

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

Translated by:

SeongJae Park
sj38.park@gmail.com

March 8, 2016

Legal Statement

This work represents the views of the editor and the authors and does not necessarily represent the view of their respective employers.

Trademarks:

- 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 trademark 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.¹ 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.² Some of this code is GPLv2-only, as it derives from the Linux kernel, while other code is GPLv2-or-later. See the comment headers of the individual source files within the CodeSamples directory in the git archive³ for the exact licenses. If you are unsure of the license for a given code fragment, you should assume GPLv2-only.

Combined work © 2005-2014 by Paul E. McKenney.

¹ <http://creativecommons.org/licenses/by-sa/3.0/us/>

² <http://www.gnu.org/licenses/gpl-2.0.html>

³ <git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git>

Contents

1	How To Use This Book	1
1.1	Roadmap	1
1.2	Quick Quizzes	2
1.3	Alternatives to This Book	2
1.4	Sample Source Code	3
1.5	Whose Book Is This?	3
2	Introduction	7
2.1	Historic Parallel Programming Difficulties	7
2.2	Parallel Programming Goals	8
2.2.1	Performance	8
2.2.2	Productivity	9
2.2.3	Generality	10
2.3	Alternatives to Parallel Programming	12
2.3.1	Multiple Instances of a Sequential Application	12
2.3.2	Use Existing Parallel Software	12
2.3.3	Performance Optimization	12
2.4	What Makes Parallel Programming Hard?	13
2.4.1	Work Partitioning	13
2.4.2	Parallel Access Control	14
2.4.3	Resource Partitioning and Replication	14
2.4.4	Interacting With Hardware	15
2.4.5	Composite Capabilities	15
2.4.6	How Do Languages and Environments Assist With These Tasks?	15
2.5	Discussion	15
3	Hardware and its Habits	17
3.1	Overview	17
3.1.1	Pipelined CPUs	17
3.1.2	Memory References	18
3.1.3	Atomic Operations	19
3.1.4	Memory Barriers	19
3.1.5	Cache Misses	20
3.1.6	I/O Operations	20
3.2	Overheads	21
3.2.1	Hardware System Architecture	21
3.2.2	Costs of Operations	22
3.3	Hardware Free Lunch?	22

3.3.1	3D Integration	23
3.3.2	Novel Materials and Processes	23
3.3.3	Light, Not Electrons	24
3.3.4	Special-Purpose Accelerators	24
3.3.5	Existing Parallel Software	24
3.4	Software Design Implications	25
4	Tools of the Trade	27
4.1	Scripting Languages	27
4.2	POSIX Multiprocessing	28
4.2.1	POSIX Process Creation and Destruction	28
4.2.2	POSIX Thread Creation and Destruction	29
4.2.3	POSIX Locking	30
4.2.4	POSIX Reader-Writer Locking	32
4.3	Atomic Operations	34
4.4	Linux-Kernel Equivalents to POSIX Operations	34
4.5	The Right Tool for the Job: How to Choose?	36
5	Counting	37
5.1	Why Isn't Concurrent Counting Trivial?	37
5.2	Statistical Counters	39
5.2.1	Design	39
5.2.2	Array-Based Implementation	39
5.2.3	Eventually Consistent Implementation	40
5.2.4	Per-Thread-Variable-Based Implementation	41
5.2.5	Discussion	42
5.3	Approximate Limit Counters	42
5.3.1	Design	43
5.3.2	Simple Limit Counter Implementation	43
5.3.3	Simple Limit Counter Discussion	48
5.3.4	Approximate Limit Counter Implementation	48
5.3.5	Approximate Limit Counter Discussion	48
5.4	Exact Limit Counters	48
5.4.1	Atomic Limit Counter Implementation	48
5.4.2	Atomic Limit Counter Discussion	52
5.4.3	Signal-Theft Limit Counter Design	52
5.4.4	Signal-Theft Limit Counter Implementation	53
5.4.5	Signal-Theft Limit Counter Discussion	56
5.5	Applying Specialized Parallel Counters	56
5.6	Parallel Counting Discussion	57
5.6.1	Parallel Counting Performance	57
5.6.2	Parallel Counting Specializations	58
5.6.3	Parallel Counting Lessons	59
6	Partitioning and Synchronization Design	61
6.1	Partitioning Exercises	61
6.1.1	Dining Philosophers Problem	61
6.1.2	Double-Ended Queue	63
6.1.3	Partitioning Example Discussion	69
6.2	Design Criteria	69

6.3	Synchronization Granularity	71
6.3.1	Sequential Program	71
6.3.2	Code Locking	72
6.3.3	Data Locking	73
6.3.4	Data Ownership	74
6.3.5	Locking Granularity and Performance	75
6.4	Parallel Fastpath	77
6.4.1	Reader/Writer Locking	77
6.4.2	Hierarchical Locking	78
6.4.3	Resource Allocator Caches	78
6.5	Beyond Partitioning	81
6.5.1	Work-Queue Parallel Maze Solver	82
6.5.2	Alternative Parallel Maze Solver	83
6.5.3	Performance Comparison I	84
6.5.4	Alternative Sequential Maze Solver	86
6.5.5	Performance Comparison II	86
6.5.6	Future Directions and Conclusions	87
6.6	Partitioning, Parallelism, and Optimization	87
7	Locking	89
7.1	Staying Alive	89
7.1.1	Deadlock	89
7.1.2	Livelock and Starvation	95
7.1.3	Unfairness	96
7.1.4	Inefficiency	96
7.2	Types of Locks	97
7.2.1	Exclusive Locks	97
7.2.2	Reader-Writer Locks	97
7.2.3	Beyond Reader-Writer Locks	97
7.2.4	Scoped Locking	98
7.3	Locking Implementation Issues	100
7.3.1	Sample Exclusive-Locking Implementation Based on Atomic Exchange	100
7.3.2	Other Exclusive-Locking Implementations	100
7.4	Lock-Based Existence Guarantees	102
7.5	Locking: Hero or Villain?	103
7.5.1	Locking For Applications: Hero!	103
7.5.2	Locking For Parallel Libraries: Just Another Tool	103
7.5.3	Locking For Parallelizing Sequential Libraries: Villain!	106
7.6	Summary	107
8	Data Ownership	109
8.1	Multiple Processes	109
8.2	Partial Data Ownership and <code>pthread</code> s	109
8.3	Function Shipping	110
8.4	Designated Thread	110
8.5	Privatization	110
8.6	Other Uses of Data Ownership	111

9 Deferred Processing	113
9.1 Reference Counting	113
9.1.1 Implementation of Reference-Counting Categories	114
9.1.2 Hazard Pointers	117
9.1.3 Linux Primitives Supporting Reference Counting	118
9.1.4 Counter Optimizations	119
9.2 Sequence Locks	119
9.3 Read-Copy Update (RCU)	121
9.3.1 Introduction to RCU	121
9.3.2 RCU Fundamentals	123
9.3.3 RCU Usage	129
9.3.4 RCU Linux-Kernel API	138
9.3.5 “Toy” RCU Implementations	143
9.3.6 RCU Exercises	155
9.4 Which to Choose?	156
9.5 What About Updates?	157
10 Data Structures	159
10.1 Motivating Application	159
10.2 Partitionable Data Structures	159
10.2.1 Hash-Table Design	160
10.2.2 Hash-Table Implementation	160
10.2.3 Hash-Table Performance	161
10.3 Read-Mostly Data Structures	163
10.3.1 RCU-Protected Hash Table Implementation	163
10.3.2 RCU-Protected Hash Table Performance	164
10.3.3 RCU-Protected Hash Table Discussion	166
10.4 Non-Partitionable Data Structures	167
10.4.1 Resizable Hash Table Design	167
10.4.2 Resizable Hash Table Implementation	168
10.4.3 Resizable Hash Table Discussion	172
10.4.4 Other Resizable Hash Tables	173
10.5 Other Data Structures	175
10.6 Micro-Optimization	176
10.6.1 Specialization	176
10.6.2 Bits and Bytes	176
10.6.3 Hardware Considerations	177
10.7 Summary	178
11 Validation	179
11.1 Introduction	179
11.1.1 Where Do Bugs Come From?	179
11.1.2 Required Mindset	180
11.1.3 When Should Validation Start?	181
11.1.4 The Open Source Way	182
11.2 Tracing	183
11.3 Assertions	183
11.4 Static Analysis	184
11.5 Code Review	184
11.5.1 Inspection	184

11.5.2 Walkthroughs	185
11.5.3 Self-Inspection	185
11.6 Probability and Heisenbugs	186
11.6.1 Statistics for Discrete Testing	187
11.6.2 Abusing Statistics for Discrete Testing	188
11.6.3 Statistics for Continuous Testing	188
11.6.4 Hunting Heisenbugs	190
11.7 Performance Estimation	193
11.7.1 Benchmarking	193
11.7.2 Profiling	194
11.7.3 Differential Profiling	194
11.7.4 Microbenchmarking	194
11.7.5 Isolation	195
11.7.6 Detecting Interference	196
11.8 Summary	197
12 Formal Verification	201
12.1 General-Purpose State-Space Search	201
12.1.1 Promela and Spin	201
12.1.2 How to Use Promela	204
12.1.3 Promela Example: Locking	206
12.1.4 Promela Example: QRCU	208
12.1.5 Promela Parable: dynticks and Preemptible RCU	212
12.1.6 Validating Preemptible RCU and dynticks	215
12.2 Special-Purpose State-Space Search	227
12.2.1 Anatomy of a Litmus Test	228
12.2.2 What Does This Litmus Test Mean?	228
12.2.3 Running a Litmus Test	229
12.2.4 PPCMEM Discussion	229
12.3 Axiomatic Approaches	230
12.4 SAT Solvers	231
12.5 Summary	232
13 Putting It All Together	235
13.1 Counter Conundrums	235
13.1.1 Counting Updates	235
13.1.2 Counting Lookups	235
13.2 RCU Rescues	235
13.2.1 RCU and Per-Thread-Variable-Based Statistical Counters	236
13.2.2 RCU and Counters for Removable I/O Devices	237
13.2.3 Array and Length	238
13.2.4 Correlated Fields	238
13.3 Hashing Hassles	239
13.3.1 Correlated Data Elements	239
13.3.2 Update-Friendly Hash-Table Traversal	240

14 Advanced Synchronization	241
14.1 Avoiding Locks	241
14.2 Memory Barriers	241
14.2.1 Memory Ordering and Memory Barriers	242
14.2.2 If B Follows A, and C Follows B, Why Doesn't C Follow A?	243
14.2.3 Variables Can Have More Than One Value	243
14.2.4 What Can You Trust?	244
14.2.5 Review of Locking Implementations	248
14.2.6 A Few Simple Rules	249
14.2.7 Abstract Memory Access Model	249
14.2.8 Device Operations	250
14.2.9 Guarantees	250
14.2.10 What Are Memory Barriers?	251
14.2.11 Locking Constraints	257
14.2.12 Memory-Barrier Examples	258
14.2.13 The Effects of the CPU Cache	260
14.2.14 Where Are Memory Barriers Needed?	261
14.3 Non-Blocking Synchronization	262
14.3.1 Simple NBS	262
14.3.2 NBS Discussion	263
15 Parallel Real-Time Computing	265
15.1 What is Real-Time Computing?	265
15.1.1 Soft Real Time	265
15.1.2 Hard Real Time	265
15.1.3 Real-World Real Time	266
15.2 Who Needs Real-Time Computing?	269
15.3 Who Needs Parallel Real-Time Computing?	270
15.4 Implementing Parallel Real-Time Systems	270
15.4.1 Implementing Parallel Real-Time Operating Systems	271
15.4.2 Implementing Parallel Real-Time Applications	280
15.4.3 The Role of RCU	282
15.5 Real Time vs. Real Fast: How to Choose?	283
16 Ease of Use	285
16.1 What is Easy?	285
16.2 Rusty Scale for API Design	285
16.3 Shaving the Mandelbrot Set	286
17 Conflicting Visions of the Future	289
17.1 The Future of CPU Technology Ain't What it Used to Be	289
17.1.1 Uniprocessor Über Alles	289
17.1.2 Multithreaded Mania	290
17.1.3 More of the Same	291
17.1.4 Crash Dummies Slamming into the Memory Wall	291
17.2 Transactional Memory	292
17.2.1 Outside World	293
17.2.2 Process Modification	296
17.2.3 Synchronization	299
17.2.4 Discussion	302

17.3	Hardware Transactional Memory	302
17.3.1	HTM Benefits WRT to Locking	303
17.3.2	HTM Weaknesses WRT Locking	305
17.3.3	HTM Weaknesses WRT to Locking When Augmented	309
17.3.4	Where Does HTM Best Fit In?	312
17.3.5	Potential Game Changers	312
17.3.6	Conclusions	314
17.4	Functional Programming for Parallelism	314
A	Important Questions	317
A.1	What Does “After” Mean?	317
A.2	What is the Difference Between “Concurrent” and “Parallel”?	319
A.3	What Time Is It?	320
B	Synchronization Primitives	321
B.1	Organization and Initialization	321
B.1.1	smp_init():	321
B.2	Thread Creation, Destruction, and Control	321
B.2.1	create_thread()	321
B.2.2	smp_thread_id()	322
B.2.3	for_each_thread()	322
B.2.4	for_each_running_thread()	322
B.2.5	wait_thread()	322
B.2.6	wait_all_threads()	322
B.2.7	Example Usage	322
B.3	Locking	323
B.3.1	spin_lock_init()	323
B.3.2	spin_lock()	323
B.3.3	spin_trylock()	323
B.3.4	spin_unlock()	323
B.3.5	Example Usage	323
B.4	Per-Thread Variables	323
B.4.1	DEFINE_PER_THREAD()	323
B.4.2	DECLARE_PER_THREAD()	324
B.4.3	per_thread()	324
B.4.4	__get_thread_var()	324
B.4.5	init_per_thread()	324
B.4.6	Usage Example	324
B.5	Performance	324
C	Why Memory Barriers?	325
C.1	Cache Structure	325
C.2	Cache-Coherence Protocols	327
C.2.1	MESI States	327
C.2.2	MESI Protocol Messages	327
C.2.3	MESI State Diagram	328
C.2.4	MESI Protocol Example	329
C.3	Stores Result in Unnecessary Stalls	329
C.3.1	Store Buffers	329
C.3.2	Store Forwarding	330

C.3.3	Store Buffers and Memory Barriers	331
C.4	Store Sequences Result in Unnecessary Stalls	333
C.4.1	Invalidate Queues	333
C.4.2	Invalidate Queues and Invalidate Acknowledge	333
C.4.3	Invalidate Queues and Memory Barriers	334
C.5	Read and Write Memory Barriers	335
C.6	Example Memory-Barrier Sequences	336
C.6.1	Ordering-Hostile Architecture	336
C.6.2	Example 1	337
C.6.3	Example 2	337
C.6.4	Example 3	337
C.7	Memory-Barrier Instructions For Specific CPUs	337
C.7.1	Alpha	340
C.7.2	AMD64	341
C.7.3	ARMv7-A/R	342
C.7.4	IA64	342
C.7.5	MIPS	343
C.7.6	PA-RISC	343
C.7.7	POWER / PowerPC	343
C.7.8	SPARC RMO, PSO, and TSO	344
C.7.9	x86	345
C.7.10	zSeries	345
C.8	Are Memory Barriers Forever?	346
C.9	Advice to Hardware Designers	346
D	Answers to Quick Quizzes	349
D.1	How To Use This Book	349
D.2	Introduction	350
D.3	Hardware and its Habits	353
D.4	Tools of the Trade	356
D.5	Counting	361
D.6	Partitioning and Synchronization Design	374
D.7	Locking	378
D.8	Data Ownership	385
D.9	Deferred Processing	386
D.10	Data Structures	403
D.11	Validation	405
D.12	Formal Verification	410
D.13	Putting It All Together	414
D.14	Advanced Synchronization	416
D.15	Parallel Real-Time Computing	419
D.16	Ease of Use	421
D.17	Conflicting Visions of the Future	421
D.18	Important Questions	424
D.19	Synchronization Primitives	424
D.20	Why Memory Barriers?	425
E	Glossary and Bibliography	429

F Credits	457
F.1 Authors	457
F.2 Reviewers	457
F.3 Machine Owners	457
F.4 Original Publications	457
F.5 Figure Credits	458
F.6 Other Support	459

Chapter 1

How To Use This Book

이 책의 목적은 당신이 shared-memory parallel machine 을 정확성을 해치지 않으면서 프로그램하는 것을 돋는 것입니다.¹ 우린 이 책의 디자인 원칙이 당신이 적어도 몇몇 병렬 프로그래밍의 함정들은 피하는데 도움을 주길 바랍니다. 즉, 당신은 이 책을 완성된 성당이라기보단 새로 건물을 지을 토대라고 생각해야 합니다. 이를 따르기로 한다면, 당신의 임무는 신나는 병렬 프로그래밍 세계에 진보 — 결국엔 이 책을 구식으로 만들게 될 진보요 — 를 가져오는것을 돋는겁니다. 병렬 프로그래밍은 사람들이 말하는 것처럼 어렵지 않습니다. 그리고 우리는 이 책이 당신의 병렬 프로그래밍 과제를 더 쉽고 재밌게 만들길 바랍니다.

짧게 말해서, 병렬 프로그래밍을 과학, 연구, 뭔가 대단한 과제에 적용하려 하면, 그것은 곧바로 엔지니어링 문제가 되어버립니다. 따라서 우리는 특정 병렬 프로그래밍 작업들을 조사하고, 거기서 얻은 것들을 어떻게 다른 곳에도 적용할 수 있는지 이야기합니다. 그러한 적용은, 일부 케이스에서는 놀랍게도 자동화도 가능합니다.

이 책은 성공적 병렬 프로그래밍 프로젝트에 숨어있는 엔지니어링 비법들을 알려주는 것이 새로운 세대의 병렬성 해커들을 느리고 고통스럽게 오래된 바퀴를 다시 만들어내는 대신, 그들의 에너지와 창의성을 새로운 개척자에게 집중하도록 도울 수 있길 바랍니다. 우린 진심으로 병렬 프로그래밍이 당신에게 최소한, 우리에게도 왔던 재미와 흥분, 그리고 도전을 당신에게 가져다 주길 바랍니다.

1.1 Roadmap

이 책은 극소수의 영역에만 적용 가능한 최적의 알고리즘의 모음이라기보다는 꽤넓게 적용될 수 있고 많이 사용된 디자인 테크닉들을 소개하는 안내서입니다. 당신은 지금 Chapter 1을 읽고 있습니다, 알고 있겠지만요.

¹또는, 좀 더 정확히는, 병렬성 없는 프로그래밍에 비해 정확성을 털 해치면서요.

Chapter 2 은 병렬 프로그래밍에 대해 전반적으로 간단히 살펴봅니다.

Chapter 3 는 shared-memory parallel hardware 에 대해 소개합니다. 무엇보다도, 당신이 코드가 동작하게 될 하드웨어에 대해 모르면 좋은 병렬성 있는 코드를 작성하기가 어렵습니다. 하드웨어는 계속 발전할테니, 이 챕터는 항상 시대에 뒤쳐질겁니다. 따라서 우리는 내용을 최신으로 유지하기 위해 최선을 다할겁니다. Chapter 4 는 이어서 shared-memory 병렬 프로그래밍의 기본을 매우 간략히 소개합니다.

Chapter 5 에서는 가장 간단한 문제, 카운팅을 병렬화하는 작업에 대해 자세히 알아봅니다. 대부분은 카운팅에 대해서는 잘 알고 있을테니, 이 챕터에서는 보다 흔한 컴퓨터 사이언스에서의 문제들에 정신을 뺏기지 않고 병렬 프로그래밍 이슈에 대해서만 집중할 수 있을 겁니다. 이 챕터는 병렬 프로그래밍 수업에서 가장 많이 활용되곤 합니다.

Chapter 6 에서는 Chapter 5 에서 알아본 문제들을 해결하는 설계 레벨의 방법들을 소개합니다. 가능할때엔 병렬성을 설계 레벨에서 해결하는 것이 중요합니다: Dijkstra [Dij68] 의 말을 바꿔쓰자면, “개선된 병렬성은 최적이 아닌 것으로 본다” [McK12c].

이어지는 세개의 챕터는 세개의 중요한 동기화 방법을 각각 설명합니다. Chapter 7 에서는 적어도 2014년에 와서는 제품 수준의 병렬 프로그래밍에 있어 가장 많이 사용되지도 않고, 일반적으로 병렬 프로그래밍의 가장 악랄한 악당으로 여겨지는 락킹에 대해 알아봅니다. Chapter 8 는 과소평가되곤 하지만 실제로는 놀랍도록 여러 분야에 사용 가능하고 강력한 방법인 데이터 소유권에 대해 알아봅니다. 마지막으로 Chapter 10 에서는 레퍼런스 카운팅, 해저드 포인터, 순차적 락킹, 그리고 RCU 를 포함한 deferred-processing 메커니즘들을 소개합니다.

챕터 10 에서는 앞에서 배운 내용들을 해시 테이블에 적용해 봅니다. 해시 테이블은 그 훌륭한 데이터 분리성으로 인해 널리 사용되고 있고, 때문에 (보통은) 훌륭한

성능과 확장성을 보입니다.

많은 사람들이 비통해하듯이, validation (실증) 없이 이루어지는 병렬 프로그래밍은 비참한 실패로의 분명한 지름길입니다. 챕터 11에서는 다양한 테스트 방법을 다룹니다. 물론 모든 부분에 대해 프로그램의 신뢰성을 테스트 하는 것은 불가능합니다. 따라서 챕터 12에서는 몇개의 실용적인 형식 검증 (formal verification) 방법에 대해 간단히 다룹니다.

챕터 13에서는 적당한 크기의 병렬 프로그래밍 문제들을 다룹니다. 이런 문제들의 어려움은 다양하지만 앞 챕터들의 내용을 이해한 사람에게는 적당할 겁니다.

챕터 14에서는 메모리 배리어와 non-blocking 동기화를 포함한 고급 동기화 방법을 알아봅니다. 이어지는 챕터 16는 몇몇 ease-of-use 기법들을 이야기합니다. 마지막으로, 챕터 17에서는 공유 메모리 병렬 시스템 설계, 소프트웨어 / 하드웨어 트랜잭션 메모리, 그리고 병렬성을 위한 함수형 프로그래밍을 포함해 몇몇 가능할 법한 미래의 방향에 대해 알아봅니다.

이 책의 끝에는 몇개의 부록이 있습니다. 그 중 가장 유명한 건 메모리 배리어에 대해 다루고 있는 Appendix C 일 겁니다. Appendix D 는 다음 섹션에서 이야기할, 퀴즈들의 답을 담고 있습니다.

1.2 Quick Quizzes

“Quick quizzes” 는 이 책 전반에 걸쳐 여기저기서 나오고, 그에 대한 답은 페이지 349 부터 시작하는 Appendix D 에서 볼 수 있습니다. 그 중 일부는 그 퀴즈가 제출된 곳의 내용에 기반하지만, 몇몇은 그 섹션 이외의 내용에 대해서도 생각해야 할 거고, 일부는 당신이 알고 있는 모든 내용을 필요로 할 수도 있습니다. 최대한 노력했다는 가정 하에, 당신이 이 책으로부터 얻을 수 있는 것은 당신이 배운 내용을 얼마나 실제로 응용하는지에 달려있습니다. 따라서, 답을 보기 전에 퀴즈를 풀기 위해 많은 노력을 기울인 독자는 향상된 병렬 프로그래밍에 대한 이해와 함께 그들의 노력이 결실로 돌아옴을 알 수 있을 것입니다.

Quick Quiz 1.1: 이 Quick Quiz 들의 답은 어디에 있을까요?

Quick Quiz 1.2: 몇몇 퀴즈는 저자의 입장이 아니라 독자의 입장에서 쓰인 것 같은데요. 그런 의도가 맞나요?

Quick Quiz 1.3: 전 퀴즈를 좋아하지 않아요. 어떡하죠?

간략히 정리하자면, 당신이 해당 내용에 대해 깊은 이해가 필요하다면, 어느 정도의 시간은 퀴즈의 답을

구하는데 사용할 필요가 있습니다. 가만히 내용을 읽기만 하는 것도 물론 의미있습니다만, 완벽한 문제 해결 능력을 갖는 것은 실질적인 문제를 풀어보는 것도 필요로 합니다.

저는 이 깨달음을 저의 늦깎이 박사 과정에서 힘들게 얻었습니다. 저는 제게 익숙한 주제를 공부했는데, 제가 해당 챕터의 연습문제 중에 제가 곧바로 머리 속에서 답할 수 있는 내용이 얼마 안된다는 사실에 놀랐습니다.² 저 스스로를 그 문제들을 풀도록 강제하는 것은 해당 내용에 대한 제 기억력을 매우 높였습니다. 따라서 저는 이 퀴즈들에 대해 제가 스스로 하지 않았던 것을 여러분에게 강요하고 있지는 않아요!

1.3 Alternatives to This Book

Knuth 가 깨달았던 것과 같이, 당신의 책의 내용에 끝이 있으려면 그 책의 내용은 어딘가에 집중되어 있어야 합니다. 이 책은 운영체계 커널, 병렬 데이터 관리 시스템, 저수준 라이브러리 등과 같은 소프트웨어 스택의 바닥 쪽에 있는 소프트웨어를 주요 대상으로 두고 공유메모리 기반 병렬 프로그래밍에 중점을 둡니다. 이 책에서 프로그래밍 언어로는 C 언어를 사용합니다.

당신이 병렬성의 다른 분야에 관심있다면, 다른 책을 보는 편이 좋을 겁니다. 만약 그렇다면, 다행히도 여러 좋은 책들이 있습니다:

1. 보다 학술적이고 정밀하게 병렬 프로그래밍을 다루고 싶다면, Herlihy 와 Shavit 의 책 [HS08]이 당신에게 적합할 겁니다. 이 책은 추상화된 하드웨어에서 제공하는 원초적 기능들의 조합으로 시작해서 락킹과 리스트, 큐, 해시 테이블, 그리고 카운터를 포함한 간단한 자료구조들을 다루고, 마지막으로 트랜잭션 메모리를 다룹니다. Michael Scott 의 책 [Sco13] 은 비슷한 내용을 보다 소프트웨어 엔지니어링에 중점을 두어서 접근합니다. 그리고, 제가 알기로는 최초로 RCU 에 대한 내용의 섹션을 담은 채로 정식으로 학계에 출간된 최초의 책입니다.
2. 당신이 프로그래밍 언어적 실용성 관점에서의 병렬 프로그래밍에 대한 학술적 처리를 알고 싶다면 Scott 의 책 [Sco06] 의 concurrency (동시성) 챕터를 재밌게 볼 수 있을 겁니다.
3. 객체 지향 패턴 전문가들이 병렬 프로그래밍을 어떻게 취급하는지 C++ 위주로 알고 싶다면 Schmidt 의 POSA 시리즈 [SSRB00, BHS07] 의 Volume 2

² 아마 그래서 제 교수님들은 제가 그 수업을 포기하지 못하게 하셨다고 생각합니다

와 4 를 읽어봐도 좋을 겁니다. 특히 Volume 4 는 그 작업물을 도매점 어플리케이션에 적용해본 내용에 대한 재미있는 챕터들이 있습니다. 이 예제가 얼마나 실제 상황에 가까운지는 병렬성에 내재된 문제들은 종종 실제 세계의 응용사례에서 시작된다는 이야기를 하는 “Partitioning the Big Ball of Mud” 라는 제목의 섹션에서 증명되었습니다.

- 당신이 리눅스 커널 디바이스 드라이버를 다루고 싶다면 Corbet, Rubini, 그리고 Kroah-Hartman 이 쓴 “Linux Device Drivers” [CRKH05], 그리고 Linux Weekly News 웹사이트 (<http://lwn.net/> 를 반드시 봐야 합니다. 리눅스 커널 내부에 대한 일반적 내용에 대해서는 많은 책과 자료들이 있습니다.
- 당신의 주요 관심사가 과학 / 기술 분야 컴퓨팅이라면, 그리고 패턴주의자의 접근방법을 선호한다면, Mattson 의 책 [MSM05] 을 읽어 보십시오. 그 책에서는 Java, C/C++, OpenMP, 그리고 MPI 를 다룹니다. 거기서 이야기하는 패턴들은 첫째로 설계, 다음으로 구현에 매우 집중되어 있습니다.
- 당신의 주요 관심사가 과학 / 기술 분야 컴퓨팅이고 GPU, CUDA, 그리고 MPI 에 관심이 있다면, Norm Matloff 의 ‘Programming on Parallel Machines’ [Mat13] 을 한번쯤 보세요.
- POSIX 쓰레드에 관심 있다면, David R. Butenhof 의 책 [But97] 을 읽어보세요. 또한, W. Richard Stevens 의 책 [Ste92] 은 UNIX 와 POSIX 를 다루고, Stewart Weiss 의 수업노트 [Wei13] 는 좋은 예제들과 함께 완전하고 접근 가능한 소개를 제공합니다.
- C++11 에 관심 있다면, Anthony Williams 의 “C++ Concurrency in Action: Practical Multithreading” [Wil12] 를 좋아할 수 있을 겁니다.
- C++ 에 관심 있지만 Windows 환경에 있다면 Dr. Dobb 의 잡지 [Sut08] 에 실린 Herb Sutter 의 “Effective Concurrency” 시리즈를 읽어보세요. 이 시리즈는 병렬성에의 일반적 접근을 잘 소개합니다.
- Intel Threading Building Blocks 를 사용해 보고 싶다면, James Reinder 의 책 [Rei07] 이 아마 찾으시는 책일 겁니다.
- 다양한 멀티 프로세서 시스템의 하드웨어 캐시 구성이 커널 내부 구현에 어떤 영향을 끼치는지 궁금한 사람이라면 이 연구 [Sch94] 에 실린 Curt Schimmel 의 고전적 접근을 한번 봐야 합니다.

- 마지막으로, Java 사용자라면 Doug Lea 의 교재 [Lea97, GPB⁺07] 가 큰 도움이 될겁니다.

하지만, 로우 레벨의, 특히 C 로 구현된 소프트웨어를 위한 병렬적 설계의 기본적 내용에 관심이 있다면, 이 책을 계속 읽으세요!

1.4 Sample Source Code

이 책은 많은 소스 코드를 인용하고 있고, 많은 경우 그 소스 코드는 이 책의 git tree 안의 CodeSamples 디렉토리 안에 있습니다. 예를 들어 UNIX 시스템에서는 다음과 같이 명령을 내릴 수 있을 겁니다:

```
find CodeSamples -name rCU_rcplS.c -print
```

이 명령문은 Section 9.3.5 에 사용된 rCU_rcplS.c 파일의 위치를 알려줄 겁니다. 다른 종류의 시스템에서는 나름대로 파일 이름으로 해당 파일의 위치를 알려주는 방법이 있을 겁니다.

1.5 Whose Book Is This?

표지에서 이야기했듯, 이 책의 편집자는 Paul E. McKenney 입니다. 하지만, Paul 은 perfbook@vger.kernel.org 이메일 리스트를 통한 기여를 받습니다. 이런 기여들은 텍스트 이메일, 책의 LATEX 소스에 대한 패치, 심지어는 git pull 요청까지 어떤 형태라도 상관 없습니다. 당신에게 가장 좋은 방법을 사용하세요.

패치를 만들거나 git pull 요청을 보내기 위해서는, [git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git](http://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git) 에 있는 이 책의 LATEX 소스가 필요할 겁니다. 또한, 당연하게도 git 과 LATEX 이 필요하겠죠. git 과 LATEX 은 대부분의 주요 리눅스 배포본에 포함되어 있습니다. 그 외에도 당신이 사용하고 있는 배포본에 따라 다른 패키지들도 필요할 수 있습니다. 일부 유명한 배포본에 대해 필요한 패키지 목록은 이 책의 LATEX 소스의 FAQ.txt 파일에 있습니다.

이 책의 LATEX 소스 트리를 만들고 보려면 Figure 1.1 에 있는 리눅스 커맨드를 사용하세요. 일부 환경에서는 `perfbook.pdf` 를 표시하는데 사용되는 `evince` 커맨드가 `acroread` 라던지 다른 커맨드로 바뀌어야 할 수도 있습니다. `git clone` 커맨드는 PDF 를 최초 만들 때 한번만 수행되면 됩니다. 한번 pdf 를 생성한 이후로는 Figure 1.2 의 커맨드를 수행함으로써 그사이 업데이트된 내용을 얻어오고 업데이트된 내용이 포함된 PDF 를 만들 수 있습니다. Figure 1.2 의 커맨드는

```

1 git clone git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git
2 cd perfbook
3 make
4 evince perfbook.pdf & # Two-column version
5 make perfbook-1c.pdf
6 evince perfbook-1c.pdf & # One-column version for e-readers

```

Figure 1.1: Creating an Up-To-Date PDF

```

1 git remote update
2 git checkout origin/master
3 make
4 evince perfbook.pdf & # Two-column version
5 make perfbook-1c.pdf
6 evince perfbook-1c.pdf & # One-column version for e-readers

```

Figure 1.2: Generating an Updated PDF

반드시 Figure 1.1에 나온 커맨드가 생성한 perfbook 디렉토리 안에서 수행되어야 합니다.

이 책의 PDF들은 가끔마다 <http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html>과 <http://www.rdrop.com/users/paulmck/perfbook/>에 업로드 됩니다.

실제 패치를 보내고 git pull 요청을 보내는 과정은 리눅스 소스 트리의 Documentation/SubmittingPatches에 문서화되어 있는 리눅스 커널의 방식과 유사합니다. 한가지 중요한 필수사항은 각 패치(git pull 요청의 경우라면 커밋)은 반드시 다음과 같은 형태의, 유효한 Signed-off-by: 라인을 포함해야 합니다:

Signed-off-by: My Name <myname@example.org>

Signed-off-by: 라인을 포함한 패치의 예를 보려면 <http://lkml.org/lkml/2007/1/15/219>를 보십시오.

Signed-off-by: 라인은 매우 중요한 의미를 갖는데, 다음과 같은 내용을 당신이 선서한다는 의미입니다:

1. 해당 기여는 모두 또는 부분적으로 저에 의해 만들어졌고, 저는 해당 파일에 표시된 오픈 소스 라이센스 아래 해당 기여를 제출할 권리를 갖습니다; 또는
2. 해당 기여는 제가 알기로는 적절한 오픈 소스 라이센스 하에 만들어진 기존의 작업물에 기초하며, 파일에 표시된 대로 저는 그 라이센스 아래 해당 작업물을 모두 또는 일부분 제가 수정하고 기준과 같은 오픈 소스 라이센스 아래(제가 다른 라이센스로 제출할 권리를 갖지 않았다면) 제출할 권리를 갖습니다; 또는

3. 해당 기여는 제게 (a), (b) 또는 (c)를 선서해 준 다른 사람에 의해 제공되었고 저는 이를 수정한 바가 없습니다.

4. 해당 기여는 다른 집단의 어떤 지적 재산권 관련 분쟁이나 권리와 무관하게 만들어졌습니다.

5. 저는 이 프로젝트와 해당 기여가 공적이며 이 기여에 대한 기록(제가 기여와 함께 제공한 모든 개인적 정보와 저의 sign-off를 포함하여)이 불특정 기간 유지되고 이 프로젝트나 사용된 오픈 소스 라이센스(들)과 연관되어 재배포 될 수 있음을 이해하고 동의합니다.

이건 리눅스 커널에서 사용되는 개발자의 유래에 대한 선서 (DCO) 1.1과 유사합니다. 다만 한가지, item #4가 추가되었습니다. 추가된 해당 항목은 당신이 해당 기여를 다른 사람으로부터 받은게 아니라 직접 만들었다고 이야기합니다. 만약 여러 사람이 하나의 기여를 함께 작성했다면, 각자가 모두 Signed-off-by: 라인을 가져야 합니다.

당신은 당신의 진짜 이름을 사용해야만 합니다: 안타깝지만 저는 익명으로 제공된 기여는 받을 수 없습니다.

이 책은 미국 영어를 사용합니다만, 오픈소스라는 환경상 이 책은 번역이 허용되고, 전 개인적으로 번역을 장려합니다. 이 책에 붙어 있는 오픈 소스 라이센스는 추가적으로, 원한다면 당신이 당신의 번역을 판매할 수도 있게 허용합니다. 번역본의 복사본을 (가능하다면 하드카피로) 보내 주실 것을 부탁드립니다만, 이는 어디까지나 부탁드리는 것이고, 당신이 이미 Creative Commons 와 GPL license로 갖는 권리에 우선하지 않습니다. 현재 진행중인 번역 작업의 목록을 보려면 FAQ.txt 파일을 봐 주십시오. 전 최소 한개의 챕터 이상이 이미 모두 번역되었다면 번역 작업이 “진행 중”이라고 여깁니다.

이 섹션의 시작에서 이야기 했듯, 전 이 책의 편집자입니다. 하지만, 당신이 기여를 하기로 하면, 이것은 당신의 책이기도 합니다. 이제, Chapter 2, introduction 을 시작합니다.

Chapter 2

Introduction

병렬 프로그래밍은 해커가 시도해 볼 수 있는 가장 어려운 영역 중 하나라는 평판을 가지고 있습니다. 논문과 서적들은 데드락, 라이브락, 레이스 컨디션, 논-디터미니즘, 확장성에서의 암달의 법칙 한계, 그리고 가혹한 리얼타임 환경에서의 응답시간의 위험을 경고하고 있고, 이런 위험들은 실제로 존재합니다. 우리 저자들은 감정적 상처, 하얗게 세는 머리카락, 그리고 탈모를 겪어가면서 셀 수 없을 만큼 오랜 시간 그 문제들을 다뤄왔습니다.

하지만, 모든 새로운 기술들이 처음엔 곧바로 사용하기엔 너무 어려웠지만, 시간에 따라 점점 쉬워져 왔습니다. 예컨대, 과거엔 자동차를 운전하는 것이 흔치 않은 능력이었지만 지금은 흔한 일입니다. 이런 극적인 변화는 두 가지 기본적 이유에서 나옵니다: (1) 자동차가 점점 싸고 흔해져서 더 많은 사람들이 운전을 배울 기회를 갖습니다, 그리고 (2) 자동 변속기, 자동 죠크, 자동 발진기, 매우 향상된 신뢰도, 그리고 다른 향상된 기술의 접목 등으로 인해 자동차를 운전하기가 보다 쉬워집니다.

컴퓨터를 포함해서, 다른 기술들도 마찬가지입니다. 더이상 프로그램을 짜기 위해 천공카드를 만질 필요 없습니다. 스프레드시트 프로그램들은 수십년전이었다면 전문가 집단이 필요했을 계산 결과를 프로그래머가 아닌 사람들도 그들의 컴퓨터에서 쉽게 얻을 수 있게 해줍니다. 가장 반박하기 어려운 예는 아마도 지금은 흔해진 소셜 네트워크 서비스들과 함께 약 2000년대 초쯤부터 전문적으로 교육받지 않은 사람도 접근하기 시작한 웹서핑과 컨텐츠 제작일 겁니다. 1968년만 해도, 그런 컨텐츠 제작은 연구 주제 [Eng68]에 가까웠습니다. 그 당시엔 “UFO 가 백악관 잔디밭에 착륙하기처럼”[Gri00] 어렵다고 이야기 하곤 했죠.

그러니, 병렬 프로그래밍이 지금 그렇듯이 영원히 어려운 주제로 남을 거라고 주장하고자 한다면, 과거 수백, 수천년의 다양한 노력에서 만들어진 반례를 생각하세요. 그래도 주장하고자 한다면, 증명은 당신의 몫입니다.

2.1 Historic Parallel Programming Difficulties

제목에서 알 수 있듯이, 이 책은 좀 다른 접근법을 취합니다. 병렬 프로그래밍의 어려움을 이야기하기보다는, 먼저 병렬 프로그래밍이 어려운 이유를 알아보고나서 독자들이 이 어려움들을 이겨낼 수 있도록 돕습니다. 이어서 설명하겠지만, 이 어려움들은 다음의 카테고리들로 분류될 수 있습니다:

1. 역사적으로 비싸고 흔치 않았던 병렬 시스템들.
2. 연구자와 전문가들의 부족한 병렬 시스템 경험.
3. 공식적으로 접할 수 있는 병렬 코드의 양의 부족.
4. 병렬 프로그래밍에 대해 널리 알려진 엔지니어링 교육의 부족.
5. 연산에 비해, 심지어 타이트하게 설계된 공유메모리 컴퓨터에서도 존재하는, 커뮤니케이션의 높은 오버헤드.

이렇게 역사적으로 존재했던 문제들 중 다수는 해결되어 가는 중입니다. 먼저, 지난 수십년간, 무어의 법칙 덕에 병렬 시스템의 가격은 집 여러채 가격에서 자전거 한대 가격 정도로 떨어졌습니다. 멀티코어 CPU의 장점에 대한 논문은 1996년초 [ONH⁺96]에도 나왔습니다. IBM은 2000년도에 동시에 수행되는 멀티쓰레딩 기능을 그들의 하이엔드 POWER 제품군에 넣었고, 2001년에는 멀티코어를 구현했습니다. 2000년 11월, Intel은 일반 시장용 제품인 Pentium에 하이퍼쓰레딩 기능을 넣었고, 2005년에는 AMD와 Intel 둘 다 듀얼코어 CPU를 내놓았습니다. Sun은 2005년 후반, 멀티코어/멀티쓰레드 기능을 갖춘 Niagara로 그 뒤를 이었습니다. 2008년에 이르러서는 사실상 싱글 CPU 데스크탑을 찾아보기 어려워졌습니다. 이 시점부터 싱글코어 CPU는 넷북이나 임베디드 기기에서나 사용되었습니다. 2012년에

이르러, 심지어 스마트폰도 멀티코어 CPU를 사용하기 시작했습니다.

둘째로, 저가에 쉽게 구할 수 있는 멀티코어 시스템이 많아진 것은, 한때엔 접하기 쉽지 않았던 병렬 프로그래밍 경험이 거의 모든 연구자와 전문가에게 가능해졌다는 뜻입니다. 실제로, 병렬 시스템은 이제 학생이나 취미로 컴퓨터 만지는 사람도 살 수 있는 가격입니다. 따라서 우리는 병렬 시스템을 이용한 발명과 혁신이 엄청나게 많아질 것임을 예상할 수 있고, 이렇게 친숙해진 병렬 시스템 환경은 한때 엄청나게 접근하기 어렵던 병렬 프로그래밍 영역을 훨씬 편하고 일반적인 영역으로 만들 것입니다.

셋째로, 20세기에는 병렬 소프트웨어로 구현된 큰 시스템은 거의 항상 독점과 비밀로 갇혀 있었습니다. 반면, 21세기에는 다행히도 리눅스 커널 [Tor03], 데이터베이스 시스템들 [Pos08, MS08], 그리고 메세지 패싱 시스템들 [The08, UoC08]을 포함해, 많은 오픈소스(따라서 공개적으로 접할 수 있는) 병렬 소프트웨어 프로젝트들이 존재합니다. 이 책은 주로 리눅스 커널의 경우를 소개할 겁니다만 유저 레벨 어플리케이션에서도 적용 가능한 것들을 많이 다룰 겁니다.

넷째로, 1980년대와 1990년대의 거대 규모의 병렬 프로그래밍 개발 프로젝트들은 모두 독점 프로젝트였던 합니다만, 커뮤니티에 상용 쿠리티의 병렬 코드를 개발하는데 필요한 엔지니어링 수련법을 이해하고 있는 개발자 요원들을 심어주었습니다. 이 책의 주요 목적은 이런 엔지니어링 수련법을 소개하는 것입니다.

불행히도, 다섯번째 문제인 연산에 비해 비싼 커뮤니케이션의 비용은 여전히 많이 남아있습니다. 이 문제는 2000년대 들어 많은 주목을 받았지만, Stephen Hawking에 따르자면 빛과 원자의 속도의 한계로 인해 이 분야의 발전은 어려울 것으로 보입니다 [Gar07, Moo03]. 다행히도, 이 문제는 1980년대부터 존재했습니다. 덕분에 앞서 이야기한 엔지니어링 훈련법은 실용적이고 효과적인 방법으로 진화되었습니다. 또한, 하드웨어 설계자들은 점점 더 이 문제에 주목하고 있습니다. 그러나 아마도 미래의 하드웨어는 Section 3.3에서 이야기한 것처럼 보다 병렬 소프트웨어에 친화적인 형태가 될 것입니다.

Quick Quiz 2.1: 여봐요!!! 병렬 프로그래밍은 수십 년간 엄청나게 어렵다고 알려졌다구요. 근데 당신은 그게 그렇게 어렵지 않다고 슬쩍 이야기하는 것 같네요. 뭔 개수작이요?

하지만, 병렬 프로그래밍이 흔히 이야기하는 것보다 애 어렵지 않다고는 해도, 일반적으로는 시퀀셜 프로그래밍보다는 어려운 경우가 많습니다.

Quick Quiz 2.2: 어떻게 병렬 프로그래밍이 시퀀셜 프로그래밍만큼 쉬운게 가능한가요?

그러니 병렬 프로그래밍의 대안을 찾아보는 것도 말이 됩니다. 하지만, 병렬 프로그래밍의 목표를 이해하지 않은채 병렬 프로그래밍의 대안을 찾아본다는 것은 말이 안됩니다. 이 주제는 다음 섹션에서 다룹니다.

2.2 Parallel Programming Goals

시퀀셜 프로그래밍을 넘어서 병렬 프로그래밍이 이루고자 하는 세개의 주된 목표는 다음과 같습니다:

1. 성능(Performance).
2. 생산성(Productivity).
3. 일반성(Generality).

불행히도, 현재로썬 어떤 병렬 프로그램도 이중 두개의 목표까지만 달성 가능합니다. 그러니까, 말하자면 이 세개의 목표는 구부릴 수 없는, 병렬 프로그래밍의 쇠로 만들어진 삼각형을 구성하는 세개의 꼭지점인 거죠.

Quick Quiz 2.3: 헐, 진짜요? 정확성, 관리성, 내구성이 같은 것들은요?

Quick Quiz 2.4: 그리고 정확성, 관리성, 내구성이 해당되지 않는데 왜 생산성과 Generality는 해당되는 거죠?

Quick Quiz 2.5: 병렬 프로그램은 정확성을 증명하기가 어렵다고 알고 있는데, 정말 정확성도 그 목록에 올라갈 수 없는 건가요?

Quick Quiz 2.6: 그냥 재미를 목표로 하는건 어떤가요?

이 목표들은 다음 섹션에서 각각 자세히 설명됩니다.

2.2.1 Performance

성능이야말로 병렬 프로그래밍에서의 최우선 목표입니다. 만약 성능이 고려대상이 아니라면, 당신 자신을 위해 그냥 시퀀셜하게 코드를 짜고 행복해지는게 나을 겁니다. 그렇게 하면 훨씬 쉽고 빠르게 일을 끝낼 수 있을 겁니다.

Quick Quiz 2.7: 성능 이외의 이유로 병렬 프로그래밍을 하는 경우도 있나요?

참고로, 여기서 “성능” 이란 용어는 확장성 (CPU 당 성능)과 효율성 (예를 들어, watt 당 성능)를 포함해 넓은 범주를 포함합니다.

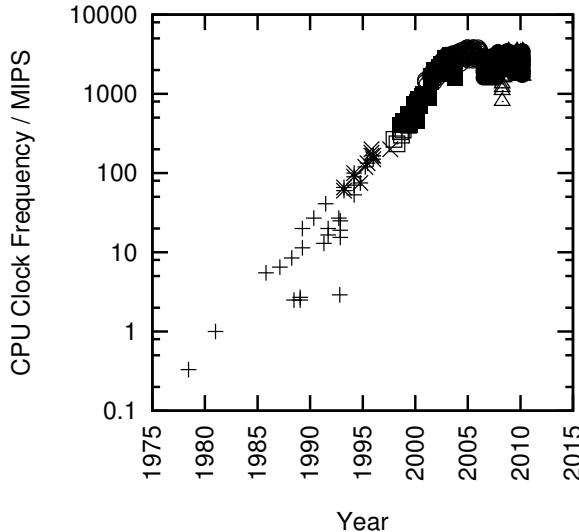


Figure 2.1: MIPS/Clock-Frequency Trend for Intel CPUs

그건 그렇고, 성능의 포커스는 하드웨어에서 병렬 소프트웨어로 옮겨졌습니다. 이는 무어의 법칙은 여전히 트랜지스터 밀집도를 높이고 있지만, 이를 전통적인 싱글쓰레드 성능 향상에 사용하는 것을 그만뒀기 때문이지요. Figure 2.1¹은 싱글쓰레드로 도는 코드를 작성하고 단순히 성능 목표를 만족할 더 빠른 컴퓨터가 나오길 1, 2년 정도 기다리는건 더이상 선택지가 아니란 걸 보여줍니다. 모든 주요 제조사가 멀티코어/멀티쓰레드 시스템으로 개발 방향을 바꾼 이상, 병렬성만이 그런 시스템의 최대 성능을 끌어낼 수 있는 방법입니다.

하지만, 첫번째 목표는 성능이지 확장성이 아닙니다. 선형적인 확장성을 얻을 수 있는 가장 쉬운 방법이 각 CPU의 성능을 떨어뜨리는 것 [Tor01]이기도 하고 말이죠. 네개 CPU를 갖는 시스템을 갖게 되었다면, 당신은 뭘 고르시겠어요? 한개 CPU 위에서 초당 100개 트랜잭션을 처리하지만 CPU 확장성이 아예 없는 프로그램? 아니면 한개 CPU 위에선 초당 10개 트랜잭션만 처리하지만 완벽하게 CPU를 늘려줌에 따라 성능이 늘어나는 프로그램? 아마 첫번째 프로그램이 더 나은 선택 같아 보이겠지만, 당신에게 32개 CPU 시스템이 생길 수 있

¹ 이 그림은 이론상으로 한 클락에 한개 이상의 명령어를 처리할 수 있는 최신 CPU에서는 클락 주파수를, 그리고 가장 간단한 명령어의 처리에도 여러 클락이 필요한 구형 CPU에서는 MIPS(오래된 Dhrystone 벤치마크에서 주로 사용하던, millions of instructions per second)를 보여줍니다. 이렇게 서로 다른 두개의 측정 결과를 보여주는 이유는, 최신 CPU의 한 클락에 여러 명령어를 처리할 수 있는 기능이 일반적으로는 메모리쪽 성능에 의해 제한되기 때문입니다. 뿐만 아니라, 구형 CPU에서 일반적으로 사용되던 벤치마크는 너무 구식이고, 최신의 벤치마크들을 구형 CPU에서 돌리는건 구형 CPU를 구하기도 어려워서 불가능에 가깝기 때문입니다.

다고 하면 또 답이 달라질겁니다.

그렇다면 하지만, 단순히 멀티 CPU 머신이 있기 때문이란 게 그 모든 CPU를 다 사용해야 한다는 이유가 되진 않습니다. 특히나 요즘은 멀티 CPU 시스템이 매우 싼 가격이기도 하니까요. 이해해야 하는 핵심 포인트는 병렬 프로그래밍은 기본적으로 성능 최적화를 위한 여러 방법 중 하나란 거죠. 당신의 프로그램이 지금도 충분히 빠르다면 프로그램을 병렬화 시키거나 병렬화 이외의 다른 방법들을 통해 최적화를 할 필요 자체도 없습니다.² 같은 이유로, 당신이 최적화를 위해 시퀀셜 프로그램에 병렬성을 도입하려 한다면, 병렬 알고리즘들을 최선의 시퀀셜 알고리즘과 비교해야 합니다. 많은 병렬 알고리즘 성능 분석 문서들이 시퀀셜 알고리즘의 케이스를 아예 무시하고 있기 때문에 주의를 해야겠지만요.

2.2.2 Productivity

Quick Quiz 2.8: 왜 이런 비기술적인 문제를 이야기하는거죠??? 그저 비기술적일 뿐 아니라, 심지어 생산성이라니요? 누가 그런걸 신경써요?

■ 최근 수십년간 생산성은 매우 중요해졌습니다. 엔지니어들의 연봉은 수천달러 정도인데 컴퓨터의 가격은 수천만 달러였던 과거를 생각해봅시다. 그 당시의 컴퓨터에 10명의 엔지니어로 구성된 팀을 만들어서 성능을 10% 라도 개선할 수 있다면, 그들은 많은 보너스를 받을 수 있었을겁니다.

CSIRAC은 그런 부류의 머신 중 하나로, 가장 오래되었지만 여전히 온전한, stored-program 컴퓨터로 1949년 [Mus04, Mel06]에 운영되었습니다. 이 머신은 트랜지스터 시대 이전에 만들어졌기 때문에, 2,000 개의 진공관으로 구성되었고 1kHz 클락 주파수로 동작했으며, 30kW의 전력을 사용하고, 그 무게는 3톤이 넘었습니다. 하지만 이 머신은 768 워드밖에 안되는 용량의 RAM을 가지고 있었기에, 오늘날의 거대한 소프트웨어 프로젝트에서 종종 골치를 앓는 생산성 문제에서 자유로웠습니다.

오늘날에 그렇게 성능이 떨어지는 기계를 구입하는건 매우 어렵습니다. 그나마 가장 비슷한 예는 Z80 [Wik08]과 같은 8-bit 임베디드 마이크로프로세서가 될 수 있을 것 같습니다만, 그 Z80 조차도 CSIRAC 보다 1,000 배는 빠른 CPU 클락 주파수를 가지고 있습니다. Z80 CPU는 8,500 개의 트랜지스터를 사용했고, 2008년도에 개당 \$2 US에 1,000개 단위로 구매할

² 물론, 당신이 그저 취미로 병렬 소프트웨어를 만드는 사람이라면 그것만으로도 그 소프트웨어가 뭐건 병렬화를 할 충분한 이유가 되지요.

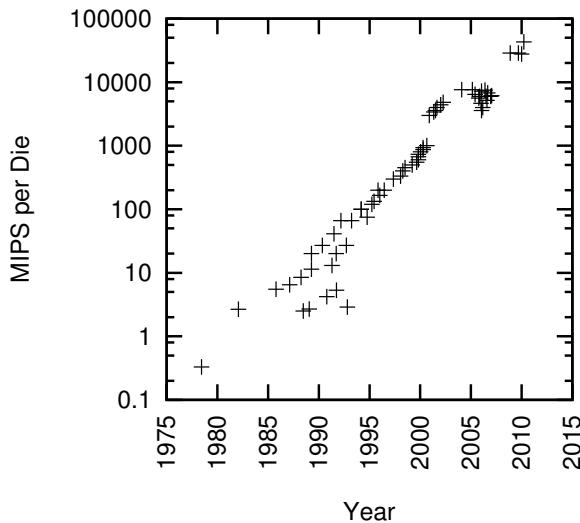


Figure 2.2: MIPS per Die for Intel CPUs

수 있었습니다. CSIRAC 와 반대로 Z80 에서 소프트웨어 개발 비용은 그렇게 크게 중요하지 않았습니다.

오랫동안의 트렌드를 Figure 2.2 를 통해 보면 CSIRAC 와 Z80 은 그 가운데 두개의 점으로 표현될 수 있습니다. 이 그림은 지난 30여년간의 다이당 컴퓨터 시스템 파워 변화를 간략히 보여주는데, 지속적으로 증가한 걸 볼 수 있습니다. 멀티코어 CPU 의 성장은 2003년에 마주한 클락 주파수의 한계 장벽에도 불구하고 이런 증가 추세를 지속시켰습니다.

이렇게 가파른 하드웨어 가격 하락은 소프트웨어 생산성이 점점 중요해짐을 의미합니다. 이제는 그저 하드웨어를 효과적으로 사용하는 것만으로는 충분하지 않습니다: 이젠 소프트웨어 개발자들을 가능한한 효과적으로 사용하는 것이 중요합니다. 시퀀셜 하드웨어에서는 예전부터 그런 문제가 있었지만, 병렬 하드웨어는 최근에서야 낮은 가격으로 상품이 나오기 시작했습니다. 따라서, 병렬 소프트웨어를 만들 때의 생산성은 최근에 들어서야 매우 중요해졌습니다.

Quick Quiz 2.9: 병렬 시스템이 그렇게 싼 가격이 되었다면, 어떤 사람이 그걸 프로그램 하라고 월급을 줘 가며 프로그래머를 고용하겠어요? ■

적어도 한때에는 병렬 소프트웨어의 유일한 목표가 성능이었습니다. 하지만, 지금은 생산성이 점점 스포트라이트를 받고 있습니다.

2.2.3 Generality

병렬 소프트웨어를 개발하는 높은 비용을 정당화 할 수 있는 한가지 방법은 최대한의 generality 를 위해 노력

하는 것입니다. 보다 일반적인 소프트웨어 제품의 개발 비용은 덜 일반적인 것에 비해 더 많은 사용자들로 나뉘어져 보상되기 때문입니다. 실제로, 이런 경제적 이유가 generality 의 중요한 특별 케이스라 할 수 있는 이식성에 대한 마니악한 관심을 설명합니다.³

불행히도, generality 는 종종 성능이나 생산성, 또는 둘 다를 하락시키기도 합니다. 예를 들어, 이식성은 종종 중간 레이어를 두는 식으로 만족되는데, 중간 레이어가 추가되는 것은 분명한 성능 하락을 야기합니다. 보다 일반적인 환경에서의 이 현상을 보기 위해, 다음의 널리 쓰이는 병렬 프로그래밍 환경들을 생각해봅시다:

C/C++ “락킹과 쓰레드” : POSIX 쓰레드 (pthreads) [Ope97], Windows 쓰레드, 그리고 많은 운영체계 커널 환경을 포함하는 이 카테고리는 (최소 하나의 SMP 시스템에서는) 엄청난 성능과 좋은 generality 를 제공합니다. 비교적 낮은 생산성은 아쉽지만요.

Java : 이 범용적이고 본질적으로 멀티쓰레드를 고려한 프로그래밍 환경은 자동 가비지 컬렉터와 당야한 클래스 라이브러리들로 인해 C 나 C++ 보다 훨씬 높은 생산성을 제공하는 것으로 널리 알려졌습니다. 하지만, 그 성능은 2000년대 초에 엄청 개선되긴 했지만 여전히 C 와 C++ 보다 부족합니다.

MPI : 이 메세지 패싱 인터페이스 [MPI08] 는 세계에서 가장 큰 과학 분야와 기술 분야 컴퓨팅 클러스터들을 뒷받침하며 병렬화 되지 않은 성능과 확장성을 제공합니다. 그러나, 이론적으로는 범용적으로 사용 가능하다고 하지만 대부분의 경우 과학 분야와 기술 분야 컴퓨팅에서 사용됩니다. 그 생산성은 심지어 C/C++ “락킹과 쓰레드” 환경보다도 떨어진다고 많은 사람들이 생각합니다.

OpenMP : 이 컴파일러 지시어 집합은 루프문을 병렬화 하는데 사용될 수 있습니다. 때문에 이 특정한 작업에 한정적이고, 이런 제약이 종종 그 성능을 제한합니다. 하지만, MPI 나 C/C++ “락킹과 쓰레드” 보다는 사용하기 쉬운 편입니다.

SQL : SQL (Structured Query Language [Int92] 는 관계형 데이터베이스 쿼리에 제한적입니다. 하지만, 그 성능은 트랜잭션 처리 성능 의회 (TPC) 벤치마크 결과들 [Tra01]로 볼 때 상당히 좋습니다. 생산성도 대단합니다; 실제로, 이 병렬 프로그래밍 환경은 병렬 프로그래밍 컨셉을 잘 모르거나 아예 모르는 사람들도 커다란 병렬 시스템을 잘 사용할 수 있게 돕습니다.

³ 이걸 지적해준 Michael Wong 에게 찬사를.

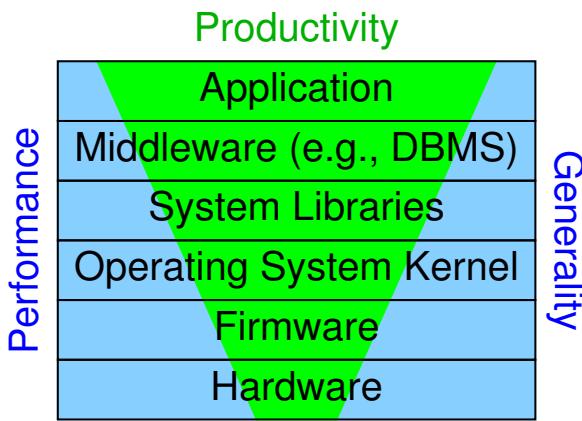


Figure 2.3: Software Layers and Performance, Productivity, and Generality

훌륭한 성능, 생산성, 그리고 generality 를 제공하는 병렬 프로그래밍 환경의 천국은 아직 존재하지 않습니다. 그런 천국이 오기 전까지는, 성능, 생산성, 그리고 generality 사이에서 엔지니어링적 트레이드오프를 가질 수밖에 없습니다. 그런 트레이드오프의 한 예가 Figure 2.3 에 있습니다; 이 그림은 시스템 스택의 상위 레이어로 갈수록 생산성은 점점 중요해지고 성능과 generality 는 하위 레이어로 갈수록 중요해진다는 점을 보여줍니다. 하위 레이어에서 발생하는 거대한 개발 비용은 똑같이 거대한 사용자 수로 나뉘어져야 하고(그래서 generality 가 중요하죠), 하위 레이어에서 잃은 성능은 위쪽에서 쉽게 회복시킬 수 없습니다. 스택의 윗단에서는 해당 특정 어플리케이션에 대해 매우 적은 사용자만 존재할겁니다. 생산성 고려가 중요해지죠. 이게 스택의 위로 갈수록 “bloatware”가 되는 경향을 설명합니다: 많은 경우 여분의 하드웨어가 여분의 개발자보다 싸게 먹히니까요. 이 책은 성능과 generality 가 주요 관심사인, 스택의 바닥 근처에 있는 개발자를 대상으로 합니다.

생산성과 generality 사이에서의 트레이드오프가 많은 영역에서 100여년간 존재했음을 알아둘 필요가 있습니다. 하나만 예를 들자면, 못을 박는데 망치보다 레일건이 생산적이지만 망치는 못박는 거 말고도 많은 영역에 사용될 수 있죠. 그러니 병렬 컴퓨팅에서도 비슷한 트레이드오프들을 발견하는건 그다지 놀라운 일 아니입니다. 이런 트레이드오프는 추상적으로 Figure 2.4 에 나타나 있습니다. 여기서 user 1, 2, 3, 그리고 4 는 컴퓨터가 도와야 하는 그들만의 특정한 작업을 가지고 있습니다. 특정 사용자에게 가장 생산적인 언어 또는 환경은 어떤 프로그래밍이나 환경 설정, 환경 구축을 할 필요 없이 간단하게 그 사용자의 작업을 해내는 언어 또는 환경입니다.

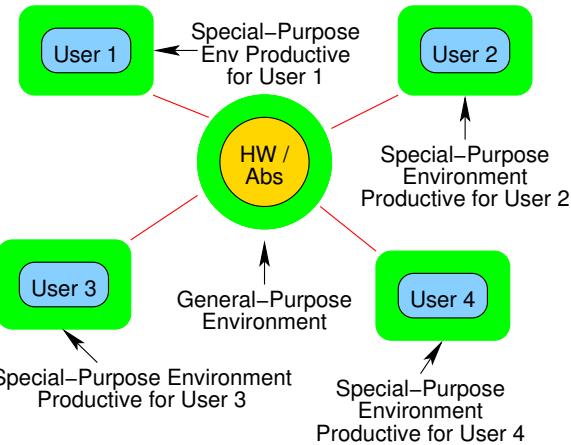


Figure 2.4: Tradeoff Between Productivity and Generality

Quick Quiz 2.10: 이건 달성 불가한 이상에 불과해요! 현실적으로 달성 가능한 무언가에 집중하는게 어때요?

■

불행히도, user 1 에 의해 요청된 작업을 수행하는 시스템은 user 2 의 작업을 처리하지 못하는 경우가 많습니다. 가장 생산성 좋은 언어와 환경은 영역이 한정되어 있어서, generality 가 부족할 수 있다는 것이죠.

다른 선택지는 Figure 2.4 의 가운데 영역처럼 프로그래밍 언어나 환경을 주어진 (assembly, C, C++, 또는 Java 같은 로우 레벨 언어들처럼) 하드웨어나 (Haskell, Prolog, 또는 Snobol 처럼) 추상화에 맞추는 것입니다. 이런 언어들은 user 1, 2, 3, 4 에 의해 요청된 작업들에 대해 모두 똑같이 맞춰져 있지 않다는 점에서 일반적이라고 할 수 있습니다. 다시 말해, 그런 언어들의 generality 는 해당 영역에 한정된 언어와 환경에 비해 생산성을 떨어뜨려야만 얻을 수 있다는 거죠. 뿐만 아니라, 특정 추상화에 맞춰진 언어는 누군가가 그 추상화와 실제 하드웨어 사이의 매핑을 효과적으로 하기 전까지는 성능과 확장성 문제를 심하게 겪을 확률이 높습니다.

성능, 생산성, generality 라는 쇠로된 삼각형의 세 가지 목표에서의 탈출은 불가능한 걸까요?

실은 종종 탈출구가 있습니다. 예를 들어, 다음 섹션에서 이야기할 병렬 프로그래밍의 대안책을 사용하는 것이죠. 병렬 프로그래밍은 엄청난 재미를 가져다 줄 수는 있지만, 항상 최고의 선택인 건 아닙니다.

2.3 Alternatives to Parallel Programming

병렬 프로그래밍의 대안을 제대로 고려해보려면, 당신은 당신이 병렬성에 기대하는게 뭔지 정확히 정해야 합니다. Section 2.2에서 본것과 같이, 병렬 프로그래밍의 주요 목표는 성능, 생산성, 그리고 generality입니다. 이 책은 소프트웨어 스택의 바닥 근처에서 성능에 큰 영향을 끼치는 코드를 만지는 개발자들을 대상으로 하고 있기 때문에, 이 섹션의 나머지 부분은 성능 개선 쪽에 포커스를 맞춥니다.

병렬성은 성능을 개선하기 위한 방법 중 하나에 불과하단 것을 명심해야 합니다. 병렬성 외의 잘 알려진 방법을 적당히 덜 어려운 순서에서 더 어려운 순서로 나열해 보자면 다음과 같습니다:

1. 시퀀셜 어플리케이션의 여러 인스턴스를 실행하는 것.
2. 어플리케이션이 이미 존재하는 병렬 소프트웨어를 사용하게 하는 것.
3. 성능 개선책을 시리얼 어플리케이션에 적용하는 것.

이 방법들 각각을 다음의 섹션들에서 설명합니다.

2.3.1 Multiple Instances of a Sequential Application

시퀀셜 어플리케이션의 인스턴스를 여러개 실행시키는 것으로 당신은 실제로는 병렬 프로그래밍을 하지 않으지만 병렬 프로그래밍을 한것과 같은 효과를 얻을 수 있습니다. 이런 접근법에는 어플리케이션의 구조에 따라 여러 방법이 있습니다.

당신의 프로그램이 매우 많은, 서로 다른 시나리오를 분석하거나 매우 많은 독립적 데이터 셋을 분석하는 것이라면 쉽고 효과적인 방법 하나는 하나의 분석을 수행하는 하나의 시퀀셜 프로그램을 만들고 (예를 들면 bash 셸 같은) 스크립트 환경을 사용해 그 시퀀셜 프로그램의 여러 인스턴스를 병렬적으로 수행시키는 것입니다. 경우에 따라서는 이런 접근법은 여러 머신으로 구성된 클러스터로도 쉽게 확장될 수 있습니다.

이런 접근법은 사기처럼 보일 수도 있겠죠, 그리고 어떤 사람들은 그런 프로그램들을 “당황 스러울 정도로 병렬적이다”라고 펌하합니다. 그리고 실제로, 이런 접근법은 증가된 메모리 소모, 같은 중간 결과를 다시 계산함으로써 발생하는 CPU 사이클의 낭비, 그리고 증가된 데이터 복사와 같은 잠재적 단점을 갖습니다. 하지만, 이 접근은 종종 매우 생산적이어서 엄청난 생산성

증가를 매우 적은, 또는 아예 없는 추가 노력으로도 달성 가능하게 해줍니다.

2.3.2 Use Existing Parallel Software

관계형 데이터베이스 [Dat82], 웹 어플리케이션 서버, 그리고 맵 리듀스 환경 등과 같이 싱글 쓰레드 프로그래밍으로 동작시킬 수 있는 병렬 소프트웨어 환경이 오늘날에는 결코 적지 않습니다. 예를 들어, 혼란 구조 중 하나는 각자 SQL 프로그램을 생성하는 별개의 프로그램을 각 유저에게 제공하는 것입니다(여주: SQL 클라이언트 같은거죠). 이렇게 유저별로 배포된 SQL 프로그램은, 자동으로 사용자들의 쿼리를 동시에 처리하는 일반적인 관계형 데이터베이스와 상호 동작합니다. 사용자에게 배포된 프로그램은 단지 사용자 인터페이스만 책임지면 되고, 병렬성과 데이터 정합성 등의 어려운 일은 모두 관계형 데이터베이스의 책임입니다.

또한, 특히 수학 계산 쪽에서, 병렬 라이브러리 함수들도 많이 생겨나고 있습니다. 심지어 일부 라이브러리는 벡터 연산 유닛들과 범용 그래픽 처리 유닛 (GPGPU)과 같은, 특정 목적으로 만들어진 하드웨어의 특성을 활용하기도 합니다.

매우 신경써서 조심스럽게 만들어진 완전히 병렬적인 어플리케이션에 비교했을 때엔 이런 접근법은 종종 성능을 희생합니다. 하지만, 그정도 희생은 대부분의 경우 커다란 개발 비용의 축소로 보상됩니다.

Quick Quiz 2.11: 잠깐만요! 이런 접근법은 단순히 개발을 위한 노력을 당신으로부터 누군가 그 존재한다는 병렬 소프트웨어를 만드는 사람에게 전가할 뿐인 거 아닌가요? ■

2.3.3 Performance Optimization

2000년대 초까지, CPU 성능은 매 18개월마다 두배씩 빨라졌습니다. 그런 환경에서는 조심스럽게 성능을 개선하는 것보다 새로운 기능을 추가하는 것이 대부분의 경우 중요합니다. 무어의 법칙은 트랜지스터 집적도와 트랜지스터당 성능을 높이는 게 아니라, “단지” 트랜지스터 집적도만을 올리기 때문에 지금은 성능 최적화의 중요성에 대해 다시 생각해볼 좋은 기회입니다. 무엇보다, 새로운 하드웨어들은 더이상 대단한 싱글 쓰레드 성능 향상을 가져오지 않습니다. 더욱기, 많은 성능 최적화는 에너지 소모를 줄입니다.

이런 관점에서, 병렬 시스템이 점점 싸고 시장에 많이 나올수록 더욱더 매력적인 성능 개선 방법이 되고 있습니다. 하지만, 병렬성을 활용해 얻을 수 있는 성능 향상 정도는 대략적으로 보면, CPU 수에 제한됨을 기억하고 있는게 현명할 것입니다. 반면, 전통적인 싱글 쓰레드 소프트웨어 최적화에서의 최적화 기법을 통한 성능 향상은 그보다 훨씬 클 수 있습니다. 예를 들어, 긴 링크드

리스트를 해쉬 테이블이나 서치 트리로 교체하는 것은 수십수백배 성능을 향상시킬 수 있습니다. 이 고도로 최적화된 싱글쓰레드 프로그램은 최적화 되지 않은 병렬 프로그램에 비해 훨씬 빠를 것이고, 병렬성은 필요치 않을 것입니다. 물론, 고도로 최적화된 병렬 프로그램은 추가적인 개발과정의 노력이 필요하겠지만 최적화된 싱글쓰레드 프로그램보다도 나은 성능을 보일테죠.

더욱이, 서로 다른 프로그램들은 서로 다른 성능 병목 지점을 가질 겁니다. 예를 들어, 당신의 프로그램이 대부분의 시간을 디스크 드라이브로부터 읽어들이는 데이터가 도착하기를 기다리고 있다면, 여러 CPU를 사용하는 것은 그저 디스크로부터 데이터를 기다리는 시간을 증가시키기만 할수도 있습니다. 실제로, 프로그램이 하드디스크처럼 회전하는 디스크에 시퀀셜하게 써여져 있는 하나의 큰 파일을 읽으려 할 때, 그 프로그램을 병렬화 하면 추가된 탐색 오버헤드 때문에 프로그램이 더 느려질 수 있습니다. 그보다는 데이터 배치를 그 파일이 더 작아질 수 있도록 (그래서 더 빨리 읽어들일 수 있도록) 하고, 그 파일을 병렬적으로 다른 드라이브에서 접근할 수 있도록, 여러 조각으로 나누거나, 자주 접근되는 데이터를 메인 메모리에 캐시해 놓거나, 만약 가능하다면, 읽어야 하는 데이터의 양 자체를 줄여야 합니다.

Quick Quiz 2.12: 어떤 다른 병목지점들이 CPU를 추가해도 성능을 개선되지 않게 할 수 있을까요?

■ 병렬성은 효과적인 최적화 기술이 될 수 있지만, 유일한 것도 아니고 모든 상황에 맞는 것도 아닙니다. 물론, 당신의 프로그램을 병렬화 하기가 쉬울수록 최적화 방법으로 병렬화가 더 매력적인 선택일 겁니다. 병렬화는 매우 어렵다는 평판을 얻어왔고, 이로 인해 “정확히 뭐가 병렬 프로그래밍을 그렇게 어렵게 하지?”라는 질문이 나옵니다.

2.4 What Makes Parallel Programming Hard?

병렬 프로그래밍의 어려움은 병렬 프로그래밍 문제의 기술적 요소의 집합이기에 인간 공학에서의 이슈만큼이나 어렵다는 것을 알아두는 것이 중요합니다. 우리는 인간이 병렬 시스템에게 뭘 해야하는지 말하는 법, 다른 말로 하자면 프로그래밍을, 알아야 할 필요가 있습니다. 하지만 병렬 프로그래밍은 프로그램의 성능과 확장성이라는, 머신으로부터 인간으로의 커뮤니케이션을 포함한 두갈래의 커뮤니케이션과 관련되어 있습니다. 짧게 말해서, 사람은 컴퓨터가 뭘 해야하는지 말하는 프로그램을 쓰고, 컴퓨터는 이 프로그램을 그 성능과 확장성으로 비평합니다. 따라서, 추상화나 수학적 분석으로의

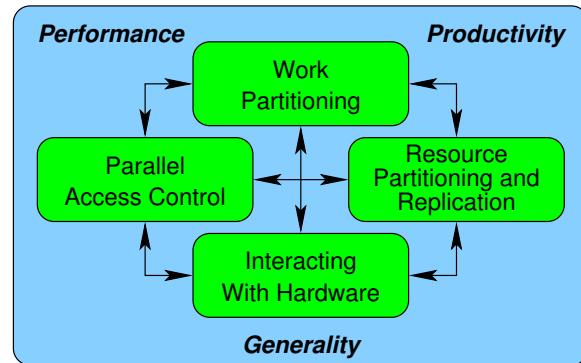


Figure 2.5: Categories of Tasks Required of Parallel Programmers

접근은 종종 매우 제한된 쓰임새만을 보일 겁니다.

산업혁명 때, 인간과 기계 사이의 인터페이스는 인간 공학 연구에 의해 평가되었고, 그때는 시간-과-움직임 연구라 불렸습니다. 비록 그때도 병렬 프로그래밍을 조사하는 인간공학 연구 [ENS05, ES05, HCS⁺05, SS94]도 있었지만, 이런 연구는 매우 좁게 포커스 되어 있었고, 따라서 일반적인 결과를 내놓지는 못했습니다. 더욱이, 일반적인 프로그래머 생산성의 범위가 10배 이상도 차이나는 점을 생각하면, 생산성의 (대략) 10% 차이를 알 수 있는 적당한 비용의 연구를 원하는 건 비현실적입니다. 그런 연구가 신빙성 있게 파악할 수 있는 수백 수천배 차이들도 매우 가치 있지만, 대부분의 의미있는 개선들은 10% 개선의 연장선입니다.

따라서 우리는 다른 접근법을 가져야 합니다.

하나의 그런 접근법은 시퀀셜 프로그래머와 달리 병렬 프로그래머가 반드시 해야 하는 작업을 신중히 고려해 보는 것입니다. 그러면 우린 주어진 프로그래밍 언어나 환경이 개발자가 그 작업을 처리하는데 도움을 주는지 평가할 수 있습니다. 이런 작업들은 Figure 2.5에 보여진, 각각 다음의 섹션들에서 다루어질 네개의 카테고리로 나누어집니다.

2.4.1 Work Partitioning

워크 파티셔닝은 병렬 수행을 위해 반드시 필요합니다: 만약 오로지 하나의 일 “덩어리”만 있다면, 한번에 한개의 CPU에 의해서만 수행될 수 있는데, 이는 곧 순차적 수행이죠. 하지만, 코드를 쪼개는데에는 매우 큰 주의가 필요합니다. 예를 들어, 균등하지 않게 일을 쪼개는 경우 작은 조각들이 수행 완료된 후에는 순차적 수행이 일어납니다 [Amd67]. 덜 극단적인 경우에는 하드웨어를 모두 사용하고 성능과 확장성을 올리기 위해 로드밸런싱이 사용될 수 있습니다.

파티셔닝이 성능과 확장성을 엄청 끌어올릴 수 있긴 하지만, 복잡도도 높일수가 있습니다. 예를 들어, 파티셔닝은 글로벌한 에러와 이벤트의 처리를 복잡하게 만들 수 있습니다: 병렬 프로그램은 그런 글로벌한 이벤트를 안전하게 처리하기 위해 사소하지 않은 동기화를 해야만 할 수 있습니다. 좀 더 일반적으로 말해, 각각의 조각은 커뮤니케이션을 필요로 합니다: 무엇보다, 한 쓰레드가 전혀 커뮤니케이션을 하지 않는다면, 그건 어떤 효과도 일으키지 못할테고 애초에 수행될 필요가 없는 거죠. 하지만, 커뮤니케이션은 오버헤드를 일으키기 때문에, 생각없는 파티셔닝은 커다란 성능 하락을 가져올 수 있습니다.

더구나, 동시에 수행되는 쓰레드들은 종종 CPU 캐시 공간과 같은 공유 자원을 모조리 사용해 벼릴 수 있기 때문에 그 갯수는 자주 조정되어야 합니다. 너무 많은 쓰레드들이 동시에 수행되도록 된다면 CPU 캐시는 넘쳐날거고, 이는 높은 캐시 미스 레이트를 가져와 성능을 떨어뜨릴 겁니다. 반대로, I/O 기기들을 모두 활용하기 위해서는 컴퓨테이션과 I/O 를 겹치게 하기 위해 많은 수의 쓰레드가 필요합니다.

Quick Quiz 2.13: CPU 캐시 용량 외에, 뭐가 동시에 수행되는 쓰레드들의 갯수를 제한해야 하게 할 수 있을까요? ■

마지막으로, 쓰레드들을 동시에 수행되도록 두는 것은 프로그램의 상태 공간을 엄청나게 증가시켜서 프로그램을 이해하고 디버깅하기 어렵게 만들어 생산성을 떨어뜨립니다. 다른것들은 똑같다 해도, 더 정규적인 구조를 갖는 더 작은 상태 공간은 이해하기 쉽습니다만, 이건 기술적 또는 수학적 표현만큼이나 인간공학적 표현입니다. 나쁜 병렬 설계는 비교적 작은 상태 공간을 갖더라도 이해하기 어렵겠지만 좋은 병렬 설계는 엄청나게 큰 상태 공간을 가지더라도 정규적 구조 덕분에 불필요하게 이해하기 어렵지는 않을 겁니다. 최고의 설계는 당황스러운 병렬성을 노출하거나 문제를 맹황스럽게 병렬적인 해결책으로 바꿔버립니다. 어떤 경우든, “당황스럽도록 병렬적임”은 사실 부자들의 골칫거리입니다. 현존하는 최고의 기술은 좋은 설계들을 나열합니다; 상태공간 크기와 구조에 대해 더 일반적 평가를 하기 위해선 더 많은 일이 필요합니다.

2.4.2 Parallel Access Control

싱글쓰레드로 도는 시퀀셜 프로그램에서는 하나의 쓰레드가 프로그램의 모든 리소스에 접근합니다. 이런 리소스들은 대부분의 경우 메모리 위의 데이터 구조들이지만 CPU들, 메모리 (캐시 포함), I/O 기기, 연산 가속 장치들, 파일, 그리고 그 외에도 다른 것들이 얼마든지 있을 수 있습니다.

첫번째 병렬-접근-제어(parallel-access-control) 문제

는 어떤 리소스에의 접근 형태가 그 리소스의 위치에 의존적인가 하는 것입니다. 예를 들어, 많은 메세지 전달 환경에서는 지역변수 접근은 expression 과 assignment 를 통해 이루어지지만, 원격 변수에의 접근은 일반적으로 메세징과 관련된, 완전 다른 문법을 사용합니다. 메세지 패싱 인터페이스 (MPI) [MPI08] 에서는 원격 데이터에 접근하는데에는 명시적 메세징이 필요하기 때문에 명시적 접근방법을 제공하는 반면, POSIX 쓰레드 환경 [Ope97], Structured Query Language (SQL) [Int92], 그리고 Universal Parallel C (UPC) [EGCD03] 같은 분할된 전체 주소공간(PGAS) 환경에서는 묵시적 접근을 제공합니다.

다른 병렬-접근-제어 문제는 쓰레드들이 리소스로의 접근을 어떻게 상호 순서 등을 맞출 것인가입니다. 이런 순서는 메세지 패싱, 락킹, 트랜잭션, 레퍼런스 카운팅, 명시적 타이밍, 공유된 어토믹 변수, 그리고 데이터 소유권 등을 포함한 다양한 병렬 언어와 환경에서 제공하는 매우 많은 동기화 메커니즘들을 통해 이루어집니다. 많은 전통적 병렬 프로그래밍이 이 순서 문제로 부터 발생하는 데드락, 라이브락, 그리고 트랜잭션 롤백 등에 많은 고려를 합니다. 락킹 vs 트랜잭션 메모리 [MMW07] 와 같은 동기화 메커니즘들의 비교를 포함하도록 할수도 있겠지만, 그런 설명은 이 섹션의 범위를 넘어갑니다. (트랜잭션 메모리에 대한 더 많은 정보를 위해선 Section 17.2 와 17.3 를 보기 바랍니다.)

2.4.3 Resource Partitioning and Replication

대부분의 효과적인 병렬 알고리즘들과 시스템들은 리소스 병렬성을 활용하는데, 대부분의 경우 쓰기가 많이 일어나는 리소스는 분할하고 자주 읽기가 많이 일어나는 리소스는 사본을 만드는 것이 현명합니다. 여기서 말하는 리소스는 대부분의 경우 컴퓨터 시스템 각자, 대용량 저장장치, NUMA 노드, CPU 코어 (또는 다이 또는 하드웨어 쓰레드들), 페이지, 캐시 라인, 동기화 수단의 인스턴스, 또는 코드의 크리티컬 섹션에 분할되곤 하는 데이터입니다. 예를 들어, 락킹으로 분할하는 것을 가리켜 “data locking” [BK85] 라고 합니다.

리소스 분할은 많은 경우 어플리케이션에 종속적입니다. 예를 들어, 수학적 어플리케이션은 종종 행렬을 행, 열, 또는 서브 행렬로 분할하고, 상용 어플리케이션은 많은 경우 쓰기 위주 데이터 구조는 분할시키고 읽기 위주 데이터 구조는 사본을 만듭니다. 즉, 상용 어플리케이션은 주어진 고객을 위한 데이터를 커다란 클러스터의 일부 컴퓨터에 할당할 것입니다. 일부 어플리케이션은 데이터를 정적으로, 또는 수행시간동안 동적으로 분할할 수도 있습니다.

리소스 분할은 매우 효과적입니다만 복잡하게 연결

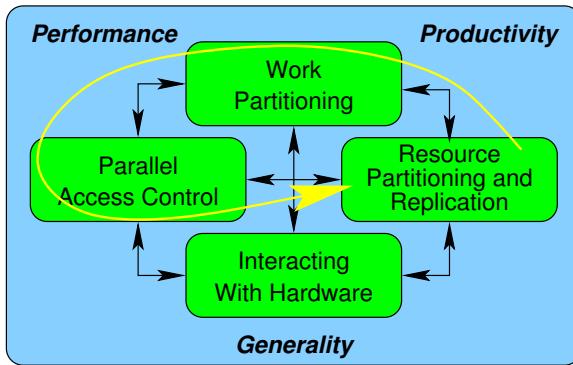


Figure 2.6: Ordering of Parallel-Programming Tasks

되어 있는 데이터 구조에서는 꽤 달성하기 어려운 문제이기도 합니다.

2.4.4 Interacting With Hardware

하드웨어와의 상호작용은 일반적으로는 운영체제, 컴파일러, 라이브러리, 또는 다른 소프트웨어 환경 인프라의 영역입니다. 하지만, 최신 하드웨어 기능과 컴포넌트를 바탕으로 일하는 개발자들 같은 경우는 종종 그런 하드웨어를 직접 다뤄야 하는 경우가 있습니다. 또한, 주어진 시스템에서 마지막 한방울까지 성능을 짜내야 하는 경우에도 하드웨어에 직접 접근해야 하는 경우가 있을 수 있습니다. 이런 경우, 개발자는 어플리케이션을 타겟 하드웨어의 캐시 구조, 시스템 구성, 또는 상호작용 프로토콜에 맞춰줘야 하게 됩니다.

어떤 경우에는, 하드웨어는 앞의 섹션에서 이야기한 것처럼 파티셔닝하거나 접근을 제어해야 하는 리소스로 여겨질 수도 있습니다.

2.4.5 Composite Capabilities

앞서 살펴본 네가지 방법이 기본적인 것들이지만, 좋은 엔지니어링적 실천법은 이 네개의 방법을 조합해서 사용하는 것입니다. 예를 들어, 데이터 병렬 접근법은 Figure 2.6에서 보여지듯이 먼저 데이터를 파티션간 커뮤니케이션이 최소화 되도록 분할하고, 코드 역시 적당하게 분할한 후에, 데이터 조각들과 쓰레드들이 쓰레드 간 커뮤니케이션을 최소화된 채로 처리량을 최대화하도록 매핑합니다. 개발자는 각 조각이 생산성을 증가시킬 수 있도록 개별적으로 관련있는 상태 공간을 얼마나 많이 줄였는지 평가해 볼 수 있습니다. 설령 일부 문제는 분할이 불가능하다 하더라도, 분할 가능한 형태로의 현명한 변환을 통해 가끔 성능과 확장성에 큰 향상을 줄 수 있기도 합니다 [Met99].

2.4.6 How Do Languages and Environments Assist With These Tasks?

많은 환경이 개발자가 이런 일을 직접 해야만 하게 하지만, 상당한 자동화를 제공하는 환경도 오래전부터 있었습니다. 이런 환경의 대표적인 예는 SQL로, 하나의 큰 쿼리를 자동으로 병렬화 시키고 독립적인 쿼리와 업데이트들의 동시수행을 자동화 해주며, 많은 구현체가 존재합니다.

이런 네 카테고리의 작업들은 반드시 모든 병렬 프로그램에서 행해져야만 합니다만 개발자가 하나 하나 손으로 해야만 한다는 의미는 아닙니다. 병렬 시스템들이 계속해서 가격이 인하되고 여러 곳에서 접할 수 있게 되어갈수록 이 네개 작업의 자동화가 늘어날 것입니다.

Quick Quiz 2.14: 병렬 프로그래밍에 다른 어려움은 없나요?

■

2.5 Discussion

이 섹션에서는 병렬 프로그래밍의 어려움과 목표, 그리고 그 대안에 대해 간략히 살펴봤습니다. 이 내용에 있어서 무엇이 병렬 프로그래밍을 어렵게 만드는지와, 그 어려움을 해결하기 위한 하이레벨에서의 접근법에 대해 알아봤습니다. 여전히 병렬 프로그래밍은 손댈 수 없을 정도로 어렵다고 생각하는 분은 병렬 프로그래밍에의 좀 더 오래된 리뷰 [Seq88, Dig89, BK85, Inm85]를 좀 보시기 바랍니다. 특히, Andrew Birrell의 전공논문 [Dig89]에서는 다음과 같이 이야기합니다:

동시성을 가진 프로그램을 짜는 것은 생소하고 어렵다는 인식이 있습니다. 전 생소하지도 어렵지도 않다고 믿습니다. 당신은 좋은 기능과 라이브러리들을 제공하는 시스템이 필요하고, 기본적인 주의와 조심이 필요하며, 유용한 테크닉들과 흔한 함정들을 알아야 합니다. 전 이 논문이 당신이 저의 믿음을 공유하는데 도움이 되었길 바랍니다.

이 오래된 가이드들의 저자들은 1980년대의 병렬 프로그래밍 문제에 잘 대응했습니다. 따라서, 21세기에 와서 병렬 프로그래밍 문제에 도전하는 것을 거절할 명분은 없습니다!

이제 다음 챕터로 넘어가도 좋을 것 같습니다. 다음 챕터에서는 우리의 병렬 소프트웨어 아래에 존재하는 병렬 하드웨어의 관련있는 부분들에 대해 깊이 이야기해 봅니다.

Chapter 3

Hardware and its Habits

대부분의 사람들은 시스템간에 메세지를 주고받는 것은 단일 시스템 안에서 간단한 계산을 하는 것보다 비싸다는 것을 직관적으로 이해하고 있습니다. 하지만, 단일 공유메모리 시스템 안에서 쓰레드간에 커뮤니케이션하는 것도 매우 비쌀 수 있다는 것은 항상 앞의 이야기만큼 분명해 보이진 않습니다. 그래서 이 챕터는 하나의 공유메모리 시스템에서 동기화와 커뮤니케이션의 비용에 대해서 알아봅니다. 이 몇장의 내용은 공유메모리 병렬 하드웨어 설계의 겉면만 훑어 볼 겁니다; 더 깊이 알고 싶은 독자분들은 Hennessy 와 Patterson 의 고전 교과서 [HP95] 의 최신판을 보면 좋을 겁니다.

Quick Quiz 3.1: 왜 병렬 프로그래머가 하드웨어의 로우 레벨 요소들까지 배워야 하죠? 하이 레벨의 추상 계층만 보는게 더 쉽고, 낫고, 더 일반적이지 않겠어요?

■

3.1 Overview

컴퓨터 시스템 스펙 문서를 생각없이 읽으면 CPU 성능이 Figure 3.1 에 그려진 것과 같은, 제일 빠른 사람이 항상 경주에서 이기는, 깨끗한 운동장에서의 도보 경주와 같다고 생각하기 쉽습니다.

Figure 3.1 에 보여진 이상적 상황으로 접근하는 CPU 바운드 벤치마크들도 몇개 있긴 합니다만, 일반적인 프로그램은 경주용 운동장보다는 장애물 코스에 더 가깝습니다. 무어의 법칙에 의해 CPU 의 내부 구조가 지난 수십년간 엄청나게 변해왔기 때문이죠. 이런 변경들을 다음 섹션들에서 설명합니다.

3.1.1 Pipelined CPUs

1980년대 초, 인스트럭션을 가져오고, 디코드하고, 실행하는 대부분의 마이크로 프로세서는 일반적으로 다음 인스트럭션을 가져오기 전에 인스트럭션 하나를 실행 완료하는데 최소 3 클럭 사이클을 소모했습니다. 대조적으로, 1990년대 후반에서 2000년대 초반의 CPU 는

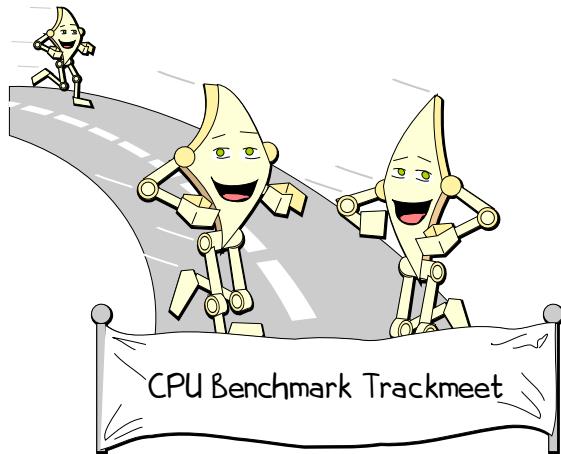


Figure 3.1: CPU Performance at its Best

CPU 로의 인스트럭션 전달 흐름을 내부적으로 조절하는데 깊은 “파이프라인”을 사용해 많은 인스트럭션들을 동시에 수행할 수 있습니다. 이러한 근래의 하드웨어의 기능들은 Figure 3.2 에 보여진 것처럼 성능을 대폭 향상시킬 수 있습니다.

긴 파이프라인을 가지고 CPU 의 최대 성능을 얻기 위해서는 프로그램의 컨트롤 플로우 (코드 실행 흐름)를 매우 정밀하게 예측할 수 있어야 합니다. 큰 행렬이나 벡터 계산과 같이 기본적으로 꽉 짜여진 루프로 구성된 프로그램에서는 적당한 컨트롤 플로우를 얻을 수 있습니다. 그런 경우 CPU 는 루프 마지막의, 루프 마지막의, 루프 처음으로 돌아갈지 루프를 빠져나갈지에 대한 분기 조건이 거의 항상 참일 거라는 것을 올바르게 예측해서 파이프라인이 꽉차 있게 해 CPU가 최대 속도로 돌아갈 수 있게 할겁니다.

하지만, 분기 예측이 항상 그렇게 쉬운건 아닙니다. 예를 들어, 작은 무작위적 횟수만큼 도는 많은 루프로

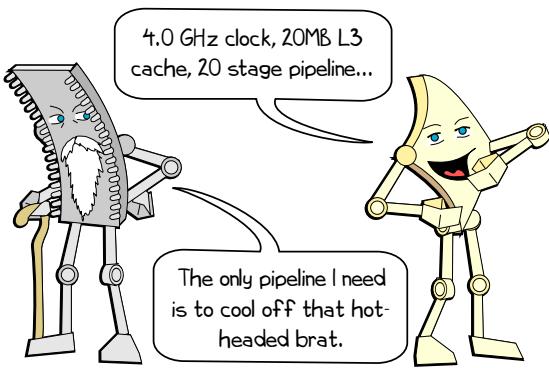


Figure 3.2: CPUs Old and New

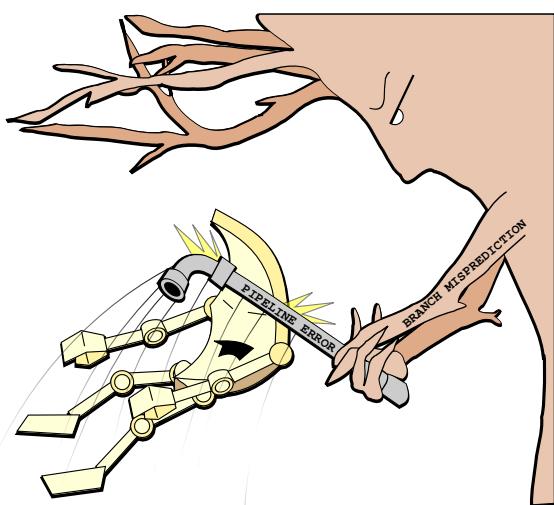


Figure 3.3: CPU Meets a Pipeline Flush

구성된 프로그램을 생각해보세요. 또 다른 예로, 수많은 다른 진짜 객체를 레퍼런스 할 수 있는 가상 객체들이 있는데, 각 진짜 객체들은 자주 호출되는 멤버 함수들을 모두 다르게 구현한 객체지향 프로그램을 생각해보세요. 이런 경우, CPU 가 다음 분기가 실행 흐름을 어디로 이끌지를 예측하는 것 조차도 불가능합니다. 그렇게 되면 CPU 는 그 브랜치가 어디로 실행 흐름을 이끌지가 확실해질 때까지 기다리며 멈춰있거나, 추측이라도 해야 합니다. 예측 가능한 컨트롤 플로우를 갖는 프로그램에서는 추측하는 방법이 매우 잘 동작하지만, (바이너리 서치와 같은) 예측 불가능한 분기문들에 대해서는 추측이 대부분 틀릴 겁니다. 추측이 잘못된 경우 CPU 는 그 분기문을 따르는, 투기적으로 그간 수행된 인스

트럭션들을 모두 폐기시켜야 하기 때문에 파이프라인 플러시를 일으키기 때문에 잘못된 예측은 매우 비싼 비용을 지불합니다. 만약 파이프라인 플러시가 너무 자주 일어난다면, Figure 3.3 에서 그려진 것처럼 엄청난 성능 하락을 겪을 수 있습니다.

불행히도, 파이프라인 플러시는 근래의 CPU 들이 달려야 하는 장애물 코스의 유일한 위협이 아닙니다. 다음 섹션에서는 메모리 참조에 존재하는 위험들을 다룹니다.

3.1.2 Memory References

1980년대에는, 대부분의 경우 마이크로프로세서가 메모리에서 값을 하나 얻어오는데 걸리는 시간이 인스트럭션 하나를 수행하는데 걸리는 시간보다 적었습니다. 2006년에 와서는, 마이크로프로세서는 메모리에 접근 한번 하는데 걸리는 시간 동안 수백, 심한 경우 수천개의 인스트럭션을 수행할 수 있습니다. 이 간극은 Moore 의 법칙이 CPU 성능 향상을 메모리 반응속도의 감소 정도에 비해 너무 크게 이끌었기 때문입니다; 메모리의 발전은 반응속도 감소보다 용량 증가에 치중되었던 것도 한 이유죠. 예를 들어, 1970년대의 평범한 미니컴퓨터는 4KB (네, 메가바이트가 아니라 킬로바이트요. 기가바이트는 입밖에 꺼내지도 마요) 메인 메모리를 가졌고, 그 메모리는 한 사이클만에 접근 가능했습니다.¹ 2008년에도 CPU 설계자들은 여전히 단일 사이클만에 접근 가능한 4KB 메모리를 만들 수 있습니다; 심지어 수 GHz 주파수의 시스템에서도요. 그리고 실제로 그런 메모리를 만듭니다만, 그들은 이제 그걸 “레벨 0 캐시”라 부르고, 그것들은 보통은 4KB 보다는 아주 약간은 크기도 합니다.

근래의 마이크로프로세서에 장착되는 큰 캐시들은 메모리 접근 시간과 맞서 싸우는데 꽤 도움을 줄 수 있습니다만, 이런 캐시들이 그 시간들을 제대로 숨기기 위해서는 고도로 예측 가능한 데이터 접근 패턴이 필요합니다. 불행히도, 링크드 리스트를 순회하는 것과 같은 많은 일들이 예측 불가능한 메모리 접근 패턴을 갖습니다. 무엇보다, 만약 패턴이 예측 가능하다면, 소프트웨어는 포인터 타입이란 것 자체를 만들지도 않았겠죠, 그렇죠? 따라서, Figure 3.4 에서 보여지듯, 메모리 참조는 종종 근래의 CPU 들에게 거대한 장애물입니다.

지금까지는 주어진 CPU 가 싱글 쓰레드 코드를 돌릴 때 만날 수 있는 문제들만 이야기했습니다. 멀티쓰레드 수행은 다음 섹션에서 설명할텐데, 추가적인 문제들을 CPU 에게 내놓습니다.

¹ 물론 그 한개의 사이클은 1.6 마이크로 세컨드 이상이었음을 언급하는게 공정하겠죠.

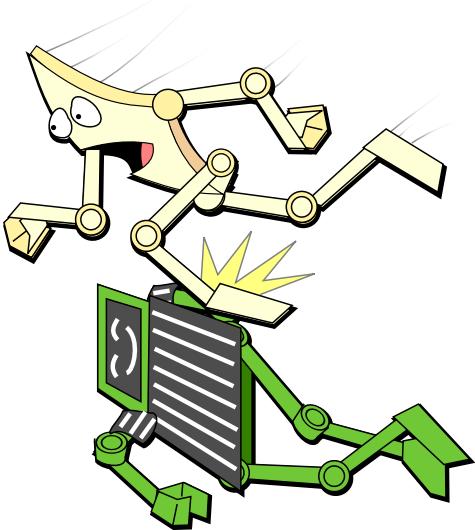


Figure 3.4: CPU Meets a Memory Reference

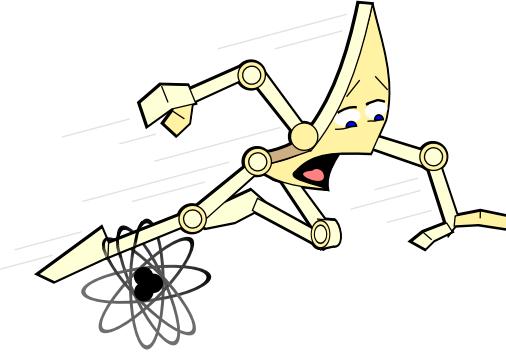


Figure 3.5: CPU Meets an Atomic Operation

위해 기다릴 필요 없이 곧바로 버퍼에 써넣을 수 있습니다. 다행히도, CPU 설계자들은 어토믹 오퍼레이션에 신경을 많이 쏟았고, 덕분에 2014년 초에 이르러서는 그 오버헤드를 상당히 감소시켰습니다. 하지만 그래도, 그 성능에 끼치는 효과는 Figure 3.5에 보이는 대로입니다.

불행히도, 어토믹 오퍼레이션들은 보통 데이터의 한 개 요소에만 적용 가능합니다. 많은 병렬 알고리즘들이 복수개의 데이터 요소들에의 업데이트 간에도 순서가 이루어지길 필요로 하기 때문에, 대부분의 CPU들은 메모리 배리어들을 제공합니다. 이런 메모리 배리어들 역시 성능에의 문제로 존재합니다. 다음 섹션에서 이를 이야기 합니다.

Quick Quiz 3.2: 어떤 기계가 복수 데이터 요소에 대한 어토믹 오퍼레이션을 허용하겠어요?

■

3.1.4 Memory Barriers

메모리 배리어에 대해서는 Section 14.2 와 Appendix C에서 좀 더 깊게 다룰 겁니다. 그 전에, 여기서는 다음의 간단한 락을 사용한 크리티컬 섹션을 생각해 봅시다:

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

만약 CPU에 코드가 보여지는 순서대로 수행되어야 한다는 제약이 존재하지 않는다면, 변수 “a”는 “mylock”의 보호 없이 값이 증가할 것이고, 이렇게 되면 락을 잡는 목표가 이뤄지지 않은 셈입니다. 그런 문제되는 순서 재배치를 막기 위해, 락킹에 사용되는 기본 기능들은 명시적이든 묵시적이든 메모리 배리어를 사용합니다. 이런 메모리 배리어의 목적은 CPU가 성능을 향상시키기

3.1.3 Atomic Operations

그런 장애 중 하나는 어토믹 오퍼레이션들입니다. 여기서의 문제는 모든 어토믹 오퍼레이션은 CPU 파이프라인의 한 순간에는 조각으로 나뉘어 수행되는 어셈블리 오퍼레이션들과 한번씩은 충돌한다는 것입니다. 하드웨어 설계자는 근래의 CPU들은 그런 오퍼레이션들이 실은 여러 조각으로 나뉘어져 여러 순간동안 수행되더라도 원자적으로 수행되는 것처럼 보이게 하기 위해 여러개의 굉장히 현명한 트릭들을 사용한다고 이야기합니다. 그런 트릭 중 흔한 한 가지는 어토믹하게 수행되어야 하는 데이터를 담고 있는 캐시라인들을 모두 파악해두고, 이 캐시라인들은 해당 어토믹 오퍼레이션을 수행하는 CPU에 소유되어 있음을 분명하게 하고, 그러고 나서야만 해당 캐시라인들이 해당 CPU에 의해 여전히 소유되어 있음을 분명히 한 채로 어토믹 오퍼레이션을 수행하는 것입니다. 모든 데이터가 해당 CPU만 접근할 수 있는 상태이기에, 다른 CPU들은 실은 조각으로 나뉘어 수행되는 CPU 파이프라인의 현실에도 불구하고 해당 어토믹 오퍼레이션에 간섭을 끼칠 수 없습니다. 말할 필요도 없겠지만, 이런 종류의 트릭은 그 셋업이 수행되도록 주어진 어토믹 오퍼레이션이 완전히 끝날 때까지 파이프라인이 지연되거나 심지어 플러시될 수도 있습니다.

반대로, 어토믹 오퍼레이션이 아닌 오퍼레이션을 수행할 때에는 CPU는 값을 캐시라인에 올라오는대로 가져올 수 있고, 수행 결과를 캐시라인 소유권을 가지기



Figure 3.6: CPU Meets a Memory Barrier

위해 할 수 있는 코드의 순서 재배치를 막기 위한 것이기 때문에, 메모리 배리어는 거의 항상 Figure 3.6에서 보여지듯 성능을 떨어뜨립니다.

어토믹 오퍼레이션과 같이, CPU 설계자들은 메모리 배리어 오버헤드를 줄이려 열심히 노력해왔고, 꽤 많은 진전을 이뤘습니다.

3.1.5 Cache Misses

또 하나의 멀티 쓰레딩에서의 CPU 성능에의 장애물은 “캐시 미스”입니다. 앞서 말했듯, 근래의 CPU들은 높은 메모리 반응속도로 발생할 수 있는 성능 하락을 줄이기 위해 큰 캐시를 장착하고 있습니다. 하지만, 이런 캐시들은 실은 CPU 간에 자주 공유되는 변수들에 대해서는 생산적이지 못합니다. 하나의 CPU가 한 변수를 수정하려 할 때, 다른 CPU가 그 값을 최근에 바꾼 경우가 있을 가능성이 크기 때문이죠. 이런 경우, 해당 변수는 지금 수정하려는 CPU의 캐시가 아니라 최근에 값을 수정한 CPU의 캐시에 있을 것이고, 이로 인해 비싼 캐시 미스(더 자세한 내용을 위해선 Section C.1를 보세요)를 일으킬 것입니다. 이런 캐시 미스들은 Figure 3.7에서 보여진 것처럼 CPU 성능의 주요 장애가 됩니다.

Quick Quiz 3.3: 그래서, CPU 설계자들은 캐시 미스 오버헤드 역시 많이 개선 했나요? ■

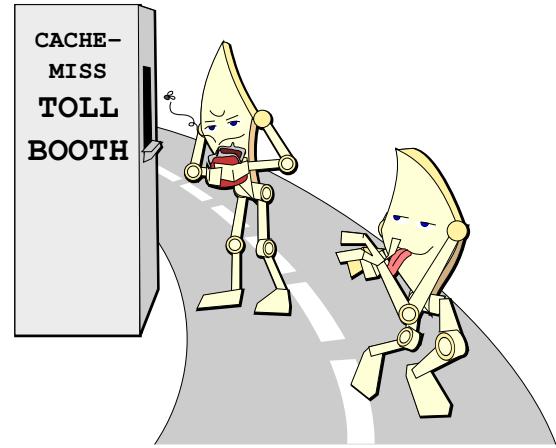


Figure 3.7: CPU Meets a Cache Miss

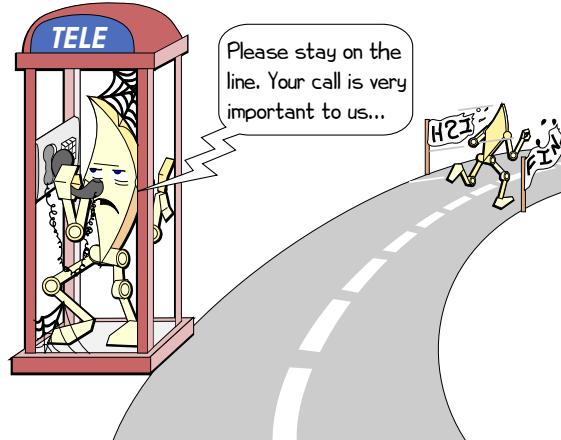


Figure 3.8: CPU Waits for I/O Completion

3.1.6 I/O Operations

캐시 미스는 곧 CPU와 CPU 사이의 I/O 오퍼레이션으로 볼 수 있고, 이것은 가능한 I/O 오퍼레이션 중 가장 비용이 낮은 것들 중 하나입니다. 네트워킹이나 대용량 저장장치, 또는 사람을 포함하는 I/O 오퍼레이션들은 Figure 3.8에서 보여지듯, 앞의 섹션들에서 이야기 되었던 내부적 장애들보다 훨씬 커다란 장애를 야기합니다.

이건 공유 메모리 병렬성과 분산시스템 병렬성 사이의 차이 중 하나입니다: 공유메모리 병렬 프로그램은 일반적으로 캐시 미스보다 더한 장애는 겪지 않습니다만, 분산 병렬 프로그램은 보통 더 큰 네트워크 커뮤니케이

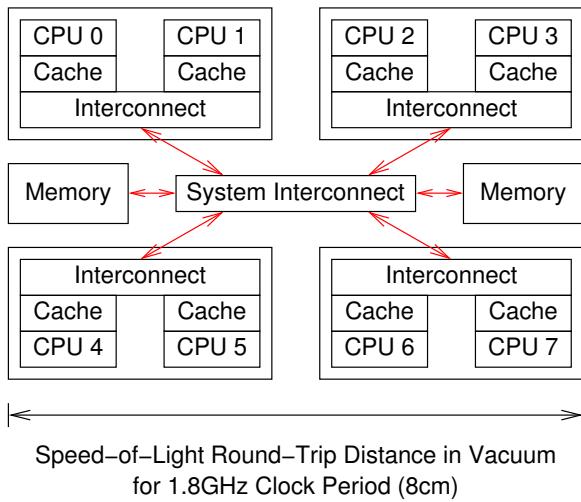


Figure 3.9: System Hardware Architecture

션 시간을 겪습니다. 두 경우 모두, 문제의 응답시간들은 커뮤니케이션의 비용으로 생각될 수 있습니다. 순차 프로그램에서는 존재하지 않는 비용이죠. 따라서, 실제 수행되는 일과 커뮤니케이션 오버헤드 간의 비율이 핵심 설계 결정 요소입니다. 병렬 하드웨어 설계의 주요 목표는 이 비율을 적절한 성능과 확장성 목표를 달성 가능한 수준으로 낮추는 것입니다. 이에 따라서, Chapter 6에서 볼테지만, 병렬 소프트웨어 설계의 중요한 목표는 커뮤니케이션 캐시 미스와 같은 비싼 동작들의 빈번도를 낮추는 것입니다.

물론, 주어진 동작이 장애라는 이야기는 그 이야기이고, 그 동작이 심각한 장애라는 건 또 따로 보여줘야 하지요. 이 차이를 다음의 섹션에서 이야기 합니다.

3.2 Overheads

이 섹션에서는 앞의 섹션에서 나열했던 각 장애들의 실제 성능 오버헤드를 보입니다. 하지만, 그 전에 하드웨어 시스템 아키텍쳐에 대해서 간략히 알아보는게 필요합니다. 다음 섹션에서 이를 다룹니다.

3.2.1 Hardware System Architecture

Figure 3.9 는 8 코어 컴퓨터 시스템의 간략한 구조를 보여줍니다. 각 디아이는 CPU 코어 한쌍을 가지고 있고, 각 코어는 캐시를 갖고, 각 캐시는 쌍을 이루는 코어와 통신을 할 수 있도록 하는 연결이 되어 있습니다. 그럼 중간의 system interconnect는 네개의 디아이가 통신할 수 있도록 하고, 메인 메모리로 각 디아이를 연결합니다.

이 시스템에서 데이터는 하나의 2의승수로 정렬된 메모리의 블락들로, 보통 32에서 256 바이트 사이인 “캐시 라인” 단위로 움직입니다. 한 CPU는 메모리로부터 자신의 레지스터로 변수를 가져오려면, 그 변수를 가지고 있는 캐시 라인을 자신의 캐시로 가져와야 합니다. 비슷하게, 한 CPU는 자신의 레지스터로부터 메모리로 어떤 값을 쓰려 할 때, 일단 그 변수를 담고 있는 캐시 라인을 자신의 캐시로 가져와야 합니다만, 또한 다른 CPU가 그 캐시 라인의 카피를 들고 있지 않음을 보장해야 합니다.

예를 들어, 만약 CPU 0 CPU 7의 캐시에 있는 캐시 라인에 있는 변수에 compare-and-swap (CAS) 오퍼레이션을 하려 하면, 다음의 간략화한 이벤트들이 일어날 겁니다:

1. CPU 0은 자신의 로컬 캐시를 확인하고, 그 캐시 라인이 없음을 발견합니다.
2. 요청은 CPU 0과 CPU 1 사이의 연결로 넘겨져서 CPU 1의 로컬 캐시를 확인해봅니다만, 역시 해당 캐시라인이 없음을 발견합니다.
3. 요청은 시스템 연결부로 넘겨지고, 다른 세 개의 디아들을 체크합니다. 결과, 해당 캐시라인은 CPU 6과 CPU 7이 위치한 디아이에 있음을 확인합니다.
4. 요청은 CPU 6과 CPU 7 연결부로 넘겨져 각 CPU의 캐시를 확인해 해당 캐시 라인이 CPU 7의 캐시에 있음을 발견합니다.
5. CPU 7은 해당 캐시라인을 자신의 연결부로 넘기고, 자신의 캐시에서 해당 캐시라인을 비워버립니다.
6. CPU 6과 CPU 7의 연결부는 해당 캐시 라인을 시스템 연결부로 넘깁니다.
7. 시스템 연결부는 해당 캐시 라인을 CPU 0과 CPU 1 연결부로 넘깁니다.
8. CPU 0과 CPU 1 연결부는 해당 캐시라인을 CPU 0의 캐시로 보냅니다.
9. CPU 0은 이제 자신의 캐시 안에 있는 변수에 CAS 오퍼레이션을 수행할 수 있습니다.

Quick Quiz 3.4: 이제 간략화된 거라구요? 이것보다 더 복잡한게 어떻게 가능하죠? ■

Quick Quiz 3.5: 왜 CPU 7의 캐시에서 해당 캐시라인을 비워야 하죠?

■
이 간략화된 시나리오는 캐시 코히런시 프로토콜 [HP95, CSG99, MHS12, SHW11]의 시작일 뿐입니다.

Operation	Cost (ns)	Ratio (cost/clock)
Clock period	0.6	1.0
Best-case CAS	37.9	63.2
Best-case lock	65.6	109.3
Single cache miss	139.5	232.5
CAS cache miss	306.0	510.0
Comms Fabric	3,000	5,000
Global Comms	130,000,000	216,000,000

Table 3.1: Performance of Synchronization Mechanisms on 4-CPU 1.8GHz AMD Opteron 844 System

3.2.2 Costs of Operations

병렬 프로그램에 중요하고 흔히 사용되는 오퍼레이션의 오버헤드들이 Table 3.1에 표시되어 있습니다. 이 시스템의 클락 시간은 약 0.6ns입니다. 많은 근래의 마이크로프로세서가 한 클락 시간동안 여러 인스트럭션을 수행 완료시킬 수 있긴 합니다만, 모든 오퍼레이션의 비용은 클락 시간을 기준으로 해서 “Ratio”라고 표시한 세번째 행의 값으로 나타낼 수 있습니다. 이 표에 대해서 가장 먼저 알아둬야 할 것은, 큰 비율의 값들입니다.

최선의 경우 compare-and-swap (CAS) 오퍼레이션은 클락 시간보다 60배 이상 큰, 약 40 나노세컨드를 소비합니다. 여기서 “최선의 경우”는 한 번수에 CAS 오퍼레이션을 수행하는 CPU와 해당 변수를 마지막으로 건든 CPU가 동일해서 관련된 캐시 라인이 이미 자신의 캐시 안에 있는 경우입니다. 비슷하게, 최선의 경우 락 오퍼레이션(락 획득과 해제 두개 오퍼레이션의 수행 시간의 합)은 60 나노 세컨드가 넘고 100 클락 사이클이 넘는 시간을 소비합니다. 다시 말하지만, “최선의 경우”는 락을 나타내는 데이터 구조가 이미 락의 획득과 해제를 수행하는 CPU의 캐시 위에 있는 경우입니다. 락 오퍼레이션은 락 데이터 구조에 대한 두번의 어토믹 오퍼레이션을 필요로 하기 때문에 CAS 보다 비쌉니다.

캐시 미스가 난 오퍼레이션은 거의 200 클락 사이클인 140 나노세컨드를 소모합니다. 이 캐시 미스 비용 측정을 위해 사용된 코드는 미스가 난 데이터를 다른 CPU의 캐시에서 얻어옵니다. 즉, 이 캐시 미스 오퍼레이션은 메모리까지 접근하지는 않습니다. 변수의 기준 값을 보는 것은 물론, 새로운 값을 쓰기도 해야 하는 CAS 오퍼레이션은 500 클락 사이클인 300 나노세컨드를 소비합니다. 이걸 조금 생각해 봅시다. CPU는 하나의 CAS 오퍼레이션을 수행하는데 필요한 시간 동안, 500 개의 보통 인스트럭션을 수행할 수도 있었습니다. 이건 fine-grained 락킹만이 아니라 fine-grained 전역적 규칙에 기반한 모든 동기화 메커니즘의 한계를 보여줍니다.

Quick Quiz 3.6: 하드웨어 설계자들은 분명 이 상황

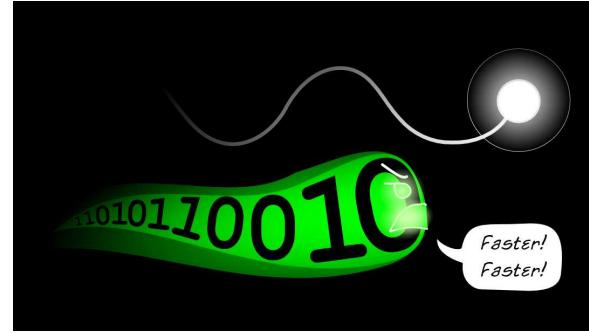


Figure 3.10: Hardware and Software: On Same Side

을 개선하려 노력할 수 있었을 거예요! 왜 그들은 이 단일 인스트럭션 오퍼레이션들의 끔찍한 성능을 만족하고 있는거죠? ■

I/O 오퍼레이션들은 이보다도 더 비쌉니다. “Comms Fabric” 열에서 나타나 있듯이, InfiniBand나 몇몇 독점 연결장치와 같은 고성능의 (그리고 고비용의!) 통신 장치들은 5 천개의 인스트럭션을 수행될 수도 있는, 약 3 마이크로세컨드의 대기시간을 갖습니다. 표준에 기반한 통신 네트워크들은 종종 몇몇 프로토콜 프로세싱을 필요로 하고, 이는 대기시간을 더욱 증가시킵니다. 물론, 지형적 거리도 대기시간을 늘립니다. 이론적으로 빛의 속도라고 해도 지구를 한바퀴 돌기 위해선 약 130 밀리세컨드의 시간을 필요로 합니다. 이는 2억 클락 사이클 이상입니다. 이 수치는 “Global Comms” 열에 있습니다.

Quick Quiz 3.7: 숫자가 미친듯이 크군요! 어떡해야 제 머리로 이걸 이해할 수 있을까요? ■

짧게 말해서, 하드웨어와 소프트웨어 엔지니어들은 사실 Figure 3.10에 우리의 데이터 스트림이 빛의 속도를 넘으려 노력하는 그림으로 나타내는 것처럼, 같은 편에서 컴퓨터들이 물리적 법칙에도 불구하고 더 빨리 동작할 수 있도록 고군분투하고 있습니다. 다음 섹션은 하드웨어 엔지니어들이 할수도 (또는 안할수도) 있는, 가능한 것들에 대해 알아봅시다. 이 싸움에서 소프트웨어가 할 수 있는 공헌은 이 책의 남은 챕터들에서 이야기 합니다.

3.3 Hardware Free Lunch?

동시성이 최근들어 기존보다 주목을 받게 된 것은 페이지 9의 Figure 2.1에 나타난대로 무어의 법칙에 의한 싱글 쓰레드 성능 증가(또는 “공짜 점심” [Sut08])가 멈췄기 때문입니다. 이 섹션에서는 하드웨어 설계자들이 “공짜 점심”을 약간이라도 다시 가져올 수 있는 몇 가지

방법을 간략히 알아봅니다.

하지만, 앞의 섹션에서는 동시성을 노출하는데 생기는 현저한 하드웨어적 문제를 알아봤습니다. 하드웨어 설계자들이 직면하는 강력한 물리적 한계점들 중 하나는 빛의 유한한 속도입니다. 페이지 21의 Figure 3.9에 보여진 것처럼, 빛은 진공에서 1.8 GHz 클락 시간동안 8 센티미터만을 왕복할 수 있습니다. 이 거리는 5 GHz 클락에서는 3 센티미터로 줄어듭니다. 근래 컴퓨터 시스템의 크기에 비교해 보면 둘 다 비교적 작은 거리입니다.

상황이 더 나빠지는게, 실리콘에서의 전자파는 진공에서의 빛에 비해 3-30 배 느리게 움직이고, 일반적인 클락 주파수에 맞춰 움직이는 논리 회로들은 더욱 느리게 동작합니다. 예를 들어, 메모리 접근은 접근 요청이 시스템의 나머지 영역으로 넘어가기 전에 로컬 캐시 검색의 완료를 기다려야 합니다. 더욱이, 예를 들어 CPU와 메인 메모리 사이의 통신의 경우와 같이, 전기 신호를 한 실리콘 다이에서 다른 다이로 넘기기 위해서는 상대적으로 느린 속도와 큰 파워의 드라이버가 필요합니다.

Quick Quiz 3.8: 하지만 개별의 전자들은 컨덕터 내에서조차도 그렇게 빠르지 않아요!!! 세미컨덕터에서 발견된 저전력의 컨덕터 안에서의 전자 이동 속도는 초당 겨우 1 밀리미터 정도라구요. 뭔가요???



(하드웨어에도 소프트웨어에도) 이를 개선할 기술은 더도 말고 덜도 말고 약간만 있습니다:

1. 3D 융합,
2. 훌륭한 물질과 프로세스들,
3. 전자 빛으로 대체하는 것,
4. 특수 목적의 가속기, 그리고
5. 존재하는 병렬 소프트웨어.

이것들 각각을 다음 섹션에서 설명합니다.

3.3.1 3D Integration

3차원 융합 (3DI)은 매우 얇은 실리콘 다이들을 수직으로 쌓아 올려 붙이는 기술입니다. 이 기술은 잠재적 이득을 제공하지만, 또한 대단한 공정적 어려움 [Kni08]을 품고 있습니다.

3DI가 가져올 수 있는 가장 중요한 이점은 Figure 3.11에 그려진대로, 시스템 전체적으로 짧아지는 경로의 길이일 것입니다. 해당 그림에서는 3 센티미터의 실리콘 다이가 제개의 1.5 센티미터 다이들의 더미로 바뀌었고, 각 레이어 사이의 거리가 상당히 얇다는

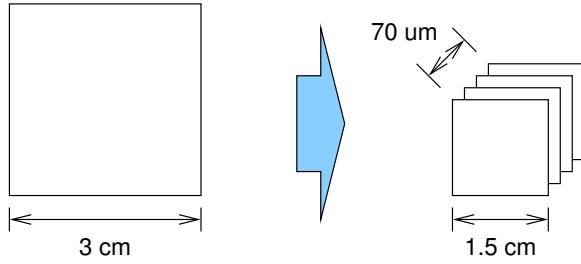


Figure 3.11: Latency Benefit of 3D Integration

것을 고려하면 이론적으로 시스템을 관통하는 최대 경로의 길이가 두배 가까이 줄어든 셈입니다. 또한, 설계와 배치에 충분한 고려를 한다면, (느리고 전력도 많이 소모함) 수평적인 전자적 연결들은 빠르기도 하고 전력 소모도 적은, 짧은 수직적 전자적 연결로 대체될 수 있을 것입니다.

하지만, 클락으로 동작하는 논리회로의 수준들로 인한 한계는 3D 융합으로 감소되진 못할 것이고, 상기한 장점을 달성하면서도 상품화하기 위해서는 3D 융합에서의 상당한 수준의 제조, 테스트, 전력 제공, 그리고 발열 처리 등의 문제들이 해결되어야 합니다. 발열 처리는 훌륭한 열의 전도체이지만 전자에는 절연체인 다이아몬드에 기반한 반도체를 사용해 해결될 수 있을 것입니다. 그렇지만, 웨이퍼를 만들기 위한 커다란 단일 다이아몬드 크리스탈을 만드는 것은 어려운 것으로 알려져 있습니다. 또한, 이런 기술들 중 어느 것도 몇몇 사람들은 이미 익숙해진 이 문제에 커다란 개선을 가져다 주진 못할 것 같습니다. 그렇지만, 짐 그레이의 “연기나는 텔루성이 골프공들” [Gra02]로 가기 위해서는 해결되어야만 하는 단계입니다.

3.3.2 Novel Materials and Processes

스티븐 호킹은 반도체 제조사들이 두개의 근본적 문제를 가지고 있다고 이야기했다고 합니다: (1) 빛의 제한된 속도와 (2) 물질의 원자성의 본질 [Gar07]. 반도체 제조사에서 이런 제약을 해결해 보려 하는 건 가능하겠지만, 이런 근본적 한계들을 비켜나가는데 집중해온 연구와 개발 과정에서 얻어진 몇 가지 방법만이 존재할 뿐입니다.

물질의 원자성의 본질에 대한 한개의 회피책은 보다 큰 기기가 실현 불가능하도록 작은 기기의 전자적 특성을 흉내내게 하는, “high-k dielectric”이라 불리는 물질들입니다. 이 물질들은 일부 쉽게 해결하기 어려운 제조 공정 문제를 가지고 있지만, 선도자들이 한걸음 더 나아가게 하는데 도움을 줄 수도 있습니다. 또 다른 좀 더 신기한 회피책은 하나의 전자는 여러 에너지 레벨을 가

질수 있다는 점을 이용해 여러 비트들을 하나의 전자에 저장하는 것입니다. 이 방법이 상품화된 반도체 기기에서 안정적으로 동작하게 될 수 있을지는 아직 확실하지 않습니다.

또 다른 제안된 회피책은 훨씬 작은 기기 크기들을 이용하는 “quantum dot” 방법입니다만 아직 연구 단계에 머물러 있습니다.

3.3.3 Light, Not Electrons

빛의 속도는 매우 가혹한 제약이지만, 반도체 물질 내부의 전자파는 진공에서의 빛의 속도의 3%에서 30% 사이 속도로 움직이기 때문에, 반도체 기기는 빛의 속도보다는 전자의 속도에 제한되고 있다고 볼 수 있습니다. 실리콘 기기에서 구리로된 접합부를 사용하는 것은 전자의 속도를 높이는 한 방법이고, 그 외에도 추가적인 방법을 사용하면 실제 빛의 속도에 더 가까이 다가갈 수 있을 것입니다. 덧붙이자면, 유리에서의 빛의 속도는 진공에서의 빛의 속도의 60% 이상이라는 사실에 기초해 칩들 사이에 작은 광섬유를 연결부로 사용한 실험도 있었습니다. 그런 광섬유 사용의 한 가지 문제는 전자와 빛 사이의 변환의 효율성이 떨어진다는 점으로, 이는 에너지 소모와 발열 처리 문제를 일으킵니다.

그렇다면 하나, 물리학 쪽에서의 근본적 진전이 없이는 데이터 흐름 속도의 폭발적인 증가는 진공에서의 빛의 속도에 제한될 것입니다.

3.3.4 Special-Purpose Accelerators

특정 문제에 사용되는 범용 CPU는 실제 문제에 크게 관련되지 않은 부분에 많은 시간과 에너지를 소모하고 있게 되는 경우가 많습니다. 예를 들어, 두개의 벡터의 내적을 구하는 경우, 범용 CPU는 일반적으로 루프 카운터를 사용해 루프 (아마도 루프 언롤링을 적용하지 않은채)를 돌릴 겁니다. 인스트럭션을 디코드하고, 루프 카운터를 증가시키고, 이 카운터의 값을 체크하고, 루프의 시작지점으로 실행흐름을 다시 옮기는 일은 어떻게 보면 좀 낭비스럽습니다: 실제 목표는 그게 아니라 두 벡터의 연관된 원소들을 곱하는 거니까요. 따라서, 벡터들을 곱하는데 특수하게 설계된 특별한 하드웨어 부품은 해당 작업을 보다 에너지를 적게 쓰고 보다 빠르게 해결할 수 있습니다.

이게 현존하는 많은 상용 마이크로프로세서에 존재하는 벡터 연산 명령어들의 실제 모티베이션이 되었습니다. 이런 명령어들은 여러 데이터 항목들에 동시적으로 수행되기 때문에, 보다 적은 인스트럭션 디코드와 루프 오버헤드만으로 내적 연산을 완료할 수 있을 겁니다.

비슷하게, 특수화된 하드웨어는 보다 효율적으로 암호화와 복호화, 압축과 압축 해제, 인코딩과 디코딩, 그리고 그외에도 여러 많은 작업을 처리할 수 있습니다.

안타깝게도, 이런 효율성은 공짜로 오진 않습니다. 이런 특수화된 하드웨어를 내장하는 컴퓨터 시스템은 더 많은 트랜지스터를 장착하게 되고, 이는 곧 일부 전력의 소모를 의미하는데, 심지어 사용중이지 않을 때도 전력을 소모할 수 있습니다. 이 특수 하드웨어의 장점을 활용하기 위해선 소프트웨어도 수정되어야 하는데, 이렇게 되면 해당 하드웨어는 충분히 범용적으로 사용될 수 있어서 해당 특수 하드웨어가 충분히 구매할 만 하도록 그 하드웨어의 웨이트 앤드 설계 비용이 충분히 많은 사용자에게 나뉘어 질 수 있어야만 합니다. 부분적으로는 이런 부류의 경제적 고려사항 때문에 특수화된 하드웨어는 그래픽 처리 (GPU), 벡터 처리기 (MMX, SSE, 그리고 VMX 명령어들), 그리고 암호화 등의 적은 어플리케이션 분야에만 나타나곤 했습니다.

서버와 PC 분야와 달리, 스마트폰은 다양한 하드웨어 가속기를 사용해 왔습니다. 이런 하드웨어 가속기는 CPU가 완전히 잡든 채로 고성능의 MP3 플레이어가 오디오를 재생 가능하도록 미디어 디코딩에 주로 사용되었습니다. 이런 가속기의 목적은 에너지 효율성을 개선해서 배터리 수명을 늘리는 것입니다: 특수 목적 하드웨어는 많은 경우 범용 CPU 보다 더 효율적으로 연산을 처리할 수 있습니다. 이건 Section 2.2.3에서 다룬 기본 요소에 대한 또 하나의 예입니다: 제너럴리티는 거의 항상 공짜가 아닙니다.

무어의 법칙으로 인한 싱글 쓰레드 성능 향상이 멈춘 이상, 앞으로는 더 다양한 특수 목적 하드웨어가 나타날 것이라고 보여집니다.

3.3.5 Existing Parallel Software

멀티코어 CPU는 컴퓨팅 산업을 놀라게 한 것 같지만, 사실 공유 메모리 병렬 컴퓨터 시스템은 25년여 전부터 판매되었습니다. 이건 중대한 병렬 소프트웨어가 나타나기에 충분한 시간이 되고도 남고, 그리고 실로 그려했습니다. 병렬 운영 체제는 상당히 흔하고, 병렬 쓰레딩 라이브러리와 병렬 관계형 데이터베이스 관리 시스템, 그리고 병렬 수학 분야 소프트웨어가 그렇습니다. 이미 존재하는 병렬 소프트웨어를 사용하는 것은 우리가 마주한 어떤 병렬 소프트웨어 위기를 해결하는데 많은 도움을 줄 수 있습니다.

아마도 가장 대표적인 예는 병렬 관계형 데이터베이스 관리 시스템일 것입니다. 종종 하이 레벨 스크립트 언어로 짜여지는 싱글 쓰레드 프로그램들에서는 중앙의 관계형 데이터베이스에 동시적으로 접근할 일이 별로 없을 겁니다. 최종적으로 사용되는 고도로 병렬화된 시스템에서는 데이터베이스 자체만이 실질적으로 병렬성을 직접 고려하면 되는 존재입니다. 제대로 먹힌다면 매우 훌륭한 트릭이죠!

3.4 Software Design Implications

Table 3.1에 나온 비율 값들은 주어진 병렬 어플리케이션의 효율성을 제한하기 때문에 매우 중요합니다. 해당 병렬 어플리케이션이 쓰레드들간에 통신을 하기 위해 CAS를 사용한다고 생각해 보세요. 이 CAS 오퍼레이션들은 쓰레드들이 자기 혼자 하는게 아니라 다른 쓰레드들과 통신을 하기 위해 사용하는 것이기 때문에 자주 캐시 미스를 낼 것입니다. 더 나아가서 각각의 CAS 통신 오퍼레이션에 뒤따르는 일의 단위가 부동 소수점 연산 작업 정도는 충분히 할 수 있는 시간인 300ns인 경우를 상상해 보세요. 그렇게 되면 실행 시간의 절반 가량이 CAS 통신 오퍼레이션만으로 소모되는 겁니다! 이건 결국 그런 병렬 프로그램을 돌리는 두개짜리 CPU로 구성된 시스템은 한개짜리 CPU 시스템에서 돌아가는 순차적 구현보다도 빠르게 동작하지는 못한다는 이야기입니다.

단일 통신 오퍼레이션의 대기시간이 수천 또는 심지어 수백만 부동소수점 연산만큼이나 느린 분산 시스템의 경우엔 더 상황이 나빠집니다. 이는 통신 작업이 극단적으로 가끔만 일어나야 하고 매우 큰 단위의 연산을 가능하게 해야 하는 것이 얼마나 중요한지를 잘 보여줍니다.

Quick Quiz 3.9: 분산 시스템에서 통신이 그렇게까지 비싸다면 누가, 그리고 왜 그걸 쓰려 하는 건가요?

■ 교훈은 분명합니다: 병렬 알고리즘들은 거의 독립적인 쓰레드들에 의해 수행되도록 명시적으로 설계되어야 합니다. 어토믹 오퍼레이션을 사용하든, 락이나 명시적 메세지를 사용하든, 쓰레드들의 커뮤니케이션이 덜 빈번할수록 어플리케이션의 성능과 확장성은 나아질 것입니다. 요약하자면, 훌륭한 병렬 성능과 확장성을 달성하는 것은 조심스럽게 데이터 구조와 알고리즘을 선택해서든, 존재하는 병렬 어플리케이션과 환경을 사용해서든, 또는 문제를 당황스럽도록 병렬적인 해결책이 존재하는 문제로 변환해서든 당황스럽도록 병렬적인 알고리즘과 구현을 위해 노력하는 것입니다.

Quick Quiz 3.10: 좋아요, 우리가 분산 프로그래밍 기법들을 공유 메모리 병렬 프로그램에 적용하려 한다면, 항상 이런 분산 기법들을 사용하고 공유 메모리 없이 살면 안되나요?

■ 자, 정리해 보죠:

1. 좋은 소식. 멀티코어 시스템이 저렴하고 어디서든 구할 수 있게 되었습니다.
2. 더 좋은 소식도 있어요: 많은 동기화 오퍼레이션의 오버헤드는 2000년대 초의 병렬 시스템에서 그랬던 것보다 훨씬 낮아졌습니다.

3. 나쁜 소식은 캐시 미스의 오버헤드는, 특히 큰 시스템에서는 여전히 높다는 것입니다.

이 책의 뒷부분에서는 이 나쁜 소식을 처리하는 방법들을 설명합니다.

특히, Chapter 4에서는 병렬 프로그래밍에서 사용되는 일부 로우 레벨 도구들을 다룰 거고, Chapter 5에서는 병렬 카운팅에서의 문제와 해결책을 알아볼겁니다. 그리고 Chapter 6에서는 성능과 확장성을 올릴 수 있는 설계 원칙을 이야기해 봅니다.

Chapter 4

Tools of the Trade

이 챕터에서는 리눅스와 유사한 운영체제에서 돌아가는 어플리케이션에 집중해서 몇몇 기본적인 병렬 프로그래밍 도구를 소개합니다. Section 4.1은 스크립트 언어로 시작을 하고, Section 4.2에서는 POSIX API로 지원되는 멀티 프로세스 병렬성을 설명하고 POSIX 쓰레드를 다뤄봅니다. Section 4.3은 어토믹 오퍼레이션들을 설명하며, Section 4.4에서는 리눅스 커널에서의 유사한 오퍼레이션들을 알아봅니다. 그리고 마지막으로, Section 4.5에서는 해당 일을 완료하기 위해 어떤 도구를 골라야 할지 선택을 도와드립니다.

이 챕터는 간략한 소개만을 제공한다는 점을 기억해 두세요. 더 자세한 내용은 인용된 참조 목록들에서 볼 수 있으며, 이 도구들을 어떻게 사용하는지 최선인지는 뒤의 챕터들에서 설명합니다.

4.1 Scripting Languages

리눅스 셸 스크립트 언어들은 병렬성을 관리하는 간단하지만 효과적인 방법들을 제공합니다. 예를 들어, 당신이 `compute_it`이라는 이름의 프로그램을 가지고 있는데 두개의 다른 인자들로 두번 수행해야 한다고 생각해 봅시다. 유닉스 셸 스크립트를 사용하면 다음과 같이 수행을 할 수 있습니다:

```
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
```

라인 1과 2는 이 프로그램의 인스턴스를 두개 실행시키고, 각 인스턴스의 결과물을 두개의 별개의 파일에 집어넣는데, & 문자는 셸이 그 두 프로그램 인스턴스를 백그라운드에서 실행하도록 합니다. 라인 3은 두개의 인스턴스 모두가 종료되길 기다리고, 라인 4와 5에서는 그들의 결과값을 화면에 출력합니다. 실행 흐름은 Figure 4.1에 나타난 대로입니다: `compute_it`의 두개의 인스턴스는 병렬적으로 수행되고, `wait`은 두개

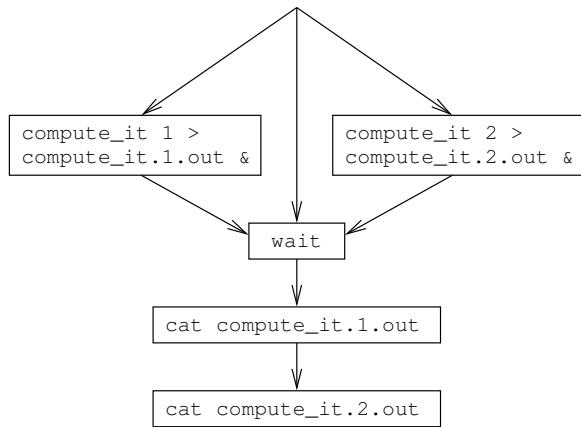


Figure 4.1: Execution Diagram for Parallel Shell Execution

인스턴스 모두가 완료된 뒤에 실행 완료되며, 그 뒤에 `cat`의 두개의 인스턴스가 순차적으로 수행됩니다.

Quick Quiz 4.1: 하지만 이 간단한 셸 스크립트는 진짜 병렬 프로그램이 아니잖아요! 왜 이런 별거아닌 걸 신경쓰는거죠???

Quick Quiz 4.2: 병렬 셸 스크립트를 작성하는 좀 더 간단한 방법은 없나요? 만약 있다면, 어떻게 하나요? 없다면, 왜 없죠?

다른 예로, `make` 소프트웨어 빌드 스크립트 언어는 얼마나 많은 병렬성이 해당 빌드 작업에 주어져야 하는지 결정하는 `-j` 옵션을 제공합니다. 예를 들어, `make -j4` 명령을 리눅스 커널 빌드를 위해 내리게 되면 최대 4개의 병렬 컴파일이 동시에 수행될 수 있습니다.

이런 간단한 예가 병렬 프로그래밍은 항상 복잡하거나 어려울 필요는 없음을 당신에게 납득시켜 주길 바랍니다.

```

1 pid = fork();
2 if (pid == 0) {
3     /* child */
4 } else if (pid < 0) {
5     /* parent, upon error */
6     perror("fork");
7     exit(-1);
8 } else {
9     /* parent, pid == child ID */
10 }

```

Figure 4.2: Using the fork() Primitive

```

1 void waitall(void)
2 {
3     int pid;
4     int status;
5
6     for (;;) {
7         pid = wait(&status);
8         if (pid == -1) {
9             if (errno == ECHILD)
10                 break;
11             perror("wait");
12             exit(-1);
13         }
14     }
15 }

```

Figure 4.3: Using the wait() Primitive

니다.

Quick Quiz 4.3: 하지만 스크립트 기반 병렬 프로그래밍이 그렇게 쉽다면, 왜 다른 것들을 신경쓰는거죠?

4.2 POSIX Multiprocessing

이 섹션에서는 이미 사용 가능하고 여러 구현체가 존재하는 POSIX 환경에 대해, `pthread` [Ope97]를 포함해 알아봅니다. Section 4.2.1에서는 POSIX `fork()` 와 관련된 것들을 살짝 훑어보고, Section 4.2.2에서는 쓰레드 생성과 소멸에 대해 간단히 알아본 후, Section 4.2.3에서는 POSIX 락킹에 대한 짧은 개요를 제공할 예정입니다. 그리고, 마지막으로, Section 4.2.4에서 여러 쓰레드에 의해 읽히고 가끔만 갱신되는 데이터에 대해 사용되곤 하는 특정한 락에 대해 설명합니다.

4.2.1 POSIX Process Creation and Destruction

프로세스는 `fork()`를 통해 생성되고, `kill()`를 통해 소멸될 수도, 스스로 `exit()`를 통해 소멸될 수도 있습니다. `fork()`를 실행하는 프로세스는 새로 생성되는 프로세스의 “부모”라고 불리웁니다. 부모는 자신의 자식을 `wait()`를 통해 기다릴 수도 있습니다.

이 섹션의 예제들은 상당히 간단한 것들이란 점을 기억해 두시기 바랍니다. 이 간단한 도구들을 사용하는 실제 어플리케이션들은 시그널, 파일 디스크립터, 공유된 메모리 조작, 그리고 또다른 많은 자원들을 사용해야만 할 수도 있을 겁니다. 또한, 어떤 어플리케이션은 어떤 자식 프로세스가 종료되었을 때 특별한 행동을 취해야 할 수도 있고, 또한 자식 프로세스가 어떤 이유로 종료되었는지에 대해서도 신경써야 할 수 있습니다. 이렇게 신경써야 하는 부분들은 물론 코드에 상당한 복잡도를 추가할 수 있습니다. 더 많은 내용을 위해서는, 해당 주제에 대한 교재들 [Ste92, Wei13]을 보시기 바랍니다.

`fork()`가 성공하면, `fork()`는 한번은 부모에게, 또한번은 자식에게 두번 리턴합니다. `fork()`가 리턴

하는 값은 Figure 4.2(`forkjoin.c`)에 보여진 것처럼 콜러가 그 차이를 알 수 있게 합니다. 라인 1은 `fork()` 함수를 실행하고, 그 리턴값을 지역 변수 `pid`에 저장합니다. 라인 2에서는 `pid` 가 0인지 체크하는데, 만약 맞다면 자신은 자식 프로세스라는 뜻이며, 이 경우 코드 실행은 라인 3으로 이어집니다. 앞서 이야기 했듯, 자식 프로세스는 `exit()` 함수를 통해 종료될 수 있습니다. 만약 `fork()` 리턴값이 0이 아니어서 자신이 부모 프로세스라면, 라인 4에서 얻은 `fork()` 리턴값을 가지고 라인 5-7에서 에러가 있었는지 확인하고 에러가 있었다면 에러를 화면에 출력하고 종료합니다. 그렇지 않고 `fork()`가 성공적으로 수행되었다면, 자식 프로세스의 프로세스 ID 값을 가지고 있는 변수 `pid` 와 함께 라인 9로 넘어갑니다.

부모 프로세스는 자식 프로세스가 완료될 때까지 `wait()` 함수를 이용해 기다릴 수도 있습니다. 하지만, 각 `wait()` 호출은 오로지 한개 자식 프로세스만 기다리기 때문에, 이 함수의 사용은 셀 스크립트에서 사용할 때보다 좀 더 복잡합니다. 그래서 Figure 4.3(`api-pthread.h`)에서 보여진, 셀 스크립트에서의 `wait` 명령과 유사한 의미를 가진, `waitall()` 함수와 비슷한 함수로 `wait()`를 감싸는게 일반적입니다. 라인 6-15에서의 루프의 각 패스에서는 한개 자식 프로세스를 기다립니다. 라인 7에서는 `wait()` 함수를 호출하는데, 이로 인해 부모 프로세스는 자식 프로세스 하나가 종료할 때까지 블록되어 있고, 자식 프로세스가 종료된 후 자식 프로세스의 프로세스 ID를 리턴합니다. 만약 프로세스 ID가 아니라 -1이 리턴된다면, 이는 `wait()` 함수가 자식 프로세스를 기다릴 수 없었음을 의미합니다. 만약 그렇다면, 라인 9에서 `errno`가 `ECHILD`로 되었는지 여부를 체크하는데, 만약 그렇다면 더이상 자식 프로세스가 존재하지 않았다는 의미로, 이 때엔 라인 10에서 루프를 빠져나옵니다. 그렇지 않다면, 라인 11과 12에서 에러를 출력하고 종료합니다.

Quick Quiz 4.4: 왜 이 `wait()` 함수는 그렇게 복잡

```

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=%d\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);

```

Figure 4.4: Processes Created Via `fork()` Do Not Share Memory

해야만 하는거죠? 왜 그냥 셸 스크립트의 `wait` 같이 동작하도록 만들지 않는 거예요?

부모와 자식 프로세스가 메모리를 공유하지 않는다는 점은 매우 중요하므로 반드시 기억해야 합니다. 이는 Figure 4.4 (`forkjoinvar.c`)에 나타난 프로그램에 보여지는데, 자식 프로세스는 라인 6에서 전역 변수 `x`를 1로 만들고 라인 7에서 메세지를 프린트한 후, 라인 8에서 종료합니다. 부모는 라인 14에서 실행 흐름을 이어서 자식 프로세스를 기다리고 라인 15에서 변수 `x`의 값을 보지만 여전히 그 값은 0입니다. 따라서 이 프로그램의 출력은 다음과 같습니다:

```

Child process set x=1
Parent process sees x=0

```

Quick Quiz 4.5: 여기서 이야기한 것 외에도 `fork()` 와 `wait()`에 대해 이야기할 것들이 많지 않나요?

가장 잘게 크리티컬 섹션을 끼갠 병렬성은 공유 메모리를 필요로 하며, 이는 Section 4.2.2에서 다릅니다. 참고로, 공유 메모리 병렬성은 `fork-join` 병렬성에 비해 상당히 복잡할 수 있습니다.

4.2.2 POSIX Thread Creation and Destruction

프로세스 내에서 쓰레드를 생성하려면 Figure 4.5(`pcreate.c`)의 라인 15와 16에 보인 것처럼 `pthread_create()` 함수를 호출해야 합니다. 첫번째 인자는 `pthread_t` 타입 변수로의 포인터로, 해당 쓰레드의 ID를 저장하게 되며, 두 번째로 예제에서는 `NULL` 값을 준 인자는 필요하면 추가하게 되는 `pthread_attr_t` 타입 변수로의 포인터이며, 세번째 인자는 새로 생성된 쓰레드에 의해

```

1 int x = 0;
2
3 void *mythread(void *arg)
4 {
5     x = 1;
6     printf("Child process set x=%d\n");
7     return NULL;
8 }
9
10 int main(int argc, char *argv[])
11 {
12     pthread_t tid;
13     void *vp;
14
15     if (pthread_create(&tid, NULL,
16                         mythread, NULL) != 0) {
17         perror("pthread_create");
18         exit(-1);
19     }
20     if (pthread_join(tid, &vp) != 0) {
21         perror("pthread_join");
22         exit(-1);
23     }
24     printf("Parent process sees x=%d\n", x);
25     return 0;
26 }

```

Figure 4.5: Threads Created Via `pthread_create()` Share Memory

호출될 함수(이 경우 `mythread()`)이고, 마지막으로 여기선 `NULL`을 준 인자는 `mythread`에게 전달될 인자입니다.

이 예제에서, `mythread()`는 단순히 리턴하지만, 대신 `pthread_exit()`를 사용할 수도 있습니다.

Quick Quiz 4.6: Figure 4.5의 `mythread()` 함수가 그냥 리턴해도 된다면, 왜 `pthread_exit()`를 신경 써야하죠?

라인 20에 있는 `pthread_join()` 함수는 `fork-join`에서의 `wait()` 함수와 유사합니다. 이 함수는 `tid` 변수로 이야기된 쓰레드가 `pthread_exit()` 함수를 통해서나 쓰레드의 탑 레벨 함수가 리턴을 하거나 해서 실행을 완료할 때까지 기다립니다. 쓰레드의 종료 값은 `pthread_join()` 함수에 두번째 인자로 넘겨진 포인터를 통해 저장됩니다. 쓰레드의 종료 값은 쓰레드가 어떻게 종료되었는지에 따라 다른데, `pthread_exit()` 함수에 넘겨진 값이거나 쓰레드의 탑 레벨 함수에서 리턴한 값입니다.

Figure 4.5에 보여진 프로그램은 다음과 같이 결과를 내놓게 되는데, 이 결과는 메모리가 두개 쓰레드간에 공유됨을 보여줍니다:

```

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

```

이 프로그램은 변수 `x`에 값을 저장하는 쓰레드가 한 번에 하나 뿐임을 확실히 하기 위해 많은 신경을 쓰고

있음을 알아 두십시오. 한 쓰레드가 어떤 변수에 값을 저장하는 동안 다른 쓰레드가 그 변수의 값을 읽거나 쓰려 하는 상황을 가리켜 “data race(데이터 레이스)”라 합니다. C 언어는 데이터 레이스의 결과에 대해 어떤 보장도 하지 않기 때문에, 우리는 다음 섹션에서 이야기할 락킹 도구들과 같이 데이터에의 동시적 접근과 수정을 안전하게 할 수 있는 방법이 필요합니다.

Quick Quiz 4.7: C 언어가 데이터 레이스에 대해 어떤 보장도 하지 않는다면, 왜 리눅스 커널은 그렇게 많은 데이터 레이스들을 가지고 있는거죠? 지금 리눅스 커널이 완전 영망이라고 이야기 하려는 거예요???

4.2.3 POSIX Locking

POSIX 표준은 프로그래머가 “POSIX 락킹”을 이용해 데이터 레이스 상황을 회피할 수 있게 합니다. POSIX 락킹은 여러개의 기본 기능을 제공하는데, 가장 기본적인 것들은 `pthread_mutex_lock()`과 `pthread_mutex_unlock()`입니다. 이 기본 기능들은 `pthread_mutex_t` 타입인 락에 대해 동작합니다. 이 락들은 정적으로 선언되고 `PTHREAD_MUTEX_INITIALIZER`를 통해 초기화 될 수도, 동적으로 할당된 후 `pthread_mutex_init()`를 통해 초기화 될 수도 있습니다. 이 섹션의 예제 코드는 앞의 경우들을 취할 것입니다.

`pthread_mutex_lock()` 함수는 특정 락을 “획득”하고, `pthread_mutex_unlock()` 함수는 특정 락을 “해제”합니다. 이것들은 “명시적” 락킹 함수들이기 때문에, 한 순간에 하나의 쓰레드만이 특정 락을 “가질” 수 있습니다. 예를 들어, 두개의 쓰레드들이 같은 락을 동시에 획득하려 하면, 그 중 하나만이 먼저 락을 얻을 수 있도록 “허락”되고, 다른 쓰레드는 첫번째 쓰레드가 락을 해제할 때까지 기다려야 합니다.

Quick Quiz 4.8: 제가 여러 쓰레드들이 한번에 같은 락을 쥐고 있게 하고 싶으면 어떻게 하죠?

Figure 4.6 (lock.c)에 명시적 락킹의 사용 예가 있습니다. 라인 1은 `lock_a`라는 이름의 POSIX 락을 정의와 함께 초기화하고, 라인 2에서는 비슷하게 `lock_b`라는 이름의 락을 정의하고 초기화 합니다. 라인 3에서는 공유 변수 `x`를 정의와 함께 초기화 합니다.

라인 5-28은 `arg`로 가리켜진 락을 잡고서 공유 변수 `x`를 반복적으로 읽는 `lock_reader()` 함수를 정의합니다. 라인 10은 `arg`를 `pthread_mutex_lock()`과 `pthread_mutex_unlock()` 함수에 사용하기 위해 `pthread_mutex_t` 포인터로 캐스팅합니다.

Quick Quiz 4.9: 왜 그냥 Figure 4.6 라인 5에서

```

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
12    if (pthread_mutex_lock(pmlp) != 0) {
13        perror("lock_reader:pthread_mutex_lock");
14        exit(-1);
15    }
16    for (i = 0; i < 100; i++) {
17        newx = ACCESS_ONCE(x);
18        if (newx != oldx) {
19            printf("lock_reader(): x = %d\n", newx);
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
36     if (pthread_mutex_lock(pmlp) != 0) {
37         perror("lock_writer:pthread_mutex_lock");
38         exit(-1);
39    }
40    for (i = 0; i < 3; i++) {
41        ACCESS_ONCE(x)++;
42        poll(NULL, 0, 5);
43    }
44    if (pthread_mutex_unlock(pmlp) != 0) {
45        perror("lock_writer:pthread_mutex_unlock");
46        exit(-1);
47    }
48    return NULL;
49 }
```

Figure 4.6: Demonstration of Exclusive Locks

```

1 printf("Creating two threads using same lock:\n");
2 if (pthread_create(&tid1, NULL,
3                     lock_reader, &lock_a) != 0) {
4     perror("pthread_create");
5     exit(-1);
6 }
7 if (pthread_create(&tid2, NULL,
8                     lock_writer, &lock_a) != 0) {
9     perror("pthread_create");
10    exit(-1);
11 }
12 if (pthread_join(tid1, &vp) != 0) {
13     perror("pthread_join");
14     exit(-1);
15 }
16 if (pthread_join(tid2, &vp) != 0) {
17     perror("pthread_join");
18     exit(-1);
19 }

```

Figure 4.7: Demonstration of Same Exclusive Lock

lock_reader() 가 곧바로 pthread_mutex_t 포인터를 받도록 하지 않는거죠?

라인 12-15 는 특정 pthread_mutex_t 를 획득하고, 에러를 체크한 후 만약 에러가 있었다면 프로그램을 종료합니다. 라인 16-23 은 반복적으로 x 의 값을 채크하고, 그 값이 바뀔 때마다 새로운 값을 화면에 출력합니다. 라인 22 는 예제가 싱글 프로세서 머신에서도 잘 돌아가도록 1 밀리세컨드씩 잠을 잡니다. 라인 24-27 에서는 pthread_mutex_t 를 해제하고, 에러를 체크한 후 에러가 났다면 프로그램을 종료합니다. 마지막으로, 라인 28 에서는 pthread_create() 에서 요구된 함수 타입을 맞춰주기 위해 NULL 을 리턴합니다.

Quick Quiz 4.10: pthread_mutex_t 의 획득과 해제에 매번 4줄이나 써야한다니 좀 고통스러울 것 같군요! 더 나은 방법은 없나요?

Figure 4.6 의 라인 31-49 는 주기적으로 공유 변수 x 를 특정 pthread_mutex_t 를 잡은 채로 업데이트 하는 lock_writer() 함수를 보여줍니다. lock_reader() 에서처럼 라인 34 에서는 arg 를 pthread_mutex_t 포인터로 캐스팅하고 라인 36-39 에서 해당 락을 얻어오고, 라인 44-47 에서 락을 놓아줍니다. 락을 잡고 있는 동안, 라인 40-43 에서는 공유 변수 x 를 5 밀리세컨드 씩 자면서 증가시킵니다. 마지막으로 라인 44-47 에서는 락을 놓아줍니다.

Figure 4.7 는 lock_reader() 와 lock_writer() 를 같은 lock_a 락을 사용하도록 하면서 쓰레드로 실행시키는 코드를 보여줍니다. 라인 2-6 은 lock_reader() 를 실행하는 쓰레드를 생성하고, 라인 7-11 에서는 lock_writer() 를 실행하는 쓰레드를 생성합니다. 라인 12-19 에서는 두

```

1 printf("Creating two threads w/different locks:\n");
2 x = 0;
3 if (pthread_create(&tid1, NULL,
4                     lock_reader, &lock_a) != 0) {
5     perror("pthread_create");
6     exit(-1);
7 }
8 if (pthread_create(&tid2, NULL,
9                     lock_writer, &lock_b) != 0) {
10    perror("pthread_create");
11    exit(-1);
12 }
13 if (pthread_join(tid1, &vp) != 0) {
14     perror("pthread_join");
15     exit(-1);
16 }
17 if (pthread_join(tid2, &vp) != 0) {
18     perror("pthread_join");
19     exit(-1);
20 }

```

Figure 4.8: Demonstration of Different Exclusive Locks

쓰레드가 완료되기를 기다립니다. 이 코드가 내놓는 결과는 다음과 같습니다:

```

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

```

두 쓰레드가 모두 같은 락을 사용하기 때문에, lock_reader() 쓰레드는 lock_writer() 가 락을 잡고서 만들어내는 x 의 중간 값들을 볼 수 없습니다.

Quick Quiz 4.11: “x = 0” 만이 Figure 4.7 의 코드에서 발생 가능한 오로지 하나의 결과인가요? 만약 그렇다면, 왜죠? 아니라면, 어떤 다른 결과가 가능할까요, 그리고 왜일까요?

Figure 4.8 에서 비슷한, 하지만 이번엔 다른 락을 사용하는 코드를 보여줍니다: lock_reader() 를 위해선 lock_a 를, lock_writer() 를 위해선 lock_b 를 사용합니다. 이 코드의 수행 결과는 다음과 같습니다:

```

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

```

두 쓰레드가 다른 락을 사용하기 때문에, 서로를 배제하지 않고, 동시에 수행됩니다. 그래서 lock_reader() 함수는 lock_writer() 가 저장한 x 중간값을 볼 수 있습니다.

Quick Quiz 4.12: 서로 다른 락을 사용하는건 쓰레드가 서로 상대의 중간 상태를 볼 수 있는등 혼란스럽게 할 수 있는 것같은데요. 잘 짜여진 병렬 프로그램은 이런 혼란을 막기 위해서는 하나의 락만을 사용해야만 하는 건가요?

Quick Quiz 4.13: Figure 4.8 에 보여진 코드에서, lock_reader() 는 lock_writer() 가 생성하는 값 모두를 보도록 보장되어 있나요? 그렇다면, 또 그렇지 않다면, 왜죠?

Quick Quiz 4.14: 잠깐만요!!! Figure 4.7 에서는 공유 변수 x 를 초기화 하지 않았는데, Figure 4.8 에서는 왜 초기화 해야 했던거죠?

이외에도 몇가지 더 POSIX 명시적 락킹이 있지만, 여기 소개한 것만으로도 좋은 시작이 될 수 있고, 많은 상황에는 이것만으로도 충분할 겁니다. 다음 섹션에서는 POSIX 리더-라이터 락킹에 대해 간략히 알아봅니다.

4.2.4 POSIX Reader-Writer Locking

POSIX API 는 pthread_rwlock_t 로 사용되는, 리더-라이터 락을 제공합니다. pthread_mutex_t 처럼, pthread_rwlock_t 는 PTHREAD_RWLOCK_INITIALIZER 를 사용해 정적으로 초기화 될 수도, pthread_rwlock_init() 함수를 이용해 동적으로 초기화 될 수도 있습니다. pthread_rwlock_rdlock() 함수는 특정 pthread_rwlock_t 의 읽기 권한을 얻어오고, pthread_rwlock_wrlock() 함수는 쓰기 권한을 얻어오며, pthread_rwlock_unlock() 함수는 락을 해제합니다. 언제든 pthread_rwlock_t 의 쓰기권한은 하나의 쓰레드만이 획득 가능하며, 읽기 권한은 여러 쓰레드가 동시에 가질 수 있습니다만, 동시에 쓰기 권한을 주는 쓰레드가 없는 경우에 국한됩니다.

예상했겠지만, 리더-라이터 락은 읽기 작업이 대부분인 경우를 위해 설계되었습니다. 이런 상황에서, 리더-라이터 락은 명시적 락에 비해 훨씬 나은 확장성을 제공할 수 있는데, 명시적 락은 기본적으로 한번에 락을 잡을 수 있는 쓰레드의 수가 하나로 제한되는데 반해 리더-라이터 락은 충분히 많은 수의 읽기 작업 하는 쓰레드가 동시에 락을 잡을 수 있기 때문입니다. 하지만, 실제로 리더-라이터 락이 얼마나 추가적인 확장성을 제공하는지 알 필요가 있습니다.

Figure 4.9 (rwlockscales.c) 는 리더-라이터 락의 확장성을 측정하는 한가지 방법을 보여줍니다. 라인 1 은 해당 리더-라이터 락의 정의와 초기화를 하고, 라인 2 에서는 각 쓰레드가 해당 리더-라이터 락을 잡는 시간을 조절하는 holdtime 인자를 보여줍니다. 라인 3 에서는 리더-라이터 락의 해제와 다음 획득 사이의 시간을 조절하는 thinktime 인자를 보여주고, 라인 4 에서는 각 리더 쓰레드가 자신이 락을 획득한 횟수를 저장하는 readcounts 배열을 정의합니다. 그리고 라인 5

```

1 pthread_rwlock_t rwl = PTHREAD_RWLOCK_INITIALIZER;
2 int holdtime = 0;
3 int thinktime = 0;
4 long long *readcounts;
5 int readersrunning = 0;
6
7 #define GOFLAG_INIT 0
8 #define GOFLAG_RUN 1
9 #define GOFLAG_STOP 2
10 char goflag = GOFLAG_INIT;
11
12 void *reader(void *arg)
13 {
14     int i;
15     long long loopcnt = 0;
16     long me = (long)arg;
17
18     __sync_fetch_and_add(&readersrunning, 1);
19     while (ACCESS_ONCE(goflag) == GOFLAG_INIT) {
20         continue;
21     }
22     while (ACCESS_ONCE(goflag) == GOFLAG_RUN) {
23         if (pthread_rwlock_rdlock(&rwl) != 0) {
24             perror("pthread_rwlock_rdlock");
25             exit(-1);
26         }
27         for (i = 1; i < holdtime; i++) {
28             barrier();
29         }
30         if (pthread_rwlock_unlock(&rwl) != 0) {
31             perror("pthread_rwlock_unlock");
32             exit(-1);
33         }
34         for (i = 1; i < thinktime; i++) {
35             barrier();
36         }
37         loopcnt++;
38     }
39     readcounts[me] = loopcnt;
40     return NULL;
41 }
```

Figure 4.9: Measuring Reader-Writer Lock Scalability

에서는 언제 모든 리더 쓰레드들이 수행을 시작했는지 알려주는 `nreadersrunning` 변수를 정의합니다.

라인 7-10은 테스트의 시작과 끝을 맞춰주는 `goflag`를 정의합니다. 이 변수는 처음엔 `GOFLAG_INIT`로 설정되어 있고, 이후에 모든 리더 쓰레드들이 시작된 이후에 `GOFLAG_RUN`으로 변경된 후, 마지막으로 테스트를 종료시키기 위해 `GOFLAG_STOP`으로 값을 바꿉니다.

라인 12-41은 리더 쓰레드인 `reader()` 함수를 정의합니다. 라인 18은 이 쓰레드가 이제 수행됨을 알리기 위해 `nreadersrunning` 변수의 값을 어ト믹하게 증가시키고, 라인 19-21은 테스트가 시작되길 기다립니다. `ACCESS_ONCE()` 함수는 컴파일러가 루프의 매 수행마다 `goflag`를 실제로 읽어오도록 합니다—그러지 않으면 컴파일러는 `goflag`의 값이 영영 변하지 않을 거라고 가정하고 동작하는 권리를 행사할 수도 있습니다.

Quick Quiz 4.15: 여기 저기 모든 곳에서 `ACCESS_ONCE()`를 쓰는 대신에, Figure 4.9의 라인 10에서 `goflag`를 `volatile`로 선언하는게 어때요?

Quick Quiz 4.16: `ACCESS_ONCE()`는 컴파일러에만 영향을 주지, CPU에는 영향을 안주죠. Figure 4.9의 `goflag`의 값의 변화가 시간 순서대로 다른 CPU에게도 전파되게 하려면 메모리 배리어도 쳐야 하지 않나요?

Quick Quiz 4.17: 예를 들어 `gcc __thread` 스토리지 클래스를 사용해 선언된 쓰레드별 변수에 접근할 때에도 `ACCESS_ONCE()`가 필요할까요?

루프가 선언된 라인 22-38은 성능 테스트를 합니다. 라인 23-26은 락을 얻어오고, 라인 27-29는 정해진 시간동안 락을 쥐고 있고 (그리고 `barrier()` 지시어가 컴파일러가 루프를 없애는 것을 막습니다), 라인 30-33에서 락을 풀어주고, 라인 34-36에서 락을 다시 얻어오기 전에 특정 시간동안 기다립니다. 라인 37은 이 락 획득 횟수를 셹니다.

라인 39에서는 해당 락 획득 횟수를 `readcounts[]` 배열의 해당 쓰레드의 원소에 저장하고, 라인 40에서는 리턴해서 쓰레드를 종료합니다.

Figure 4.10에서는 테스트를 코어당 2개 하드웨어 쓰레드를 지원해 소프트웨어에는 총 128개의 CPU가 존재하는 것으로 보이는 64-코어 Power-5 시스템에서 돌린 결과입니다. `thinktime` 패러미터는 모든 테스트에서 0이었고, `holdtime`은 1000(그림에서는 “1K”로 표시되었습니다)부터 1억(그림에선 “100M”으로 표시됩니다)까지 값을 변화시켰습니다. 그림으로 그려

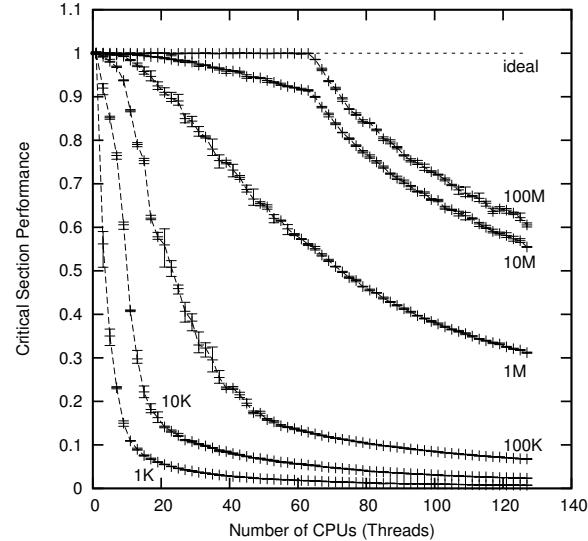


Figure 4.10: Reader-Writer Lock Scalability

진 실제 값은 다음과 같습니다:

$$\frac{L_N}{NL_1} \quad (4.1)$$

N 는 쓰레드의 갯수이고, L_N 는 N 쓰레드들의 락 획득 횟수, 그리고 L_1 는 단일 쓰레드의 락 획득 횟수입니다. 이상적인 하드웨어와 소프트웨어 확장성이 주어졌다면, 이 값은 항상 1.0이 될 것입니다.

그림에서 보여지듯이, 리더-라이터 락킹의 확장성은 이상적이진 않고, 특히 작은 크기의 크리티컬 섹션에서 더 그렇습니다. 왜 읽기 권한 락 획득이 그렇게 느린지 알아보려면, 모든 락 획득을 원하는 쓰레드들이 `pthread_rwlock_t` 데이터를 수정해야 함을 상기해 보십시오. 따라서, 모든 128개의 수행되는 쓰레드들이 리더-라이터 락으로부터 동시에 읽기 권한 락 획득을 하려 하면, 그 쓰레드들은 `pthread_rwlock_t`를 한번씩 반드시 수정해야만 합니다. 운좋은 쓰레드는 곧바로 그렇게 할 수 있을 테지만, 가장 불행한 쓰레드는 다른 127개의 쓰레드들이 수정을 완료할 때까지 기다려야 합니다. 이 상황은 CPU를 추가할수록 나빠지기만 할 겁니다.

Quick Quiz 4.18: 단일 CPU 성능에 비교하는건 좀 심한 거 아닌가요?

Quick Quiz 4.19: 하지만 1,000 개의 인스트럭션은 크리티컬 섹션 치고 그렇게 작은 크기는 아니예요. 수십 개의 인스트럭션 정도만을 가지는 훨씬 작은 크리티컬 섹션이 필요하면 어떻게 해야하죠?

Quick Quiz 4.20: Figure 4.10에서 100M에서의 경우 이외의 값들은 이상적인 선에서 부드럽게 멀어집니다. 반면, 100M에서의 값은 64 CPU에서 갑자기 이상적인 선으로부터 멀어지는군요. 또, 100M값과 10M값 사이의 거리는 10M값과 1M값 사이의 거리보다 작아요. 왜 100M값은 이렇게 남들과 다른거죠?

Quick Quiz 4.21: Power-5는 나온지 몇년이 넘었고, 최신 하드웨어는 분명 더 빠를 거예요. 그런데 왜 리더-라이터 락의 느린 속도에 걱정해야 하죠?

이런 한계점에도 불구하고, 리더-라이터 락킹은 많은 경우, 예를 들어 리더들이 대기시간이 긴 파일이나 네트워크 I/O를 하는 경우 등에 매우 유용합니다. 다른 대안들도 있는데, 그런 것들 중 일부는 Chapter 5와 Chapter 9에서 다루겠습니다.

4.3 Atomic Operations

Figure 4.10가 리더-라이터 락킹의 오버헤드는 크리티컬 섹션이 작을수록 커진다는 것을 보여줬으니, 매우 작은 크리티컬 섹션을 보호할 다른 나은 방안을 찾아봐야 하겠습니다. 그런 한가지 방안은 어토믹 오퍼레이션의 사용입니다. 우린 Figure 4.9의 라인 18에서 `__sync_fetch_and_add()`를 통해 어토믹 오퍼레이션 하나를 본 바 있습니다. 이 기능은 첫번째 인자로 참조된 값에 두번째 인자로 주어진 값을 어토믹하게 더하고, 예전 값(이 경우엔 그냥 무시되었었죠)을 리턴합니다. 두개의 쓰레드가 같은 변수에 대해 동시에 `__sync_fetch_and_add()`를 실행하면 변수의 값은 두 더하기 연산이 모두 수행된 값이 됩니다.

gcc 컴파일러는 이외에도 아래와 같이 추가적인 어토믹 오퍼레이션들을 제공합니다. 먼저 `__sync_fetch_and_sub()`, `__sync_fetch_and_or()`, `__sync_fetch_and_and()`, `__sync_fetch_and_xor()`, 그리고 `__sync_fetch_and_nand()`는 기존 값을 리턴합니다. 기존 값이 아니라 새롭게 바뀐 값이 필요하다면, 아래의 것들이 있습니다. `__sync_add_and_fetch()`, `__sync_sub_and_fetch()`, `__sync_or_and_fetch()`, `__sync_and_and_fetch()`, `__sync_xor_and_fetch()`, 그리고 `__sync_nand_and_fetch()`입니다.

Quick Quiz 4.22: 정말로 이것들이 다 필요한 거 맞나요? ■

고전의 compare-and-swap 오퍼레이션은 `__sync_bool_compare_and_swap()`과 `__sync_val_compare_and_swap()` 두개의 함수로 제공됩니다.

두 함수 모두 어토믹하게 변수의 현재 값이 제시한 값과 같을 경우 새로운 값으로 변경해 줍니다. 첫번째 함수는 오퍼레이션이 성공하면 1을, 그리고 실패하면(예를 들어, 기존 값이 제시한 값과 같지 않은 경우) 0을 리턴합니다. 두번째 함수는 기존 값을 리턴합니다. 따라서 리턴받은 값이 제시했던 기존값과 같다면 오퍼레이션이 성공했음을 의미합니다. 앞의 오퍼레이션들이 적용 가능한 영역에선 대부분 compare-and-swap 보다 더 효율적이지만 하나의 변수에 대한 어떤 어토믹 오퍼레이션도 compare-and-swap을 사용해서 구현 가능하기 때문에, compare-and-swap 오퍼레이션은 “보편적”이라 할 수 있습니다. 뿐만 아니라 compare-and-swap 오퍼레이션은 더 넓은 어토믹 오퍼레이션 집합의 토대 역할을 할 수도 있습니다. 그렇게 만들어진 것들은 보통 복잡도, 확장성, 성능 문제 [Her90]를 겪지만요.

`__sync_synchronize()` 함수는 Section 14.2에서 다루는, 컴파일러와 CPU의 오퍼레이션 재배치 기능을 제한하는 “메모리 배리어”를 불러옵니다. 어떤 경우에는 CPU의 재배치 가능성은 두고 컴파일러의 오퍼레이션 재배치를 제한하는 것만으로도 충분한데, 이 경우에는 Figure 4.9의 라인 28에서처럼 `barrier()` 기능이 사용될 수 있을 겁니다. 어떤 경우에는 컴파일러가 주어진 메모리 접근의 최적화를 하는 것을 막는 것만으로도 충분한 경우가 있을 수 있는데, 이 경우에는 Figure 4.6의 라인 17에서처럼 `ACCESS_ONCE()` 함수가 사용될 수 있을 겁니다. 이 마지막 두개의 함수들은 gcc에 의해 제공되지는 않습니다만 다음과 같이 구현될 수 있을 겁니다:

```
#define ACCESS_ONCE(x) (* (volatile typeof(x) *) &(x))
#define barrier() __asm__ __volatile__ ("": : : "memory")
```

Quick Quiz 4.23: 이 어토믹 오퍼레이션들은 기계의 인스트럭션 셋에서 바로 지원되는 한개짜리 어토믹 인스트럭션으로 변환될테니, 이것들이 일을 돌아가게 할 수 있는 가장 빠른 방법 아닌가요?

4.4 Linux-Kernel Equivalents to POSIX Operations

안타깝게도, 쓰레딩 오퍼레이션들, 락킹 도구들, 그리고 어토믹 오퍼레이션들은 많은 표준 위원회가 그들을 다루기 전에도 매우 많이 사용되었습니다. 그 결과, 이 오퍼레이션들이 지원되는 방법에는 상당히 많은 다양성이 존재합니다. 역사적 이유로든 특정 환경에서 더 나은 성능을 얻기 위해서든, 어셈블리 언어로 구현된 오퍼레이션들도 여전히 많습니다. 예를 들어, gcc `__sync_`

Category	POSIX	Linux Kernel
Thread Management	<code>pthread_t</code>	<code>struct task_struct</code>
	<code>pthread_create()</code>	<code>kthread_create</code>
	<code>pthread_exit()</code>	<code>kthread_should_stop() (rough)</code>
	<code>pthread_join()</code>	<code>kthread_stop() (rough)</code>
	<code>poll(NULL, 0, 5)</code>	<code>schedule_timeout_interruptible()</code>
POSIX Locking	<code>pthread_mutex_t</code>	<code>spinlock_t (rough)</code> <code>struct mutex</code>
	<code>PTHREAD_MUTEX_INITIALIZER</code>	<code>DEFINE_SPINLOCK()</code> <code>DEFINE_MUTEX()</code>
	<code>pthread_mutex_lock()</code>	<code>spin_lock() (and friends)</code> <code>mutex_lock() (and friends)</code>
	<code>pthread_mutex_unlock()</code>	<code>spin_unlock() (and friends)</code> <code>mutex_unlock()</code>
POSIX Reader-Writer Locking	<code>pthread_rwlock_t</code>	<code>rwlock_t (rough)</code> <code>struct rw_semaphore</code>
	<code>PTHREAD_RWLOCK_INITIALIZER</code>	<code>DEFINE_RWLOCK()</code> <code>DECLARE_RWSEM()</code>
	<code>pthread_rwlock_rdlock()</code>	<code>read_lock() (and friends)</code> <code>down_read() (and friends)</code>
	<code>pthread_rwlock_unlock()</code>	<code>read_unlock() (and friends)</code> <code>up_read()</code>
	<code>pthread_rwlock_wrlock()</code>	<code>write_lock() (and friends)</code> <code>down_write() (and friends)</code>
	<code>pthread_rwlock_unlock()</code>	<code>write_unlock() (and friends)</code> <code>up_write()</code>
Atomic Operations	C Scalar Types	<code>atomic_t</code> <code>atomic64_t</code>
	<code>__sync_fetch_and_add()</code>	<code>atomic_add_return()</code> <code>atomic64_add_return()</code>
	<code>__sync_fetch_and_sub()</code>	<code>atomic_sub_return()</code> <code>atomic64_sub_return()</code>
	<code>__sync_val_compare_and_swap()</code>	<code>cmpxchg()</code>
	<code>__sync_lock_test_and_set()</code>	<code>xchg() (rough)</code>
	<code>__sync_synchronize()</code>	<code>smp_mb()</code>

Table 4.1: Mapping from POSIX to Linux-Kernel Primitives

류 함수들은 메모리 순서 의미를 제공해서 많은 개발자들이 메모리 순서 의미가 요구되지 않는 상황을 위한 각자의 구현을 만들도록 이깁니다.

그래서, Table 4.1 on page 35 는 대략적인 POSIX 와 gcc 도구들, 그리고 리눅스 커널 사이의 대응표를 보여줍니다. 완벽한 대체물을 항상 찾긴 불가능한데, 예를 들어 리눅스 커널은 다양한 락킹 도구를 갖는 반면, gcc 는 리눅스 커널에서는 곧바로 사용할수는 없는 여러 어토믹 오퍼레이션들을 제공합니다. 물론, 사용자 레벨 코드는 리눅스 커널의 다양한 락킹 도구들이 필요하진 않으며, gcc 의 어토믹 오퍼레이션도 `cmpxchg()` 를 사용해 흉내내어질 수 있습니다.

Quick Quiz 4.24: 리눅스 커널의 `fork()` 와 `wait()` 대체물은 어디갔죠?

는 것 만큼이나 올바른 설계를 하는 것이 중요합니다.

4.5 The Right Tool for the Job: How to Choose?

대략적 경험으로 얻은 교훈으로 말씀드리건대, 해야하는 일을 완수해주는 가장 간단한 도구를 사용하세요. 만약 가능하다면, 그냥 순차적으로 프로그램을 짜세요. 그게 충분치 않다면, 병렬성을 성립시키기 위해 셸 스크립트를 사용하세요. 그로 말미암은 셸 스크립트의 `fork()`/`exec()` 오버헤드(Intel Core Duo 랩탑에서의 작은 C 프로그램의 경우 약 480 마이크로세컨드)가 지나치게 크다면, C 언어의 `fork()` 와 `wait()` 함수를 사용해 보세요. 이 함수들의 오버헤드(가장 작은 자식 프로세스에 약 80 마이크로세컨드)도 여전히 너무 크다면, POSIX 쓰레딩 도구들에서 적절한 락킹과 필요하면 어토믹 오퍼레이션을 골라서 사용해야 할겁니다. POSIX 쓰레딩 도구들의 오버헤드(대부분의 경우 마이크로세컨드 미만) 조차도 너무 크다면, Chapter 9 에서 소개되는 도구들이 필요할 수 있습니다. 프로세스간 통신과 메세지 패싱은 공유 메모리 멀티 쓰레드 실행의 좋은 대안이 될 수 있다는 점을 항상 기억하세요.

Quick Quiz 4.25: 셸은 기본적으로 `fork()` 가 아니라 `vfork()` 를 사용하지 않나요?

물론, 실제 오버헤드는 당신의 하드웨어에 따라 달라질 수 있을 것입니다만, 그보다는 당신이 해당 도구들을 어떻게 사용하느냐가 훨씬 더 영향을 끼칠 겁니다. 특히, 멀티 쓰레드로 짜여진 코드를 무작위로 해킹하는 건 공유 메모리 병렬 시스템은 당신의 지능을 당신에게 사용하기 때문에 엄청나게 나쁜 생각입니다: 당신이 똑똑할 수록, 당신은 당신이 문제 [Pok16] 에 빠져 있음을 깨닫기 전까지 점점 깊은 구멍에 빠져들 겁니다. 따라서, 뒤의 챕터들에서 이야기하겠지만 개별 도구를 고르

Chapter 5

Counting

카운팅은 아마도 가장 간단하고 가장 자연스런 컴퓨터의 일 중 하나일 것입니다. 하지만, 커다란 공유 메모리 멀티 프로세서에서 효과적이고 확장성 있게 카운팅을 하는 것은 꽤 어려운 일입니다. 더욱이, 카운팅의 간단함은 정교한 데이터 구조나 복잡한 동기화 도구로의 혼란 없이 기본적인 동시성 문제를 볼 수 있게 합니다. 따라서 카운팅은 병렬 프로그래밍으로의 훌륭한 소개 역할을 합니다.

이 챕터는 간단하고 빠르고 확장성 있는 카운팅 알고리즘들 중 일부를 다룹니다. 하지만 먼저, 당신이 얼마나 동시적 카운팅에 대해 알고 있는지 알아보죠.

Quick Quiz 5.1: 대체 왜 효과적이고 확장성 있는 카운팅이 어려운가요? 무엇보다, 컴퓨터들은 카운팅, 더하기, 빼기, 그 외에도 여러가지를 위한 전용 하드웨어도 가지고 있는데, 그걸 못하나요???

Quick Quiz 5.2: 네트워크 패킷 카운팅 문제. 당신이 송수신된 네트워크 패킷의 갯수(또는 전체 용량)에 대한 통계를 구해야 한다고 생각해 봅시다. 패킷들은 시스템의 어떤 CPU를 통해서든 송신 / 수신될 수 있을 겁니다. 나아가서 이 커다란 기계가 초당 백만개의 패킷을 다룰 수 있고, 그 갯수를 매 5초마다 읽어내야 하는 시스템 모니터링 패키지가 있다고 가정해 봅시다. 당신이라면 이 통계 카운터를 어떻게 구현하시겠어요?

Quick Quiz 5.3: 대략적 구조체 할당 한계 문제. 할당된 구조체의 갯수가 어떤 한계(한 10,000 정도)를 넘어가면 추가 할당을 막기 위해 할당된 구조체의 갯수를 유지해야 한다고 생각해 봅시다. 또, 이 구조체들은 할당되고 나서 곧바로 해제되고, 한계치를 넘기는 일은 매우 드물고, “대략적인” 한계치 설정이 가능하다고 생각해 봅시다.

Quick Quiz 5.4: 정교한 구조체 할당 한계 문제. 할당된 구조체의 갯수가 어떤 정확한 한계(여기서도, 한 10,000 정도)를 넘어가면 추가 할당을 막기 위해 할당된

구조체의 갯수를 유지해야 한다고 생각해 봅시다. 이 구조체들은 할당되고 얼마 안되 해제되고, 그 한계는 드물게 초과되며, 거의 항상 최소 한개의 구조체는 사용중이 됩니다. 또한 예를 들어, 하나의 구조체도 사용되지 않고 있다면 해제 할 수 있는 어떤 메모리를 위해 카운터가 0이 되는 시점을 정확히 알 필요가 있습니다.

Quick Quiz 5.5: 제거될 수 있는 I/O 디바이스 접속 카운트 문제. 매우 빈번하게 사용되는 제거 가능한 대용량 디바이스에 대해 사용자에게 해당 디바이스를 제거해도 안전한지 알려주기 위해 그 참조 횟수를 관리해야 한다고 가정해 봅시다. 이 디바이스는 사용자가 디바이스를 제거하고 싶을 때 그 의사를 알려주며, 시스템은 사용자에게 언제 디바이스를 제거해도 안전한지 알려주는 일반적 디바이스 제거 절차를 따릅니다. ■

이 챕터의 뒷부분은 이 문제들에 대한 답을 만들어 볼겁니다. Section 5.1에서는 왜 멀티코어 시스템에서의 카운팅이 간단하지 않은지 다루고, Section 5.2에서는 네트워크 패킷 카운팅 문제를 푸는 방법들을 봅니다. Section 5.3에서는 대략적 구조체 할당 한계 문제, Section 5.4에서는 정교한 구조체 할당 한계 문제를 풀어봅니다. Section 5.5에서는 다양한 앞의 섹션에서 소개된 특정 병렬 카운터들을 어떻게 사용할지 생각해 봅니다. 마지막으로, Section 5.6에서는 성능 측정과 함께 이 챕터를 마무리 합니다.

Section 5.1와 Section 5.2는 소개적인 내용들을 담고 있고, 이외의 섹션들은 좀 더 전문적인 학생들에게 적당할 겁니다.

5.1 Why Isn't Concurrent Counting Trivial?

일단 간단한, 예를 들어, Figure 5.1 (`count_nonatomic.c`)에 나온 것과 같이 직관적인 케이스부터 보도록 하죠. 여기서, 우리는 라인 1에

카운터를 두고, 라인 5에서 그 값을 증가시키며, 라인 10에서 그 값을 읽어옵니다. 이보다 간단할 수는 없겠죠?

이 방법은 당신이 대부분의 경우 읽기만을 하고 값의 증가는 아주 가끔만 한다면 매우 빠르다는 장점을 가지고, 또한 작은 시스템에서도 훌륭한 성능을 보일 겁니다.

다만 여기에는 한가지 문제가 있습니다: 이 방법은 카운트를 놓칠 수 있습니다. 제 듀얼 코어 랩탑에서 짧은 시간동안 `inc_count()`를 100,014,000 번 수행했을 때, 카운터의 마지막 값은 52,909,118 이었습니다. 컴퓨팅에서는 대략적인 값도 충분한 경우도 있지만 그렇다 해도 50% 이상의 정밀도는 항상 필요합니다.

Quick Quiz 5.6: 하지만 `++` 연산자는 x86의 add-to-memory 명령어를 만들지 않나요? 그리고 CPU 캐시는 그걸 어토믹하게 수행하지 않나요? ■

Quick Quiz 5.7: 실패 횟수의 8-figure 정확도는 당신이 진짜로 이 테스트를 한 것을 보여주는군요. 왜 이런 사소한 프로그램을, 특히나 버그가 이렇게 쉽게 직관적으로 보이는데 굳이 테스트 해야 하나요? ■

정확하게 카운트를 하는 직관적 방법은 Figure 5.2 (`count_atomic.c`)에 나온 것처럼 어토믹 오퍼레이션을 사용하는 것입니다. 라인 1은 어토믹 변수를 정의하고, 라인 5에서 어토믹하게 값을 증가시키고, 라인 10에서 읽어냅니다. 이건 어토믹하기 때문에, 완벽한 카운트를 유지합니다. 하지만, 느립니다: 인텔 Core Duo 랩탑에서 이것은 어토믹하지 않은 방법에 비해 싱글쓰레드에서 여섯배, 그리고 두 쓰레드를 사용했을 때엔 열배 느립니다.¹

Chapter 3에서 이야기했던 걸 떠올려 보면 성능이 느린 것도, Figure 5.3에서 보여지듯 어토믹 증가 연산의 성능이 CPU와 쓰레드의 숫자가 증가할수록 느려지는

¹ 흥미롭게도, 어토믹하지 않게 카운터를 증가시키는 두개의 쓰레드는 어토믹하게 카운터를 증가시키는 두개의 쓰레드보다 더 빨리 그 값을 증가시킵니다. 물론, 당신의 목표가 단순히 카운터를 더 빨리 증가시키는 거라면, 그냥 카운터에 큰 값을 넣으면 되겠습니다. 분명한 건, 거대한 성능과 확장성을 위해선 정확성을 주의깊게 완화한 알고리즘의 역할도 있을 수 있다는 것입니다 [And91, ACMS03, Ung11].

```

1 long counter = 0;
2
3 void inc_count(void)
4 {
5     counter++;
6 }
7
8 long read_count(void)
9 {
10    return counter;
11 }
```

Figure 5.1: Just Count!

```

1 atomic_t counter = ATOMIC_INIT(0);
2
3 void inc_count(void)
4 {
5     atomic_inc(&counter);
6 }
7
8 long read_count(void)
9 {
10    return atomic_read(&counter);
11 }
```

Figure 5.2: Just Count Atomically!

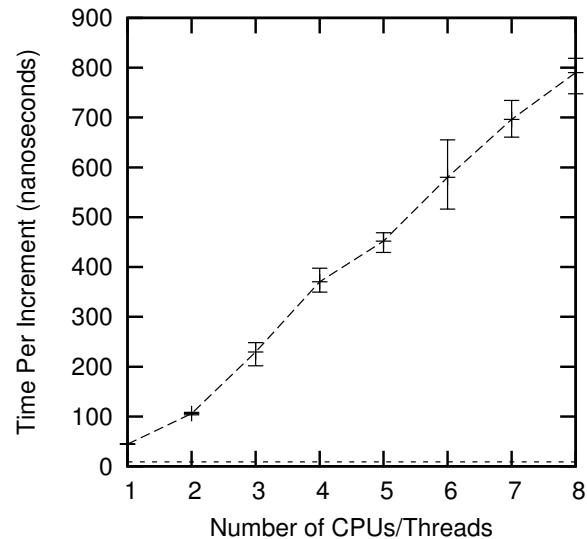


Figure 5.3: Atomic Increment Scalability on Nehalem

것도 그다지 놀라운 일은 아닙니다. 이 그림에서, x 축에 붙어있는 수평의 점선은 완벽하게 확장하는 알고리즘에 의해서만 얻어질 수 있는 이상적인 성능입니다: 그런 알고리즘이라면, 카운트 증가는 싱글쓰레드에서와 동일한 오버헤드만을 일으킬 것입니다. 하나의 전역 변수에 대한 어토믹 증가 연산은 분명하게 비이상적이며, CPU를 추가할수록 성능이 나빠집니다.

Quick Quiz 5.8: 왜 x 축의 점선은 $x = 1$ 에서 대각선의 선과 만나지 않죠? ■

Quick Quiz 5.9: 하지만 어토믹 증가 연산은 여전히 꽤 빠릅니다. 그리고 빠빠한 루프에서 하나의 변수를 증가시키는 건 제겐 꽤 비현실적인 것 같아 보이구요, 무엇보다, 프로그램의 실행은 실제로 일을 하는데 쓰여야지, 자기가 한 일을 세는데 쓰여야 하는게 아니라구요! 왜 제가 이걸 빠르게 하는걸 고민해야 하나요? ■

전역 어토믹 증가에 대한 다른 관점을 위해, Figure 5.4를 보세요. 각 CPU가 주어진 전역 변수를 증

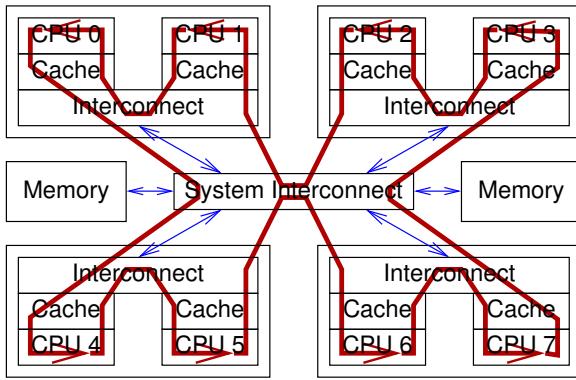


Figure 5.4: Data Flow For Global Atomic Increment

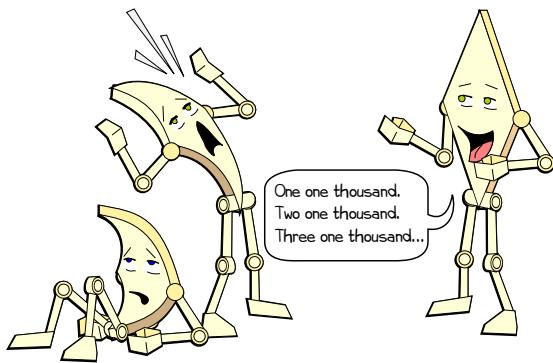


Figure 5.5: Waiting to Count

가시킬 기회를 얻기 위해서, 해당 변수를 가지고 있는 캐시 라인은 빨간 화살표로 표시된 것처럼 모든 CPU 사이를 순환해야 합니다. 이런 순환은 상당한 시간을 요할 것이고, 이는 Figure 5.5 와 같은 상황을 초래해서 Figure 5.3 에서 보여진 낮은 성능을 야기할 것입니다.

다음 섹션들에서는 이런 순환에서 발생하는 지연들을 없애고 성능 카운팅에 대해 알아봅니다.

Quick Quiz 5.10: 그런데 왜 CPU 설계자들은 단순히 데이터에의 증가 연산을 추가해서 담고 있는 캐시 라인의 순회가 증가하는걸 막지 않는거죠? ■

5.2 Statistical Counters

이 섹션은 카운트는 매우 가끔만 업데이트 되고 그 값은 웬만해서는 읽혀지지 않는, 통계적 카운터의 흔한 특수 케이스들을 다룹니다. 이것들은 Quick Quiz 5.2 에서 이

```

1 DEFINE_PER_THREAD(long, counter);
2
3 void inc_count(void)
4 {
5     __get_thread_var(counter)++;
6 }
7
8 long read_count(void)
9 {
10    int t;
11    long sum = 0;
12
13    for_each_thread(t)
14        sum += per_thread(counter, t);
15
16    return sum;
17 }
```

Figure 5.6: Array-Based Per-Thread Statistical Counters

야기한 네트워크 패킷 카운팅 문제를 푸는데 사용될 수 있을 것입니다.

5.2.1 Design

통계적 카운팅은 보통 쓰레드별로 (또는, 커널에서라면 CPU별로) 카운터를 제공해서 각 쓰레드가 자신의 카운터를 업데이트하도록 합니다. 이 카운터들의 전체 통합 값은 더하기의 상호성과 결합성에 기반해, 단순히 모든 쓰레드의 카운터의 값을 더하는 것으로 구해질 수 있습니다. 이건 Section 6.3.4 에서 소개되는 데이터 소유권 패턴의 한 예입니다.

Quick Quiz 5.11: 하지만 C 의 “정수들”은 크기와 관련한 복잡한 문제들이 있지 않나요? ■

5.2.2 Array-Based Implementation

쓰레드별 변수를 제공하는 방법 중 하나는 쓰레드별로 (아마도 false sharing 을 막기 위해 캐시 얼라인 되어 있고 패딩 되어있는)원소 하나를 갖는 배열을 만드는 겁니다.

Quick Quiz 5.12: 배열이요??? 하지만 그럼 쓰레드의 갯수가 제한되지 않나요? ■

그런 배열은 Figure 5.6 (count_stat.c) 에 보여진 것처럼 per-thread 기능을 사용할 수도 있습니다. 라인 1 은 long 타입의 쓰레드별 카운터들을 담는, counter 라는 이름의 배열을 만듭니다.

라인 3-6 은 이 카운터를 증가시키는 함수로, __get_thread_var() 함수로 현재 수행중인 쓰레드의 원소를 counter 배열에서 얻어옵니다. 이 원소는 연관된 쓰레드에 의해서만 수정되므로, 어토믹하지 않은 방식으로도 충분합니다.

라인 8-16 은 카운터의 총계 값을 얻어오는 함수로, for_each_thread() 를 이용해 현재 돌고 있는 쓰레드들의 리스트를 순회하면서 per_thread() 를 이

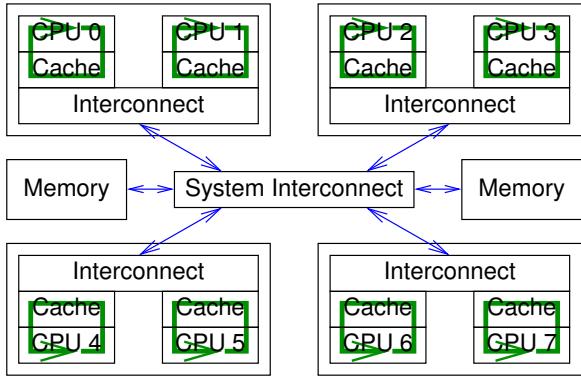


Figure 5.7: Data Flow For Per-Thread Increment

용해 특정 쓰레드의 카운터 값을 얻어옵니다. 하드웨어는 제대로 정렬되어 있는 `long` 변수는 어토믹하게 읽고 쓸 수 있기 때문에, 그리고 `gcc`는 이 기능을 잘 사용해주기 때문에, 평범한 읽기로 충분하고, 별다른 어토믹 오퍼레이션은 필요치 않습니다.

Quick Quiz 5.13: 근데, 그 외에 `gcc` 가 어떤 짓을 할 수 있죠???

Quick Quiz 5.14: Figure 5.6 의 쓰레드별 `counter` 변수는 어떻게 초기화 되나요?

Quick Quiz 5.15: Figure 5.6 의 코드가 어떻게 복수의 카운터를 가능하게 할 수 있죠?

이 방법은 `inc_count()` 를 호출하는 업데이터 쓰레드의 수가 늘어나더라도 선형적으로 성능이 확장됩니다. Figure 5.7 에 각 CPU 의 초록색 화살표로 보여지듯, 각 CPU 는 자신의 쓰레드의 변수를 시스템을 통과하는 비싼 통신 없이 빠르게 증가시킬 수 있기 때문입니다. 이와 같이, 이 섹션은 이 챕터의 시작에서 소개된 네트워크 패킷 카운팅 문제를 해결합니다.

Quick Quiz 5.16: 읽기 오퍼레이션은 쓰레드별 값을 모두 더하는 시간을 가져야 할 것이고, 그동안도 카운터는 값이 변할 수 있어요. 그럼 Figure 5.6 의 `read_count()` 는 정확하지 않다는 의미입니다. 이 카운터는 단위시간당 r 만큼 카운터 값을 증가하고 `read_count()` 는 Δ 단위시간을 소모한다고 해봅시다. 리턴되는 값의 예상 오류값은 얼마입니까?

하지만, 이 업데이트 쪽 확장성은 쓰레드의 수가 늘어날 경우 읽기 쪽에 큰 부담으로 다가옵니다. 다음 섹션에서는, 업데이트 쪽 확장성을 유지하면서 읽기 쪽 부담을 줄이는 방법을 알아봅니다.

5.2.3 Eventually Consistent Implementation

읽기 쪽 성능을 엄청나게 개선하면서 업데이트 쪽 확장성도 지키는 한가지 방법은 일관성에 대한 요구사항을 약화시키는 것입니다. 앞 섹션의 카운팅 알고리즘은 이상적인 카운터가 `read_count()` 실행 직전과 직후에 내놓을 값을 사이의 값을 내놓는 것을 보장했습니다. 최종적 일관성(Eventual consistency [Vog09]) 는 더 약한 보장사항을 제공합니다: `inc_count()` 호출이 없을 때라는 조건 하에, `read_count()` 는 최종적으로는 정확한 값을 리턴할 것입니다.

우리는 글로벌 카운터를 사용해 최종적 일관성을 제공할 것입니다. 하지만, 업데이트는 쓰레드별 카운터만을 사용합니다. 별도의 쓰레드가 존재하며 이 쓰레드가 업데이트 쓰레드들의 쓰레드별 카운터의 값을 세어 글로벌 카운터의 값을 만들어냅니다. 읽기는 단순히 글로벌 카운터의 값을 읽습니다. 업데이트가 수행 중이라면, 읽기 쓰레드가 읽어낸 값은 과거의 값을 것입니다만, 일단 업데이트가 중단되면 글로벌 카운터는 최종적으로 진짜 값을 가질 겁니다—그래서 이 방법이 최종적 일관성에 분류되는 겁니다.

Figure 5.8 (`count_stat_eventual.c`) 에 그 구현이 있습니다. 라인 1-2 는 카운터의 값을 갖는 쓰레드별 변수와 글로벌 변수를 보여주며, 세번째 라인은 프로그램을 정확한 카운터 값과 함께 종료하고자 할 경우를 위해 종료 조건을 알리는 `stopflag` 를 보입니다. 라인 5-8 의 `inc_count()` 함수는 Figure 5.6 것과 비슷합니다. 라인 10-13 의 `read_count()` 함수는 단순히 `global_count` 변수의 값을 리턴합니다.

하지만, 라인 34-42 의 `count_init()` 함수는 라인 15-32 의 `eventual()` 쓰레드를 생성하는데, 이 쓰레드는 모든 쓰레드를 순회하며 쓰레드별 `counter` 를 더해서 `global_count` 변수에 저장하는 일을 반복합니다. `eventual()` 쓰레드는 일의 반복 사이에 적당히 선택된 1 밀리세컨드 동안 기다립니다. 라인 44-50 의 `count_cleanup()` 함수는 프로그램 종료를 관리합니다.

이 방법은 여전히 선형적인 카운터 업데이트 성능을 유지하면서 극단적으로 빠른 카운터 읽기 속도를 보입니다. 하지만, 이 훌륭한 읽기 성능과 업데이트 성능 확장성은 `eventual()` 이라는 추가된 쓰레드의 수행이라는 비용을 가집니다.

Quick Quiz 5.17: Figure 5.8 의 `inc_count()` 는 왜 어토믹 명령을 사용하지 않죠? 쓰레드별 카운터를 여러 쓰레드에서 접근하고 있잖아요!

Quick Quiz 5.18: Figure 5.8 의 단일 글로벌 쓰레드인 `eventual()` 함수는 글로벌 락처럼 큰 병목이 되거나 하진 않나요?

Quick Quiz 5.19: Figure 5.8 의 `read_count()` 에

```

1 long __thread counter = 0;
2 long *counterp[NR_THREADS] = { NULL };
3 long finalcount = 0;
4 DEFINE_SPINLOCK(final_mutex);
5
6 void inc_count(void)
7 {
8     counter++;
9 }
10
11 long read_count(void)
12 {
13     int t;
14     long sum;
15
16     spin_lock(&final_mutex);
17     sum = finalcount;
18     for_each_thread(t)
19         if (counterp[t] != NULL)
20             sum += *counterp[t];
21     spin_unlock(&final_mutex);
22     return sum;
23 }
24
25 void count_register_thread(void)
26 {
27     int idx = smp_thread_id();
28
29     spin_lock(&final_mutex);
30     counterp[idx] = &counter;
31     spin_unlock(&final_mutex);
32 }
33
34 void count_unregister_thread(int nthreadsexpected)
35 {
36     int idx = smp_thread_id();
37
38     spin_lock(&final_mutex);
39     finalcount += counter;
40     counterp[idx] = NULL;
41     spin_unlock(&final_mutex);
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 5.8: Array-Based Per-Thread Eventually Consistent Counters

Figure 5.9: Per-Thread Statistical Counters

서 리턴하는 추정값은 쓰레드의 갯수가 늘어날수록 부정확해져 가지 않을까요? ■

Quick Quiz 5.20: Figure 5.8 의 최종적 일관성 알고리즘은 읽기에도 쓰기에도 매우 적은 오버헤드와 극단적인 확장성을 보이는데, 과연 누가 Section 5.2.2 같이 읽기 쪽이 비싼 구현을 사용하겠습니까? ■

5.2.4 Per-Thread-Variable-Based Implementation

다행히도, gcc 는 쓰레드별 저장소인 `__thread` 스토리지 클래스를 제공합니다. 이 스토리지 클래스는 Figure 5.9(`count_end.c`)에 보여진 것처럼 확장성 있을 뿐만 아니라, 간단하고 어토믹하지 않은 증가 방법에 비해 카운트를 증가시키는 쪽에 거의 성능 페널티를 주지 않는 통계적 카운터를 구현하는데 사용될 수 있습니다.

라인 1-4 는 필요한 변수들을 정의합니다: `counter`

는 쓰레드별 카운터 변수이고, `counterp[]` 배열은 쓰레드들이 다른 쓰레드의 카운터들을 볼 수 있게 하며, `finalcount`는 개별 쓰레드가 끝날 때마다 전체 카운트를 계산해 가지고 있으며, `final_mutex`는 쓰레드들 사이에서 전체 카운트를 구할 때와 쓰레드 종료 시에 크리티컬 섹션을 위해 사용됩니다.

Quick Quiz 5.21: 다른 쓰레드의 카운터를 찾는데 왜 별개의 배열이 필요하죠? 왜 gcc는 리눅스 커널의 `per_cpu()` 가 쓰레드들이 다른 쓰레드의 쓰레드별 변수를 쉽게 접근할 수 있도록 하는 것처럼 `per_thread()` 같은 인터페이스를 제공하지 않나요? ■

업데이터에 의해 사용되는 `inc_count()` 함수는 라인 6-9에서 볼 수 있듯이 상당히 간단합니다.

리더에 의해 사용되는 `read_count()` 함수는 약간 복잡합니다. 라인 16에서 종료되는 쓰레드들을 배제하기 위해 락을 획득하고, 라인 21에서는 락을 풁니다. 라인 17에서는 카운트의 합을 이미 종료한 쓰레드들에 의해 계산된 카운트 값으로 초기화하고, 라인 18-20에서 현재 동작 중인 쓰레드들에서 얻어온 카운트 값을 더합니다. 마지막으로, 라인 22에서 그 합을 리턴합니다.

Quick Quiz 5.22: Figure 5.9 의 라인 19에서의 NULL 체크는 브랜치 예측 실패를 가져오지 않나요? 항상 0인 변수 집합을 두고 더이상 사용되지 않는 카운터로의 포인터를 NULL로 만드는 대신 그 변수로 향하게 하는게 어떤가요? ■

Quick Quiz 5.23: 도대체 왜 Figure 5.9 의 `read_count()` 함수의 합을 계산하는 곳에서 무거운 `lock` 을 사용하는거죠? ■

라인 25-32는 `count_register_thread()` 함수를 보여주는데, 각 쓰레드는 자신의 카운터를 처음 사용하기 전에 이 함수를 반드시 호출해야 합니다. 이 함수는 단순히 해당 쓰레드를 위한 `counterp[]` 배열의 원소가 자신의 쓰레드별 `counter` 변수를 가리키도록 만듭니다.

Quick Quiz 5.24: 대체 왜 Figure 5.9 의 `count_register_thread()` 함수에서 락을 잡아야 하는거죠? 여기서 사용하는건 다른 쓰레드가 건들지 않는, 제대로 정렬된 기계의 워드 스토어 사이즈 데이터이니 어토믹할 거잖아요, 아닌가요? ■

라인 34-42는 `count_unregister_thread()` 함수를 보아는데, 각 쓰레드는 종료되기 직전에 반드시 이 함수를 호출해야 합니다. 라인 38에서는 락을 잡고, 라인 41에서 푸는데, 이로써 `read_count()` 호출과 다른 `count_unregister_thread()` 호출을 배제 시킵니다. 라인 39에서는 이 쓰레드의 `counter`를 글로벌 `finalcount`에 더하고, 그 후 라인 40에서 자신의 `counterp[]` 배열에서의 원소를 NULL로 만듭니다. 이후의 `read_count()` 호출은 종료된 쓰레드의

카운트는 글로벌 변수 `finalcount`에서 볼 수 있을 것이고, 종료된 쓰레드의 카운터는 `counterp[]` 배열을 통해 무시할 수 있으므로 올바른 전체 값을 얻을 수 있을 것입니다.

이 방법은 업데이터들을 어토믹하지 않은 더하기와 거의 똑같은 성능을 주면서도 선형적 확장성을 갖습니다. 반면, 동시적으로 수행되는 읽기 동작은 하나의 글로벌 락을 두고 경쟁하므로 성능과 확장성 모두 최악으로 떨어집니다. 하지만, 카운트의 증가가 주로 일어나고 읽기는 잘 발생하지 않는 통계적 카운터에선 문제가 되지 않습니다. 물론, 이 방법은 배열 기반의 방법보다는 쓰레드 종료 시 쓰레드별 변수의 증발 문제 처리로 인해 좀 더 복잡하긴 합니다.

Quick Quiz 5.25: 좋아요, 하지만 리눅스 커널은 CPU 별 카운터의 값을 합칠 때 락을 잡지 않아요. 유저 스페이스 코드에선 왜 이게 필요한거죠??? ■

5.2.5 Discussion

이 세개의 구현들은 병렬 머신에서 돌아감에도 통계적 카운터에 유니프로세서에서의 성능을 얻는게 가능함을 보여줍니다.

Quick Quiz 5.26: 패킷의 사이즈가 다양하다면 패킷의 갯수를 세는 것과 패킷의 전체 바이트 수를 세는 것에 어떤 기본적 차이가 있나요? ■

Quick Quiz 5.27: 리더는 쓰레드들의 카운터를 모두 더해야 하므로, 쓰레드의 갯수가 늘어나면 더 많은 시간을 쓰게 될겁니다. 리더에게도 쓸만한 성능과 확장성을 주면서 쓰기 작업도 여전히 빠르고 확장성 있게 하는 방법은 없을까요? ■

이 섹션에서 이야기된 것을 바탕으로, 독자 여러분은 이제 이 챕터의 시작에 있던 네트워킹에 대한 통계적 카운터에 관한 Quick Quiz에 답변을 할 수 있을 겁니다.

5.3 Approximate Limit Counters

카운팅에 관계된 또 다른 특수 케이스는 한계 체크입니다. 예를 들면, Quick Quiz 5.3의 대략적 구조체 할당 한계 문제처럼, 할당된 구조체의 갯수를 세고 있다가 할당된 구조체의 갯수가 특정 한계치, 대충 10,000 정도를 넘어가면 이어지는 할당을 실패 처리해야 하는 경우를 생각해 봅시다. 이 구조체들은 또한 잠시동안만 사용되어서, 한계치는 가끔만 초과되며, 이 한계치는 대략적 이어서 어느정도까지는 넘어가도 되는 경우를 생각해 봅시다 (한계치가 정확해야 한다면 Section ?? 를 참고하세요).

5.3.1 Design

한계 카운터를 위해 사용할 수 있을 설계 중 하나는 한계치 10,000 을 쓰레드의 수로 나눠서 각 쓰레드에게 크기가 고정된 구조체 풀을 주는 것입니다. 예를 들어, 100 개의 쓰레드가 있다면, 각 쓰레드가 100개의 구조체를 갖는 풀을 알아서 관리하게 하는 것입니다. 이 방법은 간단하고, 일부 경우에는 잘 동작합니다만, 구조체를 할당하는 쓰레드와 해제하는 쓰레드가 다른 [MS93] 많은 경우에는 동작하지 못합니다. 만약 한 쓰레드가 자신이 해제하는 모든 구조체에 대해 관리 책임을 갖는다면, 대부분의 해제를 하는 쓰레드가 구조체 해제 처리를 하느라 바쁜 동안 대부분의 할당을 하는 쓰레드는 풀의 구조체가 바닥나게 할겁니다. 한편으로는, 구조체를 할당한 CPU에게 해제된 구조체에 대한 관리 책임이 있다면, CPU들은 다른 CPU들을 모두 돌아봐야 할거고, 이는 비싼 어토믹 명령이나 다른 쓰레드간 통신 비용을 발생시킬 것입니다.²

짧게 말해서, 완전히 분할된 카운터는 많은 중요 워크로드에서 사용할 수 없습니다. 카운터를 분할하는 방법은 Section 5.2에서 이야기한 세가지 방법에서 훌륭한 업데이트 쪽 성능을 가져온 방법이었으나, 비관적으로 보일 수도 있겠습니다. 하지만, Section 5.2.3에서 보여진 최종적 일관성 (eventually consistent) 알고리즘이 흥미로운 힌트를 제공합니다. 이 알고리즘은 업데이트를 위한 쓰레드별 counter 변수와 읽기를 위한 global_count 변수 두개의 저장소를 유지하고, 주기적으로 global_count 를 업데이트해서 최종적으로는 쓰레드별 counter 의 값과 global_count 를 일관되게 만들어주는 eventual() 쓰레드를 운영했습니다. 쓰레드별 counter 는 카운터 값을 완벽하게 분할했고 global_count 는 전체 값을 유지했습니다.

한계 카운터를 위해 우리는 이 방법의 변종을 사용할 수 있는데, 카운터를 부분적으로 분할하는 것입니다. 예를 들어, 네개의 쓰레드가 각각 쓰레드별 counter 를 가질 수 있지만, 각각은 또한 쓰레드별 최대값 (countermax 라고 해봅시다) 을 가지는 것입니다.

그렇게 되면 한 쓰레드가 자신의 counter 를 증가시키는데, counter 는 자신의 countermax 와 동일하면 어떻게 될까요? 그 쓰레드의 counter 값의 절반을 globalcount 로 옮기고, counter 를 증가시키는 트릭을 씁니다. 예를 들어, 한 쓰레드의 counter 와 countermax 변수가 똑같이 10이라면, 다음의 일을 합니다:

1. 글로벌 락을 잡는다.

² 그렇지만, 만약 각 구조체가 항상 같은 CPU (또는 쓰레드) 에 의해 해제된다면, 이 간단한 풀개기 전략은 매우 잘 동작합니다.

2. globalcount 에 5를 더한다.
3. 앞의 합과 균형을 맞추기 위해, 이 쓰레드의 counter 에서 5를 뺀다.
4. 글로벌 락을 놓는다.
5. 이 쓰레드의 counter 를 증가시켜서 6으로 만든다.

이 방법은 여전히 글로벌 락을 필요로 하지만, 그 락은 다섯번의 증가 오퍼레이션마다 한번씩만 사용되므로, 락 경쟁을 상당히 줄여줄 겁니다. 이 경쟁정도는 countermax 값을 올려줌으로써 필요한 만큼까지 낮춰줄 수 있습니다. 하지만, countermax 를 늘리는데 대한 페널티가 존재하는데 globalcount 의 정확도가 떨어진다는 점입니다. 예를 들어 4개 CPU 가 있는 시스템에서 countermax 는 10이라 하면 globalcount 는 최대 40 의 범위값을 가질 수 있습니다. 반면, countermax 가 100 까지 늘어나면, globalcount 는 최대 400 의 범위값을 가질 수 있습니다.

이는 globalcount 와 globalcount 와 각 쓰레드의 counter 변수의 값의 합으로 만들어지는 값 사이의 차이를 얼마나 많이 신경써야 하나 하는 질문을 가져옵니다. 답은 얼마나 합으로 만들어진 값이 카운터의 리미트 (globalcountmax 라고 해두죠) 와 차이가 나나에 달려 있습니다. 이 두 값 사이의 차이가 크면 클수록 countermax 는 globalcountmax 리미트를 넘지는 않는다는 한계 내에서 커져도 괜찮습니다. 이 말은 주어진 쓰레드의 countermax 변수는 이 차이에 맞춰 설정될 수 있다는 겁니다. 리미트가 한참 남았다면, countermax 쓰레드별 변수는 성능과 확장성을 위한 최적화를 위해 큰 값을 가질 수 있고, 반대로 리미트가 가깝다면, 이 값들은 globalcountmax 리미트에의 체크의 예리를 줄이기 위해 작은 값을 설정되어야 합니다.

이 디자인은 일반적인 경우는 쓰레드간의 상호작용과 비싼 동작 없이 수행하되 결국 사용해야만 할 때에는 보수적으로 설계 된 (그리고 오버헤드가 큰) 알고리즘을 사용하는 유명한 디자인 패턴인 병렬 패스트패스 (parallel fastpath) 의 한 예입니다. 이 디자인 패턴은 Section 6.4에서 더 자세히 다룹니다.

5.3.2 Simple Limit Counter Implementation

Figure 5.10 는 이 구현에서 사용되는 쓰레드별, 그리고 글로벌 변수를 보여줍니다. 쓰레드별 counter 와

```

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);

```

Figure 5.10: Simple Limit Counter Variables

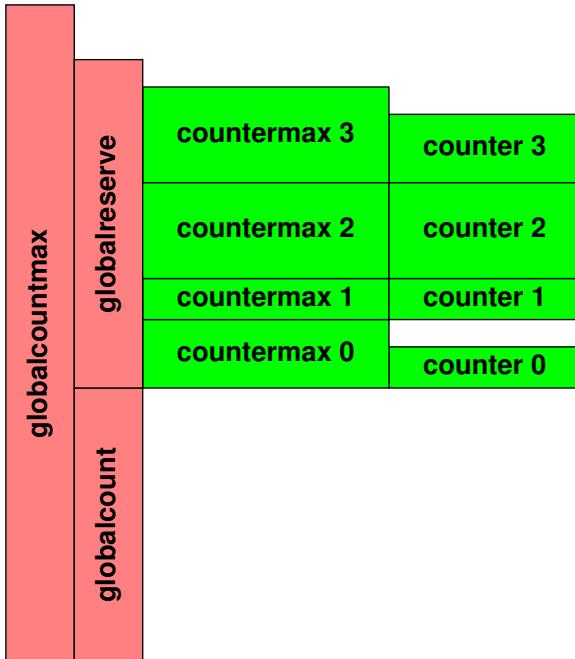


Figure 5.11: Simple Limit Counter Variable Relationships

countermax 변수들은 연관되는 쓰레드의 로컬 카운터와 그 카운터의 최대 허용값을 각각 나타냅니다. 라인 3의 globalcountmax 변수는 함께 카운터의 최대값을 가지며, 라인 4의 globalcount 변수는 글로벌 카운터입니다. globalcount 와 각 쓰레드의 counter 의 합은 전체 카운터의 합산 값을 갖습니다. 라인 5의 globalreserve 변수는 모든 쓰레드별 countermax 변수의 값의 합입니다. 이 변수들 간의 관계가 Figure 5.11에 나타나 있습니다:

1. globalcount 와 globalreserve 의 합은 globalcountmax 보다 작거나 같아야만 합니다.
2. 모든 쓰레드의 countermax 값의 합은 globalreserve 보다 작거나 같아야만 합니다.

```

1 int add_count(unsigned long delta)
2 {
3     if (countermax - counter >= delta) {
4         counter += delta;
5         return 1;
6     }
7     spin_lock(&gblcnt_mutex);
8     globalize_count();
9     if (globalcountmax -
10         globalcount - globalreserve < delta) {
11         spin_unlock(&gblcnt_mutex);
12         return 0;
13     }
14     globalcount += delta;
15     balance_count();
16     spin_unlock(&gblcnt_mutex);
17     return 1;
18 }
19
20 int sub_count(unsigned long delta)
21 {
22     if (counter >= delta) {
23         counter -= delta;
24         return 1;
25     }
26     spin_lock(&gblcnt_mutex);
27     globalize_count();
28     if (globalcount < delta) {
29         spin_unlock(&gblcnt_mutex);
30         return 0;
31     }
32     globalcount -= delta;
33     balance_count();
34     spin_unlock(&gblcnt_mutex);
35     return 1;
36 }
37
38 unsigned long read_count(void)
39 {
40     int t;
41     unsigned long sum;
42
43     spin_lock(&gblcnt_mutex);
44     sum = globalcount;
45     for_each_thread(t)
46         if (counterp[t] != NULL)
47             sum += *counterp[t];
48     spin_unlock(&gblcnt_mutex);
49     return sum;
50 }

```

Figure 5.12: Simple Limit Counter Add, Subtract, and Read

3. 각 쓰레드의 counter 는 해당 쓰레드의 countermax 보다 작거나 같아야만 합니다.

counterp[] 배열의 각 원소는 상용하는 쓰레드의 counter 변수를 가리키며, 마지막으로, gblcnt_mutex 스핀락은 모든 글로벌 변수들을 보호하는데, 달리 말하자면 어떤 쓰레드도 gblcnt_mutex 를 잡지 못했다면 어떤 글로벌 변수들도 접근하거나 수정하지 못합니다.

Figure 5.12 (count_lim.c)에서 add_count(), sub_count(), and read_count() 함수들을 보이고 있습니다.

Quick Quiz 5.28: 어째서 Figure 5.12 는 Section 5.2에서 나왔던 inc_count() 와 dec_count() 인터페이스 대신에 add_count() 와 sub_count() 를 제공하나요? ■

라인 1-18 은 add_count() 를 보여주는데, 이 함수는 특정값 delta 를 카운터에 더합니다. 라인 3 에서는 delta 를 위한 공간이 이 쓰레드의 counter 에 남아 있는지 확인하고, 만약 그렇다면 라인 4 에서 그 값을 더하고 라인 6 에서 성공했음을 리턴합니다. 이게 add_counter() 의 가장 빠른 수행경로로, 어토믹 오퍼레이션을 하지 않으며, 쓰레드별 변수만 참조하므로 어떤 캐시 미스도 만들지 않을 겁니다.

Quick Quiz 5.29: Figure 5.12 라인 3 의 저 이상한 조건문은 뭔가요? 왜 다음과 같이 더 직관적인 형태의 빠른 수행경로를 사용하지 않는거죠?

```
3 if (counter + delta <= countermax) {
4     counter += delta;
5     return 1;
6 }
```

라인 3 에서의 테스트가 실패한다면, 글로벌 변수들에 접근해야 하므로, 라인 7 의 gblcnt_mutex 를 반드시 잡아야 하고, 이 락은 실패하는 경우 라인 11 에서, 성공하는 경우는 라인 16 에서 해제합니다. 라인 8 에서는 Figure 5.13 의 globalize_count() 를 호출하는데, 이 함수는 쓰레드 로컬 변수들을 초기화 하고, 글로벌 변수들을 필요한 대로 맞춰줘서 전역적인 처리를 단순화 시킵니다. (하지만 제 말을 믿지만 말고, 직접 코딩해보세요!) 라인 9 와 10 에서는 delta 를 더하는 것이 합법적인지 Figure 5.11 에서 가장 왼쪽 두 빨간 막대의 높이 차이로 나타난 크기 규칙이 지켜지는지로 체크합니다. delta 의 합이 이루어져선 안된다면, 라인 11 은 (앞서 이야기했듯) gblcnt_mutex 를 풀고 라인 12 에서 실패를 알리는 리턴을 합니다.

만약 아니라면, 느린 수행경로를 실행하게 됩니다. 라인 14 는 delta 를 globalcount 에 더하고, 글로벌 변수들과 쓰레드별 변수들 모두 업데이트 하기 위해 라인 15 에서 (Figure 5.13 에서 나왔던) balance_count() 를 호출합니다. 이 balance_count() 호출은 대부분의 경우 이 쓰레드가 다음엔 빠른 수행경로로 수행되도록 countermax 를 조정할 것입니다. 그러고나서 라인 16 에서 gblcnt_mutex 를 (역시 앞에서 언급했듯) 풀어주고, 마지막으로 라인 17 에서 성공을 의미하는 리턴을 합니다.

Quick Quiz 5.30: Figure 5.12 에서 왜 globalize_count() 는 나중에 balance_count() 가 쓰레드별 변수를 다시 채우도록 쓰레드별 변수를 0으로 바꾸나요? 왜 그냥 쓰레드별 변수를 0이 아닌채로 놔두질 않는거죠? ■

라인 20-36 은 sub_count() 함수를 보여주는데, 이 함수는 delta 만큼을 카운터에서 제거합니다. 라인 22 는 쓰레드별 카운터가 이 빼기를 해도 되는 값인지 체크하고, 만약 그렇다면 라인 23 에서 그 빼기를 수행한 후 라인 24 에서 성공을 리턴합니다. 이 부분들이 add_count() 에서의 그것과 같은 sub_count() 의 빠른 수행경로로, 이 경로에서는 비용이 심한 동작은 하지 않습니다.

빠른 수행경로에서 delta 의 빼기가 적절치 못함으로 판단된다면, 실행은 라인 26-35 의 느린 수행경로로 이어집니다. 느린 실행 경로는 글로벌 상태에 접근해야 하기 때문에, 라인 26 에서 gblcnt_mutex 를 잡고, 이 락은 (실패의 경우) 라인 29 에서, 또는 (성공의 경우) 라인 34 에서 해제됩니다. 라인 27 은 Figure 5.13 의 globalize_count() 를 실행하는데, 앞서 설명했듯 쓰레드 로컬 변수의 값을 지우고, 글로벌 변수들을 적당한 값으로 설정합니다. 라인 28 은 카운터가 delta 의 빼기를 해도 좋을지 보고, 좋지 않다면 라인 29 에서 gblcnt_mutex 를 (앞서 설명했듯) 해제하고 라인 30 에서 실패를 리턴합니다.

Quick Quiz 5.31: Figure 5.12 에서 globalreserve 는 add_count() 에서 값이 구해지는데, 왜 sub_count() 에서 값을 구하지 않나요? ■

Quick Quiz 5.32: 한 쓰레드가 Figure 5.12 의 add_count() 를 호출하고, 다른 쓰레드가 sub_count() 를 호출한다고 해봅시다. sub_count() 는 카운터의 값이 0이 아님에도 실패하지 않겠습니까? ■

한편, 만약 라인 28 에서 카운터가 delta 빼기를 해도 적합하다고 판단되면, 느린 수행경로를 걷게 됩니다. 라인 32 에서 실제 빼기를 행하게 되고, 이후 라인 33 에서 (Figure 5.13 에 나온)balance_count() 를 호출해서 글로벌 변수들과 쓰레드별 변수들을 모두 (가능하면 다음엔 빠른 수행경로로 돌 수 있도록) 업데이트 합니다. 라인 34 에서는 gblcnt_mutex 를 해제하고, 라인 35 에서 성공을 리턴합니다.

Quick Quiz 5.33: Figure 5.12 에서는 왜 add_count() 와 sub_count() 를 모두 가지고 있는 거죠? 그냥 add_count() 에 음수를 넘기면 되지 않나요? ■

라인 38-50 은 read_count() 를 보이는데, 이 함수는 카운터의 합산된 값을 리턴합니다. 먼저 라인 48 에서 해제하는 gblcnt_mutex 를 라인 43 에서 잡는데, 이 락으로 add_count() 와 sub_count() 의 글로벌한 동작들을 배제시키고, 또한 쓰레드 생성과 종료 역시 배제시킵니다. 라인 44 는 로컬 변수 sum 을 globalcount 의 값으로 초기화 하고, 라인 45-47 의 루프는 쓰레드별 counter 변수의 합을 구합니다. 그러고나서 라인 49 에서 합을 리턴합니다.

```

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
25     spin_lock(&gblcnt_mutex);
26     counterp[idx] = &counter;
27     spin_unlock(&gblcnt_mutex);
28 }
29
30 void count_unregister_thread(int nthreadsexpected)
31 {
32     int idx = smp_thread_id();
33
34     spin_lock(&gblcnt_mutex);
35     globalize_count();
36     counterp[idx] = NULL;
37     spin_unlock(&gblcnt_mutex);
38 }

```

Figure 5.13: Simple Limit Counter Utility Functions

Figure 5.13 에서는 Figure 5.12 에서 봤던 `add_count()`, `sub_count()`, and `read_count()`에서 사용하던 함수들을 보입니다.

라인 1-7 은 `globalize_count()` 함수인데, 이 함수는 현 쓰레드의 쓰레드별 카운터를 0으로 만들고, 글로벌 변수들을 적당한 값으로 만듭니다. 이 함수는 카운터의 합산값을 바꾸진 않지만 어떻게 카운터의 현재 값이 보여야 하는지를 바꾼다는 점이 중요합니다. 라인 3 은 쓰레드의 `counter` 변수를 `globalcount`에 더하고 라인 4에서 `counter`를 0으로 만듭니다. 비슷하게, 라인 5 는 쓰레드별 `countermax`를 `globalreserve`에서 빼고, 라인 6에서 `countermax`를 0으로 만듭니다. 이어서 설명될 `balance_count()` 함수와 이 함수를 함께 이해하려면 Figure 5.11를 참고하는게 도움이 될겁니다.

라인 9-19 는 `balance_count()` 함수를 보이는 데, 이 함수는 간단히 말해서 `globalize_count()`의 반대입니다. 이 함수가 하는일은 현 쓰레드의 `countermax` 변수를 카운터가 `globalcountmax` 한계치를 넘지 않는 한계에서 가장 큰 값을 갖도록 하는 것입니다. 현재 쓰레드의 `countermax` 변수를 바꾸는건 물론 `counter`, `globalcount` 그리고 `globalreserve` 를 Figure 5.11에서 보여진 규칙을 지키도록 적절하게 조정하는 작업을 필요로 합니다. 이렇게 함으로써, `balance_count()` 는 `add_count()` 와 `sub_count()` 의 오버헤드가 적은 빠른 수행경로의 수행을 최대화 합니다.

라인 11-13 은 `globalcount` 나 `globalreserve`로 다뤄지지 않은 `globalcountmax`에서 이 쓰레드의 자분을 계산하고, 계산된 값을 이 쓰레드의 `countermax`에 대입합니다. 라인 14 는 `globalreserve`를 올바르게 조정합니다. 라인 15는 이 쓰레드의 `counter`를 `countermax` 절반으로 만듭니다. 라인 16은 이 `counter` 값이 규칙을 깨지 않는 선에서 가능한 값인지 `globalcount`를 보고, 그렇지 않다면 라인 17에서 `counter`를 적절하게 감소시킵니다. 앞의 조건과 관계없이 마지막으로, 라인 18에서 `globalcount`를 적절하게 조정합니다.

Quick Quiz 5.34: Figure 5.13 의 라인 15 에서는 왜 `counter` 를 `countermax / 2` 로 만들죠? 그냥 `countermax` 값을 가져오는게 더 간단하지 않나요? ■

Figure 5.14에 그린 것처럼 `globalize_count()` 와 `balance_count`에 의해서 카운터 값들이 어떻게 바뀌는지 개요를 그려보는 게 도움이 될 겁니다. 시간은 왼쪽에서 오른쪽으로 흐르고, 가장 왼쪽의 구성은 대략 Figure 5.11 의 그것입니다. 중간의 구성은 `globalize_count()` 가 쓰레드 0에 의해 수행되고 난 후의 각 카운터들의 관계입니다. 그럼에서 볼 수 있듯, 쓰레드 0의 `counter` (그림에서의 “c 0”)은

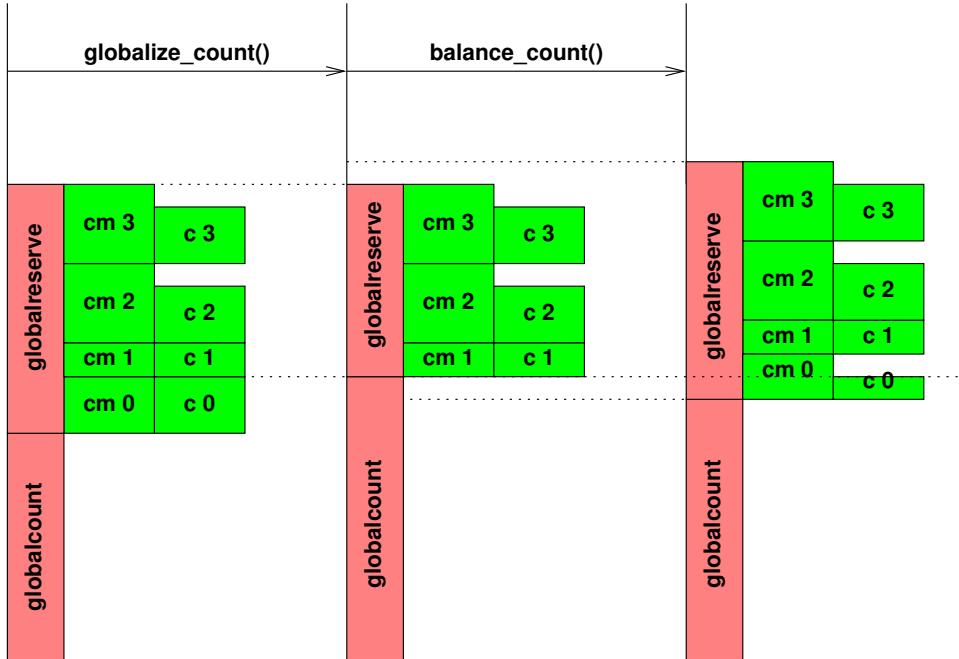


Figure 5.14: Schematic of Globalization and Balancing

는 globalcount 에 더해지고, globalreserve 는 그만큼 줄었습니다. 쓰레드 0 의 counter 와 countermax (그림에서의 “cm 0”)는 0이 되었습니다. 다른 세 쓰레드의 카운터들은 그대로입니다. 가장 왼쪽 구성과 중간 구성 사이의 바닥쪽 점선을 보면 알 수 있듯이 이 변화는 전체 카운터의 값을 바꾸지 않았음을 참고하십시오. 바꿔 말하자면, globalcount 와 네 쓰레드의 counter 변수들의 값의 합은 두 구성 모두에서 동일합니다. 비슷하게, 위쪽 점선을 보면 알 수 있듯이, 이 변경은 globalcount 와 globalreserve 의 합에도 어떤 영향을 끼치거나 하지 않았습니다.

가장 오른쪽의 구성은 역시 쓰레드 0 에 의해 balance_count() 가 수행된 이후의 카운터들의 관계를 보여줍니다. 다른 두개의 구성에 비해 올라간 수직의 선으로 보여지듯, 남은 카운트의 4분의 1이 쓰레드 0 의 countermax 에, 절반이 쓰레드 0 의 counter 에 더해졌습니다. 쓰레드 0 의 counter 에 더해진 양은 가운데와 오른쪽 구성의 가장 아래쪽의 두개의 점선으로 보여지듯이 전체 카운터의 값 (globalcount 와 세 쓰레드의 counter 변수의 값의 합) 을 바꾸지 않기 위해 globalcount 에서 빼졌습니다. globalreserve 변수 또한 이 변수가 네 쓰레드의 countermax 변수의 합과 동일하게 조정되었습니다. 쓰레드 0 의 counter 는 해당 쓰레드의

countermax 보다 작기 때문에, 쓰레드 0 은 또다시 자신의 카운터에서 곧바로 카운트 증가를 할 수 있습니다.

Quick Quiz 5.35: Figure 5.14 에서 보면, 가운데와 오른쪽 구성의 점선을 보면 알 수 있듯이, 한계까지 남은 카운트의 4분의 1이 쓰레드 0 에게 주어졌음에도, 8분의 1 만이 사용되었습니다. 왜 그런건가요? ■

라인 21-28 은 count_register_thread() 함수인데, 이 함수는 새로 생성된 쓰레드를 위한 상태를 만듭니다. 이 함수는 단순히 새로 생성된 쓰레드의 counter 변수로의 포인터를 상응하는 counterp[] 배열에서의 원소에 gblcnt_mutex 의 보호 아래 설정합니다.

마지막으로, 라인 30-38 은 count_unregister_thread() 함수를 보여주는데, 이 함수는 곧 종료될 쓰레드를 위해 상태를 정리합니다. 라인 34 에서는 라인 37 에서 해제될 gblcnt_mutex 를 잡습니다. 라인 35 에서는 globalize_count() 를 호출해서 이 쓰레드의 카운터 상태를 정리하고, 라인 36 에서 이 스레드의 counterp[] 배열에서의 상응하는 원소를 정리합니다.

```

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 5.15: Approximate Limit Counter Variables

```

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 5.16: Approximate Limit Counter Balancing

5.3.3 Simple Limit Counter Discussion

이 유형의 카운터는 `add_count()` 와 `sub_count()`의 빠른 수행경로에서의 비교와 브랜치로 인한 약간의 오버헤드가 있지만 합산값이 0에 가까울 때엔 매우 빠릅니다. 하지만, 쓰레드별 `coutnermax`의 사용으로 인해 카운터의 합산값은 `globalcountmax` 근처도 안갔는데 `add_count()` 가 실패할 수도 있습니다. 유사하게, `sub_count()` 는 합산값이 0 근처도 안갔는데 실패할 수 있습니다.

많은 경우에 이런 동작은 문제가 됩니다. 설령 `globalcountmax` 가 대략적인 한계치라고는 해도, 보통은 정확히 얼마나 대략적으로 조정되어야 하는지에 대한 한계는 존재합니다. 추정 정도를 제한하는 한 방법은 쓰레드별 `coutnermax` 인스턴스의 값에 최대값 한계를 내포시키는 것입니다. 이 방법이 다음 섹션에서 사용됩니다.

5.3.4 Approximate Limit Counter Implementation

이 구현 (`count_lim_app.c`)은 앞 섹션의 것 (Figures 5.10, 5.12, 5.13)들과 상당히 비슷하기 때문에, 여기에선 다른 부분만 이야기 합니다. Figure 5.15는 쓰레드별 `countermax`의 허용되는 최대값을 가리키는 `MAX_COUNTERMAX` 가 추가되었다는 점을 제외하고 Figure 5.10과 똑같습니다.

비슷하게, Figure 5.16는 라인 6과 7에서 쓰레드

별 `countermax` 변수에 `MAX_COUNTERMAX` 한계를 적용하는 내용이 추가된 걸 제외하고는 Figure 5.13의 `balance_count()` 함수와 동일합니다.

5.3.5 Approximate Limit Counter Discussion

이 변경들은 앞 섹션에서 보았던 한계치의 비정확성을 매우 줄여줍니다만, 다른 문제를 드러냅니다: `MAX_COUNTERMAX` 값은 워크로드에 따라 빠른 수행경로 접근 비율이 바뀝니다. 쓰레드의 수가 늘어나면, 빠른 수행경로 이외의 경로를 통하는 것은 성능과 확장성에 모두 문제가 될겁니다. 하지만, 이 문제는 일단 미뤄두고 정확한 한계치를 갖는 카운터에 대해 이야기해 봅시다.

5.4 Exact Limit Counters

Quick Quiz 5.4에서 이야기한 정확한 구조체 할당 한계 문제를 풀기 위해선 정확히 그 한계를 넘겼을 때 알려주는 리미트 카운터가 필요합니다. 그런 리미트 카운터를 구현하는 한가지 방법은 예약된 카운트를 가지고 있는 쓰레드들에게 그것을 포기하게 하는 것입니다. 이렇게 하는 방법 중 하나는 어토믹 인스트럭션들입니다. 물론, 어토믹 인스트럭션들은 빠른 수행경로를 느리게 만들겁니다만, 시도조차 해보지 않는것도 웃긴일일 겁니다.

5.4.1 Atomic Limit Counter Implementation

불행히도, 한 쓰레드가 다른 쓰레드의 카운트를 안전하게 없애려면, 두 쓰레드 모두 각 쓰레드의 `counter` 와 `countermax` 변수를 함께 어토믹하게 조정해야 할겁니다. 이걸 해결하는 일반적인 방법은, 예를 들면 32비트 변수가 있다면, 앞쪽 16비트는 `counter`를, 뒷쪽 16비트는 `countermax`를 나타내게 하는 식으로 두 변수를 한개의 변수로 합치는 것입니다.

Quick Quiz 5.36: 쓰레드의 `counter` 와 `countermax` 변수를 한번에 어토믹하게 수정해야 하는 이유가 뭐죠? 각 변수를 개별적으로 어토믹하게 수정해도 충분하지 않아요? ■

간단한 어토믹 리미트 카운터를 위한 변수와 액세스 함수들을 Figure 5.17 (`count_lim_atomic.c`)에 보였습니다. 앞의 알고리즘에서의 `counter` 와 `countermax` 변수들은 라인 1에 있는 하나의 변수 `ctrandmax`에 `counter`는 앞쪽 절반에, `countermax`는 뒷쪽 절반에 위치하도록 합쳐졌습니다. 이 변수는 `atomic_t` 타입으로, `int` 형의 값을 갖습니다.

```

1 atomic_t __thread ctrandmax = ATOMIC_INIT(0);
2 unsigned long globalcountmax = 10000;
3 unsigned long globalcount = 0;
4 unsigned long globalreserve = 0;
5 atomic_t *counterp[NR_THREADS] = { NULL };
6 DEFINE_SPINLOCK(gblcnt_mutex);
7 #define CM_BITS (sizeof	atomic_t) * 4
8 #define MAX_COUNTERMAX ((1 << CM_BITS) - 1)
9
10 static void
11 split_ctrandmax_int(int cami, int *c, int *cm)
12 {
13     *c = (cami >> CM_BITS) & MAX_COUNTERMAX;
14     *cm = cami & MAX_COUNTERMAX;
15 }
16
17 static void
18 split_ctrandmax(atomic_t *cam, int *old,
19                  int *c, int *cm)
20 {
21     unsigned int cami = atomic_read(cam);
22
23     *old = cami;
24     split_ctrandmax_int(cami, c, cm);
25 }
26
27 static int merge_ctrandmax(int c, int cm)
28 {
29     unsigned int cami;
30
31     cami = (c << CM_BITS) | cm;
32     return ((int)cami);
33 }

```

Figure 5.17: Atomic Limit Counter Variables and Access Functions

라인 2-6 은 globalcountmax, globalcount, globalreserve, counterp, 그리고 gblcnt_mutex 정의를 보이는데, 이들은 Figure 5.15 에 있는 것들과 비슷한 역할을 합니다. 라인 7 은 ctrandmax 의 각 절반에 사용되는 비트의 수를 나타내는 CM_BITS를 정의하고, 라인 8 에서는 ctrandmax 의 각 절반이 가질 수 있는 최대값을 나타내는 MAX_COUNTERMAX 를 정의합니다.

Quick Quiz 5.37: Figure 5.17 의 라인 7 에서는 C 표준을 어기는거 아닌가요? ■

라인 10-15 는 split_ctrandmax_int() 함수를 보여주는데, 이 함수는 atomic_t ctrandmax 변수로부터 얻어온 int 값을 받아서, counter (c) 와 countermax (cm) 으로 쪼갭니다. 라인 13 은 이 int 로부터 앞쪽 절반을 빼어내어 인자로 받은 c 에 넣고, 라인 14 에서는 뒷쪽 절반을 빼어내어 인자로 받은 cm 에 넣습니다.

라인 17-25 는 split_ctrandmax() 함수를 보여주는데, 이 함수는 라인 21 에서 주어진 변수로부터 int 값을 읽어내고, 라인 23 에서 그 값을 인자로 받은 old 에 저장하고, 라인 24 에서 split_ctrandmax_int() 를 호출해 쪼개는 작업을 합니다.

Quick Quiz 5.38: ctrandmax 변수는 하나 뿐인데, Figure 5.17 의 라인 18 에서는 왜 굳이 포인터로 받는거죠? ■

라인 27-33 은 merge_ctrandmax() 함수인데, 이 함수는 split_ctrandmax() 함수의 반대의 일을 하는 것으로 볼 수 있습니다. 라인 31 에서 c 와 cm 으로 받은 counter 와 countermax 값을 합쳐서 그 결과 값을 리턴합니다.

Quick Quiz 5.39: Figure 5.17 의 merge_ctrandmax() 는 왜 바로 atomic_t 에 값을 저장하지 않고 int 값을 리턴하는 거죠? ■

Figure 5.18 는 add_count(), sub_count(), 그리고 read_count() 함수를 보입니다.

라인 1-32 는 add_count() 함수를 보이는데, 이 함수는 라인 8-15 에 빠른 수행 경로가, 나머지에는 느린 수행경로가 있습니다. 빠른 수행경로의 라인 8-14는 라인 13-14 에서 실제 CAS 를 atomic_cxmpxchg() 함수로 수행하는 compare-and-swap (CAS) 루프를 구성합니다. 라인 9 는 현재 쓰레드의 ctrandmax 변수를 counter (c 에) 와 countermax (cm 에) 으로 쪼개고, 기존의 ctrandmax 가 갖는 int 값을 old 에 넣어 둡니다. 라인 10 에서는 delta 값이 로컬하게 처리 가능할지 (인티저 오버플로우를 막기 위한 처리를 하면서) 체크하고, 만약 그렇지 않다면 라인 11 의 느린 수행경로를 밟게 됩니다. 로컬하게 처리 가능하다면, 라인 12 에서 업데이트된 counter 값을 원본 countermax 값과 조합해서 new 를 만듭니다. 라

```

1 int add_count(unsigned long delta)
2 {
3     int c;
4     int cm;
5     int old;
6     int new;
7
8     do {
9         split_ctrandmax(&ctrandmax, &old, &c, &cm);
10        if (delta > MAX_COUNTERMAX || c + delta > cm)
11            goto slowpath;
12        new = merge_ctrandmax(c + delta, cm);
13    } while (atomic_cmpxchg(&ctrandmax,
14                           old, new) != old);
15    return 1;
16 slowpath:
17     spin_lock(&gblcnt_mutex);
18     globalize_count();
19     if (globalcountmax - globalcount -
20         globalreserve < delta) {
21         flush_local_count();
22         if (globalcountmax - globalcount -
23             globalreserve < delta) {
24             spin_unlock(&gblcnt_mutex);
25             return 0;
26         }
27     }
28     globalcount += delta;
29     balance_count();
30     spin_unlock(&gblcnt_mutex);
31     return 1;
32 }
33
34 int sub_count(unsigned long delta)
35 {
36     int c;
37     int cm;
38     int old;
39     int new;
40
41     do {
42         split_ctrandmax(&ctrandmax, &old, &c, &cm);
43         if (delta > c)
44             goto slowpath;
45         new = merge_ctrandmax(c - delta, cm);
46     } while (atomic_cmpxchg(&ctrandmax,
47                           old, new) != old);
48     return 1;
49 slowpath:
50     spin_lock(&gblcnt_mutex);
51     globalize_count();
52     if (globalcount < delta) {
53         flush_local_count();
54         if (globalcount < delta) {
55             spin_unlock(&gblcnt_mutex);
56             return 0;
57         }
58     }
59     globalcount -= delta;
60     balance_count();
61     spin_unlock(&gblcnt_mutex);
62     return 1;
63 }

```

Figure 5.18: Atomic Limit Counter Add and Subtract

인 13-14 에서는 `atomic_cmpxchg()` 함수를 이용해 이 쓰레드의 `ctrandmax` 변수를 `old` 와 어토믹하게 비교하고, 만약 비교 결과 같다면 그 값을 `new`로 어토믹하게 업데이트 합니다. 만약 비교 결과가 같다면 라인 15 에서 성공을 리턴하고, 그렇지 않았다면 라인 9 의 루프에서 실행을 계속합니다.

Quick Quiz 5.40: 우웩! Figure 5.18 라인 11 의 저더러운 `goto` 는 웬말이예요? `break` 몰라요???

Quick Quiz 5.41: Figure 5.18 의 라인 13-14 의 `atomic_cmpxchg()` 함수는 어떻게 실패할 수 있죠? 우린 이전 값을 라인 9 에서 가져오고 나서 바꾼 적 없잖아요!

Figure 5.18 의 라인 16-31 은 `add_count()` 의 느린 수행경로를 보이는데, 라인 17에서 얻어오고 라인 24 와 30에서 해제하는 `gblcnt_mutex`로 보호됩니다. 라인 18에서는 `globalize_count()` 를 호출하는데, 이 함수는 이 쓰레드의 상태를 글로벌 카운터로 옮겨줍니다. 라인 19-20은 `delta` 값이 현재의 글로벌 상태에 사용 가능한지 체크하고, 불가능하다면 라인 21에서 `flush_local_count()` 를 호출해서 모든 쓰레드의 로컬 스테이트를 글로벌 카운터로 몰아넣고, 라인 22-23에서 `delta` 가 이제는 사용 가능한지 다시 체크합니다. 이후에도 `delta` 더하기가 불가능하다면, 라인 24에서 `gblcnt_mutex`를 해제하고 라인 25에서 실패를 리턴합니다.

그렇지 않다면, 라인 28에서 `delta` 를 글로벌 카운터에 더하고, 라인 29에서 만약 적절하다면 카운트를 로컬 스테이트에 분산시킨 후, 라인 30에서 `gblcnt_mutex`를 해제하고, 마지막으로 라인 31에서 성공을 리턴합니다.

Figure 5.18 의 라인 34-63 에서는 `sub_count()` 를 보이는데, 이 함수는 `add_count()` 와 유사하게 구성되어 있어서, 라인 41-48에 빠른 수행경로를, 그리고 라인 49-62에 누린 수행경로를 갖습니다. 이 함수의 각 라인별 분석은 독자 여러분의 연습문제로 남겨두겠습니다.

Figure 5.19 는 `read_count()` 함수를 보입니다. 라인 9에서 `gblcnt_mutex` 를 잡고 라인 16에서 해제합니다. 라인 10에서는 로컬 변수 `sum` 을 `globalcount`의 값으로 초기화하고, 라인 11-15의 루프에서는 쓰레드별 카운터를 이 합에 더하고 라인 13에서 각 쓰레드별 카운터를 `split_ctrandmax` 를 사용해서 고립시킵니다. 마지막으로, 라인 17에서 아까 만든 합을 리턴합니다.

Figure 5.20 와 5.21 는 유ти리티처럼 사용되는 함수들인 `globalize_count()`, `flush_local_count()`, `balance_count()`, `count_register_thread()`, 그리고 `count_unregister_thread()` 의 코드를 보입니다.

```

1 unsigned long read_count(void)
2 {
3     int c;
4     int cm;
5     int old;
6     int t;
7     unsigned long sum;
8
9     spin_lock(&gblcnt_mutex);
10    sum = globalcount;
11    for_each_thread(t)
12        if (counterp[t] != NULL) {
13            split_ctrandmax(counterp[t], &old, &c, &cm);
14            sum += c;
15        }
16    spin_unlock(&gblcnt_mutex);
17    return sum;
18 }

```

Figure 5.19: Atomic Limit Counter Read

```

1 static void globalize_count(void)
2 {
3     int c;
4     int cm;
5     int old;
6
7     split_ctrandmax(&ctrandmax, &old, &c, &cm);
8     globalcount += c;
9     globalreserve -= cm;
10    old = merge_ctrandmax(0, 0);
11    atomic_set(&ctrandmax, old);
12 }
13
14 static void flush_local_count(void)
15 {
16     int c;
17     int cm;
18     int old;
19     int t;
20     int zero;
21
22     if (globalreserve == 0)
23         return;
24     zero = merge_ctrandmax(0, 0);
25     for_each_thread(t)
26         if (counterp[t] != NULL) {
27             old = atomic_xchg(counterp[t], zero);
28             split_ctrandmax_int(old, &c, &cm);
29             globalcount += c;
30             globalreserve -= cm;
31         }
32 }

```

Figure 5.20: Atomic Limit Counter Utility Functions 1

```

1 static void balance_count(void)
2 {
3     int c;
4     int cm;
5     int old;
6     unsigned long limit;
7
8     limit = globalcountmax - globalcount -
9         globalreserve;
10    limit /= num_online_threads();
11    if (limit > MAX_COUNTERMAX)
12        cm = MAX_COUNTERMAX;
13    else
14        cm = limit;
15    globalreserve += cm;
16    c = cm / 2;
17    if (c > globalcount)
18        c = globalcount;
19    globalcount -= c;
20    old = merge_ctrandmax(c, cm);
21    atomic_set(&ctrandmax, old);
22 }
23
24 void count_register_thread(void)
25 {
26     int idx = smp_thread_id();
27
28     spin_lock(&gblcnt_mutex);
29     counterp[idx] = &ctrandmax;
30     spin_unlock(&gblcnt_mutex);
31 }
32
33 void count_unregister_thread(int nthreadsexpected)
34 {
35     int idx = smp_thread_id();
36
37     spin_lock(&gblcnt_mutex);
38     globalize_count();
39     counterp[idx] = NULL;
40     spin_unlock(&gblcnt_mutex);
41 }

```

Figure 5.21: Atomic Limit Counter Utility Functions 2

`globalize_count()`의 코드는 Figure 5.20의 라인 1-12에 있는데 앞의 알고리즘과 거의 같지만 라인 7의, `counter`와 `countermax`를 `ctrandmax`에서 분할해 내기 위한 코드의 추가가 차이점입니다.

모든 쓰레드의 로컬 카운터 상태를 글로벌 카운터로 옮기는, `flush_local_count()`의 코드는 라인 14-32에 있습니다. 라인 22에서는 `globalreserve`의 값이 모든 쓰레드별 카운트를 허용하는지 체크하고 아니라면 라인 23에서 리턴합니다. 그렇지 않다면, 라인 24에서 로컬 변수 `zero`를 `counter`와 `countermax` 조합된 0으로 만듭니다. 라인 25-31의 루프에서는 각 쓰레드를 순회합니다. 라인 26에서 현재 쓰레드가 카운터 상태를 가지고 있는지 보고, 만약 그렇다면 라인 27-30에서 그 상태를 글로벌 카운터로 옮깁니다. 라인 27에서는 현재 쓰레드의 상태를 어도미하게 0으로 바꿈과 동시에 얻어옵니다. 라인 28에서는 이 상태를 자신의 `counter` (로컬 변수 `c`로)와 `countermax` (로컬 변수 `cm`으로)로 분할합니다. 라인 29에서 이 쓰레드의 `counter`를 `globalcount`에 더하고 라인 30에서 이 쓰레드의 `countermax`를 `globalreserve`에서 뺍니다.

Quick Quiz 5.42: Figure 5.20의 라인 14에서 `flush_local_count()` 가 `ctrandmax` 변수를 0으로 만든 후 그냥 다시 값을 넣을 수 없는 이유는 뭐죠? ■

Quick Quiz 5.43: Figure 5.20의 라인 27에서 `flush_local_count()` 가 `ctrandmax` 변수를 빼우는 동안 `add_count()`나 `sub_count()`의 빠른 수행경로가 `ctrandmax`를 함께 사용하면서 동시에 수행되지 못하는 이유는 뭐죠? ■

Figure 5.21의 라인 1-22는 호출한 쓰레드의 로컬 `ctrandmax` 변수를 재설정 하는 `balance_count()` 함수의 코드를 보여줍니다. 이 함수는 앞의 알고리즘과 상당히 유사합니다만, `ctrandmax` 변수의 병합 작업을 처리하는 부분이 추가되었습니다. 해당 코드의 자세한 분석은 라인 24에 시작하는 `count_register_thread()` 함수와 라인 33에서 시작하는 `count_unregister_thread()` 함수와 함께 독자 여러분의 연습문제로 남겨두겠습니다.

Quick Quiz 5.44: `atomic_set()`은 주어진 `atomic_t`에 단순히 스토어를 할 뿐인데, 어떻게 Figure 5.21의 라인 21에서의 `balance_count()`는 `flush_local_count()`의 해당 변수에 동시에 가해지는 업데이트에도 불구하고 올바르게 동작할 수 있는 거죠? ■

다음 섹션에서는 이 설계를 질적으로 평가해 봅니다.

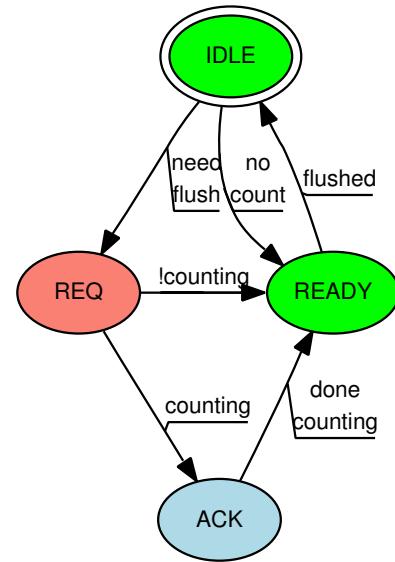


Figure 5.22: Signal-Theft State Machine

5.4.2 Atomic Limit Counter Discussion

이것은 실제로 카운터가 그 리미트를 가지면서도 어떤 방법으로든 사용될 수 있는 최초의 구현입니다만, 빠른 수행경로에 일부 시스템에서는 빠른 수행경로를 매우 느리게 만들 수 있는 어토믹 오퍼레이션을 추가하는 추가비용을 갖습니다. 일부 워크로드에서는 이 성능 저하가 조절될 수 있겠지만, 더 나은 읽기 쪽 성능을 위한 알고리즘을 찾아보는 것도 좋을 것입니다. 그런 알고리즘 중 하나는 다른 쓰레드로부터 카운트를 훔치기 위해 시그널 핸들러를 사용합니다. 시그널 핸들러는 시그널을 받은 쓰레드의 컨텍스트에서 동작하기 때문에, 다음 섹션에서 살펴보겠지만, 어토믹 오퍼레이션이 필요 없습니다.

Quick Quiz 5.45: 하지만 시그널 핸들러는 수행 중에 다른 CPU로 옮겨져서 수행될 수도 있잖아요. 이런 가능성을 쓰레드와 해당 쓰레드를 인터럽트 하는 시그널 핸들러 사이의 안정적인 통신을 위해 어토믹 인스트럭션과 메모리 배리어를 필요로 하게 만들지 않을까요? ■

5.4.3 Signal-Theft Limit Counter Design

쓰레드별 상태는 이제 연관된 쓰레드에 의해서만 조정된다고는 해도, 여전히 시그널 핸들러를 이용해 동기화를 할 필요가 있습니다. 이 동기화는 Figure 5.22에 보여지는 스테이트 머신에 의해 제공됩니다. 해당 스테이트 머신은 IDLE 상태에서 시작하고 `add_count()`나

sub_count() 가 로컬 쓰레드의 카운트의 합과 글로벌 카운트가 주어진 리퀘스트를 수용할 수 없다는 걸 알았을 때, 관련된 느린 수행 경로는 각 쓰레드의 theft 상태를 REQ로 만듭니다 (해당 쓰레드가 카운트가 있을 경우이고, 그렇지 않다면 READY로 곧바로 변환시킵니다). 초록색으로 표시되어 있듯, gblcnt_mutex 락을 쥐고 있는 느린 수행경로에게만 IDLE 상태로부터의 변환이 허가되어 있습니다.³ 해당 느린 수행 경로는 이후 각 쓰레드에 시그널을 보내고, 그에 상응하는 시그널 핸들러들이 연관된 쓰레드의 theft 와 counting 변수들을 체크합니다. 만약 theft 상태가 REQ 가 아니라면 시그널 핸들러는 상태를 바꿀 권한이 없고, 따라서 그냥 리턴합니다. 그렇지 않고, counting 변수가 값이 있어 현재 쓰레드의 빠른 수행경로가 실행 중이란 것을 알게 된다면, 시그널 핸들러는 theft 상태를 ACK로, 그렇지 않다면 READY로 바꿉니다.

파란색으로 표시되어 있듯, theft 상태가 ACK 이라면 빠른 수행 경로만이 theft 상태를 바꿀 수 있습니다. 빠른 수행 경로가 완료되면, theft 상태를 READY로 바꿉니다.

한번 느린 수행 경로가 한 쓰레드의 theft 상태가 READY임을 봤다면, 해당 느린 수행 경로는 해당 쓰레드의 카운트를 훔칠 권한을 갖습니다. 해당 느린 수행 경로는 이후 해당 쓰레드의 theft 상태를 IDLE로 만듭니다.

Quick Quiz 5.46: Figure 5.22에서 REQ theft 상태는 왜 빨간색으로 칠해졌나요? ■

Quick Quiz 5.47: Figure 5.22에서, 두개의 분리된 REQ 와 ACK theft 상태를 갖는 이유가 뭐죠? 왜 그 두 상태를 하나의 REQACK 상태로 만들어서 스테이트 머신을 간단하게 만들지 않는 거예요? 만약 그렇게 하면 그 상태에 먼저 도달하는 시그널 핸들러나 빠른 수행 경로가 상태를 READY로 바꿀 수 있을 텐데요. ■

5.4.4 Signal-Theft Limit Counter Implementation

Figure 5.23 (count_lim_sig.c)는 signal-theft based counter 구현에 사용되는 데이터 구조를 보입니다. 라인 1-7에서는 앞의 섹션에서 설명한 상태들과 쓰레드 별 theft 스테이트 머신을 위한 값들을 정의합니다. 라인 8-17은 앞의 구현과 비슷합니다만, 라인 14 와 15에 쓰레드의 countermax 와 theft 변수에의 원격에서의 접근을 허가하기 위한 코드가 추가되었습니다.

Figure 5.24는 쓰레드별 변수와 전역 변수 사이에 카운트를 옮겨주는 함수를 보입니다. 라인 1-7은 globalize_count() 함수를 보이는데, 이 함수

³ 이 책의 흑백 버전을 위해 말해두자면, IDLE과 READY는 초록, REQ는 빨강, 그리고 ACK는 파란색으로 되어 있습니다.

```

1 #define THEFT_IDLE 0
2 #define THEFT_REQ 1
3 #define THEFT_ACK 2
4 #define THEFT_READY 3
5
6 int __thread theft = THEFT_IDLE;
7 int __thread counting = 0;
8 unsigned long __thread counter = 0;
9 unsigned long __thread countermax = 0;
10 unsigned long globalcountmax = 10000;
11 unsigned long globalcount = 0;
12 unsigned long globalreserve = 0;
13 unsigned long *counterp[NR_THREADS] = { NULL };
14 unsigned long *countermaxp[NR_THREADS] = { NULL };
15 int *theftp[NR_THREADS] = { NULL };
16 DEFINE_SPINLOCK(gblcnt_mutex);
17 #define MAX_COUNTERMAX 100

```

Figure 5.23: Signal-Theft Limit Counter Data

는 앞의 구현과 동일합니다. 라인 9-19는 flush_local_count_sig() 함수로, 훔치기 과정에서 사용되는 시그널 핸들러입니다. 라인 11과 12에서는 theft 상태가 REQ인지 확인하고, 아니라면 변경 없이 리턴합니다. 라인 13에서는 theft 변수의 샘플링이 해당 변수에의 변경 이전에 이루어졌음을 분명히 하기 위해 메모리 배리어를 칩니다. 라인 14는 theft 상태를 ACK로 놓고, 라인 15에서는 이 쓰레드의 빠른 수행 경로가 수행중인지 확인 후, 라인 16에서 theft 상태를 READY로 놓습니다.

Quick Quiz 5.48: Figure 5.24의 flush_local_count_sig() 함수에서는 왜 theft 쓰레드별 변수의 사용을 ACCESS_ONCE()로 감싼거죠? ■

라인 21-49는 flush_local_count()를 보이는 데, 이 함수는 느린 수행 경로에서 모든 쓰레드의 로컬 카운트를 비우기 위해 호출됩니다. 라인 26-34의 루프에서는 로컬 카운트가 있는 모든 쓰레드의 theft 상태를 바꾸고, 해당 쓰레드들에 시그널을 날립니다. 라인 27에서는 존재하지 않는 쓰레드들을 스kip합니다. 그렇지 않다면, 라인 28에서 현재 쓰레드가 로컬 카운트를 가지고 있는지 보고, 가지고 있지 않다면 라인 29에서 해당 쓰레드의 theft 상태를 READY로 바꾸고 라인 30에서 다음 쓰레드로 넘어갑니다. 그렇지 않고 현재 쓰레드가 로컬 카운트를 가지고 있다면, 라인 32에서 해당 쓰레드의 theft 상태를 REQ로 바꾸고, 라인 33에서 시그널을 날립니다.

Quick Quiz 5.49: Figure 5.24에서, 왜 다른 쓰레드의 countermax 변수를 바로 접근해도 안전한 거죠? ■

Quick Quiz 5.50: Figure 5.24에서, 왜 라인 33은 현재 쓰레드가 자기 자신에게 시그널을 보내는지 체크하지 않나요? ■

Quick Quiz 5.51: Figure 5.24의 코드는 gcc 와 POSIX에서 동작합니다. ISO C 표준에서 동작하게 하려면 뭐가 필요할까요? ■

```

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 static void balance_count(void)
52 {
53     countermax = globalcountmax -
54     globalcount - globalreserve;
55     countermax /= num_online_threads();
56     if (countermax > MAX_COUNTERMAX)
57         countermax = MAX_COUNTERMAX;
58     globalreserve += countermax;
59     counter = countermax / 2;
60     if (counter > globalcount)
61         counter = globalcount;
62     globalcount -= counter;
63 }

```

Figure 5.24: Signal-Theft Limit Counter Value-Migration Functions

```

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 }

```

Figure 5.25: Signal-Theft Limit Counter Add Function

라인 35-48 의 루프는 각 쓰레드가 READY 상태에 도달하길 기다리고, 이후 해당 쓰레드의 카운트를 훔칩니다. 라인 36-37은 존재하지 않는 쓰레드들을 스kip하고, 라인 38-42의 루프에서 현재 쓰레드의 theft 상태가 READY 가 되길 기다립니다. 라인 39에서는 우선순위 역전(priority-inversion) 문제를 막기 위해 1 밀리세컨드 동안 블락되고, 라인 40에서 해당 쓰레드의 시그널이 아직 도착하지 않은 것으로 판단되면 라인 41에서 시그널을 다시 날립니다. 현재 살펴보고 있는 쓰레드의 상태가 마침내 READY 가 되면 실행 흐름은 라인 43에 도달해 라인 43-46에서 훔치기를 시전합니다. 라인 47은 해당 쓰레드의 theft 상태를 다시 IDLE로 되돌립니다.

Quick Quiz 5.52: Figure 5.24 의 라인 41 에서는 왜 시그널을 다시 보내죠? ■

라인 51-63에서는 balance_count() 함수를 보이는데, 앞의 예제와 비슷합니다.

Figure 5.25는 add_count() 함수를 보입니다. 라인 5-20에는 빠른 수행 경로가, 라인 21-35에는 느린 수행 경로가 있습니다. 라인 5에서는 쓰레드별 counting 변수를 1로 만들어서 이후에 이 쓰레드를

```

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 5.26: Signal-Theft Limit Counter Subtract Function

```

1 unsigned long read_count(void)
2 {
3     int t;
4     unsigned long sum;
5
6     spin_lock(&gblcnt_mutex);
7     sum = globalcount;
8     for_each_thread(t)
9         if (counterp[t] != NULL)
10             sum += *counterp[t];
11     spin_unlock(&gblcnt_mutex);
12     return sum;
13 }

```

Figure 5.27: Signal-Theft Limit Counter Read Function

인터럽트하는 어떤 시그널 핸들러도 `theft` 상태를 `READY` 가 아니라 `ACK`로 설정하게 해둠으로써, 이 빠른 수행 경로가 올바르게 완료되도록 합니다. 라인 6은 컴파일러가 빠른 수행 경로의 본체가 `counting` 설정보다 앞에 위치하도록 재배치 하는 것을 막아줍니다. 라인 7과 8은 쓰레드별 데이터가 `add_count()`를 처리할 수 있는지 확인하고 현재 수행중인 훔치기 작업이 없다면 라인 9에서 빠른 수행 경로 더하기 연산을 수행하고 라인 10에서 빠른 수행 경로가 진행됨을 알립니다.

어떤 경우든, 라인 12에서 컴파일러가 빠른 수행 경로 본체가 라인 13 뒤에 오도록 재배치해 이후의 시그널 핸들러가 훔치기 작업을 하는 사태를 만들지 못하게 합니다. 라인 14에서는 다시 컴파일러 재배치를 막고, 라인 15에서 시그널 핸들러가 `theft` 상태를 `READY`로 바꿨는지 보고, 만약 그렇다면 라인 16에서 라인 17에서 `READY`로 설정한 상태를 본 CPU는 라인 9의 결과도 보도록 해줍니다. 라인 9에서의 빠른 수행 경로 더하기가 실행되었다면, 라인 20에서 성공을 리턴합니다.

그렇지 않다면, 라인 21부터 시작되는 느린 수행 경로로 떨어집니다. 느린 수행 경로의 구조는 앞의 예제들과 유사하므로, 분석은 독자 여러분의 연습 문제로 놔두겠습니다. 비슷하게, Figure 5.26의 `sub_count()`의 구조는 `add_count()`와 동일하므로, `sub_count()` 역시, 그리고 Figure 5.27의 `read_count()`와 함께 그 분석을 독자 여러분의 몫으로 남겨 두겠습니다.

Figure 5.28의 라인 1-12는 `count_init()`를 보여주는데, 이 함수에선 `flush_local_count_sig()`를 `SIGUSR1`의 시그널 핸들러로 설정해서 `flush_local_count()`에서 `pthread_kill()` 호출로 `flush_local_count_sig()`를 실행시킬 수 있게 합니다. 쓰레드 등록과 해제를 위한 코드는 앞의 예제와 비슷하므로, 이 코드의 분석은 독자의 몫으로 남겨 두겠습니다.

5.4.5 Signal-Theft Limit Counter Discussion

Signal-theft 구현은 제 Intel Core Duo 랩탑에서 어토믹 인스트럭션 구현보다 두배 이상 빠르게 동작합니다. 항상 그럴까요?

Signal-theft 구현은 펜티엄-4 시스템에서는 어토믹 인스트럭션들이 느리기 때문에 선호될만 합니다만 과거의 80386 기반 Sequent Symmetry 시스템에서는 어토믹 구현의 더 짧은 코드를 더 빨리 수행할 것입니다. 하지만, 이 업데이트 쪽의 성능 향상은 높아진 읽는쪽 오버헤드와 함께 옵니다: POSIX 시그널은 공짜가 아닙니다. 결국 원하는 핵심이 성능이라면, 두 구현 모두를 어플리케이션이 배포될 시스템 위에서 돌려보는게 좋을 겁니다.

Quick Quiz 5.53: POSIX 시그널만 느린게 아니라, 시그널을 각 쓰레드에 보내는 행위 자체가 확장성이 없어요. 만약 10,000 개의 쓰레드가 있고 읽는쪽도 빨라야 한다면 어떻게 하시겠어요? ■

이건 왜 높은 품질의 API가 그렇게 중요한지를 보여주는 한가지 예이기도 합니다: 높은 품질의 API는 바꿔는 하드웨어 성능 특성에 따라 요구되는 구현 변경이 가능하게 해줍니다.

Quick Quiz 5.54: 아래쪽 한계는 명확하게 지키지만 위쪽 한계는 좀 정확하지 않아도 되는 한계 카운터를 원한다면 어떻게 하면 될까요? ■

```

1 void count_init(void)
2 {
3     struct sigaction sa;
4
5     sa.sa_handler = flush_local_count_sig;
6     sigemptyset(&sa.sa_mask);
7     sa.sa_flags = 0;
8     if (sigaction(SIGUSR1, &sa, NULL) != 0) {
9         perror("sigaction");
10        exit(-1);
11    }
12 }
13
14 void count_register_thread(void)
15 {
16     int idx = smp_thread_id();
17
18     spin_lock(&gblcnt_mutex);
19     counterp[idx] = &counter;
20     countermaxp[idx] = &countermax;
21     theftp[idx] = &theft;
22     spin_unlock(&gblcnt_mutex);
23 }
24
25 void count_unregister_thread(int nthreadsexpected)
26 {
27     int idx = smp_thread_id();
28
29     spin_lock(&gblcnt_mutex);
30     globalize_count();
31     counterp[idx] = NULL;
32     countermaxp[idx] = NULL;
33     theftp[idx] = NULL;
34     spin_unlock(&gblcnt_mutex);
35 }
```

Figure 5.28: Signal-Theft Limit Counter Initialization Functions

5.5 Applying Specialized Parallel Counters

Section 5.4에서 소개한 정확한 리미트 카운터 구현은 매우 유용할 수 있지만, 예를 들어 I/O 디바이스로의 액세스 수를 세는 경우와 같이 카운터의 값이 항상 0에 가깝게 유지된다면 별로 유용하지 않을 겁니다. 그런 경우에서의 높은 오버헤드는 일반적으로 얼마나 많은 레퍼런스들이 존재하는지 신경쓰지 않는다는 점을 감안하면 특히나 큰 문제가 됩니다. Quick Quiz 5.5에서 이야기한 제거 가능한 I/O 디바이스 액세스 카운트 문제에서 이야기 했듯, 액세스의 갯수는 누군가가 정말로 해당 디바이스를 제거하려 하고 있는 드문 경우가 아니고는 무의미합니다.

이 문제에 대한 간단한 해결책은 카운터에 커다란 “바이어스” (예를 들어, 10억 정도)를 더해 값이 0보다 충분히 커서 카운터가 효과적으로 동작할 수 있도록 보장해 주는 것입니다. 누군가가 디바이스를 제거하려 한다면, 이 바이어스는 카운터 값에서 빼집니다. 마지막의 몇개 액세스들을 카운팅하는 건 매우 비효율적이 되겠지만, 중요한 건, 그 앞의 많은 액세스들은 최대 속도로 카운트 될 것이라는 점입니다.

Quick Quiz 5.55: 바이어스된 카운터를 사용할 때 그 외에 뭘 하면 좋을까요? ■

바이어스된 카운터는 매우 유용하고 도움이 될 수 있지만, page 37에서 이야기된 제거 가능한 I/O 디바이스 액세스 카운트 문제에의 부분적 해결책에 불과합니다. 디바이스를 제거하려 시도할 때, 우린 현재 I/O 액세스들의 정확한 갯수를 알아야 할뿐만이 아니라, 이후의 액세스가 발생하지 않도록 막아야 합니다. 이걸 해내는 한가지 방법은 리더-라이터 락을 사용해 카운터를 업데이트 할 때 읽기 권한을 획득하고, 카운터를 체크할 때 같은 리더-라이터 락에서 쓰기 권한을 획득하는 것입니다. I/O를 위한 코드는 다음과 같이 될겁니다:

```

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 }
```

라인 1에서 락을 이용해 읽기 권한을 획득하고, 라인 3이나 7에서 이를 해제합니다. 라인 2에서는 디바이스가 삭제되는 중인지 확인하고, 그렇다면 라인 3에서 락을 해제하고 라인 4에서 I/O를 취소하거나, 디바이스가 삭제될 예정일 때 해야 할 무슨 일이든 합니다. 디바이스가 삭제중인지 않다면 라인 6에서 액세스 카운트를 증가시키고 라인 7에서 락을 해제하고, 라인 8에서 I/O를 수행한 후, 라인 9에서 액세스 카운트를 낮춥니다.

Quick Quiz 5.56: 이거 참 웃기네요! 카운터를 업데이트 하기 위해 리더-라이터 락의 읽기 권한 획득을 한다니요? 뭐하는거예요??? ■

디바이스를 제거하는 코드는 다음과 같습니다:

```

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();
```

라인 1에서는 락의 쓰기-획득을 하고 라인 4에서 해제합니다. 라인 2에서 디바이스가 제거되는 중임을 알고, 라인 5-7에서 존재하는 I/O 오퍼레이션들이 끝나길 기다립니다. 마지막으로, 라인 8에서 디바이스 제거에 필요한 작업을 진행합니다.

Quick Quiz 5.57: 실제 시스템에 적용하려면 해결해

야할 문제들이 또 뭐가 있을 수 있을까요? ■

5.6 Parallel Counting Discussion

이 챕터에서는 전통적인 카운팅 도구들의 신뢰성, 성능, 그리고 확장성 문제를 보았습니다. C 언어의 ++ 오퍼레이터는 멀티쓰레드 코드에서 신뢰성 있게 동작함을 보장하지 않고, 하나의 변수에 어토믹 오퍼레이션을 행하는 것은 성능도 떨어지고 확장성도 좋지 않습니다. 그래서 이 챕터에서는 일부 특정한 경우에 성능도 좋고 확장성도 매우 좋은 카운팅 알고리즘 일부를 보았습니다.

이 카운팅 알고리즘들에서 얻은 교훈을 한번 더 보는 것은 가치있는 일일 것입니다. 그러기 위해, Section 5.6.1에서 성능과 확장성을 정리하고, Section 5.6.2에서 특수화를 위한 필요성을 이야기해 보고, 그리고 마지막으로 Section 5.6.3에서 이번에 배운 교훈들을 나열해보고 이 교훈들을 확장해 나갈 뒤의 챕터에 대해 간단히 알아봅니다.

5.6.1 Parallel Counting Performance

Table 5.1에 네개의 병렬 통계성 카운팅 알고리즘들의 성능이 나열되어 있습니다. 네개의 알고리즘 모두 업데이트에 있어 완벽에 가까운 선형적 확장성을 제공합니다. 쓰레드별 변수 구현(count_end.c)는 특히나 어레이 기반 구현(count_stat.c)에 비해 업데이트에 있어 빠릅니다만, 코어의 수가 많을 때 읽기 쪽에선 느리며, 많은 읽기 오퍼레이션이 병렬적으로 수행될 때에는 상당한 락 경쟁 상황으로 성능이 떨어집니다. 이 경쟁 상황은 Table ??의 count_end_rcu.c 열에서 볼 수 있듯, Chapter 9에서 소개되는 지연 처리(deferred-processing) 테크닉으로 해결될 수 있습니다. 지연 처리 기법은 count_stat_eventual.c 열에서도 그 효과를 보이는데, 결과적 일관성의 덕입니다.

Quick Quiz 5.58: Table 5.1의 count_stat.c 열에 보면 읽기 성능이 쓰레드 수에 따라 선형적으로 확장되는데요. 쓰레드 수가 늘어나면 더 많은 쓰레드별 카운터의 합이 이루어져야 하는데 어떻게 그게 가능하죠? ■

Quick Quiz 5.59: Table 5.1의 마지막 열을 보더라도 통계적 카운터 구현의 읽기 쪽 성능은 매우 나쁘군요. 왜 이렇게 성능 나쁜 알고리즘을 신경쓰는거죠? ■

Figure 5.2는 병렬 리미트 카운팅 알고리즘들의 성능을 보여줍니다. 리미트가 정확히 지켜져야 한다는 제약은 상당한 성능 저하를 가져옵니다만, 적어도 이 4.7 GHz Power-6 시스템에서는 어토믹 오퍼레이션을 시그널로 대체하는 것으로 그 성능 저하를 줄일 수 있습니다. 이 구현들은 모두 동시적 읽기에 의해 유발되는 읽기 쪽의 락 경쟁으로 인한 성능 저하 문제를 갖습니다.

Algorithm	Section	Updates	Reads	
			1 Core	32 Cores
count_stat.c	5.2.2	11.5 ns	408 ns	409 ns
count_stat_eventual.c	5.2.3	11.6 ns	1 ns	1 ns
count_end.c	5.2.4	6.3 ns	389 ns	51,200 ns
count_end_rcu.c	13.2.1	5.7 ns	354 ns	501 ns

Table 5.1: Statistical Counter Performance on Power-6

Algorithm	Section	Exact?	Updates	Reads	
				1 Core	64 Cores
count_lim.c	5.3.2	N	3.6 ns	375 ns	50,700 ns
count_lim_app.c	5.3.4	N	11.7 ns	369 ns	51,000 ns
count_lim_atomic.c	5.4.1	Y	51.4 ns	427 ns	49,400 ns
count_lim_sig.c	5.4.4	Y	10.2 ns	370 ns	54,000 ns

Table 5.2: Limit Counter Performance on Power-6

Quick Quiz 5.60: Table 5.2에 보여진 성능 데이터를 놓고 보자면, 우리는 항상 어토믹 오퍼레이션보다는 시그널을 사용해야겠군요, 그렇죠? ■

Quick Quiz 5.61: Table 5.2에 보여진 읽는 쓰레드간의 락 컨텐션을 해결하기 위해 고급 테크닉들이 사용될 수 있을까요? ■

한마디로, 이 챕터는 특수한 경우들에 한해 성능도 좋고 확장성도 매우 좋은 카운팅 알고리즘들을 보였습니다. 하지만 우리의 병렬 카운팅은 특정한 경우에만 한정되어야 하는 걸까요? 모든 경우에 효율적으로 동작하는 일반적 알고리즘이 있다면 더 좋지 않을까요? 다음 섹션에서는 이런 질문에 대해 알아봅니다.

5.6.2 Parallel Counting Specializations

이 알고리즘들은 각자의 특별한 경우에 대해서만 잘 동작한다는 사실은 일반적인 병렬 프로그래밍에 있어서는 중요한 문제로 여겨질 것입니다. 무엇보다, C 언어의 `++` 오퍼레이터는 싱글 쓰레드 코드에서는, 그것도 특수한 경우에 대해서만이 아니라 일반적인 경우에서 잘 동작합니다, 그렇죠?

이 생각의 흐름은 진실을 담고 있긴 하나, 본질적으로는 잘못 유추되어졌습니다. 문제는 병렬성이 아니라, 확장성입니다. 이걸 이해하기 위해, 먼저 C 언어의 `++` 오퍼레이터를 생각해 봅시다. 사실, `++` 오퍼레이터는 일반적인 경우에 동작하지 않고, 그저 제한된 수의 영역 내에서만 동작합니다. 1,000 자리 십진수 숫자를 다뤄야 한다면, C 언어 `++` 오퍼레이터는 동작하지 않을 겁니다.

Quick Quiz 5.62: `++` 오퍼레이터는 1,000 자리 숫자

에도 잘 동작해요! 연산자 오버로딩이라고 못들어봤어요??? ■

이 문제는 수에 국한되지 않습니다. 데이터를 저장하고 찾아와야 한다고 생각해 봅시다. ASCII 파일을 써야 할까요? XML? 관계형 데이터베이스? 링크드 리스트? 덴스 어레이? B-트리? 랙드스 트리? 또는 데이터를 저장하고 찾아오는 것을 허락하는 다른 수많은 데이터 구조와 환경 중 하나? 그건 무엇을 해야 하는가, 얼마나 빨리 해야 하는가, 그리고 얼마나 데이터가 큰가에 달려 있습니다—심지어 순차적 시스템에서도요.

비슷하게, 카운팅이 필요하다면, 해결책은 얼마나 큰 숫자를 다뤄야 하고, 얼마나 많은 CPU가 동시에 그 숫자를 조정할 것이며, 어떻게 그 숫자가 사용될 것이고, 어떤 수준의 성능과 확장성이 필요한지에 따라 달라집니다.

또한 이 문제는 소프트웨어에 국한되지도 않습니다. 사람들이 작은 개울을 건널 수 있게 해주는 다리의 설계는 하나의 나무판자처럼 간단할 수도 있습니다. 하지만 당신은 수 킬로미터 폭의 콜럼비아 강을 위해서라면은 물론, 콘크리트 트럭이 지나가야 하는 다리를 설계해야 한다고만 해도 나무판자를 사용하진 않을 겁니다. 깊게 말해서, 다리 설계가 거리와 부하에 따라 달라져야만 하는 만큼, 소프트웨어 설계 역시 CPU 갯수에 따라 달라져야만 합니다. 그렇다면 하나, 이 작업을 자동화 해서 소프트웨어가 하드웨어 구성과 워크로드의 변화를 수용할 수 있도록 하면 좋을 것입니다. 실제로 그런 자동화 부류를 위한 연구 [AHS⁺03, SAH⁺03]가 있어왔고, 리눅스 커널 역시 제한된 바이너리 수정을 포함해, 부팅 타임 재구성을 합니다. 이런 종류의 적응 기능은 주요 시스템에서의 CPU 갯수가 늘어남을 유지함에 따라

더욱 중요해질 것입니다.

요약하자면, Chapter 3에서 이야기 했듯, 물리 법칙은 다리와 같은 기계적 인공물에 제약을 가하듯이 병렬 소프트웨어에도 제약을 가합니다. 이런 제약으로 인해 특수화가 필요하게 됩니다. 다만 소프트웨어의 경우에는 그 특수화를 요구되는 하드웨어와 워크로드에 걸맞는 특수화로 선택하는 것 자체를 자동화 하는게 가능할 수도 있습니다.

물론, 일반화된 카운팅조차도 상당히 특수화 되어 있습니다. 컴퓨터를 가지고는 그 외에도 수많은 일을 해야 합니다. 다음 섹션에서는 카운터로부터 배운 것과 이 책의 뒤에서 다루게 될 주제 사이의 관계를 알아봅니다.

5.6.3 Parallel Counting Lessons

이 챕터를 시작하는 문단에서는 우리의 카운팅에 대한 공부는 병렬 프로그래밍에 대한 훌륭한 소개를 제공할 것이라 이야기 했습니다. 이 섹션에서는 이 챕터에서 얻은 교훈과 뒤의 챕터들에서 다룰 것들 사이의 명시적 관계를 만들어 봅니다.

이 챕터에서의 예제들은 **분할하기** (*partitioning*) 가 확장성과 성능을 위한 중요한 도구임을 보였습니다. 카운터들은 Section 5.2에서 다룬 통계적 카운터들처럼 완벽하게 분할되어질 수도 있고, Section 5.3과 Section 5.4에서 다루어진 리미트 카운터들처럼 부분적으로 분할될 수도 있습니다. 분할은 Chapter 6에서 훨씬 꽉꽉 다루어질 것이고, 부분적 분할은 특히 Section 6.4에서 *parallel fastpath* 라 불리며 다루어질 것입니다.

Quick Quiz 5.63: 하지만 우리가 모든 것을 분할할 거라면, 왜 공유 메모리 멀티쓰레딩을 신경쓰죠? 그냥 문제를 완벽하게 분할해버리고 각 분할된 조각들을 여러 프로세스들로, 각자의 어드레스 스페이스에서 처리하도록 돌리지 않는건가요? ■

부분적 분할 카운팅 알고리즘들은 글로벌 데이터를 락킹으로 보호하고, 락킹은 Chapter 7의 주제입니다. 반면, 분할된 데이터는 완전히 연관된 쓰레드의 제어 하에 있어서 어떤 동기화도 필요하지 않게 되는 경향이 있습니다. 이 데이터 소유권은 Section 6.3.4에서 소개하고 Chapter 8에서 더 자세히 다룹니다.

정수의 더하기와 빼기는 일반적인 동기화 오퍼레이션들에 비해 매우 비용이 싸기 때문에, 합리적인 확장성을 얻기 위해선 동기화 오퍼레이션들을 아껴 쓸 것이 요구됩니다. 이를 이루는 방법 중 한 가지는 더하기와 빼기 오퍼레이션들을 몰아서 해서 (batch) 하나의 동기화 오퍼레이션으로 이 수많은 비용 적은 오퍼레이션들이 처리되게 하는 것입니다. 한 부류의 또는 또 다른 부류의 몰아서 처리하기 (Batching) 최적화는 Table 5.1과 Table 5.2에 보인 카운팅 알고리즘들에서 각각 사용되

었습니다.

마지막으로, Section 5.2.3에서 다룬 결과적으로 일관적인 통계적 카운터는 뒤로 미루는 (deferring) 행위 (이 경우, 클로벌 카운터를 업데이트 하는 것)가 상당한 성능 향상과 확장성에 도움을 줄 수 있음을 보았습니다. 이런 접근은 일반적인 경우의 코드가 다른 경우에 사용할 수 있는 것들보다 훨씬 비용이 싼 동기화 오퍼레이션들을 사용할 수 있게 합니다. Chapter 9는 여러 미루기 방법들이 성능, 확장성, 그리고 심지어는 실제 시간에서의 응답 (real-time response) 까지 개선을 줄 수 있음을 보일 겁니다.

요약을 요약하자면:

1. 분할하기는 성능과 확장성을 높입니다.
2. 부분 분할은 일반적인 코드 패스에만 분할을 적용한 것으로, 대부분 잘 동작합니다.
3. 부분 분할은 (Section 5.2의 통계적 카운터의 분할된 업데이트와 분할되지 않은 읽기처럼) 코드에만 적용 가능한 것이 아니라, (Section 5.3과 Section 5.4의 리미트 카운터들이 리미트와 멀때는 빠르게, 하지만 리미트에 가까워졌을 때는 느리게 동작하듯이) 시간에도 적용될 수 있습니다.
4. 시간을 분할하는 것은 종종 비싼 글로벌 오퍼레이션을 줄여서 동기화 오버헤드를 줄여서 결과적으로 성능과 확장성을 개선하기 위해 업데이트를 지역적으로 몰아서 하도록 합니다. Table 5.1과 Table 5.2에 보여진 모든 알고리즘들은 몰아 처리하기 (Batching)을 많이 사용합니다.
5. 읽기만 하는 코드 패스는 읽기만 하도록 유지되어야 합니다: 공유 메모리에의 의미없는 동기화된 쓰기는 Table 5.1의 `count_end.c` 열에 보여진 것처럼 성능과 확장성을 떨어뜨립니다.
6. 현명한 딜레이의 사용은 Section 5.2.3에서 보았듯이 성능과 확장성을 증진시킵니다.
7. 병렬 성능과 확장성은 대부분의 경우 균형잡기입니다: 특정한 지점 이후부터는, 일부 코드 패스를 최적화 하는 것은 다른쪽의 성능을 떨어뜨릴 겁니다. Table 5.1의 `count_stat.c`와 `count_end_rcu.c` 열이 이 지점을 보여줍니다.
8. 다른 수준의 성능과 확장성은 다른 여러 요소들이 그렇듯이 알고리즘과 데이터 구조 설계에 영향을 끼칠 겁니다. Figure 5.3가 이 점을 보이고 있습니다: 두개의 CPU로 구성된 시스템에서 어토믹 증가는 사용하기 적당합니다만, 8개 CPU로 구성된 시스템에서는 결코 적당치 않습니다.

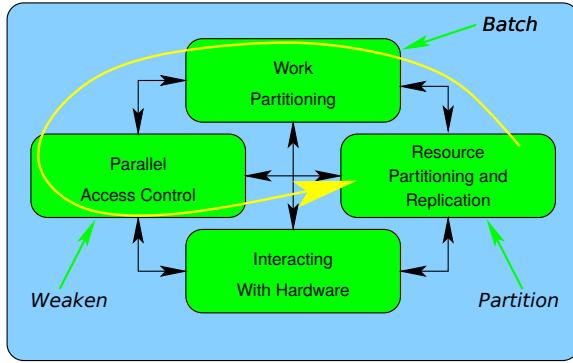


Figure 5.29: Optimization and the Four Parallel-Programming Tasks

Summarizing still further, we have the “big three” methods of increasing performance and scalability, namely (1) *partitioning* over CPUs or threads, (2) *batching* so that more work can be done by each expensive synchronization operations, and (3) *weakening* synchronization operations where feasible. As a rough rule of thumb, you should apply these methods in this order, as was noted earlier in the discussion of Figure 2.6 on page 15. The partitioning optimization applies to the “Resource Partitioning and Replication” bubble, the batching optimization to the “Work Partitioning” bubble, and the weakening optimization to the “Parallel Access Control” bubble, as shown in Figure 5.29. Of course, if you are using special-purpose hardware such as digital signal processors (DSPs), field-programmable gate arrays (FPGAs), or general-purpose graphical processing units (GPGPUs), you may need to pay close attention to the “Interacting With Hardware” bubble throughout the design process. For example, the structure of a GPGPU’s hardware threads and memory connectivity might richly reward very careful partitioning and batching design decisions.

In short, as noted at the beginning of this chapter, the simplicity of counting have allowed us to explore many fundamental concurrency issues without the distraction of complex synchronization primitives or elaborate data structures. Such synchronization primitives and data structures are covered in later chapters.

Chapter 6

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” [Ale79, GHJV95, SSRB00] 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. This chapter will help you design partitioning into your code, with some discussion of batching and weakening as well. The word “design” is very important: You should partition first, batch second, weaken third, and code fourth. Changing this order often leads to poor performance and scalability along with great frustration.

To this end, Section 6.1 presents partitioning exercises, Section 6.2 reviews partitionability design criteria, Section 6.3 discusses selecting an appropriate synchronization granularity, Section 6.4 gives an overview of important parallel-fastpath designs that provide speed and scalability in the common case with a simpler but less-scalable fallback “slow path” for unusual situations, and finally Section 6.5 takes a brief look beyond partitioning.

6.1 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.

6.1.1 Dining Philosophers Problem

Figure 6.1 shows a diagram of the classic Dining Philosophers problem [Dij71]. This problem features five philo-

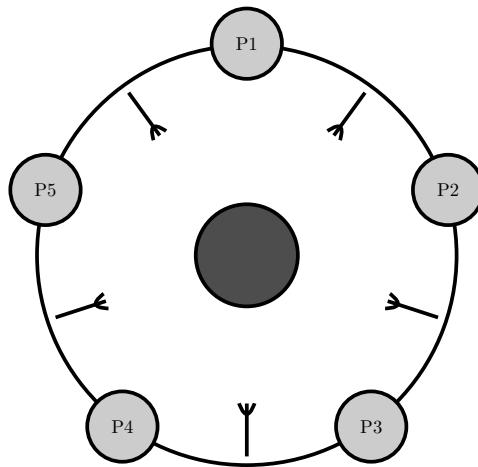


Figure 6.1: Dining Philosophers Problem

phers 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. Please note that it is not sufficient to allow at least one philosopher to eat. As Figure 6.2 shows, starvation of even a few of the philosophers is to be avoided.

¹ Readers who have difficulty imagining a food that requires two forks are invited to instead think in terms of chopsticks.

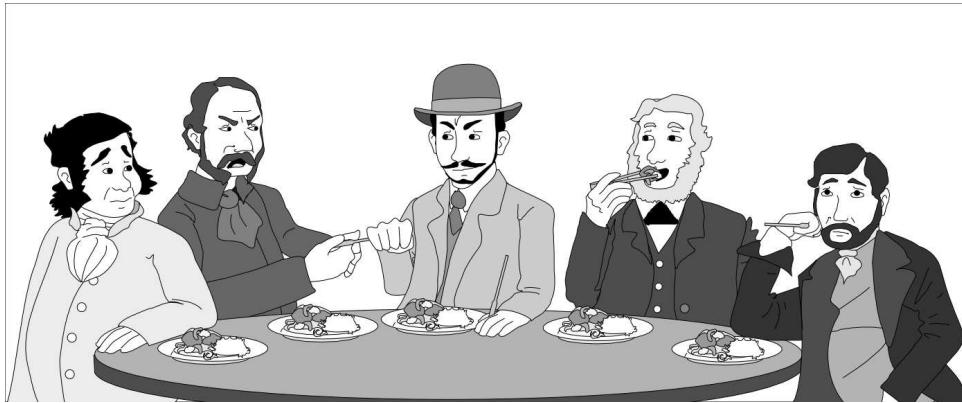


Figure 6.2: Partial Starvation Is Also Bad

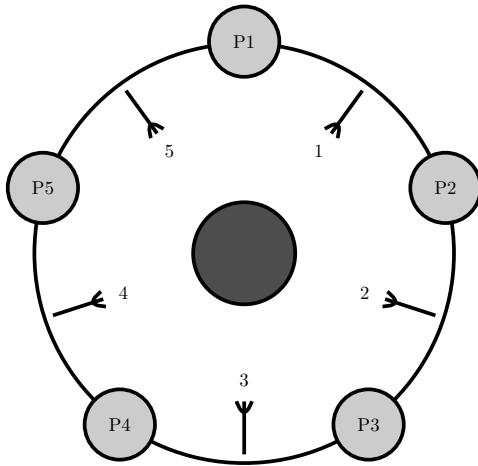


Figure 6.3: Dining Philosophers Problem, Textbook Solution

Dijkstra's solution used a global semaphore, which works fine assuming negligible communications delays, an assumption that became invalid in the late 1980s or early 1990s.² Therefore, recent solutions number the forks as shown in Figure 6.3. 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 proceed eating.

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.

In short, this algorithm can result in only one philosopher eating at a given time, even when all five philosophers are hungry, despite the fact that there are more than enough forks for two philosophers to eat concurrently.

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

² It is all too easy to denigrate Dijkstra from the viewpoint of the year 2012, more than 40 years after the fact. If you still feel the need to denigrate Dijkstra, my advice is to publish something, wait 40 years, and then see how *your* words stood the test of time.

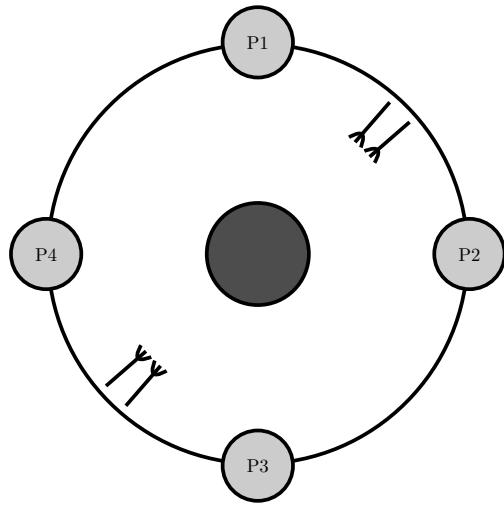


Figure 6.4: Dining Philosophers Problem, Partitioned

One approach is shown in Figure 6.4, 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 always 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 6.1: 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 pairs of philosophers. In a horizontally parallel data-processing system, a given item of data would be processed by only one of a replicated set of software components.

Quick Quiz 6.2: And in just what sense can this “horizontal parallelism” be said to be “horizontal”? ■

6.1.2 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

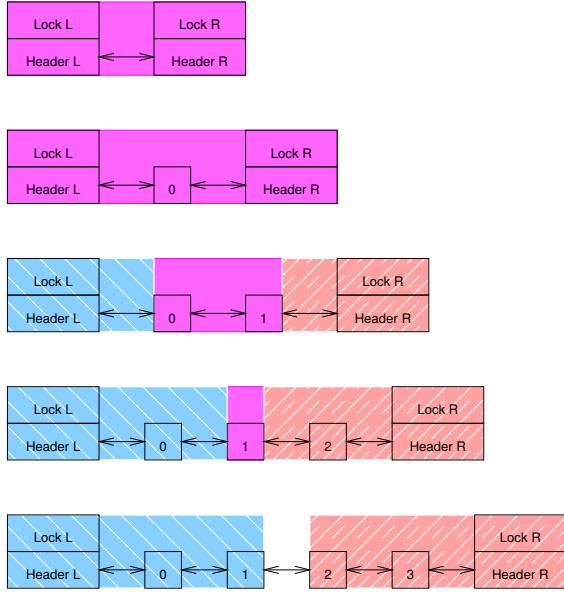


Figure 6.5: Double-Ended Queue With Left- and Right-Hand Locks

in a reasonably simple implementation, looking at three general approaches in the following sections.

6.1.2.1 Left- and Right-Hand Locks

One seemingly straightforward approach would be to use a doubly linked list with 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 6.5. 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 with downward stripes indicating the domain of the left-hand lock, red with upward stripes indicating the domain of the right-hand lock, and purple (with no stripes) 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.

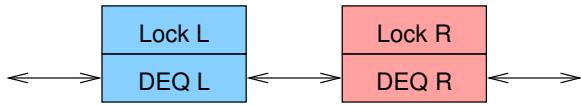


Figure 6.6: Compound Double-Ended Queue

6.1.2.2 Compound Double-Ended Queue

One way of forcing non-overlapping lock domains is shown in Figure 6.6. 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 6.3: 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 resulting code (`locktdeq.c`) is quite straightforward. 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.

6.1.2.3 Hashed Double-Ended Queue

One of the simplest and most effective ways to deterministically partition a data structure is to hash it. It is

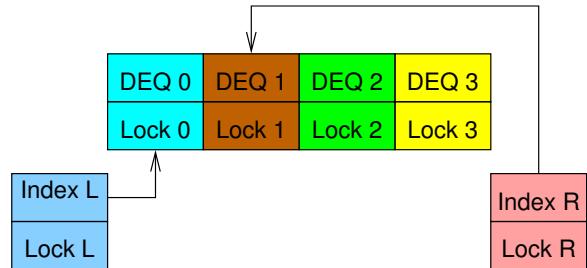


Figure 6.7: Hashed Double-Ended Queue

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 6.7 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 6.8 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 (see Figure 6.7), 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 6.8, a left-dequeue operation would return element "L-2" and leave 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

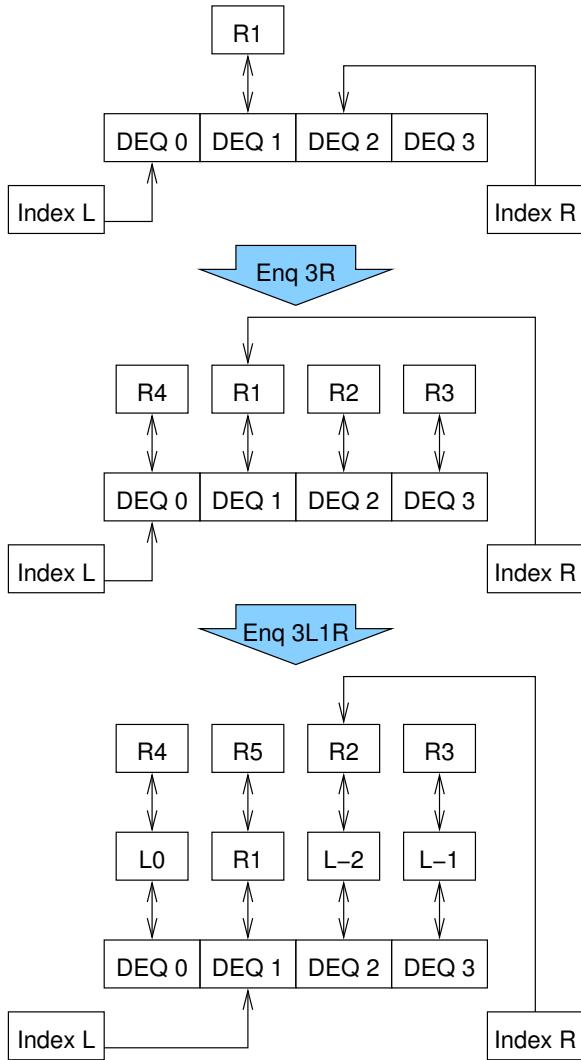


Figure 6.8: Hashed Double-Ended Queue After Insertions

result in lock contention, but the probability of such contention can be reduced to arbitrarily low levels by using a larger hash table.

Figure 6.9 shows how 12 elements would be organized in a four-hash-bucket parallel double-ended queue. Each underlying single-lock double-ended queue holds a one-quarter slice of the full parallel double-ended queue.

Figure 6.10 shows the corresponding C-language data structure, assuming an existing `struct deq` that provides a trivially locked double-ended-queue implementation. This data structure contains the left-hand lock on line 2, the left-hand index on line 3, the right-hand lock

R4	R5	R6	R7
L0	R1	R2	R3
L-4	L-3	L-2	L-1
L-8	L-7	L-6	L-5

Figure 6.9: Hashed Double-Ended Queue With 12 Elements

```

1 struct pdeq {
2     spinlock_t llock;
3     int lidx;
4     spinlock_t rlock;
5     int ridx;
6     struct deq bkt[DEQ_N_BKTS];
7 };

```

Figure 6.10: Lock-Based Parallel Double-Ended Queue Data Structure

on line 4 (which is cache-aligned in the actual implementation), the right-hand index on line 5, and, finally, the hashed array of simple lock-based double-ended queues on line 6. A high-performance implementation would of course use padding or special alignment directives to avoid false sharing.

Figure 6.11 (`lockhdeq.c`) shows the implementation of the enqueue and dequeue functions.³ Discussion will focus on the left-hand operations, as the right-hand operations are trivially derived from them.

Lines 1-13 show `pdeq_pop_1()`, 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 `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 29-38 shows `pdeq_push_1()`, which left-enqueues the specified element. Line 33 acquires the left-hand lock, and line 34 picks up the left-hand index. Line 35 left-enqueues the specified element onto the double-ended queue indexed by the left-hand index. Line 36 then updates the left-hand index and line 37 releases the lock.

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

³ One could easily create a polymorphic implementation in any number of languages, but doing so is left as an exercise for the reader.

```

1 struct cds_list_head *pdeq_pop_l(struct pdeq *d)
2 {
3     struct cds_list_head *e;
4     int i;
5
6     spin_lock(&d->llock);
7     i = moveright(d->lidx);
8     e = deq_pop_l(&d->bkt[i]);
9     if (e != NULL)
10        d->lidx = i;
11     spin_unlock(&d->llock);
12     return e;
13 }
14
15 struct cds_list_head *pdeq_pop_r(struct pdeq *d)
16 {
17     struct cds_list_head *e;
18     int i;
19
20     spin_lock(&d->rlock);
21     i = moveleft(d->ridx);
22     e = deq_pop_r(&d->bkt[i]);
23     if (e != NULL)
24        d->ridx = i;
25     spin_unlock(&d->rlock);
26     return e;
27 }
28
29 void pdeq_push_l(struct cds_list_head *e, struct pdeq *d)
30 {
31     int i;
32
33     spin_lock(&d->llock);
34     i = d->lidx;
35     deq_push_l(e, &d->bkt[i]);
36     d->lidx = moveleft(d->lidx);
37     spin_unlock(&d->llock);
38 }
39
40 void pdeq_push_r(struct cds_list_head *e, struct pdeq *d)
41 {
42     int i;
43
44     spin_lock(&d->rlock);
45     i = d->ridx;
46     deq_push_r(e, &d->bkt[i]);
47     d->ridx = moveright(d->ridx);
48     spin_unlock(&d->rlock);
49 }

```

Figure 6.11: Lock-Based Parallel Double-Ended Queue Implementation

their analysis is left as an exercise for the reader.

Quick Quiz 6.4: Is the hashed double-ended queue a good solution? Why or why not? ■

6.1.2.4 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 6.5: Move *all* the elements to the queue that became empty? In what possible universe is this brain-dead 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 6.12 shows the implementation. Unlike the hashed implementation, this compound implementation is asymmetric, so that we must consider the `pdeq_pop_l()` and `pdeq_pop_r()` implementations separately.

Quick Quiz 6.6: Why can't the compound parallel double-ended queue implementation be symmetric? ■

The `pdeq_pop_l()` implementation is shown on lines 1-16 of the figure. Line 5 acquires the left-hand lock, which line 14 releases. Line 6 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 8 acquires the right-hand lock, line 9 left-dequeues an element from the right-hand queue, and line 10 moves any remaining elements on the right-hand queue to the left-hand queue, line 11 initializes the right-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_pop_r()` implementation is shown on lines 18-38 of the figure. As before, line 22 acquires the right-hand lock (and line 36 releases it), and line 23 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 24 determines that there was no element to dequeue, line 25 releases the right-hand lock and lines 26-27 acquire both locks in the proper order. Line 28 then attempts to right-dequeue an element from the right-hand list again, and if line 29 determines that this second attempt has failed, line 30 right-dequeues an element from the left-hand queue (if there is one available), line 31 moves any remaining elements from the left-hand queue to the right-hand queue, and line 32 initializes the left-hand queue. Either way, line 34 releases the left-hand lock.

Quick Quiz 6.7: Why is it necessary to retry the right-dequeue operation on line 28 of Figure 6.12? ■

Quick Quiz 6.8: Surely the left-hand lock must *sometimes* be available!!! So why is it necessary that line 25 of Figure 6.12 unconditionally release the right-hand lock? ■

The `pdeq_push_l()` implementation is shown on lines 40-47 of Figure 6.12. 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.

6.1.2.5 Double-Ended Queue Discussion

The compound implementation is somewhat more complex than the hashed variant presented in Section 6.1.2.3, 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 [DCW¹¹] and even compared to algorithms using hardware assist [DLM¹⁰]. 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. This limitation also applies to algorithms based on non-blocking synchronization, such as the compare-and-swap-based dequeue algorithm of Michael [Mic03].⁴

Quick Quiz 6.9: Why are there not one but two solutions to the double-ended queue problem? ■

In fact, as noted by Dice et al. [DLM¹⁰], an unsynchronized single-threaded double-ended queue significantly outperforms any of the parallel implementations they studied. Therefore, the key point is that there can be significant overhead enqueueing to or dequeuing from a shared queue, regardless of implementation. This should come as no surprise given the material in Chapter 3, given the strict FIFO nature of these queues.

Furthermore, these strict FIFO queues are strictly FIFO only with respect to *linearization points* [HW90]⁵ that are not visible to the caller, in fact, in these examples, the linearization points are buried in the lock-based critical sections. These queues are not strictly FIFO with

⁴ This paper is interesting in that it showed that special double-compare-and-swap (DCAS) instructions are not needed for lock-free implementations of double-ended queues. Instead, the common compare-and-swap (e.g., x86 `cmpxchg`) suffices.

⁵ In short, a linearization point is a single point within a given function where that function can be said to have taken effect. In this lock-based implementation, the linearization points can be said to be anywhere within the critical section that does the work.

```

1 struct cds_list_head *pdeq_pop_l(struct pdeq *d)
2 {
3     struct cds_list_head *e;
4
5     spin_lock(&d->llock);
6     e = deq_pop_l(&d->ldeq);
7     if (e == NULL) {
8         spin_lock(&d->rlock);
9         e = deq_pop_l(&d->rdeq);
10    cds_list_splice(&d->rdeq.chain, &d->ldeq.chain);
11    CDS_INIT_LIST_HEAD(&d->rdeq.chain);
12    spin_unlock(&d->rlock);
13 }
14 spin_unlock(&d->llock);
15 return e;
16 }
17
18 struct cds_list_head *pdeq_pop_r(struct pdeq *d)
19 {
20     struct cds_list_head *e;
21
22     spin_lock(&d->rlock);
23     e = deq_pop_r(&d->rdeq);
24     if (e == NULL) {
25         spin_unlock(&d->rlock);
26         spin_lock(&d->llock);
27         spin_lock(&d->rlock);
28         e = deq_pop_r(&d->ldeq);
29         if (e == NULL) {
30             e = deq_pop_r(&d->ldeq);
31             cds_list_splice(&d->ldeq.chain, &d->rdeq.chain);
32             CDS_INIT_LIST_HEAD(&d->ldeq.chain);
33         }
34         spin_unlock(&d->llock);
35     }
36     spin_unlock(&d->rlock);
37     return e;
38 }
39
40 void pdeq_push_l(struct cds_list_head *e, struct pdeq *d)
41 {
42     int i;
43
44     spin_lock(&d->llock);
45     deq_push_l(e, &d->ldeq);
46     spin_unlock(&d->llock);
47 }
48
49 void pdeq_push_r(struct cds_list_head *e, struct pdeq *d)
50 {
51     int i;
52
53     spin_lock(&d->rlock);
54     deq_push_r(e, &d->rdeq);
55     spin_unlock(&d->rlock);
56 }

```

Figure 6.12: Compound Parallel Double-Ended Queue Implementation

respect to (say) the times at which the individual operations started [HKLP12]. This indicates that the strict FIFO property is not all that valuable in concurrent programs, and in fact, Kirsch et al. present less-strict queues that provide improved performance and scalability [KLP12].⁶ All that said, if you are pushing all the data used by your concurrent program through a single queue, you really need to rethink your overall design.

6.1.3 Partitioning Example Discussion

The optimal solution to the dining philosophers problem given in the answer to the Quick Quiz in Section 6.1.1 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 6.10: 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 6.11: 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. Section 6.3.5 looks briefly at a third example, matrix multiply. However, all three of these examples beg for more and better design criteria for parallel programs, a topic taken up in the next section.

6.2 Design Criteria

One way to obtain the best performance and scalability is to simply hack away until you converge on the best possible parallel program. Unfortunately, if your program is other than microscopically tiny, the space of possible parallel programs is so huge that convergence is not guaranteed in the lifetime of the universe. Besides, what exactly is the “best possible parallel program”? After

⁶ Nir Shavit produced relaxed stacks for roughly the same reasons [Sha11]. This situation leads some to believe that the linearization points are useful to theorists rather than developers, and leads others to wonder to what extent the designers of such data structures and algorithms were considering the needs of their users.

all, Section 2.2 called out no fewer than three parallel-programming goals of performance, productivity, and generality, and the best possible performance will likely come at a cost in terms of productivity and generality. We clearly need to be able to make higher-level choices at design time in order to arrive at an acceptably good parallel program before that program becomes obsolete.

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 2.2, 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-interruptible⁷ 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.

⁷ Either by masking interrupts or by being oblivious to them.

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 5.

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. Worse yet, added complexity can actually *reduce* performance and scalability.

Therefore, beyond a certain point, there may be potential sequential optimizations that are cheaper and more effective than parallelization. As noted in Section 2.2.1, 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.⁸

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

⁸ A real-world parallel system will be subject to many additional design criteria, such as data-structure layout, memory size, memory-hierarchy latencies, bandwidth limitations, and I/O issues.

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 that only one CPU may execute within a given critical section at a given time.

More specifically, 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.

2. 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.
3. 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 asymmetric primitives (see Section 9), or by moving toward a more coarse-grained design such as code locking).
4. 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, asymmetric, or data ownership.
5. 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 asymmetric primitives.
6. Many changes that improve SMP performance, for example, reducing lock contention, also improve real-time latencies [McK05c].

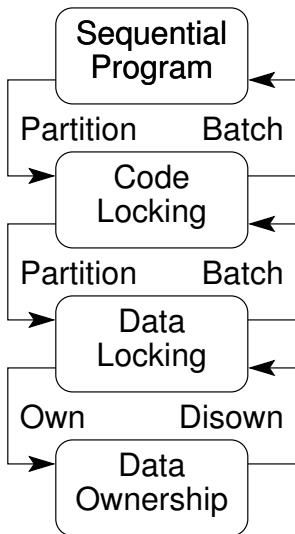


Figure 6.13: Design Patterns and Lock Granularity

Quick Quiz 6.12: Don't all these problems with critical sections mean that we should just always use non-blocking synchronization [Her90], which don't have critical sections? ■

6.3 Synchronization Granularity

Figure 6.13 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.

6.3.1 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, as can be seen in Figure 6.14, the exponential increase in single-threaded performance halted in about 2003. Therefore, increasing performance will increasingly require parallelism.⁹ The debate as to

⁹ 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

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 6.15. This growth will motivate multi-threaded servers in order to handle the communications load.

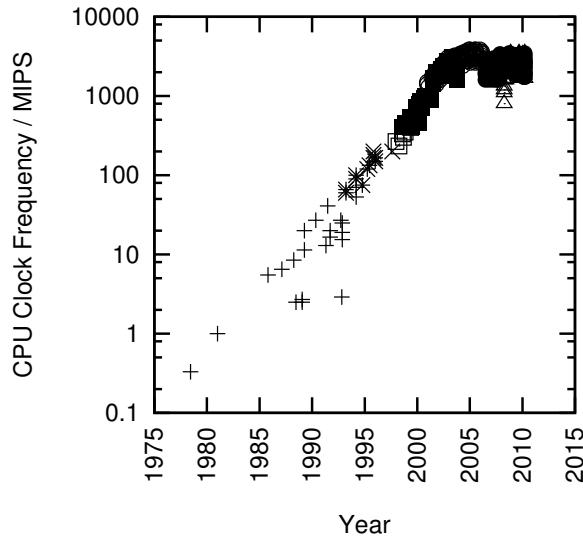


Figure 6.14: MIPS/Clock-Frequency Trend for Intel CPUs

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 processor, spare yourself the overhead and complexity of SMP synchronization primitives. The simplicity of the hash-table lookup code in Figure 6.16 underscores this point.¹⁰ A key point is that speedups due to parallelism are normally limited to the number of CPUs. In contrast, speedups due to sequential optimizations, for example, careful choice of data structure, can be arbitrarily large.

On the other hand, if you are not in this happy situation, read on!

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.

¹⁰ The examples in this section are taken from Hart et al. [HMB06], adapted for clarity by gathering related code from multiple files.

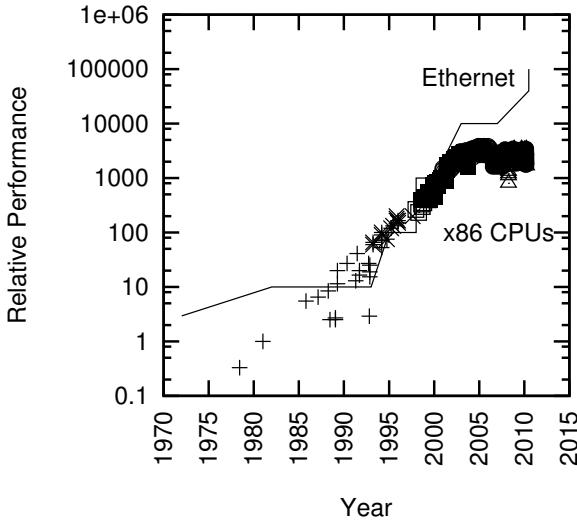


Figure 6.15: Ethernet Bandwidth vs. Intel x86 CPU Performance

```

1 struct hash_table
2 {
3     long nbuckets;
4     struct node **buckets;
5 };
6
7 typedef struct node {
8     unsigned long key;
9     struct node *next;
10 } node_t;
11
12 int hash_search(struct hash_table *h, long key)
13 {
14     struct node *cur;
15
16     cur = h->buckets[key % h->nbuckets];
17     while (cur != NULL) {
18         if (cur->key >= key) {
19             if (cur->key == key)
20                 return (cur->key == key);
21             cur = cur->next;
22         }
23     }
24     return 0;
25 }
```

Figure 6.16: Sequential-Program Hash Table Search

6.3.2 Code Locking

Code locking is quite simple due to the fact that it uses only global locks.¹¹ 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 6.17. However, note that the simple return of the comparison in `hash_search()` in Figure 6.16 has now become three statements due to the need to release the lock before returning.

Unfortunately, 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 groups of older people who are acting like children) will immediately recognize

```

1 spinlock_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     spin_lock(&hash_lock);
20     cur = h->buckets[key % h->nbuckets];
21     while (cur != NULL) {
22         if (cur->key >= key) {
23             if (cur->key == key)
24                 spin_unlock(&hash_lock);
25             return retval;
26         }
27         cur = cur->next;
28     }
29     spin_unlock(&hash_lock);
30     return 0;
31 }
```

Figure 6.17: Code-Locking Hash Table Search

¹¹ 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 6.3.3.

the danger of having only one of something, as illustrated in Figure 6.18.

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

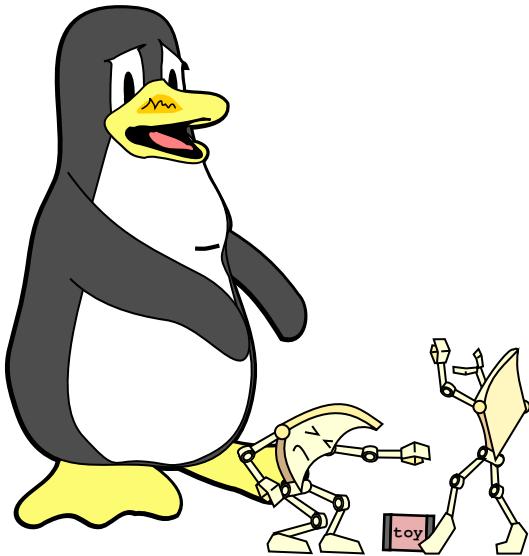


Figure 6.18: Lock Contention

6.3.3 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. You should 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 across multiple data structures, for example, maintaining per-hash-bucket critical sections in a hash table, as shown in Figure 6.19. The increased scalability again results in a slight increase in complexity in the form of an additional data structure, the `struct bucket`.

In contrast with the contentious situation shown in Figure 6.18, data locking helps promote harmony, as illustrated by Figure 6.20 — and in parallel programs, this *almost* always translates into increased performance and scalability. For this reason, data locking was heavily used

```

1 struct hash_table
2 {
3     long nbuckets;
4     struct bucket **buckets;
5 };
6
7 struct bucket {
8     spinlock_t bucket_lock;
9     node_t *list_head;
10 };
11
12 typedef struct node {
13     unsigned long key;
14     struct node *next;
15 } node_t;
16
17 int hash_search(struct hash_table *h, long key)
18 {
19     struct bucket *bp;
20     struct node *cur;
21     int retval;
22
23     bp = h->buckets[key % h->nbuckets];
24     spin_lock(&bp->bucket_lock);
25     cur = bp->list_head;
26     while (cur != NULL) {
27         if (cur->key >= key) {
28             retval = (cur->key == key);
29             spin_unlock(&bp->bucket_lock);
30             return retval;
31         }
32         cur = cur->next;
33     }
34     spin_unlock(&bp->bucket_lock);
35     return 0;
36 }
```

Figure 6.19: Data-Locking Hash Table Search

by Sequent in both its DYNIX and DYNIX/ptx operating systems [BK85, Inm85, Gar90, Dov90, MD92, MG92, MS93].

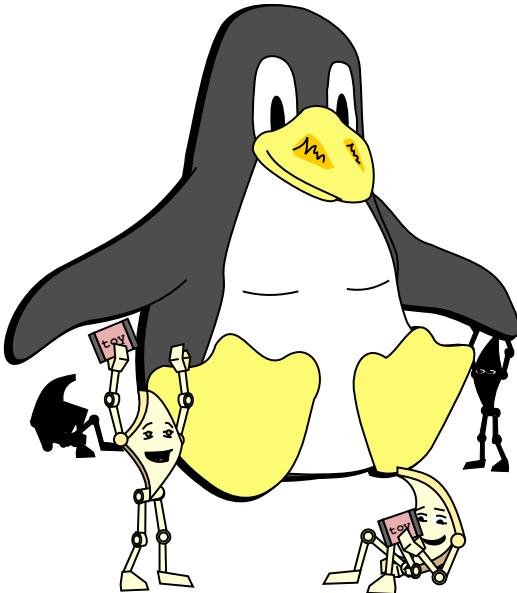


Figure 6.20: Data Locking

However, as those who 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 6.21.

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 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

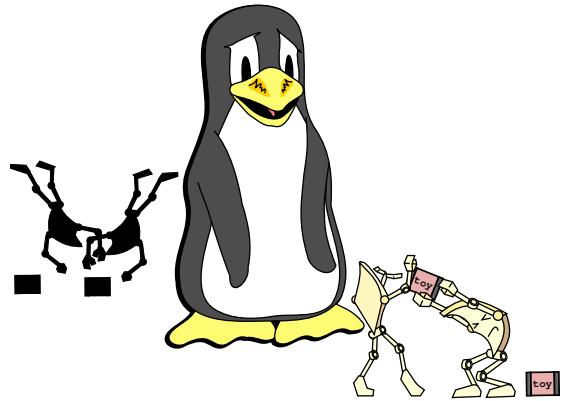


Figure 6.21: Data Locking and Skew

in existence while the lock is being acquired. The code in Figure 6.19 finesse 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 resizable, 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 6.13: What are some ways of preventing a structure from being freed while its lock is being acquired? ■

6.3.4 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 performs the operation on behalf of the first, or, alternatively, migrates the data to the first thread.

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. There are also computing frameworks designed for this type of problem [UoC08].

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 6.21. 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 6.16. Such situations are often referred to as “embarrassingly parallel”, and, in the best case, resemble the situation previously shown in Figure 6.20.

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 8.

6.3.5 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 queuing models are based on an exponentially distributed “inter-arrival rate” λ and an exponentially distributed “service rate” μ . The inter-arrival rate λ 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, λ is an inverse measure of the overhead of each non-synchronization unit of work. For example, if each unit of work was a transaction, and if each transaction took one millisecond to process, excluding synchronization overhead, then λ would be 1,000 transactions per second.

The service rate μ 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 synchronization operations, in other words, μ can be roughly thought of as the synchronization overhead in absence of contention. For example, suppose that each synchronization operation involves an atomic increment instruction, and that a computer system is able to do an atomic increment every 25 nanoseconds on each CPU to a private variable.¹² The value of μ is therefore about 40,000,000 atomic increments per second.

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

$$\lambda = n\lambda_0 \quad (6.1)$$

where n is the number of CPUs and λ_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:

$$T = \frac{1}{\mu - \lambda} \quad (6.2)$$

Substituting the above value of λ :

$$T = \frac{1}{\mu - n\lambda_0} \quad (6.3)$$

Now, the efficiency is just the ratio of the time required to process a transaction in absence of synchronization ($1/\lambda_0$) to the time required including synchronization ($T + 1/\lambda_0$):

$$e = \frac{1/\lambda_0}{T + 1/\lambda_0} \quad (6.4)$$

Substituting the above value for T and simplifying:

¹² Of course, if there are 8 CPUs all incrementing the same shared variable, then each CPU must wait at least 175 nanoseconds for each of the other CPUs to do its increment before consuming an additional 25 nanoseconds doing its own increment. In actual fact, the wait will be longer due to the need to move the variable from one CPU to another.

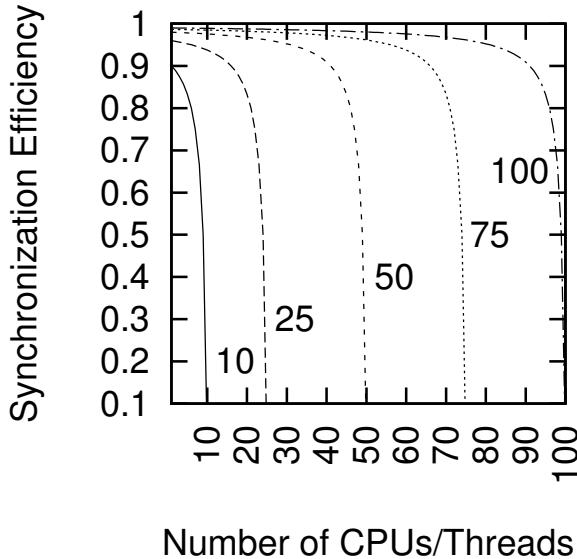


Figure 6.22: Synchronization Efficiency

$$e = \frac{\frac{\mu}{\lambda_0} - n}{\frac{\mu}{\lambda_0} - (n-1)} \quad (6.5)$$

But the value of μ/λ_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:

$$e = \frac{f - n}{f - (n-1)} \quad (6.6)$$

Figure 6.22 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 5.

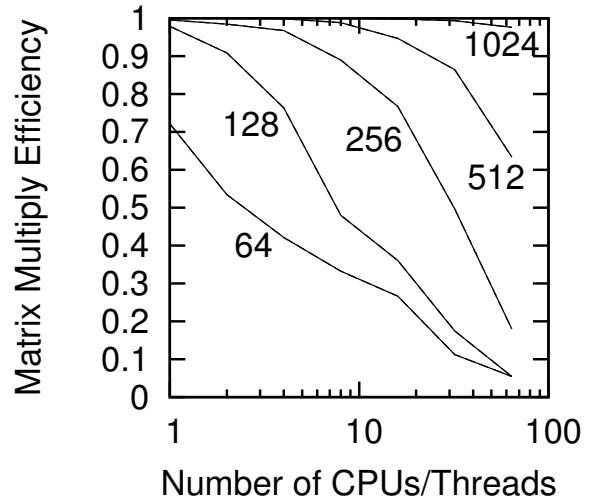


Figure 6.23: Matrix Multiply Efficiency

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 6.23 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. Nevertheless, this figure clearly demonstrates the performance and scalability benefits of batching: If you must incur synchronization overhead, you may as well get your money’s worth.

Quick Quiz 6.14: 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 6.23 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 6.3.3 or the parallel-fastpath approach discussed in the next section.

Quick Quiz 6.15: How are data-parallel techniques going to help with matrix multiply? It is *already* data parallel!!! ■

6.4 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 6.24:

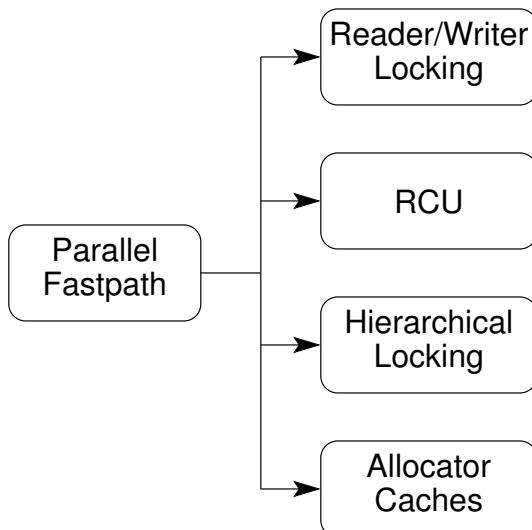


Figure 6.24: Parallel-Fastpath Design Patterns

1. Reader/Writer Locking (described below in Section 6.4.1).
2. Read-copy update (RCU), which may be used as a high-performance replacement for reader/writer locking, is introduced in Section 9.3, and will not be discussed further in this chapter.
3. Hierarchical Locking ([McK96a]), which is touched upon in Section 6.4.2.
4. Resource Allocator Caches ([McK96a, MS93]). See Section 6.4.3 for more detail.

6.4.1 Reader/Writer Locking

If synchronization overhead is negligible (for example, if the program uses coarse-grained parallelism with large critical sections), 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. There are many implementations of reader-writer locking, including the POSIX implementation described in Section 4.2.4. Figure 6.25 shows how the hash search might be implemented using reader-writer locking.

```

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             retval = (cur->key == key);
24             read_unlock(&hash_lock);
25             return retval;
26         }
27         cur = cur->next;
28     }
29     read_unlock(&hash_lock);
30     return 0;
31 }
  
```

Figure 6.25: Reader-Writer-Locking Hash Table Search

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 systems. Locking in general and reader-writer locking in particular is described extensively in Chapter 7.

6.4.2 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 6.26 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.

```

1 struct hash_table
2 {
3     long nbuckets;
4     struct bucket **buckets;
5 };
6
7 struct bucket {
8     spinlock_t bucket_lock;
9     node_t *list_head;
10 };
11
12 typedef struct node {
13     spinlock_t node_lock;
14     unsigned long key;
15     struct node *next;
16 } node_t;
17
18 int hash_search(struct hash_table *h, long key)
19 {
20     struct bucket *bp;
21     struct node *cur;
22     int retval;
23
24     bp = h->buckets[key % h->nbuckets];
25     spin_lock(&bp->bucket_lock);
26     cur = bp->list_head;
27     while (cur != NULL) {
28         if (cur->key >= key) {
29             spin_lock(&cur->node_lock);
30             spin_unlock(&bp->bucket_lock);
31             retval = (cur->key == key);
32             spin_unlock(&cur->node_lock);
33             return retval;
34         }
35         cur = cur->next;
36     }
37     spin_unlock(&bp->bucket_lock);
38     return 0;
39 }
```

Figure 6.26: Hierarchical-Locking Hash Table Search

Quick Quiz 6.16: In what situation would hierarchical locking work well? ■

6.4.3 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 [Tor03].

6.4.3.1 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].

6.4.3.2 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 6.27: 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.

6.4.3.3 Data Structures

The actual data structures for a “toy” implementation of allocator caches are shown in Figure 6.28. The “Global Pool” of Figure 6.27 is implemented by `globalmem` of type `struct globalmempool`, and the two CPU pools by the per-CPU variable `percpumem`

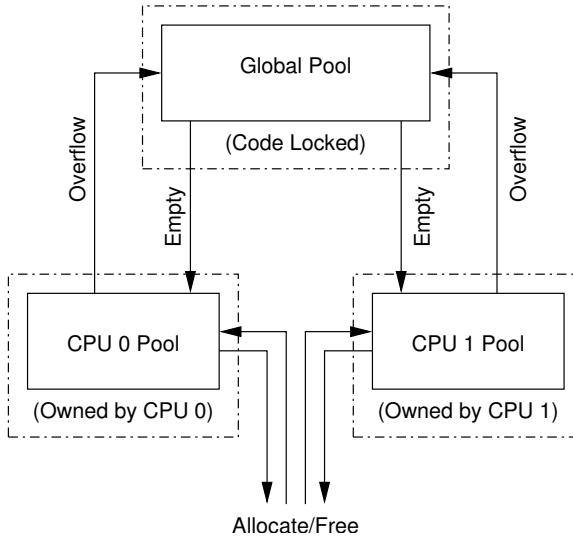


Figure 6.27: Allocator Cache Schematic

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.¹³

```

1 #define TARGET_POOL_SIZE 3
2 #define GLOBAL_POOL_SIZE 40
3
4 struct globalmempool {
5     spinlock_t mutex;
6     int cur;
7     struct memblock *pool[GLOBAL_POOL_SIZE];
8 } globalmem;
9
10 struct percpumempool {
11     int cur;
12     struct memblock *pool[2 * TARGET_POOL_SIZE];
13 };
14
15 DEFINE_PER_THREAD(struct percpumempool, percpumem);

```

Figure 6.28: Allocator-Cache Data Structures

The operation of the pool data structures is illustrated

¹³ 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.

by Figure 6.29, 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` pointers.

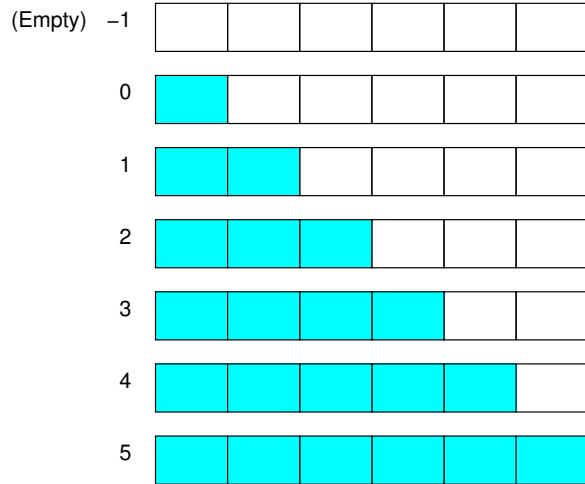


Figure 6.29: Allocator Pool Schematic

6.4.3.4 Allocation Function

The allocation function `memblock_alloc()` may be seen in Figure 6.30. 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.

6.4.3.5 Free Function

Figure 6.31 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.

```

1 struct memblock *memblock_alloc(void)
2 {
3     int i;
4     struct memblock *p;
5     struct percpumempool *pcpp;
6
7     pcpp = &__get_thread_var(percpumem);
8     if (pcpp->cur < 0) {
9         spin_lock(&globalmem.mutex);
10        for (i = 0; i < TARGET_POOL_SIZE &&
11             globalmem.cur >= 0; i++) {
12            pcpp->pool[i] = globalmem.pool[globalmem.cur];
13            globalmem.pool[globalmem.cur--] = NULL;
14        }
15        pcpp->cur = i - 1;
16        spin_unlock(&globalmem.mutex);
17    }
18    if (pcpp->cur >= 0) {
19        p = pcpp->pool[pcpp->cur];
20        pcpp->pool[pcpp->cur--] = NULL;
21        return p;
22    }
23    return NULL;
24 }

```

Figure 6.30: Allocator-Cache Allocator Function

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.

```

1 void memblock_free(struct memblock *p)
2 {
3     int i;
4     struct percpumempool *pcpp;
5
6     pcpp = &__get_thread_var(percpumem);
7     if (pcpp->cur >= 2 * TARGET_POOL_SIZE - 1) {
8         spin_lock(&globalmem.mutex);
9         for (i = pcpp->cur; i >= TARGET_POOL_SIZE; i--) {
10            globalmem.pool[+globalmem.cur] = pcpp->pool[i];
11            pcpp->pool[i] = NULL;
12        }
13        pcpp->cur = i;
14        spin_unlock(&globalmem.mutex);
15    }
16    pcpp->pool[+pcpp->cur] = p;
17 }

```

Figure 6.31: Allocator-Cache Free Function

6.4.3.6 Performance

Rough performance results¹⁴ are shown in Figure 6.32, 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 all the blocks in that group, with the number of blocks in 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.

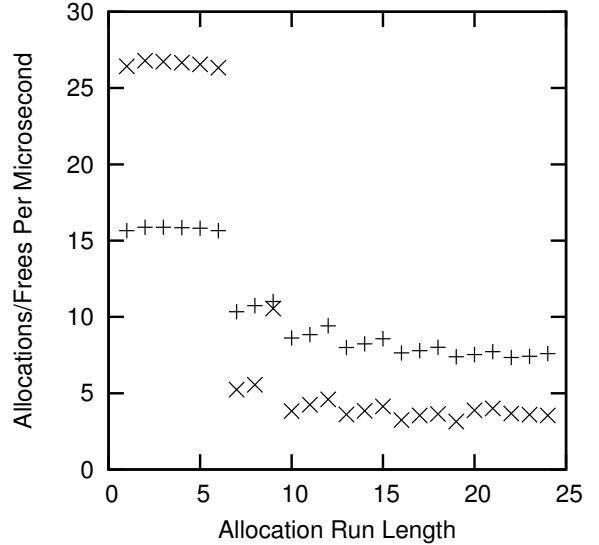


Figure 6.32: Allocator Cache Performance

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.

As can be seen from the figure, the situations where the common-case data-ownership applies (run lengths up to six) provide greatly improved performance compared to the cases where locks must be acquired. Avoiding synchronization in the common case will be a recurring theme

¹⁴ 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 11. That said, repeated runs gave similar results, and these results match more careful evaluations of similar algorithms.

through this book.

Quick Quiz 6.17: In Figure 6.32, 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 6.18: 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-thread target pool size s of three, number of threads n equal to two, and assuming that the per-thread pools are initially empty with none of the memory in use, what is the smallest allocation run length m at which failures can occur? (Recall that each thread repeatedly allocates m block of memory, and then frees the m blocks of memory.) Alternatively, given n threads each with pool size s , and where each thread repeatedly first allocates m blocks of memory and then frees those m blocks, how large must the global pool size be? *Note:* Obtaining the correct answer will require you to examine the `smpalloc.c` source code, and very likely single-step it as well. You have been warned! ■

6.4.3.7 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 6.1.

Level	Locking	Purpose
Per-thread pool	Data ownership	High-speed allocation
Global block pool	Data locking	Distributing blocks among threads
Coalescing	Data locking	Combining blocks into pages
System memory	Code locking	Memory from/to system

Table 6.1: Schematic of Real-World Parallel Allocator

6.5 Beyond Partitioning

This chapter has discussed how data partitioning can be used to design simple linearly scalable parallel programs. Section 6.3.4 hinted at the possibilities of data replication, which will be used to great effect in Section 9.3.

The main goal of applying partitioning and replication is to achieve linear speedups, in other words, to ensure that the total amount of work required does not increase significantly as the number of CPUs or threads increases. A problem that can be solved via partitioning and/or replication, resulting in linear speedups, is *embarrassingly parallel*. But can we do better?

To answer this question, let us examine the solution of labyrinths and mazes. Of course, labyrinths and mazes have been objects of fascination for millenia [Wik12], so it should come as no surprise that they are generated and solved using computers, including biological computers [Ada11], GPGPUs [Eri08], and even discrete hardware [KFC11]. Parallel solution of mazes is sometimes used as a class project in universities [ETH11, Uni10] and as a vehicle to demonstrate the benefits of parallel-programming frameworks [Fos10].

Common advice is to use a parallel work-queue algorithm (PWQ) [ETH11, Fos10]. This section evaluates this advice by comparing PWQ against a sequential algorithm (SEQ) and also against an alternative parallel algorithm, in all cases solving randomly generated square mazes. Section 6.5.1 discusses PWQ, Section 6.5.2 discusses an alternative parallel algorithm, Section 6.5.3 analyzes its anomalous performance, Section 6.5.4 derives an improved sequential algorithm from the alternative parallel algorithm, Section 6.5.5 makes further performance comparisons, and finally Section 6.5.6 presents future directions and concluding remarks.

```

1 int maze_solve(maze *mp, cell sc, cell ec)
2 {
3     cell c = sc;
4     cell n;
5     int vi = 0;
6
7     maze_try_visit_cell(mp, c, c, &n, 1);
8     for (;;) {
9         while (!maze_find_any_next_cell(mp, c, &n)) {
10            if (++vi >= mp->vi)
11                return 0;
12            c = mp->visited[vi].c;
13        }
14        do {
15            if (n == ec) {
16                return 1;
17            }
18            c = n;
19        } while (maze_find_any_next_cell(mp, c, &n));
20        c = mp->visited[vi].c;
21    }
22 }
```

Figure 6.33: SEQ Pseudocode

6.5.1 Work-Queue Parallel Maze Solver

PWQ is based on SEQ, which is shown in Figure 6.33 (`maze_seq.c`). The maze is represented by a 2D array of cells and a linear-array-based work queue named `->visited`.

Line 7 visits the initial cell, and each iteration of the loop spanning lines 8-21 traverses passages headed by one cell. The loop spanning lines 9-13 scans the `->visited[]` array for a visited cell with an unvisited neighbor, and the loop spanning lines 14-19 traverses one fork of the submaze headed by that neighbor. Line 20 initializes for the next pass through the outer loop.

The pseudocode for `maze_try_visit_cell()` is shown on lines 1-12 of Figure 6.34. Line 4 checks to see if cells `c` and `n` are adjacent and connected, while line 5 checks to see if cell `n` has not yet been visited. The `celladdr()` function returns the address of the specified cell. If either check fails, line 6 returns failure. Line 7 indicates the next cell, line 8 records this cell in the next slot of the `->visited[]` array, line 9 indicates that this slot is now full, and line 10 marks this cell as visited and also records the distance from the maze start. Line 11 then returns success.

The pseudocode for `maze_find_any_next_cell()` is shown on lines 14-28 of the figure (`maze.c`). Line 17 picks up the current cell's distance plus 1, while lines 19, 21, 23, and 25 check the cell in each direction, and lines 20, 22, 24, and 26 return true if the corresponding cell is a candidate next cell. The `prevcol()`, `nextcol()`, `prevrow()`, and `nextrow()` each do

```

1 int maze_try_visit_cell(struct maze *mp, cell c, cell t,
2                         cell *n, int d)
3 {
4     if (!maze_cells_connected(mp, c, t) ||
5         (*celladdr(mp, t) & VISITED))
6         return 0;
7     *n = t;
8     mp->visited[mp->vi] = t;
9     mp->vi++;
10    *celladdr(mp, t) |= VISITED | d;
11    return 1;
12 }
13
14 int maze_find_any_next_cell(struct maze *mp, cell c,
15                             cell *n)
16 {
17     int d = (*celladdr(mp, c) & DISTANCE) + 1;
18
19     if (maze_try_visit_cell(mp, c, prevcol(c), n, d))
20         return 1;
21     if (maze_try_visit_cell(mp, c, nextcol(c), n, d))
22         return 1;
23     if (maze_try_visit_cell(mp, c, prevrow(c), n, d))
24         return 1;
25     if (maze_try_visit_cell(mp, c, nextrow(c), n, d))
26         return 1;
27     return 0;
28 }
```

Figure 6.34: SEQ Helper Pseudocode

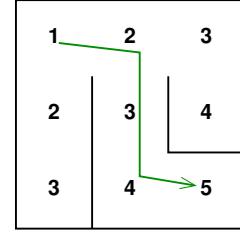


Figure 6.35: Cell-Number Solution Tracking

the specified array-index-conversion operation. If none of the cells is a candidate, line 27 returns false.

The path is recorded in the maze by counting the number of cells from the starting point, as shown in Figure 6.35, where the starting cell is in the upper left and the ending cell is in the lower right. Starting at the ending cell and following consecutively decreasing cell numbers traverses the solution.

The parallel work-queue solver is a straightforward parallelization of the algorithm shown in Figures 6.33 and 6.34. Line 10 of Figure 6.33 must use fetch-and-add, and the local variable `vi` must be shared among the various threads. Lines 5 and 10 of Figure 6.34 must be combined into a CAS loop, with CAS failure indicating a loop in the maze. Lines 8-9 of this figure must use fetch-and-

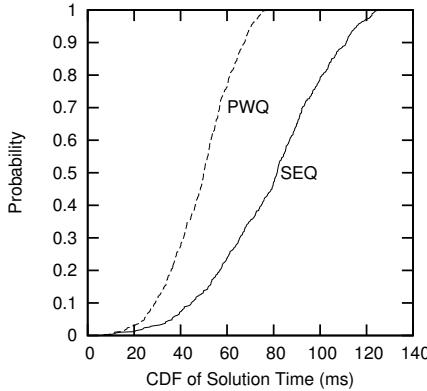


Figure 6.36: CDF of Solution Times For SEQ and PWQ

add to arbitrate concurrent attempts to record cells in the `->visited[]` array.

This approach does provide significant speedups on a dual-CPU Lenovo™ W500 running at 2.53GHz, as shown in Figure 6.36, which shows the cumulative distribution functions (CDFs) for the solution times of the two algorithms, based on the solution of 500 different square 500-by-500 randomly generated mazes. The substantial overlap of the projection of the CDFs onto the x-axis will be addressed in Section 6.5.3.

Interestingly enough, the sequential solution-path tracking works unchanged for the parallel algorithm. However, this uncovers a significant weakness in the parallel algorithm: At most one thread may be making progress along the solution path at any given time. This weakness is addressed in the next section.

6.5.2 Alternative Parallel Maze Solver

Youthful maze solvers are often urged to start at both ends, and this advice has been repeated more recently in the context of automated maze solving [Uni10]. This advice amounts to partitioning, which has been a powerful parallelization strategy in the context of parallel programming for both operating-system kernels [BK85, Inm85] and applications [Pat10]. This section applies this strategy, using two child threads that start at opposite ends of the solution path, and takes a brief look at the performance and scalability consequences.

The partitioned parallel algorithm (PART), shown in Figure 6.37 (`maze_part.c`), is similar to SEQ, but has a few important differences. First, each child thread has its own `visited` array, passed in by the parent as shown

```

1 int maze_solve_child(maze *mp, cell *visited, cell sc)
2 {
3     cell c;
4     cell n;
5     int vi = 0;
6
7     myvisited = visited; myvi = &vi;
8     c = visited[vi];
9     do {
10         while (!maze_find_any_next_cell(mp, c, &n)) {
11             if (visited[++vi].row < 0)
12                 return 0;
13             if (ACCESS_ONCE(mp->done))
14                 return 1;
15             c = visited[vi];
16         }
17         do {
18             if (ACCESS_ONCE(mp->done))
19                 return 1;
20             c = n;
21         } while (maze_find_any_next_cell(mp, c, &n));
22         c = visited[vi];
23     } while (!ACCESS_ONCE(mp->done));
24     return 1;
25 }
```

Figure 6.37: Partitioned Parallel Solver Pseudocode

on line 1, which must be initialized to all [-1,-1]. Line 7 stores a pointer to this array into the per-thread variable `myvisited` to allow access by helper functions, and similarly stores a pointer to the local visit index. Second, the parent visits the first cell on each child's behalf, which the child retrieves on line 8. Third, the maze is solved as soon as one child locates a cell that has been visited by the other child. When `maze_try_visit_cell()` detects this, it sets a `->done` field in the maze structure. Fourth, each child must therefore periodically check the `->done` field, as shown on lines 13, 18, and 23. The `ACCESS_ONCE()` primitive must disable any compiler optimizations that might combine consecutive loads or that might reload the value. A C++1x volatile relaxed load suffices [Bec11]. Finally, the `maze_find_any_next_cell()` function must use compare-and-swap to mark a cell as visited, however no constraints on ordering are required beyond those provided by thread creation and join.

The pseudocode for `maze_find_any_next_cell()` is identical to that shown in Figure 6.34, but the pseudocode for `maze_try_visit_cell()` differs, and is shown in Figure 6.38. Lines 8-9 check to see if the cells are connected, returning failure if not. The loop spanning lines 11-18 attempts to mark the new cell visited. Line 13 checks to see if it has already been visited, in which case line 16 returns failure, but only after line 14 checks to see if we have encountered the other thread, in

```

1 int maze_try_visit_cell(struct maze *mp, int c, int t,
2             int *n, int d)
3 {
4     cell_t t;
5     cell_t *tp;
6     int vi;
7
8     if (!maze_cells_connected(mp, c, t))
9         return 0;
10    tp = celladdr(mp, t);
11    do {
12        t = ACCESS_ONCE(*tp);
13        if (t & VISITED) {
14            if ((t & TID) != mytid)
15                mp->done = 1;
16            return 0;
17        }
18    } while (!CAS(tp, t, t | VISITED | myid | d));
19    *n = t;
20    vi = (*myvi)++;
21    myvisited[vi] = t;
22    return 1;
23 }

```

Figure 6.38: Partitioned Parallel Helper Pseudocode

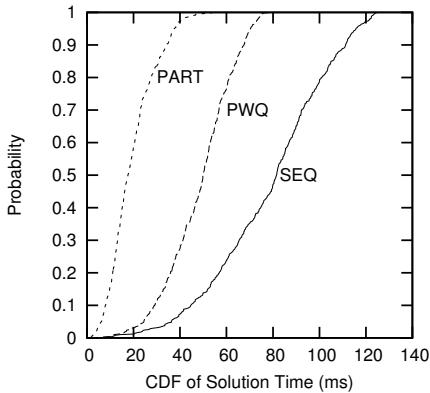


Figure 6.39: CDF of Solution Times For SEQ, PWQ, and PART

which case line 15 indicates that the solution has been located. Line 19 updates to the new cell, lines 20 and 21 update this thread's visited array, and line 22 returns success.

Performance testing revealed a surprising anomaly, shown in Figure 6.39. The median solution time for PART (17 milliseconds) is more than four times faster than that of SEQ (79 milliseconds), despite running on only two threads. The next section analyzes this anomaly.

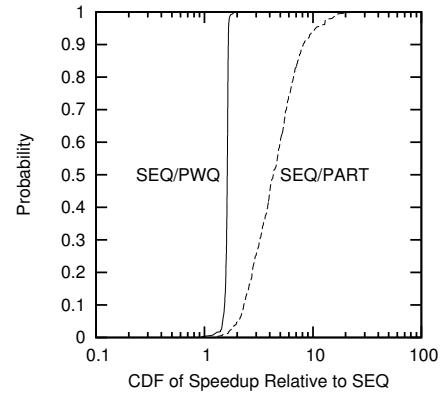


Figure 6.40: CDF of SEQ/PWQ and SEQ/PART Solution-Time Ratios

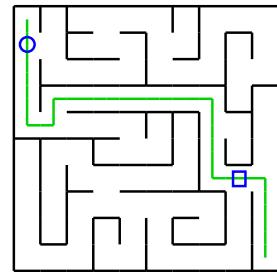


Figure 6.41: Reason for Small Visit Percentages

6.5.3 Performance Comparison I

The first reaction to a performance anomaly is to check for bugs. Although the algorithms were in fact finding valid solutions, the plot of CDFs in Figure 6.39 assumes independent data points. This is not the case: The performance tests randomly generate a maze, and then run all solvers on that maze. It therefore makes sense to plot the CDF of the ratios of solution times for each generated maze, as shown in Figure 6.40, greatly reducing the CDFs' overlap. This plot reveals that for some mazes, PART is more than *forty* times faster than SEQ. In contrast, PWQ is never more than about two times faster than SEQ. A forty-times speedup on two threads demands explanation. After all, this is not merely embarrassingly parallel, where partitionability means that adding threads does not increase the overall computational cost. It is instead *humiliatingly parallel*: Adding threads significantly reduces the overall computational cost, resulting in large algorithmic superlinear speedups.

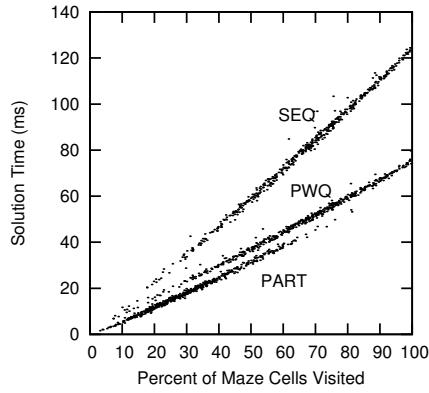


Figure 6.42: Correlation Between Visit Percentage and Solution Time

Further investigation showed that PART sometimes visited fewer than 2% of the maze’s cells, while SEQ and PWQ never visited fewer than about 9%. The reason for this difference is shown by Figure 6.41. If the thread traversing the solution from the upper left reaches the circle, the other thread cannot reach the upper-right portion of the maze. Similarly, if the other thread reaches the square, the first thread cannot reach the lower-left portion of the maze. Therefore, PART will likely visit a small fraction of the non-solution-path cells. In short, the super-linear speedups are due to threads getting in each others’ way. This is a sharp contrast with decades of experience with parallel programming, where workers have struggled to keep threads *out* of each others’ way.

Figure 6.42 confirms a strong correlation between cells visited and solution time for all three methods. The slope of PART’s scatterplot is smaller than that of SEQ, indicating that PART’s pair of threads visits a given fraction of the maze faster than can SEQ’s single thread. PART’s scatterplot is also weighted toward small visit percentages, confirming that PART does less total work, hence the observed humiliating parallelism.

The fraction of cells visited by PWQ is similar to that of SEQ. In addition, PWQ’s solution time is greater than that of PART, even for equal visit fractions. The reason for this is shown in Figure 6.43, which has a red circle on each cell with more than two neighbors. Each such cell can result in contention in PWQ, because one thread can enter but two threads can exit, which hurts performance, as noted earlier in this chapter. In contrast, PART can incur such contention but once, namely when the solution is located. Of course, SEQ never contends.

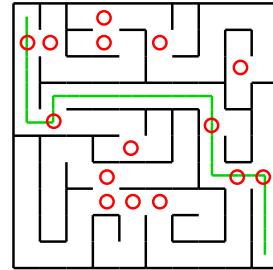


Figure 6.43: PWQ Potential Contention Points

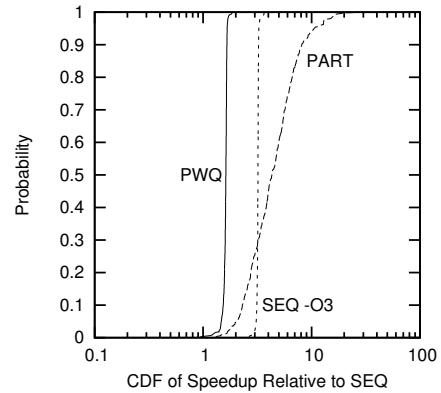


Figure 6.44: Effect of Compiler Optimization (-O3)

Although PART’s speedup is impressive, we should not neglect sequential optimizations. Figure 6.44 shows that SEQ, when compiled with -O3, is about twice as fast as unoptimized PWQ, approaching the performance of unoptimized PART. Compiling all three algorithms with -O3 gives results similar to (albeit faster than) those shown in Figure 6.40, except that PWQ provides almost no speedup compared to SEQ, in keeping with Amdahl’s Law [Amd67]. However, if the goal is to double performance compared to unoptimized SEQ, as opposed to achieving optimality, compiler optimizations are quite attractive.

Cache alignment and padding often improves performance by reducing false sharing. However, for these maze-solution algorithms, aligning and padding the maze-cell array *degrades* performance by up to 42% for 1000x1000 mazes. Cache locality is more important than avoiding false sharing, especially for large mazes. For smaller 20-by-20 or 50-by-50 mazes, aligning and padding can produce up to a 40% performance improvement for PART,

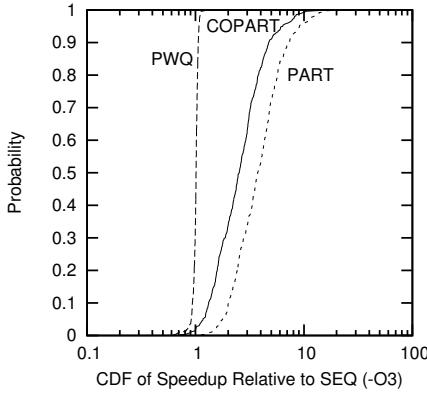


Figure 6.45: Partitioned Coroutines

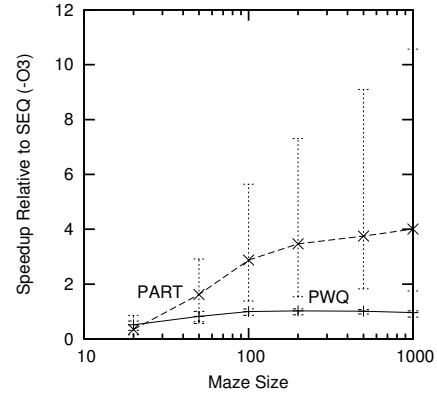


Figure 6.46: Varying Maze Size vs. SEQ

but for these small sizes, SEQ performs better anyway because there is insufficient time for PART to make up for the overhead of thread creation and destruction.

In short, the partitioned parallel maze solver is an interesting example of an algorithmic superlinear speedup. If “algorithmic superlinear speedup” causes cognitive dissonance, please proceed to the next section.

6.5.4 Alternative Sequential Maze Solver

The presence of algorithmic superlinear speedups suggests simulating parallelism via co-routines, for example, manually switching context between threads on each pass through the main do-while loop in Figure 6.37. This context switching is straightforward because the context consists only of the variables `c` and `vi`: Of the numerous ways to achieve the effect, this is a good tradeoff between context-switch overhead and visit percentage. As can be seen in Figure 6.45, this coroutine algorithm (COPART) is quite effective, with the performance on one thread being within about 30% of PART on two threads (`maze_2seq.c`).

6.5.5 Performance Comparison II

Figures 6.46 and 6.47 show the effects of varying maze size, comparing both PWQ and PART running on two threads against either SEQ or COPART, respectively, with 90%-confidence error bars. PART shows superlinear scalability against SEQ and modest scalability against COPART for 100-by-100 and larger mazes. PART exceeds theoretical energy-efficiency breakeven against COPART at roughly the 200-by-200 maze size, given that power

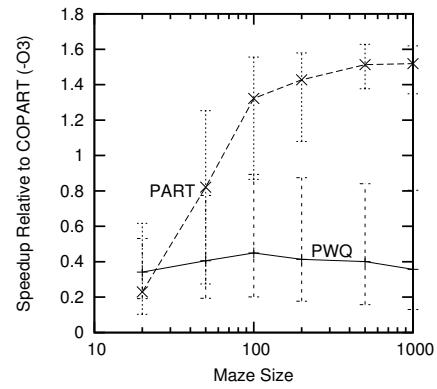


Figure 6.47: Varying Maze Size vs. COPART

consumption rises as roughly the square of the frequency for high frequencies [Mud00], so that 1.4x scaling on two threads consumes the same energy as a single thread at equal solution speeds. In contrast, PWQ shows poor scalability against both SEQ and COPART unless unoptimized: Figures 6.46 and 6.47 were generated using `-O3`.

Figure 6.48 shows the performance of PWQ and PART relative to COPART. For PART runs with more than two threads, the additional threads were started evenly spaced along the diagonal connecting the starting and ending cells. Simplified link-state routing [BG87] was used to detect early termination on PART runs with more than two threads (the solution is flagged when a thread is connected to both beginning and end). PWQ performs quite poorly, but PART hits breakeven at two threads and again at five threads, achieving modest speedups beyond five threads. Theoretical energy efficiency breakeven is within

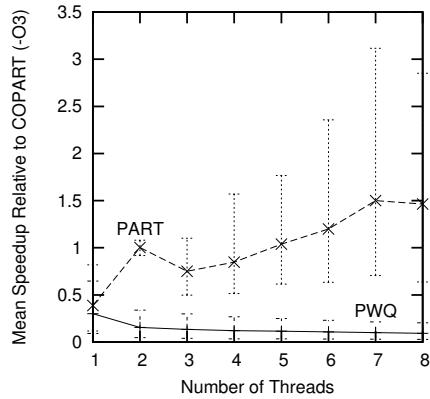


Figure 6.48: Mean Speedup vs. Number of Threads, 1000x1000 Maze

the 90% confidence interval for seven and eight threads. The reasons for the peak at two threads are (1) the lower complexity of termination detection in the two-thread case and (2) the fact that there is a lower probability of the third and subsequent threads making useful forward progress: Only the first two threads are guaranteed to start on the solution line. This disappointing performance compared to results in Figure 6.47 is due to the less-tightly integrated hardware available in the larger and older Xeon® system running at 2.66GHz.

6.5.6 Future Directions and Conclusions

Much future work remains. First, this section applied only one technique used by human maze solvers. Others include following walls to exclude portions of the maze and choosing internal starting points based on the locations of previously traversed paths. Second, different choices of starting and ending points might favor different algorithms. Third, although placement of the PART algorithm's first two threads is straightforward, there are any number of placement schemes for the remaining threads. Optimal placement might well depend on the starting and ending points. Fourth, study of unsolvable mazes and cyclic mazes is likely to produce interesting results. Fifth, the lightweight C++11 atomic operations might improve performance. Sixth, it would be interesting to compare the speedups for three-dimensional mazes (or of even higher-order mazes). Finally, for mazes, humiliating parallelism indicated a more-efficient sequential implementation using coroutines. Do humiliatingly parallel algorithms always lead to more-efficient sequential

implementations, or are there inherently humiliatingly parallel algorithms for which coroutine context-switch overhead overwhelms the speedups?

This section demonstrated and analyzed parallelization of maze-solution algorithms. A conventional work-queue-based algorithm did well only when compiler optimizations were disabled, suggesting that some prior results obtained using high-level/overhead languages will be invalidated by advances in optimization.

This section gave a clear example where approaching parallelism as a first-class optimization technique rather than as a derivative of a sequential algorithm paves the way for an improved sequential algorithm. High-level design-time application of parallelism is likely to be a fruitful field of study. This section took the problem of solving mazes from mildly scalable to humiliatingly parallel and back again. It is hoped that this experience will motivate work on parallelism as a first-class design-time whole-application optimization technique, rather than as a grossly suboptimal after-the-fact micro-optimization to be retrofitted into existing programs.

6.6 Partitioning, Parallelism, and Optimization

Most important, although this chapter has demonstrated that although applying parallelism at the design level gives excellent results, this final section shows that this is not enough. For search problems such as maze solution, this section has shown that search strategy is even more important than parallel design. Yes, for this particular type of maze, intelligently applying parallelism identified a superior search strategy, but this sort of luck is no substitute for a clear focus on search strategy itself.

As noted back in Section 2.2, parallelism is but one potential optimization of many. A successful design needs to focus on the most important optimization. Much though I might wish to claim otherwise, that optimization might or might not be parallelism.

However, for the many cases where parallelism is the right optimization, the next section covers that synchronization workhorse, locking.

Chapter 7

Locking

In recent concurrency research, the role of villain is often played by locking. In many papers and presentations, locking stands accused of promoting deadlocks, convoying, starvation, unfairness, data races, and all manner of other concurrency sins. Interestingly enough, the role of workhorse in production-quality shared-memory parallel software is played by, you guessed it, locking. This chapter will look into this dichotomy between villain and hero, as fancifully depicted in Figures 7.1 and Figure 7.2.

There are a number of reasons behind this Jekyll-and-Hyde dichotomy:

1. Many of locking's sins have pragmatic design solutions that work well in most cases, for example:
 - (a) Use of 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 10.
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 statistical counters (see Chapter 5), reference counters (see Section 9.1), hazard pointers (see Section 9.1.2), sequence-locking readers (see Section 9.2), RCU (see Section 9.3), and simple non-blocking data structures (see Section 14.3).
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. Locking works extremely well for some software artifacts and extremely poorly for others. Developers who have worked on artifacts for which locking works well can be expected to have a much more positive opinion of locking than those who have worked on artifacts for which locking works poorly, as will be discussed in Section 7.5.
6. All good stories need a villain, and locking has a long and honorable history serving as a research-paper whipping boy.

Quick Quiz 7.1: Just how can serving as a whipping boy be considered to be in any way honorable??? ■

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

7.1 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.

7.1.1 Deadlock

Deadlock occurs when each of a group of threads is holding at least one lock while at the same time waiting on a lock held by a member of that same group.

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

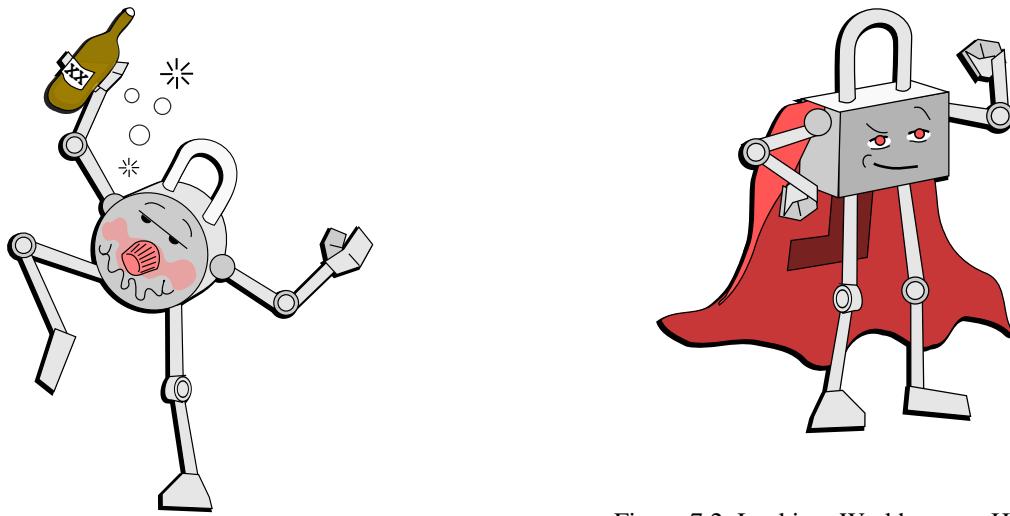


Figure 7.1: Locking: Villain or Slob?

Figure 7.2: Locking: Workhorse or Hero?

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 7.3. 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 7.3, this cycle is Thread B, Lock 3, Thread C, Lock 4, and back to Thread B.

Quick Quiz 7.2: 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 appropriate 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

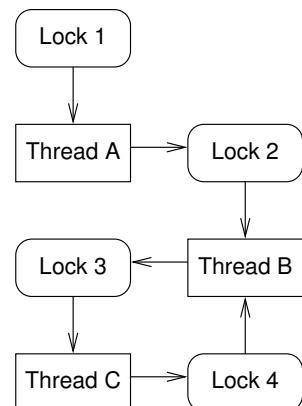


Figure 7.3: Deadlock Cycle

locking hierarchies (Section 7.1.1.1), local locking hierarchies (Section 7.1.1.2), layered locking hierarchies (Section 7.1.1.3), strategies for dealing with APIs containing pointers to locks (Section 7.1.1.4), conditional locking (Section 7.1.1.5), acquiring all needed locks first (Section 7.1.1.6), single-lock-at-a-time designs (Section 7.1.1.7), and strategies for signal/interrupt handlers (Section 7.1.1.8). 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.

7.1.1.1 Locking Hierarchies

Locking hierarchies order the locks and prohibit acquiring locks out of order. In Figure 7.3, we might order the locks numerically, so that a thread was forbidden from 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 program, it is wise to use tools to enforce your locking hierarchy [Cor06a].

7.1.1.2 Local Locking Hierarchies

However, the global nature of locking hierarchies make 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 7.3: 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 also 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 7.4: 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 7.4 and 7.5. In both figures, application func-

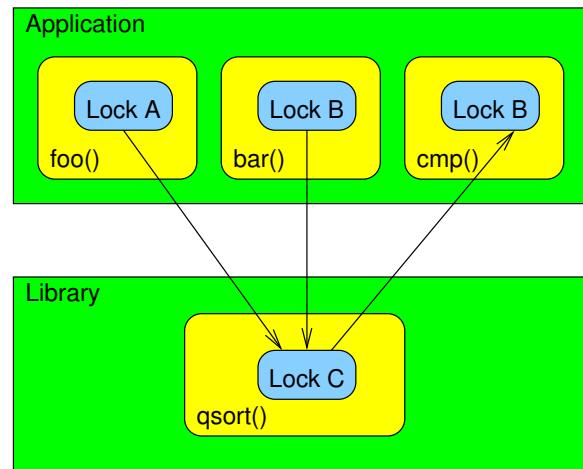


Figure 7.4: Without Local Locking Hierarchy for `qsort()`

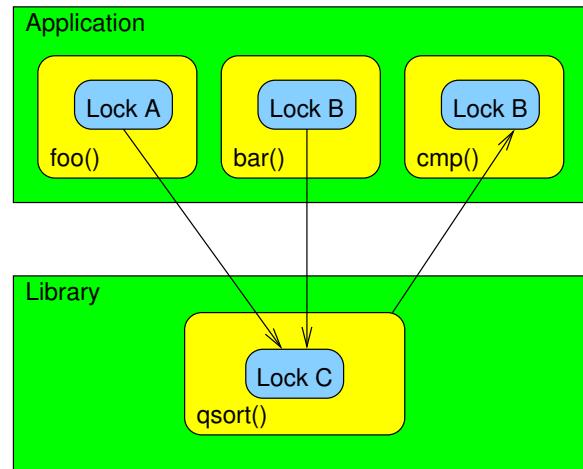


Figure 7.5: Local Locking Hierarchy for `qsort()`

tions `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 7.4, deadlock can occur. To see this, suppose that one thread invokes `foo()` while a second thread concurrently invokes `bar()`. The first thread will

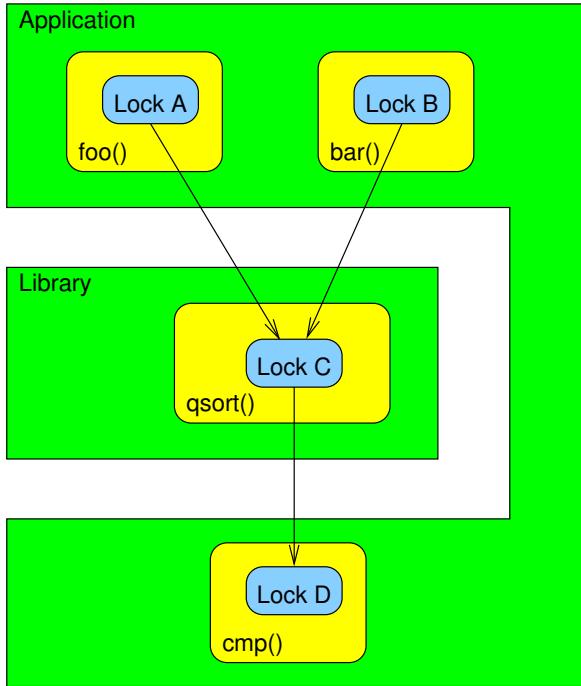


Figure 7.6: Layered Locking Hierarchy for `qsort()`

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 7.5.

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.

7.1.1.3 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 7.6. Here, the `cmp()`

```

1 struct locked_list {
2     spinlock_t s;
3     struct list_head h;
4 };
5
6 struct list_head *list_start(struct locked_list *lp)
7 {
8     spin_lock(&lp->s);
9     return list_next(lp, &lp->h);
10 }
11
12 struct list_head *list_next(struct locked_list *lp,
13                             struct list_head *np)
14 {
15     struct list_head *ret;
16
17     ret = np->next;
18     if (ret == &lp->h) {
19         spin_unlock(&lp->s);
20         ret = NULL;
21     }
22     return ret;
23 }
```

Figure 7.7: Concurrent List Iterator

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.

Please note that it is not typically possible to mechanically change `cmp()` to use the new Lock D. Quite the opposite: It is often necessary to make profound design-level modifications. Nevertheless, the effort required for such modifications is normally a small price to pay in order to avoid deadlock.

For another example where releasing all locks before invoking unknown code is impractical, imagine an iterator over a linked list, as shown in Figure 7.7 (`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 7.8 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()`

```

1 struct list_ints {
2     struct list_head n;
3     int a;
4 };
5
6 void list_print(struct locked_list *lp)
7 {
8     struct list_head *np;
9     struct list_ints *ip;
10
11    np = list_start(lp);
12    while (np != NULL) {
13        ip = list_entry(np, struct list_ints, n);
14        printf("\t%d\n", ip->a);
15        np = list_next(lp, np);
16    }
17 }

```

Figure 7.8: Concurrent List Iterator Usage

or `list_next()`, which results in deadlock. We can avoid the deadlock by layering the locking hierarchy to 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.¹ 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 9.

7.1.1.4 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 7.5: 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.

¹ One name for this is “object-oriented spaghetti code.”

```

1 spin_lock(&lock2);
2 layer_2_processing(pkt);
3 nextlayer = layer_1(pkt);
4 spin_lock(&nextlayer->lock1);
5 layer_1_processing(pkt);
6 spin_unlock(&lock2);
7 spin_unlock(&nextlayer->lock1);

```

Figure 7.9: Protocol Layering and Deadlock

```

1 retry:
2     spin_lock(&lock2);
3     layer_2_processing(pkt);
4     nextlayer = layer_1(pkt);
5     if (!spin_trylock(&nextlayer->lock1)) {
6         spin_unlock(&lock2);
7         spin_lock(&nextlayer->lock1);
8         spin_lock(&lock2);
9         if (layer_1(pkt) != nextlayer) {
10            spin_unlock(&nextlayer->lock1);
11            spin_unlock(&lock2);
12            goto retry;
13        }
14    }
15    layer_1_processing(pkt);
16    spin_unlock(&lock2);
17    spin_unlock(&nextlayer->lock1);

```

Figure 7.10: Avoiding Deadlock Via Conditional Locking

Quick Quiz 7.6: Doesn't the fact that `pthread_cond_wait()` first releases the mutex and then re-acquires it eliminate the possibility of deadlock? ■

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.

7.1.1.5 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 7.9. 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 7.10. 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.² Therefore, line 9 must recheck the decision, and if it has changed, must release the locks and start over.

Quick Quiz 7.7: Can the transformation from Figure 7.9 to Figure 7.10 be applied universally? ■

Quick Quiz 7.8: But the complexity in Figure 7.10 is well worthwhile given that it avoids deadlock, right? ■

7.1.1.6 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 7.1.2.

Quick Quiz 7.9: When using the “acquire needed locks first” approach described in Section 7.1.1.6, how can livelock be avoided? ■

A related approach, two-phase locking [BHG87], has seen long production use in transactional database systems. In the first phase of a two-phase locking transaction, locks are acquired but not released. Once all needed locks have been acquired, the transaction enters the second phase, where locks are released, but not acquired. This locking approach allows databases to provide serializability guarantees for their transactions, in other words, to guarantee that all values seen and produced by the transactions are consistent with some global ordering of all the transactions. Many such systems rely on the ability to abort transactions, although this can be simplified

by avoiding making any changes to shared data until all needed locks are acquired. Livelock and deadlock are issues in such systems, but practical solutions may be found in any of a number of database textbooks.

7.1.1.7 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 7.4 and several others are presented in Chapter 9.

7.1.1.8 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 7.10: 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 7.11: 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

² And, in contrast to the 1900s, mobility is the common case.

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.

Quick Quiz 7.12: If acquiring locks in signal handlers is such a bad idea, why even discuss ways of making it safe? ■

7.1.1.9 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 7.13: Given an object-oriented application that passes control freely among a group of objects such that there is no straightforward locking hierarchy,³ layered or otherwise, how can this application be parallelized? ■

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

7.1.2 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 7.11. This example's beauty hides an ugly livelock. To see this, consider the following sequence of events:

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` on line 6, but fails because Thread 2 holds it.
4. Thread 2 attempts to acquire `lock1` on line 20, but fails because Thread 1 holds it.
5. Thread 1 releases `lock1` on line 7, then jumps to `retry` at line 3.
6. Thread 2 releases `lock2` on line 21, and jumps to `retry` at line 17.

³ Also known as “object-oriented spaghetti code.”

```

1 void thread1(void)
2 {
3     retry:
4     spin_lock(&lock1);
5     do_one_thing();
6     if (!spin_trylock(&lock2)) {
7         spin_unlock(&lock1);
8         goto retry;
9     }
10    do_another_thing();
11    spin_unlock(&lock2);
12    spin_unlock(&lock1);
13 }
14
15 void thread2(void)
16 {
17     retry:
18     spin_lock(&lock2);
19     do_a_third_thing();
20     if (!spin_trylock(&lock1)) {
21         spin_unlock(&lock2);
22         goto retry;
23     }
24     do_a_fourth_thing();
25     spin_unlock(&lock1);
26     spin_unlock(&lock2);
27 }
```

Figure 7.11: Abusing Conditional Locking

7. The livelock dance repeats from the beginning.

Quick Quiz 7.14: How can the livelock shown in Figure 7.11 be avoided? ■

Livelock can be thought of as an extreme form of starvation where a group of threads starve, rather than just one of them.⁴

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 exponential backoff can often address livelock and starvation. The idea is to introduce exponentially increasing delays before each retry, as shown in Figure 7.12.

Quick Quiz 7.15: What problems can you spot in the code in Figure 7.12? ■

However, for better results, the backoff should be bounded, and even better high-contention results have been obtained via queued locking [And90], which is discussed more in Section 7.3.2. Of course, best of all is to use a good parallel design so that lock contention remains low.

⁴ 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 adequate forward progress is a problem that needs to be fixed, regardless of what name you choose for it.

```

1 void thread1(void)
2 {
3     unsigned int wait = 1;
4     retry:
5         spin_lock(&lock1);
6         do_one_thing();
7         if (!spin_trylock(&lock2)) {
8             spin_unlock(&lock1);
9             sleep(wait);
10            wait = wait << 1;
11            goto retry;
12        }
13        do_another_thing();
14        spin_unlock(&lock2);
15        spin_unlock(&lock1);
16    }
17
18 void thread2(void)
19 {
20     unsigned int wait = 1;
21     retry:
22     spin_lock(&lock2);
23     do_a_third_thing();
24     if (!spin_trylock(&lock1)) {
25         spin_unlock(&lock2);
26         sleep(wait);
27         wait = wait << 1;
28         goto retry;
29     }
30     do_a_fourth_thing();
31     spin_unlock(&lock1);
32     spin_unlock(&lock2);
33 }

```

Figure 7.12: Conditional Locking and Exponential Back-off

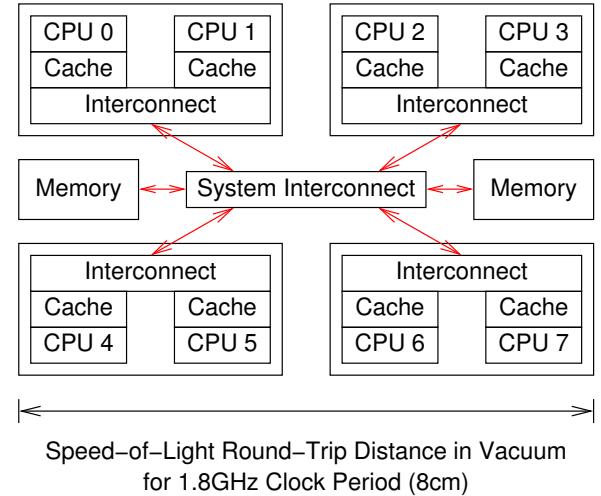


Figure 7.13: System Architecture and Lock Unfairness

7.1.3 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 7.13. 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 holds 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 0 and 1, bypassing CPUs 2-7.

Quick Quiz 7.16: Wouldn't it be better just to use a good parallel design so that lock contention was low enough to avoid unfairness? ■

7.1.4 Inefficiency

Locks are implemented using atomic instructions and memory barriers, and often involve cache misses. As we saw in Chapter 3, 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 6.3, especially Figure 6.22: 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 7.17: How might the lock holder be interfered with? ■

7.2 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 7.2.1), reader-writer locks (Section 7.2.2), multi-role locks (Section 7.2.3), and scoped locking (Section 7.2.4).

7.2.1 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.

Quick Quiz 7.18: Does it ever make sense to have an exclusive lock acquisition immediately followed by a release of that same lock, that is, an empty critical section? ■

7.2.2 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.

The classic reader-writer lock implementation involves a set of counters and flags that are manipulated atomi-

	Null (Not Held)	Concurrent Read	Concurrent Write	Protected Read	Protected Write	Exclusive
Null (Not Held)						
Concurrent Read						X
Concurrent Write			X	X	X	
Protected Read		X		X	X	
Protected Write		X	X	X	X	
Exclusive	X	X	X	X	X	

Table 7.1: VAX/VMS Distributed Lock Manager Policy

cally. 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. The canonical use case for reader-writer locking involves very long read-side critical sections, preferably measured in hundreds of microseconds or even milliseconds.

7.2.3 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

which is that of the VAX/VMS distributed lock manager (DLM) [ST87], which is shown in Table 7.1. 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 concurrent 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 7.19: 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 illustration of just how flexible the concepts behind locking can be. It also serves as a very simple introduction to the locking schemes used by modern DBMSes, which can have more than thirty locking modes, compared to VAX/VMS’s six.

7.2.4 Scoped Locking

The locking primitives discussed thus far require explicit acquisition and release primitives, for example, `spin_lock()` and `spin_unlock()`, respectively. Another approach is to use the object-oriented “resource allocation is initialization” (RAII) pattern [ES90].⁵ This pattern is often applied to auto variables in languages like C++, where the corresponding *constructor* is invoked upon entry to the object’s scope, and the corresponding *destructor* is invoked upon exit from that scope. This can be applied to locking by having the constructor acquire the lock and the destructor free it.

This approach can be quite useful, in fact in 1990 I was convinced that it was the only type of locking that was needed.⁶ One very nice property of RAII locking is that you don’t need to carefully release the lock on each and every code path that exits that scope, a property that can eliminate a troublesome set of bugs.

However, RAII locking also has a dark side. RAII makes it quite difficult to encapsulate lock acquisition and release, for example, in iterators. In many iterator implementations, you would like to acquire the lock in the iterator’s “start” function and release it in the iterator’s “stop” function. RAII locking instead requires that the lock acquisition and release take place in the same level of scoping, making such encapsulation difficult or even impossible.

RAII locking also prohibits overlapping critical sections, due to the fact that scopes must nest. This prohibition makes it difficult or impossible to express a number of useful constructs, for example, locking trees that mediate between multiple concurrent attempts to assert an event. Of an arbitrarily large group of concurrent attempts, only one need succeed, and the best strategy for the remaining attempts is for them to fail as quickly and painlessly as possible. Otherwise, lock contention becomes pathological on large systems (where “large” is many hundreds of CPUs).

⁵ Though more clearly expressed at http://www.stroustrup.com/bs_faq2.html#finally.

⁶ My later work with parallelism at Sequent Computer Systems very quickly disabused me of this misguided notion.

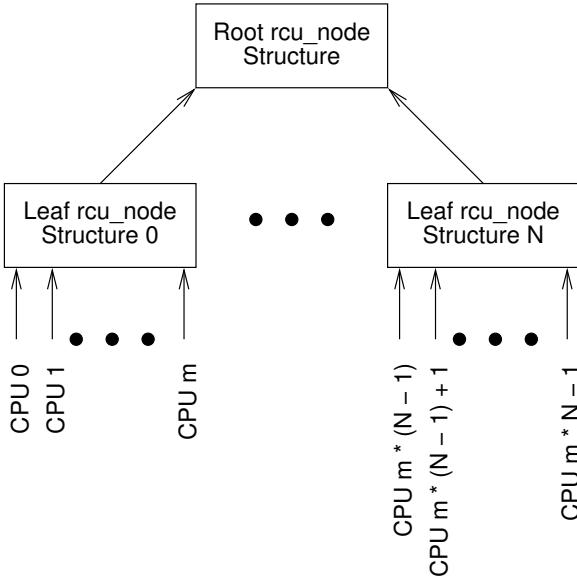


Figure 7.14: Locking Hierarchy

Example data structures (taken from the Linux kernel's implementation of RCU) are shown in Figure 7.14. Here, each CPU is assigned a leaf `rcu_node` structure, and each `rcu_node` structure has a pointer to its parent (named, oddly enough, `->parent`), up to the root `rcu_node` structure, which has a NULL `->parent` pointer. The number of child `rcu_node` structures per parent can vary, but is typically 32 or 64. Each `rcu_node` structure also contains a lock named `->fqslck`.

The general approach is a *tournament*, where a given CPU conditionally acquires its leaf `rcu_node` structure's `->fqslck`, and, if successful, attempt to acquire that of the parent, then release that of the child. In addition, at each level, the CPU checks a global `gp_flags` variable, and if this variable indicates that some other CPU has asserted the event, the first CPU drops out of the competition. This acquire-then-release sequence continues until either the `gp_flags` variable indicates that someone else won the tournament, one of the attempts to acquire an `->fqslck` fails, or the root `rcu_node` structure's `->fqslck` as been acquired.

Simplified code to implement this is shown in Figure 7.15. The purpose of this function is to mediate between CPUs who have concurrently detected a need to invoke the `do_force_quiescent_state()` function. At any given time, it only makes sense for one instance of `do_force_quiescent_state()` to be

```

1 void force_quiescent_state(struct rcu_node *rnp_leaf)
2 {
3     int ret;
4     struct rcu_node *rnp = rnp_leaf;
5     struct rcu_node *rnp_old = NULL;
6
7     for (; rnp != NULL; rnp = rnp->parent) {
8         ret = (ACCESS_ONCE(gp_flags)) ||
9             !raw_spin_trylock(&rnp->fqslck);
10        if (rnp_old != NULL)
11            raw_spin_unlock(&rnp_old->fqslck);
12        if (ret)
13            return;
14        rnp_old = rnp;
15    }
16    if (!ACCESS_ONCE(gp_flags)) {
17        ACCESS_ONCE(gp_flags) = 1;
18        do_force_quiescent_state();
19        ACCESS_ONCE(gp_flags) = 0;
20    }
21    raw_spin_unlock(&rnp_old->fqslck);
22 }
```

Figure 7.15: Conditional Locking to Reduce Contention

active, so if there are multiple concurrent callers, we need at most one of them to actually invoke `do_force_quiescent_state()`, and we need the rest to (as quickly and painlessly as possible) give up and leave.

To this end, each pass through the loop spanning lines 7-15 attempts to advance up one level in the `rcu_node` hierarchy. If the `gp_flags` variable is already set (line 8) or if the attempt to acquire the current `rcu_node` structure's `->fqslck` is unsuccessful (line 9), then local variable `ret` is set to 1. If line 10 sees that local variable `rnp_old` is non-NULL, meaning that we hold `rnp_old`'s `->fqslck`, line 11 releases this lock (but only after the attempt has been made to acquire the parent `rcu_node` structure's `->fqslck`). If line 12 sees that either line 8 or 9 saw a reason to give up, line 13 returns to the caller. Otherwise, we must have acquired the current `rcu_node` structure's `->fqslck`, so line 14 saves a pointer to this structure in local variable `rnp_old` in preparation for the next pass through the loop.

If control reaches line 16, we won the tournament, and now holds the root `rcu_node` structure's `->fqslck`. If line 16 still sees that the global variable `gp_flags` is zero, line 17 sets `gp_flags` to one, line 18 invokes `do_force_quiescent_state()`, and line 19 resets `gp_flags` back to zero. Either way, line 21 releases the root `rcu_node` structure's `->fqslck`.

Quick Quiz 7.20: The code in Figure 7.15 is ridiculously complicated! Why not conditionally acquire a single global lock? ■

Quick Quiz 7.21: Wait a minute! If we “win” the tour-

```

1 typedef int xchglock_t;
2 #define DEFINE_XCHG_LOCK(n) xchglock_t n = 0
3
4 void xchg_lock(xchglock_t *xp)
5 {
6     while (xchg(xp, 1) == 1) {
7         while (*xp == 1)
8             continue;
9     }
10 }
11
12 void xchg_unlock(xchglock_t *xp)
13 {
14     (void)xchg(xp, 0);
15 }

```

Figure 7.16: Sample Lock Based on Atomic Exchange

nement on line 16 of Figure 7.15, we get to do all the work of `do_force_quiescent_state()`. Exactly how is that a win, really? ■

This function illustrates the not-uncommon pattern of hierarchical locking. This pattern is quite difficult to implement using RAII locking, just like the iterator encapsulation noted earlier, and so the lock/unlock primitives will be needed for the foreseeable future.

7.3 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.

7.3.1 Sample Exclusive-Locking Implementation Based on Atomic Exchange

This section reviews the implementation shown in Figure 7.16. 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.

Quick Quiz 7.22: Why not rely on the C language’s default initialization of zero instead of using the explicit initializer shown on line 2 of Figure 7.16? ■

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 7.23: Why bother with the inner loop on lines 7-8 of Figure 7.16? 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 7.24: Why not simply store zero into the lock word on line 14 of Figure 7.16? ■

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.

7.3.2 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. However, it is important to avoid getting too worried about the possibility of preemption and interruption, given that this preemption and interruption might just as well happen just after the lock was acquired.⁷

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

⁷ Besides, the best way of handling high lock contention is to avoid it in the first place! However, there are some situations where high lock contention is the lesser of the available evils, and in any case, studying schemes that deal with high levels of contention is good mental exercise.

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, WKS94, Cra93, MLH94, TS93]. Queued locks avoid high cache-validation overhead by assigning each thread a queue element. These queue elements are linked together into a queue that governs the order that the lock will be granted to the waiting threads. The key point is that each thread spins on its own queue element, so that the lock holder need only invalidate the first element from the next thread's CPU's cache. This arrangement greatly reduces the overhead of lock handoff at high levels of 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 [SRL90a, Cor06b], despite some lingering controversy over this practice [Yod04a, 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 certain sizable 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, Dre11].

Interestingly enough, atomic instructions are not strictly needed to implement locks [Dij65, Lam74]. 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, with one exception described below, such study is left as an exercise for the reader.

Gamsa et al. [GKAS99, Section 5.3] describe a token-based mechanism in which a token circulates among the CPUs. When the token reaches a given CPU, it has exclusive access to anything protected by that token. There are any number of schemes that may be used to implement the token-based mechanism, for example:

1. Maintain a per-CPU flag, which is initially zero for all but one CPU. When a CPU's flag is non-zero, it holds the token. When it finishes with the token, it zeroes its flag and sets the flag of the next CPU to one (or to any other non-zero value).
2. Maintain a per-CPU counter, which is initially set to the corresponding CPU's number, which we assume to range from zero to $N - 1$, where N is the number of CPUs in the system. When a CPU's counter is greater than that of the next CPU (taking counter wrap into account), the first CPU holds the token.

When it is finished with the token, it sets the next CPU's counter to a value one greater than its own counter.

Quick Quiz 7.25: How can you tell if one counter is greater than another, while accounting for counter wrap? ■

Quick Quiz 7.26: Which is better, the counter approach or the flag approach? ■

This lock is unusual in that a given CPU cannot necessarily acquire it immediately, even if no other CPU is using it at the moment. Instead, the CPU must wait until the token comes around to it. This is useful in cases where CPUs need periodic access to the critical section, but can tolerate variances in token-circulation rate. Gamsa et al. [GKAS99] used it to implement a variant of read-copy update (see Section 9.3), but it could also be used to protect periodic per-CPU operations such as flushing per-CPU caches used by memory allocators [MS93], garbage-collecting per-CPU data structures, or flushing per-CPU data to shared storage (or to mass storage, for that matter).

As increasing numbers of people gain familiarity with parallel hardware and parallelize increasing amounts of code, we can expect more special-purpose locking primitives to appear. Nevertheless, you should carefully consider this important safety tip: Use the standard synchronization primitives whenever humanly possible. The big advantage of the standard synchronization primitives over roll-your-own efforts is that the standard primitives are typically *much* less bug-prone.⁸

7.4 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.

⁸ And yes, I have done at least my share of roll-your-own synchronization primitives. However, you will notice that my hair is much greyer than it was before I started doing that sort of work. Coincidence? Maybe. But are you *really* willing to risk your own hair turning prematurely grey?

```

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

Figure 7.17: Per-Element Locking Without Existence Guarantees

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 (directly or indirectly) from that function, then the given function has an active instance.

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

Quick Quiz 7.27: 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

```

1 int delete(int key)
2 {
3     int b;
4     struct element *p;
5     spinlock_t *sp;
6
7     b = hashfunction(key);
8     sp = &locktable[b];
9     spin_lock(sp);
10    p = hashtable[b];
11    if (p == NULL || p->key != key) {
12        spin_unlock(sp);
13        return 0;
14    }
15    hashtable[b] = NULL;
16    spin_unlock(sp);
17    kfree(p);
18    return 1;
19 }

```

Figure 7.18: Per-Element Locking With Lock-Based Existence Guarantees

that is to protect a data element in the data element itself is subject to subtle race conditions, as shown in Figure 7.17.

Quick Quiz 7.28: What if the element we need to delete is not the first element of the list on line 8 of Figure 7.17? ■

Quick Quiz 7.29: What race condition can occur in Figure 7.17? ■

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 7.18. 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 9 describes simpler—and faster—ways of providing existence guarantees.

7.5 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. In my experience, those writing whole applications are happy with locking, those writing parallel libraries are less happy, and those parallelizing existing sequential libraries are extremely unhappy. The following

sections discuss some reasons for these differences in viewpoints.

7.5.1 Locking For Applications: Hero!

When writing an entire application (or entire kernel), developers have full control of the design, including the synchronization design. Assuming that the design makes good use of partitioning, as discussed in Chapter 6, locking can be an extremely effective synchronization mechanism, as demonstrated by the heavy use of locking in production-quality parallel software.

Nevertheless, although such software usually bases most of its synchronization design on locking, such software also almost always makes use of other synchronization mechanisms, including special counting algorithms (Chapter 5), data ownership (Chapter 8), reference counting (Section 9.1), sequence locking (Section 9.2), and read-copy update (Section 9.3). In addition, practitioners use tools for deadlock detection [Cor06a], lock acquisition/release balancing [Cor04b], cache-miss analysis [The11], hardware-counter-based profiling [EGMdB11, The12], and many more besides.

Given careful design, use of a good combination of synchronization mechanisms, and good tooling, locking works quite well for applications and kernels.

7.5.2 Locking For Parallel Libraries: Just Another Tool

Unlike applications and kernels, the designer of a library cannot know the locking design of the code that the library will be interacting with. In fact, that code might not be written for years to come. Library designers therefore have less control and must exercise more care when laying out their synchronization design.

Deadlock is of course of particular concern, and the techniques discussed in Section 7.1.1 need to be applied. One popular deadlock-avoidance strategy is therefore to ensure that the library’s locks are independent subtrees of the enclosing program’s locking hierarchy. However, this can be harder than it looks.

One complication was discussed in Section 7.1.1.2, namely when library functions call into application code, with `qsort()`’s comparison-function argument being a case in point. Another complication is the interaction with signal handlers. If an application signal handler is invoked from a signal received within the library function, deadlock can ensue just as surely as if the library function

had called the signal handler directly. A final complication occurs for those library functions that can be used between a `fork()`/`exec()` pair, for example, due to use of the `system()` function. In this case, if your library function was holding a lock at the time of the `fork()`, then the child process will begin life with that lock held. Because the thread that will release the lock is running in the parent but not the child, if the child calls your library function, deadlock will ensue.

The following strategies may be used to avoid deadlock problems in these cases:

1. Don't use either callbacks or signals.
2. Don't acquire locks from within callbacks or signal handlers.
3. Let the caller control synchronization.
4. Parameterize the library API to delegate locking to caller.
5. Explicitly avoid callback deadlocks.
6. Explicitly avoid signal-handler deadlocks.

Each of these strategies is discussed in one of the following sections.

7.5.2.1 Use Neither Callbacks Nor Signals

If a library function avoids callbacks and the application as a whole avoids signals, then any locks acquired by that library function will be leaves of the locking-hierarchy tree. This arrangement avoids deadlock, as discussed in Section 7.1.1.1. Although this strategy works extremely well where it applies, there are some applications that must use signal handlers, and there are some library functions (such as the `qsort()` function discussed in Section 7.1.1.2) that require callbacks.

The strategy described in the next section can often be used in these cases.

7.5.2.2 Avoid Locking in Callbacks and Signal Handlers

If neither callbacks nor signal handlers acquire locks, then they cannot be involved in deadlock cycles, which allows straightforward locking hierarchies to once again consider library functions to be leaves on the locking-hierarchy tree. This strategy works very well for most uses of `qsort`, whose callbacks usually simply compare the two values

passed in to them. This strategy also works wonderfully for many signal handlers, especially given that acquiring locks from within signal handlers is generally frowned upon [Gro01],⁹ but can fail if the application needs to manipulate complex data structures from a signal handler.

Here are some ways to avoid acquiring locks in signal handlers even if complex data structures must be manipulated:

1. Use simple data structures based on non-blocking synchronization, as will be discussed in Section 14.3.1.
2. If the data structures are too complex for reasonable use of non-blocking synchronization, create a queue that allows non-blocking enqueue operations. In the signal handler, instead of manipulating the complex data structure, add an element to the queue describing the required change. A separate thread can then remove elements from the queue and carry out the required changes using normal locking. There are a number of readily available implementations of concurrent queues [KLP12, Des09, MS96].

This strategy should be enforced with occasional manual or (preferably) automated inspections of callbacks and signal handlers. When carrying out these inspections, be wary of clever coders who might have (unwisely) created home-brew locks from atomic operations.

7.5.2.3 Caller Controls Synchronization

Let the caller control synchronization. This works extremely well when the library functions are operating on independent caller-visible instances of a data structure, each of which may be synchronized separately. For example, if the library functions operate on a search tree, and if the application needs a large number of independent search trees, then the application can associate a lock with each tree. The application then acquires and releases locks as needed, so that the library need not be aware of parallelism at all. Instead, the application controls the parallelism, so that locking can work very well, as was discussed in Section 7.5.1.

However, this strategy fails if the library implements a data structure that requires internal concurrency, for example, a hash table or a parallel sort. In this case, the library absolutely must control its own synchronization.

⁹ But the standard's words do not stop clever coders from creating their own home-brew locking primitives from atomic operations.

7.5.2.4 Parameterize Library Synchronization

The idea here is to add arguments to the library’s API to specify which locks to acquire, how to acquire and release them, or both. This strategy allows the application to take on the global task of avoiding deadlock by specifying which locks to acquire (by passing in pointers to the locks in question) and how to acquire them (by passing in pointers to lock acquisition and release functions), but also allows a given library function to control its own concurrency by deciding where the locks should be acquired and released.

In particular, this strategy allows the lock acquisition and release functions to block signals as needed without the library code needing to be concerned with which signals need to be blocked by which locks. The separation of concerns used by this strategy can be quite effective, but in some cases the strategies laid out in the following sections can work better.

That said, passing explicit pointers to locks to external APIs must be very carefully considered, as discussed in Section 7.1.1.4. Although this practice is sometimes the right thing to do, you should do yourself a favor by looking into alternative designs first.

7.5.2.5 Explicitly Avoid Callback Deadlocks

The basic rule behind this strategy was discussed in Section 7.1.1.2: “Release all locks before invoking unknown code.” This is usually the best approach because it allows the application to ignore the library’s locking hierarchy: the library remains a leaf or isolated subtree of the application’s overall locking hierarchy.

In cases where it is not possible to release all locks before invoking unknown code, the layered locking hierarchies described in Section 7.1.1.3 can work well. For example, if the unknown code is a signal handler, this implies that the library function block signals across all lock acquisitions, which can be complex and slow. Therefore, in cases where signal handlers (probably unwisely) acquire locks, the strategies in the next section may prove helpful.

7.5.2.6 Explicitly Avoid Signal-Handler Deadlocks

Signal-handler deadlocks can be explicitly avoided as follows:

1. If the application invokes the library function from within a signal handler, then that signal must be

blocked every time that the library function is invoked from outside of a signal handler.

2. If the application invokes the library function while holding a lock acquired within a given signal handler, then that signal must be blocked every time that the library function is called outside of a signal handler.

These rules can be enforced by using tools similar to the Linux kernel’s lockdep lock dependency checker [Cor06a]. One of the great strengths of lockdep is that it is not fooled by human intuition [Ros11].

7.5.2.7 Library Functions Used Between `fork()` and `exec()`

As noted earlier, if a thread executing a library function is holding a lock at the time that some other thread invokes `fork()`, the fact that the parent’s memory is copied to create the child means that this lock will be born held in the child’s context. The thread that will release this lock is running in the parent, but not in the child, which means that the child’s copy of this lock will never be released. Therefore, any attempt on the part of the child to invoke that same library function will result in deadlock.

One approach to this problem would be to have the library function check to see if the owner of the lock is still running, and if not, “breaking” the lock by re-initializing and then acquiring it. However, this approach has a couple of vulnerabilities:

1. The data structures protected by that lock are likely to be in some intermediate state, so that naively breaking the lock might result in arbitrary memory corruption.
2. If the child creates additional threads, two threads might break the lock concurrently, with the result that both threads believe they own the lock. This could again result in arbitrary memory corruption.

The `atfork()` function is provided to help deal with these situations. The idea is to register a triplet of functions, one to be called by the parent before the `fork()`, one to be called by the parent after the `fork()`, and one to be called by the child after the `fork()`. Appropriate cleanups can then be carried out at these three points.

Be warned, however, that coding of `atfork()` handlers is quite subtle in general. The cases where `atfork()` works best are cases where the data structure in question can simply be re-initialized by the child.

7.5.2.8 Parallel Libraries: Discussion

Regardless of the strategy used, the description of the library’s API must include a clear description of that strategy and how the caller should interact with that strategy. In short, constructing parallel libraries using locking is possible, but not as easy as constructing a parallel application.

7.5.3 Locking For Parallelizing Sequential Libraries: Villain!

With the advent of readily available low-cost multicore systems, a common task is parallelizing an existing library that was designed with only single-threaded use in mind. This all-too-common disregard for parallelism can result in a library API that is severely flawed from a parallel-programming viewpoint. Candidate flaws include:

1. Implicit prohibition of partitioning.
2. Callback functions requiring locking.
3. Object-oriented spaghetti code.

These flaws and the consequences for locking are discussed in the following sections.

7.5.3.1 Partitioning Prohibited

Suppose that you were writing a single-threaded hash-table implementation. It is easy and fast to maintain an exact count of the total number of items in the hash table, and also easy and fast to return this exact count on each addition and deletion operation. So why not?

One reason is that exact counters do not perform or scale well on multicore systems, as was seen in Chapter 5. As a result, the parallelized implementation of the hash table will not perform or scale well.

So what can be done about this? One approach is to return an approximate count, using one of the algorithms from Chapter 5. Another approach is to drop the element count altogether.

Either way, it will be necessary to inspect uses of the hash table to see why the addition and deletion operations need the exact count. Here are a few possibilities:

1. Determining when to resize the hash table. In this case, an approximate count should work quite well. It might also be useful to trigger the resizing operation from the length of the longest chain, which can be

computed and maintained in a nicely partitioned per-chain manner.

2. Producing an estimate of the time required to traverse the entire hash table. An approximate count works well in this case, also.
3. For diagnostic purposes, for example, to check for items being lost when transferring them to and from the hash table. This clearly requires an exact count. However, given that this usage is diagnostic in nature, it might suffice to maintain the lengths of the hash chains, then to infrequently sum them up while locking out addition and deletion operations.

It turns out that there is now a strong theoretical basis for some of the constraints that performance and scalability place on a parallel library’s APIs [AGH⁺11a, AGH⁺11b, McK11b]. Anyone designing a parallel library needs to pay close attention to those constraints.

Although it is all too easy to blame locking for what are really problems due to a concurrency-unfriendly API, doing so is not helpful. On the other hand, one has little choice but to sympathize with the hapless developer who made this choice in (say) 1985. It would have been a rare and courageous developer to anticipate the need for parallelism at that time, and it would have required an even more rare combination of brilliance and luck to actually arrive at a good parallel-friendly API.

Times change, and code must change with them. That said, there might be a huge number of users of a popular library, in which case an incompatible change to the API would be quite foolish. Adding a parallel-friendly API to complement the existing heavily used sequential-only API is probably the best course of action in this situation.

Nevertheless, human nature being what it is, we can expect our hapless developer to be more likely to complain about locking than about his or her own poor (though understandable) API design choices.

7.5.3.2 Deadlock-Prone Callbacks

Sections 7.1.1.2, 7.1.1.3, and 7.5.2 described how undisciplined use of callbacks can result in locking woes. These sections also described how to design your library function to avoid these problems, but it is unrealistic to expect a 1990s programmer with no experience in parallel programming to have followed such a design. Therefore, someone attempting to parallelize an existing callback-heavy single-threaded library will likely have many opportunities to curse locking’s villainy.

If there are a very large number of uses of a callback-heavy library, it may be wise to again add a parallel-friendly API to the library in order to allow existing users to convert their code incrementally. Alternatively, some advocate use of transactional memory in these cases. While the jury is still out on transactional memory, Section 17.2 discusses its strengths and weaknesses. It is important to note that hardware transactional memory (discussed in Section 17.3) cannot help here unless the hardware transactional memory implementation provides forward-progress guarantees, which few do. Other alternatives that appear to be quite practical (if less heavily hyped) include the methods discussed in Sections 7.1.1.5, and 7.1.1.6, as well as those that will be discussed in Chapters 8 and 9.

7.5.3.3 Object-Oriented Spaghetti Code

Object-oriented programming went mainstream sometime in the 1980s or 1990s, and as a result there is a huge amount of object-oriented code in production, much of it single-threaded. Although object orientation can be a valuable software technique, undisciplined use of objects can easily result in object-oriented spaghetti code. In object-oriented spaghetti code, control flits from object to object in an essentially random manner, making the code hard to understand and even harder, and perhaps impossible, to accommodate a locking hierarchy.

Although many might argue that such code should be cleaned up in any case, such things are much easier to say than to do. If you are tasked with parallelizing such a beast, you can reduce the number of opportunities to curse locking by using the techniques described in Sections 7.1.1.5, and 7.1.1.6, as well as those that will be discussed in Chapters 8 and 9. This situation appears to be the use case that inspired transactional memory, so it might be worth a try as well. That said, the choice of synchronization mechanism should be made in light of the hardware habits discussed in Chapter 3. After all, if the overhead of the synchronization mechanism is orders of magnitude more than that of the operations being protected, the results are not going to be pretty.

And that leads to a question well worth asking in these situations: Should the code remain sequential? For example, perhaps parallelism should be introduced at the process level rather than the thread level. In general, if a task is proving extremely hard, it is worth some time spent thinking about not only alternative ways to accomplish that particular task, but also alternative tasks that might better solve the problem at hand.

7.6 Summary

Locking is perhaps the most widely used and most generally useful synchronization tool. However, it works best when designed into an application or library from the beginning. Given the large quantity of pre-existing single-threaded code that might need to one day run in parallel, locking should therefore 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 8

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. Interestingly enough, data ownership covers each of the “big three” parallel design techniques: It partitions over threads (or CPUs, as the case may be), it batches all local operations, and its elimination of synchronization operations is weakening carried to its logical extreme. It should therefore be no surprise that data ownership 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 reference examples from previous chapters.

Quick Quiz 8.1: What form of data ownership is extremely difficult to avoid when creating shared-memory parallel programs (for example, using pthreads) in C or C++? ■

There are a number of approaches to data ownership. Section 8.1 presents the logical extreme in data ownership, where each thread has its own private address space. Section 8.2 looks at the opposite extreme, where the data is shared, but different threads own different access rights to the data. Section 8.3 describes function shipping, which is a way of allowing other threads to have indirect access to data owned by a particular thread. Section 8.4 describes how designated threads can be assigned ownership of a specified function and the related data. Section 8.5 discusses improving performance by transforming algorithms with shared data to instead use data ownership. Finally, Section 8.6 lists a few software environments that feature data ownership as a first-class citizen.

8.1 Multiple Processes

Section 4.1 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 8.2: What synchronization remains in the example shown in Section 8.1? ■

Quick Quiz 8.3: Is there any shared data in the example shown in Section 8.1? ■

This same pattern can be written in C as well as in sh, as illustrated by Figures 4.2 and 4.3.

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

8.2 Partial Data Ownership and pthreads

Chapter 5 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 5.9 on page 41. 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 8.4: 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 6.4.3 starting on page 78. In this algorithm, each thread's cache is completely private to that thread.

8.3 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 5.4.3 beginning on page 52, in particular the `flush_local_count_sig()` and `flush_local_count()` functions in Figure 5.24 on page 54.

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 5.25 on page 54 and Figure 5.26 on page 55).

Quick Quiz 8.5: What mechanisms other than POSIX signals may be used for function shipping? ■

8.4 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 owns the rights to the data that is required to do its job. The eventually consistent counter implementation described in Section 5.2.3 provides an example. This implementation has a designated thread that runs the `eventual()` function shown on lines 15-32 of Figure 5.8. 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 8.6: But none of the data in the `eventual()` function shown on lines 15-32 of Figure 5.8 is actually owned by the `eventual()` thread! In just what way is this data ownership??? ■

8.5 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 6.1.1, 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 5.12 on page 44. This is an example of an algorithm where 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 8.7: 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.

8.6 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 [MPI08] and BOINC [UoC08].
2. Map-reduce [Jac08].
3. Client-server systems, including RPC, web services, and pretty much any system with a back-end database server.
4. Shared-nothing database systems.
5. Fork-join systems with separate per-process address spaces.
6. Process-based parallelism, such as the Erlang language.
7. Private variables, for example, C-language on-stack auto variables, in threaded environments.

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, to say nothing of leading to greater performance and scalability coupled with reduced complexity.

Chapter 9

Deferred Processing

The strategy of deferring work goes back before the dawn of recorded history. It has occasionally been derided as procrastination or even as sheer laziness. However, in the last few decades workers have recognized this strategy's value in simplifying and streamlining parallel algorithms [KL80, Mas92]. Believe it or not, "laziness" in parallel programming often outperforms and scales better than does industriousness! These performance and scalability benefits stem from the fact that deferring work often enables weakening of synchronization primitive, thereby reducing synchronization overhead. General approaches work deferral include reference counting (Section 9.1), sequence locking (Section 9.2), and RCU (Section 9.3).

9.1 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 premature disposal, there would be no need for the reference counter in the first place. But if the object can be disposed of, what prevents disposal 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.
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. If a thread does not have a reference to a given object, it may obtain one with the help of another thread that already has a reference.

Acquisition Synchronization	Release Synchronization		
	Locking	Reference Counting	RCU
Locking	-	CAM	CA
Reference Counting	A	AM	A
RCU	CA	MCA	CA

Table 9.1: Reference Counting and Synchronization Mechanisms

3. An existence guarantee is provided for the object, preventing it from being freed while some other entity might be attempting to acquire a reference. Existence guarantees are often provided by automatic garbage collectors, and, as will be seen in Section 9.3, by RCU.
4. A type-safety guarantee is provided for the object. An additional identity check must be performed once the reference is acquired. Type-safety guarantees can be provided by special-purpose memory allocators, for example, by the `SLAB_DESTROY_BY_RCU` feature within the Linux kernel, as will be seen in Section 9.3.

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: Reference counting, sequence locking, and RCU.

Quick Quiz 9.1: 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 9.1. 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,¹ 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 9.1.3. 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.

9.1.1 Implementation of Reference-Counting Categories

Simple counting protected by locking (“-”) is described in Section 9.1.1.1, atomic counting with no memory barriers (“A”) is described in Section 9.1.1.2 atomic counting with acquisition memory barrier (“AM”) is described in Section 9.1.1.3, and atomic counting with check and release memory barrier (“CAM”) is described in Section 9.1.1.4.

¹ With `atomic_read()` and `ATOMIC_INIT()` being the exceptions that prove the rule.

9.1.1.1 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 9.1 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.

```

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 9.1: Simple Reference-Count API

9.1.1.2 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 9.2.

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

The `kref` structure itself, consisting of a single atomic data item, is shown in lines 1-3 of Figure 9.2. 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 atomically decrements the counter, and if the result is zero, line 24 invokes the specified `release()` function and line 24 returns, informing the caller that `release()` was invoked. Otherwise, `kref_put()` returns zero, informing

```

1 struct kref {
2     atomic_t refcount;
3 };
4
5 void kref_init(struct kref *kref)
6 {
7     atomic_set(&kref->refcount, 1);
8 }
9
10 void kref_get(struct kref *kref)
11 {
12     WARN_ON(!atomic_read(&kref->refcount));
13     atomic_inc(&kref->refcount);
14 }
15
16 static inline int
17 kref_sub(struct kref *kref, unsigned int count,
18           void (*release)(struct kref *kref))
19 {
20     WARN_ON(release == NULL);
21
22     if (atomic_sub_and_test((int) count,
23                           &kref->refcount)) {
24         release(kref);
25         return 1;
26     }
27     return 0;
28 }
```

Figure 9.2: Linux Kernel `kref` API

the caller that `release()` was not called.

Quick Quiz 9.3: Suppose that just after the `atomic_sub_and_test()` on line 22 of Figure 9.2 is invoked, that some other CPU invokes `kref_get()`. Doesn’t this result in that other CPU now having an illegal reference to a released object? ■

Quick Quiz 9.4: Suppose that `kref_sub()` returns zero, indicating that the `release()` function was not invoked. Under what conditions can the caller rely on the continued existence of the enclosing object? ■

Quick Quiz 9.5: Why not just pass `kfree()` as the `release` function? ■

9.1.1.3 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 9.3.

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()`

```

1 static inline
2 struct dst_entry * dst_clone(struct dst_entry * dst)
3 {
4     if (dst)
5         atomic_inc(&dst->__refcnt);
6     return dst;
7 }
8
9 static inline
10 void dst_release(struct dst_entry * dst)
11 {
12     if (dst) {
13         WARN_ON(atomic_read(&dst->__refcnt) < 1);
14         smp_mb__before_atomic_dec();
15         atomic_dec(&dst->__refcnt);
16     }
17 }

```

Figure 9.3: Linux Kernel dst_clone API

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.

9.1.1.4 Atomic Counting With Check and Release Memory Barrier

Consider a situation where the caller must be able to acquire a new reference to an object to which it does not already hold a reference. The fact that initial reference-count acquisition can now 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 9.6: Why can't the check for a zero ref-

erence 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 9.4.

```

1 struct file *fget(unsigned int fd)
2 {
3     struct file *file;
4     struct files_struct *files = current->files;
5
6     rcu_read_lock();
7     file = fcheck_files(files, fd);
8     if (file) {
9         if (!atomic_inc_not_zero(&file->f_count)) {
10             rcu_read_unlock();
11             return NULL;
12         }
13     }
14     rcu_read_unlock();
15     return file;
16 }
17
18 struct file *
19 fcheck_files(struct files_struct *files, unsigned int fd)
20 {
21     struct file * file = NULL;
22     struct fdtable *fdt = rcu_dereference((files)->fdt);
23
24     if (fd < fdt->max_fds)
25         file = rcu_dereference(fdt->fd[fd]);
26     return file;
27 }
28
29 void fput(struct file *file)
30 {
31     if (atomic_dec_and_test(&file->f_count))
32         call_rcu(&file->f_u.fu_rcuhead, file_free_rcu);
33 }
34
35 static void file_free_rcu(struct rcu_head *head)
36 {
37     struct file *f;
38
39     f = container_of(head, struct file, f_u.fu_rcuhead);
40     kmem_cache_free(filp_cachep, f);
41 }

```

Figure 9.4: Linux Kernel fget/fput API

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 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.

9.1.2 Hazard Pointers

All of the reference-counting mechanisms discussed in the previous section require some other mechanism to prevent the data element from being deleted while the reference count is being acquired. This other mechanism might be a pre-existing reference held on that data element, locking, RCU, or atomic operations, but all of them either degrade performance and scalability or restrict use cases.

```

1 int hp_store(void **p, void **hp)
2 {
3     void *tmp;
4
5     tmp = ACCESS_ONCE(*p);
6     ACCESS_ONCE(*hp) = tmp;
7     smp_mb();
8     if (tmp != ACCESS_ONCE(*p) ||
9         tmp == HAZPTR_POISON) {
10        ACCESS_ONCE(*hp) = NULL;
11        return 0;
12    }
13    return 1;
14 }
15
16 void hp_erase(void **hp)
17 {
18     smp_mb();
19     ACCESS_ONCE(*hp) = NULL;
20     hp_free(hp);
21 }
```

Figure 9.5: Hazard-Pointer Storage and Erasure

One way of avoiding these problems is to implement the reference counters inside out, that is, rather than incrementing an integer stored in the data element, instead store a pointer to that data element in per-CPU (or per-thread) lists. Each element of these lists is called a *hazard pointer* [Mic04].² The value of a given data element's “virtual reference counter” can then be obtained by counting the number of hazard pointers referencing that element. Therefore, if that element has been rendered inaccessible to readers, and there are no longer any hazard pointers referencing it, that element may safely be freed.

Of course, this means that hazard-pointer acquisition must be carried out quite carefully in order to avoid destructive races with concurrent deletion. One implementation is shown in Figure 9.5, which shows `hp_store()` on lines 1-13 and `hp_erase()` on lines 15-20. The `smp_mb()` primitive will be described in detail in Section 14.2, but may be ignored for the purposes of this brief overview.

The `hp_store()` function records a hazard pointer at `hp` for the data element whose pointer is referenced by `p`, while checking for concurrent modifications. If a concurrent modification occurred, `hp_store()` refuses to record a hazard pointer, and returns zero to indicate that the caller must restart its traversal from the beginning. Otherwise, `hp_store()` returns one to indicate that it successfully recorded a hazard pointer for the data element.

Quick Quiz 9.7: Why does `hp_store()` in Figure 9.5 take a double indirection to the data element?

² Also independently invented by others [HLM02].

Why not `void *` instead of `void **`? ■

Quick Quiz 9.8: Why does `hp_store()`'s caller need to restart its traversal from the beginning in case of failure? Isn't that inefficient for large data structures? ■

Quick Quiz 9.9: Given that papers on hazard pointers use the bottom bits of each pointer to mark deleted elements, what is up with `HAZPTR_POISON`? ■

Because algorithms using hazard pointers might be restarted at any step of their traversal through the data structure, such algorithms must typically take care to avoid making any changes to the data structure until after they have acquired all relevant hazard pointers.

Quick Quiz 9.10: But don't these restrictions on hazard pointers also apply to other forms of reference counting? ■

These restrictions result in great benefits to readers, courtesy of the fact that the hazard pointers are stored local to each CPU/thread, which in turn allows traversals of the data structures themselves to be carried out in a completely read-only fashion. Referring back to Figure 5.29 on page 60, hazard pointers enable the CPU caches to do resource replication, which in turn allows weakening of the parallel-access-control mechanism, thus boosting performance and scalability. Performance comparisons with other mechanisms may be found in Chapter 10 and in other publications [HMBW07, McK13, Mic04].

9.1.3 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` (non-zero) 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` (non-zero) if the increment occurred. Issues a memory barrier and disables compiler optimizations that might otherwise move memory references across this primitive.
- `int atomic_read(atomic_t *var);` Returns the integer value of the referenced variable. This need not be an atomic operation, and it need not issue any memory-barrier instructions. Instead of thinking of as “an atomic read,” think of it as “a normal read from an atomic variable.”
- `void atomic_set(atomic_t *var, int val);` Sets the value of the referenced atomic variable to “`val`”. This need not be an atomic operation, and it is not required to either issue memory barriers or disable compiler optimizations. Instead of thinking of as “an atomic set,” think of it as “a normal set of an atomic variable.”
- `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 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_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.

Quick Quiz 9.11: An `atomic_read()` and an `atomic_set()` that are non-atomic? Is this some kind of bad joke??? ■

9.1.4 Counter Optimizations

In some cases where increments and decrements are common, but checks for zero are rare, it makes sense to maintain per-CPU or per-task counters, as was discussed in Chapter 5. See the paper on sleepable read-copy update (SRCU) for an example of this technique applied to RCU [McK06]. 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 addition, the primitives such as `synchronize_srcu()` 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 references are frequently acquired and released, but where it is rarely necessary to check for a zero reference count.

However, it is usually 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.

Quick Quiz 9.12: But hazard pointers don't write to the data structure! ■

It is therefore worthwhile to look into synchronization mechanisms that do not require readers to do writes at all. One such synchronization mechanism, sequence locks, is covered in the next section.

9.2 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, like hazard pointers, sequence locks force readers to *retry* an operation if they

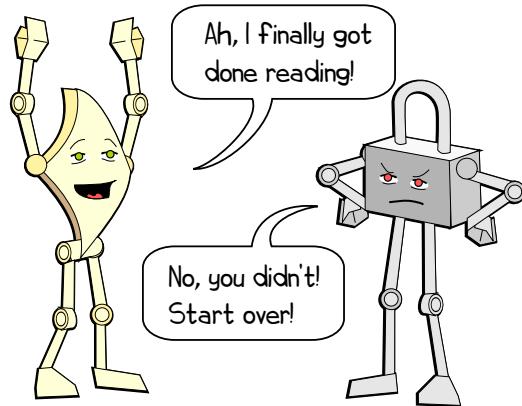


Figure 9.6: Reader And Uncooperative Sequence Lock

detect activity from a concurrent writer. As can be seen from Figure 9.6, it is important to design code using sequence locks so that readers very rarely need to retry.

Quick Quiz 9.13: Why isn't this sequence-lock discussion in Chapter 7, 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 snapshots 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_seqretry()` functions, as shown in Figure 9.7, 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_`

```
1 do {
2     seq = read_seqbegin(&test_seqlock);
3     /* read-side access. */
4 } while (read_seqretry(&test_seqlock, seq));
```

Figure 9.7: Sequence-Locking Reader

```
1 write_seqlock(&test_seqlock);
2 /* Update */
3 write_sequnlock(&test_seqlock);
```

Figure 9.8: Sequence-Locking Writer

`sequnlock()` functions, as shown in Figure 9.8, 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 9.14: 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 9.9 (`seqlock.h`). The `seqlock_t` data structure 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 begins a sequence-lock read-side critical section. Line 17 takes a snapshot of the sequence counter, and line 18 orders this snapshot operation before the caller's critical section. Line 19 checks to see if the snapshot is odd, indicating 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 9.15: Why bother with the check on line 19 of `read_seqbegin()` in Figure 9.9? 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 9.16: Why is the `smp_mb()` on line 29 of Figure 9.9 needed? ■

Quick Quiz 9.17: Can't weaker memory barriers be used in the code in Figure 9.9? ■

Quick Quiz 9.18: What prevents sequence-locking updaters from starving readers? ■

Lines 34-39 show `write_seqlock()`, which simply acquires the lock, increments the sequence number, and executes a memory barrier to ensure that this increment is ordered before the caller's critical section.

```

1 typedef struct {
2     unsigned long seq;
3     spinlock_t lock;
4 } seqlock_t;
5
6 static void seqlock_init(seqlock_t *slp)
7 {
8     slp->seq = 0;
9     spin_lock_init(&slp->lock);
10 }
11
12 static unsigned long read_seqbegin(seqlock_t *slp)
13 {
14     unsigned long s;
15
16     repeat:
17         s = ACCESS_ONCE(slp->seq);
18         smp_mb();
19         if (unlikely(s & 1))
20             goto repeat;
21         return s;
22 }
23
24 static int read_seqretry(seqlock_t *slp,
25                         unsigned long oldseq)
26 {
27     unsigned long s;
28
29     smp_mb();
30     s = ACCESS_ONCE(slp->seq);
31     return s != oldseq;
32 }
33
34 static void write_seqlock(seqlock_t *slp)
35 {
36     spin_lock(&slp->lock);
37     ++slp->seq;
38     smp_mb();
39 }
40
41 static void write_sequnlock(seqlock_t *slp)
42 {
43     smp_mb();
44     ++slp->seq;
45     spin_unlock(&slp->lock);
46 }
```

Figure 9.9: Sequence-Locking Implementation

Lines 41-46 show `write_sequunlock()`, which executes 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 9.19: What if something else serializes writers, so that the lock is not needed? ■

Quick Quiz 9.20: Why isn't `seq` on line 2 of Figure 9.9 `unsigned` rather than `unsigned long`? 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 17.2. The limitations of sequence locking are: (1) Sequence locking restricts updates and (2) sequence locking does not permit traversal of pointers to objects that might be freed by updaters. These limitations are of course overcome by transactional memory, but can also be overcome by combining other synchronization primitives with sequence locking.

Sequence locks allow writers to defer readers, but not vice versa. This can result in unfairness and even starvation 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 read-side failure, let alone starvation. In addition, it would also be nice to overcome sequence locking's limitations with pointers. The following section presents a synchronization mechanism with exactly these properties.

9.3 Read-Copy Update (RCU)

This section covers RCU from a number of different perspectives. Section 9.3.1 provides the classic introduction to RCU, Section 9.3.2 covers fundamental RCU concepts, Section 9.3.3 introduces some common uses of RCU, Section 9.3.4 presents the Linux-kernel API, Section 9.3.5 covers a sequence of "toy" implementations of user-level RCU, and finally Section 9.3.6 provides some RCU exercises.

9.3.1 Introduction to RCU

The prototypical example of an RCU-protected data structure is a networking routing table that maps from destination address to the corresponding destination port. Normally, this mapping will not change, but it could change

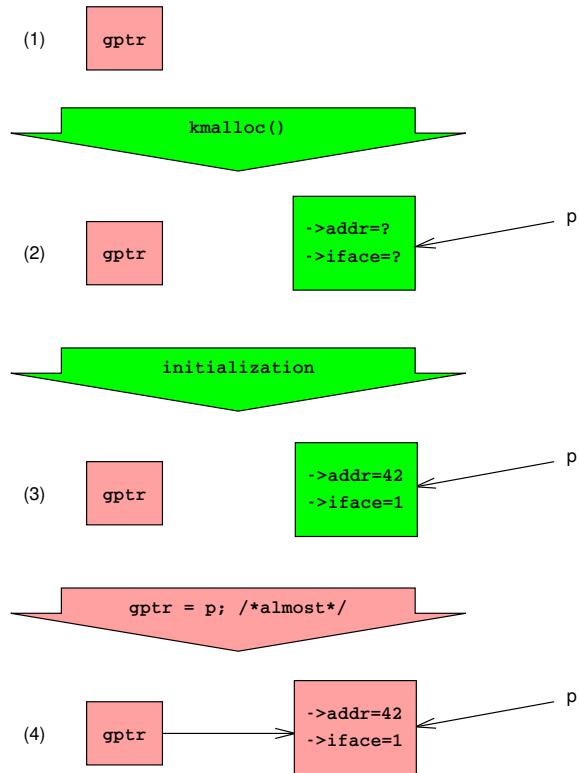


Figure 9.10: Insertion With Concurrent Readers

at any time due to hardware reconfigurations or failures. Because network packet transmissions are much more frequent than hardware reconfigurations or failures, the overhead of lookups should be minimized even if doing so causes updates to become quite expensive. In fact, if possible, the overhead of lookups should be the same as that of a single-threaded lookup. In this limiting case, the parallel lookups would execute the same sequence of assembly language instructions as would a single-threaded lookup.

Although this is a nice goal, it does raise some serious implementability questions. Insertion and deletion will be treated separately.

A classic approach for insertion is shown in Figure 9.10. The first row shows the default state, with `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

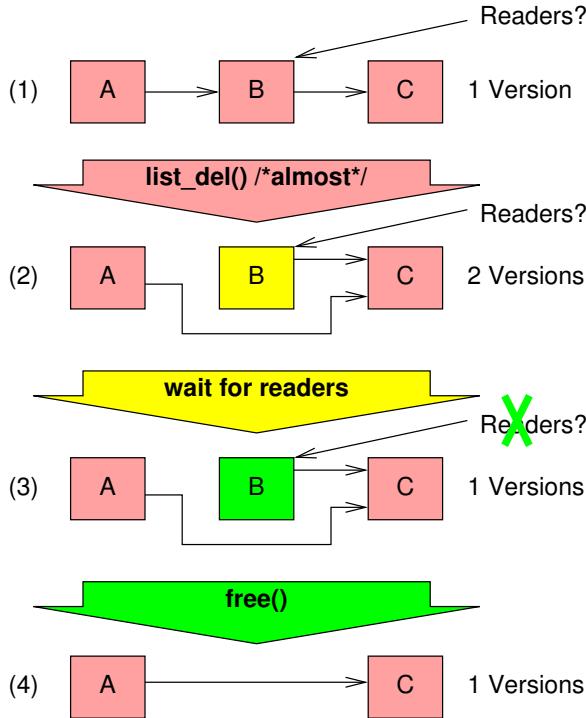


Figure 9.11: Deletion From Linked List With Concurrent Readers

assign `gptr` to reference this new element.³ 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 `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.⁴

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 9.11. This list initially contains elements A, B, and C, and we

³ 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 9.3.2.

⁴ 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 9.3.2.

need to remove element B. First, we use `list_del()` to carry out the removal,⁵ at which point all new readers will see element B as having been deleted from the list. However, there might be old readers still referencing this element. Once all these old readers have finished, we can safely free element B, resulting in the situation shown at the bottom of the figure.

But how can we tell when the readers are finished?

It is tempting to consider a reference-counting scheme, but Figure 5.3 in Chapter 5 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.

One constraint that fits well with 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.⁶ 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.

Let us impose this same constraint on reader threads traversing the linked list: such threads are not allowed to block until after completing their traversal. Returning to the second row of Figure 9.11, where the updater has just completed executing `list_del()`, imagine that CPU 0 executes a context switch. Because readers are not permitted to block while traversing the linked list, we are guaranteed that all prior readers that might have been running on CPU 0 will have completed. Extending this line of reasoning to the other CPUs, once each CPU

⁵ And yet again, this approximates reality, which will be expanded on in Section 9.3.2.

⁶ In contrast, an infinite loop in a preemptible 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.

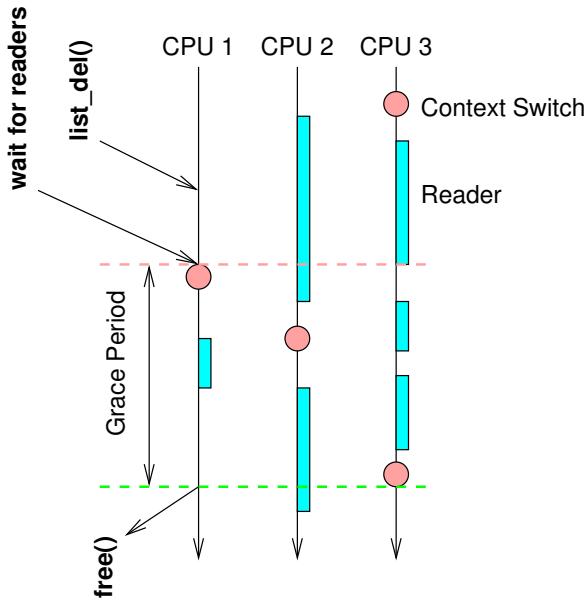


Figure 9.12: Waiting for Pre-Existing Readers

has been observed executing a context switch, we are guaranteed that all prior readers have completed, and that there are no longer any reader threads referencing element B. The updater can then safely free element B, resulting in the state shown at the bottom of Figure 9.11.

A schematic of this approach is shown in Figure 9.12, 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 implementation is exceedingly simple:

```
1 for_each_online_cpu(cpu)
2   run_on(cpu);
```

The `for_each_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_each_online_cpu()` has completed, each CPU has executed a context switch, which in turn guarantees that all pre-existing reader threads have completed.

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.

9.3.2 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 updaters 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, using replication and weakening optimizations in a manner similar to hazard pointers, but without the need for read-side retries. In some cases (non-preemptible kernels), RCU's read-side primitives have zero overhead.

Quick Quiz 9.21: But doesn't Section 9.2'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 9.3.2.1 describes the publish-subscribe mechanism used for insertion, Section 9.3.2.2 describes how waiting for pre-existing RCU readers enabled deletion, and Sec-

```

1 struct foo {
2     int a;
3     int b;
4     int c;
5 };
6 struct foo *gp = NULL;
7
8 /* . . . */
9
10 p = kmalloc(sizeof(*p), GFP_KERNEL);
11 p->a = 1;
12 p->b = 2;
13 p->c = 3;
14 gp = p;

```

Figure 9.13: Data Structure Publication (Unsafe)

tion 9.3.2.3 discusses how maintaining multiple versions of recently updated objects permits concurrent insertions and deletions. Finally, Section 9.3.2.4 summarizes RCU fundamentals.

9.3.2.1 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. 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 9.13 (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:

```

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:

```

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:⁷

```

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 read-side critical section. Their purpose is explained in Section 9.3.2.2, 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 prac-

⁷ In the Linux kernel, `rcu_dereference()` is implemented via a volatile cast, and, on DEC Alpha, a memory barrier instruction. In the C11 and C++11 standards, `memory_order_consume` is intended to provide longer-term support for `rcu_dereference()`, but no compilers implement this natively yet. (They instead strengthen `memory_order_consume` to `memory_order_acquire`, thus emitting a needless memory-barrier instruction on weakly ordered systems.)

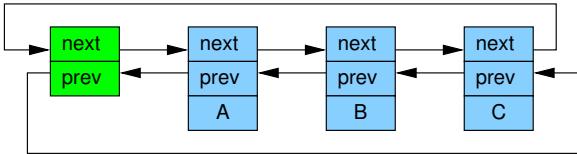


Figure 9.14: Linux Circular Linked List



Figure 9.15: Linux Linked List Abbreviated

```

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 = kmalloc(sizeof(*p), GFP_KERNEL);
12 p->a = 1;
13 p->b = 2;
14 p->c = 3;
15 list_add_rcu(&p->list, &head);

```

Figure 9.16: RCU Data Structure Publication

Since 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 9.14, where the green (leftmost) boxes represent the list header and the blue (rightmost three) boxes represent the elements in the list. This notation is cumbersome, and will therefore be abbreviated as shown in Figure 9.15, which shows only the non-header (blue) elements.

Adapting the pointer-publish example for the linked list results in the code shown in Figure 9.16.

Line 15 must be protected by some synchronization mechanism (most commonly some sort of lock) to prevent multiple `list_add_rcu()` instances from executing concurrently. However, such synchronization does not prevent this `list_add()` instance from executing concurrently with RCU readers.

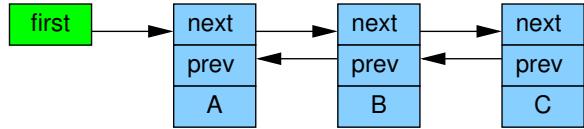


Figure 9.17: Linux Linear Linked List

```

1 struct foo {
2     struct hlist_node *list;
3     int a;
4     int b;
5     int c;
6 };
7 HLIST_HEAD(head);
8
9 /* . . . */
10
11 p = kmalloc(sizeof(*p), GFP_KERNEL);
12 p->a = 1;
13 p->b = 2;
14 p->c = 3;
15 hlist_add_head_rcu(&p->list, &head);

```

Figure 9.18: RCU hlist Publication

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, inserting it at the head of the specified list, guaranteeing that the corresponding `list_for_each_entry_rcu()` invocation will properly subscribe to this same entry.

Quick Quiz 9.22: 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 9.17. 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 9.15.

Publishing a new element to an RCU-protected `hlist` is quite similar to doing so for the circular list, as shown in Figure 9.18.

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 9.23: 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 9.2, 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.

9.3.2.2 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, 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 [McK06] 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 [McK07f,

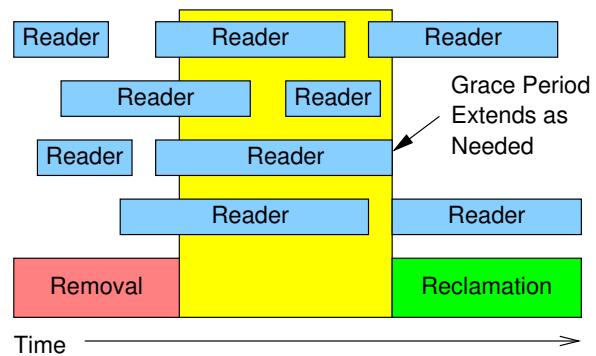


Figure 9.19: Readers and RCU Grace Period

McK07a].

In particular, as shown in Figure 9.19, 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 9.20, adapted from those in Section 9.3.2.1, 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 gives RCU (“read-copy update”) its name: while permitting concurrent *reads*, line 16 *copies* and lines 17-19 do an *update*.

As discussed in Section 9.3.1, the `synchronize_rcu()` primitive can be quite simple (see Section 9.3.5 for additional “toy” RCU implementations). However,

Category	Publish	Retract	Subscribe
Pointers	rcu_assign_pointer() list_add_rcu()	rcu_assign_pointer(..., NULL)	rcu_dereference()
Lists	list_add_tail_rcu() list_replace_rcu() hlist_add_after_rcu()	list_del_rcu()	list_for_each_entry_rcu()
Hlists	hlist_add_before_rcu() hlist_add_head_rcu() hlist_replace_rcu()	hlist_del_rcu()	hlist_for_each_entry_rcu()

Table 9.2: RCU Publish and Subscribe Primitives

```

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 9.20: Canonical RCU Replacement Example

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.

9.3.2.3 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.

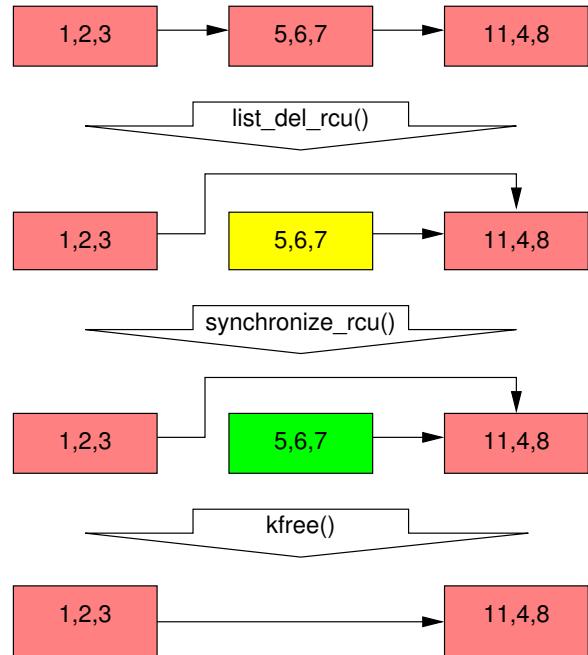


Figure 9.21: RCU Deletion From Linked List

Example 1: Maintaining Multiple Versions During Deletion We can now revisit the deletion example from Section 9.3.1, 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 9.20 to read as follows:

```

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 9.21. The triples in each element represent the values of fields a, b, and c, respectively. The red-shaded elements in-

dicate that RCU readers might be holding references to them, so in the initial state at the top of the diagram, all elements are shaded red. 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 9.21. 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 in the second row of the figure is now 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 9.21. 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 9.21. At this point, we have completed the deletion of element `5, 6, 7`. 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 9.20:

```

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 9.22.

As before, the triples in each element represent the values of fields `a`, `b`, and `c`, respectively. The red-shaded

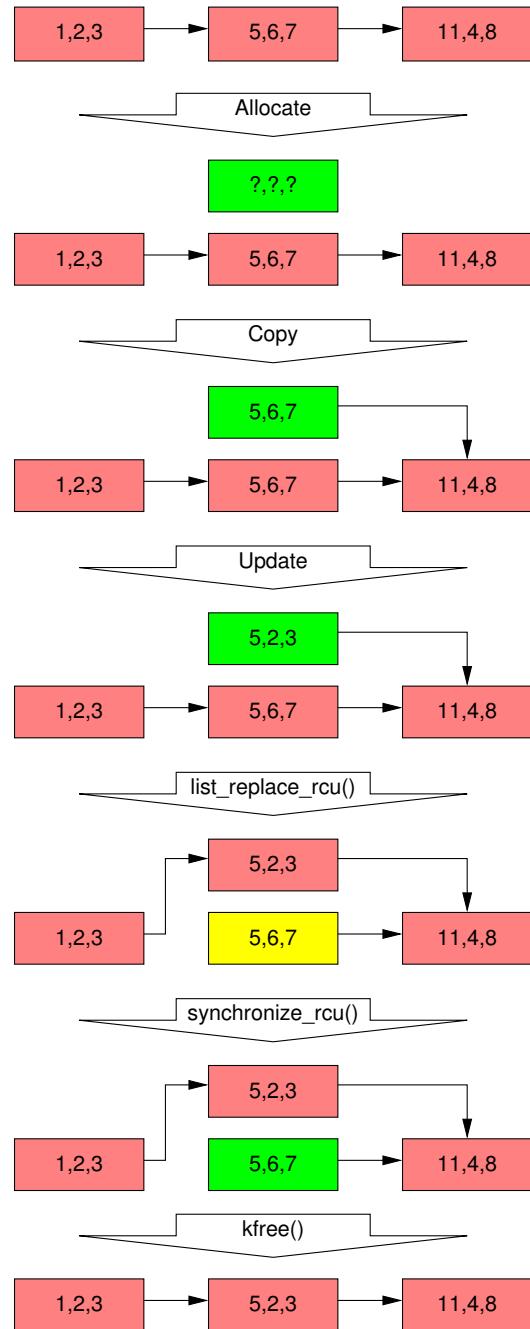


Figure 9.22: RCU Replacement in Linked List

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 `kmalloc()`s a replacement element, as follows, resulting in the state as shown in the second row of Figure 9.22. 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 9.22. 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 9.22.

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 9.22. 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 `synchronize_rcu()` on line 6 returns, a grace period will have elapsed, and so all reads that started before the `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 9.22. 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 9.22.

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 9.3.2.3.

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 9.24: How would you modify the deletion example to permit more than two versions of the list to be active? ■

Quick Quiz 9.25: 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.

9.3.2.4 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 9.26: 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.

9.3.3 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 9.3. Following the sections listed in this table, Section 9.3.3.8 provides a summary.

Mechanism RCU Replaces	Section
Reader-writer locking	Section 9.3.3.1
Restricted reference-counting mechanism	Section 9.3.3.2
Bulk reference-counting mechanism	Section 9.3.3.3
Poor man's garbage collector	Section 9.3.3.4
Existence Guarantees	Section 9.3.3.5
Type-Safe Memory	Section 9.3.3.6
Wait for things to finish	Section 9.3.3.7

Table 9.3: RCU Usage

9.3.3.1 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 a lightweight reader-writer lock [HW92]⁸ before implementing a general-purpose RCU implementation back in the early 1990s. Each and every one of the uses I envisioned for the lightweight reader-writer lock was instead implemented using RCU. In fact, it was more than three years before the lightweight reader-writer lock 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 realtime 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 9.23.

Quick Quiz 9.27: 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

⁸ Similar to `brlock` in the 2.4 Linux kernel and to `lglock` in more recent Linux kernels.

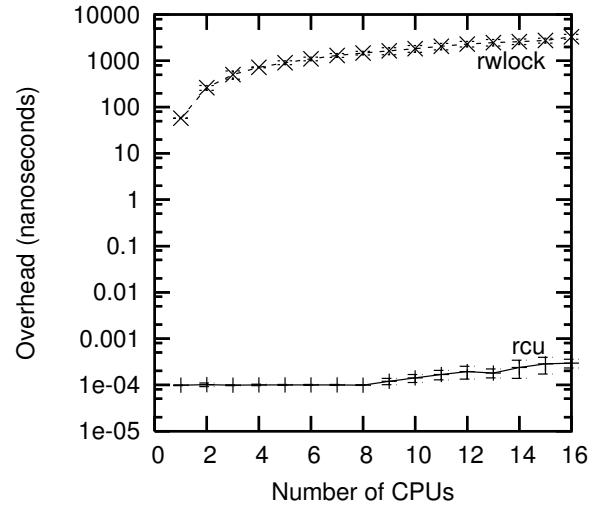


Figure 9.23: Performance Advantage of RCU Over Reader-Writer Locking

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 9.24. 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 9.24 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 9.25 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 9.28: 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.

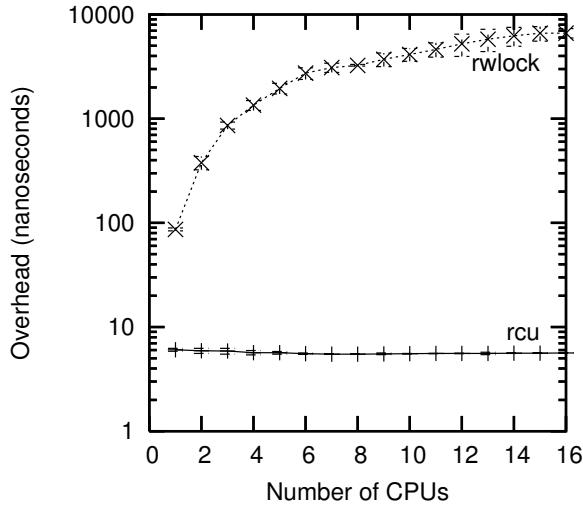


Figure 9.24: Performance Advantage of Preemptible RCU Over Reader-Writer Locking

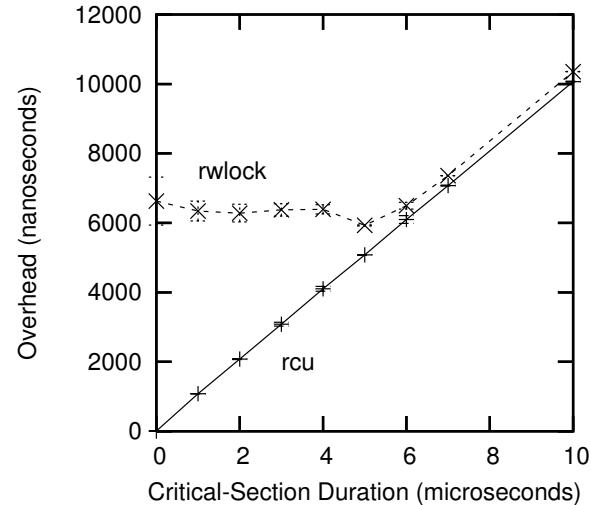


Figure 9.25: Comparison of RCU to Reader-Writer Locking as Function of Critical-Section Duration

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 9.29: 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:

```

1 rCU_read_lock();
2 list_for_each_entry_rcu(p, &head, list_field) {
3     do_something_with(p);
4     if (need_update(p)) {
5         spin_lock(my_lock);
6         do_update(p);
7         spin_unlock(&my_lock);
8     }
9 }
10 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.

Quick Quiz 9.30: Immunity to both deadlock and priority inversion??? Sounds too good to be true. Why should I believe that this is even possible? ■

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 [McK07c, 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

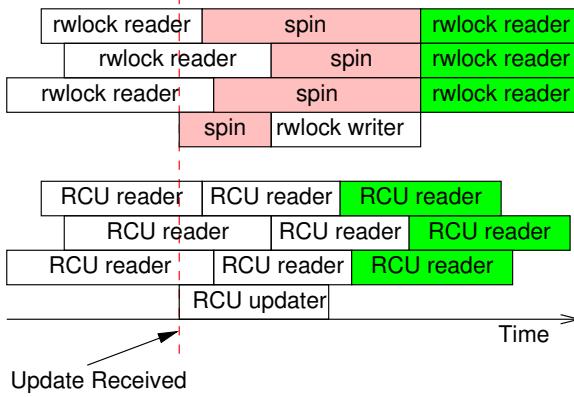


Figure 9.26: Response Time of RCU vs. Reader-Writer Locking

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 9.26.

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 shading of the rightmost boxes. 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 green-shaded rightmost RCU readers are *guaranteed* to see the updated values.

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 9.1.

Low-Priority RCU Readers Can Block High-Priority Reclaimers In Realtime RCU [GMTW08], SRCU [McK06], or QRCU [McK07e] (see Section 12.1.4), 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 [McK07c, 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 9.3.5, 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

```

1 struct el {
2     struct list_head lp;
3     long key;
4     spinlock_t mutex;
5     int data;
6     /* Other data fields */
7 };
8 DEFINE_RWLOCK(listmutex);
9 LIST_HEAD(head);

```

```

1 struct el {
2     struct list_head lp;
3     long key;
4     spinlock_t mutex;
5     int data;
6     /* Other data fields */
7 };
8 DEFINE_SPINLOCK(listmutex);
9 LIST_HEAD(head);

```

Figure 9.27: Converting Reader-Writer Locking to RCU: Data

```

1 int search(long key, int *result)
2 {
3     struct el *p;
4
5     read_lock(&listmutex);
6     list_for_each_entry(p, &head, lp) {
7         if (p->key == key) {
8             *result = p->data;
9             read_unlock(&listmutex);
10            return 1;
11        }
12    }
13    read_unlock(&listmutex);
14    return 0;
15 }

```

```

1 int search(long key, int *result)
2 {
3     struct el *p;
4
5     rCU_read_lock();
6     list_for_each_entry_rcu(p, &head, lp) {
7         if (p->key == key) {
8             *result = p->data;
9             rCU_read_unlock();
10            return 1;
11        }
12    }
13    rCU_read_unlock();
14    return 0;
15 }

```

Figure 9.28: Converting Reader-Writer Locking to RCU: Search

```

1 int delete(long key)
2 {
3     struct el *p;
4
5     write_lock(&listmutex);
6     list_for_each_entry(p, &head, lp) {
7         if (p->key == key) {
8             list_del(&p->lp);
9             write_unlock(&listmutex);
10            kfree(p);
11            return 1;
12        }
13    }
14    write_unlock(&listmutex);
15    return 0;
16 }

```

```

1 int delete(long key)
2 {
3     struct el *p;
4
5     spin_lock(&listmutex);
6     list_for_each_entry(p, &head, lp) {
7         if (p->key == key) {
8             list_del_rcu(&p->lp);
9             spin_unlock(&listmutex);
10            synchronize_rcu();
11            kfree(p);
12            return 1;
13        }
14    }
15    spin_unlock(&listmutex);
16    return 0;
17 }

```

Figure 9.29: Converting Reader-Writer Locking to RCU: Deletion

locking to RCU is quite simple, as shown in Figures 9.27, 9.28, and 9.29, all taken from Wikipedia [MPA⁺06].

More-elaborate cases of replacing reader-writer locking with RCU are beyond the scope of this document.

9.3.3.2 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:

```
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`:

```
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 9.31: 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? ■

Of course, RCU can also be combined with traditional reference counting, as has been discussed on LKML and as summarized in Section 9.1.

But why bother? Again, part of the answer is performance, as shown in Figure 9.30, again showing data taken on a 16-CPU 3GHz Intel x86 system.

Quick Quiz 9.32: 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

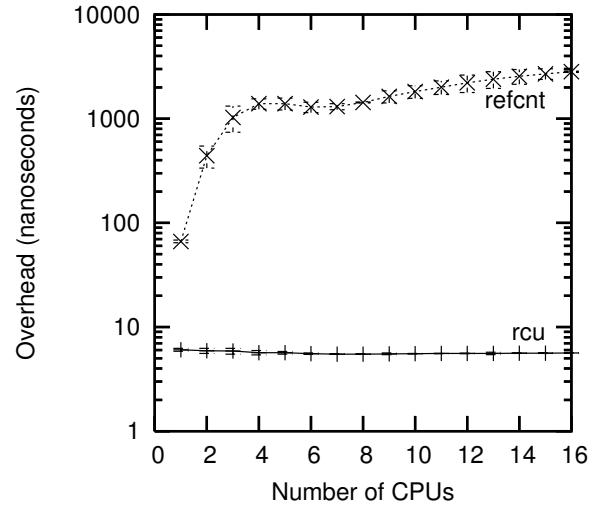


Figure 9.30: Performance of RCU vs. Reference Counting

critical sections, as shown Figure 9.31 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.

9.3.3.3 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

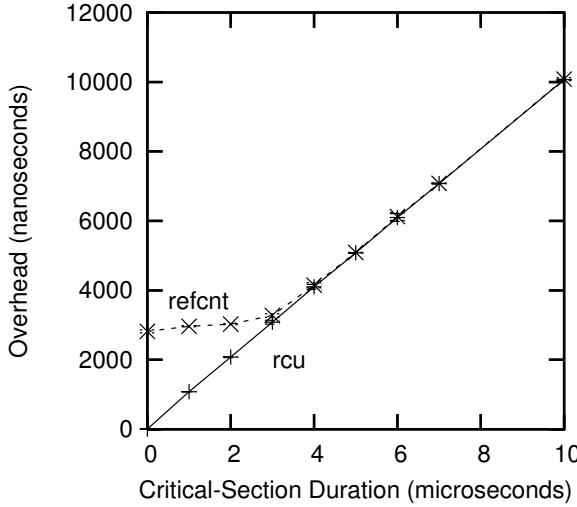


Figure 9.31: Response Time of RCU vs. Reference Counting

(SRCU) [McK06]. 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 interrupt handlers or from non-maskable interrupt (NMI) 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.

9.3.3.4 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

```

1 int delete(int key)
2 {
3     struct element *p;
4     int b;
5
6     b = hashfunction(key);
7     rCU_read_lock();
8     p = rCU_dereference(hashtable[b]);
9     if (p == NULL || p->key != key) {
10         rCU_read_unlock();
11         return 0;
12     }
13     spin_lock(&p->lock);
14     if (hashtable[b] == p && p->key == key) {
15         rCU_read_unlock();
16         rCU_assign_pointer(hashtable[b], NULL);
17         spin_unlock(&p->lock);
18         synchronize_rcu();
19         kfree(p);
20         return 1;
21     }
22     spin_unlock(&p->lock);
23     rCU_read_unlock();
24     return 0;
25 }
```

Figure 9.32: Existence Guarantees Enable Per-Element Locking

(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.

9.3.3.5 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 7.4 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 9.32 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 9.33: What if the element we need to delete is not the first element of the list on line 9 of Figure 9.32? ■

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 9.34: Why is it OK to exit the RCU read-side critical section on line 15 of Figure 9.32 before releasing the lock on line 17? ■

Quick Quiz 9.35: Why not exit the RCU read-side critical section on line 23 of Figure 9.32 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 9.3.3.7. They might also note the deadlock-immunity advantages over the lock-based existence guarantees discussed in Section 7.4.

9.3.3.6 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 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 9.36: 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!

9.3.3.7 RCU is a Way of Waiting for Things to Finish

As noted in Section 9.3.2 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 interrupts) permit you to implement interactions with non-maskable interrupt (NMI) handlers that would be quite difficult if using locking. 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.

```

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 = rcu_dereference(buf);
10
11    if (p == NULL)
12        return;
13    if (pcvalue >= p->size)
14        return;
15    atomic_inc(&p->entry[pcvalue]);
16 }
17
18 void nmi_stop(void)
19 {
20     struct profile_buffer *p = buf;
21
22    if (p == NULL)
23        return;
24    rcu_assign_pointer(buf, NULL);
25    synchronize_sched();
26    kfree(p);
27 }

```

Figure 9.33: Using RCU to Wait for NMIs to Finish

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 9.33.

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 interrupts 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 9.37: 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.

9.3.3.8 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.

In the meantime, Figure 9.34 shows some rough rules of thumb on where RCU is most helpful.

As shown in the blue box at the top of the figure, RCU works best if you have read-mostly data where stale and

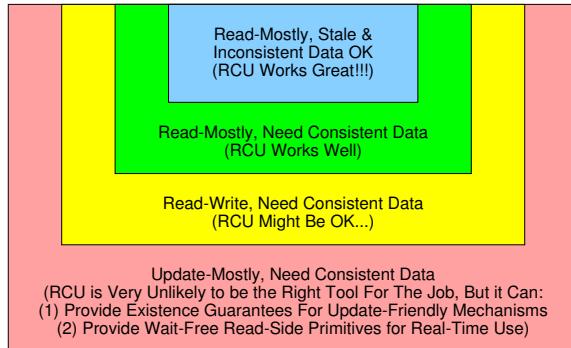


Figure 9.34: RCU Areas of Applicability

inconsistent data is permissible (but see below for more information on stale and inconsistent data). The canonical example of this case in the Linux kernel is routing tables. Because it may have taken many seconds or even minutes for the routing updates to propagate across Internet, the system has been sending packets the wrong way for quite some time. Having some small probability of continuing to send some of them the wrong way for a few more milliseconds is almost never a problem.

If you have a read-mostly workload where consistent data is required, RCU works well, as shown by the green “read-mostly, need consistent data” box. One example of this case is the Linux kernel’s mapping from user-level System-V semaphore IDs to the corresponding in-kernel data structures. Semaphores tend to be used far more frequently than they are created and destroyed, so this mapping is read-mostly. However, it would be erroneous to perform a semaphore operation on a semaphore that has already been deleted. This need for consistency is handled by using the lock in the in-kernel semaphore data structure, along with a “deleted” flag that is set when deleting a semaphore. If a user ID maps to an in-kernel data structure with the “deleted” flag set, the data structure is ignored, so that the user ID is flagged as invalid.

Although this requires that the readers acquire a lock for the data structure representing the semaphore itself, it allows them to dispense with locking for the mapping data structure. The readers therefore locklessly traverse the tree used to map from ID to data structure, which in turn greatly improves performance, scalability, and real-time response.

As indicated by the yellow “read-write” box, RCU can also be useful for read-write workloads where consistent data is required, although usually in conjunction with a

number of other synchronization primitives. For example, the directory-entry cache in recent Linux kernels uses RCU in conjunction with sequence locks, per-CPU locks, and per-data-structure locks to allow lockless traversal of pathnames in the common case. Although RCU can be very beneficial in this read-write case, such use is often more complex than that of the read-mostly cases.

Finally, as indicated by the red box at the bottom of the figure, update-mostly workloads requiring consistent data are rarely good places to use RCU, though there are some exceptions [DMS⁺12]. In addition, as noted in Section 9.3.3.6, within the Linux kernel, the `SLAB_DESTROY_BY_RCU` slab-allocator flag provides type-safe memory to RCU readers, which can greatly simplify non-blocking synchronization and other lockless algorithms.

In short, RCU is an API that includes a publish-subscribe mechanism for adding new data, a way of waiting for pre-existing RCU readers to finish, and a discipline of maintaining multiple versions to allow updates to avoid harming or unduly delaying concurrent RCU readers. This RCU API is best suited for read-mostly situations, especially if stale and inconsistent data can be tolerated by the application.

9.3.4 RCU Linux-Kernel API

This section looks at RCU from the viewpoint of its Linux-kernel API. Section 9.3.4.1 presents RCU’s wait-to-finish APIs, and Section 9.3.4.2 presents RCU’s publish-subscribe and version-maintenance APIs. Finally, Section 9.3.4.4 presents concluding remarks.

9.3.4.1 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 9.4 and 9.5, which shows the wait-for-RCU-readers portions of the non-sleepable and sleepable APIs, respectively, and by Table 9.6, 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 9.4, 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.

Attribute	RCU Classic	RCU BH	RCU Sched	Realtime RCU
Purpose	Original	Prevent DDoS attacks	Wait for preempt-disable regions, hardirqs, & NMIs	Realtime response
Availability	2.5.43	2.6.9	2.6.12	2.6.26
Read-side primitives	<code>rcu_read_lock()</code> ! <code>rcu_read_unlock()</code> !	<code>rcu_read_lock_bh()</code> <code>rcu_read_unlock_bh()</code>	<code>preempt_disable()</code> <code>preempt_enable()</code> (and friends)	<code>rcu_read_lock()</code> <code>rcu_read_unlock()</code>
Update-side primitives (synchronous)	<code>synchronize_rcu()</code> <code>synchronize_net()</code>		<code>synchronize_sched()</code>	<code>synchronize_rcu()</code> <code>synchronize_net()</code>
Update-side primitives (asynchronous/callback)	<code>call_rcu()</code> !	<code>call_rcu_bh()</code>	<code>call_rcu_sched()</code>	<code>call_rcu()</code>
Update-side primitives (wait for callbacks)	<code>rcu_barrier()</code>	<code>rcu_barrier_bh()</code>	<code>rcu_barrier_sched()</code>	<code>rcu_barrier()</code>
Type-safe memory	<code>SLAB_DESTROY_BY_RCU</code>			<code>SLAB_DESTROY_BY_RCU</code>
Read side constraints	No blocking	No irq enabling	No blocking	Only preemption and lock acquisition
Read side overhead	Preempt disable/enable (free on non-PREEMPT)	BH disable/enable	Preempt disable/enable (free on non-PREEMPT)	Simple instructions, irq disable/enable
Asynchronous update-side overhead	sub-microsecond	sub-microsecond		sub-microsecond
Grace-period latency	10s of milliseconds	10s of milliseconds	10s of milliseconds	10s of milliseconds
Non-PREEMPT_RT implementation	RCU Classic	RCU BH	RCU Classic	Preemptible RCU
PREEMPT_RT implementation	Preemptible RCU	Realtime RCU	Forced Schedule on all CPUs	Realtime RCU

Table 9.4: RCU Wait-to-Finish APIs

Attribute	SRCU	QRCU
Purpose	Sleeping readers	Sleeping readers and fast grace periods
Availability	2.6.19	
Read-side primitives	<code>srcu_read_lock()</code> <code>srcu_read_unlock()</code>	<code>qrcu_read_lock()</code> <code>qrcu_read_unlock()</code>
Update-side primitives (synchronous)	<code>synchronize_srcu()</code>	<code>synchronize_qrcu()</code>
Update-side primitives (asynchronous/callback)	N/A	N/A
Update-side primitives (wait for callbacks)	N/A	N/A
Type-safe memory		
Read side constraints	No <code>synchronize_srcu()</code>	No <code>synchronize_qrcu()</code>
Read side overhead	Simple instructions, preempt disable/enable	Atomic increment and decrement of shared variable
Asynchronous update-side overhead	N/A	N/A
Grace-period latency	10s of milliseconds	10s of nanoseconds in absence of readers
Non-PREEMPT_RT implementation	SRCU	N/A
PREEMPT_RT implementation	SRCU	N/A

Table 9.5: Sleepable RCU Wait-to-Finish APIs

If you are already familiar with RCU, these tables can serve as a useful reference.

Quick Quiz 9.38: Why do some of the cells in Table 9.4 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 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 [McK07d]. The `rcu_barrier()` primitive does this job. Note that the more recent hierarchical RCU [McK08a] implementation also adheres to “RCU Classic” semantics.

Finally, RCU may be used to provide type-safe memory [GC96], as described in Section 9.3.3.6. 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 reallocating 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 9.39: How do you prevent a huge number of RCU read-side critical sections from indefinitely blocking a `synchronize_rcu()` invocation? ■

Quick Quiz 9.40: 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 9.41: 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 9.42: 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 9.43: What happens if you mix and match RCU Classic and RCU Sched? ■

Quick Quiz 9.44: 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 9.45: Why do both SRCU and QRCU lack asynchronous `call_srcu()` or `call_qrcu()` interfaces? ■

The “SRCU” column in Table 9.5 displays a specialized RCU API that permits general sleeping in RCU read-side critical sections [McK06]. Of course, use of `synchronize_srcu()` in an SRCU read-side critical section can result in self-deadlock, so should be avoided. SRCU differs from earlier RCU implemen-

tations 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 [McK07e]. As with SRCU, use of `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 9.46: 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 four implementations behind three 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 interrupt handlers, but not including software-interrupt (`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).⁹ 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.

9.3.4.2 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

⁹ Thanks to James Bottomley for urging me to this formulation, as opposed to simply saying that there are no forward-progress guarantees.

Category	Primitives	Availability	Overhead
List traversal	list_for_each_entry_rcu()	2.5.59	Simple instructions (memory barrier on Alpha)
List update	list_add_rcu()	2.5.44	Memory barrier
	list_add_tail_rcu()	2.5.44	Memory barrier
	list_del_rcu()	2.5.44	Simple instructions
	list_replace_rcu()	2.6.9	Memory barrier
	list_splice_init_rcu()	2.6.21	Grace-period latency
Hlist traversal	hlist_for_each_entry_rcu()	2.6.8	Simple instructions (memory barrier on Alpha)
	hlist_add_after_rcu()	2.6.14	Memory barrier
	hlist_add_before_rcu()	2.6.14	Memory barrier
	hlist_add_head_rcu()	2.5.64	Memory barrier
	hlist_del_rcu()	2.5.64	Simple instructions
	hlist_replace_rcu()	2.6.15	Memory barrier
	rcu_dereference()	2.6.9	Simple instructions (memory barrier on Alpha)
Pointer traversal	rcu_assign_pointer()	2.6.10	Memory barrier

Table 9.6: RCU Publish-Subscribe and Version Maintenance APIs

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 9.47: 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` primitives in the table relate to their non-RCU counterparts in much the same way as do the `struct list_head` primitives.

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 9.48: Normally, any pointer subject to `rcu_dereference()` *must* always be updated using `rcu_assign_pointer()`. What is an exception to this rule? ■

Quick Quiz 9.49: Are there any downsides to the fact that these traversal and update primitives can be used with any of the RCU API family members? ■

9.3.4.3 Where Can RCU's APIs Be Used?

Figure 9.35 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, finally, 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.

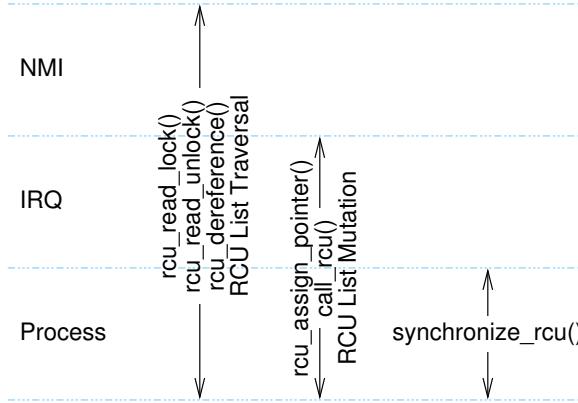


Figure 9.35: RCU API Usage Constraints

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.

9.3.4.4 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.

9.3.5 “Toy” RCU Implementations

The toy RCU implementations in this section are designed not for high performance, practicality, or any kind of production use,¹⁰ but rather for clarity. Nevertheless, you

will need a thorough understanding of Chapters 2, 3, 4, 6, and 9 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 9.3.5.1 presents a rudimentary RCU implementation based on simple locking, while Section 9.3.5.3 through 9.3.5.9 present a series of simple RCU implementations based on locking, reference counters, and free-running counters. Finally, Section 9.3.5.10 provides a summary and a list of desirable RCU properties.

9.3.5.1 Lock-Based RCU

Perhaps the simplest RCU implementation leverages locking, as shown in Figure 9.36 (`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-

```

1 static void rCU_read_lock(void)
2 {
3     spin_lock(&rcu_gp_lock);
4 }
5
6 static void rCU_read_unlock(void)
7 {
8     spin_unlock(&rcu_gp_lock);
9 }
10
11 void synchronize_rcu(void)
12 {
13     spin_lock(&rcu_gp_lock);
14     spin_unlock(&rcu_gp_lock);
15 }
```

Figure 9.36: Lock-Based RCU Implementation

¹⁰ However, production-quality user-level RCU implementations are available [Des09].

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 9.50: Why wouldn't any deadlock in the RCU implementation in Figure 9.36 also be a deadlock in any other RCU implementation? ■

Quick Quiz 9.51: Why not simply use reader-writer locks in the RCU implementation in Figure 9.36 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. 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.

9.3.5.2 Per-Thread Lock-Based RCU

Figure 9.37 (`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 9.52: Wouldn't it be cleaner to acquire all the locks, and then release them all in the loop from lines 15-18 of Figure 9.37? After all, with this change, there would be a point in time when there were no readers, simplifying things greatly. ■

Quick Quiz 9.53: Is the implementation shown in Figure 9.37 free from deadlocks? Why or why not? ■

Quick Quiz 9.54: Isn't one advantage of the RCU algorithm shown in Figure 9.37 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.

9.3.5.3 Simple Counter-Based RCU

A slightly more sophisticated RCU implementation is shown in Figure 9.38 (`rcu_rcg.h` and `rcu_rcg.c`). This implementation makes use of a global reference 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 9.3.5.1, this implementation allows parallel execution of RCU read-side critical sections. In happy contrast to the per-thread lock-based implementation shown in Section 9.3.5.2, 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 9.55: 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, this implementation still has 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.

```

1 static void rcu_read_lock(void)
2 {
3     spin_lock(&__get_thread_var(rcu_gp_lock));
4 }
5
6 static void rcu_read_unlock(void)
7 {
8     spin_unlock(&__get_thread_var(rcu_gp_lock));
9 }
10
11 void synchronize_rcu(void)
12 {
13     int t;
14
15     for_each_running_thread(t) {
16         spin_lock(&per_thread(rcu_gp_lock, t));
17         spin_unlock(&per_thread(rcu_gp_lock, t));
18     }
19 }

```

Figure 9.37: Per-Thread Lock-Based RCU Implementation

```

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 9.38: RCU Implementation Using Single Global Reference Counter

```

1 DEFINE_SPINLOCK(rcu_gp_lock);
2 atomic_t rcu_refcnt[2];
3 atomic_t rcu_idx;
4 DEFINE_PER_THREAD(int, rcu_nesting);
5 DEFINE_PER_THREAD(int, rcu_read_idx);

```

Figure 9.39: RCU Global Reference-Count Pair Data

Quick Quiz 9.56: 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 9.57: Why not simply make `rcu_read_lock()` wait when a concurrent `synchronize_rcu()` has been waiting too long in the RCU implementation in Figure 9.38? 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.

9.3.5.4 Starvation-Free Counter-Based RCU

Figure 9.40 (`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 9.39.

Design It is the two-element `rcu_refcnt` [] array that provides the freedom from starvation. The key point

```

1 static void rcu_read_lock(void)
2 {
3     int i;
4     int n;
5
6     n = __get_thread_var(rcu_nesting);
7     if (n == 0) {
8         i = atomic_read(&rcu_idx);
9         __get_thread_var(rcu_read_idx) = i;
10        atomic_inc(&rcu_refcnt[i]);
11    }
12    __get_thread_var(rcu_nesting) = n + 1;
13    smp_mb();
14 }
15
16 static void rcu_read_unlock(void)
17 {
18     int i;
19     int n;
20
21     smp_mb();
22     n = __get_thread_var(rcu_nesting);
23     if (n == 1) {
24         i = __get_thread_var(rcu_read_idx);
25         atomic_dec(&rcu_refcnt[i]);
26     }
27     __get_thread_var(rcu_nesting) = n - 1;
28 }

```

Figure 9.40: RCU Read-Side Using Global Reference-Count Pair

is that `synchronize_rcu()` is only required to wait for pre-existing readers. If a new reader starts after a given instance of `synchronize_rcu()` has already begun execution, then that instance of `synchronize_rcu()` need not wait on that new reader. At any given time, when a given reader enters its RCU read-side critical section via `rcu_read_lock()`, it increments the element of the `rcu_refcnt[]` array indicated by the `rcu_idx` variable. When that same reader exits its RCU read-side critical section via `rcu_read_unlock()`, it decrements whichever element it incremented, ignoring any possible subsequent changes to the `rcu_idx` value.

This arrangement means that `synchronize_rcu()` can avoid starvation by complementing the value of `rcu_idx`, as in `rcu_idx = !rcu_idx`. Suppose that the old value of `rcu_idx` was zero, so that the new value is one. New readers that arrive after the complement operation will increment `rcu_idx[1]`, while the old readers that previously incremented `rcu_idx[0]` will decrement `rcu_idx[0]` when they exit their RCU read-side critical sections. This means that the value of `rcu_idx[0]` will no longer be incremented, and thus will be monotonically decreasing.¹¹ This means that all that

`synchronize_rcu()` need do is wait for the value of `rcu_refcnt[0]` to reach zero.

With the background, we are ready to look at the implementation of the actual primitives.

Implementation 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 9.40 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 9.41 (`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 `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

¹¹ There is a race condition that this “monotonically decreasing” statement ignores. This race condition will be dealt with by the code for `synchronize_rcu()`. In the meantime, I suggest suspending disbelief.

```

1 void synchronize_rcu(void)
2 {
3     int i;
4
5     smp_mb();
6     spin_lock(&rcu_gp_lock);
7     i = atomic_read(&rcu_idx);
8     atomic_set(&rcu_idx, !i);
9     smp_mb();
10    while (atomic_read(&rcu_refcnt[i]) != 0) {
11        poll(NULL, 0, 10);
12    }
13    smp_mb();
14    atomic_set(&rcu_idx, i);
15    smp_mb();
16    while (atomic_read(&rcu_refcnt[!i]) != 0) {
17        poll(NULL, 0, 10);
18    }
19    spin_unlock(&rcu_gp_lock);
20    smp_mb();
21 }

```

Figure 9.41: RCU Update Using Global Reference-Count Pair

`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 9.58: Why the memory barrier on line 5 of `synchronize_rcu()` in Figure 9.41 given that there is a spin-lock acquisition immediately after? ■

Quick Quiz 9.59: Why is the counter flipped twice in Figure 9.41? 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 9.38.

Discussion 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 9.38, 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_lock()` and `rcu_read_unlock()` operations, there will be extreme memory contention on the `rcu_refcnt`

```

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 9.42: RCU Per-Thread Reference-Count Pair Data

elements, resulting in expensive cache misses. This further 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 substantial 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 9.60: 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 9.40? ■

Despite these shortcomings, one could imagine this variant of RCU being used on small tightly coupled multiprocessors, perhaps as a memory-conserving implementation that maintains API compatibility with more complex implementations. However, it would 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.

9.3.5.5 Scalable Counter-Based RCU

Figure 9.43 (`rcu_rcpl.h`) shows the read-side primitives of an RCU implementation that uses per-thread pairs of reference counters. This implementation is quite similar to that shown in Figure 9.40, the only difference being that `rcu_refcnt` is now a per-thread array (as shown in Figure 9.42). As with the algorithm in the previous section, use of this two-element array prevents readers from starving updaters. One benefit of per-thread `rcu_refcnt[]` array is that the `rcu_read_lock()` and `rcu_read_unlock()` primitives no longer perform atomic operations.

Quick Quiz 9.61: Come off it! We can see the `atomic_read()` primitive in `rcu_read_lock()`!!! So why are you trying to pretend that

```

1 static void rcu_read_lock(void)
2 {
3     int i;
4     int n;
5
6     n = __get_thread_var(rcu_nesting);
7     if (n == 0) {
8         i = atomic_read(&rcu_idx);
9         __get_thread_var(rcu_read_idx) = i;
10        __get_thread_var(rcu_refcnt)[i]++;
11    }
12    __get_thread_var(rcu_nesting) = n + 1;
13    smp_mb();
14 }
15
16 static void rcu_read_unlock(void)
17 {
18     int i;
19     int n;
20
21     smp_mb();
22     n = __get_thread_var(rcu_nesting);
23     if (n == 1) {
24         i = __get_thread_var(rcu_read_idx);
25         __get_thread_var(rcu_refcnt)[i]--;
26     }
27     __get_thread_var(rcu_nesting) = n - 1;
28 }

```

Figure 9.43: RCU Read-Side Using Per-Thread Reference-Count Pair

rcu_read_lock() contains no atomic operations???

Figure 9.44 (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 9.41, 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

```

1 static void flip_counter_and_wait(int i)
2 {
3     int t;
4
5     atomic_set(&rcu_idx, !i);
6     smp_mb();
7     for_each_thread(t) {
8         while (per_thread(rcu_refcnt, t)[i] != 0) {
9             poll(NULL, 0, 10);
10        }
11    }
12    smp_mb();
13 }
14
15 void synchronize_rcu(void)
16 {
17     int i;
18
19     smp_mb();
20     spin_lock(&rcu_gp_lock);
21     i = atomic_read(&rcu_idx);
22     flip_counter_and_wait(i);
23     flip_counter_and_wait(!i);
24     spin_unlock(&rcu_gp_lock);
25     smp_mb();
26 }

```

Figure 9.44: RCU Update Using Per-Thread Reference-Count Pair

call to rCU_read_lock().

Quick Quiz 9.62: 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? ■

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

```

1 DEFINE_SPINLOCK(rcu_gp_lock);
2 DEFINE_PER_THREAD(int [2], rcu_refcnt);
3 long rcu_idx;
4 DEFINE_PER_THREAD(int, rcu_nesting);
5 DEFINE_PER_THREAD(int, rcu_read_idx);

```

Figure 9.45: RCU Read-Side Using Per-Thread Reference-Count Pair and Shared Update Data

```

1 static void rcu_read_lock(void)
2 {
3     int i;
4     int n;
5
6     n = __get_thread_var(rcu_nesting);
7     if (n == 0) {
8         i = ACCESS_ONCE(rcu_idx) & 0x1;
9         __get_thread_var(rcu_read_idx) = i;
10    __get_thread_var(rcu_refcnt)[i]++;
11    }
12   __get_thread_var(rcu_nesting) = n + 1;
13   smp_mb();
14 }
15
16 static void rcu_read_unlock(void)
17 {
18     int i;
19     int n;
20
21     smp_mb();
22     n = __get_thread_var(rcu_nesting);
23     if (n == 1) {
24         i = __get_thread_var(rcu_read_idx);
25         __get_thread_var(rcu_refcnt)[i]--;
26     }
27   __get_thread_var(rcu_nesting) = n - 1;
28 }

```

Figure 9.46: RCU Read-Side Using Per-Thread Reference-Count Pair and Shared Update

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.

9.3.5.6 Scalable Counter-Based RCU With Shared Grace Periods

Figure 9.46 (`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 9.43 is that `rcu_idx` is now a `long` that counts freely, so that line 8 of Figure 9.46 must mask off the low-order bit. We also switched from using `atomic_read()` and `atomic_set()` to using `ACCESS_ONCE()`. The data is also quite similar,

```

1 static void flip_counter_and_wait(int ctr)
2 {
3     int i;
4     int t;
5
6     ACCESS_ONCE(rcu_idx) = ctr + 1;
7     i = ctr & 0x1;
8     smp_mb();
9     for_each_thread(t) {
10         while (per_thread(rcu_refcnt, t)[i] != 0) {
11             poll(NULL, 0, 10);
12         }
13     }
14     smp_mb();
15 }
16
17 void synchronize_rcu(void)
18 {
19     int ctr;
20     int oldctr;
21
22     smp_mb();
23     oldctr = ACCESS_ONCE(rcu_idx);
24     smp_mb();
25     spin_lock(&rcu_gp_lock);
26     ctr = ACCESS_ONCE(rcu_idx);
27     if (ctr - oldctr >= 3) {
28         spin_unlock(&rcu_gp_lock);
29         smp_mb();
30         return;
31     }
32     flip_counter_and_wait(ctr);
33     if (ctr - oldctr < 2)
34         flip_counter_and_wait(ctr + 1);
35     spin_unlock(&rcu_gp_lock);
36     smp_mb();
37 }

```

Figure 9.47: RCU Shared Update Using Per-Thread Reference-Count Pair

as shown in Figure 9.45, with `rcu_idx` now being a `long` instead of an `atomic_t`.

Figure 9.47 (`rcu_rcpls.c`) shows the implementation of `synchronize_rcu()` and its helper function `flip_counter_and_wait()`. These are similar to those in Figure 9.44. 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. There are ways of avoiding this, as was done in a production-quality real-time implementation of RCU for the Linux kernel [McK07a].

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 9.3.5.3, the read-side primitives scale extremely well, incurring roughly 115 nanoseconds of overhead regardless of the number of CPUs. The `synchronize_rcu()` primitive is still expensive, ranging from about one microsecond up to about 16 microseconds. This is nevertheless much cheaper than the roughly 200 microseconds incurred by the implementation in Section 9.3.5.5. So, despite its shortcomings, one could imagine this RCU

```

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 9.48: Data for Free-Running Counter Using RCU

implementation being used in production in real-life applications.

Quick Quiz 9.63: 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 9.46, 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.

9.3.5.7 RCU Based on Free-Running Counter

Figure 9.49 (`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 9.48. The resulting `rcu_read_lock()` implementation is extremely straightforward. Lines 3 and 4 simply add 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 5 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 10 executes a memory barrier, again to prevent the prior RCU read-side critical section from “leaking out”. Lines 11 and 12 then copy 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.

```

1 static void rCU_read_lock(void)
2 {
3     __get_thread_var(rcu_reader_gp) =
4         ACCESS_ONCE(rcu_gp_ctr) + 1;
5     smp_mb();
6 }
7
8 static void rCU_read_unlock(void)
9 {
10    smp_mb();
11    __get_thread_var(rcu_reader_gp) =
12        ACCESS_ONCE(rcu_gp_ctr);
13 }
14
15 void synchronize_rcu(void)
16 {
17     int t;
18
19     smp_mb();
20     spin_lock(&rcu_gp_lock);
21     ACCESS_ONCE(rcu_gp_ctr) += 2;
22     smp_mb();
23     for_each_thread(t) {
24         while (((per_thread(rcu_reader_gp, t) & 0x1) &&
25                 ((per_thread(rcu_reader_gp, t) -
26                  ACCESS_ONCE(rcu_gp_ctr)) < 0)) {
27             poll(NULL, 0, 10);
28         }
29     }
30     spin_unlock(&rcu_gp_lock);
31     smp_mb();
32 }

```

Figure 9.49: Free-Running Counter Using RCU

Quick Quiz 9.64: If any even value is sufficient to tell `synchronize_rcu()` to ignore a given task, why don't lines 10 and 11 of Figure 9.49 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 19 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 21. Line 20 acquires the `rcu_gp_lock` (and line 30 releases it) in order to prevent multiple `synchronize_rcu()` instances from running concurrently. Line 21 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 23-29 scan the `rcu_reader_gp` values until they all are either even (line 24) or are greater than

the global `rcu_gp_ctr` (lines 25-26). Line 27 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 31 ensures that any subsequent destruction will not be reordered into the preceding loop.

Quick Quiz 9.65: Why are the memory barriers on lines 19 and 31 of Figure 9.49 needed? Aren't the memory barriers inherent in the locking primitives on lines 20 and 30 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 9.66: Couldn't the update-side batching optimization described in Section 9.3.5.6 be applied to the implementation shown in Figure 9.49? ■

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 9.49 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 9.67: Is the possibility of readers being preempted in lines 3-4 of Figure 9.49 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? ■

9.3.5.8 Nestable RCU Based on Free-Running Counter

Figure 9.51 (`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 9.50. This is a generalization of the scheme in Section 9.3.5.7, which can be thought of as having a single low-order bit re-

```

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 9.50: Data for Nestable RCU Using a Free-Running Counter

```

1 static void rcu_read_lock(void)
2 {
3     long tmp;
4     long *rrgp;
5
6     rrgp = &__get_thread_var(rcu_reader_gp);
7     tmp = *rrgp;
8     if ((tmp & RCU_GP_CTR_NEST_MASK) == 0)
9         tmp = ACCESS_ONCE(rcu_gp_ctr);
10    tmp++;
11    *rrgp = tmp;
12    smp_mb();
13 }
14
15 static void rcu_read_unlock(void)
16 {
17     long tmp;
18
19     smp_mb();
20     __get_thread_var(rcu_reader_gp)--;
21 }
22
23 void synchronize_rcu(void)
24 {
25     int t;
26
27     smp_mb();
28     spin_lock(&rcu_gp_lock);
29     ACCESS_ONCE(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     }
38     spin_unlock(&rcu_gp_lock);
39     smp_mb();
40 }
41

```

Figure 9.51: Nestable RCU Using a Free-Running Counter

served 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, despite their `rcu_read_lock()` differences, the implementation of `rcu_read_unlock()` is broadly similar to that shown in Section 9.3.5.7. 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`

```

1 DEFINE_SPINLOCK(rcu_gp_lock);
2 long rcu_gp_ctr = 0;
3 DEFINE_PER_THREAD(long, rcu_reader_qs_gp);

```

Figure 9.52: Data for Quiescent-State-Based RCU

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 9.3.5.7. There are two differences. The first is that lines 29 and 30 adds RCU_GPI_CTR_BOTTOM_BIT to the global rcu_gp_ctr instead of adding the constant “2”, and the second is that the comparison on line 33 has been abstracted out to a separate function, where it checks the bit indicated by RCU_GPI_CTR_BOTTOM_BIT instead of unconditionally checking the low-order bit.

This approach achieves read-side performance almost equal to that shown in Section 9.3.5.7, 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 9.68: 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 that of Section 9.3.5.7, 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 9.69: Given the algorithm shown in Figure 9.51, how could you double the time required to overflow the global rcu_gp_ctr? ■

Quick Quiz 9.70: Again, given the algorithm shown in Figure 9.51, is counter overflow fatal? Why or why not? If it is fatal, what can be done to fix it? ■

```

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) =
13         ACCESS_ONCE(rcu_gp_ctr) + 1;
14     smp_mb();
15 }
16
17 static void rcu_thread_offline(void)
18 {
19     smp_mb();
20     __get_thread_var(rcu_reader_qs_gp) =
21         ACCESS_ONCE(rcu_gp_ctr);
22     smp_mb();
23 }
24
25 static void rcu_thread_online(void)
26 {
27     rcu_quiescent_state();
28 }

```

Figure 9.53: Quiescent-State-Based RCU Read Side

9.3.5.9 RCU Based on Quiescent States

Figure 9.53 (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 9.52. 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. Each of these quiescent states contains a call to rcu_quiescent_state(), which is shown from lines 9-15 in the figure. Threads entering extended quiescent states (for example, when blocking) may instead call rcu_thread_offline() (lines 17-23) when entering an extended quiescent state and then call rcu_thread_online() (lines 25-28) when leaving it. As such, rcu_thread_online() is analogous to rcu_read_lock() and rcu_thread_offline() is analogous to rcu_read_unlock(). In addition, rcu_quiescent_state() can be thought of as a rcu_thread_online() immediately followed by

a `rcu_thread_offline()`.¹² It is illegal to invoke `rcu_quiescent_state()`, `rcu_thread_offline()`, or `rcu_thread_online()` from 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 (including possible RCU read-side critical sections) 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, which prevents subsequent code (including a possible RCU read-side critical section) from being re-ordered with the lines 12-13.

Quick Quiz 9.71: Doesn't the additional memory barrier shown on line 14 of Figure 9.53, 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_online()` when starting to use RCU and `rcu_thread_offline()` 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 9.72: Why are the two memory barriers on lines 19 and 22 of Figure 9.53 needed? ■

The `rcu_thread_online()` function simply invokes `rcu_quiescent_state()`, thus marking the end of the extended quiescent state.

¹² Although the code in the figure is consistent with `rcu_quiescent_state()` being the same as `rcu_thread_online()` immediately followed by `rcu_thread_offline()`,

```

1 void synchronize_rcu(void)
2 {
3     int t;
4
5     smp_mb();
6     spin_lock(&rcu_gp_lock);
7     rCU_gp_ctr += 2;
8     smp_mb();
9     for_each_thread(t) {
10         while (rcu_gp_ongoing(t) &&
11                ((per_thread(rcu_reader_qs_gp, t) -
12                  rCU_gp_ctr) < 0)) {
13             poll(NULL, 0, 10);
14         }
15     }
16     spin_unlock(&rcu_gp_lock);
17     smp_mb();
18 }
```

Figure 9.54: RCU Update Side Using Quiescent States

Figure 9.54 (`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 9.73: To be sure, the clock frequencies of Power systems in 2008 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? ■

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 9.74: 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 9.53 and 9.54? ■

Quick Quiz 9.75: 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? ■

this relationship is obscured by performance optimizations.

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.

9.3.5.10 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 descriptions of production-quality RCU implementations [DMS⁺12, McK07a, McK08a, McK09a].

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. In particular, irrevocable operations such as I/O should be permitted.
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 9.76: 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? ■

9.3.6 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 9.77: The statistical-counter implementation shown in Figure 5.9 (`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 9.78: Section 5.5 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.)

■

9.4 Which to Choose?

Table 9.7 provides some rough rules of thumb that can help you choose among the four deferred-processing techniques presented in this chapter.

As shown in the “Existence Guarantee” column, if you need existence guarantees for linked data elements, you must use reference counting, hazard pointers, or RCU. Sequence locks do not provide existence guarantees, instead providing detection of updates, retrying any read-side critical sections that do encounter an update.

Of course, as shown in the “Updates and Readers Progress Concurrently” column, this detection of updates implies that sequence locking does not permit updates and readers to make forward progress concurrently. After all, preventing such forward progress is the whole point of using sequence locking in the first place! This situation points the way to using sequence locking in conjunction with reference counting, hazard pointers, or RCU in order to provide both existence guarantees and update detection. In fact, the Linux kernel combines RCU and sequence locking in this manner during pathname lookup.

The “Read-Side Overhead” column gives a rough sense of the read-side overhead of these techniques. The overhead of reference counting can vary widely. At the low end, a simple non-atomic increment suffices, at least in the case where the reference is acquired under the protection of a lock that must be acquired for other reasons. At the high end, a fully ordered atomic operation is required. Reference counting incurs this overhead on each and every data element traversed. Hazard pointers incur the overhead of a memory barrier for each data element traversed, and sequence locks incur the overhead of a pair of memory barriers for each attempt to execute the critical section. The overhead of RCU implementations vary from nothing to that of a pair of memory barriers for each read-side critical section, thus providing RCU with the best performance, particularly for read-side critical sections that traverse many data elements.

The “Bulk Reference” column indicates that only RCU is capable of acquiring multiple references with constant overhead. The entry for sequence locks is “N/A” because, again, sequence locks detect updates rather than acquiring references.

Quick Quiz 9.79: But can’t both reference counting and hazard pointers also acquire a reference to multiple data elements with constant overhead? A single reference count can cover multiple data elements, right? ■

The “Low Memory Footprint” column indicates which techniques enjoy low memory footprint. This column ends up being the mirror image of the “Bulk Reference” column: The ability to acquire references on large numbers of data elements implies that all these data elements must persist, which in turn implies a large memory footprint in some cases. For example, one thread might delete a large number of data elements while another thread concurrently executes a long RCU read-side critical section. Because the read-side critical section could potentially retain a reference to any of the newly deleted data elements, all those data elements must be retained for the full duration of that critical section. In contrast, reference counting and hazard pointers would retain only those specific data elements actually referenced by concurrent readers.

However, this low-memory-footprint advantage comes at a price, as shown in the “Unconditional Acquisition” column. To see this, imagine a large linked data structure in which a reference-counting or hazard-pointer reader (call it Thread A) holds a reference to an isolated data element in the middle of that structure. Consider the following sequence of events:

1. Thread B removes the data element referenced by Thread A. Because of this reference, the data element cannot yet be freed.
2. Thread B removes all the data elements adjacent to the one referenced by Thread A. Because there are no references held for these data elements, they are all immediately freed. Because Thread A’s data element has already been removed, its outgoing pointers are not updated.
3. All of Thread A’s data element’s outgoing pointers now reference the freelist, and therefore cannot safely be traversed.
4. The reference-counting or hazard-pointer implementation therefore has no choice but to fail any attempt by Thread A to acquire a reference via any of the pointers emanating from its data element.

In short, any deferred-processing technique that offers precise tracking of references must also be prepared to fail attempts to acquire references. Therefore, RCU’s memory-footprint disadvantage implies an ease-of-use advantage,

	Existence Guarantee	Updates and Readers Progress Concurrently	Read-Side Overhead	Bulk Reference	Low Memory Footprint	Unconditional Acquisition	Non-Blocking Updates
Reference Counting	Y	Y	$++ \rightarrow \text{atomic } \dagger$		Y		?
Hazard Pointers	Y	Y	MB \dagger		Y		Y
Sequence Locks			2 MB \ddagger	N/A	N/A		
RCU	Y	Y	$0 \rightarrow 2 \text{ MB}$	Y		Y	?

\dagger Incurred on each element traversed on each retry

\ddagger Incurred on each retry

atomic: Atomic operation

MB: Memory barrier

Table 9.7: Which Deferred Technique to Choose?

namely that RCU readers need not deal with acquisition failure.

This tension between memory footprint, precise tracking, and acquisition failures is sometimes resolved within the Linux kernel by combining use of RCU and reference counters. RCU is used for short-lived references, which means that RCU read-side critical sections can be short. These short RCU read-side critical sections in turn mean that the corresponding RCU grace periods can also be short, limiting the memory footprint. For the few data elements that need longer-lived references, reference counting is used. This means that the complexity of reference-acquisition failure only needs to be dealt with for those few data elements: The bulk of the reference acquisitions are unconditional, courtesy of RCU.

Finally, the “Non-Blocking Updates” column shows that hazard pointers can provide non-blocking updates [Mic04, HLM02]. Reference counting might or might not, depending on the implementation. However, sequence locking cannot provide non-blocking updates, courtesy of its update-side lock. RCU updaters must wait on readers, which also rules out fully non-blocking updates. However, there are situations in which the only blocking operation is a wait to free memory, which results in a situation that, for many purposes, is as good as non-blocking [DMS⁺12].

As more experience is gained using these techniques, both separately and in combination, the rules of thumb

laid out in this section will need to be refined. However, this section does reflect the current state of the art.

9.5 What About Updates?

The deferred-processing techniques called out in this chapter are most directly applicable to read-mostly situations, which begs the question “But what about updates?” After all, increasing the performance and scalability of readers is all well and good, but it is only natural to also want great performance and scalability for writers.

We have already seen one situation featuring high performance and scalability for writers, namely the counting algorithms surveyed in Chapter 5. These algorithms featured partially partitioned data structures so that updates can operate locally, while the more-expensive reads must sum across the entire data structure. Silas Boyd-Wickizer has generalized this notion to produce OpLog, which he has applied to Linux-kernel pathname lookup, VM reverse mappings, and the `stat()` system call [BW14].

Another approach, called “Disruptor,” is designed for applications that process high-volume streams of input data. The approach is to rely on single-producer-single-consumer FIFO queues, minimizing the need for synchronization [Sut13]. For Java applications, Disruptor also has the virtue of minimizing use of the garbage collector.

And of course, where feasible, fully partitioned or “sharded” systems provide excellent performance and scalability, as noted in Chapter 6.

The next chapter will look at updates in the context of several types of data structures.

Chapter 10

Data Structures

Efficient access to data is critically important, so that discussions of algorithms include time complexity of the related data structures [CLRS01]. However, for parallel programs, measures of time complexity must also include concurrency effects. These effects can be overwhelmingly large, as shown in Chapter 3, which means that concurrent data structure designs must focus as much on concurrency as they do on sequential time complexity.

Section 10.1 presents a motivating application that will be used to evaluate the data structures presented in this chapter.

As discussed in Chapter 6, an excellent way to achieve high scalability is partitioning. This points the way to partitionable data structures, a topic taken up by Section 10.2. Chapter 9 described how deferring some actions can greatly improve both performance and scalability. Section 9.3 in particular showed how to tap the awesome power of procrastination in pursuit of performance and scalability, a topic taken up by Section 10.3.

Not all data structures are partitionable. Section 10.4 looks at a mildly non-partitionable example data structure. This section shows how to split it into read-mostly and partitionable portions, enabling a fast and scalable implementation.

Because this chapter cannot delve into the details of every concurrent data structure that has ever been used Section 10.5 provides a brief survey of the most common and important ones. Although the best performance and scalability results design rather than after-the-fact micro-optimization, it is nevertheless the case that micro-optimization has an important place in achieving the absolute best possible performance and scalability. This topic is therefore taken up in Section 10.6.

Finally, Section 10.7 presents a summary of this chapter.

10.1 Motivating Application

We will use the Schrödinger's Zoo application to evaluate performance [McK13]. Schrödinger has a zoo containing a large number of animals, and he would like to track them using an in-memory database with each animal in the zoo represented by a data item in this database. Each animal has a unique name that is used as a key, with a variety of data tracked for each animal.

Births, captures, and purchases result in insertions, while deaths, releases, and sales result in deletions. Because Schrödinger's zoo contains a large quantity of short-lived animals, including mice and insects, the database must be able to support a high update rate.

Those interested in Schrödinger's animals can query them, however, Schrödinger has noted extremely high rates of queries for his cat, so much so that he suspects that his mice might be using the database to check up on their nemesis. This means that Schrödinger's application must be able to support a high rate of queries to a single data element.

Please keep this application in mind as various data structures are presented.

10.2 Partitionable Data Structures

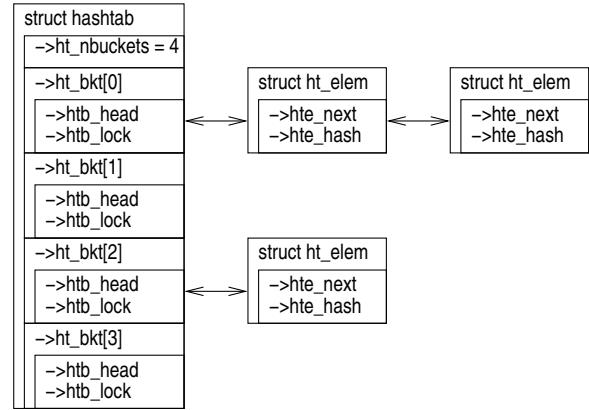
There are a huge number of data structures in use today, so much so that there are multiple textbooks covering them. This small section focuses on a single data structure, namely the hash table. This focused approach allows a much deeper investigation of how concurrency interacts with data structures, and also focuses on a data structure that is heavily used in practice. Section 10.2.1 overviews of the design, and Section 10.2.2 presents the implementation. Finally, Section 10.2.3 discusses the resulting performance and scalability.

```

1 struct ht_elem {
2     struct cds_list_head hte_next;
3     unsigned long hte_hash;
4 };
5
6 struct ht_bucket {
7     struct cds_list_head htb_head;
8     spinlock_t htb_lock;
9 };
10
11 struct hashtab {
12     unsigned long ht_nbuckets;
13     struct ht_bucket ht_bkt[0];
14 };

```

Figure 10.1: Hash-Table Data Structures



10.2.1 Hash-Table Design

Chapter 6 emphasized the need to apply partitioning in order to attain respectable performance and scalability, so partitionability must be a first-class criterion when selecting data structures. This criterion is well satisfied by that workhorse of parallelism, the hash table. Hash tables are conceptually simple, consisting of an array of *hash buckets*. A *hash function* maps from a given element’s *key* to the hash bucket that this element will be stored in. Each hash bucket therefore heads up a linked list of elements, called a *hash chain*. When properly configured, these hash chains will be quite short, permitting a hash table to access the element with a given key extremely efficiently.

Quick Quiz 10.1: But there are many types of hash tables, of which the chained hash tables described here are but one type. Why the focus on chained hash tables?

In addition, each bucket can be given its own lock, so that elements in different buckets of the hash table may be added, deleted, and looked up completely independently. A large hash table containing a large number of elements therefore offers excellent scalability.

10.2.2 Hash-Table Implementation

Figure 10.1 (`hash_bkt.c`) shows a set of data structures used in a simple fixed-sized hash table using chaining and per-hash-bucket locking, and Figure 10.2 diagrams how they fit together. The `hashtab` structure (lines 11-14 in Figure 10.1) contains four `ht_bucket` structures (lines 6-9 in Figure 10.1), with the `->ht_nbuckets` field controlling the number of buckets. Each such bucket contains a list header `->htb_head` and a lock `->htb_lock`. The list headers chain `ht_elem`

Figure 10.2: Hash-Table Data-Structure Diagram

```

1 #define HASH2BKT(htp, h) \
2     (&(htp)->ht_bkt[h % (htp)->ht_nbuckets])
3
4 static void hashtab_lock(struct hashtab *htp,
5                          unsigned long hash)
6 {
7     spin_lock(&HASH2BKT(htp, hash)->htb_lock);
8 }
9
10 static void hashtab_unlock(struct hashtab *htp,
11                           unsigned long hash)
12 {
13     spin_unlock(&HASH2BKT(htp, hash)->htb_lock);
14 }

```

Figure 10.3: Hash-Table Mapping and Locking

structures (lines 1-4 in Figure 10.1) through their `->hte_next` fields, and each `ht_elem` structure also caches the corresponding element’s hash value in the `->hte_hash` field. The `ht_elem` structure would be included in the larger structure being placed in the hash table, and this larger structure might contain a complex key.

The diagram shown in Figure 10.2 has bucket 0 with two elements and bucket 2 with one.

Figure 10.3 shows mapping and locking functions. Lines 1 and 2 show the macro `HASH2BKT()`, which maps from a hash value to the corresponding `ht_bucket` structure. This macro uses a simple modulus: if more aggressive hashing is required, the caller needs to implement it when mapping from key to hash value. The remaining two functions acquire and release the `->htb_lock` corresponding to the specified hash value.

Figure 10.4 shows `hashtab_lookup()`, which returns a pointer to the element with the specified hash and key if it exists, or `NULL` otherwise. This function takes

```

1 struct ht_elem *
2 hashtab_lookup(struct hashtab *htp,
3                 unsigned long hash,
4                 void *key,
5                 int (*cmp)(struct ht_elem *htep,
6                           void *key))
7 {
8     struct ht_bucket *htb;
9     struct ht_elem *htep;
10
11    htb = HASH2BKT(htp, hash);
12    cds_list_for_each_entry(htep,
13                            &htb->htb_head,
14                            hte_next) {
15        if (htep->hte_hash != hash)
16            continue;
17        if (cmp(htep, key))
18            return htep;
19    }
20    return NULL;
21 }

```

Figure 10.4: Hash-Table Lookup

```

1 void
2 hashtab_add(struct hashtab *htp,
3             unsigned long hash,
4             struct ht_elem *htep)
5 {
6     htep->hte_hash = hash;
7     cds_list_add(&htep->hte_next,
8                  &HASH2BKT(htp, hash)->htb_head);
9 }
10
11 void hashtab_del(struct ht_elem *htep)
12 {
13     cds_list_del_init(&htep->hte_next);
14 }

```

Figure 10.5: Hash-Table Modification

both a hash value and a pointer to the key because this allows users of this function to use arbitrary keys and arbitrary hash functions, with the key-comparison function passed in via `cmp()`, in a manner similar to `qsort()`. Line 11 maps from the hash value to a pointer to the corresponding hash bucket. Each pass through the loop spanning line 12-19 examines one element of the bucket's hash chain. Line 15 checks to see if the hash values match, and if not, line 16 proceeds to the next element. Line 17 checks to see if the actual key matches, and if so, line 18 returns a pointer to the matching element. If no element matches, line 20 returns NULL.

Quick Quiz 10.2: But isn't the double comparison on lines 15-18 in Figure 10.4 inefficient in the case where the key fits into an `unsigned long`? ■

Figure 10.5 shows the `hashtab_add()` and `hashtab_del()` functions that add and delete elements from the hash table, respectively.

```

1 struct hashtab *
2 hashtab_alloc(unsigned long nbuckets)
3 {
4     struct hashtab *htp;
5     int i;
6
7     htp = malloc(sizeof(*htp) +
8                  nbuckets *
9                  sizeof(struct ht_bucket));
10    if (htp == NULL)
11        return NULL;
12    htp->ht_nbuckets = nbuckets;
13    for (i = 0; i < nbuckets; i++) {
14        CDS_INIT_LIST_HEAD(&htp->ht_bkt[i].htb_head);
15        spin_lock_init(&htp->ht_bkt[i].htb_lock);
16    }
17    return htp;
18 }
19
20 void hashtab_free(struct hashtab *htp)
21 {
22     free(htp);
23 }

```

Figure 10.6: Hash-Table Allocation and Free

The `hashtab_add()` function simply sets the element's hash value on line 6, then adds it to the corresponding bucket on lines 7 and 8. The `hashtab_del()` function simply removes the specified element from whatever hash chain it is on, courtesy of the doubly linked nature of the hash-chain lists. Before calling either of these two functions, the caller is required to ensure that no other thread is accessing or modifying this same bucket, for example, by invoking `hashtab_lock()` beforehand.

Figure 10.6 shows `hashtab_alloc()` and `hashtab_free()`, which do hash-table allocation and freeing, respectively. Allocation begins on lines 7-9 with allocation of the underlying memory. If line 10 detects that memory has been exhausted, line 11 returns NULL to the caller. Otherwise, line 12 initializes the number of buckets, and the loop spanning lines 13-16 initializes the buckets themselves, including the chain list header on line 14 and the lock on line 15. Finally, line 17 returns a pointer to the newly allocated hash table. The `hashtab_free()` function on lines 20-23 is straightforward.

10.2.3 Hash-Table Performance

The performance results for an eight-CPU 2GHz Intel® Xeon® system using a bucket-locked hash table with 1024 buckets are shown in Figure 10.7. The performance does scale nearly linearly, but is not much more than half of the ideal performance level, even at only eight CPUs. Part of this shortfall is due to the fact that the lock acquisi-

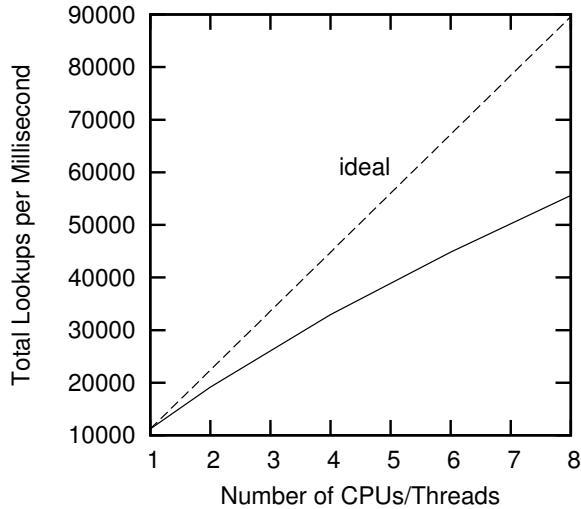


Figure 10.7: Read-Only Hash-Table Performance For Schrödinger's Zoo

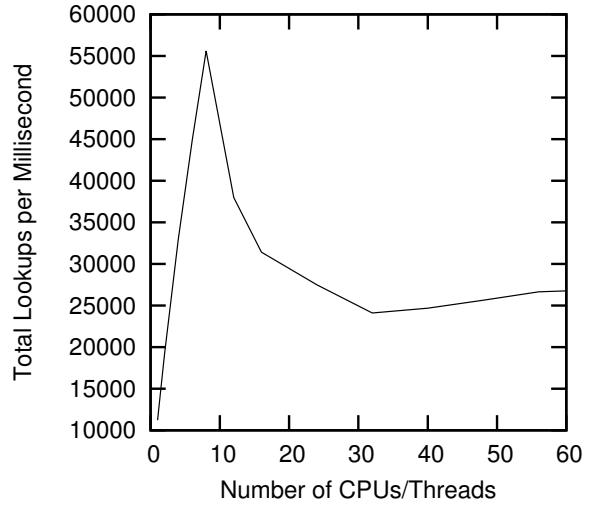


Figure 10.8: Read-Only Hash-Table Performance For Schrödinger's Zoo, 60 CPUs

tions and releases incur no cache misses on a single CPU, but do incur misses on two or more CPUs.

And things only get worse with larger number of CPUs, as can be seen in Figure 10.8. We do not need an additional line to show ideal performance: The performance for nine CPUs and beyond is worse than abysmal. This clearly underscores the dangers of extrapolating performance from a modest number of CPUs.

Of course, one possible reason for the collapse in performance might be that more hash buckets are needed. After all, we did not pad each hash bucket to a full cache line, so there are a number of hash buckets per cache line. It is possible that the resulting cache-thrashing comes into play at nine CPUs. This is of course easy to test by increasing the number of hash buckets.

Quick Quiz 10.3: Instead of simply increasing the number of hash buckets, wouldn't it be better to cache-align the existing hash buckets? ■

However, as can be seen in Figure 10.9, although increasing the number of buckets does increase performance somewhat, scalability is still abysmal. In particular, we still see a sharp dropoff at nine CPUs and beyond. Furthermore, going from 8192 buckets to 16,384 buckets produced almost no increase in performance. Clearly something else is going on.

The problem is that this is a multi-socket system, with CPUs 0-7 and 32-39 mapped to the first socket as shown in Table 10.1. Test runs confined to the first eight CPUs there-

fore perform quite well, but tests that involve socket 0's CPUs 0-7 as well as socket 1's CPU 8 incur the overhead of passing data across socket boundaries. This can severely degrade performance, as was discussed in Section 3.2.1. In short, large multi-socket systems require good locality of reference in addition to full partitioning.

Quick Quiz 10.4: Given the negative scalability of the Schrödinger's Zoo application across sockets, why not just run multiple copies of the application, with each copy having a subset of the animals and confined to run on a single socket? ■

One key property of the Schrödinger's-zoo runs discussed thus far is that they are all read-only. This makes the performance degradation due to lock-acquisition-induced cache misses all the more painful. Even though we are not updating the underlying hash table itself, we are still paying the price for writing to memory. Of course, if the hash table was never going to be updated, we could dispense entirely with mutual exclusion. This approach is quite straightforward and is left as an exercise for the reader. But even with the occasional update, avoiding writes avoids cache misses, and allows the read-mostly data to be replicated across all the caches, which in turn promotes locality of reference.

The next section therefore examines optimizations that can be carried out in read-mostly cases where updates are rare, but could happen at any time.

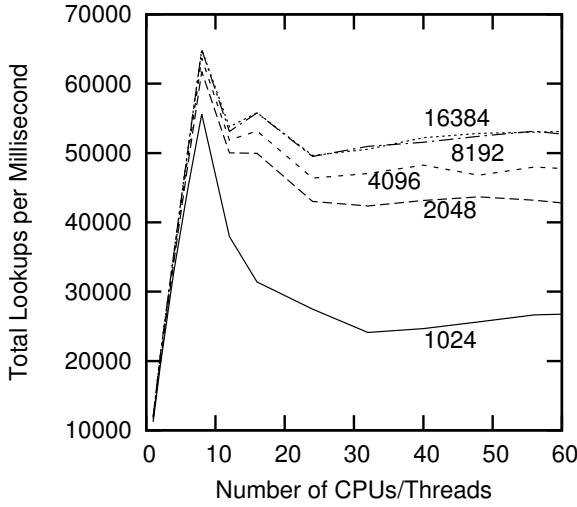


Figure 10.9: Read-Only Hash-Table Performance For Schrödinger’s Zoo, Varying Buckets

Socket	Core							
	0	1	2	3	4	5	6	7
0	32	33	34	35	36	37	38	39
1	8	9	10	11	12	13	14	15
	40	41	42	43	44	45	46	47
2	16	17	18	19	20	21	22	23
	48	49	50	51	52	53	54	55
3	24	25	26	27	28	29	30	31
	56	47	58	59	60	61	62	63

Table 10.1: NUMA Topology of System Under Test

10.3 Read-Mostly Data Structures

Although partitioned data structures can offer excellent scalability, NUMA effects can result in severe degradations of both performance and scalability. In addition, the need for readers to exclude writers can degrade performance in read-mostly situations. However, we can achieve both performance and scalability by using RCU, which was introduced in Section 9.3. Similar results can be achieved using hazard pointers (`hazptr.c`) [Mic04], which will be included in the performance results shown in this section [McK13].

```

1 static void hashtab_lock_lookup(struct hashtab *htp,
2                                unsigned long hash)
3 {
4     rcu_read_lock();
5 }
6
7 static void hashtab_unlock_lookup(struct hashtab *htp,
8                                  unsigned long hash)
9 {
10    rcu_read_unlock();
11 }

```

Figure 10.10: RCU-Protected Hash-Table Read-Side Concurrency Control

```

1 struct ht_elem
2 *hashtab_lookup(struct hashtab *htp,
3                  unsigned long hash,
4                  void *key,
5                  int (*cmp)(struct ht_elem *htep,
6                             void *key))
7 {
8     struct ht_bucket *htb;
9     struct ht_elem *htep;
10
11    htb = HASH2BKT(htp, hash);
12    cds_list_for_each_entry_rcu(htep,
13                               &htb->htb_head,
14                               hte_next) {
15        if (htep->hte_hash != hash)
16            continue;
17        if (cmp(htep, key))
18            return htep;
19    }
20    return NULL;
21 }

```

Figure 10.11: RCU-Protected Hash-Table Lookup

10.3.1 RCU-Protected Hash Table Implementation

For an RCU-protected hash table with per-bucket locking, updaters use locking exactly as described in Section 10.2, but readers use RCU. The data structures remain as shown in Figure 10.1, and the `HASH2BKT()`, `hashtab_lock()`, and `hashtab_unlock()` functions remain as shown in Figure 10.3. However, readers use the lighter-weight concurrency-control embodied by `hashtab_lock_lookup()` and `hashtab_unlock_lookup()` shown in Figure 10.10.

Figure 10.11 shows `hashtab_lookup()` for the RCU-protected per-bucket-locked hash table. This is identical to that in Figure 10.4 except that `cds_list_for_each_entry()` is replaced by `cds_list_for_each_entry_rcu()`. Both of these primitives sequence down the hash chain referenced by `htb->htb_head` but `cds_list_for_each_entry_rcu()` also correctly enforces memory ordering in case of con-

```

1 void
2 hashtab_add(struct hashtab *htp,
3             unsigned long hash,
4             struct ht_elem *htep)
5 {
6     htep->hte_hash = hash;
7     cds_list_add_rcu(&htep->hte_next,
8                      &HASH2BKT(htp, hash)->htb_head);
9 }
10
11 void hashtab_del(struct ht_elem *htep)
12 {
13     cds_list_del_rcu(&htep->hte_next);
14 }

```

Figure 10.12: RCU-Protected Hash-Table Modification

current insertion. This is an important difference between these two hash-table implementations: Unlike the pure per-bucket-locked implementation, the RCU protected implementation allows lookups to run concurrently with insertions and deletions, and RCU-aware primitives like `cds_list_for_each_entry_rcu()` are required to correctly handle this added concurrency. Note also that `hashtab_lookup()`'s caller must be within an RCU read-side critical section, for example, the caller must invoke `hashtab_lock_lookup()` before invoking `hashtab_lookup()` (and of course invoke `hashtab_unlock_lookup()` some time afterwards).

Quick Quiz 10.5: But if elements in a hash table can be deleted concurrently with lookups, doesn't that mean that a lookup could return a reference to a data element that was deleted immediately after it was looked up? ■

Figure 10.12 shows `hashtab_add()` and `hashtab_del()`, both of which are quite similar to their counterparts in the non-RCU hash table shown in Figure 10.5. The `hashtab_add()` function uses `cds_list_add_rcu()` instead of `cds_list_add()` in order to ensure proper ordering when an element is added to the hash table at the same time that it is being looked up. The `hashtab_del()` function uses `cds_list_del_rcu()` instead of `cds_list_del_init()` to allow for the case where an element is looked up just before it is deleted. Unlike `cds_list_del_init()`, `cds_list_del_rcu()` leaves the forward pointer intact, so that `hashtab_lookup()` can traverse to the newly deleted element's successor.

Of course, after invoking `hashtab_del()`, the caller must wait for an RCU grace period (e.g., by invoking `synchronize_rcu()`) before freeing or otherwise reusing the memory for the newly deleted element.

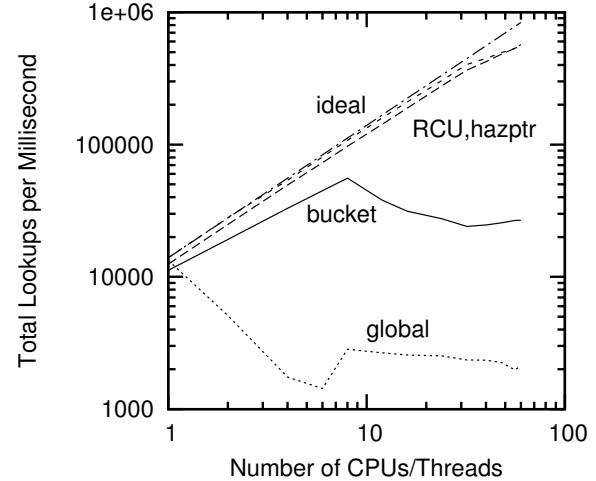


Figure 10.13: Read-Only RCU-Protected Hash-Table Performance For Schrödinger's Zoo

10.3.2 RCU-Protected Hash Table Performance

Figure 10.13 shows the read-only performance of RCU-protected and hazard-pointer-protected hash tables against the previous section's per-bucket-locked implementation. As you can see, both RCU and hazard pointers achieve near-ideal performance and scalability despite the larger numbers of threads and the NUMA effects. Results from a globally locked implementation are also shown, and as expected the results are even worse than those of the per-bucket-locked implementation. RCU does slightly better than hazard pointers, but the difference is not readily visible in this log-scale plot.

Figure 10.14 shows the same data on a linear scale. This drops the global-locking trace into the x-axis, but allows the relative performance of RCU and hazard pointers to be more readily discerned. Both show a change in slope at 32 CPUs, and this is due to hardware multithreading. At 32 and fewer CPUs, each thread has a core to itself. In this regime, RCU does better than does hazard pointers because hazard pointers's read-side memory barriers result in dead time within the core. In short, RCU is better able to utilize a core from a single hardware thread than is hazard pointers.

This situation changes above 32 CPUs. Because RCU is using more than half of each core's resources from a single hardware thread, RCU gains relatively little benefit from the second hardware thread in each core. The slope

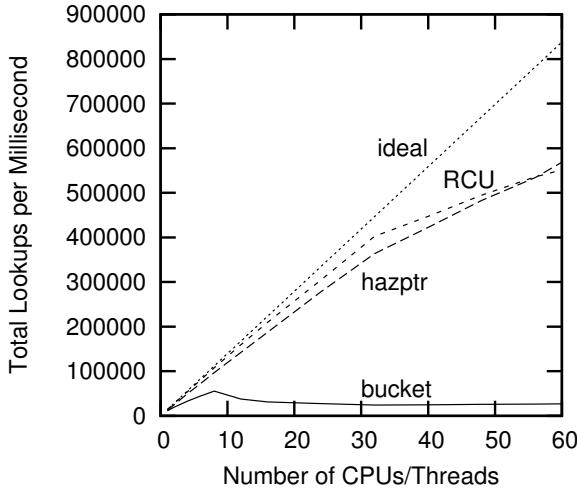


Figure 10.14: Read-Only RCU-Protected Hash-Table Performance For Schrödinger's Zoo, Linear Scale

of hazard pointers's trace also decreases at 32 CPUs, but less dramatically, because the second hardware thread is able to fill in the time that the first hardware thread is stalled due to memory-barrier latency. As we will see in later sections, hazard pointers's second-hardware-thread advantage depends on the workload.

As noted earlier, Schrödinger is surprised by the popularity of his cat [Sch35], but recognizes the need to reflect this popularity in his design. Figure 10.15 shows the results of 60-CPU runs, varying the number of CPUs that are doing nothing but looking up the cat. Both RCU and hazard pointers respond well to this challenge, but bucket locking scales negatively, eventually performing even worse than global locking. This should not be a surprise because if all CPUs are doing nothing but looking up the cat, the lock corresponding to the cat's bucket is for all intents and purposes a global lock.

This cat-only benchmark illustrates one potential problem with fully partitioned sharding approaches. Only the CPUs associated with the cat's partition is able to access the cat, limiting the cat-only throughput. Of course, a great many applications have good load-spreading properties, and for these applications sharding works quite well. However, sharding does not handle “hot spots” very well, with the hot spot exemplified by Schrödinger's cat being but one case in point.

Of course, if we were only ever going to read the data, we would not need any concurrency control to begin with. Figure 10.16 therefore shows the effect of updates. At the

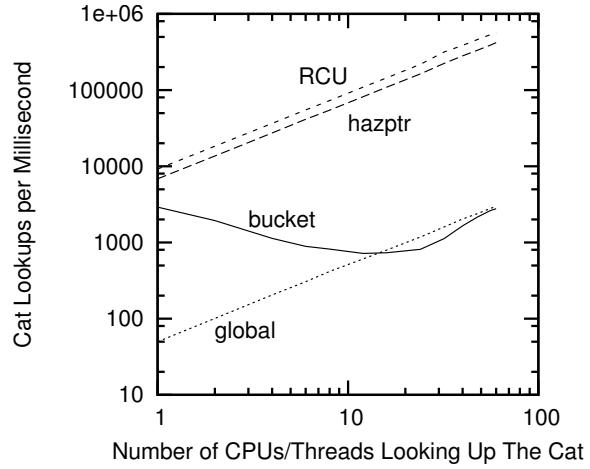


Figure 10.15: Read-Side Cat-Only RCU-Protected Hash-Table Performance For Schrödinger's Zoo at 60 CPUs

extreme left-hand side of this graph, all 60 CPUs are doing lookups, while to the right all 60 CPUs are doing updates. For all four implementations, the number of lookups per millisecond decreases as the number of updating CPUs increases, of course reaching zero lookups per millisecond when all 60 CPUs are updating. RCU does well relative to hazard pointers due to the fact that hazard pointers's read-side memory barriers incur greater overhead in the presence of updates. It therefore seems likely that modern hardware heavily optimizes memory-barrier execution, greatly reducing memory-barrier overhead in the read-only case.

Where Figure 10.16 showed the effect of increasing update rates on lookups, Figure 10.17 shows the effect of increasing update rates on the updates themselves. Hazard pointers and RCU start off with a significant advantage because, unlike bucket locking, readers do not exclude updaters. However, as the number of updating CPUs increases, update-side overhead starts to make its presence known, first for RCU and then for hazard pointers. Of course, all three of these implementations fare much better than does global locking.

Of course, it is quite possible that the differences in lookup performance are affected by the differences in update rates. One way to check this is to artificially throttle the update rates of per-bucket locking and hazard pointers to match that of RCU. Doing so does not significantly improve the lookup performance of per-bucket locking, nor does it close the gap between hazard pointers and RCU.

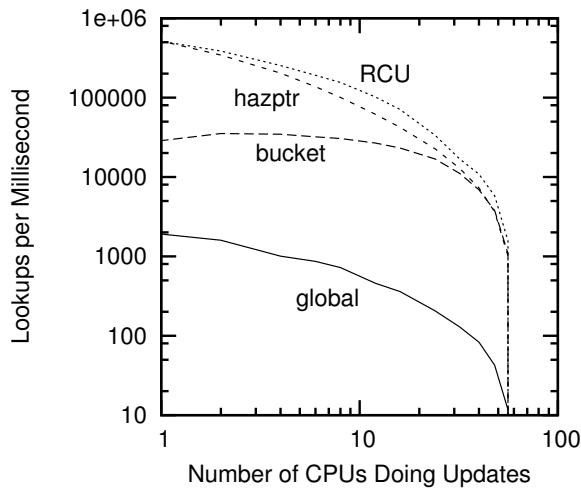


Figure 10.16: Read-Side RCU-Protected Hash-Table Performance For Schrödinger’s Zoo at 60 CPUs

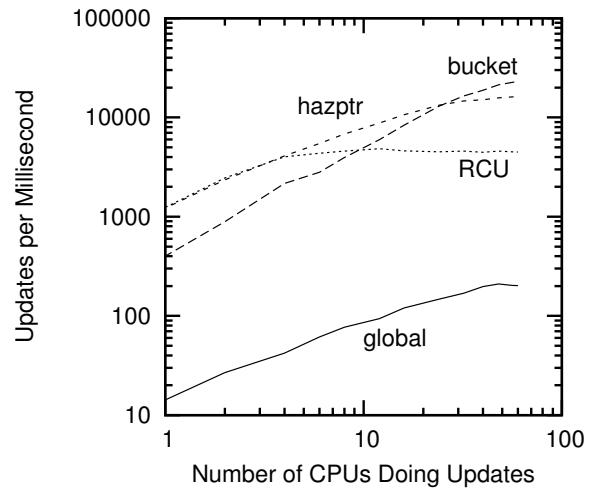


Figure 10.17: Update-Side RCU-Protected Hash-Table Performance For Schrödinger’s Zoo at 60 CPUs

However, removing hazard pointers’s read-side memory barriers (thus resulting in an unsafe implementation of hazard pointers) does nearly close the gap between hazard pointers and RCU. Although this unsafe hazard-pointer implementation will usually be reliable enough for benchmarking purposes, it is absolutely not recommended for production use.

Quick Quiz 10.6: The dangers of extrapolating from eight CPUs to 60 CPUs was made quite clear in Section 10.2.3. But why should extrapolating up from 60 CPUs be any safer? ■

10.3.3 RCU-Protected Hash Table Discussion

One consequence of the RCU and hazard-pointer implementations is that a pair of concurrent readers might disagree on the state of the cat. For example, one of the readers might have fetched the pointer to the cat’s data structure just before it was removed, while another reader might have fetched this same pointer just afterwards. The first reader would then believe that the cat was alive, while the second reader would believe that the cat was dead.

Of course, this situation is completely fitting for Schrödinger’s cat, but it turns out that it is quite reasonable for normal non-quantum cats as well.

The reason for this is that it is impossible to determine exactly when an animal is born or dies.

To see this, let’s suppose that we detect a cat’s death by

heartbeat. This raise the question of exactly how long we should wait after the last heartbeat before declaring death. It is clearly ridiculous to wait only one millisecond, because then a healthy living cat would have to be declared dead—and then resurrected—more than once every second. It is equally ridiculous to wait a full month, because by that time the poor cat’s death would have made itself very clearly known via olfactory means.



Figure 10.18: Even Veterinarians Disagree!

Because an animal’s heart can stop for some seconds and then start up again, there is a tradeoff between timely recognition of death and probability of false alarms. It is

quite possible that a pair of veterinarians might disagree on the time to wait between the last heartbeat and the declaration of death. For example, one veterinarian might declare death thirty seconds after the last heartbeat, while another might insist on waiting a full minute. In this case, the two veterinarians would disagree on the state of the cat for the second period of thirty seconds following the last heartbeat, as fancifully depicted in Figure 10.18.

Of course, Heisenberg taught us to live with this sort of uncertainty [Hei27], which is a good thing because computing hardware and software acts similarly. For example, how do you know that a piece of computing hardware has failed? Often because it does not respond in a timely fashion. Just like the cat's heartbeat, this results in a window of uncertainty as to whether or not the hardware has failed.

Furthermore, most computing systems are intended to interact with the outside world. Consistency with the outside world is therefore of paramount importance. However, as we saw in Figure 9.26 on page 132, increased internal consistency can come at the expense of external consistency. Techniques such as RCU and hazard pointers give up some degree of internal consistency to attain improved external consistency.

In short, internal consistency is not a natural part of all problem domains, and often incurs great expense in terms of performance, scalability, external consistency, or all of the above.

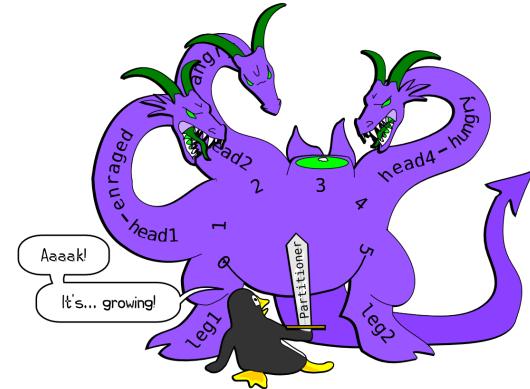


Figure 10.19: Partitioning Problems

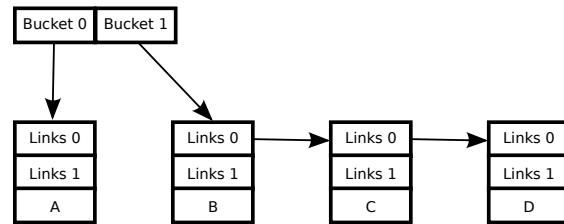


Figure 10.20: Growing a Double-List Hash Table, State (a)

The key insight behind the first hash-table implementation is that each data element can have two sets of list pointers, with one set currently being used by RCU readers (as well as by non-RCU updaters) and the other being used to construct a new resized hash table. This approach allows lookups, insertions, and deletions to all run concurrently with a resize operation (as well as with each other).

The resize operation proceeds as shown in Figures 10.20-10.23, with the initial two-bucket state shown in Figure 10.20 and with time advancing from figure to figure. The initial state uses the zero-index links to chain the elements into hash buckets. A four-bucket array is allocated, and the one-index links are used to chain the elements into these four new hash buckets. This results in state (b) shown in Figure 10.21, with readers still using the original two-bucket array.

The new four-bucket array is exposed to readers and then a grace-period operation waits for all readers, result-

10.4 Non-Partitionable Data Structures

Fixed-size hash tables are perfectly partitionable, but resizable hash tables pose partitioning challenges when growing or shrinking, as fancifully depicted in Figure 10.19. However, it turns out that it is possible to construct high-performance scalable RCU-protected hash tables, as described in the following sections.

10.4.1 Resizable Hash Table Design

In happy contrast to the situation in the early 2000s, there are now no fewer than three different types of scalable RCU-protected hash tables. The first (and simplest) was developed for the Linux kernel by Herbert Xu [Xu10], and is described in the following sections. The other two are covered briefly in Section 10.4.4.

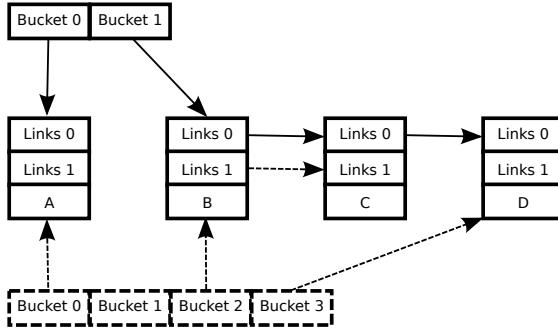


Figure 10.21: Growing a Double-List Hash Table, State (b)

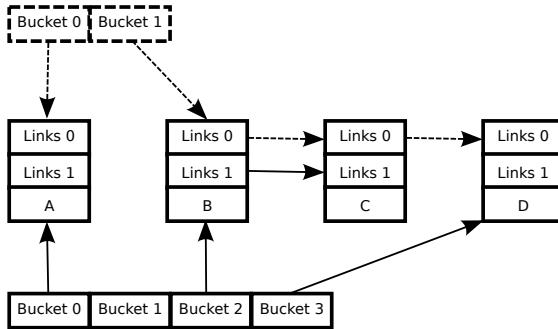


Figure 10.22: Growing a Double-List Hash Table, State (c)

ing in state (c), shown in Figure 10.22. In this state, all readers are using the new four-bucket array, which means that the old two-bucket array may now be freed, resulting in state (d), shown in Figure 10.23.

This design leads to a relatively straightforward implementation, which is the subject of the next section.

10.4.2 Resizable Hash Table Implementation

Resizing is accomplished by the classic approach of inserting a level of indirection, in this case, the `ht` structure shown on lines 12-25 of Figure 10.24. The `hashtab` structure shown on lines 27-30 contains only a pointer to the current `ht` structure along with a spinlock that is used to serialize concurrent attempts to resize the hash table. If we were to use a traditional lock- or atomic-operation-based implementation, this `hashtab` structure

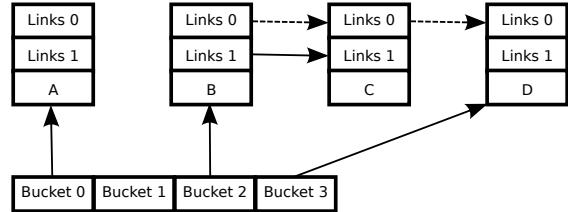


Figure 10.23: Growing a Double-List Hash Table, State (d)

```

1 struct ht_elem {
2     struct rCU_head rh;
3     struct cds_list_head hte_next[2];
4     unsigned long hte_hash;
5 };
6
7 struct ht_bucket {
8     struct cds_list_head htB_head;
9     spinlock_t htB_lock;
10 };
11
12 struct ht {
13     long ht_nbuckets;
14     long ht_resize_csr;
15     struct ht *ht_new;
16     int ht_idx;
17     void *ht_hash_private;
18     int (*ht_cmp)(void *hash_private,
19                     struct ht_elem *htep,
20                     void *key);
21     long (*ht_gethash)(void *hash_private,
22                        void *key);
23     void *(*ht_getkey)(struct ht_elem *htep);
24     struct ht_bucket ht_bkt[0];
25 };
26
27 struct hashtab {
28     struct ht *ht_csr;
29     spinlock_t ht_lock;
30 };

```

Figure 10.24: Resizable Hash-Table Data Structures

could become a severe bottleneck from both performance and scalability viewpoints. However, because resize operations should be relatively infrequent, we should be able to make good use of RCU.

The `ht` structure represents a specific size of the hash table, as specified by the `->ht_nbuckets` field on line 13. The size is stored in the same structure containing the array of buckets (`->ht_bkt[]` on line 24) in order to avoid mismatches between the size and the array. The `->ht_resize_csr` field on line 14 is equal to -1 unless a resize operation is in progress, in which case it indicates the index of the bucket whose elements are being inserted into the new hash table, which is ref-

erenced by the `->ht_new` field on line 15. If there is no resize operation in progress, `->ht_new` is `NULL`. Thus, a resize operation proceeds by allocating a new `ht` structure and referencing it via the `->ht_new` pointer, then advancing `->ht_resize_cur` through the old table's buckets. When all the elements have been added to the new table, the new table is linked into the `hashtab` structure's `->ht_cur` field. Once all old readers have completed, the old hash table's `ht` structure may be freed.

The `->ht_idx` field on line 16 indicates which of the two sets of list pointers are being used by this instantiation of the hash table, and is used to index the `->ht_e``next[]` array in the `ht_bucket` structure on line 3.

The `->ht_hash_private`, `->ht_cmp()`, `->ht_gethash()`, and `->ht_getkey()` fields on lines 17-23 collectively define the per-element key and the hash function. The `->ht_hash_private` allows the hash function to be perturbed [McK90a, McK90b, McK91], which can be used to avoid denial-of-service attacks based on statistical estimation of the parameters used in the hash function. The `->ht_cmp()` function compares a specified key with that of the specified element, the `->ht_gethash()` calculates the specified key's hash, and `->ht_getkey()` extracts the key from the enclosing data element.

The `ht_bucket` structure is the same as before, and the `ht_elem` structure differs from that of previous implementations only in providing a two-element array of list pointer sets in place of the prior single set of list pointers.

In a fixed-sized hash table, bucket selection is quite straightforward: Simply transform the hash value to the corresponding bucket index. In contrast, when resizing, it is also necessary to determine which of the old and new sets of buckets to select from. If the bucket that would be selected from the old table has already been distributed into the new table, then the bucket should be selected from the new table. Conversely, if the bucket that would be selected from the old table has not yet been distributed, then the bucket should be selected from the old table.

Bucket selection is shown in Figure 10.25, which shows `ht_get_bucket_single()` on lines 1-8 and `ht_get_bucket()` on lines 10-24. The `ht_get_bucket_single()` function returns a reference to the bucket corresponding to the specified key in the specified hash table, without making any allowances for resizing. It also stores the hash value corresponding to the key into

```

1 static struct ht_bucket *
2 ht_get_bucket_single(struct ht *http,
3                      void *key, long *b)
4 {
5     *b = http->ht_gethash(http->ht_hash_private,
6                            key) % http->ht_nbuckets;
7     return &http->ht_bkt[*b];
8 }
9
10 static struct ht_bucket *
11 ht_get_bucket(struct ht **http, void *key,
12                long *b, int *i)
13 {
14     struct ht_bucket *htbp;
15
16     htp = ht_get_bucket_single(*http, key, b);
17     if (*b <= (*http)->ht_resize_cur) {
18         *http = (*http)->ht_new;
19         htp = ht_get_bucket_single(*http, key, b);
20     }
21     if (*i)
22         *i = (*http)->ht_idx;
23     return htp;
24 }
```

Figure 10.25: Resizable Hash-Table Bucket Selection

the location referenced by parameter `b` on lines 5 and 6. Line 7 then returns a reference to the corresponding bucket.

The `ht_get_bucket()` function handles hash-table selection, invoking `ht_get_bucket_single()` on line 16 to select the bucket corresponding to the hash in the current hash table, storing the hash value through parameter `b`. If line 17 determines that the table is being resized and that line 16's bucket has already been distributed across the new hash table, then line 18 selects the new hash table and line 19 selects the bucket corresponding to the hash in the new hash table, again storing the hash value through parameter `b`.

Quick Quiz 10.7: The code in Figure 10.25 computes the hash twice! Why this blatant inefficiency? ■

If line 21 finds that parameter `i` is non-`NULL`, then line 22 stores the pointer-set index for the selected hash table. Finally, line 23 returns a reference to the selected hash bucket.

Quick Quiz 10.8: How does the code in Figure 10.25 protect against the resizing process progressing past the selected bucket? ■

This implementation of `ht_get_bucket_single()` and `ht_get_bucket()` will permit lookups and modifications to run concurrently with a resize operation.

Read-side concurrency control is provided by RCU as was shown in Figure 10.10, but the update-side concurrency-control functions `hashtab_lock_`

```

1 void hashtab_lock_mod(struct hashtab *htp_master,
2                         void *key)
3 {
4     long b;
5     struct ht *htp;
6     struct ht_bucket *htbp;
7     struct ht_bucket *htbp_new;
8
9     rcu_read_lock();
10    htp = rcu_dereference(htp_master->ht_cur);
11    htp = ht_get_bucket_single(htp, key, &b);
12    spin_lock(&htbp->htb_lock);
13    if (b > htp->ht_resize_cur)
14        return;
15    htp = htp->ht_new;
16    htp = ht_get_bucket_single(htp, key, &b);
17    spin_lock(&htbp_new->htb_lock);
18    spin_unlock(&htbp->htb_lock);
19 }
20
21 void hashtab_unlock_mod(struct hashtab *htp_master,
22                         void *key)
23 {
24     long b;
25     struct ht *htp;
26     struct ht_bucket *htbp;
27
28     htp = rcu_dereference(htp_master->ht_cur);
29     htp = ht_get_bucket(&htp, key, &b, NULL);
30     spin_unlock(&htbp->htb_lock);
31     rcu_read_unlock();
32 }

```

Figure 10.26: Resizable Hash-Table Update-Side Concurrency Control

`mod()` and `hashtab_unlock_mod()` must now deal with the possibility of a concurrent resize operation as shown in Figure 10.26.

The `hashtab_lock_mod()` spans lines 1-19 in the figure. Line 9 enters an RCU read-side critical section to prevent the data structures from being freed during the traversal, line 10 acquires a reference to the current hash table, and then line 11 obtains a reference to the bucket in this hash table corresponding to the key. Line 12 acquires that bucket's lock, which will prevent any concurrent resizing operation from distributing that bucket, though of course it will have no effect if the resizing operation has already distributed this bucket. Line 13 then checks to see if a concurrent resize operation has already distributed this bucket across the new hash table, and if not, line 14 returns with the selected hash bucket's lock held (and also within an RCU read-side critical section).

Otherwise, a concurrent resize operation has already distributed this bucket, so line 15 proceeds to the new hash table and line 16 selects the bucket corresponding to the key. Finally, line 17 acquires the bucket's lock and line 18 releases the lock for the old hash table's bucket. Once again, `hashtab_lock_mod()` exits within an

RCU read-side critical section.

Quick Quiz 10.9: The code in Figures 10.25 and 10.26 computes the hash and executes the bucket-selection logic twice for updates! Why this blatant inefficiency? ■

The `hashtab_unlock_mod()` function releases the lock acquired by `hashtab_lock_mod()`. Line 28 picks up the current hash table, and then line 29 invokes `ht_get_bucket()` in order to gain a reference to the bucket that corresponds to the key—and of course this bucket might well be in a new hash table. Line 30 releases the bucket's lock and finally line 31 exits the RCU read-side critical section.

Quick Quiz 10.10: Suppose that one thread is inserting an element into the new hash table during a resize operation. What prevents this insertion from being lost due to a subsequent resize operation completing before the insertion does? ■

Now that we have bucket selection and concurrency control in place, we are ready to search and update our resizable hash table. The `hashtab_lookup()`, `hashtab_add()`, and `hashtab_del()` functions shown in Figure 10.27.

The `hashtab_lookup()` function on lines 1-21 of the figure does hash lookups. Line 11 fetches the current hash table and line 12 obtains a reference to the bucket corresponding to the specified key. This bucket will be located in a new resized hash table when a resize operation has progressed past the bucket in the old hash table that contained the desired data element. Note that line 12 also passes back the index that will be used to select the correct set of pointers from the pair in each element. The loop spanning lines 13-19 searches the bucket, so that if line 16 detects a match, line 18 returns a pointer to the enclosing data element. Otherwise, if there is no match, line 20 returns `NULL` to indicate failure.

Quick Quiz 10.11: In the `hashtab_lookup()` function in Figure 10.27, the code carefully finds the right bucket in the new hash table if the element to be looked up has already been distributed by a concurrent resize operation. This seems wasteful for RCU-protected lookups. Why not just stick with the old hash table in this case? ■

The `hashtab_add()` function on lines 23-37 of the figure adds new data elements to the hash table. Lines 32-34 obtain a pointer to the hash bucket corresponding to the key (and provide the index), as before, and line 35 adds the new element to the table. The caller is required to handle concurrency, for example, by invoking `hashtab_lock_mod()` before the call to `hashtab_`

```

1 struct ht_elem *
2 hashtable_lookup(struct hashtable *htp_master,
3                  void *key)
4 {
5     long b;
6     int i;
7     struct ht *htp;
8     struct ht_elem *htep;
9     struct ht_bucket *htbp;
10
11    htp = rcu_dereference(htp_master->ht_cur);
12    htp = ht_get_bucket(htp, key, &b, &i);
13    cds_list_for_each_entry_rcu(htep,
14                                &htbp->htb_head,
15                                hte_next[i]) {
16        if (htp->ht_cmp(htp->ht_hash_private,
17                         htep, key))
18            return htep;
19    }
20    return NULL;
21 }
22
23 void
24 hashtable_add(struct hashtable *htp_master,
25                struct ht_elem *htep)
26 {
27     long b;
28     int i;
29     struct ht *htp;
30     struct ht_bucket *htbp;
31
32     htp = rcu_dereference(htp_master->ht_cur);
33     htp = ht_get_bucket(&htp, htp->ht_getkey(htep),
34                         &b, &i);
35     cds_list_add_rcu(&htep->hte_next[i],
36                       &htbp->htb_head);
37 }
38
39 void
40 hashtable_del(struct hashtable *htp_master,
41                struct ht_elem *htep)
42 {
43     long b;
44     int i;
45     struct ht *htp;
46     struct ht_bucket *htbp;
47
48     htp = rcu_dereference(htp_master->ht_cur);
49     htp = ht_get_bucket(&htp, htp->ht_getkey(htep),
50                         &b, &i);
51     cds_list_del_rcu(&htep->hte_next[i]);
52 }

```

Figure 10.27: Resizable Hash-Table Access Functions

add() and invoking hashtable_unlock_mod() afterwards. These two concurrency-control functions will correctly synchronize with a concurrent resize operation: If the resize operation has already progressed beyond the bucket that this data element would have been added to, then the element is added to the new table.

The hashtable_del() function on lines 39-52 of the figure removes an existing element from the hash table. Lines 48-50 provide the bucket and index as before, and line 51 removes the specified element. As with

hashtable_add(), the caller is responsible for concurrency control and this concurrency control suffices for synchronizing with a concurrent resize operation.

Quick Quiz 10.12: The hashtable_del() function in Figure 10.27 does not always remove the element from the old hash table. Doesn't this mean that readers might access this newly removed element after it has been freed? ■

The actual resizing itself is carried out by hashtable_resize, shown in Figure 10.28 on page 172. Line 17 conditionally acquires the top-level ->ht_lock, and if this acquisition fails, line 18 returns -EBUSY to indicate that a resize is already in progress. Otherwise, line 19 picks up a reference to the current hash table, and lines 21-24 allocate a new hash table of the desired size. If a new set of hash/key functions have been specified, these are used for the new table, otherwise those of the old table are preserved. If line 25 detects memory-allocation failure, line 26 releases ->htlock and line 27 returns a failure indication.

Line 29 starts the bucket-distribution process by installing a reference to the new table into the ->ht_new field of the old table. Line 30 ensures that all readers who are not aware of the new table complete before the resize operation continues. Line 31 picks up the current table's index and stores its inverse to the new hash table, thus ensuring that the two hash tables avoid overwriting each other's linked lists.

Each pass through the loop spanning lines 33-44 distributes the contents of one of the old hash table's buckets into the new hash table. Line 34 picks up a reference to the old table's current bucket, line 35 acquires that bucket's spinlock, and line 36 updates ->ht_resize_cur to indicate that this bucket is being distributed.

Quick Quiz 10.13: In the hashtable_resize() function in Figure 10.27, what guarantees that the update to ->ht_new on line 29 will be seen as happening before the update to ->ht_resize_cur on line 36 from the perspective of hashtable_lookup(), hashtable_add(), and hashtable_del()? ■

Each pass through the loop spanning lines 37-42 adds one data element from the current old-table bucket to the corresponding new-table bucket, holding the new-table bucket's lock during the add operation. Finally, line 43 releases the old-table bucket lock.

Execution reaches line 45 once all old-table buckets have been distributed across the new table. Line 45 installs the newly created table as the current one, and line 46 waits for all old readers (who might still be referencing

```

1 int hashtab_resize(struct hashtab *htp_master,
2                     unsigned long nbuckets, void *hash_private,
3                     int (*cmp)(void *hash_private, struct ht_elem *htep, void *key),
4                     long (*gethash)(void *hash_private, void *key),
5                     void *(*getkey)(struct ht_elem *htep))
6 {
7     struct ht *htp;
8     struct ht *htp_new;
9     int i;
10    int idx;
11    struct ht_elem *htep;
12    struct ht_bucket *htbp;
13    struct ht_bucket *htbp_new;
14    unsigned long hash;
15    long b;
16
17    if (!spin_trylock(&htp_master->ht_lock))
18        return -EBUSY;
19    htp = htp_master->ht_cur;
20    htp_new = ht_alloc(nbuckets,
21                      hash_private ? hash_private : htp->ht_hash_private,
22                      cmp ? cmp : htp->ht_cmp,
23                      gethash ? gethash : htp->ht_gethash,
24                      getkey ? getkey : htp->ht_getkey);
25    if (htp_new == NULL) {
26        spin_unlock(&htp_master->ht_lock);
27        return -ENOMEM;
28    }
29    htp->ht_new = htp_new;
30    synchronize_rcu();
31    idx = htp->ht_idx;
32    htp_new->ht_idx = !idx;
33    for (i = 0; i < htp->ht_nbuckets; i++) {
34        htpb = &htp->ht_bkt[i];
35        spin_lock(&htbp->htb_lock);
36        htp->ht_resize_cur = i;
37        cds_list_for_each_entry(htep, &htbp->htb_head, hte_next[idx]) {
38            htpb_new = ht_get_bucket_single(htp_new, htp_new->ht_getkey(htep), &b);
39            spin_lock(&htbp_new->htb_lock);
40            cds_list_add_rcu(&htep->hte_next[!idx], &htbp_new->htb_head);
41            spin_unlock(&htbp_new->htb_lock);
42        }
43        spin_unlock(&htbp->htb_lock);
44    }
45    rcu_assign_pointer(htp_master->ht_cur, htp_new);
46    synchronize_rcu();
47    spin_unlock(&htp_master->ht_lock);
48    free(htp);
49    return 0;
50 }

```

Figure 10.28: Resizable Hash-Table Resizing

the old table) to complete. Then line 47 releases the resize-serialization lock, line 48 frees the old hash table, and finally line 48 returns success.

10.4.3 Resizable Hash Table Discussion

Figure 10.29 compares resizing hash tables to their fixed-sized counterparts for 2048, 16,384, and 131,072 elements in the hash table. The figure shows three traces for each element count, one for a fixed-size 1024-bucket hash table, another for a fixed-size 2048-bucket hash table, and a third for a resizable hash table that shifts back and forth

between 1024 and 2048 buckets, with a one-millisecond pause between each resize operation.

The uppermost three traces are for the 2048-element hash table. The upper trace corresponds to the 2048-bucket fixed-size hash table, the middle trace to the 1024-bucket fixed-size hash table, and the lower trace to the resizable hash table. In this case, the short hash chains cause normal lookup overhead to be so low that the overhead of resizing dominates. Nevertheless, the larger fixed-size hash table has a significant performance advantage, so that resizing can be quite beneficial, at least given sufficient time between resizing operations: One millisecond

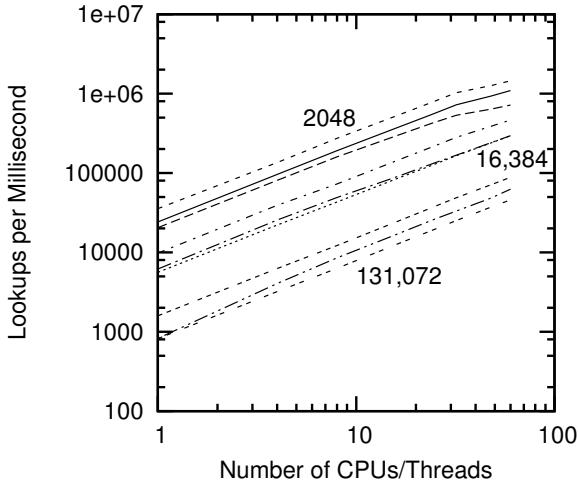


Figure 10.29: Overhead of Resizing Hash Tables

is clearly too short a time.

The middle three traces are for the 16,384-element hash table. Again, the upper trace corresponds to the 2048-bucket fixed-size hash table, but the middle trace now corresponds to the resizable hash table and the lower trace to the 1024-bucket fixed-size hash table. However, the performance difference between the resizable and the 1024-bucket hash table is quite small. One consequence of the eight-fold increase in number of elements (and thus also in hash-chain length) is that incessant resizing is now no worse than maintaining a too-small hash table.

The lower three traces are for the 131,072-element hash table. The upper trace corresponds to the 2048-bucket fixed-size hash table, the middle trace to the resizable hash table, and the lower trace to the 1024-bucket fixed-size hash table. In this case, longer hash chains result in higher lookup overhead, so that this lookup overhead dominates that of resizing the hash table. However, the performance of all three approaches at the 131,072-element level is more than an order of magnitude worse than that at the 2048-element level, suggesting that the best strategy would be a single 64-fold increase in hash-table size.

The key point from this data is that the RCU-protected resizable hash table performs and scales almost as well as does its fixed-size counterpart. The performance during an actual resize operation of course suffers somewhat due to the cache misses caused by the updates to each element's pointers, and this effect is most pronounced when the hash-tables bucket lists are short. This indicates that hash tables should be resized by substantial amounts,

and that hysteresis should be applied to prevent performance degradation due to too-frequent resize operations. In memory-rich environments, hash-table sizes should furthermore be increased much more aggressively than they are decreased.

Another key point is that although the `hashtab` structure is non-partitionable, it is also read-mostly, which suggests the use of RCU. Given that the performance and scalability of this resizable hash table is very nearly that of RCU-protected fixed-sized hash tables, we must conclude that this approach was quite successful.

Finally, it is important to note that insertions, deletions, and lookups can proceed concurrently with a resize operation. This concurrency is critically important when resizing large hash tables, especially for applications that must meet severe response-time constraints.

Of course, the `ht_elem` structure's pair of pointer sets does impose some memory overhead, which is taken up in the next section.

10.4.4 Other Resizable Hash Tables

One shortcoming of the resizable hash table described earlier in this section is memory consumption. Each data element has two pairs of linked-list pointers rather than just one. Is it possible to create an RCU-protected resizable hash table that makes do with just one pair?

It turns out that the answer is “yes.” Josh Triplett et al. [TMW11] produced a *relativistic hash table* that incrementally splits and combines corresponding hash chains so that readers always see valid hash chains at all points during the resizing operation. This incremental splitting and combining relies on the fact that it is harmless for a reader to see a data element that should be in some other hash chain: When this happens, the reader will simply ignore the extraneous data element due to key mismatches.

The process of shrinking a relativistic hash table by a factor of two is shown in Figure 10.30, in this case shrinking a two-bucket hash table into a one-bucket hash table, otherwise known as a linear list. This process works by coalescing pairs of buckets in the old larger hash table into single buckets in the new smaller hash table. For this process to work correctly, we clearly need to constrain the hash functions for the two tables. One such constraint is to use the same underlying hash function for both tables, but to throw out the low-order bit when shrinking from large to small. For example, the old two-bucket hash table would use the two top bits of the value, while the new one-bucket hash table could use the top bit of the value.

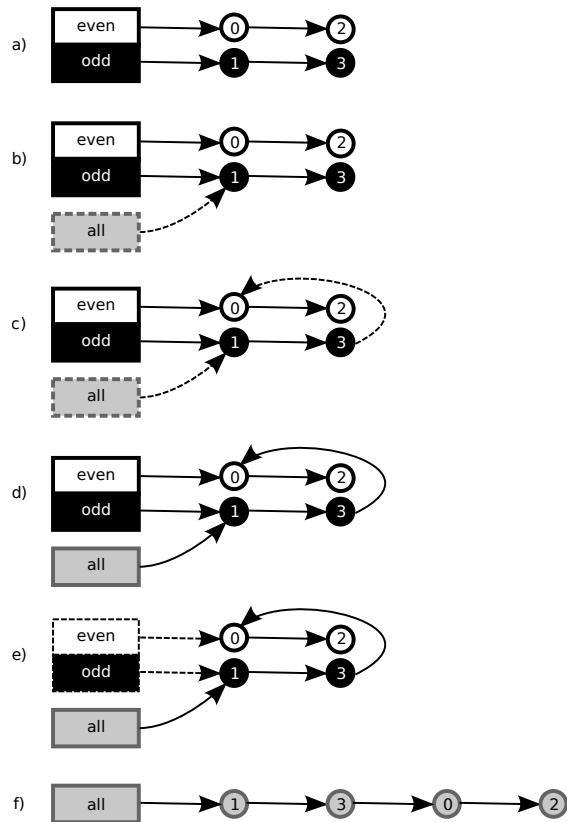


Figure 10.30: Shrinking a Relativistic Hash Table

In this way, a given pair of adjacent even and odd buckets in the old large hash table can be coalesced into a single bucket in the new small hash table, while still having a single hash value cover all of the elements in that single bucket.

The initial state is shown at the top of the figure, with time advancing from top to bottom, starting with initial state (a). The shrinking process begins by allocating the new smaller array of buckets, and having each bucket of this new smaller array reference the first element of one of the buckets of the corresponding pair in the old large hash table, resulting in state (b).

Then the two hash chains are linked together, resulting in state (c). In this state, readers looking up an even-numbered element see no change, and readers looking up elements 1 and 3 likewise see no change. However, readers looking up some other odd number will also traverse elements 0 and 2. This is harmless because any odd number will compare not-equal to these two elements. There

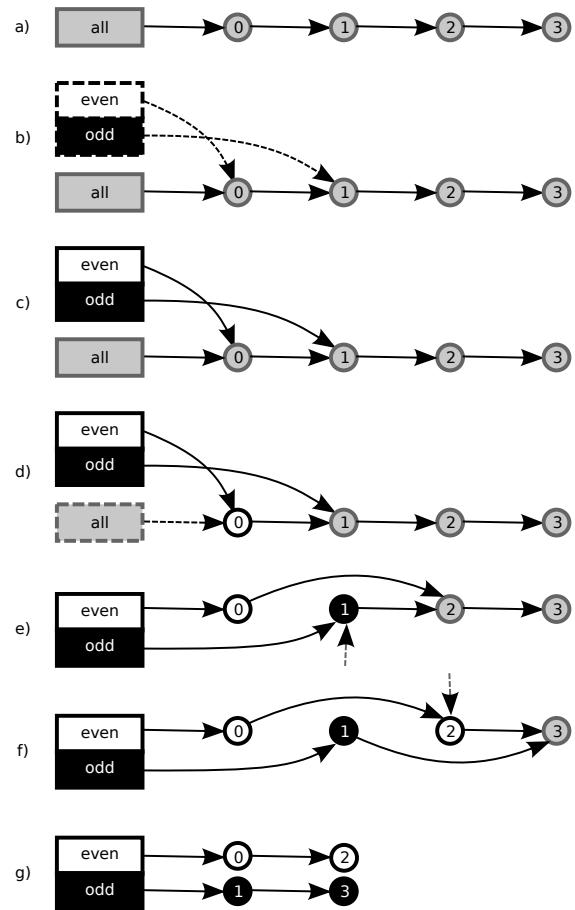


Figure 10.31: Growing a Relativistic Hash Table

is some performance loss, but on the other hand, this is exactly the same performance loss that will be experienced once the new small hash table is fully in place.

Next, the new small hash table is made accessible to readers, resulting in state (d). Note that older readers might still be traversing the old large hash table, so in this state both hash tables are in use.

The next step is to wait for all pre-existing readers to complete, resulting in state (e). In this state, all readers are using the new small hash table, so that the old large hash table's buckets may be freed, resulting in the final state (f).

Growing a relativistic hash table reverses the shrinking process, but requires more grace-period steps, as shown in Figure 10.31. The initial state (a) is at the top of this figure, with time advancing from top to bottom.

We start by allocating the new large two-bucket hash table, resulting in state (b). Note that each of these new buckets references the first element destined for that bucket. These new buckets are published to readers, resulting in state (c). After a grace-period operation, all readers are using the new large hash table, resulting in state (d). In this state, only those readers traversing the even-values hash bucket traverse element 0, which is therefore now colored white.

At this point, the old small hash buckets may be freed, although many implementations use these old buckets to track progress “unzipping” the list of items into their respective new buckets. The last even-numbered element in the first consecutive run of such elements now has its pointer-to-next updated to reference the following even-numbered element. After a subsequent grace-period operation, the result is state (e). The vertical arrow indicates the next element to be unzipped, and element 1 is now colored black to indicate that only those readers traversing the odd-values hash bucket may reach it.

Next, the last odd-numbered element in the first consecutive run of such elements now has its pointer-to-next updated to reference the following odd-numbered element. After a subsequent grace-period operation, the result is state (f). A final unzipping operation (including a grace-period operation) results in the final state (g).

In short, the relativistic hash table reduces the number of per-element list pointers at the expense of additional grace periods incurred during resizing. These additional grace periods are usually not a problem because insertions, deletions, and lookups may proceed concurrently with a resize operation.

It turns out that it is possible to reduce the per-element memory overhead from a pair of pointers to a single pointer, while still retaining $O(1)$ deletions. This is accomplished by augmenting split-order list [SS06] with RCU protection [Des09, MDJ13a]. The data elements in the hash table are arranged into a single sorted linked list, with each hash bucket referencing the first element in that bucket. Elements are deleted by setting low-order bits in their pointer-to-next fields, and these elements are removed from the list by later traversals that encounter them.

This RCU-protected split-order list is complex, but offers lock-free progress guarantees for all insertion, deletion, and lookup operations. Such guarantees can be important in real-time applications. An implementation is available from recent versions of the userspace RCU library [Des09].

10.5 Other Data Structures

The preceding sections have focused on data structures that enhance concurrency due to partitionability (Section 10.2), efficient handling of read-mostly access patterns (Section 10.3), or application of read-mostly techniques to avoid non-partitionability (Section 10.4). This section gives a brief review of other data structures.

One of the hash table’s greatest advantages for parallel use is that it is fully partitionable, at least while not being resized. One way of preserving the partitionability and the size independence is to use a radix tree, which is also called a trie. Tries partition the search key, using each successive key partition to traverse the next level of the trie. As such, a trie can be thought of as a set of nested hash tables, thus providing the required partitionability. One disadvantage of tries is that a sparse key space can result in inefficient use of memory. There are a number of compression techniques that may be used to work around this disadvantage, including hashing the key value to a smaller keyspace before the traversal [ON06]. Radix trees are heavily used in practice, including in the Linux kernel [Pig06].

One important special case of both a hash table and a trie is what is perhaps the oldest of data structures, the array and its multi-dimensional counterpart, the matrix. The fully partitionable nature of matrices is exploited heavily in concurrent numerical algorithms.

Self-balancing trees are heavily used in sequential code, with AVL trees and red-black trees being perhaps the most well-known examples [CLRS01]. Early attempts to parallelize AVL trees were complex and not necessarily all that efficient [Ell80], however, more recent work on red-black trees provides better performance and scalability by using RCU for readers and hashed arrays of locks¹ to protect reads and updates, respectively [HW11, HW13]. It turns out that red-black trees rebalance aggressively, which works well for sequential programs, but not necessarily so well for parallel use. Recent work has therefore made use of RCU-protected “bonsai trees” that rebalance less aggressively [CKZ12], trading off optimal tree depth to gain more efficient concurrent updates.

Concurrent skip lists lend themselves well to RCU readers, and in fact represents an early academic use of a technique resembling RCU [Pug90].

Concurrent double-ended queues were discussed in Section 6.1.2, and concurrent stacks and queues have a

¹ In the guise of swissTM [DFGG11], which is a variant of software transactional memory in which the developer flags non-shared accesses.

long history [Tre86], though not normally the most impressive performance or scalability. They are nevertheless a common feature of concurrent libraries [MDJ13b]. Researchers have recently proposed relaxing the ordering constraints of stacks and queues [Sha11], with some work indicating that relaxed-ordered queues actually have better ordering properties than do strict FIFO queues [HKLP12, KLP12, HHK⁺13].

It seems likely that continued work with concurrent data structures will produce novel algorithms with surprising properties.

10.6 Micro-Optimization

The data structures shown in this section were coded straightforwardly, with no adaptation to the underlying system's cache hierarchy. In addition, many of the implementations used pointers to functions for key-to-hash conversions and other frequent operations. Although this approach provides simplicity and portability, in many cases it does give up some performance.

The following sections touch on specialization, memory conservation, and hardware considerations. Please do not mistake these short sections for a definitive treatise on this subject. Whole books have been written on optimizing to a specific CPU, let alone to the set of CPU families in common use today.

10.6.1 Specialization

The resizable hash table presented in Section 10.4 used an opaque type for the key. This allows great flexibility, permitting any sort of key to be used, but it also incurs significant overhead due to the calls via of pointers to functions. Now, modern hardware uses sophisticated branch-prediction techniques to minimize this overhead, but on the other hand, real-world software is often larger than can be accommodated even by today's large hardware branch-prediction tables. This is especially the case for calls via pointers, in which case the branch prediction hardware must record a pointer in addition to branch-taken/branch-not-taken information.

This overhead can be eliminated by specializing a hash-table implementation to a given key type and hash function. Doing so eliminates the `->ht_cmp()`, `->ht_gethash()`, and `->ht_getkey()` function pointers in the `ht` structure shown in Figure 10.24 on page 168. It also eliminates the corresponding calls through these

pointers, which could allow the compiler to inline the resulting fixed functions, eliminating not only the overhead of the call instruction, but the argument marshalling as well.

In addition, the resizable hash table is designed to fit an API that segregates bucket selection from concurrency control. Although this allows a single torture test to exercise all the hash-table implementations in this chapter, it also means that many operations must compute the hash and interact with possible resize operations twice rather than just once. In a performance-conscious environment, the `hashtab_lock_mod()` function would also return a reference to the bucket selected, eliminating the subsequent call to `ht_get_bucket()`.

Quick Quiz 10.14: Couldn't the `hashtorture.h` code be modified to accommodate a version of `hashtab_lock_mod()` that subsumes the `ht_get_bucket()` functionality? ■

Quick Quiz 10.15: How much do these specializations really save? Are they really worth it? ■

All that aside, one of the great benefits of modern hardware compared to that available when I first started learning to program back in the early 1970s is that much less specialization is required. This allows much greater productivity than was possible back in the days of four-kilobyte address spaces.

10.6.2 Bits and Bytes

The hash tables discussed in this chapter made almost no attempt to conserve memory. For example, the `->ht_idx` field in the `ht` structure in Figure 10.24 on page 168 always has a value of either zero or one, yet takes up a full 32 bits of memory. It could be eliminated, for example, by stealing a bit from the `->ht_resize_key` field. This works because the `->ht_resize_key` field is large enough to address every byte of memory and the `ht_bucket` structure is more than one byte long, so that the `->ht_resize_key` field must have several bits to spare.

This sort of bit-packing trick is frequently used in data structures that are highly replicated, as is the page structure in the Linux kernel. However, the resizable hash table's `ht` structure is not all that highly replicated. It is instead the `ht_bucket` structures we should focus on. There are two major opportunities for shrinking the `ht_bucket` structure: (1) Placing the `->htb_lock` field in a low-order bit of one of the `->htb_head` pointers and (2) Reducing the number of pointers required.

The first opportunity might make use of bit-spinlocks in the Linux kernel, which are provided by the `include/linux/bit_spinlock.h` header file. These are used in space-critical data structures in the Linux kernel, but are not without their disadvantages:

1. They are significantly slower than the traditional spinlock primitives.
2. They cannot participate in the lockdep deadlock detection tooling in the Linux kernel [Cor06a].
3. They do not record lock ownership, further complicating debugging.
4. They do not participate in priority boosting in `-rt` kernels, which means that preemption must be disabled when holding bit spinlocks, which can degrade real-time latency.

Despite these disadvantages, bit-spinlocks are extremely useful when memory is at a premium.

One aspect of the second opportunity was covered in Section 10.4.4, which presented resizable hash tables that require only one set of bucket-list pointers in place of the pair of sets required by the resizable hash table presented in Section 10.4. Another approach would be to use singly linked bucket lists in place of the doubly linked lists used in this chapter. One downside of this approach is that deletion would then require additional overhead, either by marking the outgoing pointer for later removal or by searching the bucket list for the element being deleted.

In short, there is a tradeoff between minimal memory overhead on the one hand, and performance and simplicity on the other. Fortunately, the relatively large memories available on modern systems have allowed us to prioritize performance and simplicity over memory overhead. However, even with today's large-memory systems² it is sometime necessary to take extreme measures to reduce memory overhead.

10.6.3 Hardware Considerations

Modern computers typically move data between CPUs and main memory in fixed-sized blocks that range in size from 32 bytes to 256 bytes. These blocks are called *cache lines*, and are extremely important to high performance and scalability, as was discussed in Section 3.2. One time-worn way to kill both performance and scalability is to place incompatible variables into the same cacheline. For

```
struct hash_elem {
    struct ht_elem e;
    long __attribute__ ((aligned(64))) counter;
};
```

Figure 10.32: Alignment for 64-Byte Cache Lines

example, suppose that a resizable hash table data element had the `ht_elem` structure in the same cacheline as a counter that was incremented quite frequently. The frequent incrementing would cause the cacheline to be present at the CPU doing the incrementing, but nowhere else. If other CPUs attempted to traverse the hash bucket list containing that element, they would incur expensive cache misses, degrading both performance and scalability.

One way to solve this problem on systems with 64-byte cache line is shown in Figure 10.32. Here a `gcc aligned` attribute is used to force the `->counter` and the `ht_elem` structure into separate cache lines. This would allow CPUs to traverse the hash bucket list at full speed despite the frequent incrementing.

Of course, this raises the question "How did we know that cache lines are 64 bytes in size?" On a Linux system, this information may be obtained from the `/sys/devices/system/cpu/cpu*/cache` directories, and it is even possible to make the installation process rebuild the application to accommodate the system's hardware structure. However, this would be more difficult if you wanted your application to also run on non-Linux systems. Furthermore, even if you were content to run only on Linux, such a self-modifying installation poses validation challenges.

Fortunately, there are some rules of thumb that work reasonably well in practice, which were gathered into a 1995 paper [GKPS95].³ The first group of rules involve rearranging structures to accommodate cache geometry:

1. Separate read-mostly data from data that is frequently updated. For example, place read-mostly data at the beginning of the structure and frequently updated data at the end. Where possible, place data that is rarely accessed in between.
2. If the structure has groups of fields such that each group is updated by an independent code path, separate these groups from each other. Again, it can make sense to place data that is rarely accessed between the groups. In some cases, it might also make sense

³ A number of these rules are paraphrased and expanded on here with permission from Orran Krieger.

² Smartphones with gigabytes of memory, anyone?

to place each such group into a separate structure referenced by the original structure.

3. Where possible, associate update-mostly data with a CPU, thread, or task. We saw several very effective examples of this rule of thumb in the counter implementations in Chapter 5.
4. In fact, where possible, you should partition your data on a per-CPU, per-thread, or per-task basis, as was discussed in Chapter 8.

There has recently been some work towards automated trace-based rearrangement of structure fields [GDZE10]. This work might well ease one of the more painstaking tasks required to get excellent performance and scalability from multithreaded software.

An additional set of rules of thumb deal with locks:

1. Given a heavily contended lock protecting data that is frequently modified, take one of the following approaches:
 - (a) Place the lock in a different cacheline than the data that it protects.
 - (b) Use a lock that is adapted for high contention, such as a queued lock.
 - (c) Redesign to reduce lock contention. (This approach is best, but can require quite a bit of work.)
2. Place uncontended locks into the same cache line as the data that they protect. This approach means that the cache miss that brought the lock to the current CPU also brought its data.
3. Protect read-mostly data with RCU, or, if RCU cannot be used and the critical sections are of very long duration, reader-writer locks.

Of course, these are rules of thumb rather than absolute rules. Some experimentation is required to work out which are most applicable to your particular situation.

10.7 Summary

This chapter has focused primarily on hash tables, including resizable hash tables, which are not fully partitionable. Section 10.5 gave a quick overview of a few non-hash-table data structures. Nevertheless, this exposition of hash tables is an excellent introduction to the

many issues surrounding high-performance scalable data access, including:

1. Fully partitioned data structures work well on small systems, for example, single-socket systems.
2. Larger systems require locality of reference as well as full partitioning.
3. Read-mostly techniques, such as hazard pointers and RCU, provide good locality of reference for read-mostly workloads, and thus provide excellent performance and scalability even on larger systems.
4. Read-mostly techniques also work well on some types of non-partitionable data structures, such as resizable hash tables.
5. Additional performance and scalability can be obtained by specializing the data structure to a specific workload, for example, by replacing a general key with a 32-bit integer.
6. Although requirements for portability and for extreme performance often conflict, there are some data-structure-layout techniques that can strike a good balance between these two sets of requirements.

That said, performance and scalability is of little use without reliability, so the next chapter covers validation.

Chapter 11

Validation

I have had a few parallel programs work the first time, but that is only because I have written a large number 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.¹

A longer course may be found in many recent books on validation, as well as at least one rather old but quite worthwhile one [Mye79]. 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.

Section 11.1 introduces the philosophy of debugging. Section 11.2 discusses tracing, Section 11.3 discusses assertions, and Section 11.4 discusses static analysis. Section 11.5 describes some unconventional approaches to code review that can be helpful when the fabled 10,000 eyes happen not to be looking at your code. Section 11.6 gives an overview of the use of probability for validating parallel software. Because performance and scalability are first-class requirements for parallel programming, Section 11.7 covers these topics. Finally, Section 11.8 gives a fanciful summary and a short list of statistical traps to avoid.

¹ But you can leave the thumbscrews and waterboards at home. This chapter covers much more sophisticated and effective methods, especially given that most computer systems neither feel pain nor fear drowning.

11.1 Introduction

Section 11.1.1 discusses the sources of bugs, and Section 11.1.2 overviews the mindset required when validating software. Section 11.1.3 discusses when you should start validation, and Section 11.1.4 describes the surprisingly effective open-source regimen of code review and community testing.

11.1.1 Where Do Bugs Come From?

Bugs come from developers. The basic problem is that the human brain did not evolve with computer software in mind. Instead, the human brain evolved in concert with other human brains and with animal brains. Because of this history, the following three characteristics of computers often come as a shock to human intuition:

1. Computers typically lack common sense, despite decades of research sacrificed at the altar of artificial intelligence.
2. Computers generally fail to understand user intent, or more formally, computers generally lack a theory of mind.
3. Computers usually cannot do anything useful with a fragmentary plan, instead requiring that each and every detail of each and every possible scenario be spelled out in full.

The first two points should be uncontroversial, as they are illustrated by any number of failed products, perhaps most famously Clippy and Microsoft Bob. By attempting to relate to users as people, these two products raised common-sense and theory-of-mind expectations that they proved incapable of meeting. Perhaps the set of software

assistants that have recently started appearing on smartphones will fare better. That said, the developers working on them by all accounts still develop the old way: The assistants might well benefit end users, but not so much their own developers.

This human love of fragmentary plans deserves more explanation, especially given that it is a classic two-edged sword. This love of fragmentary plans is apparently due to the assumption that the person carrying out the plan will have (1) common sense and (2) a good understanding of the intent behind the plan. This latter assumption is especially likely to hold in the common case where the person doing the planning and the person carrying out the plan are one and the same: In this case, the plan will be revised almost subconsciously as obstacles arise. Therefore, the love of fragmentary plans has served human beings well, in part because it is better to take random actions that have a high probability of locating food than to starve to death while attempting to plan the unplannable. However, the past usefulness of fragmentary plans in everyday life is no guarantee of their future usefulness in stored-program computers.

Furthermore, the need to follow fragmentary plans has had important effects on the human psyche, due to the fact that throughout much of human history, life was often difficult and dangerous. It should come as no surprise that executing a fragmentary plan that has a high probability of a violent encounter with sharp teeth and claws requires almost insane levels of optimism—a level of optimism that actually is present in most human beings. These insane levels of optimism extend to self-assessments of programming ability, as evidenced by the effectiveness of (and the controversy over) interviewing techniques involving coding trivial programs [Bra07]. In fact, the clinical term for a human being with less-than-insane levels of optimism is “clinically depressed.” Such people usually have extreme difficulty functioning in their daily lives, underscoring the perhaps counter-intuitive importance of insane levels of optimism to a normal, healthy life. If you are not insanely optimistic, you are less likely to start a difficult but worthwhile project.²

Quick Quiz 11.1: When in computing is the willingness to follow a fragmentary plan critically important?

² There are some famous exceptions to this rule of thumb. One set of exceptions is people who take on difficult or risky projects in order to make at least a temporary escape from their depression. Another set is people who have nothing to lose: the project is literally a matter of life or death.

An important special case is the project that, while valuable, is not valuable enough to justify the time required to implement it. This special case is quite common, and one early symptom is the unwillingness of the decision-makers to invest enough to actually implement the project. A natural reaction is for the developers to produce an unrealistically optimistic estimate in order to be permitted to start the project. If the organization (be it open source or proprietary) is strong enough, it might survive the resulting schedule slips and budget overruns, so that the project might see the light of day. However, if the organization is not strong enough and if the decision-makers fail to cancel the project as soon as it becomes clear that the estimates are garbage, then the project might well kill the organization. This might result in another organization picking up the project and either completing it, cancelling it, or being killed by it. A given project might well succeed only after killing several organizations. One can only hope that the organization that eventually makes a success of a serial-organization-killer project manages to maintain a suitable level of humility, lest it be killed by the next project.

Important though insane levels of optimism might be, they are a key source of bugs (and perhaps failure of organizations). The question is therefore “How to maintain the optimism required to start a large project while at the same time injecting enough reality to keep the bugs down to a dull roar?” The next section examines this conundrum.

11.1.2 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. This means that if you are the type of person who enjoys breaking things, validation is just the right type of job for you.

Quick Quiz 11.2: 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? ■

But perhaps you are a super-programmer whose code is always perfect the first time every time. If so, congratulations! Feel free to skip this chapter, but I do hope that you will forgive my skepticism. You see, I have met far more people who claimed to be able to write perfect code the first time than I have people who were actually capable of carrying out this feat, which is not too surprising given the previous discussion of optimism and over-confidence. And even if you really are a super-programmer, you just might find yourself debugging lesser mortals' work.

One approach for the rest of us is to alternate between our normal state of insane optimism (Sure, I can program that!) and severe pessimism (It seems to work, but I just know that there have to be more bugs hiding in there somewhere!). It helps if you enjoy breaking things. If you don't, or if your joy in breaking things is limited to breaking *other* people's things, find someone who does love breaking your code and get them to help you test it.

Another helpful frame of mind is to hate it when other people find bugs in your code. This hatred can help motivate you to torture your code beyond reason in order to increase the probability that you find the bugs rather than someone else.

One final frame of mind is to consider the possibility that someone's life depends on your code being correct. This can also motivate you to torture your code into revealing the whereabouts of its bugs.

This wide variety of frames of mind opens the door to the possibility of multiple people with different frames of mind contributing to the project, with varying levels of optimism. This can work well, if properly organized.

Some people might see vigorous validation as a form of torture, as depicted in Figure 11.1.³ Such people might do well to remind themselves that, Tux cartoons aside, they are really torturing an inanimate object, as shown in Figure 11.2. In addition, rest assured that those who fail to torture their code are doomed to be tortured by it.

³ More cynical people might question whether these people are instead merely afraid that validation will find bugs that they will then be expected to fix.

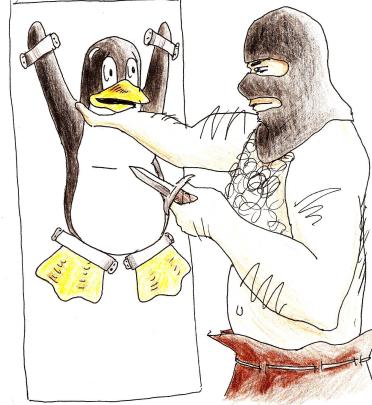


Figure 11.1: Validation and the Geneva Convention

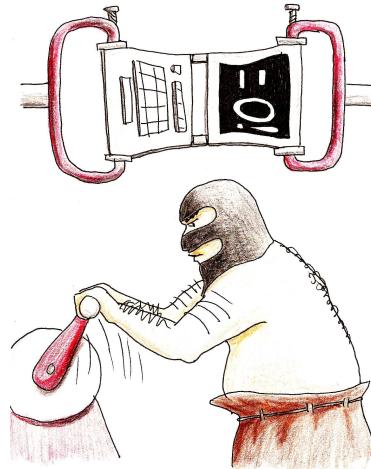


Figure 11.2: Rationalizing Validation

However, this leaves open the question of exactly when during the project lifetime validation should start, a topic taken up by the next section.

11.1.3 When Should Validation Start?

Validation should start at the same time that the project starts.

To see this, consider that tracking down a bug is much harder in a large program than in a small one. Therefore, to minimize the time and effort required to track down bugs, you should test small units of code. Although you

won't find all the bugs this way, you will find a substantial fraction, and it will be much easier to find and fix the ones you do find. Testing at this level can also alert you to larger flaws in your overall design, minimizing the time you waste writing code that is quite literally broken by design.

But why wait until you have code before validating your design?⁴ Hopefully reading Chapters 3 and 4 provided you with the information required to avoid some regrettably common design flaws, but discussing your design with a colleague or even simply writing it down can help flush out additional flaws.

However, it is all too often the case that waiting to start validation until you have a design is waiting too long. Mightn't your natural level of optimism caused you to start the design before you fully understood the requirements? The answer to this question will almost always be "yes". One good way to avoid flawed requirements is to get to know your users. To really serve them well, you will have to live among them.

Quick Quiz 11.3: You are asking me to do all this validation BS before I even start coding??? That sounds like a great way to never get started!!! ■

First-of-a-kind projects require different approaches to validation, for example, rapid prototyping. Here, the main goal of the first few prototypes is to learn how the project should be implemented, not so much to create a correct implementation on the first try. However, it is important to keep in mind that you should not omit validation, but rather take a radically different approach to it.

Now that we have established that you should start validation when you start the project, the following sections cover a number of validation techniques and methods that have proven their worth.

11.1.4 The Open Source Way

The open-source programming methodology has proven quite effective, and includes a regimen of intense code review and testing.

I can personally attest to the effectiveness of the open-source community's intense code review. One of the first patches I prepared for the Linux kernel involved a distributed filesystem where a user on one node writes to a given file at a location that a user on another node has mapped into memory. In this case, it is necessary to invalidate the affected pages from the mapping in order

to allow the filesystem to maintain coherence during the write operation. I coded up a first attempt at a patch, and, in keeping with the open-source maxim "post early, post often", I posted the patch. I then considered how I was going to test it.

But before I could even decide on an overall test strategy, I got a reply to my posting pointing out a few bugs. I fixed the bugs and reposted the patch, and returned to thinking out my test strategy. However, before I had a chance to write any test code, I received a reply to my reposted patch, pointing out more bugs. This process repeated itself many times, and I am not sure that I ever got a chance to actually test the patch.

This experience brought home the truth of the open-source saying: Given enough eyeballs, all bugs are shallow [Ray99].

However, when you post some code or a given patch, it is worth asking a few questions:

1. How many of those eyeballs are actually going to look at your code?
2. How many will be experienced and clever enough to actually find your bugs?
3. Exactly when are they going to look?

I was lucky: There was someone out there who wanted the functionality provided by my patch, who had long experience with distributed filesystems, and who looked at my patch almost immediately. If no one had looked at my patch, there would have been no review, and therefore no finding of bugs. If the people looking at my patch had lacked experience with distributed filesystems, it is unlikely that they would have found all the bugs. Had they waited months or even years to look, I likely would have forgotten how the patch was supposed to work, making it much more difficult to fix them.

However, we must not forget the second tenet of the open-source development, namely intensive testing. For example, a great many people test the Linux kernel. Some test patches as they are submitted, perhaps even yours. Others test the -next tree, which is helpful, but there is likely to be several weeks or even months delay between the time that you write the patch and the time that it appears in the -next tree, by which time the patch will not be quite as fresh in your mind. Still others test maintainer trees, which often have a similar time delay.

Quite a few people don't test code until it is committed to mainline, or the master source tree (Linus's tree in the case of the Linux kernel). If your maintainer won't accept

⁴ The old saying "First we must code, then we have incentive to think" notwithstanding.

your patch until it has been tested, this presents you with a deadlock situation: your patch won't be accepted until it is tested, but it won't be tested until it is accepted. Nevertheless, people who test mainline code are still relatively aggressive, given that many people and organizations do not test code until it has been pulled into a Linux distro.

And even if someone does test your patch, there is no guarantee that they will be running the hardware and software configuration and workload required to locate your bugs.

Therefore, even when writing code for an open-source project, you need to be prepared to develop and run your own test suite. Test development is an underappreciated and very valuable skill, so be sure to take full advantage of any existing test suites available to you. Important as test development is, we will leave further discussion of it to books dedicated to that topic. The following sections therefore discuss locating bugs in your code given that you already have a good test suite.

11.2 Tracing

When all else fails, add a `printf()`! Or a `printk()`, if you are working with user-mode C-language 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 `gdb` (for user applications) or `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, especially now that typical systems have more than 64K of memory and CPUs running faster than 4MHz. Much has been written about these tools, so this chapter will add little more.

However, these tools 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 going wrong, namely, they often have excessive overheads. There are special tracing technologies for this purpose, which typically leverage data ownership techniques (see Chapter 8) 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. Even so, enabling tracing can sometimes change timing enough to hide bugs, resulting in *heisenbugs*, which are discussed in Section 11.6 and especially Section 11.6.4.

Even if you avoid heisenbugs, other pitfalls await you. For example, although the machine really does know all, what it knows is almost always way more than your head can hold. For this reason, high-quality test suites normally come with sophisticated scripts to analyze the voluminous output. But beware—scripts won't necessarily notice surprising things. My `rcutorture` scripts are a case in point: Early versions of those scripts were quite satisfied with a test run in which RCU grace periods stalled indefinitely. This of course resulted in the scripts being modified to detect RCU grace-period stalls, but this does not change the fact that the scripts will only detect problems that I think to make them detect. The scripts are useful, but they are no substitute for occasional manual scans of the `rcutorture` output.

Another problem with tracing and especially with `printf()` calls is that their overhead is often too much for production use. In some such cases, assertions can be helpful.

11.3 Assertions

Assertions are usually implemented in the following manner:

```
1 if (something_bad_is_happening())
2 complain();
```

This pattern is often encapsulated into C-preprocessor macros or language intrinsics, for example, in the Linux kernel, this might be represented as `WARN_ON(something_bad_is_happening())`. Of course, if `something_bad_is_happening()` quite frequently, the resulting output might obscure reports of other problems, in which case `WARN_ONCE(something_bad_is_happening())` might be more appropriate.

Quick Quiz 11.4: How can you implement `WARN_ONCE()`? ■

In parallel code, one especially bad something that might happen is that a function expecting to be called under a particular lock might be called without that lock being held. Such functions sometimes have header comments stating something like “The caller must hold `foo_lock`”

`lock` when calling this function”, but such a comment does no good unless someone actually reads it. An executable statement like `lock_is_held(&foo_lock)` carries far more weight.

The Linux kernel’s lockdep facility [Cor06a, Ros11] takes this a step farther, reporting potential deadlocks as well as allowing functions to verify that the proper locks are held. Of course, this additional functionality incurs significant overhead, so that lockdep is not necessarily appropriate for production use.

So what can be done in cases where checking is necessary, but where the overhead of runtime checking cannot be tolerated? One approach is static analysis, which is discussed in the next section.

11.4 Static Analysis

Static analysis is a validation technique where one program takes a second program as input, reporting errors and vulnerabilities located in this second program. Interestingly enough, almost all programs are subjected to static analysis by their compilers or interpreters. These tools are of course far from perfect, but their ability to locate errors has improved immensely over the past few decades, in part because they now have much more than 64K bytes of memory in which to carry out their analysis.

The original UNIX `lint` tool [Joh77] was quite useful, though much of its functionality has since been incorporated into C compilers. There are nevertheless `lint`-like tools under development and in use to this day.

The sparse static analyzer [Cor04b] looks for higher-level issues in the Linux kernel, including:

1. Misuse of pointers to user-space structures.
2. Assignments from too-long constants.
3. Empty `switch` statements.
4. Mismatched lock acquisition and release primitives.
5. Misuse of per-CPU primitives.
6. Use of RCU primitives on non-RCU pointers and vice versa.

Although it is likely that compilers will continue to increase their static-analysis capabilities, the sparse static analyzer demonstrates the benefits of static analysis outside of the compiler, particularly for finding application-specific bugs.

11.5 Code Review

Various code-review activities are special cases of static analysis, but with human beings doing the analysis. This section covers inspection, walkthroughs, and self-inspection.

11.5.1 Inspection

Traditionally, formal code inspections take place in face-to-face meetings with formally defined roles: moderator, developer, and one or two other participants. The developer reads through the code, explaining what it is doing and why it works. The one or two other participants ask questions and raise issues, while the moderator’s job is to resolve any conflicts and to take notes. This process can be extremely effective at locating bugs, particularly if all of the participants are familiar with the code at hand.

However, this face-to-face formal procedure does not necessarily work well in the global Linux kernel community, although it might work well via an IRC session. Instead, individuals review code separately and provide comments via email or IRC. The note-taking is provided by email archives or IRC logs, and moderators volunteer their services as appropriate. Give or take the occasional flamewar, this process also works reasonably well, particularly if all of the participants are familiar with the code at hand.⁵

It is quite likely that the Linux kernel community’s review process is ripe for improvement:

1. There is sometimes a shortage of people with the time and expertise required to carry out an effective review.
2. Even though all review discussions are archived, they are often “lost” in the sense that insights are forgotten and people often fail to look up the discussions. This can result in re-insertion of the same old bugs.
3. It is sometimes difficult to resolve flamewars when they do break out, especially when the combatants have disjoint goals, experience, and vocabulary.

When reviewing, therefore, it is worthwhile to review relevant documentation in commit logs, bug reports, and LWN articles.

⁵ That said, one advantage of the Linux kernel community approach over traditional formal inspections is the greater probability of contributions from people *not* familiar with the code, who therefore might not be blinded by the invalid assumptions harbored by those familiar with the code.

11.5.2 Walkthroughs

A traditional code walkthrough is similar to a formal inspection, except that the group “plays computer” with the code, driven by specific test cases. A typical walkthrough team has a moderator, a secretary (who records bugs found), a testing expert (who generates the test cases) and perhaps one to two others. These can be extremely effective, albeit also extremely time-consuming.

It has been some decades since I have participated in a formal walkthrough, and I suspect that a present-day walkthrough would use single-stepping debuggers. One could imagine a particularly sadistic procedure as follows:

1. The tester presents the test case.
2. The moderator starts the code under a debugger, using the specified test case as input.
3. Before each statement is executed, the developer is required to predict the outcome of the statement and explain why this outcome is correct.
4. If the outcome differs from that predicted by the developer, this is taken as evidence of a potential bug.
5. In parallel code, a “concurrency shark” asks what code might execute concurrently with this code, and why such concurrency is harmless.

Sadistic, certainly. Effective? Maybe. If the participants have a good understanding of the requirements, software tools, data structures, and algorithms, then walkthroughs can be extremely effective. If not, walkthroughs are often a waste of time.

11.5.3 Self-Inspection

Although developers are usually not all that effective at inspecting their own code, there are a number of situations where there is no reasonable alternative. For example, the developer might be the only person authorized to look at the code, other qualified developers might all be too busy, or the code in question might be sufficiently bizarre that the developer is unable to convince anyone else to take it seriously until after demonstrating a prototype. In these cases, the following procedure can be quite helpful, especially for complex parallel code:

1. Write design document with requirements, diagrams for data structures, and rationale for design choices.

2. Consult with experts, update the design document as needed.
3. Write the code in pen on paper, correct errors as you go. Resist the temptation to refer to pre-existing nearly identical code sequences, instead, copy them.
4. If there were errors, copy the code in pen on fresh paper, correcting errors as you go. Repeat until the last two copies are identical.
5. Produce proofs of correctness for any non-obvious code.
6. Where possible, test the code fragments from the bottom up.
7. When all the code is integrated, do full-up functional and stress testing.
8. Once the code passes all tests, write code-level documentation, perhaps as an extension to the design document discussed above.

When I faithfully follow this procedure for new RCU code, there are normally only a few bugs left at the end. With a few prominent (and embarrassing) exceptions [McK11a], I usually manage to locate these bugs before others do. That said, this is getting more difficult over time as the number and variety of Linux-kernel users increases.

Quick Quiz 11.5: Why would anyone bother copying existing code in pen on paper??? Doesn’t that just increase the probability of transcription errors? ■

Quick Quiz 11.6: This procedure is ridiculously over-engineered! How can you expect to get a reasonable amount of software written doing it this way??? ■

The above procedure works well for new code, but what if you need to inspect code that you have already written? You can of course apply the above procedure for old code in the special case where you wrote one to throw away [FPB79], but the following approach can also be helpful in less desperate circumstances:

1. Using your favorite documentation tool (L^AT_EX, HTML, OpenOffice, or straight ASCII), describe the high-level design of the code in question. Use lots of diagrams to illustrate the data structures and how these structures are updated.
2. Make a copy of the code, stripping away all comments.

3. Document what the code does statement by statement.
4. Fix bugs as you find them.

This works because describing the code in detail is an excellent way to spot bugs [Mye79]. Although this second procedure is also a good way to get your head around someone else's code, in many cases, the first step suffices.

Although review and inspection by others is probably more efficient and effective, the above procedures can be quite helpful in cases where for whatever reason it is not feasible to involve others.

At this point, you might be wondering how to write parallel code without having to do all this boring paperwork. Here are some time-tested ways of accomplishing this:

1. Write a sequential program that scales through use of available parallel library functions.
2. Write sequential plug-ins for a parallel framework, such as map-reduce, BOINC, or a web-application server.
3. Do such a good job of parallel design that the problem is fully partitioned, then just implement sequential program(s) that run in parallel without communication.
4. Stick to one of the application areas (such as linear algebra) where tools can automatically decompose and parallelize the problem.
5. Make extremely disciplined use of parallel-programming primitives, so that the resulting code is easily seen to be correct. But beware: It is always tempting to break the rules "just a little bit" to gain better performance or scalability. Breaking the rules often results in general breakage. That is, unless you carefully do the paperwork described in this section.

But the sad fact is that even if you do the paperwork or use one of the above ways to more-or-less safely avoid paperwork, there will be bugs. If nothing else, more users and a greater variety of users will expose more bugs more quickly, especially if those users are doing things that the original developers did not consider. The next section describes how to handle the probabilistic bugs that occur all too commonly when validating parallel software.

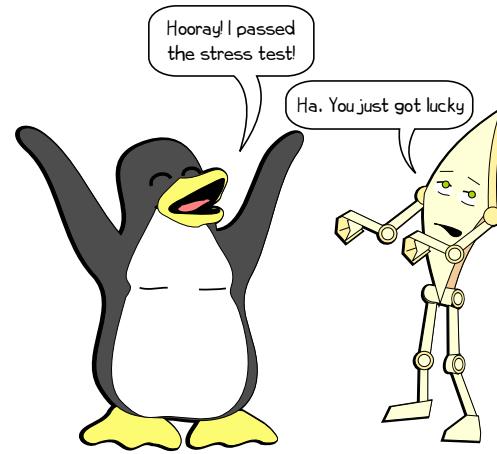


Figure 11.3: Passed on Merits? Or Dumb Luck?

11.6 Probability and Heisenbugs

So your parallel program fails. Sometimes.

But you used techniques from the earlier sections to locate the problem and now have a fix in place! Congratulations!!!

Now the question is just how much testing is required 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, having fixed only one of several related bugs on the other hand, or made some ineffectual unrelated change on yet a third hand. In short, what is the answer to the eternal question posed by Figure 11.3?

Unfortunately, the honest answer is that an infinite amount of testing is required to attain absolute certainty.

Quick Quiz 11.7: 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 please note that reading this section not a substitute for taking a good set of statistics classes.⁶

⁶ Which I most highly recommend. The few statistics courses I have

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 differ 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 therefore start with discrete tests.

11.6.1 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 probability 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

taken have provided value way out of proportion to the time I spent studying for them.

above series of steps, have at it! For those who like formulas, call the probability of a single failure f . The probability of a single success is then $1 - f$ and the probability that all of n tests will succeed is then:

$$S_n = (1 - f)^n \quad (11.1)$$

The probability of failure is $1 - S_n$, or:

$$F_n = 1 - (1 - f)^n \quad (11.2)$$

Quick Quiz 11.8: 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 11.2 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 11.2 can get tedious, so let's solve for n :

$$F_n = 1 - (1 - f)^n \quad (11.3)$$

$$1 - F_n = (1 - f)^n \quad (11.4)$$

$$\log(1 - F_n) = n \log(1 - f) \quad (11.5)$$

Finally the number of tests required is given by:

$$n = \frac{\log(1 - F_n)}{\log(1 - f)} \quad (11.6)$$

Plugging $f = 0.1$ and $F_n = 0.99$ into Equation 11.6 gives 43.7, meaning that we need 44 consecutive successful test runs to be 99% certain that our fix was a real improvement. This matches the number obtained by the previous method, which is reassuring.

Quick Quiz 11.9: In Equation 11.6, are the logarithms base-10, base-2, or base- e ? ■

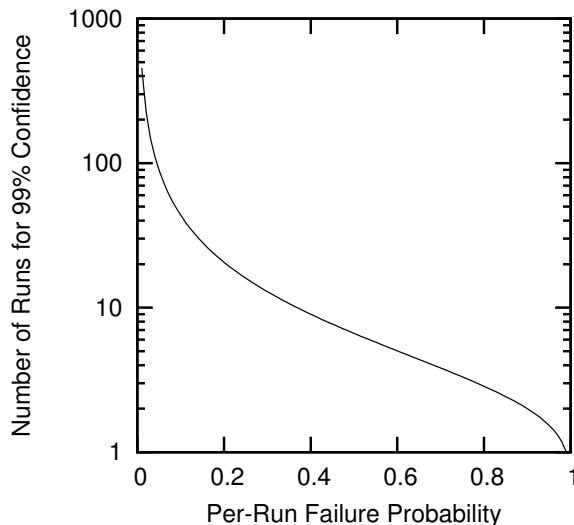


Figure 11.4: Number of Tests Required for 99 Percent Confidence Given Failure Rate

Figure 11.4 shows a plot of this function. Not surprisingly, 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. As the failure probability decreases, the number of test runs required increases, going to infinity as the failure probability goes to zero.

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 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 improvement”. Suppose you instead want to have 99% confidence that your fix reduced the failure rate by an order of magnitude. How many failure-free test runs are required?

An order of magnitude improvement from a 30% failure rate would be a 3% failure rate. Plugging these numbers into Equation 11.6 yields:

$$n = \frac{\log(1 - 0.99)}{\log(1 - 0.03)} = 151.2 \quad (11.7)$$

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 Section 11.6.4.

11.6.2 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 quite small compared to the errors inherent in your measurements of your program’s failure rates. Nevertheless, the next section describes a slightly less dodgy approach.

11.6.3 Statistics for Continuous Testing

This section contains more aggressive mathematics. If you are not in the mood for mathematical aggression, feel free to use the results of the previous section or to skip ahead to Section 11.6.3.2, possibly noting Equation 11.30 on page 190 for future reference.

11.6.3.1 Derivation of Poisson Distribution

As the number of tests n increases and the probability of per-test failure f decreases, it makes sense to move the mathematics to the continuous domain. It is convenient to define λ as nf : as we increase n and decrease f , λ will remain fixed. Intuitively, λ is the expected number of failures per unit time.

What then is the probability of all n tests succeeding? This is given by:

$$(1 - f)^n \quad (11.8)$$

But because λ is equal to nf , we can solve for f and obtain $f = \frac{\lambda}{n}$. Substituting this into the previous equation

yields:

$$\left(1 - \frac{\lambda}{n}\right)^n \quad (11.9)$$

Readers who are both alert and mathematically inclined will recognize this as approaching $e^{-\lambda}$ as n increases without limit. In other words, if we expect λ failures from a test of a given duration, the probability F_0 of zero failures from the test is given by:

$$F_0 = e^{-\lambda} \quad (11.10)$$

The next step is to compute the probability of all but one of n tests succeeding, which is:

$$\frac{n!}{1!(n-1)!} f(1-f)^{n-1} \quad (11.11)$$

The ratio of factorials accounts for the different permutations of test results. The f is the chance of the single failure, and the $(1-f)^{n-1}$ is the chance that the rest of the tests succeed. The $n!$ in the numerator allows for all permutations of n tests, while the two factors in the denominator allow for the indistinguishability of the one failure on the one hand and the $n-1$ successes on the other.

Cancelling the factorials and multiplying top and bottom by $1-f$ yields:

$$\frac{nf}{1-f}(1-f)^n \quad (11.12)$$

But because f is assumed to be arbitrarily small, $1-f$ is arbitrarily close to the value one, allowing us to dispense with the denominator:

$$nf(1-f)^n \quad (11.13)$$

Substituting $f = \frac{\lambda}{n}$ as before yields:

$$\lambda\left(1 - \frac{\lambda}{n}\right)^n \quad (11.14)$$

For large n , as before, the latter term is approximated by $e^{-\lambda}$, so that the probability of a single failure in a test from which λ failures were expected is given by:

$$F_1 = \lambda e^{-\lambda} \quad (11.15)$$

The third step is to compute the probability of all but two of the n tests succeeding, which is:

$$\frac{n!}{2!(n-2)!} f^2(1-f)^{n-2} \quad (11.16)$$

Cancelling the factorials and multiplying top and bottom by $(1-f)^2$ yields:

$$\frac{n(n-1)f^2}{2(1-f)^2}(1-f)^n \quad (11.17)$$

Once again, because f is assumed to be arbitrarily small, $(1-f)^2$ is arbitrarily close to the value one, once again allowing us to dispense with this portion of the denominator:

$$\frac{n(n-1)f^2}{2}(1-f)^n \quad (11.18)$$

Substituting $f = \frac{\lambda}{n}$ once again yields:

$$\frac{n(n-1)\lambda^2}{2n^2}\left(1 - \frac{\lambda}{n}\right)^n \quad (11.19)$$

Because n is assumed large, $n-1$ is arbitrarily close to n , allowing the $n(n-1)$ in the numerator to be cancelled with the n^2 in the denominator. And again, the final term is approximated by $e^{-\lambda}$, yielding the probability of two failures from a test from which λ failures were expected:

$$F_2 = \frac{\lambda^2}{2} e^{-\lambda} \quad (11.20)$$

We are now ready to try a more general result. Assume that there are m failures, where m is extremely small compared to n . Then we have:

$$\frac{n!}{m!(n-m)!} f^m(1-f)^{n-m} \quad (11.21)$$

Cancelling the factorials and multiplying top and bottom by $(1-f)^m$ yields:

$$\frac{n(n-1)\dots(n-m+2)(n-m+1)f^m}{m!(1-f)^m}(1-f)^n \quad (11.22)$$

And you guessed it, because f is arbitrarily small, $(1-f)^m$ is arbitrarily close to the value one and may therefore be dropped:

$$\frac{n(n-1)\dots(n-m+2)(n-m+1)f^m}{m!}(1-f)^n \quad (11.23)$$

Substituting $f = \frac{\lambda}{n}$ one more time:

$$\frac{n(n-1)\dots(n-m+2)(n-m+1)\lambda^m}{m!n^m}\left(1 - \frac{\lambda}{n}\right)^n \quad (11.24)$$

Because m is small compared to n , we can cancel all but the last of the factors in the numerator with the n^m in the denominator, resulting in:

$$\frac{\lambda^m}{m!} \left(1 - \frac{\lambda}{n}\right)^n \quad (11.25)$$

As always, for large n , the last term is approximated by $e^{-\lambda}$, yielding the probability of m failures from a test from which λ failures were expected:

$$F_m = \frac{\lambda^m}{m!} e^{-\lambda} \quad (11.26)$$

This is the celebrated Poisson distribution. A more rigorous derivation may be found in any advanced probability textbook, for example, Feller's classic "An Introduction to Probability Theory and Its Applications" [Fel50].

11.6.3.2 Use of Poisson Distribution

Let's try reworking the example from Section 11.6.2 using the Poisson distribution. Recall that this example involved a test with a 30% failure rate per hour, and that the question was how long the test would need to run on a alleged fix to be 99% certain that the fix actually reduced the failure rate. Solving this requires setting $e^{-\lambda}$ to 0.01 and solving for λ , resulting in:

$$\lambda = -\log 0.01 = 4.6 \quad (11.27)$$

Because we get 0.3 failures per hour, the number of hours required is $4.6/0.3 = 14.3$, which is within 10% of the 13 hours calculated using the method in Section 11.6.2. Given that you normally won't know your failure rate to within 10%, this indicates that the method in Section 11.6.2 is a good and sufficient substitute for the Poisson distribution in a great many situations.

More generally, if we have n failures per unit time, and we want to be $P\%$ certain that a fix reduced the failure rate, we can use the following formula:

$$T = -\frac{1}{n} \log \frac{100 - P}{100} \quad (11.28)$$

Quick Quiz 11.10: Suppose that a bug causes a test failure three times per hour on average. How long must the test run error-free to provide 99.9% confidence that the fix significantly reduced the probability of failure? ■

As before, the less frequently the bug occurs and the greater the required level of confidence, the longer the required error-free test run.

Suppose that a given test fails about once every hour, but after a bug fix, a 24-hour test run fails only twice.

Assuming that the failure leading to the bug is a random occurrence, what is the probability that the small number of failures in the second run was due to random chance? In other words, how confident should we be that the fix actually had some effect on the bug? This probability may be calculated by summing Equation 11.26 as follows:

$$F_0 + F_1 + \dots + F_{m-1} + F_m = \sum_{i=0}^m \frac{\lambda^i}{i!} e^{-\lambda} \quad (11.29)$$

This is the Poisson cumulative distribution function, which can be written more compactly as:

$$F_{i \leq m} = \sum_{i=0}^m \frac{\lambda^i}{i!} e^{-\lambda} \quad (11.30)$$

Here m is the number of errors in the long test run (in this case, two) and λ is expected number of errors in the long test run (in this case, 24). Plugging $m = 2$ and $\lambda = 24$ into this expression gives the probability of two or fewer failures as about 1.2×10^{-8} , in other words, we have a high level of confidence that the fix actually had some relationship to the bug.⁷

Quick Quiz 11.11: Doing the summation of all the factorials and exponentials is a real pain. Isn't there an easier way? ■

Quick Quiz 11.12: But wait!!! Given that there has to be *some* number of failures (including the possibility of zero failures), shouldn't the summation shown in Equation 11.30 approach the value 1 as m goes to infinity? ■

The Poisson distribution is a powerful tool for analyzing test results, but the fact is that in this last example there were still two remaining test failures in a 24-hour test run. Such a low failure rate results in very long test runs. The next section discusses counter-intuitive ways of improving this situation.

11.6.4 Hunting Heisenbugs

This line of thought also helps explain 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.

The name "heisenbug" stems from the Heisenberg Uncertainty Principle from quantum physics, which states

⁷ Of course, this result in no way excuses you from finding and fixing the bug(s) resulting in the remaining two failures!

that it is impossible to exactly quantify a given particle's position and velocity at any given point in time [Hei27]. Any attempt to more accurately measure that particle's position will result in increased uncertainty of its velocity. A similar effect occurs for heisenbugs: attempts to track down the heisenbug causes it to radically change its symptoms or even disappear completely.

If the field of physics inspired the name of this problem, it is only logical that we should look to the field of physics for the solution. Fortunately, particle physics is up to the task: Why not create an anti-heisenbug to annihilate the heisenbug?

This section describes a number of ways to do just that:

1. Add delay to race-prone regions.
2. Increase workload intensity.
3. Test suspicious subsystems in isolation.
4. Simulate unusual events.
5. Count near misses.

Although producing an anti-heisenbug for a given heisenbug is more an art than a science, the following sections give some tips on generating the corresponding species of anti-heisenbug.

11.6.4.1 Add Delay

Consider the count-lossy code in Section 5.1. Adding `printf()` statements will likely greatly reduce or even eliminate the lost counts. However, converting the load-add-store sequence to a load-add-delay-store sequence will greatly increase the incidence of lost counts (try it!). Once you spot a bug involving a race condition, it is frequently possible to create an anti-heisenbug by adding delay in this manner.

Of course, this begs the question of how to find the race condition in the first place. This is a bit of a dark art, but there are a number of things you can do to find them.

One approach is to recognize that race conditions often end up corrupting some of the data involved in the race. It is therefore good practice to double-check the synchronization of any corrupted data. Even if you cannot immediately recognize the race condition, adding delay before and after accesses to the corrupted data might change the failure rate. By adding and removing the delays in an organized fashion (e.g., binary search), you might learn more about the workings of the race condition.

Quick Quiz 11.13: How is this approach supposed to help if the corruption affected some unrelated pointer, which then caused the corruption??? ■

Another important approach is to vary the software and hardware configuration and look for statistically significant differences in failure rate. You can then look more intensively at the code implicated by the software or hardware configuration changes that make the greatest difference in failure rate. It might be helpful to test that code in isolation, for example.

One important aspect of software configuration is the history of changes, which is why `git bisect` is so useful. Bisection of the change history can provide very valuable clues as to the nature of the heisenbug.

Quick Quiz 11.14: But I did the bisection, and ended up with a huge commit. What do I do now? ■

However you locate the suspicious section of code, you can then introduce delays to attempt to increase the probability of failure. As we have seen, increasing the probability of failure makes it much easier to gain high confidence in the corresponding fix.

However, it is sometimes quite difficult to track down the problem using normal debugging techniques. The following sections present some other alternatives.

11.6.4.2 Increase Workload Intensity

It is often the case that a given test suite places relatively low stress on a given subsystem, so that a small change in timing can cause a heisenbug to disappear. One way to create an anti-heisenbug for this case is to increase the workload intensity, which has a good chance of increasing the probability of the bug appearing. If the probability is increased sufficiently, it may be possible to add lightweight diagnostics such as tracing without causing the bug to vanish.

How can you increase the workload intensity? This depends on the program, but here are some things to try:

1. Add more CPUs.
2. If the program uses networking, add more network adapters and more or faster remote systems.
3. If the program is doing heavy I/O when the problem occurs, either (1) add more storage devices, (2) use faster storage devices, for example, substitute SSDs for disks, or (3) use a RAM-based filesystem to substitute main memory for mass storage.

4. Change the size of the problem, for example, if doing a parallel matrix multiply, change the size of the matrix. Larger problems may introduce more complexity, but smaller problems often increase the level of contention. If you aren't sure whether you should go large or go small, just try both.

However, it is often the case that the bug is in a specific subsystem, and the structure of the program limits the amount of stress that can be applied to that subsystem. The next section addresses this situation.

11.6.4.3 Isolate Suspicious Subsystems

If the program is structured such that it is difficult or impossible to apply much stress to a subsystem that is under suspicion, a useful anti-heisenbug is a stress test that tests that subsystem in isolation. The Linux kernel's `rcutorture` module takes exactly this approach with RCU: By applying more stress to RCU than is feasible in a production environment, the probability that any RCU bugs will be found during `rcutorture` testing rather than during production use is increased.⁸

In fact, when creating a parallel program, it is wise to stress-test the components separately. Creating such component-level stress tests can seem like a waste of time, but a little bit of component-level testing can save a huge amount of system-level debugging.

11.6.4.4 Simulate Unusual Events

Heisenbugs are sometimes due to unusual events, such as memory-allocation failure, conditional-lock-acquisition failure, CPU-hotplug operations, timeouts, packet losses, and so on. One way to construct an anti-heisenbug for this class of heisenbug is to introduce spurious failures.

For example, instead of invoking `malloc()` directly, invoke a wrapper function that uses a random number to decide whether to return `NULL` unconditionally on the one hand, or to actually invoke `malloc()` and return the resulting pointer on the other. Inducing spurious failures is an excellent way to bake robustness into sequential programs as well as parallel programs.

Quick Quiz 11.15: Why don't existing conditional-locking primitives provide this spurious-failure functionality? ■

⁸ Though sadly not increased to probability one.

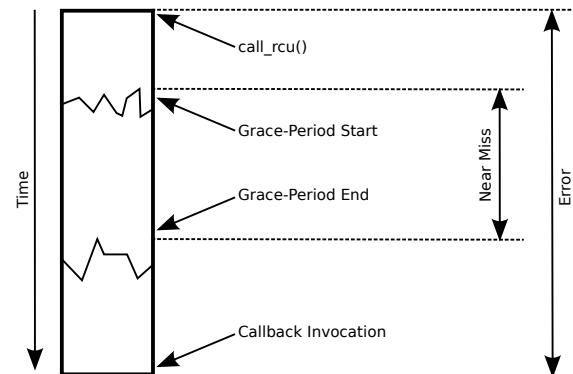


Figure 11.5: RCU Errors and Near Misses

11.6.4.5 Count Near Misses

Bugs are often an all-or-nothing thing, so that either the bug happens or it doesn't, with nothing in between. However, it is sometimes possible to define a *near miss* where the bug does not result in a failure, but has likely manifested. For example, suppose your code is making a robot walk. The robot's falling over constitutes a bug in your program, but stumbling and recovering might constitute a near miss. If the robot falls over only once per hour, but stumbles every few minutes, you might be able to speed up your debugging progress by counting the number of stumbles in addition to the number of falls.

In concurrent programs, timestamping can sometimes be used to detect near misses. For example, locking primitives incur significant delays, so if there is a too-short delay between a pair of operations that are supposed to be protected by different acquisitions of the same lock, this too-short delay might be counted as a near miss.⁹

For example, a low-probability bug in RCU priority boosting occurred roughly once every hundred hours of focused `rcutorture` testing. Because it would take almost 500 hours of failure-free testing to be 99% certain that the bug's probability had been significantly reduced, the `git bisect` process to find the failure would be painfully slow—or would require an extremely large test farm. Fortunately, the RCU operation being tested included not only a wait for an RCU grace period, but also a previous wait for the grace period to start and a subsequent wait for an RCU callback to be invoked after

⁹ Of course, in this case, you might be better off using whatever `lock_held()` primitive is available in your environment. If there isn't a `lock_held()` primitive, create one!

completion of the RCU grace period. This distinction between an `rcutorture` error and near miss is shown in Figure 11.5. To qualify as a full-fledged error, an RCU read-side critical section must extend from the `call_rcu()` that initiated a grace period, through the remainder of the previous grace period, through the entirety of the grace period initiated by the `call_rcu()` (denoted by the region between the jagged lines), and through the delay from the end of that grace period to the callback invocation, as indicated by the “Error” arrow. However, the formal definition of RCU prohibits RCU read-side critical sections from extending across a single grace period, as indicated by the “Near Miss” arrow. This suggests using near misses as the error condition, however, this can be problematic because different CPUs can have different opinions as to exactly where a given grace period starts and ends, as indicated by the jagged lines.¹⁰ Using the near misses as the error condition could therefore result in false positives, which need to be avoided in the automated `rcutorture` testing.

By sheer dumb luck, `rcutorture` happens to include some statistics that are sensitive to the near-miss version of the grace period. As noted above, these statistics are subject to false positives due to their unsynchronized access to RCU’s state variables, but these false positives turn out to be extremely rare on strongly ordered systems such as the IBM mainframe and x86, occurring less than once per thousand hours of testing.

These near misses occurred roughly once per hour, about two orders of magnitude more frequently than the actual errors. Use of these near misses allowed the bug’s root cause to be identified in less than a week and a high degree of confidence in the fix to be built in less than a day. In contrast, excluding the near misses in favor of the real errors would have required months of debug and validation time.

To sum up near-miss counting, the general approach is to replace counting of infrequent failures with more-frequent near misses that are believed to be correlated with those failures. These near-misses can be considered an anti-heisenbug to the real failure’s heisenbug because the near-misses, being more frequent, are likely to be more robust in the face of changes to your code, for example, the changes you make to add debugging code.

Thus far, we have focused solely on bugs in the parallel program’s functionality. However, because performance is

a first-class requirement for a parallel program (otherwise, why not write a sequential program?), the next section looks into finding performance bugs.

11.7 Performance Estimation

Parallel programs usually have performance and scalability requirements, after all, if performance is not an issue, why not use a sequential program? Ultimate performance and linear scalability might not be necessary, but there is little use for a parallel program that runs slower than its optimal sequential counterpart. And there really are cases where every microsecond matters and every nanosecond is needed. Therefore, for parallel programs, insufficient performance is just as much a bug as is incorrectness.

Quick Quiz 11.16: That is ridiculous!!! After all, isn’t getting the correct answer later than one would like better than getting an incorrect answer??? ■

Quick Quiz 11.17: But if you are going to put in all the hard work of parallelizing an application, why not do it right? Why settle for anything less than optimal performance and linear scalability? ■

Validating a parallel program must therefore include validating its performance. But validating performance means having a workload to run and performance criteria with which to evaluate the program at hand. These needs are often met by *performance benchmarks*, which are discussed in the next section.

11.7.1 Benchmarking

The old saying goes “There are lies, damn lies, statistics, and benchmarks.” However, benchmarks are heavily used, so it is not helpful to be too dismissive of them.

Benchmarks span the range from ad hoc test jigs to international standards, but regardless of their level of formality, benchmarks serve four major purposes:

1. Providing a fair framework for comparing competing implementations.
2. Focusing competitive energy on improving implementations in ways that matter to users.
3. Serving as example uses of the implementations being benchmarked.
4. Serving as a marketing tool to highlight your software’s strong points against your competitors’ offerings.

¹⁰ The jaggedness of these lines is seriously understated because idle CPUs might well be completely unaware of the most recent few hundred grace periods.

Of course, the only completely fair framework is the intended application itself. So why would anyone who cared about fairness in benchmarking bother creating imperfect benchmarks rather than simply using the application itself as the benchmark?

Running the actual application is in fact the best approach where it is practical. Unfortunately, it is often impractical for the following reasons:

1. The application might be proprietary, and you might not have the right to run the intended application.
2. The application might require more hardware that you have access to.
3. The application might use data that you cannot legally access, for example, due to privacy regulations.

In these cases, creating a benchmark that approximates the application can help overcome these obstacles. A carefully constructed benchmark can help promote performance, scalability, energy efficiency, and much else besides.

11.7.2 Profiling

In many cases, a fairly small portion of your software is responsible for the majority of the performance and scalability shortfall. However, developers are notoriously unable to identify the actual bottlenecks by hand. For example, in the case of a kernel buffer allocator, all attention focused on a search of a dense array which turned out to represent only a few percent of the allocator's execution time. An execution profile collected via a logic analyzer focused attention on the cache misses that were actually responsible for the majority of the problem [MS93].

An old-school but quite effective method of tracking down performance and scalability bugs is to run your programmer under a debugger, then periodically interrupt it, recording the stacks of all threads at each interruption. The theory here is that if something is slowing down your program, it has to be visible in your threads' executions.

That said, there are a number of tools that will usually do a much better job of helping you to focus your attention where it will do the most good. Two popular choices are `gprof` and `perf`. To use `perf` on a single-process program, prefix your command with `perf record`, then after the command completes, type `perf report`. There is a lot of work on tools for performance debugging of multi-threaded programs, which should make this important job easier.

11.7.3 Differential Profiling

Scalability problems will not necessarily be apparent unless you are running on very large systems. However, it is sometimes possible to detect impending scalability problems even when running on much smaller systems. One technique for doing this is called *differential profiling*.

The idea is to run your workload under two different sets of conditions. For example, you might run it on two CPUs, then run it again on four CPUs. You might instead vary the load placed on the system, the number of network adapters, the number of mass-storage devices, and so on. You then collect profiles of the two runs, and mathematically combine corresponding profile measurements. For example, if your main concern is scalability, you might take the ratio of corresponding measurements, and then sort the ratios into descending numerical order. The prime scalability suspects will then be sorted to the top of the list [McK95, McK99].

Some tools such as `perf` have built-in differential-profiling support.

11.7.4 Microbenchmarking

Microbenchmarking can be useful when deciding which algorithms or data structures are worth incorporating into a larger body of software for deeper evaluation.

One common approach to microbenchmarking is to measure the time, run some number of iterations of the code under test, then measure the time again. The difference between the two times divided by the number of iterations gives the measured time required to execute the code under test.

Unfortunately, this approach to measurement allows any number of errors to creep in, including:

1. The measurement will include some of the overhead of the time measurement. This source of error can be reduced to an arbitrarily small value by increasing the number of iterations.
2. The first few iterations of the test might incur cache misses or (worse yet) page faults that might inflate the measured value. This source of error can also be reduced by increasing the number of iterations, or it can often be eliminated entirely by running a few warm-up iterations before starting the measurement period.
3. Some types of interference, for example, random memory errors, are so rare that they can be dealt

with by running a number of sets of iterations of the test. If the level of interference was statistically significant, any performance outliers could be rejected statistically.

4. Any iteration of the test might be interfered with by other activity on the system. Sources of interference include other applications, system utilities and daemons, device interrupts, firmware interrupts (including system management interrupts, or SMIs), virtualization, memory errors, and much else besides. Assuming that these sources of interference occur randomly, their effect can be minimized by reducing the number of iterations.

The first and fourth sources of interference provide conflicting advice, which is one sign that we are living in the real world. The remainder of this section looks at ways of resolving this conflict.

Quick Quiz 11.18: But what about other sources of error, for example, due to interactions between caches and memory layout? ■

The following sections discuss ways of dealing with these measurement errors, with Section 11.7.5 covering isolation techniques that may be used to prevent some forms of interference, and with Section 11.7.6 covering methods for detecting interference so as to reject measurement data that might have been corrupted by that interference.

11.7.5 Isolation

The Linux kernel provides a number of ways to isolate a group of CPUs from outside interference.

First, let's look at interference by other processes, threads, and tasks. The POSIX `sched_setaffinity()` system call may be used to move most tasks off of a given set of CPUs and to confine your tests to that same group. The Linux-specific user-level `taskset` command may be used for the same purpose, though both `sched_setaffinity()` and `taskset` require elevated permissions. Linux-specific control groups (`cgroups`) may be used for this same purpose. This approach can be quite effective at reducing interference, and is sufficient in many cases. However, it does have limitations, for example, it cannot do anything about the per-CPU kernel threads that are often used for housekeeping tasks.

One way to avoid interference from per-CPU kernel threads is to run your test at a high real-time

priority, for example, by using the POSIX `sched_setscheduler()` system call. However, note that if you do this, you are implicitly taking on responsibility for avoiding infinite loops, because otherwise your test will prevent part of the kernel from functioning.¹¹

These approaches can greatly reduce, and perhaps even eliminate, interference from processes, threads, and tasks. However, it does nothing to prevent interference from device interrupts, at least in the absence of threaded interrupts. Linux allows some control of threaded interrupts via the `/proc/irq` directory, which contains numerical directories, one per interrupt vector. Each numerical directory contains `smp_affinity` and `smp_affinity_list`. Given sufficient permissions, you can write a value to these files to restrict interrupts to the specified set of CPUs. For example, “`sudo echo 3 > /proc/irq/23/smp_affinity`” would confine interrupts on vector 23 to CPUs 0 and 1. The same results may be obtained via “`sudo echo 0-1 > /proc/irq/23/smp_affinity_list`”. You can use “`cat /proc/interrupts`” to obtain a list of the interrupt vectors on your system, how many are handled by each CPU, and what devices use each interrupt vector.

Running a similar command for all interrupt vectors on your system would confine interrupts to CPUs 0 and 1, leaving the remaining CPUs free of interference. Or mostly free of interference, anyway. It turns out that the scheduling-clock interrupt fires on each CPU that is running in user mode.¹² In addition you must take care to ensure that the set of CPUs that you confine the interrupts to is capable of handling the load.

But this only handles processes and interrupts running in the same operating-system instance as the test. Suppose that you are running the test in a guest OS that is itself running on a hypervisor, for example, Linux running KVM. Although you can in theory apply the same techniques at the hypervisor level that you can at the guest-OS level, it is quite common for hypervisor-level operations to be restricted to authorized personnel. In addition, none of these techniques work against firmware-level interference.

Quick Quiz 11.19: Wouldn't the techniques suggested to isolate the code under test also affect that code's performance, particularly if it is running within a larger ap-

¹¹ This is an example of the Spiderman Principle: “With great power comes great responsibility.”

¹² Frederic Weisbecker is working on an adaptive-ticks project that will allow the scheduling-clock interrupt to be shut off on any CPU that has only one runnable task, but as of early 2013, this is unfortunately still work in progress.

```

1 #include <sys/time.h>
2 #include <sys/resource.h>
3
4 /* Return 0 if test results should be rejected. */
5 int runtest(void)
6 {
7     struct rusage rul;
8     struct rusage ru2;
9
10    if (getrusage(RUSAGE_SELF, &rul) != 0) {
11        perror("getrusage");
12        abort();
13    }
14    /* run test here. */
15    if (getrusage(RUSAGE_SELF, &ru2 != 0) {
16        perror("getrusage");
17        abort();
18    }
19    return (rul.ru_nvcsw == ru2.ru_nvcsw &
20            rul.runivcsw == ru2.runivcsw);
21 }

```

Figure 11.6: Using `getrusage()` to Detect Context Switches

plication? ■

If you find yourself in this painful situation, instead of preventing the interference, you might need to detect the interference as described in the next section.

11.7.6 Detecting Interference

If you cannot prevent interference, perhaps you can detect the interference after the fact and reject the test runs that were affected by that interference. Section 11.7.6.1 describes methods of rejection involving additional measurements, while Section 11.7.6.2 describes statistics-based rejection.

11.7.6.1 Detecting Interference Via Measurement

Many systems, including Linux, provide means for determining after the fact whether some forms of interference have occurred. For example, if your test encountered process-based interference, a context switch must have occurred during the test. On Linux-based systems, this context switch will be visible in `/proc/<PID>/sched` in the `nr_switches` field. Similarly, interrupt-based interference can be detected via the `/proc/interrupts` file.

Opening and reading files is not the way to low overhead, and it is possible to get the count of context switches for a given thread by using the `getrusage()` system call, as shown in Figure 11.6. This same system call can be used to detect minor page faults (`ru_minflt`) and major page faults (`ru_majflt`).

Unfortunately, detecting memory errors and firmware interference is quite system-specific, as is the detection of interference due to virtualization. Although avoidance is better than detection, and detection is better than statistics, there are times when one must avail oneself of statistics, a topic addressed in the next section.

11.7.6.2 Detecting Interference Via Statistics

Any statistical analysis will be based on assumptions about the data, and performance microbenchmarks often support the following assumptions:

1. Smaller measurements are more likely to be accurate than larger measurements.
2. The measurement uncertainty of good data is known.
3. A reasonable fraction of the test runs will result in good data.

The fact that smaller measurements are more likely to be accurate than larger measurements suggests that sorting the measurements in increasing order is likely to be productive.¹³ The fact that the measurement uncertainty is known allows us to accept measurements within this uncertainty of each other. If the effects of interference are large compared to this uncertainty, this will ease rejection of bad data. Finally, the fact that some fraction (for example, one third) can be assumed to be good allows us to blindly accept the first portion of the sorted list, and this data can then be used to gain an estimate of the natural variation of the measured data, over and above the assumed measurement error.

The approach is to take the specified number of leading elements from the beginning of the sorted list, and use these to estimate a typical inter-element delta, which in turn may be multiplied by the number of elements in the list to obtain an upper bound on permissible values. The algorithm then repeatedly considers the next element of the list. If it falls below the upper bound, and if the distance between the next element and the previous element is not too much greater than the average inter-element distance for the portion of the list accepted thus far, then the next element is accepted and the process repeats. Otherwise, the remainder of the list is rejected.

Figure 11.7 shows a simple `sh/awk` script implementing this notion. Input consists of an `x`-value followed by an arbitrarily long list of `y`-values, and output consists of one line for each input line, with fields as follows:

¹³ To paraphrase the old saying, “Sort first and ask questions later.”

1. The x-value.
2. The average of the selected data.
3. The minimum of the selected data.
4. The maximum of the selected data.
5. The number of selected data items.
6. The number of input data items.

This script takes three optional arguments as follows:

- `--divisor`: Number of segments to divide the list into, for example, a divisor of four means that the first quarter of the data elements will be assumed to be good. This defaults to three.
- `--relerr`: Relative measurement error. The script assumes that values that differ by less than this error are for all intents and purposes equal. This defaults to 0.01, which is equivalent to 1%.
- `--trendbreak`: Ratio of inter-element spacing constituting a break in the trend of the data. For example, if the average spacing in the data accepted so far is 1.5, then if the trend-break ratio is 2.0, then if the next data value differs from the last one by more than 3.0, this constitutes a break in the trend. (Unless of course, the relative error is greater than 3.0, in which case the “break” will be ignored.)

Lines 1-3 of Figure 11.7 set the default values for the parameters, and lines 4-21 parse any command-line overriding of these parameters. The `awk` invocation on lines 23 and 24 sets the values of the `divisor`, `relerr`, and `trendbreak` variables to their `sh` counterparts. In the usual `awk` manner, lines 25-52 are executed on each input line. The loop spanning lines 24 and 26 copies the input `y`-values to the `d` array, which line 27 sorts into increasing order. Line 28 computes the number of `y`-values that are to be trusted absolutely by applying `divisor` and rounding up.

Lines 29-33 compute the `maxdelta` value used as a lower bound on the upper bound of `y`-values. To this end, lines 29 and 30 multiply the difference in values over the trusted region of data by the `divisor`, which projects the difference in values across the trusted region across the entire set of `y`-values. However, this value might well be much smaller than the relative error, so line 31 computes the absolute error (`d[i] * relerr`) and adds that to the difference `delta` across the trusted portion of the

data. Lines 32 and 33 then compute the maximum of these two values.

Each pass through the loop spanning lines 34-43 attempts to add another data value to the set of good data. Lines 35-39 compute the trend-break delta, with line 36 disabling this limit if we don’t yet have enough values to compute a trend, and with lines 38 and 39 multiplying `trendbreak` by the average difference between pairs of data values in the good set. If line 40 determines that the candidate data value would exceed the lower bound on the upper bound (`maxdelta`) and line 41 determines that the difference between the candidate data value and its predecessor exceeds the trend-break difference (`maxdiff`), then line 42 exits the loop: We have the full good set of data.

Lines 44-52 then compute and print the statistics for the data set.

Quick Quiz 11.20: This approach is just plain weird! Why not use means and standard deviations, like we were taught in our statistics classes? ■

Quick Quiz 11.21: But what if all the `y`-values in the trusted group of data are exactly zero? Won’t that cause the script to reject any non-zero value? ■

Although statistical interference detection can be quite useful, it should be used only as a last resort. It is far better to avoid interference in the first place (Section 11.7.5), or, failing that, detecting interference via measurement (Section 11.7.6.1).

11.8 Summary

Although validation never will be an exact science, much can be gained by taking an organized approach to it, as an organized approach will help you choose the right validation tools for your job, avoiding situations like the one fancifully depicted in Figure 11.8.

A key choice is that of statistics. Although the methods described in this chapter work very well most of the time, they do have their limitations. These limitations are inherent because we are attempting to do something that is in general impossible, courtesy of the Halting Problem [Tur37, Pul00]. Fortunately for us, there are a huge number of special cases in which we can not only work out whether a given program will halt, but also establish estimates for how long it will run before halting, as discussed in Section 11.7. Furthermore, in cases where a given program might or might not work correctly, we can often establish estimates for what fraction of the time it will work correctly, as discussed in Section 11.6.

Nevertheless, unthinking reliance on these estimates is brave to the point of foolhardiness. After all, we are summarizing a huge mass of complexity in code and data structures down to a single solitary number. Even though we can get away with such bravery a surprisingly large fraction of the time, it is only reasonable to expect that the code and data being abstracted away will occasionally cause severe problems.

One possible problem is variability, where repeated runs might give wildly different results. This is often dealt with by maintaining a standard deviation as well as a mean, but the fact is that attempting to summarize the behavior of a large and complex program with two numbers is almost as brave as summarizing its behavior with only one number. In computer programming, the surprising thing is that use of the mean or the mean and standard deviation are often sufficient, but there are no guarantees.

One cause of variation is confounding factors. For example, the CPU time consumed by a linked-list search will depend on the length of the list. Averaging together runs with wildly different list lengths will probably not be useful, and adding a standard deviation to the mean will not be much better. The right thing to do would be control for list length, either by holding the length constant or to measure CPU time as a function of list length.

Of course, this advice assumes that you are aware of the confounding factors, and Murphy says that you probably will not be. I have been involved in projects that had confounding factors as diverse as air conditioners (which drew considerable power at startup, thus causing the voltage supplied to the computer to momentarily drop too low, sometimes resulting in failure), cache state (resulting in odd variations in performance), I/O errors (including disk errors, packet loss, and duplicate Ethernet MAC addresses), and even porpoises (which could not resist playing with an array of transponders, which, in the absence of porpoises, could be used for high-precision acoustic positioning and navigation).

In short, validation always will require some measure of the behavior of the system. Because this measure must be a severe summarization of the system, it can be misleading. So as the saying goes, “Be careful. It is a real world out there.”

But suppose you are working on the Linux kernel, which as of 2013 has about a billion instances throughout the world? In that case, a bug that would be encountered once every million years will be encountered almost three times per day across the installed base. A test with a 50%

chance of encountering this bug in a one-hour run would need to increase that bug’s probability of occurrence by more than nine orders of magnitude, which poses a severe challenge to today’s testing methodologies. One important tool that can sometimes be applied with good effect to such situations is formal verification, the subject of the next chapter.

```

1 divisor=3
2 relerr=0.01
3 trendbreak=10
4 while test $# -gt 0
5 do
6   case "$1" in
7     --divisor)
8       shift
9       divisor=$1
10    ;;
11   --relerr)
12    shift
13   relerr=$1
14    ;;
15   --trendbreak)
16    shift
17   trendbreak=$1
18    ;;
19  esac
20  shift
21 done
22
23 awk -v divisor=$divisor -v relerr=$relerr \
24   -v trendbreak=$trendbreak '(
25   for (i = 2; i <= NF; i++)
26     d[i - 1] = $i;
27   asort(d);
28   i = int((NF + divisor - 1) / divisor);
29   delta = d[i] - d[1];
30   maxdelta = delta * divisor;
31   maxdeltal = delta + d[i] * relerr;
32   if (maxdeltal > maxdelta)
33     maxdelta = maxdeltal;
34   for (j = i + 1; j < NF; j++) {
35     if (j <= 2)
36       maxdiff = d[NF - 1] - d[1];
37     else
38       maxdiff = trendbreak * \
39         (d[j - 1] - d[1]) / (j - 2);
40     if (d[j] - d[1] > maxdelta && \
41         d[j] - d[j - 1] > maxdiff)
42       break;
43   }
44   n = sum = 0;
45   for (k = 1; k < j; k++) {
46     sum += d[k];
47     n++;
48   }
49   min = d[1];
50   max = d[j - 1];
51   avg = sum / n;
52   print $1, avg, min, max, n, NF - 1;
53 )'

```

Figure 11.7: Statistical Elimination of Interference



Figure 11.8: Choose Validation Methods Wisely

Chapter 12

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. In addition, a proof of correctness cannot be expected to find errors in your assumptions, shortcomings in the requirements, misunderstandings of the underlying software or hardware primitives, or errors that you did not think to construct a proof for. This means that formal methods can never replace testing, however, formal methods are nevertheless a valuable addition to your validation toolbox.

It would be very helpful to have a tool that could somehow locate all race conditions. A number of such tools exist, for example, Section 12.1 provides an introduction to the general-purpose state-space search tools Promela and Spin, Section 12.2 similarly introduces the special-purpose ppcmem and cppmem tools, Section 12.3 looks at an example axiomatic approach, Section 12.4 briefly overviews SAT solvers, and finally Section 12.5 sums up use of formal-verification tools for verifying parallel algorithms.

12.1 General-Purpose State-Space Search

This section features the general-purpose Promela and spin tools, which may be used to carry out a full state-space search of many types of multi-threaded code. They are also quite useful for verifying data communication protocols. Section 12.1.1 introduces Promela and spin, including a couple of warm-up exercises verifying both non-atomic and atomic increment. Section 12.1.2 describes use of Promela, including example command lines and a

comparison of Promela syntax to that of C. Section 12.1.3 shows how Promela may be used to verify locking, 12.1.4 uses Promela to verify an unusual implementation of RCU named “QRCU,” and finally Section 12.1.5 applies Promela to RCU’s dyntick-idle implementation.

12.1.1 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 be 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.

```

1 #define NUMPROCS 2
2
3 byte counter = 0;
4 byte progress[NUMPROCS];
5
6 proctype incrementer(byte me)
7 {
8     int temp;
9
10    temp = counter;
11    counter = temp + 1;
12    progress[me] = 1;
13 }
14
15 init {
16     int i = 0;
17     int sum = 0;
18
19     atomic {
20         i = 0;
21         do
22             :: i < NUMPROCS ->
23                 progress[i] = 0;
24                 run incrementer(i);
25                 i++
26             :: i >= NUMPROCS -> break
27             od;
28     }
29     atomic {
30         i = 0;
31         sum = 0;
32         do
33             :: i < NUMPROCS ->
34                 sum = sum + progress[i];
35                 i++
36             :: i >= NUMPROCS -> break
37             od;
38         assert(sum < NUMPROCS || counter == NUMPROCS)
39     }
40 }

```

Figure 12.1: Promela Code for Non-Atomic Increment

12.1.1.1 Promela Warm-Up: Non-Atomic Increment

Figure 12.1 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 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 lose 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()` statement 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 12.2. 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 12.3. As can be seen, the first portion of the init block created both incrementer processes, both of which first fetched the counter, then both incremented and stored it, losing a count. The assertion then triggered, after which the global state is displayed.

```

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 12.2: 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))]
2: proc 0 (:init:) line 23 "increment.spin" (state 3) [progress[i] = 0]
Starting incrementer with pid 1
3: proc 0 (:init:) line 24 "increment.spin" (state 4) [(run incrementer(i))]
3: proc 0 (:init:) line 25 "increment.spin" (state 5) [i = (i+1)]
4: proc 0 (:init:) line 22 "increment.spin" (state 2) [((i<2))]
4: proc 0 (:init:) line 23 "increment.spin" (state 3) [progress[i] = 0]
Starting incrementer with pid 2
5: proc 0 (:init:) line 24 "increment.spin" (state 4) [(run incrementer(i))]
5: proc 0 (:init:) line 25 "increment.spin" (state 5) [i = (i+1)]
6: proc 0 (:init:) line 26 "increment.spin" (state 6) [((i>=2))]
7: proc 0 (:init:) line 21 "increment.spin" (state 10) [break]
8: proc 2 (incrementer) line 10 "increment.spin" (state 1) [temp = counter]
9: proc 1 (incrementer) line 10 "increment.spin" (state 1) [temp = counter]
10: proc 2 (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]
16: proc 0 (:init:) line 31 "increment.spin" (state 13) [sum = 0]
17: proc 0 (:init:) line 33 "increment.spin" (state 14) [((i<2))]
17: proc 0 (:init:) line 34 "increment.spin" (state 15) [sum = (sum+progress[i])]
17: proc 0 (:init:) line 35 "increment.spin" (state 16) [i = (i+1)]
18: proc 0 (:init:) line 33 "increment.spin" (state 14) [((i<2))]
18: proc 0 (:init:) line 34 "increment.spin" (state 15) [sum = (sum+progress[i])]
18: proc 0 (:init:) line 35 "increment.spin" (state 16) [i = (i+1)]
19: proc 0 (:init:) line 36 "increment.spin" (state 17) [((i>=2))]
20: proc 0 (:init:) line 32 "increment.spin" (state 21) [break]
spin: line 38 "increment.spin", Error: assertion violated
spin: text of failed assertion: assert(((sum<2) || (counter==2)))
21: proc 0 (:init:) line 38 "increment.spin" (state 22) [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 12.3: Non-Atomic Increment Error Trail

12.1.1.2 Promela Warm-Up: Atomic Increment

```

1 proctype incrementer(byte me)
2 {
3     int temp;
4
5     atomic {
6         temp = counter;
7         counter = temp + 1;
8     }
9     progress[me] = 1;
10 }

```

Figure 12.4: Promela Code for Atomic Increment

```

(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   +
    hash conflicts        0 (resolved)

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 proctype incrementer
    (0 of 5 states)
unreached in proctype :init:
    (0 of 24 states)

```

Figure 12.5: 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 12.4. 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 12.5.

Table 12.1 shows the number of states and memory consumed as a function of number of incrementers modeled (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.

# incrementers	# states	megabytes
1	11	2.6
2	52	2.6
3	372	2.6
4	3,496	2.7
5	40,221	5.0
6	545,720	40.5
7	8,521,450	652.7

Table 12.1: Memory Usage of Increment Model

12.1.2 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.

12.1.2.1 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
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.

12.1.2.2 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:

`spin { x = 1; y = 2; r1 = x; r2 = y; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

`spin { x = 1; y = 2; r1 = y; r2 = x; barrier; };`

`spin { x = 1; y = 2; r1 = x; r2 = y; barrier; };`

</

```

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 12.6: 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 12.7: Atomic Block for Complex Promela Assertion

```

1 if
2 :: 1 -> r1 = x;
3   r2 = y
4 :: 1 -> r2 = y;
5   r1 = x
6 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 12.6.

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 12.7.

3. Promela does not provide functions. You must in-

```

1 #define spin_lock(mutex) \
2   do \
3     :: 1 -> atomic { \
4       if \
5         :: mutex == 0 -> \
6           mutex = 1; \
7         break \
8       :: else -> skip \
9     fi \
10   } \
11 od
12
13 #define spin_unlock(mutex) \
14   mutex = 0

```

Figure 12.8: Promela Code for Spinlock

stead use C preprocessor macros. However, you must use them carefully in order to avoid combinatorial explosion.

Now we are ready for more complex examples.

12.1.3 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 12.8. 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.

These macros are tested by the Promela code shown in Figure 12.9. This code is similar to that used to test

```

1 #include "lock.h"
2
3 #define N_LOCKERS 3
4
5 bit mutex = 0;
6 bit havelock[N_LOCKERS];
7 int sum;
8
9 proctype locker(byte me)
10 {
11     do
12     :: 1 ->
13         spin_lock(mutex);
14         havelock[me] = 1;
15         havelock[me] = 0;
16         spin_unlock(mutex)
17     od
18 }
19
20 init {
21     int i = 0;
22     int j;
23
24 end: do
25     :: i < N_LOCKERS ->
26         havelock[i] = 0;
27         run locker(i);
28         i++
29     :: i >= N_LOCKERS ->
30         sum = 0;
31         j = 0;
32         atomic {
33             do
34                 :: j < N_LOCKERS ->
35                     sum = sum + havelock[j];
36                     j = j + 1
37                 :: j >= N_LOCKERS ->
38                     break
39             od
40         }
41         assert(sum <= 1);
42         break
43     od
44 }

```

Figure 12.9: Promela Code to Test Spinlocks

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 12.10: Output for Spinlock Test

The output will look something like that shown in Figure 12.10. As expected, this run has no assertion failures (“errors: 0”).

Quick Quiz 12.1: Why is there an unreached statement in `locker`? After all, isn't this a *full* state-space search? ■

Quick Quiz 12.2: What are some Promela code-style issues with this example? ■

12.1.4 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 [McK06] 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:

```
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.

```
1 #include "lock.h"
2
3 #define N_QRCU_READERS 2
4 #define N_QRCU_UPDATORS 2
5
6 bit idx = 0;
7 byte ctr[2];
8 byte readerprogress[N_QRCU_READERS];
9 bit mutex = 0;
```

Figure 12.11: QRCU Global Variables

Returning to the Promela code for QRCU, the global variables are as shown in Figure 12.11. 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 an 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 {
3     int myidx;
4
5     do
6     :: 1 ->
7         myidx = idx;
8         atomic {
9             if
10                :: ctr[myidx] > 0 ->
11                    ctr[myidx]++;
12                    break
13                :: else -> skip
14                fi
15            }
16        od;
17        readerprogress[me] = 1;
18        readerprogress[me] = 2;
19        atomic { ctr[myidx]-- }
```

Figure 12.12: QRCU Reader Process

QRCU readers are modeled by the `qrcu_reader()` process shown in Figure 12.12. 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.

```

1 #define sum_unordered \
2   atomic { \
3     do \
4       :: 1 -> \
5         sum = ctr[0]; \
6         i = 1; \
7         break \
8       :: 1 -> \
9         sum = ctr[1]; \
10        i = 0; \
11        break \
12     od; \
13   } \
14   sum = sum + ctr[i]

```

Figure 12.13: QRCU Unordered Summation

The C-preprocessor macro shown in Figure 12.13 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 12.3: 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

```

1 proctype qrcu_updater(byte me)
2 {
3   int i;
4   byte readerstart[N_QRCU_READERS];
5   int sum;
6
7   do
8     :: 1 ->
9
10    /* Snapshot reader state. */
11
12    atomic {
13      i = 0;
14      do
15        :: i < N_QRCU_READERS ->
16          readerstart[i] = readerprogress[i];
17          i++
18        :: i >= N_QRCU_READERS ->
19          break
20        od
21    }
22
23    sum_unordered;
24    if
25      :: sum <= 1 -> sum_unordered
26      :: else -> skip
27    fi;
28    if
29      :: sum > 1 ->
30        spin_lock(mutex);
31        atomic { ctr[!idx]++ }
32        idx = !idx;
33        atomic { ctr[!idx]-- }
34        do
35          :: ctr[!idx] > 0 -> skip
36          :: ctr[!idx] == 0 -> break
37        od;
38        spin_unlock(mutex);
39        :: else -> skip
40      fi;
41
42    /* Verify reader progress. */
43
44    atomic {
45      i = 0;
46      sum = 0;
47      do
48        :: i < N_QRCU_READERS ->
49          sum = sum + (readerstart[i] == 1 &&
50                        readerprogress[i] == 1);
51          i++
52        :: i >= N_QRCU_READERS ->
53          assert(sum == 0);
54          break
55        od
56      }
57    od
58 }

```

Figure 12.14: QRCU Updater Process

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-invoke sum_unordered if the fastpath is potentially usable.

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 12.4: 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 12.5: 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         do
9             :: i < N_QRCU_READERS ->
10            readerprogress[i] = 0;
11            run qrcu_reader(i);
12            i++
13            :: i >= N_QRCU_READERS -> break
14        od;
15        i = 0;
16        do
17            :: i < N_QRCU_UPDATER ->
18            run qrcu_updater(i);
19            i++
20            :: i >= N_QRCU_UPDATER -> break
21        od
22    }
23 }
```

Figure 12.15: QRCU Initialization Process

All that remains is the initialization block shown in Figure 12.15. 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.

12.1.4.1 Running the QRCU Example

To run the QRCU example, combine the code fragments in the previous section into a single file named `qrcu.spin`,

updaters	readers	# states	MB
1	1	376	2.6
1	2	6,177	2.9
1	3	82,127	7.5
2	1	29,399	4.5
2	2	1,071,180	75.4
2	3	33,866,700	2,715.2
3	1	258,605	22.3
3	2	169,533,000	14,979.9

Table 12.2: Memory Usage of QRCU Model

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 12.2. 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.

12.1.4.2 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.

12.1.4.3 Alternative Approach: Proof of Correctness

An informal proof [McK07b] follows:

1. For `synchronize_qrcu()` to exit too early, then by definition there must have been at least one reader present during `synchronize_qrcu()`'s full execution.
2. The counter corresponding to this reader will have been at least 1 during this time interval.
3. The `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 `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.

12.1.4.4 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.

12.1.4.5 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.

12.1.4.6 Is QRCU Really Correct?

Is QRCU really correct? We have a Promela-based mechanical proof and a by-hand proof that both say that it is. However, a recent paper by Alglave et al. [AKT13] says otherwise (see Section 5.1 of the paper at the bottom of page 12). Which is it?

I do not know, as I never have been able to track down the code in which Alglave, Kroening, and Tautschig found a flaw, though the authors did point out that the concurrency benchmarks are not necessarily equivalent to the real-world examples that they were derived from. In some

sense, it does not matter, as QRCU never was accepted into the Linux kernel, nor to the best of my knowledge was it ever used in any other production software.

However, if you do intend to use QRCU, please take care. Its proofs of correctness might or might not themselves be correct. Which is one reason why formal verification is unlikely to completely replace testing.

Quick Quiz 12.6: Given that we have two independent proofs of correctness for the QRCU algorithm described herein, and given that the proof of incorrectness covers what is likely a different algorithm, why is there any room for doubt? ■

12.1.5 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 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 CPU 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 illusory.

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 Section 12. 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 12.7: 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 12.1.5.1 gives an overview of preemptible RCU’s dynticks interface, Section 12.1.6, and Section 12.1.6.8 lists lessons (re)learned during this effort.

12.1.5.1 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.

12.1.5.2 Task Interface

When a given CPU enters dynticks-idle mode because it has no more tasks to run, it invokes `rcu_enter_nohz()`:

```
1 static inline void rCU_enter_nohz(void)
2 {
3     mb();
4     __get_cpu_var(dynticks_progress_counter)++;
5     WARN_ON(__get_cpu_var(dynticks_progress_counter) &
6             0x1);
7 }
```

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()`:

```
1 static inline void rCU_exit_nohz(void)
2 {
3     __get_cpu_var(dynticks_progress_counter)++;
4     mb();
5     WARN_ON(!__get_cpu_var(dynticks_progress_counter) &
6             0x1);
7 }
```

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.

12.1.5.3 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:

```
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,
9                  cpu) & 0x1) == 0) {
10        per_cpu(dynticks_progress_counter, cpu)++;
11        smp_mb();
12        per_cpu(rcu_update_flag, cpu)++;
13    }
14 }
```

Line 3 fetches the current CPU's number, while lines 5 and 6 increment the `rcu_update_flag` nesting counter if it is already non-zero. Lines 7-9 check to see whether we are the outermost level of interrupt, and, if so, whether `dynticks_progress_counter` needs to be incremented. If so, line 10 increments `dynticks_progress_counter`, line 11 executes a memory barrier, and line 12 increments `rcu_update_flag`. As with `rcu_exit_nohz()`, the memory barrier ensures that any other CPU that sees the effects of an RCU read-side critical section in the interrupt handler (following the `rcu_irq_enter()` invocation) will also see the increment of `dynticks_progress_counter`.

Quick Quiz 12.8: Why not simply increment `rcu_update_flag`, and then only increment `dynticks_progress_counter` if the old value of `rcu_update_flag` was zero??? ■

Quick Quiz 12.9: But if line 7 finds that we are the outermost interrupt, wouldn't we *always* need to increment `dynticks_progress_counter`? ■

Interrupt exit is handled similarly by `rcu_irq_exit()`:

```
1 void rCU_irq_exit(void)
2 {
3     int cpu = smp_processor_id();
4
5     if (per_cpu(rcu_update_flag, cpu)) {
6         if (--per_cpu(rcu_update_flag, cpu))
7             return;
8         WARN_ON(in_interrupt());
9         smp_mb();
10        per_cpu(dynticks_progress_counter, cpu)++;
11    }
12 }
```

```

11     WARN_ON(per_cpu(dynticks_progress_counter,
12                     cpu) & 0x1);
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_enter_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.

12.1.5.4 Grace-Period Interface

Of the four preemptible RCU grace-period states shown in Figure 12.16, only the `rcu_try_flip_waitack_state()` and `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 `dynticks_progress_counter` variable, placing the snapshot in another per-CPU variable, `rcu_dyntick_snapshot`. This is accomplished by invoking `dyntick_save_progress_counter`, shown below:

```

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 `rcu_try_flip_waitack_state()` state invokes `rcu_try_flip_waitack_needed()`, shown below:

```

1 static inline int
2 rCU_try_flip_waitack_needed(int cpu)

```

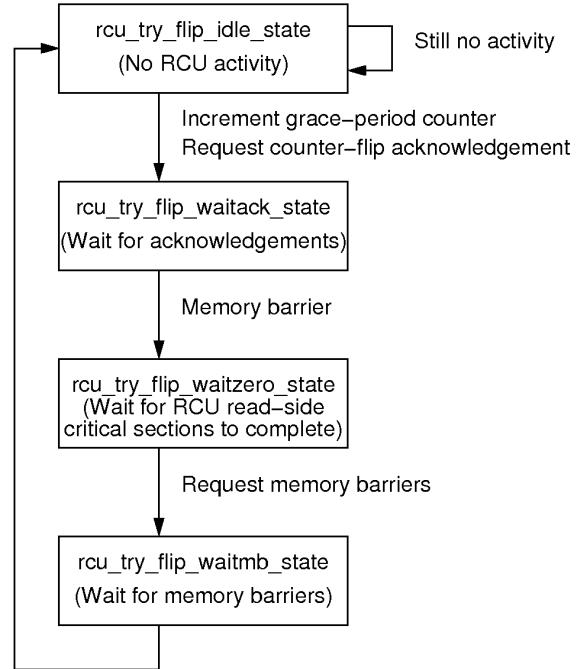


Figure 12.16: Preemptible RCU State Machine

```

3 {
4     long curr;
5     long snap;
6
7     curr = per_cpu(dynticks_progress_counter, cpu);
8     snap = per_cpu(rcu_dyntick_snapshot, cpu);
9     smp_mb();
10    if ((curr == snap) && ((curr & 0x1) == 0))
11        return 0;
12    if ((curr - snap) > 2 || (snap & 0x1) == 0)
13        return 0;
14    return 1;
15 }

```

Lines 7 and 8 pick up current and snapshot versions of `dynticks_progress_counter`, respectively. The memory barrier on line 9 ensures that the counter checks in the later `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 ex-

plicitly respond.

For its part, the `rcu_try_flip_waitmb_state` state invokes `rcu_try_flip_waitmb_needed()`, shown below:

```

1 static inline int
2 rCU_trY_flip_wAItmb_nEEdEd(int CPU)
3 {
4     long curr;
5     long snap;
6
7     curr = per_Cpu(dynticks_progress_counter, CPU);
8     snap = per_Cpu(rcu_dyntick_snapshot, CPU);
9     smp_mb();
10    if ((curr == snap) && ((curr & 0x1) == 0))
11        return 0;
12    if (curr != snap)
13        return 0;
14    return 1;
15 }
```

This is quite similar to `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 `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 12.10: Can you spot any bugs in any of the code in this section? ■

12.1.6 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.

12.1.6.1 Basic Model

This section translates the process-level dynticks entry/exit code and the grace-period processing into Promela [Hol03]. We start with `rcu_exit_nohz()` and `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:

```

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 ->
```

```

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;
21 }
```

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 12.11: Why isn't the memory barrier in `rcu_exit_nohz()` and `rcu_enter_nohz()` modeled in Promela? ■

Quick Quiz 12.12: 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? ■

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_waitack_needed()`, `rcu_try_flip_waitmb_needed()`, as well as portions of `rcu_try_flip_waitack()` and `rcu_try_flip_waitmb()`, all from the 2.6.25-rc4 kernel. The following `grace_period()` Promela process models these functions as they would be invoked during a single pass through preemptible RCU's grace-period processing.

```

1 proctype grace_period()
2 {
3     byte curr;
4     byte snap;
5 }
```

```

6  atomic {
7    printf("MDLN = %d\n", MAX_DYNTICK_LOOP_NOHZ);
8    snap = dynticks_progress_counter;
9  }
10 do
11   :: 1 ->
12   atomic {
13     curr = dynticks_progress_counter;
14     if
15       :: (curr == snap) && ((curr & 1) == 0) ->
16         break;
17       :: (curr - snap) > 2 || (snap & 1) == 0 ->
18         break;
19       :: 1 -> skip;
20     fi;
21   }
22 od;
23 snap = dynticks_progress_counter;
24 do
25   :: 1 ->
26   atomic {
27     curr = dynticks_progress_counter;
28     if
29       :: (curr == snap) && ((curr & 1) == 0) ->
30         break;
31       :: (curr != snap) ->
32         break;
33       :: 1 -> skip;
34     fi;
35   }
36 od;
37 }

```

Lines 6-9 print out the loop limit (but only into the .trail file in case of error) and models a line of code from `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_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 23 models a line from `rcu_try_flip_waitzero()` and its call to `dyntick_save_progress_counter()`, again taking a snapshot of the CPU's `dynticks_progress_counter` variable.

Finally, lines 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 12.13: 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.

12.1.6.2 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 `gp_state` variable that can take on three states as follows:

```

1 #define GP_IDLE      0
2 #define GP_WAITING  1
3 #define GP_DONE     2
4 byte gp_state = GP_DONE;

```

The `grace_period()` process sets this variable as it progresses through the grace-period phases, as shown below:

```

1 proctype grace_period()
2 {
3   byte curr;
4   byte snap;
5
6   gp_state = GP_IDLE;
7   atomic {
8     printf("MDLN = %d\n", MAX_DYNTICK_LOOP_NOHZ);
9     snap = dynticks_progress_counter;
10    gp_state = GP_WAITING;
11  }
12 do
13   :: 1 ->
14   atomic {
15     curr = dynticks_progress_counter;
16     if
17       :: (curr == snap) && ((curr & 1) == 0) ->
18         break;
19       :: (curr - snap) > 2 || (snap & 1) == 0 ->
20         break;
21       :: 1 -> skip;
22     fi;
23   }
24 od;
25 gp_state = GP_DONE;
26 gp_state = GP_IDLE;
27 atomic {
28   snap = dynticks_progress_counter;
29   gp_state = GP_WAITING;
30 }
31 do
32   :: 1 ->
33   atomic {
34     curr = dynticks_progress_counter;
35     if
36       :: (curr == snap) && ((curr & 1) == 0) ->
37         break;
38       :: (curr != snap) ->

```

```

39         break;
40         :: 1 -> skip;
41     fi;
42 }
43 od;
44 gp_state = GP_DONE;
45 }

```

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 `gp_state` variable cannot jump from `GP_IDLE` to `GP_DONE` during a time period over which RCU readers could plausibly persist.

Quick Quiz 12.14: Given there are a pair of back-to-back changes to `gp_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:

```

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 = (gp_state == GP_IDLE);
14        assert((dynticks_progress_counter & 1) == 1);
15    }
16    atomic {
17        tmp = dynticks_progress_counter;
18        assert(!old_gp_idle ||
19               gp_state != GP_DONE);
20    }
21    atomic {
22        dynticks_progress_counter = tmp + 1;
23        assert((dynticks_progress_counter & 1) == 0);
24    }
25    i++;
26 od;
27 dyntick_nohz_done = 1;
28 }

```

Line 13 sets a new `old_gp_idle` flag if the value of the `gp_state` variable is `GP_IDLE` at the beginning of task execution, and the assertion at lines 18 and 19 fire if the `gp_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.

12.1.6.3 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:

```

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 = (gp_state == GP_IDLE);
14        assert((dynticks_progress_counter & 1) == 1);
15    }
16    atomic {
17        tmp = dynticks_progress_counter;
18        assert(!old_gp_idle ||
19               gp_state != GP_DONE);
20    }
21    atomic {
22        dynticks_progress_counter = tmp + 1;
23        assert((dynticks_progress_counter & 1) == 0);
24    }
25    i++;
26 od;
27 dyntick_nohz_done = 1;
28 }

```

With this variable in place, we can add assertions to `grace_period()` to check for unnecessary blockage as follows:

```

1 proctype grace_period()
2 {
3     byte curr;
4     byte snap;
5     bit shouldexit;
6
7     gp_state = GP_IDLE;
8     atomic {
9         printf("MDLN = %d\n", MAX_DYNTICK_LOOP_NOHZ);
10        shouldexit = 0;
11        snap = dynticks_progress_counter;
12        gp_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 gp_state = GP_DONE;
30 gp_state = GP_IDLE;
31 atomic {
32     shouldexit = 0;
33     snap = dynticks_progress_counter;
34     gp_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 gp_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
7     curr = per_cpu(dynticks_progress_counter, cpu);
8     snap = per_cpu(rcu_dyntick_snapshot, cpu);
9     smp_mb();
10    if ((curr == snap) && ((curr & 0x1) == 0))
11        return 0;
12    if ((curr - snap) > 2 || (curr & 0x1) == 0)

```

```

13     return 0;
14     return 1;
15 }
```

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
7     curr = per_cpu(dynticks_progress_counter, cpu);
8     snap = per_cpu(rcu_dyntick_snapshot, cpu);
9     smp_mb();
10    if ((curr - snap) >= 2 || (curr & 0x1) == 0)
11        return 0;
12    return 1;
13 }
```

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.

12.1.6.4 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:

```

1 #define EXECUTE_MAINLINE(label, stmt) \
2 label: skip; \
3     atomic { \
4         if \
5             :: in_dyntick_irq -> goto label; \
6         :: else -> stmt; \
7         fi; \
8     } \
```

One might use this macro as follows:

```

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.

12.1.6.5 Validating Interrupt Handlers

The first step is to convert `dyntick_nohz()` to `EXECUTE_MAINLINE()` form, as follows:

```

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 = (gp_state == GP_IDLE);
15                           assert((dynticks_progress_counter & 1) == 1))
16        EXECUTE_MAINLINE(stmt3,
17                           tmp = dynticks_progress_counter;
18                           assert(!old_gp_idle ||
19                                 gp_state != GP_DONE))
20        EXECUTE_MAINLINE(stmt4,
21                           dynticks_progress_counter = tmp + 1;
22                           assert((dynticks_progress_counter & 1) == 0))
23        i++;
24    od;
25    dyntick_nohz_done = 1;
26 }
```

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 12.15: But what would you do if you needed the statements in a single EXECUTE_MAINLINE() group to execute non-atomically? ■

Quick Quiz 12.16: 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:

```

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 &&
19        (dynticks_progress_counter & 1) == 0 ->
20        tmp = dynticks_progress_counter;
21        dynticks_progress_counter = tmp + 1;
22        tmp = rcu_update_flag;
23        rcu_update_flag = tmp + 1;
24    :: else -> skip;
25    fi;
26    tmp = in_interrupt;
27    in_interrupt = tmp + 1;
28    old_gp_idle = (gp_state == GP_IDLE);
29    assert(!old_gp_idle || gp_state != GP_DONE);
30    tmp = in_interrupt;
31    in_interrupt = tmp - 1;
32    if
33    :: rcu_update_flag != 0 ->
34        tmp = rcu_update_flag;
35        rcu_update_flag = tmp - 1;
36    if
37    :: rcu_update_flag == 0 ->
38        tmp = dynticks_progress_counter;
39        dynticks_progress_counter = tmp + 1;
40    :: else -> skip;
41    fi;
42    :: else -> skip;
43    fi;
44    atomic {
45        in_dyntick_irq = 0;
46        i++;
47    }
48    od;
49    dyntick_irq_done = 1;
50 }

```

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, just like the corresponding line of dyntick_nohz().

Quick Quiz 12.17: 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 verifies 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 12.18: What property of interrupts is this dynticks_irq() process unable to model? ■

The grace_period process then becomes as follows:

```

1 proctype grace_period()
2 {
3     byte curr;
4     byte snap;
5     bit shouldexit;
6
7     gp_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        gp_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    gp_state = GP_DONE;
29    gp_state = GP_IDLE;
30    atomic {
31        shouldexit = 0;
32        snap = dynticks_progress_counter;
33        gp_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    gp_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.

12.1.6.6 Validating Nested Interrupt Handlers

Nested interrupt handlers may be modeled by splitting the body of the loop in `dyntick_irq()` as follows:

```

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        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 = (gp_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                        gp_state != GP_DONE);
48                :: else -> skip;
49            fi;
50        }

```

```

51    tmp = in_interrupt;
52    in_interrupt = tmp - 1;
53    if
54        :: rcu_update_flag != 0 ->
55            tmp = rcu_update_flag;
56            rcu_update_flag = tmp - 1;
57        if
58            :: rcu_update_flag == 0 ->
59                tmp = dynticks_progress_counter;
60                dynticks_progress_counter = tmp + 1;
61            :: else -> skip;
62        fi;
63        :: else -> skip;
64    fi;
65    atomic {
66        j++;
67        in_dyntick_irq = (i != j);
68    }
69 od;
70 dyntick_irq_done = 1;
71 }

```

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 `gp_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 `gp_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.

12.1.6.7 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:

```

1 proctype dyntick_nmi()
2 {
3     byte tmp;
4     byte i = 0;
5     bit old_gp_idle;
6
7     do
8         :: i >= MAX_DYNTICK_LOOP_NMI -> break;
9         :: i < MAX_DYNTICK_LOOP_NMI ->
10            in_dyntick_nmi = 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 &&
19                (dynticks_progress_counter & 1) == 0 ->
20                    tmp = dynticks_progress_counter;
21                    dynticks_progress_counter = tmp + 1;
22                    tmp = rcu_update_flag;
23                    rcu_update_flag = tmp + 1;
24            :: else -> skip;
25        fi;
26        tmp = in_interrupt;
27        in_interrupt = tmp + 1;
28        old_gp_idle = (gp_state == GP_IDLE);
29        assert(!old_gp_idle || gp_state != GP_DONE);
30        tmp = in_interrupt;
31        in_interrupt = tmp - 1;
32    if
33        :: rcu_update_flag != 0 ->
34            tmp = rcu_update_flag;
35            rcu_update_flag = tmp - 1;
36        if
37            :: rcu_update_flag == 0 ->
38                tmp = dynticks_progress_counter;
39                dynticks_progress_counter = tmp + 1;
40            :: else -> skip;
41        fi;
42        :: else -> skip;
43    fi;
44    atomic {
45        i++;
46        in_dyntick_nmi = 0;
47    }
48    od;
49    dyntick_nmi_done = 1;
50 }

```

Of course, the fact that we have NMIs requires adjustments in the other components. For example, the EXECUTE_MAINLINE() macro now needs to pay attention to the NMI handler (in_dyntick_nmi) as well as the interrupt handler (in_dyntick_irq) by checking the dyntick_nmi_done variable as follows:

```

1 #define EXECUTE_MAINLINE(label, stmt) \
2 label: skip; \
3     atomic { \
4         if \
5             :: in_dyntick_irq || \
6                 in_dyntick_nmi -> goto label; \
7             :: else -> stmt; \
8         fi; \
9     } \

```

We will also need to introduce an EXECUTE_IRQ() macro that checks in_dyntick_nmi in order to allow

dyntick_irq() to exclude dyntick_nmi():

```

1 #define EXECUTE_IRQ(label, stmt) \
2 label: skip; \
3     atomic { \
4         if \
5             :: in_dyntick_nmi -> goto label; \
6             :: else -> stmt; \
7         fi; \
8     } \

```

It is further necessary to convert dyntick_irq() to EXECUTE_IRQ() as follows:

```

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
20                :: in_dyntick_nmi -> goto stmt1;
21                :: !in_dyntick_nmi && rcu_update_flag ->
22                    goto stmt1_then;
23                :: else -> goto stmt1_else;
24            fi;
25        }
26        stmt1_then: skip;
27        EXECUTE_IRQ(stmt1_1, tmp = rcu_update_flag)
28        EXECUTE_IRQ(stmt1_2, rcu_update_flag = tmp + 1)
29    stmt1_else: skip;
30    stmt2: skip; atomic {
31        if
32            :: in_dyntick_nmi -> goto stmt2;
33            :: !in_dyntick_nmi &&
34                !in_interrupt &&
35                (dynticks_progress_counter & 1) == 0 ->
36                    goto stmt2_then;
37            :: else -> goto stmt2_else;
38        fi;
39    }
40    stmt2_then: skip;
41    EXECUTE_IRQ(stmt2_1, tmp = dynticks_progress_counter)
42    EXECUTE_IRQ(stmt2_2,
43        dynticks_progress_counter = tmp + 1)
44    EXECUTE_IRQ(stmt2_3, tmp = rcu_update_flag)
45    EXECUTE_IRQ(stmt2_4, rcu_update_flag = tmp + 1)
46    stmt2_else: skip;
47    EXECUTE_IRQ(stmt3, tmp = in_interrupt)
48    EXECUTE_IRQ(stmt4, in_interrupt = tmp + 1)
49    stmt5: skip;
50    atomic {
51        if
52            :: in_dyntick_nmi -> goto stmt4;
53            :: !in_dyntick_nmi && outermost ->
54                old_gp_idle = (gp_state == GP_IDLE);
55            :: else -> skip;
56        fi;
57    }
58    i++;
59    :: j < i ->

```

```

60 stmt6: skip;
61     atomic {
62         if
63             :: in_dyntick_nmi -> goto stmt6;
64             :: !in_dyntick_nmi && j + 1 == i ->
65                 assert(!old_gp_idle ||
66                         gp_state != GP_DONE);
67             :: else -> skip;
68         fi;
69     }
70 EXECUTE_IRQ(stmt7, tmp = in_interrupt);
71 EXECUTE_IRQ(stmt8, in_interrupt = tmp - 1);
72
73 stmt9: skip;
74     atomic {
75         if
76             :: in_dyntick_nmi -> goto stmt9;
77             :: !in_dyntick_nmi && rcu_update_flag != 0 ->
78                 goto stmt9_then;
79             :: else -> goto stmt9_else;
80         fi;
81     }
82 stmt9_then: skip;
83     EXECUTE_IRQ(stmt9_1, tmp = rcu_update_flag)
84     EXECUTE_IRQ(stmt9_2, rcu_update_flag = tmp - 1)
85 stmt9_3: skip;
86     atomic {
87         if
88             :: in_dyntick_nmi -> goto stmt9_3;
89             :: !in_dyntick_nmi && rcu_update_flag == 0 ->
90                 goto stmt9_3_then;
91             :: else -> goto stmt9_3_else;
92         fi;
93     }
94 stmt9_3_then: skip;
95     EXECUTE_IRQ(stmt9_3_1,
96                 tmp = dynticks_progress_counter)
97     EXECUTE_IRQ(stmt9_3_2,
98                 dynticks_progress_counter = tmp + 1)
99 stmt9_3_else:
100 stmt9_else: skip;
101     atomic {
102         j++;
103         in_dyntick_irq = (i != j);
104     }
105 od;
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:

```

1 proctype grace_period()
2 {
3     byte curr;
4     byte snap;
5     bit shouldexit;
6
7     gp_state = GP_IDLE;
8     atomic {
9         printf("MDLN = %d\n", MAX_DYNTICK_LOOP_NOHZ);
10        printf("MDLI = %d\n", MAX_DYNTICK_LOOP_IRQ);
11        printf("MDLN = %d\n", MAX_DYNTICK_LOOP_NMI);
12        shouldexit = 0;
13        snap = dynticks_progress_counter;
14        gp_state = GP_WAITING;
15    }

```

```

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     gp_state = GP_DONE;
32     gp_state = GP_IDLE;
33     atomic {
34         shouldexit = 0;
35         snap = dynticks_progress_counter;
36         gp_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     gp_state = GP_DONE;
54 }

```

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 12.19: Does Paul *always* write his code in this painfully incremental manner? ■

12.1.6.8 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 12.17.

```

static inline void rCU_enter_noHZ(void)
{
+    mb();
-    __get_cpu_var(dynticks_progress_counter)++;
-    mb();
}

static inline void rCU_exit_noHZ(void)
{
-    mb();
+    __get_cpu_var(dynticks_progress_counter)++;
+    mb();
}

```

Figure 12.17: Memory-Barrier Fix Patch

```

-    if ((curr - snap) > 2 || (snap & 0x1) == 0)
+    if ((curr - snap) > 2 || (curr & 0x1) == 0)

```

Figure 12.18: 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 12.18.
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. You will then see that the obvious verify-the-verification technique is to deliberately insert bugs in the code being verified. If the verification fails to find them, the verification clearly is buggy.
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.**

To this last point, it turns out that there is a much simpler solution to the dynticks problem, which is presented in the next section.

```

1 struct rCU_dynticks {
2     int dynticks_nesting;
3     int dynticks;
4     int dynticks_nmi;
5 };
6
7 struct rCU_data {
8     ...
9     int dynticks_snap;
10    int dynticks_nmi_snap;
11 ...
12 };

```

Figure 12.19: Variables for Simple Dynticks Interface

12.1.6.9 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 leads to the notion of providing separate code paths and variables for irqs and NMIs, as has been done for hierarchical RCU [McK08a] as indirectly suggested by Manfred Spraul [Spr08].

12.1.6.10 State Variables for Simplified Dynticks Interface

Figure 12.19 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 dyntick-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 12.20: 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? ■

12.1.6.11 Entering and Leaving Dynticks-Idle Mode

Figure 12.20 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.

Line 6 ensures that any prior memory accesses (which might include accesses from RCU read-side critical sections) are seen by other CPUs before those marking entry to dynticks-idle mode. Lines 7 and 12 disable and reenable irqs. Line 8 acquires a pointer to the current CPU's `rcu_dynticks` structure, and line 9 increments the current CPU's `dynticks` counter, which should now be even, given that we are entering dynticks-idle mode in process context. Finally, line 10 decrements `dynticks_nesting`, which should now be zero.

The `rcu_exit_nohz()` function is quite similar, but increments `dynticks_nesting` rather than decrementing it and checks for the opposite `dynticks` polarity.

12.1.6.12 NMIs From Dynticks-Idle Mode

Figure 12.21 shows the `rcu_nmi_enter()` and `rcu_nmi_exit()` functions, which inform RCU of NMI en-

```

1 void rcu_enter_nohz(void)
2 {
3     unsigned long flags;
4     struct rcu_dynticks *rdtp;
5
6     smp_mb();
7     local_irq_save(flags);
8     rdtp = &__get_cpu_var(rcu_dynticks);
9     rdtp->dynticks++;
10    rdtp->dynticks_nesting--;
11    WARN_ON(rdtp->dynticks & 0x1);
12    local_irq_restore(flags);
13 }
14
15 void rcu_exit_nohz(void)
16 {
17     unsigned long flags;
18     struct rcu_dynticks *rdtp;
19
20     local_irq_save(flags);
21     rdtp = &__get_cpu_var(rcu_dynticks);
22     rdtp->dynticks++;
23     rdtp->dynticks_nesting++;
24     WARN_ON(!(rdtp->dynticks & 0x1));
25     local_irq_restore(flags);
26     smp_mb();
27 }
```

Figure 12.20: Entering and Exiting Dynticks-Idle Mode

```

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     WARN_ON(!(rdtp->dynticks_nmi & 0x1));
10    smp_mb();
11 }
12
13 void rcu_nmi_exit(void)
14 {
15     struct rcu_dynticks *rdtp;
16
17     rdtp = &__get_cpu_var(rcu_dynticks);
18     if (rdtp->dynticks & 0x1)
19         return;
20     smp_mb();
21     rdtp->dynticks_nmi++;
22     WARN_ON(rdtp->dynticks_nmi & 0x1);
23 }
```

Figure 12.21: NMIs From Dynticks-Idle Mode

```

1 void rCU_irk_enter(void)
2 {
3     struct rCU_dynticks *rdtp;
4
5     rdtp = &__get_cpu_var(rCU_dynticks);
6     if (rdtp->dynticks_nesting++)
7         return;
8     rdtp->dynticks++;
9     WARN_ON(!(rdtp->dynticks & 0x1));
10    smp_mb();
11 }
12
13 void rCU_irk_exit(void)
14 {
15     struct rCU_dynticks *rdtp;
16
17     rdtp = &__get_cpu_var(rCU_dynticks);
18     if (--rdtp->dynticks_nesting)
19         return;
20     smp_mb();
21     rdtp->dynticks++;
22     WARN_ON(rdp->dynticks & 0x1);
23     if (__get_cpu_var(rCU_data).nxtlist ||
24         __get_cpu_var(rCU_bh_data).nxtlist)
25         set_need_resched();
26 }

```

Figure 12.22: Interrupts From Dynticks-Idle Mode

try and exit, respectively, from dynticks-idle mode. However, if the NMI arrives during an irq handler, then RCU will already be on the lookout for RCU read-side critical sections from this CPU, so lines 6 and 7 of `rcu_nmi_enter` and lines 18 and 19 of `rcu_nmi_exit` silently return if `dynticks` is odd. Otherwise, the two functions increment `dynticks_nmi`, with `rcu_nmi_enter()` leaving it with an odd value and `rcu_nmi_exit()` leaving it with an even value. Both functions execute memory barriers between this increment and possible RCU read-side critical sections on lines 11 and 21, respectively.

12.1.6.13 Interrupts From Dynticks-Idle Mode

Figure 12.22 shows `rcu_irk_enter()` and `rcu_irk_exit()`, which inform RCU of entry to and exit from, respectively, irq context. Line 6 of `rcu_irk_enter()` increments `dynticks_nesting`, and if this variable was already non-zero, line 7 silently returns. Otherwise, line 8 increments `dynticks`, which will then have an odd value, consistent with the fact that this CPU can now execute RCU read-side critical sections. Line 10 therefore executes a memory barrier to ensure that the increment of `dynticks` is seen before any RCU read-side critical sections that the subsequent irq handler might execute.

Line 18 of `rcu_irk_exit` decrements `dynticks_`

```

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) &&
14        ((snap_nmi & 0x1) == 0);
15    if (ret)
16        rdp->dynticks_qs++;
17    return ret;
18 }

```

Figure 12.23: Saving Dyntick Progress Counters

nesting, and if the result is non-zero, line 19 silently returns. Otherwise, line 20 executes a memory barrier to ensure that the increment of `dynticks` on line 21 is seen after any RCU read-side critical sections that the prior irq handler might have executed. Line 22 verifies that `dynticks` is now even, consistent with the fact that no RCU read-side critical sections may appear in dynticks-idle mode. Lines 23-25 check to see if the prior irq handlers enqueued any RCU callbacks, forcing this CPU out of dynticks-idle mode via a reschedule API if so.

12.1.6.14 Checking For Dynticks Quiescent States

Figure 12.23 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 12.20, 12.21, and 12.22. Lines 11 and 12 record the snapshots for later calls to `rcu_implicit_dynticks_qs`, and lines 13 and 14 check to see if the CPU is in dynticks-idle mode with neither irqs nor NMIs in progress (in other words, both snapshots have even values), hence in an extended quiescent state. If so, lines 15 and 16 count this event, and line 17 returns true if the CPU was in a quiescent state.

Figure 12.24 shows `dyntick_save_progress_counter`, which is called to check whether a CPU has entered dyntick-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,

```

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
9     curr = rdp->dynticks->dynticks;
10    snap = rdp->dynticks_snap;
11    curr_nmi = rdp->dynticks->dynticks_nmi;
12    snap_nmi = rdp->dynticks_nmi_snap;
13    smp_mb();
14    if ((curr != snap || (curr & 0x1) == 0) &&
15        (curr_nmi != snap_nmi ||
16         (curr_nmi & 0x1) == 0)) {
17        rdp->dynticks_fqs++;
18        return 1;
19    }
20    return rCU_implicit_offline_qs(rdp);
21 }

```

Figure 12.24: Checking Dyntick Progress Counters

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 12.20, 12.21, and 12.22. Lines 14-16 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 dyntick-idle quiescent state, then line 17 counts that fact and line 18 returns an indication of this fact. Either way, line 20 checks for race conditions that can result in RCU waiting for a CPU that is offline.

Quick Quiz 12.21: This is still pretty complicated. Why not just have a `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? ■

12.1.6.15 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 12.1.5, 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.

12.2 Special-Purpose State-Space Search

Although Promela and spin allow you to verify pretty much any (smallish) algorithm, their very generality can sometimes be a curse. For example, Promela does not understand memory models or any sort of reordering semantics. This section therefore describes some state-space search tools that understand memory models used by production systems, greatly simplifying the verification of weakly ordered code.

For example, Section 12.1.4 showed how to convince Promela to account for weak memory ordering. Although this approach can work well, it requires that the developer fully understand the system's memory model. Unfortunately, few (if any) developers fully understand the complex memory models of modern CPUs.

Therefore, another approach is to use a tool that already understands this memory ordering, such as the PPCMEM tool produced by Peter Sewell and Susmit Sarkar at the University of Cambridge, Luc Maranget, Francesco Zappa Nardelli, and Pankaj Pawan at INRIA, and Jade Alglave at Oxford University, in cooperation with Derek Williams of IBM [AMP⁺11]. This group formalized the memory models of Power, ARM, x86, as well as that of the C/C++11 standard [Bec11], and produced the PPCMEM tool based on the Power and ARM formalizations.

Quick Quiz 12.22: But x86 has strong memory ordering! Why would you need to formalize its memory model? ■

The PPCMEM tool takes *litmus tests* as input. A sample litmus test is presented in Section 12.2.1. Section 12.2.2 relates this litmus test to the equivalent C-language program, Section 12.2.3 describes how to apply PPCMEM to this litmus test, and Section 12.2.4 discusses the implications.

```

1 PPC SB+lwsync-RMW-lwsync+isync-simple
2 ""
3 {
4 0:r2=x; 0:r3=2; 0:r4=y; 0:r10=0; 0:r11=0; 0:r12=z;
5 1:r2=y; 1:r4=x;
6 }
7 P0 | P1 ;
8 li r1,1 | li r1,1 ;
9 stw r1,0(r2) | stw r1,0(r2) ;
10 lwsync | sync ;
11 | lwz r3,0(r4) ;
12 lwarx r11,r10,r12 | ;
13 stwcx. r11,r10,r12 | ;
14 bne Fail1 | ;
15 isync | ;
16 lwz r3,0(r4) | ;
17 Fail1: | ;
18
19 exists
20 (0:r3=0 /\ 1:r3=0)

```

Figure 12.25: PPCMEM Litmus Test

12.2.1 Anatomy of a Litmus Test

An example PowerPC litmus test for PPCMEM is shown in Figure 12.25. The ARM interface works exactly the same way, but with ARM instructions substituted for the Power instructions and with the initial “PPC” replaced by “ARM”. You can select the ARM interface by clicking on “Change to ARM Model” at the web page called out above.

In the example, line 1 identifies the type of system (“ARM” or “PPC”) and contains the title for the model. Line 2 provides a place for an alternative name for the test, which you will usually want to leave blank as shown in the above example. Comments can be inserted between lines 2 and 3 using the OCaml (or Pascal) syntax of `(* *)`.

Lines 3-6 give initial values for all registers; each is of the form `P : R=V`, where `P` is the process identifier, `R` is the register identifier, and `V` is the value. For example, process 0’s register `r3` initially contains the value 2. If the value is a variable (`x`, `y`, or `z` in the example) then the register is initialized to the address of the variable. It is also possible to initialize the contents of variables, for example, `x=1` initializes the value of `x` to 1. Uninitialized variables default to the value zero, so that in the example, `x`, `y`, and `z` are all initially zero.

Line 7 provides identifiers for the two processes, so that the `0 : r3=2` on line 4 could instead have been written `P0 : r3=2`. Line 7 is required, and the identifiers must be of the form `Pn`, where `n` is the column number, starting from zero for the left-most column. This may seem unnecessarily strict, but it does prevent considerable confusion

in actual use.

Quick Quiz 12.23: Why does line 8 of Figure 12.25 initialize the registers? Why not instead initialize them on lines 4 and 5? ■

Lines 8-17 are the lines of code for each process. A given process can have empty lines, as is the case for P0’s line 11 and P1’s lines 12-17. Labels and branches are permitted, as demonstrated by the branch on line 14 to the label on line 17. That said, too-free use of branches will expand the state space. Use of loops is a particularly good way to explode your state space.

Lines 19-20 show the assertion, which in this case indicates that we are interested in whether P0’s and P1’s `r3` registers can both contain zero after both threads complete execution. This assertion is important because there are a number of use cases that would fail miserably if both P0 and P1 saw zero in their respective `r3` registers.

This should give you enough information to construct simple litmus tests. Some additional documentation is available, though much of this additional documentation is intended for a different research tool that runs tests on actual hardware. Perhaps more importantly, a large number of pre-existing litmus tests are available with the online tool (available via the “Select ARM Test” and “Select POWER Test” buttons). It is quite likely that one of these pre-existing litmus tests will answer your Power or ARM memory-ordering question.

12.2.2 What Does This Litmus Test Mean?

P0’s lines 8 and 9 are equivalent to the C statement `x=1` because line 4 defines P0’s register `r2` to be the address of `x`. P0’s lines 12 and 13 are the mnemonics for load-linked (“load register exclusive” in ARM parlance and “load reserve” in Power parlance) and store-conditional (“store register exclusive” in ARM parlance), respectively. When these are used together, they form an atomic instruction sequence, roughly similar to the compare-and-swap sequences exemplified by the x86 `colock;cmpxchg` instruction. Moving to a higher level of abstraction, the sequence from lines 10-15 is equivalent to the Linux kernel’s `atomic_add_return(&z, 0)`. Finally, line 16 is roughly equivalent to the C statement `r3=y`.

P1’s lines 8 and 9 are equivalent to the C statement `y=1`, line 10 is a memory barrier, equivalent to the Linux kernel statement `smp_mb()`, and line 11 is equivalent to the C statement `r3=x`.

Quick Quiz 12.24: But whatever happened to line 17

```

1 void P0(void)
2 {
3     int r3;
4
5     x = 1; /* Lines 8 and 9 */
6     atomic_add_return(&z, 0); /* Lines 10-15 */
7     r3 = y; /* Line 16 */
8 }
9
10 void P1(void)
11 {
12     int r3;
13
14     y = 1; /* Lines 8-9 */
15     smp_mb(); /* Line 10 */
16     r3 = x; /* Line 11 */
17 }

```

Figure 12.26: Meaning of PPCMEM Litmus Test

```

./ppcmem -model lwsync_read_block \
          -model coherence_points filename.litmus
...
States 6
0:r3=0; 1:r3=0;
0:r3=0; 1:r3=1;
0:r3=1; 1:r3=0;
0:r3=1; 1:r3=1;
0:r3=2; 1:r3=0;
0:r3=2; 1:r3=1;
Ok
Condition exists (0:r3=0 /\ 1:r3=0)
Hash=e2240ce2072a2610c034cccd4fc964e77
Observation SB+lwsync-RMW-lwsync+isync Sometimes 1

```

Figure 12.27: PPCMEM Detects an Error

of Figure 12.25, the one that is the `Fail: label? ■`

Putting all this together, the C-language equivalent to the entire litmus test is as shown in Figure 12.26. The key point is that if `atomic_add_return()` acts as a full memory barrier (as the Linux kernel requires it to), then it should be impossible for `P0()`'s and `P1()`'s `r3` variables to both be zero after execution completes.

The next section describes how to run this litmus test.

12.2.3 Running a Litmus Test

Although litmus tests may be run interactively via <http://www.cl.cam.ac.uk/~pes20/ppcmem/>, which can help build an understanding of the memory model. However, this approach requires that the user manually carry out the full state-space search. Because it is very difficult to be sure that you have checked every possible sequence of events, a separate tool is provided for this purpose [McK11c].

Because the litmus test shown in Figure 12.25 contains read-modify-write instructions, we must add `-model` ar-

```

./ppcmem -model lwsync_read_block \
          -model coherence_points filename.litmus
...
States 5
0:r3=0; 1:r3=1;
0:r3=1; 1:r3=0;
0:r3=1; 1:r3=1;
0:r3=2; 1:r3=0;
0:r3=2; 1:r3=1;
No (allowed not found)
Condition exists (0:r3=0 /\ 1:r3=0)
Hash=77dd723cda9981248ea4459fcdf6097d
Observation SB+lwsync-RMW-lwsync+sync Never 0 5

```

Figure 12.28: PPCMEM on Repaired Litmus Test

guments to the command line. If the litmus test is stored in `filename.litmus`, this will result in the output shown in Figure 12.27, where the `...` stands for voluminous making-progress output. The list of states includes `0:r3=0; 1:r3=0;`, indicating once again that the old PowerPC implementation of `atomic_add_return()` does not act as a full barrier. The “Sometimes” on the last line confirms this: the assertion triggers for some executions, but not all of the time.

The fix to this Linux-kernel bug is to replace `P0`'s `isync` with `sync`, which results in the output shown in Figure 12.28. As you can see, `0:r3=0; 1:r3=0;` does not appear in the list of states, and the last line calls out “Never”. Therefore, the model predicts that the offending execution sequence cannot happen.

Quick Quiz 12.25: Does the ARM Linux kernel have a similar bug? ■

12.2.4 PPCMEM Discussion

These tools promise to be of great help to people working on low-level parallel primitives that run on ARM and on Power. These tools do have some intrinsic limitations:

1. These tools are research prototypes, and as such are unsupported.
2. These tools do not constitute official statements by IBM or ARM on their respective CPU architectures. For example, both corporations reserve the right to report a bug at any time against any version of any of these tools. These tools are therefore not a substitute for careful stress testing on real hardware. Moreover, both the tools and the model that they are based on are under active development and might change at any time. On the other hand, this model was developed in consultation with the relevant hardware

- experts, so there is good reason to be confident that it is a robust representation of the architectures.
3. These tools currently handle a subset of the instruction set. This subset has been sufficient for my purposes, but your mileage may vary. In particular, the tool handles only word-sized accesses (32 bits), and the words accessed must be properly aligned. In addition, the tool does not handle some of the weaker variants of the ARM memory-barrier instructions, nor does it handle arithmetic.
 4. The tools are restricted to small loop-free code fragments running on small numbers of threads. Larger examples result in state-space explosion, just as with similar tools such as Promela and spin.
 5. The full state-space search does not give any indication of how each offending state was reached. That said, once you realize that the state is in fact reachable, it is usually not too hard to find that state using the interactive tool.
 6. These tools are not much good for complex data structures, although it is possible to create and traverse extremely simple linked lists using initialization statements of the form “`x=y; y=z; z=42;`”.
 7. These tools do not handle memory mapped I/O or device registers. Of course, handling such things would require that they be formalized, which does not appear to be in the offing.
 8. The tools will detect only those problems for which you code an assertion. This weakness is common to all formal methods, and is yet another reason why testing remains important. In the immortal words of Donald Knuth, “Beware of bugs in the above code; I have only proved it correct, not tried it.”

That said, one strength of these tools is that they are designed to model the full range of behaviors allowed by the architectures, including behaviors that are legal, but which current hardware implementations do not yet inflict on unwary software developers. Therefore, an algorithm that is vetted by these tools likely has some additional safety margin when running on real hardware. Furthermore, testing on real hardware can only find bugs; such testing is inherently incapable of proving a given usage correct. To appreciate this, consider that the researchers

routinely ran in excess of 100 billion test runs on real hardware to validate their model. In one case, behavior that is allowed by the architecture did not occur, despite 176 billion runs [AMP⁺11]. In contrast, the full-state-space search allows the tool to prove code fragments correct.

It is worth repeating that formal methods and tools are no substitute for testing. The fact is that producing large reliable concurrent software artifacts, the Linux kernel for example, is quite difficult. Developers must therefore be prepared to apply every tool at their disposal towards this goal. The tools presented in this paper are able to locate bugs that are quite difficult to produce (let alone track down) via testing. On the other hand, testing can be applied to far larger bodies of software than the tools presented in this paper are ever likely to handle. As always, use the right tools for the job!

Of course, it is always best to avoid the need to work at this level by designing your parallel code to be easily partitioned and then using higher-level primitives (such as locks, sequence counters, atomic operations, and RCU) to get your job done more straightforwardly. And even if you absolutely must use low-level memory barriers and read-modify-write instructions to get your job done, the more conservative your use of these sharp instruments, the easier your life is likely to be.

12.3 Axiomatic Approaches

Although the PPCMEM tool can solve the famous “independent reads of independent writes” (IRIW) litmus test shown in Figure 12.29, doing so requires no less than fourteen CPU hours and generates no less than ten gigabytes of state space. That said, this situation is a great improvement over that before the advent of PPCMEM, where solving this problem required perusing volumes of reference manuals, attempting proofs, discussing with experts, and being unsure of the final answer. Although fourteen hours can seem like a long time, it is much shorter than weeks or even months.

However, the time required is a bit surprising given the simplicity of the litmus test, which has two threads storing to two separate variables and two other threads loading from these two variables in opposite orders. The assertion triggers if the two loading threads disagree on the order of the two stores. This litmus test is simple, even by the standards of memory-order litmus tests.

One reason for the amount of time and space consumed is that PPCMEM does a trace-based full-state-space search, which means that it must generate and eval-

```

1 PPC IRIW.litmus
2 ""
3 (* Traditional IRIW. *)
4 {
5 0:r1=1; 0:r2=x;
6 1:r1=1;           1:r4=y;
7           2:r2=x; 2:r4=y;
8           3:r2=x; 3:r4=y;
9 }
10 P0      | P1      | P2      | P3      ;
11 stw r1,0(r2) | stw r1,0(r4) | lwz r3,0(r2) | lwz r3,0(r4) ;
12           |           | sync      | sync      ;
13           |           | lwz r5,0(r4) | lwz r5,0(r2) ;
14
15 exists
16 (2:r3=1 /\ 2:r5=0 /\ 3:r3=1 /\ 3:r5=0)

```

Figure 12.29: IRIW Litmus Test

uate all possible orders and combinations of events at the architectural level. At this level, both loads and stores correspond to ornate sequences of events and actions, resulting in a very large state space that must be completely searched, in turn resulting in large memory and CPU consumption.

Of course, many of the traces are quite similar to one another, which suggests that an approach that treated similar traces as one might improve performance. One such approach is the axiomatic approach of Alglave et al. [AMT14], which creates a set of axioms to represent the memory model and then converts litmus tests to theorems that might be proven or disproven over this set of axioms. The resulting tool, called “herd,” conveniently takes as input the same litmus tests as PPCMEM, including the IRIW litmus test shown in Figure 12.29.

However, where PPCMEM requires 14 CPU hours to solve IRIW, herd does so in 17 milliseconds, which represents a speedup of more than six orders of magnitude. That said, the problem is exponential in nature, so we should expect herd to exhibit exponential slowdowns for larger problems. And this is exactly what happens, for example, if we add four more writes per writing CPU as shown in Figure 12.30, herd slows down by a factor of more than 50,000, requiring more than 15 *minutes* of CPU time. Adding threads also results in exponential slowdowns [MS14].

Despite their exponential nature, both PPCMEM and herd have proven quite useful for checking key parallel algorithms, including the queued-lock handoff on x86 systems. The weaknesses of the herd tool are similar to those of PPCMEM, which were described in Section 12.2.4. There are some obscure (but very real) cases for which the PPCMEM and herd tools disagree, and as of late 2014

resolving these disagreements was ongoing.

Longer term, the hope is that the axiomatic approaches incorporate axioms describing higher-level software artifacts. This could potentially allow axiomatic verification of much larger software systems. Another alternative is to press the axioms of boolean logic into service, as described in the next section.

12.4 SAT Solvers

Any finite program with bounded loops and recursion can be converted into a logic expression, which might express that program’s assertions in terms of its inputs. Given such a logic expression, it would be quite interesting to know whether any possible combinations of inputs could result in one of the assertions triggering. If the inputs are expressed as combinations of boolean variables, this is simply SAT, also known as the satisfiability problem. SAT solvers are heavily used in verification of hardware, which has motivated great advances. A world-class early 1990s SAT solver might be able to handle a logic expression with 100 distinct boolean variables, but by the early 2010s million-variable SAT solvers were readily available [KS08].

In addition, front-end programs for SAT solvers can automatically translate C code into logic expressions, taking assertions into account and generating assertions for error conditions such as array-bounds errors. One example is the C bounded model checker, or `cbmc`, which is available as part of many Linux distributions. This tool is quite easy to use, with `cbmc test.c` sufficing to validate `test.c`. This ease of use is exceedingly important because it opens the door to formal verification being incorporated into regression-testing frameworks.

```

1 PPC IRIW5.litmus
2 """
3 (* Traditional IRIW, but with five stores instead of just one. *)
4 {
5 0:r1=1; 0:r2=x;
6 1:r1=1;           1:r4=y;
7           2:r2=x; 2:r4=y;
8           3:r2=x; 3:r4=y;
9 }
10 P0      | P1      | P2      | P3      ;
11 stw r1,0(r2) | stw r1,0(r4) | lwz r3,0(r2) | lwz r3,0(r4) ;
12 addi r1,r1,1 | addi r1,r1,1 | sync      | sync      ;
13 stw r1,0(r2) | stw r1,0(r4) | lwz r5,0(r4) | lwz r5,0(r2) ;
14 addi r1,r1,1 | addi r1,r1,1 |           |           ;
15 stw r1,0(r2) | stw r1,0(r4) |           |           ;
16 addi r1,r1,1 | addi r1,r1,1 |           |           ;
17 stw r1,0(r2) | stw r1,0(r4) |           |           ;
18 addi r1,r1,1 | addi r1,r1,1 |           |           ;
19 stw r1,0(r2) | stw r1,0(r4) |           |           ;
20
21 exists
22 (2:r3=1 /\ 2:r5=0 /\ 3:r3=1 /\ 3:r5=0)

```

Figure 12.30: Expanded IRIW Litmus Test

In contrast, the traditional tools that require non-trivial translation to a special-purpose language are confined to design-time verification.

More recently, SAT solvers have appeared that handle parallel code. These solvers operate by converting the input code into single static assignment (SSA) form, then generating all permitted access orders. This approach seems promising, but it remains to be seen how well it works in practice. For example, it is not clear what types and sizes of programs this technique handles. However, there is some reason to hope that SAT solvers will be useful for verifying parallel code.

12.5 Summary

The formal-verification techniques described in this chapter are very powerful tools for validating small parallel algorithms, but they should not be the only tools in your toolbox. Despite decades of focus on formal verification, testing remains the validation workhorse for large parallel software systems [Cor06a, Jon11].

It is nevertheless quite possible that this will not always be the case. To see this, consider that there are more than one billion instances of the Linux kernel as of 2013. Suppose that the Linux kernel has a bug that manifests on average every million years of runtime. As noted at the end of the preceding chapter, this bug will be appearing three times *per day* across the installed base. But the fact remains that most formal validation techniques can be used only on very small code bases. So what is a

concurrency coder to do?

One approach is to think in terms of finding the first bug, the first relevant bug, the last relevant bug, and the last bug.

The first bug is normally found via inspection or compiler diagnostics. Although the increasingly sophisticated diagnostics provided by modern compilers might be considered to be a lightweight sort of formal verification, it is not common to think of them in those terms. This is in part due to an odd practitioner prejudice which says “If I am using it, it cannot be formal verification” on the one hand, and the large difference in sophistication between compiler diagnostics and verification research on the other.

Although the first relevant bug might be located via inspection or compiler diagnostics, it is not unusual for these two steps to find only typos and false positives. Either way, the bulk of the relevant bugs, that is, those bugs that might actually be encountered in production, will often be found via testing.

When testing is driven by anticipated or real use cases, it is not uncommon for the last relevant bug to be located by testing. This situation might motivate a complete rejection of formal verification, however, irrelevant bugs have a bad habit of suddenly becoming relevant at the least convenient moment possible, courtesy of black-hat attacks. For security-critical software, which appears to be a continually increasing fraction of the total, there can thus be strong motivation to find and fix the last bug. Testing is demonstrably unable to find the last bug, so there

is a possible role for formal verification. That is, there is such a role if and only if formal verification proves capable of growing into it. As this chapter has shown, current formal verification systems are extremely limited.

Another approach is to consider that formal verification is often much harder to use than is testing. This is of course in part a cultural statement, and there is every reason to hope that formal verification will be perceived to be easier as more people become familiar with it. That said, very simple test harnesses can find significant bugs in arbitrarily large software systems. In contrast, the effort required to apply formal verification seems to increase dramatically as the system size increases.

I have nevertheless made occasional use of formal verification for more than 20 years, playing to formal verification's strengths, namely design-time verification of small complex portions of the overarching software construct. The larger overarching software construct is of course validated by testing.

Quick Quiz 12.26: In light of the full verification of the L4 microkernel, isn't this limited view of formal verification just a little bit obsolete? ■

One final approach is to consider the following two axioms and the theorem that they immediately imply:

Axiom: The only bug-free programs are trivial programs.

Axiom: Reliable programs have no known bugs.

Theorem: Any non-trivial reliable program contains at least one as-yet-unknown bug.

From this viewpoint, any advances in validation and verification can have but two effects: (1) An increase in the number of trivial programs or (2) A decrease in the number of reliable programs. Of course, the human race's increasing reliance on multicore systems and software provides extreme motivation for a very sharp increase in the number of trivial programs!

However, if your code is so complex that you find yourself relying too heavily on formal-verification tools, you should carefully rethink your design, especially if your formal-verification tools require your code to be hand-translated to a special-purpose language. For example, a complex implementation of the dynticks interface for preemptible RCU that was presented in Section 12.1.5 turned out to have a much simpler alternative implementation, as discussed in Section 12.1.6.9. All else being equal, a simpler implementation is much better than a mechanical proof for a complex implementation!

And the open challenge to those working on formal verification techniques and systems is prove this summary wrong!

Chapter 13

Putting It All Together

This chapter gives a few hints on handling some concurrent-programming puzzles, starting with counter conundrums in Section 13.1, continuing with some RCU rescues in Section 13.2, and finishing off with some hashing hassles in Section 13.3.

13.1 Counter Conundrums

This section outlines possible solutions to some counter conundrums.

13.1.1 Counting Updates

Suppose that Schrödinger (see Section 10.1) wants to count the number of updates for each animal, and that these updates are synchronized using a per-data-element lock. How can this counting best be done?

Of course, any number of counting algorithms from Chapter 5 might be considered, but the optimal approach is much simpler in this case. Just place a counter in each data element, and increment it under the protection of that element’s lock!

13.1.2 Counting Lookups

Suppose that Schrödinger also wants to count the number of lookups for each animal, where lookups are protected by RCU. How can this counting best be done?

One approach would be to protect a lookup counter with the per-element lock, as discussed in Section 13.1.1. Unfortunately, this would require all lookups to acquire this lock, which would be a severe bottleneck on large systems.

Another approach is to “just say no” to counting, following the example of the `noatime` mount option. If this approach is feasible, it is clearly the best: After all,

nothing is faster than doing nothing. If the lookup count cannot be dispensed with, read on!

Any of the counters from Chapter 5 could be pressed into service, with the statistical counters described in Section 5.2 being perhaps the most common choice. However, this results in a large memory footprint: The number of counters required is the number of data elements multiplied by the number of threads.

If this memory overhead is excessive, then one approach is to keep per-socket counters rather than per-CPU counters, with an eye to the hash-table performance results depicted in Figure 10.8. This will require that the counter increments be atomic operations, especially for user-mode execution where a given thread could migrate to another CPU at any time.

If some elements are looked up very frequently, there are a number of approaches that batch updates by maintaining a per-thread log, where multiple log entries for a given element can be merged. After a given log entry has a sufficiently large increment or after sufficient time has passed, the log entries may be applied to the corresponding data elements. Silas Boyd-Wickizer has done some work formalizing this notion [BW14].

13.2 RCU Rescues

This section 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.

13.2.1 RCU and Per-Thread-Variable-Based Statistical Counters

Section 5.2.4 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 5.9 on Page 41.

Quick Quiz 13.1: Why on earth did we need that global lock in the first place? ■

13.2.1.1 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 13.2: 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.

13.2.1.2 Implementation

Lines 1-4 of Figure 13.1 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.

```

1 struct countarray {
2     unsigned long total;
3     unsigned long *counterp[NR_THREADS];
4 };
5
6 long __thread counter = 0;
7 struct countarray *countarrayp = NULL;
8 DEFINE_SPINLOCK(final_mutex);
9
10 void inc_count(void)
11 {
12     counter++;
13 }
14
15 long read_count(void)
16 {
17     struct countarray *cap;
18     unsigned long sum;
19     int t;
20
21     rCU_read_lock();
22     cap = rCU_dereference(countarrayp);
23     sum = cap->total;
24     for_each_thread(t)
25         if (cap->counterp[t] != NULL)
26             sum += *cap->counterp[t];
27     rCU_read_unlock();
28     return sum;
29 }
30
31 void count_init(void)
32 {
33     countarrayp = malloc(sizeof(*countarrayp));
34     if (countarrayp == NULL) {
35         fprintf(stderr, "Out of memory\n");
36         exit(-1);
37     }
38     memset(countarrayp, '\0', sizeof(*countarrayp));
39 }
40
41 void count_register_thread(void)
42 {
43     int idx = smp_thread_id();
44
45     spin_lock(&final_mutex);
46     countarrayp->counterp[idx] = &counter;
47     spin_unlock(&final_mutex);
48 }
49
50 void count_unregister_thread(int nthreadsexpected)
51 {
52     struct countarray *cap;
53     struct countarray *capold;
54     int idx = smp_thread_id();
55
56     cap = malloc(sizeof(*countarrayp));
57     if (cap == NULL) {
58         fprintf(stderr, "Out of memory\n");
59         exit(-1);
60     }
61     spin_lock(&final_mutex);
62     *cap = *countarrayp;
63     cap->total += counter;
64     cap->counterp[idx] = NULL;
65     capold = countarrayp;
66     rCU_assign_pointer(countarrayp, cap);
67     spin_unlock(&final_mutex);
68     synchronize_rcu();
69     free(capold);
70 }
```

Figure 13.1: RCU and Per-Thread Statistical Counters

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 5.9.

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 initializes `sum` to `cap->total`, which is the sum of the counts of threads that have previously exited. Lines 24-26 add up the per-thread counters corresponding to currently running threads, and, finally, line 28 returns the sum.

The initial value for `countarrayp` is provided by `count_init()` on lines 31-39. This function runs before the first thread is created, and its job is to allocate and zero the initial structure, and then assign it to `countarrayp`.

Lines 41-48 show the `count_register_thread()` function, which is invoked by each newly created thread. Line 43 picks up the current thread's index, line 45 acquires `final_mutex`, line 46 installs a pointer to this thread's counter, and line 47 releases `final_mutex`.

Quick Quiz 13.3: Hey!!! Line 45 of Figure 13.1 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.

13.2.1.3 Discussion

Quick Quiz 13.4: Wow! Figure 13.1 contains 69 lines of code, compared to only 42 in Figure 5.9. 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.

13.2.2 RCU and Counters for Removable I/O Devices

Section 5.5 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 a RCU read-side critical section being substituted for the reader-writer lock read-side critical section in the original:

```

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.

```

1 struct foo {
2     int length;
3     char *a;
4 };

```

Figure 13.2: RCU-Protected Variable-Length Array

The updated code fragment removing a device is as follows:

```

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();

```

Here we replace the reader-writer lock with an exclusive spinlock and add a `synchronize_rcu()` to wait for all of the RCU read-side critical sections to complete. 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.

13.2.3 Array and Length

Suppose we have an RCU-protected variable-length array, as shown in Figure 13.2. The length of the array `->a[]` can change dynamically, and at any given time, its length is given by the field `->length`. Of course, this introduces the following race condition:

1. The array is initially 16 characters long, and thus `->length` is equal to 16.
2. CPU 0 loads the value of `->length`, obtaining the value 16.
3. CPU 1 shrinks the array to be of length 8, and assigns a pointer to a new 8-character block of memory into `->a[]`.
4. CPU 0 picks up the new pointer from `->a[]`, and stores a new value into element 12. Because the array has only 8 characters, this results in a SEGV or (worse yet) memory corruption.

```

1 struct foo_a {
2     int length;
3     char a[0];
4 };
5
6 struct foo {
7     struct foo_a *fa;
8 };

```

Figure 13.3: Improved RCU-Protected Variable-Length Array

How can we prevent this?

One approach is to make careful use of memory barriers, which are covered in Section 14.2. This works, but incurs read-side overhead and, perhaps worse, requires use of explicit memory barriers.

A better approach is to put the value and the array into the same structure, as shown in Figure 13.3. Allocating a new array (`foo_a` structure) then automatically provides a new place for the array length. This means that if any CPU picks up a reference to `->fa`, it is guaranteed that the `->length` will match the `->a[]` length [ACMS03].

1. The array is initially 16 characters long, and thus `->length` is equal to 16.
2. CPU 0 loads the value of `->fa`, obtaining a pointer to the structure containing the value 16 and the 16-byte array.
3. CPU 0 loads the value of `->fa->length`, obtaining the value 16.
4. CPU 1 shrinks the array to be of length 8, and assigns a pointer to a new `foo_a` structure containing an 8-character block of memory into `->a[]`.
5. CPU 0 picks up the new pointer from `->a[]`, and stores a new value into element 12. But because CPU 0 is still referencing the old `foo_a` structure that contains the 16-byte array, all is well.

Of course, in both cases, CPU 1 must wait for a grace period before freeing the old array.

A more general version of this approach is presented in the next section.

13.2.4 Correlated Fields

Suppose that each of Schrödinger's animals is represented by the data element shown in Figure 13.4. The `meas_1`, `meas_2`, and `meas_3` fields are a set of correlated

```

1 struct animal {
2     char name[40];
3     double age;
4     double meas_1;
5     double meas_2;
6     double meas_3;
7     char photo[0]; /* large bitmap. */
8 };

```

Figure 13.4: Uncorrelated Measurement Fields

```

1 struct measurement {
2     double meas_1;
3     double meas_2;
4     double meas_3;
5 };
6
7 struct animal {
8     char name[40];
9     double age;
10    struct measurement *mp;
11    char photo[0]; /* large bitmap. */
12 };

```

Figure 13.5: Correlated Measurement Fields

measurements that are updated periodically. It is critically important that readers see these three values from a single measurement update: If a reader sees an old value of `meas_1` but new values of `meas_2` and `meas_3`, that reader will become fatally confused. How can we guarantee that readers will see coordinated sets of these three values?

One approach would be to allocate a new animal structure, copy the old structure into the new structure, update the new structure's `meas_1`, `meas_2`, and `meas_3` fields, and then replace the old structure with a new one by updating the pointer. This does guarantee that all readers see coordinated sets of measurement values, but it requires copying a large structure due to the `->photo[]` field. This copying might incur unacceptably large overhead.

Another approach is to insert a level of indirection, as shown in Figure 13.5. When a new measurement is taken, a new measurement structure is allocated, filled in with the measurements, and the animal structure's `->mp` field is updated to point to this new measurement structure using `rcu_assign_pointer()`. After a grace period elapses, the old measurement structure can be freed.

Quick Quiz 13.5: But can't the approach shown in Figure 13.5 result in extra cache misses, in turn resulting in additional read-side overhead? ■

This approach enables readers to see correlated values for selected fields with minimal read-side overhead.

13.3 Hashing Hassles

This section looks at some issues that can arise when dealing with hash tables. Please note that these issues also apply to many other search structures.

13.3.1 Correlated Data Elements

This situation is analogous to that in Section 13.2.4: We have a hash table where we need correlated views of two or more of the elements. These elements are updated together, and we do not want to see an old version of the first element along with new versions of the other elements. For example, Schrödinger decided to add his extended family to his in-memory database along with all his animals. Although Schrödinger understands that marriages and divorces do not happen instantaneously, he is also a traditionalist. As such, he absolutely does not want his database ever to show that the bride is now married, but the groom is not, and vice versa. In other words, Schrödinger wants to be able to carry out a wedlock-consistent traversal of his database.

One approach is to use sequence locks (see Section 9.2), so that wedlock-related updates are carried out under the protection of `write_seqlock()`, while reads requiring wedlock consistency are carried out within a `read_seqbegin() / read_seqretry()` loop. Note that sequence locks are not a replacement for RCU protection: Sequence locks protect against concurrent modifications, but RCU is still needed to protect against concurrent deletions.

This approach works quite well when the number of correlated elements is small, the time to read these elements is short, and the update rate is low. Otherwise, updates might happen so quickly that readers might never complete. Although Schrödinger does not expect that even his least-sane relatives will marry and divorce quickly enough for this to be a problem, he does realize that this problem could well arise in other situations. One way to avoid this reader-starvation problem is to have the readers use the update-side primitives if there have been too many retries, but this can degrade both performance and scalability.

In addition, if the update-side primitives are used too frequently, poor performance and scalability will result due to lock contention. One way to avoid this is to maintain a per-element sequence lock, and to hold both spouses' locks when updating their marital status. Readers can do their retry looping on either of the spouses' locks to gain a stable view of any change in marital status in-

volving both members of the pair. This avoids contention due to high marriage and divorce rates, but complicates gaining a stable view of all marital statuses during a single scan of the database.

If the element groupings are well-defined and persistent, which marital status is hoped to be, then one approach is to add pointers to the data elements to link together the members of a given group. Readers can then traverse these pointers to access all the data elements in the same group as the first one located.

Other approaches using version numbering are left as exercises for the interested reader.

13.3.2 Update-Friendly Hash-Table Traversal

Suppose that a statistical scan of all elements in a hash table is required. For example, Schrödinger might wish to compute the average length-to-weight ratio over all of his animals.¹ Suppose further that Schrödinger is willing to ignore slight errors due to animals being added to and removed from the hash table while this statistical scan is being carried out. What should Schrödinger do to control concurrency?

One approach is to enclose the statistical scan in an RCU read-side critical section. This permits updates to proceed concurrently without unduly impeding the scan. In particular, the scan does not block the updates and vice versa, which allows scan of hash tables containing very large numbers of elements to be supported gracefully, even in the face of very high update rates.

Quick Quiz 13.6: But how does this scan work while a resizable hash table is being resized? In that case, neither the old nor the new hash table is guaranteed to contain all the elements in the hash table! ■

¹ Why would such a quantity be useful? Beats me! But group statistics in general are often useful.

Chapter 14

Advanced Synchronization

This section discusses a number of ways of using weaker, and hopefully lower-cost, synchronization primitives. This weakening can be quite helpful, in fact, some have argued that weakness is a virtue [Alg13]. Nevertheless, in parallel programming, as in many other aspects of life, weakness is not a panacea. For example, as noted at the end of Chapter 5, you should thoroughly apply partitioning, batching, and well-tested packaged weak APIs (see Chapter 8 and 9) before even thinking about unstructured weakening.

But after doing all that, you still might find yourself needing the advanced techniques described in this chapter. To that end, Section 14.1 summarizes techniques used thus far for avoiding locks, Section 14.2 covers use of memory barriers, and finally Section 14.3 gives a brief overview of non-blocking synchronization.

14.1 Avoiding Locks

Although locking is the workhorse of parallelism in production, in many situations performance, scalability, and real-time response can all be greatly improved though use of lockless techniques. A particularly impressive example of such a lockless technique are the statistical counters described in Section 5.2, which avoids not only locks, but also atomic operations, memory barriers, and even cache misses for counter increments. Other examples we have covered include:

1. The fastpaths through a number of other counting algorithms in Chapter 5.
2. The fastpath through resource allocator caches in Section 6.4.3.
3. The maze solver in Section 6.5.

4. The data-ownership techniques described in Section 8.
5. The reference-counting and RCU techniques described in Chapter 9.
6. The lookup code paths described in Chapter 10.
7. Many of the techniques described in Chapter 13.

In short, lockless techniques are quite useful and are heavily used.

However, it is best if lockless techniques are hidden behind a well-defined API, such as the `inc_count()`, `memblock_alloc()`, `rcu_read_lock()`, and so on. The reason for this is that undisciplined use of lockless techniques is a good way to create difficult bugs.

A key component of many lockless techniques is the memory barrier, which is described in the following section.

14.2 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. For example, the litmus test in Table 14.1 appears to guarantee that the assertion never fires. After all, if `r1 != 1`, we might

Thread 1	Thread 2
<code>x = 1;</code>	<code>y = 1;</code>
<code>r1 = y;</code>	<code>r2 = x;</code>
<code>assert(r1 == 1 r2 == 1);</code>	

Table 14.1: Memory Misordering: Dekker

hope that Thread 1’s load from `y` must have happened before Thread 2’s store to `y`, which might raise further hopes that Thread 2’s load from `x` must happen after Thread 1’s store to `x`, so that `r2 == 1`, as required by the assertion. The example is symmetric, so similar hopeful reasoning might lead us to hope that `r2 != 1` guarantees that `r1 == 1`. Unfortunately, the lack of memory barriers in Table 14.1 dashes these hopes. Both the compiler and the CPU are within their rights to reorder the statements within both Thread 1 and Thread 2, even on relatively strongly ordered systems such as x86.

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 14.2.1 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 14.2.2, which shows an intuitively correct code fragment that fails miserably on real hardware, and by Section 14.2.3, 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 14.2.4 provides the basic rules that memory barriers follow, rules that we will build upon. These rules are further refined in Sections 14.2.5 through 14.2.14.

14.2.1 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 the time required to fetch a single variable

from memory. CPUs therefore sport increasingly large caches, as shown in Figure 14.1. 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.

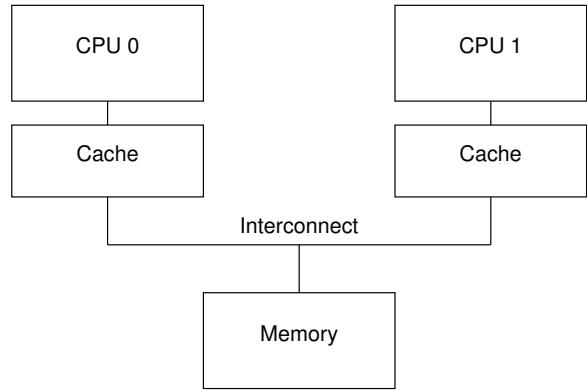


Figure 14.1: Modern Computer System Cache Structure

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 14.2. 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.

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.

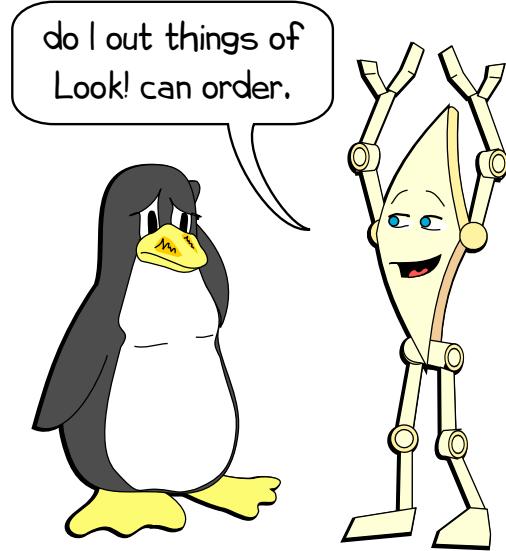


Figure 14.2: CPUs Can Do Things Out of Order

14.2.2 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 14.3 executing in parallel where variables A, B, and C are initially zero:

```

1 thread0(void)
2 {
3     A = 1;
4     smp_wmb();
5     B = 1;
6 }
7
8 thread1(void)
9 {
10    while (B != 1)
11        continue;
12    barrier();
13    C = 1;
14 }
15
16 thread2(void)
17 {
18    while (C != 1)
19        continue;
20    barrier();
21    assert(A != 0);
22 }
```

Figure 14.3: 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 14.1: How on earth could the assertion on line 21 of the code in Figure 14.3 on page 243 *possibly* fail? ■

Quick Quiz 14.2: 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.

14.2.3 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 14.4. 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 a 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 6-8.

Quick Quiz 14.3: What assumption is the code fragment in Figure 14.4 making that might not be valid on real hardware? ■

```

1 state.variable = mycpu;
2 lasttbb = oldtb = firsttbb = gettbb();
3 while (state.variable == mycpu) {
4     lasttbb = oldtb;
5     oldtb = gettbb();
6     if (lasttbb - firsttbb > 1000)
7         break;
8 }
```

Figure 14.4: Software Logic Analyzer

Upon exit from the loop, `firsttbb` will hold a timestamp taken shortly after the assignment and `lasttbb` will hold a timestamp taken before the last sampling of the shared variable that still retained the assigned value, or a value equal to `firsttbb` if the shared variable had changed before entry into the loop. This allows us to plot each CPU’s view of the value of `state.variable` over a 532-nanosecond time period, as shown in Figure 14.5. This data was collected on 1.5GHz POWER5 system with 8 cores, each containing a pair of hardware threads. CPUs 1, 2, 3, and 4 recorded the values, while CPU 0 controlled the test. The timebase counter period was about 5.32ns, sufficiently fine-grained to allow observations of intermediate cache states.

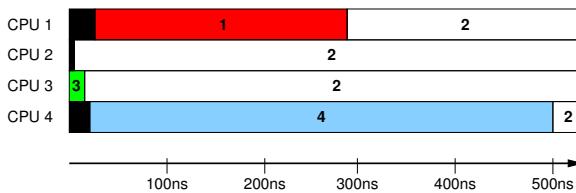


Figure 14.5: A Variable With Multiple Simultaneous Values

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 14.4: How could CPUs possibly have different views of the value of a single variable *at the same time*? ■

Quick Quiz 14.5: 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 bid a fond farewell to comfortable intuitions about values of variables and the passage of time. This is the regime where memory barriers are needed.

14.2.4 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 14.2.6.

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.¹ Each of these properties is described in the following sections.

¹ Or, better yet, you can avoid explicit use of memory barriers entirely. But that would be the subject of other sections.

14.2.4.1 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 user 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.

14.2.4.2 Single-Variable Memory Consistency

Because current commercially available computer systems provide *cache coherence*, if a group of CPUs all do concurrent non-atomic 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 14.5, 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 sequences of these four numbers that end in “2”. Thus, there will be agreement on the sequence of values taken on by a single variable, but there might be ambiguity.

In contrast, 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. One of the `atomic_inc_return()` invocations would happen first, and would change the value from 0 to 1, the second from 1 to 2, and so on. The CPUs could compare notes afterwards and come to agreement on the exact ordering of the sequence of `atomic_inc_return()` invocations. This does not work for the non-atomic stores described earlier because the non-atomic stores do not return any indication of the earlier value, hence the possibility of ambiguity.

Please note well that this section applies *only* when all CPUs’ accesses are to one single variable. In this single-variable case, cache coherence guarantees the global ordering, at least assuming that some of the more aggressive compiler optimizations are disabled via the Linux kernel’s `ACCESS_ONCE()` directive or C++11’s relaxed atomics [Bec11]. In contrast, if there are multiple variables,

memory barriers are required for the CPUs to consistently agree on the order for current commercially available computer systems.

14.2.4.3 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
<code>access (A);</code>	<code>access (B);</code>
<code>smp_mb();</code>	<code>smp_mb();</code>
<code>access (B);</code>	<code>access (A);</code>

Quick Quiz 14.6: 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 14.2 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.

14.2.4.4 Pair-Wise Memory Barriers: Portable Combinations

The following pairings from Table 14.2, 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
<code>A=1;</code>	<code>Y=B;</code>
<code>smp_mb();</code>	<code>smp_mb();</code>
<code>B=1;</code>	<code>X=A;</code>

After both CPUs have completed executing these code sequences, if $Y==1$, then we must also have $X==1$. In this

	CPU 1		CPU 2		Description
0	load(A)	load(B)	load(B)	load(A)	Ears to ears.
1	load(A)	load(B)	load(B)	store(A)	Only one store.
2	load(A)	load(B)	store(B)	load(A)	Only one store.
3	load(A)	load(B)	store(B)	store(A)	Pairing 1.
4	load(A)	store(B)	load(B)	load(A)	Only one store.
5	load(A)	store(B)	load(B)	store(A)	Pairing 2.
6	load(A)	store(B)	store(B)	load(A)	Mouth to mouth, ear to ear.
7	load(A)	store(B)	store(B)	store(A)	Pairing 3.
8	store(A)	load(B)	load(B)	load(A)	Only one store.
9	store(A)	load(B)	load(B)	store(A)	Mouth to mouth, ear to ear.
A	store(A)	load(B)	store(B)	load(A)	Ears to mouths.
B	store(A)	load(B)	store(B)	store(A)	Stores “pass in the night”.
C	store(A)	store(B)	load(B)	load(A)	Pairing 1.
D	store(A)	store(B)	load(B)	store(A)	Pairing 3.
E	store(A)	store(B)	store(B)	load(A)	Stores “pass in the night”.
F	store(A)	store(B)	store(B)	store(A)	Stores “pass in the night”.

Table 14.2: Memory-Barrier Combinations

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 $X==1$.

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.

14.2.4.5 Pair-Wise Memory Barriers: Semi-Portable Combinations

The following pairings from Table 14.2 can be used on modern hardware, but might fail on some systems that were produced in the 1900s. However, these *can* safely be used on all mainstream hardware introduced since the year 2000. So if you think that memory barriers are difficult to deal with, please keep in mind that they used to be a *lot* harder on some systems!

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, 21st-century 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).

Quick Quiz 14.7: How do we know that modern hardware guarantees that at least one of the loads will see the value stored by the other thread in the ears-to-mouths scenario? ■

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.

CPU 1	CPU 2
<code>A=1;</code>	<code>B=2;</code>
<code>smp_mb();</code>	<code>smp_mb();</code>
<code>B=1;</code>	<code>A=2;</code>

Unfortunately, although this conclusion is correct on 21st-century systems, it does not necessarily hold on all antique 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 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 these ordering intuitions, and *do* permit this combination to be used safely.

14.2.4.6 Pair-Wise Memory Barriers: Dubious Combinations

In the following combinations from Table 14.2, the memory barriers have very limited use in portable code, even on 21st-century hardware. However, “limited use” is different than “no use”, so let’s see what can be done! Avid readers will want to write toy programs that rely on each of these combinations in order to fully understand how this works.

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. However, if we know that CPU 2’s load from B returned a newer value than CPU 1’s load from B, then we also know that CPU 2’s load from A returned either the same value as CPU 1’s load from A or some later value.

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.

However, it is possible to determine which store happened last, but this requires an additional load from B. If this additional load from B is executed after both CPUs 1 and 2 complete, and if it turns out that CPU 2’s store to B happened last, then we know that CPU 2’s load from A returned either the same value as CPU 1’s load from A or some later value.

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.

At least not straightforwardly. But suppose that in combination 1 from Table 14.2, CPU 1’s load from A returns the value that CPU 2 stored to A. Then we know that CPU 1’s load from B returned either the same value as CPU 2’s load from A or some later value.

Quick Quiz 14.8: How can the other “Only one store” entries in Table 14.2 be used? ■

14.2.4.7 Semantics Sufficient to Implement Locking

Suppose we have an exclusive lock (`spinlock_t` in the Linux kernel, `pthread_mutex_t` in `pthread` 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 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.

Why are these properties necessary?

Suppose the first property did not hold. Then the assertion in the following code might well fail!

```
a = 1;
b = 1 + a;
assert(b == 2);
```

Quick Quiz 14.9: How could the assertion `b==2` on page 248 possibly fail? ■

Suppose that the second property did not hold. Then the following code might leak memory!

```
spin_lock(&mylock);
if (p == NULL)
    p = kmalloc(sizeof(*p), GFP_KERNEL);
spin_unlock(&mylock);
```

Quick Quiz 14.10: How could the code on page 248 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 true with pairwise memory barriers.

```
spin_lock(&mylock);
ctr = ctr + 1;
spin_unlock(&mylock);
```

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.

Quick Quiz 14.11: How could the code on page 248 possibly count backwards? ■

If you are convinced that these rules are necessary, let's look at how they interact with a typical locking implementation.

14.2.5 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, and that the implicit barrier after the atomic exchange operation 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 14.2.4.

```
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:

CPU 1	CPU 2
(critical section)	atomic_xchg(&lck->a, 1) -> 1
smp_mb();	lck->a->1
lck->a=0;	lck->a->1
	lck->a->0
	(implicit smp_mb(1))
	atomic_xchg(&lck->a, 1) -> 0
	(implicit smp_mb(2))
	(critical section)

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.

14.2.6 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 CPU's series will be consistent.³
3. If one CPU does ordered stores to variables A and B,⁴ and if a second CPU does ordered loads from B and A,⁵ 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 to A, and if the first CPU's load from A gives the value stored by the second CPU, then the first 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

The next section takes a more operational view of these rules.

14.2.7 Abstract Memory Access Model

Consider the abstract model of the system shown in Figure 14.6.

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.

⁴ For example, by executing the store to A, a memory barrier, and then the store to B.

⁵ For example, by executing the load from B, a memory barrier, and then the load from A.

⁶ Or, for the more competitively oriented, the first CPU's store to B "wins".

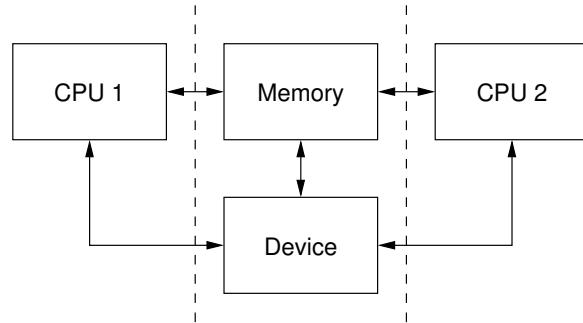


Figure 14.6: Abstract Memory Access Model

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 $\{A = 1, B = 2\}$:

CPU 1	CPU 2
$A = 3;$	$x = A;$
$B = 4;$	$y = B;$

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":

$st\ A=3,$	$st\ B=4,$	$x=ld\ A\rightarrow 3,$	$y=ld\ B\rightarrow 4$
$st\ A=3,$	$st\ B=4,$	$y=ld\ B\rightarrow 4,$	$x=ld\ A\rightarrow 3$
$st\ A=3,$	$x=ld\ A\rightarrow 3,$	$st\ B=4,$	$y=ld\ B\rightarrow 4$
$st\ A=3,$	$x=ld\ A\rightarrow 3,$	$y=ld\ B\rightarrow 2,$	$st\ B=4$
$st\ A=3,$	$y=ld\ B\rightarrow 2,$	$st\ B=4,$	$x=ld\ A\rightarrow 3$
$st\ A=3,$	$y=ld\ B\rightarrow 2,$	$x=ld\ A\rightarrow 3,$	$st\ B=4$
$st\ B=4,$	$st\ A=3,$	$x=ld\ A\rightarrow 3,$	$y=ld\ B\rightarrow 4$
$st\ B=4,$	$...$		

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 $\{A = 1, B = 2, C = 3, P = \&A, Q = \&C\}$:

CPU 1	CPU 2
$B = 4;$	$Q = P;$
$P = \&B$	$D = *Q;$

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:

$(Q == \&A) \text{ and } (D == 1)$
 $(Q == \&B) \text{ and } (D == 2)$
 $(Q == \&B) \text{ and } (D == 4)$

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 $*Q$.

14.2.8 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:

```
*A = 5;
x = *D;
```

but this might show up as either of the following two sequences:

```
STORE *A = 5, x = LOAD *D
x = LOAD *D, STORE *A = 5
```

the second of which will almost certainly result in a malfunction, since it set the address *after* attempting to read the register.

14.2.9 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:

```
Q = P; D = *Q;
```

the CPU will issue the following memory operations:

```
Q = LOAD P, D = LOAD *Q
```

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:

```
a = *X; *X = b;
```

the CPU will only issue the following sequence of memory operations:

```
a = LOAD *X, STORE *X = b
```

And for:

```
*X = c; d = *X;
```

the CPU will only issue:

```
STORE *X = c, d = LOAD *X
```

(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, though 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:

```
X = *A; Y = *B; *D = Z;
```

we may get any of the following sequences:

```

X = LOAD *A,      Y = LOAD *B,      STORE *D = Z
X = LOAD *A,      STORE *D = Z,    Y = LOAD *B
Y = LOAD *B,      X = LOAD *A,      STORE *D = Z
Y = LOAD *B,      STORE *D = Z,    X = LOAD *A
STORE *D = Z,     X = LOAD *A,      Y = LOAD *B
STORE *D = Z,     Y = LOAD *B,      X = LOAD *A

```

2. It *must* be assumed that overlapping memory accesses may be merged or discarded. This means that for:

```
X = *A; Y = *(A + 4);
```

we may get any one of the following sequences:

```

X = LOAD *A; Y = LOAD *(A + 4);
Y = LOAD *(A + 4); X = LOAD *A;
{X, Y} = LOAD {*A, *(A + 4)};

```

And for:

```
*A = X; Y = *A;
```

we may get any of:

```

STORE *A = X; STORE *(A + 4) = Y;
STORE *(A + 4) = Y; STORE *A = X;
STORE {*A, *(A + 4)} = {X, Y};

```

Finally, for:

```
*A = X; *A = Y;
```

we may get either of:

```

STORE *A = X; STORE *A = Y;
STORE *A = Y;

```

14.2.10 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.

14.2.10.1 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 inter-dependent 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.

14.2.10.2 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 14.12: What effect does the following sequence have on the order of stores to variables “a” and “b”?

```
a = 1;  
b = 1;  
<write barrier> ■
```

14.2.10.3 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 hardware⁷ 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.

14.2.10.4 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	CPU 2
$B = 4;$ <write barrier> $P = \&B;$	$Q = P;$ $D = *Q;$

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:

$(Q == \&A) \text{ implies } (D == 1)$
 $(Q == \&B) \text{ implies } (D == 4)$

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:

$(Q == \&B) \text{ and } (D == 2) \text{ ????}$

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 $\{A = 1, B = 2, C = 3, P = \&A, Q = \&C\}$):

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 [Tor03].

CPU 1	CPU 2
$B = 4;$ <write barrier> $P = \&B;$	$Q = P;$ <data dependency barrier> $D = *Q;$

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 2).

Another example of where data dependency barriers might be required is where a number is read from memory and then used to calculate the index for an array access with initial values $\{M[0] = 1, M[1] = 2, M[3] = 3, P = 0, Q = 3\}$:

CPU 1	CPU 2
$M[1] = 4;$ <write barrier> $P = 1;$	$Q = P;$ <data dependency barrier> $D = M[Q];$

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 Section 14.2.13.1 for a larger example.

14.2.10.5 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;

```

14.2.10.6 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 a write barrier, though, again, a general barrier is viable:

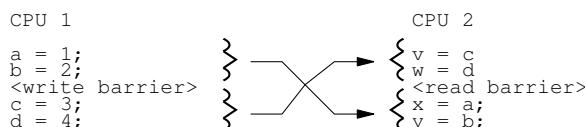
CPU 1	CPU 2
<pre> A = 1; <write barrier> B = 2; </pre>	<pre> X = B; <read barrier> Y = A; </pre>

Or:

CPU 1	CPU 2
<pre> A = 1; <write barrier> B = &A; </pre>	<pre> X = B; <data dependency barrier> Y = *X; </pre>

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:



14.2.10.7 Examples of Memory Barrier Pairings

Firstly, write barriers act as partial orderings on store operations. Consider the following sequence of events:

⁸ By “weaker”, we mean “makes fewer ordering guarantees”. A weaker barrier is usually also lower-overhead than is a stronger barrier.

```

STORE A = 1
STORE B = 2
STORE C = 3
<write barrier>
STORE D = 4
STORE E = 5

```

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 14.7.

Secondly, data dependency barriers act as partial orderings on data-dependent loads. Consider the following sequence of events with initial values {B = 7, X = 9, Y = 8, C = &Y}:

CPU 1	CPU 2
<pre> A = 1; B = 2; <write barrier> C = &B; D = 4; </pre>	<pre> LOAD X LOAD C (gets &B) LOAD *C (reads B) </pre>

Without intervention, CPU 2 may perceive the events on CPU 1 in some effectively random order, despite the write barrier issued by CPU 1, as shown in Figure 14.8.

In the above example, CPU 2 perceives that B is 7, despite the load of *C (which would be B) coming after the LOAD of C.

If, however, a data dependency barrier were to be placed between the load of C and the load of *C (i.e.: B) on CPU 2, again with initial values of {B = 7, X = 9, Y = 8, C = &Y}:

CPU 1	CPU 2
<pre> A = 1; B = 2; <write barrier> C = &B; D = 4; </pre>	<pre> LOAD X LOAD C (gets &B) <data dependency barrier> LOAD *C (reads B) </pre>

then ordering will be as intuitively expected, as shown in Figure 14.9.

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}:

CPU 1	CPU 2
<pre> A = 1; <write barrier> B = 2; </pre>	<pre> LOAD B LOAD A </pre>

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

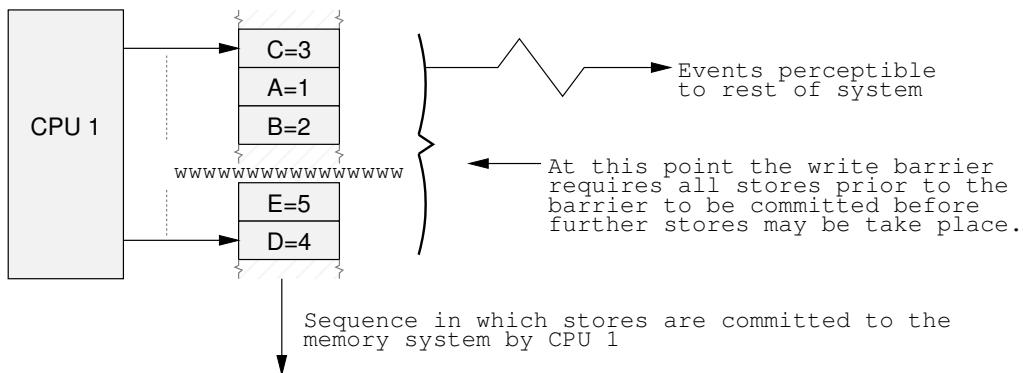


Figure 14.7: Write Barrier Ordering Semantics

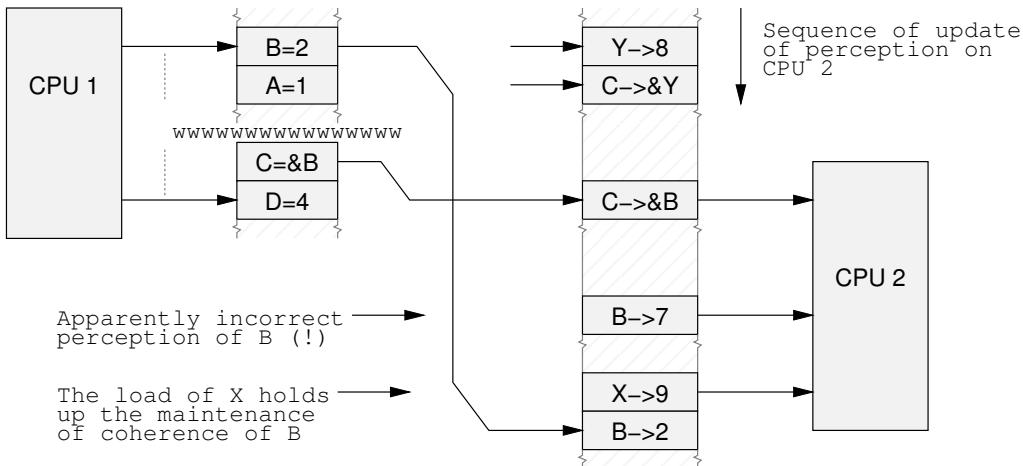


Figure 14.8: Data Dependency Barrier Omitted

in Figure 14.10.

If, however, a read barrier were to be placed between the load of B and the load of A on CPU 2, again with initial values of {A = 0, B = 9}:

CPU 1	CPU 2
<pre>A = 1; <write barrier> B = 2;</pre>	<pre>LOAD B <read barrier> LOAD A</pre>

then the partial ordering imposed by CPU 1's write barrier will be perceived correctly by CPU 2, as shown in Figure 14.11.

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}:

CPU 1	CPU 2
<pre>A = 1; <write barrier> B = 2;</pre>	<pre>LOAD B LOAD A (1st) <read barrier> LOAD A (2nd)</pre>

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 14.12.

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 14.13.

The guarantee is that the second load will always come

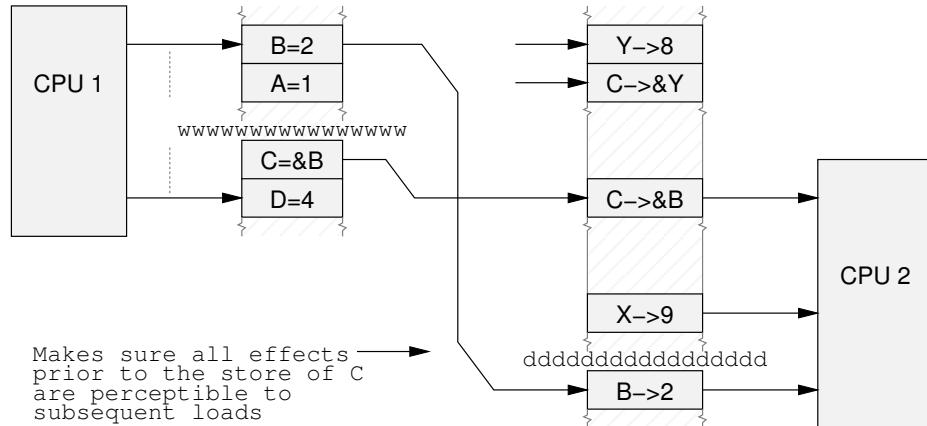


Figure 14.9: Data Dependency Barrier Supplied

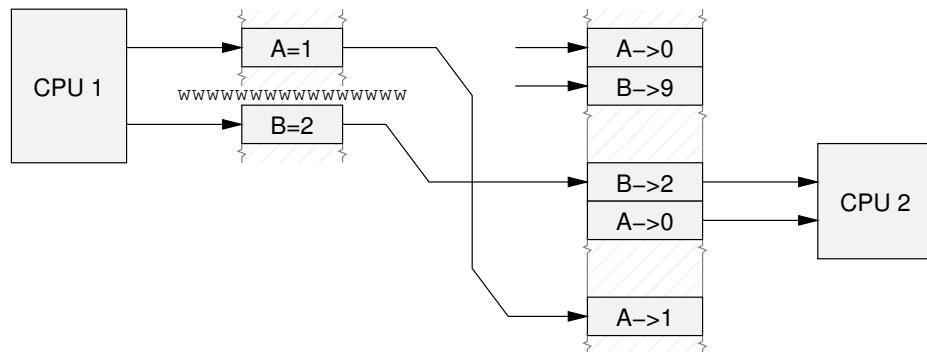


Figure 14.10: Read Barrier Needed

up with $A == 1$ if the load of B came up with $B == 2$. No such guarantee exists for the first load of A ; that may come up with either $A == 0$ or $A == 1$.

14.2.10.8 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:

CPU 1	CPU 2
LOAD B	
DIVIDE	
DIVIDE	
LOAD A	

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 14.14.

Placing a read barrier or a data dependency barrier just

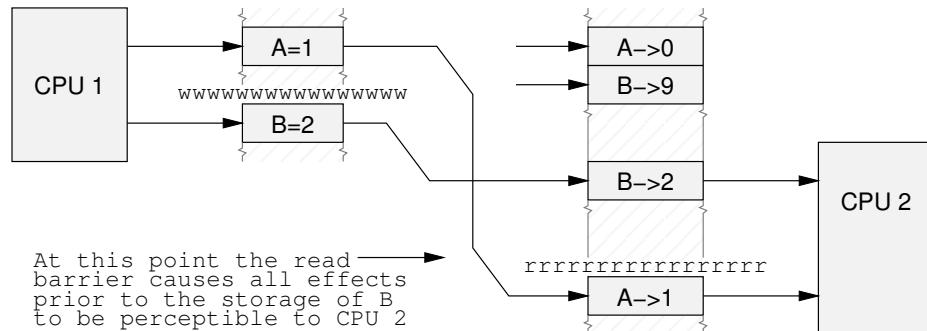


Figure 14.11: Read Barrier Supplied

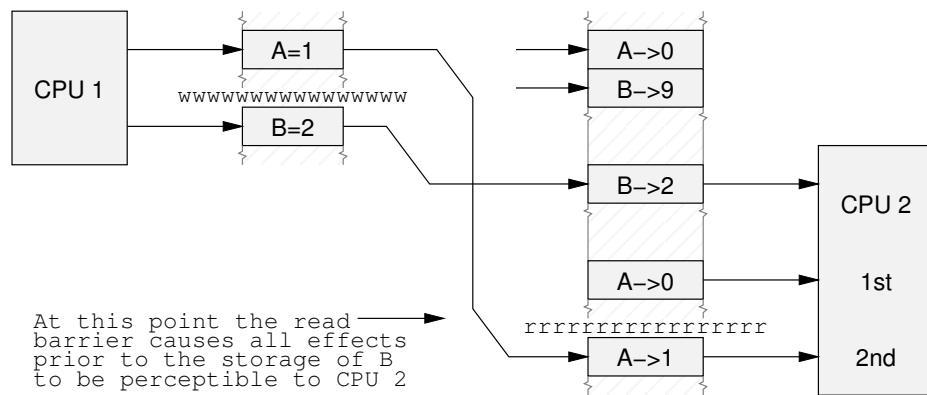


Figure 14.12: Read Barrier Supplied, Double Load

before the second load:

CPU 1	CPU 2
LOAD B	
DIVIDE	
DIVIDE	
<read barrier>	
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 14.15. On the other hand, if there was an update or invalidation to A from some other CPU, then the speculation will be cancelled and the value of A will be reloaded, as shown in Figure 14.16.

14.2.11 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** operation has completed.
 - Memory operations issued after the **UNLOCK** may be completed before the **UNLOCK** operation has completed.

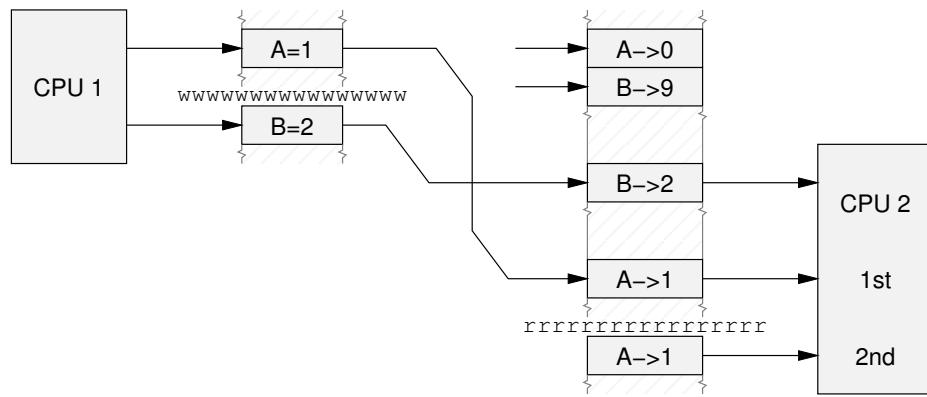


Figure 14.13: Read Barrier Supplied, Take Two

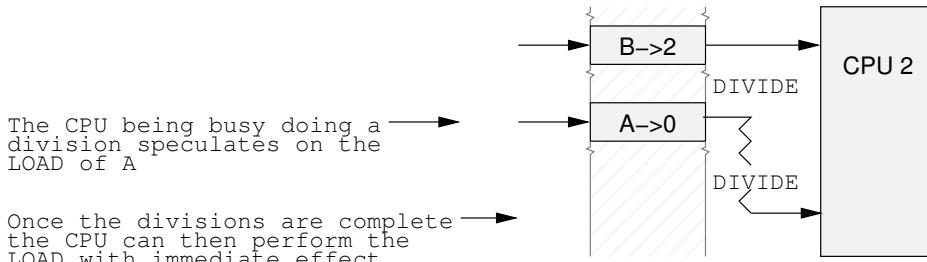


Figure 14.14: Speculative Load

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.

14.2.12 Memory-Barrier Examples

14.2.12.1 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:

```

1 *A = a;
2 LOCK
3 UNLOCK
4 *B = b;

```

might well execute in the following order:

```

2 LOCK
4 *B = b;
1 *A = a;
3 UNLOCK

```

Again, always remember that both LOCK and UNLOCK are permitted to let preceding operations “bleed

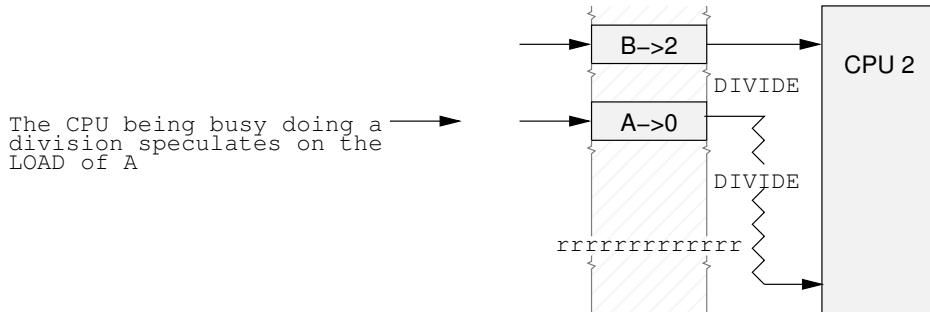


Figure 14.15: Speculative Load and Barrier

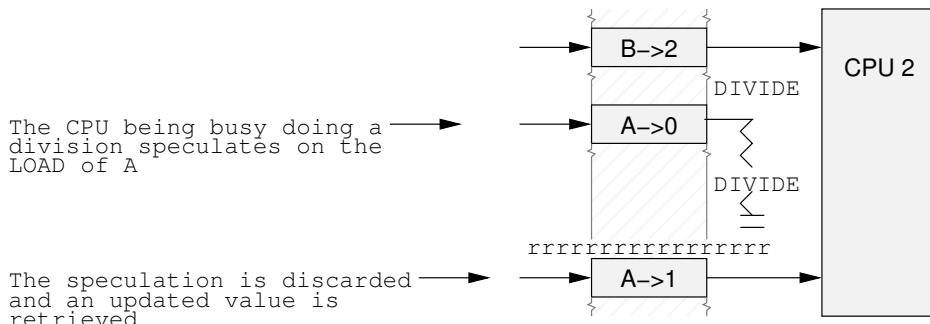


Figure 14.16: Speculative Load Cancelled by Barrier

in" to the critical section.

Quick Quiz 14.13: What sequence of LOCK-UNLOCK operations *would* act as a full memory barrier?

Quick Quiz 14.14: 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:

```

3  LOCK
1  *A = a;  *F = f,
7  *E = e;
4  *C = c;  *D = d,
2  *B = b;
6  UNLOCK

```

#	Ordering: legitimate or not?
1	*A; *B; LOCK; *C; *D; UNLOCK; *E; *F;
2	*A; {*B; LOCK;} *C; *D; UNLOCK; *E; *F;
3	{*F; *A; } *B; LOCK; *C; *D; UNLOCK; *E;
4	*A; *B; {*LOCK; *C;} *D; {UNLOCK; *E;} *F;
5	*B; LOCK; *C; *D; *A; UNLOCK; *E; *F;
6	*A; *B; *C; LOCK; *D; UNLOCK; *E; *F;
7	*A; *B; LOCK; *C; UNLOCK; *D; *E; *F;
8	{*B; *A; LOCK;} {*D; *C;} {UNLOCK; *F; *E; }
9	*B; LOCK; *C; *D; UNLOCK; {*F; *A;} *F;

Table 14.3: Lock-Based Critical Sections

Quick Quiz 14.15: Given that operations grouped in curly braces are executed concurrently, which of the rows of Table 14.3 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? ■

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 14.4, which uses a pair of locks named “M” and “Q”.

CPU 1	CPU 2
<code>A = a;</code>	<code>E = e;</code>
<code>LOCK M;</code>	<code>LOCK Q;</code>
<code>B = b;</code>	<code>F = f;</code>
<code>C = c;</code>	<code>G = g;</code>
<code>UNLOCK M;</code>	<code>UNLOCK Q;</code>
<code>D = d;</code>	<code>H = h;</code>

Table 14.4: Ordering With Multiple Locks

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 14.16: What are the constraints for Table 14.4? ■

Ordering with Multiple CPUs on One Lock: Suppose, instead of the two different locks as shown in Table 14.4, both CPUs acquire the same lock, as shown in Table 14.5?

CPU 1	CPU 2
<code>A = a;</code>	<code>E = e;</code>
<code>LOCK M;</code>	<code>LOCK M;</code>
<code>B = b;</code>	<code>F = f;</code>
<code>C = c;</code>	<code>G = g;</code>
<code>UNLOCK M;</code>	<code>UNLOCK M;</code>
<code>D = d;</code>	<code>H = h;</code>

Table 14.5: Ordering With Multiple CPUs on One Lock

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, and D.

14.2.13 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 14.17, 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.

14.2.13.1 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 14.18, 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,⁹ 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.

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.

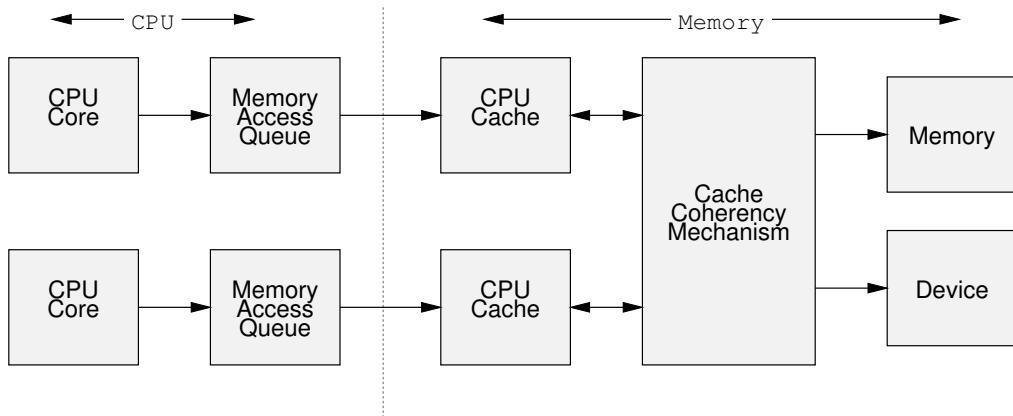


Figure 14.17: Memory Architecture

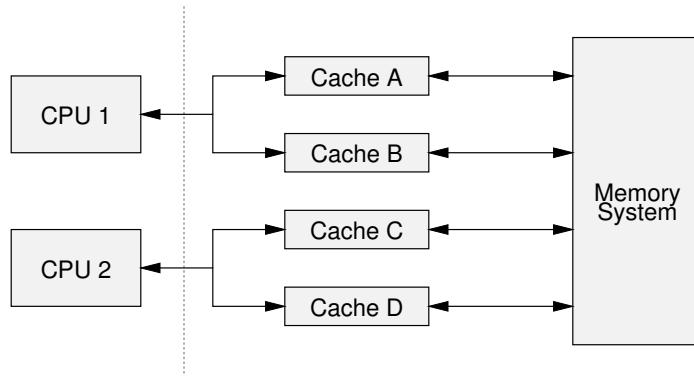


Figure 14.18: Split Caches

- 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.

14.2.14 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, 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.

14.3 Non-Blocking Synchronization

The term *non-blocking synchronization* (NBS) describes six classes of linearizable algorithms with differing *forward-progress guarantees*. These forward-progress guarantees are orthogonal to those that form the basis of real-time programming:

1. Real-time forward-progress guarantees usually have some definite time associated with them, for example, “scheduling latency must be less than 100 microseconds.” In contrast, the most popular forms of NBS only guarantees that progress will be made in finite time, with no definite bound.
2. Real-time forward-progress guarantees are sometimes probabilistic, as in the soft-real-time guarantee that “at least 99.9% of the time, scheduling latency must be less than 100 microseconds.” In contrast, NBS’s forward-progress guarantees have traditionally been unconditional.
3. Real-time forward-progress guarantees are often conditioned on environmental constraints, for example, only being honored for the highest-priority tasks, when each CPU spends at least a certain fraction of its time idle, or when I/O rates are below some specified maximum. In contrast, NBS’s forward-progress guarantees are usually unconditional.¹⁰
4. Real-time forward-progress guarantees usually apply only in the absence of software bugs. In contrast, most NBS guarantees apply even in the face of fail-stop bugs.¹¹
5. NBS forward-progress guarantee classes imply linearizability. In contrast, real-time forward progress guarantees are often independent of ordering constraints such as linearizability.

¹⁰ As we will see below, some recent NBS work relaxes this guarantee.

¹¹ Again, some recent NBS work relaxes this guarantee.

Despite these differences, a number of NBS algorithms are extremely useful in real-time programs.

There are currently seven levels in the NBS hierarchy [ACHS13], which are roughly as follows:

1. *Bounded wait-free synchronization*: Every thread will make progress within a specific finite period of time [Her91]. (Note that this level is widely considered to be unachievable, which might be why Alitarh et al. [ACHS13] omitted it.)
2. *Wait-free synchronization*: Every thread will make progress in finite time [Her93].
3. *Lock-free synchronization*: At least one thread will make progress in finite time [Her93].
4. *Obstruction-free synchronization*: Every thread will make progress in finite time in the absence of contention [HLM03].
5. *Clash-free synchronization*: At least one thread will make progress in finite time in the absence of contention [ACHS13].
6. *Starvation-free synchronization*: Every thread will make progress in finite time in the absence of failures [ACHS13].
7. *Deadlock-free synchronization*: At least one thread will make progress in finite time in the absence of failures [ACHS13].

NBS classes 1, 2 and 3 were first formulated in the early 1990s, class 4 was first formulated in the early 2000s, and class 5 was first formulated in 2013. The final two classes have seen informal use for a great many decades, but were reformulated in 2013.

In theory, any parallel algorithm can be cast into wait-free form, but there are a relatively small subset of NBS algorithms that are in common use. A few of these are listed in the following section.

14.3.1 Simple NBS

Perhaps the simplest NBS algorithm is atomic update of an integer counter using `fetch-and-add` (`atomic_add_return()`) primitives.

Another simple NBS algorithm implements a set of integers in an array. Here the array index indicates a value that might be a member of the set and the array element indicates whether or not that value actually is a set member.

```

1 static inline bool
2 __cds_wfcq_append(struct cds_wfcq_head *head,
3                     struct cds_wfcq_tail *tail,
4                     struct cds_wfcq_node *new_head,
5                     struct cds_wfcq_node *new_tail)
6 {
7     struct cds_wfcq_node *old_tail;
8
9     old_tail = uatomic_xchg(&tail->p, new_tail);
10    CMM_STORE_SHARED(old_tail->next, new_head);
11    return old_tail != &head->node;
12 }
13
14 static inline bool
15 __cds_wfcq_enqueue(struct cds_wfcq_head *head,
16                     struct cds_wfcq_tail *tail,
17                     struct cds_wfcq_node *new_tail)
18 {
19     return __cds_wfcq_append(head, tail,
20                             new_tail, new_tail);
21 }

```

Figure 14.19: NBS Enqueue Algorithm

The linearizability criterion for NBS algorithms requires that reads from and updates to the array either use atomic instructions or be accompanied by memory barriers, but in the not-uncommon case where linearizability is not important, simple volatile loads and stores suffice, for example, using `ACCESS_ONCE()`.

An NBS set may also be implemented using a bitmap, where each value that might be a member of the set corresponds to one bit. Reads and updates must normally be carried out via atomic bit-manipulation instructions, although compare-and-swap (`cmpxchg()` or CAS) instructions can also be used.

The statistical counters algorithm discussed in Section 5.2 can be considered wait-free, but only by using a cute definitional trick in which the sum is considered approximate rather than exact.¹² Given sufficiently wide error bounds that are a function of the length of time that the `read_count()` function takes to sum the counters, it is not possible to prove that any non-linearizable behavior occurred. This definitely (if a bit artificially) classifies the statistical-counters algorithm as wait-free. This algorithm is probably the most heavily used NBS algorithm in the Linux kernel.

Another common NBS algorithm is the atomic queue where elements are enqueued using an atomic exchange instruction [MS98b], followed by a store into the `->next` pointer of the new element's predecessor, as shown in Figure 14.19, which shows the userspace-RCU library implementation [Des09]. Line 9 updates the tail pointer to reference the new element while returning a

reference to its predecessor, which is stored in local variable `old_tail`. Line 10 then updates the predecessor's `->next` pointer to reference the newly added element, and finally line 11 returns an indication as to whether or not the queue was initially empty.

Although mutual exclusion is required to dequeue a single element (so that dequeue is blocking), it is possible to carry out a non-blocking removal of the entire contents of the queue. What is not possible is to dequeue any given element in a non-blocking manner: The enqueuer might have failed between lines 9 and 10 of the figure, so that the element in question is only partially enqueued. This results in a half-NBS algorithm where enqueues are NBS but dequeues are blocking. This algorithm is nevertheless used in practice, in part because most production software is not required to tolerate arbitrary fail-stop errors.

14.3.2 NBS Discussion

It is possible to create fully non-blocking queues [MS96], however, such queues are much more complex than the half-NBS algorithm outlined above. The lesson here is to carefully consider what your requirements really are. Relaxing irrelevant requirements can often result in great improvements in both simplicity and performance.

Recent research points to another important way to relax requirements. It turns out that systems providing fair scheduling can enjoy most of the benefits of wait-free synchronization even when running algorithms that provide only non-blocking synchronization, both in theory [ACHS13] and in practice [AB13]. Because a great many schedulers used in production do in fact provide fairness, the more-complex algorithms providing wait-free synchronization usually provide no practical advantages over their simpler and often faster non-blocking-synchronization counterparts.

Interestingly enough, fair scheduling is but one beneficial constraint that is often respected in practice. Other sets of constraints can permit blocking algorithms to achieve deterministic real-time response. For example, given fair locks that are granted to requesters in FIFO order at a given priority level, a method of avoiding priority inversion (such as priority inheritance [TS95, WTS96] or priority ceiling), a bounded number of threads, bounded critical sections, bounded load, and avoidance of fail-stop bugs, lock-based applications can provide deterministic response times [Bra11]. This approach of course blurs the distinction between blocking and wait-free synchronization, which is all to the good. Hopefully theoretical

¹² Citation needed. I hear of this trick verbally from Mark Moir.

frameworks continue to grow, further increasing their ability to describe how software is actually constructed in practice.

Chapter 15

Parallel Real-Time Computing

An important emerging area in computing is that of parallel real-time computing. Section 15.1 looks at a number of definitions of “real-time computing,” moving beyond the usual sound bites to more meaningful criteria. Section 15.2 surveys the sorts of applications that need real-time response. Section 15.3 notes that parallel real-time computing is upon us, and discusses when and why parallel real-time computing can be useful. Section 15.4 gives a brief overview of how parallel real-time systems may be implemented, and finally, Section 15.5 outlines how to decide whether or not your application needs real-time facilities.

15.1 What is Real-Time Computing?

One traditional way of classifying real-time computing is into the categories of *hard real time* and *soft real time*, where the macho hard real-time applications never ever miss their deadlines, but the wimpy soft real-time applications might well miss their deadlines frequently and often.

15.1.1 Soft Real Time

It should be easy to see problems with this definition of soft real time. For one thing, by this definition, *any* piece of software could be said to be a soft real-time application: “My application computes million-point fourier transforms in half a picosecond.” “No way!!! The clock cycle on this system is more than *three hundred* picoseconds!” “Ah, but it is a *soft* real-time application!” If the term “soft real time” is to be of any use whatsoever, some limits are clearly required.

We might therefore say that a given soft real-time application must meet its response-time requirements at least some fraction of the time, for example, we might say that it must execute in less than 20 microseconds 99.9% of the time.

This of course raises the question of what is to be done when the application fails to meet its response-time requirements. The answer varies with the application, but one possibility is that the system being controlled has sufficient stability and inertia to render harmless the occasional late control action. Another possibility is that the application has two ways of computing the result, a fast and deterministic but inaccurate method on the one hand and a very accurate method with unpredictable compute time on the other. One reasonable approach would be to start both methods in parallel, and if the accurate method fails to finish in time, kill it and use the answer from the fast but inaccurate method. One candidate for the fast but inaccurate method is to take no control action during the current time period, and another candidate is to take the same control action as was taken during the preceding time period.

In short, it does not make sense to talk about soft real time without some measure of exactly how soft it is.

15.1.2 Hard Real Time

In contrast, the definition of hard real time is quite definite. After all, a given system either always meets its deadlines or it doesn’t. Unfortunately, a strict application of this definition would mean that there can never be any hard real time systems. The reason for this is fancifully depicted in Figure 15.1. It is true that you could construct a more robust system, perhaps even with added redundancy. But it is also true that I can always get a bigger hammer.

Then again, perhaps it is unfair to blame the software



Figure 15.1: Real-Time Response Guarantee, Meet Hammer

for what is clearly not just a hardware problem, but a bona fide big-iron hardware problem at that.¹ This suggests that we define hard-real-time software as software that will always meet its deadlines, but only in the absence of a hardware failure. Unfortunately, failure is not always an option, as fancifully depicted in Figure 15.2. We simply cannot expect the poor gentleman depicted in that figure to be reassured our saying “Rest assured that if a missed deadline results in your tragic death, it most certainly will not have been due to a software problem!” Hard real-time response is a property of the entire system, not just of the software.

But if we cannot demand perfection, perhaps we can make do with notification, similar to the soft real-time approach noted earlier. Then if the Life-a-Tron in Figure 15.2 is about to miss its deadline, it can alert the hospital staff.

Unfortunately, this approach has the trivial solution fancifully depicted in Figure 15.3. A system that always immediately issues a notification that it won’t be able to meet its deadline complies with the letter of the law, but is completely useless. There clearly must also be a requirement that the system meet its deadline some fraction of the time, or perhaps that it be prohibited from missing its deadlines on more than a certain number of consecutive operations.

We clearly cannot take a sound-bite approach to either hard or soft real time. The next section therefore takes a

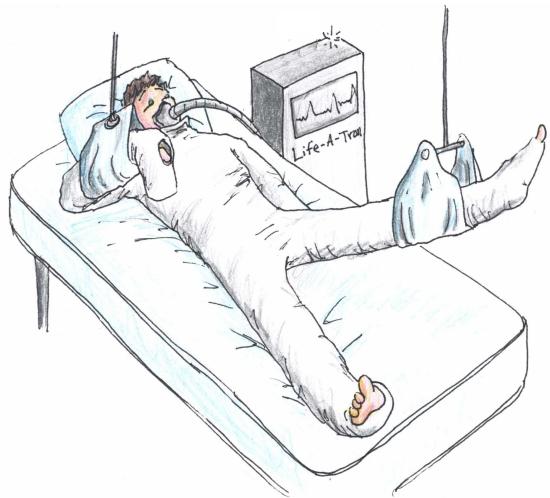


Figure 15.2: Real-Time Response: Hardware Matters

more real-world approach.

15.1.3 Real-World Real Time

Although sentences like “Hard real time systems *always* meet their deadlines!” can be catchy and are no doubt easy to memorize, something else is needed for real-world real-time systems. Although the resulting specifications are harder to memorize, they can simplify construction of a real-time system by imposing constraints on the environment, the workload, and the real-time application itself.

15.1.3.1 Environmental Constraints

Constraints on the environment address the objection to open-ended promises of response times implied by “hard real time.” These constraints might specify permissible operating temperatures, air quality, levels and types of electromagnetic radiation, and, to Figure 15.1’s point, levels of shock and vibration.

Of course, some constraints are easier to meet than others. Any number of people have learned the hard way that commodity computer components often refuse to operate at sub-freezing temperatures, which suggests a set of climate-control requirements.

An old college friend once had to meet the challenge of operating a real-time system in an atmosphere featuring some rather aggressive chlorine compounds, a challenge that he wisely handed off to his colleagues design-

¹ Or, given modern hammers, a big-steel problem.



Figure 15.3: Real-Time Response: Notification Insufficient

ing the hardware. In effect, my colleague imposed an atmospheric-composition constraint on the environment immediately surrounding the computer, a constraint that the hardware designers met through use of physical seals.

Another old college friend worked on a computer-controlled system that sputtered ingots of titanium using an industrial-strength arc in a vacuum. From time to time, the arc would decide that it was bored with its path through the ingot of titanium and choose a far shorter and more entertaining path to ground. As we all learned in our physics classes, a sudden shift in the flow of electrons creates an electromagnetic wave, with larger shifts in larger flows creating higher-power electromagnetic waves. And in this case, the resulting electromagnetic pulses were sufficient to induce a quarter of a volt potential difference in the leads of a small “rubber ducky” antenna located more than 400 meters away. This means that nearby conductors saw larger voltages, courtesy of the inverse-square law. This includes those conductors making up the computer controlling the sputtering process. In particular, the voltage induced on that computer’s reset line was sufficient to actually reset the computer, to the consternation of everyone involved. In this case, the challenge was also met using hardware, including some elaborate shielding and a fiber-optic network with the lowest bitrate I have ever heard of, namely 9600 baud. That said, less spectacular electromagnetic environments can often be handled by software through use of error detection and correction codes. That said, it is important to remember that although

error detection and correction codes can reduce failure rates, they normally cannot reduce them all the way down to zero, which can form yet another obstacle to achieving hard real-time response.

There are also situations where a minimum level of energy is required, for example, through the power leads of the system and through the devices through which the system is to communicate with that portion of the outside world that is to be monitored or controlled.

Quick Quiz 15.1: But what about battery-powered systems? They don’t require energy flowing into the system as a whole. ■

A number of systems are intended to operate in environments with impressive levels of shock and vibration, for example, engine control systems. More strenuous requirements may be found when we move away from continuous vibrations to intermittent shocks. For example, during my undergraduate studies, I encountered an old Athena ballistics computer, which was designed to continue operating normally even if a hand grenade went off nearby.² And finally, the “black boxes” used in airliners must continue operating before, during, and after a crash.

Of course, it is possible to make hardware more robust against environmental shocks and insults. Any number of ingenious mechanical shock-absorbing devices can reduce the effects of shock and vibration, multiple layers

² Decades later, the acceptance tests for some types of computer systems involve large detonations, and some types of communications networks must deal with what is delicately termed “ballistic jamming.”

of shielding can reduce the effects of low-energy electromagnetic radiation, error-correction coding can reduce the effects of high-energy radiation, various potting and sealing techniques can reduce the effect of air quality, and any number of heating and cooling systems can counter the effects of temperature. In extreme cases, triple modular redundancy can reduce the probability that a fault in one part of the system will result in incorrect behavior from the overall system. However, all of these methods have one thing in common: Although they can reduce the probability of failure, they cannot reduce it to zero.

Although these severe environmental conditions are often addressed by using more robust hardware, the workload and application constraints in the next two sections are often handled in software.

15.1.3.2 Workload Constraints

Just as with people, it is often possible to prevent a real-time system from meeting its deadlines by overloading it. For example, if the system is being interrupted too frequently, it might not have sufficient CPU bandwidth to handle its real-time application. A hardware solution to this problem might limit the rate at which interrupts were delivered to the system. Possible software solutions include disabling interrupts for some time if they are being received too frequently, resetting the device generating too-frequent interrupts, or even avoiding interrupts altogether in favor of polling.

Overloading can also degrade response times due to queueing effects, so it is not unusual for real-time systems to overprovision CPU bandwidth, so that a running system has (say) 80% idle time. This approach also applies to storage and networking devices. In some cases, separate storage and networking hardware might be reserved for the sole use of high-priority portions of the real-time application. It is of course not unusual for this hardware to be mostly idle, given that response time is more important than throughput in real-time systems.

Quick Quiz 15.2: But given the results from queueing theory, won't low utilization merely improve the average response time rather than improving the worst-case response time, which is the only response time that many real-time systems care about? ■

Of course, maintaining sufficiently low utilization requires great discipline throughout the design and implementation. There is nothing quite like a little feature creep to destroy deadlines.

15.1.3.3 Application Constraints

It is easier to provide bounded response time for some operations than for others. For example, it is quite common to see response-time specifications for interrupts and for wake-up operations, but quite rare for (say) filesystem unmount operations. One reason for this is that it is quite difficult to bound the amount of work that a filesystem-unmount operation might need to do, given that the unmount is required to flush all of that filesystem's in-memory data to mass storage.

This means that real-time applications must be confined to operations for which bounded latencies can reasonably be provided. Other operations must either be pushed out into the non-real-time portions of the application or forgone entirely.

There might also be constraints on the non-real-time portions of the application. For example, is the non-real-time application permitted to use CPUs used by the real-time portion? Are there time periods during which the real-time portion of the application is expected to be unusually busy, and if so, is the non-real-time portion of the application permitted to run at all during those times? Finally, by what amount is the real-time portion of the application permitted to degrade the throughput of the non-real-time portion?

15.1.3.4 Real-World Real-Time Specifications

As can be seen from the preceding sections, a real-world real-time specification needs to include constraints on the environment, on the workload, and on the application itself. In addition, for the operations that the real-time portion of the application is permitted to make use of, there must be constraints on the hardware and software implementing those operations.

For each such operation, these constraints might include a maximum response time (and possibly also a minimum response time) and a probability of meeting that response time. A probability of 100% indicates that the corresponding operation must provide hard real-time service.

In some cases, both the response times and the required probabilities of meeting them might vary depending on the parameters to the operation in question. For example, a network operation over a local LAN would be much more likely to complete in (say) 100 microseconds than would that same network operation over a transcontinental WAN. Furthermore, a network operation over a copper or fiber LAN might have an extremely high probability

of completing without time-consuming retransmissions, while that same networking operation over a lossy WiFi network might have a much higher probability of missing tight deadlines. Similarly, a read from a tightly coupled solid-state disk (SSD) could be expected to complete much more quickly than that same read to an old-style USB-connected rotating-rust disk drive.³

Some real-time applications pass through different phases of operation. For example, a real-time system controlling a plywood lathe that peels a thin sheet of wood (called “veneer”) from a spinning log must: (1) Load the log into the lathe, (2) Position the log on the lathe’s chucks so as to expose the largest cylinder contained in the log to the blade, (3) Start spinning the log, (4) Continuously vary the knife’s position so as to peel the log into veneer, (5) Remove the remaining core of the log that is too small to peel, and (6) Wait for the next log. Each of these six phases of operation might well have its own set of deadlines and environmental constraints, for example, one would expect phase 4’s deadlines to be much more severe than those of phase 6, milliseconds instead of seconds. One might therefore expect that low-priority work would be performed in phase 6 rather than in phase 4. That said, careful choices of hardware, drivers, and software configuration would be required to support phase 4’s more severe requirements.

A key advantage of this phase-by-phase approach is that the latency budgets can be broken down, so that the application’s various components can be developed independently, each with its own latency budget. Of course, as with any other kind of budget, there will likely be the occasional conflict as to which component gets which fraction of the overall budget, and as with any other kind of budget, strong leadership and a sense of shared goals can help to resolve these conflicts in a timely fashion. And, again as with other kinds of technical budget, a strong validation effort is required in order to ensure proper focus on latencies and to give early warning of latency problems. A successful validation effort will almost always include a good test suite, which might be unsatisfying to the theorists, but has the virtue of helping to get the job done. As a point of fact, as of early 2015, most real-world real-time system use an acceptance test rather than formal proofs.

That said, the widespread use of test suites to validate real-time systems does have a very real disadvantage,

namely that real-time software is validated only on specific hardware in specific hardware and software configurations. Adding additional hardware and configurations requires additional costly and time-consuming testing. Perhaps the field of formal verification will advance sufficiently to change this situation, but as of early 2015, rather large advances are required.

Quick Quiz 15.3: Formal verification is already quite capable, benefiting from decades of intensive study. Are additional advances *really* required, or is this just a practitioner’s excuse to continue to be lazy and ignore the awesome power of formal verification? ■

In addition to latency requirements for the real-time portions of the application, there will likely be performance and scalability requirements for the non-real-time portions of the application. These additional requirements reflect the fact that ultimate real-time latencies are often attained by degrading scalability and average performance.

Software-engineering requirements can also be important, especially for large applications that must be developed and maintained by large teams. These requirements often favor increased modularity and fault isolation.

This is a mere outline of the work that would be required to specify deadlines and environmental constraints for a production real-time system. It is hoped that this outline clearly demonstrates the inadequacy of the sound-bite-based approach to real-time computing.

15.2 Who Needs Real-Time Computing?

It is possible to argue that all computing is in fact real-time computing. For one moderately extreme example, when you purchase a birthday gift online, you would like the gift to arrive before the recipient’s birthday. And in fact even turn-of-the-millennium web services observed sub-second response constraints [Boh01], and requirements have not eased with the passage of time [DHJ⁺07]. It is nevertheless useful to focus on those real-time applications whose response-time requirements cannot be achieved straightforwardly by non-real-time systems and applications. Of course, as hardware costs decrease and bandwidths and memory sizes increase, the line between real-time and non-real-time will continue to shift, but such progress is by no means a bad thing.

Quick Quiz 15.4: Differentiating real-time from non-real-time based on what can “be achieved straightforwardly by non-real-time systems and applications” is a

³ Important safety tip: Worst-case response times from USB devices can be extremely long. Real-time systems should therefore take care to place any USB devices well away from critical paths.

travesty! There is absolutely no theoretical basis for such a distinction!!! Can't we do better than that??? ■

Real-time computing is used in industrial-control applications, ranging from manufacturing to avionics; scientific applications, perhaps most spectacularly in the adaptive optics used by large Earth-bound telescopes to de-twinkle starlight; military applications, including the afore-mentioned avionics; and financial-services applications, where the first computer to recognize an opportunity is likely to reap most of the resulting profit. These four areas could be characterized as “in search of production,” “in search of life,” “in search of death,” and “in search of money.”

Financial-services applications differ subtly from applications in the other three categories in that money is non-material, meaning that non-computational latencies are quite small. In contrast, mechanical delays inherent in the other three categories provide a very real point of diminishing returns beyond which further reductions in the application’s real-time response provide little or no benefit. This means that financial-services applications, along with other real-time information-processing applications, face an arms race, where the application with the lowest latencies normally wins. Although the resulting latency requirements can still be specified as described in Section 15.1.3.4, the unusual nature of these requirements has led some to refer to financial and information-processing applications as “low latency” rather than “real time.”

Regardless of exactly what we choose to call it, there is substantial need for real-time computing [Pet06, Inm07].

15.3 Who Needs Parallel Real-Time Computing?

It is less clear who really needs parallel real-time computing, but the advent of low-cost multicore systems has brought it to the fore regardless. Unfortunately, the traditional mathematical basis for real-time computing assumes single-CPU systems, with a few exceptions that prove the rule [Bra11]. That said, there are a couple of ways of squaring modern computing hardware to fit the real-time mathematical circle, and a few Linux-kernel hackers have been encouraging academics to make this transition [Gle10].

One approach is to recognize the fact that many real-time systems reflect biological nervous systems, with responses ranging from real-time reflexes to non-real-time strategizing and planning, as depicted in Figure 15.4. The

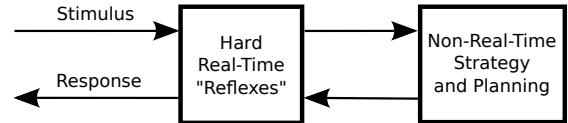


Figure 15.4: Real-Time Reflexes

hard real-time reflexes, which read from sensors and control actuators, run real-time on a single CPU, while the non-real-time strategy and planning portion of the application runs on the multiple remaining CPUs. Strategy and planning activities might include statistical analysis, periodic calibration, user interface, supply-chain activities, and preparation. For an example of high-compute-load preparation activities, think back to the veneer-peeling application discussed in Section 15.1.3.4. While one CPU is attending to the high-speed real-time computations required to peel one log, the other CPUs might be analyzing the size and shape of the next log in order to determine how to position the next log so as to obtain the greatest possible quantity of high-quality veneer. It turns out that many applications have non-real-time and real-time components [BMP08], so this approach can often be used to allow traditional real-time analysis to be combined with modern multicore hardware.

Another trivial approach is to shut off all but one hardware thread so as to return to the settled mathematics of uniprocessor real-time computing. However, this approach gives up potential cost and energy-efficiency advantages. That said, obtaining these advantages requires overcoming the parallel performance obstacles covered in Chapter 3, and not merely on average, but instead in the worst case.

Implementing parallel real-time systems can therefore be quite a challenge. Ways of meeting this challenge are outlined in the following section.

15.4 Implementing Parallel Real-Time Systems

We will look at two major styles of real-time systems, event-driven and polling. An event-driven real-time system remains idle much of the time, responding in real time to events passed up through the operating system to the application. Alternatively, the system could be running a background non-real-time workload instead of remain-

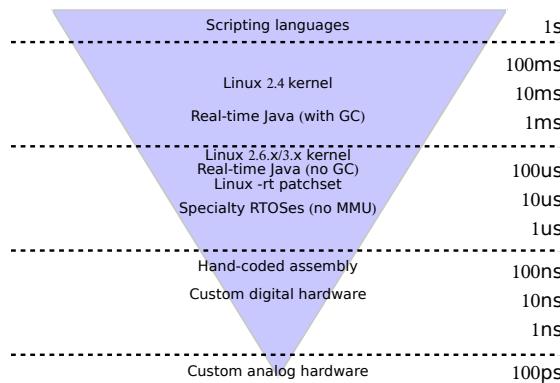


Figure 15.5: Real-Time Response Regimes

ing mostly idle. A polling real-time system features a real-time thread that is CPU bound, running in a tight loop that polls inputs and updates outputs on each pass through the loop. This tight polling loop often executes entirely in user mode, reading from and writing to hardware registers that have been mapped into the user-mode application’s address space. Alternatively, some applications place the polling loop into the kernel, for example, via use of loadable kernel modules.

Regardless of the style chosen, the approach used to implement a real-time system will depend on the deadlines, for example, as shown in Figure 15.5. Starting from the top of this figure, if you can live with response times in excess of one second, you might well be able to use scripting languages to implement your real-time application—and scripting languages are in fact used surprisingly often, not that I necessarily recommend this practice. If the required latencies exceed several tens of milliseconds, old 2.4 versions of the Linux kernel can be used, not that I necessarily recommend this practice, either. Special real-time Java implementations can provide real-time response latencies of a few milliseconds, even when the garbage collector is used. The Linux 2.6.x and 3.x kernels can provide real-time latencies of a few hundred microseconds if carefully configured, tuned, and run on real-time friendly hardware. Special real-time Java implementations can provide real-time latencies below 100 microseconds if use of the garbage collector is carefully avoided. (But note that avoiding the garbage collector means also avoiding Java’s large standard libraries, thus also avoiding Java’s productivity advantages.) A Linux kernel incorporating the -rt patchset can provide latencies below 20 microseconds,

and specialty real-time operating systems (RTOSes) running without memory translation can provide sub-ten-microsecond latencies. Achieving sub-microsecond latencies typically requires hand-coded assembly or even special-purpose hardware.

Of course, careful configuration and tuning are required all the way down the stack. In particular, if the hardware or firmware fails to provide real-time latencies, there is nothing that the software can do to make up the lost time. And high-performance hardware sometimes sacrifices worst-case behavior to obtain greater throughput. In fact, timings from tight loops run with interrupts disabled can provide the basis for a high-quality random-number generator [MOZ09]. Furthermore, some firmware does cycle-stealing to carry out various housekeeping tasks, in some cases attempting to cover its tracks by reprogramming the victim CPU’s hardware clocks. Of course, cycle stealing is expected behavior in virtualized environment, but people are nevertheless working towards real-time response in virtualized environments [Gle12, Kis14]. It is therefore critically important to evaluate your hardware’s and firmware’s real-time capabilities. There are organizations who carry out such evaluations, including the Open Source Automation Development Lab (OSADL).

But given competent real-time hardware and firmware, the next layer up the stack is the operating system, which is covered in the next section.

15.4.1 Implementing Parallel Real-Time Operating Systems

There are a number of strategies that may be used to implement a real-time system. One approach is to port a general-purpose non-realtime OS on top of a special purpose real-time operating system (RTOS), as shown in Figure 15.6. The green “Linux Process” boxes represent non-realtime processes running on the Linux kernel, while the yellow “RTOS Process” boxes represent real-time processes running on the RTOS.

This was a very popular approach before the Linux kernel gained real-time capabilities, and is still in use today [xen14, Yod04b]. However, this approach requires that the application be split into one portion that runs on the RTOS and another that runs on Linux. Although it is possible to make the two environments look similar, for example, by forwarding POSIX system calls from the RTOS to a utility thread running on Linux, there are invariably rough edges.

In addition, the RTOS must interface to both the hard-

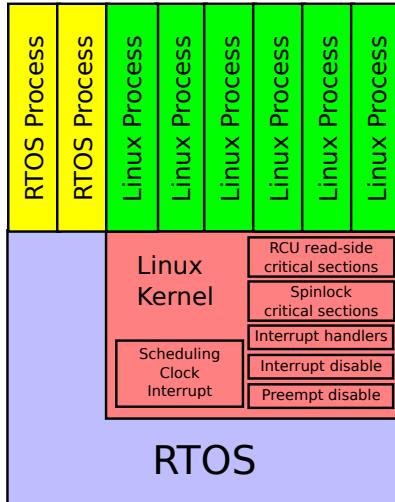


Figure 15.6: Linux Ported to RTOS

ware and to the Linux kernel, thus requiring significant maintenance with changes in both hardware and kernel. Furthermore, each such RTOS often has its own system-call interface and set of system libraries, which can balkanize both ecosystems and developers. In fact, these problems seem to be what drove the combination of RTOSes with Linux, as this approach allowed access to the full real-time capabilities of the RTOS, while allowing the application's non-real-time code full access to Linux's rich and vibrant open-source ecosystem.

Although pairing RTOSes with the Linux kernel was a clever and useful short-term response during the time that the Linux kernel had minimal real-time capabilities, it also motivated adding real-time capabilities to the Linux kernel. Progress towards this goal is shown in Figure 15.7. The upper row shows a diagram of the Linux kernel with preemption disabled, thus having essentially no real-time capabilities. The middle row shows a set of diagrams showing the increasing real-time capabilities of the mainline Linux kernel with preemption enabled. Finally, the bottom row shows a diagram of the Linux kernel with the `-rt` patchset applied, maximizing real-time capabilities. Functionality from the `-rt` patchset is added to mainline, hence the increasing capabilities of the mainline Linux kernel over time. Nevertheless, the most demanding real-time applications continue to use the `-rt` patchset.

The non-preemptible kernel shown at the top of Figure 15.7 is built with `CONFIG_PREEMPT=n`, so that ex-

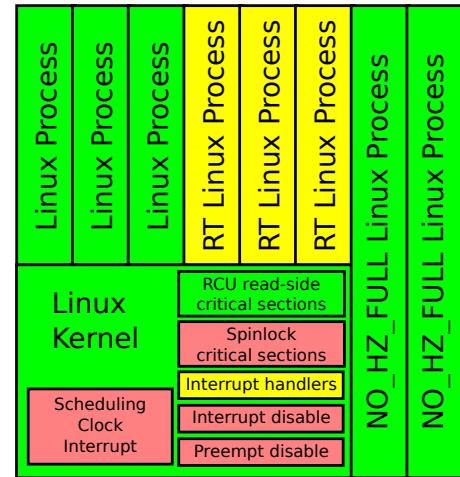


Figure 15.8: CPU Isolation

ecution within the Linux kernel cannot be preempted. This means that the kernel's real-time response latency is bounded below by the longest code path in the Linux kernel, which is indeed long. However, user-mode execution is preemptible, so that one of the real-time Linux processes shown in the upper right may preempt any of the non-real-time Linux processes shown in the upper left anytime the non-real-time process is executing in user mode.

The preemptible kernels shown in the middle row of Figure 15.7 are built with `CONFIG_PREEMPT=y`, so that most process-level code within the Linux kernel can be preempted. This of course greatly improves real-time response latency, but preemption is still disabled within RCU read-side critical sections, spinlock critical sections, interrupt handlers, interrupt-disabled code regions, and preempt-disabled code regions, as indicated by the red boxes in the left-most diagram in the middle row of the figure. The advent of preemptible RCU allowed RCU read-side critical sections to be preempted, as shown in the central diagram, and the advent of threaded interrupt handlers allowed device-interrupt handlers to be preempted, as shown in the right-most diagram. Of course, a great deal of other real-time functionality was added during this time, however, it cannot be as easily represented on this diagram. It will instead be discussed in Section 15.4.1.1.

A final approach is simply to get everything out of the way of the real-time process, clearing all other processing off of any CPUs that this process needs. This was imple-

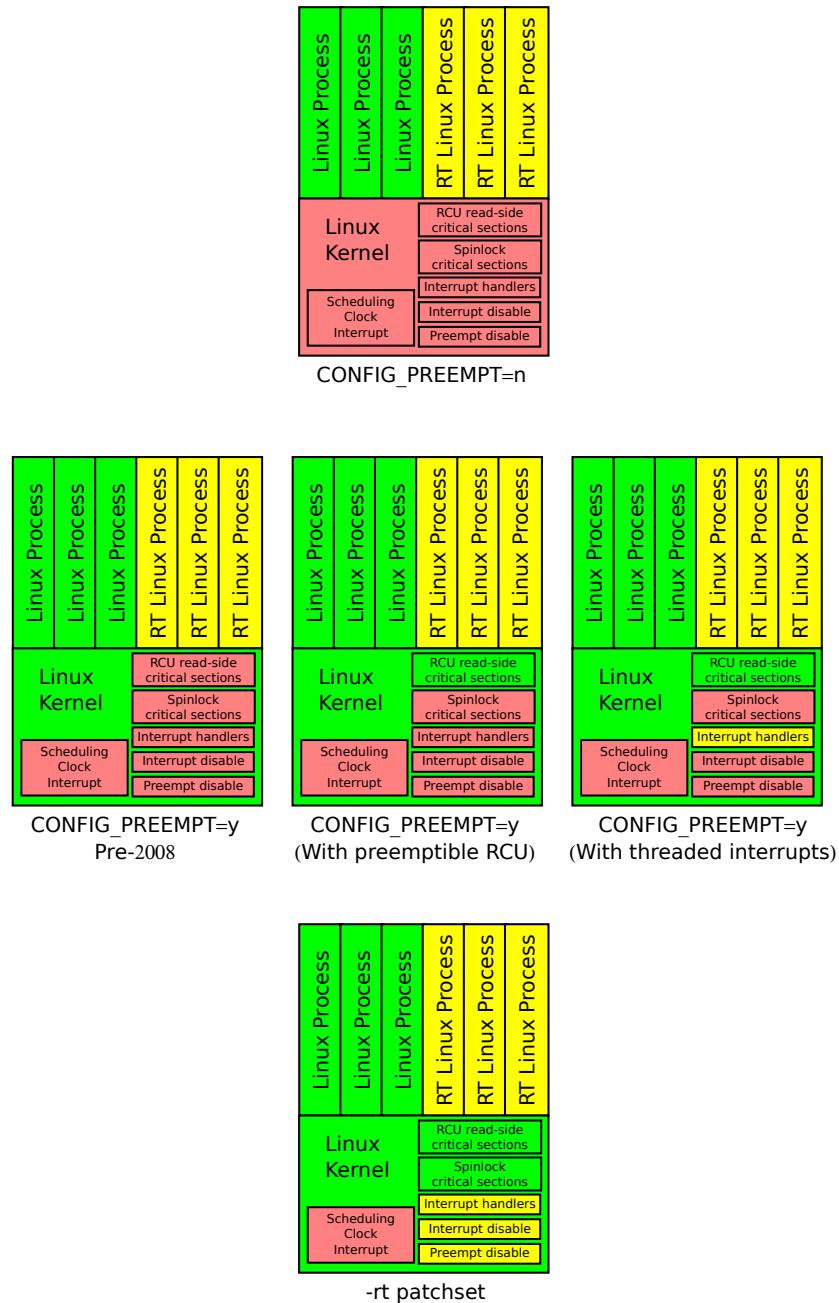


Figure 15.7: Linux-Kernel Real-Time Implementations

mented in the 3.10 Linux kernel via the `CONFIG_NO_HZ_FULL` Kconfig parameter [Wei12]. It is important to note that this approach requires at least one *housekeeping CPU* to do background processing, for example running kernel daemons. However, when there is only one runnable task on a given non-housekeeping CPU, scheduling-clock interrupts are shut off on that CPU, removing an important source of interference and *OS jitter*.⁴ With a few exceptions, the kernel does not force other processing off of the non-housekeeping CPUs, but instead simply provides better performance when only one runnable task is present on a given CPU. If configured properly, a non-trivial undertaking, `CONFIG_NO_HZ_FULL` offers real-time threads levels of performance nearly rivaling that of bare-metal systems.

There has of course been much debate over which of these approaches is best for real-time systems, and this debate has been going on for quite some time [Cor04a, Cor04c]. As usual, the answer seems to be “it depends,” as discussed in the following sections. Section 15.4.1.1 considers event-driven real-time systems, and Section 15.4.1.2 considers real-time systems that use a CPU-bound polling loop.

15.4.1.1 Event-Driven Real-Time Support

The operating-system support required for event-driven real-time applications is quite extensive, however, this section will focus on only a few items, namely timers, threaded interrupts, priority inheritance, preemptible RCU, and preemptible spinlocks.

Timers are clearly critically important for real-time operations. After all, if you cannot specify that something be done at a specific time, how are you going to respond by that time? Even in non-real-time systems, large numbers of timers are generated, so they must be handled extremely efficiently. Example uses include retransmit timers for TCP connections (which are almost always cancelled before they have a chance to fire),⁵ timed delays (as in `sleep(1)`, which are rarely cancelled), and timeouts for the `poll()` system call (which are often cancelled before they have a chance to fire). A good data structure for such timers would therefore be a priority queue whose addition and deletion primitives were fast and $O(1)$ in the number of timers posted.

⁴ A once-per-second residual scheduling-clock interrupt remains due to process-accounting concerns. Future work includes addressing these concerns and eliminating this residual interrupt.

⁵ At least assuming reasonably low packet-loss rates!

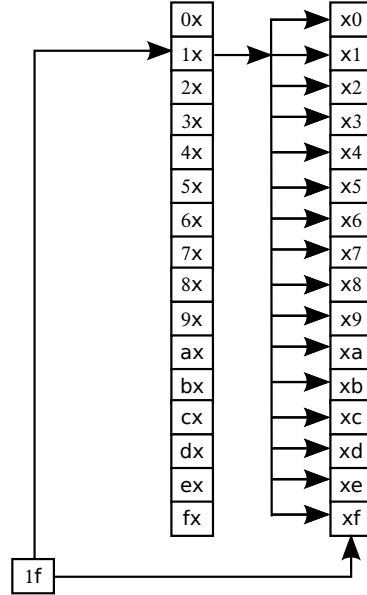


Figure 15.9: Timer Wheel

The classic data structure for this purpose is the *calendar queue*, which in the Linux kernel is called the *timer wheel*. This age-old data structure is also heavily used in discrete-event simulation. The idea is that time is quantized, for example, in the Linux kernel, the duration of the time quantum is the period of the scheduling-clock interrupt. A given time can be represented by an integer, and any attempt to post a timer at some non-integral time will be rounded to a convenient nearby integral time quantum.

One straightforward implementation would be to allocate a single array, indexed by the low-order bits of the time. This works in theory, but in practice systems create large numbers of long-duration timeouts (for example, 45-minute keepalive timeouts for TCP sessions) that are almost always cancelled. These long-duration timeouts cause problems for small arrays because much time is wasted skipping timeouts that have not yet expired. On the other hand, an array that is large enough to gracefully accommodate a large number of long-duration timeouts would consume too much memory, especially given that performance and scalability concerns require one such array for each and every CPU.

A common approach for resolving this conflict is to provide multiple arrays in a hierarchy. At the lowest level of this hierarchy, each array element represents one unit of



Figure 15.10: Timer Wheel at 1kHz



Figure 15.11: Timer Wheel at 100kHz

time. At the second level, each array element represents N units of time, where N is the number of elements in each array. At the third level, each array element represents N^2 units of time, and so on up the hierarchy. This approach allows the individual arrays to be indexed by different bits, as illustrated by Figure 15.9 for an unrealistically small eight-bit clock. Here, each array has 16 elements, so the low-order four bits of the time (currently 0xf) index the low-order (rightmost) array, and the next four bits (currently (0x1) index the next level up. Thus, we have two arrays each with 16 elements, for a total of 32 elements, which, taken together, is much smaller than the 256-element array that would be required for a single array.

This approach works extremely well for throughput-based systems. Each timer operation is $O(1)$ with small constant, and each timer element is touched at most $m + 1$ times, where m is the number of levels.

Unfortunately, timer wheels do not work well for real-time systems, and for two reasons. The first reason is that there is a harsh tradeoff between timer accuracy and timer overhead, which is fancifully illustrated by Figures 15.10 and 15.11. In Figure 15.10, timer processing happens only once per millisecond, which keeps overhead acceptably low for many (but not all!) workloads, but which also means that timeouts cannot be set for finer than one-millisecond granularities. On the other hand, Figure 15.11 shows timer processing taking place every ten microseconds, which provides acceptably fine timer granularity for

most (but not all!) workloads, but which processes timers so frequently that the system might well not have time to do anything else.

The second reason is the need to cascade timers from higher levels to lower levels. Referring back to Figure 15.9, we can see that any timers enqueued on element 1x in the upper (leftmost) array must be cascaded down to the lower (rightmost) array so that may be invoked when their time arrives. Unfortunately, there could be a large number of timeouts waiting to be cascaded, especially for timer wheels with larger numbers of levels. The power of statistics causes this cascading to be a non-problem for throughput-oriented systems, but cascading can result in problematic degradations of latency in real-time systems.

Of course, real-time systems could simply choose a different data structure, for example, some form of heap or tree, giving up $O(1)$ bounds on insertion and deletion operations to gain $O(\log n)$ limits on data-structure-maintenance operations. This can be a good choice for special-purpose RTOSes, but is inefficient for general-purpose systems such as Linux, which routinely support extremely large numbers of timers.

The solution chosen for the Linux kernel's -rt patch-set is to differentiate between timers that schedule later activity and timeouts that schedule error handling for low-probability errors such as TCP packet losses. One key observation is that error handling is normally not particularly time-critical, so that a timer wheel's millisecond-level granularity is good and sufficient. Another key observation is that error-handling timeouts are normally cancelled very early, often before they can be cascaded. A final observation is that systems commonly have many

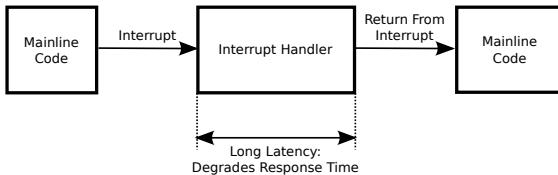


Figure 15.12: Non-Threaded Interrupt Handler

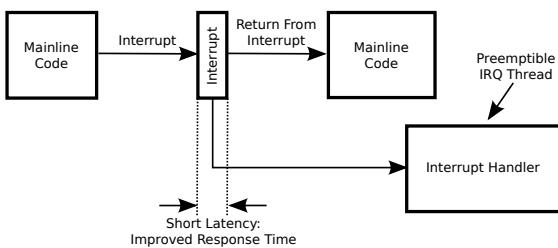


Figure 15.13: Threaded Interrupt Handler

more error-handling timeouts than they do timer events, so that an $O(\log n)$ data structure should provide acceptable performance for timer events.

In short, the Linux kernel's -rt patchset uses timer wheels for error-handling timeouts and a tree for timer events, providing each category the required quality of service.

Threaded interrupts are used to address a significant source of degraded real-time latencies, namely long-running interrupt handlers, as shown in Figure 15.12. These latencies can be especially problematic for devices that can deliver a large number of events with a single interrupt, which means that the interrupt handler will run for an extended period of time processing all of these events. Worse yet are devices that can deliver new events to a still-running interrupt handler, as such an interrupt handler might well run indefinitely, thus indefinitely degrading real-time latencies.

One way of addressing this problem is the use of threaded interrupts shown in Figure 15.13. Interrupt handlers run in the context of a preemptible IRQ thread, which runs at a configurable priority. The device interrupt handler then runs for only a short time, just long enough to make the IRQ thread aware of the new event. As shown in the figure, threaded interrupts can greatly improve real-time latencies, in part because interrupt handlers running

in the context of the IRQ thread may be preempted by high-priority real-time threads.

However, there is no such thing as a free lunch, and there are downsides to threaded interrupts. One downside is increased interrupt latency. Instead of immediately running the interrupt handler, the handler's execution is deferred until the IRQ thread gets around to running it. Of course, this is not a problem unless the device generating the interrupt is on the real-time application's critical path.

Another downside is that poorly written high-priority real-time code might starve the interrupt handler, for example, preventing networking code from running, in turn making it very difficult to debug the problem. Developers must therefore take great care when writing high-priority real-time code. This has been dubbed the *Spiderman principle*: With great power comes great responsibility.

Priority inheritance is used to handle priority inversion, which can be caused by, among other things, locks acquired by preemptible interrupt handlers [SRL90b]. Suppose that a low-priority thread holds a lock, but is preempted by a group of medium-priority threads, at least one such thread per CPU. If an interrupt occurs, a high-priority IRQ thread will preempt one of the medium-priority threads, but only until it decides to acquire the lock held by the low-priority thread. Unfortunately, the low-priority thread cannot release the lock until it starts running, which the medium-priority threads prevent it from doing. So the high-priority IRQ thread cannot acquire the lock until after one of the medium-priority threads releases its CPU. In short, the medium-priority threads are indirectly blocking the high-priority IRQ threads, a classic case of priority inversion.

Note that this priority inversion could not happen with non-threaded interrupts because the low-priority thread would have to disable interrupts while holding the lock, which would prevent the medium-priority threads from preempting it.

In the priority-inheritance solution, the high-priority thread attempting to acquire the lock donate its priority to the low-priority thread holding the lock until such time as the lock is released, thus preventing long-term priority inversion.

Of course, priority inheritance does have its limitations. For example, if you can design your application to avoid priority inversion entirely, you will likely obtain somewhat better latencies [Yod04b]. This should be no surprise, given that priority inheritance adds a pair of context switches to the worst-case latency. That said, priority



Figure 15.14: Priority Inversion and User Input

inheritance can convert indefinite postponement into a limited increase in latency, and the software-engineering benefits of priority inheritance may outweigh its latency costs in many applications.

Another limitation is that it addresses only lock-based priority inversions within the context of a given operating system. One priority-inversion scenario that it cannot address is a high-priority thread waiting on a network socket for a message that is to be written by a low-priority process that is preempted by a set of CPU-bound medium-priority processes. In addition, a potential disadvantage of applying priority inheritance to user input is fancifully depicted in Figure 15.14.

A final limitation involves reader-writer locking. Suppose that we have a very large number of low-priority threads, perhaps even thousands of them, each of which read-holds a particular reader-writer lock. Suppose that all of these threads are preempted by a set of medium-priority threads, with at least one medium-priority thread per CPU. Finally, suppose that a high-priority thread awakens and attempts to write-acquire this same reader-writer lock. No matter how vigorously we boost the priority of the threads read-holding this lock, it could well be a good long time before the high-priority thread can complete its write-acquisition.

There are a number of possible solutions to this reader-writer lock priority-inversion conundrum:

1. Only allow one read-acquisition of a given reader-writer lock at a time. (This is the approach traditionally taken by the Linux kernel's -rt patchset.)
2. Only allow N read-acquisitions of a given reader-writer lock at a time, where N is the number of CPUs.

```

1 void __rcu_read_lock(void)
2 {
3     current->rcu_read_lock_nesting++;
4     barrier();
5 }
6
7 void __rcu_read_unlock(void)
8 {
9     struct task_struct *t = current;
10
11    if (t->rcu_read_lock_nesting != 1) {
12        --t->rcu_read_lock_nesting;
13    } else {
14        barrier();
15        t->rcu_read_lock_nesting = INT_MIN;
16        barrier();
17        if (ACCESS_ONCE(t->rcu_read_unlock_special.s))
18            rcu_read_unlock_special(t);
19        barrier();
20        t->rcu_read_lock_nesting = 0;
21    }
22 }

```

Figure 15.15: Preemptible Linux-Kernel RCU

3. Only allow N read-acquisitions of a given reader-writer lock at a time, where N is a number specified somehow by the developer. There is a good chance that the Linux kernel's -rt patchset will someday take this approach.
4. Prohibit high-priority threads from write-acquiring reader-writer locks that are ever read-acquired by threads running at lower priorities. (This is a variant of the *priority ceiling* protocol [SRL90b].)

Quick Quiz 15.5: But if you only allow one reader at a time to read-acquire a reader-writer lock, isn't that the same as an exclusive lock??? ■

In some cases, reader-writer lock priority inversion can be avoided by converting the reader-writer lock to RCU, as briefly discussed in the next section.

Preemptible RCU can sometimes be used as a replacement for reader-writer locking [MW07, MBWW12, McK14], as was discussed in Section 9.3. Where it can be used, it permits readers and updaters to run concurrently, which prevents low-priority readers from inflicting any sort of priority-inversion scenario on high-priority updaters. However, for this to be useful, it is necessary to be able to preempt long-running RCU read-side critical sections [GMTW08]. Otherwise, long RCU read-side critical sections would result in excessive real-time latencies.

A preemptible RCU implementation was therefore added to the Linux kernel. This implementation avoids

the need to individually track the state of each and every task in the kernel by keeping lists of tasks that have been preempted within their current RCU read-side critical section. A grace period is permitted to end: (1) Once all CPUs have completed any RCU read-side critical sections that were in effect before the start of the current grace period and (2) Once all tasks that were preempted while in one of those pre-existing critical sections have removed themselves from their lists. A simplified version of this implementation is shown in Figure 15.15. The `__rcu_read_lock()` function spans lines 1-5 and the `__rcu_read_unlock()` function spans lines 7-22.

Line 3 of `__rcu_read_lock()` increments a per-task count of the number of nested `rcu_read_lock()` calls, and line 4 prevents the compiler from reordering the subsequent code in the RCU read-side critical section to precede the `rcu_read_lock()`.

Line 11 of `__rcu_read_unlock()` checks to see if the nesting level count is one, in other words, if this corresponds to the outermost `rcu_read_unlock()` of a nested set. If not, line 12 decrements this count, and control returns to the caller. Otherwise, this is the outermost `rcu_read_unlock()`, which requires the end-of-critical-section handling carried out by lines 14-20.

Line 14 prevents the compiler from reordering the code in the critical section with the code comprising the `rcu_read_unlock()`. Line 15 sets the nesting count to a large negative number in order to prevent destructive races with RCU read-side critical sections contained within interrupt handlers [McK11a], and line 16 prevents the compiler from reordering this assignment with line 17's check for special handling. If line 17 determines that special handling is required, line 18 invokes `rcu_read_unlock_special()` to carry out that special handling.

There are several types of special handling that can be required, but we will focus on that required when the RCU read-side critical section has been preempted. In this case, the task must remove itself from the list that it was added to when it was first preempted within its RCU read-side critical section. However, it is important to note that these lists are protected by locks, which means that `rcu_read_unlock()` is no longer lockless. However, the highest-priority threads will not be preempted, and therefore, for those highest-priority threads, `rcu_read_unlock()` will never attempt to acquire any locks. In addition, if implemented carefully, locking can be used to synchronize real-time software [Bra11].

Whether or not special handling is required, line 19

prevents the compiler from reordering the check on line 17 with the zeroing of the nesting count on line 20.

Quick Quiz 15.6: Suppose that preemption occurs just after the load from `t->rcu_read_unlock_special.s` on line 17 of Figure 15.15. Mightn't that result in the task failing to invoke `rcu_read_unlock_special()`, thus failing to remove itself from the list of tasks blocking the current grace period, in turn causing that grace period to extend indefinitely? ■

This preemptible RCU implementation enables real-time response for read-mostly data structures without the delays inherent to priority boosting of large numbers of readers.

Preemptible spinlocks are an important part of the -rt patchset due to the long-duration spinlock-based critical sections in the Linux kernel. This functionality has not yet reached mainline: Although they are a conceptually simple substitution of sleeplocks for spinlocks, they have proven relatively controversial.⁶ However, they are quite necessary to the task of achieving real-time latencies down in the tens of microseconds.

There are of course any number of other Linux-kernel components that are critically important to achieving world-class real-time latencies, most recently deadline scheduling, however, those listed in this section give a good feeling for the workings of the Linux kernel augmented by the -rt patchset.

15.4.1.2 Polling-Loop Real-Time Support

At first glance, use of a polling loop might seem to avoid all possible operating-system interference problems. After all, if a given CPU never enters the kernel, the kernel is completely out of the picture. And the traditional approach to keeping the kernel out of the way is simply not to have a kernel, and many real-time applications do indeed run on bare metal, particularly those running on eight-bit microcontrollers.

One might hope to get bare-metal performance on a modern operating-system kernel simply by running a single CPU-bound user-mode thread on a given CPU, avoiding all causes of interference. Although the reality is of course more complex, it is becoming possible to do just that, courtesy of the NO_HZ_FULL implementation led

⁶ In addition, development of the -rt patchset has slowed in recent years, perhaps because the real-time functionality that is already in the mainline Linux kernel suffices for a great many use cases [Edg13, Edg14]. However, OSADL (<http://osadl.org/>) is working to raise funds to move the remaining code from the -rt patchset to mainline.

by Frederic Weisbecker [Cor13] that has been accepted into version 3.10 of the Linux kernel. Nevertheless, considerable care is required to properly set up such an environment, as it is necessary to control a number of possible sources of OS jitter. The discussion below covers the control of several sources of OS jitter, including device interrupts, kernel threads and daemons, scheduler realtime throttling (this is a feature, not a bug!), timers, non-real-time device drivers, in-kernel global synchronization, scheduling-clock interrupts, page faults, and finally, non-real-time hardware and firmware.

Interrupts are an excellent source of large amounts of OS jitter. Unfortunately, in most cases interrupts are absolutely required in order for the system to communicate with the outside world. One way of resolving this conflict between OS jitter and maintaining contact with the outside world is to reserve a small number of house-keeping CPUs, and to force all interrupts to these CPUs. The `Documentation/IRQ-affinity.txt` file in the Linux source tree describes how to direct device interrupts to specified CPU, which as of early 2015 involves something like the following:

```
echo 0f > /proc/irq/44/smp_affinity
```

This command would confine interrupt #44 to CPUs 0-3. Note that scheduling-clock interrupts require special handling, and are discussed later in this section.

A second source of OS jitter is due to kernel threads and daemons. Individual kernel threads, such as RCU's grace-period kthreads (`rcu_bh`, `rcu_preempt`, and `rcu_sched`), may be forced onto any desired CPUs using the `taskset` command, the `sched_setaffinity()` system call, or `cgroups`.

Per-CPU kthreads are often more challenging, sometimes constraining hardware configuration and workload layout. Preventing OS jitter from these kthreads requires either that certain types of hardware not be attached to real-time systems, that all interrupts and I/O initiation take place on housekeeping CPUs, that special kernel Kconfig or boot parameters be selected in order to direct work away from the worker CPUs, or that worker CPUs never enter the kernel. Specific per-kthread advice may be found in the Linux kernel source `Documentation` directory at `kernel-per-CPU-kthreads.txt`.

A third source of OS jitter in the Linux kernel for CPU-bound threads running at real-time priority is the scheduler itself. This is an intentional debugging feature, designed to ensure that important non-realtime work is allotted at least 50 milliseconds out of each second, even if

there is an infinite-loop bug in your real-time application. However, when you are running a polling-loop-style real-time application, you will need to disable this debugging feature. This can be done as follows:

```
echo -1 > /proc/sys/kernel/sched_rt_runtime_us
```

You will of course need to be running as root to execute this command, and you will also need to carefully consider the Spiderman principle. One way to minimize the risks is to offload interrupts and kernel threads/daemons from all CPUs running CPU-bound realtime threads, as described in the paragraphs above. In addition, you should carefully read the material in the `Documentation/scheduler` directory. The material in the `sched-rt-group.txt` file is particularly important, especially if you are using the `cgroups` real-time features enabled by the `CONFIG_RT_GROUP_SCHED` Kconfig parameter, in which case you should also read the material in the `Documentation/cgroups` directory.

A fourth source of OS jitter comes from timers. In most cases, keeping a given CPU out of the kernel will prevent timers from being scheduled on that CPU. One important exception are recurring timers, where a given timer handler posts a later occurrence of that same timer. If such a timer gets started on a given CPU for any reason, that timer will continue to run periodically on that CPU, inflicting OS jitter indefinitely. One crude but effective way to offload recurring timers is to use CPU hotplug to offline all worker CPUs that are to run CPU-bound real-time application threads, online these same CPUs, then start your real-time application.

A fifth source of OS jitter is provided by device drivers that were not intended for real-time use. For an old canonical example, in 2005, the VGA driver would blank the screen by zeroing the frame buffer with interrupts disabled, which resulted in tens of milliseconds of OS jitter. One way of avoiding device-driver-induced OS jitter is to carefully select devices that have been used heavily in real-time systems, and which have therefore had their real-time bugs fixed. Another way is to confine the devices' interrupts and all code using that device to designated housekeeping CPUs. A third way is to test the device's ability to support real-time workloads and fix any real-time bugs.⁷

A sixth source of OS jitter is provided by some in-kernel full-system synchronization algorithms, perhaps

⁷ If you take this approach, please submit your fixes upstream so that others can benefit. Keep in mind that when you need to port your application to a later version of the Linux kernel, *you* will be one of those “others.”

most notably the global TLB-flush algorithm. This can be avoided by avoiding memory-unmapping operations, and especially avoiding unmapping operations within the kernel. As of early 2015, the way to avoid in-kernel unmapping operations is to avoid unloading kernel modules.

A seventh source of OS jitter is provided by scheduling-clock interrupts and RCU callback invocation. These may be avoided by building your kernel with the `NO_HZ_FULL` Kconfig parameter enabled, and then booting with the `nohz_full=` parameter specifying the list of worker CPUs that are to run real-time threads. For example, `nohz_full=2-7` would designate CPUs 2, 3, 4, 5, 6, and 7 as worker CPUs, thus leaving CPUs 0 and 1 as housekeeping CPUs. The worker CPUs would not incur scheduling-clock interrupts as long as there is no more than one runnable task on each worker CPU, and each worker CPU's RCU callbacks would be invoked on one of the housekeeping CPUs. A CPU that has suppressed scheduling-clock interrupts due to there only being one runnable task on that CPU is said to be in *adaptive ticks mode*.

As an alternative to the `nohz_full=` boot parameter, you can build your kernel with `NO_HZ_FULL_ALL`, which will designate CPU 0 as a housekeeping CPU and all other CPUs as worker CPUs. Either way, it is important to ensure that you have designated enough housekeeping CPUs to handle the housekeeping load imposed by the rest of the system, which requires careful benchmarking and tuning.

Of course, there is no free lunch, and `NO_HZ_FULL` is no exception. As noted earlier, `NO_HZ_FULL` makes kernel/user transitions more expensive due to the need for delta process accounting and the need to inform kernel subsystems (such as RCU) of the transitions. It also prevents CPUs running processes with POSIX CPU timers enabled from entering adaptive-ticks mode. Additional limitations, tradeoffs, and configuration advice may be found in `Documentation/timers/NO_HZ.txt`.

An eighth source of OS jitter is page faults. Because most Linux implementations use an MMU for memory protection, real-time applications running on these systems can be subject to page faults. Use the `mlock()` and `mlockall()` system calls to pin your application's pages into memory, thus avoiding major page faults. Of course, the Spiderman principle applies, because locking down too much memory may prevent the system from getting other work done.

A ninth source of OS jitter is unfortunately the hardware and firmware. It is therefore important to use systems

```

1 cd /sys/kernel/debug/tracing
2 echo 1 > max_graph_depth
3 echo function_graph > current_tracer
4 # run workload
5 cat per_cpu/cpuN/trace

```

Figure 15.16: Locating Sources of OS Jitter

that have been designed for real-time use. OSADL runs long-term tests of systems, so referring to their website ([\(http://osadl.org/\)](http://osadl.org/)) can be helpful.

Unfortunately, this list of OS-jitter sources can never be complete, as it will change with each new version of the kernel. This makes it necessary to be able to track down additional sources of OS jitter. Given a CPU N running a CPU-bound usermode thread, the commands shown in Figure 15.16 will produce a list of all the times that this CPU entered the kernel. Of course, the N on line 5 must be replaced with the number of the CPU in question, and the 1 on line 2 may be increased to show additional levels of function call within the kernel. The resulting trace can help track down the source of the OS jitter.

As you can see, obtaining bare-metal performance when running CPU-bound real-time threads on a general-purpose OS such as Linux requires painstaking attention to detail. Automation would of course help, and some automation has been applied, but given the relatively small number of users, automation can be expected to appear relatively slowly. Nevertheless, the ability to gain near-bare-metal performance while running a general-purpose operating system promises to ease construction of some types of real-time systems.

15.4.2 Implementing Parallel Real-Time Applications

Developing real-time applications is a wide-ranging topic, and this section can only touch on a few aspects. To this end, Section 15.4.2.1 looks at a few software components commonly used in real-time applications, Section 15.4.2.2 provides a brief overview of how polling-loop-based applications may be implemented, Section 15.4.2.3 gives a similar overview of streaming applications, and Section 15.4.2.4 briefly covers event-based applications.

15.4.2.1 Real-Time Components

As in all areas of engineering, a robust set of components is essential to productivity and reliability. This section

is not a full catalog of real-time software components—such a catalog would fill an entire book—but rather a brief overview of the types of components available.

A natural place to look for real-time software components would be algorithms offering wait-free synchronization [Her91], and in fact lockless algorithms are very important to real-time computing. However, wait-free synchronization only guarantees forward progress in finite time, and real-time computing requires algorithms that meet the far more stringent guarantee of forward progress in bounded time. After all, a century is finite, but unhelpful when your deadlines are measured in milliseconds.

Nevertheless, there are some important wait-free algorithms that do provide bounded response time, including atomic test and set, atomic exchange, atomic fetch-and-add, single-producer/single-consumer FIFO queues based on circular arrays, and numerous per-thread partitioned algorithms. In addition, recent research has confirmed the observation that algorithms with lock-free guarantees⁸ provide the same latencies in practice assuming a stochastically fair scheduler and freedom from fail-stop bugs [ACHS13].⁹ This means that lock-free stacks and queues are appropriate for real-time use.

Quick Quiz 15.7: But isn't correct operation despite fail-stop bugs a valuable fault-tolerance property? ■

In practice, locking is often used in real-time programs, theoretical concerns notwithstanding. However, under more severe constraints, lock-based algorithms can also provide bounded latencies [Bra11]. These constraints include:

1. Fair scheduler. In the common case of a fixed-priority scheduler, the bounded latencies are provided only to the highest-priority threads.
2. Sufficient bandwidth to support the workload. An implementation rule supporting this constraint might be “There will be at least 50% idle time on all CPUs during normal operation,” or, more formally, “The offered load will be sufficiently low to allow the workload to be schedulable at all times.”
3. No fail-stop bugs.
4. FIFO locking primitives with bounded acquisition, handoff, and release latencies. Again, in the common case of a locking primitive that is FIFO within

⁸ Wait-free algorithms guarantee that all threads make progress in finite time, while lock-free algorithms only guarantee that at least one thread will make progress in finite time.

⁹ This paper also introduces the notion of *bounded minimal progress*, which is a welcome step on the part of theory towards real-time practice.

priorities, the bounded latencies are provided only to the highest-priority threads.

5. Some way of preventing unbounded priority inversion. The priority-ceiling and priority-inheritance disciplines mentioned earlier in this chapter suffice.
6. Bounded nesting of lock acquisitions. We can have an unbounded number of locks, but only as long as a given thread never acquires more than a few of them (ideally only one of them) at a time.
7. Bounded number of threads. In combination with the earlier constraints, this constraint means that there will be a bounded number of threads waiting on any given lock.
8. Bounded time spent in any given critical section. Given a bounded number of threads waiting on any given lock and a bounded critical-section duration, the wait time will be bounded.

Quick Quiz 15.8: I couldn't help but spot the word “includes” before this list. Are there other constraints? ■

This result opens a vast cornucopia of algorithms and data structures for use in real-time software—and validates long-standing real-time practice.

Of course, a careful and simple application design is also extremely important. The best real-time components in the world cannot make up for a poorly thought-out design. For parallel real-time applications, synchronization overheads clearly must be a key component of the design.

15.4.2.2 Polling-Loop Applications

Many real-time applications consist of a single CPU-bound loop that reads sensor data, computes a control law, and writes control output. If the hardware registers providing sensor data and taking control output are mapped into the application's address space, this loop might be completely free of system calls. But beware of the Spiderman principle: With great power comes great responsibility, in this case the responsibility to avoid bricking the hardware by making inappropriate references to the hardware registers.

This arrangement is often run on bare metal, without the benefits of (or the interference from) an operating system. However, increasing hardware capability and increasing levels of automation motivates increasing software functionality, for example, user interfaces, logging, and reporting, all of which can benefit from an operating system.

One way of gaining much of the benefit of running on bare metal while still having access to the full features and functions of a general-purpose operating system is to use the Linux kernel’s `NO_HZ_FULL` capability, described in Section 15.4.1.2. This support first became available in version 3.10 of the Linux kernel.

15.4.2.3 Streaming Applications

A popular sort of big-data real-time application takes input from numerous sources, processes it internally, and outputs alerts and summaries. These *streaming applications* are often highly parallel, processing different information sources concurrently.

One approach for implementing streaming applications is to use dense-array circular FIFOs to connect different processing steps [Sut13]. Each such FIFO has only a single thread producing into it and a (presumably different) single thread consuming from it. Fan-in and fan-out points use threads rather than data structures, so if the output of several FIFOs needed to be merged, a separate thread would input from them and output to another FIFO for which this separate thread was the sole producer. Similarly, if the output of a given FIFO needed to be split, a separate thread would input from this FIFO and output to several FIFOs as needed.

This discipline might seem restrictive, but it allows communication among threads with minimal synchronization overhead, and minimal synchronization overhead is important when attempting to meet tight latency constraints. This is especially true when the amount of processing for each step is small, so that the synchronization overhead is significant compared to the processing overhead.

The individual threads might be CPU-bound, in which case the advice in Section 15.4.2.2 applies. On the other hand, if the individual threads block waiting for data from their input FIFOs, the advice of the next section applies.

15.4.2.4 Event-Driven Applications

We will use fuel injection into a mid-sized industrial engine as a fanciful example for event-driven applications. Under normal operating conditions, this engine requires that the fuel be injected within a one-degree interval surrounding top dead center. If we assume a 1,500-RPM rotation rate, we have 25 rotations per second, or about 9,000 degrees of rotation per second, which translates to 111 microseconds per degree. We therefore need to

```

1 if (clock_gettime(CLOCK_REALTIME, &timestart) != 0) {
2     perror("clock_gettime 1");
3     exit(-1);
4 }
5 if (nanosleep(&timewait, NULL) != 0) {
6     perror("nanosleep");
7     exit(-1);
8 }
9 if (clock_gettime(CLOCK_REALTIME, &timeend) != 0) {
10    perror("clock_gettime 2");
11    exit(-1);
12 }
```

Figure 15.17: Timed-Wait Test Program

schedule the fuel injection to within a time interval of about 100 microseconds.

Suppose that a timed wait was to be used to initiate fuel injection, although if you are building an engine, I hope you supply a rotation sensor. We need to test the timed-wait functionality, perhaps using the test program shown in Figure 15.17. Unfortunately, if we run this program, we can get unacceptable timer jitter, even in a `-rt` kernel.

One problem is that POSIX `CLOCK_REALTIME` is, oddly enough, not intended for real-time use. Instead, it means “realtime” as opposed to the amount of CPU time consumed by a process or thread. For real-time use, you should instead use `CLOCK_MONOTONIC`. However, even with this change, results are still unacceptable.

Another problem is that the thread must be raised to a real-time priority by using the `sched_setscheduler()` system call. But even this change is insufficient, because we can still see page faults. We also need to use the `mlockall()` system call to pin the application’s memory, preventing page faults. With all of these changes, results might finally be acceptable.

In other situations, further adjustments might be needed. It might be necessary to affinity time-critical threads onto their own CPUs, and it might also be necessary to affinity interrupts away from those CPUs. It might be necessary to carefully select hardware and drivers, and it will very likely be necessary to carefully select kernel configuration.

As can be seen from this example, real-time computing can be quite unforgiving.

15.4.3 The Role of 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

```

1 struct calibration {
2     short a;
3     short b;
4     short c;
5 };
6 struct calibration default_cal = { 62, 33, 88 };
7 struct calibration cur_cal = &default_cal;
8
9 short calc_control(short t, short h, short press)
10 {
11     struct calibration *p;
12
13     p = rcu_dereference(cur_cal);
14     return do_control(t, h, press, p->a, p->b, p->c);
15 }
16
17 bool update_cal(short a, short b, short c)
18 {
19     struct calibration *p;
20     struct calibration *old_p;
21
22     old_p = rcu_dereference(cur_cal);
23     p = malloc(sizeof(*p));
24     if (!p)
25         return false;
26     p->a = a;
27     p->b = b;
28     p->c = c;
29     rcu_assign_pointer(cur_cal, p);
30     if (old_p == &default_cal)
31         return true;
32     synchronize_rcu();
33     free(p);
34     return true;
35 }

```

Figure 15.18: Real-Time Calibration Using RCU

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 and hazard pointers. 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 `cur_cal` that normally references `default_cal`, which is a statically allocated and initialized structure that contains the default calibration values in fields imaginatively named `a`, `b`, and `c`. Otherwise, `cur_cal` points to a dynamically allocated structure providing the current calibration values.

Figure 15.18 shows how RCU can be used to solve this problem. Lookups are deterministic, as shown in `alc_control()` on lines 9-15, consistent with real-



Figure 15.19: The Dark Side of Real-Time Computing

time requirements. Updates are more complex, as shown by `update_cal()` on lines 17-35.

Quick Quiz 15.9: Given that real-time systems are often used for safety-critical applications, and given that runtime memory allocation is forbidden in many safety-critical situations, what is with the call to `malloc()` ???

■ **Quick Quiz 15.10:** Don't you need some kind of synchronization to protect `update_cal()` ? ■

This example shows how RCU can provide deterministic read-side data-structure access to real-time programs.

15.5 Real Time vs. Real Fast: How to Choose?

The choice between real-time and real-fast computing can be a difficult one. Because real-time systems often inflict a throughput penalty on non-real-time computing, using real-time when it is not required can cause problems, as fancifully depicted by Figure 15.19. On the other hand, failing to use real-time when it *is* required can also cause problems, as fancifully depicted by Figure 15.20. It is almost enough to make you feel sorry for the boss!

One rule of thumb uses the following four questions to help you choose:

1. Is average long-term throughput the only goal?
2. Is it permissible for heavy loads to degrade response times?



Figure 15.20: The Dark Side of Real-Fast Computing

3. Is there high memory pressure, ruling out use of the `mlockall()` system call?
4. Does the basic work item of your application take more than 100 milliseconds to complete?

If the answer to any of these questions is “yes,” you should choose real-fast over real-time, otherwise, real-time might be for you.

Choose wisely, and if you do choose real-time, make sure that your hardware, firmware, and operating system are up to the job!

Chapter 16

Ease of Use

“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.”

16.1 What is Easy?

“Easy” is a relative term. For example, many people would consider a 15-hour airplane flight to be a bit of an ordeal—unless they stopped to consider alternative modes of transportation, especially swimming. This means that creating an easy-to-use API requires that you know quite a bit about your intended users.

The following question illustrates this point: “Given a randomly chosen person among everyone alive today, what one change would improve his or her life?”

There is no single change that would be guaranteed to help everyone’s life. After all, there is an extremely wide range of people, with a correspondingly wide range of needs, wants, desires, and aspirations. A starving person might need food, but additional food might well hasten the death of a morbidly obese person. The high level of excitement so fervently desired by many young people might well be fatal to someone recovering from a heart attack. Information critical to the success of one person might contribute to the failure of someone suffering from information overload. In short, if you are working on a software project that is intended to help someone you know nothing about, you should not be surprised when that someone is less than impressed with your efforts.

If you really want to help a given group of people, there is simply no substitute for working closely with them over an extended period of time. Nevertheless, there are some simple things that you can do to increase the odds of your users being happy with your software, and some of these things are covered in the next section.

16.2 Rusty Scale for API Design

This section is adapted from portions of Rusty Russell’s 2003 Ottawa Linux Symposium keynote address [Rus03, Slides 39–57]. Rusty’s key point is that the goal should not be merely to make an API easy to use, but rather to make the API hard to misuse. To that end, Rusty proposed his “Rusty Scale” in decreasing order of this important hard-to-misuse property.

The following list attempts to generalize the Rusty Scale beyond the Linux kernel:

1. It is impossible to get wrong. Although this is the standard to which all API designers should strive, only the mythical `dwim()`¹ command manages to come close.
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. The `malloc()` library function is a good example. Although it is easy to get memory allocation wrong, a great many projects do manage to get it right, at least most of the time. Using `malloc()` in conjunction with Valgrind [The11] moves `malloc()` almost up to the “do it right or it will always break at runtime” point on the scale.

¹ The `dwim()` function is an acronym that expands to “do what I mean”.

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 original non-CONFIG_PREEMPT implementation of `rcu_read_lock()` [McK07a] is an infamous example of this point on the scale.
13. Read the documentation and you will get it wrong. For example, the DEC Alpha `wmb` instruction's documentation [SW95] fooled a number of developers into thinking that that this instruction had much stronger memory-order semantics than it actually does. Later documentation clarified this point [Com01], moving the `wmb` instruction up to the “read the documentation and you will get it right” point on the scale.
14. Follow common convention and you will get it wrong. The `printf()` statement is an example of this point on the scale because developers almost always fail to check `printf()`'s 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. The Linux kernel `smp_mb()` function is an example of this point on the scale. Many developers assume that this function has much stronger ordering semantics than it possesses. Section 14.2 contains the information needed to avoid this mistake, as does the Linux-kernel source tree's Documentation directory.
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. The `gets()` function is a famous example of this point on the scale. In fact, `gets()` can perhaps best be described as an unconditional buffer-overflow security hole.

16.3 Shaving the Mandelbrot Set

The set of useful programs resembles the Mandelbrot set (shown in Figure 16.1) 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”,² restricting the use of the API to an easily described subset of the full set of potential uses.

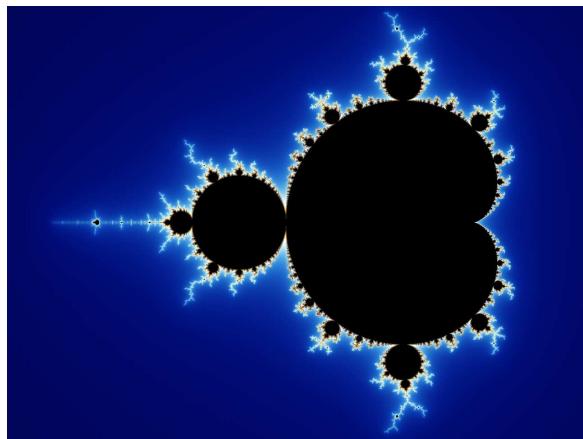


Figure 16.1: Mandelbrot Set (Courtesy of Wikipedia)

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.

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.³ 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 suc-

² Due to Josh Triplett.

³ Those of you with strong operating-system backgrounds, please suspend disbelief. If you are unable to suspend disbelief, send us a better example.

cessor 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 16.1: 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 16.2: 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 16.3: 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 16.2.

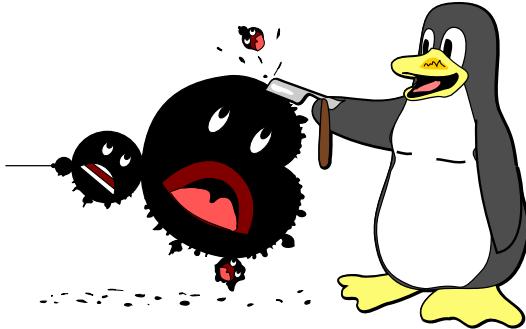


Figure 16.2: Shaving the Mandelbrot Set

Chapter 17

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, hardware transactional memory, and parallel functional programming. But first, a cautionary tale on prognostication taken from the early 2000s.

17.1 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 17.1),
2. Multithreaded Mania (Figure 17.2),
3. More of the Same (Figure 17.3), and

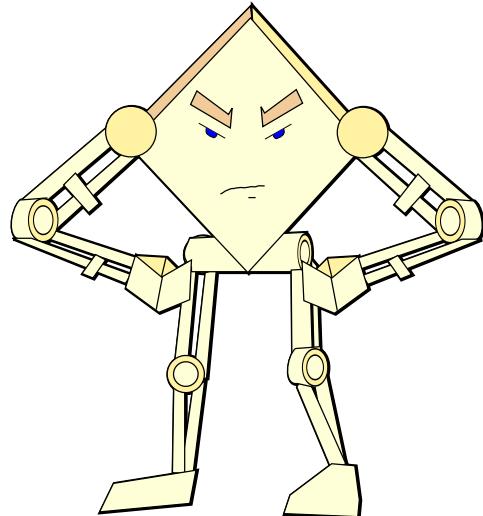


Figure 17.1: Uniprocessor Über Alles

4. Crash Dummies Slamming into the Memory Wall (Figure 17.4).

Each of these scenarios are covered in the following sections.

17.1.1 Uniprocessor Über Alles

As was said in 2004 [McK04]:

In this scenario, the combination of Moore's Law increases in CPU clock rate and continued progress in horizontally scaled computing render SMP systems irrelevant. This scenario is

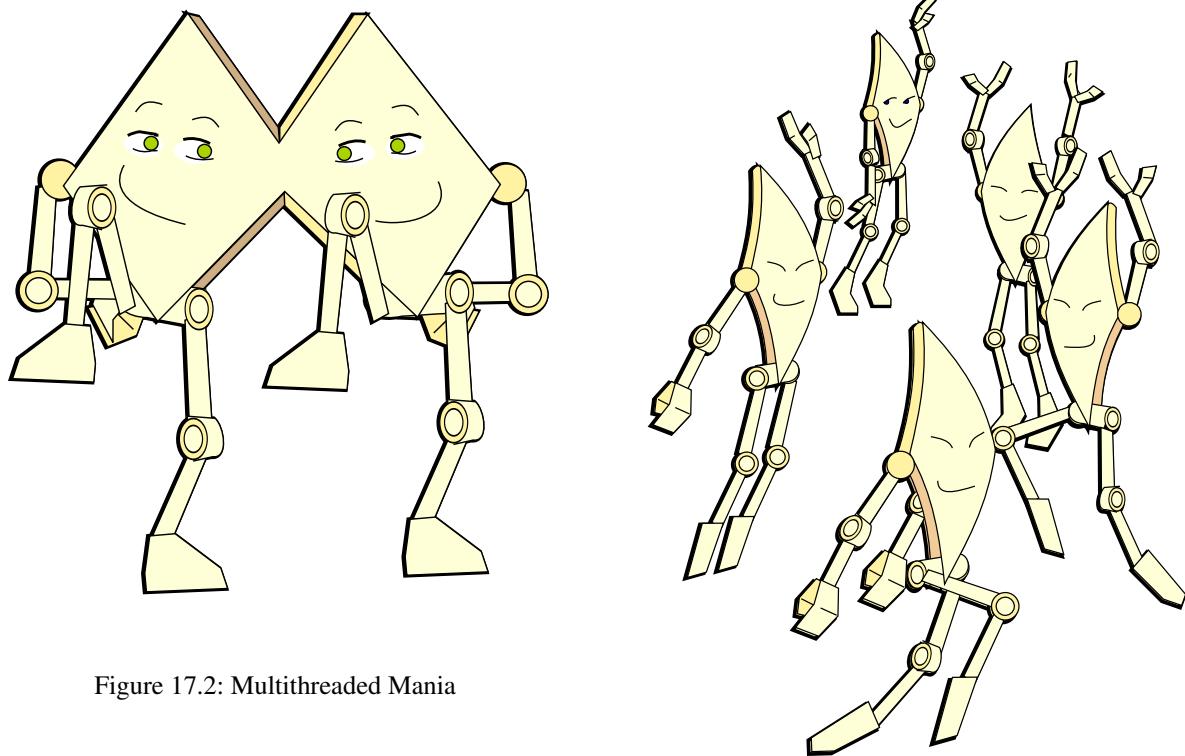


Figure 17.2: Multithreaded Mania

therefore dubbed “Uniprocessor Über Alles”, literally, uniprocessors above all else.

These uniprocessor systems would be subject only to instruction overhead, since memory barriers, 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.

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!

Figure 17.3: More of the Same

17.1.2 Multithreaded Mania

Also from 2004 [McK04]:

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 ap-

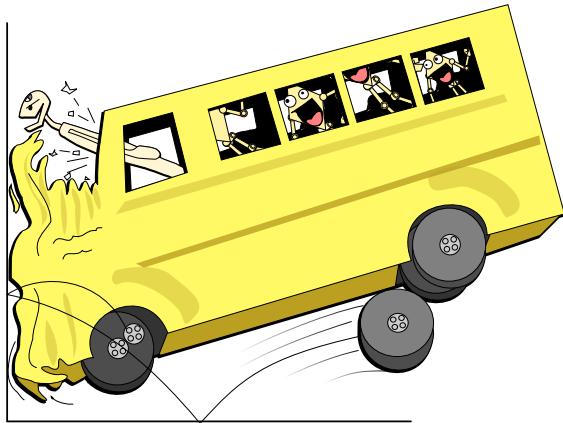


Figure 17.4: Crash Dummies Slamming into the Memory Wall

plications with large cache footprints. There is also some possibility that the restricted amount of cache available will cause RCU-based 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.

17.1.3 More of the Same

Again from 2004 [McK04]:

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, intercon-

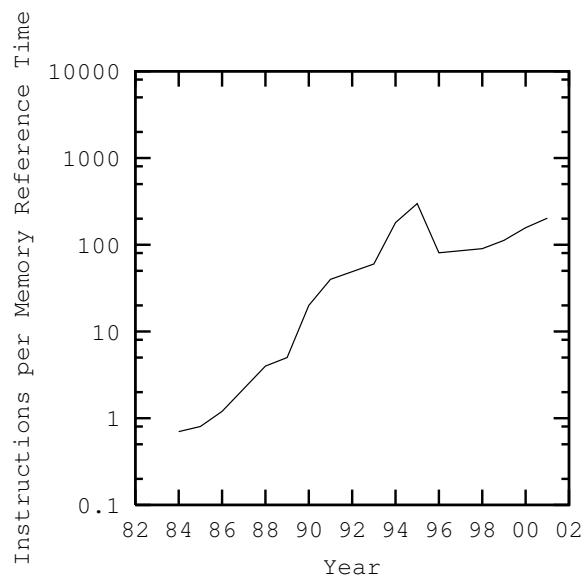


Figure 17.5: Instructions per Local Memory Reference for Sequent Computers

nect 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.

17.1.4 Crash Dummies Slamming into the Memory Wall

And one more quote from 2004 [McK04]:

If the memory-latency trends shown in Figure 17.5 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 17.6. As can be seen in this figure, if RCU

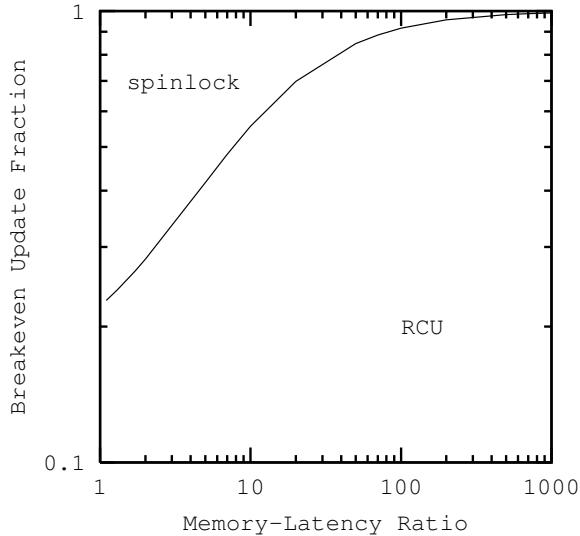


Figure 17.6: Breakevens vs. r , λ Large, Four CPUs

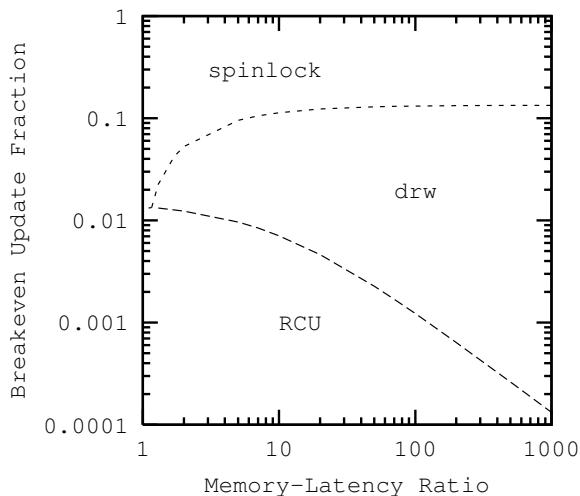


Figure 17.7: Breakevens vs. r , λ Small, Four CPUs

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 17.7. 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 for such workloads. 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.

17.2 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 14.3).

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” [McK09b]. Section 17.2.1 examines the challenges faced interacting with the outside world, Section 17.2.2 looks at interactions with process modification primitives, Section 17.2.3 explores interactions with other synchronization primitives, and finally Section 17.2.4 closes with some discussion.

17.2.1 Outside World

In the words of Donald Knuth:

Many computer users feel that input and output are not actually part of “real programming,” they are merely things that (unfortunately) must be done in order to get information in and out of the machine.

Whether we believe that input and output are “real programming” or not, the fact is that for most computer systems, interaction with the outside world is a first-class requirement. This section therefore critiques transactional memory’s ability to so interact, whether via I/O operations, time delays, or persistent storage.

17.2.1.1 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 revocable, so that executing the operation twice has the same effect as executing it once. Unfortunately, I/O is in general the prototypical irrevocable operation, making it difficult to include general I/O operations in transactions. In fact, general I/O is irrevocable: Once you have pushed the button launching the nuclear warheads, there is no turning back.

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 “a+” 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 *irrevocable* transaction [SMS08] to proceed at any given time, thus allowing irrevocable transactions to contain I/O operations.¹ This works in general, but severely limits

¹ In earlier literature, irrevocable transactions are termed *inevitable* transactions.

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 irrevocability to tolerate I/O operations seems to prohibit use of manual transaction-abort operations.² Finally, if there is an irrevocable transaction manipulating a given data item, any other transaction manipulating that same data item cannot have non-blocking semantics.

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.

17.2.1.2 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:

```

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:

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 work, but will require TM to interact with other synchronization primitives.
2. Permit only one special irrevocable transaction [SMS08] to proceed at any given time, thus allowing irrevocable 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 irrevocable transactions to permit RPC operations rules out manual transaction-abort operations once the RPC operation has started. Finally, if there is an irrevocable transaction manipulating a given data item, any other transaction manipulating that same data item cannot have non-blocking semantics.
3. Identify special cases where the success of the transaction may be determined before the RPC response is received, and automatically convert these to irrevocable 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

² This difficulty was pointed out by Michael Factor.

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.

17.2.1.3 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 (or

possibly even hardware) at the other end of the transaction may require a substantial redesign to tolerate aborted transactions. This need for redesign would make it more difficult to apply transactional memory to legacy code.

17.2.1.4 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 disappear, 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 include 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 even span multiple computer systems via distributed lock managers and distributed filesystems—and persist across reboots of any or all of these computer systems.

Persistent locks can be used by any application, including applications written using multiple languages and software environments. In fact, a persistent lock might well be acquired by an application written in C and released by an application written in Python.

How could a similar persistent functionality be provided 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 devices and/or filesystems. Unfortunately, this does not

handle network communication, nor does it handle I/O to devices that do not provide snapshot capabilities, 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.

17.2.2 Process Modification

Processes are not eternal: They are created and destroyed, their memory mappings are modified, they are linked to dynamic libraries, and they are debugged. These sections look at how transactional memory can handle an ever-changing execution environment.

17.2.2.1 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:

```

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 under 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 compiler to enforce `pthread_create()`-free transactions.

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 questions about the nature of conflicting accesses, given that the parent and children are presumably permitted 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 transaction. 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 reasonably 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 commonplace 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 textbook 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 thoroughly.

17.2.2.2 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 17.1: What about non-persistent primitives represented by data structures in `mmap()` regions of memory? What happens when there 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 `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 `transaction_safe` and `transaction_unsafe` attributes.³ This approach has some advantages over aborting the transaction at runtime, but again requires non-TM synchronization primitives for use in conjunction with `exec()`.
3. Treat the transaction in a manner similar to non-persistent Locking primitives, so that the transaction survives if `exec()` fails, and silently commits if the `exec()` succeeds. The case where some of the variables affected by the transaction reside in `mmap()` ed memory (and thus could survive a successful `exec()` system call) is left as an exercise for the reader.
4. Abort the transaction (and the `exec()` system call) if the `exec()` system call would have succeeded, but allow the transaction to continue if the `exec()` system call would fail. This is in some sense the “correct” approach, but it would require considerable work for a rather unsatisfying result.

³ Thanks to Mark Moir for pointing me at this spec, and to Michael Wong for having pointed me at an earlier revision some time back.

The `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 execs in real life. That said, the two options prohibiting `exec()` within transactions are perhaps the most logical of the group.

Similar issues surround the `exit()` and `kill()` system calls.

17.2.2.3 Dynamic Linking and Loading

Both lock-based critical sections and RCU read-side critical sections can legitimately contain code that invokes dynamically linked and loaded functions, including C/C++ shared libraries and Java class libraries. Of course, the code contained in these libraries is by definition unknowable 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 dynamically 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 theory. 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 operations 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 implementations of TM seem to have different restrictions.

So what can TM do about dynamically linked and loaded library functions? Options for part (a), the actual 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, possibilities include the following:

1. Just execute the code: if there are any TM-unfriendly operations in the function, simply abort the transaction. Unfortunately, this approach makes it impossible for the compiler to determine whether a given group of transactions may be safely composed. One way to permit composability regardless is irrevocable transactions, however, current implementations permit only a single irrevocable transaction to proceed at any given time, which can severely limit performance and scalability. Irrevocable transactions also seem to rule out use of manual transaction-abort operations. Finally, if there is an irrevocable transaction manipulating a given data item, any other transaction manipulating that same data item cannot have non-blocking semantics.
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.)

17.2.2.4 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 conflicting transactions.

It is interesting to note that `munmap()` leaves the relevant region of memory unmapped, which could have additional interesting implications.⁴

17.2.2.5 Debugging

The usual debugging operations such as breakpoints work normally within lock-based critical sections and from RCU read-side critical sections. However,

⁴ This difference between mapping and unmapping was noted by Josh Triplett.

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.

17.2.3 Synchronization

If transactional memory someday proves that it can be everything to everyone, it will not need to interact with

any other synchronization mechanism. Until then, it will need to work with synchronization mechanisms that can do what it cannot, or that work more naturally in a given situation. The following sections outline the current challenges in this area.

17.2.3.1 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 participate 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 be 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.

17.2.3.2 Reader-Writer Locking

It is commonplace to read-acquire reader-writer locks while holding other locks, which just works, at least 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 seems 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.

17.2.3.3 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.⁵ 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., `rcu_ctrlblk.lock`) were converted to transactions, it is silent about what happened to locks used in RCU-based updates (e.g., `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.

⁵ However, the in-kernel excuse is wearing thin with the advent of user-space RCU [Des09, DMS⁺12].

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 `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. Nevertheless, there are popular TM implementations that can be easily and efficiently integrated with RCU in this manner [PW07, HW11, HW13].
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 approach 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.⁶

17.2.3.4 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:

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.

⁶ Kudos to the TxLinux group, Maged Michael, and Josh Triplett for coming up with a number of the above alternatives.

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.
5. Introduce weak semantics to transactional memory. One approach is the combination with RCU described in Section 17.2.3.3, while Gramoli and Guerraoui survey a number of other weak-transaction approaches [GG14], for example, restricted partitioning of large “elastic” transactions into smaller transactions, thus reducing conflict probabilities (albeit with tepid performance and scalability). Perhaps further experience will show that some uses of extra-transactional accesses can be replaced by weak 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.

17.2.4 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.

But for the moment, the current state of STM can best be summarized with a series of cartoons. First, Figure 17.8 shows the STM vision. As always, the reality is a bit more nuanced, as fancifully depicted by Figures 17.9, 17.10, and 17.11.

Recent advances in commercially available hardware have opened the door for variants of HTM, which are addressed in the following section.

17.3 Hardware Transactional Memory

As of early 2012, hardware transactional memory (HTM) is starting to emerge into commercially available commod-



Figure 17.8: The STM Vision

ity computer systems. This section makes a first attempt to find its place in the parallel programmer’s toolbox.

From a conceptual viewpoint, HTM uses processor caches and speculative execution to make a designated group of statements (a “transaction”) take effect atomically from the viewpoint of any other transactions running on other processors. This transaction is initiated by a begin-transaction machine instruction and completed by a commit-transaction machine instruction. There is typically also an abort-transaction machine instruction, which squashes the speculation (as if the begin-transaction instruction and all following instructions had not executed) and commences execution at a failure handler. The location of the failure handler is typically specified by the begin-transaction instruction, either as an explicit failure-handler address or via a condition code set by the instruction itself. Each transaction executes atomically with respect to all other transactions.

HTM has a number of important benefits, including automatic dynamic partitioning of data structures, reducing synchronization-primitive cache misses, and supporting a fair number of practical applications.

However, it always pays to read the fine print, and HTM is no exception. A major point of this section is determining under what conditions HTM’s benefits outweigh the complications hidden in its fine print. To this end, Sec-



Figure 17.9: The STM Reality: Conflicts

tion 17.3.1 describes HTM’s benefits and Section 17.3.2 describes its weaknesses. This is the same approach used in earlier papers [MMW07, MMTW10], but focused on HTM rather than TM as a whole.⁷

Section 17.3.3 then describes HTM’s weaknesses with respect to the combination of synchronization primitives used in the Linux kernel (and in some user-space applications). Section 17.3.4 looks at where HTM might best fit into the parallel programmer’s toolbox, and Section 17.3.5 lists some events that might greatly increase HTM’s scope and appeal. Finally, Section 17.3.6 presents concluding remarks.

17.3.1 HTM Benefits WRT to Locking

The primary benefits of HTM are (1) its avoidance of the cache misses that are often incurred by other synchronization primitives, (2) its ability to dynamically partition data structures, and (3) the fact that it has a fair number of practical applications. I break from TM tradition by not

⁷ And I gratefully acknowledge many stimulating discussions with the other authors, Maged Michael, Josh Triplett, and Jonathan Walpole, as well as with Andi Kleen.



Figure 17.10: The STM Reality: Irrevocable Operations



Figure 17.11: The STM Reality: Realtime Response

listing ease of use separately for two reasons. First, ease of use should stem from HTM's primary benefits, which this paper focuses on. Second, there has been considerable controversy surrounding attempts to test for raw programming talent [Bow06, DBA09] and even around the use of small programming exercises in job interviews [Bra07]. This indicates that we really do not have a grasp on what makes programming easy or hard. Therefore, this paper focuses on the three benefits listed above, each in one of the following sections.

17.3.1.1 Avoiding Synchronization Cache Misses

Most synchronization mechanisms are based on data structures that are operated on by atomic instructions. Because these atomic instructions normally operate by first causing the relevant cache line to be owned by the CPU that they are running on, a subsequent execution of the same instance of that synchronization primitive on some other CPU will result in a cache miss. These communications cache misses severely degrade both the performance and scalability of conventional synchronization mechanisms [ABD⁺97, Section 4.2.3].

In contrast, HTM synchronizes by using the CPU's cache, avoiding the need for a synchronization data structure and resultant cache misses. HTM's advantage is great-

est in cases where a lock data structure is placed in a separate cache line, in which case, converting a given critical section to an HTM transaction can reduce that critical section's overhead by a full cache miss. These savings can be quite significant for the common case of short critical sections, at least for those situations where the elided lock does not share a cache line with an oft-written variable protected by that lock.

Quick Quiz 17.2: Why would it matter that oft-written variables shared the cache line with the lock variable? ■

17.3.1.2 Dynamic Partitioning of Data Structures

A major obstacle to the use of some conventional synchronization mechanisms is the need to statically partition data structures. There are a number of data structures that are trivially partitionable, with the most prominent example being hash tables, where each hash chain constitutes a partition. Allocating a lock for each hash chain then trivially parallelizes the hash table for operations confined to a given chain.⁸ Partitioning is similarly trivial for arrays, radix trees, and a few other data structures.

However, partitioning for many types of trees and

⁸ And it is also easy to extend this scheme to operations accessing multiple hash chains by having such operations acquire the locks for all relevant chains in hash order.

graphs is quite difficult, and the results are often quite complex [Ell80]. Although it is possible to use two-phased locking and hashed arrays of locks to partition general data structures, other techniques have proven preferable [Mil06], as will be discussed in Section 17.3.3. Given its avoidance of synchronization cache misses, HTM is therefore a very real possibility for large non-partitionable data structures, at least assuming relatively small updates.

Quick Quiz 17.3: Why are relatively small updates important to HTM performance and scalability? ■

17.3.1.3 Practical Value

Some evidence of HTM's practical value has been demonstrated in a number of hardware platforms, including Sun Rock [DLMN09] and Azul Vega [Cli09]. It is reasonable to assume that practical benefits will flow from the more recent IBM Blue Gene/Q, Intel Haswell TSX, and AMD ASF systems.

Expected practical benefits include:

1. Lock elision for in-memory data access and update [MT01, RG02].
2. Concurrent access and small random updates to large non-partitionable data structures.

However, HTM also has some very real shortcomings, which will be discussed in the next section.

17.3.2 HTM Weaknesses WRT Locking

The concept of HTM is quite simple: A group of accesses and updates to memory occurs atomically. However, as is the case with many simple ideas, complications arise when you apply it to real systems in the real world. These complications are as follows:

1. Transaction-size limitations.
2. Conflict handling.
3. Aborts and rollbacks.
4. Lack of forward-progress guarantees.
5. Irrevocable operations.
6. Semantic differences.

Each of these complications is covered in the following sections, followed by a summary.

17.3.2.1 Transaction-Size Limitations

The transaction-size limitations of current HTM implementations stem from the use of the processor caches to hold the data affected by the transaction. Although this allows a given CPU to make the transaction appear atomic to other CPUs by executing the transaction within the confines of its cache, it also means that any transaction that does not fit must be aborted. Furthermore, events that change execution context, such as interrupts, system calls, exceptions, traps, and context switches either must abort any ongoing transaction on the CPU in question or must further restrict transaction size due to the cache footprint of the other execution context.

Of course, modern CPUs tend to have large caches, and the data required for many transactions would fit easily in a one-megabyte cache. Unfortunately, with caches, sheer size is not all that matters. The problem is that most caches can be thought of hash tables implemented in hardware. However, hardware caches do not chain their buckets (which are normally called *sets*), but rather provide a fixed number of cachelines per set. The number of elements provided for each set in a given cache is termed that cache's *associativity*.

Although cache associativity varies, the eight-way associativity of the level-0 cache on the laptop I am typing this on is not unusual. What this means is that if a given transaction needed to touch nine cache lines, and if all nine cache lines mapped to the same set, then that transaction cannot possibly complete, never mind how many megabytes of additional space might be available in that cache. Yes, given randomly selected data elements in a given data structure, the probability of that transaction being able to commit is quite high, but there can be no guarantee.

There has been some research work to alleviate this limitation. Fully associative *victim caches* would alleviate the associativity constraints, but there are currently stringent performance and energy-efficiency constraints on the sizes of victim caches. That said, HTM victim caches for unmodified cache lines can be quite small, as they need to retain only the address: The data itself can be written to memory or shadowed by other caches, while the address itself is sufficient to detect a conflicting write [RD12].

Unbounded transactional memory (UTM) schemes [AAKL06, MBM⁺06] use DRAM as an extremely large victim cache, but integrating such schemes into a production-quality cache-coherence mechanism is still an unsolved problem. In addition, use of DRAM as a victim cache may have unfortunate perfor-

mance and energy-efficiency consequences, particularly if the victim cache is to be fully associative. Finally, the “unbounded” aspect of UTM assumes that all of DRAM could be used as a victim cache, while in reality the large but still fixed amount of DRAM assigned to a given CPU would limit the size of that CPU’s transactions. Other schemes use a combination of hardware and software transactional memory [KCH⁺06] and one could imagine using STM as a fallback mechanism for HTM.

However, to the best of my knowledge, currently available systems do not implement any of these research ideas, and perhaps for good reason.

17.3.2.2 Conflict Handling

The first complication is the possibility of *conflicts*. For example, suppose that transactions A and B are defined as follows:

Transaction A	Transaction B
$x = 1;$	$y = 2;$
$y = 3;$	$x = 4;$

Suppose that each transaction executes concurrently on its own processor. If transaction A stores to x at the same time that transaction B stores to y , neither transaction can progress. To see this, suppose that transaction A executes its store to y . Then transaction A will be interleaved within transaction B, in violation of the requirement that transactions execute atomically with respect to each other. Allowing transaction B to execute its store to x similarly violates the atomic-execution requirement. This situation is termed a *conflict*, which happens whenever two concurrent transactions access the same variable where at least one of the accesses is a store. The system is therefore obligated to abort one or both of the transactions in order to allow execution to progress. The choice of exactly which transaction to abort is an interesting topic that will very likely retain the ability to generate Ph.D. dissertations for some time to come, see for example [ATC⁺11].⁹ For the purposes of this section, we can assume that the system makes a random choice.

Another complication is conflict detection, which is comparatively straightforward, at least in the simplest case. When a processor is executing a transaction, it marks every cache line touched by that transaction. If the processor’s cache receives a request involving a cache line

that has been marked as touched by the current transaction, a potential conflict has occurred. More sophisticated systems might try to order the current processors’ transaction to precede that of the processor sending the request, and optimization of this process will likely also retain the ability to generate Ph.D. dissertations for quite some time. However this section assumes a very simple conflict-detection strategy.

However, for HTM to work effectively, the probability of conflict must be suitably low, which in turn requires that the data structures be organized so as to maintain a sufficiently low probability of conflict. For example, a red-black tree with simple insertion, deletion, and search operations fits this description, but a red-black tree that maintains an accurate count of the number of elements in the tree does not.¹⁰ For another example, a red-black tree that enumerates all elements in the tree in a single transaction will have high conflict probabilities, degrading performance and scalability. As a result, many serial programs will require some restructuring before HTM can work effectively. In some cases, practitioners will prefer to take the extra steps (in the red-black-tree case, perhaps switching to a partitionable data structure such as a radix tree or a hash table), and just use locking, particularly during the time before HTM is readily available on all relevant architectures [Cli09].

Quick Quiz 17.4: How could a red-black tree possibly efficiently enumerate all elements of the tree regardless of choice of synchronization mechanism??? ■

Furthermore, the fact that conflicts can occur brings failure handling into the picture, as discussed in the next section.

17.3.2.3 Aborts and Rollbacks

Because any transaction might be aborted at any time, it is important that transactions contain no statements that cannot be rolled back. This means that transactions cannot do I/O, system calls, or debugging breakpoints (no single stepping in the debugger for HTM transactions!!!). Instead, transactions must confine themselves to accessing normal cached memory. Furthermore, on some systems, interrupts, exceptions, traps, TLB misses, and other events will also abort transactions. Given the number of bugs that have resulted from improper handling of error conditions, it is fair to ask what impact aborts and rollbacks have on ease of use.

⁹ Liu’s and Spear’s paper entitled “Toxic Transactions” [LS11] is particularly instructive in this regard.

¹⁰ The need to update the count would result in additions to and deletions from the tree conflicting with each other, resulting in strong non-commutativity [AGH⁺11a, AGH⁺11b, McK11b].

Quick Quiz 17.5: But why can't a debugger emulate single stepping by setting breakpoints at successive lines of the transaction, relying on the retry to retrace the steps of the earlier instances of the transaction? ■

Of course, aborts and rollbacks raise the question of whether HTM can be useful for hard realtime systems. Do the performance benefits of HTM outweigh the costs of the aborts and rollbacks, and if so under what conditions? Can transactions use priority boosting? Or should transactions for high-priority threads instead preferentially abort those of low-priority threads? If so, how is the hardware efficiently informed of priorities? The literature on realtime use of HTM is quite sparse, perhaps because researchers are finding more than enough problems in getting transactions to work well in non-realtime environments.

Because current HTM implementations might deterministically abort a given transaction, software must provide fallback code. This fallback code must use some other form of synchronization, for example, locking. If the fallback is used frequently, then all the limitations of locking, including the possibility of deadlock, reappear. One can of course hope that the fallback isn't used often, which might allow simpler and less deadlock-prone locking designs to be used. But this raises the question of how the system transitions from using the lock-based fallbacks back to transactions.¹¹ One approach is to use a test-and-test-and-set discipline [MT02], so that everyone holds off until the lock is released, allowing the system to start from a clean slate in transactional mode at that point. However, this could result in quite a bit of spinning, which might not be wise if the lock holder has blocked or been preempted. Another approach is to allow transactions to proceed in parallel with a thread holding a lock [MT02], but this raises difficulties in maintaining atomicity, especially if the reason that the thread is holding the lock is because the corresponding transaction would not fit into cache.

Finally, dealing with the possibility of aborts and rollbacks seems to put an additional burden on the developer, who must correctly handle all combinations of possible error conditions.

It is clear that users of HTM must put considerable validation effort into testing both the fallback code paths and transition from fallback code back to transactional code.

¹¹ The possibility of an application getting stuck in fallback mode has been termed the “lemming effect”, a term that Dave Dice has been credited with coining.

17.3.2.4 Lack of Forward-Progress Guarantees

Even though transaction size, conflicts, and aborts/rollbacks can all cause transactions to abort, one might hope that sufficiently small and short-duration transactions could be guaranteed to eventually succeed. This would permit a transaction to be unconditionally retried, in the same way that compare-and-swap (CAS) and load-linked/store-conditional (LL/SC) operations are unconditionally retried in code that uses these instructions to implement atomic operations.

Unfortunately, most currently available HTM implementation refuse to make any sort of forward-progress guarantee, which means that HTM cannot be used to avoid deadlock on those systems.¹² Hopefully future implementations of HTM will provide some sort of forward-progress guarantees. Until that time, HTM must be used with extreme caution in real-time applications.¹³

The one exception to this gloomy picture as of 2013 is upcoming versions of the IBM mainframe, which provides a separate instruction that may be used to start a special *constrained transaction* [JSG12]. As you might guess from the name, such transactions must live within the following constraints:

1. Each transaction's data footprint must be contained within four 32-byte blocks of memory.
2. Each transaction is permitted to execute at most 32 assembler instructions.
3. Transactions are not permitted to have backwards branches (e.g., no loops).
4. Each transaction's code is limited to 256 bytes of memory.
5. If a portion of a given transaction's data footprint resides within a given 4K page, then that 4K page is prohibited from containing any of that transaction's instructions.

These constraints are severe, but the nevertheless permit a wide variety of data-structure updates to be implemented, including stacks, queues, hash tables, and so on. These operations are guaranteed to eventually complete, and are free of deadlock and livelock conditions.

¹² HTM might well be used to reduce the probability of deadlock, but as long as there is some possibility of the fallback code being executed, there is some possibility of deadlock.

¹³ As of mid-2012, there has been surprisingly little work on transactional memory's real-time characteristics.

It will be interesting to see how hardware support of forward-progress guarantees evolves over time.

17.3.2.5 Irrevocable Operations

Another consequence of aborts and rollbacks is that HTM transactions cannot accommodate irrevocable operations. Current HTM implementations typically enforce this limitation by requiring that all of the accesses in the transaction be to cacheable memory (thus prohibiting MMIO accesses) and aborting transactions on interrupts, traps, and exceptions (thus prohibiting system calls).

Note that buffered I/O can be accommodated by HTM transactions as long as the buffer fill/flush operations occur extra-transactionally. The reason that this works is that adding data to and removing data from the buffer is revocable: Only the actual buffer fill/flush operations are irrevocable. Of course, this buffered-I/O approach has the effect of including the I/O in the transaction's footprint, increasing the size of the transaction and thus increasing the probability of failure. ■

17.3.2.6 Semantic Differences

Although HTM can in many cases be used as a drop-in replacement for locking (hence the name transactional lock elision [DHL⁺08]), there are subtle differences in semantics. A particularly nasty example involving coordinated lock-based critical sections that results in deadlock or livelock when executed transactionally was given by Blundell [BLM06], but a much simpler example is the empty critical section.

In a lock-based program, an empty critical section will guarantee that all processes that had previously been holding that lock have now released it. This idiom was used by the 2.4 Linux kernel's networking stack to coordinate changes in configuration. But if this empty critical section is translated to a transaction, the result is a no-op. The guarantee that all prior critical sections have terminated is lost. In other words, transactional lock elision preserves the data-protection semantics of locking, but loses locking's time-based messaging semantics.

Quick Quiz 17.6: But why would *anyone* need an empty lock-based critical section??? ■

Quick Quiz 17.7: Can't transactional lock elision trivially handle locking's time-based messaging semantics by simply choosing not to elide empty lock-based critical sections? ■

Quick Quiz 17.8: Given modern hardware [MOZ09], how can anyone possibly expect parallel software relying

```

1 void boostee(void)
2 {
3     int i = 0;
4
5     acquire_lock(&boost_lock[i]);
6     for (;;) {
7         acquire_lock(&boost_lock[!i]);
8         release_lock(&boost_lock[i]);
9         i = i ^ 1;
10        do_something();
11    }
12 }
13
14 void booster(void)
15 {
16     int i = 0;
17
18     for (;;) {
19         usleep(1000); /* sleep 1 ms. */
20         acquire_lock(&boost_lock[i]);
21         release_lock(&boost_lock[i]);
22         i = i ^ 1;
23     }
24 }
```

Figure 17.12: Exploiting Priority Boosting

on timing to work? ■

One important semantic difference between locking and transactions is the priority boosting that is used to avoid priority inversion in lock-based real-time programs. One way in which priority inversion can occur is when a low-priority thread holding a lock is preempted by a medium-priority CPU-bound thread. If there is at least one such medium-priority thread per CPU, the low-priority thread will never get a chance to run. If a high-priority thread now attempts to acquire the lock, it will block. It cannot acquire the lock until the low-priority thread releases it, the low-priority thread cannot release the lock until it gets a chance to run, and it cannot get a chance to run until one of the medium-priority threads gives up its CPU. Therefore, the medium-priority threads are in effect blocking the high-priority process, which is the rationale for the name “priority inversion.”

One way to avoid priority inversion is *priority inheritance*, in which a high-priority thread blocked on a lock temporarily donates its priority to the lock's holder, which is also called *priority boosting*. However, priority boosting can be used for things other than avoiding priority inversion, as shown in Figure 17.12. Lines 1-12 of this figure show a low-priority process that must nevertheless run every millisecond or so, while lines 14-24 of this same figure show a high-priority process that uses priority boosting to ensure that `boostee()` runs periodically as needed.

The `boostee()` function arranges this by always

holding one of the two `boost_lock []` locks, so that lines 20-21 of `booster()` can boost priority as needed.

Quick Quiz 17.9: But the `boostee()` function in Figure 17.12 alternatively acquires its locks in reverse order! Won't this result in deadlock? ■

This arrangement requires that `boostee()` acquire its first lock on line 5 before the system becomes busy, but this is easily arranged, even on modern hardware.

Unfortunately, this arrangement can break down in presence of transactional lock elision. The `boostee()` function's overlapping critical sections become one infinite transaction, which will sooner or later abort, for example, on the first time that the thread running the `boostee()` function is preempted. At this point, `boostee()` will fall back to locking, but given its low priority and that the quiet initialization period is now complete (which after all is why `boostee()` was preempted), this thread might never again get a chance to run.

And if the `boostee()` thread is not holding the lock, then the `booster()` thread's empty critical section on lines 20 and 21 of Figure 17.12 will become an empty transaction that has no effect, so that `boostee()` never runs. This example illustrates some of the subtle consequences of transactional memory's rollback-and-retry semantics.

Given that experience will likely uncover additional subtle semantic differences, application of HTM-based lock elision to large programs should be undertaken with caution. That said, where it does apply, HTM-based lock elision can eliminate the cache misses associated with the lock variable, which has resulted in tens of percent performance increases in large real-world software systems as of early 2015. We can therefore expect to see substantial use of this technique on hardware supporting it.

Quick Quiz 17.10: So a bunch of people set out to supplant locking, and they mostly end up just optimizing locking??? ■

17.3.2.7 Summary

Although it seems likely that HTM will have compelling use cases, current implementations have serious transaction-size limitations, conflict-handling complications, abort-and-rollback issues, and semantic differences that will require careful handling. HTM's current situation relative to locking is summarized in Table 17.1. As can be seen, although the current state of HTM alleviates some serious shortcomings of locking,¹⁴ it does so by

¹⁴ In fairness, it is important to emphasize that locking's shortcomings do have well-known and heavily used engineering solutions, including

introducing a significant number of shortcomings of its own. These shortcomings are acknowledged by leaders in the TM community [MS12].¹⁵

In addition, this is not the whole story. Locking is not normally used by itself, but is instead typically augmented by other synchronization mechanisms, including reference counting, atomic operations, non-blocking data structures, hazard pointers [Mic04, HLM02], and read-copy update (RCU) [MS98a, MAK⁺01, HMBW07, McK12b]. The next section looks at how such augmentation changes the equation.

17.3.3 HTM Weaknesses WRT to Locking When Augmented

Practitioners have long used reference counting, atomic operations, non-blocking data structures, hazard pointers, and RCU to avoid some of the shortcomings of locking. For example, deadlock can be avoided in many cases by using reference counts, hazard pointers, or RCU to protect data structures, particularly for read-only critical sections [Mic04, HLM02, DMS⁺12, GMTW08, HMBW07]. These approaches also reduce the need to partition data structures [McK12a]. RCU further provides contention-free wait-free read-side primitives [DMS⁺12]. Adding these considerations to Table 17.1 results in the updated comparison between augmented locking and HTM shown in Table 17.2. A summary of the differences between the two tables is as follows:

1. Use of non-blocking read-side mechanisms alleviates deadlock issues.
2. Read-side mechanisms such as hazard pointers and RCU can operate efficiently on non-partitionable data.
3. Hazard pointers and RCU do not contend with each other or with updaters, allowing excellent performance and scalability for read-mostly workloads.

deadlock detectors [Cor06a], a wealth of data structures that have been adapted to locking, and a long history of augmentation, as discussed in Section 17.3.3. In addition, if locking really were as horrible as a quick skim of many academic papers might reasonably lead one to believe, where did all the large lock-based parallel programs (both FOSS and proprietary) come from, anyway?

¹⁵ In addition, in early 2011, I was invited to deliver a critique of some of the assumptions underlying transactional memory [McK11d]. The audience was surprisingly non-hostile, though perhaps they were taking it easy on me due to the fact that I was heavily jet-lagged while giving the presentation.

	Locking		Hardware Transactional Memory	
Basic Idea	Allow only one thread at a time to access a given set of objects.			Cause a given operation over a set of objects to execute atomically.
Scope	+ Handles all operations.		+ Handles revocable operations. - Irrevocable operations force fallback (typically to locking).	
Composability	⇓ Limited by deadlock.		⇓ Limited by irrevocable operations, transaction size, and deadlock (assuming lock-based fallback code).	
Scalability & Performance	- Data must be partitionable to avoid lock contention.		- Data must be partitionable to avoid conflicts.	
	⇓ Partitioning must typically be fixed at design time.		+ Dynamic adjustment of partitioning carried out automatically down to cacheline boundaries. - Partitioning required for fallbacks (less important for rare fallbacks).	
	⇓ Locking primitives typically result in expensive cache misses and memory-barrier instructions.		- Transactions begin/end instructions typically do not result in cache misses, but do have memory-ordering consequences.	
	+ Contention effects are focused on acquisition and release, so that the critical section runs at full speed.		- Contention aborts conflicting transactions, even if they have been running for a long time.	
	+ Privatization operations are simple, intuitive, performant, and scalable.		- Privatized data contributes to transaction size.	
	+ Commodity hardware suffices. + Performance is insensitive to cache-geometry details.		- New hardware required (and is starting to become available). - Performance depends critically on cache geometry.	
Software Support	+ APIs exist, large body of code and experience, debuggers operate naturally.		- APIs emerging, little experience outside of DBMS, breakpoints mid-transaction can be problematic.	
Interaction With Other Mechanisms	+ Long experience of successful interaction.		⇓ Just beginning investigation of interaction.	
Practical Apps	+ Yes.		+ Yes.	
Wide Applicability	+ Yes.		- Jury still out, but likely to win significant use.	

Table 17.1: Comparison of Locking and HTM (“+” is Advantage, “-” is Disadvantage, “⇓” is Strong Disadvantage)

	Locking with RCU or Hazard Pointers		Hardware Transactional Memory	
Basic Idea	Allow only one thread at a time to access a given set of objects.		Cause a given operation over a set of objects to execute atomically.	
Scope	+	Handles all operations.	+	Handles revocable operations.
				– Irrevocable operations force fallback (typically to locking).
Composability	+	Readers limited only by grace-period-wait operations.	↓	Limited by irrevocable operations, transaction size, and deadlock. (Assuming lock-based fallback code.)
	–	Updaters limited by deadlock. Readers reduce deadlock.		
Scalability & Performance	–	Data must be partitionable to avoid lock contention among updaters.	–	Data must be partitionable to avoid conflicts.
	+	Partitioning not needed for readers.		
	↓	Partitioning for updaters must typically be fixed at design time.	+	Dynamic adjustment of partitioning carried out automatically down to cacheline boundaries.
	+	Partitioning not needed for readers.	–	Partitioning required for fallbacks (less important for rare fallbacks).
	↓	Updater locking primitives typically result in expensive cache misses and memory-barrier instructions.	–	Transactions begin/end instructions typically do not result in cache misses, but do have memory-ordering consequences.
	+	Update-side contention effects are focused on acquisition and release, so that the critical section runs at full speed.	–	Contention aborts conflicting transactions, even if they have been running for a long time.
	+	Readers do not contend with updaters or with each other.		
	+	Read-side primitives are typically wait-free with low overhead. (Lock-free for hazard pointers.)	–	Read-only transactions subject to conflicts and rollbacks. No forward-progress guarantees other than those supplied by fallback code.
	+	Privatization operations are simple, intuitive, performant, and scalable when data is visible only to updaters.	–	Privatized data contributes to transaction size.
	–	Privatization operations are expensive (though still intuitive and scalable) for reader-visible data.		
Hardware Support	+	Commodity hardware suffices.	–	New hardware required (and is starting to become available).
	+	Performance is insensitive to cache-geometry details.	–	Performance depends critically on cache geometry.
Software Support	+	APIs exist, large body of code and experience, debuggers operate naturally.	–	APIs emerging, little experience outside of DBMS, breakpoints mid-transaction can be problematic.
Interaction With Other Mechanisms	+	Long experience of successful interaction.	↓	Just beginning investigation of interaction.
Practical Apps	+	Yes.	+	Yes.
Wide Applicability	+	Yes.	–	Jury still out, but likely to win significant use.

Table 17.2: Comparison of Locking (Augmented by RCU or Hazard Pointers) and HTM (“+” is Advantage, “–” is Disadvantage, “↓” is Strong Disadvantage)

4. Hazard pointers and RCU provide forward-progress guarantees (lock freedom and wait-freedom, respectively).
5. Privatization operations for hazard pointers and RCU are straightforward.

Of course, it is also possible to augment HTM, as discussed in the next section.

17.3.4 Where Does HTM Best Fit In?

Although it will likely be some time before HTM's area of applicability can be as crisply delineated as that shown for RCU in Figure 9.34 on page 138, that is no reason not to start moving in that direction.

HTM seems best suited to update-heavy workloads involving relatively small changes to disparate portions of relatively large in-memory data structures running on large multiprocessors, as this meets the size restrictions of current HTM implementations while minimizing the probability of conflicts and attendant aborts and rollbacks. This scenario is also one that is relatively difficult to handle given current synchronization primitives.

Use of locking in conjunction with HTM seems likely to overcome HTM's difficulties with irrevocable operations, while use of RCU or hazard pointers might alleviate HTM's transaction-size limitations for read-only operations that traverse large fractions of the data structure. Current HTM implementations unconditionally abort an update transaction that conflicts with an RCU or hazard-pointer reader, but perhaps future HTM implementations will interoperate more smoothly with these synchronization mechanisms. In the meantime, the probability of an update conflicting with a large RCU or hazard-pointer read-side critical section should be much smaller than the probability of conflicting with the equivalent read-only transaction.¹⁶ Nevertheless, it is quite possible that a steady stream of RCU or hazard-pointer readers might starve updaters due to a corresponding steady stream of conflicts. This vulnerability could be eliminated (perhaps at significant hardware cost and complexity) by giving extra-transactional reads the pre-transaction copy of the memory location being loaded.

The fact that HTM transactions must have fallbacks might in some cases force static partitionability of data

structures back onto HTM. This limitation might be alleviated if future HTM implementations provide forward-progress guarantees, which might eliminate the need for fallback code in some cases, which in turn might allow HTM to be used efficiently in situations with higher conflict probabilities.

In short, although HTM is likely to have important uses and applications, it is another tool in the parallel programmer's toolbox, not a replacement for the toolbox in its entirety.

17.3.5 Potential Game Changers

Game changers that could greatly increase the need for HTM include the following:

1. Forward-progress guarantees.
2. Transaction-size increases.
3. Improved debugging support.
4. Weak atomicity.

These are expanded upon in the following sections.

17.3.5.1 Forward-Progress Guarantees

As was discussed in Section 17.3.2.4, current HTM implementations lack forward-progress guarantees, which requires that fallback software be available to handle HTM failures. Of course, it is easy to demand guarantees, but not always to easy to provide them. In the case of HTM, obstacles to guarantees can include cache size and associativity, TLB size and associativity, transaction duration and interrupt frequency, and scheduler implementation.

Cache size and associativity was discussed in Section 17.3.2.1, along with some research intended to work around current limitations. However, HTM forward-progress guarantees would come with size limits, large though these limits might one day be. So why don't current HTM implementations provide forward-progress guarantees for small transactions, for example, limited to the associativity of the cache? One potential reason might be the need to deal with hardware failure. For example, a failing cache SRAM cell might be handled by deactivating the failing cell, thus reducing the associativity of the cache and therefore also the maximum size of transactions that can be guaranteed forward progress. Given that this would simply decrease the guaranteed transaction size, it seems likely that other reasons are at work. Perhaps providing forward progress guarantees on production-quality

¹⁶ It is quite ironic that strictly transactional mechanisms are appearing in shared-memory systems at just about the time that NoSQL databases are relaxing the traditional database-application reliance on strict transactions.

hardware is more difficult than one might think, an entirely plausible explanation given the difficulty of making forward-progress guarantees in software. Moving a problem from software to hardware does not necessarily make it easier to solve.

Given a physically tagged and indexed cache, it is not enough for the transaction to fit in the cache. Its address translations must also fit in the TLB. Any forward-progress guarantees must therefore also take TLB size and associativity into account.

Given that interrupts, traps, and exceptions abort transactions in current HTM implementations, it is necessary that the execution duration of a given transaction be shorter than the expected interval between interrupts. No matter how little data a given transaction touches, if it runs too long, it will be aborted. Therefore, any forward-progress guarantees must be conditioned not only on transaction size, but also on transaction duration.

Forward-progress guarantees depend critically on the ability to determine which of several conflicting transactions should be aborted. It is all too easy to imagine an endless series of transactions, each aborting an earlier transaction only to itself be aborted by a later transactions, so that none of the transactions actually commit. The complexity of conflict handling is evidenced by the large number of HTM conflict-resolution policies that have been proposed [ATC⁺11, LS11]. Additional complications are introduced by extra-transactional accesses, as noted by Blundell [BLM06]. It is easy to blame the extra-transactional accesses for all of these problems, but the folly of this line of thinking is easily demonstrated by placing each of the extra-transactional accesses into its own single-access transaction. It is the pattern of accesses that is the issue, not whether or not they happen to be enclosed in a transaction.

Finally, any forward-progress guarantees for transactions also depend on the scheduler, which must let the thread executing the transaction run long enough to successfully commit.

So there are significant obstacles to HTM vendors offering forward-progress guarantees. However, the impact of any of them doing so would be enormous. It would mean that HTM transactions would no longer need software fallbacks, which would mean that HTM could finally deliver on the TM promise of deadlock elimination.

And as of late 2012, the IBM Mainframe announced an HTM implementation that includes *constrained transactions* in addition to the usual best-effort HTM implementation [JSG12]. A constrained transaction starts with

the `tbeginC` instruction instead of the `tbegin` instruction that is used for best-effort transactions. Constrained transactions are guaranteed to always complete (eventually), so if a transaction aborts, rather than branching to a fallback path (as is done for best-effort transactions), the hardware instead restarts the transaction at the `tbeginC` instruction.

The Mainframe architects needed to take extreme measures to deliver on this forward-progress guarantee. If a given constrained transaction repeatedly fails, the CPU might disable branch prediction, force in-order execution, and even disable pipelining. If the repeated failures are due to high contention, the CPU might disable speculative fetches, introduce random delays, and even serialize execution of the conflicting CPUs. “Interesting” forward-progress scenarios involve as few as two CPUs or as many as one hundred CPUs. Perhaps these extreme measures provide some insight as to why other CPUs have thus far refrained from offering constrained transactions.

As the name implies, constrained transactions are in fact severely constrained:

1. The maximum data footprint is four blocks of memory, where each block can be no larger than 32 bytes.
2. The maximum code footprint is 256 bytes.
3. If a given 4K page contains a constrained transaction’s code, then that page may not contain that transaction’s data.
4. The maximum number of assembly instructions that may be executed is 32.
5. Backwards branches are forbidden.

Nevertheless, these constraints support a number of important data structures, including linked lists, stacks, queues, and arrays. Constrained HTM therefore seems likely to become an important tool in the parallel programmer’s toolbox.

17.3.5.2 Transaction-Size Increases

Forward-progress guarantees are important, but as we saw, they will be conditional guarantees based on transaction size and duration. It is important to note that even small-sized guarantees will be quite useful. For example, a guarantee of two cache lines is sufficient for a stack, queue, or dequeue. However, larger data structures require larger guarantees, for example, traversing a tree in order

requires a guarantee equal to the number of nodes in the tree.

Therefore, increasing the size of the guarantee also increases the usefulness of HTM, thereby increasing the need for CPUs to either provide it or provide good-and-sufficient workarounds.

17.3.5.3 Improved Debugging Support

Another inhibitor to transaction size is the need to debug the transactions. The problem with current mechanisms is that a single-step exception aborts the enclosing transaction. There are a number of workarounds for this issue, including emulating the processor (slow!), substituting STM for HTM (slow and slightly different semantics!), playback techniques using repeated retries to emulate forward progress (strange failure modes!), and full support of debugging HTM transactions (complex!).

Should one of the HTM vendors produce an HTM system that allows straightforward use of classical debugging techniques within transactions, including breakpoints, single stepping, and print statements, this will make HTM much more compelling. Some transactional-memory researchers are starting to recognize this problem as of 2013, with at least one proposal involving hardware-assisted debugging facilities [GKP13]. Of course, this proposal depends on readily available hardware gaining such facilities.

17.3.5.4 Weak Atomicity

Given that HTM is likely to face some sort of size limitations for the foreseeable future, it will be necessary for HTM to interoperate smoothly with other mechanisms. HTM's interoperability with read-mostly mechanisms such as hazard pointers and RCU would be improved if extra-transactional reads did not unconditionally abort transactions with conflicting writes—instead, the read could simply be provided with the pre-transaction value. In this way, hazard pointers and RCU could be used to allow HTM to handle larger data structures and to reduce conflict probabilities.

This is not necessarily simple, however. The most straightforward way of implementing this requires an additional state in each cache line and on the bus, which is a non-trivial added expense. The benefit that goes along with this expense is permitting large-footprint readers without the risk of starving updaters due to continual conflicts.

17.3.6 Conclusions

Although current HTM implementations appear to be poised to deliver real benefits, they also have significant shortcomings. The most significant shortcomings appear to be limited transaction sizes, the need for conflict handling, the need for aborts and rollbacks, the lack of forward-progress guarantees, the inability to handle irrevocable operations, and subtle semantic differences from locking.

Some of these shortcomings might be alleviated in future implementations, but it appears that there will continue to be a strong need to make HTM work well with the many other types of synchronization mechanisms, as noted earlier [MMW07, MMTW10].

In short, current HTM implementations appear to be welcome and useful additions to the parallel programmer's toolbox, and much interesting and challenging work is required to make use of them. However, they cannot be considered to be a magic wand with which to wave away all parallel-programming problems.

17.4 Functional Programming for Parallelism

When I took my first-ever functional-programming class in the early 1980s, the professor asserted that the side-effect-free functional-programming style was well-suited to trivial parallelization and analysis. Thirty years later, this assertion remains, but mainstream production use of parallel functional languages is minimal, a state of affairs that might well stem from this professor's additional assertion that programs should neither maintain state nor do I/O. There is niche use of functional languages such as Erlang, and multithreaded support has been added to several other functional languages, but mainstream production usage remains the province of procedural languages such as C, C++, Java, and Fortran (usually augmented with OpenMP, MPI, or, in the case of Fortran, coarrays).

This situation naturally leads to the question “If analysis is the goal, why not transform the procedural language into a functional language before doing the analysis?” There are of course a number of objections to this approach, of which I list three:

1. Procedural languages often make heavy use of global variables, which can be updated independently by different functions, or, worse yet, by multiple threads. Note that Haskell's *monads* were invented to deal

with single-threaded global state, and that multi-threaded access to global state requires additional violence to the functional model.

2. Multithreaded procedural languages often use synchronization primitives such as locks, atomic operations, and transactions, which inflict added violence upon the functional model.
3. Procedural languages can *alias* function arguments, for example, by passing a pointer to the same structure via two different arguments to the same invocation of a given function. This can result in the function unknowingly updating that structure via two different (and possibly overlapping) code sequences, which greatly complicates analysis.

Of course, given the importance of global state, synchronization primitives, and aliasing, clever functional-programming experts have proposed any number of attempts to reconcile the function programming model to them, monads being but one case in point.

Another approach is to compile the parallel procedural program into a functional program, the use functional-programming tools to analyze the result. But it is possible to do much better than this, given that any real computation is a large finite-state machine with finite input that runs for a finite time interval. This means that any real program can be transformed into an expression, possibly albeit an impractically large one [DHK12].

However, a number of the low-level kernels of parallel algorithms transform into expressions that are small enough to fit easily into the memories of modern computers. If such an expression is coupled with an assertion, checking to see if the assertion would ever fire becomes a satisfiability problem. Even though satisfiability problems are NP-complete, they can often be solved in much less time than would be required to generate the full state space. In addition, the solution time appears to be independent of the underlying memory model, so that algorithms running on weakly ordered systems can be checked just as quickly as they could on sequentially consistent systems [AKT13].

The general approach is to transform the program into single-static-assignment (SSA) form, so that each assignment to a variable creates a separate version of that variable. This applies to assignments from all the active threads, so that the resulting expression embodies all possible executions of the code in question. The addition of an assertion entails asking whether any combination

of inputs and initial values can result in the assertion firing, which, as noted above, is exactly the satisfiability problem.

One possible objection is that it does not gracefully handle arbitrary looping constructs. However, in many cases, this can be handled by unrolling the loop a finite number of times. In addition, perhaps some loops will also prove amenable to collapse via inductive methods.

Another possible objection is that spinlocks involve arbitrarily long loops, and any finite unrolling would fail to capture the full behavior of the spinlock. It turns out that this objection is easily overcome. Instead of modeling a full spinlock, model a trylock that attempts to obtain the lock, and aborts if it fails to immediately do so. The assertion must then be crafted so as to avoid firing in cases where a spinlock aborted due to the lock not being immediately available. Because the logic expression is independent of time, all possible concurrency behaviors will be captured via this approach.

A final objection is that this technique is unlikely to be able to handle a full-sized software artifact such as the millions of lines of code making up the Linux kernel. This is likely the case, but the fact remains that exhaustive validation of each of the much smaller parallel primitives within the Linux kernel would be quite valuable. And in fact the researchers spearheading this approach have applied it to non-trivial real-world code, including the RCU implementation in the Linux kernel (albeit to verify one of the less-profound properties of RCU).

It remains to be seen how widely applicable this technique is, but it is one of the more interesting innovations in the field of formal verification. And it might be more well-received than the traditional advice of writing all programs in functional form.

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 pro-

```
1 /* WARNING: BUGGY CODE. */
2 void *producer(void *ignored)
3 {
4     int i = 0;
5
6     producer_ready = 1;
7     while (!goflag)
8         sched_yield();
9     while (goflag) {
10         ss.t = dgettimeofday();
11         ss.a = ss.c + 1;
12         ss.b = ss.a + 1;
13         ss.c = ss.b + 1;
14         i++;
15     }
16     printf("producer exiting: %d samples\n", i);
17     producer_done = 1;
18     return (NULL);
19 }
```

Figure A.1: “After” Producer Function

ducer 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.

seq	time (seconds)	delta	a	b	c
17563:	1152396.251585	(-16.928)	27	27	27
18004:	1152396.252581	(-12.875)	24	24	24
18163:	1152396.252955	(-19.073)	18	18	18
18765:	1152396.254449	(-148.773)	216	216	216
19863:	1152396.256960	(-6.914)	18	18	18
21644:	1152396.260959	(-5.960)	18	18	18
23408:	1152396.264957	(-20.027)	15	15	15

Table A.1: “After” Program Sample Output

```

1 /* WARNING: BUGGY CODE. */
2 void *consumer(void *ignored)
3 {
4     struct snapshot_consumer curssc;
5     int i = 0;
6     int j = 0;
7
8     consumer_ready = 1;
9     while (ss.t == 0.0) {
10         sched_yield();
11     }
12     while (goflag) {
13         curssc.tc = dgettimeofday();
14         curssc.t = ss.t;
15         curssc.a = ss.a;
16         curssc.b = ss.b;
17         curssc.c = ss.c;
18         curssc.sequence = curseq;
19         curssc.iserror = 0;
20         if ((curssc.t > curssc.tc) ||
21             modgreater(ssc[i].a, curssc.a) ||
22             modgreater(ssc[i].b, curssc.b) ||
23             modgreater(ssc[i].c, curssc.c) ||
24             modgreater(curssc.a, ssc[i].a + maxdelta) ||
25             modgreater(curssc.b, ssc[i].b + maxdelta) ||
26             modgreater(curssc.c, ssc[i].c + maxdelta)) {
27             i++;
28             curssc.iserror = 1;
29         } else if (ssc[i].iserror)
30             i++;
31         ssc[i] = curssc;
32         curseq++;
33         if (i + 1 >= NSNAPS)
34             break;
35     }
36     printf("consumer exited, collected %d items of %d\n",
37           i, curseq);
38     if (ssc[0].iserror)
39         printf("0/%d: %.6f %.6f (%.3f) %d %d %d\n",
40               ssc[0].sequence, ssc[j].t, ssc[j].tc,
41               (ssc[j].tc - ssc[j].t) * 1000000,
42               ssc[j].a, ssc[j].b, ssc[j].c);
43     for (j = 0; j <= i; j++)
44         if (ssc[j].iserror)
45             printf("%d: %.6f (%.3f) %d %d %d\n",
46                   ssc[j].sequence,
47                   ssc[j].t, (ssc[j].tc - ssc[j].t) * 1000000,
48                   ssc[j].a - ssc[j - 1].a,
49                   ssc[j].b - ssc[j - 1].b,
50                   ssc[j].c - ssc[j - 1].c);
51     consumer_done = 1;
52 }

```

Figure A.2: “After” Consumer Function

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 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.

seq	time (seconds)	delta	a	b	c
58597:	1156521.556296	(3.815)	1485	1485	1485
403927:	1156523.446636	(2.146)	2583	2583	2583

Table A.2: Locked “After” Program Sample Output

Quick Quiz A.2: How could there be such a large gap between successive consumer reads? See `timelocked.c` for full code. ■

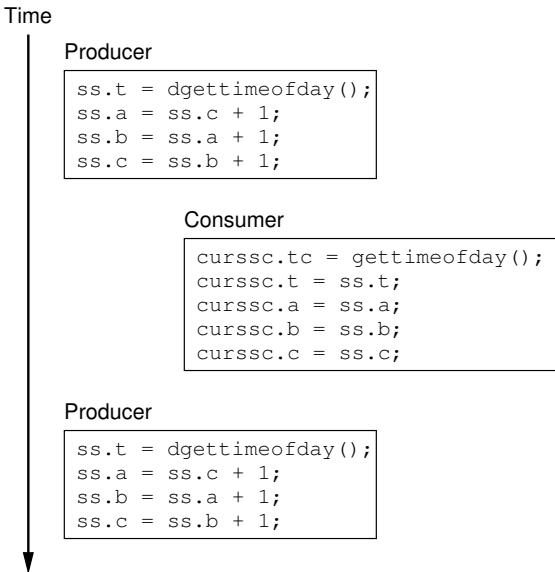


Figure A.3: Effect of Locking on Snapshot Collection

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 6 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.

A.2 What is the Difference Between “Concurrent” and “Parallel”?

From a classic computing perspective, “concurrent” and “parallel” are clearly synonyms. However, this has not stopped many people from drawing distinctions between the two, and it turns out that these distinctions can be understood from a couple of different perspectives.

The first perspective treats “parallel” as an abbreviation for “data parallel,” and treats “concurrent” as pretty much everything else. From this perspective, in parallel computing, each partition of the overall problem can proceed completely independently, with no communication with other partitions. In this case, little or no coordination among partitions is required. In contrast, concurrent computing might well have tight interdependencies, in the form of contended locks, transactions, or other synchronization mechanisms.

Quick Quiz A.3: Suppose a portion of a program uses RCU read-side primitives as its only synchronization mechanism. Is this parallelism or concurrency? ■

This of course begs the question of why such a distinction matters, which brings us to the second perspective, that of the underlying scheduler. Schedulers come in a wide range of complexities and capabilities, and as a rough rule of thumb, the more tightly and irregularly a set of parallel processes communicate, the higher the level of sophistication is required from the scheduler. As such, parallel computing’s avoidance of interdependencies means that parallel-computing programs run well on the least-capable schedulers. In fact, a pure parallel-computing program can run successfully after being arbitrarily subdivided and interleaved onto a uniprocessor.¹ In contrast, concurrent-computing programs might well require extreme subtlety on the part of the scheduler.

One could argue that we should simply demand a reasonable level of competence from the scheduler, so that we could simply ignore any distinctions between parallelism and concurrency. Although this is often a good strategy, there are important situations where efficiency, performance, and scalability concerns sharply limit the level of competence that the scheduler can reasonably offer. One important example is when the scheduler is implemented in hardware, as it often is in SIMD units or GPGPUs. Another example is a workload where the units of work are quite short, so that even a software-based scheduler must make hard choices between subtlety on the one hand and efficiency on the other.

Now, this second perspective can be thought of as making the workload match the available scheduler, with parallel workloads able to operate on a simple scheduler and concurrent workloads requiring more sophisticated schedulers.

Unfortunately, this perspective does not always align with the dependency-based distinction put forth by the

¹ Yes, this does mean that parallel-computing programs are best-suited for sequential execution. Why did you ask?

first perspective. For example, a highly interdependent lock-based workload with one thread per CPU can make do with a trivial scheduler because no scheduler decisions are required. In fact, some workloads of this type can even be run one after another on a sequential machine. Therefore, such a workload would be labeled “concurrent” by the first perspective and “parallel” by many taking the second perspective.

Quick Quiz A.4: In what part of the second (scheduler-based) perspective would the lock-based single-thread-per-CPU workload be considered “concurrent”? ■

Which is just fine. No rule that humankind writes carries any weight against objective reality, including the rule dividing multiprocessor programs into categories such as “concurrent” and “parallel.”

This categorization failure does not mean such rules are useless, but rather that you should take on a suitably skeptical frame of mind when attempting to apply them to new situations. As always, use such rules where they apply and ignore them otherwise.

In fact, it is likely that new categories will arise in addition to parallel, concurrent, map-reduce, task-based, and so on. Some will stand the test of time, but good luck guessing which!

A.3 What Time Is It?

A key issue with timekeeping on multicore computer systems is illustrated by Figure A.4. One problem is that it takes time to read out the time. An instruction might read from a hardware clock, and might have to go off-core (or worse yet, off-socket) to complete this read operation. It might also be necessary to do some computation on the value read out, for example, to convert it to the desired format, to apply network time protocol (NTP) adjustments, and so on. So does the time eventually returned correspond to the beginning of the resulting time interval, the end, or somewhere in between?

Worse yet, the thread reading the time might be interrupted or preempted. Furthermore, there will likely be some computation between reading out the time and the actual use of the time that has been read out. Both of these possibilities further extend the interval of uncertainty.

One approach is to read the time twice, and take the arithmetic mean of the two readings, perhaps one on each side of the operation being timestamped. The difference between the two readings is then a measure of uncertainty of the time at which the intervening operation occurred.

Of course, in many cases, the exact time is not necessary. For example, when printing the time for the benefit of a human user, we can rely on slow human reflexes to render internal hardware and software delays irrelevant. Similarly, if a server need to timestamp the response to a client, any time between the reception of the request and the transmission of the response will do equally well.

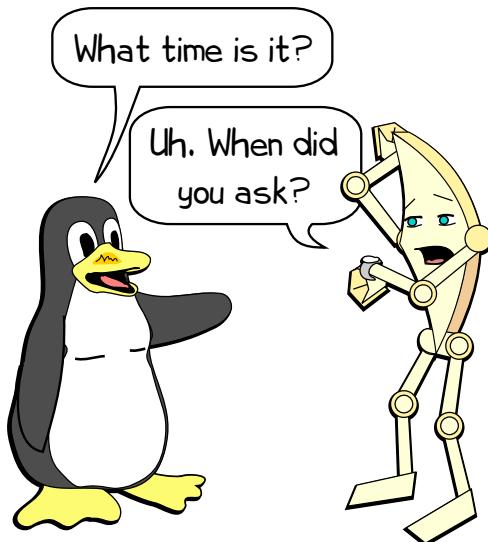


Figure A.4: What Time Is It?

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.

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

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.¹ 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,² which includes program counter and stack.

The thread API is shown in Figure B.1, and members are described in the following sections.

B.2.1 `create_thread()`

The `create_thread()` primitive creates a new thread, starting the new thread’s execution at the function

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?

```

int smp_thread_id(void)
thread_id_t create_thread(void *(*func)(void *), void *arg)
for_each_thread(t)
for_each_running_thread(t)
void *wait_thread(thread_id_t tid)
void wait_all_threads(void)

```

Figure B.1: Thread API

`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 at the end of 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.

```

1 void *thread_test(void *arg)
2 {
3     int myarg = (int)arg;
4
5     printf("child thread %d: smp_thread_id() = %d\n",
6           myarg, smp_thread_id());
7     return NULL;
8 }

```

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.

```

1 int main(int argc, char *argv[])
2 {
3     int i;
4     int nkids = 1;
5
6     smp_init();
7     if (argc > 1) {
8         nkids = strtoul(argv[1], NULL, 0);
9         if (nkids > NR_THREADS) {
10             fprintf(stderr, "nkids=%d too big, max=%d\n",
11                     nkids, NR_THREADS);
12             usage(argv[0]);
13         }
14     }
15     printf("Parent spawning %d threads.\n", nkids);
16     for (i = 0; i < nkids; i++)
17         create_thread(thread_test, (void *)i);
18     wait_all_threads();
19     printf("All threads completed.\n", nkids);
20     exit(0);
21 }
```

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.

```

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.

B.3.5 Example Usage

A spinlock named `mutex` may be used to protect a variable `counter` as follows:

```

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 simplify coding.

```

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-thread variable. Such a variable can be defined as follows:

```
DEFINE_PER_THREAD(int, counter);
```

The counter must be initialized as follows:

```
init_per_thread(counter, 0);
```

A thread can increment its instance of this counter as follows:

```
__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:

```
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++`”).

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.¹

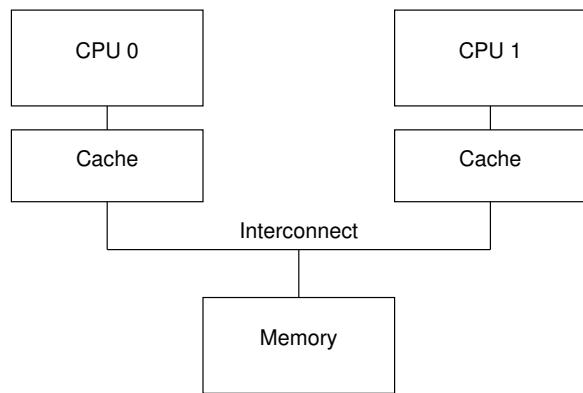


Figure C.1: Modern Computer System Cache Structure

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.

¹ 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.

	Way 0	Way 1
0x0	0x12345000	
0x1	0x12345100	
0x2	0x12345200	
0x3	0x12345300	
0x4	0x12345400	
0x5	0x12345500	
0x6	0x12345600	
0x7	0x12345700	
0x8	0x12345800	
0x9	0x12345900	
0xA	0x12345A00	
0xB	0x12345B00	
0xC	0x12345C00	
0xD	0x12345D00	
0xE	0x12345E00	0x43210E00
0xF		

Figure C.2: CPU Cache Structure

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 size, 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 0x12345F00. 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,² 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.

Quick Quiz C.1: Where does a writeback message originate from and where does it go to? ■

² 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.

Interestingly enough, a shared-memory multiprocessor system really is a message-passing computer under the 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.2: What happens if two CPUs attempt to invalidate the same cache line concurrently? ■

Quick Quiz C.3: 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.4: 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.

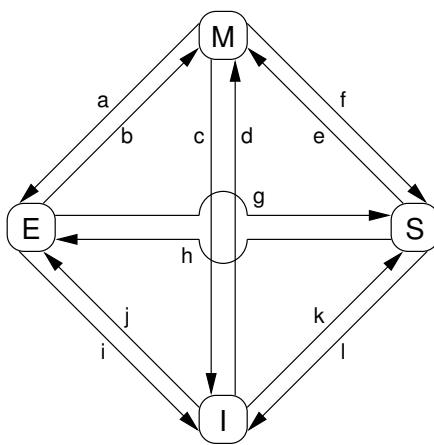


Figure C.3: MESI Cache-Coherency State Diagram

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 an atomic 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 an atomic 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, this 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.5: 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.6: 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.

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.

Sequence #	CPU #	Operation	CPU Cache				Memory	
			0	1	2	3	0	8
0		Initial State	-/I	-/I	-/I	-/I	V	V
1	0	Load	0/S	-/I	-/I	-/I	V	V
2	3	Load	0/S	-/I	-/I	0/S	V	V
3	0	Invalidation	8/S	-/I	-/I	0/S	V	V
4	2	RMW	8/S	-/I	0/E	-/I	V	V
5	2	Store	8/S	-/I	0/M	-/I	I	V
6	1	Atomic Inc	8/S	0/M	-/I	-/I	I	V
7	1	Writeback	8/S	8/S	-/I	-/I	V	V

Table C.1: Cache Coherence Example

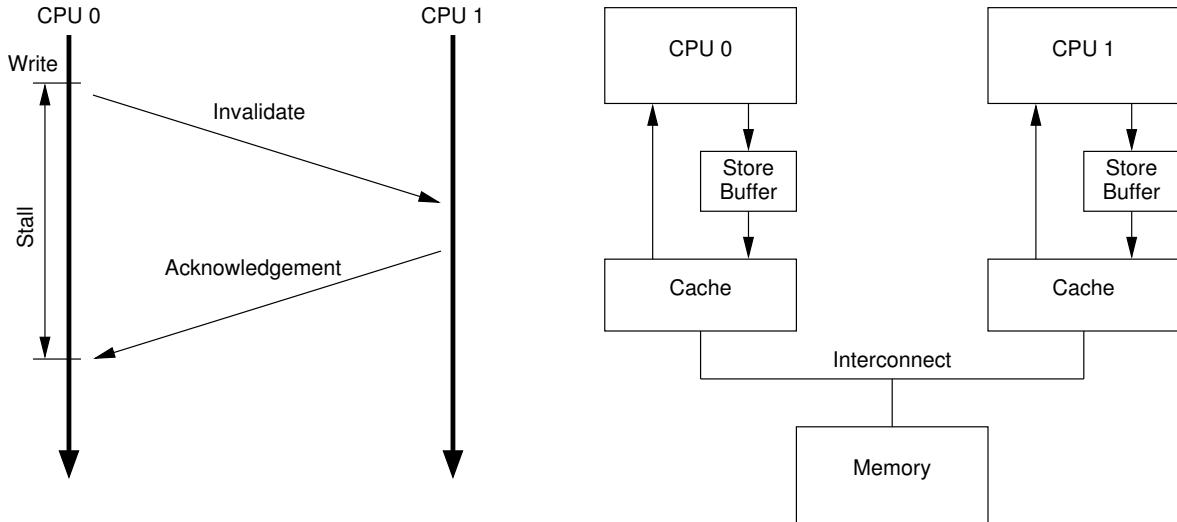


Figure C.4: Writes See Unnecessary Stalls

Figure C.5: Caches With Store Buffers

Quick Quiz C.7: But if the main purpose of store buffers is to hide acknowledgment latencies in multiprocessor cache-coherence protocols, why do uniprocessors also have store buffers? ■

These store buffers are local to a given CPU or, on systems with hardware multithreading, local to a given core. Either way, a given CPU is permitted to access only the store buffer assigned to it. For example, in Figure C.5, CPU 0 cannot access CPU 1's store buffer and vice versa. This restriction simplifies the hardware by separating concerns: The store buffer improves performance for consecutive writes, while the responsibility for communicating among CPUs (or cores, as the case may be) is fully shouldered by the cache-coherence protocol. However, even given this restriction, there are complications that must be addressed, which are covered in the next two sections.

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:

```

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”.
8. CPU 0 loads “a” from its cache, finding the value zero.
9. CPU 0 applies the entry from its store buffer 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.

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.

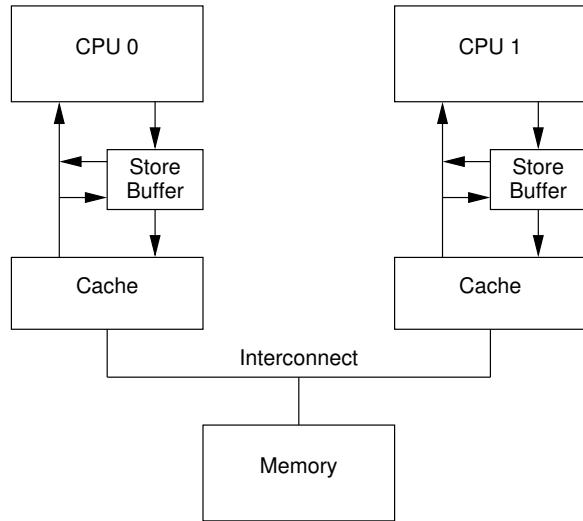


Figure C.6: Caches With Store Forwarding

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.
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.8: 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:

```

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 }
```

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 CPU 1. It also marks its own copy of this cache line as “shared”.
16. CPU 1 receives the cache line containing “b” and installs it in its cache.
17. CPU 1 can now load the value of “b”, and since it finds that the value of “b” is 1, it exits the `while` loop and proceeds to the next statement.
18. CPU 1 executes the `assert (a == 1)`, but the cache line containing “a” is no longer in its cache. Once it gets this cache from CPU 0, it will be working with the up-to-date value of “a”, and the assertion therefore passes.

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.

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.

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.

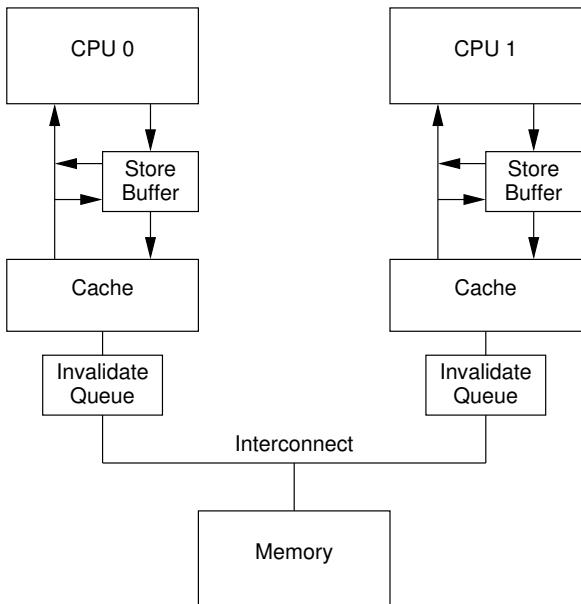


Figure C.7: Caches With Invalidate Queues

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:

```

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.9: 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:

```

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.10: 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 buffer.

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:

```

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

A number of ordering-hostile computer systems have been produced over the decades, 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 de-

sign 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.11: 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

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], Peter Sewell's work [Sew], or the excellent hardware-oriented primer by Sorin, Hill, and Wood [SHW11].

⁵ Any real hardware architect or designer will no doubt be objecting strenuously, 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".

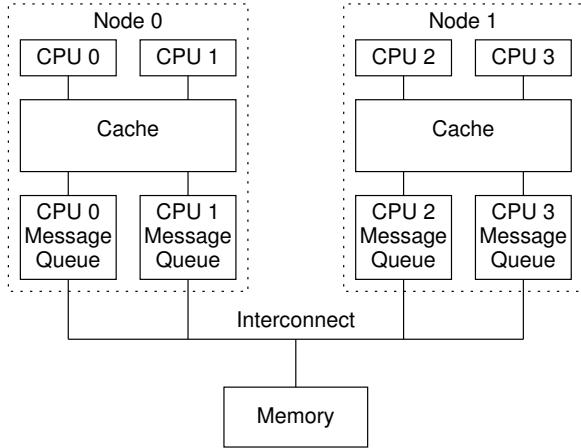


Figure C.8: Example Ordering-Hostile Architecture

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. 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.

Therefore, portable code cannot rely on this assertion not firing, as both the compiler and the CPU can reorder the code so as to trip the assertion.

Quick Quiz C.12: Could this code be fixed by inserting a memory barrier between CPU 1’s “while” and assignment to “c”? Why or why not? ■

C.6.3 Example 2

Table C.3 shows three code fragments, executed concurrently by CPUs 0, 1, and 2. Both “a” and “b” are initially zero.

Again, 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” 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 “b” will sail through CPU 1’s previously empty queue. Therefore, CPU 2 might well see CPU 1’s assignment to “b” before it sees CPU 0’s assignment to “a”, causing the assertion to fire, despite the memory barriers.

In theory, portable code should not rely on this example code fragment, however, as before, in practice it actually does work on most mainstream computer systems.

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.

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.13: 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.14: If CPU 2 executed an assert ($e == 0 \mid c == 1$) 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.

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

CPU 0	CPU 1	CPU 2
<pre>a = 1; smp_wmb(); b = 1;</pre>	<pre>while (b == 0); c = 1;</pre>	<pre>z = c; smp_rmb(); x = a; assert(z == 0 x == 1);</pre>

Table C.2: Memory Barrier Example 1

CPU 0	CPU 1	CPU 2
<pre>a = 1;</pre>	<pre>while (a == 0); smp_mb(); b = 1;</pre>	<pre>y = b; smp_rmb(); x = a; assert(y == 0 x == 1);</pre>

Table C.3: Memory Barrier Example 2

	CPU 0	CPU 1	CPU 2
1	<pre>a = 1;</pre>		
2	<pre>smp_wmb();</pre>		
3	<pre>b = 1;</pre>	<pre>while (b == 0);</pre>	<pre>while (b == 0);</pre>
4		<pre>smp_mb();</pre>	<pre>smp_mb();</pre>
5		<pre>c = 1;</pre>	<pre>d = 1;</pre>
6	<pre>while (c == 0);</pre>		
7	<pre>while (d == 0);</pre>		
8	<pre>smp_mb();</pre>		
9	<pre>e = 1;</pre>		<pre>assert(e == 0 a == 1);</pre>

Table C.4: Memory Barrier Example 3

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.

	Loads Reordered After Loads?	Loads Reordered After Stores?	Stores Reordered After Stores?	Stores Reordered After Loads?	Atomic Instructions Reordered With Loads?	Atomic Instructions Reordered With Stores?	Dependent Loads Reordered?	Incoherent Instruction Cache/Pipeline?
Alpha	Y	Y	Y	Y	Y	Y	Y	Y
AMD64				Y				
ARMv7-A/R	Y	Y	Y	Y	Y	Y		Y
IA64	Y	Y	Y	Y	Y	Y		Y
MIPS	Y	Y	Y	Y	Y	Y		Y
(PA-RISC)	Y	Y	Y	Y				
PA-RISC CPUs								
POWER™	Y	Y	Y	Y	Y	Y		Y
(SPARC RMO)	Y	Y	Y	Y	Y	Y		Y
(SPARC PSO)			Y	Y		Y		Y
SPARC TSO				Y				Y
x86				Y				Y
(x86 OOStore)	Y	Y	Y	Y				Y
zSeries®				Y				Y

Table C.5: Summary of Memory Ordering

The platforms with a non-no-op `mmiowb()` 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 14.2 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 (spin-locks, 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.

```

1 struct el *insert(long key, long data)
2 {
3     struct el *p;
4     p = kmalloc(sizeof(*p), GFP_ATOMIC);
5     spin_lock(&mutex);
6     p->next = head.next;
7     p->key = key;
8     p->data = data;
9     smp_wmb();
10    head.next = p;
11    spin_unlock(&mutex);
12 }
13
14 struct el *search(long key)
15 {
16     struct el *p;
17     p = head.next;
18     while (p != &head) {
19         /* BUG ON ALPHA!!! */
20         if (p->key == key) {
21             return (p);
22         }
23         p = p->next;
24     };
25     return (NULL);
26 }
```

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.

Figure C.10 shows how this can happen on an aggressively parallel machine with partitioned caches, so that alternating cache 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

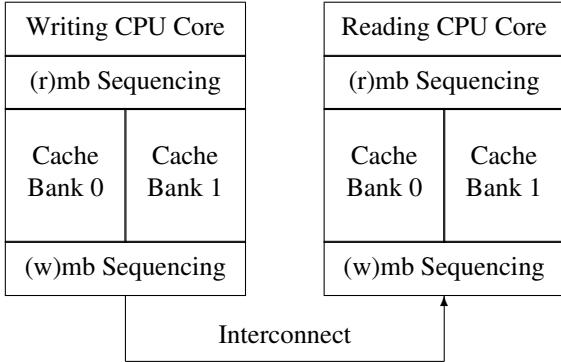


Figure C.10: Why `smp_read_barrier_depends()` is Required

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

One could place an `smp_rmb()` primitive between the pointer fetch and dereference. 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.

It is also possible to implement a software barrier that could be used 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,

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.

```

1 struct el *insert(long key, long data)
2 {
3     struct el *p;
4     p = kmalloc(sizeof(*p), GFP_ATOMIC);
5     spin_lock(&mutex);
6     p->next = head.next;
7     p->key = key;
8     p->data = data;
9     smp_wmb();
10    head.next = p;
11    spin_unlock(&mutex);
12 }
13
14 struct el *search(long key)
15 {
16     struct el *p;
17     p = head.next;
18     while (p != &head) {
19         smp_read_barrier_depends();
20         if (p->key == key) {
21             return (p);
22         }
23         p = p->next;
24     }
25     return (NULL);
26 }

```

Figure C.11: Safe Insert and Lock-Free Search

implementing a memory-barrier shutdown. 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.15: 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 updated its documented 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.7.7, 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:

```

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,⁷ 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

⁷ ARMv8 has recently added load-acquire and store-release instructions.

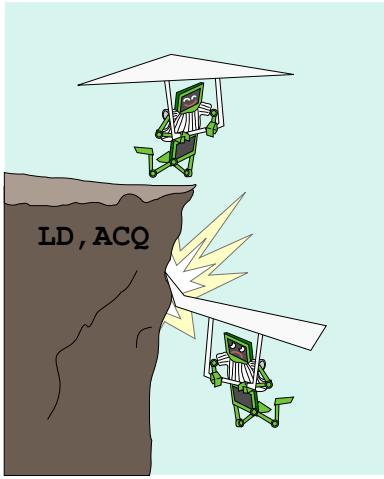


Figure C.12: Half Memory Barrier

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 MIPS

The MIPS memory model [Ima15, Table 6.6] appears to resemble that of ARM, IA64, and Power, being weakly ordered by default, but respecting dependencies. MIPS has a wide variety of memory-barrier instructions, but ties them not to hardware considerations, but rather to the use cases provided by the Linux kernel and the C++11 standard [Smi15] in a manner similar to the ARM64 additions:

SYNC Full barrier for a number of hardware operations in addition to memory references.

SYNC_WMB Write memory barrier, which can be used to implement the `smp_wmb()` primitive in the Linux kernel.

SYNC_MB Full memory barrier, but only for memory operations. This may be used to implement the Linux-kernel `smp_mb()` and the C++ `atomic_thread_fence(memory_order_seq_cst)`.

SYNC_ACQUIRE Acquire memory barrier, which may be used to implement the Linux-kernel

`smp_load_acquire()` and the C++ `atomic_load_explicit(..., memory_order_acquire)`.

SYNC_RELEASE Release memory barrier, which may be used to implement the Linux-kernel `smp_store_release()` and the C++ `atomic_store_explicit(..., memory_order_release)`.

SYNC_RMB Read memory barrier, which can be used to implement the `smp_rmb()` primitive in the Linux kernel.

Informal discussions with MIPS architects indicates that MIPS has a definition of transitivity or cumulativity similar to that of ARM and Power. However, it appears that different MIPS implementations can have different memory-ordering properties, so it is important to consult the documentation for the specific MIPS implementation you are using.

C.7.6 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.7.7 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. The `lwsync` instruction may be used to implement load-acquire and store-release operations.

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 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 Silveira [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.7.8 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.7.9 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; addl`. This atomic instruction acts as a barrier to both loads and stores.

Intel has also published a memory model for x86 [Int07]. It turns out that Intel's actual CPUs enforced tighter ordering than was claimed in the previous specifi-

cations, 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; addl`.

Although newer x86 implementations accommodate self-modifying code without any special instructions, to be fully compatible with past and potential future x86 implementations, a given CPU must execute a jump instruction or a serializing instruction (e.g., `cpuid`) between modifying the code and executing it [Int11, Section 8.1.3].

C.7.10 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

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 `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. Similarly, you need to flush the CPU caches of any location in any DMA buffer after DMA to that buffer completes. 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

Answers to Quick Quizzes

그 외의 퀴즈들은 컨퍼런스 발표 중에, 그리고 이 책에서 다루는 내용을 다루는 수업 중에 받은 실제 질문과 유사합니다. 그리고 일부 퀴즈는 저자의 관점에서 쓰이기도 했습니다.

D.1 How To Use This Book

Quick Quiz 1.1:

이 Quick Quiz 들의 답은 어디에 있을까요?



Answer:

페이지 349에서 시작하는 Appendix D.

쉽죠?

Quick Quiz 1.2:

몇몇 퀴즈는 저자의 입장이 아니라 독자의 입장에서 쓰인 것 같은데요. 그런 의도가 맞나요?



Answer:

실제로 그렇답니다! 많은 질문들은 Paul E. Mckenny 가 이 내용들을 다루는 수업을 듣는 학생이었다면 질문했을 법한 것들입니다. Paul 은 이 내용들을 교수님으로부터가 아니라 병렬 하드웨어와 소프트웨어로부터 배웠다는 내용도 짚어둬야 할 것 같네요. Paul 의 경험에 의하면, 교수님들은 Watson 같은 실제 병렬 시스템과는 달리 말로 이야기되는 문제에 대해 답을 주려 하곤 합니다. 물론, 어떤 교수님들이나 병렬 시스템들이 이런 종류의 문제들에 대해 가장 유용한 답을 주는지에 대해서는 많은 토론이 가능하겠습니까만, 지금은 일단 실제 교수님들과 병렬 시스템들에 따라 그들이 주는 답의 유용성이 다를 수 있다는 점만 동의하고 넘어갑시다.

Quick Quiz 1.3:

전 퀴즈를 좋아하지 않아요. 어떡하죠?



Answer:

여기 몇가지 전략이 있습니다:

1. 그냥 퀴즈를 무시하고 책을 읽으세요. 퀴즈의 흥미로운 내용들을 놓치게 되겠지만 이 책의 퀴즈를 제외한 부분도 훌륭한 내용을 많이 담고 있습니다. 만약 당신의 목표가 일반적인 내용에 대한 이해를 얻는 것이거나 이 책을 통해 특정 문제에 대한 해결책을 찾는 것이라면 충분히 합리적인 접근법입니다.
2. 퀴즈가 집중을 방해하지만 무시하기엔 중요하다고 생각한다면, 언제든 이 책의 소스를 git 저장소에서 클론할 수 있음을 기억하세요. 그리고나서 Makefile 과 qqz.sty 를 수정해서 퀴즈가 PDF에서 사라지게 할 수 있습니다. 또는, 해당 파일들을 수정해서 답변이 문제 바로 아래 나오도록 수정할 수도 있습니다.
3. 당신의 답을 구하느라 너무 많은 시간을 보내지 말고 곧바로 답을 보세요. 현재 퀴즈의 답이 당신이 해결하려는 문제의 핵심을 쥐고 있는 게 아니라면 이것도 합리적인 접근법입니다. 또한, 당신이 원하는게 해당 내용에 대한 깊은 이해이지, 새로이 병렬성을 활용한 해결책을 맨바닥부터 만들려 하는게 아닌 경우에도 이는 합리적인 접근법입니다.

D.2 Introduction

Quick Quiz 2.1:

여봐요!!! 병렬 프로그래밍은 수십년간 엄청나게 어렵다고 알려졌다구요. 근데 당신은 그게 그렇게 어렵지 않다고 슬쩍 이야기하는 것 같네요. 뭘 개수작이요?



Answer:

정말 병렬 프로그래밍이 엄청나게 어렵다고 믿는다면, “왜 병렬 프로그래밍이 어려운가?”라는 질문에 대답할 준비가 되어 있을 겁니다. 누군가는 여러 이유를 댈 수 있겠죠, 테드락부터 레이스 컨디션, 테스팅 커버리지 등등. 하지만 진짜 답은 그건 그렇게까지 어렵지는 않다입니다. 일단, 만약 병렬 프로그래밍이 정말로 그렇게 소름끼치도록 어렵다면, 어떻게 Apache 나 MySQL, 리눅스 커널 같은 많은 오픈 소스 프로젝트들이 그걸 잘 사용하고 있겠어요?

보다 나은 질문은 아마도 이렇겠죠: “왜 병렬 프로그래밍은 그렇게 어렵다고 알려져 있을까?” 답을 알기 위해, 1991년으로 돌아가 봅시다. Paul McKenney는 주차장에서 6개의 dual-80486 Sequent Symmetry CPU 보드를 들고 Sequent의 벤치마킹 센터로 걸어가던 중, 문득 자신이 집 몇채 가격의 물건을 들고 있음을 깨달았습니다.¹ 이렇게 엄청난 병렬 시스템의 가격은 병렬 프로그래밍이 병렬 시스템을 직접 제작하거나 — 1991년의 미국 달러로 — \$100,000 이상의 가격의 기계를 구매할 수 있는 회사에서 일하는 제한된 일부 특권층의 일부에게만 가능했음을 의미합니다.

하지만, 2006년, Paul은 자신이 이 글을 dual-core x86 노트북에서 쓰고 있음을 발견합니다. 앞서 이야기한 dual-80486 CPU 보드와 달리, 이 노트북은 2GB 메인 메모리, 60GB 디스크 드라이브, 모니터, 이더넷, USB 포트, 무선랜, 그리고 블루투스까지 달려 있습니다. 그리고 그 노트북은 그간의 인플레이션을 고려하지 않더라도 dual-80486 CPU 보드보다 열배가 넘게 싹니다.

병렬 시스템이 정말로 세상에 도래했습니다. 병렬 시스템은 더이상 일부 특권층의 소유물이 아니라 거의 모든 사람에게 가능한 물건입니다.

기존의 제한적이었던 병렬 하드웨어 접근성이야말로 병렬 프로그래밍이 그렇게 어렵다고 여겨지게 만들었던 진짜 이유입니다. 무엇보다, 아무리 단순한 기계라도 직접 만져볼 수가 없다면 프로그램하기는 매우 어렵습니다. 찾기 어렵고 비싼 패럴렐 머신들의 시대는 갔으니 병렬 프로그래밍이 미치도록 어렵다고 생각하

¹ 그래요. 이 갑작스런 깨달음은 그가 좀 더 조심히 걷게 만들었습 니다. 왜 그런걸 물어야죠?

던 시대는 곧 지나갈 겁니다.²

Quick Quiz 2.2:

어떻게 병렬 프로그래밍이 시퀀셜 프로그래밍만큼 쉬운게 가능한가요?



Answer:

그건 프로그래밍 환경에 달려 있습니다. SQL [Int92]는 잘 알려지지 않은 성공 사례인데요, 병렬성에 대해 잘 모르는 프로그래머도 거대한 병렬 시스템을 바삐 동작하게 만들 수 있도록 해주기 때문이죠. 병렬 컴퓨터는 갈수록 싸고 어디서나 접할 수 있게 되어가고 있기 때문에 이런 류의 다양한 예를 볼 수 있을 겁니다. 예를 들어, 과학 / 기술 컴퓨팅 쪽에서의 가능할 법한 경쟁자는 흔한 행렬 연산을 자동으로 병렬화 시켜주는 MATLAB*P입니다.

마지막으로, 리눅스와 유닉스 시스템에서의 다음 셀 커맨드를 생각해 보세요:

```
get_input | grep "interesting" | sort
```

이 셀 파이프라인은 get_input, grep, 그리고 sort를 병렬적으로 처리합니다. 어때요, 어렵지 않았죠, 됐죠?

요약하자면, 병렬 프로그래밍은 시퀀셜 프로그래밍 만큼이나 쉽습니다. 적어도 병렬성을 사용자에게서 숨겨주는 환경에서는요!

Quick Quiz 2.3:

헐, 진짜요? 정확성, 관리성, 내구성 같은 것들은요?



Answer:

그것들도 중요한 목표들이죠, 하지만 병렬 프로그램에서 그런 목표들의 중요도는 시퀀셜 프로그램에서의 그것 정도일 뿐입니다. 따라서, 그것들은 중요한 목표들이긴 하지만 병렬 프로그래밍만의 목표에는 속하지 않습니다.

Quick Quiz 2.4:

그리고 정확성, 관리성, 내구성이 해당되지 않는데 왜 생산성과 Generality는 해당되는거죠?



² 병렬 프로그래밍은 시퀀셜 프로그래밍보다는 어렵습니다. 예를 들어, 병렬적으로 validation을 하는 것은 더 어렵습니다. 하지만 더 이상 미칠듯이 어렵진 않아요.

Answer:

병렬 프로그래밍이 시퀀셜 프로그래밍보다 훨씬 어렵다고 인식되고 있는 만큼, 생산성도 달성하기 어려운 목표로 여겨지고 있고, 따라서 반드시 이 목표를 이뤄야 합니다. 또한, SQL과 같이 높은 생산성을 갖는 병렬 프로그래밍 환경은 특정한 용도에만 사용 가능하기 때문에, Generality도 반드시 목표에 들어가야 합니다.

Quick Quiz 2.5:

병렬 프로그램은 정확성을 증명하기가 어렵다고 알고 있는데, 정말 정확성도 그 목록에 올라갈 수 없는 건가요?

**Answer:**

엔지니어링 관점에서 형식적이든 비형식적이든 정확성을 증명하는 건 엔지니어링 관점에서의 최대 목표인 생산성에 어떤 영향을 미치느냐에 따라 중요도가 정해집니다. 따라서, 정확성 증명이 중요한 경우라면 “생산성” 아래 포함된다고 볼 수 있겠죠.

**Quick Quiz 2.6:**

그냥 재미를 목표로 하는 건 어떤가요?

**Answer:**

재미도 물론 중요하죠, 하지만 당신이 취미로만 사는 사람이 아니라면 재미가 당신의 최우선 목표는 아닐 겁니다. 거꾸로 말하자면, 당신이 취미로만 사는 사람이 맞다면 좋은 자세입니다!

Quick Quiz 2.7:

성능 이외의 이유로 병렬 프로그래밍을 하는 경우도 있나요?

**Answer:**

풀어야 하는 문제가 본질적으로 병렬적인 경우가 있습니다. 예를 들어, Monte Carlo method들과 일부 숫자 계산들이요. 하지만, 이런 경우에도 병렬성을 관리하기 위해 많은 추가 작업이 필요합니다.

병렬성은 가끔 신뢰성(reliability)을 위해 사용되기도 합니다. 일단 예를 하나들자면, triple-modulo redundancy는 병렬로 동작하는 세 개의 시스템을 가지고 결과에 대해 투표를 합니다. 극단적 경우에는 세 개의 시스템이 서로 다른 알고리즘과 기술을 가지고 독립적으로

구현될 수도 있습니다.

Quick Quiz 2.8:

왜 이런 비기술적인 문제를 이야기하는 거죠??? 그저 비기술적일 뿐 아니라, 심지어 생산성이라니요? 누가 그런 걸 신경 써요?

**Answer:**

당신이 순수히 취미로만 사는 사람이라면 아마 당신은 신경쓰지 않아도 될 겁니다. 하지만 설명 그렇다 해도 얼마나 빨리 그리고 얼마나 많이 일을 할 수 있는지는 신경쓸 겁니다. 무엇보다, 가장 유명한 취미가용 도구는 보통 그 목적에 가장 적합한 도구이고, “가장 적합한” 이란 말의 정의의 가장 중요한 부분은 생산성과 연결되어 있죠. 그리고 만약 누군가가 당신에게 병렬 코드를 작성하라고 돈을 준다면, 그들은 당신의 생산성에 대해 매우 신경쓸 겁니다. 그리고 그 고용주가 뭔가에 신경쓴다면, 당신은 거기에 적어도 관심을 가져야겠죠!

그리고, 만약 당신이 정말로 생산성에 신경쓰지 않는다면, 애초에 컴퓨터를 사용하지 않고 손으로 일을 했겠죠!

Quick Quiz 2.9:

병렬 시스템이 그렇게 싼 가격이 되었다면, 어떤 사람이 그걸 프로그램 하라고 월급을 줘가며 프로그래머를 고용하겠어요?



이 질문에는 몇 가지 답이 있습니다:

1. 거대한, 여러 병렬머신들로 구성된 클러스터가 있다고 하면, 이 클러스터의 전체 비용은 상당한 개발 노력을 정당화합니다. 개발 비용은 수많은 머신들 전체에게 적용되기 때문이죠.
2. 수천만명이 넘는 사용자들이 사용하는 유명한 소프트웨어라면 상당한 개발 노력이 정당화 됩니다. 그 개발 노력은 수천만 사용자를 위한 거니까요. 커널이나 시스템 라이브러리 같은 것들도 이 경우에 들어감을 참고하세요.
3. 낮은 가격의 병렬 머신이 중요한 어떤 장비의 운영에 사용되고 있다면 그 장비의 가격 일부분이 상당한 개발 비용을 정당화 할 수 있습니다.
4. 안전을 위해 사용되는 주요 시스템은 사람의 목숨을 보호합니다. 따라서 이 경우에는 매우 큰 개발 비용을 정당화 하죠.

5. 취미가와 연구자들은 돈보다는 지식, 경험, 재미, 그리고 명예를 추구합니다.

그러니까 하락하는 하드웨어 가격은 소프트웨어를 의미 없게 만들지 않고, 오히려 소프트웨어 개발 비용을 하드웨어 가격에 “숨기는” 것이 불가능해진 겁니다. 적어도 엄청나게 많은 수의 하드웨어를 사용하는 경우가 아니라면요.

Quick Quiz 2.10:

이건 달성 불가한 이상에 불과해요! 현실적으로 달성 가능한 무언가에 집중하는게 어때요? ■

Answer:

이건 분명 달성 가능합니다. 휴대폰은 프로그래밍이나 환경구성 없이 최종 사용자가 전화 통화를 하고 텍스트 메세지를 주고 받을 수 있게 해주는 컴퓨터입니다.

일견 사소한 예처럼 보일 수 있겠지만, 천천히 생각해 보면 이건 간단하기도 하고 심오하기도 한 이야기입니다. generality 를 희생하면 우리는 놀랍도록 높은 생산성 향상을 얻을 수 있습니다. 과한 generality 에 빠진 사람들은 그래서 소프트웨어 스택의 최대치까지 성능을 끌어올리는데 실패하곤 합니다. 이 삶의 진리는 약자도 있죠: YAGNI, 즉 “You Ain’t Gonna Need It.”

Quick Quiz 2.11:

잠깐만요! 이런 접근법은 단순히 개발을 위한 노력을 당신으로부터 누군가 그 존재한다는 병렬 소프트웨어를 만드는 사람에게 전가할 뿐인 거 아닌가요? ■

Answer:

바로 그겁니다! 그리고 그게 바로 이미 있는 소프트웨어를 쓰는 것의 요점이죠. 한 팀의 작업물이 많은 다른 팀에 의해 사용되어서 모든 팀이 불필요하게 바퀴를 재발명하는 것에 비해 훨씬 노력을 줄이게 되는것이요.

Quick Quiz 2.12:

어떤 다른 병목지점들이 CPU 를 추가해도 성능을 개선되지 않게 할 수 있을까요? ■

Answer:

잠재적 병목지점이 얼마든지 있습니다:

1. 메인 메모리. 싱글 쓰레드가 모든 가용한 메모리를 사용하고 있다면, 추가된 쓰레드는 단순히 명령하게 자신을 페이지 아웃 시키겠죠.

2. 캐시. 싱글 쓰레드의 캐시 사용량이 모든 공유 CPU 캐시(들)을 꽉 채운다면, 쓰레드를 추가하는 것은 그저 영향받는 캐시들을 쓰래쉬 하기만 할겁니다.
3. 메모리 밴드위쓰. 싱글 쓰레드가 모든 메모리 밴드위쓰를 소모한다면, 추가된 쓰레드들은 그저 메모리로의 시스템 접점에 줄을 서 있을겁니다.
4. I/O 밴드위쓰. 싱글쓰레드가 I/O 에 바운드 되어 있다면, 쓰레드들을 추가하는 것은 그저 그들 모두 관련된 I/O 자원에 줄을 서서 기다리고만 있게 될겁니다.

특정 하드웨어 시스템들은 추가적인 병목지점을 얼마든지 가지고 있을 수 있습니다. 다만 분명한 건 여러 CPU 들이나 쓰레드들 간에 공유되고 있는 자원은 잠재적 병목지점입니다.

Quick Quiz 2.13:

CPU 캐시 용량 외에, 뭐가 동시에 수행되는 쓰레드들의 갯수를 제한해야 하게 할 수 있을까요? ■

Answer:

쓰레드 갯수에 영향을 끼치는 여러 잠재적 요소들이 있습니다:

1. 메인 메모리. 각 쓰레드는 (최소한 스택을 위해) 메모리를 일부 사용하므로, 너무 많은 쓰레드는 메모리를 모조리 사용해버려서 엄청나게 과도한 페이징이나 메모리 할당 실패를 일으킬 수 있습니다.
2. I/O 밴드위쓰. 각 쓰레드가 많은 스토리지 I/O 나 네트워크 트래픽을 만든다면 너무 많은 수의 쓰레드는 과도한 I/O 큐잉 딜레이를 일으키고, 결국 성능이 또 저하될 것입니다. 일부 네트워킹 프로토콜은 너무 많은 쓰레드가 네트워킹 이벤트를 만들어 시간 내에 그 응답을 받지 못할 경우 타임아웃이나 다른 문제상황을 낼 수 있습니다.
3. 동기화 오버헤드. 많은 동기화 프로토콜에서 과도한 수의 쓰레드는 지나친 스피닝, 블락킹, 또는 롤백을 일으켜서 성능을 떨어뜨릴 수 있습니다.

특정한 어플리케이션이나 플랫폼에 따라서는 이외에도 추가적인 요소가 얼마든지 있을 수 있습니다.

Quick Quiz 2.14:

병렬 프로그래밍에 다른 어려움은 없나요?



Answer:

병렬 프로그래밍에의 수많은 잠재적 문제들이 존재합니다. 여기 그 중 일부를 이야기 해보죠:

1. 주어진 프로젝트의 하나 뿐인 알고리즘이 본질적으로 순차적일 수 있습니다. 이 경우에는 (당신의 프로젝트가 반드시 병렬로 돌아야 한다는 법적 조항이 없다면) 병렬 프로그래밍을 관두거나 새로운 병렬 알고리즘을 고안해내야 합니다.
2. 프로젝트가 동일 어드레스 스페이스를 사용하지만 바이너리로만 제공되는 플러그인을 허용해서 모든 개발자가 프로젝트의 모든 소스 코드에 접근할 수는 없는 경우가 있을 수 있습니다. 데드락을 포함해 많은 병렬성에 기인한 버그들이 여기저기 있기 때문에, 그런 바이너리로만 제공되는 플러그인은 현재의 소프트웨어 개발 방법 하에서는 상당한 어려움을 안겨줄 수 있습니다. 물론 미래에는 상황이 바뀔 수도 있지만 현재로썬 주어진 어드레스 스페이스를 공유하는 병렬 코드의 모든 개발자는 그 어드레스 스페이스에서 돌아가는 모든 모드를 들여다 볼 수 있어야 합니다.
3. 프로젝트가 병렬성을 고려하지 않은채 설계된 API [AGH⁺11a, CKZ⁺13]를 엄청나게 사용하는 경우. System V 메세지 큐 API의 매우 화려한 기능들이 이 경우에 속합니다. 물론, 만약 당신의 프로젝트가 수십년 넘게 존속되었다면, 그리고 그 개발자들이 병렬 하드웨어를 접해본 적 없었다면 그 프로젝트는 분명 그런 API들을 최소한 사용은 하고 있을 겁니다.
4. 프로젝트가 병렬성에 대한 고려 없이 구현된 경우. 순차적 환경에서는 매우 잘 동작하지만 병렬 환경에서는 처참하게 동작하는 기술이 있기 때문에, 만약 당신의 프로젝트가 순차적 하드웨어에서만 그 동안 사용되어왔다면 당신의 프로젝트는 분명 병렬성에 친화적이지 못한 코드를 최소한 사용은 하고 있을 겁니다.
5. 프로젝트가 좋은 소프트웨어 개발 관습에 대한 고려 없이 구현된 경우. 잔혹한 사실은, 공유 메모리 병렬 환경은 종종 순차적 환경에 비해 대충 만들어진 개발 관습에 더 엄혹하다는 것입니다. 이 경우에는 병렬성을 도입하기 전에 먼저 기존의 설계와 코드를 재정리 해야할 겁니다.
6. 당신의 프로젝트를 처음 개발한 사람들이 여전히 관리 권한을 쥐고 있거나 작은 기능 정도는 추가할 수 있는 기능을 가지고 있지만 “커다란” 변경은 할

수 없는 경우. 이런 경우에는 당신이 매우 간단하게 당신의 프로젝트를 병렬화 할 수 있다 해도, 순차적인 채로 놔두는게 최선일 수 있습니다. 그렇다 해도 여러 인스턴스를 수행시킨다면지, 많이 사용하는 라이브러리의 병렬적 구현체를 사용한다던지, database 와 같은 다른 병렬 프로젝트의 사용을 하도록 한다던지와 같이 간단하게 당신의 프로젝트를 병렬화 시킬 수 있는 방법이 있습니다.

이런 문제들은 비기술적인 요소들이라고 말할 수도 있겠죠, 하지만 그렇다고 이것들이 비현실적이지도 않습니다. 요약하자면, 커다란 코드의 병렬화는 크고 복잡한 노력을 필요로 할 수 있습니다. 그리고 크고 복잡한 노력이 필요하다면, 그 숙제를 가능한 빨리 해결하는게 낫겠죠.

D.3 Hardware and its Habits

Quick Quiz 3.1:

왜 병렬 프로그래머가 하드웨어의 로우 레벨 요소들까지 배워야 하죠? 하이 레벨의 추상 계층만 보는게 더 쉽고, 낫고, 더 일반적이지 않겠어요?

**Answer:**

하드웨어의 세세한 내용들은 무시하는게 더 쉬울 수 있을 겁니다만, 많은 경우 그건 바보같은 짓일 수 있습니다. 병렬성의 모든 목적이 성능 향상일 뿐이란걸 인정하신다면, 그리고 성능은 하드웨어의 디테일한 부분들에 의존적인 걸 인정하신다면, 논리적으로 병렬 프로그래머들은 하드웨어에 대해 최소 조금은 알아야 한다는 결론을 얻을 수 있을 겁니다.

이건 대부분의 엔지니어링 교훈에서 나오는 이야기입니다. 당신이라면 콘크리트와 철강에 대해 이해하지 못하는 엔지니어가 설계한 다리를 사용하시겠습니까? 아니라면, 왜 병렬 프로그래머가 최소한 조금의 하드웨어에 대한 이해 없이 훌륭한 병렬 소프트웨어를 만들 수 있을 거라고 생각하시나요?

Quick Quiz 3.2:

어떤 기계가 복수 데이터 요소에 대한 어토믹 오퍼레이션을 허용하겠어요?

**Answer:**

이 질문에 대한 한가지 답은 종종 복수개의 데이터

요소를 어토믹하게 다뤄질 수 있는, 단일 머신 워드 안에 모아넣을 수 있다는 겁니다.

좀 더 트렌디한 답은 트랜잭션 메모리 [Lom77]를 지원하는 기계가 되겠습니다. 2014년 초에 이르러서는 일부 주요 시스템들이 제한되긴 했지만 하드웨어 트랜잭션 메모리 구현을 제공합니다. 더 자세한 내용은 Section 17.3에서 다루고 있습니다. 소프트웨어 트랜잭션 메모리 [MMW07, PW07, RHP⁺07, CBM⁺08, DFGG11, MS12]에 대해서는 아직 적합하지 않다는 평가입니다. 소프트웨어 트랜잭션 메모리에 대한 더 많은 내용은 Section 17.2에서 볼 수 있을 겁니다.

Quick Quiz 3.3:

그래서, CPU 설계자들은 캐시 미스 오버헤드 역시 많이 개선 했나요? ■

Answer:

안타깝지만, 그렇게 많은 개선은 하지 못했습니다. 약간 오버헤드를 줄인 CPU들도 있었습니다만, 빛의 속도의 한계와 물질의 원자성의 자연 법칙이 큰 시스템에서 캐시 미스 오버헤드를 줄일 수 있는 방법을 제한하고 있습니다. Section 3.3에서 가능할 법한 미래의 개선 방법들을 논의해 봅니다.

Quick Quiz 3.4:

이제 간략화된 거라구요? 이것보다 더 복잡한게 어떻게 가능하죠? ■

Answer:

이 예는 다음을 포함해 몇 가지 가능한 복잡한 경우를 봅습니다:

1. 해당 캐시라인에 대해 다른 CPU들도 동사에 CAS 오퍼레이션을 수행하려 하고 있을 수 있습니다.
2. 해당 캐시라인은 리드 온리로 다른 CPU들의 캐시들에 복사되어 있을 수 있는데, 이 경우엔 그 캐시들도 비워야 할 필요가 생깁니다.
3. CPU 7은 해당 요청이 도착했을 때 해당 캐시라인에 뭔가 연산을 수행하고 있었을 수 있고, 이 경우 CPU 7은 자신의 연산이 끝날 때까지 해당 요청을 잠시 대기하고 있게 해야 합니다.
4. CPU 7은 (예를 들어, 다른 데이터를 위한 공간을 만들기 위해) 해당 캐시라인을 캐시에서 없앴을 수 있고, 이로 인해 요청이 도착한 시점에서는 캐시라인이 메모리에 있을 수 있습니다.

5. 캐시라인에서 고칠 수 있는 에러가 났을 수 있는데, 그렇다면 해당 데이터가 사용되기 전에 그 에러는 고쳐져야 합니다.

제품 품질의 캐시 일관성 메커니즘들은 이런 종류의 여러 복잡한 경우 [HP95, CSG99, MHS12, SHW11] 때문에 엄청나게 복잡합니다.

Quick Quiz 3.5:

왜 CPU 7의 캐시에서 해당 캐시라인을 비워야 하죠?

■

Answer:

만약 해당 캐시라인이 CPU 7의 캐시에서 비워지지 않는다면, CPU 0과 CPU 7은 같은 변수에 대해 서로 다른 값을 보게 될 겁니다. 이런 종류의 비일관성은 병렬 소프트웨어를 매우 복잡하게 만들 수 있고, 때문에 하드웨어 설계자들은 그런 문제를 없애려 노력해 왔습니다.

Quick Quiz 3.6:

하드웨어 설계자들은 분명 이 상황을 개선하려 노력할 수 있었을 거예요! 왜 그들은 이 단일 인스트럭션 오퍼레이션들의 끔찍한 성능을 만족하고 있는거죠? ■

Answer:

하드웨어 설계자들은 이 문제를 해결하려 노력했었습니다만, 물리학자 스티븐 호킹 정도의 권위자에게만 조언을 얻었습니다. 호킹은 하드웨어 설계자들이 두개의 기본 문제 [Gar07]를 가지고 있음을 발견했습니다:

1. 빛의 한계 속도, 그리고
2. 물질의 원자적 본성.

첫번째 문제는 기본 속도를 제한하고, 두번째 문제는 동작의 소형화를 제한하여 결과적으로 단위시간당 가능한 오퍼레이션의 갯수를 제한합니다. 그리고 이 문제는 설령, 현재 상품화된 CPU들의 속도를 10 GHz 아래로 제한하고 있는, 에너지 소비 문제를 피해간다 해도 존재합니다.

Table D.1과 페이지 22의 Table 3.1를 비교해 보면 알 수 있겠지만, 분명 개선은 이루어지고 있습니다. 하드웨어 쓰레드들을 단일 코어에 집어넣고 복수의 코어들을 하나의 다이에 넣는 것은 적어도 싱글 코어내에서 또는 단일 다이 내에서의 반응속도는 엄청나게 개선했습니다. 전체 시스템 반응속도에서도 일부 개선이 있었습니다만, 겨우 대략 두배 정도입니다. 안타깝지만, 지난 몇년간 빛의 속도나 물질의 원자성의 본질은 변하지 않았습니다.

Operation	Cost (ns)	Ratio (cost/clock)
Clock period	0.4	1.0
“Best-case” CAS	12.2	33.8
Best-case lock	25.6	71.2
Single cache miss	12.9	35.8
CAS cache miss	7.0	19.4
Off-Core		
Single cache miss	31.2	86.6
CAS cache miss	31.2	86.5
Off-Socket		
Single cache miss	92.4	256.7
CAS cache miss	95.9	266.4
Comms Fabric	4,500	7,500
Global Comms	195,000,000	324,000,000

Table D.1: Performance of Synchronization Mechanisms on 16-CPU 2.8GHz Intel X5550 (Nehalem) System

Section 3.3 에서는 병렬 프로그래머들의 곤경을 완화시키기 위해 어떤 일을 해줄 수 있을지 알아봅니다.

Quick Quiz 3.7:

숫자가 미친듯이 크군요! 어떡해야 제 머리로 이걸 이해할 수 있을까요? ■

Answer:

휴지 한 룰을 가져오세요. 미제 휴지라면, 한 룰의 휴지는 약 350-500 조각의 휴지로 구성됩니다. 한개 조각이 하나의 클락 사이클이라고 생각하고, 휴지를 모두 풀어 보세요.

그게 하나의 CAS 캐시 미스를 의미한다고 보면 됩니다.

더 비싼 시스템간 커뮤니케이션 대기시간을 보려면, 몇개의 휴지 룰 (또는 여러 휴지 케이스들) 이 필요할 겁니다.

중요한 팁 하나: 당신이 살아가면서 필요한 휴지가 얼마나 되는지도 생각해보세요!

Quick Quiz 3.8:

하지만 개별의 전자들은 컨덕터 내에서조차도 그렇게 빠르지 않아요!!! 세미컨덕터에서 발견된 저전력의 컨덕터 안에서의 전자 이동 속도는 초당 겨우 1 밀리미터 정도라구요. 뭔가요???

■

Answer:

전자 이동 속도는 긴 시간동안의 개별 전자들의 이동을 추적합니다. 개별 전자들은 꽤 무작위적으로 튀어다니고, 따라서 그들의 순간 속도는 매우 빠르지만 긴 시간으로 보게 되면 그렇게 멀리 이동하지는 않습니다. 여기서, 전자들은 대부분의 시간을 고속으로 이동하는데 소모하지만 긴 시간으로 보면 어디에도 가지 않는 통근자와도 같습니다. 이런 통근자들의 속도는 시속 70 마일(113 킬로미터) 정도지만, 지구의 표면에 비교해 보는 긴 시간동안의 이동 속도는 제로에 가까울 겁니다.

따라서, 우리는 전자의 이동속도가 아니라 순간적 속도에 주의를 기울여야 합니다. 하지만, 전자의 순간적 속도라 하더라도 빛의 속도에는 밸끌도 따라가지 못합니다. 컨덕터에서 측정된 전자파의 속도는 더도 아니고 덜도 아니고 빛의 속도의 밸끌은 따라가고 있는데, 이 때문에 여전히 미스테리는 풀리지 않습니다.

하나 더 있는 트릭은 전자는 그 음극의 성질로 인해 다른 전자와 상당히(원자적 관점에서요) 상호작용을 한다는 것입니다. 이 상호작용은 광자에 의해 이끌어지는 데, 광자는 바로 빛의 속도로 움직입니다. 따라서 전기학에서의 전자라 해도, 대부분의 일을 하는건 광자입니다.

통근자 비유를 이어가 보자면, 운전자는 다른 운전자에게 사고나 교통 혼잡들을 알리는데 스마트폰을 사용할 수 있고, 이로 인해 교통 상황의 변화를 개별 차들의 순간 속도보다 훨씬 빠르게 전파할 수 있는 겁니다. 이 전기학과 교통상황 사이의 비유를 요약하자면 다음과 같습니다:

1. 전자의 (매우 낮은) 이동 속도는 통근자의 장시간 속도와 비슷해서, 둘 다 제로에 가깝습니다.
2. 전자의 (여전히 낮은) 순간 속도는 통행 중인 차의 순간 속도와 비슷합니다. 둘 다 이동 속도에 비해선 높지만, 변화가 전달되는 속도에 비교하면 굉장히 작습니다.
3. 전자파의 (훨씬 높은) 전달 속도는 대부분 전자들 사이에서 전자기력을 전달하는 광자의 덕분입니다. 유사하게, 교통 상황은 운전자 사이의 커뮤니케이션으로 인해 훨씬 빠르게 바뀔 수 있습니다. 이 것은 이미 교통 혼잡에 빠져 있는 운전자에겐 큰 도움이 되지 않듯이, 이미 주어진 캐퍼시티에 잡혀 있는 전자들에겐 큰 도움이 되지 않습니다.

물론, 이 주제를 완전히 이해하려면 전자기학을 공부해야 할겁니다.

Quick Quiz 3.9:

분산 시스템에서 통신이 그렇게까지 비싸다면 누가, 그리고 왜 그걸 쓰려 하는 건가요? ■

Answer:

몇가지 이유가 있지요:

1. 공유 메모리 멀티 프로세서 시스템은 크기 제한이 있습니다. 수천개 이상의 CPU가 필요하다면, 분산 시스템을 사용하는 것밖에 선택지가 없습니다.
2. 극단적으로 거대한 공유 메모리 시스템은 매우 비싸고, Table 3.1에 나타난 것처럼 작은 네개 CPU로 구성된 시스템에서보다도 긴 캐시 미스 대기시간을 갖는 경향을 보입니다..
3. 분산 시스템에서의 통신 대기시간은 CPU를 사용하지 않고, 따라서 메세지가 전달되는 동안 컴퓨팅 연산을 병렬적으로 수행할 수 있습니다.
4. 많은 중요한 문제들은 “당황스럽도록 병렬적”이라서 극단적일 정도로 거대한 연산 단위들이 매우 작은 수의 메세지만으로 가능해 질수도 있습니다. SETI@HOME [aCB08]은 그런 어플리케이션의 한 예입니다. 이런 부류의 어플리케이션들은 극단적으로 긴 통신 대기시간에도 불구하고 컴퓨터 네트워크를 훌륭하게 사용할 수 있습니다.

병렬 어플리케이션에서의 향후 노력은 긴 통신 대기 시간을 가진 기계와, 또는 클러스터에서 잘돌아갈 수 있는 당황스럽도록 병렬적인 어플리케이션의 수를 늘려가는 것을 계속할 것입니다. 그렇다면 해도, 하드웨어 대기시간을 크게 줄이는 것은 개발에 크게 도움이 될겁니다.

Quick Quiz 3.10:

좋아요, 우리가 분산 프로그래밍 기법들을 공유 메모리 병렬 프로그램에 적용하려 한다면, 항상 이런 분산 기법들을 사용하고 공유 메모리 없이 살면 안되나요? ■

Answer:

많은 경우 프로그램의 작은 부분만이 성능에 민감하기 때문입니다. 공유 메모리 병렬성은 우리가 그 작은 부분에의 분산 프로그래밍에 집중하고, 성능에 민감하지 않은 프로그램의 대체분의 영역은 간단한 공유 메모리 기법을 사용하도록 해줍니다.

D.4 Tools of the Trade

Quick Quiz 4.1:

하지만 이 간단한 셸 스크립트는 진짜 병렬 프로그램이 아니잖아요! 왜 이런 별거아닌 걸 신경쓰는거죠???

■

Answer:

당신은 결코 이 간단한 것을 잊을 수 없을 것이기 때문입니다!

이 책의 제목이 “Is Parallel Programming Hard, And, If So, What Can You Do About It?” 이란 걸 마음에 새겨 두십시오. 당신이 할 수 있는 가장 효과적인 일은 그 간단한 것을 잊지 않도록 하는 것입니다! 무엇보다, 당신이 병렬 프로그래밍을 어려운 방법으로 하기로 선택했다면, 당신의 선택이니, 당신은 당신 자신 외의 누구에게도 불평 할 수 없습니다.

Quick Quiz 4.2:

병렬 셸 스크립트를 작성하는 좀 더 간단한 방법은 없나요? 만약 있다면, 어떻게 하나요? 없다면, 왜 없죠?

■

Answer:

가장 직관적인 방법은 셸 파이프라인입니다:

```
grep $pattern1 | sed -e 's/a/b/' | sort
```

충분히 커다란 입력 파일에 대해서, grep의 패턴 매칭, sed의 수정과 sort의 입력물 처리는 병렬적으로 수행될 겁니다. parallel.sh 파일에 셸 스크립트 병렬성과 파이프라인에 대한 데모가 있습니다.

Quick Quiz 4.3:

하지만 스크립트 기반 병렬 프로그래밍이 그렇게 쉽다면, 왜 다른 것들을 신경쓰는거죠?

■

Answer:

사실 오늘날 사용되는 병렬 프로그램들의 매우 많은 부분들이 스크립트에 기반합니다. 하지만, 스크립트 기반 병렬성은 한계점도 지니고 있습니다:

1. 새 프로세스의 생성은 보통 비싼 시스템콜인 `fork()`와 `exec()`를 포함하기 때문에 상당히 무거운 작업입니다.

2. 파일라이닝을 포함해서 데이터의 공유는 일반적으로 비싼 file I/O를 포함합니다.
3. 스크립트에서 믿고 쓸 수 있는 사용 가능한 동기화 기본 도구들 역시 일반적으로 비싼 file I/O를 포함합니다.

이런 제한점들은 스크립트 기반 병렬성이 coarse-grained 병렬성을 사용하고 각 일의 단위들은 최소 수십 밀리세컨드, 그리고 가능하다면 그보다도 훨씬 긴 시간을 가질 것을 요구합니다.

finer-grained 병렬성을 필요로 하는 작업들은 그 작업의 문제가 coarse-grained 형태로 표현될 수는 없을지 좀 고민해 보도록 추천됩니다. 만약 불가능하다면, Section 4.2에서 다루는 것과 같은 다른 병렬 프로그래밍 환경을 고려해 봐야 합니다.

Quick Quiz 4.4:

왜 이 `wait()` 함수는 그렇게 복잡해야만 하는거죠? 왜 그냥 쉘 스크립트의 `wait` 같이 동작하도록 만들지 않는 거예요?



Answer:

일부 병렬 어플리케이션은 특정 자식 프로세스가 끝났을 때 특별한 행동을 취해야 할 수 있고, 그 때문에 각 자식 프로세스에 대해 개별적으로 대기를 할 필요가 있습니다. 또한, 일부 병렬 어플리케이션들은 자식 프로세스가 종료된 이유를 알 필요도 있습니다. Figure 4.3에서 본 것처럼, `wait()` 함수를 가지고 `waitall()` 함수를 만드는 건 어렵지 않습니다만 그 반대는 불가능하겠지요. 한번 특정 자식 프로세스에 대한 정보를 잃어버리면, 그건 복구될 수 없습니다.

Figure 4.4의 부모 프로세스는 자식 프로세스가 종료될 때까지 자신의 `printf()`를 위해 기다리고 있다는 것을 기억해 둘 필요가 있습니다. `printf()`의 buffered I/O를 같은 파일에 대해 여러 프로세스에서 동시적으로 사용하는 것은 일반적이지 않고, 그러지 않는 게 최선입니다. 정말로 동시적으로 buffered I/O를 해야만 한다면, 당신의 OS의 문서를 보세요. UNIX/Linux 시스템에서는 Stewart Weiss의 강의 노트가 예제 [Wei13]와 함께 좋은 소개를 제공합니다.

Quick Quiz 4.6:

Figure 4.5의 `mythread()` 함수가 그냥 리턴해도 된다면, 왜 `pthread_exit()`를 신경써야하죠?



Answer:

이 간단한 예제에서는 `pthread_exit()`를 신경 쓸 이유가 없는게 맞습니다. 하지만, `mythread()`가 별도로 컴파일된 다른 함수를 호출하는 경우를 생각해 봅시다. 그런 경우, `pthread_exit()`는 이런 다른 함수들에서도 별도의 다른 에러들을 리턴하거나 해서 실행 흐름을 `mythread()`에 되돌리거나 할 필요 없이 곧바로 쓰레드의 실행을 종료시킬 수 있게 합니다.

Quick Quiz 4.7:

C 언어가 데이터 레이스에 대해 어떤 보장도 하지 않는다면, 왜 리눅스 커널은 그렇게 많은 데이터 레이스들을 가지고 있는거죠? 지금 리눅스 커널이 완전 영망이라고 이야기 하려는 거예요???



Answer:

아, 하지만 리눅스 커널은 조심스럽게 선택된, 데이터 레이스 상황에서도 안전한 실행을 가능하게 하는 `asm`과 같은 `gcc`의 특수한 확장 기능을 포함하는 C 언어의 슈퍼셋으로 작성되었습니다. 또한, 리눅스 커널은 데이터 레이스가 특히나 문제가 되는 플랫폼들 위에서는 동작하지 않습니다. 예를 들어, 32비트 포인터와 16비트 버스를 갖는 임베디드 시스템을 생각해 보세요. 그런 시스템에서는 하나의 포인터에 값을 저장하고 읽어오는 데이터 레이스에서 읽기는 아래쪽 16비트는 예전 값이고 위쪽 16비트는 새 값인 값을 읽어올 수도 있을 겁니다.

Quick Quiz 4.8:

제가 여러 쓰레드들이 한번에 같은 락을 쥐고 있게 하고 싶으면 어떻게 하죠?

Answer:

가장 먼저 당신이 해야할 일은 왜 그려질 원하는지 스스로에게 물어보는 겁니다. 만약 답이 “나는 많은 쓰레드에 의해 읽혀지고 아주 가끔 수정되는 많은 데이터를 가지고 있기 때문”이라면, POSIX 리더-라이터 락이 당신이 찾고 있는 것일 수 있습니다. 이것들은 Section 4.2.4에 소개되어 있습니다.

여러 쓰레드가 같은 락을 잡고 있는 것과 같은 효과를 얻는 또 다른 방법은 한 쓰레드가 락을 획득하고 나서 `pthread_create()` 함수를 이용해 다른 쓰레드들을 생성하는 것입니다. 왜 이게 좋은 방법인지는 독자 여러분께서 생각해 보시기 바랍니다.

Quick Quiz 4.9:

왜 그냥 Figure 4.6 라인 5에서 `lock_reader()` 가 곧바로 `pthread_mutex_t` 포인터를 받도록 하지 않는거죠?

Answer:

`lock_reader()` 를 `pthread_create()` 에 넘겨야 하기 때문이죠. 물론 함수를 `pthread_create()` 에 넘길 때 캐스팅을 해서 넘길 수도 있지만, 함수 캐스팅은 좀 보기도 안좋고 간단한 포인터 캐스팅에 비해 잘 하기가 어렵습니다.

Quick Quiz 4.10:

`pthread_mutex_t` 의 획득과 해제에 매번 4줄이나 써야한다니 좀 고통스러울 것 같군요! 더 나은 방법은 없나요?

Answer:

실로 그렇습니다! 그리고 그런 이유로, `pthread_mutex_lock()` 과 `pthread_mutex_unlock()` 함수들은 보통 이 에러 체킹을 해주는 함수로 감싸져서 사용되곤 합니다. 뒤에서, 우리는 이들을 리눅스 커널의 `spin_lock()` 과 `spin_unlock()` API 들로 감싸서 사용할 겁니다.

Quick Quiz 4.11:

“ $x = 0$ ” 만이 Figure 4.7 의 코드에서 발생 가능한 오로지

하나의 결과인가요? 만약 그렇다면, 왜죠? 아니라면, 어떤 다른 결과가 가능할까요, 그리고 왜일까요?

Answer:

아닙니다. “ $x = 0$ ” 가 나온 이유는 `lock_reader()` 가 락을 먼저 잡았기 때문입니다. `lock_writer()` 가 먼저 락을 잡았다면, 결과는 “ $x = 3$ ” 가 되었을 것입니다. 하지만, 해당 코드에서는 `lock_reader()` 를 먼저 시작시키고 이 실행은 멀티프로세서에서 이루어졌기 때문에, 대부분은 일반적으로 `lock_reader()` 가 락을 먼저 잡을 것으로 예상할 수 있을 겁니다. 하지만, 보장된 건 아니지요, 특히나 바쁜 시스템에서는요.

Quick Quiz 4.12:

서로 다른 락을 사용하는건 쓰레드가 서로 상대의 중간 상태를 볼 수 있는등 혼란스럽게 할 수 있는 것같은데요. 잘 짜여진 병렬 프로그램은 이런 혼란을 막기 위해서는 하나의 락만을 사용해야만 하는 건가요?

Answer:

가끔은 프로그램을 하나의 전역적인 락만을 사용하면서 잘 동작하고 확장성도 좋게 작성하는 것도 가능하지만, 그런 프로그램은 좀 예외적인 경우입니다. 당신은 좋은 성능과 확장성을 위해선 보통은 여러개의 락을 사용해야 할겁니다.

이 규칙에 대해 하나의 가능한 예외는 아직은 연구 단계에 머물러 있는, “트랜잭션 메모리”입니다. 트랜잭션 메모리는 하나의 전역 락을 사용하면서 허용된 최적화를 사용하고, 추가적으로 롤백을 지원하는 케이스 [Boe09]로 간략히 생각할 수 있습니다.

Quick Quiz 4.13:

Figure 4.8 에 보여진 코드에서, `lock_reader()` 는 `lock_writer()` 가 생성하는 값 모두를 보도록 보장되어 있나요? 그렇다면, 또 그렇지 않다면, 왜죠?

Answer:

아닙니다. 바쁜 시스템에서라면, `lock_reader()` 는 `lock_writer()` 의 실행이 완료될 때까지 CPU 를 선점당해 `lock_writer()` 의 x 중간 값을 전혀 볼 수 없을 수도 있습니다.

Quick Quiz 4.14:

잠깐만요!!! Figure 4.7 에서는 공유 변수 `x` 를 초기화 하지 않았는데, Figure 4.8 에서는 왜 초기화 해야 했던거죠?



Answer:

Figure 4.6 의 라인 3 을 보세요. Figure 4.7 의 코드는 먼저 수행되었기 때문에, `x` 의 컴파일 타임 초기화에 의존할 수도 있었습니다. Figure 4.8 는 그 다음에 돌았기 때문에, `x` 를 다시 초기화 해야 합니다.



Quick Quiz 4.15:

여기 저기 모든 곳에서 `ACCESS_ONCE()` 를 쓰는 대신에, Figure 4.9 의 라인 10에서 `goflag` 를 `volatile` 로 선언하는게 어때요?



Answer:

이 경우에는 `volatile` 로의 선언도 합리적인 대안입니다. 하지만, `ACCESS_ONCE()` 의 사용은 코드를 읽는 사람에게 `goflag` 가 동시적 리드와 업데이트 동작에 연관되어 있음을 분명하게 보여줍니다. 하지만, `ACCESS_ONCE()` 는 특히나 대부분의 접근이 락에 의해 보호되고 있지만 (따라서 변화에 종속되지 않지만) 락 바깥에서의 접근도 약간 있는 경우에 유용합니다. `volatile` 선언을 이런 경우에 사용하는 것은 코드를 읽는 사람이 락 바깥에서의 특수한 접근의 경우를 알아채기가 어렵게 만들고 컴파일러가 락 아래의 코드에 대해 좋은 코드를 만들기 어렵게 할 수 있습니다.



Quick Quiz 4.16:

`ACCESS_ONCE()` 는 컴파일러에만 영향을 주지, CPU 에는 영향을 안주죠. Figure 4.9 의 `goflag` 의 값의 변화가 시간 순서대로 다른 CPU 에게도 전파되게 하려면 메모리 배리어도 쳐야 하지 않나요?



Answer:

아니오, 메모리 배리어는 여기선 필요하지도 않고 도움을 주지도 않습니다. 메모리 배리어들은 그저 여러 메모리 참조들 사이의 순서만을 잡아줍니다: 그것들은 시스템의 한 부분에서 다른 곳으로 데이터 전파를 촉진시키는 어떤 일도 하지 않습니다. 이것이 하나의 경험적 법칙을 일깨웁니다: 여러 쓰레드들 사이에 두개 이상의 변수를 사용해 통신하고 있지 않다면 메모리 배리어는 필요하지 않습니다.

하지만 `nreadersrunning` 의 경우는 어떨까요?

통신에 사용되는 두번째 변수 아닙니까? 맞습니다, 그리고 `__sync_fetch_and_add()` 아래에 해당 쓰레드가 시작해야 하는지 보기 전에 자신의 존재를 분명히 알리기 위해 필요한 메모리 배리어가 있습니다.

Quick Quiz 4.17:

예를 들어 `gcc __thread` 스토리지 클래스를 사용해 선언된 쓰레드별 변수에 접근할 때에도 `ACCESS_ONCE()` 가 필요할까요?



Answer:

경우에 따라 다릅니다. 만약 그 쓰레드별 변수가 그 쓰레드에서만 접근되었다면, 그리고 시그널 핸들러에서 접근되지 않았다면, 필요하지 않습니다. 하지만 그렇지 않다면, `ACCESS_ONCE()` 가 필요할 수 있습니다. 각 상황을 모두 Section 5.4.4 에서 보겠습니다.

이 이야기는 어떻게 한 쓰레드가 다른 쓰레드의 `__thread` 변수에 접근할 수 있는지 질문을 가져오는데, 답은 두번째 쓰레드가 자신의 `__thread` 변수로의 포인터를 첫번째 쓰레드가 접근할 수 있는 곳에 저장해 둠으로써 가능하다입니다. 이런 코드를 작성하는 흔한 경우 중 하나는 쓰레드당 한개씩의 원소를 갖는 링크드 리스트에 대해 각 쓰레드의 `__thread` 변수를 해당하는 원소에 저장하는 경우입니다.

Quick Quiz 4.18:

단일 CPU 성능에 비교하는건 좀 심한 거 아닌가요?



Answer:

전혀요. 사실, 이 비교는 지나치게 관대한 편입니다. 더 균형잡힌 비교를 위해선 락을 전혀 사용하지 않는 단일 CPU 성능과 비교해야 하겠죠.

Quick Quiz 4.19:

하지만 1,000 개의 인스트럭션은 크리티컬 섹션 치고 그렇게 작은 크기는 아니예요. 수십 개의 인스트럭션 정도만을 가지는 훨씬 작은 크리티컬 섹션이 필요하면 어떻게 해야하죠?



Answer:

읽혀지는 데이터가 절대 변하지 않는다면, 거기 접근하는데 어떤 락도 잡을 필요가 없습니다. 만약 데이터가 충분히 가끔만 변경된다면, 실행된 단계를 기록해두

고, 모든 쓰레드를 종료시키고, 데이터를 변경한 후, 기록된 단계부터 쓰레드들을 다시 실행시키면 됩니다.

다른 방법은 쓰레드당 하나씩의 명시적 락을 두고, 자신의 락을 획득함으로써 커다란 리더-라이터 락의 읽기 락 획득을 하고, 모든 쓰레드의 락을 획득함으로써 쓰기 락 획득을 하는 것 [HW92]과 같은 효과를 얻는 것입니다. 이 방법은 리더들을 위해선 상당히 잘 동작합니다만, 라이터들은 쓰레드의 수가 늘어날수록 큰 오버헤드를 갖게 만들 수 있습니다.

그 외의 매우 작은 크리티컬 섹션을 처리하기 위한 방법들은 Section 9.3에서 다루고 있습니다.

Quick Quiz 4.20:

Figure 4.10에서 100M에서의 경우 이외의 값들은 이상적인 선에서 부드럽게 멀어집니다. 반면, 100M에서의 값은 64 CPU에서 갑자기 이상적인 선으로부터 멀어지는군요. 또, 10M 값과 1M 값 사이의 거리는 10M 값과 1M 값 사이의 거리보다 작아요. 왜 100M 값은 이렇게 남들과 다른거죠?

■

Answer:

일반적으로, 최신 하드웨어에선 개선되어 있습니다. 하지만, 128 CPU에서 리더-라이터 락이 이상적인 성능을 달성하기 위해선 100배 이상의 개선이 필요합니다. 게다가, CPU의 갯수가 커질수록, 필요한 성능 향상 정도도 커집니다. 따라서 리더-라이터 락의 성능 문제는 당분간은 존재할 것입니다.

Quick Quiz 4.22:

정말로 이것들이 다 필요한 거 맞나요? ■

Answer:

엄격하게 말하면, 아닙니다. 필요하면 첫번째 분류의 것들을 이용해서 두번째 분류의 것들을 구현할 수가 있습니다. 예를 들어, 누군가는 `__sync_fetch_and_nand()`를 이용해서 아래와 같이 `__sync_nand_and_fetch()`를 구현할 수 있겠죠.

```
tmp = v;
ret = __sync_fetch_and_nand(p, tmp);
ret = ~ret & tmp;
```

비슷하게 `__sync_fetch_and_add()`, `__sync_fetch_and_sub()`, 그리고 `__sync_fetch_and_xor()`를 그들의 나중값 리턴하는 대응 함수들을 이용해 구현하는 것도 가능합니다.

하지만, 이를 대신해주는 기능이 있는 게 프로그래머에게도 컴파일러/라이브러리를 구현하는 사람에게도 편리할 것입니다.

Quick Quiz 4.23:

이 어토믹 오퍼레이션들은 기계의 인스트럭션 셋에서 바로 지원되는 한개짜리 어토믹 인스트럭션으로 변환될텐데, 이것들이 일을 돌아가게 할 수 있는 가장 빠른 방법 아닙까요?

■

Answer:

안타깝지만, 아닙니다. 극명한 반례를 위해 Chapter 5을 보시기 바랍니다.

Quick Quiz 4.24:

리눅스 커널의 `fork()`과 `wait()` 대체물은 어디 갔죠?

■

Quick Quiz 4.21:

Power-5는 나온지 몇년이 넘었고, 최신 하드웨어는 분명 더 빠를 거예요. 그런데 왜 리더-라이터 락의 느린 속도에 걱정해야 하죠?

Answer:

그런건 없습니다. 리눅스 커널 내에서 실행되는 모든 태스크들은 메모리를 공유합니다. 당신이 거대한 메모리 매핑을 손으로 일일히 할 생각이 아니라면 말이죠.

Quick Quiz 4.25:

셸은 기본적으로 `fork()` 가 아니라 `vfork()` 를 사용하지 않나요?

**Answer:**

아마 그럴겁니다만, 확인해보는건 독자의 몫입니다. 하지만 그렇다 해도, 전 우리가 `vfork()` 는 `fork()` 의 변종일 뿐이고, 따라서 `fork()` 를 둘 다를 이야기하는 일반적 용어로 사용해도 된다는데 합의했으면 합니다.

D.5 Counting

Quick Quiz 5.1:

대체 왜 효과적이고 확장성 있는 카운팅이 어려운가요? 무엇보다, 컴퓨터들은 카운팅, 더하기, 빼기, 그 외에도 여러가지를 위한 전용 하드웨어도 가지고 있는데, 그걸 못하나요???

**Answer:**

공유된 카운터에 대한 어토믹 오퍼레이션과 같은 기본적인 카운팅 알고리즘들은 Section 5.1에서 이야기하듯 느리고 확장성이 나쁘거나 정확도가 떨어지기 때문입니다.

Quick Quiz 5.2:

네트워크 패킷 카운팅 문제. 당신이 송수신된 네트워크 패킷의 갯수 (또는 전체 용량)에 대한 통계를 구해야 한다고 생각해 봅시다. 패킷들은 시스템의 어떤 CPU를 통해서든 송신 / 수신될 수 있을 겁니다. 나아가서 이 커다란 기계가 초당 백만개의 패킷을 다룰 수 있고, 그 갯수를 매 5초마다 읽어내야 하는 시스템 모니터링 패키지가 있다고 가정해 봅시다. 당신이라면 이 통계 카운터를 어떻게 구현하시겠어요?

**Answer:**

힌트: 카운터의 업데이트는 엄청 빨라야 합니다만,

카운터는 500만번의 업데이트마다 한번만 일어나기 때문에, 카운터를 읽어내는 행동은 꽤 느려도 될 겁니다. 또한, 일반적으로 읽어지는 값은 완전히 정교하진 않아도 될겁니다—무엇보다, 카운터는 1밀리세컨드당 1000번 업데이트되기 때문에, 우린 “진짜 값”에서 수천정도는 오차값을 가질 수밖에 없을 겁니다. “진짜 값” 이란게 이 문맥에서 뭘 의미하던지요. 하지만, 읽혀지는 값은 어느정도는 절대적인 오차를 유지해야 할겁니다. 예를 들어, 카운트가 수백만 정도일 때 1% 오차는 문제없지만, 조단위가 된다면 문제가 있겠죠. Section 5.2를 참고하세요.

Quick Quiz 5.3:

대략적 구조체 할당 한계 문제. 할당된 구조체의 갯수가 어떤 한계(한 10,000 정도)를 넘어가면 추가 할당을 막기 위해 할당된 구조체의 갯수를 유지해야 한다고 생각해 봅시다. 또, 이 구조체들은 할당되고 나서 곧바로 해제되고, 한계치를 넘기는 일은 매우 드물고, “대략적인” 한계치 설정이 가능하다고 생각해 봅시다.

**Answer:**

힌트: 카운터의 업데이트 동작은 여기서도 매우 빨라야 합니다만, 카운터는 카운터가 증가될 때마다 읽혀야 합니다. 하지만, 읽혀지는 값은 그 값이 한계치 아래인지 위인지를 대략적으로는 구분해 내야 한다는 점을 제외하고는 정교하지 않아도 됩니다.

Quick Quiz 5.4:

정교한 구조체 할당 한계 문제. 할당된 구조체의 갯수가 어떤 정확한 한계(여기서도, 한 10,000 정도)를 넘어가면 추가 할당을 막기 위해 할당된 구조체의 갯수를 유지해야 한다고 생각해 봅시다. 이 구조체들은 할당되고 얼마 안되 해제되고, 그 한계는 드물게 초과되며, 거의 항상 최소 한개의 구조체는 사용중이 됩니다. 또한 예를 들어, 하나의 구조체도 사용되지 않고 있다면 해제 할 수 있는 어떤 메모리를 위해 카운터가 0이 되는 시점을 정확히 알 필요가 있습니다.

**Answer:**

힌트: 카운터의 업데이트 동작은 역시 엄청 빨라야 합니다만, 카운터의 값이 증가될 때마다 그 값 역시 읽혀야 합니다. 하지만, 그 값은 한계치 경계를 넘어서는지와 0인지를 분명하게 체크해야 한다는 점을 제외하고는 정교할 필요가 없습니다. Section ?? 를

참고하세요.

Quick Quiz 5.5:

제거될 수 있는 I/O 디바이스 접속 카운트 문제. 매우 빈번하게 사용되는 제거 가능한 대용량 디바이스에 대해 사용자에게 해당 디바이스를 제거해도 안전한지 알려주기 위해 그 참조 횟수를 관리해야 한다고 가정해 봅시다. 이 디바이스는 사용자가 디바이스를 제거하고 싶을 때 그 의사를 알려주며, 시스템은 사용자에게 언제 디바이스를 제거해도 안전한지 알려주는 일반적 디바이스 제거 절차를 따릅니다. ■

Answer:

힌트: 여기서도 카운터의 업데이트 동작은 I/O 오퍼레이션이 느려지지 않게 매우 빠르고 확장성 있어야 합니다. 하지만 카운터는 사용자가 디바이스를 제거하고자 할 때에만 읽혀지기 때문에, 카운터의 읽기 동작은 매우 느려도 됩니다. 또한, 사용자가 디바이스를 제거하고자 하는 의사를 밝히지 않았다면 그 카운터는 읽혀질 필요가 아예 없습니다. 또한, 그 값은 디바이스가 제거를 위한 절차 중일 때로 한정해서 0인지 0이 아닌지만 분명히 구분할 수 있어야 한다는 점을 제외하고는 정교할 필요가 없습니다. 하지만, 한번 0이라는 값을 읽었다면, 차후에 다른 쓰레드가 제거중인 해당 디바이스에 접근을 하거나 하는 일을 막기 위해 해당 값을 0으로 유지해야 합니다. Section 5.5를 참고하세요.

Quick Quiz 5.6:

하지만 ++ 연산자는 x86 의 add-to-memory 명령어를 만들지 않나요? 그리고 CPU 캐시는 그걸 어토믹하게 수행하지 않나요? ■

Answer:

++ 연산자는 어토믹할 수도 있지만, 그래야만 한다는 규칙은 없습니다. 그리고 실제로, gcc 는 값을 레지스터에 읽어오고, 레지스터의 값을 증가시킨 후, 메모리에 그 값을 저장하는, 어토믹하지 않은 방법을 종종 택합니다.

Quick Quiz 5.7:

실패 횟수의 8-figure 정확도는 당신이 진짜로 이 테스트를 한 것을 보여주는군요. 왜 이런 사소한 프로그램을, 특히나 버그가 이렇게 쉽게 직관적으로 보이는데도 굳이 테스트 해야 하나요? ■

Answer:

사소한 병렬 프로그램이 아주 조금만 존재하지는 않고, 순차적 프로그램에도 마찬가지라고 저는 생각합니다. 프로그램이 얼마나 작거나 간단한지와는 상관 없이, 테스트 해보지 않았다면, 그건 동작하지 않는 것입니다. 그리고 설령 테스트 해봤다 해도, 머피의 법칙에 의하면 여전히 숨어있는 버그가 몇개는 있을 수 있을 수 있습니다.

또한, 정확성의 증명은 분명 그 의미를 갖지만, 여기 사용된 counttorture.h 테스트를 포함해 테스트를 대체하는 일은 결코 없을 겁니다. 무엇보다, 증명은 그것이 바닥에 깔고 있는 가정에 국한됩니다. 게다가, 증명은 프로그램이 그렇듯 버그를 가지고 있기 쉽습니다!

Quick Quiz 5.8:

왜 x 축의 점선은 $x = 1$ 에서 대각선의 선과 만나지 않죠? ■

Answer:

어토믹 오퍼레이션의 오버헤드 때문입니다. x 축의 점선은 싱글 어토믹하지 않은 증가 연산의 오버헤드를 나타냅니다. 이상적인 알고리즘은 선형적으로 확장될 뿐만 아니라, 싱글 쓰레드 코드에 비해서도 성능 하락이 없어야 할 것입니다.

이런 수준의 이상론은 좀 지나쳐 보일 수 있습니다. 다만 리눅스 토발즈에게 충분하다면, 당신에게도 충분 할 겁니다.

Quick Quiz 5.9:

하지만 어토믹 증가 연산은 여전히 꽤 빠릅니다. 그리고 빽빽한 루프에서 하나의 변수를 증가시키는 건 제겐 꽤 비현실적인 것 같아 보이구요. 무엇보다, 프로그램의 실행은 실제로 일을 하는데 쓰여야지, 자기가 한 일을 세는데 쓰여야 하는게 아니라구요! 왜 제가 이걸 빠르게 하는걸 고민해야 하나요? ■

Answer:

많은 경우에 어토믹 증가 연산은 분명히 당신에겐 충분히 빠를 겁니다. 그런 경우에 당신은 당연히 어토믹 증가 연산을 사용해야죠. 그렇지만, 더 나은 카운팅 알고리즘이 필요한 실제 상황도 상당히 많이 존재합니다. 그런 상황의 예는 고도로 최적화된 네트워킹 스택에서의 패킷과 용량 카운팅으로, 이런 예에서는, 특히나 커다란 멀티프로세서에서는 대부분의 실행시간을 이런류의 카운팅 작업에 보내게 됩니다.

게다가, 이 챕터의 시작에서 이야기했듯, 카운팅은 공유 메모리 병렬 프로그램에서 마주칠 수 있는 문제들을

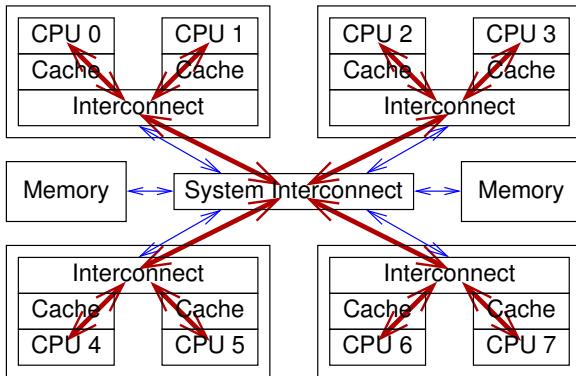


Figure D.1: Data Flow For Global Combining-Tree Atomic Increment

보여줍니다.

Quick Quiz 5.10:

그런데 왜 CPU 설계자들은 단순히 데이터에의 증가 연산을 추가해서 담고 있는 캐시 라인의 순회가 증가하는 걸 막지 않는거죠? ■

Answer:

어떤 경우에는 그런 방법도 가능하겠죠. 하지만, 그러면 좀 복잡합니다:

1. 만약 그 변수의 값이 필요하다면, 그 쓰레드는 해당 오퍼레이션이 그 데이터로 향할 때까지 기다리고, 그리고나선 돌아오기까지 기다려야 합니다.
2. 그 어토믹 증가 오퍼레이션이 이전의, 그리고 이후의 오퍼레이션들과 순서를 맞춰야 한다면, 해당 쓰레드는 오퍼레이션이 데이터까지 가고, 돌아올 준비가 완료될 때까지 기다려야 합니다.
3. 오퍼레이션들을 CPU들 사이에서 보내게 되면 시스템 접속부를 지나야 하고, 이는 더 많은 디아이 간과 전력을 소모하게 될겁니다.

하지만 앞의 두가지 조건이 없다면? 그럼 당신은 Section 5.2에서 논의되는, 실제 상용화된 하드웨어에서 이상적 상황에 근접하는 성능을 보이는 알고리즘을 잘 고려해 봄야 할 겁니다.

앞의 두가지 조건이 하나라도 걸려 있다면, 개선된 하드웨어에 약간의 희망이 있습니다. 캠바이닝 트리(combining tree)를 하드웨어에서 구현해서, 여러 CPU에서의 증가 요청이 하드웨어에 의해 결합되어 하나의 더하기 연산으로 변환되는 방법을 생각해 볼 수 있을 것입니다. 해당 하드웨어는 또한 요청들에 순서를 잡아줄

수도 있으므로, 각 CPU에게 각자의 어토믹 증가에 의한 값의 반환도 가능할 겁니다. Figure D.1에서 보여지듯, 이는 인스트럭션의 대기시간을 $O(\log N)$ 로 만들어 줍니다. 여기서 N 은 CPU의 갯수입니다. 그리고 이런 하드웨어 최적화를 포함하는 CPU는 2011년부터 나오기 시작했습니다.

Figure 5.4에 보여진 현재 하드웨어의 $O(N)$ 성능에 비하면 엄청난 향상이고, 하드웨어 대기시간은 3차원 제조 공정이 현실화되면 더욱 낮아질 수 있을 것입니다. 물론, 일부 중요한 특수 케이스에서는 소프트웨어가 훨씬 나은 일을 할 수 있을 것입니다.

Quick Quiz 5.11:

하지만 C의 “정수들”은 크기와 관련한 복잡한 문제들이 있지 않나요? ■

Answer:

아닙니다. 모듈로 더하기도 역시 상호성과 결합성을 가지니까요. 적어도 부호 없는 정수형을 사용한다면 말입니다. 오버플로우가 났을 때 넘쳐난 값을 감추는 것 외에 별다른 일을 하는 기계들은 요즘 거의 없지만, C 표준에 의하면 부호를 갖는 정수형의 오버플로우는 예상외 동작을 유발할 수 있으므로, 요즘 기계들은 대부분 안전하다는 사실은 신경쓰지 마십시오. 불행히도, 컴파일러들은 종종 부호 있는 정수형들은 오버플로우 나지 않을 것이라는 가정 하에 최적화를 하기 때문에, 당신의 코드가 부호 있는 정수형을 오버플로우 나게 한다면, 2의 보수를 사용하는 하드웨어를 사용하고 있다 해도 문제에 직면할 수 있습니다.

그렇다면 해도, 32-비트 쓰레드별 카운터에서 (대략) 64-비트 합을 모으는데에도 추가적인 복잡한 일이 숨어 있습니다. 이걸 처리하는 것은 독자 여러분들에게 연습 문제로 남겨두겠습니다만, 이 챕터의 뒤에 소개될 기법들이 큰 도움이 될 수도 있습니다.

Quick Quiz 5.12:

배열이요??? 하지만 그럼 쓰레드의 갯수가 제한되지 않나요? ■

Answer:

그럴 수 있고, 이 간단한 구현에서는 그렇습니다. 하지만 임의의 갯수의 쓰레드를 지원하는 구현을 만드는 건 그렇게 어렵지 않은데요, 예를 들면, Section 5.2.4 `gcc __thread`를 사용하는 거죠.

Quick Quiz 5.13:

근데, 그 외에 gcc 가 어떤 짓을 할 수 있죠??? ■

Answer:

C 표준대로라면, 다른 쓰레드에 의해 동시에 수정되고 있을 수 있는 변수를 읽어들이는 행동의 효과에 대해선 정의되어 있지 않습니다. C 는 어토믹하게 long 타입 변수를 읽어들일 수가 없는 (예를 들면) 8-비트 아키텍처를 지원해야만 했기에 C 표준은 다른 선택의 여지가 없었습니다. 다음 버전의 C 표준에서는 이 현실과의 격차를 해결해 보려 합니다만, 그 전까지는 gcc 개발자들의 친절함에 의존해야 합니다.

대신, 하드웨어가 한번의 메모리 참조 명령으로 필요한 값을 읽을 수 있는 경우라면, ACCESS_ONCE() [Cor12] 같은 volatile 접근법을 사용해서 컴파일러에 제약을 가할 수 있습니다.

Quick Quiz 5.14:

Figure 5.6 의 쓰레드별 counter 변수는 어떻게 초기화 되나요? ■

Answer:

C 표준은 전역 변수의 초기값은 명시적으로 초기화되지 않는 이상 0이라고 명시합니다. 그리고, 사용자가 통계적 카운터들의 연속되는 값 사이의 차이에 관심있는게 아니라면, 초기값은 의미가 없습니다.

Quick Quiz 5.15:

Figure 5.6 의 코드가 어떻게 복수의 카운터를 가능하게 할 수 있죠? ■

Answer:

실제로, 이 예제는 한개 이상의 카운터는 지원하지 않습니다. 이 예제를 수정해서 복수의 카운터를 제공하게 하는건 독자 여러분에게 과제로 남겨 두겠습니다.

Quick Quiz 5.16:

읽기 오퍼레이션은 쓰레드별 값을 모두 더하는 시간을 가져야 할 것이고, 그동안도 카운터는 값이 변할 수 있어요. 그럼 Figure 5.6 의 read_count() 는 정확하지 않다는 의미입니다. 이 카운터는 단위시간당 r 만큼 카운터 값을 증가하고 read_count() 는 Δ 단위시간을 소모한다고 해봅시다. 리턴되는 값의 예상 오류값은 얼마입니까? ■

Answer:

최악의 경우에 대한 분석부터 해보고, 좀 나은 경우들을

보죠.

최악의 경우는, 읽기 오퍼레이션이 실제 동작은 금방 끝냈지만 리턴하기 전에 Δ 단위시간동안 대기를 하는 경우로, 이 경우의 오류값은 간단히 $r\Delta$ 입니다.

이 최악의 경우 동작은 별로 현실성이 없으니 각 N 카운터들에 대한 각 읽기 동작이 시간 간격 Δ 안에 동일한 간격으로 일어나는 경우를 생각해보죠. N 회의 읽기 사이에 $\frac{\Delta}{N+1}$ 길이의 $N+1$ 개 간격이 존재할 겁니다. 마지막 쓰레드의 카운터에서의 읽기 이후로의 지연으로 발생하는 에러치는 $\frac{r\Delta}{N(N+1)}$ 이므로, 두번째에서 마지막 쓰레드 사이의 카운터는 $\frac{2r\Delta}{N(N+1)}$, 세번째에서 마지막 사이는 $\frac{3r\Delta}{N(N+1)}$, 그리고 그렇게 계속 진행됩니다. 전체 오류값은 각 쓰레드의 카운터에서의 읽기로 발생한 에러의 총 합으로 주어지는데, 다음과 같습니다:

$$\frac{r\Delta}{N(N+1)} \sum_{i=1}^N i \quad (D.1)$$

합 부분은 다음과 같이 표현 가능하구요:

$$\frac{r\Delta}{N(N+1)} \frac{N(N+1)}{2} \quad (D.2)$$

불필요한 부분을 제거하면 다음과 같이 직관적인 결과가 나옵니다:

$$\frac{r\Delta}{2} \quad (D.3)$$

읽기 오퍼레이션 호출자가 해당 오퍼레이션으로 리턴받은 수를 가지고 어떤 일을 하는 코드를 수행하는 중에도 오류는 쌓여감을 기억해둘 필요가 있습니다. 예를 들어, 읽기 오퍼레이션 호출자가 리턴받은 값을 가지고 어떤 계산을 하는데 t 시간을 사용한다면, 최악의 경우 오류치는 $r(\Delta+t)$ 로 늘어날 겁니다.

예상 오류치도 비슷하게 늘어나겠죠:

$$r \left(\frac{\Delta}{2} + t \right) \quad (D.4)$$

물론, 읽기 중에 카운터가 계속 증가하는건 용납될 수 없는 경우도 있겠습니다. Section 5.5 에서는 이런 상황을 해결하는 방법을 알아봅니다.

이렇게, 우리는 감소는 없고 증가만 이루어지는 카운터를 알아보았습니다. 만약 카운터 값이 단위시간당 r 만큼 감소로든 증가로든 바뀐다면, 오류값은 줄어들 거라고 예상할 수 있을 겁니다. 하지만, 카운터가 어느 쪽으로든 움직일 수 있을지 몰라도 최악의 경우는 해당 읽기 오퍼레이션이 금방 끝났지만 Δ 단위 시간동안 기다려야 하는데, 그 시간 동안 카운터의 값은 같은 방향으로만 이동하는 경우이므로 여전히 $r\Delta$ 오류값을 내므로 차이가 없습니다.

평균 에러치를 계산하는데는 값의 증감 패턴에 대한 다양한 가정을 바탕으로 하는 여러가지 방법이 있습니다. 일단은 간단하게, 1의 오퍼레이션 중 f 만큼이 감소 오퍼레이션이고, 관심있는 오류값은 카운터의 장시간 추세선에서의 굴곡이라고 가정해 봅시다. 이 가정 하에서는 f 가 0.5 이하라면, 각 감소는 증가에 의해 무효화 되고, 따라서 $2f$ 오퍼레이션은 서로를 무효화 시키고, $1-2f$ 의 오퍼레이션들은 무효화 되지 않은 증가가 됩니다. 반면, f 가 0.5 보다 크다면, $1-f$ 의 감소는 증가에 의해 무효화 되고, 카운터는 음의 방향으로 $-1+2(1-f)$ 만큼 이동하는데, 이는 $1-2f$ 로 정리되고, 따라서 카운터는 평균적으로 오퍼레이션당 $1-2f$ 만큼씩 어느 경우든 이동하게 됩니다. 따라서, 긴 시점에서의 카운터의 변화는 $(1-2f)r\Delta$ 로 주어집니다. 이걸 Equation D.3에 대입하면:

$$\frac{(1-2f)r\Delta}{2} \quad (D.5)$$

그렇지만, 대부분의 통계적 카운터 사용에서 `read_count()`에 의해 리턴되는 값의 오류치는 별 의미가 없습니다. `read_count()`가 수행하는데 필요한 시간은 일반적으로 `read_count()` 호출 사이의 시간에 비하면 극단적으로 작기 때문입니다.

Quick Quiz 5.17:

Figure 5.8의 `inc_count()`는 왜 어토믹 명령을 사용하지 않죠? 쓰레드별 카운터를 여러 쓰레드에서 접근하고 있잖아요! ■

Answer:

두 쓰레드 중 하나는 읽기만 하고 있고, 변수는 정렬되어 있으며 기계가 지원하는 워드 크기이기에, 어토믹하지 않은 명령들만으로도 충분합니다. 다만, `ACCESS_ONCE()` 매크로는 카운터 업데이트가 `eventual()`에게 보여지는 것을 막을 수도 있는 [Cor12] 컴파일러 최적화를 막기 위해 사용되었습니다.

이 알고리즘의 예전 버전은 어토믹 명령을 사용했습니다만, 감사하게도 Ersoy Bayramoglu 가 그것들이 필요 없다는 것을 지적해 줬습니다. 그렇다면 하나, 쓰레드별 `counter` 변수가 `global_counter` 보다 작았다면 어토믹 명령이 필요했을 겁니다. 하지만, 32-bit 시스템에서 쓰레드별 `counter` 변수는 정확하게 합을 구하기 위해 32 비트로 제한되어야 하고 오버플로를 막기 위해 `global_count` 변수는 64-bit 이 되어야 할 겁니다. 이 경우엔, 오버플로를 막기 위해 쓰레드별 `counter` 변수를 주기적으로 0으로 초기화 시켜줘야 할 겁니다. 0으로의 주기적 초기화는 너무 오래 지연되면 쓰레드별 변수의 오버플로가 가능하단 것을 반드시

기억해 둬야만 합니다. 따라서 이 방법은 프로그램이 돌아가는 시스템이 리얼-타임 속성을 가지고 있어야 하며, 매우 조심스럽게 사용되어야 함을 의미합니다.

대조적으로, 모든 변수가 같은 크기이면 어떤 변수에 오버플로가 나더라도 최종적 합은 워드 크기로 절삭될 테니 별 문제 없습니다.

Quick Quiz 5.18:

Figure 5.8의 단일 글로벌 쓰레드인 `eventual()` 함수는 글로벌 락처럼 큰 병목이 되거나 하진 않나요? ■

Answer:

이 경우엔, 아닙니다. 그 대신 쓰레드의 갯수가 늘어나면 `read_count()`에 리턴되는 카운터 값이 더 부정확해질 겁니다.

Quick Quiz 5.19:

Figure 5.8의 `read_count()`에서 리턴하는 추정값은 쓰레드의 갯수가 늘어날수록 부정확해져 가지 않을까요? ■

Answer:

맞습니다. 이게 문제가 된다면, 여러 `eventual()` 쓰레드를 만들고, 각 쓰레드가 일을 나눠서 해야 하는게 한가지 해결책이 될 수 있습니다. 더 극단적인 경우에는, `tree` 같은 `eventual()` 쓰레드 계층 관리가 필요할 수도 있습니다.

Quick Quiz 5.20:

Figure 5.8의 최종적 일관성 알고리즘은 읽기에도 쓰기에도 매우 적은 오버헤드와 극단적인 확장성을 보이는데, 과연 누가 Section 5.2.2 같이 읽기 쪽이 비싼 구현을 사용하겠습니까? ■

Answer:

`eventual()` 쓰레드를 돌리는 것은 CPU 시간을 소모합니다. 이 최종적으로 일관적인 카운터가 추가되어가면 언젠가는 `eventual()` 쓰레드들이 모든 CPU를 차지할 겁니다. 따라서 이 구현은 확장성이 쓰레드나 CPU의 갯수가 아니라 최종적으로 일관적인 카운터의 갯수에 제한되는, 또 다른 종류의 확장성 한계 문제를 갖습니다.

Quick Quiz 5.21:

다른 쓰레드의 카운터를 찾는데 왜 별개의 배열이 필요하죠? 왜 gcc 는 리눅스 커널의 `per_cpu()` 가 쓰레드들이 다른 쓰레드의 쓰레드별 변수를 쉽게 접근할 수 있도록 하는 것처럼 `per_thread()` 같은 인터페이스를 제공하지 않나요? ■

Answer:

정말 왜일까요?

gcc 에는 리눅스 커널은 무시할 수 있는 몇가지 문제들이 존재합니다. 유저 레벨 쓰레드가 종료될 때, 그 쓰레드의 쓰레드별 변수는 모두 사라지는데 이로 인해, 적어도 유저 레벨 RCU(Section 9.3) 가 충분히 개선되기 전까지는, 쓰레드별 변수에의 액세스 문제는 복잡해집니다. 반면, 리눅스 커널에서는 한 CPU 가 오프라인이 되더라도 그 CPU 의 CPU 별 변수는 여전히 매핑 되어 있고 액세스 가능한 상태로 남습니다.

비슷하게, 새 유저 레벨 쓰레드가 생성되면, 그 쓰레드의 쓰레드별 변수는 갑자기 생겨나야 합니다. 반면, 리눅스 커널에서는 특정 CPU 가 아직 존재하지 않거나 나중에도 존재하게 되는 일이 없더라도 부팅 과정에서 모든 CPU 별 변수의 매핑과 초기화를 합니다.

리눅스 커널이 가지고 있는 중요 제약은 컴파일 시간의 길이가 CPU 갯수인 `CONFIG_NR_CPUS` 에 바운드되며, 부팅 타임의 길이 역시 `nr_cpu_ids` 에 바운드 된다는 점입니다. 반면, 유저 스페이스에서는 쓰레드의 갯수에 의한, 하드코딩된 제약이 존재하지 않습니다.

물론, 두 환경 모두 다이나믹하게 로드되는 코드(유저 스페이스라면 동적 라이브러리, 리눅스 커널에서는 커널 모듈) 가 쓰레드별 변수의 복잡도를 증가시키므로 해당 경우도 처리해야 합니다.

이런 복잡성이 유저 스페이스 환경에서 다른 쓰레드의 쓰레드별 변수에의 접근을 제공하기 어렵게 합니다. 하지만 분명한건, 그런 접근은 상당히 유용하고, 따라서 언젠가는 그런 인터페이스가 생겨나면 좋겠죠.

Quick Quiz 5.22:

Figure 5.9 의 라인 19 에서의 NULL 체크는 브랜치 예측 실패를 가져오지 않나요? 항상 0인 변수 집합을 두고 더이상 사용되지 않는 카운터로의 포인터를 NULL 로 만드는 대신 그 변수로 향하게 하는게 어떤가요? ■

Answer:

말 되는 이야기입니다. 다만 성능이 어떻게 달라지는지는 독자의 몫으로 남겨두겠습니다. 다만, 이 코드가 빠르게 하고자 하는 곳은 `read_count()` 가 아니라 `inc_count()` 임을 항상 기억해 두시기 바랍니다.

Quick Quiz 5.23:

도대체 왜 Figure 5.9 의 `read_count()` 함수의 합을 계산하는 곳에서 무거운 `lock` 을 사용하는거죠? ■

Answer:

쓰레드가 종료될 때, 그 쓰레드의 쓰레드별 변수는 사라짐을 기억하세요. 따라서, 한 쓰레드의 쓰레드별 변수를 그 쓰레드가 종료된 후에 접근하려 하면 세그먼테이션 폴트가 날 겁니다. 해당 락은 합 계산과 쓰레드 종료 작업을 중재해서 그런 일이 발생하지 않게 해줍니다.

물론, 대신 reader-writer 락을 사용해 `read-acquire` 할 수도 있겠습니다만 Chapter 9 에서 이 중재작업을 그보다도 가볍게 해줄 수 있는 메커니즘을 소개할 겁니다.

다른 방법으로는 쓰레드별 변수 대신 배열을 사용하는 방법이 있겠는데요, Alexey Roytman 이 이야기한 대로 NULL 테스트를 없앨 수 있겠죠. 하지만, 배열에의 접근은 대부분의 경우 쓰레드별 변수보다 느리고, 쓰레드의 갯수의 최대값에 대한 제한을 가져올 겁니다. 또한, 테스트도 락도 우리가 빠르게 하고자 하는 부분인 `inc_count()` 에서는 사용되지 않고 있음을 기억하세요.

Quick Quiz 5.24:

대체 왜 Figure 5.9 의 `count_register_thread()` 함수에서 락을 잡아야 하는거죠? 여기서 사용하는건 다른 쓰레드가 건들지 않는, 제대로 정렬된 기계의 워드 스토어 사이즈 데이터이니 어토믹할 거잖아요, 아닌가요? ■

Answer:

이 락은 실제로 없앨 수도 있습니다만, 특히 이 함수가 쓰레드 시작 시점에서만 실행되고, 따라서 성능에 중요한 영역이 아닌만큼 좀 더 안전에 치중했습니다. 만약 우리가 이 코드를 수천개의 CPU 를 가진 기계에서 테스트 한다면야 이 락을 없애야 할 수도 있습니다만 “겨우” 수백개 CPU 의 기계라면 굳이 그렇게 할 필요 없겠죠.

Quick Quiz 5.25:

좋아요, 하지만 리눅스 커널은 CPU 별 카운터의 값을 합칠 때 락을 잡지 않아요. 유저 스페이스 코드에선 왜 이게 필요한거죠??? ■

Answer:

기억해보세요, 리눅스 커널의 CPU 별 변수들은 항상, 심지어 해당 CPU 가 꺼져 있다 해도 접근 가능해요 —

```

1 long __thread counter = 0;
2 long *counterp[NR_THREADS] = { NULL };
3 int finalthreadcount = 0;
4 DEFINE_SPINLOCK(final_mutex);
5
6 void inc_count(void)
7 {
8     counter++;
9 }
10
11 long read_count(void)
12 {
13     int t;
14     long sum = 0;
15
16     for_each_thread(t)
17     {
18         if (counterp[t] != NULL)
19             sum += *counterp[t];
20     }
21
22 void count_init(void)
23 {
24 }
25
26 void count_register_thread(void)
27 {
28     counterp[smp_thread_id()] = &counter;
29 }
30
31 void count_unregister_thread(int nthreadsexpected)
32 {
33     spin_lock(&final_mutex);
34     finalthreadcount++;
35     spin_unlock(&final_mutex);
36     while (finalthreadcount < nthreadsexpected)
37         poll(NULL, 0, 1);
38 }

```

Figure D.2: Per-Thread Statistical Counters With Lockless Summation

심지어 해당 CPU 가 한번도 켜졌던 적 없고 앞으로도 켜질 일이 없다 해도요.

다만 문제를 회피하는 한가지 방법은 Figure D.2 (count_tstat.c)에 나온 것처럼 각 쓰레드가 모든 쓰레드가 끝날 때까지 종료하지 않게 하는 겁니다. 이 코드의 분석은 독자의 뜻으로 남겨두겠습니다만 이건 counttorture.h의 카운터 성능 평가 방법과는 맞지 않음을 알아 두세요.(왜일까요?) Chapter 9에서는 이 상황을 훨씬 우아한 방법으로 해결하는 동기화 메커니즘을 소개합니다.

Quick Quiz 5.26:

패킷의 사이즈가 다양하다면 패킷의 갯수를 세는 것과 패킷의 전체 바이트 수를 세는 것에 어떤 기본적 차이가 있나요? ■

Answer:

패킷의 갯수를 세 때, 카운터는 한번에 1씩 증가합니다.

반면, 바이트를 세 때에는, 카운터는 큰 수만큼 증가할 수도 있습니다.

왜 이걸 신경써야 할까요? 1씩 증가하는 경우에 리턴되는 값은 비록 그 값이 이루어진 시점이 언제인지는 알 수 없더라도 분명히 어떤 시점의 값인 것은 분명하기 때문입니다. 반면, 바이트를 세는 경우에는 두개의 서로 다른 쓰레드는 오퍼레이션들의 순서에 따라 비일관적인 값을 리턴할 수도 있습니다.

쓰레드 0이 자신의 카운터에 3을, 쓰레드 1이 자신의 카운터에 5를 더하고, 쓰레드 2와 쓰레드 3이 카운터를 더하는 경우를 생각해 봅시다. 만약 시스템이 “취약한 순서”를 가지거나 컴파일러가 강력한 최적화를 사용한다면, 쓰레드 2는 합이 3이고 쓰레드 3은 합이 5라고 볼 수 있습니다. 일관적인 시스템이라면 값이 변하는 순서는 0,3,8 또는 0,5,8 만 가능하므로 이 예에서 쓰레드들이 발견한 결과는 일관적이지 못합니다.

이걸 깜박했다 해도, 당신만 그런게 아닙니다. Michael Scott은 Paul E. McKenney의 박사 학위 심사 과정에서 이 질문을 했습니다.

Quick Quiz 5.27:

리더는 쓰레드들의 카운터를 모두 더해야 하므로, 쓰레드의 갯수가 늘어나면 더 많은 시간을 쓰게 될 겁니다. 리더에게도 쓸만한 성능과 확장성을 주면서 쓰기 작업도 여전히 빠르고 확장성 있게 하는 방법은 없을까요? ■

Answer:

글로벌한 추정값을 두는 게 한 방법이 될 겁니다. 리더들은 각자의 쓰레드별 변수를 증가시키되 어떤 미리 지정된 한계에 도달했을 때에는 그 값을 글로벌 변수에 어토믹하게 더하고, 쓰레드별 변수를 0으로 초기화 시키는 겁니다. 이 방법은 병균적 쓰기 오버헤드와 읽어지는 값의 정확성 사이의 협상을 가능하게 할겁니다.

독자분들은 다른 방법들, 예를 들어 컴바이닝 트리와 같은 것들을 생각해보고 시도해보는 것도 좋을 겁니다.

Quick Quiz 5.28:

어째서 Figure 5.12 는 Section 5.2에서 나왔던 inc_count() 와 dec_count() 인터페이스 대신에 add_count() 와 sub_count() 를 제공하나요? ■

Answer:

서로 다른 크기의 구조체들이 할당 요청되기 때문입니다. 물론, 특정 크기의 구조체에만 사용되는 한계 카운터는 여전히 inc_count() 와 dec_count()

를 사용할 수 있을 겁니다.

Quick Quiz 5.29:

Figure 5.12 라인 3 의 저 이상한 조건문은 뭘까요? 왜 다음과 같이 더 직관적인 형태의 빠른 수행경로를 사용하지 않는거죠?

```
3 if (counter + delta <= countermax) {
4     counter += delta;
5     return 1;
6 }
```

Answer:

두단어로 설명하죠. “인터저 오버플로우.”

앞의 코드를 10의 값을 갖는 counter 와 ULONG_MAX 값을 갖는 delta에 대해 수행해 보세요. 그리고 나서 Figure 5.12 의 코드로 한번 더 해보세요.

이 예제의 뒷부분은 인터저 오버플로우에 대한 깊은 이해를 필요로 하므로, 인터저 오버플로우 문제를 한번도 겪어본 적 없다면, 몇몇 예제를 가지고 이해해 보려 노력해보세요. 일부 경우에 있어서는 인터저 오버플로우가 병렬 알고리즘보다도 제대로 처리하기가 어렵습니다!

Quick Quiz 5.30:

Figure 5.12 에서 왜 globalize_count () 는 나중에 balance_count () 가 쓰레드별 변수를 다시 채우도록 쓰레드별 변수를 0으로 바꾸나요? 왜 그냥 쓰레드별 변수를 0이 아닌 채로 놔두질 않는거죠? ■

Answer:

사실 이전의 버전의 이 코드에서는 그렇게 했습니다. 하지만 더하기와 빼기는 매우 비용이 짜동작이고, 모든 특수 케이스를 처리하는건 상당히 복잡합니다. 다시 말하지만, 직접 한번 해보세요, 다만 인터저 오버플로우를 조심하구요!

Quick Quiz 5.31:

Figure 5.12 에서 globalreserve 는 add_count () 에서 값이 구해지는데, 왜 sub_count () 에서 값을 구하지 않나요? ■

Answer:

globalreserve 변수는 모든 쓰레드의 countermax 변수의 합을 따라갑니다. 이 쓰레드의 counter 변수들의 합은 0 부터 globalreserve

사이 어딘가일 것입니다. 따라서 우리는 모든 쓰레드의 counter 변수가 add_count () 에서 꽉 차고 sub_count () 에서 비어버린다 가정하는, 보수적 방법을 취합니다.

하지만 나중에 다시 한번 이야기할테니, 이 질문을 기억해 두세요. ■

Quick Quiz 5.32:

한 쓰레드가 Figure 5.12 의 add_count () 를 호출하고, 다른 쓰레드가 sub_count () 를 호출한다고 해봅시다. sub_count () 는 카운터의 값이 0이 아님에도 실패하지 않겠습니까? ■

Answer:

실제로 그럴 것입니다! 많은 경우에, 이것은 Section 5.3.3 에서 이야기되는 것처럼 문제가 될 것이고, 그런 경우에는 Section 5.4 에서 다루는 알고리즘을 사용하는게 좋을 겁니다.

Quick Quiz 5.33:

Figure 5.12 에서는 왜 add_count () 와 sub_count () 를 모두 가지고 있는 거죠? 그냥 add_count () 에 음수를 넘기면 되지 않나요? ■

Answer:

add_count () unsigned long 타입을 인자로 받기 때문에, 음수를 넘기는건 조금 어려울 겁니다. 그리고 설령 반물질 메모리를 가지고 있다 해도, 사용중인 구조체의 수를 세는데에 음수를 넘긴다는건 좀 말이 이상하죠!

Quick Quiz 5.34:

Figure 5.13 의 라인 15 에서는 왜 counter 를 countermax / 2 로 만들죠? 그냥 countermax 값을 가져오는게 더 간단하지 않나요? ■

Answer:

첫째로, countermax 카운트는 이미 예약되어 있긴 합니다만 (라인 14 를 참고하세요), 이 코드는 해당 순간에 해당 쓰레드에 의해서 실제로는 그 반만 사용하고 있다고 알리는 것입니다. 이렇게 함으로써 해당 쓰레드는 globalcount 까지 찾아가지 않고도 최소 countermax / 2 만큼까지는 증가 또는 감소 작업을 할 수 있게 합니다.

globalcount 의 수는 라인 18 에서의 조정 덕에 여전히 정확함을 참고하세요.

Quick Quiz 5.35:

Figure 5.14 에서 보면, 가운데와 오른쪽 구성을 잇는 뒤쪽의 점선을 보면 알 수 있듯이, 한계까지 남은 카운트의 4분의 1이 쓰레드 0에게 주어졌음에도, 8분의 1 만이 사용되었습니다. 왜 그런건가요? ■

Answer:

쓰레드 0의 counter 가 countermax 의 절반으로 책정되었기 때문입니다. 따라서, 쓰레드 0에 할당된 4분의 1 중 절반(8분의 1)은 globalcount 에서 오고, 나머지 절반(역시 8분의 1)은 남은 카운트에서 오도록 남겨두는 것이죠.

이런 방법을 취하는데에는 두가지 목적이 있습니다: (1) 쓰레드 0가 증가만이 아니라 감소에서도 빠른 수행 경로를 사용할 수 있도록 하는것, 그리고 (2) 모든 쓰레드가 단조적으로 한계점을 향해 증가만 하고 있다면 비정확성을 줄이기 위해서입니다. 마지막 이야기를 이해하려면, 알고리즘에 한발 더 다가가 자세히 살펴보세요.

Quick Quiz 5.36:

쓰레드의 counter 와 countermax 변수를 한번에 어토믹하게 수정해야 하는 이유가 뭐죠? 각 변수를 개별적으로 어토믹하게 수정해도 충분하지 않아요? ■

Answer:

그렇게도 할 수 있겠지만, 엄청난 주의가 필요합니다. counter 를 countermax 를 먼저 0으로 하지 않고 없애는 것은 counter 를 0이 된 직후 증가시키는 같은 쓰레드가 카운터를 0으로 만든 효과를 없애버리게 만듭니다.

반대로, countermax 를 0으로 만들고 counter 를 없애는 것 역시 0이 아닌 counter 를 만들 수 있습니다. 이걸 자세히 보기 위해 다음의 이벤트 시퀀스를 봅시다:

1. Thread A 가 자신의 countermax 를 가져오고, 0 이 아님을 확인합니다.
2. Thread B 가 Thread A 의 countermax 를 0 으로 만듭니다.
3. Thread B 가 Thread A 의 counter 를 제거합니다.
4. 자신의 countermax 가 0 이 아님을 확인했던 Thread A 는 counter 에 값을 더하고, 이로 인해 counter 는 0 이 아닌 값을 갖습니다.

다시 말하지만, countermax 와 counter 를 별개의 변수로 두고서 어토믹하게 조정하는 것도 가능하긴 할겁니다만, 많은 주의가 필요할 것임은 분명합니다. 또한 그렇게 하는 것은 빠른 수행경로를 느리게 만들 확률이 큽니다.

이런 가능성을 더 알아보는건 독자 여러분의 숙제로 남겨두겠습니다.

Quick Quiz 5.37:

Figure 5.17 의 라인 7 에서는 C 표준을 어기는거 아닌가요? ■

Answer:

해당 코드는 바이트당 비트가 8개라 가정합니다. 이 가정은 공유 메모리 멀티프로세서에 쉽게 장착될 수 있는 현재의 모든 상용화된 마이크로프로세서에 성립합니다만, 물론 C 코드가 돌아갈 수 있는 모든 컴퓨터 시스템에 성립하진 않습니다. (C 표준에 맞추려면 대신 어떻게 할 수 있을까요? 그리고 그 때의 단점은 무엇일까요?)

Quick Quiz 5.38:

ctrandmax 변수는 하나 뿐인데, Figure 5.17 의 라인 18 에서는 왜 굳이 포인터로 받는거죠? ■

Answer:

ctrandmax 변수는 쓰레드당 한개씩만 있습니다. 뒤에서 우리는 다른 쓰레드의 ctrandmax 변수를 split_ctrandmax() 에 넘기는 코드도 보게 될겁니다.

Quick Quiz 5.39:

Figure 5.17 의 merge_ctrandmax() 는 왜 바로 atomic_t 에 값을 저장하지 않고 int 값을 리턴하는 거죠? ■

Answer:

나중에, atomic_cmpxchg() 함수에 넘기기 위해 int 리턴이 필요한 부분을 보게 될겁니다.

Quick Quiz 5.40:

우웩! Figure 5.18 라인 11 의 저 더러운 goto 는 웬말이예요? break 몰라요??? ■

Answer:

해당 `goto` 를 `break` 로 대체하려면 라인 15에서 리턴해야 할지 말아야 할지를 결정하기 위한 플래그를 하나 더 만들어야 할텐데, 이건 빠른 수행 경로에서 하고자 하는 일은 아닐 겁니다. 정말로 `goto` 를 그렇게나 싫어한다면, 이 빠른 수행 경로를 별도의 함수로 집어넣고 그 함수에서 성공인지 실패인지를 리턴하게 하고 “실패”는 느린 수행경로의 수행 필요를 나타내도록 하는게 최선일 겁니다. 이건 `goto` 싫어하는 독자분들의 연습문제로 남겨두겠습니다.

Quick Quiz 5.41:

Figure 5.18 의 라인 13-14 의 `atomic_cmpxchg()` 함수는 어떻게 실패할 수 있죠? 우린 이전 값을 라인 9에서 가져오고 나서 바꾼 적 없잖아요! ■

Answer:

나중에, Figure 5.20 의 `flush_local_count()` 함수에서 어떻게 이 쓰레드의 `ctrandmax` 변수를 Figure 5.18 의 라인 8-14 의 빠른 수행 경로 실행과 동시에 수정할 수 있는지 알아볼 겁니다.

Quick Quiz 5.42:

Figure 5.20 의 라인 14에서 `flush_local_count()` 가 `ctrandmax` 변수를 0 으로 만든 후 그냥 다시 값을 넣을 수 없는 이유는 뭐죠? ■

Answer:

이 다른 쓰레드는 `flush_local_count()` 를 호출한 쪽에서 `gblcnt_mutex` 를 해제하기 전까지는 자신의 `ctrandmax` 를 재설정 할 수 없습니다. `gblcnt_mutex` 가 해제되는 시점에선, `flush_local_count()` 호출자 쪽에서는 카운트 값의 사용을 이미 끝냈을 거고, 따라서 재설정에 문제는 없습니다 — `globalcount` 가 재설정을 허용할 만큼 충분히 크다는 가정 하에요.

Quick Quiz 5.43:

Figure 5.20 의 라인 27에서 `flush_local_count()` 가 `ctrandmax` 변수를 비우는 동안 `add_count()` 나 `sub_count()` 의 빠른 수행경로가 `ctrandmax` 를 함께 사용하면서 동시에 수행되지 못하는 이유는 뭐죠? ■

Answer:

그런 이유는 없습니다. 다음의 세가지 경우를 생각해보죠:

- 만약 `flush_local_count()` 의 `atomic_xchg()` 가 이야기된 빠른 수행경로 두개의 `split_ctrandmax()` 이전에 수행된다면, 빠른 수행경로에서는 0 이 된 `counter` 와 `countermax` 를 보게 될거고, 따라서 (물론 `delta` 가 0 이 아니라면) 그냥 느린 수행경로로 넘어갈 겁니다.
- 만약 `flush_local_count()` 의 `atomic_xchg()` 가 두 빠른 수행경로의 `split_ctrandmax()` 뒤에, 그러나 빠른 수행경로의 `atomic_cmpxchg()` 보단 앞에 수행된다면, `atomic_cmpxchg()` 를 실패할거고, 빠른 수행경로를 재시작해서 앞의 case 1 의 상황으로 돌아갈 겁니다.
- 만약 `flush_local_count()` 의 `atomic_xchg()` 가 두 빠른 수행경로의 `atomic_cmpxchg()` 뒤에 수행된다면, 빠른 수행경로는 `flush_local_count()` 가 해당 쓰레드의 `ctrandmax` 변수를 0 으로 만들기 이전에 이미 성공적으로 완료될 겁니다.

어느쪽이든, 경주는 올바르게 마무리 됩니다.

Quick Quiz 5.44:

`atomic_set()` 은 주어진 `atomic_t` 에 단순히 스도어를 할 뿐인데, 어떻게 Figure 5.21 의 라인 21에서의 `balance_count()` 는 `flush_local_count()` 의 해당 변수에 동시에 가해지는 업데이트에도 불구하고 올바르게 동작할 수 있는 거죠? ■

Answer:

`balance_count()` 와 `flush_local_count()` 의 호출자 모두 `gblcnt_mutex` 를 잡고 있으므로, 한번에 한쪽만 수행될 수 있습니다.

Quick Quiz 5.45:

하지만 시그널 핸들러는 수행 중에 다른 CPU 로 옮겨져서 수행될 수도 있잖아요. 이런 가능성은 쓰레드와 해당 쓰레드를 인터럽트 하는 시그널 핸들러 사이의 안정적인 통신을 위해 어토믹 인스트럭션과 메모리 배리어를 필요로 하게 만들지 않을까요? ■

Answer:

아니요. 시그널 핸들러가 다른 CPU 로 옮겨가면, 인터럽트된 쓰레드 역시 그리로 옮겨집니다.

Quick Quiz 5.46:

Figure 5.22에서 REQ `theft` 상태는 왜 빨간색으로 칠해졌나요? ■

Answer:

빠른 수행 경로만이 `theft` 상태를 바꿀 수 있음과 해당 쓰레드가 이 상태에 너무 오래 머무르면, 느린 수행 경로를 수행하고 있는 쓰레드는 POSIX 시그널을 다시 보낼 것임을 알리기 위해서입니다.

Quick Quiz 5.47:

Figure 5.22에서, 두개의 분리된 REQ 와 ACK `theft` 상태를 갖는 이유가 뭐죠? 왜 그 두 상태를 하나의 REQACK 상태로 만들어서 스테이트 머신을 간단하게 만들지 않는 거예요? 만약 그렇게 하면 그 상태에 먼저 도달하는 시그널 핸들러나 빠른 수행 경로가 상태를 READY로 바꿀 수 있을 텐데요. ■

Answer:

REQ 와 ACK 상태를 합치는게 나쁜 이유를 들어보자면:

1. 해당 느린 수행 경로는 REQ 와 ACK 상태를 사용해 언제 시그널이 다시 보내져야 할지 결정합니다. 만약 해당 상태들이 합쳐진다면, 해당 느린 수행 경로는 반복적으로 시그널을 보내는 수밖에 없고, 빠른 수행경로를 불필요하게 느리게 만드는 효과를 만들 겁니다.
2. 다음과 같은 레이스가 일어날 수 있습니다:
 - (a) 느린 수행 경로가 주어진 쓰레드의 상태를 REQACK 으로 만듭니다.
 - (b) 해당 쓰레드는 방금 빠른 수행 경로를 끝낸 참이었고, REQACK 상태임을 확인합니다.
 - (c) 해당 쓰레드는 시그널을 받고, 여기서도 REQACK 상태임을 확인합니다만, 빠른 수행 경로는 아무 효과를 발휘하지 못한 채이므로, 상태를 READY로 바꿉니다.
 - (d) 느린 수행 경로는 READY 상태를 확인하고, 카운트를 가져가고 상태를 IDLE로 돌려놓고 완료됩니다.
 - (e) 빠른 수행 경로는 상태를 READY로 바꾸고, 이 쓰레드에서의 다음 빠른 수행 경로 수행을 막아버립니다.

여기서의 기본적 문제는 합쳐진 REQACK 상태는 시그널 핸들러와 빠른 수행 경로 둘 다 볼 수 있다는 겁니다. 네개의 상태로 관리되는 명확한 분리 상태는 순서화된 상태 전환을 분명히 보장합니다.

그렇다면 하지만, 세개의 상태만으로도 제대로 동작하도록 할 수 있을 수도 있습니다. 만약 성공하면 네개 상태 버전과 잘 비교해 보세요. 세개 상태 버전이 정말 더 낫나요, 그리고 왜죠 또는 왜 아니죠?

Quick Quiz 5.48:

Figure 5.24의 `flush_local_count_sig()` 함수에서는 왜 `theft` 쓰레드별 변수의 사용을 `ACCESS_ONCE()`로 감싼거죠? ■

Answer:

첫번째 `ACCESS_ONCE` 사용은 (라인 11)은 필요 없는 것이라 주장할 수도 있습니다. 다음의 두 군데 사용은 (라인 14 와 16) 중요합니다. 이것들이 없어지면, 컴파일러는 라인 14-17 을 다음과 같이 바꿀 수도 있습니다:

```
14     theft = THEFT_READY;
15     if (counting) {
16         theft = THEFT_ACK;
17     }
```

느린 수행 경로는 잠깐 들어오는 값인 `THEFT_READY`를 보고서 연관된 쓰레드가 준비되기도 전에 값을 훔쳐가기 시작할테니 위험합니다.

Quick Quiz 5.49:

Figure 5.24에서, 왜 다른 쓰레드의 `countermax` 변수를 바로 접근해도 안전한 거죠? ■

Answer:

그 다른 쓰레드는 자신의 `countermax` 변수의 값을 `gblcnt_mutex` 락을 쥐지 않은 한 수정할 수 없기 때문입니다. 하지만 이 함수 호출 코드는 락을 잡고 함수를 호출하기 때문에, 그 다른 쓰레드는 해당 락을 잡을 수가 없고, 따라서 그 다른 쓰레드는 `countermax` 변수를 수정할 수 없습니다. 따라서 바로 접근해도 안전합니다 — 하지만 바꾸진 않습니다.

Quick Quiz 5.50:

Figure 5.24에서, 왜 라인 33은 현재 쓰레드가 자기 자신에게 시그널을 보내는지 체크하지 않나요? ■

Answer:

또한번 체크할 필요가 없습니다. `flush_local_count()` 는 이미 `globalize_count()` 를 호출했으니, 라인 28에서의 체크가 성공해서 뒤의 `pthread_kill()` 은 스kip될 겁니다.

Quick Quiz 5.51:

Figure 5.24 의 코드는 gcc 와 POSIX 에서 동작합니다. ISO C 표준에서 동작하게 하려면 뭐가 필요할까요? ■

Answer:

theft 변수는 안전하게 시그널 핸들러와 시그널에 인터럽트되는 코드 사이에서 안전하게 공유될 수 있도록 sig_atomic_t 타입이어야만 합니다.

Quick Quiz 5.52:

Figure 5.24 의 라인 41 에서는 왜 시그널을 다시 보내죠? ■

Answer:

지난 수십년간 많은 운영 체제는 갑자기 시그널을 잃어버리는 특성을 가졌기 때문입니다. 이게 가능인지 버그인지는 논쟁거리이지만, 그건 무의미합니다. 사용자가 보기에 분명한 증상은 커널 버그가 아니라 사용자 어플리케이션의 문제입니다.

당신의 어플리케이션의 문제입니다!

Quick Quiz 5.53:

POSIX 시그널만 느린게 아니라, 시그널을 각 쓰레드에 보내는 행위 자체가 확장성이 없어요. 만약 10,000 개의 쓰레드가 있고 읽는 쪽도 빨라야 한다면 어떻게 하시겠어요? ■

Answer:

한가지 방법은 Section 5.2.3 에서 보였던, 한개의 카운터 변수에 추정치를 합하는 방법입니다. 또 다른 방법으로는 각각 업데이트를 하는 쓰레드의 일부와 상호작용하면서 읽기 작업을 함께 하는 복수의 쓰레드를 사용하는 방법도 있겠습니다.

Quick Quiz 5.54:

아래쪽 한계는 명확하게 지키지만 위쪽 한계는 좀 정확하지 않아도 되는 한계 카운터를 원한다면 어떻게 하면 될까요? ■

Answer:

한가지 간단한 해결책은 위쪽 한계를 원하는 만큼 더 높게 잡아주는 것입니다. 그렇게 더 높게 리미트를 잡아주는 것의 한계는 카운터가 표현할 수 있는 최대의 값이 될 것입니다.

Quick Quiz 5.55:

바이어스된 카운터를 사용할 때 그 외에 뭘 하면 좋을까요? ■

Answer:

카운터가 액세스의 수가 최대값에 가까울 때에도 효과적으로 동작할 수 있도록 위쪽 리미트를 바이어스, 예상되는 최대 액세스 수, 그리고 충분한 “출렁거림”을 수용하기 충분하도록 크게 잡는게 좋을 겁니다.

Quick Quiz 5.56:

이거 참 웃기네요! 카운터를 업데이트 하기 위해 리더-라이터 락의 읽기 권한 획득을 한다니요? 뭐하는 거예요??? ■

Answer:

이상해 보일 수 있겠죠, 하지만 진짜예요! “리더-라이터 락”이라는 이름은 사실 완벽하게 의미를 설명하지 못한다는 점을 상기하면 이해가 될 거예요, 그렇죠?

Quick Quiz 5.57:

실제 시스템에 적용하려면 해결해야할 문제들이 또 뭐가 있을 수 있을까요? ■

Answer:

엄청나게 많죠!

일단 몇가지 생각을 시작할 것들은:

1. 디바이스는 여려개가 있을 수 있으니, 전역 변수는 적절치 못하고, do_io() 에 인자가 없는 것도 마찬가지죠.
2. 폴링하는 루프는 실제 시스템에서는 문제가 있을 수 있습니다. 많은 경우, 마지막으로 I/O 를 완료하는 쪽에서 디바이스 제거 쓰레드를 깨우는 편이 낫습니다.
3. I/O 는 실패할 수 있으므로, do_io() 는 리턴 값을 가져야 할 겁니다.
4. 디바이스가 고장나면, 마지막 I/O 는 성공하지 못할 것입니다. 이런 경우, 여러 복구를 위한 어떤 타임아웃 같은 것이 필요할 것입니다.
5. add_count() 와 sub_count() 모두 실패할 수 있는데 리턴값을 체크하지 않았습니다.
6. 리더-라이터 락은 확장성이 그다지 좋지 않습니다. 리더-라이터 락의 읽기 권한 획득의 높은 비용을

회피하는 방법 한가지가 Chapter 7,9 에 소개되어 있습니다.

7. 폴링 루프는 매우 낮은 에너지 효율성을 초래할 것입니다. 이벤트 기반 설계가 나을 겁니다.

Quick Quiz 5.58:

Table 5.1 의 `count_stat.c` 열에 보면 읽기 성능이 쓰레드 수에 따라 선형적으로 확장되는데요. 쓰레드 수가 늘어나면 더 많은 쓰레드별 카운터의 합이 이루어져야 하는데 어떻게 그게 가능하죠? ■

Answer:

읽는 쪽의 코드는 쓰레드의 수와 상관 없이 고정된 크기의 배열 전체를 읽어야 하기 때문에 성능에 차이가 없습니다. 반면, 뒤의 두개 알고리즘의 경우 쓰레드가 늘어나면 더 많은 일을 하게 됩니다. 더불어, 뒤의 두개 알고리즘은 쓰레드 ID 와 연관된 `__thread` 변수 사이의 맵핑을 유지하는 추가적인 계층을 갖습니다.

Quick Quiz 5.59:

Table 5.1 의 마지막 열을 보더라도 통계적 카운터 구현의 읽기쪽 성능은 매우 나쁘군요. 왜 이렇게 성능 나쁜 알고리즘을 신경쓰는거죠? ■

Answer:

“해야할 일에 걸맞는 도구를 사용하세요.”

Figure 5.3 에서 볼 수 있듯이, 하나의 변수에 어토믹 증가 오퍼레이션을 사용하는 방법은 상당한 양의 병렬적 업데이트가 있는 작업에 사용되어선 안됩니다. 반면, Table 5.1 에 보인 알고리즘들은 업데이트가 많은 상황에서 일을 훌륭하게 처리할 것입니다. 물론, 읽기가 대부분인 상황이라면, 다른걸 사용해야 합니다. 예를 들자면, Section 5.2.3 에 사용된 것과 비슷하게 한번의 로드 오퍼레이션으로 읽어낼 수 있는, 어토믹하게 증가되는 변수를 사용하는 결과적 일관성 설계와 같은 거요.

Quick Quiz 5.60:

Table 5.2 에 보여진 성능 데이터를 놓고 보자면, 우리는 항상 어토믹 오퍼레이션보다는 시그널을 사용해야겠군요, 그렇죠? ■

Answer:

그건 워크로드에 따라 달라집니다. 64-코어 시스템이라면, 단지 한개의 시그널 (약 40-나노세컨드 성능 향상) 을 만들기 위해 100 개가 넘는 어토믹하지 않은

오퍼레이션들의 실행 (약 5-마이크로세컨드 성능 저하) 이 필요합니다. 더욱 읽기 위주인 워크로드는 여전히 존재하지만, 현재 처리해야하는 특정 워크로드에 신경쓸 필요가 있습니다.

또한, 역사적으로 메모리 배리어는 일반 인스트럭션들에 비해 비용이 비쌌지만, 당신이 운용하게 될 특정 하드웨어에서도 그러한지 확인해 봐야 합니다. 컴퓨터 하드웨어의 특성은 시간에 따라 변하고, 알고리즘도 그에 맞춰 변해야만 합니다.

Quick Quiz 5.61:

Table 5.2 에 보여진 읽는 쓰레드간의 락 컨텐션을 해결하기 위해 고급 테크닉들이 사용될 수 있을까요? ■

Answer:

한가지 해결책은 scalable non-zero indicators(SNZI) [ELLM07] 처럼 업데이트 쪽 성능을 약간 포기하는 겁니다. SNZI 외에도 이 해결책을 구현하는 여러 방법이 있겠지만, 그건 독자의 몫으로 남겨두겠습니다. 자주 획득이 요청되는 글로벌 락을 낮은 레벨의 계층의 로컬 락의 획득들로 대체하는 계층적 방법들도 이 문제를 잘 해결할 겁니다.

Quick Quiz 5.62:

++ 오퍼레이터는 1,000 자리 숫자에도 잘 동작해요!
연산자 오버로딩이라고 못들어봤어요??? ■

Answer:

C++ 언어에서라면 그런 수를 구현하는 클래스에 액세스가 가능하다는 가정 하에 1,000 자리 숫자에도 ++ 을 사용할 수 있겠죠. 하지만 최소 2010년 까지는, C 언어는 오퍼레이터 오버로딩을 허용하지 않습니다.

Quick Quiz 5.63:

하지만 우리가 모든 것을 분할할 거라면, 왜 공유 메모리 멀티쓰레딩을 신경쓰죠? 그냥 문제를 완벽하게 분할해버리고 각 분할된 조각들을 여러 프로세스들로, 각자의 어드레스 스페이스에서 처리하도록 돌리지 않는건가요? ■

Answer:

사실, 별도의 어드레스 스페이스를 갖는 여러 프로세스들은 병렬성을 보일 수 있는 훌륭한 방법으로, 포크-조인 방법론의 지지자들과 Erlang 언어가 그 사실을 잘 입증합니다. 하지만, 공유 메모리 병렬성만의

장점 역시 일부 있습니다:

1. 어플리케이션의 성능에 치명적인 부분만이 분할되어야 하고, 그런 성능에 치명적인 부분은 일반적으로 어플리케이션의 작은 부분입니다.
2. 캐시 미스는 개별적 레지스터간 인스트럭션들에 비하면 매우 느리지만, TCP/IP 네트워킹과 같은 것들보다는 빠른 프로세스간 통신 (inter-process-communication) 기능들에 비해서도 상당히 빠릅니다.
3. 공유 메모리 멀티프로세서들은 이미 시장에 나와 있고 상당히 저렴하므로, 1990년대와는 정반대로, 공유 메모리 병렬성을 사용하는데 비용 문제는 거의 없습니다.

항상 말하듯이, 처리해야 하는 일에 걸맞는 도구를 사용하세요!

D.6 Partitioning and Synchronization Design

Quick Quiz 6.1:

Is there a better solution to the Dining Philosophers Problem? ■

Answer:

One such improved solution is shown in Figure D.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, this approach offers greatly improved disease control.

This solution might seem like cheating to some, but such “cheating” is key to finding good solutions to many concurrency problems.

Quick Quiz 6.2:

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

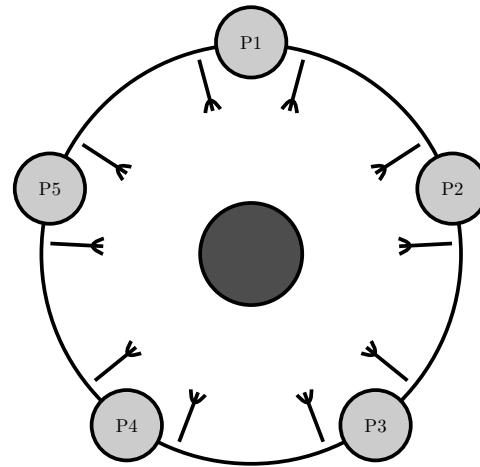


Figure D.3: Dining Philosophers Problem, Fully Partitioned

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 6.3:

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 non-empty queue, release both locks, and return.

Quick Quiz 6.4:

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.³ Do not forget to compare to a sequential implementation!

Quick Quiz 6.5:

Move *all* the elements to the queue that became empty? In what possible universe is this brain-dead 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 another question, namely, in what possible universe emptying from both ends concurrently would be a reasonable thing to do. Work-stealing queues are one possible answer to this question.

Quick Quiz 6.6:

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 (see Section 6.1.1).

Quick Quiz 6.7:

Why is it necessary to retry the right-dequeue operation on line 28 of Figure 6.12? ■

Answer:

This retry is necessary because some other thread might have enqueued an element between the time that this thread dropped `d->rlock` on line 25 and the time that it reacquired this same lock on line 27.

Quick Quiz 6.8:

Surely the left-hand lock must *sometimes* be available!!! So why is it necessary that line 25 of Figure 6.12 unconditionally release the right-hand lock? ■

³ The studies by Dalessandro et al. [DCW⁺11] and Dice et al. [DLM⁺10] are good starting points.

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 6.9:

Why are there not one but two solutions to the double-ended queue problem? ■

Answer:

There are actually at least three. The third, by Dominik Dingel, makes interesting use of reader-writer locking, and may be found in `lockrwdeq.c`.

Quick Quiz 6.10:

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 create a double-ended queue that allows multiple concurrent operations at each end? If so, how? If not, why not?

Quick Quiz 6.11:

Is there a significantly better way of handling concurrency for double-ended queues? ■

Answer:

One approach is to 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 achieve 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 dequeue simply cannot provide strong ordering guarantees. After all, the fact that a given thread removed an item first in no way implies that it will process that item first [HKLP12]. And if there are no guarantees, we may as well obtain the performance benefits that come with refusing to provide these guarantees.

Regardless of whether or not the problem can be transformed to use multiple queues, it is worth asking whether work can be batched so that each enqueue and dequeue operation corresponds to larger units of work. This batching approach decreases contention on the queue data structures, which increases both performance and scalability, as will be seen in Section 6.3. After all, if you must incur high synchronization overheads, be sure you are getting your money’s worth.

Other researchers are working on other ways to take advantage of limited ordering guarantees in queues [KLP12].

Quick Quiz 6.12:

Don’t all these problems with critical sections mean that we should just always use non-blocking synchronization [Her90], which don’t have critical sections? ■

Answer:

Although non-blocking synchronization can be very useful in some situations, it is no panacea. Also, non-blocking synchronization really does have critical sections, as noted by Josh Triplett. For example, in a non-blocking algorithm based on compare-and-swap operations, the code starting at the initial load and continuing to the compare-and-swap is in many ways analogous to a lock-based critical section.

Quick Quiz 6.13:

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 6.4.2). 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 7. 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 9.

For more on providing existence guarantees, see Chapters 7 and 9.

Quick Quiz 6.14:

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 6.23 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 6.15:

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 minuscule 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 incur the overhead required to spawn a thread, now is it?

Quick Quiz 6.16:

In what situation would hierarchical locking work well? ■

Answer:

If the comparison on line 31 of Figure 6.26 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 6.17:

In Figure 6.32, 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 6.18:

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-thread target pool size s of three, number of threads n equal to two, and assuming that the per-thread pools are initially empty with none of the memory in use, what is the smallest allocation run length m at which failures can occur? (Recall that each thread repeatedly allocates m blocks of memory, and then frees the m blocks of memory.) Alternatively, given n threads each with pool size s , and where each thread repeatedly first allocates m blocks of memory and then frees those m blocks, how large must the global pool size be? *Note:* Obtaining the correct answer will require you to examine the `smpalloc.c` source code, and very likely single-step it as well. You have been warned! ■

Answer:

This solution is adapted from one put forward by Alexey Roystman. It is based on the following definitions:

g Number of blocks globally available.

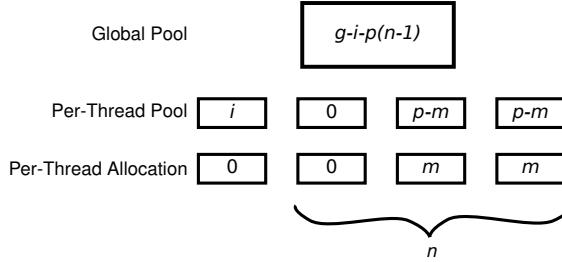


Figure D.4: Allocator Cache Run-Length Analysis

- i* Number of blocks left in the initializing thread's per-thread pool. (This is one reason you needed to look at the code!)
- m* Allocation/free run length.
- n* Number of threads, excluding the initialization thread.
- p* Per-thread maximum block consumption, including both the blocks actually allocated and the blocks remaining in the per-thread pool.

The values g , m , and n are given. The value for p is m rounded up to the next multiple of s , as follows:

$$p = s \left\lceil \frac{m+s-1}{s} \right\rceil \quad (\text{D.6})$$

The value for i is as follows:

$$i = \begin{cases} g & (\text{mod } 2s) = 0 : 2s \\ g & (\text{mod } 2s) \neq 0 : g \quad (\text{mod } 2s) \end{cases} \quad (\text{D.7})$$

The relationships between these quantities is shown in Figure D.4. The global pool is shown on the top of this figure, and the “extra” initializer thread’s per-thread pool and per-thread allocations are the left-most pair of boxes. The initializer thread has no blocks allocated, but has i blocks stranded in its per-thread pool. The rightmost two pairs of boxes are the per-thread pools and per-thread allocations of threads holding the maximum possible number of blocks, while the second-from-left pair of boxes represents the thread currently trying to allocate.

The total number of blocks is g , and adding up the per-thread allocations and per-thread pools, we see that the global pool contains $g - i - p(n - 1)$ blocks. If the allocating thread is to be successful, it needs at least m blocks in the global pool, in other words:

$$g - i - p(n - 1) \geq m \quad (\text{D.8})$$

The question has $g = 40$, $s = 3$, and $n = 2$. Equation D.7 gives $i = 4$, and Equation D.6 gives $p = 18$ for $m = 18$ and $p = 21$ for $m = 19$. Plugging these into Equation D.8 shows that $m = 18$ will not overflow, but that $m = 19$ might well do so.

The presence of i could be considered to be a bug. After all, why allocate memory only to have it stranded in the initialization thread’s cache? One way of fixing this would be to provide a `memblock_flush()` function that flushed the current thread’s pool into the global pool. The initialization thread could then invoke this function after freeing all of the blocks.

D.7 Locking

Quick Quiz 7.1:

Just how can serving as a whipping boy be considered to be in any way honorable??? ■

Answer:

The reason locking serves as a research-paper whipping boy is because it is heavily used in practice. In contrast, if no one used or cared about locking, most research papers would not bother even mentioning it.

Quick Quiz 7.2:

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 be 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 7.3:

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 are! 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 7.4:

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 8) or through use of deferral mechanisms such as reference counting (as discussed in Chapter 9).

Quick Quiz 7.5:

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 7.6:

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 7.7:

Can the transformation from Figure 7.9 to Figure 7.10 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 7.8:

But the complexity in Figure 7.10 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 7.1.2).

Quick Quiz 7.9:

When using the “acquire needed locks first” approach described in Section 7.1.1.6, how can livelock be avoided? ■

Answer:

Provide an additional global lock. If a given thread has repeatedly tried and failed to acquire the needed locks, then have that thread unconditionally acquire the new global lock, and then unconditionally acquire any needed locks. (Suggested by Doug Lea.)

Quick Quiz 7.10:

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 a another lock that is acquired within a signal handler.

Quick Quiz 7.11:

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 7.12:

If acquiring locks in signal handlers is such a bad idea,

why even discuss ways of making it safe? ■

Answer:

Because these same rules apply to the interrupt handlers used in operating-system kernels and in some embedded applications.

In many application environments, acquiring locks in signal handlers is frowned upon [Ope97]. However, that does not stop clever developers from (usually unwisely) fashioning home-brew locks out of atomic operations. And atomic operations are in many cases perfectly legal in signal handlers.

Quick Quiz 7.13:

Given an object-oriented application that passes control freely among a group of objects such that there is no straightforward locking hierarchy,⁴ 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, and likely also avoids synchronization overhead.
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 7.1.1.7.
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 7.1.1.5.
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.

⁴ Also known as “object-oriented spaghetti code.”

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 7.1.1.6.

6. Use transactional memory. This approach has a number of advantages and disadvantages which will be discussed in Section 17.2.
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 7.14:

How can the livelock shown in Figure 7.11 be avoided? ■

Answer:

Figure 7.10 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”.

That said, one good-and-sufficient approach due to Doug Lea is to use conditional locking as described in Section 7.1.1.5, but combine this with acquiring all needed locks first, before modifying shared data, as described in Section 7.1.1.6. If a given critical section retries too many times, unconditionally acquire a global lock, then unconditionally acquire all the needed locks. This avoids both deadlock and livelock, and scales reasonably assuming that the global lock need not be acquired too often.

Quick Quiz 7.15:

What problems can you spot in the code in Figure 7.12? ■

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 7.16:

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 7.17:

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. 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. 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.

Of course, the downside of placing the lock and data into separate cache lines is that the code will incur two cache misses rather than only one in the uncontended case.

Quick Quiz 7.18:

Does it ever make sense to have an exclusive lock acquisition immediately followed by a release of that same lock, that is, an empty critical section? ■

Answer:

This usage is rare, but is occasionally used. The point is that the semantics of exclusive locks have two components: (1) the familiar data-protection semantic and (2) a messaging semantic, where releasing a given lock notifies a waiting acquisition of that same lock. An empty critical section uses the messaging component without the data-protection component.

The rest of this answer provides some example uses of empty critical sections, however, these examples should be considered “gray magic.”⁵ As such, empty critical sections are almost never used in practice. Nevertheless, pressing on into this gray area...

One historical use of empty critical sections appeared in the networking stack of the 2.4 Linux kernel. This usage pattern can be thought of as a way of approximating the effects of read-copy update (RCU), which is discussed in Section 9.3.

The empty-lock-critical-section idiom can also be used to reduce lock contention in some situations. For example, consider a multithreaded user-space application where each thread processes unit of work maintained in a per-thread list, where threads are prohibited from touching each others’ lists. There could also be updates that require that all previously scheduled units of work have completed before the update can progress. One way to handle this is to schedule a unit of work on each thread, so that when all of these units of work complete, the update may proceed.

In some applications, threads can come and go. For example, each thread might correspond to one user of the application, and thus be removed when that user logs out or otherwise disconnects. In many applications, threads cannot depart atomically: They must instead explicitly unravel themselves from various portions of the application using a specific sequence of actions. One specific action will be refusing to accept further requests from other threads, and another specific action will be disposing of any remaining units of work on its list, for example, by placing these units of work in a global work-item-disposal list to be taken by one of the remaining threads. (Why not just drain the thread’s work-item list by executing each item? Because a given work item might generate more work items, so that the list could not be drained in a timely fashion.)

If the application is to perform and scale well, a good locking design is required. One common solution is to have a global lock (call it G) protecting the entire pro-

cess of departing (and perhaps other things as well), with finer-grained locks protecting the individual unraveling operations.

Now, a departing thread must clearly refuse to accept further requests before disposing of the work on its list, because otherwise additional work might arrive after the disposal action, which would render that disposal action ineffective. So simplified pseudocode for a departing thread might be as follows:

1. Acquire lock G .
2. Acquire the lock guarding communications.
3. Refuse further communications from other threads.
4. Release the lock guarding communications.
5. Acquire the lock guarding the global work-item-disposal list.
6. Move all pending work items to the global work-item-disposal list.
7. Release the lock guarding the global work-item-disposal list.
8. Release lock G .

Of course, a thread that needs to wait for all pre-existing work items will need to take departing threads into account. To see this, suppose that this thread starts waiting for all pre-existing work items just after a departing thread has refused further communications from other threads. How can this thread wait for the departing thread’s work items to complete, keeping in mind that threads are not allowed to access each others’ lists of work items?

One straightforward approach is for this thread to acquire G and then the lock guarding the global work-item-disposal list, then move the work items to its own list. The thread then releases both locks, places a work item on the end of its own list, and then waits for all of the work items that it placed on each thread’s list (including its own) to complete.

This approach does work well in many cases, but if special processing is required for each work item as it is pulled in from the global work-item-disposal list, the result could be excessive contention on G . One way to avoid that contention is to acquire G and then immediately release it. Then the process of waiting for all prior work items looks something like the following:

⁵ Thanks to Alexey Roystman for this description.

1. Set a global counter to one and initialize a condition variable to zero.
2. Send a message to all threads to cause them to atomically increment the global counter, and then to enqueue a work item. The work item will atomically decrement the global counter, and if the result is zero, it will set a condition variable to one.
3. Acquire G , which will wait on any currently departing thread to finish departing. Because only one thread may depart at a time, all the remaining threads will have already received the message sent in the preceding step.
4. Release G .
5. Acquire the lock guarding the global work-item-disposal list.
6. Move all work items from the global work-item-disposal list to this thread's list, processing them as needed along the way.
7. Release the lock guarding the global work-item-disposal list.
8. Enqueue an additional work item onto this thread's list. (As before, this work item will atomically decrement the global counter, and if the result is zero, it will set a condition variable to one.)
9. Wait for the condition variable to take on the value one.

Once this procedure completes, all pre-existing work items are guaranteed to have completed. The empty critical sections are using locking for messaging as well as for protection of data.

Quick Quiz 7.19:

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 7.20:

The code in Figure 7.15 is ridiculously complicated! Why not conditionally acquire a single global lock? ■

Answer:

Conditionally acquiring a single global lock does work very well, but only for relatively small numbers of CPUs. To see why it is problematic in systems with many hundreds of CPUs, look at Figure 5.3 and extrapolate the delay from eight to 1,000 CPUs.

Quick Quiz 7.21:

Wait a minute! If we "win" the tournament on line 16 of Figure 7.15, we get to do all the work of `do_force_quiescent_state()`. Exactly how is that a win, really? ■

Answer:

How indeed? This just shows that in concurrency, just as in life, one should take care to learn exactly what winning entails before playing the game.

Quick Quiz 7.22:

Why not rely on the C language's default initialization of zero instead of using the explicit initializer shown on line 2 of Figure 7.16? ■

Answer:

Because this default initialization does not apply to locks allocated as auto variables within the scope of a function.

Quick Quiz 7.23:

Why bother with the inner loop on lines 7-8 of Figure 7.16? 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 7.24:

Why not simply store zero into the lock word on line 14 of Figure 7.16? ■

Answer:

This can be a legitimate implementation, but only if this store is preceded by a memory barrier and makes use of `ACCESS_ONCE()`. 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 7.25:

How can you tell if one counter is greater than another, while accounting for counter wrap? ■

Answer:

In the C language, the following macro correctly handles this:

```
#define ULONG_CMP_LT(a, b) \
    (ULONG_MAX / 2 < (a) - (b))
```

Although it is tempting to simply subtract two signed integers, this should be avoided because signed overflow is undefined in the C language. For example, if the compiler knows that one of the values is positive and the other negative, it is within its rights to simply assume that the positive number is greater than the negative number, even though subtracting the negative number from the positive number might well result in overflow and thus a negative number.

How could the compiler know the signs of the two numbers? It might be able to deduce it based on prior assignments and comparisons. In this case, if the per-CPU counters were signed, the compiler could deduce that they were always increasing in value, and then might assume that they would never go negative. This assumption could well lead the compiler to generate unfortunate code [McK12d, Reg10].

Quick Quiz 7.26:

Which is better, the counter approach or the flag approach? ■

Answer:

The flag approach will normally suffer fewer cache

misses, but a better answer is to try both and see which works best for your particular workload.

Quick Quiz 7.27:

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 7.28:

What if the element we need to delete is not the first element of the list on line 8 of Figure 7.17? ■

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 7.29:

What race condition can occur in Figure 7.17? ■

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)`, reaching line 10, but spins on the lock because Thread 0 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!

D.8 Data Ownership

Quick Quiz 8.1:

What form of data ownership is extremely difficult to avoid when creating shared-memory parallel programs (for example, using `pthread`s) 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 8.2:

What synchronization remains in the example shown in Section 8.1? ■

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 8.3:

Is there any shared data in the example shown in Section 8.1? ■

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 8.4:

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 8.5:

What mechanisms other than POSIX signals may be used for function shipping? ■

Answer:

There is a very large number of such mechanisms, including:

1. System V message queues.
2. Shared-memory dequeue (see Section 6.1.2).
3. Shared-memory mailboxes.
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 8.6:

But none of the data in the `eventual()` function shown on lines 15-32 of Figure 5.8 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 8.2.

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 8.7:

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.

D.9 Deferred Processing

Quick Quiz 9.1:

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 9.2:

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 9.2.

Quick Quiz 9.3:

Suppose that just after the `atomic_sub_and_test()` on line 22 of Figure 9.2 is invoked, that some other CPU invokes `kref_get()`. Doesn't this result in that other CPU now having an illegal reference to a released object? ■

Answer:

This cannot happen if these functions are used correctly. It is illegal to invoke `kref_get()` unless you already hold a reference, in which case the `kref_sub()` could not possibly have decremented the counter to zero.

Quick Quiz 9.4:

Suppose that `kref_sub()` returns zero, indicating that the `release()` function was not invoked. Under what conditions can the caller rely on the continued existence of the enclosing object? ■

Answer:

The caller cannot rely on the continued existence of the object unless it knows that at least one reference will continue to exist. Normally, the caller will have no way of knowing this, and must therefore carefully avoid referencing the object after the call to `kref_sub()`.

Quick Quiz 9.5:

Why not just pass `kfree()` as the release function? ■

Answer:

Because the `kref` structure normally is embedded in a larger structure, and it is necessary to free the entire structure, not just the `kref` field. This is normally accomplished by defining a wrapper function that does a `container_of()` and then a `kfree()`.

Quick Quiz 9.6:

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 9.7:

Why does `hp_store()` in Figure 9.5 take a double indirection to the data element? Why not `void *` instead of `void **`? ■

Answer:

Because `hp_record()` must check for concurrent modifications. To do that job, it needs a pointer to a pointer to the element, so that it can check for a modification to the pointer to the element.

Quick Quiz 9.8:

Why does `hp_store()`'s caller need to restart its traversal from the beginning in case of failure? Isn't that inefficient for large data structures? ■

Answer:

It might be inefficient in some sense, but the fact is that such restarting is absolutely required for correctness. To see this, consider a hazard-pointer-protected linked list containing elements A, B, and C that is subject to the following sequence of events:

1. Thread 0 stores a hazard pointer to element B (having presumably traversed to element B from element A).
2. Thread 1 removes element B from the list, which sets the pointer from element B to element C to a special `HAZPTR_POISON` value in order to mark the deletion. Because Thread 0 has a hazard pointer to element B, it cannot yet be freed.
3. Thread 1 removes element C from the list. Because there are no hazard pointers referencing element C, it is immediately freed.
4. Thread 0 attempts to acquire a hazard pointer to now-removed element B's successor, but sees the `HAZPTR_POISON` value, and thus returns zero, forcing the caller to restart its traversal from the beginning of the list.

Which is a very good thing, because otherwise Thread 0 would have attempted to access the now-freed element C, which might have resulted in arbitrarily horrible memory corruption, especially if the memory for element C had since been re-allocated for some other purpose.

Quick Quiz 9.9:

Given that papers on hazard pointers use the bottom bits of each pointer to mark deleted elements, what is up with HAZPTR_POISON? ■

Answer:

The published implementations of hazard pointers used non-blocking synchronization techniques for insertion and deletion. These techniques require that readers traversing the data structure “help” updaters complete their updates, which in turn means that readers need to look at the successor of a deleted element.

In contrast, we will be using locking to synchronize updates, which does away with the need for readers to help updaters complete their updates, which in turn allows us to leave pointers’ bottom bits alone. This approach allows read-side code to be simpler and faster.

Quick Quiz 9.10:

But don’t these restrictions on hazard pointers also apply to other forms of reference counting? ■

Answer:

These restrictions apply only to reference-counting mechanisms whose reference acquisition can fail.

Quick Quiz 9.11:

An `atomic_read()` and an `atomic_set()` that are non-atomic? Is this some kind of bad joke??? ■

Answer:

It might well seem that way, but in situations where no other CPU has access to the atomic variable in question, the overhead of an actual atomic instruction would be wasteful. Two examples where no other CPU has access are during initialization and cleanup.

Quick Quiz 9.12:

But hazard pointers don’t write to the data structure! ■

Answer:

Indeed, they do not. However, they do write to the hazard pointers themselves, and, more important, require that possible failures be handled for all `hp_store()` calls, each of which might fail. Therefore, although hazard pointers are extremely useful, it is still worth looking for improved mechanisms.

Quick Quiz 9.13:

Why isn’t this sequence-lock discussion in Chapter 7, 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 9.14:

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 9.3.3.6. But in that case, you would be using some other synchronization mechanism in addition to sequence locks!

Quick Quiz 9.15:

Why bother with the check on line 19 of `read_seqbegin()` in Figure 9.9? 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 9.9.

Quick Quiz 9.16:

Why is the `smp_mb()` on line 29 of Figure 9.9 needed? ■

Answer:

If it was omitted, both the compiler and the CPU would be within their rights to move the critical section preceding the call to `read_seqretry()` down below this function. This would prevent the sequence lock from protecting the critical section. The `smp_mb()` primitive prevents such reordering.

Quick Quiz 9.17:

Can't weaker memory barriers be used in the code in Figure 9.9? ■

Answer:

In older versions of the Linux kernel, no.

In very new versions of the Linux kernel, line 17 could use `smp_load_acquire()` instead of `ACCESS_ONCE()`, which in turn would allow the `smp_mb()` on line 18 to be dropped. Similarly, line 44 could use an `smp_store_release()`, for example, as follows:

```
smp_store_release(&slp->seq, ACCESS_ONCE(slp->seq) + 1);
```

This would allow the `smp_mb()` on line 43 to be dropped.

Quick Quiz 9.18:

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 with the sequence-locking updates!

Quick Quiz 9.19:

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 9.20:

Why isn't `seq` on line 2 of Figure 9.9 `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 9.21:

But doesn't Section 9.2'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 9.22:

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 9.23:

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 9.24:

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 D.5. Note that this means that multiple concurrent deletions might be waiting in `synchronize_rcu()`.

Quick Quiz 9.25:

How many RCU versions of a given list can be active at any given time? ■

```

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 D.5: Concurrent RCU Deletion

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, suppose that only the search, the update, and the `list_replace_rcu()` were protected by a lock, so that the `synchronize_rcu()` was outside of that lock, similar to the code shown in Figure D.5. Suppose further that a large number of threads undertook an RCU replacement at about the same time, and that readers are also constantly traversing the data structure.

Then the following sequence of events could occur, starting from the end state of Figure 9.22:

1. Thread A traverses the list, obtaining a reference to the 5,2,3 element.
2. Thread B replaces the 5,2,3 element with a new 5,2,4 element, then waits for its `synchronize_rcu()` call to return.
3. Thread C traverses the list, obtaining a reference to the 5,2,4 element.
4. Thread D replaces the 5,2,4 element with a new 5,2,5 element, then waits for its `synchronize_rcu()` call to return.
5. Thread E traverses the list, obtaining a reference to the 5,2,5 element.
6. Thread F replaces the 5,2,5 element with a new 5,2,6 element, then waits for its `synchronize_rcu()` call to return.
7. Thread G traverses the list, obtaining a reference to the 5,2,6 element.

8. And the previous two steps repeat quickly, so that all of them happen before any of the `synchronize_rcu()` calls return.

Thus, there can 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 9.26:

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 9.27:

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:

```

1 for (i = 0; i < CSCOUNT_SCALE; i++) {
2     rCU_read_lock();
3     rCU_read_unlock();
4 }

```

Next, consider the effective definitions of `rcu_read_lock()` and `rcu_read_unlock()`:

```

1 #define rCU_read_lock()    do { } while (0)
2 #define rCU_read_unlock() do { } while (0)

```

Consider also that the compiler does simple optimizations, allowing it to replace the loop with:

```
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 9.28:

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 9.29:

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:

```
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 9.30:

Immunity to both deadlock and priority inversion??? Sounds too good to be true. Why should I believe that this is even possible? ■

Answer:

It really does work. After all, if it didn’t work, the Linux kernel would not run.

Quick Quiz 9.31:

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 9.32:

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% of measured values in some cases. The dip in overhead therefore might well be a statistical aberration.

Quick Quiz 9.33:

What if the element we need to delete is not the first element of the list on line 9 of Figure 9.32? ■

Answer:

As with Figure 7.17, 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 9.34:

Why is it OK to exit the RCU read-side critical section on line 15 of Figure 9.32 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 9.35:

Why not exit the RCU read-side critical section on line 23 of Figure 9.32 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 re-allocated as some other type of data structure. This is a fatal memory-corruption error.

Quick Quiz 9.36:

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 9.3.3.6 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 9.37:

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()`,

```

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 D.6: Using RCU to Wait for Mythical Preemptible NMIs to Finish

and to replace the `synchronize_sched()` with `synchronize_rcu()`, perhaps as shown in Figure D.6.

Quick Quiz 9.38:

Why do some of the cells in Table 9.4 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 9.39:

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 9.40:

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 9.41:

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.

However, the `rcu_dereference()` family of functions apply to all flavors of RCU. (There was an attempt to have per-flavor variants of `rcu_dereference()`, but it was just too messy.)

Quick Quiz 9.42:

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 pre-emptible 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 9.43:

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 9.44:

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 9.45:

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 9.46:

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 D.7 could still result in deadlock.

```

1 idx = srcu_read_lock(&ssa);
2 synchronize_srcu(&ssb);
3 srcu_read_unlock(&ssa, idx);
4
5 /* . . . */
6
7 idx = srcu_read_lock(&ssb);
8 synchronize_srcu(&ssa);
9 srcu_read_unlock(&ssb, idx);

```

Figure D.7: Multistage SRCU Deadlocks

Quick Quiz 9.47:

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 9.48:

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 9.49:

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 D.8.

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 9.50:

Why wouldn’t any deadlock in the RCU implementation

```

1  rcu_read_lock();
2  preempt_disable();
3  p = rcu_dereference(global_pointer);
4
5  /* . . . */
6
7  preempt_enable();
8  rcu_read_unlock();

```

Figure D.8: Diverse RCU Read-Side Nesting

```

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 D.9: Deadlock in Lock-Based RCU Implementation

in Figure 9.36 also be a deadlock in any other RCU implementation? ■

Answer:

Suppose the functions `foo()` and `bar()` in Figure D.9 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.

Quick Quiz 9.51:

Why not simply use reader-writer locks in the RCU implementation in Figure 9.36 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 9.52:

Wouldn't it be cleaner to acquire all the locks, and then release them all in the loop from lines 15-18 of Figure 9.37? 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 9.53:

Is the implementation shown in Figure 9.37 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 could 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 imple-

mentation 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 9.54:

Isn't one advantage of the RCU algorithm shown in Figure 9.37 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 9.55:

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 9.3.5.9.

Quick Quiz 9.56:

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 9.57:

Why not simply make `rcu_read_lock()` wait when a concurrent `synchronize_rcu()` has been waiting too long in the RCU implementation in Figure 9.38? 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 9.3.5.4 for a proper cure.

Quick Quiz 9.58:

Why the memory barrier on line 5 of `synchronize_rcu()` in Figure 9.41 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 9.41 are really needed. See Section 12 for information on using these tools. The first correct and complete response will be credited.

Quick Quiz 9.59:

Why is the counter flipped twice in Figure 9.41? 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 9.40 picks up `rcu_idx`, finding its value to be zero.
2. Line 8 of `synchronize_rcu()` in Figure 9.41 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 9.60:

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 9.40? ■

Answer:

Using non-atomic operations would cause increments and decrements to be lost, in turn causing the implementation to fail. See Section 9.3.5.5 for a safe way to use

non-atomic operations in `rcu_read_lock()` and `rcu_read_unlock()`.

Quick Quiz 9.61:

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 9.62:

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 9.63:

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 [McK09a].

Quick Quiz 9.64:

If any even value is sufficient to tell `synchronize_rcu()` to ignore a given task, why don't lines 10 and 11 of Figure 9.49 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_crt` 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 9.65:

Why are the memory barriers on lines 19 and 31 of Figure 9.49 needed? Aren't the memory barriers inherent in the locking primitives on lines 20 and 30 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 9.66:

Couldn't the update-side batching optimization described in Section 9.3.5.6 be applied to the implementation shown in Figure 9.49? ■

Answer:

Indeed it could, with a few modifications. This work is

left as an exercise for the reader.

Quick Quiz 9.67:

Is the possibility of readers being preempted in lines 3-4 of Figure 9.49 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 9.3.5.8. 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 9.68:

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 end up with!

Quick Quiz 9.69:

Given the algorithm shown in Figure 9.51, 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 9.70:

Again, given the algorithm shown in Figure 9.51, 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 9.3.5.7.

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 9.71:

Doesn't the additional memory barrier shown on line 14 of Figure 9.53, 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 9.72:

Why are the two memory barriers on lines 19 and 22 of Figure 9.53 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 9.73:

To be sure, the clock frequencies of Power systems in 2008 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 9.74:

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 9.53 and 9.54? ■

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 9.75:

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 9.76:

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 9.77:

The statistical-counter implementation shown in Figure 5.9 (`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?

See Section 13.2.1 on page 236 for more details.

Quick Quiz 9.78:

Section 5.5 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 13.2.2 on Page 237 for one solution to this problem.

Quick Quiz 9.79:

But can't both reference counting and hazard pointers can also acquire a reference to multiple data elements with constant overhead? A single reference count can cover multiple data elements, right? ■

Answer:

Almost. As we will see in the "Unconditional Acquisition" column, neither reference counting nor hazard pointers provide unconditional acquisition of references, so

acquiring a reference can have non-constant overhead in the face of conflicting updates.

In addition, using a single reference count to cover multiple data items can have severe consequences, for example, you cannot remove any of the data items until all references to all of them have been released. This can result in more complex data-element-cleanup code, and can also increase memory footprint to rival that of RCU. In other words, the increased memory footprint is a consequence not of RCU in particular, but of bulk reference-count acquisition in general.

D.10 Data Structures

Quick Quiz 10.1:

But there are many types of hash tables, of which the chained hash tables described here are but one type. Why the focus on chained hash tables? ■

Answer:

Chained hash tables are completely partitionable, and thus well-suited to concurrent use. There are other completely-partitionable hash tables, for example, split-ordered list [SS06], but they are considerably more complex. We therefore start with chained hash tables.

Quick Quiz 10.2:

But isn't the double comparison on lines 15-18 in Figure 10.4 inefficient in the case where the key fits into an unsigned long? ■

Answer:

Indeed it is! However, hash tables quite frequently store information with keys such as character strings that do not necessarily fit into an unsigned long. Simplifying the hash-table implementation for the case where keys always fit into unsigned longs is left as an exercise for the reader.

Quick Quiz 10.3:

Instead of simply increasing the number of hash buckets, wouldn't it be better to cache-align the existing hash buckets? ■

Answer:

The answer depends on a great many things. If the hash table has a large number of elements per bucket, it would clearly be better to increase the number of hash buckets. On the other hand, if the hash table is lightly loaded, the answer depends on the hardware, the effectiveness of the hash function, and the workload. Interested readers are encouraged to experiment.

Quick Quiz 10.4:

Given the negative scalability of the Schrödinger's Zoo application across sockets, why not just run multiple copies of the application, with each copy having a subset of the animals and confined to run on a single socket? ■

Answer:

You can do just that! In fact, you can extend this idea to large clustered systems, running one copy of the application on each node of the cluster. This practice is called "sharding", and is heavily used in practice by large web-based retailers [DHJ⁺07].

However, if you are going to shard on a per-socket basis within a multisocket system, why not buy separate smaller and cheaper single-socket systems, and then run one shard of the database on each of those systems?

Quick Quiz 10.5:

But if elements in a hash table can be deleted concurrently with lookups, doesn't that mean that a lookup could return a reference to a data element that was deleted immediately after it was looked up? ■

Answer:

Yes it can! This is why `hashtab_lookup()` must be invoked within an RCU read-side critical section, and it is why `hashtab_add()` and `hashtab_del()` must also use RCU-aware list-manipulation primitives. Finally, this is why the caller of `hashtab_del()` must wait for a grace period (e.g., by calling `synchronize_rcu()`) before freeing the deleted element.

Quick Quiz 10.6:

The dangers of extrapolating from eight CPUs to 60 CPUs was made quite clear in Section 10.2.3. But why should extrapolating up from 60 CPUs be any safer? ■

Answer:

It isn't any safer, and a useful exercise would be to run these programs on larger systems. That said, other testing has shown that RCU read-side primitives offer consistent performance and scalability up to at least 1024 CPUs.

Quick Quiz 10.7:

The code in Figure 10.25 computes the hash twice! Why this blatant inefficiency? ■

Answer:

The reason is that the old and new hash tables might have completely different hash functions, so that a hash computed for the old table might be completely irrelevant to the new table.

Quick Quiz 10.8:

How does the code in Figure 10.25 protect against the resizing process progressing past the selected bucket? ■

Answer:

It does not provide any such protection. That is instead the job of the update-side concurrency-control functions described next.

Quick Quiz 10.9:

The code in Figures 10.25 and 10.26 computes the hash and executes the bucket-selection logic twice for updates! Why this blatant inefficiency? ■

Answer:

This approach allows the `hashtorture.h` testing infrastructure to be reused. That said, a production-quality resizable hash table would likely be optimized to avoid this double computation. Carrying out this optimization is left as an exercise for the reader.

Quick Quiz 10.10:

Suppose that one thread is inserting an element into the new hash table during a resize operation. What prevents this insertion from being lost due to a subsequent resize operation completing before the insertion does? ■

Answer:

The second resize operation will not be able to move beyond the bucket into which the insertion is taking

place due to the insertion holding the lock on one of the hash buckets in the new hash table (the second hash table of three in this example). Furthermore, the insertion operation takes place within an RCU read-side critical section. As we will see when we examine the `hashtab_resize()` function, this means that the first resize operation will use `synchronize_rcu()` to wait for the insertion's read-side critical section to complete.

Quick Quiz 10.11:

In the `hashtab_lookup()` function in Figure 10.27, the code carefully finds the right bucket in the new hash table if the element to be looked up has already been distributed by a concurrent resize operation. This seems wasteful for RCU-protected lookups. Why not just stick with the old hash table in this case? ■

Answer:

Suppose that a resize operation begins and distributes half of the old table's buckets to the new table. Suppose further that a thread adds a new element that goes into one of the already-distributed buckets, and that this same thread now looks up this newly added element. If lookups unconditionally traversed only the old hash table, this thread would get a lookup failure for the element that it just added, which certainly sounds like a bug to me!

Quick Quiz 10.12:

The `hashtab_del()` function in Figure 10.27 does not always remove the element from the old hash table. Doesn't this mean that readers might access this newly removed element after it has been freed? ■

Answer:

No. The `hashtab_del()` function omits removing the element from the old hash table only if the resize operation has already progressed beyond the bucket containing the just-deleted element. But this means that new `hashtab_lookup()` operations will use the new hash table when looking up that element. Therefore, only old `hashtab_lookup()` operations that started before the `hashtab_del()` might encounter the newly removed element. This means that `hashtab_del()` need only wait for an RCU grace period to avoid inconveniencing `hashtab_lookup()` operations.

D.11 Validation

Quick Quiz 10.13:

In the `hashtab_resize()` function in Figure 10.27, what guarantees that the update to `->ht_new` on line 29 will be seen as happening before the update to `->ht_resize_cur` on line 36 from the perspective of `hashtab_lookup()`, `hashtab_add()`, and `hashtab_del()`? ■

Answer:

The `synchronize_rcu()` on line 30 of Figure 10.27 ensures that all pre-existing RCU readers have completed between the time that we install the new hash-table reference on line 29 and the time that we update `->ht_resize_cur` on line 36. This means that any reader that sees a non-negative value of `->ht_resize_cur` cannot have started before the assignment to `->ht_new`, and thus must be able to see the reference to the new hash table.

Quick Quiz 11.1:

When in computing is the willingness to follow a fragmentary plan critically important? ■

Answer:

There are any number of situations, but perhaps the most important situation is when no one has ever created anything resembling the program to be developed. In this case, the only way to create a credible plan is to implement the program, create the plan, and implement it a second time. But whoever implements the program for the first time has no choice but to follow a fragmentary plan because any detailed plan created in ignorance cannot survive first contact with the real world.

And perhaps this is one reason why evolution has favored insanely optimistic human beings who are happy to follow fragmentary plans!

Quick Quiz 10.14:

Couldn't the `hashtorture.h` code be modified to accommodate a version of `hashtab_lock_mod()` that subsumes the `ht_get_bucket()` functionality? ■

Answer:

It probably could, and doing so would benefit all of the per-bucket-locked hash tables presented in this chapter. Making this modification is left as an exercise for the reader.

Quick Quiz 11.2:

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.)

Quick Quiz 10.15:

How much do these specializations really save? Are they really worth it? ■

Answer:

The answer to the first question is left as an exercise to the reader. Try specializing the resizable hash table and see how much performance improvement results. The second question cannot be answered in general, but must instead be answered with respect to a specific use case. Some use cases are extremely sensitive to performance and scalability, while others are less so.

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?
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 11.3:

You are asking me to do all this validation BS before I even start coding??? That sounds like a great way to never get started!!! ■

Answer:

If it is your project, for example, a hobby, do what you like. Any time you waste will be your own, and you have no one else to answer to for it. And there is a good chance that the time will not be completely wasted. For example, if you are embarking on a first-of-a-kind project, the requirements are in some sense unknowable anyway. In this case, the best approach might be to quickly prototype a number of rough solutions, try them out, and see what works best.

On the other hand, if you are being paid to produce a system that is broadly similar to existing systems, you owe it to your users, your employer, and your future self to validate early and often.

Quick Quiz 11.4:

How can you implement `WARN_ON_ONCE()`? ■

Answer:

If you don't mind having a `WARN_ON_ONCE()` that will sometimes warn twice or three times, simply maintain a static variable that is initialized to zero. If the condition triggers, check the static variable, and if it is non-zero, return. Otherwise, set it to one, print the message, and return.

If you really need the message to never appear more than once, perhaps because it is huge, you can use an atomic exchange operation in place of “set it to one” above. Print the message only if the atomic exchange operation returns zero.

Quick Quiz 11.5:

Why would anyone bother copying existing code in pen

on paper??? Doesn't that just increase the probability of transcription errors? ■

Answer:

If you are worried about transcription errors, please allow me to be the first to introduce you to a really cool tool named `diff`. In addition, carrying out the copying can be quite valuable:

1. If you are copying a lot of code, you are probably failing to take advantage of an opportunity for abstraction. The act of copying code can provide great motivation for abstraction.
2. Copying the code gives you an opportunity to think about whether the code really works in its new setting. Is there some non-obvious constraint, such as the need to disable interrupts or to hold some lock?
3. Copying the code also gives you time to consider whether there is some better way to get the job done.

So, yes, copy the code!

Quick Quiz 11.6:

This procedure is ridiculously over-engineered! How can you expect to get a reasonable amount of software written doing it this way??? ■

Answer:

Indeed, repeatedly copying code by hand is laborious and slow. However, when combined with heavy-duty stress testing and proofs of correctness, this approach is also extremely effective for complex parallel code where ultimate performance and reliability are required and where debugging is difficult. The Linux-kernel RCU implementation is a case in point.

On the other hand, if you are writing a simple single-threaded shell script to manipulate some data, then you would be best-served by a different methodology. For example, you might enter each command one at a time into an interactive shell with a test data set to make sure that it did what you wanted, then copy-and-paste the successful commands into your script. Finally, test the script as a whole.

If you have a friend or colleague who is willing to help out, pair programming can work very well, as can any number of formal design- and code-review processes.

And if you are writing code as a hobby, then do whatever you like.

In short, different types of software need different development methodologies.

Quick Quiz 11.7:

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 Section 12. But beware: formal validation of your code will not find errors in your assumptions, misunderstanding of the requirements, misunderstanding of the software or hardware primitives you use, or errors that you did not think to construct a proof for.

Quick Quiz 11.8:

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 11.9:

In Equation 11.6, are the logarithms base-10, base-2, or base- e ? ■

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 11.10:

Suppose that a bug causes a test failure three times per hour on average. How long must the test run error-free to provide 99.9% confidence that the fix significantly reduced the probability of failure? ■

Answer:

We set n to 3 and P to 99.9 in Equation 11.28, resulting in:

$$T = -\frac{1}{3} \log \frac{100 - 99.9}{100} = 2.3 \quad (\text{D.9})$$

If the test runs without failure for 2.3 hours, we can be 99.9% certain that the fix reduced the probability of failure.

Quick Quiz 11.11:

Doing the summation of all the factorials and exponentials is a real pain. Isn't there an easier way? ■

Answer:

One approach is to use the open-source symbolic manipulation program named “maxima”. Once you have installed this program, which is a part of many Debian-based Linux distributions, you can run it and give the `load(distrib)`; command followed by any number of `bfloat(cdf_poisson(m, 1))`; commands, where the `m` is replaced by the desired value of m and the `1` is replaced by the desired value of λ .

In particular, the `bfloat(cdf_poisson(2, 24))`; command results in `1.181617112359357b-8`, which matches the value given by Equation 11.30.

Alternatively, you can use the rough-and-ready method described in Section 11.6.2.

Quick Quiz 11.12:

But wait!!! Given that there has to be *some* number of failures (including the possibility of zero failures), shouldn't the summation shown in Equation 11.30 approach the value 1 as m goes to infinity? ■

Answer:

Indeed it should. And it does.

To see this, note that $e^{-\lambda}$ does not depend on i , which means that it can be pulled out of the summation as follows:

$$e^{-\lambda} \sum_{i=0}^{\infty} \frac{\lambda^i}{i!} \quad (\text{D.10})$$

The remaining summation is exactly the Taylor series for e^{λ} , yielding:

$$e^{-\lambda} e^{\lambda} \quad (\text{D.11})$$

The two exponentials are reciprocals, and therefore cancel, resulting in exactly 1, as required.

Quick Quiz 11.13:

How is this approach supposed to help if the corruption affected some unrelated pointer, which then caused the corruption??? ■

Answer:

Indeed, that can happen. Many CPUs have hardware-debugging facilities that can help you locate that unrelated pointer. Furthermore, if you have a core dump, you can search the core dump for pointers referencing the corrupted region of memory. You can also look at the data layout of the corruption, and check pointers whose type matches that layout.

You can also step back and test the modules making up your program more intensively, which will likely confine the corruption to the module responsible for it. If this makes the corruption vanish, consider adding additional argument checking to the functions exported from each module.

Nevertheless, this is a hard problem, which is why I used the words “a bit of a dark art”.

Quick Quiz 11.14:

But I did the bisection, and ended up with a huge commit. What do I do now? ■

Answer:

A huge commit? Shame on you! This is but one reason why you are supposed to keep the commits small.

And that is your answer: Break up the commit into bite-sized pieces and bisect the pieces. In my experience, the act of breaking up the commit is often sufficient to make the bug painfully obvious.

Quick Quiz 11.15:

Why don't existing conditional-locking primitives provide this spurious-failure functionality? ■

Answer:

There are locking algorithms that depend on conditional-locking primitives telling them the truth. For example, if conditional-lock failure signals that some other thread is already working on a given job, spurious failure might cause that job to never get done, possibly resulting in a hang.

Quick Quiz 11.16:

That is ridiculous!!! After all, isn't getting the correct answer later than one would like better than getting an incorrect answer??? ■

Answer:

This question fails to consider the option of choosing not to compute the answer at all, and in doing so, also fails to consider the costs of computing the answer. For example, consider short-term weather forecasting, for which accurate models exist, but which require large (and expensive) clustered supercomputers, at least if you want to actually run the model faster than the weather.

And in this case, any performance bug that prevents the model from running faster than the actual weather prevents any forecasting. Given that the whole purpose of purchasing the large clustered supercomputer was to forecast weather, if you cannot run the model faster than the weather, you would be better off not running the model at all.

More severe examples may be found in the area of safety-critical real-time computing.

Quick Quiz 11.17:

But if you are going to put in all the hard work of parallelizing an application, why not do it right? Why settle for anything less than optimal performance and linear scalability? ■

Answer:

Although I do heartily salute your spirit and aspirations, you are forgetting that there may be high costs due to delays in the program's completion. For an extreme example, suppose that a 40% performance shortfall from a single-threaded application is causing one person to die each day. Suppose further that in a day you could hack together a quick and dirty parallel program that ran 50% faster on an eight-CPU system than the sequential version, but that an optimal parallel program would require four months of painstaking design, coding, debugging, and tuning.

It is safe to say that more than 100 people would prefer the quick and dirty version.

Quick Quiz 11.18:

But what about other sources of error, for example, due to interactions between caches and memory layout? ■

Answer:

Changes in memory layout can indeed result in unrealistic decreases in execution time. For example, suppose that a given microbenchmark almost always overflows the L0 cache's associativity, but with just the right memory layout, it all fits. If this is a real concern, consider running your microbenchmark using huge pages (or within the kernel or on bare metal) in order to completely control the memory layout.

Quick Quiz 11.19:

Wouldn't the techniques suggested to isolate the code under test also affect that code's performance, particularly if it is running within a larger application? ■

Answer:

Indeed it might, although in most microbenchmarking efforts you would extract the code under test from the enclosing application. Nevertheless, if for some reason you must keep the code under test within the application, you will very likely need to use the techniques discussed

in Section 11.7.6.

Quick Quiz 11.20:

This approach is just plain weird! Why not use means and standard deviations, like we were taught in our statistics classes? ■

Answer:

Because mean and standard deviation were not designed to do this job. To see this, try applying mean and standard deviation to the following data set, given a 1% relative error in measurement:

```
49,548.4 49,549.4 49,550.2 49,550.9 49,550.9
49,551.0 49,551.5 49,552.1 49,899.0 49,899.3
49,899.7 49,899.8 49,900.1 49,900.4 52,244.9
53,333.3 53,333.3 53,706.3 53,706.3 54,084.5
```

The problem is that mean and standard deviation do not rest on any sort of measurement-error assumption, and they will therefore see the difference between the values near 49,500 and those near 49,900 as being statistically significant, when in fact they are well within the bounds of estimated measurement error.

Of course, it is possible to create a script similar to that in Figure 11.7 that uses standard deviation rather than absolute difference to get a similar effect, and this is left as an exercise for the interested reader. Be careful to avoid divide-by-zero errors arising from strings of identical data values!

Quick Quiz 11.21:

But what if all the y-values in the trusted group of data are exactly zero? Won't that cause the script to reject any non-zero value? ■

Answer:

Indeed it will! But if your performance measurements often produce a value of exactly zero, perhaps you need to take a closer look at your performance-measurement code.

Note that many approaches based on mean and standard deviation will have similar problems with this sort of dataset.

D.12 Formal Verification

Quick Quiz 12.1:

Why is there an unreached statement in locker? After all, isn't this a *full* state-space search? ■

Answer:

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 12.2:

What are some Promela code-style issues with this example? ■

Answer:

There are several:

1. The declaration of `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 `sum` and `j`, and also to cover the assertion. This also reduces the state space (again, by how much?).

Quick Quiz 12.3:

Is there a more straightforward way to code the `do-od` statement? ■

Answer:

Yes. Replace it with `if- fi` and remove the two `break` statements.

Quick Quiz 12.4:

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 12.5:

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 `ctr[0]` is zero and the value of `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 `ctr[0]`, which is zero.
4. The first updater adds one to `ctr[0]`, flips the index (which now becomes zero), then subtracts one from `ctr[1]` (which now becomes one).
5. The second updater fetches the value of `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 12.6:

Given that we have two independent proofs of correctness for the QRCU algorithm described herein, and given that the proof of incorrectness covers what is likely a different algorithm, why is there any room for doubt? ■

Answer:

There is always room for doubt. In this case, it is important to keep in mind that the two proofs of correctness preceded the formalization of real-world memory models, raising the possibility that these two proofs are based on

incorrect memory-ordering assumptions. Furthermore, since both proofs were constructed by the same person, it is quite possible that they contain a common error. Again, there is always room for doubt.

Quick Quiz 12.7:

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.
5. Wait for memory sizes of affordable systems to expand to fit your problem.
6. Use one of a number of cloud-computing services to rent a large system for a short time period.

Quick Quiz 12.8:

Why not simply increment `rcu_update_flag`, and then only increment `dynticks_progress_counter` if the old value of `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 12.9:

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 12.10:

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 12.11:

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 12.13 on page 209.

Quick Quiz 12.12:

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 12.13:

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 12.14:

Given there are a pair of back-to-back changes to `gp_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 12.15:

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 statement in its own `EXECUTE_MAINLINE()` statement.

Quick Quiz 12.16:

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 12.17:

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 12.18:

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 12.19:

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 12.20:

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 12.21:

This is still pretty complicated. Why not just have a `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.

Quick Quiz 12.22:

But x86 has strong memory ordering! Why would you need to formalize its memory model? ■

Answer:

Actually, academics consider the x86 memory model to be weak because it can allow prior stores to be reordered with subsequent loads. From an academic viewpoint, a strong memory model is one that allows absolutely no reordering, so that all threads agree on the order of all operations visible to them.

Quick Quiz 12.23:

Why does line 8 of Figure 12.25 initialize the registers? Why not instead initialize them on lines 4 and 5? ■

Answer:

Either way works. However, in general, it is better to use initialization than explicit instructions. The explicit instructions are used in this example to demonstrate their use. In addition, many of the litmus tests available on the tool’s web site (<http://www.cl.cam.ac.uk/~pes20/ppcmem/>) were automatically generated, which generates explicit initialization instructions.

Quick Quiz 12.24:

But whatever happened to line 17 of Figure 12.25, the one that is the Fail: label? ■

Answer:

The implementation of powerpc version of `atomic_add_return()` loops when the `stwcx` instruction fails, which it communicates by setting non-zero status in the condition-code register, which in turn is tested by the `bne` instruction. Because actually modeling the loop would result in state-space explosion, we instead branch to the `Fail:` label, terminating the model with the initial value of 2 in thread 1’s `r3` register, which will not trigger the `exists` assertion.

There is some debate about whether this trick is universally applicable, but I have not seen an example where it

fails.

Quick Quiz 12.25:

Does the ARM Linux kernel have a similar bug? ■

Answer:

ARM does not have this particular bug because that it places `smp_mb()` before and after the `atomic_add_return()` function’s assembly-language implementation. PowerPC no longer has this bug; it has long since been fixed. Finding any other bugs that the Linux kernel might have is left as an exercise for the reader.

Quick Quiz 12.26:

In light of the full verification of the L4 microkernel, isn’t this limited view of formal verification just a little bit obsolete? ■

Answer:

Unfortunately, no.

The full verification of the L4 microkernel was a tour de force, with a large number of Ph.D. students hand-verifying code at a very slow per-student rate. This level of effort could not be applied to most software projects because the rate of change is just too great. Furthermore, although the L4 microkernel is a large software artifact from the viewpoint of formal verification, it is tiny compared to the a great number of projects, including LLVM, gcc, the Linux kernel, Hadoop, MongoDB, and a great many others.

Although formal verification is finally starting to show some promise, it currently has no chance of completely displacing testing in the foreseeable future. And although I would dearly love to be proven wrong on this point, please note that such a proof will be in the form of a real tool that verifies real software, not in the form of a large body of rousing rhetoric.

D.13 Putting It All Together

Quick Quiz 13.1:

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 13.2:

Just what is the accuracy of `read_count()`, anyway?

■

Answer:

Refer to Figure 5.9 on Page 41. 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:

$$\epsilon = \frac{T_r R_i}{2} \quad (\text{D.12})$$

where ϵ 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 13.3:

Hey!!! Line 45 of Figure 13.1 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 13.4:

Wow! Figure 13.1 contains 69 lines of code, compared to only 42 in Figure 5.9. 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 5.9 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 5.9, but with all the scalability and performance benefits of the implementation shown in Figure 13.1!

```

1 struct measurement {
2     double meas_1;
3     double meas_2;
4     double meas_3;
5 };
6
7 struct animal {
8     char name[40];
9     double age;
10    struct measurement *mp;
11    struct measurement meas;
12    char photo[0]; /* large bitmap. */
13 };

```

Figure D.10: Localized Correlated Measurement Fields

Quick Quiz 13.5:

But can't the approach shown in Figure 13.5 result in extra cache misses, in turn resulting in additional read-side overhead? ■

Answer:

Indeed it can.

One way to avoid this cache-miss overhead is shown in Figure D.10: Simply embed an instance of a measurement structure named `meas` into the `animal` structure, and point the `->mp` field at this `->meas` field.

Measurement updates can then be carried out as follows:

1. Allocate a new measurement structure and place the new measurements into it.
2. Use `rcu_assign_pointer()` to point `->mp` to this new structure.
3. Wait for a grace period to elapse, for example using either `synchronize_rcu()` or `call_rcu()`.
4. Copy the measurements from the new measurement structure into the embedded `->meas` field.
5. Use `rcu_assign_pointer()` to point `->mp` back to the old embedded `->meas` field.
6. After another grace period elapses, free up the new measurement field.

This approach uses a heavier weight update procedure to eliminate the extra cache miss in the common case. The extra cache miss will be incurred only while an update is actually in progress.

Quick Quiz 13.6:

But how does this scan work while a resizable hash table is being resized? In that case, neither the old nor the new hash table is guaranteed to contain all the elements in the hash table! ■

Answer:

True, resizable hash tables as described in Section 10.4 cannot be fully scanned while being resized. One simple way around this is to acquire the `hashtab` structure's `->ht_lock` while scanning, but this prevents more than one scan from proceeding concurrently.

Another approach is for updates to mutate the old hash table as well as the new one while resizing is in progress. This would allow scans to find all elements in the old hash table. Implementing this is left as an exercise for the reader.

D.14 Advanced Synchronization

Quick Quiz 14.1:

How on earth could the assertion on line 21 of the code in Figure 14.3 on page 243 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 14.2:

Great... So how do I fix it? ■

Answer:

The easiest fix is to replace each of the `barrier()`s on line 12 and line 20 with an `smp_mb()`.

Of course, some hardware is more forgiving than other hardware. For example, on x86 the assertion on line 21 of Figure 14.3 on page 243 cannot trigger. On PowerPC, only the `barrier()` on line 20 need be replaced with `smp_mb()` to prevent the assertion from triggering.

Quick Quiz 14.3:

What assumption is the code fragment in Figure 14.4

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 14.4:

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 14.5:

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 14.6:

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, as discussed in Section 14.2.8.

Quick Quiz 14.7:

How do we know that modern hardware guarantees that at least one of the loads will see the value stored by the other thread in the ears-to-mouths scenario? ■

Answer:

The scenario is as follows, with A and B both initially zero:

```
CPU 0: A=1; smp_mb(); r1=B;
CPU 1: B=1; smp_mb(); r2=A;
```

If neither of the loads see the corresponding store, when both CPUs finish, both r_1 and r_2 will be equal to zero. Let's suppose that r_1 is equal to zero. Then we know that CPU 0's load from B happened before CPU 1's store to B: After all, we would have had r_1 equal to one otherwise. But given that CPU 0's load from B happened before CPU 1's store to B, memory-barrier pairing guarantees that CPU 0's store to A happens before CPU 1's load from A, which in turn guarantees that r_2 will be equal to one, not zero.

Therefore, at least one of r_1 and r_2 must be nonzero, which means that at least one of the loads saw the value from the corresponding store, as claimed.

Quick Quiz 14.8:

How can the other “Only one store” entries in Table 14.2 be used? ■

Answer:

For combination 2, if CPU 1's load from B sees a value prior to CPU 2's store to B, then we know that CPU 2's load from A will return the same value as CPU 1's load from A, or some later value.

For combination 4, if CPU 2's load from B sees the value from CPU 1's store to B, then we know that CPU 2's load from A will return the same value as CPU 1's load from A, or some later value.

For combination 8, if CPU 2's load from A sees CPU 1's store to A, then we know that CPU 1's load from B will return the same value as CPU 2's load from A, or some later value.

Quick Quiz 14.9:

How could the assertion $b==2$ on page 248 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 14.10:

How could the code on page 248 possibly leak memory?

■

Answer:

Only the first execution of the critical section should see $p==NULL$. 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 $p==NULL$, 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 14.11:

How could the code on page 248 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 14.12:

What effect does the following sequence have on the order of stores to variables “a” and “b”?

```
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 14.13:

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 14.14:

What (if any) CPUs have memory-barrier instructions from which these semi-permeable locking primitives might be constructed? ■

Answer:

Itanium is one example. The identification of any others is left as an exercise for the reader.

Quick Quiz 14.15:

Given that operations grouped in curly braces are executed concurrently, which of the rows of Table 14.3 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 14.16:

What are the constraints for Table 14.4? ■

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.

D.15 Parallel Real-Time Computing

Quick Quiz 15.1:

But what about battery-powered systems? They don’t require energy flowing into the system as a whole. ■

Answer:

Sooner or later, either the battery must be recharged, which requires energy to flow into the system, or the system will stop operating.

Quick Quiz 15.2:

But given the results from queueing theory, won’t low utilization merely improve the average response time rather than improving the worst-case response time, which is the only response time that many real-time systems care about? ■

Answer:

It depends. One situation where the worst-case response time is improved by lowering utilization is where there is only one real-time thread using the device in question, but where all the threads use the device in question. Restricting use of that device to the single real-time thread eliminates queueing delays, at least assuming that the real-time thread refrains from overdriving that device.

Quick Quiz 15.3:

Formal verification is already quite capable, benefiting from decades of intensive study. Are additional advances *really* required, or is this just a practitioner’s excuse to continue to be lazy and ignore the awesome power of formal verification? ■

Answer:

Perhaps this situation is just a theoretician’s excuse to avoid diving into the messy world of real software? Perhaps more constructively, the following advances are required:

1. Formal verification needs to handle larger software artifacts. The largest verification efforts have been for systems of only about 10,000 lines of code, and those have been verifying much simpler properties than real-time latencies.
2. Hardware vendors will need to publish formal timing guarantees. This used to be common practice back when hardware was much simpler, but today’s complex hardware results in excessively complex expressions for worst-case performance. Unfortunately, energy-efficiency concerns are pushing vendors in the direction of even more complexity.
3. Timing analysis needs to be integrated into development methodologies and IDEs.

All that said, there is hope, given recent work formalizing the memory models of real computer systems [AMP¹¹, AKNT13].

Quick Quiz 15.4:

Differentiating real-time from non-real-time based on what can “be achieved straightforwardly by non-real-time systems and applications” is a travesty! There is absolutely no theoretical basis for such a distinction!!! Can’t we do better than that??? ■

Answer:

This distinction is admittedly unsatisfying from a strictly theoretical perspective. But on the other hand, it is exactly what the developer needs in order to decide whether the application can be cheaply and easily developed using standard non-real-time approaches, or whether the more difficult and expensive real-time approaches are required. In other words, theory is quite important, however, for those of us who like to get things done, theory supports practice, never the other way around.

Quick Quiz 15.5:

But if you only allow one reader at a time to read-acquire a reader-writer lock, isn’t that the same as an exclusive lock??? ■

Answer:

Indeed it is, other than the API. And the API is important because it allows the Linux kernel to offer real-time capabilities without having the -rt patchset grow to ridiculous sizes.

However, this approach clearly and severely limits read-side scalability. The Linux kernel’s -rt patchset has been able to live with this limitation for several reasons: (1) Real-time systems have traditionally been relatively small, (2) Real-time systems have generally focused on process control, thus being unaffected by scalability limitations in the I/O subsystems, and (3) Many of the Linux kernel’s reader-writer locks have been converted to RCU.

All that aside, it is quite possible that the Linux kernel will some day permit limited read-side parallelism for reader-writer locks subject to priority boosting.

Quick Quiz 15.6:

Suppose that preemption occurs just after the load from `t->rcu_read_unlock_special.s` on line 17 of Figure 15.15. Mightn’t that result in the task failing to invoke `rcu_read_unlock_special()`, thus

failing to remove itself from the list of tasks blocking the current grace period, in turn causing that grace period to extend indefinitely? ■

Answer:

That is a real problem, and it is solved in RCU’s scheduler hook. If that scheduler hook sees that the value of `t->rcu_read_lock_nesting` is negative, it invokes `rcu_read_unlock_special()` if needed before allowing the context switch to complete.

Quick Quiz 15.7:

But isn’t correct operation despite fail-stop bugs a valuable fault-tolerance property? ■

Answer:

Yes and no.

Yes in that non-blocking algorithms can provide fault tolerance in the face of fail-stop bugs, but no in that this is grossly insufficient for practical fault tolerance. For example, suppose you had a wait-free queue, and further suppose that a thread has just dequeued an element. If that thread now succumbs to a fail-stop bug, the element it has just dequeued is effectively lost. True fault tolerance requires way more than mere non-blocking properties, and is beyond the scope of this book.

Quick Quiz 15.8:

I couldn’t help but spot the word “includes” before this list. Are there other constraints? ■

Answer:

Indeed there are, and lots of them. However, they tend to be specific to a given situation, and many of them can be thought of as refinements of some of the constraints listed above. For example, the many constraints on choices of data structure will help meeting the “Bounded time spent in any given critical section” constraint.

Quick Quiz 15.9:

Given that real-time systems are often used for safety-critical applications, and given that runtime memory allocation is forbidden in many safety-critical situations, what is with the call to `malloc()` ??? ■

Answer:

In early 2016, situations forbidding runtime memory were also not so excited with multithreaded computing. So the runtime memory allocation is not an additional obstacle to safety criticality.

Quick Quiz 15.10:

Don't you need some kind of synchronization to protect `update_cal()`? ■

Answer:

Indeed you do, and you could use any of a number of techniques discussed earlier in this book.

D.16 Ease of Use

Quick Quiz 16.1:

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 16.2:

Yech! 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 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 16.3:

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.

D.17 Conflicting Visions of the Future

Quick Quiz 17.1:

What about non-persistent primitives represented by data structures in `mmap()` regions of memory? What happens when there is an `exec()` within a critical section of such a primitive? ■

Answer:

If the `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.

Quick Quiz 17.2:

Why would it matter that oft-written variables shared the cache line with the lock variable? ■

Answer:

If the lock is in the same cacheline as some of the variables that it is protecting, then writes to those variables by one CPU will invalidate that cache line for all the other CPUs. These invalidations will generate large numbers of conflicts and retries, perhaps even degrading performance and scalability compared to locking.

Quick Quiz 17.3:

Why are relatively small updates important to HTM performance and scalability? ■

Answer:

The larger the updates, the greater the probability of conflict, and thus the greater probability of retries, which degrade performance.

Quick Quiz 17.4:

How could a red-black tree possibly efficiently enumerate all elements of the tree regardless of choice of synchronization mechanism??? ■

Answer:

In many cases, the enumeration need not be exact. In these cases, hazard pointers or RCU may be used to protect readers with low probability of conflict with any given insertion or deletion.

Quick Quiz 17.5:

But why can't a debugger emulate single stepping by setting breakpoints at successive lines of the transaction, relying on the retry to retrace the steps of the earlier instances of the transaction? ■

Answer:

This scheme might work with reasonably high probability, but it can fail in ways that would be quite surprising to most users. To see this, consider the following transaction:

```

1 begin_trans();
2 if (a) {
3   do_one_thing();
4   do_another_thing();
5 } else {
6   do_a_third_thing();
7   do_a_fourth_thing();
8 }
9 end_trans();

```

Suppose that the user sets a breakpoint at line 3, which triggers, aborting the transaction and entering the debugger. Suppose that between the time that the breakpoint triggers and the debugger gets around to stopping all the threads, some other thread sets the value of `a` to zero.

When the poor user attempts to single-step the program, surprise! The program is now in the `else`-clause instead of the `then`-clause.

This is *not* what I call an easy-to-use debugger.

Quick Quiz 17.6:

But why would *anyone* need an empty lock-based critical section??? ■

Answer:

See the answer to the Quick Quiz in Section 7.2.1.

However, it is claimed that given a strongly atomic HTM implementation without forward-progress guarantees, any memory-based locking design based on empty critical sections will operate correctly in the presence of transactional lock elision. Although I have not seen a proof of this statement, there is a straightforward rationale for this claim. The main idea is that in a strongly atomic HTM implementation, the results of a given transaction are not visible until after the transaction completes successfully. Therefore, if you can see that a transaction has started, it is guaranteed to have already completed, which means that a subsequent empty lock-based critical section will successfully “wait” on it—after all, there is no waiting required.

This line of reasoning does not apply to weakly atomic systems (including many STM implementation), and it also does not apply to lock-based programs that use means other than memory to communicate. One such means is the passage of time (for example, in hard real-time systems) or flow of priority (for example, in soft real-time systems).

Locking designs that rely on priority boosting are of particular interest.

Quick Quiz 17.7:

Can't transactional lock elision trivially handle locking's time-based messaging semantics by simply choosing not to elide empty lock-based critical sections? ■

Answer:

It could do so, but this would be both unnecessary and insufficient.

It would be unnecessary in cases where the empty critical section was due to conditional compilation. Here, it might well be that the only purpose of the lock was to protect data, so eliding it completely would be the right

thing to do. In fact, leaving the empty lock-based critical section would degrade performance and scalability.

On the other hand, it is possible for a non-empty lock-based critical section to be relying on both the data-protection and time-based and messaging semantics of locking. Using transactional lock elision in such a case would be incorrect, and would result in bugs.

Quick Quiz 17.8:

Given modern hardware [MOZ09], how can anyone possibly expect parallel software relying on timing to work? ■

Answer:

The short answer is that on commonplace commodity hardware, synchronization designs based on any sort of fine-grained timing are foolhardy and cannot be expected to operate correctly under all conditions.

That said, there are systems designed for hard real-time use that are much more deterministic. In the (very unlikely) event that you are using such a system, here is a toy example showing how time-based synchronization can work. Again, do *not* try this on commodity microprocessors, as they have highly nondeterministic performance characteristics.

This example uses multiple worker threads along with a control thread. Each worker thread corresponds to an outbound data feed, and records the current time (for example, from the `clock_gettime()` system call) in a per-thread `my_timestamp` variable after executing each unit of work. The real-time nature of this example results in the following set of constraints:

1. It is a fatal error for a given worker thread to fail to update its timestamp for a time period of more than `MAX_LOOP_TIME`.
2. Locks are used sparingly to access and update global state. item Locks are granted in strict FIFO order within a given thread priority.

When worker threads complete their feed, they must disentangle themselves from the rest of the application and place a status value in a per-thread `my_status` variable that is initialized to -1. Threads do not exit; they instead are placed on a thread pool to accommodate later processing requirements. The control thread assigns (and re-assigns) worker threads as needed, and also maintains a histogram of thread statuses. The control thread runs

at a real-time priority no higher than that of the worker threads.

Worker threads' code is as follows:

```

1  int my_status = -1; /* Thread local. */
2
3  while (continue_working()) {
4      enqueue_any_new_work();
5      wp = dequeue_work();
6      do_work(wp);
7      my_timestamp = clock_gettime(...);
8  }
9
10 acquire_lock(&departing_thread_lock);
11 /*
12  * Disentangle from application, might
13  * acquire other locks, can take much longer
14  * than MAX_LOOP_TIME, especially if many
15  * threads exit concurrently.
16  */
17 my_status = get_return_status();
18 release_lock(&departing_thread_lock);
19
20 /* thread awaits repurposing. */
21

```

The control thread's code is as follows:

```

1  for (;;) {
2      for_each_thread(t) {
3          ct = clock_gettime(...);
4          d = ct - per_thread(my_timestamp, t);
5          if (d >= MAX_LOOP_TIME) {
6              /* thread departing. */
7              acquire_lock(&departing_thread_lock);
8              release_lock(&departing_thread_lock);
9              i = per_thread(my_status, t);
10             status_hist[i]++;
11             /* Bug if TLE! */
12         }
13     /* Repurpose threads as needed. */
14 }

```

Line 5 uses the passage of time to deduce that the thread has exited, executing lines 6-10 if so. The empty lock-based critical section on lines 7 and 8 guarantees that any thread in the process of exiting completes (remember that locks are granted in FIFO order!).

Once again, do not try this sort of thing on commodity microprocessors. After all, it is difficult enough to get right on systems specifically designed for hard real-time use!

Quick Quiz 17.9:

But the `boostee()` function in Figure 17.12 alternatively acquires its locks in reverse order! Won't this result in deadlock? ■

Answer:

No deadlock will result. To arrive at deadlock, two different threads must each acquire the two locks in

opposite orders, which does not happen in this example. However, deadlock detectors such as lockdep [Cor06a] will flag this as a false positive.

Quick Quiz 17.10:

So a bunch of people set out to supplant locking, and they mostly end up just optimizing locking??? ■

Answer:

At least they accomplished something useful! And perhaps there will be additional HTM progress over time.

D.18 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.
2. Missing Memory barriers on update side.
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).

Quick Quiz A.3:

Suppose a portion of a program uses RCU read-side primitives as its only synchronization mechanism. Is this parallelism or concurrency? ■

Answer:

Yes.

Quick Quiz A.4:

In what part of the second (scheduler-based) perspective would the lock-based single-thread-per-CPU workload be considered “concurrent”? ■

Answer:

The people who would like to arbitrarily subdivide and interleave the workload. Of course, an arbitrary subdivision might end up separating a lock acquisition from the corresponding lock release, which would prevent any other thread from acquiring that lock. If the locks were pure spinlocks, this could even result in deadlock.

D.19 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!

D.20 Why Memory Barriers?

Quick Quiz C.1:

Where does a writeback message originate from and where does it go to? ■

Answer:

The writeback message originates from a given CPU, or in some designs from a given level of a given CPU's cache—or even from a cache that might be shared among several CPUs. The key point is that a given cache does

not have room for a given data item, so some other piece of data must be ejected from the cache to make room. If there is some other piece of data that is duplicated in some other cache or in memory, then that piece of data may be simply discarded, with no writeback message required.

On the other hand, if every piece of data that might be ejected has been modified so that the only up-to-date copy is in this cache, then one of those data items must be copied somewhere else. This copy operation is undertaken using a “writeback message”.

The destination of the writeback message has to be something that is able to store the new value. This might be main memory, but it also might be some other cache. If it is a cache, it is normally a higher-level cache for the same CPU, for example, a level-1 cache might write back to a level-2 cache. However, some hardware designs permit cross-CPU writebacks, so that CPU 0's cache might send a writeback message to CPU 1. This would normally be done if CPU 1 had somehow indicated an interest in the data, for example, by having recently issued a read request.

In short, a writeback message is sent from some part of the system that is short of space, and is received by some other part of the system that can accommodate the data.

Quick Quiz C.2:

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.3:

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.4:

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.5:

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.6:

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.7:

But if the main purpose of store buffers is to hide acknowledgement latencies in multiprocessor cache-coherence protocols, why do uniprocessors also have store buffers? ■

Answer:

Because the purpose of store buffers is not just to hide acknowledgement latencies in multiprocessor cache-coherence protocols, but to hide memory latencies in general. Because memory is much slower than is cache on uniprocessors, store buffers on uniprocessors can help to hide write-miss latencies.

Quick Quiz C.8:

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.9:

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 CPUs to discard their copies of this cache line. An “invalidate” message therefore suffices.

Quick Quiz C.10:

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. Furthermore, compilers normally assume that only the currently executing thread is updating the variables, and this assumption allows the compiler to hoist the load of `a` to precede the loop.

In fact, some compilers would transform the loop to a branch around an infinite loop as follows:

```

1 void foo(void)
2 {
3     a = 1;
4     smp_mb();
5     b = 1;
6 }
7
8 void bar(void)
9 {
10    if (b == 0)
11        for (;;)
12            continue;
13    smp_mb();
14    assert(a == 1);
15 }
```

Given this optimization, the assertion could clearly fire. You should use volatile casts or (where available) C++ relaxed atomics to prevent the compiler from optimizing your parallel code into oblivion.

In short, both compilers and CPUs are quite aggressive about optimizing, so you must clearly communicate your constraints to them, using compiler directives and memory barriers.

Quick Quiz C.11:

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.12:

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.13:

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.14:

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.15:

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:

```
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`.

Appendix E

Glossary and Bibliography

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. See Section 14.2.4.2 for more information.

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 set¹ 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 threads 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.

Embarrassingly Parallel: A problem or algorithm where adding threads does not significantly increase the overall cost of the computation, resulting in linear speedups as threads are added (assuming sufficient CPUs are available).

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.

¹ In hardware-cache terminology, the word "set" is used in the same way that the word "bucket" is used when discussing software caches.

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 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.

Heisenbug: A timing-sensitive bug that disappears from sight when you add print statements or tracing in an attempt to track it down.

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.

Humiliatingly Parallel: A problem or algorithm where adding threads significantly *decreases* the overall cost of the computation, resulting in large superlinear speedups as threads are added (assuming sufficient CPUs are available).

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 can-

not 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.

Teachable: A topic, concept, method, or mechanism that the teacher understands completely and is therefore comfortable teaching.

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 (hardware transactional memory, or HTM), in software (software transactional memory, or STM), or in a combination of hardware and software (“unbounded” transactional memory, or UTM).

Unteachable: A topic, concept, method, or mechanism that the teacher does not understand well is therefore uncomfortable teaching.

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

- [AAKL06] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, and Charles E. Leiserson. Unbounded transactional memory. *IEEE Micro*, pages 59–69, January–February 2006. Available: <http://www.cag.csail.mit.edu/scale/papers/utm-ieeeemicro2006.pdf> [Viewed December 21, 2006].
- [AB13] Samy Al Bahra. Nonblocking algorithms and scalable multicore programming. *Commun. ACM*, 56(7):50–61, July 2013.
- [ABD⁺97] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevenne, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 1–14, New York, NY, October 1997.
- [aCB08] University at California Berkeley. SETI@HOME. Available: <http://setiathome.berkeley.edu/> [Viewed January 31, 2008], December 2008.
- [ACHS13] Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. Are lock-free concurrent algorithms practically wait-free? ArXiv:1311.3200v2, December 2013.
- [ACMS03] Andrea Arcangeli, Mingming Cao, Paul E. McKenney, and Dipankar Sarma. Using read-copy update techniques for System V IPC in the Linux 2.5 kernel. In *Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track)*, pages 297–310. USENIX Association, June 2003.
- [Ada11] Andrew Adamatzky. Slime mould solves maze in one pass . . . assisted by gradient of chemo-attractants. <http://arxiv.org/abs/1108.4956>, August 2011.
- [Adv02] Advanced Micro Devices. *AMD x86-64 Architecture Programmer’s Manual Volumes 1-5*, 2002.
- [Adv07] Advanced Micro Devices. *AMD x86-64 Architecture Programmer’s Manual Volume 2: System Programming*, 2007.
- [AGH⁺11a] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 487–498, New York, NY, USA, 2011. ACM.
- [AGH⁺11b] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. *SIGPLAN Not.*, 46(1):487–498, January 2011.
- [AHS⁺03] J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. M. Da Silva, O. Krieger, M. A. Auslander, D. J. Edelsohn, B. Gamsa, G. R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis. Enabling autonomic behavior in systems software with hot swapping. *IBM Systems Journal*, 42(1):60–76, January 2003.
- [AKNT13] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschig. Software verification for weak memory via program

- transformation. In *Proceedings of the 22nd European conference on Programming Languages and Systems*, ESOP'13, pages 512–532, Berlin, Heidelberg, 2013. Springer-Verlag.
- [AKT13] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient Bounded Model Checking of concurrent software. In *Computer Aided Verification (CAV)*, volume 8044 of *LNCS*, pages 141–157. Springer, 2013.
- [Ale79] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, New York, 1979.
- [Alg13] Jade Alglave. Weakness is a virtue. In *(EC)² 2013: 6th International Workshop on Exploiting Concurrency Efficiently and Correctly*, page 3, 2013.
- [Amd67] Gene Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, pages 483–485, Washington, DC, USA, 1967. IEEE Computer Society.
- [AMP⁺11] Jade Alglave, Luc Maranget, Pankaj Pawan, Susmit Sarkar, Peter Sewell, Derek Williams, and Francesco Zappa Nardelli. PPCMEM/ARMMEM: A tool for exploring the POWER and ARM memory models. <http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/pldi105-sarkar.pdf>, June 2011.
- [AMT14] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data-mining for weak memory. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 40–40, New York, NY, USA, 2014. ACM.
- [And90] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.
- [And91] Gregory R. Andrews. *Concurrent Programming, Principles, and Practices*. Benjamin Cummins, 1991.
- [ARM10] ARM Limited. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R Edition*, 2010.
- [ATC⁺11] Ege Akpinar, Sasa Tomic, Adrian Cristal, Osman Unsal, and Mateo Valero. A comprehensive study of conflict resolution policies in hardware transactional memory. In *TRANSACT 2011*. ACM SIGPLAN, June 2011.
- [ATS09] Ali-Reza Adl-Tabatabai and Tatiana Shpeisman. Draft specification of transactional language constructs for c++. <http://research.sun.com/scalable/pubs/C++-transactional-constructs-1.0.pdf>, August 2009.
- [BA01] Jeff Bonwick and Jonathan Adams. Magazines and vmem: Extending the slab allocator to many CPUs and arbitrary resources. In *USENIX Annual Technical Conference, General Track 2001*, pages 15–33, 2001.
- [Bec11] Pete Becker. Working draft, standard for programming language C++. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf>, February 2011.
- [BG87] D. Bertsekas and R. Gallager. *Data Networks*. Prentice-Hall, Inc., 1987.
- [BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company, 1987.
- [BHS07] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. Wiley, Chichester, West Sussex, England, 2007.
- [BK85] Bob Beck and Bob Kasten. VLSI assist in building a multiprocessor UNIX system. In *USENIX Conference Proceedings*, pages 255–275, Portland, OR, June 1985. USENIX Association.

- [BLM05] C. Blundell, E. C. Lewis, and M. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, June 2005. Available: http://www.cis.upenn.edu/acg/papers/wddd05_atomic_semantics.pdf [Viewed June 4, 2009].
- [BLM06] C. Blundell, E. C. Lewis, and M. Martin. Subtleties of transactional memory and atomicity semantics. *Computer Architecture Letters*, 5(2), 2006. Available: http://www.cis.upenn.edu/acg/papers/cal06_atomic_semantics.pdf [Viewed June 4, 2009].
- [BMMM05] Luke Browning, Thomas Mathews, Paul E. McKenney, and James Moody. Apparatus, method, and computer program product for converting simple locks in a multiprocessor system. Technical Report US Patent 6,842,809, US Patent and Trademark Office, Washington, DC, January 2005.
- [BMP08] R. F. Berry, P. E. McKenney, and F. N. Parr. Responsive systems: An introduction. *IBM Systems Journal*, 47(2):197–206, April 2008. Available: <http://www.research.ibm.com/journal/sj/472/berry.pdf> [Viewed May 8, 2008].
- [Boe09] Hans-J. Boehm. Transactional memory should be an implementation technique, not a programming interface. In *HOTPAR 2009*, page 6, Berkeley, CA, USA, March 2009. Available: http://www.usenix.org/event/hotpar09/tech/full_papers/boehm/boehm.pdf [Viewed May 24, 2009].
- [Boh01] Kristoffer Bohmann. Response time still matters. Available: http://www.bohmann.dk/articles/response_time_still_matters.html [Viewed July 23, 2007], July 2001.
- [Bow06] Maggie Bowman. Dividing the sheep from the goats. [url=http://www.cs.kent.ac.uk/news/2006/RBornat/](http://www.cs.kent.ac.uk/news/2006/RBornat/), February 2006.
- [Bra07] Reg Braithwaite. Don't overthink fizzbuzz. [url=http://weblog.raganwald.com/2007/01/dont-overthink-fizzbuzz.html](http://weblog.raganwald.com/2007/01/dont-overthink-fizzbuzz.html), January 2007.
- [Bra11] Björn Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.
- [But97] David Butenhof. *Programming with POSIX Threads*. Addison-Wesley, Boston, MA, USA, 1997.
- [BW14] Silas Boyd-Wickizer. *Optimizing Communications Bottlenecks in Multiprocessor Operating Systems Kernels*. PhD thesis, Massachusetts Institute of Technology, 2014.
- [CBM⁺08] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *ACM Queue*, September 2008.
- [CHP71] P. J. Courtois, F. Heymans, and D. L. Parناس. Concurrent control with “readers” and “writers”. *Communications of the ACM*, 14(10):667–668, October 1971.
- [CKZ12] Austin Clements, Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using RCU balanced trees. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*, pages 199–210, London, UK, March 2012. ACM.
- [CKZ⁺13] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP ’13*, pages 1–17, New York, NY, USA, 2013. ACM.

- [Cli09] Cliff Click. And now some hardware transactional memory comments... <http://www.azulsystems.com/blog/cliff-click/2009-02-25-and-now-some-hardware-transactional-memory-comments>, February 2009.
- [CLRS01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT electrical engineering and computer science series. MIT Press, 2001.
- [Com01] Compaq Computer Corporation. Shared memory, threads, interprocess communication. Available: http://h71000.www7.hp.com/wizard/wiz_2637.html, August 2001.
- [Cor04a] Jonathan Corbet. Approaches to realtime Linux. Available: <http://lwn.net/Articles/106010/> [Viewed March 25, 2008], October 2004.
- [Cor04b] Jonathan Corbet. Finding kernel problems automatically. <http://lwn.net/Articles/87538/>, June 2004.
- [Cor04c] Jonathan Corbet. Realtime preemption, part 2. Available: <http://lwn.net/Articles/107269/> [Viewed March 25, 2008], October 2004.
- [Cor06a] Jonathan Corbet. The kernel lock validator. Available: <http://lwn.net/Articles/185666/> [Viewed: March 26, 2010], May 2006.
- [Cor06b] Jonathan Corbet. Priority inheritance in the kernel. Available: <http://lwn.net/Articles/178253/> [Viewed June 29, 2009], April 2006.
- [Cor12] Jon Corbet. ACCESS_ONCE(). <http://lwn.net/Articles/508991/>, August 2012.
- [Cor13] Jonathan Corbet. (nearly) full tickless operation in 3.10. <http://lwn.net/Articles/549580/>, May 2013.
- [Cra93] Travis Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical Report 93-02-02, University of Washington, Seattle, Washington, February 1993.
- [CRKH05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, Inc., third edition, 2005.
- [CSG99] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: a Hardware/Software Approach*. Morgan Kaufman, 1999.
- [Dat82] C. J. Date. *An Introduction to Database Systems*, volume 1. Addison-Wesley Publishing Company, 1982.
- [DBA09] Saeed Dehnadi, Richard Bornat, and Ray Adams. Meta-analysis of the effect of consistency on success in early learning of programming. In *PPIG 2009*, pages 1–13, University of Limerick, Ireland, June 2009. Psychology of Programming Interest Group.
- [DCW⁺11] Luke Dalessandro, Francois Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS '11, pages ???–???, New York, NY, USA, 2011. ACM.
- [DD09] Mathieu Desnoyers and Michel R. Dagenais. Lttng, filling the gap between kernel instrumentation and a widely usable kernel tracer. Available: http://events.linuxfoundation.org/slides/lfcs09_desnoyers_paper.pdf [Viewed: August 28, 2011], April 2009.
- [Des09] Mathieu Desnoyers. [RFC git tree] userspace RCU (urcu) for Linux. <http://liburcu.org>, February 2009.

- [DFGG11] Aleksandar Dragovejic, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a research toy. *Communications of the ACM*, pages 70–77, April 2011.
- [DHJ⁺07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon’s highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.
- [DHK12] Vijay D’Silva, Leopold Haller, and Daniel Kroening. Satisfiability solvers are static analyzers. In *Static Analysis Symposium (SAS)*, volume 7460 of *LNCS*, pages 317–333. Springer, 2012.
- [DHL⁺08] Dave Dice, Maurice Herlihy, Doug Lea, Yossi Lev, Victor Luchangco, Wayne Mesard, Mark Moir, Kevin Moore, and Dan Nussbaum. Applications of the adaptive transactional memory test platform. In *3rd ACM SIGPLAN Workshop on Transactional Computing*, pages 1–10, Salt Lake City, UT, USA, February 2008.
- [Dig89] Digital Systems Research Center. *An Introduction to Programming with Threads*, January 1989.
- [Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, Sept 1965.
- [Dij68] Edsger W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Commun. ACM*, 11(3):147–148, March 1968.
- [Dij71] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971. Available: <http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF> [Viewed January 13, 2008].
- [DLM⁺10] Dave Dice, Yossi Lev, Virendra J. Marathe, Mark Moir, Dan Nussbaum, and Marek Oleszewski. Simplifying concurrent algorithms by exploiting hardware transactional memory. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures, SPAA ’10*, pages 325–334, New York, NY, USA, 2010. ACM.
- [DLMN09] Dave Dice, Yossi Lev, Mark Moir, and Dan Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’09)*, pages 157–168, Washington, DC, USA, March 2009. Available: <http://research.sun.com/scalable/pubs/ASPLOS2009-RockHTM.pdf> [Viewed February 4, 2009].
- [DMS⁺12] Mathieu Desnoyers, Paul E. McKenney, Alan Stern, Michel R. Dagenais, and Jonathan Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems*, 23:375–382, 2012.
- [Dov90] Ken F. Dove. A high capacity TCP/IP in parallel STREAMS. In *UKUUG Conference Proceedings*, London, June 1990.
- [Dre11] Ulrich Drepper. Futexes are tricky. Technical Report FAT2011, Red Hat, Inc., Raleigh, NC, USA, November 2011.
- [DSS06] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proc. International Symposium on Distributed Computing*. Springer Verlag, 2006. Available: <http://www.springerlink.com/content/5688h5q0w72r54x0/> [Viewed March 10, 2008].
- [Edg13] Jake Edge. The future of realtime Linux. <http://lwn.net/Articles/572740/>, November 2013.
- [Edg14] Jake Edge. The future of the realtime patch set. <http://lwn.net/Articles/617140/>, October 2014.
- [EGCD03] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. UPC language specifications v1.1. Available: <http://upc.cs.umn.edu/>.

- gwu.edu [Viewed September 19, 2008], May 2003.
- [EGMdB11] Stephane Eranian, Eric Gouriou, Tipp Moseley, and Willem de Bruijn. Linux kernel profiling with perf. <https://perf.wiki.kernel.org/index.php/Tutorial>, June 2011.
- [Ell80] Carla Schlatter Ellis. Concurrent search and insertion in avl trees. *IEEE Transactions on Computers*, C-29(9):811–817, September 1980.
- [ELLM07] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. Snzi: scalable nonzero indicators. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC ’07, pages 13–22, New York, NY, USA, 2007. ACM.
- [Eng68] Douglas Engelbart. The demo. Available: <http://video.google.com/videoplay?docid=-8734787622017763097> [Viewed November 28, 2008], December 1968.
- [ENS05] Ryan Eccles, Blair Nonneck, and Deborah A. Stacey. Exploring parallel programming knowledge in the novice. In *HPCS ’05: Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications*, pages 97–102, Washington, DC, USA, 2005. IEEE Computer Society.
- [Eri08] Christer Ericson. Aiding pathfinding with cellular automata. <http://realtimecollisiondetection.net/blog/?p=57>, June 2008.
- [ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, 1990.
- [ES05] Ryan Eccles and Deborah A. Stacey. Understanding the parallel programmer. In *HPCS ’05: Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications*, pages 156–160, Washington, DC, USA, 2005. IEEE Computer Society.
- [ETH11] ETH Zurich. Parallel solver for a perfect maze. <http://nativesystems.inf.ethz.ch/pub/Main/WebHomeLecturesParallelProgrammingExercises2011hw04.pdf>, March 2011.
- [Fel50] W. Feller. *An Introduction to Probability Theory and its Applications*. John Wiley, 1950.
- [Fos10] Ron Fosner. Scalable multithreaded programming with tasks. *MSDN Magazine*, 2010(11):60–69, November 2010. <http://msdn.microsoft.com/en-us/magazine/gg309176.aspx>.
- [FPB79] Jr. Frederick P. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1979.
- [FRK02] Hubertus Francke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Ottawa Linux Symposium*, pages 479–495, June 2002. Available: <http://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf> [Viewed May 22, 2011].
- [Gar90] Arun Garg. Parallel STREAMS: a multi-processor implementation. In *USENIX Conference Proceedings*, pages 163–176, Berkeley CA, February 1990. USENIX Association.
- [Gar07] Bryan Gardiner. Idf: Gordon moore predicts end of moore’s law (again). Available: <http://blog.wired.com/business/2007/09/idf-gordon-mo-1.html> [Viewed: November 28, 2008], September 2007.
- [GC96] Michael Greenwald and David R. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 123–136, Seattle, WA, October 1996. USENIX Association.
- [GDZE10] Olga Golovanevsky, Alon Dayan, Ayal Zaks, and David Edelsohn. Trace-based

- [GG14] Vincent Gramoli and Rachid Guerraoui. Democratizing transactional programming. *Commun. ACM*, 57(1):86–93, January 2014.
- [Gha95] Kourosh Gharachorloo. Memory consistency models for shared-memory multiprocessors. Technical Report CSL-TR-95-685, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA, December 1995. Available: <http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-9.pdf> [Viewed: October 11, 2004].
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [GKAS99] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation*, pages 87–100, New Orleans, LA, February 1999.
- [GKP13] Justin Gottschlich, Rob Knauerhase, and Gilles Pokam. But how do we really debug transactional memory? In *5th USENIX Workshop on Hot Topics in Parallelism (HotPar 2013)*, San Jose, CA, USA, June 2013.
- [GKPS95] Ben Gamsa, Orran Krieger, E. Parsons, and Michael Stumm. Performance issues for multiprocessor operating systems. Technical Report CSRI-339, Available: <ftp://ftp.cs.toronto.edu/pub/reports/csri/339/339.ps>, November 1995.
- [Gle10] Thomas Gleixner. Realtime linux: academia v. reality. Available: <http://lwn.net/Articles/397422/> [Viewed July 27, 2010], July 2010.
- [Gle12] Thomas Gleixner. Linux -rt kvm guest demo. Personal communication, December 2012.
- [GMTW08] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal*, 47(2):221–236, May 2008.
- [GPB⁺07] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java: Concurrency in Practice*. Addison Wesley, Upper Saddle River, NJ, USA, 2007.
- [Gra02] Jim Gray. Super-servers: Commodity computer clusters pose a software challenge. Available: [http://research.microsoft.com/en-us/um/people/gray/papers/superservers\(4t_computers\).doc](http://research.microsoft.com/en-us/um/people/gray/papers/superservers(4t_computers).doc) [Viewed: June 23, 2004], April 2002.
- [Gri00] Scott Griffen. Internet pioneers: Doug Englebart. Available: <http://www.ibiblio.org/pioneers/englebart.html> [Viewed November 28, 2008], May 2000.
- [Gro01] The Open Group. Single UNIX specification. <http://www.opengroup.org/onlinepubs/007908799/index.html>, July 2001.
- [Gro07] Dan Grossman. The transactional memory / garbage collection analogy. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 695–706, New York, NY, USA, October 2007. ACM. Available: http://www.cs.washington.edu/homes/djg/papers/analogy_oopsla07.pdf [Viewed December 19, 2008].

- [GT90] Gary Graunke and Shreekant Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–69, June 1990.
- [HCS⁺05] Lorin Hochstein, Jeff Carver, Forrest Shull, Sima Asgari, and Victor Basili. Parallel programmer productivity: A case study of novice parallel programmers. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 35, Washington, DC, USA, 2005. IEEE Computer Society.
- [Hei27] W. Heisenberg. Über den anschaulichen Inhalt der quantentheoretischen Kinetik und Mechanik. *Zeitschrift für Physik*, 43(3–4):172–198, 1927. English translation in “Quantum theory and measurement” by Wheeler and Zurek.
- [Her90] Maurice P. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206, March 1990.
- [Her91] Maurice Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, January 1991.
- [Her93] Maurice Herlihy. Implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.
- [Her05] Maurice Herlihy. The transactional manifesto: software engineering and non-blocking synchronization. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 280–280, New York, NY, USA, 2005. ACM Press.
- [HHK⁺13] A. Haas, T.A. Henzinger, C.M. Kirsch, M. Lippautz, H. Payer, A. Sezgin, and A. Sokolova. Distributed queues in shared memory—multicore performance and scalability through quantitative relaxation. In *Proc. International Conference on Computing Frontiers*. ACM, 2013.
- [HKLP12] Andreas Haas, Christoph M. Kirsch, Michael Lippautz, and Hannes Payer. How FIFO is your concurrent FIFO queue? In *Proceedings of the Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, Tucson, AZ USA, October 2012.
- [HLM02] Maurice Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of 16th International Symposium on Distributed Computing*, pages 339–353, October 2002.
- [HLM03] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 73–82, Providence, RI, May 2003. The Institute of Electrical and Electronics Engineers, Inc.
- [HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *The 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.
- [HMB06] Thomas E. Hart, Paul E. McKenney, and Angela Demke Brown. Making lockless synchronization fast: Performance implications of memory reclamation. In *20th IEEE International Parallel and Distributed Processing Symposium*, Rhodes, Greece, April 2006. Available: http://www.rdrop.com/users/paulmck/RCU/hart_ipdps06.pdf [Viewed April 28, 2008].
- [HMBW07] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.*, 67(12):1270–1285, 2007.
- [Hol03] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, MA, USA, 2003.

- [HP95] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 1995.
- [HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, Burlington, MA, USA, 2008.
- [HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.
- [HW92] Wilson C. Hsieh and William E. Weihl. Scalable reader-writer locks for parallel systems. In *Proceedings of the 6th International Parallel Processing Symposium*, pages 216–230, Beverly Hills, CA, USA, March 1992.
- [HW11] Philip W. Howard and Jonathan Walpole. A relativistic enhancement to software transactional memory. In *Proceedings of the 3rd USENIX conference on Hot topics in parallelism*, HotPar’11, pages 1–6, Berkeley, CA, USA, 2011. USENIX Association.
- [HW13] Philip W. Howard and Jonathan Walpole. Relativistic red-black trees. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2013.
- [IBM94] IBM Microelectronics and Motorola. *PowerPC Microprocessor Family: The Programming Environments*, 1994.
- [Ima15] Imagination Technologies, LTD. *MIPS®Architecture For Programmers Volume II-A: The MIPS64®Instruction Set Reference Manual*, 2015. <https://imgtec.com/?do-download=4302>.
- [Inm85] Jack Inman. Implementing loosely coupled functions on tightly coupled engines. In *USENIX Conference Proceedings*, pages 277–298, Portland, OR, June 1985. USENIX Association.
- [Inm07] Bill Inmon. Time value of information. Available: <http://www.b-eye-network.de/view-articles/3365> [Viewed July 6, 2007], January 2007.
- [Int92] International Standards Organization. *Information Technology - Database Language SQL*. ISO, 1992. Available: <http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt> [Viewed September 19, 2008].
- [Int02a] Intel Corporation. *Intel Itanium Architecture Software Developer’s Manual Volume 3: Instruction Set Reference*, 2002.
- [Int02b] Intel Corporation. *Intel Itanium Architecture Software Developer’s Manual Volume 3: System Architecture*, 2002.
- [Int04a] Intel Corporation. *IA-32 Intel Architecture Software Developer’s Manual Volume 2B: Instruction Set Reference, N-Z*, 2004. Available: <ftp://download.intel.com/design/Pentium4/manuals/25366714.pdf> [Viewed: February 16, 2005].
- [Int04b] Intel Corporation. *IA-32 Intel Architecture Software Developer’s Manual Volume 3: System Programming Guide*, 2004. Available: <ftp://download.intel.com/design/Pentium4/manuals/25366814.pdf> [Viewed: February 16, 2005].
- [Int04c] International Business Machines Corporation. z/Architecture principles of operation. Available: <http://publibz.boulder.ibm.com/epubs/pdf/dz9zr003.pdf> [Viewed: February 16, 2005], May 2004.
- [Int07] Intel Corporation. *Intel 64 Architecture Memory Ordering White Paper*, 2007. Available: <http://developer.intel.com/products/processor/manuals/318147.pdf> [Viewed: September 7, 2007].
- [Int11] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A: System Programming Guide, Part 1*, 2011. Available: <http://www.intel.com/Assets/PDF/manual/253668.pdf> [Viewed: February 12, 2011].

- [Jac08] Daniel Jackson. MapReduce course. Available: <https://sites.google.com/site/mriap2008/> [Viewed January 3, 2013], January 2008.
- [JMRR02] Benedict Joseph Jackson, Paul E. McKenney, Ramakrishnan Rajamony, and Ronald Lynn Rockhold. Scalable interruptible queue locks for shared-memory multiprocessor. Technical Report US Patent 6,473,819, US Patent and Trademark Office, Washington, DC, October 2002.
- [Joh77] Stephen Johnson. Lint, a C program checker. Computer Science Technical Report 65, Bell Laboratories, December 1977.
- [Jon11] Dave Jones. Trinity: A system call fuzzer. In *Proceedings of the 13th Ottawa Linux Symposium*, pages ???–???, Ottawa, Canada, June 2011.
- [JSG12] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional memory architecture and implementation for IBM System z. The 45th Annual IEEE/ACM International Symposium on MicroArchitecture <http://www.microsymposia.org/micro45/talks-posters/3-jacobi-presentation.pdf>, December 2012.
- [Kan96] Gerry Kane. *PA-RISC 2.0 Architecture*. Hewlett-Packard Professional Books, 1996.
- [KCH⁺06] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kumar, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of the ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming*. ACM SIGPLAN, 2006. Available: <http://www.cs.princeton.edu/~skumar/papers/ppopp06/ppopp06.pdf> [Viewed December 21, 2006].
- [KFC11] KFC. Memristor processor solves mazes. <http://www.technologyreview.com/blog/arxiv/26467/>, March 2011.
- [Kis14] Jan Kiszka. Real-time virtualization - how crazy are we? In *Linux Plumbers Conference*, Duesseldorf, Germany, October 2014. <http://www.linuxplumbersconf.org/2014/ocw/sessions/1935>.
- [KL80] H. T. Kung and P. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems*, 5(3):354–382, September 1980.
- [KLP12] Christoph M. Kirsch, Michael Lippautz, and Hannes Payer. Fast and scalable k-fifo queues. Technical Report 2012-04, University of Salzburg, Salzburg, Austria, June 2012.
- [Kni08] John U. Knickerbocker. 3D chip technology. *IBM Journal of Research and Development*, 52(6), November 2008. Available: <http://www.research.ibm.com/journal/rd52-6.html> [Viewed: January 1, 2009].
- [Knu73] Donald Knuth. *The Art of Computer Programming*. Addison-Wesley, 1973.
- [KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.
- [KWS97] Leonidas Kontothanassis, Robert W. Wisniewski, and Michael L. Scott. Scheduler-conscious synchronization. *Communications of the ACM*, 15(1):3–40, January 1997.
- [LA94] Beng-Hong Lim and Anant Agarwal. Reactive synchronization algorithms for multiprocessors. 03/28/94 FTP <hing.lcs.mit.edu/pub/papers/reactive.ps.Z>, March 1994.
- [Lam74] Leslie Lamport. A new solution of Dijkstra’s concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.
- [Lea97] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison Wesley Longman, Reading, MA, USA, 1997.

- [LLO09] Yossi Lev, Victor Luchangco, and Marek Olszewski. Scalable reader-writer locks. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 101–110, New York, NY, USA, 2009. ACM.
- [Loc02] Doug Locke. Priority inheritance: The real story. Available: <http://www.linuxdevices.com/articles/AT5698775833.html> [Viewed June 29, 2005], July 2002.
- [Lom77] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *SIGSOFT Softw. Eng. Notes*, 2(2):128–137, 1977. Available: <http://portal.acm.org/citation.cfm?id=808319#> [Viewed June 27, 2008].
- [LR80] Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, 1980.
- [LS86] Vladimir Lanin and Dennis Shasha. A symmetric concurrent b-tree algorithm. In *ACM '86: Proceedings of 1986 ACM Fall joint computer conference*, pages 380–389, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.
- [LS11] Yujie Liu and Michael Spear. Toxic transactions. In *TRANSACT 2011*. ACM SIGPLAN, June 2011.
- [LSH02] Michael Lyons, Ed Silha, and Bill Hay. PowerPC storage model and AIX programming. Available: <http://www-106.ibm.com/developerworks/eserver/articles/powerpc.html> [Viewed: January 31, 2005], August 2002.
- [MAK⁺01] Paul E. McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *Ottawa Linux Symposium*, July 2001. Available: <http://www.linuxsymposium.org/2001/abstracts/readcopy.php> <http://www.rdrop.com/users/paulmck/readcopy/>
- [paulmck/RCU/rclock_OLS.2001.05.01c.pdf] [Viewed June 23, 2004].
- [Mas92] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, New York, NY, 1992.
- [Mat13] Norm Matloff. *Programming on Parallel Machines*. University of California, Davis, Davis, CA, USA, 2013.
- [MBM⁺06] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th Annual International Symposium on High Performance Computer Architecture (HPCA-12)*, Washington, DC, USA, 2006. IEEE. Available: http://www.cs.wisc.edu/multifacet/papers/hpca06_logtm.pdf [Viewed December 21, 2006].
- [MBWW12] Paul E. McKenney, Silas Boyd-Wickizer, and Jonathan Walpole. RCU usage in the linux kernel: One decade later. Technical report paulmck.2012.09.17, September 2012.
- [McK90a] Paul E. McKenney. Stochastic fairness queuing. Technical Report ITSTD-7186-PA-89-11, SRI International, Menlo Park, CA, March 1990. To appear in INFOCOM'90.
- [McK90b] Paul E. McKenney. Stochastic fairness queuing. In *IEEE INFOCOM'90 Proceedings*, pages 733–740, San Francisco, June 1990. The Institute of Electrical and Electronics Engineers, Inc. Revision available: <http://www.rdrop.com/users/paulmck/scalability/paper/sfq.2002.06.04.pdf> [Viewed May 26, 2008].
- [McK91] Paul E. McKenney. Stochastic fairness queuing. *Internetworking: Theory and Experience*, 2:113–131, 1991.
- [McK95] Paul E. McKenney. Differential profiling. In *MASCOTS 1995*, pages 237–241, Toronto, Canada, January 1995.

- [McK96a] Paul E. McKenney. *Pattern Languages of Program Design*, volume 2, chapter 31: Selecting Locking Designs for Parallel Programs, pages 501–531. Addison-Wesley, June 1996. Available: <http://www.rdrop.com/users/paulmck/scalability/paper/mutexdesignpat.pdf> [Viewed February 17, 2005].
- [McK96b] Paul E. McKenney. Selecting locking primitives for parallel programs. *Communications of the ACM*, 39(10):75–82, October 1996.
- [McK99] Paul E. McKenney. Differential profiling. *Software - Practice and Experience*, 29(3):219–234, 1999.
- [McK03] Paul E. McKenney. Using RCU in the Linux 2.5 kernel. *Linux Journal*, 1(114):18–26, October 2003. Available: <http://www.linuxjournal.com/article/6993> [Viewed November 14, 2007].
- [McK04] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.
- [McK05a] Paul E. McKenney. Memory ordering in modern microprocessors, part I. *Linux Journal*, 1(136):52–57, August 2005. Available: <http://www.linuxjournal.com/article/8211> <http://www.rdrop.com/users/paulmck/scalability/paper/ordering.2007.09.19a.pdf> [Viewed November 30, 2007].
- [McK05b] Paul E. McKenney. Memory ordering in modern microprocessors, part II. *Linux Journal*, 1(137):78–82, September 2005. Available: <http://www.linuxjournal.com/article/8212> <http://www.rdrop.com/users/paulmck/scalability/paper/>
- [McK05c] Paul E. McKenney. A realtime pre-emption overview. Available: <http://lwn.net/Articles/146861/> [Viewed August 22, 2005], August 2005.
- [McK06] Paul E. McKenney. Sleepable RCU. Available: <http://lwn.net/Articles/202847/> Revised: <http://www.rdrop.com/users/paulmck/RCU/srcu.2007.01.14a.pdf> [Viewed August 21, 2006], October 2006.
- [McK07a] Paul E. McKenney. The design of preemptible read-copy-update. Available: <http://lwn.net/Articles/253651/> [Viewed October 25, 2007], October 2007.
- [McK07b] Paul E. McKenney. [PATCH] QRCU with lockless fastpath. Available: <http://lkml.org/lkml/2007/2/25/18> [Viewed March 27, 2008], February 2007.
- [McK07c] Paul E. McKenney. Priority-boosting RCU read-side critical sections. <http://lwn.net/Articles/220677/>, February 2007.
- [McK07d] Paul E. McKenney. RCU and unloadable modules. Available: <http://lwn.net/Articles/217484/> [Viewed November 22, 2007], January 2007.
- [McK07e] Paul E. McKenney. Using Promela and Spin to verify parallel algorithms. Available: <http://lwn.net/Articles/243851/> [Viewed September 8, 2007], August 2007.
- [McK07f] Paul E. McKenney. What is RCU? Available: <http://www.rdrop.com/users/paulmck/RCU/whatisRCU.html> [Viewed July 6, 2007], 07 2007.
- [McK08a] Paul E. McKenney. Hierarchical RCU. <http://lwn.net/Articles/305782/>, November 2008.
- [McK08b] Paul E. McKenney. RCU part 3: the RCU API. Available: <http://lwn.net/>

- [McK08c] Paul E. McKenney. What is RCU? part 2: Usage. Available: <http://lwn.net/Articles/263130/> [Viewed January 4, 2008], January 2008.
- [McK09a] Paul E. McKenney. Re: [PATCH fyi] RCU: the bloatwatch edition. Available: <http://lkml.org/lkml/2009/1/14/449> [Viewed January 15, 2009], January 2009.
- [McK09b] Paul E. McKenney. Transactional memory everywhere? <http://paulmck.livejournal.com/tag/transactional%20memory%20everywhere>, September 2009.
- [McK11a] Paul E. McKenney. 3.0 and RCU: what went wrong. <http://lwn.net/Articles/453002/>, July 2011.
- [McK11b] Paul E. McKenney. Concurrent code and expensive instructions. Available: <http://lwn.net/Articles/423994> [Viewed January 28, 2011], January 2011.
- [McK11c] Paul E. McKenney. Validating memory barriers and atomic instructions. <http://lwn.net/Articles/470681/>, December 2011.
- [McK11d] Paul E. McKenney. Verifying parallel software: Can theory meet practice? <http://www.rdrop.com/users/paulmck/scalability/paper/VericoTheoryPractice.2011.01.28a.pdf>, January 2011.
- [McK12a] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org, Corvallis, OR, USA, 2012.
- [McK12b] Paul E. McKenney. Making RCU safe for battery-powered devices. Available: <http://www.rdrop.com/users/paulmck/RCU/RCUdynticks.2012.02.15b.pdf> [Viewed March 1, 2012], February 2012.
- [McK12c] Paul E. McKenney. Retrofitted parallelism considered grossly sub-optimal. In *4th USENIX Workshop on Hot Topics on Parallelism*, page 7, Berkeley, CA, USA, June 2012.
- [McK12d] Paul E. McKenney. Signed overflow optimization hazards in the kernel. <http://lwn.net/Articles/511259/>, August 2012.
- [McK13] Paul E. McKenney. Structured deferral: synchronization via procrastination. *Commun. ACM*, 56(7):40–49, July 2013.
- [McK14] Paul E. McKenney. The RCU API, 2014 edition. <http://lwn.net/Articles/609904/>, September 2014.
- [MCM02] Paul E. McKenney, Kevin A. Clossen, and Raghupathi Malige. Lingering locks with fairness control for multi-node computer systems. Technical Report US Patent 6,480,918, US Patent and Trademark Office, Washington, DC, November 2002.
- [MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *Transactions of Computer Systems*, 9(1):21–65, February 1991.
- [MD92] Paul E. McKenney and Ken F. Dove. Efficient demultiplexing of incoming tcp packets. In *SIGCOMM '92, Proceedings of the Conference on Communications Architecture & Protocols*, pages 269–279, Baltimore, MD, August 1992. Association for Computing Machinery.
- [MDJ13a] Paul E. McKenney, Mathieu Desnoyers, and Lai Jiangshan. URCU-protected hash tables. <http://lwn.net/Articles/573431/>, November 2013.
- [MDJ13b] Paul E. McKenney, Mathieu Desnoyers, and Lai Jiangshan. URCU-protected queues and stacks. <https://lwn.net/Articles/573433/>, November 2013.
- [Mel06] Melbourne School of Engineering. CSIRAC. Available: <http://>

- //www.csse.unimelb.edu.au/dept/about/csirac/ [Viewed: December 7, 2008], 2006.
- [Met99] Panagiotis Takis Metaxas. Fast dithering on a data-parallel computer. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 570–576, Cambridge, MA, USA, 1999. IASTED.
- [MG92] Paul E. McKenney and Gary Graunke. Efficient buffer allocation on shared-memory multiprocessors. In *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, pages 194–199, Tucson, AZ, February 1992. The Institute of Electrical and Electronics Engineers, Inc.
- [MGM⁺09] Paul E. McKenney, Manish Gupta, Maged M. Michael, Phil Howard, Joshua Triplett, and Jonathan Walpole. Is parallel programming hard, and if so, why? Technical Report TR-09-02, Portland State University, Portland, OR, USA, February 2009. Available: <http://www.cs.pdx.edu/pdfs/tr0902.pdf> [Viewed February 19, 2009].
- [MHS12] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why on-chip coherence is here to stay. *Communications of the ACM*, 55(7):78–89, July 2012.
- [Mic03] Maged M. Michael. Cas-based lock-free algorithm for shared deques. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par*, volume 2790 of *Lecture Notes in Computer Science*, pages 651–660. Springer, 2003.
- [Mic04] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.
- [Mil06] David S. Miller. Re: [PATCH, RFC] RCU : OOM avoidance and lower latency. Available: <https://lkml.org/lkml/2006/1/7/22> [Viewed February 29, 2012], January 2006.
- [MK88] Marshall Kirk McKusick and Michael J. Karels. Design of a general purpose memory allocator for the 4.3BSD UNIX kernel. In *USENIX Conference Proceedings*, Berkeley CA, June 1988.
- [MLH94] Peter Magnusson, Anders Landin, and Erik Hagersten. Efficient software synchronization on large cache coherent multiprocessors. Technical Report T94:07, Swedish Institute of Computer Science, Kista, Sweden, February 1994.
- [MM00] Ingo Molnar and David S. Miller. brlock. Available: http://www.tm.kernel.org/pub/linux/kernel/v2.3/patch-html/patch-2.3.49/linux_include_linux_brlock.h.html [Viewed September 3, 2004], March 2000.
- [MMTW10] Paul E. McKenney, Maged M. Michael, Josh Triplett, and Jonathan Walpole. Why the grass may not be greener on the other side: a comparison of locking vs. transactional memory. *ACM Operating Systems Review*, 44(3), July 2010.
- [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.
- [Mol05] Ingo Molnar. Index of /pub/linux/kernel/projects/rt. Available: <http://www.kernel.org/pub/linux/kernel/projects/rt/> [Viewed February 15, 2005], February 2005.
- [Mol06] Ingo Molnar. Lightweight robust futexes. Available: <http://lxr.linux.no/#linux+v2.6.39/Documentation/robust-futexes.txt> [Viewed May 22, 2011], March 2006.
- [Moo03] Gordon Moore. No exponential is forever—but we can delay forever. In *IBM Academy*

- of Technology 2003 Annual Meeting*, San Francisco, CA, October 2003.
- [MOZ09] Nicholas Mc Guire, Peter Odhiambo Okech, and Qingguo Zhou. Analysis of inherent randomness of the linux kernel. In *Eleventh Real Time Linux Workshop*, Dresden, Germany, September 2009.
- [MPA⁺06] Paul E. McKenney, Chris Purcell, Algae, Ben Schumin, Gaius Cornelius, Qwertyus, Neil Conway, Sbw, Blainster, Canis Rufus, Zoicon5, Anome, and Hal Eisen. Read-copy update. <http://en.wikipedia.org/wiki/Read-copy-update>, July 2006.
- [MPI08] MPI Forum. Message passing interface forum. Available: <http://www mpi-forum.org/> [Viewed September 9, 2008], September 2008.
- [MR08] Paul E. McKenney and Steven Rostedt. Integrating and validating dynticks and preemptable RCU. Available: <http://lwn.net/Articles/279077/> [Viewed April 24, 2008], April 2008.
- [MS93] Paul E. McKenney and Jack Slingwine. Efficient kernel memory allocation on shared-memory multiprocessors. In *USENIX Conference Proceedings*, pages 295–306, Berkeley CA, February 1993. USENIX Association. Available: <http://www.rdrop.com/users/paulmck/scalability/paper/mpalloc.pdf> [Viewed January 30, 2005].
- [MS96] M.M Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc of the Fifteenth ACM Symposium on Principles of Distributed Computing*, pages 267–275, May 1996. Available: <http://www.research.ibm.com/people/m/michael/podc-1996.pdf> [Viewed January 26, 2009].
- [MS98a] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.
- [MS98b] Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51(1):1–26, 1998.
- [MS08] MySQL AB and Sun Microsystems. MySQL Downloads. Available: <http://dev.mysql.com/downloads/> [Viewed November 26, 2008], November 2008.
- [MS09] Paul E. McKenney and Raul Silvera. Example power implementation for c/c++ memory model. Available: <http://www.rdrop.com/users/paulmck/scalability/paper/N2745r.2009.02.27a.html> [Viewed: April 5, 2009], February 2009.
- [MS12] Alexander Matveev and Nir Shavit. Towards a fully pessimistic STM model. In *TRANSACT 2012*. ACM SIGPLAN, February 2012.
- [MS14] Paul E. McKenney and Alan Stern. Axiomatic validation of memory barriers and atomic instructions. <http://lwn.net/Articles/608550/>, August 2014.
- [MSK01] Paul E. McKenney, Jack Slingwine, and Phil Krueger. Experience with an efficient parallel kernel memory allocator. *Software – Practice and Experience*, 31(3):235–257, March 2001.
- [MSM05] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. Addison Wesley, Boston, MA, USA, 2005.
- [MSS04] Paul E. McKenney, Dipankar Sarma, and Maneesh Soni. Scaling dcache with RCU. *Linux Journal*, 1(118):38–46, January 2004.
- [MT01] Jose F. Martinez and Josep Torrellas. Speculative locks for concurrent execution

- of critical sections in shared-memory multiprocessors. In *Workshop on Memory Performance Issues, International Symposium on Computer Architecture*, Gothenburg, Sweden, June 2001. Available: http://iacoma.cs.uiuc.edu/iacoma-papers/wmpo_locks.pdf [Viewed June 23, 2004].
- [MT02] Jose F. Martinez and Josep Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 18–29, San Jose, CA, October 2002.
- [Mud00] Trevor Mudge. POWER: A first-class architectural design constraint. *IEEE Computer*, 33(4):52–58, April 2000.
- [Mus04] Museum Victoria Australia. CSIRAC: Australia’s first computer. Available: <http://museumvictoria.com.au/CSIRAC/> [Viewed: December 7, 2008], 2004.
- [MW07] Paul E. McKenney and Jonathan Walpole. What is RCU, fundamentally? Available: <http://lwn.net/Articles/262464/> [Viewed December 27, 2007], December 2007.
- [Mye79] Glenford J. Myers. *The Art of Software Testing*. Wiley, 1979.
- [Nes06a] Oleg Nesterov. Re: [patch] cpufreq: mark cpufreq_tsc() as core_initcall_sync. Available: <http://lkml.org/lkml/2006/11/19/69> [Viewed May 28, 2007], November 2006.
- [Nes06b] Oleg Nesterov. Re: [rfc, patch 1/2] qrcu: "quick" srcu implementation. Available: <http://lkml.org/lkml/2006/11/29/330> [Viewed November 26, 2008], November 2006.
- [ON06] Robert Olsson and Stefan Nilsson. TRASH: A dynamic LC-trie and hash data structure. <http://www.nada.kth.se/~snilsson/publications/TRASH/trash.pdf>, August 2006.
- [ONH⁺96] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *ASPLOS VII*, October 1996.
- [Ope97] Open Group. The single UNIX specification, version 2: Threads. Available: <http://www.opengroup.org/onlinepubs/007908799/xsh/threads.html> [Viewed September 19, 2008], 1997.
- [Pat10] David Patterson. The trouble with multicore. *IEEE Spectrum*, 2010:28–32, 52–53, July 2010.
- [Pet06] Jeremy Peters. From reuters, automatic trading linked to news events. Available: <http://www.nytimes.com/2006/12/11/technology/11reuters.html?ei=5088&en=e5e9416415a9eeb2&ex=1323493200...> [Viewed July 7, 2007], December 2006.
- [Pig06] Nick Piggin. [patch 3/3] radix-tree: RCU lockless readside. Available: <http://lkml.org/lkml/2006/6/20/238> [Viewed March 25, 2008], June 2006.
- [Pok16] Michael Pokorny. The deadlock empire. <https://deadlockempire.github.io/>, February 2016.
- [Pos08] PostgreSQL Global Development Group. PostgreSQL. Available: <http://www.postgresql.org/> [Viewed November 26, 2008], November 2008.
- [Pug90] William Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222.1, Institute of Advanced Computer Science Studies, Department of Computer Science, University of Maryland, College Park, Maryland, June 1990.

- [Pul00] Geoffrey K. Pullum. How Dr. Seuss would prove the halting problem undecidable. *Mathematics Magazine*, 73(4):319–320, 2000. <http://www.lcl.ed.ac.uk/~gpullum/loopsnoop.html>.
- [PW07] Donald E. Porter and Emmett Witchel. Lessons from large transactional systems. Personal communication <20071214220521.GA5721@olive-green.cs.utexas.edu>, December 2007.
- [Ray99] Eric S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly, 1999.
- [RD12] Ravi Rajwar and Martin Dixon. Intel transactional synchronization extensions. Intel Developer Forum (IDF) 2012 ARCS004, September 2012.
- [Reg10] John Regehr. A guide to undefined behavior in c and c++, part 1. <http://blog.regehr.org/archives/213>, July 2010.
- [Rei07] James Reinders. *Intel Threading Building Blocks*. O'Reilly, Sebastopol, CA, USA, 2007.
- [RG01] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 294–305, Austin, TX, December 2001. The Institute of Electrical and Electronics Engineers, Inc.
- [RG02] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, Austin, TX, October 2002.
- [RH02] Zoran Radović and Erik Hagersten. Efficient synchronization for nonuniform communication architectures. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 1–13, Baltimore, Maryland, USA, November 2002. The Institute of Electrical and Electronics Engineers, Inc.
- [RH03] Zoran Radović and Erik Hagersten. Hierarchical backoff locks for nonuniform communication architectures. In *Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, pages 241–252, Anaheim, California, USA, February 2003.
- [RHP⁺07] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Aditya Bhandari, and Emmett Witchel. TxLinux: Using and managing hardware transactional memory in an operating system. In *SOSP'07: Twenty-First ACM Symposium on Operating Systems Principles*. ACM SIGOPS, October 2007. Available: <http://www.sosp2007.org/papers/sosp056-rossbach.pdf> [Viewed October 21, 2007].
- [Ros06] Steven Rostedt. Lightweight PI-futexes. Available: <http://lxr.linux.no/#linux+v2.6.39/Documentation/pi-futex.txt> [Viewed May 22, 2011], June 2006.
- [Ros10a] Steven Rostedt. tracing: Harry Potter and the Deathly Macros. Available: <http://lwn.net/Articles/418710/> [Viewed: August 28, 2011], December 2010.
- [Ros10b] Steven Rostedt. Using the TRACE_EVENT() macro (part 1). Available: <http://lwn.net/Articles/379903/> [Viewed: August 28, 2011], March 2010.
- [Ros10c] Steven Rostedt. Using the TRACE_EVENT() macro (part 2). Available: <http://lwn.net/Articles/381064/> [Viewed: August 28, 2011], March 2010.
- [Ros10d] Steven Rostedt. Using the TRACE_EVENT() macro (part 3). Available: <http://lwn.net/Articles/>

- [Ros11] Steven Rostedt. lockdep: How to read its cryptic output. <http://www.linuxplumbersconf.org/2011/ocw/sessions/153>, September 2011.
- [Rus03] Rusty Russell. Hanging out with smart people: or... things I learned being a kernel monkey. 2003 Ottawa Linux Symposium Keynote <http://ozlabs.org/~rusty/ols-2003-keynote/ols-keynote-2003.html>, July 2003.
- [SAH⁺03] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Robert W. Wisniewski, Marc Auslander, Michal Ostrowski, Bryan Rosenberg, and Jimi Xenidis. System support for online reconfiguration. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 141–154. USENIX Association, June 2003.
- [SATG⁺09] Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Robert Geva, Yang Ni, and Adam Welc. Towards transactional memory semantics for c++. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 49–58, New York, NY, USA, 2009. ACM.
- [Sch35] E. Schrödinger. Die gegenwärtige Situation in der Quantenmechanik. *Naturwissenschaften*, 23:807–812; 823–828; 844–949, November 1935. English translation: <http://www.tuhh.de/rzt/rzt/it/QM/cat.html>.
- [Sch94] Curt Schimmel. *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.
- [Sco06] Michael Scott. *Programming Language Pragmatics*. Morgan Kaufmann, Burlington, MA, USA, 2006.
- [Sco13] Michael L. Scott. *Shared-Memory Synchronization*. Morgan & Claypool, San Rafael, CA, USA, 2013.
- [Seq88] Sequent Computer Systems, Inc. *Guide to Parallel Programming*, 1988.
- [Sew] Peter Sewell. The semantics of multiprocessor programs. Available: <http://www.cl.cam.ac.uk/~pes20/weakmemory/> [Viewed: June 7, 2010].
- [Sha11] Nir Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, March 2011.
- [SHW11] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2011.
- [SM04] Dipankar Sarma and Paul E. McKenney. Making RCU safe for deep sub-millisecond response realtime applications. In *Proceedings of the 2004 USENIX Annual Technical Conference (FREENIX Track)*, pages 182–191. USENIX Association, June 2004.
- [Smi15] Richard Smith. Working draft, standard for programming language C++. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4527.pdf>, May 2015.
- [SMS08] Michael Spear, Maged Michael, and Michael Scott. Inevitability mechanisms for software transactional memory. In *3rd ACM SIGPLAN Workshop on Transactional Computing*, New York, NY, USA, February 2008. ACM. Available: http://www.cs.rochester.edu/u/scott/papers/2008_TRANSACT_inevitability.pdf [Viewed January 10, 2009].
- [SPA94] SPARC International. *The SPARC Architecture Manual*, 1994.
- [Spi77] Keith R. Spitz. Tell which is which and you'll be rich. Inscription on wall of dungeon, 1977.

- [Spr01] Manfred Spraul. Re: RFC: patch to allow lock-free traversal of lists with insertion. Available: <http://marc.theaimsgroup.com/?l=linux-kernel&m=100264675012867&w=2> [Viewed June 23, 2004], October 2001.
- [Spr08] Manfred Spraul. [RFC, PATCH] state machine based rcu. Available: <http://lkml.org/lkml/2008/8/21/336> [Viewed December 8, 2008], August 2008.
- [SR84] Z. Segall and L. Rudolf. Dynamic decentralized cache schemes for MIMD parallel processors. In *11th Annual International Symposium on Computer Architecture*, pages 340–347, June 1984.
- [SRL90a] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.
- [SRL90b] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.
- [SS94] Duane Szafron and Jonathan Schaeffer. Experimentally assessing the usability of parallel programming systems. In *IFIP WG10.3 Programming Environments for Massively Parallel Distributed Systems*, pages 19.1–19.7, 1994.
- [SS06] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, May 2006.
- [SSHT93] Janice S. Stone, Harold S. Stone, Philip Heidelberger, and John Turek. Multiple reservations and the Oklahoma update. *IEEE Parallel and Distributed Technology Systems and Applications*, 1(4):58–71, November 1993.
- [SSRB00] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, Chichester, West Sussex, England, 2000.
- [SSVM02] S. Swaminathan, John Stultz, Jack Vogel, and Paul E. McKenney. Fairlocks – a high performance fair locking scheme. In *Proceedings of the 14th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 246–251, Cambridge, MA, USA, November 2002.
- [ST87] William E. Snaman and David W. Thiel. The VAX/VMS distributed lock manager. *Digital Technical Journal*, 5:29–44, September 1987.
- [ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, Ottawa, Ontario, Canada, August 1995.
- [Ste92] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison Wesley, 1992.
- [Sut08] Herb Sutter. Effective concurrency. Series in Dr. Dobbs Journal, 2008.
- [Sut13] Adrian Sutton. Concurrent programming with the Disruptor. http://lca2013.linux.org.au/schedule/30168/view_talk, January 2013.
- [SW95] Richard L. Sites and Richard T. Witek. *Alpha AXP Architecture*. Digital Press, second edition, 1995.
- [The08] The Open MPI Project. MySQL Downloads. Available: <http://www.open-mpi.org/software/> [Viewed November 26, 2008], November 2008.
- [The11] The Valgrind Developers. Valgrind. <http://www.valgrind.org/>, November 2011.
- [The12] The OProfile Developers. Oprofile. <http://oprofile.sourceforge.net>, April 2012.

- [TMW11] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *Proceedings of the 2011 USENIX Annual Technical Conference*, pages 145–158, Portland, OR USA, June 2011. The USENIX Association.
- [Tor01] Linus Torvalds. Re: [Lse-tech] Re: RFC: patch to allow lock-free traversal of lists with insertion. Available: <http://lkml.org/lkml/2001/10/13/105> [Viewed August 21, 2004], October 2001.
- [Tor03] Linus Torvalds. Linux 2.6. Available: <ftp://kernel.org/pub/linux/kernel/v2.6> [Viewed June 23, 2004], August 2003.
- [Tra01] Transaction Processing Performance Council. TPC. Available: <http://www.tpc.org/> [Viewed December 7, 2008], 2001.
- [Tre86] R. K. Treiber. Systems programming: Coping with parallelism. *RJ* 5118, April 1986.
- [TS93] Hiroaki Takada and Ken Sakamura. A bounded spin lock algorithm with preemption. Technical Report 93-02, University of Tokyo, Tokyo, Japan, 1993.
- [TS95] H. Takada and K. Sakamura. Real-time scalability of nested spin locks. In *Proceedings of the 2Nd International Workshop on Real-Time Computing Systems and Applications*, RTCSA ’95, pages 160–, Washington, DC, USA, 1995. IEEE Computer Society.
- [Tur37] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 42 of 2, pages 230–265, 1937.
- [Ung11] David Ungar. Everything you know (about parallel programming) is wrong!: A wild screed about the future. In *Proceedings of the 2011 Systems, Programming Languages and Applications: Software for Humanity (SPLASH) Conference*, pages ???–???, Portland, OR, USA, October 2011.
- [Uni10] University of Maryland. Parallel maze solving. <http://www.cs.umd.edu/class/fall2010/cmsc433/p3/>, November 2010.
- [UoC08] Berkeley University of California. BOINC: compute for science. Available: <http://boinc.berkeley.edu/> [Viewed January 31, 2008], October 2008.
- [VGS08] Haris Volos, Neelam Goyal, and Michael M. Swift. Pathological interaction of locks with transactional memory. In *3rd ACM SIGPLAN Workshop on Transactional Computing*, New York, NY, USA, February 2008. ACM. Available: http://www.cs.wisc.edu/multifacet/papers/transact08_txlock.pdf [Viewed September 7, 2009].
- [Vog09] Werner Vogels. Eventually consistent. *Commun. ACM*, 52:40–44, January 2009.
- [Wei12] Frédéric Weisbecker. Interruption timer périodique. https://kernel-recipes.org/?page_id=410, 2012.
- [Wei13] Stewart Weiss. Unix lecture notes. Available: http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/unix_lecture_notes/ [Viewed April 8, 2014], May 2013.
- [Wik08] Wikipedia. Zilog Z80. Available: <http://en.wikipedia.org/wiki/Z80> [Viewed: December 7, 2008], 2008.
- [Wik12] Wikipedia. Labyrinth. <http://en.wikipedia.org/wiki/Labyrinth>, January 2012.
- [Wil12] Anthony Williams. *C++ Concurrency in Action: Practical Multithreading*. Manning, Shelter Island, NY, USA, 2012.
- [WKS94] Robert W. Wisniewski, Leonidas Konothanassis, and Michael L. Scott. Scalable spin locks for multiprogrammed systems. In *8th IEEE Int’l. Parallel Processing Symposium*, Cancun, Mexico, April 1994. The Institute of Electrical and Electronics Engineers, Inc.

- [WTS96] Cai-Dong Wang, Hiroaki Takada, and Ken Sakamura. Priority inheritance spin locks for multiprocessor real-time systems. In *Proceedings of the 2Nd International Workshop on Real-Time Computing Systems and Applications*, ISPAN '96, pages 70–76, Washington, DC, USA, 1996. IEEE Computer Society.
- [xen14] xenomai.org. Xenomai. <http://xenomai.org/>, December 2014.
- [Xu10] Herbert Xu. bridge: Add core IGMP snooping support. Available: <http://marc.info/?t=126719855400006&r=1&w=2> [Viewed March 20, 2011], February 2010.
- [Yod04a] Victor Yodaiken. Against priority inheritance. Available: <http://www.yodaiken.com/papers/inherit.pdf> [Viewed May 26, 2007], September 2004.
- [Yod04b] Victor Yodaiken. Temporal inventory and real-time synchronization in RTLinuxPro. Available: <http://www.yodaiken.com/papers/sync.pdf> [Viewed December 24, 2008], September 2004.

Appendix F

Credits

F.1 Authors

F.2 Reviewers

- Alan Stern (Section 14.2).
- Andy Whitcroft (Section 9.3.2, Section 9.3.4).
- Artem Bityutskiy (Section 14.2, Appendix C).
- Dave Keck (Chapter C).
- David S. Horner (Section 12.1.5).
- Gautham Shenoy (Section 9.3.2, Section 9.3.4).
- “jarkao2”, AKA LWN guest #41960 (Section 9.3.4).
- Jonathan Walpole (Section 9.3.4).
- Josh Triplett (Section 12).
- Michael Factor (Section 17.2).
- Mike Fulton (Section 9.3.2).
- Peter Zijlstra (Section 9.3.3).
- Richard Woodruff (Section C).
- Suparna Bhattacharya (Section 12).
- Vara Prasad (Section 12.1.5).

Reviewers whose feedback took the extremely welcome form of a patch are credited in the git logs.

F.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, Andy Whitcroft, Anton Blanchard, Chris McDermott, Cody Schaefer, Darrick Wong, David “Shaggy” Kleikamp, Jon M. Tollefson, Jose R. Santos, Marvin Heffler, Nathan Lynch, Nishanth Aravamudan, Tim Pepper, and Tony Breeds.

F.4 Original Publications

1. Section 2.4 (“What Makes Parallel Programming Hard?”) on page 13 originally appeared in a Portland State University Technical Report [MGM⁺09].
2. Section 6.5 (“Retrofitted Parallelism Considered Grossly Sub-Optimal”) on page 81 originally appeared in 4th USENIX Workshop on Hot Topics on Parallelism [McK12c].
3. Section 9.3.2 (“RCU Fundamentals”) on page 123 originally appeared in Linux Weekly News [MW07].
4. Section 9.3.3 (“RCU Usage”) on page 129 originally appeared in Linux Weekly News [McK08c].
5. Section 9.3.4 (“RCU Linux-Kernel API”) on page 138 originally appeared in Linux Weekly News [McK08b].
6. Appendix 12 (“Formal Verification”) on page 201 originally appeared in Linux Weekly News [McK07e, MR08, McK11c].

7. Section 12.3 (“Axiomatic Approaches”) on page 230 originally appeared in *Linux Weekly News* [MS14].
8. Appendix C.7 (“Memory-Barrier Instructions For Specific CPUs”) on page 337 originally appeared in *Linux Journal* [McK05a, McK05b].
26. Figure 14.2 (p 243) by Melissa Broussard.
27. Figure 14.6 (p 249) by David Howells.
28. Figure 14.7 (p 255) by David Howells.
29. Figure 14.8 (p 255) by David Howells.
30. Figure 14.9 (p 256) by David Howells.
31. Figure 14.10 (p 256) by David Howells.
32. Figure 14.11 (p 257) by David Howells.
33. Figure 14.12 (p 257) by David Howells.
34. Figure 14.13 (p 258) by David Howells.
35. Figure 14.14 (p 258) by David Howells.
36. Figure 14.15 (p 259) by David Howells.
37. Figure 14.16 (p 259) by David Howells.
38. Figure 14.17 (p 261) by David Howells.
39. Figure 14.18 (p 261) by David Howells.
40. Figure 15.1 (p 266) by Melissa Broussard.
41. Figure 15.2 (p 266) by Melissa Broussard.
42. Figure 15.3 (p 267) by Melissa Broussard.
43. Figure 15.10 (p 275) by Melissa Broussard.
44. Figure 15.11 (p 275) by Melissa Broussard.
45. Figure 15.14 (p 277) by Melissa Broussard.
46. Figure 15.19 (p 283) by Sarah McKenney.
47. Figure 15.20 (p 284) by Sarah McKenney.
48. Figure 16.2 (p 287) by Melissa Broussard.
49. Figure 17.1 (p 289) by Melissa Broussard.
50. Figure 17.2 (p 290) by Melissa Broussard.
51. Figure 17.3 (p 290) by Melissa Broussard.
52. Figure 17.4 (p 291) by Melissa Broussard.
53. Figure 17.8 (p 303) by Melissa Broussard.
54. Figure 17.9 (p 303) by Melissa Broussard.
55. Figure 17.10 (p 304) by Melissa Broussard.

F.5 Figure Credits

1. Figure 3.1 (p 17) by Melissa Broussard.
2. Figure 3.2 (p 18) by Melissa Broussard.
3. Figure 3.3 (p 18) by Melissa Broussard.
4. Figure 3.4 (p 19) by Melissa Broussard.
5. Figure 3.5 (p 19) by Melissa Broussard.
6. Figure 3.6 (p 20) by Melissa Broussard.
7. Figure 3.7 (p 20) by Melissa Broussard.
8. Figure 3.8 (p 20) by Melissa Broussard.
9. Figure 3.10 (p 22) by Melissa Broussard.
10. Figure 5.5 (p 39) by Melissa Broussard.
11. Figure 6.1 (p 61) by Korniliios Kourtis.
12. Figure 6.2 (p 62) by Melissa Broussard.
13. Figure 6.3 (p 62) by Korniliios Kourtis.
14. Figure 6.4 (p 63) by Korniliios Kourtis.
15. Figure 6.18 (p 73) by Melissa Broussard.
16. Figure 6.20 (p 74) by Melissa Broussard.
17. Figure 6.21 (p 74) by Melissa Broussard.
18. Figure 7.1 (p 90) by Melissa Broussard.
19. Figure 7.2 (p 90) by Melissa Broussard.
20. Figure 10.18 (p 166) by Melissa Broussard.
21. Figure 10.19 (p 167) by Melissa Broussard.
22. Figure 11.1 (p 181) by Melissa Broussard.
23. Figure 11.2 (p 181) by Melissa Broussard.
24. Figure 11.3 (p 186) by Melissa Broussard.
25. Figure 11.8 (p 199) by Melissa Broussard.

56. Figure 17.11 (p 304) by Melissa Broussard.
57. Figure A.4 (p 320) by Melissa Broussard.
58. Figure C.12 (p 343) by Melissa Brossard.
59. Figure D.3 (p 374) by Kornilios Kourtis.

F.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.