6.36, with some cleanups deferred to 2.6.37 due to dependencies on commits flowing up other maintainer trees. A key piece of Arnd Bergmann’s sparse RCU checking appeared in 2.6.36, with the remainder deferred to 2.6.37, again due to dependencies on commits flowing up other maintainer trees. Finally, a patch from Eric Dumazet fixed an error in `rcu_dereference_bh()`’s error checking.

### E.2.10 2.6.37 Linux Kernel

The final cleanups from Mathieu Desnoyers’s debugobjects work appeared in 2.6.37, as did the remainder of Arnd Bergmann’s sparse-based checking work. Lai Jiangshan added some preemption nastiness to rcutorture and made some simplifications to Tree RCU’s handling of per-CPU data. Tetsuo Handa fixed an RCU lockdep splat, Christian Dietrich removed a redundant `#ifdef`, and Dongdong Deng added an `ACCESS_ONCE()` that help call out lockless accesses to some Tree RCU control data.

Paul’s implementation of preemptible Tiny RCU also appeared in 2.6.37, as did a number of enhancements to the RCU CPU stall-warning code, docbook fixes, coalescing of duplicate code, Tree RCU speedups, added tracing to support queuing models on RCU callback flow, and several miscellaneous fixes and cleanups.

### E.2.11 2.6.38 Linux Kernel

Lai Jiangshan moved `synchronize_sched_expedited()` out of `kernel/sched.c` and into `kernel/rcutree.c` and `kernel/rcu_tiny.c` where it belongs. He also simplified RCU-callback handling during CPU-hotplug operations by eliminating the `orphan_cbs_list`, so that RCU callbacks orphaned by a CPU that is going offline are immediately adopted by the CPU that is orchestrating the offlining sequence. Tejun Heo improved `synchronize_sched_expedited()`’s batching capabilities, which in turn improves performance and scalability for workloads with many concurrent `synchronize_sched_expedited` operations. Frederic Weisbecker provided a couple of subtle changes to the RCU core code that make RCU more power-efficient when idle. Mariusz Kozłowski fixed an embarrassing syntax error in `__list_for_each_rcu()`, which was then removed. (But the fixed version is there in the git tree should it be needed.) Nick Piggin added the `hlist_bl_set_first_rcu()`, `hlist_bl_first_rcu()`, `hlist_bl_del_init_rcu()`, `hlist_bl_del_rcu()`, `hlist_bl_add_head_rcu()`, and `hlist_bl_for_each_entry_rcu()` primitives for RCU-protected use of bit-locked doubly-linked lists. Christoph Lameter implemented `__this_cpu_read()`, which is an optimized variant of `__get_cpu_var()` for use in cases where the variable is accessed directly.

In addition, `TINY_RCU` gained priority boosting, a race condition in `synchronize_sched_expedited()` was fixed, `synchronize_srcu_expedited()` was modified to retain its expedited nature in the face of concurrent readers, grace-period begin/end checks were improved, and the `TREE_RCU` leaf-level fanout was limited to 16 in order to fix lock-contention problems. This last change reduces the maximum number of CPUs that `TREE_RCU` and `TREE_PREEMPT_RCU` can support down to 4,194,304, which is (again, perhaps naïvely) believed to be sufficient.

### E.2.12 2.6.39 Linux Kernel

Lai Jiangshan made `TINY_RCU`’s `exit_rcu()` invoke `__rcu_read_unlock()` rather than `rcu_read_unlock()` in case of a task exiting while in an RCU read-side critical section in order to preserve debugging state. Jesper Juhl removed a duplicate include of `sched.h` from rcutorture, and Amerigo Wang removed some dead code from `rcu_fixup_free()`.

In addition, a new `rcu_access_index()` was created for use in the MCE subsystem.

### E.2.13 What Comes After 2.6.39?

At long last, priority boosting for Tree RCU should appear in 2.6.40.

Mathieu Desnoyers enabled `DEBUG_OBJECTS_RCU_HEAD` checking to be carried out in non-preemptible RCU implementations. Lai Jiangshan created a fire-and-forget `kfree_rcu()` (and applied it throughout the kernel), and also made `TREE_RCU`’s `exit_rcu()` invoke `__rcu_read_unlock()` rather than `rcu_read_unlock()` in case of a task exiting while in an
RCU read-side critical section in order to preserve debugging state. Eric Dumazet further shrunk TINY_RCU and applied the new \texttt{kthread\_create\_on\_node()} primitive to ensure that RCU’s kthreads have memory placed optimally on NUMA systems. Gleb Natapov added RCU hooks to allow virtualization to call RCU’s attention to quiescent states that occur when switching context to and from a guest OS. Peter Zijlstra streamlined RCU kthread blocking and wakeup.

The design for pulling SRCU into the Tree RCU implementation is still looking reasonably good, aside from interactions with the soft-lockup code. However, it will still likely be a few releases before SRCU is pulled into Tree RCU and Tiny RCU.

There has been an initial request for \texttt{rcu\_barrier\_expedited()}, but given that the requester found another way to solve this problem, this has relatively low priority.
Appendix F

Formal Verification

Parallel algorithms can be hard to write, and even harder to debug. Testing, though essential, is insufficient, as fatal race conditions can have extremely low probabilities of occurrence. Proofs of correctness can be valuable, but in the end are just as prone to human error as is the original algorithm.

It would be very helpful to have a tool that could somehow locate all race conditions. A number of such tools exist, for example, the language Promela and its compiler Spin, which are described in this chapter. Section F.1 provides an introduction to Promela and Spin, Section F.2 demonstrates use of Promela and Spin to find a race in a non-atomic increment example, Section F.3 uses Promela and Spin to validate a similar atomic-increment example, Section F.4 gives an overview of using Promela and Spin, Section F.5 demonstrates a Promela model of a spinlock, Section F.6 applies Promela and Spin to validate a simple RCU implementation, Section F.7 applies Promela to validate an interface between preemptible RCU and the dyntick-idle energy-conservation feature in the Linux kernel, Section F.8 presents a simpler interface that does not require formal verification, and finally Section F.9 sums up use of formal-verification tools for verifying parallel algorithms.

F.1 What are Promela and Spin?

Promela is a language designed to help verify protocols, but which can also be used to verify small parallel algorithms. You recode your algorithm and correctness constraints in the C-like language Promela, and then use Spin to translate it into a C program that you can compile and run. The resulting program conducts a full state-space search of your algorithm, either verifying or finding counter-examples for assertions that you can include in your Promela program.

This full-state search can extremely powerful, but can also be a two-edged sword. If your algorithm is too complex or your Promela implementation is careless, there might be more states than fit in memory. Furthermore, even given sufficient memory, the state-space search might well run for longer than the expected lifetime of the universe. Therefore, use this tool for compact but complex parallel algorithms. Attempts to naively apply it to even moderate-scale algorithms (let alone the full Linux kernel) will end badly.

Promela and Spin may be downloaded from http://spinroot.com/spin/whatispin.html.

The above site also gives links to Gerard Holzmann’s excellent book [Hol03] on Promela and Spin, as well as searchable online references starting at: http://www.spinroot.com/spin/Man/index.html.

The remainder of this article describes how to use Promela to debug parallel algorithms, starting with simple examples and progressing to more complex uses.

F.2 Promela Example: Non-Atomic Increment

Figure F.1 demonstrates the textbook race condition resulting from non-atomic increment. Line 1 defines the number of processes to run (we will vary this to see the effect on state space), line 3 defines the counter, and line 4 is used to implement the assertion that appears on lines 29-39.

Lines 6-13 define a process that increments the counter non-atomically. The argument me is the process number, set by the initialization block later in the code. Because simple Promela statements are each assumed atomic, we must break the increment into the two statements on lines 10-11. The assignment on line 12 marks the process’s
#define NUMPROCS 2
byte counter = 0;
byte progress[NUMPROCS];

proctype incrementer(byte me)
{
    int temp;
    temp = counter;
    counter = temp + 1;
    progress[me] = 1;
}

init {
    int i = 0;
    int sum = 0;
    atomic {
        i = 0;
        do
            :: i < NUMPROCS ->
                progress[i] = 0;
                run incrementer(i);
                i++
            :: i >= NUMPROCS -> break
        od;
    }
    atomic {
        sum = 0;
        do
            :: i < NUMPROCS ->
                sum = sum + progress[i];
                i++
            :: i >= NUMPROCS -> break
        od;
        assert(sum < NUMPROCS || counter == NUMPROCS)
    }
}

Figure F.1: Promela Code for Non-Atomic Increment completion. Because the Spin system will fully search the state space, including all possible sequences of states, there is no need for the loop that would be used for conventional testing.

Lines 15-40 are the initialization block, which is executed first. Lines 19-28 actually do the initialization, while lines 29-39 perform the assertion. Both are atomic blocks in order to avoid unnecessarily increasing the state space: because they are not part of the algorithm proper, we loose no verification coverage by making them atomic.

The do-od construct on lines 21-27 implements a Promela loop, which can be thought of as a C for (;; ) loop containing a switch statement that allows expressions in case labels. The condition blocks (prefixed by :: ) are scanned non-deterministically, though in this case only one of the conditions can possibly hold at a given time. The first block of the do-od from lines 22-25 initializes the i-th incrementer’s progress cell, runs the i-th incrementer’s process, and then increments the variable i. The second block of the do-od on line 26 exits the loop once these processes have been started.

The atomic block on lines 29-39 also contains a similar do-od loop that sums up the progress counters. The assert ( ) function on line 38 verifies that if all processes have been completed, then all counts have been correctly recorded.

You can build and run this program as follows:

```
spin -a increment.spin # Translate the model to C
cc -DSAFETY -o pan pan.c # Compile the model
./pan # Run the model
```

This will produce output as shown in Figure F.2. The first line tells us that our assertion was violated (as expected given the non-atomic increment!). The second line that a trail file was written describing how the assertion was violated. The “Warning” line reiterates that all was not well with our model. The second paragraph describes the type of state-search being carried out, in this case for assertion violations and invalid end states. The third paragraph gives state-size statistics: this small model had only 45 states. The final line shows memory usage.

The trail file may be rendered human-readable as follows:

```
spin -t -p increment.spin
```

This gives the output shown in Figure F.3. As can be seen, the first portion of the init block created both incrementer processes, both of which first fetched the
F.2. PROMELA EXAMPLE: NON-ATOMIC INCREMENT

pan: assertion violated ((sum<2)||(counter==2)) (at depth 20)
pan: wrote increment.spin.trail
(Spin Version 4.2.5 -- 2 April 2005)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
never claim - (none specified)
assertion violations +
cycle checks - (disabled by -DSAFETY)
invalid end states +

State-vector 40 byte, depth reached 22, errors: 1
45 states, stored
13 states, matched
58 transitions (= stored+matched)
51 atomic steps
hash conflicts: 0 (resolved)
2.622 memory usage (Mbyte)

Figure F.2: Non-Atomic Increment spin Output

Starting :init: with pid 0
1: proc 0 (:init:) line 20 "increment.spin" (state 1) [i = 0]
2: proc 0 (:init:) line 22 "increment.spin" (state 2) [((i<2))]

Starting incrementer with pid 1
3: proc 0 (:init:) line 24 "increment.spin" (state 4) [(run incrementer(i))]
4: proc 0 (:init:) line 22 "increment.spin" (state 2) [((i<2))]
5: proc 0 (:init:) line 23 "increment.spin" (state 3) [progress[i] = 0]

Starting incrementer with pid 2
6: proc 0 (:init:) line 24 "increment.spin" (state 4) [(run incrementer(i))]
7: proc 0 (:init:) line 22 "increment.spin" (state 2) [((i<2))]
8: proc 0 (:init:) line 25 "increment.spin" (state 5) [i = (i+1)]
9: proc 1 (incrementer) line 10 "increment.spin" (state 1) [temp = counter]
10: proc 1 (incrementer) line 11 "increment.spin" (state 2) [counter = (temp+1)]

11: proc 2 (incrementer) line 12 "increment.spin" (state 3) [progress[me] = 1]
12: proc 2 terminates

13: proc 1 (incrementer) line 11 "increment.spin" (state 2) [counter = (temp+1)]
14: proc 1 (incrementer) line 12 "increment.spin" (state 3) [progress[me] = 1]
15: proc 1 terminates
16: proc 0 (:init:) line 30 "increment.spin" (state 12) [i = 0]
17: proc 0 (:init:) line 31 "increment.spin" (state 13) [sum = 0]
18: proc 0 (:init:) line 33 "increment.spin" (state 14) [((i<2))]
19: proc 0 (:init:) line 34 "increment.spin" (state 15) [sum = (sum+progress[i])]
20: proc 0 (:init:) line 35 "increment.spin" (state 16) [i = (i+1)]
21: proc 0 (:init:) line 38 "increment.spin" (state 22) [assert(((sum<2)||(counter==2)))]

spin: line 38 "increment.spin", Error: assertion violated
spin: text of failed assertion: assert(((sum<2)||(counter==2)))

spin: trail ends after 21 steps
#processes: 1
  counter = 1
  progress[0] = 1
  progress[1] = 1
21: proc 0 (:init:) line 40 "increment.spin" (state 24) <valid end state>
3 processes created

Figure F.3: Non-Atomic Increment Error Trail
counter, then both incremented and stored it, losing a count. The assertion then triggered, after which the global state is displayed.

F.3 Promela Example: Atomic Increment

```promela
1 proc type incrementer(byte me)
2 {
3     int temp;
4     atomic {
5         temp = counter;
6         counter = temp + 1;
7     }
8     progress[me] = 1;
9 }
```

Figure F.4: Promela Code for Atomic Increment

(Spin Version 4.2.5 -- 2 April 2005)

Full statespace search for:
- never claim - (none specified)
- assertion violations +
- cycle checks - (disabled by -DSAFETY)
- invalid end states +

State-vector 40 byte, depth reached 20, errors: 0
52 states, stored
21 states, matched
73 transitions (= stored+matched)
66 atomic steps
hash conflicts: 0 (resolved)
2.622 memory usage (Mbyte)
unreached in proc type incrementer
(0 of 5 states)
unreached in proc type :init:
(0 of 24 states)

Figure F.5: Atomic Increment spin Output

It is easy to fix this example by placing the body of the incrementer processes in an atomic blocks as shown in Figure F.4. One could also have simply replaced the pair of statements with `counter = counter + 1;`, because Promela statements are atomic. Either way, running this modified model gives us an error-free traversal of the state space, as shown in Figure F.5.

F.3.1 Combinatorial Explosion

Table F.1 shows the number of states and memory consumed as a function of number of incrementers modeled

<table>
<thead>
<tr>
<th># incrementers</th>
<th># states</th>
<th>megabytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>2.6</td>
</tr>
<tr>
<td>2</td>
<td>52</td>
<td>2.6</td>
</tr>
<tr>
<td>3</td>
<td>372</td>
<td>2.6</td>
</tr>
<tr>
<td>4</td>
<td>3,496</td>
<td>2.7</td>
</tr>
<tr>
<td>5</td>
<td>40,221</td>
<td>5.0</td>
</tr>
<tr>
<td>6</td>
<td>545,720</td>
<td>40.5</td>
</tr>
<tr>
<td>7</td>
<td>8,521,450</td>
<td>652.7</td>
</tr>
</tbody>
</table>

Table F.1: Memory Usage of Increment Model
(by redefining NUMPROCS):
Running unnecessarily large models is thus subtly discouraged, although 652MB is well within the limits of modern desktop and laptop machines.

With this example under our belt, let’s take a closer look at the commands used to analyze Promela models and then look at more elaborate examples.

F.4 How to Use Promela

Given a source file qrcu.spin, one can use the following commands:

- `spin -a qrcu.spin` Create a file pan.c that fully searches the state machine.
- `cc -DSAFETY -o pan pan.c` Compile the generated state-machine search. The `-DSAFETY` generates optimizations that are appropriate if you have only assertions (and perhaps never statements). If you have liveness, fairness, or forward-progress checks, you may need to compile without `-DSAFETY`. If you leave off `-DSAFETY` when you could have used it, the program will let you know.

The optimizations produced by `-DSAFETY` greatly speed things up, so you should use it when you can. An example situation where you cannot use `-DSAFETY` is when checking for livelocks (AKA “non-progress cycles”) via `-DNP`.

- `./pan` This actually searches the state space. The number of states can reach into the tens of millions with very small state machines, so you will need a machine with large memory. For example, qrcu.spin with 3 readers and 2 updaters required 2.7GB of memory.
If you aren’t sure whether your machine has enough memory, run `top` in one window and `./pan` in another. Keep the focus on the `./pan` window so that you can quickly kill execution if need be. As soon as CPU time drops much below 100%, kill `./pan`. If you have removed focus from the window running `./pan`, you may wait a long time for the windowing system to grab enough memory to do anything for you.

Don’t forget to capture the output, especially if you are working on a remote machine.

If your model includes forward-progress checks, you will likely need to enable “weak fairness” via the `-f` command-line argument to `./pan`. If your forward-progress checks involve accept labels, you will also need the `-a` argument.

- `spin -t -p qrcu.spin` Given trail file output by a run that encountered an error, output the sequence of steps leading to that error. The `-g` flag will also include the values of changed global variables, and the `-l` flag will also include the values of changed local variables.

F.4.1 Promela Peculiarities

Although all computer languages have underlying similarities, Promela will provide some surprises to people used to coding in C, C++, or Java.

1. In C, “;” terminates statements. In Promela it separates them. Fortunately, more recent versions of Spin have become much more forgiving of “extra” semicolons.

2. Promela’s looping construct, the `do` statement, takes conditions. This `do` statement closely resembles a looping if-then-else statement.

3. In C’s `switch` statement, if there is no matching case, the whole statement is skipped. In Promela’s equivalent, confusingly called `if`, if there is no matching guard expression, you get an error without a recognizable corresponding error message. So, if the error output indicates an innocent line of code, check to see if you left out a condition from an `if` or `do` statement.

4. When creating stress tests in C, one usually races suspect operations against each other repeatedly. In Promela, one instead sets up a single race, because Promela will search out all the possible outcomes from that single race. Sometimes you do need to loop in Promela, for example, if multiple operations overlap, but doing so greatly increases the size of your state space.

5. In C, the easiest thing to do is to maintain a loop counter to track progress and terminate the loop. In Promela, loop counters must be avoided like the plague because they cause the state space to explode. On the other hand, there is no penalty for infinite loops in Promela as long as the none of the variables monotonically increase or decrease – Promela will figure out how many passes through the loop really matter, and automatically prune execution beyond that point.

6. In C torture-test code, it is often wise to keep per-task control variables. They are cheap to read, and greatly aid in debugging the test code. In Promela, per-task control variables should be used only when there is no other alternative. To see this, consider a 5-task verification with one bit each to indicate completion. This gives 32 states. In contrast, a simple counter would have only six states, more than a five-fold reduction. That factor of five might not seem like a problem, at least not until you are struggling with a verification program possessing more than 150 million states consuming more than 10GB of memory!

7. One of the most challenging things both in C torture-test code and in Promela is formulating good assertions. Promela also allows `never` claims that act sort of like an assertion replicated between every line of code.

8. Dividing and conquering is extremely helpful in Promela in keeping the state space under control. Splitting a large model into two roughly equal halves will result in the state space of each half being roughly the square root of the whole. For example, a million-state combined model might reduce to a pair of thousand-state models. Not only will Promela handle the two smaller models much more quickly with much less memory, but the two smaller algorithms are easier for people to understand.
APPENDIX F. FORMAL VERIFICATION

1 i = 0;
2 sum = 0;
3 do
4 :: i < N_QRCU_READERS ->
5 sum = sum + (readerstart[i] == 1 &&
6 readerprogress[i] == 1);
7 i++
8 :: i >= N_QRCU_READERS ->
9 assert(sum == 0);
10 break
11 od

Figure F.6: Complex Promela Assertion

1 atomic {
2 i = 0;
3 sum = 0;
4 do
5 :: i < N_QRCU_READERS ->
6 sum = sum + (readerstart[i] == 1 &&
7 readerprogress[i] == 1);
8 i++
9 :: i >= N_QRCU_READERS ->
10 assert(sum == 0);
11 break
12 od
13 }

Figure F.7: Atomic Block for Complex Promela Assertion

F.4.2 Promela Coding Tricks

Promela was designed to analyze protocols, so using it on parallel programs is a bit abusive. The following tricks can help you to abuse Promela safely:

1. Memory reordering. Suppose you have a pair of statements copying globals x and y to locals r1 and r2, where ordering matters (e.g., unprotected by locks), but where you have no memory barriers. This can be modeled in Promela as follows:

1 if
2 :: 1 -> r1 = x;
3 r2 = y
4 :: 1 -> r2 = y;
5 fi

The two branches of the if statement will be selected nondeterministically, since they both are available. Because the full state space is searched, both choices will eventually be made in all cases.

Of course, this trick will cause your state space to explode if used too heavily. In addition, it requires you to anticipate possible reorderings.

2. State reduction. If you have complex assertions, evaluate them under atomic. After all, they are not part of the algorithm. One example of a complex assertion (to be discussed in more detail later) is as shown in Figure F.6.

There is no reason to evaluate this assertion non-atomically, since it is not actually part of the algorithm. Because each statement contributes to state, we can reduce the number of useless states by enclosing it in an atomic block as shown in Figure F.7

3. Promela does not provide functions. You must instead use C preprocessor macros. However, you must use them carefully in order to avoid combinatorial explosion.

Now we are ready for more complex examples.

F.5 Promela Example: Locking

Since locks are generally useful, spin_lock() and spin_unlock() macros are provided in lock.h, which may be included from multiple Promela models, as shown in Figure F.8. The spin_lock() macro contains an infinite do-od loop spanning lines 2-11, courtesy of the single guard expression of “1” on line 3. The body of this loop is a single atomic block that contains an if-fi statement. The if-fi construct is similar to the do-od construct, except that it takes a single pass rather than looping. If the lock is not held on line 5, then line 6 acquires it and line 7 breaks out of the enclosing do-od loop (and also exits the atomic block). On the other hand, if the lock is already held on line 8, we do nothing (skip), and fall out of the if-fi and the atomic block so as to take another pass through the outer loop, repeating until the lock is available.

The spin_unlock() macro simply marks the lock as no longer held.
Note that memory barriers are not needed because Promela assumes full ordering. In any given Promela state, all processes agree on both the current state and the order of state changes that caused us to arrive at the current state. This is analogous to the “sequentially consistent” memory model used by a few computer systems (such as MIPS and PA-RISC). As noted earlier, and as will be seen in a later example, weak memory ordering must be explicitly coded.

```c
#include "lock.h"

#define N_LOCKERS 3

bit mutex = 0;
bit havelock[N_LOCKERS];
int sum;

proctype locker(byte me)
{
  do :: 1 ->
    spin_lock(mutex);
    havelock[me] = 1;
    havelock[me] = 0;
    spin_unlock(mutex)
  od
}

init {
  int i = 0;
  int j;
  do :: i < N_LOCKERS ->
    spin_lock(mutex);
    havelock[i] = 1;
    run locker(i);
    i++
  :: i >= N_LOCKERS ->
  end: do :: i < N_LOCKERS ->
    sum = 0;
    j = 0;
    atomic {
      do :: j < N_LOCKERS ->
        sum = sum + havelock[j];
        j = j + 1
      :: j >= N_LOCKERS ->
      break
    od
    assert(sum <= 1);
    break
  od

assert(sum <= 1);
break
od
```

Figure F.9: Promela Code to Test Spinlocks

These macros are tested by the Promela code shown in Figure F.9. This code is similar to that used to test the increments, with the number of locking processes defined by the N_LOCKERS macro definition on line 3. The mutex itself is defined on line 5, an array to track the lock owner on line 6, and line 7 is used by assertion code to verify that only one process holds the lock.

The locker process is on lines 9-18, and simply loops forever acquiring the lock on line 13, claiming it on line 14, unclaiming it on line 15, and releasing it on line 16.

The init block on lines 20-44 initializes the current locker’s havelock array entry on line 26, starts the current locker on line 27, and advances to the next locker on line 28. Once all locker processes are spawned, the do-od loop moves to line 29, which checks the assertion. Lines 30 and 31 initialize the control variables, lines 32-40 atomically sum the havelock array entries, line 41 is the assertion, and line 42 exits the loop.

We can run this model by placing the above two code fragments into files named lock.h and lock.spin, respectively, and then running the following commands:

```
spin -a lock.spin
cc -DSAFETY -o pan pan.c ./pan
```

(Spin Version 4.2.5 -- 2 April 2005)
+ Partial Order Reduction

Full statespace search for:
never claim  - (none specified)
assertion violations +
cycle checks  - (disabled by -DSAFETY)
invalid end states +

State-vector 40 byte, depth reached 357, errors: 0
564 states, stored
929 states, matched
1493 transitions (= stored+matched)
368 atomic steps
hash conflicts: 0 (resolved)
2.622 memory usage (Mbyte)

unreached in proctype locker
line 18, state 20, "-end-
(1 of 20 states)
unreached in proctype init:
(0 of 22 states)
```

Figure F.10: Output for Spinlock Test

The output will look something like that shown in Figure F.10. As expected, this run has no assertion failures ("errors: 0").

Quick Quiz F.1: Why is there an unreached statement in locker? After all, isn’t this a full state-space search? ■

Quick Quiz F.2: What are some Promela code-style issues with this example? ■

F.6 Promela Example: QRCU

This final example demonstrates a real-world use of Promela on Oleg Nesterov’s QRCU [Nes06a, Nes06b],
But modified to speed up the `synchronize_qrcu()` fastpath.

But first, what is QRCU?

QRCU is a variant of SRCU [McK06b] that trades somewhat higher read overhead (atomic increment and decrement on a global variable) for extremely low grace-period latencies. If there are no readers, the grace period will be detected in less than a microsecond, compared to the multi-millisecond grace-period latencies of most other RCU implementations.

1. There is a `qrcu_struct` that defines a QRCU domain. Like SRCU (and unlike other variants of RCU) QRCU’s action is not global, but instead focused on the specified `qrcu_struct`.

2. There are `qrcu_read_lock()` and `qrcu_read_unlock()` primitives that delimit QRCU read-side critical sections. The corresponding `qrcu_struct` must be passed into these primitives, and the return value from `rcu_read_lock()` must be passed to `rcu_read_unlock()`.

For example:

```c
idx = qrcu_read_lock(&my_qrcu_struct);
/* read-side critical section. */
qrcu_read_unlock(&my_qrcu_struct, idx);
```

3. There is a `synchronize_qrcu()` primitive that blocks until all pre-existing QRCU read-side critical sections complete, but, like SRCU’s `synchronize_srcu()`, QRCU’s `synchronize_qrcu()` need wait only for those read-side critical sections that are using the same `qrcu_struct`.

For example, `synchronize_qrcu(&your_qrcu_struct)` would not need to wait on the earlier QRCU read-side critical section. In contrast, `synchronize_qrcu(&my_qrcu_struct)` would need to wait, since it shares the same `qrcu_struct`.

A Linux-kernel patch for QRCU has been produced [McK07b], but has not yet been included in the Linux kernel as of April 2008.

Returning to the Promela code for QRCU, the global variables are as shown in Figure F.11. This example uses locking, hence including `lock.h`. Both the number of readers and writers can be varied using the two `#define` statements, giving us not one but two ways to create combinatorial explosion. The `idx` variable controls which of the two elements of the `ctr` array will be used by readers, and the `readerprogress` variable allows to assertion to determine when all the readers are finished (since a QRCU update cannot be permitted to complete until all pre-existing readers have completed their QRCU read-side critical sections). The readerprogress array elements have values as follows, indicating the state of the corresponding reader:

1. 0: not yet started.
2. 1: within QRCU read-side critical section.
3. 2: finished with QRCU read-side critical section.

Finally, the `mutex` variable is used to serialize updaters’ slowpaths.

```
1 proctype qrcu_reader(byte me)
2 { int myidx;
3  do
4   :: 1 ->
5   myidx = idx;
6  atomic {
7    if :: ctr[myidx] > 0 ->
8      ctr[myidx]++;
9     else            -> skip
10   fi
11  }
12  od;
13  readerprogress[me] = 1;
14  atomic { ctr[myidx]-- }
15 }
```

Figure F.11: QRCU Global Variables

QRCU readers are modeled by the `qrcu_reader()` process shown in Figure F.12. A do-od loop spans lines 5-16, with a single guard of “1” on line 6 that makes it an infinite loop. Line 7 captures the current value of the
global index, and lines 8-15 atomically increment it (and break from the infinite loop) if its value was non-zero (atomic_inc_not_zero()). Line 17 marks entry into the RCU read-side critical section, and line 18 marks exit from this critical section, both lines for the benefit of the assert() statement that we shall encounter later. Line 19 atomically decrements the same counter that we incremented, thereby exiting the RCU read-side critical section.

```c
#define sum_unordered
atomic {
  do { 
    sum = ctr[0]; 
    i = 1; 
    break 
  } 
    i >= 0 
  od; 
  sum = sum + ctr[i]
```

Figure F.13: QRCU Unordered Summation

The C-preprocessor macro shown in Figure F.13 sums the pair of counters so as to emulate weak memory ordering. Lines 2-13 fetch one of the counters, and line 14 fetches the other of the pair and sums them. The atomic block consists of a single do-od statement. This do-od statement (spanning lines 3-12) is unusual in that it contains two unconditional branches with guards on lines 4 and 8, which causes Promela to non-deterministically choose one of the two (but again, the full state-space search causes Promela to eventually make all possible choices in each applicable situation). The first branch fetches the zero-th counter and sets i to 1 (so that line 14 will fetch the first counter), while the second branch does the opposite, fetching the first counter and setting i to 0 (so that line 14 will fetch the second counter).

**Quick Quiz F.3:** Is there a more straightforward way to code the do-od statement?

With the sum_unordered macro in place, we can now proceed to the update-side process shown in Figure. The update-side process repeats indefinitely, with the corresponding do-od loop ranging over lines 7-57. Each pass through the loop first snapshots the global readerprogress array into the local readerstart array on lines 12-21. This snapshot will be used for the assertion on line 53. Line 23 invokes sum_unordered, and then lines 24-27 re-invokes sum_unordered if the fastpath is potentially usable.

```c
proctype qrcu_updater(byte me) {
  int i;
  byte readerstart[N_QRCU_READERS];
  int sum;
  do { 
    if (sum <= 1 -> sum_unordered 
      i = 0; 
      do { 
        idx = !idx;
        atomic { ctr[!idx]++ 
          do { 
            :: ctr[!idx] > 0 -> skip
            :: ctr[!idx] == 0 -> break
          od;
          spin_unlock(mutex);
        fi;
      } 
        if (sum > 1 ->
          spin_lock(mutex);
          atomic { ctr[!idx]-- 
            do { 
              :: ctr[!idx] > 0 -> skip
              :: ctr[!idx] == 0 -> break
            od;
            spin_unlock(mutex);
          fi;
        fi;
      } 
        /\ Verify reader progress. */ 
        atomic ( 
          i = 0;
          sum = 0;
          do { 
            :: i < N_QRCU_READERS ->
              sum = sum + (readerstart[i] == 1 &&
                readerprogress[i] == 1));
            i++
            :: i >= N_QRCU_READERS ->
              assert(sum == 0);
        break
        od 
    } 
  od 
```

Figure F.14: QRCU Updater Process
Lines 28-40 execute the slowpath code if need be, with lines 30 and 38 acquiring and releasing the update-side lock, lines 31-33 flipping the index, and lines 34-37 waiting for all pre-existing readers to complete.

Lines 44-56 then compare the current values in the readerprogress array to those collected in the readerstart array, forcing an assertion failure should any readers that started before this update still be in progress.

Quick Quiz F.4: Why are there atomic blocks at lines 12-21 and lines 44-56, when the operations within those atomic blocks have no atomic implementation on any current production microprocessor?

Quick Quiz F.5: Is the re-summing of the counters on lines 24-27 really necessary?

```
1 init {
2   int i;
3
4   atomic {
5     ctr[idx] = 1;
6     ctr[!idx] = 0;
7     i = 0;
8   }
9   do
10      :: i < N_QRCU_READERS ->
11         readerprogress[i] = 0;
12         run qrcu_reader(i);
13      i++
14      :: i >= N_QRCU_READERS -> break
15   od;
16   i = 0;
17   do
18      :: i < N_QRCU_UPDATERS ->
19         run qrcu_updater(i);
20      i++
21      :: i >= N_QRCU_UPDATERS -> break
22   od
23 }
```

Figure F.15: QRCU Initialization Process

All that remains is the initialization block shown in Figure F.15. This block simply initializes the counter pair on lines 5-6, spawns the reader processes on lines 7-14, and spawns the updater processes on lines 15-21. This is all done within an atomic block to reduce state space.

F.6.1 Running the QRCU Example

To run the QRCU example, combine the code fragments in the previous section into a single file named qrcu.spin, and place the definitions for spin_lock() and spin_unlock() into a file named lock.h. Then use the following commands to build and run the QRCU model:

```
spin -a qrcu.spin
cc -DSAFETY -o pan pan.c
./pan
```

The resulting output shows that this model passes all of the cases shown in Table F.2. Now, it would be nice to run the case with three readers and three updaters, however, simple extrapolation indicates that this will require on the order of a terabyte of memory best case. So, what to do? Here are some possible approaches:

1. See whether a smaller number of readers and updaters suffice to prove the general case.
2. Manually construct a proof of correctness.
3. Use a more capable tool.
4. Divide and conquer.

The following sections discuss each of these approaches.

F.6.2 How Many Readers and Updaters Are Really Needed?

One approach is to look carefully at the Promela code for qrcu_updater() and notice that the only global state change is happening under the lock. Therefore, only one updater at a time can possibly be modifying state visible to either readers or other updaters. This means that any sequences of state changes can be carried out serially by a single updater due to the fact that Promela does a full state-space search. Therefore, at most two updaters are required: one to change state and a second to become confused.

The situation with the readers is less clear-cut, as each reader does only a single read-side critical section then terminates. It is possible to argue that the useful number
of readers is limited, due to the fact that the fastpath must see at most a zero and a one in the counters. This is a fruitful avenue of investigation, in fact, it leads to the full proof of correctness described in the next section.

F.6.3 Alternative Approach: Proof of Correctness

An informal proof [McK07b] follows:

1. For \texttt{synchronize_qrcu()} to exit too early, then by definition there must have been at least one reader present during \texttt{synchronize_qrcu()}’s full execution.

2. The counter corresponding to this reader will have been at least 1 during this time interval.

3. The \texttt{synchronize_qrcu()} code forces at least one of the counters to be at least 1 at all times.

4. Therefore, at any given point in time, either one of the counters will be at least 2, or both of the counters will be at least one.

5. However, the \texttt{synchronize_qrcu()} fastpath code can read only one of the counters at a given time. It is therefore possible for the fastpath code to fetch the first counter while zero, but to race with a counter flip so that the second counter is seen as one.

6. There can be at most one reader persisting through such a race condition, as otherwise the sum would be two or greater, which would cause the updater to take the slowpath.

7. But if the race occurs on the fastpath’s first read of the counters, and then again on its second read, there have to have been two counter flips.

8. Because a given updater flips the counter only once, and because the update-side lock prevents a pair of updaters from concurrently flipping the counters, the only way that the fastpath code can race with a flip twice is if the first updater completes.

9. But the first updater will not complete until after all pre-existing readers have completed.

10. Therefore, if the fastpath races with a counter flip twice in succession, all pre-existing readers must have completed, so that it is safe to take the fastpath.

Of course, not all parallel algorithms have such simple proofs. In such cases, it may be necessary to enlist more capable tools.

F.6.4 Alternative Approach: More Capable Tools

Although Promela and Spin are quite useful, much more capable tools are available, particularly for verifying hardware. This means that if it is possible to translate your algorithm to the hardware-design VHDL language, as it often will be for low-level parallel algorithms, then it is possible to apply these tools to your code (for example, this was done for the first realtime RCU algorithm). However, such tools can be quite expensive.

Although the advent of commodity multiprocessing might eventually result in powerful free-software model-checkers featuring fancy state-space-reduction capabilities, this does not help much in the here and now.

As an aside, there are Spin features that support approximate searches that require fixed amounts of memory, however, I have never been able to bring myself to trust approximations when verifying parallel algorithms.

Another approach might be to divide and conquer.

F.6.5 Alternative Approach: Divide and Conquer

It is often possible to break down a larger parallel algorithm into smaller pieces, which can then be proven separately. For example, a 10-billion-state model might be broken into a pair of 100,000-state models. Taking this approach not only makes it easier for tools such as Promela to verify your algorithms, it can also make your algorithms easier to understand.

F.7 Promela Parable: dynticks and Preemptible RCU

In early 2008, a preemptible variant of RCU was accepted into mainline Linux in support of real-time workloads, a variant similar to the RCU implementations in the -rt patchset [Mol05] since August 2005. Preemptible RCU is needed for real-time workloads because older RCU implementations disable preemption across RCU read-side critical sections, resulting in excessive real-time latencies.

However, one disadvantage of the older -rt implementation (described in Appendix D.4) was that each grace
period requires work to be done on each CPU, even if that CPU is in a low-power “dynticks-idle” state, and thus incapable of executing RCU read-side critical sections. The idea behind the dynticks-idle state is that idle CPUs should be physically powered down in order to conserve energy. In short, preemptible RCU can disable a valuable energy-conservation feature of recent Linux kernels. Although Josh Triplett and Paul McKenney had discussed some approaches for allowing CPUs to remain in low-power state throughout an RCU grace period (thus preserving the Linux kernel’s ability to conserve energy), matters did not come to a head until Steve Rostedt integrated a new dyntick implementation with preemptible RCU in the -rt patchset.

This combination caused one of Steve’s systems to hang on boot, so in October, Paul coded up a dynticks-friendly modification to preemptible RCU’s grace-period processing. Steve coded up `rcu_irq_enter()` and `rcu_irq_exit()` interfaces called from the `irq_enter()` and `irq_exit()` interrupt entry/exit functions. These `rcu_irq_enter()` and `rcu_irq_exit()` functions are needed to allow RCU to reliably handle situations where a dynticks-idle CPUs is momentarily powered up for an interrupt handler containing RCU read-side critical sections. With these changes in place, Steve’s system booted reliably, but Paul continued inspecting the code periodically on the assumption that we could not possibly have gotten the code right on the first try.

Paul reviewed the code repeatedly from October 2007 to February 2008, and almost always found at least one bug. In one case, Paul even coded and tested a fix before realizing that the bug was illusory, and in fact in all cases, the “bug” turned out to be illusionary.

Near the end of February, Paul grew tired of this game. He therefore decided to enlist the aid of Promela and spin [Hol03], as described in Appendix F. The following presents a series of seven increasingly realistic Promela models, the last of which passes, consuming about 40GB of main memory for the state space.

More important, Promela and Spin did find a very subtle bug for me!

Quick Quiz F.6: Yeah, that’s just great! Now, just what am I supposed to do if I don’t happen to have a machine with 40GB of main memory???

Still better would be to come up with a simpler and faster algorithm that has a smaller state space. Even better would be an algorithm so simple that its correctness was obvious to the casual observer!

Section F.7.1 gives an overview of preemptible RCU’s dynticks interface, Section F.7.2, and Section F.7.3 lists lessons (re)learned during this effort.

F.7.1 Introduction to Preemptible RCU and dynticks

The per-CPU dynticks_progress_counter variable is central to the interface between dynticks and preemptible RCU. This variable has an even value whenever the corresponding CPU is in dynticks-idle mode, and an odd value otherwise. A CPU exits dynticks-idle mode for the following three reasons:

1. to start running a task,
2. when entering the outermost of a possibly nested set of interrupt handlers, and
3. when entering an NMI handler.

Preemptible RCU’s grace-period machinery samples the value of the dynticks_progress_counter variable in order to determine when a dynticks-idle CPU may safely be ignored.

The following three sections give an overview of the task interface, the interrupt/NMI interface, and the use of the dynticks_progress_counter variable by the grace-period machinery.

F.7.1.1 Task Interface

When a given CPU enters dynticks-idle mode because it has no more tasks to run, it invokes `rcu_enter_nohz()`:

```c
static inline void rcu_enter_nohz(void)
{
  mb();
  __get_cpu_var(dynticks_progress_counter)++;
  WARN_ON(__get_cpu_var(dynticks_progress_counter) & 0x1);
}
```

This function simply increments dynticks_progress_counter and checks that the result is even, but first executing a memory barrier to ensure that any other CPU that sees the new value of dynticks_progress_counter will also see the completion of any prior RCU read-side critical sections.

Similarly, when a CPU that is in dynticks-idle mode prepares to start executing a newly runnable task, it invokes `rcu_exit_nohz`:
This function again increments `dynticks_progress_counter`, but follows it with a memory barrier to ensure that if any other CPU sees the result of any subsequent RCU read-side critical section, then that other CPU will also see the incremented value of `dynticks_progress_counter`. Finally, `rcu_exit_nohz()` checks that the result of the increment is an odd value.

The `rcu_enter_nohz()` and `rcu_exit_nohz()` functions handle the case where a CPU enters and exits dynticks-idle mode due to task execution, but does not handle interrupts, which are covered in the following section.

### F.7.1.2 Interrupt Interface

The `rcu_irq_enter()` and `rcu_irq_exit()` functions handle interrupt/NMI entry and exit, respectively. Of course, nested interrupts must also be properly accounted for. The possibility of nested interrupts is handled by a second per-CPU variable, `rcu_update_flag`, which is incremented upon entry to an interrupt or NMI handler (in `rcu_irq_enter()`) and is decremented upon exit (in `rcu_irq_exit()`). In addition, the pre-existing `in_interrupt()` primitive is used to distinguish between an outermost or a nested interrupt/NMI.

Interrupt entry is handled by the `rcu_irq_enter()` shown below:

```c
1 void rcu_irq_enter(void)
2 { 
3   int cpu = smp_processor_id();
4   
5   if (per_cpu(rcu_update_flag, cpu)) {
6     per_cpu(rcu_update_flag, cpu)++;
7     if (!in_interrupt() &&
8         (per_cpu(dynticks_progress_counter, cpu) & 0x1) == 0) {
9       per_cpu(dynticks_progress_counter, cpu)++;
10       smp_mb();
11       per_cpu(rcu_update_flag, cpu)++;
12     }
13   }
14 }
```

Line 3 fetches the current CPU’s number, as before. Line 5 checks to see if the `rcu_update_flag` is non-zero, returning immediately (via falling off the end of the function) if not. Otherwise, lines 6 through 12 come into play. Line 6 decrements `rcu_update_flag`, returning if the result is not zero. Line 8 verifies that we are indeed leaving the outermost level of nested interrupts, line 9 executes a memory barrier, line 10 increments `dynticks_progress_counter`, and lines 11 and 12 verify that this variable is now even. As with `rcu_exit_nohz()`, the memory barrier ensures that any other CPU that sees the increment of `dynticks_progress_counter` will also see the effects of an RCU read-side critical section in the interrupt handler (preceding the `rcu_irq_exit()` invocation).

These two sections have described how the `dynticks_progress_counter` variable is maintained during entry to and exit from dynticks-idle mode, both by tasks and by interrupts and NMIs. The following section describes how this variable is used by preemptible RCU’s grace-period machinery.
F.7.1.3 Grace-Period Interface

Of the four preemptible RCU grace-period states shown in Figure D.63 on page 265 in Appendix D.4, only the \texttt{rcu\_try\_flip\_waitack\_state()} and \texttt{rcu\_try\_flip\_waitmb\_state()} states need to wait for other CPUs to respond.

Of course, if a given CPU is in dynticks-idle state, we shouldn’t wait for it. Therefore, just before entering one of these two states, the preceding state takes a snapshot of each CPU’s \texttt{dynticks\_progress\_counter} variable, placing the snapshot in another per-CPU variable, \texttt{rcu\_dyntick\_snapshot}. This is accomplished by invoking \texttt{dyntick\_save\_progress\_counter}, shown below:

```c
1 static void dyntick_save_progress_counter(int cpu)
2 {
3     per_cpu(rcu_dyntick_snapshot, cpu) =
4         per_cpu(dynticks_progress_counter, cpu);
5 }
```

The \texttt{rcu\_try\_flip\_waitack\_state()} state invokes \texttt{rcu\_try\_flip\_waitack\_needed()}, shown below:

```c
1 static inline int
2     rcu_try_flip_waitack_needed(int cpu)
3 {
4     long curr;
5     long snap;
6     curr = per_cpu(dynticks_progress_counter, cpu);
7     snap = per_cpu(rcu_dyntick_snapshot, cpu);
8     smp_mb();
9     if ((curr == snap) && ((curr & 0x1) == 0))
10    return 0;
11   if ((curr - snap) > 2 || (snap & 0x1) == 0)
12    return 0;
13   return 1;
14 }
```

Lines 7 and 8 pick up current and snapshot versions of \texttt{dynticks\_progress\_counter}, respectively. The memory barrier on line ensures that the counter checks in the later \texttt{rcu\_try\_flip\_waitzero\_state} follow the fetches of these counters. Lines 10 and 11 return zero (meaning no communication with the specified CPU is required) if that CPU has remained in dynticks-idle state since the time that the snapshot was taken. Similarly, lines 12 and 13 return zero if that CPU was initially in dynticks-idle state or if it has completely passed through a dynticks-idle state. In both these cases, there is no way that that CPU could have retained the old value of the grace-period counter. If neither of these conditions hold, line 14 returns one, meaning that the CPU needs to explicitly respond.

For its part, the \texttt{rcu\_try\_flip\_waitmb\_state} state invokes \texttt{rcu\_try\_flip\_waitmb\_needed()}, shown below:

```c
1 static inline int
2     rcu_try_flip_waitmb_needed(int cpu)
3 {
4     long curr;
5     long snap;
6     curr = per_cpu(dynticks_progress_counter, cpu);
7     snap = per_cpu(rcu_dyntick_snapshot, cpu);
8     smp_mb();
9     if ((curr == snap) && ((curr & 0x1) == 0))
10    return 0;
11   if (curr != snap)
12    return 0;
13   return 1;
14 }
```

This is quite similar to \texttt{rcu\_try\_flip\_waitack\_needed}, the difference being in lines 12 and 13, because any transition either to or from dynticks-idle state executes the memory barrier needed by the \texttt{rcu\_try\_flip\_waitmb\_state()} state.

We now have seen all the code involved in the interface between RCU and the dynticks-idle state. The next section builds up the Promela model used to verify this code.

Quick Quiz F.9: Can you spot any bugs in any of the code in this section?

F.7.2 Validating Preemptible RCU and dynticks

This section develops a Promela model for the interface between dynticks and RCU step by step, with each of the following sections illustrating one step, starting with the process-level code, adding assertions, interrupts, and finally NMIs.

F.7.2.1 Basic Model

This section translates the process-level dynticks entry/exit code and the grace-period processing into Promela [Hol03]. We start with \texttt{rcu\_exit\_nohz()} and \texttt{rcu\_enter\_nohz()} from the 2.6.25-rc4 kernel, placing these in a single Promela process that models exiting and entering dynticks-idle mode in a loop as follows:

```c
1 proctype dyntick_nohz()
2 {
3 byte tmp;
4 byte i = 0;
5
6 do
7 :: i >= MAX\_DYNTICK\_LOOP\_NOHZ -> break;
8 :: i < MAX\_DYNTICK\_LOOP\_NOHZ ->
```
F.7. PROMELA PARABLE: DYNTICKS AND PREEMPTIBLE RCU

9   tmp = dynticks_progress_counter;
10  atomic {
11     dynticks_progress_counter = tmp + 1;
12     assert((dynticks_progress_counter & 1) == 1);
13   }
14  tmp = dynticks_progress_counter;
15  atomic {
16     dynticks_progress_counter = tmp + 1;
17     assert((dynticks_progress_counter & 1) == 0);
18   }
19  i++;
20 od;

Lines 6 and 20 define a loop. Line 7 exits the loop once the loop counter i has exceeded the limit MAX_DYNTICK_LOOP_NOHZ. Line 8 tells the loop construct to execute lines 9-19 for each pass through the loop. Because the conditionals on lines 7 and 8 are exclusive of each other, the normal Promela random selection of true conditions is disabled. Lines 9 and 11 model rcu_exit_nohz()'s non-atomic increment of dynticks_progress_counter, while line 12 models the WARN_ON(). The atomic construct simply reduces the Promela state space, given that the WARN_ON() is not strictly speaking part of the algorithm. Lines 14-18 similarly models the increment and WARN_ON() for rcu_enter_nohz(). Finally, line 19 increments the loop counter.

Each pass through the loop therefore models a CPU exiting dynticks-idle mode (for example, starting to execute a task), then re-entering dynticks-idle mode (for example, that same task blocking).

Quick Quiz F.10: Why isn’t the memory barrier in rcu_exit_nohz() and rcu_enter_nohz() modeled in Promela? ■

Quick Quiz F.11: Isn’t it a bit strange to model rcu_exit_nohz() followed by rcu_enter_nohz() modeled in Promela? Wouldn’t it be more natural to instead model entry before exit? ■

The next step is to model the interface to RCU’s grace-period processing. For this, we need to model dyntick_save_progress_counter(), rcu.try_flip_idle(), and its call to dyntick_save_progress_counter(), which takes a snapshot of the current CPU’s dynticks_progress_counter variable. These two lines are executed atomically to reduce state space.

Lines 10-22 model the relevant code in rcu_try_flip_idle() and its call to dyntick_save_progress_counter(), which takes a snapshot of the current CPU’s dynticks_progress_counter variable. These two lines are executed atomically to reduce state space.

Line 23 models a line from rcu_try_flip_waitack() and its call to rcu.try_flip_waitack_needed(). This loop is modeling the grace-period state machine waiting for a counter-flip acknowledgement from each CPU, but only that part that interacts with dynticks-idle CPUs.

Line 24-36 model the relevant code in rcu_try_flip_waitack() and its call to rcu_try_flip_waitack_needed(). This loop is modeling the grace-period state machine waiting for each CPU to execute a memory barrier, but again only that part that interacts with dynticks-idle CPUs.

Quick Quiz F.12: Wait a minute! In the Linux kernel, both dynticks_progress_counter and rcu_dyntick_snapshot are per-CPU variables. So
why are they instead being modeled as single global variables?

The resulting model (dyntickRCU-base.spin), when run with the runspin.sh script, generates 691 states and passes without errors, which is not at all surprising given that it completely lacks the assertions that could find failures. The next section therefore adds safety assertions.

F.7.2.2 Validating Safety

A safe RCU implementation must never permit a grace period to complete before the completion of any RCU readers that started before the start of the grace period. This is modeled by a grace_period_state variable that can take on three states as follows:

```c
#define GP_IDLE 0
#define GP_WAITING 1
#define GP_DONE 2
byte grace_period_state = GP_DONE;
```

The grace_period() process sets this variable as it progresses through the grace-period phases, as shown below:

```c
proctype grace_period()
{
byte curr;
byte snap;

grace_period_state = GP_IDLE;

atomic {
    printf("MDLN = %d\n", MAX_DYNTICK_LOOP_NOHZ);
    snap = dynticks_progress_counter;
    grace_period_state = GP_WAITING;
}

do 
:: 1 ->
atomic {
    curr = dynticks_progress_counter;
    if 
    :: (curr == snap) && ((curr & 1) == 0) -> break;
    :: (curr - snap) > 2 || (snap & 1) == 0 -> break;
    :: 1 -> skip;

    fi;
}

grace_period_state = GP_DONE;
}
```

Lines 6, 10, 25, 26, 29, and 44 update this variable (combining atomically with algorithmic operations where feasible) to allow the dyntick_nohz() process to verify the basic RCU safety property. The form of this verification is to assert that the value of the grace_period_state variable cannot jump from GP_IDLE to GP_DONE during a time period over which RCU readers could plausibly persist.

Quick Quiz F.13: Given there are a pair of back-to-back changes to grace_period_state on lines 25 and 26, how can we be sure that line 25’s changes won’t be lost?

The dyntick_nohz() Promela process implements this verification as shown below:

```c
proctype dyntick_nohz()
{
byte tmp;
byte i = 0;
bite old_gp_idle;

do 
:: i >= MAX_DYNTICK_LOOP_NOHZ -> break;
:: i < MAX_DYNTICK_LOOP_NOHZ ->
tmp = dynticks_progress_counter;

atomic {
    dynticks_progress_counter = tmp + 1;
    old_gp_idle = (grace_period_state == GP_IDLE);
    assert((dynticks_progress_counter & 1) == 1);
}

atomic {
    tmp = dynticks_progress_counter;
    assert(!old_gp_idle || grace_period_state != GP_DONE);
}

atomic {
    dynticks_progress_counter = tmp + 1;
    assert((dynticks_progress_counter & 1) == 0);
    i++;
}

od;
}
```

Line 13 sets a new old_gp_idle flag if the value of the grace_period_state variable is GP_IDLE at the beginning of task execution, and the assertion at lines 18 and 19 fire if the grace_period_state variable has advanced to GP_DONE during task execution, which would be illegal given that a single RCU read-side critical section could span the entire intervening time period.

The resulting model (dyntickRCU-base-s.spin), when run with the runspin.sh script,
generates 964 states and passes without errors, which is reassuring. That said, although safety is critically important, it is also quite important to avoid indefinitely stalling grace periods. The next section therefore covers verifying liveness.

F.7.2.3 Validating Liveness

Although liveness can be difficult to prove, there is a simple trick that applies here. The first step is to make `dyntick_nohz()` indicate that it is done via a `dyntick_nohz_done` variable, as shown on line 27 of the following:

```plaintext
1 proctype dyntick_nohz()
2 {
3 byte tmp;
4 byte i = 0;
5 bit old_gp_idle;
6
7 do
8 :: i >= MAX_DYNTICK_LOOP_NOHZ -> break;
9 :: i < MAX_DYNTICK_LOOP_NOHZ ->
10   tmp = dynticks_progress_counter;
11   atomic {
12     dynticks_progress_counter = tmp + 1;
13     old_gp_idle = (grace_period_state == GP_IDLE);
14     assert((dynticks_progress_counter & 1) == 1);
15   }
16   atomic {
17     tmp = dynticks_progress_counter;
18     assert(!old_gp_idle ||
19       grace_period_state != GP_DONE);
20   }
21   i++;
22 od;
23 dyntick_nohz_done = 1;
24 }
```

With this variable in place, we can add assertions to `grace_period()` to check for unnecessary blockage as follows:

```plaintext
1 proctype grace_period()
2 {
3 byte curr;
4 byte snap;
5 bit shouldexit;
6
7 grace_period_state = GP_IDLE;
8 atomic {
9   printf("MDLN = %d\n", MAX_DYNTICK_LOOP_NOHZ);
10   shouldexit = 0;
11   snap = dynticks_progress_counter;
12   grace_period_state = GP_WAITING;
13 }
14 do
15 :: 1 ->
16   atomic {
17     assert(!shouldexit);
18     shouldexit = dyntick_nohz_done;
19     curr = dynticks_progress_counter;
20     if
21       :: (curr == snap) && ((curr & 1) == 0) ->
22     break;
23       :: (curr - snap) > 2 || (snap & 1) == 0 ->
24     break;
25       :: else -> skip;
26   fi;
27   }
28 od;
29 grace_period_state = GP_DONE;
30 grace_period_state = GP_IDLE;
31 atomic {
32     shouldexit = 0;
33     snap = dynticks_progress_counter;
34     grace_period_state = GP_WAITING;
35 }
36 do
37 :: 1 ->
38   atomic {
39     assert(!shouldexit);
40     shouldexit = dyntick_nohz_done;
41     curr = dynticks_progress_counter;
42     if
43       :: (curr == snap) && ((curr & 1) == 0) ->
44     break;
45       :: (curr != snap) ->
46     break;
47       :: else -> skip;
48   fi;
49   }
50 od;
51 grace_period_state = GP_DONE;
52 }
```

We have added the `shouldexit` variable on line 5, which we initialize to zero on line 10. Line 17 asserts that `shouldexit` is not set, while line 18 sets `shouldexit` to the `dyntick_nohz_done` variable maintained by `dyntick_nohz()`. This assertion will therefore trigger if we attempt to take more than one pass through the wait-for-counter-flip-acknowledgement loop after `dyntick_nohz()` has completed execution. After all, if `dyntick_nohz()` is done, then there cannot be any more state changes to force us out of the loop, so going through twice in this state means an infinite loop, which in turn means no end to the grace period.

Lines 32, 39, and 40 operate in a similar manner for the second (memory-barrier) loop.

However, running this model (`dyntickRCU-base-sl-busted.spin`) results in failure, as line 23 is checking that the wrong variable is even. Upon failure, `spin` writes out a "trail" file (`dyntickRCU-base-sl-busted.spin.trail`) file, which records the sequence of states that lead to the failure. Use the `spin -t -p -g -l dyntickRCU-base-sl-busted.spin` command to cause `spin` to retrace this sequence of state, printing the statements executed and the values of variables (`dyntickRCU-base-sl-busted.spin.trail.txt`). Note that the line numbers do not match the listing above due to the fact that `spin` takes both functions in a
single file. However, the line numbers do match the full model (dyntickRCU-base-sl-busted.spin).

We see that the dyntick_nohz() process completed at step 34 (search for “34:”), but that the grace_period() process nonetheless failed to exit the loop. The value of curr is 6 (see step 35) and that the value of snap is 5 (see step 17). Therefore the first condition on line 21 above does not hold because curr != snap, and the second condition on line 23 does not hold either because snap is odd and because curr is only one greater than snap.

So one of these two conditions has to be incorrect. Referring to the comment block in rcu_try_flip_waitack_needed() for the first condition:

If the CPU remained in dynticks mode for the entire time and didn’t take any interrupts, NMIs, SMIs, or whatever, then it cannot be in the middle of an rcu_read_lock(), so the next rcu_read_lock() it executes must use the new value of the counter. So we can safely pretend that this CPU already acknowledged the counter.

The first condition does match this, because if curr == snap and if curr is even, then the corresponding CPU has been in dynticks-idle mode the entire time, as required. So let’s look at the comment block for the second condition:

If the CPU passed through or entered a dynticks idle phase with no active irq handlers, then, as above, we can safely pretend that this CPU already acknowledged the counter.

The first part of the condition is correct, because if curr and snap differ by two, there will be at least one even number in between, corresponding to having passed completely through a dynticks-idle phase. However, the second part of the condition corresponds to having started in dynticks-idle mode, not having finished in this mode. We therefore need to be testing curr rather than snap for being an even number.

The corrected C code is as follows:

```
1 static inline int
2 rcu_try_flip_waitack_needed(int cpu)
3 {
4    long curr;
5    long snap;
6    curr = per_cpu(dynticks_progress_counter, cpu);
7    snap = per_cpu(rcu_dyntick_snapshot, cpu);
8    smp_mb();
9    if ((curr - snap) >= 2 || (curr & 0x1) == 0)
10       return 0;
11    return 1;
12 }
```

Lines 10-13 can now be combined and simplified, resulting in the following. A similar simplification can be applied to rcu_try_flip_waitmb_needed.

```
1 static inline int
2 rcu_try_flip_waitack_needed(int cpu)
3 {
4    long curr;
5    long snap;
6    curr = per_cpu(dynticks_progress_counter, cpu);
7    snap = per_cpu(rcu_dyntick_snapshot, cpu);
8    smp_mb();
9    if ((curr - snap) >= 2 || (curr & 0x1) == 0)
10       return 0;
11    return 1;
12 }
```

Making the corresponding correction in the model (dyntickRCU-base-sl.spin) results in a correct verification with 661 states that passes without errors. However, it is worth noting that the first version of the liveness verification failed to catch this bug, due to a bug in the liveness verification itself. This liveness-verification bug was located by inserting an infinite loop in the grace_period() process, and noting that the liveness-verification code failed to detect this problem!

We have now successfully verified both safety and liveness conditions, but only for processes running and blocking. We also need to handle interrupts, a task taken up in the next section.

### F.7.2.4 Interrupts

There are a couple of ways to model interrupts in Promela:

1. using C-preprocessor tricks to insert the interrupt handler between each and every statement of the dynticks_nohz() process, or
2. modeling the interrupt handler with a separate process.

A bit of thought indicated that the second approach would have a smaller state space, though it requires that the interrupt handler somehow run atomically with respect to the dynticks_nohz() process, but not with respect to the grace_period() process.

Fortunately, it turns out that Promela permits you to branch out of atomic statements. This trick allows us to have the interrupt handler set a flag, and recode
dynticks_nohz() to atomically check this flag and execute only when the flag is not set. This can be accomplished with a C-preprocessor macro that takes a label and a Promela statement as follows:

```c
#define EXECUTE_MAINLINE(label, stmt) 
  atomic { 
    if :: in_dyntick_irq -> goto label; 
    :: else -> stmt; 
    fi; 
  } 
```

One might use this macro as follows:

```c
EXECUTE_MAINLINE(stmt1, 
  tmp = dynticks_progress_counter) 
```

Line 2 of the macro creates the specified statement label. Lines 3-8 are an atomic block that tests the in_dyntick_irq variable, and if this variable is set (indicating that the interrupt handler is active), branches out of the atomic block back to the label. Otherwise, line 6 executes the specified statement. The overall effect is that mainline execution stalls any time an interrupt is active, as required.

F.7.2.5 Validating Interrupt Handlers

The first step is to convert dynticks_nohz() to EXECUTE_MAINLINE() form, as follows:

```c
1 proctype dyntick_nohz() 
 2 
 3 byte tmp; 
 4 byte i = 0; 
 5 bit old_gp_idle; 
 6 
 7 do 
 8 :: i >= MAX_DYNTICK_LOOP_NOHZ -> break; 
 9 :: i < MAX_DYNTICK_LOOP_NOHZ -> 
 10 EXECUTE_MAINLINE(stmt1, 
 11 tmp = dynticks_progress_counter) 
 12 EXECUTE_MAINLINE(stmt2, 
 13 dynticks_progress_counter = tmp + 1; 
 14 old_gp_idle = (grace_period_state == GP_IDLE); 
 15 assert((dynticks_progress_counter & 1) == 1)) 
 16 EXECUTE_MAINLINE(stmt3, 
 17 tmp = dynticks_progress_counter; 
 18 assert(!old_gp_idle || grace_period_state != GP_DONE)) 
 19 EXECUTE_MAINLINE(stmt4, 
 20 dynticks_progress_counter = tmp + 1; 
 21 assert((dynticks_progress_counter & 1) == 0)) 
 22 i++; 
 23 od; 
 24 dynticks_nohz_done = 1; 
```

It is important to note that when a group of statements is passed to EXECUTE_MAINLINE(), as in lines 11-14, all statements in that group execute atomically.

Quick Quiz F.14: But what would you do if you needed the statements in a single EXECUTE_MAINLINE() group to execute non-atomically?

Quick Quiz F.15: But what if the dynticks_nohz() process had “if” or “do” statements with conditions, where the statement bodies of these constructs needed to execute non-atomically?

The next step is to write a dyntick_irq() process to model an interrupt handler:

```c
1 proctype dyntick_irq() 
 2 
 3 byte tmp; 
 4 byte i = 0; 
 5 bit old_gp_idle; 
 6 
 7 do 
 8 :: i >= MAX_DYNTICK_LOOP_IRQ -> break; 
 9 :: i < MAX_DYNTICK_LOOP_IRQ -> 
 10 in_dyntick_irq = 1; 
 11 if 
 12 :: rcu_update_flag > 0 -> 
 13 tmp = rcu_update_flag; 
 14 rcu_update_flag = tmp + 1; 
 15 :: else -> skip; 
 16 fi; 
 17 if 
 18 :: !in_interrupt && (dynticks_progress_counter & 1) == 0 -> 
 19 tmp = dynticks_progress_counter; 
 20 dynticks_progress_counter = tmp + 1; 
 21 tmp = rcu_update_flag; 
 22 rcu_update_flag = tmp + 1; 
 23 :: else -> skip; 
 24 fi; 
 25 tmp = in_interrupt; 
 26 in_interrupt = tmp + 1; 
 27 old_gp_idle = (grace_period_state == GP_IDLE); 
 28 assert(!(old_gp_idle || grace_period_state != GP_DONE)); 
 29 tmp = in_interrupt; 
 30 in_interrupt = tmp - 1; 
 31 if 
 32 :: rcu_update_flag != 0 -> 
 33 tmp = rcu_update_flag; 
 34 rcu_update_flag = tmp - 1; 
 35 if 
 36 :: rcu_update_flag == 0 -> 
 37 tmp = dynticks_progress_counter; 
 38 dynticks_progress_counter = tmp + 1; 
 39 :: else -> skip; 
 40 fi; 
 41 :: else -> skip; 
 42 fi; 
 43 atomic { 
 44 in_dyntick_irq = 0; 
 45 i++; 
 46 } 
 47 od; 
 48 dyntick_irq_done = 1; 
```

The loop from line 7-48 models up to MAX_DYNTICK_LOOP_IRQ interrupts, with lines 8 and 9 forming the loop condition and line 45 incrementing the control variable. Line 10 tells dyntick_nohz() that an interrupt handler is running, and line 45 tells dyntick_nohz() that this handler has completed.
Line 49 is used for liveness verification, much as is the corresponding line of `dyntick_nohz()`.

**Quick Quiz F.16:** Why are lines 45 and 46 (the `in_dyntick_irq = 0;` and the `i++;`) executed atomically?

Lines 11-25 model `rcu_irq_enter()`, and lines 26 and 27 model the relevant snippet of `__irq_enter()`. Lines 28 and 29 verify safety in much the same manner as do the corresponding lines of `dynticks_nohz()`. Lines 30 and 31 model the relevant snippet of `__irq_exit()`, and finally lines 32-43 model `rcu_irq_exit()`.

**Quick Quiz F.17:** What property of interrupts is this `dynticks_irq()` process unable to model?

The `grace_period()` process then becomes as follows:

```plaintext
1 proctype grace_period()
2 {
3 byte curr;
4 byte snap;
5 bit shouldexit;
6
7 grace_period_state = GP_IDLE;
8 atomic {
9 printf("MDLN = %d\n", MAX_DYNTICK_LOOP_NOHZ);
10 printf("MDLI = %d\n", MAX_DYNTICK_LOOP_IRQ);
11 shouldexit = 0;
12 snap = dynticks_progress_counter;
13 grace_period_state = GP_WAITING;
14 }
15 do
16 :: 1 ->
17 atomic {
18 assert(!shouldexit);
19 shouldexit = dyntick_nohz_done && dyntick_irq_done;
20 curr = dynticks_progress_counter;
21 if
22 :: (curr - snap) >= 2 || (curr & 1) == 0 ->
23 break;
24 :: else -> skip;
25 fi;
26 }
27 od;
28 grace_period_state = GP_DONE;
29 grace_period_state = GP_IDLE;
30 atomic {
31 shouldexit = 0;
32 snap = dynticks_progress_counter;
33 grace_period_state = GP_WAITING;
34 }
35 do
36 :: 1 ->
37 atomic {
38 assert(!shouldexit);
39 shouldexit = dyntick_nohz_done && dyntick_irq_done;
40 curr = dynticks_progress_counter;
41 if
42 :: (curr != snap) || (curr & 1) == 0 ->
43 break;
44 :: else -> skip;
45 fi;
46 }
47 od;
48 grace_period_state = GP_DONE;
49 }
```

The implementation of `grace_period()` is very similar to the earlier one. The only changes are the addition of line 10 to add the new interrupt-count parameter, changes to lines 19 and 39 to add the new `dyntick_irq_done` variable to the liveness checks, and of course the optimizations on lines 22 and 42.

This model (`dyntickRCU-irqnn-ssl.spin`) results in a correct verification with roughly half a million states, passing without errors. However, this version of the model does not handle nested interrupts. This topic is taken up in the next section.

### F.7.2.6 Validating Nested Interrupt Handlers

Nested interrupt handlers may be modeled by splitting the body of the loop in `dyntick_irq()` as follows:

```plaintext
1 proctype dyntick_irq()
2 {
3 byte tmp;
4 byte i = 0;
5 byte j = 0;
6 bit old_gp_idle;
7 bit outermost;
8
9 do
10 :: i >= MAX_DYNTICK_LOOP_IRQ &&
11 j >= MAX_DYNTICK_LOOP_IRQ -> break;
12 :: i < MAX_DYNTICK_LOOP_IRQ ->
13 atomic {
14 outermost = (in_dyntick_irq == 0);
15 in_dyntick_irq = i;
16 }
17 if
18 :: rcu_update_flag > 0 ->
19 tmp = rcu_update_flag;
20 rcu_update_flag = tmp + 1;
21 :: else -> skip;
22 fi;
23 if
24 :: !in_interrupt &&
25 (dynticks_progress_counter & 1) == 0 ->
26 tmp = dynticks_progress_counter;
27 dynticks_progress_counter = tmp + 1;
28 tmp = rcu_update_flag;
29 rcu_update_flag = tmp + 1;
30 :: else -> skip;
31 fi;
32 tmp = in_interrupt;
33 in_interrupt = tmp + 1;
34 atomic {
35 if
36 :: outermost ->
37 old_gp_idle = (grace_period_state == GP_IDLE);
38 :: else -> skip;
39 fi;
40 }
41 i++;
42 :: j < i ->
43 atomic {
44 if
45 :: j + 1 == i ->
46 assert((old_gp_idle ||
47 grace_period_state != GP_DONE));
48 :: else -> skip;
49 fi;
50 }
```
This is similar to the earlier dynticks_irq() process. It adds a second counter variable j on line 5, so that i counts entries to interrupt handlers and j counts exits. The outermost variable on line 7 helps determine when the grace_period_state variable needs to be sampled for the safety checks. The loop-exit check on lines 10 and 11 is updated to require that the specified number of interrupt handlers are exited as well as entered, and the increment of i is moved to line 41, which is the end of the interrupt-entry model. Lines 13-16 set the outermost variable to indicate whether this is the outermost of a set of nested interrupts and to set the in_dyntick_irq variable that is used by the dyntick_nohz() process. Lines 34-40 capture the state of the grace_period_state variable, but only when in the outermost interrupt handler.

Line 42 has the do-loop conditional for interrupt-exit modeling: as long as we have exited fewer interrupts than we have entered, it is legal to exit another interrupt. Lines 43-50 check the safety criterion, but only if we are exiting from the outermost interrupt level. Finally, lines 65-68 increment the interrupt-exit count j and, if this is the outermost interrupt level, clears in_dyntick_irq.

This model (dyntickRCU-irq-ssl.spin) results in a correct verification with a bit more than half a million states, passing without errors. However, this version of the model does not handle NMIs, which are taken up in the next section.

F.7.2.7 Validating NMI Handlers

We take the same general approach for NMIs as we do for interrupts, keeping in mind that NMIs do not nest. This results in a dyntick_nmi() process as follows:
dyntick_irq() to exclude dyntick_nmi():

```c
#define EXECUTE_IRQ(label, stmt) 
  1 label: skip; 
  2 atomic { 
  3   if :: in_dyntick_nmi -> goto label; \ 
  4   :: else -> stmt; \ 
  5   fi; \ 
  6 } \ 
```

It is further necessary to convert dyntick_irq() to EXECUTE_IRQ() as follows:

```c
1 proctype dyntick_irq() 
  2 { 
  3   byte tmp; 
  4   byte i = 0; 
  5   byte j = 0; 
  6   bit old_gp_idle; 
  7   bit outermost; 
  8 
  9   do 
 10     :: i >= MAX_DYNTICK_LOOP_IRQ && 
 11     j >= MAX_DYNTICK_LOOP_IRQ -> break; 
 12     :: i < MAX_DYNTICK_LOOP_IRQ -> 
 13       atomic { 
 14         outermost = (in_dyntick_irq == 0); 
 15         in_dyntick_irq = 1; 
 16       } 
 17     stmt1: skip; 
 18     atomic { 
 19       if :: in_dyntick_nmi -> goto stmt1; 
 20       :: !in_dyntick_nmi && rcu_update_flag -> 
 21         goto stmt1_then; 
 22       :: else -> goto stmt1_else; 
 23     fi; 
 24     } 
 25     stmt1_then: skip; 
 26     EXECUTE_IRQ(stmt1_1, tmp = rcu_update_flag) 
 27     EXECUTE_IRQ(stmt1_2, rcu_update_flag = tmp + 1) 
 28     stmt1_else: skip; 
 29     stmt2: skip; atomic { 
 30       if :: in_dyntick_nmi -> goto stmt2; 
 31       :: !in_dyntick_nmi && 
 32       !in_interrupt && 
 33       (dynticks_progress_counter & 1) == 0 -> 
 34       goto stmt2_then; 
 35     :: else -> goto stmt2_else; 
 36     fi; 
 37     stmt2_then: skip; 
 38     EXECUTE_IRQ(stmt2_1, tmp = dynticks_progress_counter) 
 39     EXECUTE_IRQ(stmt2_2, dynticks_progress_counter = tmp + 1) 
 40     stmt2_else: skip; 
 41     EXECUTE_IRQ(stmt3, tmp = in_interrupt) 
 42     EXECUTE_IRQ(stmt4, in_interrupt = tmp + 1) 
 43     stmt5: skip; 
 44     atomic { 
 45       if :: in_dyntick_nmi -> goto stmt4; 
 46       :: !in_dyntick_nmi && outermost -> 
 47       old_gp_idle = (grace_period_state == GP_IDLE); 
 48     :: else -> skip; 
 49     fi; 
 50     stmt6: skip; 
 51     atomic { 
 52       if :: in_dyntick_nmi -> goto stmt6; 
 53       :: !in_dyntick_nmi && j + 1 == i -> 
 54       assert((old_gp_idle || 
 55       grace_period_state != GP_DONE)); 
 56     :: else -> skip; 
 57     fi; 
 58     stmt7: skip; 
 59     EXECUTE_IRQ(stmt7, tmp = in_interrupt); 
 60     EXECUTE_IRQ(stmt8, in_interrupt = tmp - 1); 
 61     stmt9: skip; 
 62     atomic { 
 63       if :: in_dyntick_nmi -> goto stmt9; 
 64       :: !in_dyntick_nmi && rcu_update_flag != 0 -> 
 65       goto stmt9_then; 
 66     :: else -> goto stmt9_else; 
 67     fi; 
 68     stmt9_then: skip; 
 69     EXECUTE_IRQ(stmt9_1, tmp = rcu_update_flag) 
 70     EXECUTE_IRQ(stmt9_2, rcu_update_flag = tmp - 1) 
 71     stmt9_3: skip; 
 72     atomic { 
 73       if :: in_dyntick_nmi -> goto stmt9_3; 
 74       :: !in_dyntick_nmi && rcu_update_flag == 0 -> 
 75       goto stmt9_3_then; 
 76     :: else -> goto stmt9_3_else; 
 77     fi; 
 78     stmt9_3_then: skip; 
 79     EXECUTE_IRQ(stmt9_3_1, 
 80     EXECUTE_IRQ(stmt9_3_2, 
 81     dynticks_progress_counter = tmp + 1) 
 82     stmt9_3_else: skip; 
 83     stmt9_else: skip; 
 84     atomic { 
 85       if :: in_dyntick_nmi -> goto stmt9_3; 
 86       :: !in_dyntick_nmi && rcu_update_flag == 0 -> 
 87       goto stmt9_3_then; 
 88       :: else -> goto stmt9_3_else; 
 89     fi; 
 90     stmt9_3_then: skip; 
 91     EXECUTE_IRQ(stmt9_3_1, 
 92     EXECUTE_IRQ(stmt9_3_2, 
 93     dynticks_progress_counter = tmp + 1) 
 94     stmt9_3_else: skip; 
 95     stmt9_else: skip; 
 96     atomic { 
 97       if :: in_dyntick_nmi -> goto stmt9_3; 
 98       :: !in_dyntick_nmi && rcu_update_flag == 0 -> 
 99       goto stmt9_3_then; 
100     :: else -> goto stmt9_3_else; 
101     fi; 
102     stmt9_3_then: skip; 
103     EXECUTE_IRQ(stmt9_3_1, 
104     EXECUTE_IRQ(stmt9_3_2, 
105     dynticks_progress_counter = tmp + 1) 
106     dyntick_irq_done = 1; 
107 } 
```

Note that we have open-coded the “if” statements (for example, lines 17-29). In addition, statements that process strictly local state (such as line 58) need not exclude dyntick_nmi().

Finally, grace_period() requires only a few changes:

```c
1 proctype grace_period() 
  2 { 
  3   byte curr; 
  4   byte snap; 
  5   bit shouldexit; 
  6 
  7   grace_period_state = GP_IDLE; 
  8   atomic { 
  9     printf("MDLN = %d
", MAX_DYNTICK_LOOP_NOHZ); 
10     printf("MDLI = %d
", MAX_DYNTICK_LOOP_IRQ); 
11     printf("MDLN = %d
", MAX_DYNTICK_LOOP_NMI); 
12     shouldexit = 0; 
13     snap = dynticks_progress_counter; 
14     grace_period_state = GP_WAITING; 
15 } 
```
F.8. SIMPLICITY AVOIDS FORMAL VERIFICATION

```c
16 do
17 :: 1 ->
18 atomic {
19 assert(!shouldexit);
20 shouldexit = dyntick_nohz_done &&
21 dyntick_irq_done &&
22 dyntick_nmi_done;
23 curr = dynticks_progress_counter;
24 if
25 :: (curr - snap) >= 2 || (curr & 1) == 0 ->
26 break;
27 :: else -> skip;
28 fi;
29 }
30 od;
31 grace_period_state = GP_DONE;
32 grace_period_state = GP_IDLE;
33 atomic {
34 shouldexit = 0;
35 snap = dynticks_progress_counter;
36 grace_period_state = GP_WAITING;
37 }
38 do
39 :: 1 ->
40 atomic {
41 assert(!shouldexit);
42 shouldexit = dyntick_nohz_done &&
43 dyntick_irq_done &&
44 dyntick_nmi_done;
45 curr = dynticks_progress_counter;
46 if
47 :: (curr != snap) || (curr & 1) == 0 ->
48 break;
49 :: else -> skip;
50 fi;
51 }
52 od;
53 grace_period_state = GP_DONE;
```

Figure F.16: Memory-BARRIER Fix Patch

```c
- if ((curr - snap) > 2 || (curr & 0x1) == 0)
+ if ((curr - snap) > 2 || (curr & 0x1) == 0)
```

Figure F.17: Variable-NAME-Typo Fix Patch

3. Validate your code early, often, and up to the point of destruction. This effort located one subtle bug in `rcu_try_flip_waitack_needed()` that would have been quite difficult to test or debug, as shown by the patch in Figure F.17.

4. Always verify your verification code. The usual way to do this is to insert a deliberate bug and verify that the verification code catches it. Of course, if the verification code fails to catch this bug, you may also need to verify the bug itself, and so on, recursing infinitely. However, if you find yourself in this position, getting a good night’s sleep can be an extremely effective debugging technique.

5. Use of atomic instructions can simplify verification. Unfortunately, use of the `cmpxchg` atomic instruction would also slow down the critical irq fastpath, so they are not appropriate in this case.

6. The need for complex formal verification often indicates a need to re-think your design. In fact the design verified in this section turns out to have a much simpler solution, which is presented in the next section.

F.8 Simplicity Avoids Formal Verification

The complexity of the dynticks interface for preemptible RCU is primarily due to the fact that both irqs and NMIs use the same code path and the same state variables. This

We have added the `printf()` for the new `MAX_DYNTICK_LOOP_NMI` parameter on line 11 and added `dyntick_nmi_done` to the `shouldexit` assignments on lines 22 and 44.

The model (`dyntickRCU-irq-nmi-ssl.spin`) results in a correct verification with several hundred million states, passing without errors.

Quick Quiz F.18: Does Paul always write his code in this painfully incremental manner?

F.7.3 Lessons (Re)Learned

This effort provided some lessons (re)learned:

1. Promela and spin can verify interrupt/NMI-handler interactions.
2. Documenting code can help locate bugs. In this case, the documentation effort located a misplaced memory barrier in `rcu_enter_nohz()` and `rcu_exit_nohz()`, as shown by the patch in Figure F.16.
leads to the notion of providing separate code paths and variables for irqs and NMI, as has been done for hierarchical RCU [McK08a] as indirectly suggested by Manfred Spraul [Spr08b].

F.8.1 State Variables for Simplified Dynticks Interface

Figure F.18 shows the new per-CPU state variables. These variables are grouped into structs to allow multiple independent RCU implementations (e.g., rcu and rcu_bh) to conveniently and efficiently share dynticks state. In what follows, they can be thought of as independent per-CPU variables.

The `dynticks_nesting`, `dynticks`, and `dynticks_snap` variables are for the irq code paths, and the `dynticks_nmi` and `dynticks_nmi_snap` variables are for the NMI code paths, although the NMI code path will also reference (but not modify) the `dynticks_nesting` variable. These variables are used as follows:

• `dynticks_nesting`: This counts the number of reasons that the corresponding CPU should be monitored for RCU read-side critical sections. If the CPU is in dynticks-idle mode, then this counts the irq nesting level, otherwise it is one greater than the irq nesting level.

• `dynticks`: This counter’s value is even if the corresponding CPU is in dynticks-idle mode and there are no irq handlers currently running on that CPU, otherwise the counter’s value is odd. In other words, if this counter’s value is odd, then the corresponding CPU might be in an RCU read-side critical section.

• `dynticks_nmi`: This counter’s value is odd if the corresponding CPU is in an NMI handler, but only if the NMI arrived while this CPU was in dynticks-idle mode with no irq handlers running. Otherwise, the counter’s value will be even.

• `dynticks_snap`: This will be a snapshot of the `dynticks` counter, but only if the current RCU grace period has extended for too long a duration.

• `dynticks_nmi_snap`: This will be a snapshot of the `dynticks_nmi` counter, but again only if the current RCU grace period has extended for too long a duration.

If both `dynticks` and `dynticks_nmi` have taken on an even value during a given time interval, then the corresponding CPU has passed through a quiescent state during that interval.

Quick Quiz F.19: But what happens if an NMI handler starts running before an irq handler completes, and if that NMI handler continues running until a second irq handler starts?

F.8.2 Entering and Leaving Dynticks-Idle Mode

Figure F.19 shows the `rcu_enter_nohz()` and `rcu_exit_nohz()`, which enter and exit dynticks-idle mode, also known as “nohz” mode. These two functions are invoked from process context.
F.8. SIMPLICITY AVOIDS FORMAL VERIFICATION

1 void rcu_nmi_enter(void)
2 {
3     struct rcu_dynticks *rdtp;
4     rdtp = &__get_cpu_var(rcu_dynticks);
5     if (rdtp->dynticks & 0x1)
6         return;
7     rdtp->dynticks_nmi++;
8     WARN_ON_RATELIMIT(!(rdtp->dynticks_nmi & 0x1), &rcu_rs);
9     smp_mb();
10 }
11
12 void rcu_nmi_exit(void)
13 {
14     struct rcu_dynticks *rdtp;
15     rdtp = &__get_cpu_var(rcu_dynticks);
16     if (rdtp->dynticks & 0x1)
17         return;
18     smp_mb();
19     rdtp->dynticks_nmi++;
20     WARN_ON_RATELIMIT(rdtp->dynticks_nmi & 0x1, &rcu_rs);
21     }
APPENDIX F. FORMAL VERIFICATION

1 static int
dyntick_save_progress_counter(struct rcu_data *rdp)
{
  int ret;
  int snap;
  int snap_nmi;

  snap = rdp->dynticks->dynticks;
  snap_nmi = rdp->dynticks->dynticks_nmi;
  smp_mb();
  rdp->dynticks_snap = snap;
  rdp->dynticks_nmi_snap = snap_nmi;
  ret = ((snap & 0x1) == 0) && ((snap_nmi & 0x1) == 0);
  if (ret)
    rdp->dynticks_fqs++;
  return ret;
}

Figure F.22: Saving Dyntick Progress Counters

so.

F.8.5 Checking For Dynticks Quiescent States

Figure F.22 shows dyntick_save_progress_counter(), which takes a snapshot of the specified CPU’s dynticks and dynticks_nmi counters. Lines 8 and 9 snapshot these two variables to locals, line 10 executes a memory barrier to pair with the memory barriers in the functions in Figures F.19, F.20, and F.21. Lines 11 and 12 record the snapshots for later calls to rcu_implicit_dynticks_qs, and 13 checks to see if the CPU is in dynticks-idle mode with neither irqs nor NMI’s in progress (in other words, both snapshots have even values), hence in an extended quiescent state. If so, lines 14 and 15 count this event, and line 16 returns true if the CPU was in a quiescent state.

Figure F.23 shows dyntick_save_progress_counter(), which is called to check whether a CPU has entered dynticks-idle mode subsequent to a call to dynticks_save_progress_counter(). Lines 9 and 11 take new snapshots of the corresponding CPU’s dynticks and dynticks_nmi variables, while lines 10 and 12 retrieve the snapshots saved earlier by dynticks_save_progress_counter(). Line 13 then executes a memory barrier to pair with the memory barriers in the functions in Figures F.19, F.20, and F.21. Lines 14 and 15 then check to see if the CPU is either currently in a quiescent state (curr and curr_nmi having even values) or has passed through a quiescent state since the last call to dynticks_save_progress_counter() (the values of dynticks and dynticks_nmi having changed). If these checks confirm that the CPU has passed through a dynticks-idle quiescent state, then line 16 counts that fact and line 16 returns an indication of this fact. Either way, line 19 checks for race conditions that can result in RCU waiting for a CPU that is offline.

Quick Quiz F.20: This is still pretty complicated. Why not just have a cpu_mask_t that has a bit set for each CPU that is in dynticks-idle mode, clearing the bit when entering an irq or NMI handler, and setting it upon exit?

F.8.6 Discussion

A slight shift in viewpoint resulted in a substantial simplification of the dynticks interface for RCU. The key change leading to this simplification was minimizing of sharing between irq and NMI contexts. The only sharing in this simplified interface is references from NMI context to irq variables (the dynticks variable). This type of sharing is benign, because the NMI functions never update this variable, so that its value remains constant through the lifetime of the NMI handler. This limitation of sharing allows the individual functions to be understood one at a time, in happy contrast to the situation described in Section F.7, where an NMI might change shared state at any point during execution of the irq functions.

Verification can be a good thing, but simplicity is even better.
F.9 Summary

Promela is a very powerful tool for validating small parallel algorithms. It is a useful tool in the parallel kernel hacker’s toolbox, but it should not be the only tool. The QRCU experience is a case in point: given the Promela validation, the proof of correctness, and several rcutorture runs, I now feel reasonably confident in the QRCU algorithm and its implementation. But I would certainly not feel so confident given only one of the three!

Nevertheless, if your code is so complex that you find yourself relying too heavily on validation tools, you should carefully rethink your design. For example, a complex implementation of the dynticks interface for pre-emptible RCU that was presented in Section F.7 turned out to have a much simpler alternative implementation, as discussed in Section F.8. All else being equal, a simpler implementation is much better than a mechanical proof for a complex implementation!
Appendix G

Answers to Quick Quizzes

G.1 Introduction

Quick Quiz 1.1:
Come on now!!! Parallel programming has been known to be exceedingly hard for many decades. You seem to be hinting that it is not so hard. What sort of game are you playing?

Answer:
If you really believe that parallel programming is exceedingly hard, then you should have a ready answer to the question “Why is parallel programming hard?” One could list any number of reasons, ranging from deadlocks to race conditions to testing coverage, but the real answer is that it is not really all that hard. After all, if parallel programming was really so horribly difficult, how could a large number of open-source projects, ranging from Apache to MySQL to the Linux kernel, have managed to master it?

A better question might be: “Why is parallel programming perceived to be so difficult?” To see the answer, let’s go back to the year 1991. Paul McKenney was walking across the parking lot to Sequent’s benchmarking center carrying six dual-80486 Sequent Symmetry CPU boards, when he suddenly realized that he was carrying several times the price of the house he had just purchased.¹ This high cost of parallel systems meant that parallel programming was restricted to a privileged few who worked for an employer who either manufactured or could afford to purchase machines costing upwards of $100,000 — in 1991 dollars US.

In contrast, in 2006, Paul finds himself typing these words on a dual-core x86 laptop. Unlike the dual-80486 CPU boards, this laptop also contains 2GB of main memory, a 60GB disk drive, a display, Ethernet, USB ports, wireless, and Bluetooth. And the laptop is more than an order of magnitude cheaper than even one of those dual-80486 CPU boards, even before taking inflation into account.

Parallel systems have truly arrived. They are no longer the sole domain of a privileged few, but something available to almost everyone.

The earlier restricted availability of parallel hardware is the real reason that parallel programming is considered so difficult. After all, it is quite difficult to learn to program even the simplest machine if you have no access to it. Since the age of rare and expensive parallel machines is for the most part behind us, the age during which parallel programming is perceived to be mind-crushingly difficult is coming to a close.²

Quick Quiz 1.2:
How could parallel programming ever be as easy as sequential programming?

Answer:
It depends on the programming environment. SQL [Int92] is an underappreciated success story, as it permits programmers who know nothing about parallelism to keep a large parallel system productively busy. We can expect

¹ Yes, this sudden realization did cause him to walk quite a bit more carefully. Why do you ask?

² Parallel programming is in some ways more difficult than sequential programming, for example, parallel validation is more difficult. But no longer mind-crushingly difficult.
more variations on this theme as parallel computers continue to become cheaper and more readily available. For example, one possible contender in the scientific and technical computing arena is MATLAB®P, which is an attempt to automatically parallelize common matrix operations.

Finally, on Linux and UNIX systems, consider the following shell command:

```
get_input | grep "interesting" | sort
```

This shell pipeline runs the `get_input`, `grep`, and `sort` processes in parallel. There, that wasn’t so hard, now was it?

**Quick Quiz 1.3:**
Oh, really???. What about correctness, maintainability, robustness, and so on? ■

**Answer:**
These are important goals, but they are just as important for sequential programs as they are for parallel programs. Therefore, important though they are, they do not belong on a list specific to parallel programming.

**Quick Quiz 1.4:**
And if correctness, maintainability, and robustness don’t make the list, why do productivity and generality? ■

**Answer:**
Given that parallel programming is perceived to be much harder than is sequential programming, productivity is tantamount and therefore must not be omitted. Furthermore, high-productivity parallel-programming environments such as SQL have been special purpose, hence generality must also be added to the list.

**Quick Quiz 1.5:**
Given that parallel programs are much harder to prove correct than are sequential programs, again, shouldn’t correctness really be on the list? ■

**Answer:**
From an engineering standpoint, the difficulty in proving correctness, either formally or informally, would be important insofar as it impacts the primary goal of productivity. So, in cases where correctness proofs are important, they are subsumed under the “productivity” rubric.

**Quick Quiz 1.6:**
What about just having fun? ■

**Answer:**
Having fun is important as well, but, unless you are a hobbyist, would not normally be a primary goal. On the other hand, if you are a hobbyist, go wild!

**Quick Quiz 1.7:**
Are there no cases where parallel programming is about something other than performance? ■

**Answer:**
There are certainly cases where the problem to be solved is inherently parallel, for example, Monte Carlo methods and some numerical computations. Even in these cases, however, there will be some amount of extra work managing the parallelism.

**Quick Quiz 1.8:**
Why all this prattling on about non-technical issues??? And not just any non-technical issue, but productivity of all things? Who cares? ■

**Answer:**
If you are a pure hobbyist, perhaps you don’t need to care. But even pure hobbyists will often care about how much they can get done, and how quickly. After all, the most popular hobbyist tools are usually those that are the best suited for the job, and an important part of the definition of “best suited” involves productivity. And if someone is paying you to write parallel code, they will very likely care deeply about your productivity. And if the person paying you cares about something, you would be most wise to pay at least some attention to it!

Besides, if you really didn’t care about productivity, you would be doing it by hand rather than using a computer!

**Quick Quiz 1.9:**
Given how cheap parallel hardware has become, how can anyone afford to pay people to program it? ■
Answer:
There are a number of answers to this question:

1. Given a large computational cluster of parallel machines, the aggregate cost of the cluster can easily justify substantial developer effort, because the development cost can be spread over the large number of machines.

2. Popular software that is run by tens of millions of users can easily justify substantial developer effort, as the cost of this development can be spread over the tens of millions of users. Note that this includes things like kernels and system libraries.

3. If the low-cost parallel machine is controlling the operation of a valuable piece of equipment, then the cost of this piece of equipment might easily justify substantial developer effort.

4. If the software for the low-cost parallel produces an extremely valuable result (e.g., mineral exploration), then the valuable result might again justify substantial developer cost.

5. Safety-critical systems protect lives, which can clearly justify very large developer effort.

6. Hobbyists and researchers might seek knowledge, experience, fun, or glory rather than mere money.

So it is not the case that the decreasing cost of hardware renders software worthless, but rather that it is no longer possible to “hide” the cost of software development within the cost of the hardware, at least not unless there are extremely large quantities of hardware.

Quick Quiz 1.10:
This is a ridiculously unachievable ideal! Why not focus on something that is achievable in practice?

Answer:
This is eminently achievable. The cellphone is a computer that can be used to make phone calls and to send and receive text messages with little or no programming or configuration on the part of the end user. This might seem to be a trivial example at first glance, but if you consider it carefully you will see that it is both simple and profound. When we are willing to sacrifice generality, we can achieve truly astounding increases in productivity. Those who cling to generality will therefore fail to set the productivity bar high enough to succeed in production environments.

Quick Quiz 1.11:
What other bottlenecks might prevent additional CPUs from providing additional performance?

Answer:
There are any number of potential bottlenecks:

1. Main memory. If a single thread consumes all available memory, additional threads will simply page themselves silly.

2. Cache. If a single thread’s cache footprint completely fills any shared CPU cache(s), then adding more threads will simply thrash the affected caches.

3. Memory bandwidth. If a single thread consumes all available memory bandwidth, additional threads will simply result in additional queuing on the system interconnect.

4. I/O bandwidth. If a single thread is I/O bound, adding more threads will simply result in them all waiting in line for the affected I/O resource.

Specific hardware systems may have any number of additional bottlenecks.

Quick Quiz 1.12:
What besides CPU cache capacity might require limiting the number of concurrent threads?

Answer:
There are any number of potential limits on the number of threads:

1. Main memory. Each thread consumes some memory (for its stack if nothing else), so that excessive numbers of threads can exhaust memory, resulting in excessive paging or memory-allocation failures.

2. I/O bandwidth. If each thread initiates a given amount of mass-storage I/O or networking traffic, excessive numbers of threads can result in excessive I/O queuing delays, again degrading performance. Some networking protocols may be subject to time-outs or other failures if there are so many threads
that networking events cannot be responded to in a timely fashion.

3. Synchronization overhead. For many synchronization protocols, excessive numbers of threads can result in excessive spinning, blocking, or rollbacks, thus degrading performance.

Specific applications and platforms may have any number of additional limiting factors.

Quick Quiz 1.13:
Are there any other obstacles to parallel programming?

Answer:
There are a great many other potential obstacles to parallel programming. Here are a few of them:

1. The only known algorithms for a given project might be inherently sequential in nature. In this case, either avoid parallel programming (there being no law saying that your project has to run in parallel) or invent a new parallel algorithm.

2. The project allows binary-only plugins that share the same address space, such that no one developer has access to all of the source code for the project. Because many parallel bugs, including deadlocks, are global in nature, such binary-only plugins pose a severe challenge to current software development methodologies. This might well change, but for the time being, all developers of parallel code sharing a given address space need to be able to see all of the code running in that address space.

3. The project contains heavily used APIs that were designed without regard to parallelism. Some of the more ornate features of the System V message-queue API form a case in point. Of course, if your project has been around for a few decades, and if its developers did not have access to parallel hardware, your project undoubtedly has at least its share of such APIs.

4. The project was implemented without regard to parallelism. Given that there are a great many techniques that work extremely well in a sequential environment, but that fail miserably in parallel environments, if your project ran only on sequential hardware for most of its lifetime, then your project undoubtedly has at least its share of parallel-unfriendly code.

5. The project was implemented without regard to good software-development practice. The cruel truth is that shared-memory parallel environments are often much less forgiving of sloppy development practices than are sequential environments. You may be well-served to clean up the existing design and code prior to attempting parallelization.

6. The people who originally did the development on your project have since moved on, and the people remaining, while well able to maintain it or add small features, are unable to make “big animal” changes. In this case, unless you can work out a very simple way to parallelize your project, you will probably be best off leaving it sequential. That said, there are a number of simple approaches that you might use to parallelize your project, including running multiple instances of it, using a parallel implementation of some heavily used library function, or making use of some other parallel project, such as a database.

One can argue that many of these obstacles are non-technical in nature, but that does not make them any less real. In short, parallelization can be a large and complex effort. As with any large and complex effort, it makes sense to do your homework beforehand.

Quick Quiz 1.14:
Where are the answers to the Quick Quizzes found?

Answer:
In Appendix G starting on page 309.

Hey, I thought I owed you an easy one!

Quick Quiz 1.15:
Some of the Quick Quiz questions seem to be from the viewpoint of the reader rather than the author. Is that really the intent?

Answer:
Indeed it is! Many are modeled after Paul—just ask anyone who has had the misfortune of being assigned to teach him. Others are quite similar to actual questions that have been asked during conference presentations and lectures covering the material in this book. Still others are from the viewpoint of the author.
Quick Quiz 1.16:
These Quick Quizzes just are not my cup of tea. What do you recommend?

Answer:
There are a number of alternatives available to you:

1. Just ignore the Quick Quizzes and read the rest of the book. You might miss out on the interesting material in some of the Quick Quizzes, but the rest of the book has lots of good material as well.

2. If you prefer a more academic and rigorous treatment of parallel programming, you might like Herlihy’s and Shavit’s textbook [HS08]. This book starts with an interesting combination of low-level primitives at high levels of abstraction from the hardware, and works its way through locking and simple data structures including lists, queues, hash tables, and counters, culminating with transactional memory.

3. If you would like an academic treatment of parallel programming from a programming-language-pragmatics viewpoint, you might be interested in the concurrency chapter from Scott’s textbook [Sco06] on programming languages.

4. If you are interested in an object-oriented patternist treatment of parallel programming focussing on C++, you might try Volumes 2 and 4 of Schmidt’s POSA series [SSRB00, BHS07]. Volume 4 in particular has some interesting chapters applying this work to a warehouse application. The realism of this example is attested to by the section entitled “Partitioning the Big Ball of Mud”, wherein the problems inherent in parallelism often take a back seat to the problems inherent in getting one’s head around a real-world application.

5. If your primary focus is scientific and technical computing, and you prefer a patternist approach, you might try Mattson et al.’s textbook [MSM05]. It covers Java, C/C++, OpenMP, and MPI. Its patterns are admirably focused first on design, then on implementation.

6. If you are interested in POSIX Threads, you might take a look at David R. Butenhof’s book [But97].

7. If you are interested in C++, but in a Windows environment, you might try Herb Sutter’s “Effective Concurrency” series in Dr. Dobbs Journal [Sut08].

This series does a reasonable job of presenting a commonsense approach to parallelism.

8. If you want to try out Intel Threading Building Blocks, then perhaps James Reinders’s book [Rei07] is what you are looking for.

9. Finally, those preferring to work in Java might be well-served by Doug Lea’s textbooks [Lea97, GPB+07].

In contrast, this book meshes real-world machines with real-world algorithms. If your sole goal is to find an optimal parallel queue, you might be better served by one of the above books. However, if you are interested in principles of parallel design that allow multiple such queues to operate in parallel, read on!

G.2 Hardware and its Habits

Quick Quiz 2.1:
Why should parallel programmers bother learning low-level properties of the hardware? Wouldn’t it be easier, better, and more general to remain at a higher level of abstraction?

Answer:
It might well be easier to ignore the detailed properties of the hardware, but in most cases it would be quite foolish to do so. If you accept that the only purpose of parallelism is to increase performance, and if you further accept that performance depends on detailed properties of the hardware, then it logically follows that parallel programmers are going to need to know at least a few hardware properties.

This is the case in most engineering disciplines. Would you want to use a bridge designed by an engineer who did not understand the properties of the concrete and steel making up that bridge? If not, why would you expect a parallel programmer to be able to develop competent parallel software without at least some understanding of the underlying hardware?

Quick Quiz 2.2:
What types of machines would allow atomic operations on multiple data elements?
Answer:
One answer to this question is that it is often possible to pack multiple elements of data into a single machine word, which can then be manipulated atomically.

A more trendy answer would be machines supporting transactional memory [Lom77]. However, such machines are still (as of 2008) research curiosities. The jury is still out on the applicability of transactional memory [MMW07, PW07, RHP+07].

Quick Quiz 2.3:
This is a simplified sequence of events? How could it possibly be any more complex?

Answer:
This sequence ignored a number of possible complications, including:

1. Other CPUs might be concurrently attempting to perform CAS operations involving this same cacheline.
2. The cacheline might have been replicated read-only in several CPUs’ caches, in which case, it would need to be flushed from their caches.
3. CPU 7 might have been operating on the cache line when the request for it arrived, in which case CPU 7 would need to hold of the request until its own operation completed.
4. CPU 7 might have ejected the cacheline from its cache (for example, in order to make room for other data), so that by the time that the request arrived, the cacheline was on its way to memory.
5. A correctable error might have occurred in the cacheline, which would then need to be corrected at some point before the data was used.

Production-quality cache-coherence mechanisms are extremely complicated due to these sorts of considerations.

Quick Quiz 2.4:
Why is it necessary to flush the cacheline from CPU 7’s cache?

Answer:
If the cacheline was not flushed from CPU 7’s cache, then CPUs 0 and 7 might have different values for the same set of variables in the cacheline. This sort of incoherence would greatly complicate parallel software, and so hardware architects have been convinced to avoid it.

Quick Quiz 2.5:
Surely the hardware designers could be persuaded to improve this situation! Why have they been content with such abysmal performance for these single-instruction operations?

Answer:
The hardware designers have been working on this problem, and have consulted with no less a luminary than the physicist Stephen Hawking. Hawking’s observation was that the hardware designers have two basic problems [Gar07]:

1. the finite speed of light, and
2. the atomic nature of matter.

The first problem limits raw speed, and the second limits miniaturization, which in turn limits frequency. And even this sidesteps the power-consumption issue that is currently holding production frequencies to well below 10 GHz.

Nevertheless, some progress is being made, as may be seen by comparing Table G.1 with Table 2.1 on page 2.1.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost (ns)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock period</td>
<td>0.4</td>
<td>1.0</td>
</tr>
<tr>
<td>“Best-case” CAS</td>
<td>12.2</td>
<td>33.8</td>
</tr>
<tr>
<td>Best-case lock</td>
<td>25.6</td>
<td>71.2</td>
</tr>
<tr>
<td>Single cache miss</td>
<td>12.9</td>
<td>35.8</td>
</tr>
<tr>
<td>CAS cache miss</td>
<td>7.0</td>
<td>19.4</td>
</tr>
<tr>
<td>Off-Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single cache miss</td>
<td>31.2</td>
<td>86.6</td>
</tr>
<tr>
<td>CAS cache miss</td>
<td>31.2</td>
<td>86.5</td>
</tr>
<tr>
<td>Off-Socket</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single cache miss</td>
<td>92.4</td>
<td>256.7</td>
</tr>
<tr>
<td>CAS cache miss</td>
<td>95.9</td>
<td>266.4</td>
</tr>
<tr>
<td>Comms Fabric</td>
<td>4,500</td>
<td>7,500</td>
</tr>
<tr>
<td>Global Comms</td>
<td>195,000,000</td>
<td>324,000,000</td>
</tr>
</tbody>
</table>

Table G.1: Performance of Synchronization Mechanisms on 16-CPU 2.8GHz Intel X5550 (Nehalem) System
Integration of hardware threads in a single core and multiple cores on a die have improved latencies greatly, at least within the confines of a single core or single die. There has been some improvement in overall system latency, but only by about a factor of two. Unfortunately, neither the speed of light nor the atomic nature of matter has changed much in the past few years.

Section 2.3 looks at what else hardware designers might be able to do to ease the plight of parallel programmers.

Quick Quiz 2.6:
These numbers are insanely large! How can I possibly get my head around them?

Answer:
Get a roll of toilet paper. In the USA, each roll will normally have somewhere around 350-500 sheets. Tear off one sheet to represent a single clock cycle, setting it aside. Now unroll the rest of the roll. The resulting pile of toilet paper will likely represent a single CAS cache miss.

For the more-expensive inter-system communications latencies, use several rolls (or multiple cases) of toilet paper to represent the communications latency.

Important safety tip: make sure to account for the needs of those you live with when appropriating toilet paper!

Quick Quiz 2.7:
Given that distributed-systems communication is so horribly expensive, why does anyone bother with them?

Answer:
There are a number of reasons:

1. Shared-memory multiprocessor systems have strict size limits. If you need more than a few thousand CPUs, you have no choice but to use a distributed system.

2. Extremely large shared-memory systems tend to be quite expensive and to have even longer cache-miss latencies than does the small four-CPU system shown in Table 2.1.

3. The distributed-systems communications latencies do not necessarily consume the CPU, which can often allow computation to proceed in parallel with message transfer.

4. Many important problems are “embarrassingly parallel”, so that extremely large quantities of processing may be enabled by a very small number of messages. SETI@HOME [aCB08] is but one example of such an application. These sorts of applications can make good use of networks of computers despite extremely long communications latencies.

It is likely that continued work on parallel applications will increase the number of embarrassingly parallel applications that can run well on machines and/or clusters having long communications latencies. That said, greatly reduced hardware latencies would be an extremely welcome development.

G.3 Tools of the Trade

Quick Quiz 3.1:
But this silly shell script isn’t a real parallel program! Why bother with such trivia?

Answer:
Because you should never forget the simple stuff!

Please keep in mind that the title of this book is “Is Parallel Programming Hard, And, If So, What Can You Do About It?”. One of the most effective things you can do about it is to avoid forgetting the simple stuff! After all, if you choose to do parallel programming the hard way, you have no one but yourself to blame for it being hard.

Quick Quiz 3.2:
Is there a simpler way to create a parallel shell script? If so, how? If not, why not?

Answer:
One straightforward approach is the shell pipeline:
```
grep $pattern1 | sed -e 's/a/b/' | sort
```
For a sufficiently large input file, grep will pattern-match in parallel with sed editing and with the input processing of sort. See the file parallel.sh for a demonstration of shell-script parallelism and pipelining.

Quick Quiz 3.3:
But if script-based parallel programming is so easy, why bother with anything else?

**Answer:**
In fact, it is quite likely that a very large fraction of parallel programs in use today are script-based. However, script-based parallelism does have its limitations:

1. Creation of new processes is usually quite heavyweight, involving the expensive `fork()` and `exec()` system calls.
2. Sharing of data, including pipelining, typically involves expensive file I/O.
3. The reliable synchronization primitives available to scripts also typically involve expensive file I/O.

These limitations require that script-based parallelism use coarse-grained parallelism, with each unit of work having execution time of at least tens of milliseconds, and preferably much longer.

Those requiring finer-grained parallelism are well advised to think hard about their problem to see if it can be expressed in a coarse-grained form. If not, they should consider using other parallel-programming environments, such as those discussed in Section 3.2.

**Quick Quiz 3.4:**
Why does this `wait()` primitive need to be so complicated? Why not just make it work like the shell-script `wait` does?

**Answer:**
Some parallel applications need to take special action when specific children exit, and therefore need to wait for each child individually. In addition, some parallel applications need to detect the reason that the child died. As we saw in Figure 3.3, it is not hard to build a `waitall()` function out of the `wait()` function, but it would be impossible to do the reverse. Once the information about a specific child is lost, it is lost.

**Quick Quiz 3.5:**
Isn’t there a lot more to `fork()` and `wait()` than discussed here?

**Answer:**
Indeed there is, and it is quite possible that this section will be expanded in future versions to include messaging features (such as UNIX pipes, TCP/IP, and shared file I/O) and memory mapping (such as `mmap()` and `shmget()`). In the meantime, there are any number of textbooks that cover these primitives in great detail, and the truly motivated can read manpages, existing parallel applications using these primitives, as well as the source code of the Linux-kernel implementations themselves.

**Quick Quiz 3.6:**
If the `mythread()` function in Figure 3.5 can simply return, why bother with `pthread_exit()`?

**Answer:**
In this simple example, there is no reason whatsoever. However, imagine a more complex example, where `mythread()` invokes other functions, possibly separately compiled. In such a case, `pthread_exit()` allows these other functions to end the thread’s execution without having to pass some sort of error return all the way back up to `mythread()`.

**Quick Quiz 3.7:**
If the C language makes no guarantees in presence of a data race, then why does the Linux kernel have so many data races? Are you trying to tell me that the Linux kernel is completely broken???

**Answer:**
Ah, but the Linux kernel is written in a carefully selected superset of the C language that includes special gcc extensions, such as asms, that permit safe execution even in presence of data races. In addition, the Linux kernel does not run on a number of platforms where data races would be especially problematic. For an example, consider embedded systems with 32-bit pointers and 16-bit busses. On such a system, a data race involving a store to and a load from a given pointer might well result in the load returning the low-order 16 bits of the old value of the pointer concatenated with the high-order 16 bits of the new value of the pointer.

**Quick Quiz 3.8:**
What if I want several threads to hold the same lock at the same time?
The first thing you should do is to ask yourself why you would want to do such a thing. If the answer is “because I have a lot of data that is read by many threads, and only occasionally updated”, then POSIX reader-writer locks might be what you are looking for. These are introduced in Section 3.2.4.

Another way to get the effect of multiple threads holding the same lock is for one thread to acquire the lock, and then use `pthread_create()` to create the other threads. The question of why this would ever be a good idea is left to the reader.

Quick Quiz 3.9:
Why not simply make the argument to `lock_reader()` on line 5 of Figure 3.6 be a pointer to a `pthread_mutex_t`?

Answer:
Because we will need to pass `lock_reader()` to `pthread_create()`. Although we could cast the function when passing it to `pthread_create()`, function casts are quite a bit uglier and harder to get right than are simple pointer casts.

Quick Quiz 3.10:
Writing four lines of code for each acquisition and release of a `pthread_mutex_t` sure seems painful! Isn’t there a better way?

Answer:
Indeed! And for that reason, the `pthread_mutex_lock()` and `pthread_mutex_unlock()` primitives are normally wrapped in functions that do this error checking. Later on, we will wrapper them with the Linux kernel `spin_lock()` and `spin_unlock()` APIs.

Quick Quiz 3.11:
Is “x = 0” the only possible output from the code fragment shown in Figure 3.7? If so, why? If not, what other output could appear, and why?

Answer:
No. The reason that “x = 0” was output was that `lock_reader()` acquired the lock first. Had `lock_writer()` instead acquired the lock first, then the output would have been “x = 3”. However, because the code fragment started `lock_reader()` first and because this run was performed on a multiprocessor, one would normally expect `lock_reader()` to acquire the lock first. However, there are no guarantees, especially on a busy system.

Quick Quiz 3.12:
Using different locks could cause quite a bit of confusion, what with threads seeing each others’ intermediate states. So should well-written parallel programs restrict themselves to using a single lock in order to avoid this kind of confusion?

Answer:
Although it is sometimes possible to write a program using a single global lock that both performs and scales well, such programs are exceptions to the rule. You will normally need to use multiple locks to attain good performance and scalability.

One possible exception to this rule is “transactional memory”, which is currently a research topic. Transactional-memory semantics can be thought of as those of a single global lock with optimizations permitted and with the addition of rollback [Boe09].

Quick Quiz 3.13:
In the code shown in Figure 3.8, is `lock_reader()` guaranteed to see all the values produced by `lock_writer()`? Why or why not?

Answer:
No. On a busy system, `lock_reader()` might be preempted for the entire duration of `lock_writer()`’s execution, in which case it would not see any of `lock_writer()`’s intermediate states for x.

Quick Quiz 3.14:
Wait a minute here!!! Figure 3.7 didn’t initialize shared variable x, so why does it need to be initialized in Figure 3.8?

Answer:
See line 3 of Figure 3.6. Because the code in Figure 3.7 ran first, it could rely on the compile-time initialization of x. The code in Figure 3.8 ran next, so it had to
re-initialize x.

Quick Quiz 3.15:
Isn’t comparing against single-CPU throughput a bit harsh? ■

Answer:
Not at all. In fact, this comparison was, if anything, overly lenient. A more balanced comparison would be against single-CPU throughput with the locking primitives commented out.

Quick Quiz 3.16:
But 1,000 instructions is not a particularly small size for a critical section. What do I do if I need a much smaller critical section, for example, one containing only a few tens of instructions? ■

Answer:
If the data being read never changes, then you do not need to hold any locks while accessing it. If the data changes sufficiently infrequently, you might be able to checkpoint execution, terminate all threads, change the data, then restart at the checkpoint.

Another approach is to keep a single exclusive lock per thread, so that a thread read-acquires the larger aggregate reader-writer lock by acquiring its own lock, and write-acquires by acquiring all the per-thread locks [HW92]. This can work quite well for readers, but causes writers to incur increasingly large overheads as the number of threads increases.

Some other ways of handling very small critical sections are described in Section 8.3.

Quick Quiz 3.17:
In Figure 3.10, all of the traces other than the 100M trace deviate gently from the ideal line. In contrast, the 100M trace breaks sharply from the ideal line at 64 CPUs. In addition, the spacing between the 100M trace and the 10M trace is much smaller than that between the 10M trace and the 1M trace. Why does the 100M trace behave so much differently than the other traces? ■

Answer:
Your first clue is that 64 CPUs is exactly half of the 128 CPUs on the machine. The difference is an artifact of hardware threading. This system has 64 cores with two hardware threads per core. As long as fewer than 64 threads are running, each can run in its own core. But as soon as there are more than 64 threads, some of the threads must share cores. Because the pair of threads in any given core share some hardware resources, the throughput of two threads sharing a core is not quite as high as that of two threads each in their own core. So the performance of the 100M trace is limited not by the reader-writer lock, but rather by the sharing of hardware resources between hardware threads in a single core.

This can also be seen in the 10M trace, which deviates gently from the ideal line up to 64 threads, then breaks sharply down, parallel to the 10M trace. Up to 64 threads, the 10M trace is limited primarily by reader-writer lock scalability, and beyond that, also by sharing of hardware resources between hardware threads in a single core.

Quick Quiz 3.18:
Power-5 is several years old, and new hardware should be faster. So why should anyone worry about reader-writer locks being slow? ■

Answer:
In general, newer hardware is improving. However, it will need to improve more than two orders of magnitude to permit reader-writer lock to achieve ideal performance on 128 CPUs. Worse yet, the greater the number of CPUs, the larger the required performance improvement. The performance problems of reader-writer locking are therefore very likely to be with us for quite some time to come.

Quick Quiz 3.19:
Is it really necessary to have both sets of primitives? ■

Answer:
Strictly speaking, no. One could implement any member of the second set using the corresponding member of the first set. For example, one could implement __sync_nand_and_fetch() in terms of __sync_fetch_and_nand() as follows:

```c
"tmp = v;
ret = __sync_fetch_and_nand(p, tmp);
ret = ~ret & tmp;
```

It is similarly possible to implement __sync_fetch_and_add(), __sync_fetch_and_
sub(), and __sync_fetch_and_xor() in terms of their post-value counterparts.

However, the alternative forms can be quite convenient, both for the programmer and for the compiler/library implementor.

Quick Quiz 3.20:
Given that these atomic operations will often be able to generate single atomic instructions that are directly supported by the underlying instruction set, shouldn’t they be the fastest possible way to get things done? 

Answer:
Unfortunately, no. See Chapter 4 for some stark counterexamples.

Quick Quiz 3.21:
What happened to the Linux-kernel equivalents to fork() and join()? 

Answer:
They don’t really exist. All tasks executing within the Linux kernel share memory, at least unless you want to do a huge amount of memory-mapping work by hand.

G.4 Counting

Quick Quiz 4.1:
Why on earth should efficient and scalable counting be hard? After all, computers have special hardware for the sole purpose of doing counting, addition, subtraction, and lots more besides, don’t they???

Answer:
Because the straightforward counting algorithms, for example, atomic operations on a shared counter, are slow and scale badly, as will be seen in Section 4.1.

Quick Quiz 4.2:
Network-packet counting problem. Suppose that you need to collect statistics on the number of networking packets (or total number of bytes) transmitted and/or received. Packets might be transmitted or received by any CPU on the system. Suppose further that this large machine is capable of handling a million packets per second, and that there is a systems-monitoring package that reads out the count every five seconds. How would you implement this statistical counter? 

Answer:
Hint: the act of updating the counter must be blazingly fast, but because the counter is read out only about once in five million updates, the act of reading out the counter can be quite slow. In addition, the value read out normally need not be all that accurate—after all, since the counter is updated a thousand times per millisecond, we should be able to work with a value that is within a few thousand counts of the “true value”, whatever “true value” might mean in this context. However, the value read out should maintain roughly the same absolute error over time. For example, a 1% error might be just fine when the count is on the order of a million or so, but might be absolutely unacceptable once the count reaches a trillion. See Section 4.2.

Quick Quiz 4.3:
Approximate structure-allocation limit problem. Suppose that you need to maintain a count of the number of structures allocated in order to fail any allocations once the number of structures in use exceeds a limit (say, 10,000). Suppose further that these structures are short-lived, that the limit is rarely exceeded, and that a “sloppy” approximate limit is acceptable.

Answer:
Hint: the act of updating the counter must be blazingly fast, but the counter is read out each time that the counter is increased. However, the value read out need not be accurate except that it absolutely must distinguish perfectly between values below the limit and values greater than or equal to the limit. See Section 4.3.

Quick Quiz 4.4:
Exact structure-allocation limit problem. Suppose that you need to maintain a count of the number of structures allocated in order to fail any allocations once the number of structures in use exceeds an exact limit (say, 10,000). Suppose further that these structures are short-lived, and that the limit is rarely exceeded, that there is almost always at least one structure in use, and suppose further still that it is necessary to know exactly
when this counter reaches zero, for example, in order to free up some memory that is not required unless there is at least one structure in use.

**Answer:**

Hint: the act of updating the counter must be blazingly fast, but the counter is read out each time that the counter is increased. However, the value read out need not be accurate except that it absolutely must distinguish perfectly between values between the limit and zero on the one hand, and values that either are less than or equal to zero or are greater than or equal to the limit on the other hand. See Section 4.4.

**Quick Quiz 4.5:**

**Removable I/O device access-count problem.** Suppose that you need to maintain a reference count on a heavily used removable mass-storage device, so that you can tell the user when it is safe to remove the device. This device follows the usual removal procedure where the user indicates a desire to remove the device, and the system tells the user when it is safe to do so.

**Answer:**

Hint: the act of updating the counter must be blazingly fast and scalable in order to avoid slowing down I/O operations, but because the counter is read out only when the user wishes to remove the device, the counter read-out operation can be extremely slow. Furthermore, there is no need to be able to read out the counter at all unless the user has already indicated a desire to remove the device. In addition, the value read out need not be accurate except that it absolutely must distinguish perfectly between non-zero and zero values. However, once it has read out a zero value, it must act to keep the value at zero until it has taken some action to prevent subsequent threads from gaining access to the device being removed. See Section 4.5.

**Quick Quiz 4.6:**

But doesn’t the `++` operator produce an x86 add-to-memory instruction? And won’t the CPU cache cause this to be atomic?

**Answer:**

Although the `++` operator could be atomic, there is no requirement that it be so. Furthermore, the `ACCESS_ONCE()` primitive forces most version of GCC to load the value to a register, increment the register, then store the value to memory, which is decidedly non-atomic.

**Quick Quiz 4.7:**

The 8-figure accuracy on the number of failures indicates that you really did test this. Why would it be necessary to test such a trivial program, especially when the bug is easily seen by inspection?

**Answer:**

There are no trivial parallel programs, and most days I am not so sure that there are trivial sequential programs, either.

No matter how small or simple the program, if you haven’t tested it, it does not work. And even if you have tested it, Murphy says there are at least a few bugs still lurking.

Furthermore, while proofs of correctness certainly do have their place, they never will replace testing, including the `counttorture.h` test setup used here. After all, proofs can have bugs just as easily as programs can!

**Quick Quiz 4.8:**

Why doesn’t the dashed line on the x axis meet the diagonal line at $y = 1$?

**Answer:**

Because of the overhead of the atomic operation. The dashed line on the x axis represents the overhead of a single non-atomic increment. After all, an ideal algorithm would not only scale linearly, it would also incur no performance penalty compared to single-threaded code.

This level of idealism may seem severe, but if it is good enough for Linus Torvalds, it is good enough for you.

**Quick Quiz 4.9:**

But atomic increment is still pretty fast. And incrementing a single variable in a tight loop sounds pretty unrealistic to me, after all, most of the program’s execution should be devoted to actually doing work, not accounting for the work it has done! Why should I care about making this go faster?

**Answer:**

In many cases, atomic increment will in fact be fast.
enough for you. In those cases, you should by all means use atomic increment. That said, there are many real-world situations where more elaborate counting algorithms are required. The canonical example of such a situation is counting packets and bytes in highly optimized networking stacks, where it is all too easy to find much of the execution time going into these sorts of accounting tasks, especially on large multiprocessors.

In addition, counting provides an excellent view of the issues encountered in shared-memory parallel programs.

**Quick Quiz 4.10:**
But why can’t CPU designers simply ship the operation to the data, avoiding the need to circulate the cache line containing the global variable being incremented?

**Answer:**
It might well be possible to do this in some cases. However, there are a few complications:

1. If the value of the variable is required, then the thread will be forced to wait for the operation to be shipped to the data, and then for the result to be shipped back.

2. If the atomic increment must be ordered with respect to prior and/or subsequent operations, then the thread will be forced to wait for the operation to be shipped to the data, and for an indication that the operation completed to be shipped back.

3. Shipping operations among CPUs will likely require more signals, which will consume more die area and more electrical power.

But what if neither of the first two conditions holds? Then you should think carefully about the algorithms discussed in Section 4.2, which achieve near-ideal performance on commodity hardware.

If either or both of the first two conditions hold, there is some hope for improvement. One could imagine the hardware implementing a combining tree, so that the increment requests from multiple CPUs are combined by the hardware into a single addition when the combined request reaches the hardware. The hardware could also apply an order to the requests, thus returning to each CPU the return value corresponding to its particular atomic increment. This results in instruction latency that varies as $O(\log N)$, where $N$ is the number of CPUs, as shown in Figure G.1.

This is a great improvement over the $O(N)$ performance of current hardware shown in Figure 4.4, and it is possible that hardware latencies might decrease somewhat if innovations such as three-D fabrication prove practical. Nevertheless, we will see that in some important special cases, software can do much better.

**Quick Quiz 4.11:**
But doesn’t the fact that C’s “integers” are limited in size complicate things?

**Answer:**
No, because modulo addition is still commutative and associative. At least as long as you use unsigned integer. Recall that in the C standard, overflow of signed integers results in undefined behavior (never mind the fact that machines that do anything other than wrap on overflow are quite rare these days.

That said, one potential source of additional complexity arises when attempting to gather (say) a 64-bit sum from 32-bit per-thread counters. For the moment, dealing with this added complexity is left as an exercise for the reader.

**Quick Quiz 4.12:**
An array?? But doesn’t that limit the number of threads?

**Answer:**
It can, and in this toy implementation, it does. But it is not that hard to come up with an alternative implementation that permits an arbitrary number of threads. However,
this is left as an exercise for the reader.

**Quick Quiz 4.13:**
What other choice does gcc have, anyway???

**Answer:**
According to the C standard, the effects of fetching a variable that might be concurrently modified by some other thread are undefined. It turns out that the C standard really has no other choice, given that C must support (for example) eight-bit architectures which are incapable of atomically loading a `long`. An upcoming version of the C standard aims to fill this gap, but until then, we depend on the kindness of the gcc developers.

**Quick Quiz 4.14:**
How does the per-thread `counter` variable in Figure 4.5 get initialized?

**Answer:**
The C standard specifies that the initial value of global variables is zero, unless they are explicitly initialized. So the initial value of all the instances of `counter` will be zero.

That said, one often takes differences of consecutive reads from statistical counters, in which case the initial value is irrelevant.

**Quick Quiz 4.15:**
How is the code in Figure 4.5 supposed to permit more than one counter?

**Answer:**
Indeed, this toy example does not support more than one counter. Modifying it so that it can provide multiple counters is left as an exercise to the reader.

**Quick Quiz 4.16:**
Why does `inc_count()` in Figure 4.7 need to use atomic instructions?

**Answer:**
If non-atomic instructions were used, counts could be lost.

**Quick Quiz 4.17:**
Won’t the single global thread in the function `eventual()` of Figure 4.7 be just as severe a bottleneck as a global lock would be?

**Answer:**
In this case, no. What will happen instead is that the estimate of the counter value returned by `read_count()` will become more inaccurate.

**Quick Quiz 4.18:**
Won’t the estimate returned by `read_count()` in Figure 4.7 become increasingly inaccurate as the number of threads rises?

**Answer:**
Yes. If this proves problematic, one fix is to provide multiple `eventual()` threads, each covering its own subset of the other threads. In even more extreme cases, a tree-like hierarchy of `eventual()` threads might be required.

**Quick Quiz 4.19:**
Why do we need an explicit array to find the other threads’ counters? Why doesn’t gcc provide a `per_thread()` interface, similar to the Linux kernel’s `per_cpu()` primitive, to allow threads to more easily access each others’ per-thread variables?

**Answer:**
Why indeed?

To be fair, gcc faces some challenges that the Linux kernel gets to ignore. When a user-level thread exits, its per-thread variables all disappear, which complicates the problem of per-thread-variable access, particularly before the advent of user-level RCU. In contrast, in the Linux kernel, when a CPU goes offline, that CPU’s per-CPU variables remain mapped and accessible.

Similarly, when a new user-level thread is created, its per-thread variables suddenly come into existence. In contrast, in the Linux kernel, all per-CPU variables are mapped and initialized at boot time, regardless of whether the corresponding CPU exists yet, or indeed, whether the corresponding CPU will ever exist.

A key limitation that the Linux kernel imposes is a compile-time maximum limit on the number of CPUs,
namely, CONFIG_NR_CPUS. In contrast, in user space, there is no hard-coded upper limit on the number of threads.

Of course, both environments must deal with dynamically loaded code (dynamic libraries in user space, kernel modules in the Linux kernel), which increases the complexity of per-thread variables in both environments.

These complications make it significantly harder for user-space environments to provide access to other threads’ per-thread variables. Nevertheless, such access is highly useful, and it is hoped that it will someday appear.

Quick Quiz 4.20:
Why on earth do we need something as heavyweight as a lock guarding the summation in the function read_count() in Figure 4.8? ■

Answer:
Remember, when a thread exits, its per-thread variables disappear. Therefore, if we attempt to access a given thread’s per-thread variables after that thread exits, we will get a segmentation fault. The lock coordinates summation and thread exit, preventing this scenario.

Of course, we could instead read-acquire a reader-writer lock, but Chapter 8 will introduce even lighter-weight mechanisms for implementing the required coordination.

Quick Quiz 4.21:
Why on earth do we need to acquire the lock in count_register_thread() in Figure 4.8? It is a single properly aligned machine-word store to a location that no other thread is modifying, so it should be atomic anyway, right? ■

Answer:
This lock could in fact be omitted, but better safe than sorry, especially given that this function is executed only at thread startup, and is therefore not on any critical path. Now, if we were testing on machines with thousands of CPUs, we might need to omit the lock, but on machines with “only” a hundred or so CPUs, no need to get fancy.

Quick Quiz 4.22:
Fine, but the Linux kernel doesn’t have to acquire a lock when reading out the aggregate value of per-CPU counters. So why should user-space code need to do this??? ■

Answer:
Remember, the Linux kernel’s per-CPU variables are always accessible, even if the corresponding CPU is offline — even if the corresponding CPU never existed and never will exist.

One workaround is to ensure that each thread sticks around until all threads are finished, as shown in Figure G.2. Analysis of this code is left as an exercise to the reader, however, please note that it does not fit well into the counttorture.h counter-evaluation scheme. (Why not?) Chapter 8 will introduce synchronization mechanisms that handle this situation in a much more graceful manner.

Quick Quiz 4.23:
APPENDIX G. ANSWERS TO QUICK QUIZZES

What fundamental difference is there between counting packets and counting the total number of bytes in the packets, given that the packets vary in size?

**Answer:**
When counting packets, the counter is only incremented by the value one. On the other hand, when counting bytes, the counter might be incremented by largish numbers.

Why does this matter? Because in the increment-by-one case, the value returned will be exact in the sense that the counter must necessarily have taken on that value at some point in time, even if it is impossible to say precisely when that point occurred. In contrast, when counting bytes, two different threads might return values that are inconsistent with any global ordering of operations.

To see this, suppose that thread 0 adds the value three to its counter, thread 1 adds the value five to its counter, and threads 2 and 3 sum the counters. If the system is “weakly ordered” or if the compiler uses aggressive optimizations, thread 2 might find the sum to be three and thread 3 might find the sum to be five. The only possible global orders of the sequence of values of the counter are 0,3,8 and 0,5,8, and neither order is consistent with the results obtained.

If you missed this one, you are not alone. Michael Scott used this question to stump Paul McKenney during Paul’s Ph.D. defense.

**Quick Quiz 4.24:**
Given that the reader must sum all the threads’ counters, this could take a long time given large numbers of threads. Is there any way that the increment operation can remain fast and scalable while allowing readers to also enjoy reasonable performance and scalability?

**Answer:**
One approach would be to maintain a global approximation to the value. Readers would increment their per-thread variable, but when it reached some predefined limit, atomically add it to a global variable, then zero their per-thread variable. This would permit a tradeoff between average increment overhead and accuracy of the value read out.

The reader is encouraged to think up and try out other approaches, for example, using a combining tree.

**Quick Quiz 4.25:**
What is with the strange form of the condition on line 3 of Figure 4.11? Why not the following more intuitive form of the fastpath?

```
3 if (counter + delta <= countermax){
4    counter += delta;
5    return 1;
6 }
```

**Answer:**
Two words. “Integer overflow.”

Try the above formulation with `counter` equal to 10 and `delta` equal to `ULONG_MAX`. Then try it again with the code shown in Figure 4.11.

A good understanding of integer overflow will be required for the rest of this example, so if you have never dealt with integer overflow before, please try several examples to get the hang of it. Integer overflow can sometimes be more difficult to get right than parallel algorithms!

**Quick Quiz 4.26:**
Why does `globalize_count()` zero the per-thread variables, only to later call `balance_count()` to refill them in Figure 4.11? Why not just leave the per-thread variables non-zero?

**Answer:**
That is in fact what an earlier version of this code did. But addition and subtraction are extremely cheap, and handling all of the special cases that arise is quite complex. Again, feel free to try it yourself, but beware of integer overflow!

**Quick Quiz 4.27:**
Given that `globalreserve` counted against us in `add_count()`, why doesn’t it count for us in `sub_count()` in Figure 4.11?

**Answer:**
The `globalreserve` variable tracks the sum of all threads’ `countermax` variables. The sum of these threads’ `counter` variables might be anywhere from zero to `globalreserve`. We must therefore take a conservative approach, assuming that all threads’ `counter` variables are full in `add_count()` and that they are all empty in `sub_count()`.

But remember this question, as we will come back to it
G.4. COUNTING

Later.

Quick Quiz 4.28:
Why have both add_count() and sub_count() in Figure 4.11? Why not simply pass a negative number to add_count()?

Answer:
Given that add_count() takes an unsigned long as its argument, it is going to be a bit tough to pass it a negative number. And unless you have some anti-matter memory, there is little point in allowing negative numbers when counting the number of structures in use!

Quick Quiz 4.29:
In what way does line 7 of Figure 4.15 violate the C standard?

Answer:
It assumes eight bits per byte. This assumption does hold for all current commodity microprocessors that can be easily assembled into shared-memory multiprocessors, but certainly does not hold for all computer systems that have ever run C code. (What could you do instead in order to comply with the C standard? What drawbacks would it have?)

Quick Quiz 4.30:
Given that there is only one ctrandmax variable, why bother passing in a pointer to it on line 18 of Figure 4.15?

Answer:
There is only one ctrandmax variable per thread. Later, we will see code that needs to pass other threads’ ctrandmax variables to split_ctrandmax().

Quick Quiz 4.31:
Why does merge_ctrandmax() in Figure 4.15 return an int rather than storing directly into an atomic_t?

Answer:
Later, we will see that we need the int return to pass to the atomic_cmpxchg() primitive.

Quick Quiz 4.32:
Yecch! Why the ugly goto on line 11 of Figure 4.16? Haven’t you heard of the break statement???

Answer:
Replacing the goto with a break would require keeping a flag to determine whether or not line 15 should return, which is not the sort of thing you want on a fastpath. If you really hate the goto that much, your best bet would be to pull the fastpath into a separate function that returned success or failure, with “failure” indicating a need for the slowpath. This is left as an exercise for goto-hating readers.

Quick Quiz 4.33:
Why would the atomic_cmpxchg() primitive at lines 13-14 of Figure 4.16 ever fail? After all, we picked up its old value on line 9 and have not changed it!

Answer:
Later, we will see how the flush_local_count() function in Figure 4.18 might update this thread’s ctrandmax variable concurrently with the execution of the fastpath on lines 8-14 of Figure 4.16.

Quick Quiz 4.34:
What stops a thread from simply refilling its ctrandmax variable immediately after flush_local_count() on line 14 of Figure 4.18 empties it?

Answer:
This other thread cannot refill its ctrandmax until the caller of flush_local_count() releases the gblcnt_mutex. By that time, the caller of flush_local_count() will have finished making use of the counts, so there will be no problem with this other thread refilling — assuming that the value of globalcount is large enough to permit a refill.

Quick Quiz 4.35:
What prevents concurrent execution of the fastpath of either atomic_add() or atomic_sub() from interfering with the ctrandmax variable while
flush_local_count() is accessing it on line 27 of Figure 4.18 empties it? ■

**Answer:**
Nothing. Consider the following three cases:

1. If flush_local_count()’s atomic_xchg() executes before the split_ctrandmax() of either fastpath, then the fastpath will see a zero counter and countermax, and will thus transfer to the slowpath (unless of course delta is zero).

2. If flush_local_count()’s atomic_xchg() executes after the split_ctrandmax() of either fastpath, but before that fastpath’s atomic_cmpxchg(), then the atomic_cmpxchg() will fail, causing the fastpath to restart, which reduces to case 1 above.

3. If flush_local_count()’s atomic_xchg() executes after the atomic_cmpxchg() of either fastpath, then the fastpath will (most likely) complete successfully before flush_local_count() zeroes the thread’s ctrandmax variable.

Either way, the race is resolved correctly.

**Quick Quiz 4.36:**
Given that the atomic_set() primitive does a simple store to the specified atomic_t, how can line 53 of balance_count() in Figure 4.18 work correctly in face of concurrent flush_local_count() updates to this variable? ■

**Answer:**
The caller of both balance_count() and flush_local_count() hold gblcnt_mutex, so only one may be executing at a given time.

**Quick Quiz 4.37:**
But signal handlers can be migrated to some other CPU while running. Doesn’t this possibility require that atomic instructions and memory barriers are required to reliably communicate between a thread and a signal handler that interrupts that thread? ■

**Answer:**
No. If the signal handler is migrated to another CPU, then the interrupted thread is also migrated along with it.

**Quick Quiz 4.38:**
In Figure 4.19, why is the REQ theft state colored blue? ■

**Answer:**
To indicate that only the fastpath is permitted to change the theft state.

**Quick Quiz 4.39:**
In Figure 4.19, what is the point of having separate REQ and ACK theft states? Why not simplify the state machine by collapsing them into a single state? Then whichever of the signal handler or the fastpath gets there first could set the state to READY. ■

**Answer:**
Reasons why collapsing the REQ and ACK states would be a very bad idea include:

1. The slowpath uses the REQ and ACK states to determine whether the signal should be retransmitted. If the states were collapsed, the slowpath would have no choice but to send redundant signals, which would have the unhelpful effect of slowing down the fast-path.

2. The following race would result:

   (a) The slowpath sets a given thread’s state to REQACK.

   (b) That thread has just finished its fastpath, and notes the REQACK state.

   (c) The thread receives the signal, which also notes the REQACK state, and, because there is no fastpath in effect, sets the state to READY.

   (d) The slowpath notes the READY state, steals the count, and sets the state to IDLE, and completes.

   (e) The fastpath sets the state to READY, disabling further fastpath execution for this thread.

The basic problem here is that the combined REQACK state can be referenced by both the signal handler and the fastpath. The clear separation maintained by the four-state setup ensures orderly state transitions.
That said, you might well be able to make a three-state setup work correctly. If you do succeed, compare carefully to the four-state setup. Is the three-state solution really preferable, and why or why not?

Quick Quiz 4.40:
In Figure 4.21 function flush_local_count_sig(), why are there ACCESS_ONCE() wrappers around the uses of the theft per-thread variable?

Answer:
The first one (on line 11) can be argued to be unnecessary. The last two (lines 14 and 16) are important. If these are removed, the compiler would be within its rights to rewrite lines 14-17 as follows:

```
14 theft = THEFT_READY;
15 if (counting) {
16   theft = THEFT_ACK;
17 }
```

This would be fatal, as the slowpath might see the transient value of THEFT_READY, and start stealing before the corresponding thread was ready.

Quick Quiz 4.41:
In Figure 4.21, why is it safe for line 28 to directly access the other thread’s countermax variable?

Answer:
Because the other thread is not permitted to change the value of its countermax variable unless it holds the gblcnt_mutex lock. But the caller has acquired this lock, so it is not possible for the other thread to hold it, and therefore the other thread is not permitted to change its countermax variable. We can therefore safely access it — but not change it.

Quick Quiz 4.42:
In Figure 4.21, why doesn’t line 33 check for the current thread sending itself a signal?

Answer:
There is no need for an additional check. The caller of flush_local_count() has already invoked globalize_count(), so the check on line 28 will have succeeded, skipping the later pthread_kill().

Quick Quiz 4.43:
The code in Figure 4.21, works with gcc and POSIX. What would be required to make it also conform to the ISO C standard?

Answer:
The theft variable must be of type sig_atomic_t to guarantee that it can be safely shared between the signal handler and the code interrupted by the signal.

Quick Quiz 4.44:
In Figure 4.21, why does line 41 resend the signal?

Answer:
Because many operating systems over several decades have had the property of losing the occasional signal. Whether this is a feature or a bug is debatable, but irrelevant. The obvious symptom from the user’s viewpoint will not be a kernel bug, but rather a user application hanging.

Quick Quiz 4.45:
What if you want an exact limit counter to be exact only for its lower limit?

Answer:
One simple solution is to overstate the upper limit by the desired amount. The limiting case of such overstatement results in the upper limit being set to the largest value that the counter is capable of representing.

Quick Quiz 4.46:
What else had you better have done when using a biased counter?

Answer:
You had better have set the upper limit to be large enough to accommodate the bias, the expected maximum number of accesses, and enough “slop” to allow the counter to work efficiently even when the number of accesses is at its maximum.

Quick Quiz 4.47:
This is ridiculous! We are read-acquiring a reader-writer lock to update the counter? What are you playing at??? ■

Answer:
Strange, perhaps, but true! Almost enough to make you think that the name “reader-writer lock” was poorly chosen, isn’t it?

Quick Quiz 4.48:
What other issues would need to be accounted for in a real system? ■

Answer:
A huge number!
Here are a few to start with:

1. There could be any number of devices, so that the global variables are inappropriate, as are the lack of arguments to functions like do_io().
2. Polling loops can be problematic in real systems. In many cases, it is far better to have the last completing I/O wake up the device-removal thread.
3. The I/O might fail, and so do_io() will likely need a return value.
4. If the device fails, the last I/O might never complete. In such cases, there might need to be some sort of timeout to allow error recovery.
5. Both add_count() and sub_count() can fail, but their return values are not checked.
6. Reader-writer locks do not scale well. One way of avoiding the high read-acquisition costs of reader-writer locks is presented in Chapter 8.

Quick Quiz 4.49:
On the count_stat.c row of Table 4.1, we see that the update side scales linearly with the number of threads. How is that possible given that the more threads there are, the more per-thread counters must be summed up? ■

Answer:
The read-side code must scan the entire fixed-size array, regardless of the number of threads, so there is no difference in performance. In contrast, in the last two algorithms, readers must do more work when there are more threads. In addition, the last two algorithms interpose an additional level of indirection because they map from integer thread ID to the corresponding __thread variable.

Quick Quiz 4.50:
Even on the last row of Table 4.1, the read-side performance of these statistical counter implementations is pretty horrible. So why bother with them? ■

Answer:
“Use the right tool for the job.”

As can be seen from Figure 4.3, single-variable atomic increment need not apply for any job involving heavy use of parallel updates. In contrast, the algorithms shown in Table 4.1 do an excellent job of handling update-heavy situations. Of course, if you have a read-mostly situation, you should use something else, for example, a single atomically incremented variable that can be read out using a single load.

Quick Quiz 4.51:
Given the performance data shown in Table 4.2, we should always prefer update-side signals over read-side atomic operations, right? ■

Answer:
That depends on the workload. Note that you need a million readers (with roughly a 40-nanosecond performance gain) to make up for even one writer (with almost a 40-millisecond performance loss). Although there are no shortage of workloads with far greater read intensity, you will need to consider your particular workload.

In addition, although memory barriers have historically been expensive compared to ordinary instructions, you should check this on the specific hardware you will be running. The properties of computer hardware do change over time, and algorithms must change accordingly.

Quick Quiz 4.52:
Can advanced techniques be applied to address the lock contention for readers seen in Table 4.2? ■

Answer:
There are a number of ways one might go about this, and
these are left as exercises for the reader.

Quick Quiz 4.53:
The ++ operator works just fine for 1,000-digit numbers! Haven’t you heard of operator overloading???

Answer:
In the C++ language, you might well be able to use ++ on a 1,000-digit number, assuming that you had access to a class implementing such numbers. But as of 2010, the C language does not permit operator overloading.

Quick Quiz 4.54:
But if we are going to have to partition everything, why bother with shared-memory multithreading? Why not just partition the problem completely and run as multiple processes, each in its own address space?

Answer:
Indeed, multiple processes with separate address spaces can be an excellent way to exploit parallelism, as the proponents of the fork-join methodology and the Erlang language would be very quick to tell you. However, there are also some advantages to shared-memory parallelism:

1. Only the most performance-critical portions of the application must be partitioned, and such portions are usually a small fraction of the application.

2. Although cache misses are quite slow compared to individual register-to-register instructions, they are typically considerably faster than inter-process-communication primitives, which in turn are considerably faster than things like TCP/IP networking.

3. Shared-memory multiprocessors are readily available and quite inexpensive, so, in stark contrast to the 1990s, there is little cost penalty for use of shared-memory parallelism.

As always, use the right tool for the job!

G.5 Partitioning and Synchronization Design

Quick Quiz 5.1:

Figure G.3: Dining Philosophers Problem, Fully Partitioned

Is there a better solution to the Dining Philosophers Problem?

Answer:
One such improved solution is shown in Figure G.3, where the philosophers are simply provided with an additional five forks. All five philosophers may now eat simultaneously, and there is never any need for philosophers to wait on one another. In addition, the improved disease control provided by this approach should not be underestimated.

This solution can seem like cheating to some, but such “cheating” is key to finding good solutions to many concurrency problems.

Quick Quiz 5.2:
And in just what sense can this “horizontal parallelism” be said to be “horizontal”?

Answer:
Inman was working with protocol stacks, which are normally depicted vertically, with the application on top and the hardware interconnect on the bottom. Data flows up and down this stack. “Horizontal parallelism” processes packets from different network connections in parallel, while “vertical parallelism” handles different protocol-processing steps for a given packet in parallel.
“Vertical parallelism” is also called “pipelining”.

**Quick Quiz 5.3:**
In this compound double-ended queue implementation, what should be done if the queue has become non-empty while releasing and reacquiring the lock? ☐

**Answer:**
In this case, simply dequeue an item from the now-nonempty queue, release both locks, and return.

**Quick Quiz 5.4:**
Is the hashed double-ended queue a good solution? Why or why not? ☐

**Answer:**
The best way to answer this is to run `lockhdeq.c` on a number of different multiprocessor systems, and you are encouraged to do so in the strongest possible terms. One reason for concern is that each operation on this implementation must acquire not one but two locks.

The first well-designed performance study will be cited. Do not forget to compare to a sequential implementation!

**Quick Quiz 5.5:**
Move all the elements to the queue that became empty? In what possible universe is this braindead solution in any way optimal?? ☐

**Answer:**
It is optimal in the case where data flow switches direction only rarely. It would of course be an extremely poor choice if the double-ended queue was being emptied from both ends concurrently. This of course raises the question as to what possible universe emptying from both ends concurrently would be a reasonable thing to do...

**Quick Quiz 5.6:**
Why can’t the compound parallel double-ended queue implementation be symmetric? ☐

**Answer:**
The need to avoid deadlock by imposing a lock hierarchy forces the asymmetry, just as it does in the fork-numbering solution to the Dining Philosophers Problem.

**Quick Quiz 5.7:**
Why is it necessary to retry the right-dequeue operation on line 29 of Figure 5.11? ☐

**Answer:**
This retry is necessary because some other thread might have enqueued an element between the time that this thread dropped the lock and the time that it reacquired the lock.

**Quick Quiz 5.8:**
Surely the left-hand lock must sometimes be available!!!
So why is it necessary that line 26 of Figure 5.11 unconditionally release the right-hand lock? ☐

**Answer:**
It would be possible to use `spin_trylock()` to attempt to acquire the left-hand lock when it was available. However, the failure case would still need to drop the right-hand lock and then re-acquire the two locks in order. Making this transformation (and determining whether or not it is worthwhile) is left as an exercise for the reader.

**Quick Quiz 5.9:**
The tandem double-ended queue runs about twice as fast as the hashed double-ended queue, even when I increase the size of the hash table to an insanely large number. Why is that? ☐

**Answer:**
The hashed double-ended queue’s locking design only permits one thread at a time at each end, and further requires two lock acquisitions for each operation. The tandem double-ended queue also permits one thread at a time at each end, and in the common case requires only one lock acquisition per operation. Therefore, the tandem double-ended queue should be expected to outperform the hashed double-ended queue.

Can you created a double-ended queue that allows multiple concurrent operations at each end? If so, how? If not, why not?
Quick Quiz 5.10:
Is there a significantly better way of handling concurrency for double-ended queues? ■

Answer:
Transform the problem to be solved so that multiple double-ended queues can be used in parallel, allowing the simpler single-lock double-ended queue to be used, and perhaps also replace each double-ended queue with a pair of conventional single-ended queues. Without such “horizontal scaling”, the speedup is limited to 2.0. In contrast, horizontal-scaling designs can enable very large speedups, and are especially attractive if there are multiple threads working either end of the queue, because in this multiple-thread case the deque simply cannot provide strong ordering guarantees. And if there are no guarantees, we may as well obtain the performance benefits that come with refusing to provide the guarantees, right?

Quick Quiz 5.11:
What are some ways of preventing a structure from being freed while its lock is being acquired? ■

Answer:
Here are a few possible solutions to this existence guarantee problem:

1. Provide a statically allocated lock that is held while the per-structure lock is being acquired, which is an example of hierarchical locking (see Section 5.4.2). Of course, using a single global lock for this purpose can result in unacceptably high levels of lock contention, dramatically reducing performance and scalability.

2. Provide an array of statically allocated locks, hashing the structure’s address to select the lock to be acquired, as described in Chapter 6. Given a hash function of sufficiently high quality, this avoids the scalability limitations of the single global lock, but in read-mostly situations, the lock-acquisition overhead can result in unacceptably degraded performance.

3. Use a garbage collector, in software environments providing them, so that a structure cannot be deallocated while being referenced. This works very well, removing the existence-guarantee burden (and much else besides) from the developer’s shoulders, but imposes the overhead of garbage collection on the program. Although garbage-collection technology has advanced considerably in the past few decades, its overhead may be unacceptably high for some applications. In addition, some applications require that the developer exercise more control over the layout and placement of data structures than is permitted by most garbage collected environments.

4. As a special case of a garbage collector, use a global reference counter, or a global array of reference counters.

5. Use hazard pointers [Mic04], which can be thought of as an inside-out reference count. Hazard-pointer-based algorithms maintain a per-thread list of pointers, so that the appearance of a given pointer on any of these lists acts as a reference to the corresponding structure. Hazard pointers are an interesting research direction, but have not yet seen much use in production (written in 2008).

6. Use transactional memory (TM) [HM93, Lom77, ST95], so that each reference and modification to the data structure in question is performed atomically. Although TM has engendered much excitement in recent years, and seems likely to be of some use in production software, developers should exercise some caution [BLM05, BLM06, MMW07], particularly in performance-critical code. In particular, existence guarantees require that the transaction cover the full path from a global reference to the data elements being updated.

7. Use RCU, which can be thought of as an extremely lightweight approximation to a garbage collector. Updaters are not permitted to free RCU-protected data structures that RCU readers might still be referencing. RCU is most heavily used for read-mostly data structures, and is discussed at length in Chapter 8.

For more on providing existence guarantees, see Chapters 6 and 8.

Quick Quiz 5.12:
How can a single-threaded 64-by-64 matrix multiple possibly have an efficiency of less than 1.0? Shouldn’t all of the traces in Figure 5.22 have efficiency of exactly 1.0 when running on only one thread? ■
Answer:
The `matmul.c` program creates the specified number of worker threads, so even the single-worker-thread case incurs thread-creation overhead. Making the changes required to optimize away thread-creation overhead in the single-worker-thread case is left as an exercise to the reader.

Quick Quiz 5.13:
How are data-parallel techniques going to help with matrix multiply? It is already data parallel!!!

Answer:
I am glad that you are paying attention! This example serves to show that although data parallelism can be a very good thing, it is not some magic wand that automatically wards off any and all sources of inefficiency. Linear scaling at full performance, even to “only” 64 threads, requires care at all phases of design and implementation.

In particular, you need to pay careful attention to the size of the partitions. For example, if you split a 64-by-64 matrix multiply across 64 threads, each thread gets only 64 floating-point multiplies. The cost of a floating-point multiply is miniscule compared to the overhead of thread creation.

Moral: If you have a parallel program with variable input, always include a check for the input size being too small to be worth parallelizing. And when it is not helpful to parallelize, it is not helpful to spawn a single thread, now is it?

Quick Quiz 5.14:
In what situation would hierarchical locking work well?

Answer:
If the comparison on line 31 of Figure 5.25 were replaced by a much heavier-weight operation, then releasing `bp->bucket_lock` might reduce lock contention enough to outweigh the overhead of the extra acquisition and release of `cur->node_lock`.

Quick Quiz 5.15:
In Figure 5.31, there is a pattern of performance rising with increasing run length in groups of three samples, for example, for run lengths 10, 11, and 12. Why?

Answer:
This is due to the per-CPU target value being three. A run length of 12 must acquire the global-pool lock twice, while a run length of 13 must acquire the global-pool lock three times.

Quick Quiz 5.16:
Allocation failures were observed in the two-thread tests at run lengths of 19 and greater. Given the global-pool size of 40 and the per-CPU target pool size of three, what is the smallest allocation run length at which failures can occur?

Answer:
The exact solution to this problem is left as an exercise to the reader. The first solution received will be credited to its submitter. As a rough rule of thumb, the global pool size should be at least \( m + 2sn \), where “\( m \)” is the maximum number of elements allocated at a given time, “\( s \)” is the per-CPU pool size, and “\( n \)” is the number of CPUs.

G.6 Locking

Quick Quiz 6.1:
But the definition of deadlock only said that each thread was holding at least one lock and waiting on another lock that was held by some thread. How do you know that there is a cycle?

Answer:
Suppose that there is no cycle in the graph. We would then have a directed acyclic graph (DAG), which would have at least one leaf node.

If this leaf node was a lock, then we would have a thread that was waiting on a lock that wasn’t held by any thread, which violates the definition. (And in this case the thread would immediately acquire the lock.)

On the other hand, if this leaf node was a thread, then we would have a thread that was not waiting on any lock, again violating the definition. (And in this case, the thread would either be running or blocked on something that is not a lock.)

Therefore, given this definition of deadlock, there must be a cycle in the corresponding graph.
Quick Quiz 6.2:
Are there any exceptions to this rule, so that there really could be a deadlock cycle containing locks from both the library and the caller, even given that the library code never invokes any of the caller’s functions?

Answer:
Indeed there can! Here are a few of them:

1. If one of the library function’s arguments is a pointer to a lock that this library function acquires, and if the library function holds one of its locks while acquiring the caller’s lock, then we could have a deadlock cycle involving both caller and library locks.

2. If one of the library functions returns a pointer to a lock that is acquired by the caller, and if the caller acquires one of its locks while holding the library’s lock, we could again have a deadlock cycle involving both caller and library locks.

3. If one of the library functions acquires a lock and then returns while still holding it, and if the caller acquires one of its locks, we have yet another way to create a deadlock cycle involving both caller and library locks.

4. If the caller has a signal handler that acquires locks, then the deadlock cycle can involve both caller and library locks. In this case, however, the library’s locks are innocent bystanders in the deadlock cycle. That said, please note that acquiring a lock from within a signal handler is a no-no in most environments—it is not just a bad idea, it is unsupported.

Quick Quiz 6.3:
But if qsort() releases all its locks before invoking the comparison function, how can it protect against races with other qsort() threads?

Answer:
By privatizing the data elements being compared (as discussed in Chapter 7) or through use of deferral mechanisms such as reference counting (as discussed in Chapter 8).

Quick Quiz 6.4:
Name one common exception where it is perfectly reasonable to pass a pointer to a lock into a function.

Answer:
Locking primitives, of course!

Quick Quiz 6.5:
 Doesn’t the fact that pthread_cond_wait() first releases the mutex and then re-acquires it eliminate the possibility of deadlock?

Answer:
Absolutely not!

Consider the a program that acquires mutex_a, and then mutex_b, in that order, and then passes mutex_a to pthread_cond_wait. Now, pthread_cond_wait will release mutex_a, but will re-acquire it before returning. If some other thread acquires mutex_a in the meantime and then blocks on mutex_b, the program will deadlock.

Quick Quiz 6.6:
Can the transformation from Figure 6.4 to Figure 6.10 be applied universally?

Answer:
Absolutely not!

This transformation assumes that the layer_2_processing() function is idempotent, given that it might be executed multiple times on the same packet when the layer_1() routing decision changes. Therefore, in real life, this transformation can become arbitrarily complex.

Quick Quiz 6.7:
But the complexity in Figure 6.10 is well worthwhile given that it avoids deadlock, right?

Answer:
Maybe.

If the routing decision in layer_1() changes often enough, the code will always retry, never making forward progress. This is termed “livelock” if no thread makes any forward progress or “starvation” if some threads make forward progress but other do not (see Section 6.1.2).
Quick Quiz 6.8:
Why is it illegal to acquire a Lock A that is acquired outside of a signal handler without blocking signals while holding a Lock B that is acquired within a signal handler?

Answer:
Because this would lead to deadlock. Given that Lock A is held outside of a signal handler without blocking signals, a signal might be handled while holding this lock. The corresponding signal handler might then acquire Lock B, so that Lock B is acquired while holding Lock A. Therefore, if we also acquire Lock A while holding Lock B as called out in the question, we will have a deadlock cycle.

Therefore, it is illegal to acquire a lock that is acquired outside of a signal handler without blocking signals while holding another lock that is acquired within a signal handler.

Quick Quiz 6.9:
How can you legally block signals within a signal handler?

Answer:
One of the simplest and fastest ways to do so is to use the sa_mask field of the struct sigaction that you pass to sigaction() when setting up the signal.

Quick Quiz 6.10:
Given an object-oriented application that passes control freely among a group of objects such that there is no reasonable locking hierarchy, layered or otherwise, how can this application be parallelized?

Answer:
There are a number of approaches:

1. In the case of parametric search via simulation, where a large number of simulations will be run in order to converge on (for example) a good design for a mechanical or electrical device, leave the simulation single-threaded, but run many instances of the simulation in parallel. This retains the object-oriented design, and gains parallelism at a higher level.

2. Partition the objects into groups such that there is no need to operate on objects in more than one group at a given time. Then associate a lock with each group. This is an example of a single-lock-at-a-time design, which discussed in Section 6.1.1.7.

3. Partition the objects into groups such that threads can all operate on objects in the groups in some group-wise ordering. Then associate a lock with each group, and impose a locking hierarchy over the groups.

4. Impose an arbitrarily selected hierarchy on the locks, and then use conditional locking if it is necessary to acquire a lock out of order, as was discussed in Section 6.1.1.5.

5. Before carrying out a given group of operations, predict which locks will be acquired, and attempt to acquire them before actually carrying out any updates. If the prediction turns out to be incorrect, drop all the locks and retry with an updated prediction that includes the benefit of experience. This approach was discussed in Section 6.1.1.6.

6. Use transactional memory. This approach has a number of advantages and disadvantages which will be discussed in Section 15.2.

7. Refactor the application to be more concurrency-friendly. This would likely also have the side effect of making the application run faster even when single-threaded, but might also make it more difficult to modify the application.

8. Use techniques from later chapters in addition to locking.

Quick Quiz 6.11:
How can the livelock shown in Figure 6.11 be avoided?

Answer:
This is left as an exercise to the reader. Figure 6.10 provides some good hints. In many cases, livelocks are a hint that you should revisit your locking design. Or visit it in the first place if your locking design “just grew”.

Quick Quiz 6.12:
What problems can you spot in the code in Figure 6.12?
Answer:
Here are a couple:

1. A one-second wait is way too long for most uses. Wait intervals should begin with roughly the time required to execute the critical section, which will normally be in the microsecond or millisecond range.

2. The code does not check for overflow. On the other hand, this bug is nullified by the previous bug: 32 bits worth of seconds is more than 50 years.

Quick Quiz 6.13:
Wouldn’t it be better just to use a good parallel design so that lock contention was low enough to avoid unfairness?

Answer:
It would be better in some sense, but there are situations where it can be appropriate to use designs that sometimes result in high lock contentions.

For example, imagine a system that is subject to a rare error condition. It might well be best to have a simple error-handling design that has poor performance and scalability for the duration of the rare error condition, as opposed to a complex and difficult-to-debug design that is helpful only when one of those rare error conditions is in effect.

That said, it is usually worth putting some effort into attempting to produce a design that both simple as well as efficient during error conditions, for example by partitioning the problem.

Quick Quiz 6.14:
How might the lock holder be interfered with?

Answer:
If the data protected by the lock is in the same cache line as the lock itself, then attempts by other CPUs to acquire the lock will result in expensive cache misses on the part of the CPU holding the lock. In contrast, if the lock is in a different cache line than the data that it protects, the CPU holding the lock will usually suffer a cache miss only on first access to a given variable. This is a special case of false sharing, which can also occur if a pair of variables protected by different locks happen to share a cache line.

Quick Quiz 6.15:
Is there any other way for the VAX/VMS DLM to emulate a reader-writer lock?

Answer:
There are in fact several. One way would be to use the null, protected-read, and exclusive modes. Another way would be to use the null, protected-read, and concurrent-write modes. A third way would be to use the null, concurrent-read, and exclusive modes.

Quick Quiz 6.16:
Why bother with the inner loop on lines 7-8 of Figure 6.14? Why not simply repeatedly do the atomic exchange operation on line 6?

Answer:
Suppose that the lock is held and that several threads are attempting to acquire the lock. In this situation, if these threads all loop on the atomic exchange operation, they will ping-pong the cache line containing the lock among themselves, imposing load on the interconnect. In contrast, if these threads are spinning in the inner loop on lines 7-8, they will each spin within their own caches, putting negligible load on the interconnect.

Quick Quiz 6.17:
Why not simply store zero into the lock word on line 14 of Figure 6.14?

Answer:
This can be a legitimate implementation, but only if this store is preceded by a memory barrier. The memory barrier is not required when the xchg() operation is used because this operation implies a full memory barrier due to the fact that it returns a value.

Quick Quiz 6.18:
How can relying on implicit existence guarantees result in a bug?

Answer:
Here are some bugs resulting from improper use of implicit existence guarantees:

1. A program writes the address of a global variable...
to a file, then a later instance of that same program reads that address and attempts to dereference it. This can fail due to address-space randomization, to say nothing of recompilation of the program.

2. A module can record the address of one of its variables in a pointer located in some other module, then attempt to dereference that pointer after the module has been unloaded.

3. A function can record the address of one of its on-stack variables into a global pointer, which some other function might attempt to dereference after that function has returned.

I am sure that you can come up with additional possibilities.

Quick Quiz 6.19:
What if the element we need to delete is not the first element of the list on line 8 of Figure 6.15? ■

Answer:
This is a very simple hash table with no chaining, so the only element in a given bucket is the first element. The reader is invited to adapt this example to a hash table with full chaining.

Quick Quiz 6.20:
What race condition can occur in Figure 6.15? ■

Answer:
Consider the following sequence of events:

1. Thread 0 invokes delete(0), and reaches line 10 of the figure, acquiring the lock.
2. Thread 1 concurrently invokes delete(0), and reaches line 10, but spins on the lock because Thread 1 holds it.
3. Thread 0 executes lines 11-14, removing the element from the hashtable, releasing the lock, and then freeing the element.
4. Thread 0 continues execution, and allocates memory, getting the exact block of memory that it just freed.
5. Thread 0 then initializes this block of memory as some other type of structure.
6. Thread 1’s spin_lock() operation fails due to the fact that what it believes to be p->lock is no longer a spinlock.

Because there is no existence guarantee, the identity of the data element can change while a thread is attempting to acquire that element’s lock on line 10!

G.7 Data Ownership

Quick Quiz 7.1:
What form of data ownership that is extremely difficult to avoid using when creating shared-memory parallel programs (for example, using pthreads) in C or C++? ■

Answer:
Use of auto variables in functions. By default, these are private to the thread executing the current function.

Quick Quiz 7.2:
What synchronization remains in the example shown in Section 7.1? ■

Answer:
The creation of the threads via the sh & operator and the joining of thread via the sh wait command. Of course, if the processes explicitly share memory, for example, using the shmget() or mmap() system calls, explicit synchronization might well be needed when accessing or updating the shared memory. The processes might also synchronize using any of the following inter-process communications mechanisms:

1. System V semaphores.
2. System V message queues.
3. UNIX-domain sockets.
4. Networking protocols, including TCP/IP, UDP, and a whole host of others.
5. File locking.
6. Use of the open() system call with the O_CREAT and O_EXCL flags.
7. Use of the rename() system call.
A complete list of possible synchronization mechanisms is left as an exercise to the reader, who is warned that it will be an extremely long list. A surprising number of unassuming system calls can be pressed into service as synchronization mechanisms.

Quick Quiz 7.3:
Is there any shared data in the example shown in Section 7.1? ■

Answer:
That is a philosophical question.
Those wishing the answer “no” might argue that processes by definition do not share memory.
Those wishing to answer “yes” might list a large number of synchronization mechanisms that do not require shared memory, note that the kernel will have some shared state, and perhaps even argue that the assignment of process IDs (PIDs) constitute shared data.
Such arguments are excellent intellectual exercise, and are also a wonderful way of feeling intelligent, scoring points against hapless classmates or colleagues, and (especially!) avoiding getting anything useful done.

Quick Quiz 7.4:
Does it ever make sense to have partial data ownership where each thread reads only its own instance of a per-thread variable, but writes to other threads’ instances? ■

Answer:
Amazingly enough, yes. One example is a simple message-passing system where threads post messages to other threads’ mailboxes, and where each thread is responsible for removing any message it sent once that message has been acted on. Implementation of such an algorithm is left as an exercise for the reader, as is the task of identifying other algorithms with similar ownership patterns.

Quick Quiz 7.5:
What mechanisms other than POSIX signals may be used to ship functions? ■

Answer:
There is a very large number of such mechanisms, including:
1. System V message queues.
2. Shared-memory dequeue (see Section 5.1.2).
4. UNIX-domain sockets.
5. TCP/IP or UDP, possibly augmented by any number of higher-level protocols, including RPC, HTTP, XML, SOAP, and so on.
Compilation of a complete list is left as an exercise to sufficiently single-minded readers, who are warned that the list will be extremely long.

Quick Quiz 7.6:
But none of the data in the eventual() function shown on lines 15-32 of Figure 4.7 is actually owned by the eventual() thread! In just what way is this data ownership??? ■

Answer:
The key phrase is “owns the rights to the data”. In this case, the rights in question are the rights to access the per-thread counter variable defined on line 1 of the figure. This situation is similar to that described in Section 7.2.
However, there really is data that is owned by the eventual() thread, namely the t and sum variables defined on lines 17 and 18 of the figure.
For other examples of designated threads, look at the kernel threads in the Linux kernel, for example, those created by kthread_create() and kthread_run().

Quick Quiz 7.7:
Is it possible to obtain greater accuracy while still maintaining full privacy of the per-thread data? ■

Answer:
Yes. One approach is for read_count() to add the value of its own per-thread variable. This maintains full ownership and performance, but only a slight improvement in accuracy, particularly on systems with very large numbers of threads.
Another approach is for read_count() to use function shipping, for example, in the form of per-thread signals. This greatly improves accuracy, but at a significant performance cost for read_count().
However, both of these methods have the advantage of eliminating cache-line bouncing for the common case of updating counters.

### G.8 Deferred Processing

**Quick Quiz 8.1:**
Why not implement reference-acquisition using a simple compare-and-swap operation that only acquires a reference if the reference counter is non-zero?

**Answer:**
Although this can resolve the race between the release of the last reference and acquisition of a new reference, it does absolutely nothing to prevent the data structure from being freed and reallocated, possibly as some completely different type of structure. It is quite likely that the “simple compare-and-swap operation” would give undefined results if applied to the differently typed structure.

In short, use of atomic operations such as compare-and-swap absolutely requires either type-safety or existence guarantees.

**Quick Quiz 8.2:**
Why isn’t it necessary to guard against cases where one CPU acquires a reference just after another CPU releases the last reference?

**Answer:**
Because a CPU must already hold a reference in order to legally acquire another reference. Therefore, if one CPU releases the last reference, there cannot possibly be any CPU that is permitted to acquire a new reference. This same fact allows the non-atomic check in line 22 of Figure 8.2.

**Quick Quiz 8.3:**
If the check on line 22 of Figure 8.2 fails, how could the check on line 23 possibly succeed?

**Answer:**
Suppose that kref_put() is protected by RCU, so that two CPUs might be executing line 22 concurrently. Both might see the value “2”, causing both to then execute line 23. One of the two instances of atomic_dec_and_test() will decrement the value to zero and thus return 1.

**Quick Quiz 8.4:**
How can it possibly be safe to non-atomically check for equality with “1” on line 22 of Figure 8.2?

**Answer:**
Remember that it is not legal to call either kref_get() or kref_put() unless you hold a reference. If the reference count is equal to “1”, then there can’t possibly be another CPU authorized to change the value of the reference count.

**Quick Quiz 8.5:**
Why can’t the check for a zero reference count be made in a simple “if” statement with an atomic increment in its “then” clause?

**Answer:**
Suppose that the “if” condition completed, finding the reference counter value equal to one. Suppose that a release operation executes, decrementing the reference counter to zero and therefore starting cleanup operations. But now the “then” clause can increment the counter back to a value of one, allowing the object to be used after it has been cleaned up.

**Quick Quiz 8.6:**
Why isn’t this sequence-lock discussion in Chapter 6, you know, the one on locking?

**Answer:**
The sequence-lock mechanism is really a combination of two separate synchronization mechanisms, sequence counts and locking. In fact, the sequence-count mechanism is available separately in the Linux kernel via the write_seqcount_begin() and write_seqcount_end() primitives.

However, the combined write_seqlock() and write_sequnlock() primitives are used much more heavily in the Linux kernel. More importantly, many more people will understand what you mean if you say “sequence lock” than if you say “sequence count”.

So this section is entitled “Sequence Locks” so that
people will understand what it is about just from the title, and it appears in the “Deferred Processing” because (1) of the emphasis on the “sequence count” aspect of “sequence locks” and (2) because a “sequence lock” is much more than merely a lock.

Quick Quiz 8.7:
Can you use sequence locks as the only synchronization mechanism protecting a linked list supporting concurrent addition, deletion, and search?

Answer:
One trivial way of accomplishing this is to surround all accesses, including the read-only accesses, with write_seqlock() and write_sequnlock(). Of course, this solution also prohibits all read-side parallelism, and furthermore could just as easily be implemented using simple locking.

If you do come up with a solution that uses read_seqbegin() and read_seqretry() to protect read-side accesses, make sure that you correctly handle the following sequence of events:

1. CPU 0 is traversing the linked list, and picks up a pointer to list element A.
2. CPU 1 removes element A from the list and frees it.
3. CPU 2 allocates an unrelated data structure, and gets the memory formerly occupied by element A. In this unrelated data structure, the memory previously used for element A’s ->next pointer is now occupied by a floating-point number.
4. CPU 0 picks up what used to be element A’s ->next pointer, gets random bits, and therefore gets a segmentation fault.

One way to protect against this sort of problem requires use of “type-safe memory”, which will be discussed in Section 8.3.3.6. But in that case, you would be using some other synchronization mechanism in addition to sequence locks!

Quick Quiz 8.8:
Why bother with the check on line 19 of read_seqbegin() in Figure 8.7? Given that a new writer could begin at any time, why not simply incorporate the check into line 31 of read_seqretry()?

Answer:
That would be a legitimate implementation. However, it would not save anything to move the check down to read_seqretry(): There would be roughly the same number of instructions. Furthermore, the reader’s accesses from its doomed read-side critical section could inflict overhead on the writer in the form of cache misses. We can avoid these cache misses by placing the check in read_seqbegin() as shown on line 19 of Figure 8.7.

Quick Quiz 8.9:
What prevents sequence-locking updaters from starving readers?

Answer:
Nothing. This is one of the weaknesses of sequence locking, and as a result, you should use sequence locking only in read-mostly situations. Unless of course read-side starvation is acceptable in your situation, in which case, go wild the sequence-locking updates!

Quick Quiz 8.10:
What if something else serializes writers, so that the lock is not needed?

Answer:
In this case, the ->lock field could be omitted, as it is in seqcount_t in the Linux kernel.

Quick Quiz 8.11:
Why isn’t seq on line 2 of Figure 8.7 unsigned rather than unsigned long? After all, if unsigned is good enough for the Linux kernel, shouldn’t it be good enough for everyone?

Answer:
Not at all. The Linux kernel has a number of special attributes that allow it to ignore the following sequence of events:

1. Thread 0 executes read_seqbegin(), picking up ->seq in line 17, noting that the value is even, and thus returning to the caller.
2. Thread 0 starts executing its read-side critical section, but is then preempted for a long time.
3. Other threads repeatedly invoke `write_seqlock()` and `write_sequnlock()`, until the value of `->seq` overflows back to the value that Thread 0 fetched.

4. Thread 0 resumes execution, completing its read-side critical section with inconsistent data.

5. Thread 0 invokes `read_seqretry()`, which incorrectly concludes that Thread 0 has seen a consistent view of the data protected by the sequence lock.

The Linux kernel uses sequence locking for things that are updated rarely, with time-of-day information being a case in point. This information is updated at most once per millisecond, so that seven weeks would be required to overflow the counter. If a kernel thread was preempted for seven weeks, the Linux kernel’s soft-lockup code would be emitting warnings every two minutes for that entire time.

In contrast, with a 64-bit counter, more than five centuries would be required to overflow, even given an update every nanosecond. Therefore, this implementation uses a type for `->seq` that is 64 bits on 64-bit systems.

Quick Quiz 8.12:
But doesn’t Section 8.2’s seqlock also permit readers and updaters to get work done concurrently?

Answer:
Yes and no. Although seqlock readers can run concurrently with seqlock writers, whenever this happens, the `read_seqretry()` primitive will force the reader to retry. This means that any work done by a seqlock reader running concurrently with a seqlock updater will be discarded and redone. So seqlock readers can run concurrently with updaters, but they cannot actually get any work done in this case.

In contrast, RCU readers can perform useful work even in presence of concurrent RCU updaters.

Quick Quiz 8.13:
What prevents the `list_for_each_entry_rcu()` from getting a segfault if it happens to execute at exactly the same time as the `list_add_rcu()`?

Answer:
On all systems running Linux, loads from and stores to pointers are atomic, that is, if a store to a pointer occurs at the same time as a load from that same pointer, the load will return either the initial value or the value stored, never some bitwise mashup of the two. In addition, the `list_for_each_entry_rcu()` always proceeds forward through the list, never looking back. Therefore, the `list_for_each_entry_rcu()` will either see the element being added by `list_add_rcu()` or it will not, but either way, it will see a valid well-formed list.

Quick Quiz 8.14:
Why do we need to pass two pointers into `hlist_for_each_entry_rcu()` when only one is needed for `list_for_each_entry_rcu()`?

Answer:
Because in an hlist it is necessary to check for NULL rather than for encountering the head. (Try coding up a single-pointer `hlist_for_each_entry_rcu()` If you come up with a nice solution, it would be a very good thing!)

Quick Quiz 8.15:
How would you modify the deletion example to permit more than two versions of the list to be active?

Answer:
One way of accomplishing this is as shown in Figure G.4.

```
1 spin_lock(&mylock);
2 p = search(head, key);
3 if (p == NULL)
4   spin_unlock(&mylock);
5 else {
6   list_del_rcu(&p->list);
7   spin_unlock(&mylock);
8   synchronize_rcu();
9   kfree(p);
10 }
```

Figure G.4: Concurrent RCU Deletion

Note that this means that multiple concurrent deletions might be waiting in `synchronize_rcu()`.
Quick Quiz 8.16:
How many RCU versions of a given list can be active at any given time? ■

Answer:
That depends on the synchronization design. If a semaphore protecting the update is held across the grace period, then there can be at most two versions, the old and the new.

However, if only the search, the update, and the list_replace_rcu() were protected by a lock, then there could be an arbitrary number of versions active, limited only by memory and by how many updates could be completed within a grace period. But please note that data structures that are updated so frequently probably are not good candidates for RCU. That said, RCU can handle high update rates when necessary.

Quick Quiz 8.17:
How can RCU updaters possibly delay RCU readers, given that the rcu_read_lock() and rcu_read_unlock() primitives neither spin nor block? ■

Answer:
The modifications undertaken by a given RCU updater will cause the corresponding CPU to invalidate cache lines containing the data, forcing the CPUs running concurrent RCU readers to incur expensive cache misses. (Can you design an algorithm that changes a data structure without inflicting expensive cache misses on concurrent readers? On subsequent readers?)

Quick Quiz 8.18:
WTF? How the heck do you expect me to believe that RCU has a 100-femtosecond overhead when the clock period at 3GHz is more than 300 picoseconds? ■

Answer:
First, consider that the inner loop used to take this measurement is as follows:

```c
for (i = 0; i < CSCOUNT_SCALE; i++) { 
  rcu_read_lock(); 
  rcu_read_unlock(); 
}
```

Next, consider the effective definitions of rcu_read_lock() and rcu_read_unlock():

```c
#define rcu_read_lock() do {} while (0) 
#define rcu_read_unlock() do {} while (0)
```

Consider also that the compiler does simple optimizations, allowing it to replace the loop with:

```c
i = CSCOUNT_SCALE;
```

So the "measurement" of 100 femtoseconds is simply the fixed overhead of the timing measurements divided by the number of passes through the inner loop containing the calls to rcu_read_lock() and rcu_read_unlock(). And therefore, this measurement really is in error, in fact, in error by an arbitrary number of orders of magnitude. As you can see by the definition of rcu_read_lock() and rcu_read_unlock() above, the actual overhead is precisely zero.

It certainly is not every day that a timing measurement of 100 femtoseconds turns out to be an overestimate!

Quick Quiz 8.19:
Why does both the variability and overhead of rwlock decrease as the critical-section overhead increases? ■

Answer:
Because the contention on the underlying rwlock_t decreases as the critical-section overhead increases. However, the rwlock overhead will not quite drop to that on a single CPU because of cache-thrashing overhead.

Quick Quiz 8.20:
Is there an exception to this deadlock immunity, and if so, what sequence of events could lead to deadlock? ■

Answer:
One way to cause a deadlock cycle involving RCU read-side primitives is via the following (illegal) sequence of statements:

```c
idx = srcu_read_lock(&srcucb);
synchronize_srcu(&srcucb);
srcu_read_unlock(&srcucb, idx);
```

The synchronize_srcu() cannot return until all pre-existing SRCU read-side critical sections complete, but is enclosed in an SRCU read-side critical section that cannot complete until the synchronize_srcu() returns. The result is a classic self-deadlock—you get the
same effect when attempting to write-acquire a reader-writer lock while read-holding it.

Note that this self-deadlock scenario does not apply to RCU Classic, because the context switch performed by the `synchronize_rcu()` would act as a quiescent state for this CPU, allowing a grace period to complete. However, this is if anything even worse, because data used by the RCU read-side critical section might be freed as a result of the grace period completing.

In short, do not invoke synchronous RCU update-side primitives from within an RCU read-side critical section.

**Quick Quiz 8.21:**
But wait! This is exactly the same code that might be used when thinking of RCU as a replacement for reader-writer locking! What gives?

**Answer:**
This is an effect of the Law of Toy Examples: beyond a certain point, the code fragments look the same. The only difference is in how we think about the code. However, this difference can be extremely important. For but one example of the importance, consider that if we think of RCU as a restricted reference counting scheme, we would never be fooled into thinking that the updates would exclude the RCU read-side critical sections.

It nevertheless is often useful to think of RCU as a replacement for reader-writer locking, for example, when you are replacing reader-writer locking with RCU.

**Quick Quiz 8.22:**
Why the dip in refcnt overhead near 6 CPUs?

**Answer:**
Most likely NUMA effects. However, there is substantial variance in the values measured for the refcnt line, as can be seen by the error bars. In fact, standard deviations range in excess of 10 values in some cases. The dip in overhead therefore might well be a statistical aberration.

**Quick Quiz 8.23:**
What if the element we need to delete is not the first element of the list on line 9 of Figure 8.30?

**Answer:**
As with Figure 6.15, this is a very simple hash table with no chaining, so the only element in a given bucket is the first element. The reader is again invited to adapt this example to a hash table with full chaining.

**Quick Quiz 8.24:**
Why is it OK to exit the RCU read-side critical section on line 15 of Figure 8.30 before releasing the lock on line 17?

**Answer:**
First, please note that the second check on line 14 is necessary because some other CPU might have removed this element while we were waiting to acquire the lock. However, the fact that we were in an RCU read-side critical section while acquiring the lock guarantees that this element could not possibly have been re-allocated and re-inserted into this hash table. Furthermore, once we acquire the lock, the lock itself guarantees the element’s existence, so we no longer need to be in an RCU read-side critical section.

The question as to whether it is necessary to re-check the element’s key is left as an exercise to the reader.

**Quick Quiz 8.25:**
Why not exit the RCU read-side critical section on line 23 of Figure 8.30 before releasing the lock on line 22?

**Answer:**
Suppose we reverse the order of these two lines. Then this code is vulnerable to the following sequence of events:

1. CPU 0 invokes `delete()`, and finds the element to be deleted, executing through line 15. It has not yet actually deleted the element, but is about to do so.
2. CPU 1 concurrently invokes `delete()`, attempting to delete this same element. However, CPU 0 still holds the lock, so CPU 1 waits for it at line 13.
3. CPU 0 executes lines 16 and 17, and blocks at line 18 waiting for CPU 1 to exit its RCU read-side critical section.
4. CPU 1 now acquires the lock, but the test on line 14 fails because CPU 0 has already removed the element. CPU 1 now executes line 22 (which we switched with line 23 for the purposes of this Quick Quiz) and exits its RCU read-side critical section.
5. CPU 0 can now return from `synchronize_rcu()`, and thus executes line 19, sending the element to the freelist.

6. CPU 1 now attempts to release a lock for an element that has been freed, and, worse yet, possibly reallocated as some other type of data structure. This is a fatal memory-corruption error.

**Quick Quiz 8.26:**
But what if there is an arbitrarily long series of RCU read-side critical sections in multiple threads, so that at any point in time there is at least one thread in the system executing in an RCU read-side critical section? Wouldn’t that prevent any data from a `SLAB_DESTROY_BY_RCU` slab ever being returned to the system, possibly resulting in OOM events? ■

**Answer:**
There could certainly be an arbitrarily long period of time during which at least one thread is always in an RCU read-side critical section. However, the key words in the description in Section 8.3.3.6 are “in-use” and “pre-existing”. Keep in mind that a given RCU read-side critical section is conceptually only permitted to gain references to data elements that were in use at the beginning of that critical section. Furthermore, remember that a slab cannot be returned to the system until all of its data elements have been freed, in fact, the RCU grace period cannot start until after they have all been freed.

Therefore, the slab cache need only wait for those RCU read-side critical sections that started before the freeing of the last element of the slab. This in turn means that any RCU grace period that begins after the freeing of the last element will do—the slab may be returned to the system after that grace period ends.

**Quick Quiz 8.27:**
Suppose that the `nmi_profile()` function was preemptible. What would need to change to make this example work correctly? ■

**Answer:**
One approach would be to use `rcu_read_lock()` and `rcu_read_unlock()` in `nmi_profile()`, and to replace the `synchronize_sched()` with `synchronize_rcu()`, perhaps as shown in Figure G.5.

```c
1 struct profile_buffer {
2    long size;
3    atomic_t entry[0];
4};
5 static struct profile_buffer *buf = NULL;
6
7 void nmi_profile(unsigned long pcvalue)
8 {
9    struct profile_buffer *p;
10
11    rcu_read_lock();
12    p = rcu_dereference(buf);
13    if (p == NULL) {
14        rcu_read_unlock();
15        return;
16    }
17    if (pcvalue >= p->size) {
18        rcu_read_unlock();
19        return;
20    }
21    atomic_inc(&p->entry[pcvalue]);
22    rcu_read_unlock();
23)
24
25    void nmi_stop(void)
26 {
27        struct profile_buffer *p = buf;
28
29        if (p == NULL)
30            return;
31        rcu_assign_pointer(buf, NULL);
32        synchronize_rcu();
33        kfree(p);
34    }
```

Figure G.5: Using RCU to Wait for Mythical Preemptible NMIs to Finish
Quick Quiz 8.28:
Why do some of the cells in Table 8.4 have exclamation marks ("!")?

Answer:
The API members with exclamation marks (rcu_read_lock(), rcu_read_unlock(), and call_rcu()) were the only members of the Linux RCU API that Paul E. McKenney was aware of back in the mid-90s. During this timeframe, he was under the mistaken impression that he knew all that there is to know about RCU.

Quick Quiz 8.29:
How do you prevent a huge number of RCU read-side critical sections from indefinitely blocking a synchronize_rcu() invocation?

Answer:
There is no need to do anything to prevent RCU read-side critical sections from indefinitely blocking a synchronize_rcu() invocation, because the synchronize_rcu() invocation need wait only for pre-existing RCU read-side critical sections. So as long as each RCU read-side critical section is of finite duration, there should be no problem.

Quick Quiz 8.30:
The synchronize_rcu() API waits for all pre-existing interrupt handlers to complete, right?

Answer:
Absolutely not! And especially not when using preemptible RCU! You instead want synchronize_irq(). Alternatively, you can place calls to rcu_read_lock() and rcu_read_unlock() in the specific interrupt handlers that you want synchronize_rcu() to wait for.

Quick Quiz 8.31:
What happens if you mix and match? For example, suppose you use rcu_read_lock() and rcu_read_unlock() to delimit RCU read-side critical sections, but then use call_rcu_bh() to post an RCU callback?

Answer:
If there happened to be no RCU read-side critical sections delimited by rcu_read_lock_bh() and rcu_read_unlock_bh() at the time call_rcu_bh() was invoked, RCU would be within its rights to invoke the callback immediately, possibly freeing a data structure still being used by the RCU read-side critical section! This is not merely a theoretical possibility: a long-running RCU read-side critical section delimited by rcu_read_lock() and rcu_read_unlock() is vulnerable to this failure mode.

This vulnerability disappears in -rt kernels, where RCU Classic and RCU BH both map onto a common implementation.

Quick Quiz 8.32:
Hardware interrupt handlers can be thought of as being under the protection of an implicit rcu_read_lock_bh(), right?

Answer:
Absolutely not! And especially not when using preemptible RCU! If you need to access "rcu_bh"-protected data structures in an interrupt handler, you need to provide explicit calls to rcu_read_lock_bh() and rcu_read_unlock_bh().

Quick Quiz 8.33:
What happens if you mix and match RCU Classic and RCU Sched?

Answer:
In a non-PREEMPT or a PREEMPT kernel, mixing these two works "by accident" because in those kernel builds, RCU Classic and RCU Sched map to the same implementation. However, this mixture is fatal in PREEMPT_RT builds using the -rt patchset, due to the fact that Realtime RCU's read-side critical sections can be preempted, which would permit synchronize_sched() to return before the RCU read-side critical section reached its rcu_read_unlock() call. This could in turn result in a data structure being freed before the read-side critical section was finished with it, which could in turn greatly increase the actuarial risk experienced by your kernel.

In fact, the split between RCU Classic and RCU Sched...
was inspired by the need for preemptible RCU read-side critical sections.

Quick Quiz 8.34:
In general, you cannot rely on synchronize_sched() to wait for all pre-existing interrupt handlers, right?

Answer:
That is correct! Because -rt Linux uses threaded interrupt handlers, there can be context switches in the middle of an interrupt handler. Because synchronize_sched() waits only until each CPU has passed through a context switch, it can return before a given interrupt handler completes.

If you need to wait for a given interrupt handler to complete, you should instead use synchronize_irq() or place explicit RCU read-side critical sections in the interrupt handlers that you wish to wait on.

Quick Quiz 8.35:
Why do both SRCU and QRCU lack asynchronous call_srcu() or call_qrcu() interfaces?

Answer:
Given an asynchronous interface, a single task could register an arbitrarily large number of SRCU or QRCU callbacks, thereby consuming an arbitrarily large quantity of memory. In contrast, given the current synchronous synchronize_srcu() and synchronize_qrcu() interfaces, a given task must finish waiting for a given grace period before it can start waiting for the next one.

Quick Quiz 8.36:
Under what conditions can synchronize_srcu() be safely used within an SRCU read-side critical section?

Answer:
In principle, you can use synchronize_srcu() with a given srcu_struct within an SRCU read-side critical section that uses some other srcu_struct. In practice, however, doing this is almost certainly a bad idea. In particular, the code shown in Figure G.6 could still result in deadlock.

Quick Quiz 8.37:
Why doesn't list_del_rcu() poison both the next and prev pointers?

Answer:
Poisoning the next pointer would interfere with concurrent RCU readers, who must use this pointer. However, RCU readers are forbidden from using the prev pointer, so it may safely be poisoned.

Quick Quiz 8.38:
Normally, any pointer subject to rcu_dereference() must always be updated using rcu_assign_pointer(). What is an exception to this rule?

Answer:
One such exception is when a multi-element linked data structure is initialized as a unit while inaccessible to other CPUs, and then a single rcu_assign_pointer() is used to plant a global pointer to this data structure. The initialization-time pointer assignments need not use rcu_assign_pointer(), though any such assignments that happen after the structure is globally visible must use rcu_assign_pointer().

However, unless this initialization code is on an impressively hot code-path, it is probably wise to use rcu_assign_pointer() anyway, even though it is in theory unnecessary. It is all too easy for a "minor" change to invalidate your cherished assumptions about the initialization happening privately.
Quick Quiz 8.39:
Are there any downsides to the fact that these traversal and update primitives can be used with any of the RCU API family members?

Answer:
It can sometimes be difficult for automated code checkers such as “sparse” (or indeed for human beings) to work out which type of RCU read-side critical section a given RCU traversal primitive corresponds to. For example, consider the code shown in Figure G.7.

```c
1 rcu_read_lock();
2 preempt_disable();
3 p = rcu_dereference(global_pointer);
4 /* . . . */
5 7 preempt_enable();
6 rcu_read_unlock();
```

Figure G.7: Diverse RCU Read-Side Nesting

Is the `rcu_dereference()` primitive in an RCU Classic or an RCU Sched critical section? What would you have to do to figure this out?

Quick Quiz 8.40:
Why wouldn’t any deadlock in the RCU implementation in Figure 8.33 also be a deadlock in any other RCU implementation?

Answer:
Suppose the functions `foo()` and `bar()` in Figure G.8 are invoked concurrently from different CPUs. Then `foo()` will acquire `my_lock()` on line 3, while `bar()` will acquire `rcu_gp_lock` on line 13. When `foo()` advances to line 4, it will attempt to acquire `rcu_gp_lock`, which is held by `bar()`. Then when `bar()` advances to line 14, it will attempt to acquire `my_lock`, which is held by `foo()`.

Each function is then waiting for a lock that the other holds, a classic deadlock.

Other RCU implementations neither spin nor block in `rcu_read_lock()`, hence avoiding deadlocks.

```c
1 void foo(void)
2 {
3     spin_lock(&my_lock);
4     rcu_read_lock();
5     do_something();
6     rcu_read_unlock();
7     do_something_else();
8     spin_unlock(&my_lock);
9 }
10
11 void bar(void)
12 {
13     rcu_read_lock();
14     spin_lock(&my_lock);
15     do_some_other_thing();
16     spin_unlock(&my_lock);
17     do_whatever();
18     rcu_read_unlock();
19 }
```

Figure G.8: Deadlock in Lock-Based RCU Implementation

Quick Quiz 8.41:
Why not simply use reader-writer locks in the RCU implementation in Figure 8.33 in order to allow RCU readers to proceed in parallel?

Answer:
One could in fact use reader-writer locks in this manner. However, textbook reader-writer locks suffer from memory contention, so that the RCU read-side critical sections would need to be quite long to actually permit parallel execution [McK03].

On the other hand, use of a reader-writer lock that is read-acquired in `rcu_read_lock()` would avoid the deadlock condition noted above.

Quick Quiz 8.42:
Wouldn’t it be cleaner to acquire all the locks, and then release them all in the loop from lines 15-18 of Figure 8.34? After all, with this change, there would be a point in time when there were no readers, simplifying things greatly.

Answer:
Making this change would re-introduce the deadlock, so no, it would not be cleaner.

Quick Quiz 8.43:
Is the implementation shown in Figure 8.34 free from deadlocks? Why or why not?
Answer:
One deadlock is where a lock is held across `synchronize_rcu()`, and that same lock is acquired within an RCU read-side critical section. However, this situation will deadlock any correctly designed RCU implementation. After all, the `synchronize_rcu()` primitive must wait for all pre-existing RCU read-side critical sections to complete, but if one of those critical sections is spinning on a lock held by the thread executing the `synchronize_rcu()`, we have a deadlock inherent in the definition of RCU.

Another deadlock happens when attempting to nest RCU read-side critical sections. This deadlock is peculiar to this implementation, and might be avoided by using recursive locks, or by using reader-writer locks that are read-acquired by `rcu_read_lock()` and write-acquired by `synchronize_rcu()`.

However, if we exclude the above two cases, this implementation of RCU does not introduce any deadlock situations. This is because only time some other thread’s lock is acquired is when executing `synchronize_rcu()`, and in that case, the lock is immediately released, prohibiting a deadlock cycle that does not involve a lock held across the `synchronize_rcu()` which is the first case above.

Quick Quiz 8.44:
Isn’t one advantage of the RCU algorithm shown in Figure 8.34 that it uses only primitives that are widely available, for example, in POSIX pthreads?

Answer:
This is indeed an advantage, but do not forget that `rcu_dereference()` and `rcu_assign_pointer()` are still required, which means volatile manipulation for `rcu_dereference()` and memory barriers for `rcu_assign_pointer()`. Of course, many Alpha CPUs require memory barriers for both primitives.

Quick Quiz 8.45:
But what if you hold a lock across a call to `synchronize_rcu()`, and then acquire that same lock within an RCU read-side critical section?

Answer:
Indeed, this would deadlock any legal RCU implementation. But is `rcu_read_lock()` really participating in the deadlock cycle? If you believe that it is, then please ask yourself this same question when looking at the RCU implementation in Section 8.3.5.9.

Quick Quiz 8.46:
How can the grace period possibly elapse in 40 nanoseconds when `synchronize_rcu()` contains a 10-millisecond delay?

Answer:
The update-side test was run in absence of readers, so the `poll()` system call was never invoked. In addition, the actual code has this `poll()` system call commented out, the better to evaluate the true overhead of the update-side code. Any production uses of this code would be better served by using the `poll()` system call, but then again, production uses would be even better served by other implementations shown later in this section.

Quick Quiz 8.47:
Why not simply make `rcu_read_lock()` wait when a concurrent `synchronize_rcu()` has been waiting too long in the RCU implementation in Figure 8.35? Wouldn’t that prevent `synchronize_rcu()` from starving?

Answer:
Although this would in fact eliminate the starvation, it would also mean that `rcu_read_lock()` would spin or block waiting for the writer, which is in turn waiting on readers. If one of these readers is attempting to acquire a lock that the spinning/blocking `rcu_read_lock()` holds, we again have deadlock.

In short, the cure is worse than the disease. See Section 8.3.5.4 for a proper cure.

Quick Quiz 8.48:
Why the memory barrier on line 5 of `synchronize_rcu()` in Figure 8.38 given that there is a spin-lock acquisition immediately after?

Answer:
The spin-lock acquisition only guarantees that the spin-lock’s critical section will not “bleed out” to precede the acquisition. It in no way guarantees that code preceding the spin-lock acquisition won’t be reordered
into the critical section. Such reordering could cause a removal from an RCU-protected list to be reordered to follow the complementing of rcu_idx, which could allow a newly starting RCU read-side critical section to see the recently removed data element.

Exercise for the reader: use a tool such as Promela/spin to determine which (if any) of the memory barriers in Figure 8.38 are really needed. See Section F for information on using these tools. The first correct and complete response will be credited.

Quick Quiz 8.49:
Why is the counter flipped twice in Figure 8.38? Shouldn’t a single flip-and-wait cycle be sufficient?

Answer:
Both flips are absolutely required. To see this, consider the following sequence of events:

1. Line 8 of rcu_read_lock() in Figure 8.37 picks up rcu_idx, finding its value to be zero.
2. Line 8 of synchronize_rcu() in Figure 8.38 complements the value of rcu_idx, setting its value to one.
3. Lines 10-13 of synchronize_rcu() find that the value of rcu_refcnt[0] is zero, and thus returns. (Recall that the question is asking what happens if lines 14-20 are omitted.)
4. Lines 9 and 10 of rcu_read_lock() store the value zero to this thread’s instance of rcu_read_idx and increments rcu_refcnt[0], respectively. Execution then proceeds into the RCU read-side critical section.
5. Another instance of synchronize_rcu() again complements rcu_idx, this time setting its value to zero. Because rcu_refcnt[1] is zero, synchronize_rcu() returns immediately. (Recall that rcu_read_lock() incremented rcu_refcnt[0], not rcu_refcnt[1]!)
6. The grace period that started in step 5 has been allowed to end, despite the fact that the RCU read-side critical section that started beforehand in step 4 has not completed. This violates RCU semantics, and could allow the update to free a data element that the RCU read-side critical section was still referencing.

Exercise for the reader: What happens if rcu_read_lock() is preempted for a very long time (hours!) just after line 8? Does this implementation operate correctly in that case? Why or why not? The first correct and complete response will be credited.

Quick Quiz 8.50:
Given that atomic increment and decrement are so expensive, why not just use non-atomic increment on line 10 and a non-atomic decrement on line 25 of Figure 8.37?

Answer:
Using non-atomic operations would cause increments and decrements to be lost, in turn causing the implementation to fail. See Section 8.3.5.5 for a safe way to use non-atomic operations in rcu_read_lock() and rcu_read_unlock().

Quick Quiz 8.51:
Come off it! We can see the atomic_read() primitive in rcu_read_lock()!!! So why are you trying to pretend that rcu_read_lock() contains no atomic operations???

Answer:
The atomic_read() primitives does not actually execute atomic machine instructions, but rather does a normal load from an atomic_t. Its sole purpose is to keep the compiler’s type-checking happy. If the Linux kernel ran on 8-bit CPUs, it would also need to prevent “store tearing”, which could happen due to the need to store a 16-bit pointer with two eight-bit accesses on some 8-bit systems. But thankfully, it seems that no one runs Linux on 8-bit systems.

Quick Quiz 8.52:
Great, if we have N threads, we can have 2N ten-millisecond waits (one set per flip_counter_and_wait() invocation, and even that assumes that we wait only once for each thread. Don’t we need the grace period to complete much more quickly?

Answer:
Keep in mind that we only wait for a given thread if that thread is still in a pre-existing RCU read-side critical
section, and that waiting for one hold-out thread gives all the other threads a chance to complete any pre-existing RCU read-side critical sections that they might still be executing. So the only way that we would wait for 2N intervals would be if the last thread still remained in a pre-existing RCU read-side critical section despite all the waiting for all the prior threads. In short, this implementation will not wait unnecessarily.

However, if you are stress-testing code that uses RCU, you might want to comment out the poll() statement in order to better catch bugs that incorrectly retain a reference to an RCU-protected data element outside of an RCU read-side critical section.

Quick Quiz 8.53:
All of these toy RCU implementations have either atomic operations in rcu_read_lock() and rcu_read_unlock(), or synchronize_rcu() overhead that increases linearly with the number of threads. Under what circumstances could an RCU implementation enjoy light-weight implementations for all three of these primitives, all having deterministic (O(1)) overheads and latencies?

Answer:
Special-purpose uniprocessor implementations of RCU can attain this ideal [McK09c].

Quick Quiz 8.54:
If any even value is sufficient to tell synchronize_rcu() to ignore a given task, why doesn’t line 10 of Figure 8.46 simply assign zero to rcu_reader_gp?

Answer:
Assigning zero (or any other even-numbered constant) would in fact work, but assigning the value of rcu_gp_ctr can provide a valuable debugging aid, as it gives the developer an idea of when the corresponding thread last exited an RCU read-side critical section.

Quick Quiz 8.55:
Why are the memory barriers on lines 17 and 29 of Figure 8.46 needed? Aren’t the memory barriers inherent in the locking primitives on lines 18 and 28 sufficient?

Answer:
These memory barriers are required because the locking primitives are only guaranteed to confine the critical section. The locking primitives are under absolutely no obligation to keep other code from bleeding in to the critical section. The pair of memory barriers are therefore required to prevent this sort of code motion, whether performed by the compiler or by the CPU.

Quick Quiz 8.56:
Couldn’t the update-side optimization described in Section 8.3.5.6 be applied to the implementation shown in Figure 8.46?

Answer:
Indeed it could, with a few modifications. This work is left as an exercise for the reader.

Quick Quiz 8.57:
Is the possibility of readers being preempted in line 3 of Figure 8.46 a real problem, in other words, is there a real sequence of events that could lead to failure? If not, why not? If so, what is the sequence of events, and how can the failure be addressed?

Answer:
It is a real problem, there is a sequence of events leading to failure, and there are a number of possible ways of addressing it. For more details, see the Quick Quizzes near the end of Section 8.3.5.8. The reason for locating the discussion there is to (1) give you more time to think about it, and (2) because the nesting support added in that section greatly reduces the time required to overflow the counter.

Quick Quiz 8.58:
Why not simply maintain a separate per-thread nesting-level variable, as was done in previous section, rather than having all this complicated bit manipulation?

Answer:
The apparent simplicity of the separate per-thread variable is a red herring. This approach incurs much greater complexity in the guise of careful ordering of operations, especially if signal handlers are to be permitted to contain RCU read-side critical sections. But don’t take my word for it, code it up and see what you
Quick Quiz 8.59:
Given the algorithm shown in Figure 8.48, how could you double the time required to overflow the global rcu_gp_ctr? ■

Answer:
One way would be to replace the magnitude comparison on lines 33 and 34 with an inequality check of the per-thread rcu_reader_gp variable against rcu_gp_ctr+RCU_GP_CTR_BOTTOM_BIT.

Quick Quiz 8.60:
Again, given the algorithm shown in Figure 8.48, is counter overflow fatal? Why or why not? If it is fatal, what can be done to fix it? ■

Answer:
It can indeed be fatal. To see this, consider the following sequence of events:

1. Thread 0 enters rcu_read_lock(), determines that it is not nested, and therefore fetches the value of the global rcu_gp_ctr. Thread 0 is then preempted for an extremely long time (before storing to its per-thread rcu_reader_gp variable).

2. Other threads repeatedly invoke synchronize_rcu(), so that the new value of the global rcu_gp_ctr is now RCU_GP_CTR_BOTTOM_BIT less than it was when thread 0 fetched it.

3. Thread 0 now starts running again, and stores into its per-thread rcu_reader_gp variable. The value it stores is RCU_GP_CTR_BOTTOM_BIT+1 greater than that of the global rcu_gp_ctr.

4. Thread 0 acquires a reference to RCU-protected data element A.

5. Thread 1 now removes the data element A that thread 0 just acquired a reference to.

6. Thread 1 invokes synchronize_rcu(), which increments the global rcu_gp_ctr by RCU_GP_CTR_BOTTOM_BIT. It then checks all of the per-thread rcu_reader_gp variables, but thread 0’s value (incorrectly) indicates that it started after thread 1’s call to synchronize_rcu(), so thread 1 does not wait for thread 0 to complete its RCU read-side critical section.

7. Thread 1 then frees up data element A, which thread 0 is still referencing.

Note that scenario can also occur in the implementation presented in Section 8.3.5.7.

One strategy for fixing this problem is to use 64-bit counters so that the time required to overflow them would exceed the useful lifetime of the computer system. Note that non-antique members of the 32-bit x86 CPU family allow atomic manipulation of 64-bit counters via the cmpxchg64b instruction.

Another strategy is to limit the rate at which grace periods are permitted to occur in order to achieve a similar effect. For example, synchronize_rcu() could record the last time that it was invoked, and any subsequent invocation would then check this time and block as needed to force the desired spacing. For example, if the low-order four bits of the counter were reserved for nesting, and if grace periods were permitted to occur at most ten times per second, then it would take more than 300 days for the counter to overflow. However, this approach is not helpful if there is any possibility that the system will be fully loaded with CPU-bound high-priority real-time threads for the full 300 days. (A remote possibility, perhaps, but best to consider it ahead of time.)

A third approach is to administratively abolish real-time threads from the system in question. In this case, the preempted process will age up in priority, thus getting to run long before the counter had a chance to overflow. Of course, this approach is less than helpful for real-time applications.

A final approach would be for rcu_read_lock() to recheck the value of the global rcu_gp_ctr after storing to its per-thread rcu_reader_gp counter, retrying if the new value of the global rcu_gp_ctr is inappropriate. This works, but introduces non-deterministic execution time into rcu_read_lock(). On the other hand, if your application is being preempted long enough for the counter to overflow, you have no hope of deterministic execution time in any case!

Quick Quiz 8.61:
Doesn’t the additional memory barrier shown on line 14 of Figure 8.50, greatly increase the overhead of rcu_quiescent_state? ■
Answer:
Indeed it does! An application using this implementation of RCU should therefore invoke `rcu_quiescent_state` sparingly, instead using `rcu_read_lock()` and `rcu_read_unlock()` most of the time.

However, this memory barrier is absolutely required so that other threads will see the store on lines 12-13 before any subsequent RCU read-side critical sections executed by the caller.

Quick Quiz 8.62:
Why are the two memory barriers on lines 19 and 22 of Figure 8.50 needed? ■

Answer:
The memory barrier on line 19 prevents any RCU read-side critical sections that might precede the call to `rcu_thread_offline()` won't be reordered by either the compiler or the CPU to follow the assignment on lines 20-21. The memory barrier on line 22 is, strictly speaking, unnecessary, as it is illegal to have any RCU read-side critical sections following the call to `rcu_thread_offline()`.

Quick Quiz 8.63:
To be sure, the clock frequencies of ca-2008 Power systems were quite high, but even a 5GHz clock frequency is insufficient to allow loops to be executed in 50 picoseconds! What is going on here? ■

Answer:
Since the measurement loop contains a pair of empty functions, the compiler optimizes it away. The measurement loop takes 1,000 passes between each call to `rcu_quiescent_state()`, so this measurement is roughly one thousandth of the overhead of a single call to `rcu_quiescent_state()`.

Quick Quiz 8.64:
Why would the fact that the code is in a library make any difference for how easy it is to use the RCU implementation shown in Figures 8.50 and 8.51? ■

Answer:
A library function has absolutely no control over the caller, and thus cannot force the caller to invoke `rcu_quiescent_state()` periodically. On the other hand, a library function that made many references to a given RCU-protected data structure might be able to invoke `rcu_thread_online()` upon entry, `rcu_quiescent_state()` periodically, and `rcu_thread_offline()` upon exit.

Quick Quiz 8.65:
But what if you hold a lock across a call to `synchronize_rcu()`, and then acquire that same lock within an RCU read-side critical section? This should be a deadlock, but how can a primitive that generates absolutely no code possibly participate in a deadlock cycle? ■

Answer:
Please note that the RCU read-side critical section is in effect extended beyond the enclosing `rcu_read_lock()` and `rcu_read_unlock()`, out to the previous and next call to `rcu_quiescent_state()`. This `rcu_quiescent_state` can be thought of as a `rcu_read_unlock()` immediately followed by an `rcu_read_lock()`.

Even so, the actual deadlock itself will involve the lock acquisition in the RCU read-side critical section and the `synchronize_rcu()`, never the `rcu_quiescent_state()`.

Quick Quiz 8.66:
Given that grace periods are prohibited within RCU read-side critical sections, how can an RCU data structure possibly be updated while in an RCU read-side critical section? ■

Answer:
This situation is one reason for the existence of asynchronous grace-period primitives such as `call_rcu()`. This primitive may be invoked within an RCU read-side critical section, and the specified RCU callback will in turn be invoked at a later time, after a grace period has elapsed.

The ability to perform an RCU update while within an RCU read-side critical section can be extremely convenient, and is analogous to a (mythical) unconditional read-to-write upgrade for reader-writer locking.
Quick Quiz 8.67:
The statistical-counter implementation shown in Figure 4.8 (count_end.c) used a global lock to guard the summation in read_count(), which resulted in poor performance and negative scalability. How could you use RCU to provide read_count() with excellent performance and good scalability. (Keep in mind that read_count()’s scalability will necessarily be limited by its need to scan all threads’ counters.)

Answer:
Hint: place the global variable finalcount and the array counterp[] into a single RCU-protected struct. At initialization time, this structure would be allocated and set to all zero and NULL.

The inc_count() function would be unchanged.

The read_count() function would use rcu_read_lock() instead of acquiring final_mutex, and would need to use rcu_dereference() to acquire a reference to the current structure.

The count_register_thread() function would set the array element corresponding to the newly created thread to reference that thread’s per-thread counter variable.

The count_unregister_thread() function would need to allocate a new structure, acquire final_mutex, copy the old structure to the new one, add the outgoing thread’s counter variable to the total, NULL the pointer to this same counter variable, use rcu_assign_pointer() to install the new structure in place of the old one, release final_mutex, wait for a grace period, and finally free the old structure.

Does this really work? Why or why not?

Quick Quiz 8.68:
Section 4.5 showed a fanciful pair of code fragments that dealt with counting I/O accesses to removable devices. These code fragments suffered from high overhead on the fastpath (starting an I/O) due to the need to acquire a reader-writer lock. How would you use RCU to provide excellent performance and scalability? (Keep in mind that the performance of the common-case first code fragment that does I/O accesses is much more important than that of the device-removal code fragment.)

Answer:
Hint: replace the read-acquisitions of the reader-writer lock with RCU read-side critical sections, then adjust the device-removal code fragment to suit.

See Section 9.2 on Page 131 for one solution to this problem.

G.9 Applying RCU

Quick Quiz 9.1:
Why on earth did we need that global lock in the first place?

Answer:
A given thread’s __thread variables vanish when that thread exits. It is therefore necessary to synchronize any operation that accesses other threads’ __thread variables with thread exit. Without such synchronization, accesses to __thread variable of a just-exited thread will result in segmentation faults.

Quick Quiz 9.2:
Just what is the accuracy of read_count(), anyway?

Answer:
Refer to Figure 4.8 on Page 33. Clearly, if there are no concurrent invocations of inc_count(), read_count() will return an exact result. However, if there are concurrent invocations of inc_count(), then the sum is in fact changing as read_count() performs its summation. That said, because thread creation and exit are excluded by final_mutex, the pointers in counterp remain constant.

Let’s imagine a mythical machine that is able to take an instantaneous snapshot of its memory. Suppose that this machine takes such a snapshot at the beginning of read_count()’s execution, and another snapshot at the end of read_count()’s execution. Then read_count() will access each thread’s counter at some time between these two snapshots, and will therefore obtain a result that is bounded by those of the two snapshots, inclusive. The overall sum will therefore be bounded by the pair of sums that would have been obtained from each of the two snapshots (again, inclusive).

The expected error is therefore half of the difference between the pair of sums that would have been obtained from each of the two snapshots, that is to say, half of the execution time of read_count() multiplied by...
the number of expected calls to `inc_count()` per unit time.

Or, for those who prefer equations:

\[ \epsilon = \frac{T_r R_i}{2} \]  (G.1)

where \( \epsilon \) is the expected error in `read_count()`’s return value, \( T_r \) is the time that `read_count()` takes to execute, and \( R_i \) is the rate of `inc_count()` calls per unit time. (And of course, \( T_r \) and \( R_i \) should use the same units of time: microseconds and calls per microsecond, seconds and calls per second, or whatever, as long as they are the same units.)

Quick Quiz 9.3:
Hey!!! Line 45 of Figure 9.1 modifies a value in a pre-existing `countarray` structure! Didn’t you say that this structure, once made available to `read_count()`, remained constant???

Answer:
Indeed I did say that. And it would be possible to make `count_register_thread()` allocate a new structure, much as `count_unregister_thread()` currently does.

But this is unnecessary. Recall the derivation of the error bounds of `read_count()` that was based on the snapshots of memory. Because new threads start with initial counter values of zero, the derivation holds even if we add a new thread partway through `read_count()`’s execution. So, interestingly enough, when adding a new thread, this implementation gets the effect of allocating a new structure, but without actually having to do the allocation.

Quick Quiz 9.4:
Wow! Figure 9.1 contains 69 lines of code, compared to only 42 in Figure 4.8. Is this extra complexity really worth it???

Answer:
This of course needs to be decided on a case-by-case basis. If you need an implementation of `read_count()` that scales linearly, then the lock-based implementation shown in Figure 4.8 simply will not work for you. On the other hand, if calls to `count_read()` are sufficiently rare, then the lock-based version is simpler and might thus be better, although much of the size difference is due to the structure definition, memory allocation, and NULL return checking.

Of course, a better question is "why doesn’t the language implement cross-thread access to __thread variables?" After all, such an implementation would make both the locking and the use of RCU unnecessary. This would in turn enable an implementation that was even simpler than the one shown in Figure 4.8, but with all the scalability and performance benefits of the implementation shown in Figure 9.1!

G.10 Validation

Quick Quiz 10.1:
Suppose that you are writing a script that processes the output of the `time` command, which looks as follows:

```
real 0m0.132s
user 0m0.040s
sys 0m0.008s
```

The script is required to check its input for errors, and to give appropriate diagnostics if fed erroneous `time` output. What test inputs should you provide to this program to test it for use with `time` output generated by single-threaded programs???

Answer:

1. Do you have a test case in which all the time is consumed in user mode by a CPU-bound program?
2. Do you have a test case in which all the time is consumed in system mode by a CPU-bound program?
3. Do you have a test case in which all three times are zero?
4. Do you have a test case in which the “user” and “sys” times sum to more than the “real” time? (This would of course be completely legitimate in a multithreaded program.)
5. Do you have a set of tests cases in which one of the times uses more than one second?
6. Do you have a set of tests cases in which one of the times uses more than ten second?
APPENDIX G. ANSWERS TO QUICK QUIZZES

7. Do you have a set of test cases in which one of the times has non-zero minutes? (For example, “15m36.342s”.)

8. Do you have a set of test cases in which one of the times has a seconds value of greater than 60?

9. Do you have a set of test cases in which one of the times overflows 32 bits of milliseconds? 64 bits of milliseconds?

10. Do you have a set of test cases in which one of the times is negative?

11. Do you have a set of test cases in which one of the times has a positive minutes value but a negative seconds value?

12. Do you have a set of test cases in which one of the times omits the “m” or the “s”?

13. Do you have a set of test cases in which one of the times is non-numeric? (For example, “Go Fish”.)

14. Do you have a set of test cases in which one of the lines is omitted? (For example, where there is a “real” value and a “sys” value, but no “user” value.)

15. Do you have a set of test cases where one of the lines is duplicated? Or duplicated, but with a different time value for the duplicate?

16. Do you have a set of test cases where a given line has more than one time value? (For example, “real 0m0.132s 0m0.008s”.)

17. Do you have a set of test cases containing random characters?

18. In all test cases involving invalid input, did you generate all permutations?

19. For each test case, do you have an expected outcome for that test?

If you did not generate test data for a substantial number of the above cases, you will need to cultivate a more destructive attitude in order to have a chance of generating high-quality tests.

Of course, one way to economize on destructiveness is to generate the tests with the to-be-tested source code at hand, which is called white-box testing (as opposed to black-box testing). However, this is no panacea: You will find that it is all too easy to find your thinking limited by what the program can handle, thus failing to generate truly destructive inputs.

Quick Quiz 10.2:
Suppose that you had a very large number of systems at your disposal. For example, at current cloud prices, you can purchase a huge amount of CPU time at a reasonably low cost. Why not use this approach to get close enough to certainty for all practical purposes?

Answer:
This approach might well be a valuable addition to your validation arsenal. But it does have a few limitations:

1. Some bugs have extremely low probabilities of occurrence, but nevertheless need to be fixed. For example, suppose that the Linux kernel’s RCU implementation had a bug that is triggered only once per century of machine time on average. A century of CPU time is hugely expensive even on the cheapest cloud platforms, but we could expect this bug to result in more than 2,000 failures per day on the more than 100 million Linux instances in the world as of 2011.

2. The bug might well have zero probability of occurrence on your test setup, which means that you won’t see it no matter how much machine time you burn testing it.

Of course, if your code is small enough, formal validation may be helpful, as discussed in Appendix F.

Quick Quiz 10.3:
Say what??? When I plug the earlier example of five tests each with a 10% failure rate into the formula, I get 59.050% and that just doesn’t make sense!!!

Answer:
You are right, that makes no sense at all.

Remember that a probability is a number between zero and one, so that you need to divide a percentage by 100 to get a probability. So 10% is a probability of 0.1, which gets a probability of 0.4095, which rounds to 41%, which quite sensibly matches the earlier result.

Quick Quiz 10.4:
In Equation 10.7, are the logarithms base-10, base-2, or
Answer:
It does not matter. You will get the same answer no matter what base of logarithms you use because the result is a pure ratio of logarithms. The only constraint is that you use the same base for both the numerator and the denominator.

Quick Quiz 12.1:
How on earth could the assertion on line 21 of the code in Figure 12.3 on page 140 possibly fail?

Answer:
The key point is that the intuitive analysis missed is that there is nothing preventing the assignment to C from overtaking the assignment to A as both race to reach thread2(). This is explained in the remainder of this section.

Quick Quiz 12.2:
Great... So how do I fix it?

Answer:
The easiest fix is to replace the barrier() on line 12 with an smp_mb().

Quick Quiz 12.3:
What assumption is the code fragment in Figure 12.4 making that might not be valid on real hardware?

Answer:
The code assumes that as soon as a given CPU stops seeing its own value, it will immediately see the final agreed-upon value. On real hardware, some of the CPUs might well see several intermediate results before converging on the final value.

Quick Quiz 12.4:
How could CPUs possibly have different views of the value of a single variable at the same time?

Answer:
Many CPUs have write buffers that record the values of recent writes, which are applied once the corresponding cache line makes its way to the CPU. Therefore, it is quite possible for each CPU to see a different value for a given variable at a single point in time — and for main memory to hold yet another value. One of the reasons that memory barriers were invented was to allow software to deal gracefully with situations like this one.

Quick Quiz 12.5:
Why do CPUs 2 and 3 come to agreement so quickly, when it takes so long for CPUs 1 and 4 to come to the party?

Answer:
CPUs 2 and 3 are a pair of hardware threads on the same core, sharing the same cache hierarchy, and therefore have very low communications latencies. This is a NUMA, or, more accurately, a NUCA effect.

This leads to the question of why CPUs 2 and 3 ever disagree at all. One possible reason is that they each might have a small amount of private cache in addition to a larger shared cache. Another possible reason is instruction reordering, given the short 10-nanosecond duration of the disagreement and the total lack of memory barriers in the code fragment.

Quick Quiz 12.6:
But if the memory barriers do not unconditionally force ordering, how the heck can a device driver reliably execute sequences of loads and stores to MMIO registers?

Answer:
MMIO registers are special cases: because they appear in uncached regions of physical memory. Memory barriers do unconditionally force ordering of loads and stores to uncached memory. See Section @@@ for more information on memory barriers and MMIO regions.

Quick Quiz 12.7:
How could the assertion b==2 on page 145 possibly fail?

Answer:
If the CPU is not required to see all of its loads and stores
in order, then the $b = 1 + a$ might well see an old version of the variable “$a$”.

This is why it is so very important that each CPU or thread see all of its own loads and stores in program order.

Quick Quiz 12.8:
How could the code on page 145 possibly leak memory?

Answer:
Only the first execution of the critical section should see $p == \text{NULL}$. However, if there is no global ordering of critical sections for mylock, then how can you say that a particular one was first? If several different executions of that critical section thought that they were first, they would all see $p == \text{NULL}$, and they would all allocate memory. All but one of those allocations would be leaked.

This is why it is so very important that all the critical sections for a given exclusive lock appear to execute in some well-defined order.

Quick Quiz 12.9:
How could the code on page 145 possibly count backwards?

Answer:
Suppose that the counter started out with the value zero, and that three executions of the critical section had therefore brought its value to three. If the fourth execution of the critical section is not constrained to see the most recent store to this variable, it might well see the original value of zero, and therefore set the counter to one, which would be going backwards.

This is why it is so very important that loads from a given variable in a given critical section see the last store from the last prior critical section to store to that variable.

Quick Quiz 12.10:
What effect does the following sequence have on the order of stores to variables “$a$” and “$b$”?

```plaintext
a = 1;
b = 1;
<write barrier>
```

Answer:
Absolutely none. This barrier would ensure that the assignments to “$a$” and “$b$” happened before any subsequent assignments, but it does nothing to enforce any order of assignments to “$a$” and “$b$” themselves.

Quick Quiz 12.11:
What sequence of LOCK-UNLOCK operations would act as a full memory barrier?

Answer:
A series of two back-to-back LOCK-UNLOCK operations, or, somewhat less conventionally, an UNLOCK operation followed by a LOCK operation.

Quick Quiz 12.12:
What sequence of LOCK-UNLOCK operations would act as a full memory barrier?

Answer:
Itanium is one example. The identification of any others is left as an exercise for the reader.

Quick Quiz 12.13:
Given that operations grouped in curly braces are executed concurrently, which of the rows of Table 12.2 are legitimate reorderings of the assignments to variables “$A$” through “$F$” and the LOCK/UNLOCK operations? (The order in the code is A, B, LOCK, C, D, UNLOCK, E, F.) Why or why not?

Answer:
1. Legitimate, executed in order.
2. Legitimate, the lock acquisition was executed concurrently with the last assignment preceding the critical section.
3. Illegitimate, the assignment to “$F$” must follow the LOCK operation.
4. Illegitimate, the LOCK must complete before any operation in the critical section. However, the UNLOCK may legitimately be executed concurrently with subsequent operations.
5. Legitimate, the assignment to “A” precedes the UNLOCK, as required, and all other operations are in order.
6. Illegitimate, the assignment to “C” must follow the LOCK.
7. Illegitimate, the assignment to “D” must precede the UNLOCK.
8. Legitimate, all assignments are ordered with respect to the LOCK and UNLOCK operations.
9. Illegitimate, the assignment to “A” must precede the UNLOCK.

Quick Quiz 12.14: What are the constraints for Table 12.3?

Answer: All CPUs must see the following ordering constraints:
1. LOCK M precedes B, C, and D.
2. UNLOCK M follows A, B, and C.
3. LOCK Q precedes F, G, and H.
4. UNLOCK Q follows E, F, and G.

G.12 Ease of Use

Quick Quiz 13.1: Can a similar algorithm be used when deleting elements?

Answer: Yes. However, since each thread must hold the locks of three consecutive elements to delete the middle one, if there are \( N \) threads, there must be \( 2N + 1 \) elements (rather than just \( N + 1 \)) in order to avoid deadlock.

Quick Quiz 13.2: Yetch! What ever possessed someone to come up with an algorithm that deserves to be shaved as much as this one does???

Answer: That would be Paul.

He was considering the Dining Philosopher’s Problem, which involves a rather unsanitary spaghetti dinner attended by five philosophers. Given that there are five plates and but five forks on the table, and given that each philosopher requires two forks at a time to eat, one is supposed to come up with a fork-allocation algorithm that avoids deadlock. Paul’s response was “Sheesh! Just get five more forks!”.

This in itself was OK, but Paul then applied this same solution to circular linked lists.

This would not have been so bad either, but he had to go and tell someone about it!

Quick Quiz 13.3: Give an exception to this rule.

Answer: One exception would be a difficult and complex algorithm that was the only one known to work in a given situation. Another exception would be a difficult and complex algorithm that was nonetheless the simplest of the set known to work in a given situation. However, even in these cases, it may be very worthwhile to spend a little time trying to come up with a simpler algorithm! After all, if you managed to invent the first algorithm to do some task, it shouldn’t be that hard to go on to invent a simpler one.

G.13 Conflicting Visions of the Future

Quick Quiz 15.1: What about non-persistent primitives represented by data structures in \( 	ext{mmap}() \) regions of memory? What happens when there is an \( 	ext{exec}() \) within a critical section of such a primitive?

Answer: If the \( 	ext{exec}() \) ed program maps those same regions of memory, then this program could in principle simply release the lock. The question as to whether this approach is sound from a software-engineering viewpoint is left as an exercise for the reader.
APPENDIX G. ANSWERS TO QUICK QUIZZES

G.14 Important Questions

Quick Quiz A.1:
What SMP coding errors can you see in these examples? See time.c for full code.

Answer:
1. Missing barrier() or volatile on tight loops.
3. Lack of synchronization between producer and consumer.

Quick Quiz A.2:
How could there be such a large gap between successive consumer reads? See timelocked.c for full code.

Answer:
1. The consumer might be preempted for long time periods.
2. A long-running interrupt might delay the consumer.
3. The producer might also be running on a faster CPU than is the consumer (for example, one of the CPUs might have had to decrease its clock frequency due to heat-dissipation or power-consumption constraints).

G.15 Synchronization Primitives

Quick Quiz B.1:
Give an example of a parallel program that could be written without synchronization primitives.

Answer:
There are many examples. One of the simplest would be a parametric study using a single independent variable. If the program run_study took a single argument, then we could use the following bash script to run two instances in parallel, as might be appropriate on a two-CPU system:

```
run_study 1 > 1.out & run_study 2 > 2.out; wait
```

One could of course argue that the bash ampersand operator and the "wait" primitive are in fact synchronization primitives. If so, then consider that this script could be run manually in two separate command windows, so that the only synchronization would be supplied by the user himself or herself.

Quick Quiz B.2:
What problems could occur if the variable counter were incremented without the protection of mutex?

Answer:
On CPUs with load-store architectures, incrementing counter might compile into something like the following:

```
LOAD counter, r0
INC r0
STORE r0, counter
```

On such machines, two threads might simultaneously load the value of counter, each increment it, and each store the result. The new value of counter will then only be one greater than before, despite two threads each incrementing it.

Quick Quiz B.3:
How could you work around the lack of a per-thread-variable API on systems that do not provide it?

Answer:
One approach would be to create an array indexed by smp_thread_id(), and another would be to use a hash table to map from smp_thread_id() to an array index — which is in fact what this set of APIs does in pthread environments.

Another approach would be for the parent to allocate a structure containing fields for each desired per-thread variable, then pass this to the child during thread creation. However, this approach can impose large software-engineering costs in large systems. To see this, imagine if all global variables in a large system had to be declared in a single file, regardless of whether or not they were C static variables!
G.16 Why Memory Barriers?

Quick Quiz C.1:
What happens if two CPUs attempt to invalidate the same cache line concurrently? ■

Answer:
One of the CPUs gains access to the shared bus first, and that CPU “wins”. The other CPU must invalidate its copy of the cache line and transmit an “invalidate acknowledge” message to the other CPU. Of course, the losing CPU can be expected to immediately issue a “read invalidate” transaction, so the winning CPU’s victory will be quite ephemeral.

Quick Quiz C.2:
When an “invalidate” message appears in a large multiprocessor, every CPU must give an “invalidate acknowledge” response. Wouldn’t the resulting “storm” of “invalidate acknowledge” responses totally saturate the system bus? ■

Answer:
It might, if large-scale multiprocessors were in fact implemented that way. Larger multiprocessors, particularly NUMA machines, tend to use so-called “directory-based” cache-coherence protocols to avoid this and other problems.

Quick Quiz C.3:
If SMP machines are really using message passing anyway, why bother with SMP at all? ■

Answer:
There has been quite a bit of controversy on this topic over the past few decades. One answer is that the cache-coherence protocols are quite simple, and therefore can be implemented directly in hardware, gaining bandwidths and latencies unattainable by software message passing. Another answer is that the real truth is to be found in economics due to the relative prices of large SMP machines and that of clusters of smaller SMP machines. A third answer is that the SMP programming model is easier to use than that of distributed systems, but a rebuttal might note the appearance of HPC clusters and MPI. And so the argument continues.

Quick Quiz C.4:
How does the hardware handle the delayed transitions described above? ■

Answer:
Usually by adding additional states, though these additional states need not be actually stored with the cache line, due to the fact that only a few lines at a time will be transitioning. The need to delay transitions is but one issue that results in real-world cache coherence protocols being much more complex than the over-simplified MESI protocol described in this appendix. Hennessy and Patterson’s classic introduction to computer architecture [HP95] covers many of these issues.

Quick Quiz C.5:
What sequence of operations would put the CPUs’ caches all back into the “invalid” state? ■

Answer:
There is no such sequence, at least in absence of special “flush my cache” instructions in the CPU’s instruction set. Most CPUs do have such instructions.

Quick Quiz C.6:
In step 1 above, why does CPU 0 need to issue a “read invalidate” rather than a simple “invalidate”? ■

Answer:
Because the cache line in question contains more than just the variable a.

Quick Quiz C.7:
In step 1 of the first scenario in Section C.4.3, why is an “invalidate” sent instead of a ”read invalidate” message? Doesn’t CPU 0 need the values of the other variables that share this cache line with “a”? ■

Answer:
CPU 0 already has the values of these variables, given that it has a read-only copy of the cache line containing “a”. Therefore, all CPU 0 need do is to cause the other...
APPENDIX G. ANSWERS TO QUICK QUIZZES

CPUs to discard their copies of this cache line. An “invalidate” message therefore suffices.

Quick Quiz C.8:
Say what??? Why do we need a memory barrier here, given that the CPU cannot possibly execute the `assert()` until after the `while` loop completes?

Answer:
CPUs are free to speculatively execute, which can have the effect of executing the assertion before the `while` loop completes.

Quick Quiz C.9:
Does the guarantee that each CPU sees its own memory accesses in order also guarantee that each user-level thread will see its own memory accesses in order? Why or why not?

Answer:
No. Consider the case where a thread migrates from one CPU to another, and where the destination CPU perceives the source CPU’s recent memory operations out of order. To preserve user-mode sanity, kernel hackers must use memory barriers in the context-switch path. However, the locking already required to safely do a context switch should automatically provide the memory barriers needed to cause the user-level task to see its own accesses in order. That said, if you are designing a super-optimized scheduler, either in the kernel or at user level, please keep this scenario in mind!

Quick Quiz C.10:
Could this code be fixed by inserting a memory barrier between CPU 1’s “while” and assignment to “c”? Why or why not?

Answer:
No. Such a memory barrier would only force ordering local to CPU 1. It would have no effect on the relative ordering of CPU 0’s and CPU 1’s accesses, so the assertion could still fail. However, all mainstream computer systems provide one mechanism or another to provide “transitivity”, which provides intuitive causal ordering: if B saw the effects of A’s accesses, and C saw the effects of B’s accesses, then C must also see the effects of A’s accesses. In short, hardware designers have taken at least a little pity on software developers.

Quick Quiz C.11:
Suppose that lines 3-5 for CPUs 1 and 2 in Table C.4 are in an interrupt handler, and that the CPU 2’s line 9 is run at process level. What changes, if any, are required to enable the code to work correctly, in other words, to prevent the assertion from firing?

Answer:
The assertion will need to be written to ensure that the load of “e” precedes that of “a”. In the Linux kernel, the barrier() primitive may be used to accomplish this in much the same way that the memory barrier was used in the assertions in the previous examples.

Quick Quiz C.12:
If CPU 2 executed an `assert(e==0||c==1)` in the example in Table C.4, would this assert ever trigger?

Answer:
The result depends on whether the CPU supports “transitivity.” In other words, CPU 0 stored to “e” after seeing CPU 1’s store to “c”, with a memory barrier between CPU 0’s load from “c” and store to “e”. If some other CPU sees CPU 0’s store to “e”, is it also guaranteed to see CPU 1’s store?

All CPUs I am aware of claim to provide transitivity.

Quick Quiz C.13:
Why is Alpha’s `smp_read_barrier_depends()` an `smp_mb()` rather than `smp_rmb()`?

Answer:
First, Alpha has only `mb` and `wmb` instructions, so `smp_rmb()` would be implemented by the Alpha `mb` instruction in either case.

More importantly, `smp_read_barrier_depends()` must order subsequent stores. For example, consider the following code:

```c
1 p = global_pointer;
2 smp_read_barrier_depends();
3 if (do_something_with(p->a, p->b) == 0)
4   p->hey_look = 1;
```

Here the store to `p->hey_look` must be ordered, not
just the loads from $p->a$ and $p->b$.

**G.17 Read-Copy Update Implementations**

Quick Quiz D.1:
Why is sleeping prohibited within Classic RCU read-side critical sections?

Answer:
Because sleeping implies a context switch, which in Classic RCU is a quiescent state, and RCU’s grace-period detection requires that quiescent states never appear in RCU read-side critical sections.

Quick Quiz D.2:
Why not permit sleeping in Classic RCU read-side critical sections by eliminating context switch as a quiescent state, leaving user-mode execution and idle loop as the remaining quiescent states?

Answer:
This would mean that a system undergoing heavy kernel-mode execution load (e.g., due to kernel threads) might never complete a grace period, which would cause it to exhaust memory sooner or later.

Quick Quiz D.3:
Why is it OK to assume that updates separated by `synchronize_sched()` will be performed in order?

Answer:
Because this property is required for the `synchronize_sched()` aspect of RCU to work at all. For example, consider a code sequence that removes an object from a list, invokes `synchronize_sched()`, then frees the object. If this property did not hold, then that object might appear to be freed before it was removed from the list, which is precisely the situation that `synchronize_sched()` is supposed to prevent!

Quick Quiz D.4:
Why must line 17 in `synchronize_srcu()` (Figure D.10) precede the release of the mutex on line 18? What would have to change to permit these two lines to be interchanged? Would such a change be worthwhile? Why or why not?

Answer:
Suppose that the order was reversed, and that CPU 0 has just reached line 13 of `synchronize_srcu()`, while both CPU 1 and CPU 2 start executing another `synchronize_srcu()` each, and CPU 3 starts executing a `srcu_read_lock()`. Suppose that CPU 1 reaches line 6 of `synchronize_srcu()` just before CPU 0 increments the counter on line 13. Most importantly, suppose that CPU 3 executes `srcu_read_lock()` out of order with the following SRCU read-side critical section, so that it acquires a reference to some SRCU-protected data structure before CPU 0 increments `sp->completed`, but executes the `srcu_read_lock()` after CPU 0 does this increment.

Then CPU 0 will not wait for CPU 3 to complete its SRCU read-side critical section before exiting the “while” loop on lines 15-16 and releasing the mutex (remember, the CPU could be reordering the code).

Now suppose that CPU 2 acquires the mutex next, and again increments `sp->completed`. This CPU will then have to wait for CPU 3 to exit its SRCU read-side critical section before exiting the loop on lines 15-16 and releasing the mutex. But suppose that CPU 3 again executes out of order, completing the `srcu_read_unlock()` prior to executing a final reference to the pointer it obtained when entering the SRCU read-side critical section.

CPU 1 will then acquire the mutex, but see that the `sp->completed` counter has incremented twice, and therefore take the early exit. The caller might well free up the element that CPU 3 is still referencing (due to CPU 3’s out-of-order execution).

To prevent this perhaps improbable, but entirely possible, scenario, the final `synchronize_sched()` must precede the mutex release in `synchronize_srcu()`.

Another approach would be to change to comparison on line 7 of `synchronize_srcu()` to check for at least three increments of the counter. However, such a change would increase the latency of a “bulk update” scenario, where a hash table is being updated or unloaded using multiple threads. In the current code, the latency of the resulting concurrent `synchronize_srcu()` calls would take at most two SRCU grace periods, while with this change, three would be required.
More experience will be required to determine which approach is really better. For one thing, there must first be some use of SRCU with multiple concurrent updaters.

Quick Quiz D.5:
Wait a minute! With all those new locks, how do you avoid deadlock?

**Answer:**
Deadlock is avoided by never holding more than one of the *rcu_node* structures’ locks at a given time. This algorithm uses two more locks, one to prevent CPU hotplug operations from running concurrently with grace-period advancement (*onofflock*) and another to permit only one CPU at a time from forcing a quiescent state to end quickly (*fqslock*). These are subject to a locking hierarchy, so that *fqslock* must be acquired before *onofflock*, which in turn must be acquired before any of the *rcu_node* structures’ locks.

Also, as a practical matter, refusing to ever hold more than one of the *rcu_node* locks means that it is unnecessary to track which ones are held. Such tracking would be painful as well as unnecessary.

Quick Quiz D.6:
Why stop at a 64-times reduction? Why not go for a few orders of magnitude instead?

**Answer:**
RCU works with no problems on systems with a few hundred CPUs, so allowing 64 CPUs to contend on a single lock leaves plenty of headroom. Keep in mind that these locks are acquired quite rarely, as each CPU will check in about one time per grace period, and grace periods extend for milliseconds.

Quick Quiz D.7:
But I don’t care about McKenney’s lame excuses in the answer to Quick Quiz 2!! I want to get the number of CPUs contending on a single lock down to something reasonable, like sixteen or so!!!

**Answer:**
OK, have it your way, then! Set `CONFIG_RCU_FANOUT=16` and (for `NR_CPUS=4096`) you will get a three-level hierarchy with 256 *rcu_node* structures at the lowest level, 16 *rcu_node* structures as intermediate nodes, and a single root-level *rcu_node*. The penalty you will pay is that more *rcu_node* structures will need to be scanned when checking to see which CPUs need help completing their quiescent states (256 instead of only 64).

Quick Quiz D.8:
OK, so what is the story with the colors?

**Answer:**
Data structures analogous to *rcu_state* (including *rcu_ctrlblk*) are yellow, those containing the bitmaps used to determine when CPUs have checked in are pink, and the per-CPU *rcu_data* structures are blue. The data structures used to conserve energy (such as *rcu_dynticks*) will be colored green.

Quick Quiz D.9:
Given such an egregious bug, why does Linux run at all?

**Answer:**
Because the Linux kernel contains device drivers that are (relatively) well behaved. Few if any of them spin in RCU read-side critical sections for the many milliseconds that would be required to provoke this bug. The bug nevertheless does need to be fixed, and this variant of RCU does fix it.

Quick Quiz D.10:
But doesn’t this state diagram indicate that dyntick-idle CPUs will get hit with reschedule IPIs? Won’t that wake them up?

**Answer:**
No. Keep in mind that RCU is handling groups of CPUs. One particular group might contain both dyntick-idle CPUs and CPUs in normal mode that have somehow managed to avoid passing through a quiescent state. Only the latter group will be sent a reschedule IPI; the dyntick-idle CPUs will merely be marked as being in an extended quiescent state.

Quick Quiz D.11:
G.17. READ-COPY UPDATE IMPLEMENTATIONS

But what happens if a CPU tries to report going through a quiescent state (by clearing its bit) before the bit-setting CPU has finished? ■

Answer:
There are three cases to consider here:

1. A CPU corresponding to a non-yet-initialized leaf rcu_node structure tries to report a quiescent state. This CPU will see its bit already cleared, so will give up on reporting its quiescent state. Some later quiescent state will serve for the new grace period.

2. A CPU corresponding to a leaf rcu_node structure that is currently being initialized tries to report a quiescent state. This CPU will see that the rcu_node structure’s ->lock is held, so will spin until it is released. But once the lock is released, the rcu_node structure will have been initialized, reducing to the following case.

3. A CPU corresponding to a leaf rcu_node that has already been initialized tries to report a quiescent state. This CPU will find its bit set, and will therefore clear it. If it is the last CPU for that leaf node, it will move up to the next level of the hierarchy. However, this CPU cannot possibly be the last CPU in the system to report a quiescent state, given that the CPU doing the initialization cannot yet have checked in.

So, in all three cases, the potential race is resolved correctly.

Quick Quiz D.12:
And what happens if all CPUs try to report going through a quiescent state before the bit-setting CPU has finished, thus ending the new grace period before it starts? ■

Answer:
The bit-setting CPU cannot pass through a quiescent state during initialization, as it has irqs disabled. Its bits therefore remain non-zero, preventing the grace period from ending until the data structure has been fully initialized.

Quick Quiz D.13:
And what happens if one CPU comes out of dyntick-idle mode and then passed through a quiescent state just as another CPU notices that the first CPU was in dyntick-idle mode? Couldn’t they both attempt to report a quiescent state at the same time, resulting in confusion? ■

Answer:
They will both attempt to acquire the lock on the same leaf rcu_node structure. The first one to acquire the lock will report the quiescent state and clear the appropriate bit, and the second one to acquire the lock will see that this bit has already been cleared.

Quick Quiz D.14:
But what if all the CPUs end up in dyntick-idle mode? Wouldn’t that prevent the current RCU grace period from ever ending? ■

Answer:
Indeed it will! However, CPUs that have RCU callbacks are not permitted to enter dyntick-idle mode, so the only way that all the CPUs could possibly end up in dyntick-idle mode would be if there were absolutely no RCU callbacks in the system. And if there are no RCU callbacks in the system, then there is no need for the RCU grace period to end. In fact, there is no need for the RCU grace period to even start.

RCU will restart if some irq handler does a call_rcu(), which will cause an RCU callback to appear on the corresponding CPU, which will force that CPU out of dyntick-idle mode, which will in turn permit the current RCU grace period to come to an end.

Quick Quiz D.15:
Given that force_quiescent_state() is a three-phase state machine, don’t we have triple the scheduling latency due to scanning all the CPUs? ■

Answer:
Ah, but the three phases will not execute back-to-back on the same CPU, and, furthermore, the first (initialization) phase doesn’t do any scanning. Therefore, the scheduling-latency hit of the three-phase algorithm is no different than that of a single-phase algorithm. If the scheduling latency becomes a problem, one approach would be to recode the state machine to scan the CPUs incrementally, most likely by keeping state on a per-leaf-rcu_node basis. But first show me a problem in the real world, then I will consider fixing it!
APPENDIX G. ANSWERS TO QUICK QUIZZES

Quick Quiz D.16:
But the other reason to hold ->onofflock is to prevent multiple concurrent online/offline operations, right?

Answer:
Actually, no! The CPU-hotplug code’s synchronization design prevents multiple concurrent CPU online/offline operations, so only one CPU online/offline operation can be executing at any given time. Therefore, the only purpose of ->onofflock is to prevent a CPU online or offline operation from running concurrently with grace-period initialization.

Quick Quiz D.17:
Given all these acquisitions of the global ->onofflock, won’t there be horrible lock contention when running with thousands of CPUs?

Answer:
Actually, there can be only three acquisitions of this lock per grace period, and each grace period lasts many milliseconds. One of the acquisitions is by the CPU initializing for the current grace period, and the other two onlining and offline some CPU. These latter two cannot run concurrently due to the CPU-hotplug locking, so at most two CPUs can be contending for this lock at any given time.

Lock contention on ->onofflock should therefore be no problem, even on systems with thousands of CPUs.

Quick Quiz D.18:
Why not simplify the code by merging the detection of dyntick-idle CPUs with that of offline CPUs?

Answer:
It might well be that such merging may eventually be the right thing to do. In the meantime, however, there are some challenges:

1. CPUs are not allowed to go into dyntick-idle mode while they have RCU callbacks pending, but CPUs are allowed to go offline with callbacks pending. This means that CPUs going offline need to have their callbacks migrated to some other CPU, thus, we cannot allow CPUs to simply go quietly offline.

2. Present-day Linux systems run with NR_CPUS much larger than the actual number of CPUs. A unified approach could thus end up uselessly waiting on CPUs that are not just offline, but which never existed in the first place.

3. RCU is already operational when CPUs get online one at a time during boot, and therefore must handle the online process. This onlining must exclude grace-period initialization, so the ->onofflock must still be used.

4. CPUs often switch into and out of dyntick-idle mode extremely frequently, so it is not reasonable to use the heavyweight online/offline code path for entering and exiting dyntick-idle mode.

Quick Quiz D.19:
Why not simply disable bottom halves (softirq) when acquiring the rcu_data structure’s lock? Wouldn’t this be faster?

Answer:
Because this lock can be acquired from functions called by call_rcu(), which in turn can be invoked from irq handlers. Therefore, irqs must be disabled when holding this lock.

Quick Quiz D.20:
How about the qsmask and qsmaskinit fields for the leaf rcu_node structures? Doesn’t there have to be some way to work out which of the bits in these fields corresponds to each CPU covered by the rcu_node structure in question?

Answer:
Indeed there does! The grpmask field in each CPU’s rcu_data structure does this job.

Quick Quiz D.21:
But why bother setting qs_pending to one when a CPU is coming online, given that being offline is an extended quiescent state that should cover any ongoing grace period?

Answer:
Because this helps to resolve a race between a CPU
coming online just as a new grace period is starting.

Quick Quiz D.22:
Why record the last completed grace period number in passed_quiesc_completed? Doesn’t that cause this RCU implementation to be vulnerable to quiescent states seen while no grace period was in progress being incorrectly applied to the next grace period that starts?

Answer:
We record the last completed grace period number in order to avoid races where a quiescent state noted near the end of one grace period is incorrectly applied to the next grace period, especially for dyntick and CPU-offline grace periods. Therefore, force_quiescent_state() and friends all check the last completed grace period number to avoid such races.

Now these dyntick and CPU-offline grace periods are only checked for when a grace period is actually active. The only quiescent states that can be recorded when no grace period is in progress are self-detected quiescent states, which are recorded in the passed_quiesc_completed, passed_quiesc, and qs_pending. These variables are initialized every time the corresponding CPU notices that a new grace period has started, preventing any obsolete quiescent states from being applied to the new grace period.

All that said, optimizing grace-period latency may require that gpnum be tracked in addition to completed.

Quick Quiz D.23:
What is the point of running a system with NR_CPUS way bigger than the actual number of CPUs?

Answer:
Because this allows producing a single binary of the Linux kernel that runs on a wide variety of systems, greatly easing administration and validation.

Quick Quiz D.24:
Why not simply have multiple lists rather than this funny multi-tailed list?

Answer:
Because this multi-tailed approach, due to Lai Jiangshan, simplifies callback processing.

Quick Quiz D.25:
So some poor CPU has to note quiescent states on behalf of each and every offline CPU? Yecch! Won’t that result in excessive overheads in the not-uncommon case of a system with a small number of CPUs but a large value for NR_CPUS?

Answer:
Actually, no it will not!
Offline CPUs are excluded from both the qsmask and qsmaskinit bit masks, so RCU normally ignores them. However, there are races with online/offline operations that can result in an offline CPU having its qsmask bit set. These races must of course be handled correctly, and the way they are handled is to permit other CPUs to note that RCU is waiting on a quiescent state from an offline CPU.

Quick Quiz D.26:
So what guards the earlier fields in this structure?

Answer:
Nothing does, as they are constants set at compile time or boot time. Of course, the fields internal to each rcu_node in the ->node array may change, but they are guarded separately.

Quick Quiz D.27:
I thought that RCU read-side processing was supposed to be fast! The functions shown in Figure D.21 have so much junk in them that they just have to be slow! What gives here?

Answer:
Appearances can be deceiving. The preempt_disable(), preempt_enable(), local_bh_disable(), and local_bh_enable() each do a single non-atomic manipulation of local data. Even that assumes CONFIG_PREEMPT, otherwise, the preempt_disable() and preempt_enable() functions emit no code, not even compiler directives. The __acquire() and __release() functions emit no code (not even compiler directives), but are instead used by the sparse semantic-parsing bug-finding program.
Finally, \texttt{rcu\_read\_acquire()} and \texttt{rcu\_read\_release()} emit no code (not even compiler directives) unless the “lockdep” lock-order debugging facility is enabled, in which case they can indeed be somewhat expensive.

In short, unless you are a kernel hacker who has enabled debugging options, these functions are extremely cheap, and in some cases, absolutely free of overhead. And, in the words of a Portland-area furniture retailer, “free is a very good price”.

\textbf{Quick Quiz D.28:}
Why not simply use \texttt{__get\_cpu\_var()} to pick up a reference to the current CPU’s \texttt{rcu\_data} structure on line 13 in Figure D.22?

\textbf{Answer:}
Because we might be called either from \texttt{call\_rcu()} (in which case we would need \texttt{__get\_cpu\_var(rcu\_data)}) or from \texttt{call\_rcu\_bh()} (in which case we would need \texttt{__get\_cpu\_var(rcu\_bh\_data)}). Using the \texttt{->rda[]} array of whichever \texttt{rcu\_state} structure we were passed works correctly regardless of which API \texttt{__call\_rcu()} was invoked from (suggested by Lai Jiangshan [Jia08]).

\textbf{Quick Quiz D.29:}
Given that \texttt{rcu\_pending()} is always called twice on lines 29-32 of Figure D.23, shouldn’t there be some way to combine the checks of the two structures?

\textbf{Answer:}
Sorry, but this was a trick question. The C language’s short-circuit boolean expression evaluation means that \texttt{__rcu\_pending()} is invoked on \texttt{rcu\_bh\_state} only if the prior invocation on \texttt{rcu\_state} returns zero.

The reason the two calls are in this order is that “rcu” is used more heavily than is “rcu\_bh”, so the first call is more likely to return non-zero than is the second.

\textbf{Quick Quiz D.30:}
Shouldn’t line 42 of Figure D.23 also check for \texttt{in\_hardirq()}?

\textbf{Answer:}
No. The \texttt{rcu\_read\_lock\_bh()} primitive disables softirq, not hardirq. Because \texttt{call\_rcu\_bh()} need only wait for pre-existing “rcu\_bh” read-side critical sections to complete, we need only check \texttt{in\_softirq()}.

\textbf{Quick Quiz D.31:}
But don’t we also need to check that a grace period is actually in progress in \texttt{__rcu\_process\_callbacks} in Figure D.24?

\textbf{Answer:}
Indeed we do! And the first thing that \texttt{force\_quiescent\_state()} does is to perform exactly that check.

\textbf{Quick Quiz D.32:}
What happens if two CPUs attempt to start a new grace period concurrently in Figure D.24?

\textbf{Answer:}
One of the CPUs will be the first to acquire the root \texttt{rcu\_node} structure’s lock, and that CPU will start the grace period. The other CPU will then acquire the lock and invoke \texttt{rcu\_start\_gp()}, which, seeing that a grace period is already in progress, will immediately release the lock and return.

\textbf{Quick Quiz D.33:}
How does the code traverse a given path through the \texttt{rcu\_node} hierarchy from root to leaves?

\textbf{Answer:}
It turns out that the code never needs to do such a traversal, so there is nothing special in place to handle this.

\textbf{Quick Quiz D.34:}
C-preprocessor macros are so 1990s! Why not get with the times and convert \texttt{RCU\_DATA\_PTR\_INIT()} in Figure D.29 to be a function?

\textbf{Answer:}
Because, although it is possible to pass a reference to a particular CPU’s instance of a per-CPU variable to a function, there does not appear to be a good way pass a
G.17. READ-COPY UPDATE IMPLEMENTATIONS

reference to the full set of instances of a given per-CPU variable to a function. One could of course build an array of pointers, then pass a reference to the array in, but that is part of what the RCU_DATA_PTR_INIT() macro is doing in the first place.

Quick Quiz D.35:
What happens if a CPU comes online between the time that the last online CPU is notified on lines 25-26 of Figure D.29 and the time that register_cpu_notifier() is invoked on line 27?

Answer:
Only one CPU is online at this point, so the only way another CPU can come online is if this CPU puts it online, which it is not doing.

Quick Quiz D.36:
Why call cpu_quiet() on line 41 of Figure D.30, given that we are excluding grace periods with various locks, and given that any earlier grace periods would not have been waiting on this previously-offlined CPU?

Answer:
A new grace period might have started just after the ->onofflock was released on line 40. The cpu_quiet() will help expedite such a grace period.

Quick Quiz D.37:
But what if the rcu_node hierarchy has only a single structure, as it would on a small system? What prevents concurrent grace-period initialization in that case, given the code in Figure D.32?

Answer:
The later acquisition of the sole rcu_node structure’s ->lock on line 16 excludes grace-period initialization, which must acquire this same lock in order to initialize this sole rcu_node structure for the new grace period.

The ->onofflock is needed only for multi-node hierarchies, and is used in that case as an alternative to acquiring and holding all of the rcu_node structures’ ->lock fields, which would be incredibly painful on large systems.

Quick Quiz D.38:
But does line 25 of Figure D.32 ever really exit the loop? Why or why not?

Answer:
The only way that line 25 could exit the loop is if all CPUs were to be put offline. This cannot happen in the Linux kernel as of 2.6.28, though other environments have been designed to offline all CPUs during the normal shutdown procedure.

Quick Quiz D.39:
Suppose that line 26 got executed seriously out of order in Figure D.32, so that lastcomp is set to some prior grace period, but so that the current grace period is still waiting on the now-offline CPU? In this case, won’t the call to cpu_quiet() fail to report the quiescent state, thus causing the grace period to wait forever for this now-offline CPU?

Answer:
First, the lock acquisitions on lines 16 and 12 would prevent the execution of line 26 from being pushed that far out of order. Nevertheless, even if line 26 managed to be misordered that dramatically, what would happen is that force_quiescent_state() would eventually be invoked, and would notice that the current grace period was waiting for a quiescent state from an offline CPU. Then force_quiescent_state() would report the extended quiescent state on behalf of the offlined CPU.

Quick Quiz D.40:
Given that an offline CPU is in an extended quiescent state, why does line 28 of Figure D.32 need to care which grace period it is dealing with?

Answer:
It really does not need to care in this case. However, because it does need to care in many other cases, the cpu_quiet() function does take the grace-period number as an argument, so some value must be supplied.

Quick Quiz D.41:
But this list movement in Figure D.32 makes all of the going-offline CPU’s callbacks go through another grace
period, even if they were ready to invoke. Isn’t that inefficient? Furthermore, couldn’t an unfortunate pattern of CPUs going offline then coming back online prevent a given callback from ever being invoked?

**Answer:**
It is inefficient, but it is simple. Given that this is not a commonly executed code path, this is the right tradeoff. The starvation case would be a concern, except that the online and offline process involves multiple grace periods.

**Quick Quiz D.42:**
Why not just expand `note_new_gpnum()` inline into `check_for_new_grace_period()` in Figure D.34?

**Answer:**
Because `note_new_gpnum()` must be called for each new grace period, including those started by this CPU and those started by other CPUs. In contrast, `check_for_new_grace_period()` is called only for the case where some other CPU started the grace period.

**Quick Quiz D.43:**
But there has been no initialization yet at line 15 of Figure D.37! What happens if a CPU notices the new grace period and immediately attempts to report a quiescent state? Won’t it get confused?

**Answer:**
There are two cases of interest.

1. In the first case, there is only a single `rcu_node` structure in the hierarchy. Since the CPU executing in `rcu_start_gp()` is currently holding the `rcu_node` structure’s lock, the CPU attempting to report the quiescent state will not be able to acquire this lock until initialization is complete, at which point the quiescent state will be reported normally.

2. In the second case, there are multiple `rcu_node` structures, and the leaf `rcu_node` structure corresponding to the CPU that is attempting to report the quiescent state already has that CPU’s `->qsmask` bit cleared. Therefore, the CPU attempting to report the quiescent state will give up, and some later quiescent state for that CPU will be applied to the new grace period.

**Quick Quiz D.44:**
Hey! Shouldn’t we hold the non-leaf `rcu_node` structures’ locks when munging their state in line 37 of Figure D.37?

**Answer:**
There is no need to hold their locks. The reasoning is as follows:

1. The new grace period cannot end, because the running CPU (which is initializing it) won’t pass through a quiescent state. Therefore, there is no race with another invocation of `rcu_start_gp()`.

2. The running CPU holds `->onofflock`, so there is no race with CPU-hotplug operations.

3. The leaf `rcu_node` structures are not yet initialized, so they have all of their `->qsmask` bits cleared. This means that any other CPU attempting to report a quiescent state will stop at the leaf level, and thus cannot race with the current CPU for non-leaf `rcu_node` structures.

4. The RCU tracing functions access, but do not modify, the `rcu_node` structures’ fields. Races with these functions is therefore harmless.

**Quick Quiz D.45:**
Why can’t we merge the loop spanning lines 36-37 with the loop spanning lines 40-44 in Figure D.37?

**Answer:**
If we were to do so, we would either be needlessly acquiring locks for the non-leaf `rcu_node` structures or would need ugly checks for a given node being a leaf node on each pass through the loop. (Recall that we must acquire the locks for the leaf `rcu_node` structures due to races with CPUs attempting to report quiescent states.) Nevertheless, it is quite possible that experience on very large systems will show that such merging is in fact the right thing to do.
a quiescent state from a prior grace period against the current grace period? ■

**Answer:**
If this could occur, it would be a serious bug, since the CPU in question might be in an RCU read-side critical section that started before the beginning of the current grace period.

There are several cases to consider for the CPU in question:

1. It remained online and active throughout.
2. It was in dynticks-idle mode for at least part of the current grace period.
3. It was offline for at least part of the current grace period.

In the first case, the prior grace period could not have ended without this CPU explicitly reporting a quiescent state, which would leave \( \rightarrow qs\_pending \) zero. This in turn would mean that lines 7-8 would return, so that control would not reach `cpu_quiet()` unless `check_for_new_grace_period()` had noted the new grace period. However, if the current grace period had been noted, it would also have set \( \rightarrow passed\_quiesc \) to zero, in which case lines 9-10 would have returned, again meaning that `cpu_quiet()` would not be invoked. Finally, the only way that \( \rightarrow passed\_quiesc \) could be invoked would be if `rcu_check_quiescent_state()` was invoked by a scheduling-clock interrupt that occurred somewhere between lines 5 and 9 of `rcu_check_quiescent_state()` in Figure D.39. However, this would be a case of a quiescent state occurring in the *current* grace period, which would be totally legitimate to report against the current grace period. So this case is correctly covered.

In the second case, where the CPU in question spent part of the new quiescent state in dynticks-idle mode, note that dynticks-idle mode is an extended quiescent state, hence it is again permissible to report this quiescent state against the current grace period.

In the third case, where the CPU in question spent part of the new quiescent state offline, note that offline CPUs are in an extended quiescent state, which is again permissible to report against the current grace period.

So quiescent states from prior grace periods are never reported against the current grace period.

**Quick Quiz D.47:**
How do lines 22-23 of Figure D.40 know that it is safe to promote the running CPU’s RCU callbacks? ■

**Answer:**
Because the specified CPU has not yet passed through a quiescent state, and because we hold the corresponding leaf node’s lock, we know that the current grace period cannot possibly have ended yet. Therefore, there is no danger that any of the callbacks currently queued were registered after the next grace period started, given that they have already been queued and the next grace period has not yet started.

**Quick Quiz D.48:**
Given that argument `mask` on line 2 of Figure D.41 is an unsigned long, how can it possibly deal with systems with more than 64 CPUs? ■

**Answer:**
Because `mask` is specific to the specified leaf `rcu_node` structure, it need only be large enough to represent the CPUs corresponding to that particular `rcu_node` structure. Since at most 64 CPUs may be associated with a given `rcu_node` structure (32 CPUs on 32-bit systems), the unsigned long `mask` argument suffices.

**Quick Quiz D.49:**
How do RCU callbacks on dynticks-idle or offline CPUs get invoked? ■

**Answer:**
They don’t. CPUs with RCU callbacks are not permitted to enter dynticks-idle mode, so dynticks-idle CPUs never have RCU callbacks. When CPUs go offline, their RCU callbacks are migrated to an online CPU, so offline CPUs never have RCU callbacks, either. Thus, there is no need to invoke callbacks on dynticks-idle or offline CPUs.

**Quick Quiz D.50:**
Why would lines 14-17 in Figure D.43 need to adjust the tail pointers? ■

**Answer:**
If any of the tail pointers reference the last callback in the sublist that was ready to invoke, they must be changed to
instead reference the \texttt{\textbf{-texlist}} pointer. This situation occurs when the sublists immediately following the ready-to-invocable sublist are empty.

**Quick Quiz D.51:**
But how does the code in Figure D.45 handle nested NMIs?

**Answer:**
It does not have to handle nested NMIs, because NMIs do not nest.

**Quick Quiz D.52:**
Why isn’t there a memory barrier between lines 8 and 9 of Figure D.47? Couldn’t this cause the code to fetch even-numbered values from both the \texttt{\textbf{-dynticks}} and \texttt{\textbf{-dynticks\_nmi}} fields, even though these two fields never were zero at the same time?

**Answer:**
First, review the code in Figures D.44, D.45, and D.46, and note that \texttt{\textbf{dynticks}} and \texttt{\textbf{dynticks\_nmi}} will never have odd values simultaneously (see especially lines 6 and 17 of Figure D.45, and recall that interrupts cannot happen from NMIs).

Of course, given the placement of the memory barriers in these functions, it might \textit{appear} to another CPU that both counters were odd at the same time, but logically this cannot happen, and would indicate that the CPU had in fact passed through \texttt{\textbf{dynticks\_idle}} mode.

Now, let’s suppose that at the time line 8 fetches \texttt{-dynticks}, the value of \texttt{-dynticks\_nmi} was at odd number, and that at the time line 9 fetches \texttt{-dynticks\_nmi}, the value of \texttt{-dynticks} was an odd number. Given that both counters cannot be odd simultaneously, there must have been a time between these two fetches when both counters were even, and thus a time when the CPU was in \texttt{\textbf{dynticks\_idle}} mode, which is a quiescent state, as required.

So, why can’t the \texttt{\textbf{&&}} on line 13 of Figure D.47 be replaced with an \texttt{\textbf{==}}? Well, it could be, but this would likely be more confusing than helpful.

**Quick Quiz D.53:**
Why wait the extra couple jiffies on lines 12-13 in Figure D.55?

**Answer:**
This added delay gives the offending CPU a better chance of reporting on itself, thus getting a decent stack trace of the stalled code. Of course, if the offending CPU is spinning with interrupts disabled, it will never report on itself, so other CPUs do so after a short delay.

**Quick Quiz D.54:**
What prevents the grace period from ending before the stall warning is printed in Figure D.56?

**Answer:**
The caller checked that this CPU still had not reported a quiescent state, and because preemption is disabled, there is no way that a quiescent state could have been reported in the meantime.

**Quick Quiz D.55:**
Why does \texttt{\textbf{print\_other\_cpu\_stall()}} in Figure D.57 need to check for the grace period ending when \texttt{\textbf{print\_cpu\_stall()}} did not?

**Answer:**
The other CPUs might pass through a quiescent state at any time, so the grace period might well have ended in the meantime.

**Quick Quiz D.56:**
Why is it important that blocking primitives called from within a preemptible-RCU read-side critical section be subject to priority inheritance?

**Answer:**
Because blocked readers stall RCU grace periods, which can result in OOM. For example, if a reader did a \texttt{\textbf{wait\_event()}} within an RCU read-side critical section, and that event never occurred, then RCU grace periods would stall indefinitely, guaranteeing that the system would OOM sooner or later. There must therefore be some way to cause these readers to progress through their read-side critical sections in order to avoid such OOMs. Priority boosting is one way to force such progress, but only if readers are restricted to blocking such that they can be awakened via priority boosting.

Of course, there are other methods besides priority
inheritance that handle the priority inversion problem, including priority ceiling, preemption disabling, and so on. However, there are good reasons why priority inheritance is the approach used in the Linux kernel, so this is what is used for RCU.

Quick Quiz D.57:
Could the prohibition against using primitives that would block in a non-CONFIG_PREEMPT kernel be lifted, and if so, under what conditions?

Answer:
If testing and benchmarking demonstrated that the preemptible RCU worked well enough that classic RCU could be dispensed with entirely, and if priority inheritance was implemented for blocking synchronization primitives such as semaphores, then those primitives could be used in RCU read-side critical sections.

Quick Quiz D.58:
How is it possible for lines 38-43 of __rcu_advance_callbacks() to be executed when lines 7-37 have not? Won’t they both be executed just after a counter flip, and never at any other time?

Answer:
Consider the following sequence of events:

1. CPU 0 executes lines 5-12 of rcu_try_flip_idle().
2. CPU 1 executes __rcu_advance_callbacks(). Because rcu_c trìblk.completed has been incremented, lines 7-37 execute. However, none of the rcu_flip_flag variables have been set, so lines 38-43 do not execute.
3. CPU 0 executes lines 13-15 of rcu_try_flip_idle().
4. Later, CPU 1 again executes __rcu_advance_callbacks(). The counter has not been incremented since the earlier execution, but the rcu_flip_flag variables have all been set, so only lines 38-43 are executed.

Quick Quiz D.59:
What problems could arise if the lines containing ACCESS_ONCE() in rcu_read_unlock() were reordered by the compiler?

Answer:
1. If the ACCESS_ONCE() were omitted from the fetch of rcu_flipctr_idx (line 14), then the compiler would be within its rights to eliminate idx. It would also be free to compile the rcu_flipctr decrement as a fetch-increment-store sequence, separately fetching rcu_flipctr_idx for both the fetch and the store. If an NMI were to occur between the fetch and the store, and if the NMI handler contained an rcu_read_lock(), then the value of rcu_flipctr_idx would change in the meantime, resulting in corruption of the rcu_flipctr values, destroying the ability to correctly identify grace periods.

2. Another failure that could result from omitting the ACCESS_ONCE() from line 14 is due to the compiler reordering this statement to follow the decrement of rcu_read_lock_nesting (line 16). In this case, if an NMI were to occur between these two statements, then any rcu_read_lock() in the NMI handler could corrupt rcu_flipctr_idx, causing the wrong rcu_flipctr to be decremented. As with the analogous situation in rcu_read_lock(), this could result in premature grace-period termination, an indefinite grace period, or even both.

3. If ACCESS_ONCE() macros were omitted such that the update of rcu_read_lock_nesting could be interchanged by the compiler with the decrement of rcu_flipctr, and if an NMI occurred in between, any rcu_read_lock() in the NMI handler would incorrectly conclude that it was protected by an enclosing rcu_read_lock(), and fail to increment the rcu_flipctr variables.

It is not clear that the ACCESS_ONCE() on the fetch of rcu_read_lock_nesting (line 7) is required.

Quick Quiz D.60:
What problems could arise if the lines containing ACCESS_ONCE() in rcu_read_unlock() were
reordered by the CPU?

Answer:
Absolutely none! The code in rcu_read_unlock() interacts with the scheduling-clock interrupt handler running on the same CPU, and is thus insensitive to reorderings because CPUs always see their own accesses as if they occurred in program order. Other CPUs do access the rcu_flipctr, but because these other CPUs don’t access any of the other variables, ordering is irrelevant.

Quick Quiz D.61:
What problems could arise in rcu_read_unlock() if irqs were not disabled?

Answer:

1. Disabling irqs has the side effect of disabling preemption. Suppose that this code were to be preempted in the midst of line 17 between selecting the current CPU’s copy of the rcu_flipctr array and the decrement of the element indicated by rcu_flipctr_idx. Execution might well resume on some other CPU. If this resumption happened concurrently with an rcu_read_lock() or rcu_read_unlock() running on the original CPU, an increment or decrement might be lost, resulting in either premature termination of a grace period, indefinite extension of a grace period, or even both.

2. Failing to disable preemption can also defeat RCU priority boosting, which relies on rcu_read_lock_nesting to determine which tasks to boost. If preemption occurred between the update of rcu_read_lock_nesting (line 16) and of rcu_flipctr (line 17), then a grace period might be stalled until this task resumed. But because the RCU priority booster has no way of knowing that this particular task is stalling grace periods, needed boosting will never occur. Therefore, if there are CPU-bound realtime tasks running, the preempted task might never resume, stalling grace periods indefinitely, and eventually resulting in OOM.

Of course, both of these situations could be handled by disabling preemption rather than disabling irqs. (The CPUs I have access to do not show much difference between these two alternatives, but others might.)

Quick Quiz D.62:
Suppose that the irq disabling in rcu_read_lock() was replaced by preemption disabling. What effect would that have on GP_STAGES?

Answer:
No finite value of GP_STAGES suffices. The following scenario, courtesy of Oleg Nesterov, demonstrates this:

Suppose that low-priority Task A has executed rcu_read_lock() on CPU 0, and thus has incremented per_cpu(rcu_flipctr, 0)[0], which thus has a value of one. Suppose further that Task A is now preempted indefinitely.

Given this situation, consider the following sequence of events:

1. Task B starts executing rcu_read_lock(), also on CPU 0, picking up the low-order bit of rcu_ctrlblk.completed, which is still equal to zero.

2. Task B is interrupted by a sufficient number of scheduling-clock interrupts to allow the current grace-period stage to complete, and also be sufficient long-running interrupts to allow the RCU grace-period state machine to advance the rcu_ctrlblk.complete counter so that its bottom bit is now equal to one and all CPUs have acknowledged this increment operation.

3. CPU 1 starts summing the index==0 counters, starting with per_cpu(rcu_flipctr, 0)[0], which is equal to one due to Task A’s increment. CPU 1’s local variable sum is therefore equal to one.

4. Task B returns from interrupt, resuming its execution of rcu_read_lock(), incrementing per_cpu(rcu_flipctr, 0)[0], which now has a value of two.

5. Task B is migrated to CPU 2.

6. Task B completes its RCU read-side critical section, and executes rcu_read_unlock(), which decrements per_cpu(rcu_flipctr, 2)[0], which is now -1.
The locker process is an infinite loop, so control never reaches the end of this process. However, since there are no monotonically increasing variables, Promela is able to model this infinite loop with a small number of states.

Quick Quiz F.2:
What are some Promela code-style issues with this example?
APPENDIX G. ANSWERS TO QUICK QUIZZES

Answer:
There are several:

1. The declaration of \texttt{sum} should be moved to within the init block, since it is not used anywhere else.

2. The assertion code should be moved outside of the initialization loop. The initialization loop can then be placed in an atomic block, greatly reducing the state space (by how much?).

3. The atomic block covering the assertion code should be extended to include the initialization of \texttt{sum} and \texttt{j}, and also to cover the assertion. This also reduces the state space (again, by how much?).

Quick Quiz F.3:
Is there a more straightforward way to code the do-od statement?

Answer:
Yes. Replace it with \texttt{if-fi} and remove the two \texttt{break} statements.

Quick Quiz F.4:
Why are there atomic blocks at lines 12-21 and lines 44-56, when the operations within those atomic blocks have no atomic implementation on any current production microprocessor?

Answer:
Because those operations are for the benefit of the assertion only. They are not part of the algorithm itself. There is therefore no harm in marking them atomic, and so marking them greatly reduces the state space that must be searched by the Promela model.

Quick Quiz F.5:
Is the re-summing of the counters on lines 24-27 really necessary?

Answer:
Yes. To see this, delete these lines and run the model. Alternatively, consider the following sequence of steps:

1. One process is within its RCU read-side critical section, so that the value of \texttt{ctr[0]} is zero and the value of \texttt{ctr[1]} is two.

2. An updater starts executing, and sees that the sum of the counters is two so that the fastpath cannot be executed. It therefore acquires the lock.

3. A second updater starts executing, and fetches the value of \texttt{ctr[0]}, which is zero.

4. The first updater adds one to \texttt{ctr[0]}, flips the index (which now becomes zero), then subtracts one from \texttt{ctr[1]} (which now becomes one).

5. The second updater fetches the value of \texttt{ctr[1]}, which is now one.

6. The second updater now incorrectly concludes that it is safe to proceed on the fastpath, despite the fact that the original reader has not yet completed.

Quick Quiz F.6:
Yeah, that’s just great! Now, just what am I supposed to do if I don’t happen to have a machine with 40GB of main memory???

Answer:
Relax, there are a number of lawful answers to this question:

1. Further optimize the model, reducing its memory consumption.

2. Work out a pencil-and-paper proof, perhaps starting with the comments in the code in the Linux kernel.

3. Devise careful torture tests, which, though they cannot prove the code correct, can find hidden bugs.

4. There is some movement towards tools that do model checking on clusters of smaller machines. However, please note that we have not actually used such tools myself, courtesy of some large machines that Paul has occasional access to.

Quick Quiz F.7:
Why not simply increment \texttt{rcu_update_flag}, and then only increment \texttt{dynticks_progress_counter} if the old value of \texttt{rcu_update_flag} was zero???

Answer:
This fails in presence of NMIs. To see this, suppose an NMI was received just after `rcu_irq_enter()` incremented `rcu_update_flag`, but before it incremented `dynticks_progress_counter`. The instance of `rcu_irq_enter()` invoked by the NMI would see that the original value of `rcu_update_flag` was non-zero, and would therefore refrain from incrementing `dynticks_progress_counter`. This would leave the RCU grace-period machinery no clue that the NMI handler was executing on this CPU, so that any RCU read-side critical sections in the NMI handler would lose their RCU protection.

The possibility of NMI handlers, which, by definition cannot be masked, does complicate this code.

**Quick Quiz F.8:**
But if line 7 finds that we are the outermost interrupt, wouldn’t we always need to increment `dynticks_progress_counter`?

**Answer:**
Not if we interrupted a running task! In that case, `dynticks_progress_counter` would have already been incremented by `rcu_exit_nohz()`, and there would be no need to increment it again.

**Quick Quiz F.9:**
Can you spot any bugs in any of the code in this section?

**Answer:**
Read the next section to see if you were correct.

**Quick Quiz F.10:**
Why isn’t the memory barrier in `rcu_exit_nohz()` and `rcu_enter_nohz()` modeled in Promela?

**Answer:**
Promela assumes sequential consistency, so it is not necessary to model memory barriers. In fact, one must instead explicitly model lack of memory barriers, for example, as shown in Figure F.13 on page 289.

**Quick Quiz F.11:**
Isn’t it a bit strange to model `rcu_exit_nohz()` followed by `rcu_enter_nohz()`? Wouldn’t it be more natural to instead model entry before exit?

**Answer:**
It probably would be more natural, but we will need this particular order for the liveness checks that we will add later.

**Quick Quiz F.12:**
Wait a minute! In the Linux kernel, both `dynticks_progress_counter` and `rcu_dyntick_snapshot` are per-CPU variables. So why are they instead being modeled as single global variables?

**Answer:**
Because the grace-period code processes each CPU’s `dynticks_progress_counter` and `rcu_dyntick_snapshot` variables separately, we can collapse the state onto a single CPU. If the grace-period code were instead to do something special given specific values on specific CPUs, then we would indeed need to model multiple CPUs. But fortunately, we can safely confine ourselves to two CPUs, the one running the grace-period processing and the one entering and leaving dynticks-idle mode.

**Quick Quiz F.13:**
Given there are a pair of back-to-back changes to `grace_period_state` on lines 25 and 26, how can we be sure that line 25’s changes won’t be lost?

**Answer:**
Recall that Promela and spin trace out every possible sequence of state changes. Therefore, timing is irrelevant: Promela/spin will be quite happy to jam the entire rest of the model between those two statements unless some state variable specifically prohibits doing so.

**Quick Quiz F.14:**
But what would you do if you needed the statements in a single `EXECUTE_MAINLINE()` group to execute non-atomically?

**Answer:**
The easiest thing to do would be to put each such state-
Quick Quiz F.15:
But what if the dynticks_nohz() process had “if” or “do” statements with conditions, where the statement bodies of these constructs needed to execute non-atomically?

Answer:
One approach, as we will see in a later section, is to use explicit labels and “goto” statements. For example, the construct:

```
if
:: i == 0 -> a = -1;
:: else -> a = -2;
fi;
```

could be modeled as something like:

```
EXECUTE_MAINLINE(stmt1,
if
:: i == 0 -> goto stmt1_then;
:: else -> goto stmt1_else;
fi)
stmt1_then: skip;
EXECUTE_MAINLINE(stmt1_then1, a = -1; goto stmt1_end)
stmt1_else: skip;
EXECUTE_MAINLINE(stmt1_then1, a = -2)
stmt1_end: skip;
```

However, it is not clear that the macro is helping much in the case of the “if” statement, so these sorts of situations will be open-coded in the following sections.

Quick Quiz F.16:
Why are lines 45 and 46 (the in_dyntick_irq = 0; and the i++;) executed atomically?

Answer:
These lines of code pertain to controlling the model, not to the code being modeled, so there is no reason to model them non-atomically. The motivation for modeling them atomically is to reduce the size of the state space.

Quick Quiz F.17:
What property of interrupts is this dynticks_irq() process unable to model?

Answer:
One such property is nested interrupts, which are handled in the following section.

Quick Quiz F.18:
Does Paul always write his code in this painfully incremental manner?

Answer:
Not always, but more and more frequently. In this case, Paul started with the smallest slice of code that included an interrupt handler, because he was not sure how best to model interrupts in Promela. Once he got that working, he added other features. (But if he was doing it again, he would start with a “toy” handler. For example, he might have the handler increment a variable twice and have the mainline code verify that the value was always even.)

Why the incremental approach? Consider the following, attributed to Brian W. Kernighan:

```
Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.
```

This means that any attempt to optimize the production of code should place at least 66% of its emphasis on optimizing the debugging process, even at the expense of increasing the time and effort spent coding. Incremental coding and testing is one way to optimize the debugging process, at the expense of some increase in coding effort. Paul uses this approach because he rarely has the luxury of devoting full days (let alone weeks) to coding and debugging.

Quick Quiz F.19:
But what happens if an NMI handler starts running before an irq handler completes, and if that NMI handler continues running until a second irq handler starts?

Answer:
This cannot happen within the confines of a single CPU. The first irq handler cannot complete until the NMI handler returns. Therefore, if each of the dynticks and dynticks_nmi variables have taken on an even value during a given time interval, the corresponding CPU really was in a quiescent state at some time during that interval.
Quick Quiz F.20:
This is still pretty complicated. Why not just have a \texttt{cpumask_t} that has a bit set for each CPU that is in dyntick-idle mode, clearing the bit when entering an irq or NMI handler, and setting it upon exit?

Answer:
Although this approach would be functionally correct, it would result in excessive irq entry/exit overhead on large machines. In contrast, the approach laid out in this section allows each CPU to touch only per-CPU data on irq and NMI entry/exit, resulting in much lower irq entry/exit overhead, especially on large machines.
Appendix H

Glossary

Associativity: The number of cache lines that can be held simultaneously in a given cache, when all of these cache lines hash identically in that cache. A cache that could hold four cache lines for each possible hash value would be termed a “four-way set-associative” cache, while a cache that could hold only one cache line for each possible hash value would be termed a “direct-mapped” cache. A cache whose associativity was equal to its capacity would be termed a “fully associative” cache. Fully associative caches have the advantage of eliminating associativity misses, but, due to hardware limitations, fully associative caches are normally quite limited in size. The associativity of the large caches found on modern microprocessors typically range from two-way to eight-way.

Associativity Miss: A cache miss incurred because the corresponding CPU has recently accessed more data hashing to a given set of the cache than will fit in that set. Fully associative caches are not subject to associativity misses (or, equivalently, in fully associative caches, associativity and capacity misses are identical).

Atomic: An operation is considered “atomic” if it is not possible to observe any intermediate state. For example, on most CPUs, a store to a properly aligned pointer is atomic, because other CPUs will see either the old value or the new value, but are guaranteed not to see some mixed value containing some pieces of the new and old values.

Cache: In modern computer systems, CPUs have caches in which to hold frequently used data. These caches can be thought of as hardware hash tables with very simple hash functions, but in which each hash bucket (termed a “set” by hardware types) can hold only a limited number of data items. The number of data items that can be held by each of a cache’s hash buckets is termed the cache’s “associativity”. These data items are normally called “cache lines”, which can be thought of a fixed-length blocks of data that circulate among the CPUs and memory.

Cache Coherence: A property of most modern SMP machines where all CPUs will observe a sequence of values for a given variable that is consistent with at least one global order of values for that variable. Cache coherence also guarantees that at the end of a group of stores to a given variable, all CPUs will agree on the final value for that variable. Note that cache coherence applies only to the series of values taken on by a single variable. In contrast, the memory consistency model for a given machine describes the order in which loads and stores to groups of variables will appear to occur.

Cache Coherence Protocol: A communications protocol, normally implemented in hardware, that enforces memory consistency and ordering, preventing different CPUs from seeing inconsistent views of data held in their caches.

Cache Geometry: The size and associativity of a cache is termed its geometry. Each cache may be thought of as a two-dimensional array, with rows of cache lines (“sets”) that have the same hash value, and columns of cache lines (“ways”) in which every cache line has a different hash value. The associativity of a given cache is its number of columns (hence the name “way” – a two-way set-associative cache has two “ways”), and the size of the cache is its number of rows multiplied by its number of columns.
**Cache Line:** (1) The unit of data that circulates among the CPUs and memory, usually a moderate power of two in size. Typical cache-line sizes range from 16 to 256 bytes.  
(2) A physical location in a CPU cache capable of holding one cache-line unit of data.  
(3) A physical location in memory capable of holding one cache-line unit of data, but that it also aligned on a cache-line boundary. For example, the address of the first word of a cache line in memory will end in 0x00 on systems with 256-byte cache lines.

**Cache Miss:** A cache miss occurs when data needed by the CPU is not in that CPU’s cache. The data might be missing because of a number of reasons, including: (1) this CPU has never accessed the data before (“startup” or “warmup” miss), (2) this CPU has recently accessed more data than would fit in its cache, so that some of the older data had to be removed (“capacity” miss), (3) this CPU has recently accessed more data in a given set\(^1\) than that set could hold (“associativity” miss), (4) some other CPU has written to the data (or some other data in the same cache line) since this CPU has accessed it (“communication miss”), or (5) this CPU attempted to write to a cache line that is currently read-only, possibly due to that line being replicated in other CPUs’ caches.

**Capacity Miss:** A cache miss incurred because the corresponding CPU has recently accessed more data than will fit into the cache.

**Code Locking:** A simple locking design in which a “global lock” is used to protect a set of critical sections, so that access by a given thread to that set is granted or denied based only on the set of critical sections currently occupying the set of critical sections, not based on what data the thread intends to access. The scalability of a code-locked program is limited by the code; increasing the size of the data set will normally not increase scalability (in fact, will typically decrease scalability by increasing “lock contention”). Contrast with “data locking”.

**Communication Miss:** A cache miss incurred because the some other CPU has written to the cache line since the last time this CPU accessed it.

**Critical Section:** A section of code guarded by some synchronization mechanism, so that its execution constrained by that primitive. For example, if a set of critical sections are guarded by the same global lock, then only one of those critical sections may be executing at a given time. If a thread is executing in one such critical section, any other threads must wait until the first thread completes before executing any of the critical sections in the set.

**Data Locking:** A scalable locking design in which each instance of a given data structure has its own lock. If each thread is using a different instance of the data structure, then all of the threads may be executing in the set of critical sections simultaneously. Data locking has the advantage of automatically scaling to increasing numbers of CPUs as the number of instances of data grows. Contrast with “code locking”.

**Direct-Mapped Cache:** A cache with only one way, so that it may hold only one cache line with a given hash value.

**Exclusive Lock:** An exclusive lock is a mutual-exclusion mechanism that permits only one thread at a time into the set of critical sections guarded by that lock.

**False Sharing:** If two CPUs each frequently write to one of a pair of data items, but the pair of data items are located in the same cache line, this cache line will be repeatedly invalidated, “ping-ponging” back and forth between the two CPUs’ caches. This is a common cause of “cache thrashing”, also called “cacheline bouncing” (the latter most commonly in the Linux community). False sharing can dramatically reduce both performance and scalability.

**Fragmentation:** A memory pool that has a large amount of unused memory, but not laid out to permit satisfying a relatively small request is said to be fragmented. External fragmentation occurs when the space is divided up into small fragments lying between allocated blocks of memory, while internal fragmentation occurs when specific requests or types of requests have been allotted more memory than they actually requested.

**Fully Associative Cache:** A fully associative cache contains only one set, so that it can hold any subset of memory that fits within its capacity.

**Grace Period:** A grace period is any contiguous time interval such that any RCU read-side critical section

---
\(^1\) In hardware-cache terminology, the word “set” is used in the same way that the word “bucket” is used when discussing software caches.
that began before the start of that interval has completed before the end of that same interval. Many RCU implementations define a grace period to be a time interval during which each thread has passed through at least one quiescent state. Since RCU read-side critical sections by definition cannot contain quiescent states, these two definitions are almost always interchangeable.

**Hot Spot:** Data structure that is very heavily used, resulting in high levels of contention on the corresponding lock. One example of this situation would be a hash table with a poorly chosen hash function.

**Invalidation:** When a CPU wishes to write to a data item, it must first ensure that this data item is not present in any other CPUs’ cache. If necessary, the item is removed from the other CPUs’ caches via “invalidation” messages from the writing CPUs to any CPUs having a copy in their caches.

**IPI:** Inter-processor interrupt, which is an interrupt sent from one CPU to another. IPIs are used heavily in the Linux kernel, for example, within the scheduler to alert CPUs that a high-priority process is now runnable.

**IRQ:** Interrupt request, often used as an abbreviation for “interrupt” within the Linux kernel community, as in “irq handler”.

**Linearizable:** A sequence of operations is “linearizable” if there is at least one global ordering of the sequence that is consistent with the observations of all CPUs/threads.

**Lock:** A software abstraction that can be used to guard critical sections, as such, an example of a “mutual exclusion mechanism”. An “exclusive lock” permits only one thread at a time into the set of critical sections guarded by that lock, while a “reader-writer lock” permits any number of reading threads, or but one writing thread, into the set of critical sections guarded by that lock. (Just to be clear, the presence of a writer thread in any of a given reader-writer lock’s critical sections will prevent any reader from entering any of that lock’s critical sections and vice versa.)

**Lock Contention:** A lock is said to be suffering contention when it is being used so heavily that there is often a CPU waiting on it. Reducing lock contention is often a concern when designing parallel algorithms and when implementing parallel programs.

**Memory Consistency:** A set of properties that impose constraints on the order in which accesses to groups of variables appear to occur. Memory consistency models range from sequential consistency, a very constraining model popular in academic circles, through process consistency, release consistency, and weak consistency.

**MESI Protocol:** The cache-coherence protocol featuring modified, exclusive, shared, and invalid (MESI) states, so that this protocol is named after the states that the cache lines in a given cache can take on. A modified line has been recently written to by this CPU, and is the sole representative of the current value of the corresponding memory location. An exclusive cache line has not been written to, but this CPU has the right to write to it at any time, as the line is guaranteed not to be replicated into any other CPU’s cache (though the corresponding location in main memory is up to date). A shared cache line is (or might be) replicated in some other CPUs’ cache, meaning that this CPU must interact with those other CPUs before writing to this cache line. An invalid cache line contains no value, instead representing “empty space” in the cache into which data from memory might be loaded.

**Mutual-Exclusion Mechanism:** A software abstraction that regulates threads’ access to “critical sections” and corresponding data.

**NMI:** Non-maskable interrupt. As the name indicates, this is an extremely high-priority interrupt that cannot be masked. These are used for hardware-specific purposes such as profiling. The advantage of using NMIs for profiling is that it allows you to profile code that runs with interrupts disabled.

**NUCA:** Non-uniform cache architecture, where groups of CPUs share caches. CPUs in a group can therefore exchange cache lines with each other much more quickly than they can with CPUs in other groups. Systems comprised of CPUs with hardware threads will generally have a NUCA architecture.

**NUMA:** Non-uniform memory architecture, where memory is split into banks and each such bank is “close” to a group of CPUs, the group being termed a
“NUMA node”. An example NUMA machine is Sequent’s NUMA-Q system, where each group of four CPUs had a bank of memory near by. The CPUs in a given group can access their memory much more quickly than another group’s memory.

**NUMA Node**: A group of closely placed CPUs and associated memory within a larger NUMA machines. Note that a NUMA node might well have a NUCA architecture.

**Pipelined CPU**: A CPU with a pipeline, which is an internal flow of instructions internal to the CPU that is in some way similar to an assembly line, with many of the same advantages and disadvantages. In the 1960s through the early 1980s, pipelined CPUs were the province of supercomputers, but started appearing in microprocessors (such as the 80486) in the late 1980s.

**Process Consistency**: A memory-consistency model in which each CPU’s stores appear to occur in program order, but in which different CPUs might see accesses from more than one CPU as occurring in different orders.

**Program Order**: The order in which a given thread’s instructions would be executed by a now-mythical “in-order” CPU that completely executed each instruction before proceeding to the next instruction. (The reason such CPUs are now the stuff of ancient myths and legends is that they were extremely slow. These dinosaurs were one of the many victims of Moore’s-Law-driven increases in CPU clock frequency. Some claim that these beasts will roam the earth once again, others vehemently disagree.)

**Quiescent State**: In RCU, a point in the code where there can be no references held to RCU-protected data structures, which is normally any point outside of an RCU read-side critical section. Any interval of time during which all threads pass through at least one quiescent state each is termed a “grace period”.

**Read-Copy Update (RCU)**: A synchronization mechanism that can be thought of as a replacement for reader-writer locking or reference counting. RCU provides extremely low-overhead access for readers, while writers incur additional overhead maintaining old versions for the benefit of pre-existing readers. Readers neither block nor spin, and thus cannot participate in deadlocks, however, they also can see stale data and can run concurrently with updates. RCU is thus best-suited for read-mostly situations where stale data can either be tolerated (as in routing tables) or avoided (as in the Linux kernel’s System V IPC implementation).

**Read-Side Critical Section**: A section of code guarded by read-acquisition of some reader-writer synchronization mechanism. For example, if one set of critical sections are guarded by read-acquisition of a given global reader-writer lock, while a second set of critical section are guarded by write-acquisition of that same reader-writer lock, then the first set of critical sections will be the read-side critical sections for that lock. Any number of threads may concurrently execute the read-side critical sections, but only if no thread is executing one of the write-side critical sections.

**Reader-Writer Lock**: A reader-writer lock is a mutual-exclusion mechanism that permits any number of reading threads, or but one writing thread, into the set of critical sections guarded by that lock. Threads attempting to write must wait until all pre-existing reading threads release the lock, and, similarly, if there is a pre-existing writer, any threads attempting to write must wait for the writer to release the lock. A key concern for reader-writer locks is “fairness”: can an unending stream of readers starve a writer or vice versa.

**Sequential Consistency**: A memory-consistency model where all memory references appear to occur in an order consistent with a single global order, and where each CPU’s memory references appear to all CPUs to occur in program order.

**Store Buffer**: A small set of internal registers used by a given CPU to record pending stores while the corresponding cache lines are making their way to that CPU. Also called “store queue”.

**Store Forwarding**: An arrangement where a given CPU refers to its store buffer as well as its cache so as to ensure that the software sees the memory operations performed by this CPU as if they were carried out in program order.

**Super-Scalar CPU**: A scalar (non-vector) CPU capable of executing multiple instructions concurrently. This is a step up from a pipelined CPU that executes multiple instructions in an assembly-line fashion — in a
super-scalar CPU, each stage of the pipeline would be capable of handling more than one instruction. For example, if the conditions were exactly right, the Intel Pentium Pro CPU from the mid-1990s could execute two (and sometimes three) instructions per clock cycle. Thus, a 200MHz Pentium Pro CPU could “retire”, or complete the execution of, up to 400 million instructions per second.

Transactional Memory (TM): Shared-memory synchronization scheme featuring “transactions”, each of which is an atomic sequence of operations that offers atomicity, consistency, isolation, but differ from classic transactions in that they do not offer durability. Transactional memory may be implemented either in hardware (hardwire transactional memory, or HTM), in software (software transactional memory, or STM), or in a combination of hardware and software (“unbounded” transactional memory, or UTM).

Vector CPU: A CPU that can apply a single instruction to multiple items of data concurrently. In the 1960s through the 1980s, only supercomputers had vector capabilities, but the advent of MMX in x86 CPUs and VMX in PowerPC CPUs brought vector processing to the masses.

Write Miss: A cache miss incurred because the corresponding CPU attempted to write to a cache line that is read-only, most likely due to its being replicated in other CPUs’ caches.

Write-Side Critical Section: A section of code guarded by write-acquisition of some reader-writer synchronization mechanism. For example, if one set of critical sections are guarded by write-acquisition of a given global reader-writer lock, while a second set of critical section are guarded by read-acquisition of that same reader-writer lock, then the first set of critical sections will be the write-side critical sections for that lock. Only one thread may execute in the write-side critical section at a time, and even then only if there are no threads are executing concurrently in any of the corresponding read-side critical sections.
Bibliography


symposium on Parallelism in algorithms and architectures, SPAA ’10, pages 325–334, New York, NY, USA, 2010. ACM.


[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


Paul E. McKenney. Re: [fwd: Re: [patch] real-time preemption, -rt-2.6.13-rc4-v0.7.52-01]. Available: [link], [Viewed March 14, 2006].

Paul E. McKenney. RCU Linux usage. Available: [link], [Viewed January 14, 2007].

Paul E. McKenney. Sleepable RCU. Available: [link], Revised: [link], [Viewed August 21, 2006].

Paul E. McKenney. The design of preemptible read-copy-update. Available: [link], [Viewed October 25, 2007].

Paul E. McKenney. [PATCH] QRCU with lockless fastpath. Available: [link], [Viewed March 27, 2008].

Paul E. McKenney. [patch rfc 0/9] RCU: Preemptible RCU. Available: [link], [Viewed October 25, 2007].

Paul E. McKenney. Priority-boosting RCU read-side critical sections. Available: [link], Revised: [link], [Viewed September 7, 2007].

Paul E. McKenney. RCU and unloadable modules. Available: [link], [Viewed November 22, 2007].

Paul E. McKenney. Using Promela and Spin to verify parallel algorithms. Available: [link], [Viewed August 7, 2007].

Paul E. McKenney. What is RCU? Available: [link], [Viewed July 6, 2007].

Paul E. McKenney. Hierarchical RCU. Available: [link], [Viewed November 6, 2008].

Paul E. McKenney. RCU part 3: the RCU API. Available: [link], [Viewed January 10, 2008].


[MMW07] Paul E. McKenney, Maged Michael, and Jonathan Walpole. Why the grass may not be greener on the other side: A comparison of locking vs. transactional memory.
In Programming Languages and Operating Systems, pages 1–5, New York, NY, USA, October 2007. ACM SIGOPS.


[Spi77] Keith R. Spitz. Tell which is which and you’ll be rich. Inscription on wall of dungeon, 1977.


Appendix I

Credits

I.1 Authors

I.2 Reviewers

- Alan Stern (Section 12.2).
- Andi Kleen (Section D.2).
- Andrew Morton (Section D.2).
- Andy Whitcroft (Section 8.3.2, Section 8.3.4, Section D.2, Section D.4).
- Artem Bityutskiy (Section 12.2, Appendix C).
- Dave Keck (Chapter C).
- David S. Horner (Section F.7, Section D.2).
- Gautham Shenoy (Section 8.3.2, Section 8.3.4, Section D.2).
- Ingo Molnar (Section D.2).
- “jarkao2”, AKA LWN guest #41960 (Section 8.3.4).
- Jonathan Walpole (Section 8.3.4).
- Josh Triplett (Section F, Section D.4, Section D.1, Section D.2).
- Lai Jiangshan (Section D.2).
- Manfred Spraul (Section D.2).
- Mathieu Desnoyers (Section D.2).
- Michael Factor (Section 15.2).
- Mike Fulton (Section 8.3.2).
- Nivedita Singhvi (Section D.4).
- Oleg Nesterov (Section D.4).
- Peter Zijlstra (Section 8.3.3, Section D.2).
- Richard Woodruff (Section C).
- Robert Bauer (Section D.4).
- Steve Rostedt (Section D.4).
- Suparna Bhattacharya (Section F).
- Vara Prasad (Section F.7).

Reviewers whose feedback took the extremely welcome form of a patch are credited in the git logs.

I.3 Machine Owners

A great debt of thanks goes to Martin Bligh, who originated the Advanced Build and Test (ABAT) system at IBM’s Linux Technology Center, as well as to Andy Whitcroft, Dustin Kirkland, and many others who extended this system.

Many thanks go also to a great number of machine owners:

- Andrew Theurer (Section F.7, Section D.2).
- Andy Whitcroft (Section D.2).
- Anton Blanchard (Section D.2).
- Chris McDermott (Section 2.2.2).
- Darrick Wong (Section D.2).
- David “Shaggy” Kleikamp (Section D.2).
- Jon M. Tollefson (Section D.2).
• Jose R. Santos (Section D.2).
• Nathan Lynch (Section D.2).
• Nishanth Aravamudan (Section D.2).
• Tim Pepper (Section D.2).

I.4 Original Publications
2. Section 8.3.2 (“RCU Fundamentals”) on page 96 originally appeared in Linux Weekly News [MW07].
3. Section 8.3.3 (“RCU Usage”) on page 102 originally appeared in Linux Weekly News [McK08c].
4. Section 8.3.4 (“RCU Linux-Kernel API”) on page 110 originally appeared in Linux Weekly News [McK08b].
5. Section C.7 (“Memory-Barrier Instructions For Specific CPUs”) on page 202 originally appeared in Linux Journal [McK05a, McK05b].
6. Section D.1 (“Sleepable RCU Implementation”) on page 211 originally appeared in Linux Weekly News [McK06b].
7. Section D.2 (“Hierarchical RCU Overview”) on page 217 originally appeared in Linux Weekly News [McK08a].

I.5 Figure Credits
1. Figure 2.1 (p 11) by Melissa McKenney.
2. Figure 2.2 (p 12) by Melissa McKenney.
3. Figure 2.3 (p 12) by Melissa McKenney.
4. Figure 2.4 (p 13) by Melissa McKenney.
5. Figure 2.5 (p 13) by Melissa McKenney.
6. Figure 2.6 (p 14) by Melissa McKenney.
7. Figure 2.7 (p 14) by Melissa McKenney.
8. Figure 2.8 (p 15) by Melissa McKenney.
9. Figure 5.1 (p 49) by Kornilios Kourtis.
10. Figure 5.2 (p 50) by Kornilios Kourtis.
11. Figure 5.3 (p 51) by Kornilios Kourtis.
12. Figure 5.17 (p 60) by Melissa McKenney.
13. Figure 5.19 (p 61) by Melissa McKenney.
14. Figure 5.20 (p 62) by Melissa McKenney.
15. Figure 6.1 (p 71) by Sarah McKenney.
16. Figure 6.2 (p 72) by Sarah McKenney.
17. Figure 12.2 (p 140) by Melissa McKenney.
18. Figure 12.6 (p 146) by David Howells.
19. Figure 12.7 (p 152) by David Howells.
20. Figure 12.8 (p 152) by David Howells.
21. Figure 12.9 (p 153) by David Howells.
22. Figure 12.10 (p 153) by David Howells.
23. Figure 12.11 (p 154) by David Howells.
24. Figure 12.12 (p 154) by David Howells.
25. Figure 12.13 (p 155) by David Howells.
26. Figure 12.14 (p 155) by David Howells.
27. Figure 12.15 (p 156) by David Howells.
28. Figure 12.16 (p 156) by David Howells.
29. Figure 12.17 (p 158) by David Howells.
30. Figure 12.18 (p 158) by David Howells.
31. Figure 13.2 (p 163) by Melissa McKenney.
32. Figure 15.1 (p 167) by Melissa McKenney.
33. Figure 15.2 (p 168) by Melissa McKenney.
34. Figure 15.3 (p 168) by Melissa McKenney.
35. Figure 15.4 (p 168) by Melissa McKenney.
36. Figure C.12 (p 206) by Melissa McKenney.
37. Figure D.1 (p 211) by Melissa McKenney.
38. Figure G.3 (p 329) by Kornilios Kourtis.
I.6 Other Support

We owe thanks to many CPU architects for patiently explaining the instruction- and memory-reordering features of their CPUs, particularly Wayne Cardoza, Ed Silha, Anton Blanchard, Tim Slegel, Juergen Probst, Ingo Adlung, and Ravi Arimilli. Wayne deserves special thanks for his patience in explaining Alpha’s reordering of dependent loads, a lesson that Paul resisted quite strenuously!

Portions of this material are based upon work supported by the National Science Foundation under Grant No. CNS-0719851.