

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

February 12, 2017 (m)

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 © 2005-2017 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	11
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	14
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	18
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	21
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	24
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
4.2.5	Atomic Operations (gcc Classic)	34
4.2.6	Atomic Operations (C11)	34
4.2.7	Per-Thread Variables	35
4.3	Alternatives to POSIX Operations	35
4.3.1	Organization and Initialization	35
4.3.2	Thread Creation, Destruction, and Control	35
4.3.3	Locking	36
4.3.4	Atomic Operations	38
4.3.5	Per-CPU Variables	38
4.3.6	Performance	39
4.4	The Right Tool for the Job: How to Choose?	39
5	Counting	41
5.1	Why Isn't Concurrent Counting Trivial?	41
5.2	Statistical Counters	43
5.2.1	Design	43
5.2.2	Array-Based Implementation	43
5.2.3	Eventually Consistent Implementation	44
5.2.4	Per-Thread-Variable-Based Implementation	45
5.2.5	Discussion	46
5.3	Approximate Limit Counters	46
5.3.1	Design	46
5.3.2	Simple Limit Counter Implementation	47
5.3.3	Simple Limit Counter Discussion	51
5.3.4	Approximate Limit Counter Implementation	52
5.3.5	Approximate Limit Counter Discussion	52
5.4	Exact Limit Counters	52
5.4.1	Atomic Limit Counter Implementation	52
5.4.2	Atomic Limit Counter Discussion	55
5.4.3	Signal-Theft Limit Counter Design	56
5.4.4	Signal-Theft Limit Counter Implementation	56
5.4.5	Signal-Theft Limit Counter Discussion	59
5.5	Applying Specialized Parallel Counters	60
5.6	Parallel Counting Discussion	60
5.6.1	Parallel Counting Performance	61
5.6.2	Parallel Counting Specializations	61

5.6.3	Parallel Counting Lessons	62
6	Partitioning and Synchronization Design	65
6.1	Partitioning Exercises	65
6.1.1	Dining Philosophers Problem	65
6.1.2	Double-Ended Queue	69
6.1.3	Partitioning Example Discussion	75
6.2	Design Criteria	75
6.3	Synchronization Granularity	76
6.3.1	Sequential Program	76
6.3.2	Code Locking	78
6.3.3	Data Locking	78
6.3.4	Data Ownership	79
6.3.5	Locking Granularity and Performance	80
6.4	Parallel Fastpath	82
6.4.1	Reader/Writer Locking	82
6.4.2	Hierarchical Locking	83
6.4.3	Resource Allocator Caches	83
6.5	Beyond Partitioning	87
6.5.1	Work-Queue Parallel Maze Solver	87
6.5.2	Alternative Parallel Maze Solver	88
6.5.3	Performance Comparison I	89
6.5.4	Alternative Sequential Maze Solver	91
6.5.5	Performance Comparison II	91
6.5.6	Future Directions and Conclusions	92
6.6	Partitioning, Parallelism, and Optimization	93
7	Locking	95
7.1	Staying Alive	95
7.1.1	Deadlock	95
7.1.2	Livelock and Starvation	101
7.1.3	Unfairness	102
7.1.4	Inefficiency	103
7.2	Types of Locks	103
7.2.1	Exclusive Locks	103
7.2.2	Reader-Writer Locks	103
7.2.3	Beyond Reader-Writer Locks	104
7.2.4	Scoped Locking	105
7.3	Locking Implementation Issues	106
7.3.1	Sample Exclusive-Locking Implementation Based on Atomic Exchange	107
7.3.2	Other Exclusive-Locking Implementations	107
7.4	Lock-Based Existence Guarantees	109
7.5	Locking: Hero or Villain?	110
7.5.1	Locking For Applications: Hero!	110
7.5.2	Locking For Parallel Libraries: Just Another Tool	110
7.5.3	Locking For Parallelizing Sequential Libraries: Villain!	113
7.6	Summary	114

8 Data Ownership	115
8.1 Multiple Processes	115
8.2 Partial Data Ownership and <code>pthread</code> s	115
8.3 Function Shipping	116
8.4 Designated Thread	116
8.5 Privatization	116
8.6 Other Uses of Data Ownership	117
9 Deferred Processing	119
9.1 Running Example	119
9.2 Reference Counting	120
9.3 Hazard Pointers	122
9.4 Sequence Locks	125
9.5 Read-Copy Update (RCU)	128
9.5.1 Introduction to RCU	128
9.5.2 RCU Fundamentals	130
9.5.3 RCU Usage	136
9.5.4 RCU Linux-Kernel API	147
9.5.5 “Toy” RCU Implementations	151
9.5.6 RCU Exercises	163
9.6 Which to Choose?	163
9.7 What About Updates?	165
10 Data Structures	167
10.1 Motivating Application	167
10.2 Partitionable Data Structures	167
10.2.1 Hash-Table Design	168
10.2.2 Hash-Table Implementation	168
10.2.3 Hash-Table Performance	170
10.3 Read-Mostly Data Structures	171
10.3.1 RCU-Protected Hash Table Implementation	171
10.3.2 RCU-Protected Hash Table Performance	172
10.3.3 RCU-Protected Hash Table Discussion	174
10.4 Non-Partitionable Data Structures	175
10.4.1 Resizable Hash Table Design	175
10.4.2 Resizable Hash Table Implementation	176
10.4.3 Resizable Hash Table Discussion	179
10.4.4 Other Resizable Hash Tables	181
10.5 Other Data Structures	183
10.6 Micro-Optimization	184
10.6.1 Specialization	184
10.6.2 Bits and Bytes	184
10.6.3 Hardware Considerations	185
10.7 Summary	186
11 Validation	187
11.1 Introduction	187
11.1.1 Where Do Bugs Come From?	187
11.1.2 Required Mindset	188
11.1.3 When Should Validation Start?	189

11.1.4 The Open Source Way	190
11.2 Tracing	191
11.3 Assertions	192
11.4 Static Analysis	192
11.5 Code Review	192
11.5.1 Inspection	192
11.5.2 Walkthroughs	193
11.5.3 Self-Inspection	193
11.6 Probability and Heisenbugs	194
11.6.1 Statistics for Discrete Testing	195
11.6.2 Abusing Statistics for Discrete Testing	197
11.6.3 Statistics for Continuous Testing	197
11.6.4 Hunting Heisenbugs	198
11.7 Performance Estimation	201
11.7.1 Benchmarking	201
11.7.2 Profiling	201
11.7.3 Differential Profiling	202
11.7.4 Microbenchmarking	202
11.7.5 Isolation	202
11.7.6 Detecting Interference	203
11.8 Summary	205
12 Formal Verification	207
12.1 General-Purpose State-Space Search	207
12.1.1 Promela and Spin	207
12.1.2 How to Use Promela	210
12.1.3 Promela Example: Locking	212
12.1.4 Promela Example: QRCU	213
12.1.5 Promela Parable: dynticks and Preemptible RCU	218
12.1.6 Validating Preemptible RCU and dynticks	221
12.2 Special-Purpose State-Space Search	233
12.2.1 Anatomy of a Litmus Test	233
12.2.2 What Does This Litmus Test Mean?	234
12.2.3 Running a Litmus Test	235
12.2.4 PPCMEM Discussion	235
12.3 Axiomatic Approaches	236
12.4 SAT Solvers	237
12.5 Summary	238
13 Putting It All Together	241
13.1 Counter Conundrums	241
13.1.1 Counting Updates	241
13.1.2 Counting Lookups	241
13.2 Refurbish Reference Counting	241
13.2.1 Implementation of Reference-Counting Categories	242
13.2.2 Linux Primitives Supporting Reference Counting	246
13.2.3 Counter Optimizations	246
13.3 RCU Rescues	247
13.3.1 RCU and Per-Thread-Variable-Based Statistical Counters	247
13.3.2 RCU and Counters for Removable I/O Devices	249

13.3.3 Array and Length	249
13.3.4 Correlated Fields	250
13.4 Hashing Hassles	250
13.4.1 Correlated Data Elements	250
13.4.2 Update-Friendly Hash-Table Traversal	251
14 Advanced Synchronization	253
14.1 Avoiding Locks	253
14.2 Memory Barriers	253
14.2.1 Memory Ordering and Memory Barriers	254
14.2.2 If B Follows A, and C Follows B, Why Doesn't C Follow A?	255
14.2.3 Variables Can Have More Than One Value	255
14.2.4 What Can You Trust?	256
14.2.5 Review of Locking Implementations	261
14.2.6 A Few Simple Rules	262
14.2.7 Abstract Memory Access Model	262
14.2.8 Device Operations	263
14.2.9 Guarantees	263
14.2.10 What Are Memory Barriers?	264
14.2.11 Locking Constraints	272
14.2.12 Memory-Barrier Examples	273
14.2.13 The Effects of the CPU Cache	275
14.2.14 Where Are Memory Barriers Needed?	276
14.3 Non-Blocking Synchronization	277
14.3.1 Simple NBS	278
14.3.2 NBS Discussion	278
15 Parallel Real-Time Computing	281
15.1 What is Real-Time Computing?	281
15.1.1 Soft Real Time	281
15.1.2 Hard Real Time	281
15.1.3 Real-World Real Time	282
15.2 Who Needs Real-Time Computing?	285
15.3 Who Needs Parallel Real-Time Computing?	286
15.4 Implementing Parallel Real-Time Systems	286
15.4.1 Implementing Parallel Real-Time Operating Systems	287
15.4.2 Implementing Parallel Real-Time Applications	297
15.4.3 The Role of RCU	299
15.5 Real Time vs. Real Fast: How to Choose?	300
16 Ease of Use	301
16.1 What is Easy?	301
16.2 Rusty Scale for API Design	301
16.3 Shaving the Mandelbrot Set	302
17 Conflicting Visions of the Future	305
17.1 The Future of CPU Technology Ain't What it Used to Be	305
17.1.1 Uniprocessor Über Alles	305
17.1.2 Multithreaded Mania	306
17.1.3 More of the Same	307

17.1.4 Crash Dummies Slamming into the Memory Wall	307
17.2 Transactional Memory	308
17.2.1 Outside World	309
17.2.2 Process Modification	312
17.2.3 Synchronization	315
17.2.4 Discussion	318
17.3 Hardware Transactional Memory	319
17.3.1 HTM Benefits WRT to Locking	320
17.3.2 HTM Weaknesses WRT Locking	321
17.3.3 HTM Weaknesses WRT to Locking When Augmented	325
17.3.4 Where Does HTM Best Fit In?	328
17.3.5 Potential Game Changers	328
17.3.6 Conclusions	330
17.4 Functional Programming for Parallelism	330
A Important Questions	333
A.1 What Does “After” Mean?	333
A.2 What is the Difference Between “Concurrent” and “Parallel”?	335
A.3 What Time Is It?	336
B Why Memory Barriers?	337
B.1 Cache Structure	337
B.2 Cache-Coherence Protocols	339
B.2.1 MESI States	339
B.2.2 MESI Protocol Messages	339
B.2.3 MESI State Diagram	340
B.2.4 MESI Protocol Example	341
B.3 Stores Result in Unnecessary Stalls	341
B.3.1 Store Buffers	342
B.3.2 Store Forwarding	342
B.3.3 Store Buffers and Memory Barriers	343
B.4 Store Sequences Result in Unnecessary Stalls	345
B.4.1 Invalidate Queues	345
B.4.2 Invalidate Queues and Invalidate Acknowledge	345
B.4.3 Invalidate Queues and Memory Barriers	346
B.5 Read and Write Memory Barriers	347
B.6 Example Memory-Barrier Sequences	348
B.6.1 Ordering-Hostile Architecture	348
B.6.2 Example 1	349
B.6.3 Example 2	349
B.6.4 Example 3	349
B.7 Memory-Barrier Instructions For Specific CPUs	351
B.7.1 Alpha	352
B.7.2 AMD64	354
B.7.3 ARMv7-A/R	354
B.7.4 IA64	354
B.7.5 MIPS	355
B.7.6 PA-RISC	355
B.7.7 POWER / PowerPC	355
B.7.8 SPARC RMO, PSO, and TSO	356

B.7.9	x86	357
B.7.10	zSeries	358
B.8	Are Memory Barriers Forever?	358
B.9	Advice to Hardware Designers	358
C	Read-Copy Update in Linux	361
C.1	RCU Usage Within Linux	361
C.2	RCU Evolution	361
C.2.1	2.6.27 Linux Kernel	361
C.2.2	2.6.28 Linux Kernel	362
C.2.3	2.6.29 Linux Kernel	362
C.2.4	2.6.31 Linux Kernel	363
C.2.5	2.6.32 Linux Kernel	363
C.2.6	2.6.33 Linux Kernel	363
C.2.7	2.6.34 Linux Kernel	363
C.2.8	2.6.35 Linux Kernel	364
C.2.9	2.6.36 Linux Kernel	364
C.2.10	2.6.37 Linux Kernel	364
C.2.11	2.6.38 Linux Kernel	364
C.2.12	2.6.39 Linux Kernel	364
C.2.13	3.0 Linux Kernel	364
C.2.14	3.1 Linux Kernel	365
C.2.15	3.2 Linux Kernel	365
C.2.16	3.3 Linux Kernel	365
C.2.17	3.4 Linux Kernel	365
C.2.18	3.5 Linux Kernel	366
C.2.19	3.6 Linux Kernel	366
C.2.20	3.7 Linux Kernel	366
C.2.21	3.8 Linux Kernel	366
C.2.22	3.9 Linux Kernel	367
C.2.23	3.10 Linux Kernel	367
C.2.24	3.11 Linux Kernel	367
C.2.25	3.12 Linux Kernel	367
C.2.26	3.13 Linux Kernel	367
C.2.27	3.14 Linux Kernel	368
D	Answers to Quick Quizzes	369
D.1	How To Use This Book	369
D.2	Introduction	370
D.3	Hardware and its Habits	373
D.4	Tools of the Trade	376
D.5	Counting	381
D.6	Partitioning and Synchronization Design	393
D.7	Locking	398
D.8	Data Ownership	404
D.9	Deferred Processing	406
D.10	Data Structures	422
D.11	Validation	424
D.12	Formal Verification	429
D.13	Putting It All Together	433

D.14 Advanced Synchronization	436
D.15 Parallel Real-Time Computing	438
D.16 Ease of Use	440
D.17 Conflicting Visions of the Future	440
D.18 Important Questions	443
D.19 Why Memory Barriers?	443
E Glossary and Bibliography	447
F Credits	475
F.1 Authors	475
F.2 Reviewers	475
F.3 Machine Owners	475
F.4 Original Publications	475
F.5 Figure Credits	476
F.6 Other Support	477

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 성당이라기보다 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.

¹ 또는, 좀 더 정확히는, 병렬성 없는 프로그래밍에 비해 정확성을 훨씬 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” [McK12b].

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 B 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 369 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. 어떠하죠?

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!

마지막으로, 가장 흔한 학습 장애는 당신이 이미 알고 있다고 생각하는 것입니다. 퀴즈들은 그걸 낫게 하는데 매우 효과적일 수 있습니다.

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.

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

3. 객체 지향 패턴 전문가들이 병렬 프로그래밍을 어떻게 취급하는지 C++ 위주로 알고 싶다면 Schmidt의 POSA 시리즈 [SSRB00, BHS07] 의 Volume 2 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.
4. 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.
5. 당신의 주요 관심사가 과학 / 기술 분야 컴퓨팅이 라면, 그리고 패턴주의자의 접근방법을 선호한다면, Mattson의 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.
6. 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.
7. 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.
8. C++11 에 관심 있다면, Anthony Williams 의 “C++ Concurrency in Action: Practical Multithreading” [Wil12] 를 좋아할 수 있을 겁니다.
9. 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.
10. Intel Threading Building Blocks 를 사용해 보고 싶다면, James Reinder 의 책 [Rei07] 이 아마 찾으시는 책일 겁니다.
11. 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.
12. 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 9.5.5 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` 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-BUILD.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

```

1 git clone git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git
2 cd perfbook
3 # You may need to install a font here. See item 1 in FAQ.txt.
4 make
5 evince perfbook.pdf & # Two-column version
6 make perfbook-1c.pdf
7 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

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

이 책의 PDF 들은 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/SubmittingPatches 에 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. 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) 1.1 과 유사합니다. 다만 한가지, item #4 가 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 license 로 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 다뤄 왔습니다.

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.

If parallel programming is so hard, why are there any parallel programs?

Unknown

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 [Tor03c], 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 퀄리티의 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. 그러니 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. 뭘 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.

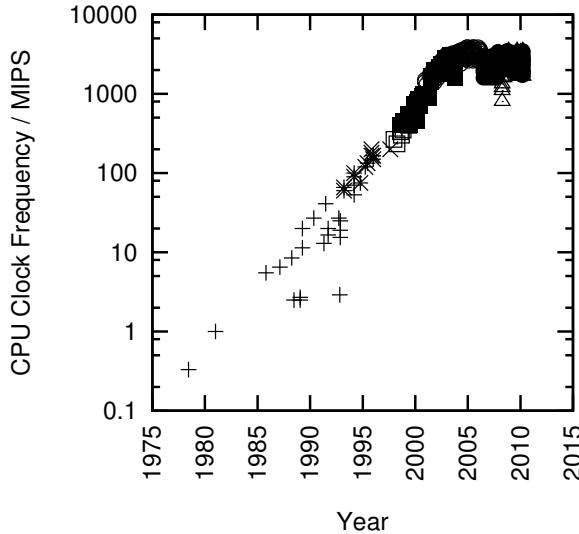


Figure 2.1: MIPS/Clock-Frequency Trend for Intel CPUs

그친 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 있다고 하면 또 답이 달라질겁니다.

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.

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, Dep06]에 운영되었습니다. 이 머신은 트랜지스터 시대 이전에 만들어졌기 때문에, 2,000개의 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 있었습니다. Z80 CPU는 8,500 개의 트랜지스터를 사용했고, 2008년도에 개당 \$2 US에 1,000개 단위로 구매할 수 있었습니다. CSIRAC 와 반대로 Z80에서 소프트웨어 개발 비용은 그렇게 크게 중요하지 않았습니다.

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

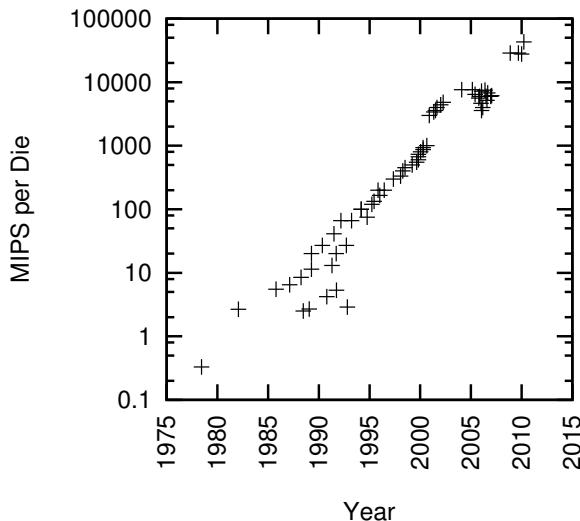


Figure 2.2: MIPS per Die for Intel CPUs

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

훌륭한 성능, 생산성, 그리고 generality 를 제공하는 병렬 프로그래밍 환경의 천국은 아직 존재하지 않습니다. 그런 천국이 오기 전까지는, 성능, 생산성, 그리고

³ 이걸 지적해준 Michael Wong 에게 찬사를.

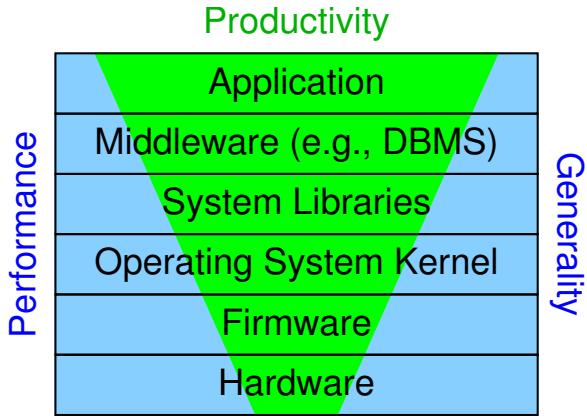


Figure 2.3: Software Layers and Performance, Productivity, and Generality

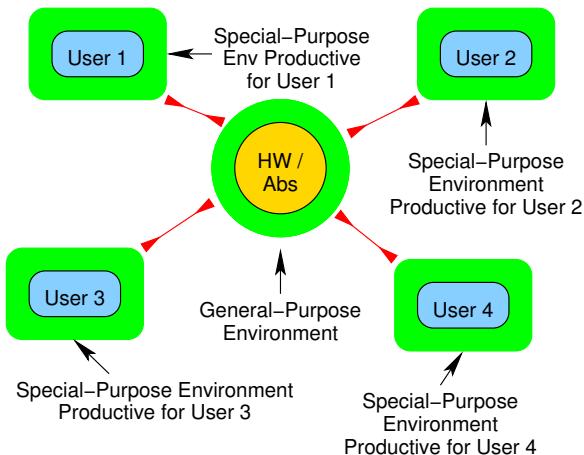


Figure 2.4: Tradeoff Between Productivity and Generality

generality 사이에서 엔지니어링적 트레이드오프를 가질 수밖에 없습니다. 그런 트레이드오프의 한 예가 Figure 2.3에 있습니다; 이 그림은 시스템 스택의 상위 레이어로 갈수록 생산성은 점점 중요해지고 성능과 generality는 하위 레이어로 갈수록 중요해진다는 점을 보여줍니다. 하위 레이어에서 발생하는 거대한 개발 비용은 똑같이 거대한 사용자 수로 나뉘어져야 하고(그래서 generality가 중요하죠), 하위 레이어에서 잊은 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 generality가 주요 관심사인, 스택의 바닥 근처에 있는 개발자를 대상으로 합니다.

생산성과 generality 사이에서의 트레이드오프가 많은 영역에서 100여년간 존재했음을 알아둘 필요가 있습니다. 하나만 예를 들자면, 뜻을 박는데 망치보다 레일건이 생산적이지만 망치는 뜻박는 거 말고도 많은 영역에 사용될 수 있죠. 그러니 병렬 컴퓨팅에서도 비슷한 트레이드오프들을 발견하는건 그다지 놀라운 일이 아닙니다. 이런 트레이드오프는 추상적으로 Figure 2.4에 나타나 있습니다. 여기서 user 1, 2, 3, 그리고 4는 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.

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

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

다른 선택지는 Figure 2.4의 가운데 영역처럼 프로그래밍 언어나 환경을 주어진 (assembly, C, C++, 또는 Java 같은 로우 레벨 언어들처럼) 하드웨어나 (Haskell, Prolog, 또는 Snobol처럼) 추상화에 맞추는 것입니다. 이런 언어들은 user 1, 2, 3, 4에 의해 요청된 작업들에 대해 모두 똑같이 맞춰져 있지 않다는 점에서 일반적이라고 할 수 있습니다. 다시 말해, 그런 언어들의 generality는 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, generality라는 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 영

향을 끼치는 코드를 만지는 개발자들을 대상으로 하고 있기 때문에, 이 섹션의 나머지 부분은 성능 개선 쪽에 포커스를 맞춥니다.

병렬성을 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 않으면서도 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, 혼한 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) 와 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. 더우기, 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 링크드리스트를 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

지점은 가질 겁니다. 예를 들어, 당신의 프로그램이 대부분의 시간을 디스크 드라이브로부터 읽어들이는 데 데이터가 도착하기를 기다리고 있다면, 여러 CPU를 사용하는 것은 그저 디스크로부터 데이터를 기다리는 시간을 증가시키기만 할 수도 있습니다. 실제로, 프로그램이 하드디스크처럼 회전하는 디스크에 시퀀셜하게 써여 있는 하나의 큰 파일을 읽으려 할 때, 그 프로그램을 병렬화하면 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 병렬화하기가 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 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 포커스되어 있었고, 따라서 일반적인 결과를 내놓지는 못했습니다. 더욱이, 일반적인 프로그래머 생산성의 범위가 10배 이상도

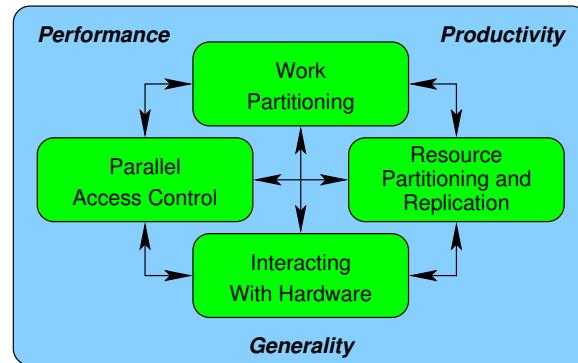


Figure 2.5: Categories of Tasks Required of Parallel Programmers

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 가치있지만, 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 높일 수가 있습니다. 예를 들어, 파티셔닝은 글로벌한 애러와 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 베릴 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 컴퓨터이션과 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 당황스럽게 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) 문제는 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 문제로부터 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 연결되어 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 컴포넌

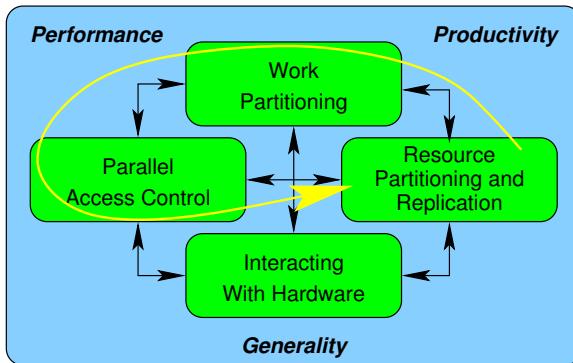


Figure 2.6: Ordering of Parallel-Programming Tasks

트를 바탕으로 일하는 개발자들같은 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 업

데이트들의 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 여러곳에서 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, Imm85]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 하나님의 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 클락 사이클을 소모했습니다. 대조적으로, 1990년대 후반에서 2000년대 초반의 CPU 는

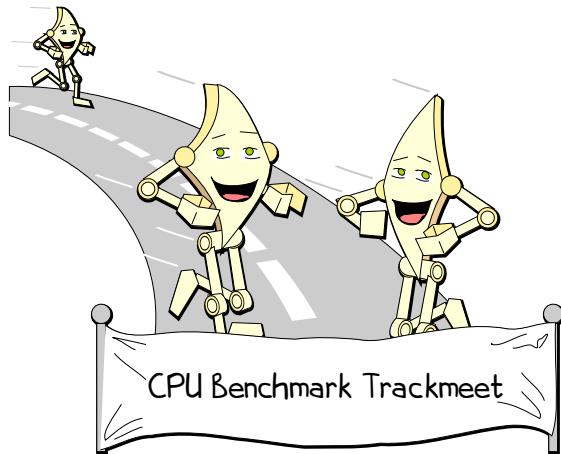


Figure 3.1: CPU Performance at its Best

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 구성된 프로그램을 생각해보세요. 또 다른 예로, 수많은

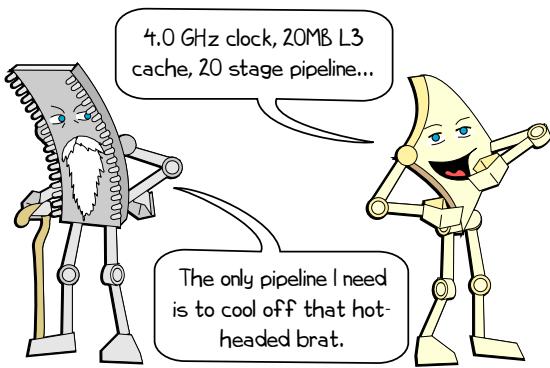


Figure 3.2: CPUs Old and New

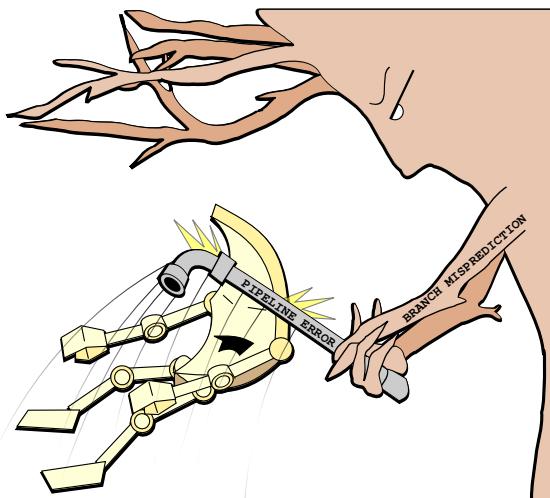


Figure 3.3: CPU Meets a Pipeline Flush

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 인스트럭션들을 모두 폐기시켜야 하기 때문에 파이프라인 플러시를 일으키기 때문에 잘못된 예측은 매우 비싼 비용을 지불합니다. 만약 파이프라인 플러시가 너무 자주

일어난다면, Figure 3.3에서 그려진 것처럼 엄청난 성능 하락을 겪을 수 있습니다.

불행히도, 파이프라인 플러시는 근래의 CPU 들이 달려야 하는 장애물 코스의 유일한 위험이 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 Moore의 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 만들니다만, 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 3.4에서 보여지듯, 메모리 참조는 종종 근래의 CPU 들에게 거대한 장애물입니다.

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

3.1.3 Atomic Operations

그런 장애 중 하나는 어토믹 오퍼레이션들입니다. 여기서의 문제는 모든 어토믹 오퍼레이션은 CPU 파이프라

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

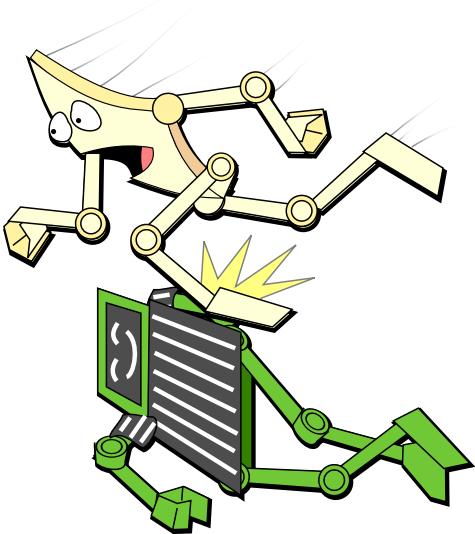


Figure 3.4: CPU Meets a Memory Reference

인의 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, 그리고 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 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

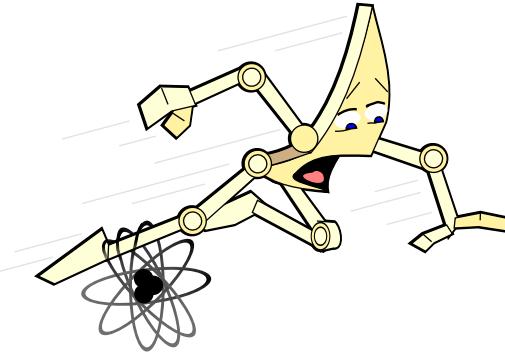


Figure 3.5: CPU Meets an Atomic Operation

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 14.2와 Appendix B에서 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 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



Figure 3.6: CPU Meets a Memory Barrier

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 CPU 가 한 변수를 수정하려 할 때, 다른 CPU 가 그 값을 최근에 바꾼 경우가 있을 가능성이 크기 때문이죠. 이런 경우, 해당 변수는 지금 수정하려는 CPU 의 캐시가 아니라 최근에 값을 수정한 CPU 의 캐시에 있을 것이고, 이로 인해 비싼 캐시 미스(더 자세한 내용을 위해선 Section B.1 를 보세요)를 일으킬 것입니다. 이런 캐시 미스들은 Figure 3.7 에서 보여진 것처럼 CPU 성능의 주요 장애가 됩니다.

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

3.1.6 I/O Operations

캐시 미스는 곧 CPU 와 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 3.8 에서 보여지듯, 앞의 섹션들에서 이야기 되었던 내부적 장애들보다 훨씬 커다란 장애를 야기합니다.

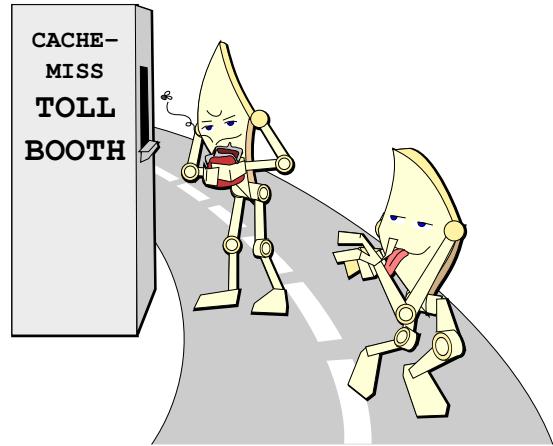


Figure 3.7: CPU Meets a Cache Miss

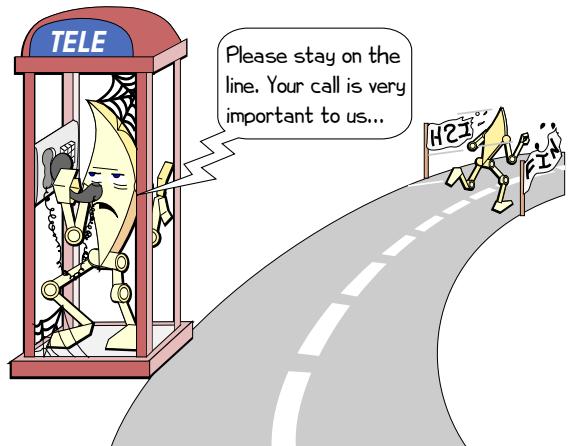


Figure 3.8: CPU Waits for I/O Completion

이전 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 커뮤니케이션 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 6 에서 볼테지만, 병렬 소프트웨어 설계의 중요한 목표는 커뮤니케이션 캐시 미스와 같은 비싼 동작들의 빈번도

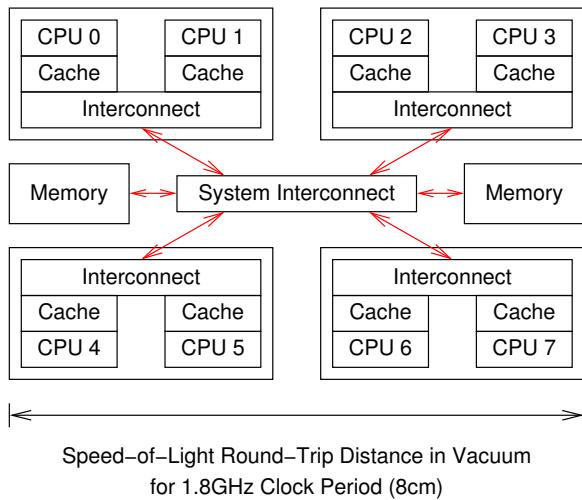


Figure 3.9: System Hardware Architecture

를 낮추는 것입니다.

물론, 주어진 동작이 장애라는 이야기는 그 이야기이고, 그 동작이 심각한 장애라는 건 또 따로 보여줘야하지요. 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 다이어그램은 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 interconnect 는 네개의 다이어그램을 통신할 수 있도록 하고, 메인 메모리로 각 다이어그램을 연결합니다.

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

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

1. CPU 0 는 자신의 로컬 캐시를 확인하고, 그 캐시라인이 없음을 발견합니다.
2. 요청은 CPU 0 와 CPU 1 사이의 연결로 넘겨져서 CPU 1 의 로컬 캐시를 확인해봅니다만, 역시 해당 캐시라인이 없음을 발견합니다.
3. 요청은 시스템 연결부로 넘겨지고, 다른 세개의 다이어그램을 체크합니다. 결과, 해당 캐시라인은 CPU 6 과 CPU 7 이 위치한 다이어그램에 있음을 확인합니다.
4. 요청은 CPU 6 과 CPU 7 연결부로 넘겨져 각 CPU 의 캐시를 확인해 해당 캐시 라인이 CPU 7 의 캐시에 있음을 발견합니다.
5. CPU 7 은 해당 캐시라인을 자신의 연결부로 넘기고, 자신의 캐시에서 해당 캐시라인을 비워버립니다.
6. CPU 6 와 CPU 7 의 연결부는 해당 캐시 라인을 시스템 연결부로 넘깁니다.
7. 시스템 연결부는 해당 캐시 라인을 CPU 0 과 CPU 1 연결부로 넘깁니다.
8. CPU 0 과 CPU 1 연결부는 해당 캐시라인을 CPU 0 의 캐시로 보냅니다.
9. CPU 0 는 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 7 의 캐시에서 해당 캐시라인을 비워야 하죠?

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

3.2.2 Costs of Operations

병렬 프로그램에 중요하고 흔히 사용되는 오퍼레이션의 오버헤드들이 Table 3.1 에 표시되어 있습니다. 이 시스템의 클락 시간은 약 0.6ns 입니다. 많은 근래의 마이크로프로세서가 한 클락 시간동안 여러 인스트럭션을 수행 완료시킬 수 있긴 합니다만, 모든 오퍼레이션의 비용은 클락 시간을 기준으로 해서 “Ratio” 라고 표시한

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	5,000	8,330
Global Comms	195,000,000	325,000,000

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

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 기존 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 상황을 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, InfiniBand 나 몇몇 독점 연결장치와 같은 고성능의 (그리고 고비용의!) 통신 장치들은 5 천개의 인스트럭션이 수행될 수도 있는, 약

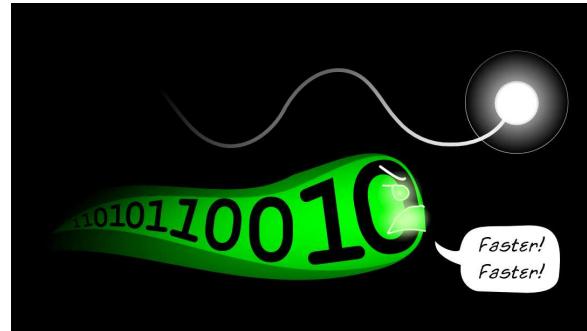


Figure 3.10: Hardware and Software: On Same Side

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 시

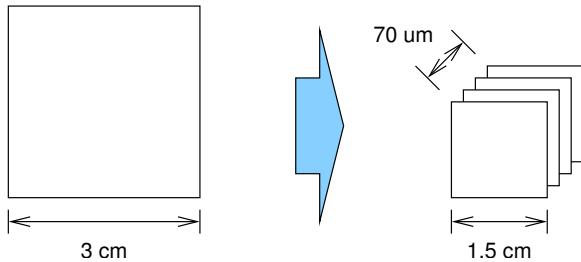


Figure 3.11: Latency Benefit of 3D Integration

스템의 크기에 비교해 보면 둘 다 비교적 작은 거리입니다.

상황이 더 나빠지는게, 실리콘에서의 전자파는 전공에서의 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 CPU 와 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 말도 말고 약간만 있습니다:

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

이것들 각각을 다음 섹션에서 설명합니다.

3.3.1 3D Integration

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

3DI 가 가져올 수 있는 가장 중요한 이점은 Figure 3.11 에 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 것을 고려하면 이론적으로 시스템을 관통하는 최대 경로의 길이가 두배 가까이 줄어든 셈입니다. 또한, 설계와 배치에 충분한 고려를 한다면, (느리고 전력도 많이 소모할) 수평적인 전자적 연결들은 빠르기도 하고 전력 소모도 적은, 짧은 수직적 전자적 연결로 대체될 수 있을 것입니다.

하지만, 클락으로 동작하는 논리회로의 수준들로 인한 지연은 3D 융합으로 감소되진 못할 것이고, 상기한 장점들을 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 하는건 가능하겠지만, 이런 근본적 한계들을 비켜나가는데 집중해온 연구와 개발 과정에서 얻어진 몇가지 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 한결음 더 나아가게 하는데 도움을 줄수도 있습니다. 또다른 좀 더 신기한 회피법은 하나의 전자는 여러 에너지 레벨을 가질 수 있다는 점을 이용해 여러 비트들을 하나의 전자에 저장하는 것입니다. 이 방법이 상품화된 반도체 기기에서 안정적으로 동작하게 될 수 있을지는 아직 확실하지 않습니다.

또다른 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 없는 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 않은 채)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.

이제 현존하는 많은 상용 마이크로프로세서에 존재하는 벡터 연산 명령어들의 실제 모티베이션이 되었습니다. 이런 명령어들은 여러 데이터 항목들에 동시에 으로 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 그 외에도 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 하도

록 그 하드웨어의 윗단 프론트엔드 설계 비용이 충분히 많은 사용자에게 나뉘어 질 수 있어야만 합니다. 부분적으로 이런 부류의 경제적 고려사항 때문에 특수화된 하드웨어는 그래픽 처리 (GPU), 벡터 처리기 (MMX, SSE, 그리고 VMX 명령어들), 그리고 암호화 등의 적은 어플리케이션 분야에만 나타나온 했습니다.

서버와 PC 분야와 달리, 스마트폰은 다양한 하드웨어 가속기를 사용해 왔습니다. 이런 하드웨어 가속기는 CPU가 완전히 잠든 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 CPU보다 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 실제로 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 동시에 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 쓰레드들 간에 통신을 하기 위해 CAS를 사용한다고 생각해 보세요. 이 CAS 오퍼레이션들은 쓰레드들이 자기 혼자 하는 게 아니라 다른 쓰레드들과 통신을 하기 위해 사용하는 것이기 때문에 자

주 캐시 미스를 낼 것입니다. 더 나아가서 각각의 CAS 통신 오퍼레이션에 뒤따르는 일의 단위가 부동 소수점 연산 작업 정도는 충분히 할 수 있는 시간인 300ns 인 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 극 단적으로 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 그런 시스템을 쓰려 하는 건가요? ■

교훈은 분명합니다: 병렬 알고리즘들은 이런 하드웨어 특성을 분명히 마음 속에 기억해 둔 채 분명하게 설계되어야만 합니다. 그런 한가지 방법은 거의 독립적인 쓰레드들을 수행시키는 것입니다. 어토믹 오퍼레이션을 사용하든, 락이나 명시적 메세지를 사용하든, 쓰레드들의 커뮤니케이션이 덜 빈번할수록 어플리케이션의 성능과 확장성은 나아질 것입니다. 이런 방법은 Chapter 5에서 간단히 알아보고, Chapter 6에서 자세히 알아본 후, 그 논리적 극단에 대해 Chapter 8에서 알아봅니다.

또 다른 방법은 공유된 것들에 가해지는 접근은 읽기가 대부분이도록 하는 것인데, 이렇게 되면 CPU들이 캐시에 읽기만 대부분 가해지는 데이터를 복사해 둘 수 있게 해서, 모든 CPU들이 빠른 접근을 할 수 있게 합니다. 이런 방법은 Section 5.2.3에서 간단히 알아보고, Chapter 9에서 좀 더 깊게 다뤄봅니다.

요약하자면, 훌륭한 병렬 성능과 확장성을 달성하는 것은 조심스럽게 데이터 구조와 알고리즘을 선택해 든, 존재하는 병렬 어플리케이션과 환경을 사용해 서든, 또는 문제를 당황스럽도록 병렬적인 해결책이 존재하는 문제로 변환해 서든 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

You are only as good as your tools, and your tools are only as good as you are.

Unknown

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 다뤄보고, Section 4.3은 다른 환경들에서의 비슷한 오퍼레이션들을 선보이며, 마지막으로, Section 4.4에서는 일을 완료하기 위해 어떤 도구를 골라야 할지 선택을 도와드립니다.

Quick Quiz 4.1: 이것들을 도구라고 하셨나요??? 제게 이것들은 도구라기보다는 낮은 단계의 동기화 기능들처럼 보이는데요! ■

이 챕터는 간략한 소개만을 제공한다는 점을 기억해 두세요. 더 자세한 내용은 인용된 참조 목록들에서 볼 수 있으며, 이 도구들을 어떻게 사용하는게 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

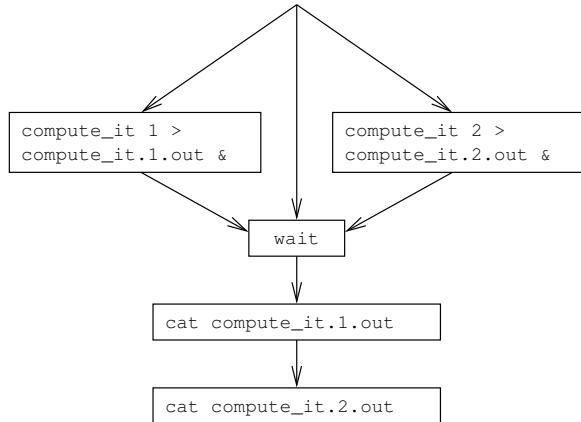


Figure 4.1: Execution Diagram for Parallel Shell Execution

Figure 4.1에 나타난 대로입니다: `compute_it`의 두 개의 인스턴스는 병렬적으로 수행되고, `wait`은 두개 인스턴스 모두가 완료된 뒤에 실행 완료되면, 그 뒤에 `cat`의 두개의 인스턴스가 순차적으로 수행됩니다.

Quick Quiz 4.2: 하지만 이 간단한 셸 스크립트는 진짜 병렬 프로그램이 아니잖아요! 왜 이런 별거아닌 결신경쓰는거죠??? ■

Quick Quiz 4.3: 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

니다.

Quick Quiz 4.4: 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 리턴하는 값은 Figure 4.2(forkjoin.c)에 보여진 것처럼

```

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

콜러가 그 차이를 알 수 있게 합니다. 라인 1은 fork() 함수를 실행하고, 그 리턴값을 지역 변수 pid에 저장합니다. 라인 2에서는 pid가 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 pid와 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 셸 스크립트에서 사용할 때보다 좀 더 복잡합니다. 그래서 Figure 4.3(api-pthread.h)에서 보여진, 셸 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 errno가 ECHILD로 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 4.5: 왜 이 wait() 함수는 그렇게 복잡해야만 하는거죠? 왜 그냥 셸 스크립트의 wait 같이

```

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

```

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

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

가장 잘게 크리티컬 섹션을 쪼갠 병렬성은 공유 메모리를 필요로 하며, 이는 Section 4.2.2에서 다룹니다. 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, 두번째로 예제에서는 `NULL` 값을 준 인자는 필요하면 추가하게 되는 `pthread_attr_t` 타입 변수로의 포인터이며, 세번째 인자는 새로 생성된 쓰레드에 의해 호출될 함수(이 경우, `mythread()`)이고, 마지막으로 여기선 `NULL`을 준 인자는 `mythread`에게 전달될 인자입니다.

```

1 int x = 0;
2
3 void *mythread(void *arg)
4 {
5     x = 1;
6     printf("Child process set x=1\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

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

Quick Quiz 4.7: Figure 4.5의 `mythread()` 함수가 그냥 리턴해도 된다면, 왜 `pthread_exit()`를 신경 써야하죠?

라인 20에 있는 `pthread_join()` 함수는 fork-join에서의 `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 한번에 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 보

장도 하지 않기 때문에, 우리는 다음 섹션에서 이야기할 락킹 도구들과 같이 데이터에의 동시적 접근과 수정을 안전하게 할 수 있는 방법이 필요합니다.

Quick Quiz 4.8: C 언어가 데이터 레이스에 대해 어떤 보장도 하지 않는다면, 왜 리눅스 커널은 그렇게 많은 데이터 레이스들을 가지고 있는거죠? 지금 리눅스 커널이 완전 엉망이라고 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. 간단하고 합리적 수준으로 유용한 프로그래밍 모델은 특정 데이터 아이템이 연관된 락을 잡고 있을 때에만 접근될 수 있도록 합니다 [Hoa74].

Quick Quiz 4.9: 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 초기화 하고, 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 캐스팅 합니다.

Quick Quiz 4.10: 왜 그냥 Figure 4.6 라인 5에서 `lock_reader()`가 곧바로 `pthread_mutex_t` 포인

```

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 = READ_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        WRITE_ONCE(x, READ_ONCE(x) + 1);
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

터를 받도록 하지 않는거죠?

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

Quick Quiz 4.11: `pthread_mutex_t`의 획득과 해제에 매번 4줄이나 써야한다니 좀 고통스러울 것 같군요! 더 나은 방법은 없나요?

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

Figure 4.7는 `lock_reader()`와 `lock_writer()`를 같은 `lock_a` 락을 사용하도록 하면서 쓰레드로 실행시키는 코드를 보여줍니다. 라인 2-6은 `lock_reader()`를 실행하는 쓰레드를 생성하고, 라인 7-11에서는 `lock_writer()`를 실행하는 쓰레드를 생성합니다. 라인 12-19에서는 두 쓰레드가 완료되기를 기다립니다. 이 코드가 내놓는 결과는 다음과 같습니다:

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

```

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

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

Figure 4.8에서 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` 중간 값을 볼 수 있습니다.

Quick Quiz 4.13: 서로 다른 락을 사용하는건 쓰레드가 서로 상대의 중간 상태를 볼 수 있는등 혼란스럽게 할 수 있는 것같은데요. 잘 짜여진 복잡 프로그램은 이런 혼란을 막기 위해서는 하나의 락만을 사용해야만 하는 건가요?

Quick Quiz 4.14: Figure 4.8에 보여진 코드에서, `lock_reader()`는 `lock_writer()` 가 생성하는 값 모두를 보도록 보장되어 있나요? 그렇다면, 또 그렇지

않다면, 왜죠?

Quick Quiz 4.15: 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 전 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 에서는 언제 모든 리더 쓰레드들이 수행을 시작했는지 알려주는 `nreadersrunning` 변수를 정의합니다.

라인 7-10 은 테스트의 시작과 끝을 맞춰주는 `goflag` 를 정의합니다. 이 변수는 처음엔 `GOFLAG_`

```

1 pthread_rwlock_t rwl = PTHREAD_RWLOCK_INITIALIZER;
2 int holdtime = 0;
3 int thinktime = 0;
4 long long *readcounts;
5 int nreadersrunning = 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(&nreadersrunning, 1);
19     while (READ_ONCE(goflag) == GOFLAG_INIT) {
20         continue;
21     }
22     while (READ_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

INIT로 설정되어 있고, 이후에 모든 리더 쓰레드들이 시작된 이후에 GOFLAG_RUN으로 변경된 후, 마지막으로 테스트를 종료시키기 위해 GOFLAG_STOP으로 값을 바꿉니다.

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

Quick Quiz 4.16: 여기 저기 모든 곳에서 READ_ONCE()를 쓰는 대신에, Figure 4.9의 라인 10에서 goflag를 volatile로 선언하는게 어때요?

Quick Quiz 4.17: READ_ONCE()는 컴파일러에만 영향을 주지, CPU에는 영향을 안주죠. Figure 4.9의 goflag의 값의 변화가 시간 순서대로 다른 CPU에게도 전파되게 하려면 메모리 배리어도 쳐야 하지 않나요?

Quick Quiz 4.18: 예를 들어 gcc __thread 스토리지 클래스를 사용해 선언된 쓰레드별 변수에 접근할 때에도 READ_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, holdtime은 1000(그림에서는 “1K”로 표시되었습니다) 부터 1억(그림에선 “100M”으로 표시됩니다) 까지 값을 변화시켰습니다. 그림으로 그려진 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.

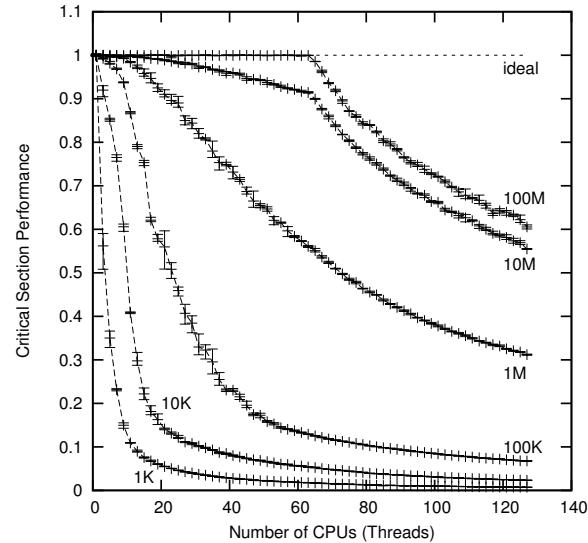


Figure 4.10: Reader-Writer Lock Scalability

다. 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_t를 한번씩 반드시 수정해야만 합니다. 운좋은 쓰레드는 곧바로 그렇게 할 수 있을테지만, 가장 불행한 쓰레드는 다른 127개의 쓰레드들이 수정을 완료할 때까지 기다려야 합니다. 이 상황은 CPU를 추가할수록 나빠지기만 할 겁니다.

Quick Quiz 4.19: 단일 CPU 성능에 비교하는건 좀 심한 거 아닌가요?

Quick Quiz 4.20: 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 4.21: Figure 4.10에서 100M에서의 경우 이외의 값들은 이상적인 선에서 부드럽게 멀어집니다. 반면, 100M에서의 값은 64 CPU에서 갑자기 이상적인 선으로부터 멀어지는군요. 또, 100M 값과 10M 값 사이의 거리는 10M 값과 1M 값 사이의 거리보다 작아요. 왜

100M 값은 이렇게 남들과 다른거죠?

Quick Quiz 4.22: Power-5 는 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/O 를 하는 경우 등에 매우 유용합니다. 다른 대안들도 있는데, 그런 것들 중 일부는 Chapter 5 와 Chapter 9 에서 다루겠습니다.

4.2.5 Atomic Operations (gcc Classic)

Figure 4.10 가 리더-라이터 락킹의 오버헤드는 크리티컬 섹션이 작을수록 커진다는 것을 보여줬으나, 매우 작은 크리티컬 섹션을 보호할 다른 나은 방안을 찾아봐야 하겠습니다. 그런 한가지 방안은 어토믹 오퍼레이션의 사용입니다. 우린 Figure 4.9 의 라인 18 에서 `__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 4.23: 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-swap 을 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 할수도 있습니다. 그렇게 만들어진 것들은 보통 복잡도, 확장성, 성능 문제 [Her90]를 겪지만요.

`__sync_synchronize()` 함수는 Section 14.2 에서 다른, 컴파일러와 CPU 의 오퍼레이션 재배치 기능을 제한하는 “메모리 배리어” 를 불러옵니다. 어떤 경우에는 CPU 의 재배치 가능성은 두고 컴파일러의 오퍼레이션 재배치를 제한하는 것만으로도 충분한데, 이 경우에는 Figure 4.9 의 라인 28 에서처럼 `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 4.6 의 라인 17 에서처럼 `READ_ONCE()` 함수가 사용될 수 있을 겁니다. 비슷하게, 컴파일러가 특정 메모리로의 쓰기 를 최적화해 버리는 것을 막기 위해 `WRITE_ONCE()` 기능이 사용될 수 있습니다. 이 마지막 두개의 함수들은 gcc 에 의해 제공되지는 않습니다만 다음과 같이 구현될 수 있을 겁니다:

```
#define ACCESS_ONCE(x) (*(volatile typeof(x) *) &(x))
#define READ_ONCE(x) ACCESS_ONCE(x)
#define WRITE_ONCE(x, val) ({ ACCESS_ONCE(x) = (val); })
#define barrier() __asm __volatile__ ("": : : "memory")
```

Quick Quiz 4.24: 이 어토믹 오퍼레이션들은 기계의 인스트럭션 셋에서 바로 지원되는 한개짜리 어토믹 인스트럭션으로 변환될테니, 이것들이 일을 돌아가게 할 수 있는 가장 빠른 방법 아닌가요? ■

4.2.6 Atomic Operations (C11)

C11 표준은 어토믹 오퍼레이션들을 추가했는데, 이는 로드 (`atomic_load()`), 스토어 (`atomic_store()`), 메모리 배리어 (`atomic_thread_fence()` 와 `atomic_signal_fence()`), 그리고 `read-modify-write` 어토믹 오퍼레이션들을 포함합니다. `Read-modify-write` 어토믹 오퍼레이션들은 `atomic_fetch_add()`, `atomic_fetch_sub()`, `atomic_fetch_and()`, `atomic_fetch_xor()`, `atomic_exchange()`, `atomic_compare_exchange_strong()`, 그리고 `atomic_compare_exchange_weak()` 를 포함합니다. 이것들은 Section 4.2.5 에서 설명한 것들과 비슷한 방식으로 동작합니다만 모든 오퍼레이션의 `_explicit` 변종들은 메모리 순서 인자를 추가로 받습니다. 메모리 순서 인자 없이는, 모든 어토믹

오퍼레이션들은 완전히 순서를 맞추고, 인자가 주어진다면 완화된 순서규칙을 허용합니다. 예를 들어, “`memory_order_explicit(&a, memory_order_relaxed)`”는 리눅스 커널의 “`READ_ONCE()`” 와 대략적으로 유사합니다.¹

C11 어토믹 오퍼레이션들의 한가지 한계점은 특수한 어토믹 타입들에만 적용된다는 점입니다. 따라서 gcc 컴파일러는 `__atomic_load()`, `__atomic_load_n()`, `__atomic_store()`, `__atomic_store_n()`, 등을 제공합니다. 이런 기능들은 C11에서 대응되는 것들과 동일한 semantic을 제공합니다만, 평범한 `atomic` 타입이 아닌 오브젝트에 대해서도 사용될 수도 있습니다.

4.2.7 Per-Thread Variables

Thread-specific 데이터, thread-local storage, 그리고 그 외의 다른 이름으로 불리는 per-thread 변수는 동시성 코드에서 매우 많이 사용되는데, Chapter 5 와 8에서 다루어질 겁니다. POSIX는 하나의 per-thread 변수를 만들기 위해 (그리고 연관된 키를 리턴받기 위해) `pthread_key_create()` 함수를, 해당 키에 연관된 해당 per-thread 변수를 삭제하기 위해 `pthread_key_delete()` 를, 현재 쓰레드의 특정 키에 연관된 변수에 값을 지정하기 위해 `pthread_setspecific()` 을, 그리고 그 값을 리턴받기 위해 `pthread_getspecific()` 을 제공합니다.

여러개의 (gcc를 포함한) 컴파일러들이 변수의 정의에 해당 변수를 per-thread로 지정하기 위해 사용될 수 있는 `__thread` 지시어를 제공합니다. 그렇게 지정되면 해당 변수의 이름은 현재 쓰레드의 해당 변수의 인스턴스의 값을 접근하는데에 평범하게 사용될 수 있습니다. 물론, `__thread`는 POSIX thread-specific data 보다 사용하기가 쉽고, `__thread`는 gcc나 `__thread`를 지원하는 컴파일러에서만 빌드될 코드에서는 일반적으로 더 선호됩니다.

다행히도, C11 표준은 `__thread` 대신 사용될 수 있는 `Thread_local` 키워드를 새로 도입했습니다. 시간이 충분히 지나면, 이 새로운 키워드는 `__thread`의 간편한 사용성과 POSIX thread-specific data의 호환성을 모두 결합해서 제공할 겁니다.

¹ 메모리 순서 규칙은 Section 14.2와 Appendix B에서 더 자세히 설명됩니다.

4.3 Alternatives 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 이유로 특정 환경에서 더 나은 성능을 얻기 위해서든, 어셈블리 언어로 구현된 오퍼레이션들도 여전히 많습니다. 예를 들어, gcc `__sync_` 류 함수들은 메모리 순서 의미를 제공해서 많은 개발자들이 메모리 순서 의미가 요구되지 않는 상황을 위한 각자의 구현을 만들도록 이릅니다. 다음 섹션들은 리눅스 커널에서의 일부 대안들과 이 책의 예제 코드에서 사용된 역사적 기능들 일부를 보입니다.

4.3.1 Organization and Initialization

많은 환경들이 특수한 초기화 코드를 필요로 하지 않지만, 이 책의 샘플 코드는 `pthrad_t`로부터 연속적인 정수들 사이의 매핑을 초기화하는 `smp_init()` 함수의 호출로 시작합니다. Userspace RCU 라이브러리는 비슷하게 `rcu_init()` 호출을 필요로 합니다. 이런 함수 호출들은 생성자를 지원하는 (gcc와 같은) 환경에서는 숨겨질 수 있긴 하지만, userspace RCU 라이브러리에 의해 지원되는 대부분의 RCU 기능들은 각 쓰레드가 쓰레드 생성 시에 `rcu_register_thread()` 를, 그리고 쓰레드 종료 전에 `rcu_unregister_thread()` 를 호출할 것을 필요로 합니다.

리눅스 커널의 경우에 있어서는, 커널이 특수한 초기화 코드로의 호출을 필요로 하지 않는지 여부나 커널의 부팅 시점 코드가 실제로 필요시되는 초기화 코드인가 여부와 같은 것들은 철학적인 질문입니다.

4.3.2 Thread Creation, Destruction, and Control

리눅스 커널은 `kthread`들을 쓰는데에 `struct task_struct` 포인터를, 그것들을 생성하는데 `kthread_create()` 를, 그것들이 멈출 것을 외부로 제안하는데에 (POSIX에는 똑같은 일을 하는 게 없습니다) `kthread_should_stop()` 을, 그것들이 멈추길 기다리는데에 `kthread_stop()` 을, 그리고 시간이 지정된 대기를 위해서 `schedule_timeout_interruptible()` 을 사용합니다. 몇가지 더 `kthread` 관리 API들이 있습니다만, 이것만으로도 괜찮은 시작점이고 괜찮은 검색어 키워드가 됩니다.

CodeSamples API는 “`thread`”에 집중하는데, 이는

제어의 한 층입니다.² 그런 쓰레드 각각은 `thread_id_t` 타입의 식별자를 가지고, 동시에 수행중인 어떤 두 쓰레드도 같은 식별자를 갖지는 않습니다. 쓰레드들은 `program counter` 와 `stack` 을 포함한 per-thread local state³ 를 제외하고는 어떤 것도 공유하지 않습니다.

이 쓰레드 API 는 Figure 4.11 에 보여져 있으며, 이 멤버들은 다음 섹션들에서 설명됩니다.

4.3.2.1 `create_thread()`

`create_thread()` 기능은 새로운 쓰레드를 생성하고, 이 새로운 쓰레드의 수행을 `create_thread()` 의 첫번째 인자로 지정된 `func` 함수부터 시작하는데, 이 때 `create_thread()` 의 두번째 인자로 지정된 인자를 넘겨줍니다. 이 새로 생성된 쓰레드는 `func` 로 지정된 시작 함수에서 리턴할 때에 종료됩니다. `create_thread()` 기능은 새로 생성된 자식 쓰레드와 연관된 `thread_id_t` 를 리턴합니다.

이 기능은 프로그램의 수행동안 생성된 쓰레드의 갯수를 내부적으로 세며, 만약 `NR_THREADS` 보다 많은 쓰레드가 생성되면 프로그램을 종료시킵니다. `NR_THREADS` 는 컴파일 시점에서 지정되는 상수로, 일부 시스템에서는 허용 가능한 쓰레드의 수에 최대값이 정해져 있을 수 있지만, 수정될 수 있습니다.

4.3.2.2 `smp_thread_id()`

`create_thread()` 로부터 리턴받는 `thread_id_t` 는 시스템에 종속적이므로, `smp_thread_id()` 기능은 요청받은 쓰레드에 연관된 쓰레드 인덱스를 리턴합니다. 이 인덱스는 프로그램이 시작한 이래로 존재한 쓰레드의 최대 숫자보다 작을 것이 보장되어지고, 따라서 bitmask, 배열 인덱스, 등등에 유용합니다.

4.3.2.3 `for_each_thread()`

`for_each_thread()` 매크로는 생성되면 존재하게 될 모든 쓰레드들을 포함해서 모든 존재하는 쓰레드들을 루프합니다. 이 매크로는 Section 4.2.7 에서 보게 될 per-thread 변수들을 다루는데에 유용합니다.

4.3.2.4 `for_each_running_thread()`

`for_each_running_thread()` 매크로는 현재 존재하는 쓰레드들만을 루프합니다. 필요할 경우 쓰레드 생성과 삭제에 동기화를 하는 것은 호출자의 역할입니다.

² 비슷한 소프트웨어 구성물을 위한 많은 다른 이름들이 있는데, “process”, “task”, “fiber”, “event”, 등등이 있습니다. 이것들 모두에도 비슷한 설계 철학이 적용될 수 있습니다.

³ 순환점의에서는 이게 어떻게 될까요?

4.3.2.5 `wait_thread()`

`wait_thread()` 기능은 넘겨받은 `thread_id_t` 로 특정되는 쓰레드의 완료를 기다립니다. 이 기능은 해당 쓰레드의 수행에 어떠한 간섭도 끼치지 않습니다; 그저, 그것을 기다릴 뿐입니다. `wait_thread()` 는 연관된 쓰레드로부터 리턴된 값을 리턴함을 알아두세요.

4.3.2.6 `wait_all_threads()`

`wait_all_threads()` 기능은 현재 수행중인 모든 쓰레드의 완료를 기다립니다. 필요하다면 쓰레드 생성, 소멸과 동기화를 맞추는건 호출자의 역할입니다. 하지만, 이 기능은 일반적으로 프로그램 수행 종료 시에 정리를 하는데에 사용되므로, 그런 동기화는 보통은 필요하지 않습니다.

4.3.2.7 Example Usage

Figure 4.12 는 hello-world 같은 자식 쓰레드의 예제를 보입니다. 앞서서 이야기 되었듯, 각각의 쓰레드는 자신의 스택을 할당받고, 따라서 각각의 쓰레드는 자신만의 `arg` 인자와 `myarg` 변수를 갖습니다. 각각의 자식은 단순히 각자의 인자와 각자의 `smp_thread_id()` 를 종료되기 전에 출력합니다. Line 7 의 `return` 문은 쓰레드를 종료시키고 이 쓰레드에 대해 `wait_thread()` 를 호출한 누군가에게 `NULL` 을 리턴함을 알아두시기 바랍니다.

그 부모 프로그램이 Figure 4.13 에 보여져 있습니다. 이 프로그램은 line 6 에서 쓰레드 시스템을 초기화 하기 위해 `smp_init()` 를 호출하고, line 7-14 에서 인자들을 분석한 후, line 15 에서 자신의 존재를 공지합니다. 이 프로그램은 이어서 지정된 수의 자식 쓰레드들을 line 16-17 에서 생성하고, 그것들이 완료되기를 line 18 에서 기다립니다. `wait_all_threads()` 는 쓰레드들이 리턴하는, 이 경우에는 그 값이 `NULL` 로써 큰 의미가 없는, 값들을 무시함을 알아두시기 바랍니다.

Quick Quiz 4.25: 리눅스 커널의 `fork()` 와 `wait()` 대체물은 어디갔죠?



4.3.3 Locking

리눅스 커널의 락킹 API 를 알아보기 시작하는데 알아보기 좋은 부분집합들이 Figure 4.14 에 보여져 있는데, 각각의 API 원소는 다음 섹션들에서 설명됩니다. 이 책의 CodeSamples 락킹 API 는 리눅스 커널의 그것을 거의 유사하게 따라갑니다.

```

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 4.11: Thread API

```

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 4.12: Example Child Thread

```

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 4.13: Example Parent Thread

4.3.3.1 spin_lock_init()

`spin_lock_init()` 기능은 명시된 `spinlock_t` 변수를 초기화하며, 이 변수가 다른 `spinlock` 기능들에 전달되기 전에 반드시 호출되어야 합니다.

4.3.3.2 spin_lock()

`spin_lock()` 기능은 명시된 스피너를 획득하며, 필요하다면 해당 스피너가 획득 가능해질 때까지 기다립니다. Pthread 와 같은 일부 환경에서는 이 기다림은 “spinning”을 포함할 수도 있는 반면에, 리눅스 커널과 같은 다른 경우에는 `blocking`을 포함합니다.

여기서의 핵심은, 언제든 하나의 스피너는 하나의 쓰레드만이 잡을 수 있다는 것입니다.

4.3.3.3 spin_trylock()

`spin_trylock()` 기능은 명시된 스피너를 획득합니다만, 곧바로 획득이 가능할 때에만 그렇게 합니다. 스피너를 획득할 수 있었다면 `true`를 리턴하고 그렇지 않다면 `false`를 리턴합니다.

4.3.3.4 spin_unlock()

`spin_unlock()` 기능은 명시된 스피너를 놓아주어서 다른 쓰레드들이 해당 스피너를 획득할 수 있도록 해줍니다.

4.3.3.5 Example Usage

변수 `counter`를 보호하는데에 `mutex` 라 이름지어진 스피너가 다음과 같이 사용될 수 있습니다:

```

spin_lock(&mutex);
counter++;
spin_unlock(&mutex);

```

Quick Quiz 4.26: 변수 `counter`가 `mutex`의 보호 없이 값 증가된다면 어떤 문제가 있을 수 있나요? ■

하지만, `spin_lock()` 과 `spin_unlock()` 기능들은 성능상의 문제가 있는데, Section 4.3.6에서 다루어집니다.

Figure 4.14: Locking API

4.3.4 Atomic Operations

리눅스 커널은 다양한 어토믹 오퍼레이션들을 제공합니다만, `atomic_t` 타입으로 정의되어있는 것들이 좋은 시작점이 될겁니다. 일반적인 `non-tearing read` 와 `store` 들은 `atomic_read()` 와 `atomic_set()` 으로 각각 제공됩니다. `Acquire load` 는 `smp_load_acquire()` 로, `release store` 는 `smp_store_release()` 로 제공됩니다.

값을 리턴하지 않는 `fetch-and-add` 오퍼레이션들은 `atomic_add()`, `atomic_sub()`, `atomic_inc()`, 그리고 `atomic_dec()` 으로 제공됩니다. 값이 0이 되는 경우에 대한 알림을 리턴하는 어토믹 값 감소 연산은 `atomic_dec_and_test()` 와 `atomic_sub_and_test()` 로 제공됩니다. 새로운 값을 리턴하는 어토믹 값 증가 연산은 `atomic_add_return()` 으로 제공됩니다. `atomic_add_unless()` 와 `atomic_inc_not_zero()` 두개의 기능은 어토믹 변수들의 원래 값이 명시된 값과 다르다면 아무일도 일어나지 않는 조건적 어토믹 오퍼레이션들을 제공합니다(이것들은, 예컨대 레퍼런스 카운터를 관리하는데 매우 유용합니다).

어토믹 값 교환 오퍼레이션은 `atomic_xchg()` 로 제공되고, 찬양되는 `compare-and-swap` (CAS) 오퍼레이션은 `atomic_cmpxchg()` 로 제공됩니다. 이것들을 둘다 기존 값을 리턴합니다. 리눅스 커널에서는 더 많은 어토믹 RMW 기능들이 사용 가능한데, 리눅스 커널 소스 트리의 `Documentation/atomic_ops.txt` 파일을 참고하시기 바랍니다.

이 책의 `CodeSamples API` 는 리눅스 커널의 것들을 거의 비슷하게 따라합니다.

4.3.5 Per-CPU Variables

리눅스 커널은 per-CPU 변수를 정의하기 위해 `DEFINE_PER_CPU()` 를, 특정 per-CPU 변수의 현재 CPU 의 인스턴스를 참조하는데에 `this_cpu_ptr()` 를, 특정 per-CPU 변수의 특정 CPU 의 인스턴스를 접근하는데에 `per_cpu()` 를 사용하며, 그 외에도 많은 특수 목적의 per-CPU 오퍼레이션들이 존재합니다.

Figure 4.15 는 이 책의 per-thread-variable API 를 보아는데, 리눅스 커널의 per-CPU-variable API 의 패턴을 따랐습니다. 이 API 는 전역 변수의 per-thread 로 동일한 것을 제공합니다. 엄격하게 말하자면 이 API 가 꼭 필요한 건 아닙니다만⁴, 이 API 는 리눅스 커널 코드와 유사한 유저스페이스를 제공할 수 있습니다.

Quick Quiz 4.27: Per-thread-variable API 를 제공하지 않는 시스템에서는 어떻게 이를 우회할 수 있을까요? ■

⁴ 이 API 대신에 `__thread` 나 `_Thread_local` 을 사용하실 수도 있습니다.

```
DEFINE_PER_THREAD(type, name)
DECLARE_PER_THREAD(type, name)
per_thread(name, thread)
__get_thread_var(name)
init_per_thread(name, v)
```

Figure 4.15: Per-Thread-Variable API

4.3.5.1 DEFINE_PER_THREAD()

`DEFINE_PER_THREAD()` 기능은 per-thread 변수를 정의합니다. 불행하게도, 리눅스 커널의 `DEFINE_PER_THREAD()` 기능과 같은 방식으로 초기화 기능을 제공하는 건 불가능합니다만, 수행시간의 초기화를 쉽게 할 수 있게 해주는 `init_per_thread()` 기능이 있습니다.

4.3.5.2 DECLARE_PER_THREAD()

`DECLARE_PER_THREAD()` 기능은 정의에 반대되는, C에서의 방식의 선언입니다. 따라서, `DECLARE_PER_THREAD()` 기능은 다른 파일에서 정의된 per-thread 변수에 접근하기 위해 사용될 수 있습니다.

4.3.5.3 per_thread()

`per_thread()` 기능은 명시된 쓰레드의 변수를 접근합니다.

4.3.5.4 __get_thread_var()

`__get_thread_var()` 기능은 현재 쓰레드의 변수를 접근합니다.

4.3.5.5 init_per_thread()

`init_per_thread()` 기능은 모든 쓰레드의 명시된 변수 인스턴스들을 명시된 값으로 설정합니다. 리눅스 커널은 이를 링커 스크립트와 CPU-online 프로세스 동안 수행되는 코드를 잘 사용해서 평범한 C 초기화를 통해 해냅니다.

4.3.5.6 Usage Example

매우 자주 그 값이 증가하지만 아주 가끔씩만 그 값이 읽어지는 카운터를 가지고 있다고 해봅시다. Section 4.3.6 에서 자명해지겠지만, 그런 카운터는 per-thread 변수를 사용해 구현하는게 좋습니다. 그런 변수는 다음과 같이 정의될 수 있습니다:

```
DEFINE_PER_THREAD(int, counter);
```

이 카운터는 다음과 같이 초기화 되어야 합니다:

```
init_per_thread(counter, 0);
```

쓰레드는 이 카운터의 자신의 인스턴스의 값을 다음과 같이 증가시킬 수 있습니다:

```
__get_thread_var(counter)++;
```

따라서 이 카운터의 값은 그 인스턴스들의 값의 합입니다. 따라서 이 카운터의 값의 스냅샷은 다음과 같이 얻어집니다:

```
for_each_thread(i)
    sum += per_thread(counter, i);
```

다시 말하지만, 다른 메커니즘을 사용해서도 비슷한 효과를 얻을 수 있습니다만, per-thread 변수들은 높은 성능과 편리성을 함께 제공합니다.

4.3.6 Performance

Section 4.3.4에서 보여진, 락으로 보호되는 값 증가 오퍼레이션의 성능을 per-CPU(또는 per-thread) 변수 사용 방법 (Section 4.3.5을 참고하세요), 그리고 통념적인 값 증가 방법 (“counter++”)과 비교해 보는 것이 도움이 될 것입니다.

삼가 말하자면, 성능의 차이는 상당히 큽니다. 이 책의 목표는 여러분이 필요하다면 realtime 반응속도와 함께 이러한 성능상의 문제를 막으면서 SMP 프로그램을 작성하는 것을 돋는 것입니다. Chapter 5는 몇가지 병렬 카운팅 알고리즘들을 설명함으로써 이 일을 시작합니다.

4.4 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 9에서

소개되는 도구들이 필요할 수 있습니다. 프로세스간 통신과 메세지 패싱은 공유 메모리 멀티 쓰레드 실행의 좋은 대안이 될 수 있다는 점을 항상 기억하세요.

Quick Quiz 4.28: 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 고르는 것 만큼이나 올바른 설계를 하는 것이 중요합니다.

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. 컴퓨터에서는 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 보이는데도 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, 느릅니다: 인텔 Core Duo 랩탑에서 이것은 어토믹하지 않은 방법에 비해 싱글쓰레드에서 여섯배, 그리고 두 쓰레드를 사용했을 때엔 열배 느릅니다.¹

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

```

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 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. 분명한건, 거대한 성능과 확장성을 위해선 정확성을 주의깊게 완화한 알고리즘의 역할도 있을 수 있다는 것입니다 [And91, ACMS03, Ung11].

```

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!

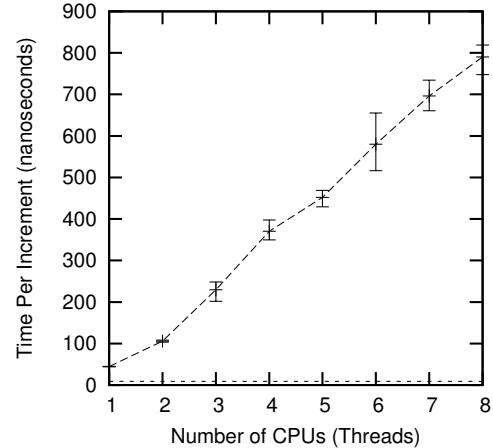


Figure 5.3: Atomic Increment Scalability on Nehalem

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 증가시킬 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 5.5와 같은 상황을 초래해서 Figure 5.3에서 보여진 낮은 성능을 야기할 것입니다.

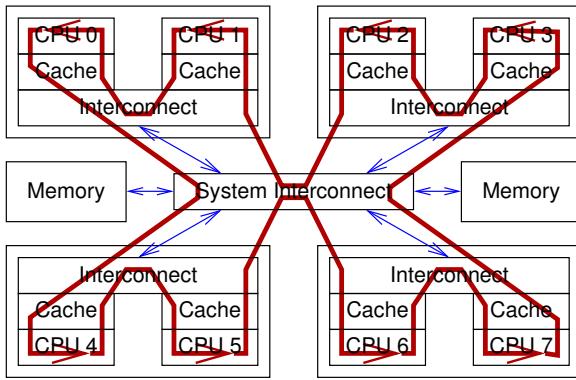


Figure 5.4: Data Flow For Global Atomic Increment

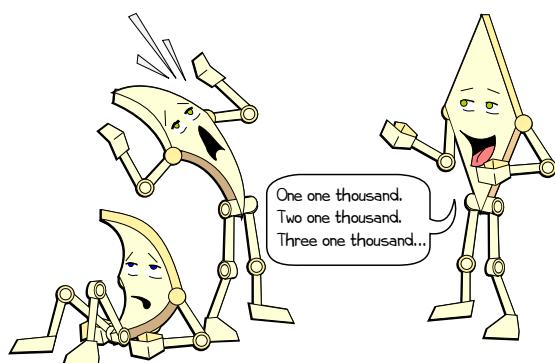


Figure 5.5: Waiting to Count

다음 섹션들에서는 이런 순환에서 발생하는 지연들을 없앤 고성능 카운팅에 대해 알아봅니다.

Quick Quiz 5.10: 그런데 왜 CPU 설계자들은 단순히 데이터에의 증가 연산을 추가해서 담고 있는 캐시 라인의 순회가 증가하는 걸 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 5.2에서 이야기한 네트워크 패킷 카운팅 문제를 푸는데 사용될 수 있을 것입니다.

```

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    return sum;
16 }
```

Figure 5.6: Array-Based Per-Thread Statistical Counters

5.2.1 Design

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

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

5.2.2 Array-Based Implementation

쓰레드별 변수를 제공하는 방법 중 하나는 쓰레드별로 (아마도 false sharing을 막기 위해 캐시 일라인 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 1은 long 타입의 쓰레드별 카운터들을 담는, counter라는 이름의 배열을 만듭니다.

라인 3-6은 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 8-16은 카운터의 총계 값을 얻어오는 함수로, for_each_thread()를 이용해 현재 돌고 있는 쓰레드들의 리스트를 순회하면서 per_thread()를 이용해 특정 쓰레드의 카운터 값을 얻어옵니다. 하드웨어는 제대로 정렬되어 있는 long 변수는 어토믹하게 읽고 쓸 수 있기 때문에, 그리고 gcc는 이 기능을 잘 사용해 주기 때문에, 평범한 읽기로 충분하고, 별다른 어토믹

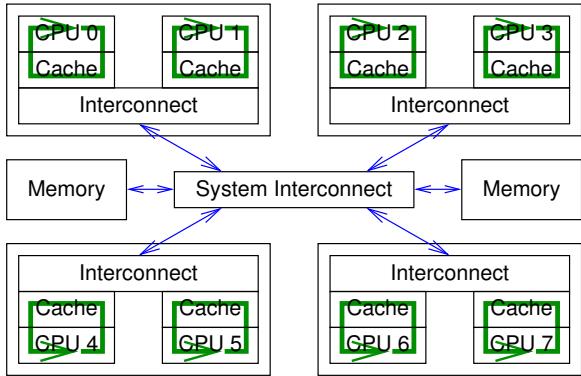


Figure 5.7: Data Flow For Per-Thread Increment

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 값은 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 값들 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 로컬 `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()` 에서 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?

```

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 }

```

Figure 5.9: Per-Thread Statistical Counters

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 gcc 는 리눅스 커널

Figure 5.8: Array-Based Per-Thread Eventually Consistent Counters

의 `per_cpu()` 가 쓰레드들이 다른 쓰레드의 쓰레드 별 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 5.9의 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 5.9의 `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 5.9의 `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 보이는데, 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 배제시킵니다. 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 중발 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 별별 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 또다른 특수 케이스는 한계 체크입니다. 예를 들면, 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 대략적이어서 어느정도 까지는 넘어가도 되는 경우를 생각해 봅시다(한계치가 정확해야 한다면 Section 5.4를 참고하세요).

5.3.1 Design

한계 카운터를 위해 사용할 수 있을 설계 중 하나는 한계치 10,000을 쓰레드의 수로 나눠서 각 쓰레드에게 크기가 고정된 구조체 풀을 주는 것입니다. 예를 들어, 100개의 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_count를 업데이트해서 최종적으로는 쓰레드별 counter의 값과 global_count를 일관되게 만들어주는 eventual() 쓰레드를 운용했습니다. 쓰레드별 counter는 카운터 값을 완벽하게 분할했고 global_count는 전체 값을 유지했습니다.

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

그렇게 되면 한 쓰레드가 자신의 counter를 증가시키는데, counter는 자신의 countermax와 동일하면 어떻게 될까요? 그 쓰레드의 counter 값의 절반을 globalcount로 옮기고, counter를 증가시키는 트릭을 씁니다. 예를 들어, 한 쓰레드의 counter와 countermax 변수가 똑같이 10이라면, 다음의 일을 합니다:

1. 글로벌 락을 잡는다.
2. globalcount에 5를 더한다.
3. 앞의 합과 균형을 맞추기 위해, 이 쓰레드의 counter에서 5를 뺀다.
4. 글로벌 락을 놓는다.
5. 이 쓰레드의 counter를 증가시켜서 6으로 만든다.

이 방법은 여전히 글로벌 락을 필요로 하지만, 그들은 다섯번의 증가 오퍼레이션마다 한번씩만 사용되므로, 락 경쟁을 상당히 줄여줄 겁니다. 이 경쟁정도는

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

```

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

countermax 값을 올려줌으로써 필요한 만큼까지 낮춰줄 수 있습니다. 하지만, countermax를 늘리는데 대한 페널티가 존재하는데 globalcount의 정확도가 떨어진다는 점입니다. 예를 들어 4개 CPU가 있는 시스템에서 countermax는 10이라 하면 globalcount는 최대 40의 에러값을 가질 수 있습니다. 반면, countermax가 100까지 늘어나면, globalcount는 최대 400의 에러값을 가질 수 있습니다.

이는 globalcount와 globalcount와 각 쓰레드의 counter 변수의 값의 합으로 만들어지는 값 사이의 차이를 얼마나 많이 신경써야 하나 하는 질문을 가져옵니다. 답은 얼마나 합으로 만들어진 값이 카운터의 리미트 (globalcountmax라고 해두죠)와 차이가 나나에 달려 있습니다. 이 두 값 사이의 차이가 크면 클수록 countermax는 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 에러를 줄이기 위해 작은 값으로 설정되어야 합니다.

이 디자인은 일반적인 경우는 쓰레드간의 상호작용과 비싼 동작 없이 수행하되 결국 사용해야만 할 때에는 보수적으로 설계된 (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 5.10는 이 구현에서 사용되는 쓰레드별, 그리고 글로벌 변수를 보여줍니다. 쓰레드별 counter와 countermax 변수들은 연관되는 쓰레드의 로컬 카운터와 그 카운터의 최대 허용값을 각각 나타냅니다. 라인 3의 globalcountmax 변수는 합계 카운터의 최대값을 가지며, 라인 4의 globalcount 변수는 글로벌 카운터입니다. globalcount와 각 쓰레드의 counter의 합은 전체 카운터의 합산 값을 갖습니다.

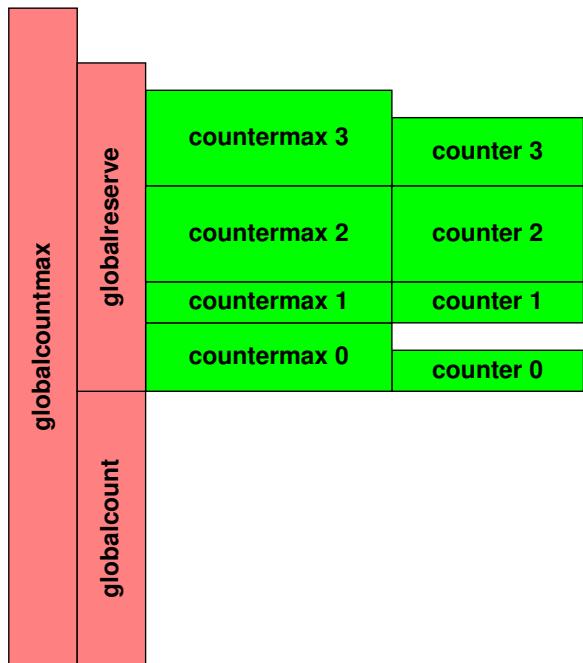


Figure 5.11: Simple Limit Counter Variable Relationships

다. 라인 5 의 `globalreserve` 변수는 모든 쓰레드별 `countermax` 변수의 값의 합입니다. 이 변수들 간의 관계가 Figure 5.11 에 나타나 있습니다:

1. `globalcount` 와 `globalreserve` 의 합은 `globalcountmax` 보다 작거나 같아야만 합니다.
2. 모든 쓰레드의 `countermax` 값의 합은 `globalreserve` 보다 작거나 같아야만 합니다.
3. 각 쓰레드의 `counter` 는 해당 쓰레드의 `countermax` 보다 작거나 같아야만 합니다.

`counterp[]` 배열의 각 원소는 상응하는 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? ■

```

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

라인 1-18은 `add_count()`를 보여주는데, 이 함수는 특정값 `delta`를 카운터에 더합니다. 라인 3에서는 `delta`를 위한 공간이 이 쓰레드의 `counter`에 남아 있는지 확인하고, 만약 그렇다면 라인 4에서 그 값을 더하고 라인 6에서 성공했음을 리턴합니다. 이게 `add_count()`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 3의 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 3에서의 테스트가 실패한다면, 글로벌 변수들에 접근해야 하므로, 라인 7의 `gblcnt_mutex`를 반드시 잡아야 하고, 이 락은 실패하는 경우 라인 11에서, 성공하는 경우는 라인 16에서 해제합니다. 라인 8에서는 Figure 5.13의 `globalize_count()`를 호출하는데, 이 함수는 쓰레드로컬 변수들을 초기화하고, 글로벌 변수들을 필요한 대로 맞춰줘서 전역적인 처리를 단순화 시킵니다. (하지만 제 말을 믿지만 말고, 직접 코딩해보세요!) 라인 9와 10에서는 `delta`를 더하는 것이 합법적인지 Figure 5.11에서 가장 왼쪽 두 빨간 막대의 높이 차이로 나타난 크기 규칙이 지켜지는지로 체크합니다. `delta`의 합이 이루어져선 안된다면, 라인 11은 (앞서 이야기했듯) `gblcnt_mutex`를 풀고 라인 12에서 실패를 알리는 리턴을 합니다.

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

Quick Quiz 5.30: Figure 5.12에서 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 20-36은 `sub_count()` 함수를 보여주는데, 이 함수는 `delta`만큼을 카운터에서 제거합니다. 라인 22는 쓰레드별 카운터가 이 빠기를 해도 되는 값인지 체크하고, 만약 그렇다면 라인 23에서 그 빠기를 수행한

후 라인 24에서 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 빠기가 적절치 못함으로 판단된다면, 실행은 라인 26-35의 느린 수행경로로 이어집니다. 느린 실행 경로는 글로벌 상태에 접근해야 하기 때문에, 라인 26에서 `gblcnt_mutex`를 잡고, 이 락은 (실패의 경우) 라인 29에서, 또는 (성공의 경우) 라인 34에서 해제됩니다. 라인 27은 Figure 5.13의 `globalize_count()`를 실행하는데, 앞서 설명했듯 쓰레드로컬 변수의 값을 지우고, 글로벌 변수들을 적당한 값으로 설정합니다. 라인 28은 카운터가 `delta`의 빠기를 해도 좋을지 보고, 좋지 않다면 라인 29에서 `gblcnt_mutex`를 (앞서 설명했듯) 해제하고 라인 30에서 실패를 리턴합니다.

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

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

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

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

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

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

라인 1-7은 `globalize_count()` 함수인데, 이 함수는 현 쓰레드의 쓰레드별 카운터를 0으로 만들고, 글로벌 변수들을 적당한 값으로 만듭니다. 이 함수는 카운

```

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

터의 합산값을 바꾸진 않지만 어떻게 카운터의 현재 값이 보여야 하는지를 바꾼다는 점이 중요합니다. 라인 3은 쓰레드의 counter 변수를 globalcount에 더하고 라인 4에서 counter를 0으로 만듭니다. 비슷하게, 라인 5는 쓰레드별 countermax를 globalreserve에서 빼고, 라인 6에서 countermax를 0으로 만듭니다. 이어서 설명될 balance_count() 함수와 이 함수를 함께 이해하려면 Figure 5.11을 참고하는게 도움이 될겁니다.

라인 9-19는 balance_count() 함수를 보아는데, 이 함수는 간단히 말해서 globalize_count()의 반대입니다. 이 함수가 하는일은 현 쓰레드의 countermax 변수를 카운터가 globalcountmax한 계치를 넘지 않는 한계에서 가장 큰 값을 갖도록 하는 것입니다. 현재 쓰레드의 countermax 변수를 바꾸는건 물론 counter, globalcount 그리고 globalreserve를 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 11-13은 globalcount나 globalreserve로 다뤄지지 않은 globalcountmax에서 이 쓰레드의 자분을 계산하고, 계산된 값을 이 쓰레드의 countermax에 대입합니다. 라인 14는 globalreserve를 올바르게 조정합니다. 라인 15는 이 쓰레드의 counter를 countermax 절반으로 만듭니다. 라인 16은 이 counter 값이 규칙을 깨지 않는 선에서 가능한 값인지 globalcount를 보고, 그렇지 않다면 라인 17에서 counter를 적절하게 감소시킵니다. 앞의 조건과 관계없이 마지막으로, 라인 18에서는 globalcount를 적절하게 조정합니다.

Quick Quiz 5.34: Figure 5.13의 라인 15에서는 왜 counter를 countermax / 2로 만들까요? 그냥 countermax 값을 가져오는게 더 간단하지 않나요? ■

Figure 5.14에 그린 것처럼 globalize_count()와 balance_count에 의해서 카운터 값들이 어떻게 바뀌는지 개요를 그려보는 게 도움이 될 겁니다. 시간은 왼쪽에서 오른쪽으로 흐르고, 가장 왼쪽의 구성은 대략 Figure 5.11의 그것입니다. 중간의 구성은 globalize_count()가 쓰레드 0에 의해 수행되고 난 후의 각 카운터들의 관계입니다. 그림에서 볼 수 있듯, 쓰레드 0의 counter(그림에서의 “c 0”)는 globalcount에 더해지고, globalreserve는 그만큼 줄었습니다. 쓰레드 0의 counter와 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 참고하십

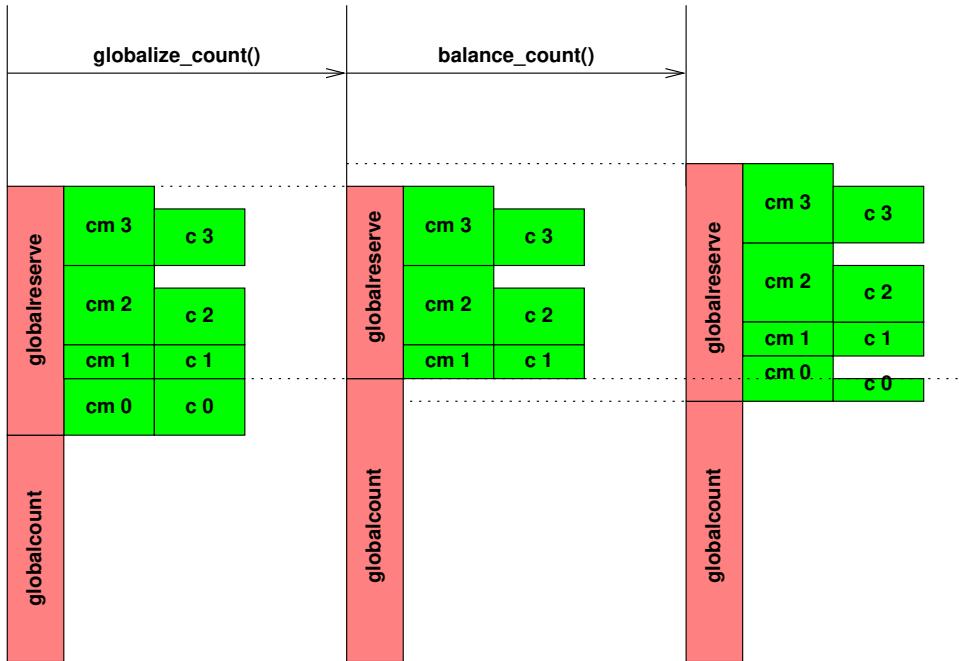


Figure 5.14: Schematic of Globalization and Balancing

시오. 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 0에 의해 `balance_count()` 가 수행된 이후의 카운터들의 관계를 보여줍니다. 다른 두개의 구성에 비해 올라간 수직의 선으로 보여지듯, 남은 카운트의 4분의 1이 쓰레드 0의 `countermax`에, 절반이 쓰레드 0의 `counter`에 더해졌습니다. 쓰레드 0의 `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 0의 `counter`는 해당 쓰레드의 `countermax` 보다 작기 때문에, 쓰레드 0은 또다시 자신의 카운터에서 곧바로 카운트 증가를 할 수 있습니다.

Quick Quiz 5.35: Figure 5.14에서 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? ■

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 34에서는 라인 37에서 해제될 `gblcnt_mutex` 를 잡습니다. 라인 35에서는 `globalize_count()` 를 호출해서 이 쓰레드의 카운터 상태를 정리하고, 라인 36에서 이 스레드의 `counterp[]` 배열에서의 상응하는 원소를 정리합니다.

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 `countermax` 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.

```

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

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 countermax 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 제외하고는 Figure 5.10 와 똑같습니다.

비슷하게, Figure 5.16 는 라인 6 과 7에서 쓰레드별 countermax 변수에 MAX_COUNTERMAX 한계를 적용하는 내용이 추가된 걸 제외하고는 Figure 5.13 의 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 수행경로로 접근 비율이 바뀝니다. 쓰레드의 수가 늘어나면, 빠른 수

행경로 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

5.4 에서 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, 예를들면 32-비트 변수가 있다면, 앞쪽 16 비트는 counter를, 뒷쪽 16 비트는 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.

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

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

라인 10-15는 split_ctrandmax_int () 함수를 보여주는데, 이 함수는 atomic_t ctrandmax 변수

```

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

로부터 얻어온 int 값을 받아서, counter (c) 와 countermax (cm) 으로 쪼갭니다. 라인 13 은 이 int로부터 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 18에서는 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 결과값을 리턴합니다.

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

Figure 5.18 는 add_count() 와 sub_count() 함수를 보입니다.

```

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

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

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

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

Figure 5.18의 라인 16-31은 `add_count()`의 느린 수행경로를 보이는데, 라인 17에서 얻어오고 라인 24와 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 5.18의 라인 34-63에서는 `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.

```

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

니다.

Figure 5.19는 `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 5.20와 5.21는 유ти리티처럼 사용되는 함수들인 `globalize_count()`, `flush_local_count()`, `balance_count()`, `count_register_thread()`, 그리고 `count_unregister_thread()`의 코드를 보입니다. `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 어토믹하게 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.

```

1 static void globalize_count(void)
2 {
3     int c;
4     int cm;
5     int old;
6
7     split_crandmax(&ctrandmax, &old, &c, &cm);
8     globalcount += c;
9     globalreserve -= cm;
10    old = merge_crandmax(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_crandmax(0, 0);
25     for_each_thread(t)
26         if (counterp[t] != NULL) {
27             old = atomic_xchg(counterp[t], zero);
28             split_crandmax_int(old, &c, &cm);
29             globalcount += c;
30             globalreserve -= cm;
31         }
32 }

```

Figure 5.20: Atomic Limit Counter Utility Functions 1

다.

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?

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 수행 경로가 ctrandmax 를 함께 사용하면서 동시에 수행되지 못하는 이유는 뭐죠? ■

Figure 5.21 의 라인 1-22 는 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 5.21 의 라인 21에서의 balance_count() 는 flush_local_count() 의 해당 변수에 동시에 가해지는 업데이트에도 불구하고 올바르게 동작할 수 있는 거죠? ■

```

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_crandmax(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

다음 섹션에서는 이 설계를 질적으로 평가해 봅니다.

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 CPU 로 옮겨져서 수행될 수도 있잖아요. 이런 가능성을 쓰레드와 해당 쓰레드를 인터럽트 하는 시그널

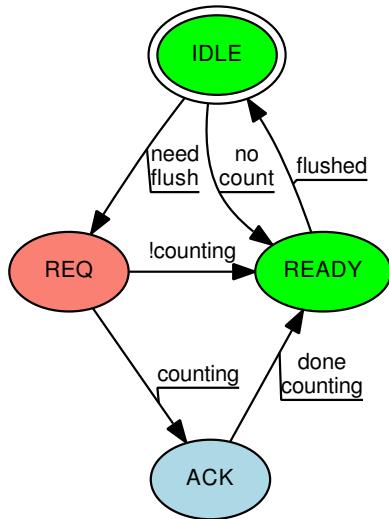


Figure 5.22: Signal-Theft State Machine

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 만들습니다(해당 쓰레드가 카운트가 있을 경우이고, 그렇지 않다면 READY로 곧바로 변환시킵니다). 초록색으로 표시되어 있듯, `gblcnt_mutex` 락을 쥐고 있는 느린 수행 경로에게만 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 REQ가 아니라면 시그널 핸들러는 상태를 바꿀 권한이 없고, 따라서 그냥 리턴합니다. 그렇지 않고, `counting` 변수가 값이 있어 현재 쓰레드의 빠른 수행 경로가 실행 중이란 것을 알게 된다면, 시그널 핸들러는 `theft` 상태를 ACK로, 그렇지 않다면 READY로 바꿉니다.

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

파란색으로 표시되어 있듯, `theft` 상태가 ACK라면 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 REQ와 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 쓰레드별 `theft` 스테이트 머신을 위한 값들을 정의합니다. 라인 8-17은 앞의 구현과 비슷합니다만, 라인 14와 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 함수는 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.

```

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

다. 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 5.24 의 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 스킵합니다. 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? ■

라인 35-48의 루프는 각 쓰레드가 READY 상태에도 달하길 기다리고, 이후 해당 쓰레드의 카운트를 훔칩니다. 라인 36-37은 존재하지 않는 쓰레드들을 스킵하고, 라인 38-42의 루프에서 현재 쓰레드의 theft 상태가 READY가 되길 기다립니다. 라인 39에서는 우선순위 역전(priority-inversion) 문제를 막기 위해 1밀리세컨드 동안 블락되고, 라인 40에서 해당 쓰레드의 시그널이 아직 도착하지 않은 것으로 판단되면 라인 41에서 시그널을 다시 날립니다. 현재 살펴보고 있는 쓰레드의 상태가 마침내 READY가 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 5.24의 라인 41에서는 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 느

```

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

```

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

린 수행 경로가 있습니다. 라인 5에서는 쓰레드별 counting 변수를 1로 만들어서 이후에 이 쓰레드를 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 CPU 는 라인 9의 결과도 보도록 해줍니다. 라인 9에서의 빠른 수행 경로 더하기가 실행되었다면, 라인 20에서 성공을 리턴합니다.

```

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

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 SIGUSR1의 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 선호될 만 합니다만 과거의 80386 기반 Sequent Symmetry 시스템에서는 어토믹 구현의 더 짧은 코드를 더 빨리 수행할 것입니다. 하지만, 이 업데이트 쪽의 성능 향상은 높아진 읽는 쪽 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 읽는 쪽도 빨라야 한다면 어떻게 하시겠어요? ■

이건 왜 높은 품질의 API가 그렇게 중요한지를 보여주는 한가지 예이기도 합니다: 높은 품질의 API는 바

```

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

뀌는 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? ■

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. 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 카운팅 하는 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 41에서 이야기된 제거 가능한 I/O 디바이스 액세스 카운트 문제에의 부분적 해결책에 불과합니다. 디바이스를 제거하려 시도할 때, 우린 현재 I/O 액세스들의 정확한 갯수를 알아야 할 뿐만이 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 일인든 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 같을 겁니다:

```

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 알리고, 라인 5-7에서 존재하는 I/O 오퍼레이션들이 끝나길 기다립니다. 마지막으로, 라인 8에서 디바이스 제거에 필요한 작업을 진행합니다.

Quick Quiz 5.57: 실제 시스템에 적용하려면 해결해야 할 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 5.1의 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 일관성이 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.

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 대해서만 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 숫자에도 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? 랙디스 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. 짧게 말해서, 다리 설계가 거리와 부하에 따라 달라져야만 하는 만큼, 소프트웨어 설계 역시 CPU 갯수에 따라 달라져야만 합니다. 그렇다고 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 적용 기능은 주요 시스템에서의 CPU 갯수가 늘어남을 유지함에 따라

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	13.3.1	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

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 다른 통계적 카운터들처럼 완벽하게 분할되어질 수도 있고, Section 5.3과 Section 5.4에서 다루어진 리미트 카운터들처럼 부분적으로 분할될 수도 있습니다. 분할은 Chapter 6에서 훨씬 꼭넓게 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 여

러 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 한가지는 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 사용되었습니다.

마지막으로, Section 5.2.3에서 다른 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)에까지 개선을 줄 수 있음을 보일 겁니다.

요약을 요약하자면:

1. 분할하기는 성능과 확장성을 높입니다.
2. 부분 분할은 일반적인 코드 패스에만 분할을 적용

한 것으로, 대부분 잘 동작합니다.

3. 부분 분할은 (Section 5.2 의 통계적 카운터의 분할된 업데이트와 분할되지 않은 읽기처럼) 코드에만 적용 가능한 것이 아니라, (Section 5.3 과 Section 5.4 의 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 5.1 과 Table 5.2 에 보여진 모든 알고리즘들은 몰아 처리하기 (Batching) 을 많이 사용합니다.
5. 읽기만 하는 코드 패스는 읽기만 하도록 유지되어야 합니다: 공유 메모리에의 의미없는 동기화된 쓰기는 Table 5.1 의 count_end.c 열에 보여진 것처럼 성능과 확장성을 떨어뜨립니다.
6. 현명한 딜레이의 사용은 Section 5.2.3 에서 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 5.1 의 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 5.3 가 이 점을 보이고 있습니다: 두개의 CPU 로 구성된 시스템에서 어토믹 증가는 사용하기 적당합니다만, 8개 CPU 로 구성된 시스템에서는 결코 적당치 않습니다.

계속해서 요약해 보자면, 우린 성능과 확장성을 개선할 수 있는 “세개의 커다란” 방법을 가지고 있는데, 각각 (1) CPU 와 쓰레드들을 분할하기 (partitioning), (2) 한번의 비싼 동기화 오퍼레이션마다 더 많은 일이 처리되도록 몰아서 처리하기 (batching), 그리고 (3) 가능하다면 동기화 오퍼레이션들을 약화시키기. 대략적 경험적 법칙상으로는, page 15 의 Figure 2.6 에서 이야기 한것처럼 이 방법들을 이 순서대로 적용해야 합니다. 분할하기 최적화 기법, 몰아서 처리하기 최적화 기법, 그리고 동기화 약화 최적화 기법은 각각 Figure 5.29 의 “리소스 분할하기와 복사하기”, “일 분할하기”, “병렬 접근 제어” 에 적용될 수 있습니다. 물론, 당신이 digital signal processors (DSPs), field-programmable gate

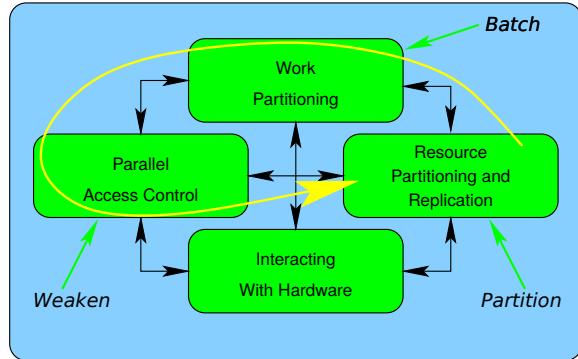


Figure 5.29: Optimization and the Four Parallel Programming Tasks

arrays (FPGAs), 또는 general-purpose graphical processing units (GPGPUs) 와 같은 특별한 용도의 하드웨어를 사용한다면, 설계 과정에서 “하드웨어와 상호작용하기” 부분에 큰 집중을 기울여야 할 겁니다. 예를 들어, GPGPU 의 하드웨어 쓰레드와 메모리 연결 구조는 조심스런 분할과 몰아처리하기 설계 결정에 큰 영향을 끼칠 겁니다.

짧게 말해서, 이 챕터의 시작에서 이야기했듯, 카운팅의 단순성이 우리가 복잡한 동기화 도구들이나 정교한 구조체에 정신을 팔리지 않은채 기본적인 동시성 문제를 알아볼 수 있게 했습니다. 그런 동기화 도구들과 데이터 구조들은 뒤의 챕터들에서 다룹니다.

Chapter 6

Partitioning and Synchronization Design

이 챕터는 상용 시스템의 트렌드인 여러개의 CPU 들로부터 이점을 얻기 위해서는 어떻게 소프트웨어를 설계해야 하는지 설명합니다. 이를 위해 성능, 확장성, 그리고 반응시간 사이의 균형을 잡는데 도움을 줄 수 있을, 여러개의 관용구나 “디자인 패턴” [Ale79, GHJV95, SSRB00] 들을 소개합니다. 앞의 챕터에서 이야기 했듯, 병렬 소프트웨어를 만들 때 하게 되는 가장 중요한 결정은 파티셔닝을 어떻게 할것인지입니다. 잘 분할된 문제들은 간단하고, 확장성 있으며, 고성능을 갖는 해결책을 이끌어냅니다만, 자못 분할된 문제들은 느리고 복잡한 해결책을 만듭니다. 이 챕터는 몰아서 처리하기 (batching) 와 약화시키기 (weakening) 에 대한 토론과 함께 파티셔닝을 코드로 설계하는 것을 도울 것입니다. “설계”란 말은 매우 중요합니다: 파티셔닝이 첫번째, 몰아 처리하기가 두번째, 약화하기가 세번째이며, 코딩은 네번째입니다. 이 순서를 바꾸는 행위는 낮은 성능과 확장성에다가 엄청난 좌절을 일으킬 것입니다.

이를 위해, Section 6.1 에서는 파티셔닝 연습문제들을 소개하고, Section 6.2 에서 분할가능성 설계 기준을 알아보고, Section 6.3 에서 적절한 동기화 정도에 대해 이야기 하고, Section 6.4 에서 일반적인 경우 속도와 확장성을 제공하는 중요한 병렬성의 빠른 수행 경로와 간단하지만 일반적이지 않은 상황을 위한 덜 확장성 있는 대안인 “슬로우 패스” 설계의 개요를 알아본 후, 마지막으로 Section 6.5 에서 파티셔닝 다음을 간략히 봅니다.

6.1 Partitioning Exercises

이 섹션은 파티셔닝의 가치를 보이기 위해 한 째의 연습 문제 (고전적인 식사하는 철학자들 문제와 양극단 큐)를 사용합니다.

6.1.1 Dining Philosophers Problem

Figure 6.1 는 고전적인 식사하는 철학자들 문제 [Dij71]를 보이고 있습니다. 이 문제는 다섯 명의 철학자들로

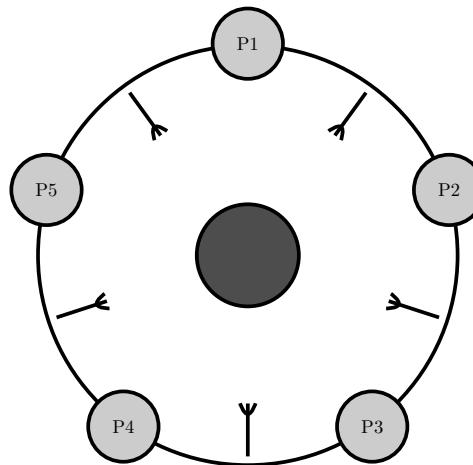


Figure 6.1: Dining Philosophers Problem

구성되는데, 이 철학자들은 아무 일도 하지 않고 그저 생각을 하고 “매우 먹기 어려운 종류의 스파게티”여서 먹는데 두개의 포크가 필요한 스파게티를 먹는 것만을 반복합니다. 철학자는 그(또는 그녀)의 오른쪽과 왼쪽에 놓인 포크만을 사용해야 하는데, 한번 포크를 골라서 들었으면, 배를 채우기 전까지는 다시 내려놓지 않습니다.¹

목표는 스타베이션 (starvation: 기아) 을 막는 알고리즘을 만드는 것입니다. 스타베이션 시나리오 중 하나는 모든 철학자가 동시에 각자의 왼쪽 포크를 집어드는 것입니다. 그들은 스파게티를 다 먹기 전까지는 아무도 포크를 내려놓지 않을 것이고, 최소 한명은 식사를 끝내기 전까지는 두번째 포크를 들 수 없으므로, 그들 모두가 굶어죽게 됩니다. 최소 한명의 철학자는 먹게 해주는 것만으로는 문제 해결에 충분치 않음에 유의하시기 바랍니다. Figure 6.2 가 보이듯이, 일부 철학자의

¹ 두개의 포크를 사용하는 음식을 떠올리기 어렵다면 젓가락을 생각해 봅시다.

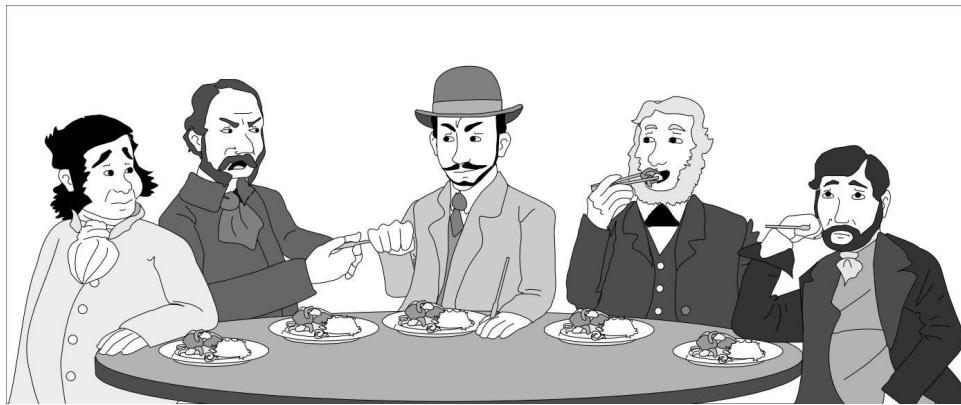


Figure 6.2: Partial Starvation Is Also Bad

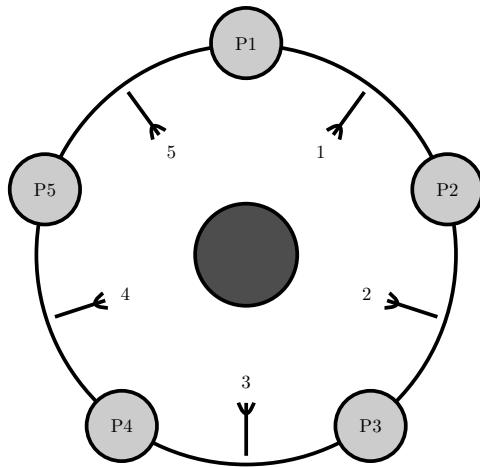


Figure 6.3: Dining Philosophers Problem, Textbook Solution

스타베이션도 없어야 합니다.

Dijkstra의 해결책은 글로벌 세마포어를 사용하는 것으로, 무시해도 좋은 통신 딜레이를 가정하면 잘 동작 하지만, 이 가정은 1980년대 후반이나 1990년대 초반 들어서는 타당하지 않게 되었습니다.² 따라서, 최근의 해결책은 Figure 6.3처럼 포크에 숫자를 매기는 것입니다. 각 철학자는 자신의 접시 옆의 것 중 가장 낮은 숫자의 포크를 먼저 들고, 이후에 가장 높은 숫자의 포크를 듭니다. 이 그림에서 가장 위쪽에 자리한 철학자는 따라서 왼쪽 포크를, 그 후 오른쪽 포크를 집어들게 되고,

² 그 해결책이 나온지 40년이 넘게 지난 2012년의 관점에서 Dijkstra를 모욕하는 것은 너무도 쉽습니다. 여전히 Dijkstra를 모욕해야겠다고 생각한다면, 제가 드리고 싶은 말은, 뭔가 출간하고, 40년 동안 기다린 후, 당신의 말들이 그 시간동안 어떻게 겸중되었는지 보시기 바랍니다.

그동안 다른 철학자들은 오른쪽 포크를 먼저 들게 됩니다. 두명의 철학자들은 포크 1을 먼저 집어들으려 할 것이기 때문에, 그리고 두 철학자 중 한명만 성공할 것이기 때문에, 네명의 철학자들에게는 다섯개의 포크가 사용 가능할 것입니다. 이 네명 중 최소 한명은 두개의 포크를 집어드는데 성공함이 보장되고, 따라서 식사를 계속할 수 있습니다.

이 일반적인 리소스에 숫자를 매기고 숫자 순서대로 획득하기 방법은 데드락 예방 기술에서 매우 많이 사용되고 있습니다. 하지만, 모두가 배고픈데 한번에 한명의 철학자만 식사를 하게 되는 경우를 초래하는 일련의 이벤트를 떠올리기는 쉽습니다:

1. P2 가 포크 1 을 집어들고, P1 이 포크를 집어들지 못하게 합니다.
2. P3 가 포크 2 를 집어듭니다.
3. P4 가 포크 3 을 집어듭니다.
4. P5 가 포크 4 를 집어듭니다.
5. P5 가 포크 5 를 집어들고 식사를 합니다.
6. P5 가 포크 4 와 5 를 내려놓습니다.
7. P4 가 포크 4 를 집어들고 식사를 합니다.

즉, 다섯명의 철학자들이 모두 배고풀 때에도, 두명의 철학자들이 동시에 식사를 해도 될만큼 충분한 양의 포크가 있음에도 한번에 한명의 철학자만 식사를 할 수 있는 경우가 존재합니다.

계속해서 읽기 전에 식사하는 철학자들 문제를 분할할 방법을 생각해 보시기 바랍니다.

(Intentional blank page)

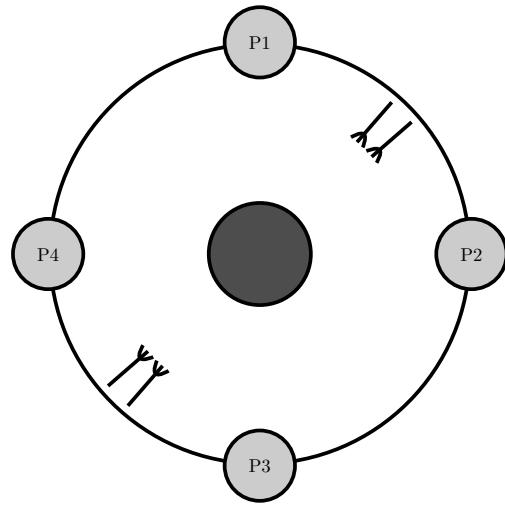


Figure 6.4: Dining Philosophers Problem, Partitioned

한가지 방법이 Figure 6.4 이 그려져 있는데, 이 방법을 좀 더 잘 그리기 위해 철학자들의 수를 다섯명에서 네명으로 바꿨습니다. 여기서 위쪽과 오른쪽의 철학자들은 한 짹의 포크를 공유하고, 아래쪽과 왼쪽의 철학자들은 다른 한 짹의 포크를 공유합니다. 모든 철학자들이 동시에 배고파진다면, 최소 두명은 항상 동시적으로 식사를 할 수 있을 것입니다. 또한, 이 그림에서 보여지듯, 포크들은 동시에 집어들고 내려놓을 수 있도록 뭉음으로 취급될 수 있어서 그 획득과 해제 알고리즘을 간단하게 만들어 줍니다.

Quick Quiz 6.1: 식사하는 철학자들 문제를 위한 더 나은 해결책은 없을까요? ■

이것은 두명씩으로 이루어진 철학자 그룹들 사이에 종속성이 없기 때문에, “수평적 병렬성” [Inm85] 또는 “데이터 병렬성”의 한가지 예입니다. 수평적으로 병렬적인 데이터 처리 시스템에서 데이터의 주어진 항목은 소프트웨어 컴포넌트들의 하나의 복제된 집합에서만 처리될 것입니다.

Quick Quiz 6.2: 그리고 어떤 관점에서 이 “수평적 병렬성”은 “수평적”이라 이야기 될 수 있는 걸까요? ■

6.1.2 Double-Ended Queue

Double-ended 큐는 원소가 양쪽의 끝 중 어디로든 삽입되고 삭제될 수 있는 자료구조입니다 [Knu73]. 동시에 양 끝으로 원소가 삽입 / 삭제될 수 있는 Double-ended 큐를 락 기반으로 구현하는 것은 어렵다고 알려져 있습니다 [Gro07]. 이 섹션은 세개의 일반적인 방법을 다음 섹션들에서 봄으로써 파티셔닝 디자인 전략이 어떻게 합리적으로 간단한 구현이 가능하게 하는지 알아봅니다.

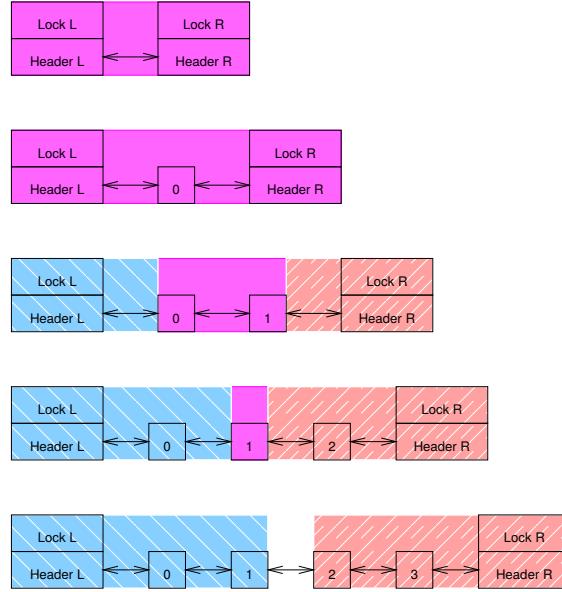


Figure 6.5: Double-Ended Queue With Left- and Right-Hand Locks

다.

6.1.2.1 Left- and Right-Hand Locks

하나의 당연해 보이는 방법은 양방향 링크드 리스트를 왼쪽 인큐와 디큐를 위해 왼쪽 락을, 오른쪽의 같은 작업들을 위해 오른쪽 락을 사용하는 것으로, Figure 6.5에 그림으로 그려져 있습니다. 하지만, 이 방법의 문제는 두 락의 도메인들이 이 리스트에 원소가 네개보다 적을 때 겹치지 않아야 한다는 것입니다. 이 겹침은 어떤 원소의 삭제가 그 원소만이 아니라, 그것의 왼쪽 또는 오른쪽 이웃 원소에게도 영향을 끼치기 때문입니다. 이런 도메인들은 그림에 색으로 표시되었는데, 아래로 향하는 줄무늬의 파란색이 왼쪽 락의 도메인을, 위쪽 줄무늬의 붉은색이 오른쪽 락의 도메인을, 그리고 줄문의 없는 보라색은 겹치는 도메인들을 의미합니다. 이 방법으로 동작하는 알고리즘을 만드는 것도 가능하겠지만, 다섯개 미만의 특별한 케이스들이 있다는 사실은 커다란 문제인데, 특히나 리스트의 다른 끝에서의 동시적인 동작인 큐를 하나의 특별한 케이스에서 다른 케이스로 언제든지 바꿀 수 있습니다. 다른 설계를 생각해 보는게 아무래도 낫습니다.

6.1.2.2 Compound Double-Ended Queue

락 도메인들이 겹치지 않게 하는 방법 한가지가 Figure 6.6에 그려져 있습니다. 두개의 별도의 double-

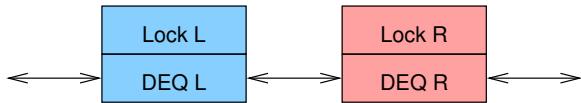


Figure 6.6: Compound Double-Ended Queue

ended 큐를 직렬로 연결되며, 각각은 각자의 락으로 보호됩니다. 이는 원소들은 결국은 하나의 double-ended 큐에서 다른 것으로 옮겨가게 된다는 것을 의미하며, 이 경우 두 락을 모두 잡아야만 하게 됩니다. 데드락을 방지하기 위해 간단한 락 계층구조를 사용할 수 있는데, 예를 들면, 항상 오른쪽 락을 잡기 전에 왼쪽 락부터 잡는 것입니다. 이 방법은 조건 없이 왼쪽에 들어온 원소를 왼쪽의 큐에, 그리고 오른쪽으로 들어온 원소를 오른쪽 큐에 넣을 수 있기 때문에, 두개의 락을 하나의 double-ended 큐에 사용하는 것보다 훨씬 간단합니다. 빈 큐에서 원소를 꺼내려 할 때 좀 복잡한 문제가 생기는데, 이 경우에는 다음과 같은 작업이 필요합니다:

1. 오른쪽 락을 잡고 있다면, 그것을 해제부터 하고 왼쪽 락을 다시 잡는다.
2. 오른쪽 락을 잡는다.
3. 두 큐 사이의 원소들의 균형을 다시 잡는다.
4. 원소가 존재하면 요청받은대로 원소를 제거한다.
5. 두 락을 모두 놓는다.

Quick Quiz 6.3: 이 compound double-ended 큐 구현에서, 비어있던 큐가 락을 놓고 다시 잡는 과정 사이 더이상 비어있지 않게 된다면 어떻게 해야 할까요? ■

그 결과로 만들어지는 코드 (locktdeq.c)는 매우 직접적입니다. 앞서 언급한 다시 균형잡는 작업은 원소를 두 큐 사이에서 옮길 수도 있을 것이고, 이로 인해 시간을 소모할 수 있으므로 최적의 성능을 위해선 실제 워크로드에 따른, 휴리스틱이 필요할 것입니다. 어떤 경우엔 이게 최선의 방법이겠지만, 더 결정론적인 알고리즘을 시도해 보는 것도 재미있을 겁니다.

6.1.2.3 Hashed Double-Ended Queue

데이터 구조를 결정론적으로 쪼개는 가장 간단하고 효과적인 방법은 해싱입니다. 각 원소에 그 리스트에서의 위치에 기반해서, 빈 큐에 왼쪽으로 들어온 첫번째 원소는 0을 가지고, 빈 큐에 오른쪽으로 들어온 원소에는 1을 할당하는식으로 숫자를 할당함으로써 double-ended 큐를 간단히 해시하는 것이 가능합니다. 이후 큐에 왼쪽으로 들어온 원소들은 감소하는 숫자 (-1, -2, -3, ...)

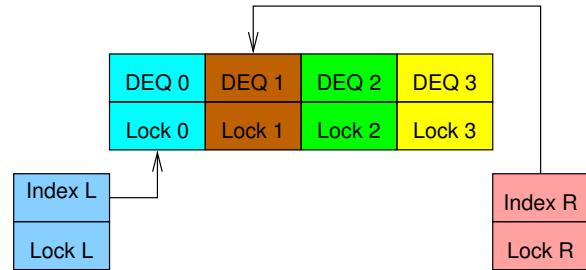


Figure 6.7: Hashed Double-Ended Queue

를 갖고, 오른쪽으로 들어온 원소들은 증가하는 수를 갖습니다 (2, 3, 4, ...). 핵심은, 이 숫자는 해당 큐에서의 위치를 의미하기 때문에, 특정 원소의 수를 실제로 나타낼 필요는 없다는 것입니다.

이 방법에서, 왼쪽 인덱스를 지키는데 한 락을 할당하고, 오른쪽 인덱스를 지키는데 또 하나를, 그리고 각 해시 체인을 위해 락을 하나 더 할당합니다. Figure 6.7 가 이로 인해 구성되는 데이터 구조를 네개의 해시 체인들로 표현하고 있습니다. 락 도메인들은 오버랩하지 않고, 데드락은 체인 락 이전에 인덱스 락을 획득하고 한번에 한 타입의 락 (인덱스 또는 체인) 만을 획득함으로써 함으로써 발생 불가능하게 되었습니다.

각 해시 체인은 그 자체로 double-ended 큐이며, 이 예제에서, 각각이 네개의 원소를 모두 가지고 있습니다. Figure 6.8 의 가장 위에는 하나의 원소 ("R₁₁") 이 오른쪽으로 들어온 후의, 오른쪽 인덱스가 해시 체인 2를 침조하게 증가된 상황이 그려져 있습니다. 이 그림의 중간 부분에는 세개의 원소가 더 오른쪽으로 들어온 후의 상황을 그립니다. 볼 수 있듯, 인덱스들은 최초의 상태 (Figure 6.7 를 참고하세요)로 돌아갔지만, 각 해시 체인은 이제 비어있지 않습니다. 이 그림의 아랫쪽은 추가적으로 원소들이 왼쪽으로, 그리고 오른쪽으로 들어온 후의 상태를 그립니다.

Figure 6.8 에 보여진 마지막 상황에서, 왼쪽 꺼내기 오퍼레이션은 "L₋₂" 를 리턴하고 왼쪽 인덱스가 이제는 하나의 원소 ("R₂")만을 가지고 있는 해시 체인 2를 가리키도록 할 것입니다. 이 상태에서, 동시에 왼쪽 집어넣기와 오른쪽 집어넣기가 실행되면 락 경쟁 상황에 빠지게 됩니다만, 그런 경쟁 상황은 더 큰 해시 테이블을 사용함으로써 비교적 적은 수준으로 경감될 수 있을 겁니다.

Figure 6.9 는 네개의 해시 버킷을 갖는 병렬 double-ended 큐에 16개의 원소가 들어가면 어떻게 구성되는지 보입니다. 전체 double-ended queue 를 구성하는, 단일 락으로 지켜지는 각각의 double-ended 큐는 전체 병렬 double-ended queue 의 4분의 1 조각을 갖는다고 볼 수 있겠습니다.

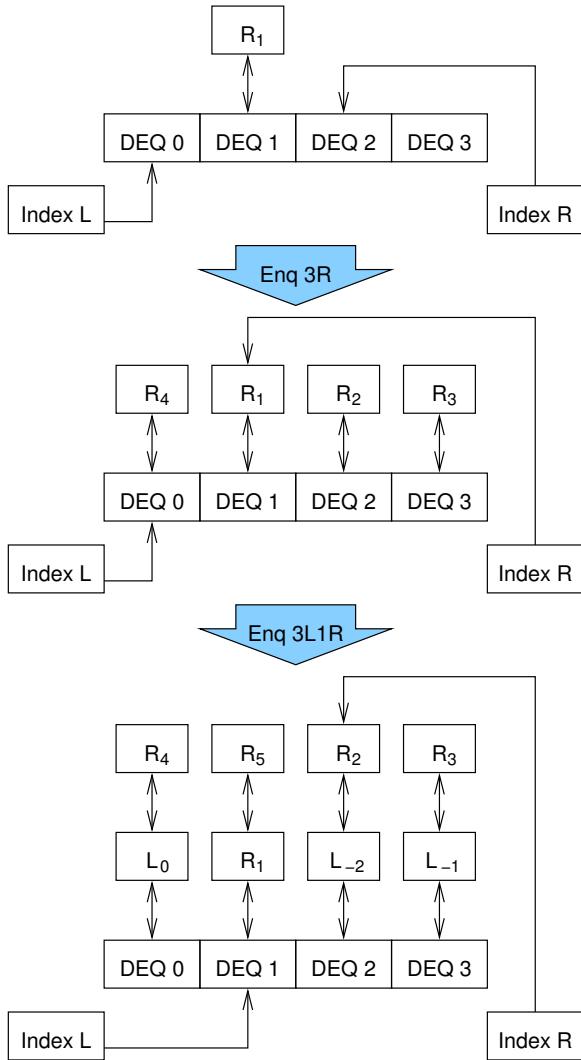


Figure 6.8: Hashed Double-Ended Queue After Insertions

Figure 6.10 는 관련된 C-언어 데이터 구조를 보여주는데, `struct deq` 가 간단히 락을 사용하는 double-ended 큐 구현을 제공한다고 가정합니다. 이 데이터 구조는 왼쪽 락을 line 2에서 포함하고 있고, 왼쪽 인덱스는 line 3에, 오른쪽 락을 line 4에서(실제 구현에서는 캐시 라인 크기로 정렬되어있겠습니다만), 오른쪽 인덱스를 line 5에서, 그리고 마지막으로 간단한 락 기반의 double-ended 큐들의 해시된 배열을 line 6에서 갖습니다. 고성능의 구현은 거짓 공유 (false sharing)를 막기 위해 패딩이나 특별한 정렬 지시어를 사용할 것입니다.

Figure 6.11 (`lockhdeq.c`)는 집어넣기 (enqueue) 와

R ₄	R ₅	R ₆	R ₇
L ₀	R ₁	R ₂	R ₃
L ₋₄	L ₋₃	L ₋₂	L ₋₁
L ₋₈	L ₋₇	L ₋₆	L ₋₅

Figure 6.9: Hashed Double-Ended Queue With 16 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

꺼내기 (dequeue) 기능 구현을 보입니다.³ 이야기는 왼쪽 오퍼레이션들에 집중해서 진행될텐데, 오른쪽 오퍼레이션들은 왼쪽의 그것과 비슷하기 때문입니다.

Line 1-13 은 왼쪽에서 꺼내고 원소가 있었다면 그 원소를, 그렇지 않다면 NULL 을 리턴하는 `pdeq_pop_1()` 를 보입니다. Line 6 에서는 왼쪽 스핀락을 얻고, line 7에서 꺼내어질 인덱스를 계산합니다. Line 8에서 해당 원소를 꺼내고, line 9에서 그 결과가 NULL이 아니라는 걸 확인하면, line 10에서 새 왼쪽 인덱스를 기록합니다. 어느쪽이건, line 11에서 락을 풀고, line 12에서 마침내 원소가 있었다면 그것을, 아니면 NULL 을 리턴합니다.

Line 29-38 은 왼쪽에 특정 원소를 집어넣는 `pdeq_push_1()` 을 보이고 있습니다. Line 33 은 왼쪽 락을 잡고, line 34에서 왼쪽 인덱스를 잡습니다. Line 35에서 왼쪽 인덱스로 가리켜진 double-ended 큐에 원소를 왼쪽으로 집어넣습니다. Line 36에서는 왼쪽 인덱스를 업데이트 하고 line 37에서 락을 풁니다.

앞서 언급되었듯, 오른쪽 오퍼레이션들은 왼쪽의 그것들과 비슷하므로, 그쪽의 분석은 독자 여러분의 연습 문제로 남겨두겠습니다.

Quick Quiz 6.4: 해시를 사용한 double-ended 큐는 좋은 해결책인가요? 그렇다면 왜고 그렇지 않다면 또 왜죠? ■

6.1.2.4 Compound Double-Ended Queue Revisited

이 섹션에서는 비어있지 않은 큐의 모든 원소들을 이제 비어있는 큐로 모두 옮기는 간단한 균형잡기 방식을

³ 다른 언어들로 다양한 형태의 구현을 하는 것도 쉬울 것입니다만, 그건 독자의 몫으로 남겨두겠습니다.

```

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

사용해서 compound double-ended 큐를 다시 생각해 봅니다.

Quick Quiz 6.5: 비어진 큐로 모든 원소들을 옮긴다 구요? 대체 어떤 미친 세상에서는 이런 방법이 최적인 거죠???

앞의 섹션에서 이야기한 해시 기반 구현과 대조적으로, compound 기반 구현은 락도 어토믹 오퍼레이션도 사용하지 않는, 순차적 구현으로 만들어진 double-ended 큐 위에서 만들어집니다.

Figure 6.12에 그 구현이 있습니다. 해시 기반 구현과 달리, 이 compound 기반 구현은 왼쪽과 오른쪽이 비대칭적이므로, `pdeq_pop_l()` 과 `pdeq_pop_r()` 구현을 각각 신경써야 합니다.

Quick Quiz 6.6: 왜 compound parallel double-ended queue 구현은 대칭적으로 만들어질 수 없는거죠?

이 그림의 line 1-16에 `pdeq_pop_l()`의 구현이 보여져 있습니다. Line 5는 왼쪽 락을 얻어오고, 이 락은 line 14에서 해제합니다. Line 6에서는 왼쪽의 아래에 있는 double-ended 큐에서 왼쪽 디큐를 하려 시도하고, 성공하면, 그냥 그 원소를 리턴하기 위해 line 8-13을 건너뜁니다. 그렇지 않다면, line 8에서 오른쪽 락을 잡고, line 9에서 오른쪽 큐로부터 왼쪽 디큐하고, line 10에서 오른쪽 큐의 모든 남아있는 원소들을 왼쪽 큐로 옮겨버리며, line 11에서 오른쪽 큐를 초기화하고, line 12에서 오른쪽 락을 놓습니다. 만약 존재한다면, line 10에서 디큐된 원소가 리턴됩니다.

`pdeq_pop_r()` 구현은 이 그림의 line 18-38에 보여져 있습니다. 앞에서 그랬듯, line 22에서 오른쪽 락을 얻고 (line 36에서 해제합니다), line 23에서 오른쪽 큐로부터 원소 하나를 오른쪽 디큐하려 시도하고, 성공하면 line 24-35를 건너뛰고 그냥 이 원소를 리턴합니다. 하지만, line 24가 디큐할 원소가 없음을 알게 된다면, line 25에서 오른쪽 락을 놓고 line 26-27에서 올바른 순서로 두 락을 모두 잡습니다. Line 28에서는 이제 오른쪽 큐에서 원소 하나를 오른쪽 디큐하려 시도하고, line 29에서 이 두번째 시도도 실패했음을 깨닫게 되면, line 30에서 왼쪽 큐로부터 (원소가 있다면) 원소 하나를 오른쪽 디큐하고, line 31에서 왼쪽 큐로부터 남아있는 모든 원소를 오른쪽 큐로 옮기고, line 32에서 왼쪽 큐를 초기화 합니다. 어떤 경우든, line 34에서 왼쪽 락을 놓습니다.

Quick Quiz 6.7: Figure 6.12의 line 28에서 왜 오른쪽 디큐를 또 시도하는거죠?

Quick Quiz 6.8: 왼쪽 락은 언젠가는 사용 가능할 겁니다!!! 그런데 Figure 6.12의 line 25에서는 오른쪽 락을 조건없이 놔버려야 하는거죠?

`pdeq_push_l()`의 구현이 Figure 6.12의 line 40-47에 있습니다. Line 44에서는 왼쪽 스판락을 잡고, line 45에서 왼쪽 큐에 왼쪽 인큐를 하고, 마지막으로

line 46에서 이 락을 놓습니다. `pdeq_enqueue_r()` 구현 (line 49-56에 있습니다)은 상당히 비슷합니다.

6.1.2.5 Double-Ended Queue Discussion

Compound 기반 구현은 Section 6.1.2.3에서 이야기한 해시 기반 구현에 비해 좀 복잡하지만, 여전히 간단한 편이라 말할 수 있겠습니다. 물론, 더 똑똑한 균형잡기 방법들은 어느정도 복잡할 수 있겠습니다만, 여기 소개한 간단한 방법들은 다른 소프트웨어 기반 방법들 [DCW¹¹]과 비교해서도, 그리고 심지어 하드웨어의 도움을 받는 알고리즘들 [DLM¹⁰]과 비교해서도 잘 동작함을 보이기 위한 것들입니다. 더도 아니고 덜도 아니고, 이런 방법에서 우리가 최대한 희망할 수 있는 건 2배까지의 확장성인데, 최대 두개의 쓰레드가 디큐를 위한 락들을 동시에 잡고 있을 수 있기 때문입니다. 이런 제약은 Michael의 compare-and-swap 기반 디큐 알고리즘 [Mic03]과 같은 non-blocking 동기화 알고리즘에도 적용됩니다.⁴

Quick Quiz 6.9: double-ended 큐 문제에 왜 한개가 아니라 두개나 해결책이 있는 거죠?

사실, Dice 등 [DLM¹⁰]에 의해 이야기 되었듯, 동기화 되지 않은 단일 쓰레드 기반 double-ended 큐는 그들이 알아본 다른 어떤 병렬 구현들보다도 성능이 좋습니다. 따라서, 핵심은 실제 구현과 관계 없이 공유된 큐에 인큐와 디큐를 하는 데에는 커다란 오버헤드가 존재한다는 점입니다. Chapter 3에서의 내용이 이 큐들의 FIFO의 섭리를 생각하면 크게 놀라운 일도 아닙니다.

더우기, 이런 철저한 FIFO 큐들은 사실 그 사용자들에게는 보이지 않는 *linearization point*⁵들에 대해서만 엄격하게 FIFO 인데, 이 예에서, linearization point 들은 락 기반의 크리티컬 섹션들 안에 있습니다. 이런 큐들은 개별적인 오퍼레이션들이 시작된 시점의 관점에서는 철저한 FIFO가 아닙니다 [HKLP12]. 이는 동시적 프로그램들에서 철저한 FIFO 속성을 그렇게 가치있는 것은 아님을 의미하며, 실제로 Kirsch 등은 향상된 성능과 확장성을 제공하는 덜 철저한 큐들을 선보였습니다 [KLP12].⁶ 그렇다고는 하나, 당신이 당신의 동시적

⁴ 이 논문은 double-ended 큐들의 lock-free 구현에 특별한 double-compare-and-swap (DCAS) 인스트럭션들이 필요치 않음을 보였다는 점에서 흥미롭습니다. 대신, 일반적인 compare-and-swap (예: x86 cmpxchq)만으로도 충분합니다.

⁵ 짧게 요약해서, linearization point는 한 함수 내에서 그 함수가 수행 결과를 만들어냈다고 할 수 있는 하나의 지점입니다. 이 락 기반의 구현에서 linearization point들은 실제 일을 하는 크리티컬 섹션 안의 어딘가라고 할 수 있습니다.

⁶ Nir Shavit은 대략적으로는 같은 이유로 완화된 스택들을 만들었습니다 [Sha11]. 이런 상황은 몇몇 사람들에게 linearization point들은 개별자들보다는 이론가들에게 유용하다고 믿게 하고, 그런 데 이터 스트럭쳐들과 알고리즘들의 설계자들은 얼마나 사용자들의 필요에 고려를 하고 있었는지 의아해하게 합니다.

```

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

프로그램에서 사용하는 모든 데이터를 하나의 큐를 통해 집어넣는다면, 전체 설계를 다시 한번 생각해 볼 필요가 있습니다.

6.1.3 Partitioning Example Discussion

Section 6.1.1 의 큐 퀴즈에 대한 답으로 이야기된 식사하는 철학자들 문제의 최적화된 해결책은 “수평적 병렬성” 또는 “데이터 병렬성”의 훌륭한 예 중 하나입니다. 이 경우의 동기화 오버헤드는 거의 (또는 정확히) 없습니다. 반면, double-ended 큐 구현은 “수직적 병렬성” 또는 “파이프라이닝”의 예로, 데이터를 한 쓰레드에서 다른 쓰레드로 옮깁니다. 파이프라이닝에 필요한 더 밀착된 협동은 일의 더 큰 단위가 주어진 수준의 효율성을 가질 것을 필요로 합니다.

Quick Quiz 6.10: 직렬화된 double-ended 큐는 해시 기반 double-ended 큐보다 두배나 빠른데, 이런 현상은 제가 해시 테이블의 크기를 미친듯이 크게 만들어줘도 그렇습니다. 왜 그런거죠? ■

Quick Quiz 6.11: double-ended 큐들을 위한 동시성 제어에 대한 훨씬 나은 방법이 있나요? ■

이 두개의 예들은 병렬 알고리즘을 고안하는데 파티셔닝이 얼마나 강력하지 보여줍니다. Section 6.3.5 에서는 세번째 예, 행렬 곱셈을 간단히 볼 겁니다. 하지만, 이 세개의 예 모두 병렬 프로그램들의 더 나은 디자인 표준을 필요로 하는데, 이는 다음 섹션의 주제입니다.

6.2 Design Criteria

최고의 성능과 확장성을 얻는 한가지 방법은 최고의 달성 가능한 병렬 프로그램에 이르를 때까지 핵을 지속하는 것입니다. 불행히도, 당신의 프로그램이 현미경으로 봐야할 정도로 작지 않다면, 가능한 병렬 프로그램의 공간은 우주의 수명이라는 거대한 시간 동안에도 최고의 달성 가능한 프로그램에 이르기를 보장하지 못할 정도로 커다랗습니다. 그리고 또, “최고의 달성 가능한 병렬 프로그램”이란 무엇일까요? 어쨌든, Section 2.2 에서는 성능, 생산성, 그리고 일반성 (generality) 의 세가지 병렬 프로그래밍 목표를 이야기 했고, 최고의 달성 가능한 병렬 프로그램은 생산성과 일반성에의 비용으로 다가올 확률이 큽니다. 프로그램이 구식이 되기 전에 충분히 받아들여질만한 좋은 병렬 프로그램까지는 만들기 위해 설계 단계에서 높은 수준에서의 선택들을 만들 수 있게 할 필요가 분명 있습니다.

하지만, 정말로 실제 세계의 설계를 하기 위해선 디자인 규범들이 필요한데, 이 섹션에서 다룰 것입니다. 실제 세계에서, 이 규범들은 종종 더 크거나 작은 정도에서 충돌하므로, 설계자들은 그로 인한 트레이드오프들을 잘 균형 맞춰야 합니다.

이 규범들은 그 자체로써 설계에 동작하는 “힘”들로 생각될 수 있으며, 특히 이런 힘들 간의 좋은 트레이드 오프들은 “디자인 패턴” [Ale79, GHJV95] 이라 불릴 수 있을 것입니다.

세개의 병렬 프로그래밍 목표들을 얻기 위한 디자인 규범들은 속도 향상, 경쟁, 오버헤드, 읽기-쓰기 비율, 그리고 복잡도입니다:

Speedup: Section 2.2 에서 이야기했듯, 성능이야말로 대부분의 시간을 쏟아야 하는 곳이고 병렬화를 해야 하는 문제입니다. 속도 향상은 순차적 버전의 프로그램을 돌리는데 드는 시간 대비 병렬 버전을 돌리는데 드는 시간 사이의 비율입니다.

Contention: 더 많은 CPU 들이 한 병렬 프로그램에 사용되고 그 프로그램에 의해 열심히 일을 하게 된다면, 너무 많은 CPU 들은 서로의 경쟁으로 인해 유의미한 일을 하지 못하게 되어버립니다. 이는 락 경쟁, 메모리 경쟁, 또는 다른 성능 문제 부분으로 부터의 것일 수도 있습니다.

Work-to-Synchronization Ratio: 유니프로세서, 싱글 쓰레드, 프리엠션 불가, 그리고 인터럽트 불가⁷ 한 버전의 병렬 프로그램은 어떤 동기화 도구들도 필요가 없을 것입니다. 따라서, 이런 도구들에 소모되는 (커뮤니케이션 캐시 미스들과 메세지 대기시간, 락킹 도구, 어토믹 인스트럭션들, 그리고 메모리 배리어들을 포함하는) 시간들은 그로그램이 완수하려 의도한 유용한 일에 직접적으로 도움을 주거나 하지 않는 오버헤드일 뿐입니다. 중요하게 측정해 봐야 할 것은 동기화 오버헤드와 크리티컬 섹션의 코드의 오버헤드 사이의 관계로, 더 큰 크리티컬 섹션은 더 큰 동기화 오버헤드를 견딜 수 있게 합니다. 일-대-동기화 비율은 동기화 효율성과 연관됩니다.

Read-to-Write Ratio: 아주 가끔만 업데이트 되는 데이터 구조체는 파티션 되기보다는 복사가 될 수 있을 것이고, 더 나아가서 읽는 쪽의 동기화 오버헤드를 쓰는 쪽을 부담시키는 대신 완화시켜주는 비대칭적 동기화 도구를 사용해서 보호될 수 있어서 전체 동기화 오버헤드를 줄일 수 있을 것입니다. 관련된 최적화들은 Chapter 5 에서 이야기 되었듯 자주 업데이트 되는 데이터 구조체에도 적용될 수 있습니다.

Complexity: 병렬 프로그램은 똑같은 순차적 프로그램에 비해 복잡한데, 병렬 프로그램은 순차적 프로그램에 비해 훨씬 큰 상태 공간을 갖기 때문입니다만, 이 커다란 상태 공간은 충분한 질서와 구조가

⁷ 인터럽트 마스킹을 하거나 그것들을 감지하지 못해서.

주어진다면 쉽게 이해될 수 있긴 합니다. 병렬 프로그램을 만드는 사람은 이 커다란 상태 공간의 문맥에서 동기화 도구들, 메세지, 락킹 설계, 크리티컬 섹션 식별, 그리고 테드락을 고려해야 합니다.

이 거대한 복잡도는 종종 높은 개발과 유지 비용으로 이야기되곤 합니다. 따라서, 예산의 제한이 존재하는 프로그램에 가할 수 있는 변경의 수와 종류를 제한할 수 있는데, 속도 향상은 많은 시간과 문제를 개선할 때에만 가치가 있기 때문입니다. 더 나쁜 것은, 추가된 복잡도가 실제로 성능과 확장성을 줄일 수 있다는 것입니다.

따라서, 어떤 특정한 지점 이후부터는 병렬화보다는 더 싸고 효과적인 순차적 최적화가 잠재하고 있을 수 있습니다. Section 2.2.1에서 이야기 했듯, 병렬화는 많은 것들 중 하나의 성능 최적화일 뿐이고, CPU-기반의 보틀넥들에 적용될 수 있는 최적화입니다.

이런 규범들은 최대의 속도 향상을 위해 함께 동작할 것입니다. 앞의 세개의 규범들은 깊게 관계되어 있으므로, 이 섹션의 뒷부분은 이 상호관계에 대해 분석해 보겠습니다.⁸

이런 규범들은 또한 요구사항의 일부분으로 나타날 수도 있음을 알아 두십시오. 예를 들어, 속도 향상은 상대적 요구사항으로 (“더 빠르게, 너 좋게”) 나올 수도 있고 워크로드의 절대적 요구사항으로 (“시스템은 최소한 초당 1,000,000 웹 헷을 지원해야 한다”) 나올 수도 있습니다. 고전의 디자인 패턴 언어들은 상대적 요구사항을 효력으로, 그리고 절대적 요구사항을 문맥으로 이야기 합니다.

이 디자인 규범들 사이의 관계에 대한 이해는 한 병렬 프로그램을 위한 적절한 설계 트레이드오프를 정하는데 매우 도움이 될 수 있을 것입니다.

1. 프로그램이 크리티컬 섹션들에서 더 적은 시간을 보낼수록, 잠재적인 속도 향상은 커집니다. 이는 Amdhal의 법칙 [Amd67]과, 주어진 시간 동안 크리티컬 섹션은 오로지 하나의 CPU에 의해서만 실행될 수 있다는 사실로 인한 결과입니다.

더 자세히 이야기 하자면, 특정한 갯수의 CPU들에서 실제 성능 향상을 얻기 위해선, 프로그램이 배타적 크리티컬 섹션에서 소모하는 시간의 비율은 CPU들의 수의 역수보다 작아야 합니다. 예를 들어, 10 개의 CPU들을 사용하는 한 프로그램은 잘 확장하기 위해선 가장 한정적인 크리티컬 섹션에서는 자신의 시간 중 10분의 1 미만만을 사용해야 합니다.

⁸ 실제 세계의 병렬 시스템은 많은 디자인 규범들에 반하는 사례가 있을 것인데, 데이터 구조체 레이아웃, 메모리 사이즈, 메모리 구조 대기시간, 대역폭 제한, I/O 문제등이 그것입니다.

2. 경쟁은 많은 CPU와, 또는 벽시계 시간을 소모할 것이어서 실제 성능 향상은 사용 가능한 CPU들의 수보다 작을 것입니다. CPU들의 수와 실제 속도 향상 사이의 차이가 클수록, CPU들은 더 비효율적으로 사용될 것입니다. 유사하게, 원하는 효율성이 클수록 얻을 수 있는 속도 향상은 줄어들 것입니다.
3. 사용 가능한 동기화 도구들이 그것들이 지키는 크리티컬 섹션들에 비해 높은 오버헤드를 갖는다면, 속도 향상을 개선하는 최선의 방법은 그 도구들이 사용되는 횟수를 줄이는 것입니다(크리티컬 섹션들을 합치거나, 데이터 소유권을 사용하거나, 비대칭적으로 도구를 사용하거나(Section 9을 참고하세요), 코드 락킹과 같이 더 큰 단위를 사용하는 디자인으로 옮겨가거나 하는 방법으로).
4. 만약 크리티컬 섹션들이 그것들을 지키는 도구들에 비해 높은 오버헤드를 갖는다면, 속도 향상을 개선하는 최선의 방법은 reader/writer 락킹, 데이터 락킹, 비대칭적, 또는 데이터 소유권을 사용하는 쪽으로 옮겨가서 병렬성을 높이는 것입니다.
5. 크리티컬 섹션들이 그것들을 지키는 도구들에 비해 높은 오버헤드를 갖고 보호되는 데이터 구조체에는 수정보다 읽기가 훨씬 많이 수행된다면, 병렬성을 높이는 최고의 방법은 reader/writer 락킹이나 비대칭적 도구들을 사용하는 것입니다.
6. SMP 성능을 개선하는 많은 변경들, 예를 들어 락경쟁을 줄이는 것은 실시간 대기시간도 향상을 시킵니다 [McK05c].

Quick Quiz 6.12: 크리티컬 섹션들과 관련한 이 모든 문제들은 우리가 크리티컬 섹션이 아예 없는 non-blocking 동기화 [Her90]를 사용해야 한다는 의미는 아닌가요? ■

6.3 Synchronization Granularity

Figure 6.13는 서로 다른 동기화 빈도의 단계를 그림으로 보여주는데, 각각의 단계는 뒤의 섹션에서 다루어질 것입니다. 이 섹션들은 주로 락킹에 중점을 맞춥니다만, 비슷한 빈도 문제가 모든 동기화 문제에 깔려 있습니다.

6.3.1 Sequential Program

프로그램이 싱글 프로세서 위에서 충분히 빨리 돌아간다면, 그리고 다른 프로세스들이나 쓰레드, 또는 인터럽트 핸들러들과의 상호작용이 없다면, 동기화 기능들을

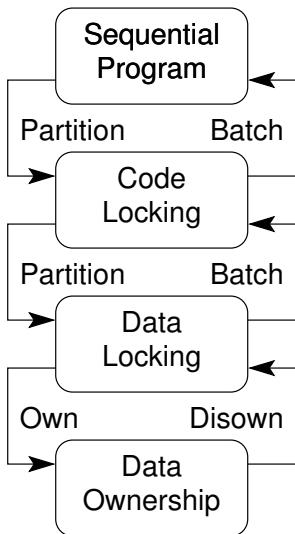


Figure 6.13: Design Patterns and Lock Granularity

모두 제거해서 그것들의 오버헤드와 복잡성을 당신으로부터 떼어놓기 바랍니다. 몇년 전에는, Moore의 법칙이 결국은 모든 프로그램을 이 카테고리로 만들 것이라 주장하던 사람들도 있었습니다. 하지만, Figure 6.14에서 볼 수 있듯이, 싱글 쓰레드의 성능의 기하급수적인 증가는 2003년 정도에 멈췄습니다. 따라서, 성능을 늘리기 위해선 병렬성을 필요로 할 것입니다.⁹ 이 새로운 트렌드가 수천개의 CPU들을 갖는 하나의 칩을 나타나게 할 것인지에 대한 논쟁은 금방 끝나지 않겠지만, Paul이 이 문장을 듀얼코어 랩탑으로 쓰고 있다는 점을 볼 때, SMP의 시대는 우리에게 와 있는 것으로 보입니다. 이더넷 대역폭이 Figure 6.15에서 볼 수 있듯이 지속적으로 성장하고 있음을 알아두는 것 역시 중요합니다. 이런 추세는 커뮤니케이션 부하를 처리하기 위해 멀티쓰레드 서버들이 나타나도록 촉진하는 역할을 할 것입니다.

이는 당신이 모든 프로그램을 멀티쓰레드 방식으로 코딩해야 한다는 의미가 아닙니다. 다시 말하지만, 프로그램이 싱글 프로세서에서 충분히 빨리 동작한다면, SMP 동기화 기능들의 오버헤드와 복잡성으로부터 당신을 멀리 하십시오. Figure 6.16에 나와 있는 해시 테이블 탐색 코드의 단순성이 이 점을 강조합니다.¹⁰ 키 포

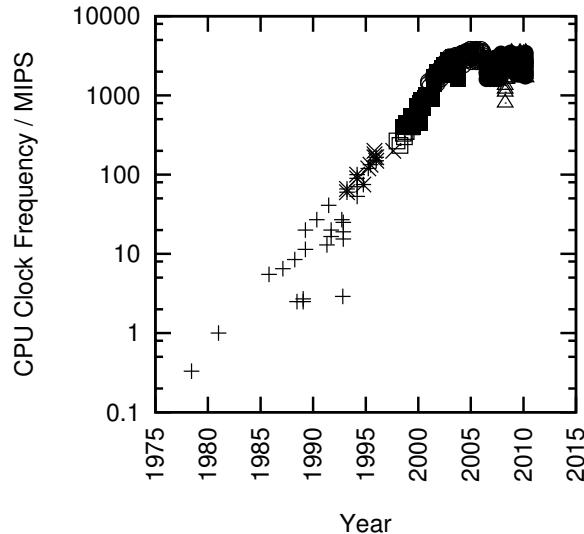


Figure 6.14: MIPS/Clock-Frequency Trend for Intel CPUs

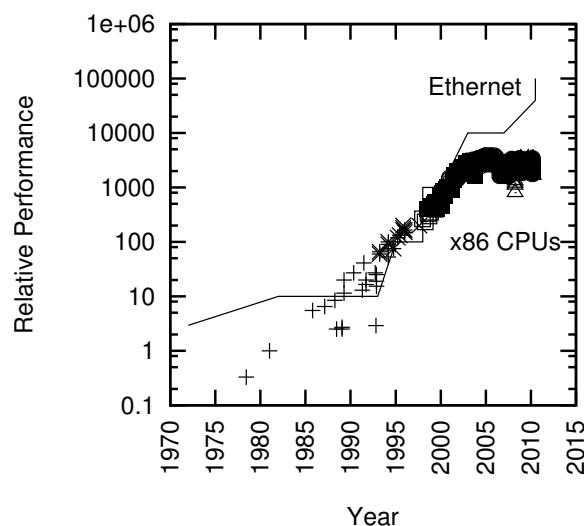


Figure 6.15: Ethernet Bandwidth vs. Intel x86 CPU Performance

⁹ 이 그림은 이론적으로 클락당 하나 이상의 인스트럭션들을 처리할 수 있는 최신 CPU들의 경우 클락 주파수를, 그리고 하나의 간단한 인스트럭션을 처리하는데 여러 클락을 필요로 하는 오래된 CPU들의 경우 MIPS를 보입니다. 이런 접근을 취하는 이유는 클락당 여러 인스트럭션들을 처리하는 최신 CPU들의 기능들은 일반적으로 메모리 시스템 성능에 제한되기 때문입니다.

¹⁰ 이 섹션의 예들은 Hart 등 [HMB06]으로부터 얻어졌으며, 여러 파일들로부터 관련된 코드를 모음으로써 경쾌함을 위해 적용되었습니다.

인트는 병렬성으로 인한 속도향상은 CPU들의 갯수에 제한된다는 것입니다. 반면, 예컨대 조심스럽게 선택된 데이터 구조와 같은 순차적 최적화를 통한 속도향상은 얼마든지 클 수 있습니다.

```

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             return (cur->key == key);
20         }
21         cur = cur->next;
22     }
23     return 0;
24 }
```

Figure 6.16: Sequential-Program Hash Table Search

반면, 이런 행복한 상황에 처해 있는 게 아니라면, 계속 읽으세요!

6.3.2 Code Locking

코드 락킹은 글로벌 락들만을 사용하기 때문에 상당히 간단합니다.¹¹ 이 방법은 기존의 프로그램을 컬티 프로세서 위에서 동작하도록 하기 위해 코드 락킹을 사용하도록 수정하는 것은 특히 쉽습니다. 만약 프로그램이 하나의 공유자원만을 가지고 있다면, 코드 락킹은 최적의 성능을 제공할 것입니다. 하지만, 많은 거대하고 복잡한 프로그램들은 많은 수행이 크리티컬 섹션에서 일어나야만 할 것을 요구하고, 이는 곧 코드 락킹이 그 확장성을 크게 제한하게 하는 결과를 초래합니다.

따라서, 전체 실행시간 중 작은 부분만을 크리티컬 섹션에서 수행하거나 작은 확장성만이 필요한 프로그램들에 대해서만 코드 락킹을 사용해야 합니다. 이런 경우, 코드 락킹은 Figure 6.17에서 볼 수 있듯, 순차적인 버전과 매우 유사하고 상대적으로 간단한 프로그램을 제공할 것입니다. 하지만, Figure 6.16의 `hash_search()`에서의 단순한 비교값 리턴은 이제 리턴 전에 락을 풀어야 하기 때문에 세개의 문장이 되었음을

니다.

¹¹ 그게 아니라 데이터 구조체 안에 락들을 가지고 있거나, 자바의 경우, `synchronized` 인스턴스로 클래스들을 사용한다면, Section 6.3.3에 설명된 “데이터 락킹”을 사용하고 있는 겁니다.

알아 두시기 바랍니다.

```

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             retval = (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

안타깝게도, 코드 락킹은 여러 CPU들이 락을 동시에 획득하려 하는, “락 경쟁” 상황에 빠지기 쉬운 경향이 있습니다. 한 무리의 어린 아이들(또는 아이처럼 행동하는 어른들의 무리들)을 보살펴야 하는 SMP 프로그래머들은 곧바로 Figure 6.18에 그려진 것처럼 뭉가 단 하나를 사용하는 것의 위험성을 깨달을 것입니다.

이 문제에 대한 해결책인 “데이터 락킹”이 다음 섹션에 설명됩니다.

6.3.3 Data Locking

많은 데이터 구조체들이 각자 자기의 락을 갖는 조각들로 분할될 수 있습니다. 이렇게 되면 데이터 구조체의 각 조각들의 크리티컬 섹션들은 병렬적으로 실행될 수 있습니다. 각 조각의 크리티컬 섹션의 인스턴스는 한번에 하나씩만 수행될 수 있긴 하지만요. 경쟁상황이 줄어야만 하고, 동기화 오버헤드가 속도향상을 제한하지 않는 경우에는 데이터 락킹을 사용해야 합니다. 데이터 락킹은 여러 데이터 구조체 사이에 존재하는 너무 큰 크리티컬 섹션의 인스턴스들을 분산시킴으로써 경쟁상황을 줄여주는데, 예를 들어 Figure 6.19처럼 해시 테이블에서 해시 베킷 별로 크리티컬 섹션을 두는 식입니다. 향상된 확장성은 약간 추가적인 데이터 구조체인 `struct bucket`의 형태로 복잡도를 약간 증가시킵니다.

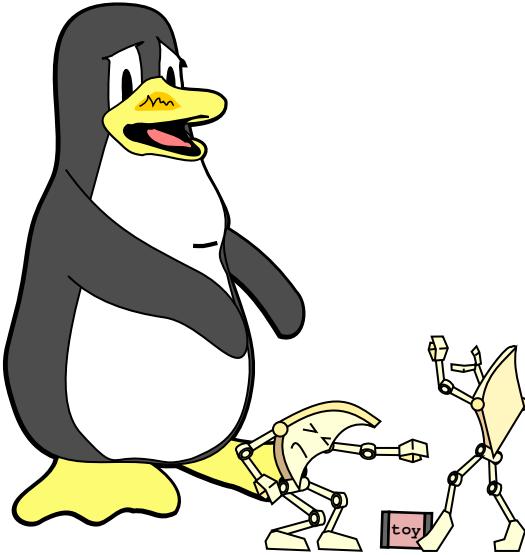


Figure 6.18: Lock Contention

다.

Figure 6.18에서 보였던 경쟁적인 상황과 달리, 데이터 락킹은 Figure 6.20에 그려진 것처럼 조화를 향상시킵니다 — 그리고 병렬 프로그램에서, 이는 거의 항상 향상된 성능과 확장성으로 이어집니다. 이런 이유로, 데이터 락킹은 DYNIX 와 DYNIX/ptx 운영체제의 Sequent에서 매우 많이 사용되었습니다 [BK85, Inm85, Gar90, Dov90, MD92, MG92, MS93].

하지만, 어린 아이들을 돌봐 본 사람이라면 증명할 수 있듯이, 충분히 가서 놀 수 있는 공간을 주는 것이 평안을 보장하지는 않습니다. 예를 들어, 리눅스 커널은 파일과 디렉토리들의 캐시 (“dcache” 라 불립니다)를 갖습니다. 이 캐시의 각 원소들은 자신의 락을 갖습니다만, 루트 디렉토리에 해당하는 원소들과 그것의 직접적인 자식들은 다른 원소들에 비해 훨씬 많이 순회됩니다. 이는 많은 CPU들이 이 자주 접근되는 원소들의 락에 경쟁을 하게 되는 결과를 초래해서, Figure 6.21에 보여진 것과 같은 상황을 초래하고 맙니다.

많은 경우, 알고리즘들은 데이터 스케줄링을 줄이도록 설계될 수 있고, 어떤 경우에는 아예 그것들을 없애버릴 수도 있습니다 (리눅스 커널의 dcache에서도 가능한 것으로 나타난 것처럼요 [MSS04]). 데이터 락킹은 종종 해시 테이블처럼 분할될 수 있는 데이터 구조체에 사용됩니다만, 여러 존재가 각각 어떤 데이터 구조체의 인스턴스로 표현될 수 있는 경우에도 사용됩니다. 리눅스 커널 2.6.17 버전의 태스크 리스트는 후자의 한 예로, 각 태스크 구조체는 자신의 `proc_lock`을 갖습니다.

```

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

다이나믹하게 할당되는 구조체들에서의 데이터 락킹의 핵심 문제는 해당 구조체가 해당 락이 잡혀있는 동안은 존재하는 상태를 유지해야 한다는 것입니다. Figure 6.19의 코드는 이 문제를 락들을 정적으로 할당된 해시 버킷들에 넣어두고, 절대 메모리에서 해제시키지 않는 것으로 해결합니다. 하지만, 이 트릭은 해시 테이블의 크기가 바뀔 수 있다면 락들이 다이나믹하게 할당되어야 하므로 제대로 동작하지 않을 것입니다. 이 경우, 해시 버킷들을 그 락들이 잡혀있는 동안은 메모리 해제되지 않도록 하는 어떤 수단이 필요할 것입니다.

Quick Quiz 6.13: 구조체가 그것의 락이 잡혀 있는 동안은 메모리 해제 되지 않도록 할 수 있는 방법들은 어떤 것들이 있을까요? ■

6.3.4 Data Ownership

데이터 소유권은 주어진 데이터 구조체를 쓰레드들이나 CPU 들로 쪼개서, 각 쓰레드/CPU 는 데이터의 자신에게 할당된 부분집합을 어떤 동기화 오버헤드 없이 접근할 수 있습니다. 하지만, 어떤 쓰레드가 다른 쓰레드의 데이터에 접근하길 원한다면, 이는 곧바로 될 수는 없습니다. 대신, 이 쓰레드는 다른 쓰레드와 먼저 통신

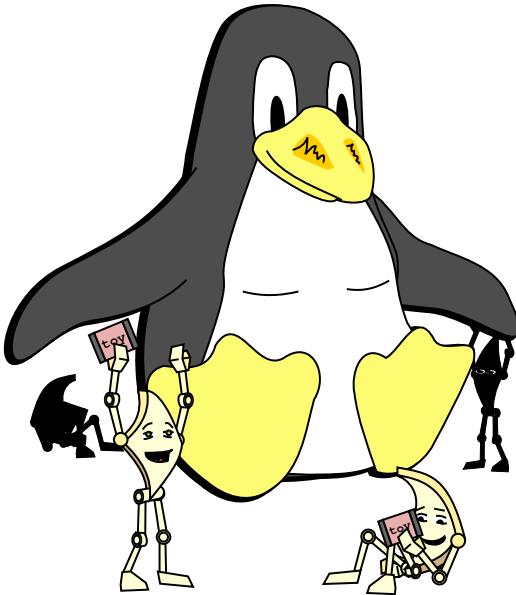


Figure 6.20: Data Locking

을 해서 다른 쓰레드가 그 일을 대신 해주거나, 또는 그 데이터의 소유권을 이전해 주도록 해야 합니다.

데이터 소유권은 불가사의해 보일 수 있지만, 매우 자주 사용됩니다:

1. 한 CPU나 쓰레드에 의해서만 접근될 수 있는 변수(C와 C++에서의 `auto` 변수와 같은)들은 모두 해당 CPU나 프로세스에게 소유되어 있습니다.
2. 사용자 인터페이스의 한 인스턴스는 해당 사용자의 컨텍스트를 소유합니다. 병렬 데이터베이스 엔진과 상호작용하는 어플리케이션들은 그것들이 순차적 프로그램인 것마냥 작성되는 것이 매우 흔한 일입니다. 그런 어플리케이션들은 사용자 인터페이스와 그/그녀의 현재 동작을 소유합니다. 명시적인 병렬성은 따라서 데이터베이스 엔진 그 자체에 국한되어 있습니다.
3. 파라미터를 사용하는 시뮬레이션들은 종종 각 쓰레드가 파라미터 공간의 특정 영역에 소유권을 갖게 하는 방법으로 병렬화 되곤 합니다. 이런 타입의 문제를 위한 컴퓨팅 프레임워크도 존재합니다 [UoC08].

상당히 많은 공유가 존재한다면, 이 쓰레드들이나 CPU들 사이의 통신은 상당한 복잡도와 오버헤드를 만들어낼 것입니다. 더 나아가서, 가장 많이 사용되는 데이터가 단일 CPU에게 소유되어 있게 된다면, 이 CPU

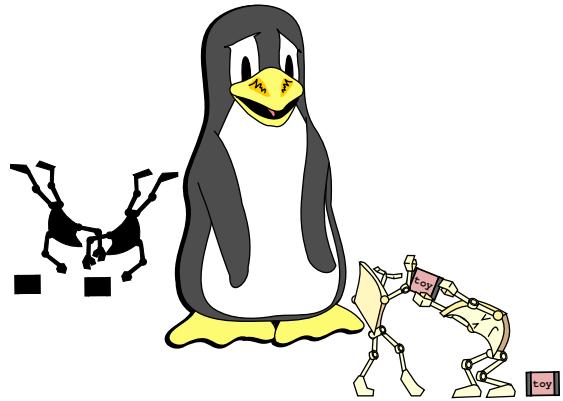


Figure 6.21: Data Locking and Skew

는 “핫스팟”이 될 것이고, Figure 6.21에 그려진 것과 같은 결과를 낼 것입니다. 하지만, 공유가 필요하지 않은 상황이라면, Figure 6.16에 보여진 것처럼 데이터 소유권은 이상적인 성능을 내고, 순차적 프로그램처럼 간단해질 수 있을 것입니다. 그런 상황은 종종 “당황스러울 정도로 병렬적”이라 불리곤 하고, Figure 6.20에서 앞서 본것과 같은 상황과 닮아 있습니다.

또 다른 중요한 데이터 소유권 상황은 데이터가 읽기 전용인 경우, 모든 쓰레드가 복사본을 통해 그것을 “소유” 할 수 있는 경우입니다.

데이터 소유권은 Chapter 8에서 좀 더 자세히 다뤄질 겁니다.

6.3.5 Locking Granularity and Performance

이 섹션은 락킹 빈도와 성능 사이의 관계를 수학적인 동기화 효율성 관점에서 살펴봅니다. 수학이 지루한 독자분들은 이 섹션을 건너뛰셔도 됩니다.

사용할 방법은 하나의 공유되는 글로벌 변수에 대해 동작하는 동기화 메커니즘의 효율성을 위한 간단한 큐잉 모델을 M/M/1 큐를 기반으로 사용해 보는 것입니다. M/M/1 큐잉 모델들은 지수적으로 분포되는 “도착간 비율 (inter-arrival rate)” λ 와 지수적으로 분포되는 “서비스 비율 (service rate)” μ 에 기반합니다. 도착간 비율 λ 는 만약 동기화 비용이 전혀 들지 않는다면 시스템이 처리할 수 있는, 초당 동기화 오퍼레이션들의 평균 숫자로 생각될 수 있는데, 달리 말하자면 λ 는 작업의 비동기 유닛의 오버헤드의 역수입니다. 예를 들어, 일의 각 유닛이 트랜잭션이고, 각 트랜잭션이 처리되는데 동기화 오버헤드를 제외하고 1 밀리세컨드가 걸린다면, λ 는 초당 1,000 트랜잭션이 될 것입니다.

서비스 비율 μ 는 비슷하게 정의됩니다만, CPU 들이 각자의 동기화 오퍼레이션들을 완료하기를 기다려야 한다는 사실을 무시하고 각 트랜잭션의 오버헤드가 존재하지 않는다면 시스템이 1초 안에 처리할 수 있는 동기화 오퍼레이션들의 수의 평균으로, 달리 말하자면, μ 는 컨텐션이 없을 때의 동기화 오버헤드라고 생각될 수 있겠습니다. 예를 들어, 각 동기화 오퍼레이션이 어토믹 값 증가 인스트럭션을 내포하고 있고, 컴퓨터 시스템은 각 CPU 에서 각자의 변수에 25 나노세컨드마다 어토믹 값 증가 인스트럭션을 수행할 수 있다고 해 봅시다.¹² 따라서 μ 의 값은 초당 40,000,000 어토믹 값 증가 일 것입니다.

물론, λ 의 값은 CPU 수가 늘어나면 함께 늘어날텐데, 각 CPU 가 독립적으로 트랜잭션들을 처리할 수 있기 때문입니다 (다시 말하지만, 동기화를 무시합니다):

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

n 은 CPU 들의 갯수이고 λ_0 는 단일 CPU 의 트랜잭션 처리 가능량입니다. 단일 CPU 가 하나의 트랜잭션을 처리하는데 걸릴 것으로 기대되는 시간은 $1/\lambda_0$ 임을 기억해 두십시오.

이 CPU 들은 다른 CPU 들이 각각 하나의 공유 변수의 값을 증가시킬 동안 “줄을 서서 기다려야” 하기 때문에, 기대되는 전체 대기 시간을 표현하는데 M/M/1 큐잉 모델을 다음과 같이 사용할 수 있습니다:

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

앞의 λ 값을 대입하면:

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

이제, 효율성은 동기화 없을 때 트랜잭션 하나를 처리하는데 필요한 시간 ($1/\lambda_0$) 과 동기화를 포함해서 필요한 시간 ($T + 1/\lambda_0$) 사이의 비율입니다:

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

앞의 T 값을 대입하고 간략화 하면:

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

¹² 물론, 같은 공유 변수의 값을 증가시키는 8 개의 CPU 가 존재한다면, 각각의 CPU 는 자신의 값 증가 작업을 위해 25 나노세컨드를 소모하기 전에 다른 CPU 들이 각자의 값 증가 작업을 마무리 할 때 까지 175 나노세컨드를 기다려야 할 것입니다. 실제로는, 이 대기는 더 길어질텐데 그 변수를 한 CPU 에서 다른 CPU 로 옮기는 시간도 걸리기 때문입니다.

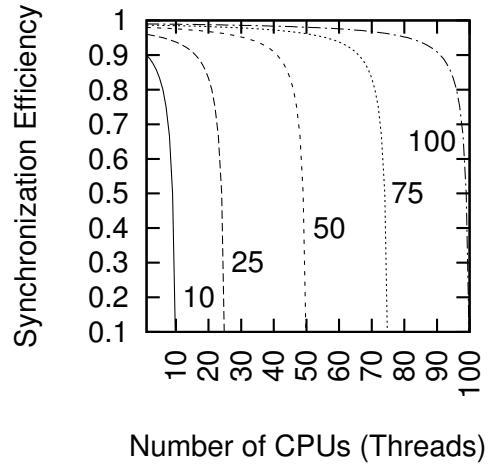


Figure 6.22: Synchronization Efficiency

하지만 μ/λ_0 의 값은 그저 트랜잭션을 처리하는데 필요한 시간과 (경쟁상황이 없는 상태에서의) 동기화 자체 오버헤드 간의 비율입니다. 이 비율을 f 라고 하면, 이렇게 됩니다:

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

Figure 6.22 는 동기화 효율성 e 가 CPU/쓰레드 수 n 에 의해 변화되는 모습을 오버헤드 비율 f 몇개의 값과 함께 보여줍니다. 예를 들어, 25 나노세컨드 걸리는 어토믹 값 증가 연산을 사용하면 $f = 10$ 선은 각 CPU 가 매 250 나노세컨드마다 어토믹 값 증가 연산을 시도하고, $f = 100$ 라인은 각 CPU 가 수천개의 인스트럭션을 처리될 수 있는 시간인 2.5 마이크로세컨드마다 어토믹 값 증가 연산을 시도합니다. 각 조합의 결과가 CPU 나 쓰레드의 수가 늘어남에 따라 급격하게 떨어지는 것으로 보아, 하나의 글로벌 공유 변수에의 어토믹 조정을 통한 동기화 메커니즘은 현재의 하드웨어에서 많이 사용되면 잘 확장되지 못할 것이라 결론 내릴 수 있습니다. 이건 이 규칙들을 수학적으로 그려본 것으로, Chapter 5 에서 이야기한 병렬 카운팅 알고리즘을 이끌어내게 합니다.

이 효율성 컨셉은 정규적인 동기화가 적거나 아예 없을 때에도 효과적입니다. 예를 들어, 한 행렬의 행이 (“dot product”로) 다른 행렬의 열로 곱해져 세번째 행렬을 만들어내는 행렬 곱셈을 생각해 봅시다. 이 오퍼레이션들은 서로 겹치지 않기 때문에, 첫번째 행렬의 행들을 쓰레드들에 분할시키고 각 쓰레드는 결과 행렬의 연관된 행을 계산하는 것이 가능합니다. 따라서 이 쓰레드들은 matmul.c 에서와 같이 아무런 동기화 오버헤드 없이 완전히 독립적으로 동작할 수 있습니다. 따라서 병

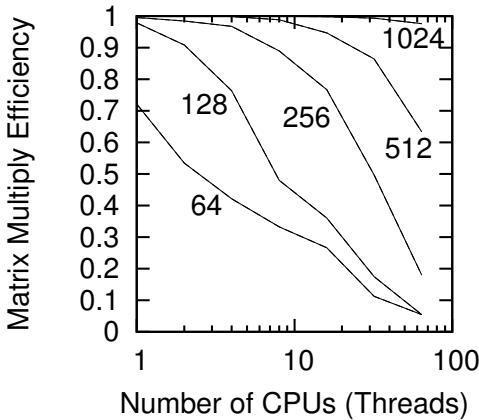


Figure 6.23: Matrix Multiply Efficiency

렬적 행렬 곱셈은 완벽한 효율성, 1.0을 가질 것이라 예상할 수도 있습니다.

하지만, Figure 6.23는 다르게 이야기하는데, 특히 64 행 64 열 행렬 곱셈에서는 0.7보다 낮은 효율성을 절대 갖지 못합니다. 싱글쓰레드로 동작하는데도 불구하고 말이지요. 512 행 512 열 행렬 곱셈의 효율성은 10 쓰레드 아래에서는 1.0보다 조금 작은 것으로 측정되며 심지어 1024 행 1024 열 행렬 곱셈조차도 수십 쓰레드에서는 완벽한 효율성을 벗어나게 되고 맙니다. 더도 아니고 덜도 아니고, 이 그림은 몰아서 처리하기의 성능과 확장성에서의 이점을 분명하게 보여주고 있으며, 이로 인해 당신의 돈의 가치도 알 수 있게 해줍니다.

Quick Quiz 6.14: 싱글쓰레드로 동작하는 64 행 64 열 행렬 곱셈이 어떻게 1.0보다 낮은 효율성을 가질 수 있죠? Figure 6.23의 모든 조합에서의 결과들이 한 쓰레드에서만 돌아갈 때에는 정확히 1.0의 효율성을 보여야 하는 거 아닌가요? ■

이런 비효율성 아래, Section 6.3.3에서 이야기한 데 이터 락킹과 같이 더 확장성 있는 방법을 생각해 보거나 다음 섹션에서 다룰 병렬 빠른 수행 경로 해결책을 고려해 보는 것도 가치가 있을 것입니다.

Quick Quiz 6.15: 행렬 곱셈에서 데이터 병렬화 기법이 어떻게 도움이 될 수 있나요? 그건 이미 병렬적인 데이터잖아요!!! ■

6.4 Parallel Fastpath

잘게 쪼개진 (그리고 따라서 일반적으로 높은 성능을 갖는) 설계들은 굵게 쪼개진 설계들에 비해 일반적으로 더 복잡합니다. 많은 경우, 대부분의 오버헤드는 코드의 작은 부분에서 발생합니다 [Knu73]. 그러니 그 작은 부분에 집중하는 노력을 가져보는게 어떨까요?

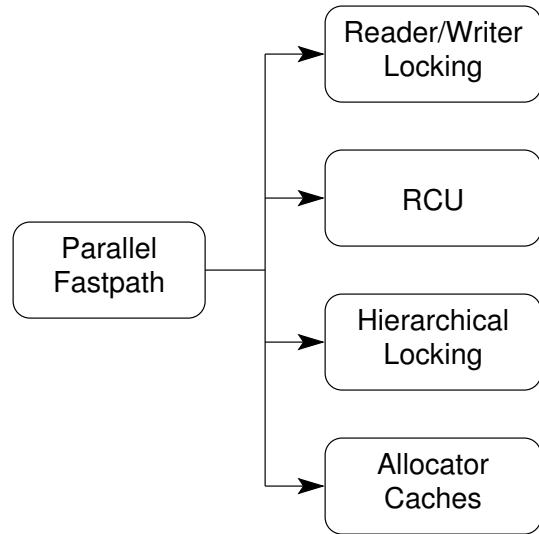


Figure 6.24: Parallel-Fastpath Design Patterns

이게 병렬 빠른 수행 경로 디자인 패턴을 뒷받침하는 아이디어로, 전체 알고리즘을 적극적으로 병렬화 하려 하면 요구되는 복잡성을 일으키지 않고 일반적인 경우를 위한 코드 수행 경로를 적극적으로 병렬화 시키는 것입니다. 이를 위해선 병렬화 하려는 특정 알고리즘만 이해해선 안되고, 그 알고리즘이 목표로 하는 워크로드에 대해서도 이해해야만 합니다. 병렬 빠른 수행 경로를 만들기 위해선 커다란 창조성과 설계 노력이 종종 필요하곤 합니다.

병렬 빠른 수행 경로는 서로 다른 패턴들 (하나는 빠른 수행 경로를 위해, 다른 경우를 위해 또 하나)을 조합하고 따라서 임시적 패턴입니다. 다음의 병렬 빠른 수행 경로의 예들은 Figure 6.24에 그려진 것처럼 그 자신의 패턴을 정당화 하기 충분할 만큼 자주 일어납니다:

1. Reader/Writer 락킹 (아래의 Section 6.4.1에서 설명합니다).
2. 고성능을 위해 Reader/Writer 락킹을 대체할 수 있으며 이 챕터에서는 더이상 설명되지 않을 Read-copy update (RCU).
3. Section 6.4.2에서 다뤄질 계층적 락킹 ([McK96]).
4. 리소스 할당자 캐시 ([McK96, MS93]). 더 자세한 내용을 위해선 Section 6.4.3을 참고하십시오.

6.4.1 Reader/Writer Locking

동기화 오버헤드가 무시할만 하다면 (예를 들어, 프로그램이 커다란 크리티컬 섹션들에 굵게 쪼개진 병렬성

```

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

을 사용한다면), 그리고 그 크리티컬 섹션들의 작은 부분들만이 데이터를 수정한다면, 여러 읽는 작업들을 병렬로 수행될 수 있도록 하면 확장성을 크게 개선시킬 것입니다. 쓰기 작업은 읽기 작업과도, 다른 쓰기 작업들과도 배타적으로 수행되어야 합니다. reader-writer 락킹의 많은 구현이 존재하는데, Section 4.2.4 에 설명된 POSIX 구현도 그 중 하나입니다. Figure 6.25 는 해시 테이블이 reader-writer 락킹을 사용해 어떻게 구현될 수 있는지 보입니다.

Reader/writer 락킹은 비대칭적 락킹에 대한 하나의 간단한 예입니다. Snaman [ST87] 은 여러 클러스터 시스템에서 사용되는, 더 화려한 여섯개 모드의 비대칭적 락킹 디자인을 설명합니다. 일반적인 락킹과 reader-writer 락킹은 Chapter 7 에서 특별히 자세히 설명됩니다.

6.4.2 Hierarchical Locking

계층적 락킹의 아이디어는 잘게 쪼개진 락을 잡는 동작을 하는 동안만 잡는 굵게 쪼개진 락을 두자는 것입니다. Figure 6.26 는 해시 테이블 탐색에 계층적 락킹이 어떻게 사용될 수 있을지 보여주는데, 또한 이 방법의 커다란 단점 역시 보여줍니다: 두번째 락을 잡기 위한 오버헤드를 감내했지만, 그 락은 짧은 시간동안만 잡습니다. 이 경우, 간단한 데이터 락킹 방법은 더 간단하고

```

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

더 좋은 성능을 보일 것입니다.

Quick Quiz 6.16: 계층적 락킹이 잘 동작할 만한 상황은 뭐가 있을까요? ■

6.4.3 Resource Allocator Caches

이 섹션에서는 병렬 고정 크기 블럭 메모리 얼로케이터의 단순화된 개요를 제공합니다. 더 자세한 설명은 literature [MG92, MS93, BA01, MSK01] 나 리눅스 커널 [Tor03c] 에서 찾을 수 있습니다.

6.4.3.1 Parallel Resource Allocation Problem

병렬 메모리 할당자가 마주하는 기본적인 문제는 일반적인 경우의 매우 빠른 메모리 할당과 해제 기능을 제공해야 하는 필요와 불리한 할당 / 해제 패턴들을 마주했을 때 효과적으로 메모리를 분산시켜야 하는 필요성 사이의 갈등입니다.

이 갈등을 보기 위해, 이 문제에 데이터 소유권을 직접적으로 활용한 경우를 생각해 봅시다 — 단순히 각

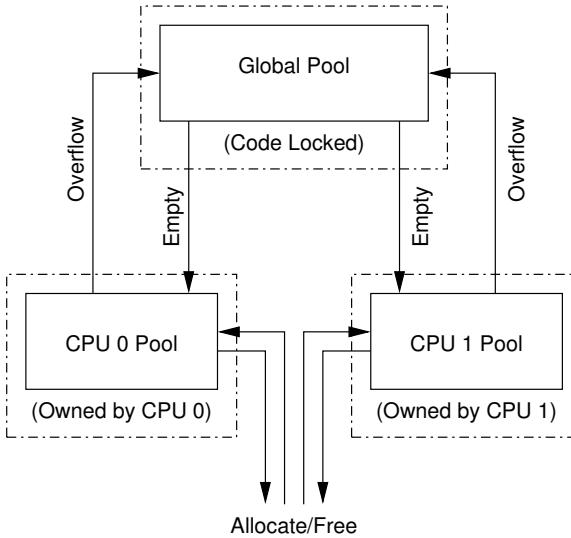


Figure 6.27: Allocator Cache Schematic

CPU 가 자기 뭉을 소유하도록 메모리를 분할하는 방법입니다. 예를 들어, 두개의 CPU 를 갖고 2 기가바이트의 메모리를 갖는 시스템 (제가 바로 지금 글을 쓰고 있는 기계와 같습니다) 이라고 생각해 봅시다. 간단히 각 CPU 에 1 기가바이트씩 메모리를 할당해 주고 각 CPU 가 그 자신이 소유하고 있는 메모리 덩어리에 접근하도록 할 수 있는데, 이렇게 되면 락킹의 필요성과 복잡성, 그리고 오버헤드들이 없습니다. 불행히도, 이 간단한 계획은 간단한 생산자-소비자 워크로드 같은 경우에서 일어날 수 있는, CPU 0 이 모든 메모리를 할당받고 CPU 1 이 그걸 해제하는 알고리즘이 있다면 망가질 수 있습니다.

다른 극단적 방법인 코드 락킹의 경우에는 과한 락경쟁과 오버헤드로 고통받게 될 수 있습니다 [MS93].

6.4.3.2 Parallel Fastpath for Resource Allocation

일반적으로 사용되는 방법은 각각의 CPU 가 적당한 캐시나 블럭들을 소유하는 별별 빠른 수행 경로를 사용하고 추가적인 블럭들을 위한 공유 풀에는 커다란 코드 락킹을 사용해서 관리하는 방법입니다. 한 CPU 가 메모리 블럭들을 독점하는 것을 막기 위해, 각 CPU 의 캐시에 존재할 수 있는 블럭들의 갯수에 제한을 걸어 둡니다. 두개의 CPU 가 있는 시스템에서, 메모리 블럭들의 흐름은 Figure 6.27 에 보여진 대로일 것입니다: 한 CPU 가 자신의 풀이 꽉 차서 블럭을 하나 해제하려고 할 때에는, 블럭들을 글로벌 풀에 보내고, 비슷하게, 그 CPU 가 자신의 풀이 비어서 블럭을 할당받으려 할 때에는 글로벌 풀로부터 블럭들을 얻어옵니다.

```

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

6.4.3.3 Data Structures

Figure 6.28 에 할당자 캐시들의 “장난감” 구현을 위한 실제 데이터 구조체가 있습니다. Figure 6.27 의 “Global Pool” 은 struct globalmempool 타입의 globalmem 으로 구현되었고 struct percpumempool 타입의 per-CPU 변수 percpumem 으로 두개의 CPU 별 풀들이 구현되었습니다. 이 데이터 구조체들 둘 다 각자의 pool 필드 안의 블럭들로의 포인터들의 배열을 가지고 있는데, 이것들은 인덱스 0부터 위쪽으로 채워져 나갑니다. 따라서, 만약 globalmem.pool[3] 이 NULL 이라면, 인덱스 4부터 위쪽인 이 배열의 나머지들 역시 모두 NULL 이어야 합니다. cur 필드는 pool 배열의 꽉찬 원소들 중 가장 높은 숫자의 인덱스를 갖는데, 만약 모든 원소들이텅 비어있다면 -1을 갖습니다. globalmem.pool[0] 부터 globalmem.pool[globalmem.cur] 사이의 모든 원소들은 반드시 꽉 차 있어야 하고, 나머지 것들은 모두 비어있어야만 합니다.¹³

풀 데이터 구조체의 동작이 어떻게 될지에 대한 그림이 Figure 6.29 에 있는데, 여섯개의 박스들은 pool 필드를 구성하는 포인터들의 배열을 의미하며, 그 앞의 숫자는 cur 필드를 의미합니다. 색이 칠해진 박스들은 NULL 이 아닌 포인터들을 의미하며, 비어있는 박스들은 NULL 포인터들을 의미합니다. 중요하지만 좀 혼란스러울 수도 있는 이야기입니다만 이 데이터 구조체가 항상 지키게 되는 사실 (불변식, invariant) 은, cur 필드는 항상 NULL 이 아닌 포인터들의 수보다 하나 작을 것이란 점입니다.

¹³ 두 풀 사이즈 (TARGET_POOL_SIZE 와 GLOBAL_POOL_SIZE) 모두 비현실적으로 작습니다만, 이 작은 크기가 이 프로그램의 행동이 어떻게 이루어지는지 이해하기 위해 프로그램을 단계별로 실행시켜 나가기 편하게 도와줄 겁니다.

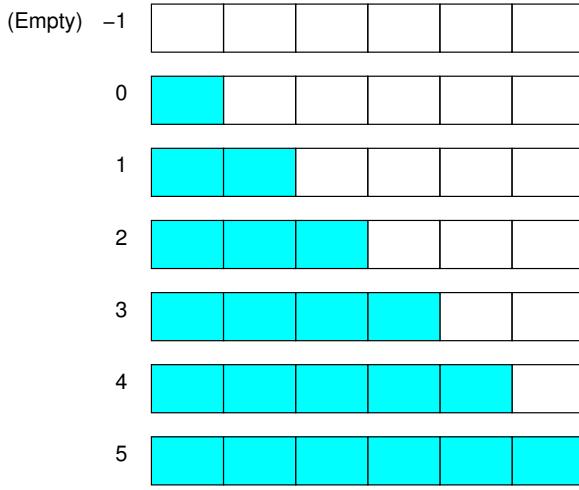


Figure 6.29: Allocator Pool Schematic

6.4.3.4 Allocation Function

할당을 하는 함수인 `memblock_alloc()` 을 Figure 6.30에서 확인할 수 있습니다. Line 7에서는 현재 쓰레드의 per-thread 풀을 가져오고, line 8에서 그에 비어있는지 확인합니다.

만약 그렇다면, line 9-16에서 line 9에서 획득하고 line 16에서 해제하는 스핀락 아래 글로벌 풀로부터 해당 per-thread 풀을 채우려 시도합니다. Line 10-14는 블럭들을 글로벌에서 per-thread 풀로 로컬 풀이 목표로 하는 크기 (절반)에 도달하거나 글로벌 풀이 텅 빌 때까지 옮기고, line 15에서는 per-thread 풀의 수를 올바른 값으로 설정합니다.

어떤 경우든, line 18에서 per-thread 풀이 여전히 비어 있는지 확인하고, 만약 그렇지 않다면, line 19-21에서 블럭 하나를 제거하고 그걸 리턴합니다. 그렇지 않다면, line 23에서 메모리가 부족하다는 슬픈 이야기를 전합니다.

6.4.3.5 Free Function

Figure 6.31는 메모리 블럭 해제 함수를 보입니다. Line 6에서는 이 쓰레드의 풀로의 포인터를 얻어오고 line 7에서 이 per-thread 풀이 꽉 차 있는지 확인합니다.

만약 그렇다면, line 8-15에서 이 per-thread 풀의 절반을 글로벌 풀로 비워내는데, 이 때 line 8과 14에서 글로벌 풀을 위한 스핀락을 각각 잡고 풁니다. Line 9-12에서는 로컬에서 글로벌 풀로 블럭들을 옮기는 루프를 구현하고 있으며, line 13에서는 per-thread 풀의 카운트를 올바른 값으로 재조정 합니다.

어떤 경우든, line 16에서는 새로 해제된 블럭을 per-

```

1 struct memblock *memblock_alloc(void)
2 {
3     int i;
4     struct membblock *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

```

1 void membblock_free(struct membblock *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

thread 풀에 넣습니다.

6.4.3.6 Performance

대략적인 성능 결과¹⁴ 가 Figure 6.32에 있는데, 이 테이터를 위한 성능 평가 실험은 1GHz로 동작하는 듀얼 코어 Intel x86 (CPU 당 4300 bogomips)에서 각 CPU의 캐시에 최대 여섯개의 블럭들을 주고 수행되었습니다. 이 마이크로 벤치마크에서, 각 쓰레드는 반복적으로 한 그룹의 블럭들을 할당받고 그 그룹의 모든 블럭들을 해제하는데, 이 그룹에 속하는 블럭들의 갯수는 x축에

¹⁴ 이 테이터는 통계적으로 유의미한 방식으로 수집되지는 않았습니다. 따라서 많은 비판적 시각과 의심을 가지고 봐야만 합니다. 좋은 테이터 수집과 요약을 위한 방법은 Chapter 11에서 이야기 됩니다. 그렇다고 하니, 반복해서 진행한 수행은 비슷한 결과를 내놓았고, 이 값들은 비슷한 알고리즘들의 더 세심하게 설계된 성능 평가 실험과 비슷한 결과를 보입니다.

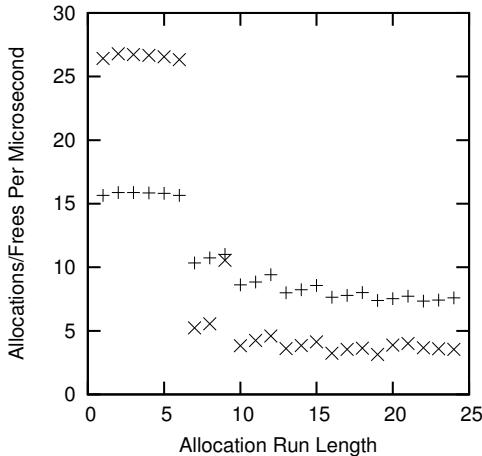


Figure 6.32: Allocator Cache Performance

표시되는 “allocation run length (할당 수행 시간 길이)”와 같습니다. Y 축은 마이크로세컨드당 성공한 할당/해제 짝들의 갯수를 보입니다 — 실패한 할당들은 카운트되지 않습니다. “X”는 두 쓰레드 수행의 결과이고, “+”는 단일 쓰레드 수행의 결과입니다.

할당 수행 시간 길이가 6일 때 까지는 선형적으로 확장되고 훌륭한 성능을 보이지만, 6보다 큰 할당 수행 시간 길이에서는 부족한 성능을 보이고 거의 항상 음의 확장도를 보임에 주의하시기 바랍니다. 따라서 TARGET_POOL_SIZE의 크기를 충분히 크게 잡는게 상당히 중요한데, 다행히도 실제 사례 [MSK01]에서는 그렇게 하기가 상당히 쉬운 경우가 일반적이고, 특히 오늘날의 큰 메모리에선 더욱 그렇습니다. 예를 들어, 대부분의 시스템에서 TARGET_POOL_SIZE를 100으로 잡는 것은 상당히 합리적으로, 이 경우 전체 시간의 99%는 할당과 해제 작업들은 per-thread 풀에서만 이루어질 것이 보장됩니다.

이 그림에서 볼 수 있듯이, 데이터 소유권이 적용될 수 있는 일반적인 경우의 상황 (할당 수행 시간이 6 미만일 때)에서는 락을 반드시 사용해야만 하는 상황들에 비해 훨씬 향상된 성능을 보입니다. 일반적인 경우에 동기화를 피하는 것은 이 책 전체에 걸쳐 반복되는 주제가 될 것입니다.

Quick Quiz 6.17: Figure 6.32에서, 세개의 샘플들마다 할당 수행 시간 길이가 증가함에 따라 성능이 따라 증가하는 패턴이 존재하는데, 예를 들어, 할당 수행 시간 길이 10, 11, 12의 경우입니다. 왜 그런거죠? ■

Quick Quiz 6.18: 할당 실패 횟수들은 두개 쓰레드를 사용한 테스트에서 할당 수행 길이가 19 이상일 때 발견되었습니다. 글로벌 풀 사이즈가 40이고 per-thread 타겟 풀 사이즈 s 가 3이고, 쓰레드의 갯수 n 가 2이며,

per-thread 풀들은 초기에 비어있고 메모리를 아무도 사용하고 있지 않다고 가정하는 조건 하에서, 할당 실패가 발생할 수 있는 최소의 할당 수행 시간 길이 m 은 무엇일까요? (각 쓰레드는 메모리의 m 블럭을 할당받고 m 블럭의 메모리를 해제하는 것을 반복함을 다시 이야기 합니다.) 그 대신에, n 쓰레드들이 각각 풀 사이즈 s 를 갖는다면, 그리고 각 쓰레드가 먼저 메모리의 m 개 블럭을 할당받고 그 m 블럭들을 해제하는 것을 반복한다고 하는 조건이라면, 글로벌 풀 사이즈는 얼마나 커야 할까요? Note: 올바른 답을 얻기 위해서는 `smpalloc.c` 소스 코드를 자세히 들여다보고, 한단계 한단계씩 들여다봐야 할 겁니다. 분명 경고했어요! ■

6.4.3.7 Real-World Design

이 장난감 병렬 리소스 할당자는 상당히 간단했습니다만 실제 세계에서의 설계는 이 방법으로부터 여러가지 방법으로 확장됩니다.

먼저, 실제 세계에서의 할당자는 넓은 범위의 할당 크기들을 다룰 수 있어야 하는데, 이 장난감 예제에서 보았던 단일한 크기와 반대됩니다. 이를 해결하는 대중적인 방법 하나는 1980년대 후반 BSD 메모리 할당자 [MK88] 같은 방식으로, 고정된 크기들의 집합을 제공하는 것인데, 외부의, 그리고 내부의 파편화 문제를 균형 맞추기 위해 그렇게 공간을 사용합니다. 이렇게 하는 것은 곧 “globalmem” 변수가 크기별로 복사본이 존재해야 함을 의미하고, 연관된 락 역시 비슷하게 복사되어야 해서, 이 장난감 프로그램의 코드 락킹보다는 데이터 락킹에 가까운 결과를 이끌어낼 것입니다.

두번째로, 상품 단계 품질의 시스템들은 메모리의 용도를 바꿀수도 있어야만 하는데, 다시 말해 그것들은 블럭들을 합체해서 페이지 [MS93] 와 같은 더 큰 구조체로 만들어질 수도 있어야만 합니다. 이 합체 역시 락으로 보호되어야 하는데, 이것 역시 앞의 것들과 마찬가지로 크기별로 복사본이 있어야 할 수 있습니다.

세번째로, 합쳐진 메모리는 아랫단의 메모리 시스템에 반환되어야만 하고, 메모리의 페이지들은 또한 그 아랫단의 메모리 시스템들로부터 할당되어야만 합니다. 이 단계에서 필요한 락킹은 이 아랫단의 메모리 시스템에 의존적일 것입니다만, 코드 락킹이 될 수도 있습니다. 코드 락킹은 종종 이 단계에서 받아들여질 수 있는데, 이 단계에 도달하는 것은 잘 설계된 시스템에서는 매우 희박하게 이루어지기 때문입니다 [MSK01].

이 실제 세계 설계의 커다란 복잡도에도 불구하고, 그 아래 자리잡고 있는 아이디어는 똑같습니다 — Table 6.1에 보인것과 같은 병렬 빠른 수행 경로의 반복적인 적용.

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

이 챕터에서는 데이터 파티셔닝이 간단한 선형적으로 확장 가능한 병렬 프로그램을 설계하는데 사용될 수 있는지 알아봤습니다. Section 6.3.4 에서는 데이터 복사 가능성에서 힌트를 얻었는데, 이는 Section 9.5 에서 커다란 효과를 가져올 것입니다.

파티셔닝과 복사본 사용을 적용하는 주요 목표는 선형적인 속도 향상을 얻기 위한 것으로, 달리 말하자면 CPU 나 쓰레드의 수가 늘어남에 따라 전체적으로 필요한 일의 양이 크게 늘어나지는 않음을 보장하기 위한 것입니다. 파티셔닝과 복사본 사용을 통해 해결될 수 있어서 선형적인 속도 향상이 가능한 문제들은 당혹스럽게 병렬적입니다. 하지만 이보다 더 잘할 수는 없을까요?

이 질문에 답을 하기 위해, 미궁과 미로의 해결책을 생각해 보도록 합시다. 물론, 미궁과 미로는 수천년 동안 매력적인 것이었으며 [Wik12], 따라서 그것들이 바이오 컴퓨터 [Ada11], GPGPU [Eri08], 심지어는 분리된 하드웨어 [KFC11] 등의 컴퓨터들을 사용해서 만들어지고 해결되었음은 별로 놀라운 일도 아닙니다. 미로의 병렬적 해결책은 대학 수업에서의 과제 프로젝트 [ETH11, Uni10]로도 사용되었고, 병렬 프로그래밍 프레임워크의 이점을 보이기 위한 매개물 [Fos10]로도 사용되었습니다.

흔한 조언은 병렬 일거리-대기열 알고리즘(PWQ: Parallel work-queue algorithm) [ETH11, Fos10]을 사용하라는 것입니다. 이 섹션은 무작위적으로 생성된 정사각형의 미로를 해결하는 모든 경우에 대해 순차적 알고리즘(SEQ)과 대안적인 병렬 알고리즘에 대해 PWQ를 비교하는 것으로 이 조언을 평가해 보겠습니다. Section 6.5.1 에서는 PWQ를 이야기하고, Section 6.5.2에서 대안적인 병렬 알고리즘을 설명하며, Section 6.5.3에서는 그것의 문제 있는 성능에 대해 이야기 한 후, Section 6.5.4에서 앞의 대안적 병렬 알고리즘으로부터 향상된 순차적 알고리즘을 소개하며, Section 6.5.5에서 성능을 비교해 보고, 마지막으로 Section 6.5.6에서 미래의 방향을 알아보고 결론을 내려봅니다.

```

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

PWQ는 Figure 6.33(maze_seq.c)에 있는 SEQ에 기반합니다. 미로는 셀들의 2D 배열과 ->visited로 이름 붙여진 선형적 배열 기반 일거리 대기열로 나타내어집니다.

Line 7에서 첫번째 셀에 들어가고, line 8-21에 있는 루프의 매 반복에서 하나의 셀에 의해 향해지는 통로를 횡단합니다. Line 9-13의 루프에서는 ->visited[] 배열을 방문되지 않은 이웃을 가지고 방문된 셀을 위해 스캔하고, line 14-19의 루프에서는 그 이웃을 통해 향해지는 작은 미로를 횡단합니다. Line 20에서는 밖의 루프에 의해 통과될 다음 경로를 위해 초기화를 합니다.

maze_try_visit_cell()의 슈도코드가 Figure 6.34 (maze.c)의 line 1-12에 나타나 있습니다. Line 4에서 셀 c와 n이 근처에 있고 연결되어 있는지 체크해 보고, line 5에서는 셀 n이 아직 방문되지 않았는지 확인해 봅니다. celladdr() 함수는 지목된 셀의 주소를 리턴합니다. 두 체크 중 하나라도 실패하면, line 6에서 실패했음을 리턴합니다. Line 7에서는 다음 셀을 알리고, line 8에서 이 셀을 ->visited[] 배열의 다음 슬롯에 기록해 두고, line 9에서 이 슬롯이 이제 채워졌음을 알리며, line 10에서 이 셀을 방문되었음으로 마크하고 미로의 시작점으로부터의 거리를 기록해둡니다. Line 11은 이제 성공했음을 리턴합니다.

maze_find_any_next_cell()의 슈도코드가 Figure 6.34 (maze.c)의 line 14-28에 있습니다. Line 17에서는 현재 셀의 거리 더하기 1을 얻어오고, 라인 19, 21, 23, 25에서는 각 방향의 해당 셀들을 체크하고, line 20, 22, 24, 26에서는 연관된 셀이 다음 셀 후보라면 true를 리턴합니다. prevcol(), nextcol(), prevrow(), 그리고 nextrow()는 각각 배열 인덱스

```

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

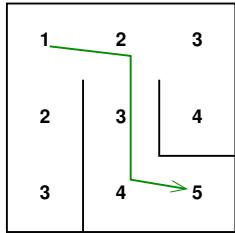


Figure 6.35: Cell-Number Solution Tracking

변환 작업을 수행합니다. 어떤 셀도 후보가 아니라면, line 27에서 `false`를 리턴합니다.

해당 경로는 Figure 6.35에 보여지는 것처럼, 미로에 시작점으로부터의 셀들의 수를 세는 것으로 기록되는데, 시작 셀은 좌상단에 위치해 있고 끝의 셀은 우하단에 위치해 있습니다. 끝 셀로부터 시작해서 연속적으로 줄어드는 셀 숫자들을 따라가는 것으로 해결 경로를 따라 횡단할 수 있습니다.

병렬 작업 대기열 처리자 (work-queue solver)는 Figures 6.33 와 6.34에 보여진 알고리즘의 직선적인 병렬화입니다. Figure 6.33의 line 10은 `fetch-and-add`를 사용해야만 하고 지역 변수인 `vi`는 여러 쓰레드를 사이에 공유되어야만 합니다. Figure 6.34의 Line 5와 10은 CAS 루프로 구성되어야만 하는데, 이 때 CAS의 실패는 미로 루프를 의미하게 됩니다. 이 그림의 Line 8-9는 셀들을 `->visited[]` 배열에 동시적으로 기록하려 시도하는 것을 처리하기 위해 `fetch-and-add`를 사용해

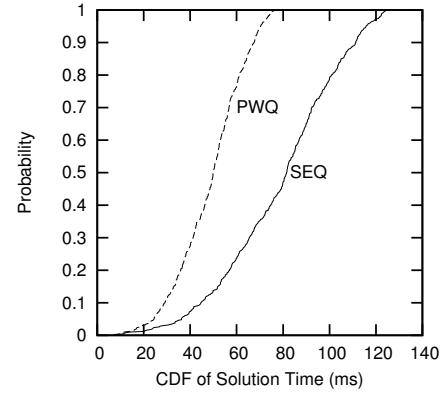


Figure 6.36: CDF of Solution Times For SEQ and PWQ

야만 합니다.

이 접근법은 Figure 6.36에서 볼 수 있듯이 2.53GHz의 속도로 동작하는 dual-CPU Lenovo™ W500에서 상당한 속도 향상을 보여주는데, 두 알고리즘의 해결책을 얻는데 걸리는 시간의 누적 분포 함수들 (CDF)을 500개의 다른 500행 500열의 정사각으로 무작위적으로 만들어진 미로들에 대해 측정되었습니다. 이 CDF들을 x 축에 투영해서 만들어지는 실질적인 겹쳐진 모습은 Section 6.5.3에서 다루어질 것입니다.

상당히 흥미롭게도, 순차적인 해결책의 경로 탐색은 병렬 알고리즘에서도 바뀌지 않았습니다. 하지만, 이는 병렬 알고리즘의 상당한 약점을 드러냈습니다: 어떤 주어진 시간 동안 최대 하나의 쓰레드만이 해결책 경로로의 진행을 만들 수 있습니다. 이 약점은 다음 섹션에서 다루어집니다.

6.5.2 Alternative Parallel Maze Solver

유용한 미로 풀기 방법들은 종종 양 끝에서 시작할 것을 주장했고, 이런 조언은 자동화된 미로 해법 [Uni10]의 맥락에서 최근들어 더 반복되었습니다. 이 조언은 파티셔닝과 같은 것으로, 파티셔닝은 병렬 프로그래밍의 맥락에서 운영체제 커널에 대해서도 [BK85, Inm85] 어플리케이션에 대해서도 [Pat10] 강력한 병렬화 전략이 되어왔습니다. 이 섹션에서는 이 전략을 적용해 보는데, 해결책 경로의 양 끝단에서 시작하는 두개의 자식 쓰레드를 사용하고, 성능과 확장성에 대해 짧게 결과를 알아봅니다.

Figure 6.37 (`maze_part.c`)에 보여진 파티션을 사용한 병렬 알고리즘 (PART)은 SEQ와 비슷하지만 몇 가지 중요한 차이점이 있습니다. 첫번째로, 각 자식 쓰레드는 자신의 `visited` 배열을 가지고 있는데, line 1에서 보듯이 부모로부터 전달받은 것으로, 모두 `[-1, -1]`로 초기화되어 있어야만 합니다. Line 7에서는 이 배

```

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

열로의 포인터를 per-thread 변수 myvisited에 저장해서 도우미 함수들로부터의 접근을 가능하게 하고, 비슷하게 지역적으로 방문한 곳의 인덱스로의 포인터를 저장합니다. 두번째로, 부모는 각 자식의 입장에서 첫 번째 셀을 방문하는데, 이는 자식들이 line 8에서 얻어옵니다. 세번째로, 미로는 한 자식이 다른 자식에 의해 방문되었던 셀을 발견하면 그 즉시 풀이됩니다. maze_try_visit_cell()이 이를 발견하게 되면, 이 함수는 해당 미로 구조체의 ->done 필드에 값을 넣습니다. 넷째로, 따라서 각 자식은 line 13, 18, 23에 보여진 것처럼 주기적으로 ->done 필드를 체크해야만 합니다. ACCESS_ONCE() 기능은 다음의 연속적인 로드들을 합치거나 값을 다시 읽어올 수도 있는 어떤 컴파일러 최적화도 무력화 되도록 해야만 합니다. 이를 위해선 C++1x volatile relaxed load로도 충분합니다 [Bec11]. 마지막으로, maze_find_and_next_cell() 함수는 한 셀을 방문된 것으로 마크하기 위해 compare-and-swap을 사용해야만 합니다만, 쓰레드 생성과 합치기에 의해 만들어지는 순서 이후에 어떤 순서 제약도 필요로 되지 않습니다.

maze_find_any_next_cell()의 슈도코드는 Figure 6.34에 보여진 것과 동일합니다만, maze_try_visit_cell()의 슈도코드는 좀 다른데, Figure 6.38에 보여져 있습니다. Line 8-9는 해당 셀들이 연결되어 있는지 체크하고, 그렇지 않다면 failure를 리턴합니다. Line 11-18의 루프에서는 새 셀을 방문된 것으로 마크합니다. Line 13은 해당 셀이 이미 방문된 적 있는지 체크하고, 이 경우엔 line 16에서 failure를 리턴합니다만, line 14에서 다른 쓰레드와 마주친 것인지 체크한

```

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

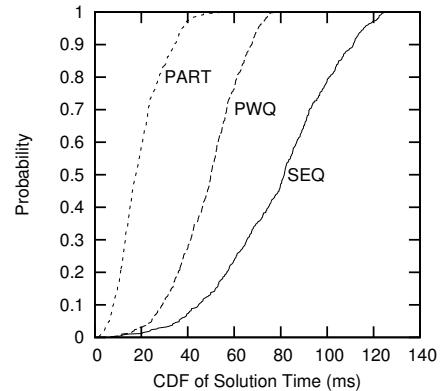


Figure 6.39: CDF of Solution Times For SEQ, PWQ, and PART

이후로, 마주친 경우라면 line 15에서 해법이 찾아졌음을 알립니다. Line 19에서는 새 셀에 업데이트를 하고, line 20과 21에서 이 쓰레드의 방문된 곳 배열을 업데이트 하며, line 22에서 success를 리턴합니다.

성능 테스트는 Figure 6.39에 보여진 것과 같이 심각한 변칙적 결과를 보였습니다. PART의 평균 해법 탐색 시간(17밀리세컨드)은 SEQ의 그것(79밀리세컨드)보다 두개의 쓰레드만 사용함에도 네배 넘게 빨랐습니다. 다음 섹션에서 이 결과를 분석해 봅니다.

6.5.3 Performance Comparison I

이례적 성능 결과에 대한 첫번째 대응은 버그 유무를 체크하는 것입니다. 이 알고리즘들은 모두 실제로 올바른 해결책을 찾아내고 있습니다만, Figure 6.39에 나타난

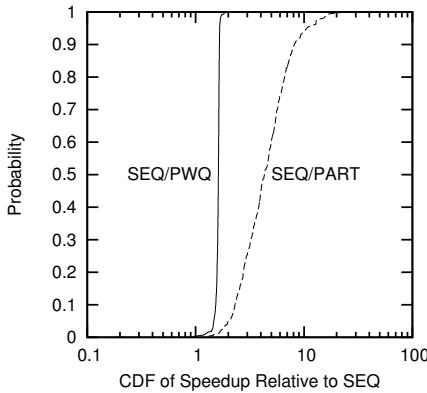


Figure 6.40: CDF of SEQ/PWQ and SEQ/PART Solution Time Ratios

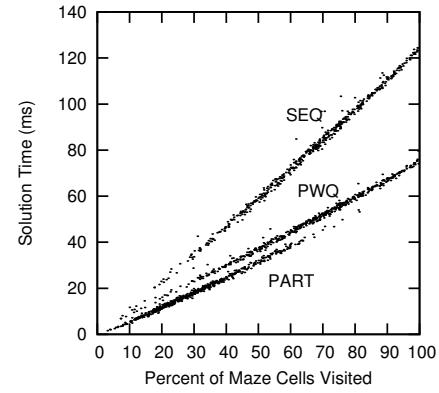


Figure 6.42: Correlation Between Visit Percentage and Solution Time

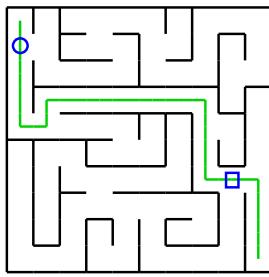


Figure 6.41: Reason for Small Visit Percentages

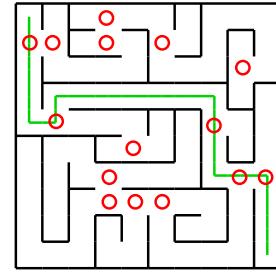


Figure 6.43: PWQ Potential Contention Points

CDF 그림은 독립적인 데이터를 가정하고 있습니다. 이건 올바른 경우가 아닙니다: 이 성능 테스트들은 무작위적으로 미로를 생성하고, 그 미로에 대해 모든 알고리즘들을 돌려보고 있습니다. 따라서 각각의 생성된 미로에 대해 해결책을 찾는데 걸린 시간의 비율의 CDF를 그려보는게 좀 더 말이 될텐데, Figure 6.40에 그 그림이 그려져 있으며, 여기선 CDF들의 중복이 훨씬 줄어들었습니다. 쓰레드 두개로 이루어진 40배의 성능 향상은 설명이 필요합니다. 무엇보다, 이것은 단지 파티션으로 쪼갤 수 있음이 쓰레드를 추가하는 것으로 인해 전체 연산 비용의 증가로 이어지지 않음을 의미하는 당혹스러울 정도의 병렬성도 아닙니다. 그 대신, 이것은 굴욕적인 병렬성입니다: 쓰레드를 추가하는 것이 전체 연산 비용을 상당히 줄여줘서 커다란 알고리즘적인 선형적인 성능향상을 초월하는 결과를 이끌어낸 것입니다.

더 나아가서 들여다본 결과 PART는 가끔 미로의 셀들 중 2% 미만만을 방문했는데, SEQ와 PWQ는 9% 미만을 방문한 적이 없었습니다. 이런 차이점에 대한 이유는 Figure 6.41에 보여져 있습니다. 만약 좌상단부터 시작해서 해결책을 찾는 쓰레드가 원에 도달하면 다른 쓰레드는 미로의 우상단에 도달할 수 없습니다. 비

슷하게, 만약 다른 쓰레드가 네모에 도달하면, 첫번째 쓰레드는 미로의 좌하단에 도착할 수 없습니다. 따라서, PART는 해법이 아닌 셀들로 이루어진 경로 중 더 작은 부분만을 방문하게 될 것입니다. 한마디로, 선형을 초월하는 속도 향상은 쓰레드들이 서로의 길을 만들어 주기 때문입니다. 이는 일하는 쓰레드들이 쓰레드들 각자의 길에서 다른 쓰레드들을 벗어나게 하기 위해 노력해왔던 수십년의 병렬 프로그래밍의 경험에 상당히 반대되는 이야기입니다.

Figure 6.42는 세개의 모든 방법들에 대해 방문된 셀들의 수와 해결책 탐색에 걸리는 시간 사이의 강한 상관관계를 확실히 보여주고 있습니다. PART의 그림의 경사도는 SEQ의 그것에 비해 작은데, 이것이 PART의 쓰레드들이 SEQ의 단일 쓰레드에 비해 미로의 주어진 부분을 더 빠르게 방문할 것임을 이야기 합니다. PART의 그림은 또한 작은 방문 퍼센티지에 몰려 있는데, 이는 PART가 더 적은 일을 하게 되며, 따라서 관측된 굴욕적인 병렬성을 보이게 되는 것입니다.

PWQ에 의해 방문되는 셀들로 이루어진 부분들은 SEQ의 그것과 유사합니다. 또한, PWQ의 해결책 탐색에 걸리는 시간은 동일한 방문 부분들에도 불구하고

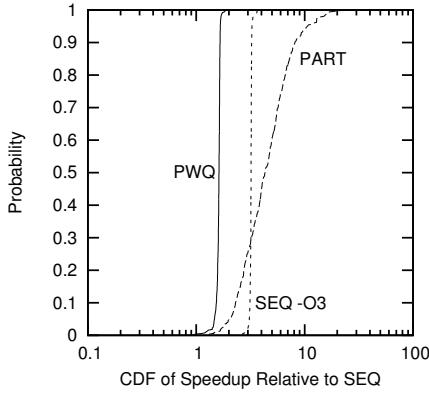


Figure 6.44: Effect of Compiler Optimization (-O3)

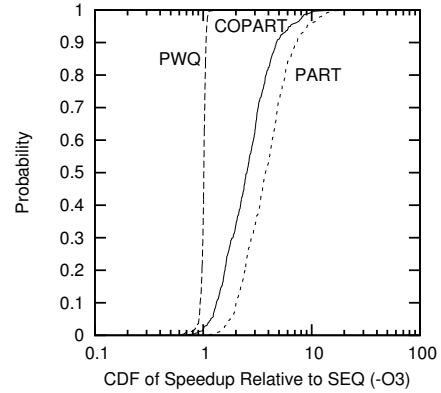


Figure 6.45: Partitioned Coroutines

PART 의 그것에 비해 훨씬 큽니다. 이에 대한 이유가 Figure 6.43 에 그려져 있는데, 이 그림에는 두개 이상의 이웃을 가지고 있는 셀마다 붉은 원을 그려두었습니다. 그러한 각각의 셀은 PWQ 에서 경쟁 상황을 초래할 수 있는데, 한 쓰레드는 들어갈 수 있지만 두 쓰레드들이 나갈 수 있기 때문에, 이 챕터의 앞부분에서 설명했듯이 성능을 악화시킬 수 있기 때문입니다. 반면에 PART 는 그런 경쟁상황을 한번만 일으키는데, 해법이 찾아졌을 때입니다. 물론, SEQ 는 경쟁상황은 일으키지 않습니다.

PART 의 성능 향상이 인상적이긴 하지만, 순차적인 최적화를 게을리 하지는 않아야 합니다. Figure 6.44 는 SEQ 가 -O3 옵션을 가지고 컴파일 되었을 때에는 최적화 되지 않은 PWQ 보다 두배 정도나 빠르고, 최적화 되지 않은 PART 의 성능에 근접합니다. 세개의 알고리즘을 모두 -O3 옵션을 주고 컴파일한 결과는 (더 빠르긴 하지만) Figure 6.40 에 보여진 그것들과 비슷한 양상을 보입니다만, PWQ 가 SEQ 에 비해 속도향상을 거의 보이지 않는다는 예외를 갖는데, 이는 Amdahl 의 법칙 [Amd67] 에 의함입니다. 하지만, 최적화 되지 않은 SEQ 에 비해 두배의 속도를 갖는게 목표라면, 컴파일러 최적화는 상당히 매력적인 방법이라 할 수 있겠습니다.

캐시 정렬과 패딩이 종종 거짓 공유 (false sharing) 을 줄여서 성능을 개선시키곤 합니다. 하지만, 이 미로 해법 알고리즘들에 있어서는, 미로 셀 배열에 정렬과 패딩을 적용하는 것은 1000x1000 미로에 대해 42% 까지 성능을 떨어뜨립니다. 캐시 로컬리티는 거짓 공유를 없애는 것보다 더 중요한데, 특히나 큰 미로에선 더욱 그러합니다. 작은 20x20 이나 50x50 미로에서라면, 정렬과 패딩이 PART 에 대해 성능을 40% 까지 향상시킬 수 있습니다만 이런 작은 크기의 미로에 대해서는 PART 가 쓰레드를 생성하고 소멸시키는데 드는 오버헤드를 상쇄시키기에 충분한 시간을 갖지 못하기에 SEQ 가 더 좋은 성능을 보입니다.

정리하자면, 파티셔닝을 사용한 병렬 미로 해결책은 알고리즘적으로 선형을 초월한 속도향상의 하나의 재미있는 예입니다. 만약 “알고리즘적으로 선형을 초월한 속도 향상” 이 인지 부조화를 일으킨다면, 다음 섹션으로 넘어가 보시기 바랍니다.

6.5.4 Alternative Sequential Maze Solver

알고리즘적으로 선형을 초월한 속도 향상의 존재는 코루틴 (co-routine) 을 통한 병렬성 모의실험을 제시하는데, 예를 들어, Figure 6.37 의 do-while 루프의 각 패스에서 수동으로 컨텍스트 스위칭을 해보는 겁니다. 이 컨텍스트 스위칭은 직접적인데 이 컨텍스트는 변수들 c 와 v_i 로만 구성되어 있기 때문입니다. 이 효과를 낼 수 있는 많은 방법들 중, 이 방법이 컨텍스트 스위칭 오버 헤드와 방문 퍼센티지 사이의 좋은 트레이드오프입니다. Figure 6.45 에서 볼 수 있듯이 이 코루틴 알고리즘 (COPART) 은 상당히 효과적으로, 한 쓰레드에서의 성능이 두 쓰레드를 사용한 PART 의 30% 입니다 (maze_2seq.c).

6.5.5 Performance Comparison II

Figures 6.46 와 6.47 는 미로의 크기의 변화에 따른 효과를 두개의 쓰레드로 동작하는 PWQ 와 PART 를 SEQ 또는 COPART 와 각각 비교해 90% 의 정확성에 끼어 막대와 함께 보여주고 있습니다. PART 는 100행 100열 이상 크기의 미로들에서 SEQ 에 비해 선형성을 초월한 확장성을 보이고 COPART 에 비해서는 적당한 확장성을 보입니다. 에너지 소모가 높은 주파수에서는 대략 주파수의 제곱 정도로 증가한다는 가정 [Mud00] 에 기반해서 보면 두 쓰레드를 사용해서 1.4 배의 확장성을 갖는 것은 단일 쓰레드가 같은 해법 탐색 시간을 필요로 할 때 소모하는 에너지와 동일하므로 PART 는 COPART

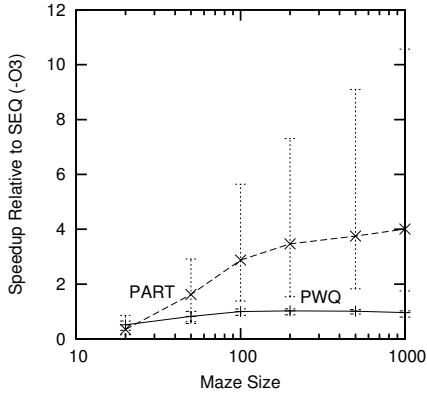


Figure 6.46: Varying Maze Size vs. SEQ

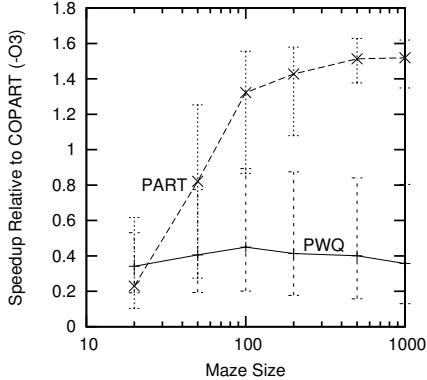


Figure 6.47: Varying Maze Size vs. COPART

에 비교했을 때 대략 200행 200열 크기 미로에서 이론적인 에너지 효율성 손익분기를 넘어섭니다. 반면, PWQ는 SEQ에 대해서도 COPART에 대해서도 빈약한 확장성을 보이는데, 이는 최적화가 적용되었을 때 이야기입니다: Figure 6.46과 6.47는 -O3 옵션을 사용해 만들어졌습니다.

Figure 6.48는 PWQ와 PART의 성능을 COPART에 비교해서 보여주고 있습니다. 두개가 넘는 쓰레드들을 사용해 동작하는 PART의 경우, 추가된 쓰레드들은 미로의 시작점과 끝점 사이를 대각선으로 동일한 거리로 나뉘어진 위치에서부터 경로 탐색을 시작합니다. 두개가 넘는 쓰레드들을 사용해 동작하는 PART의 이론 종료를 알아채기 위해서는 간략화된 링크 상태 라우팅 [BG87]이 사용되었습니다 (해법은 한 쓰레드가 시작점과 끝점과 연결되면 찾아진 것으로 표시됩니다). PWQ는 상당히 나쁜 성능 결과를 보입니다만, PART는 두개 쓰레드에서, 그리고 다섯개 쓰레드에서 한번 더 손익분기에 도달하는데 다섯개 쓰레드를 넘어서고 부터는 적당한 속도 향상을 이루어냅니다. 이론적인 예

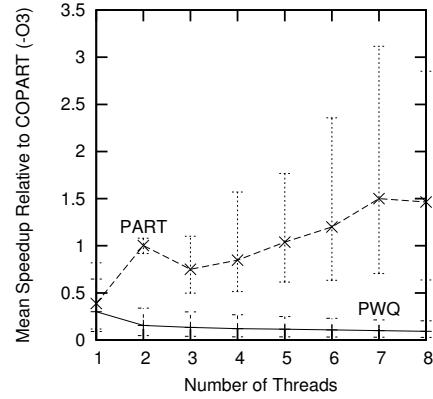


Figure 6.48: Mean Speedup vs. Number of Threads, 1000x1000 Maze

너지 효율성 손익분기는 7개와 8개 쓰레드들에 대해서는 90% 신뢰구간 안에 있습니다. 두개 쓰레드에서 성능이 뾰족하게 솟아오르는 이유는 (1) 두개 쓰레드의 경우의 덜 복잡한 해법 탐색 종료 파악 과정과 (2) 세 번째와 그 뒤의 쓰레드들이 유용한 진전을 만들 확률이 좀 더 낮은 편이라는 사실입니다: 오직 처음의 두 쓰레드만이 해법의 경로에 직결됨이 보장됩니다. Figure 6.47에 비해 비교적 실망스러운 이 성능 결과는 2.66GHz로 동작하는 더 크고 오래된 Xeon®에 사용되는 덜 밀접하게 융합된 하드웨어의 탓입니다.

6.5.6 Future Directions and Conclusions

너무 많은 해야 할 일들이 남아있습니다. 첫째로, 이 섹션은 사람이 미로 해법 탐색 사용하는 방법 가운데 하나의 방법만을 적용해 보았습니다. 이외의 방법으로는 미로의 일부를 제거하기 위해 벽을 따라가는 방법과 이전에 탐색한 경로의 위치에 기반해서 내부 시작 지점을 고르는 방법 등이 있습니다. 둘째로, 시작지점과 끝지점의 다른 선택은 다른 알고리즘에 좀 더 효과적일 수 있습니다. 셋째로, PART 알고리즘의 첫번째 적용했던 방법인 두개 쓰레드 사용이 직선적이긴 하지만, 다른 여분의 쓰레드들을 사용하기 위한 방법들이 여럿 있습니다. 최적의 쓰레드 사용은 시작지점과 끝지점에도 의존적일 것입니다. 넷째로, 해결이 불가능한 미로들과 순환적인 미로들에 대한 연구는 재미있는 결과를 내놓을 수 있을 것입니다. 다섯째로, 가벼운 C++11 어토믹 오퍼레이션들은 성능을 개선시킬 수도 있습니다. 여섯째로, 3차원 미로(또는 그보다도 높은 차원의 미로들)의 속도 향상을 비교해 보는 것도 재미있을 것입니다. 마지막으로, 미로의 경우, 굴욕적 병렬성은 코루틴들을 사용한 보다 효과적인 순차적 구현을 의미했습니다. 굴욕적 병렬성 알고리즘은 항상 더 효과적인 순차적 구현을 이끌

게 될까요, 아니면 코루틴 컨텍스트 스위치 오버헤드가 속도향상을 압도해버리는 고유의 굴욕적 병렬성 알고리즘이 존재할까요?

이 섹션은 미로 해법 탐색 알고리즘들의 병렬화를 선보이고 분석해 보았습니다. 일반적인 일거리 대기열 기반의 알고리즘은 컴파일러 최적화가 꺼져있을 때에만 잘 동작했는데, 이는 기준에 고수준의 오버헤드가 많은 언어를 사용해 얻어졌던 결과들 중 일부는 진보된 최적화에 의해서 무효화 될 수도 있음을 의미합니다.

이 섹션은 병렬화를 적용하는 것을 순차적 알고리즘의 파생물로 생각하기보다는 최적화를 위한 첫번째 선택지로 생각하는 것이 개선된 순차적 알고리즘을 위한 길의 기반을 닦음의 확인한 예 하나를 선보였습니다. 고수준 설계 레벨에서의 병렬성의 응용은 결실 있는 연구가 될 가능성이 큽니다. 이 섹션은 미로 탈출 경로를 탐색하는 문제를 느슨하게 확장성 있는 경우부터 굴욕적으로 병렬적인 경우까지 그리고 그 반대로 풀어보았습니다. 이 경험이 병렬성에 기반한 작업을 무식하게 별로 최적이지 않은 경우가 많은, 이미 존재하는 프로그램에 개조의 형태로 적용되는 기존 성능 결과에 기반한 작은 최적화와 같은 것보다는 설계시점에서의 전체 어플리케이션을 위한 최적화 기법의 첫번째 선택지로 여겨지도록 하는데 동기를 부여하길 바랍니다.

6.6 Partitioning, Parallelism, and Optimization

가장 중요한 건, 이 챕터가 설계 단계에서부터 병렬성을 적용하는 것이 훌륭한 결과를 내놓기는 함에도 불구하고, 이 마지막 섹션에서는 이것만으로 충분하지는 않다는 걸 이야기 합니다. 미로 해법 찾기와 같은 탐색 문제들을 위해, 이 섹션은 병렬 설계보다도 검색 전략이 더 중요함을 보였습니다. 그래요, 이 특별한 미로의 타입에 대해서만큼은 현명하게 병렬성을 적용하는 것이 훌륭한 검색 전략이었습니다만, 이런 부류의 행운은 검색 전략 그 자체에 대한 명확한 초점으로 충분하지 않습니다.

Section 2.2에서 이전에 이야기 했듯이, 병렬성은 많은 최적화 방법 중 하나의 잠재적인 최적화 수단일 뿐입니다. 성공적인 설계는 가장 중요한 최적화에 집중을 해야 합니다. 저는 다르게 주장하고 싶은 마음이 간절하긴 하지만, 그 최적화는 병렬성일 수도, 병렬성이 아닐 수도 있습니다.

하지만, 병렬성이 올바른 최적화인 많은 경우들을 위해서, 다음 섹션에서는 동기화 작업을 대부분의 경우 처리하는 도구인 락킹에 대해 다룹니다.

Chapter 7

Locking

최근의 동시성 관련한 연구에서, 악당 역할을 맡는 경우 많은 경우 락킹이 되곤 했습니다. 많은 논문과 발표들에서, 락킹은 데드락, 컨보잉 (convoying), 스타베이션, 비공정성, 데이터 레이스, 그리고 그외의 모든 동시성에서의 죄악들을 일으킨다고 비난받아왔습니다. 그런데 흥미롭게도, 상품 품질의 공유 메모리 병렬 소프트웨어에서 대부분의 일을 처리하는 역할은, 짐작하시겠지만, 락킹에 의해 맡아졌습니다. 이 챕터는 이런 악당과 영웅 사이의 이분법에 대해 Figure 7.1 와 7.2 에 그려진 것처럼 그럴싸하게 들여다 보겠습니다.

이런 지킬과 하이드 같은 이분법 뒤에는 많은 이유가 있습니다:

1. 많은 락킹의 죄악들은 대부분의 경우 잘 동작하는 실용적인 설계상의 해결책이 존재하는데, 예를 들면:
 - (a) 데드락을 막기 위한 락 계층의 사용.
 - (b) 리눅스 커널의 lockdep 기능 [Cor06a] 과 같은 데드락 탐지 도구들.
 - (c) Chapter 10에서 다루어지게 되는 배열, 해시 테이블, 그리고 래디스 트리와 같은, 락킹에 친화적인 데이터 구조.
2. 락킹의 죄악들 중 일부는 높은 수준의 경쟁 상황에서만 발생하는데, 그런 수준은 잘못 설계된 프로그램들에서만 도달 가능한 수준입니다.
3. 락킹의 죄악들 중 일부는 락킹과 함께 사용되는 다른 동기화 메커니즘의 사용으로 막아질 수도 있습니다. 이런 다른 메커니즘들에는 통계적 카운터 (Chapter 5 을 참고하세요), 레퍼런스 카운터들 (Section 9.2 을 참고하세요), 해저드 포인터들 (Section 9.3 을 참고하세요), 순서 락킹 읽기들 (Section 9.4 을 참고하세요), RCU (Section 9.5 을 참고하세요), 그리고 간단한 non-blocking 데이터 구조체들이 있습니다 (Section 14.3 을 참고하세요).

Locking is the worst general-purpose synchronization mechanism except for all those other mechanisms that have been tried from time to time.

*With apologies to the memory of Winston Churchill
and to whoever he was quoting*

4. 아주 최근까지는, 거의 모든 커다란 공유 메모리 병렬 프로그램들은 비밀리에 개발되었고, 따라서 대부분의 연구자들은 이런 실용적인 해결책에 대해 배우기가 어려웠습니다.

5. 락킹은 일부 소프트웨어 제품들에서는 매우 잘 동작하지만 다른 것들에서는 매우 안좋게 동작합니다. 락킹이 잘 동작하는 소프트웨어를 작업했던 개발자들은 락킹이 나쁘게 동작하는 소프트웨어를 작업했던 사람들에 비해 락킹에 대해 훨씬 더 긍정적인 의견을 가질 수 있는데 이는 Section 7.5에서 다루어질 것입니다.

6. 모든 좋은 이야기는 악당이 필요하고, 락킹은 연구 논문의 희생양으로써의 길고 명예로운 역사를 맡아왔습니다.

Quick Quiz 7.1: 희생양 역할을 한게 어떻게 명예로운 것으로 여겨질 수가 있나요???

이 챕터는 락킹의 더 심각한 죄악들을 막는 여러 방법들에 대해 간략히 살펴봅니다.

7.1 Staying Alive

락킹이 데드락과 스타베이션의 피의자 혐의를 받는다는 사실을 생각해 보면, 공유 메모리 병렬 개발자들의 중요한 관심사는 그저 살아있는 상태를 유지하는 것입니다. 따라서 다음의 섹션들에서는 데드락, 라이브락, 스타베이션, 비공정성, 그리고 비효율성에 대해 다뤄봅니다.

7.1.1 Deadlock

데드락은 한 무리의 쓰레드들이 각각 최소한 하나의 락을 잡은 상태로 같은 무리의 다른 멤버가 이미 잡고 있는 락을 놓기를 기다리고 있을 때 발생합니다.

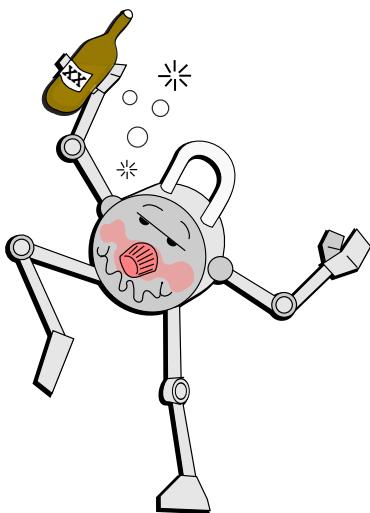


Figure 7.1: Locking: Villain or Slob?

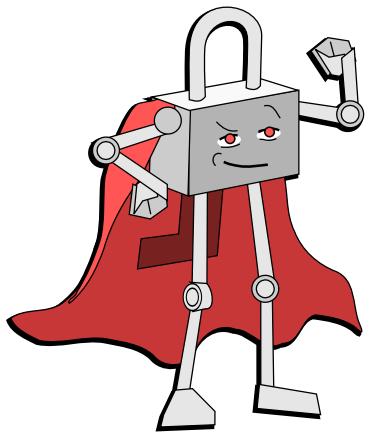


Figure 7.2: Locking: Workhorse or Hero?

어떤 외부의 간섭 같은 것이 없이는 데드락은 영원히 존재합니다. 어떤 쓰레드도 자신이 기다리고 있는 락을 쥐고 있는 쓰레드가 그 락을 놓아주기 전까지는 얻을 수 없습니다만, 그 락을 쥐고 있는 쓰레드는 그 락을 얻기 위해 기다리고 있는 쓰레드가 쥐고 있는 락을 얻기 전까지는 자신의 락을 놓을 수 없습니다.

데드락 시나리오를 쓰레드와 락을 노드로 표현한 방향성이 존재하는 그래프로 표현해 볼 수 있는데, Figure 7.3에 그러한 표현이 그려져 있습니다. 하나의 락에서 하나의 쓰레드로의 화살표는 해당 쓰레드가 해당 락을 쥐고 있음을 의미하는데, 예를 들어 Thread B는 Lock 2와 4를 쥐고 있습니다. 한 쓰레드에서 한 락으로

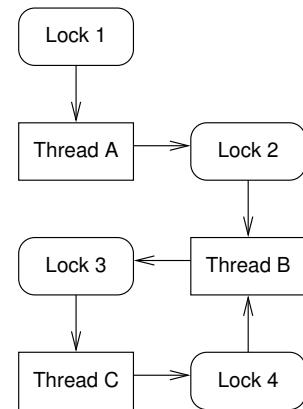


Figure 7.3: Deadlock Cycle

의 화살표는 해당 쓰레드가 해당 락을 얻기 위해 기다리고 있음을 의미하는데, 예를 들어 Thread B는 Lock 3을 얻기 위해 기다리고 있습니다.

데드락 시나리오는 항상 최소한 하나 이상의 데드락 사이클을 갖게 됩니다. Figure 7.3에서는 Thread B, Lock 3, Thread C, Lock 4, 그리고 다시 Thread B로 이어지는 사이클이 존재합니다.

Quick Quiz 7.2: 하지만 데드락의 정의는 단지 각 쓰레드가 최소 하나 이상의 락을 잡고 일부 쓰레드에 의해 잡혀 있는 또 다른 락을 기다리고 있는 것이잖아요. 어떻게 거기에 사이클이 존재하는지 알 수 있죠? ■

존재하는 데드락 상태로부터 시스템을 복구하는 데이터베이스 시스템과 같은 소프트웨어 환경도 존재합니다만, 이런 방법은 한 쓰레드가 죽거나 락이 강제로 다른 쓰레드로부터 강탈되는 동작을 필요로 합니다. 이런 죽기고 강탈하는 방법은 트랜잭션들에 대해서는 적당할 수 있겠지만, 커널과 어플리케이션 수준에서의 락킹의 사용에는 많은 경우 문제가 될 수 있습니다: 그로 인해 발생하는 부분적으로만 업데이트된 구조체를 처리하는 것은 상당히 복잡하고 위험하고 오류를 만들기 쉽습니다.

따라서 커널과 어플리케이션들은 데드락으로부터 회복을 하기보다는 데드락을 회피하는 방법을 사용하려 합니다. 여러개의 데드락 회피 전력이 존재하는데, 계층적 락킹 (Section 7.1.1.1), 지역적 계층적 락킹 (Section 7.1.1.2), 레이어를 사용한 계층적 락킹 (Section 7.1.1.3), 락들로의 포인터들을 갖는 API들을 다루는 전략들 (Section 7.1.1.4), 조건적 락킹 (Section 7.1.1.5), 필요한 락들 전체를 처음에 획득하기 (Section 7.1.1.6), 한번에 하나의 락만 사용하기 설계 (Section 7.1.1.7), 그리고 시그널/인터럽트 핸들러 전략이 포함됩니다. (Section 7.1.1.8). 모든 상황에 완벽하게 동작하는 데드락 회피 전략이 존재하기는 합니다만, 그 중에서

도 데드락 회피 도구를 잘 선택하는 방법이 있습니다.

7.1.1.1 Locking Hierarchies

락킹 계층은 락들을 순서세워서 순서에 상관없이 락들을 획득하는 행위를 막습니다. Figure 7.3 의 상황에서는 한 쓰레드가 획득하려 하는 락과 같거나 더 큰 숫자의 락을 이미 쥐고 있다면 해당 그래프에서 사라지도록, 락들을 숫자에 기반한 규칙으로 순서세울 수 있을겁니다. Thread B 는 이 계층을 위반한 셈인데, Lock 4 를쥔 상태에서 Lock 3 를 획득하려 하고 있기 때문이고, 이로인해 데드락이 발생했습니다.

다시 말하지만, 락킹 계층을 적용하기 위해서는 락들의 순서를 잡아주고 순서에 상관없이 락을 획득하는 행위를 막아야 합니다. 커다란 프로그램에서는 락킹 계층을 강제하기 위해 도구를 사용하는 것이 현명할겁니다 [Cor06a].

7.1.1.2 Local Locking Hierarchies

하지만, 전역적이라는 락킹 계층의 본연적 특성은 해당 방법을 라이브러리 함수에 적용하기 어렵게 합니다. 무엇보다도, 주어진 라이브러리 함수를 사용하는 프로그램은 아직 작성되지도 않았고, 따라서 어떻게 이 불쌍한 라이브러리 함수 구현자가 아직 작성되지 않은 프로그램의 락킹 계층에 충실할 수 있기를 바랄 수 있겠어요?

다행히도 일반적인 특수한 경우가 있는데 해당 라이브러리 함수가 호출자의 코드를 전혀 사용하지 않을 때입니다. 이 경우, 호출자의 락들은 라이브러리의 락들을 잡은 상태에서 잡혀지지 않고, 따라서 라이브러리와 호출자 둘 다의 락들로부터 생성되는 데드락 사이클은 존재할 수 없습니다.

Quick Quiz 7.3: 이 규칙에 대한 어떤 예외가 존재해서, 라이브러리 코드가 호출자의 어떤 함수도 실행하지 않는다는 조건 하에서도 라이브러리와 호출자 둘 다의 락을 포함하는 데드락 사이클이 실제로는 존재할 수도 있을 수 있나요? ■

하지만 라이브러리 함수가 실제로 호출자의 코드를 실행하는 경우를 생각해 봅시다. 예를 들어, `qsort()` 함수는 호출자가 제공하는 비교 함수를 실행합니다. `qsort()` 의 동시적인 구현은 아마도 락킹을 사용할텐데, 이는 아마도 그려지 않을 확률이 크긴 한 케이스지만 비교 함수가 락킹에도 관여되는 좀 복잡한 함수인 경우라면 데드락을 일으킬 수 있습니다. 이 라이브러리 함수는 어떻게 데드락을 막을 수 있을까요?

이 경우에의 항금률은 “알 수 없는 코드를 실행하기 전에 모든 락을 놓아버리기”입니다. 이 규칙을 따르기 위해선, 앞서 언급한 `qsort()` 함수는 비교 함수를 실행하기 전에 모든 락을 해제해야만 합니다.

Quick Quiz 7.4: 하지만 `qsort()` 가 비교 함수를 실

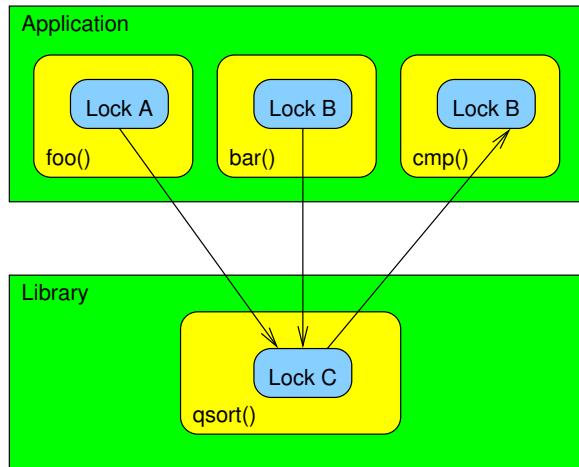


Figure 7.4: Without Local Locking Hierarchy for `qsort()`

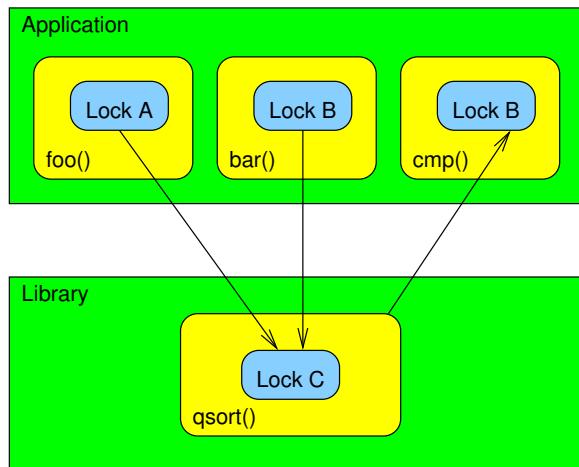


Figure 7.5: Local Locking Hierarchy for `qsort()`

행하기 전에 그 자신의 락들을 모두 해제해버리면, 다른 `qsort()` 쓰레드들과의 경주 문제를 어떻게 지킬 수 있나요? ■

지역적 락킹 계층화의 장점을 보기 위해, Figure 7.4 와 7.5 를 비교해 보시기 바랍니다. 두개의 그림에서, 어플리케이션의 함수들인 `foo()` 와 `bar()` 는 각각 `Lock A` 와 `Lock B` 를 잡은 상태로 `qsort()` 를 호출합니다. 이것은 `qsort()` 의 병렬화된 구현이기 때문에, 이 함수는 `Lock C` 를 잡습니다. 함수 `foo()` 는 `cmp()` 함수를 `qsort()` 에게 전달하고, `cmp()` 함수는 `Lock B` 를 잡습니다. 함수 `bar()` 는 간단한 정수 비교 함수 (그림에는 보이지 않습니다)를 `qsort()` 에게 전달하고, 이 간단한 함수는 어떤 락을 잡거나 하지 않습니다.

이제, `qsort()` 가 앞서 이야기한 모든 락을 놓기 황금률을 어기게 되어서 Figure 7.4에 보인 것처럼 `cmp()` 를 호출하는 동안 Lock C 를 잡고 있다면 데드락이 일어날 수 있습니다. 이를 확실히 보기 위해, 한 쓰레드가 `foo()` 를 호출했고 두번째 쓰레드가 그사이 동시에 `bar()` 를 호출하는 경우를 생각해 봅시다. 첫번째 쓰레드는 Lock A 를 잡고 두번째 쓰레드는 Lock B 를 잡게 됩니다. 만약 첫번째 쓰레드의 `qsort()` 함수 호출이 Lock C 를 잡게 된다면, 이제 `qsort()` 안에서 `cmp()` 함수를 호출했을 때 Lock B 를 잡는 것은 불가능해집니다. 하지만 첫번째 쓰레드가 Lock C 를 잡고 있어서 두번째 쓰레드의 `qsort()` 호출은 그걸 잡지 못하게 되고, 따라서 Lock B 를 놓을 수도 없게 되므로, 데드락을 초래하게 됩니다.

반면에, 만약 `qsort()` 가 비교 함수 (`qsort()` 의 관점에서 보기에는 알 수 없는 코드입니다) 를 호출하기 전에 Lock C 를 놓게 된다면, 데드락은 Figure 7.5에 보이는 것처럼 막아집니다.

만약 각 모듈이 알 수 없는 코드를 실행하기 전에 모든 락들을 내려놓게 된다면, 그리고 각 모듈이 개별적으로 데드락을 막고 있다면 데드락은 생기지 않게 됩니다. 따라서 이 규칙은 데드락 분석을 크게 간략화 시키고 모듈성을 크게 향상시킵니다.

7.1.1.3 Layered Locking Hierarchies

불행하게도, `qsort()` 가 자신의 모든 락들을 앞서 이야기한 비교 함수를 호출하기 전에 해제하는 것은 불가능할 수 있습니다. 이런 경우, 알 수 없는 코드를 실행하기 전에 모든 락들을 해제하는 지역적 락킹 계층을 사용할 수 없습니다. 하지만, 그 대신에 층을 이룬 락킹 계층을 사용할 수 있는데, 이 방법은 Figure 7.6에 그려져 있습니다. 여기서, `cmp()` 함수는 Lock A, B, C 가 잡힌 이후에 새로운 Lock D 를 사용함으로써 데드락을 막습니다. 따라서 전역적인 데드락 계층에 세개의 층이 존재하는 셈인 것으로, 첫번째 층은 Lock A 와 B 에, 그리고 두번째 층은 Lock C 를 가지며, 세번째 층은 Lock D 를 포함합니다.

`cmp()` 함수가 새로운 락인 Lock D 를 사용하도록 수정하는 것을 기계적으로 하는 건 일반적으로 불가능하다는 것을 알아 두시기 바랍니다. 오히려 그 반대입니다: 그러한 변경을 위해서는 대부분의 경우 기초적인 설계 레벨에서의 수정을 필요로 하게 됩니다. 하지만 더도 아니고 덜도 아니고, 그런 변경을 위해 필요한 노력은 일반적으로 데드락을 막기 위해서 치뤄야 하는 대가 치고는 적은 편입니다.

알 수 없는 코드를 실행하기 전에 모든 락을 해제하는 것이 비현실적인 또 다른 예를 들어보기 위해서는, Figure 7.7 (`locked_list.c`)에 보여진 것과 같이 링크드 리스트를 순회하는 반복자 (iterator) 를 생각해 보

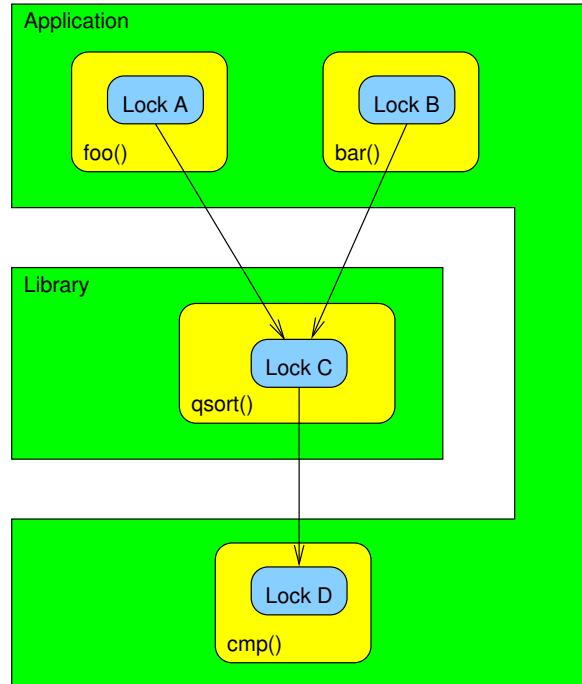


Figure 7.6: Layered Locking Hierarchy for `qsort()`

시기 바랍니다. `list_start()` 함수는 리스트의 락을 잡고서 첫번째 원소를 리턴하고 (원소가 존재한다면), `list_next()` 함수는 리스트의 그 다음 원소로의 포인터를 리턴하는데, 리스트의 끝에 도달한 경우에는 앞서 잡았던 락을 해제하고 `NULL` 을 리턴합니다.

Figure 7.8는 이 리스트 반복자가 어떻게 사용될 수 있는지 보여줍니다. Line 1-4는 하나의 정수를 포함하는 `list_ints` 원소를 정의하고 line 6-17에서는 이 원소가 포함된 리스트에의 순회를 어떻게 반복하는지 보입니다. Line 11에서는 이 리스트를 잡고 첫번째 원소로의 포인터를 얻어온 후, line 13에서 해당 원소에 동봉되어 있는 `list_ints` 구조체로의 포인터를 얻어오고, line 14에서 연관된 정수를 출력하고서, line 15에서 다음 원소로 넘어갑니다. 이 코드는 상당히 간단하고, 락킹을 모두 숨겼습니다.

각 리스트의 원소를 처리하는 코드가 그 스스로 `list_start()` 나 `list_next()` 를 호출하는 다른 코드와 함께 사용하는, 데드락을 초래할 수 있는, 락을 사용하지 않는 한은 이 락킹은 계속 숨겨져 있을 수 있습니다. 이렇게 리스트 반복자 락킹을 구현하면서 데드락을 막기 위해 락킹 계층을 여러 레이어에 층층이 놓을 수 있습니다.

이 층쌓기 방법은 임의의 많은 수의 레이어들로 확장될 수도 있습니다만, 각각의 추가된 레이어는 락킹 설계

```

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

```

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

의 복잡도를 증가시킵니다. 그런 증가된 복잡도는 특히 객체지향적 설계의 어떤 타입들에는 불편할 수 있는데, 코드의 재어와 수행이 많은 수의 객체들 사이를 규칙이 정해지지 않은 형태로 왔다 갔다 하게 되는 경우가 그런 경우입니다.¹ 이런 객체지향적 설계의 습관들과 데드락 회피의 필요성 사이의 미스매치는 병렬 프로그래밍이 어떤 사람들에게는 매우 어렵다고 인지되는데 대한 심각한 이유입니다.

수많은 레이어를 사용하는 락킹 계층들에 대한 일부 대안책은 Chapter 9에서 다뤄집니다.

7.1.1.4 Locking Hierarchies and Pointers to Locks

일부 예외가 있긴 하지만, 락으로의 포인터를 갖는 외부의 API는 대부분의 경우 잘못 설계된 API입니다. 내부의 락을 어떤 다른 소프트웨어 컴포넌트에서 처리하는 것은 일단 정보 은닉이라는 핵심 설계 원칙에 위배되는 행위입니다.

Quick Quiz 7.5: 락으로의 포인터를 함수에 넘기는 게 완벽하게 합리적인 예외를 하나만 들어 보세요. ■

예외 가운데 하나는 어떤 것을 떠맡기게 되는 함수들인데, 호출자의 락은 떠맡기기 하기 전에 반드시 잡혀야만 하지만 그 락은 해당 함수가 리턴하기 전에는 해제되어야만 하는 경우입니다. 그런 함수의 한 예는 POSIX `pthread_cond_wait()` 와 같은 함수가 될텐데, `pthread_mutex_t` 로의 포인터를 넘김으로써 일어날 조건을 놓쳐서 계속 깨어날 조건을 기다리고만 있게 되는 일을 방지합니다.

Quick Quiz 7.6: `pthread_cond_wait()` 함수가 먼저 그 뮤텍스를 해제하고 나서 다시 획득한다는 사실은 데드락의 가능성을 제거하는 거 아닌가요? ■

다시 말해, 어떤 락으로의 포인터를 인자로 받거나 리턴 값으로 취하는 API를 외부로 노출하고자 한다면 그 API 설계를 다시한번 조심스럽게 고려해 보시기 바랍니다. 그건 해야만 할 옳은 일일 수도 있을지 모르겠습니다만, 그간의 경험은 그렇지 않을 가능성이 높음을 이야기 합니다.

7.1.1.5 Conditional Locking

하지만 합리적인 락킹 계층이 존재하지 않는다고 생각해 봅시다. 이런 상황은 실제 세상에 존재할 수 있는데, 예를 들어보자면 패킷이 양방향에서 오가는 계층적 네트워크 프로토콜 스택을 생각해 볼 수 있겠습니다. 이 네트워킹의 경우, 한 패킷을 한 계층에서 다른 계층으로 보내고자 할 때 양 계층에서 모두 락을 잡아야 할 필요가 있을 수 있습니다. 패킷들은 이 프로토콜 스택의 위로도 아래로도 이동될 수 있다는 점을 생각해 보면,

¹ 이런 것들을 일컫기를 “객체지향적 스파게티 코드”라고 합니다.

```

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

Figure 7.9에 보여진 것처럼 데드락을 유발하기 위한 좋은 방법입니다. 여기서, 랜선을 향해 스택의 아래 방향으로 이동하게 되는 패킷은 다음 계층의 락을 위에서 아래 순서대로 잡아야만 합니다. 랜선으로부터 스택의 위 방향으로 이동하게 되는 패킷들은 그 락들을 아래에서 위 순서대로 잡아야 한다는 점을 상기해 보면, 해당 그림의 line 4에서의 락 획득은 데드락을 초래할 수 있습니다.

이 경우에 데드락을 막을 수 있는 한가지 방법은 락킹 계층을 적용하되, 락을 위에서 아래 순서로 잡아야 할 경우에는 Figure 7.10에 보인 것처럼 조건적으로 잡는 것입니다. 무조건적으로 layer-1 락을 획득하는 대신에, line 5에서는 그 락을 `spin_trylock()` 기능을 사용해 조건적으로 획득합니다. 이 기능은 만약 락이 획득 가능하다면 곧바로 락을 획득해 버리고 (0이 아닌 값을 리턴합니다), 그렇지 않다면 락을 잡지 않고 0을 리턴합니다.

만약 `spin_trylock()`이 성공했다면, line 15는 필요한 layer-1의 처리를 진행합니다. 그렇지 않다면, line 6에서 락을 해제하고 line 7과 8에서 올바른 순서대로 그 락들을 다시 잡습니다. 불행히도, 시스템에는 여러 네트워킹 디바이스들이 존재할 수 있는데 (예: 이더넷과 WiFi), 따라서 `layer_1()` 함수는 라우팅 결정을 해야만 합니다. 이 결정은 언제든 바뀔 수도 있는데, 특히 그 시스템이 모바일 기기라면 더욱 그렇습니다.² 따라서, line 9는 그 결정을 다시 한번 체크해 봐야 하고,

바뀌었다면 그 락들을 놓고 일을 처음부터 다시 시작해야 합니다.

Quick Quiz 7.7: Figure 7.9에서 Figure 7.10로의 변경이 어디서나 적용될 수 있는 걸까요? ■

Quick Quiz 7.8: 그렇지만 Figure 7.10에서의 추가된 복잡도는 그게 데드락을 막는다는 점을 생각하면 꽤 가치있는 거예요, 맞죠? ■

7.1.1.6 Acquire Needed Locks First

조건적 락킹의 중요하고 특수한 한 케이스에는 모든 필요한 락들이 추후의 작업 진행을 이루기 전에 획득됩니다. 이런 경우에, 처리하는 일이 면등성을 가질 필요는 없습니다: 이미 쥐고 있는 락들 가운데 하나를 먼저 놓지 않고는 어떤 락을 잡는 것이 불가능함이 밝혀진다면, 그냥 모든 락을 놓아버리고 다시 락 획득을 시도합니다. 모든 필요한 락들을 한번이라도 모두 잡기만 한다면 추후의 작업이 진행됩니다.

하지만, 이런 처리 방법은 *livelock*을 초래할 수 있는데, 이에 대해서는 Section 7.1.2에서 이야기 합니다.

Quick Quiz 7.9: Section 7.1.1.6에서 이야기한 “필요한 락들을 먼저 획득하기” 방법을 사용할 때에, 어떻게 하면 *livelock*을 막을 수 있을까요? ■

관련된 방법으로 two-phase 락킹 [BHG87]이 있는데, 이 방법은 트랜잭션을 제공하는 상용 수준의 데이터베이스 시스템들에서 오랫동안 사용되어 왔습니다. Two-phase 락킹 트랜잭션의 첫번째 페이스 (phase)에서는 락들은 획득은 되지만 해제는 되지 않습니다. 모든 필요한 락들이 획득된다면, 이 트랜잭션은 두번째 페이스로 들어가게 되는데, 이 페이스에서는 락들이 해제만 되지, 획득은 되지 않습니다. 이 락킹 방법은 데이터베이스들이 `serializability`를 트랜잭션들에게 보장해줄 수 있게 하는데, 달리 말하자면, 트랜잭션들에 의해 만들어지고 보여지는 모든 값들이 모든 트랜잭션들의 어떤 전체적인 순서에 대해 일관적이게 될 것을 보장해 주기 위해 사용됩니다. 그런 많은 시스템들은 트랜잭션들을 중단 (abort) 시킬 수 있는 기능에 의존하는데, 사실 모든 필요한 락들이 획득되기 전까지는 공유된 데이터에 어떤 변경도 가하지 않도록 하는 것으로 단순화 시킬 수도 있습니다. *Livelock*과 *deadlock*은 그런 시스템들에서의 문제거리들인데, 실용적인 해결책들은 많은 데이터베이스 서적들에서 찾을 수 있을 겁니다.

7.1.1.7 Single-Lock-at-a-Time Designs

어떤 경우에는, 락들을 다른 락 안에 넣는 것을 막을 수 있는데, 따라서 데드락을 막을 수 있습니다. 예를 들어, 만약 문제가 완벽하게 분리될 수 있는 성질의 것이라면, 각 분리된 파티션에 하나의 락을 각각 할당할 수 있을 겁니다. 이렇게 되면 하나의 주어진 파티션을 작업하는

² 그리고, 1900년대에는 다르게, 모바일 기기는 상당히 흔해졌죠.

쓰레드는 그에 연관된 락 하나만을 잡으면 됩니다. 어떤 쓰레드도 한번에 하나 이상의 락을 잡지는 않으므로, 데드락은 발생할 수 없습니다.

하지만, 어떤 락도 잡혀있지 않은 때에도 필요한 데이터 구조체들은 존재함을 보장해줄 어떤 메커니즘이 반드시 있어야만 합니다. 그런 메커니즘 가운데 하나를 Section 7.4에서 이야기 해보고, Chapter 9에서 그 외에 몇 가지 다른 메커니즘들을 살펴봅니다.

7.1.1.8 Signal/Interrupt Handlers

시그널 핸들러와 연관된 데드락들은 시그널 핸들러 안에서 `pthread_mutex_lock()` 함수를 호출하는 것은 허용되지 않는 행위라는 것을 알리는 것만으로도 빠르게 없앨 수 있습니다 [Ope97]. 하지만, 시그널 핸들러 안에서 사용될 수 있는 손으로 만든 락킹 도구 (대부분의 경우 지혜롭지 못한 행동입니다만)를 사용할 가능성도 존재합니다. 그와는 별개로, 모든 운영체제 커널들은 커널에서는 시그널 핸들러라는 유사한 이름으로 다뤄지는 인터럽트 핸들러 안에서의 락 획득을 허가합니다.

여기서의 트릭은 인터럽트 핸들러 안에서 잡을 수도 있는 락이라면 어떤 락이든지 잡을 때마다 시그널들을 막아버리는 (또는, 경우에 따라서, 인터럽트를 무효화 시킬 수도 있습니다) 겁니다. 더 나아가서, 그런 락을 잡고 있다면, 시그널들을 블락시키지 않은 채로 시그널 핸들러 밖에서 한번이라도 잡은 적 있는 락을 잡으려는 시도 역시 해선 안됩니다.

Quick Quiz 7.10: 시그널 핸들러 안에서 획득되는 Lock B 를 잡은 채로 시그널들을 블락시키지 않고 시그널 핸들러 밖에서 획득되는 Lock A 를 잡는 행위가 해선 안되는 행위이죠? ■

일부 시그널들을 위한 핸들러들에서 한 락을 잡았다면, 그 시그널들 모두는 그 락이 획득될 때마다, 심지어 그 락이 시그널 핸들러 안에서 잡힐 때 조차도 블락되어야만 합니다.

Quick Quiz 7.11: 시그널 핸들러 안에서 어떻게 시그널들을 블락시킬 수가 있죠? ■

안타깝게도, 시그널들을 블락하고 블락 해제하는 행위는 일부 운영체제에서는 많은 비용을 필요로 할 수 있는데, 특히 리눅스도 포함되고, 따라서 성능에 주안점을 둔다는 말은 곧 시그널 핸들러들 안에서 획득되는 락들은 시그널 핸들러들 안에서만 획득되며, 어플리케이션 코드와 시그널 핸들러 사이의 통신에는 락을 사용하지 않는 동기화 메커니즘만을 사용한다는 의미가 되곤 합니다.

또는 치명적인 문제 상황을 처리하기 위한 경우를 제외하고는 시그널 핸들러를 아예 사용하지 않는 방법도 있습니다.

Quick Quiz 7.12: 시그널 핸들러들 안에서 락을 잡는

```

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

행위가 그렇게 나쁜 생각이라면, 그걸 안전하게 해내기 위한 생각 자체는 대체 왜 하는거죠? ■

7.1.1.9 Discussion

공유 메모리 병렬 프로그래머가 사용할 수 있는 수많은 데드락 방지 전략이 존재합니다만 그 중 어느것도 잘 들어맞지 않는 순차적 프로그램들도 존재하지요. 이게 전문적인 프로그래머라면 자신의 도구상자에 두개 이상의 도구를 갖추고 있는 이유입니다: 락킹은 훌륭한 동시성 제어 도구이지만, 다른 도구들을 가지고 다룰 수 있는 일들도 많습니다.

Quick Quiz 7.13: 다양한 객체들 사이로 실행 제어가 완전 자유롭게 오가는 객체 지향 어플리케이션이어서 직관적인 락킹 계층, 레이어, 또는 다른 것들이 존재하지 않는 경우³ 라면 그 어플리케이션은 어떻게 병렬화 시켜야 할까요? ■

과장 없이 이야기 하건대, 이 섹션에서 이야기된 전략들은 많은 환경에서 꽤 유용한 것으로 입증되었습니다.

7.1.2 Livelock and Starvation

조건적 락킹은 효과적인 데드락 예방책이긴 하지만, 남용될 수도 있습니다. 예를 들어서, Figure 7.11에 보여진 것과 같이 아름답도록 대칭적인 예제 코드에 대해 생각해 봅시다. 이 예제의 아름다운 부분은 추한 livelock의 가능성을 감추고 있습니다. 이를 드러내 보이기 위해,

³ “객체 지향 스파게티 코드”라고도 알려져 있죠.

다음과 같은 일련의 이벤트들이 일어나는 경우를 생각해 봅시다:

1. Thread 1 이 line 4에서 lock1 을 잡는데 성공하고 do_one_thing() 을 호출합니다.
2. Thread 2 가 line 18에서 lock2 를 잡고 do_a_third_thing() 을 호출합니다.
3. Thread 1 이 line 6에서 lock2 를 잡으려 시도합니다만 Thread 2 가 그 락을 이미 잡고 있기 때문에 실패합니다.
4. Thread 2 가 line 20에서 lock1 을 획득하려 시도하지만, Thread 1 이 그 락을 이미 잡고 있으므로 실패합니다.
5. Thread 1 이 line 7에서 lock1 을 해제하고 line 3의 retry 로 점프합니다.
6. Thread 2 가 line 21에서 lock2 를 해제하고 line 17의 retry 로 점프합니다.
7. 이런 livelock 덴스가 시작부터 다시 반복됩니다.

Quick Quiz 7.14: Figure 7.11에 보여진 livelock 은 어떻게 예방할 수 있나요? ■

Livelock 은 starvation 의 한 극단적인 예로 생각되어 질 수 있는데, 여러 쓰레드들 중 하나만이 아니라 모든 쓰레드들이 굽게 되는 경우라고 볼 수 있는 것입니다.⁴

Livelock 과 starvation 은 소프트웨어 트랜잭션을 예방하는 문제입니다. 따라서 이런 문제들을 해결해 주기 위해 *contention manager* 의 개념이 만들어졌습니다. 락킹의 경우에는 exponential backoff 방법으로 이 livelock 과 starvation 을 해결할 수 있습니다. 이 방법은 Figure 7.12에 보여진 것처럼 매번 재시도를 할 때마다 시도에 실패한 다음 재시도하기 전까지 기다리는 딜레이 시간을 지수적으로 증가시키는 방법입니다.

Quick Quiz 7.15: Figure 7.12에서 어떤 문제를 발견 할 수 있나요? ■

하지만, 더 나은 결과를 위해, 이 backoff 의 최대 크기는 어딘가에 바운드 되어야만 하고, 심지어 이보다도 높은 경쟁상황에서의 더 나은 결과가 Section 7.3.2에서 이야기 되는 대기열을 서는 락킹 [And90]을 통해 관찰되었습니다. 물론, 가장 최선의 방법은 락 경쟁이 낮은 수준으로 유지되는 좋은 병렬 설계를 사용하는 것입니다.

⁴ Livelock, starvation, 그리고 unfairness 같은 단어들의 정확한 정의에 대해 너무 매달려 있지 마시기 바랍니다. 어떤 쓰레드들의 무리가 앞으로의 진행을 만들거나 못하고 있는 상황은 그 상황에 어떤 이름을 붙일지에 상관없이 해결되어야만 하는 문제일 뿐입니다.

```

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

7.1.3 Unfairness

불공정성은 starvation 의 덜 심각한 형태라고 생각되어 질 수 있는데, 어떤 주어진 락을 가지고 경쟁하고 있는 쓰레드들 가운데 일부 쓰레드들이 그 락을 획득하는데 더 좋은 우선순위를 갖는 경우입니다. 이는 공유된 캐시들이나 NUMA 특성을 가지고 있는 기계들에서 일어날 수 있는 일인데, 예를 들어 Figure 7.13 같은 경우입니다. CPU 0 가 다른 모든 CPU 들이 획득하려 하고 있는 락을 놓게 되는 경우, CPU 0 과 1 사이의 공유된 접합부는 CPU 1 이 CPU 2-7 에 비해 혜택을 갖게 될 것임을 의미합니다. 따라서 CPU 1 은 그 락을 잡는데 성공할 확률이 큽니다. CPU 1 이 그 락을 CPU 1 이 놓게 되는 시점에 CPU 0 이 그 락을 다시 잡으려 시도할 만큼 충분히 오래 잡고 있다가 놓게 되거나 그 반대 경우라면, 이 락은 CPU 0 과 1 사이에서만 오가게 되고 CPU 2-7 은 소외되어 버리게 됩니다.

Quick Quiz 7.16: 락 경쟁정도가 충분히 낮아서 비공정성을 예방할 수 있을 정도가 되는 좋은 병렬 설계를 사용하는 편이 더 낫지 않을까요? ■

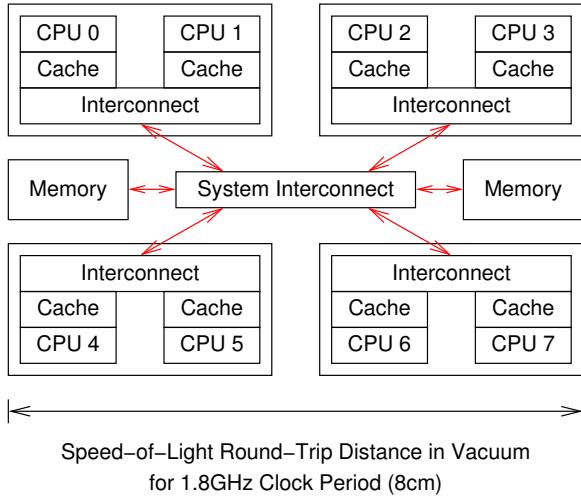


Figure 7.13: System Architecture and Lock Unfairness

7.1.4 Inefficiency

락들은 어토믹 인스트럭션들과 메모리 배리어들을 사용해서 구현되고 자주 캐시 미스들을 유발합니다. Chapter 3에서 봤듯이, 이런 인스트럭션들은 상당히 그 비용이 비싼데, 간단한 인스트럭션들에 비해 대략 수백 배에 달할 정도로 더 큰 비용입니다. 이는 락킹에 있어 상당한 문제가 될 수 있습니다: 만약 하나의 인스트럭션을 락으로 보호하게 된다면 그 오버헤드를 약 100배 정도 증가시키는 겁니다. 완벽한 확장성이 제공된다고 가정한다고 해도, 똑같은 코드를 락킹 없이 수행하는 하나의 CPU의 성능을 따라잡기 위해서는 100개의 CPU들이 필요해질 수 있는 겁니다.

이런 상황이 Section 6.3에서, 특히 Figure 6.22를 통해 이야기한 동기화의 단위에 대한 트레이드오프의 중요성을 강조합니다: 너무 큰 단위로 동기화를 하게 되는 것은 확장성을 제한하지만, 너무 작은 단위로 동기화를 하게 되면 지나치게 큰 동기화 오버헤드를 초래하게 됩니다.

그렇다고는 하나, 일단 락이 잡히면, 그 락에 의해 보호되는 데이터는 그 락을 잡은 쓰레드에 의해서는 별다른 간섭 없이 접근될 수 있습니다. 락을 잡는 행위는 비용이 비쌀 수도 있습니다, 하지만 일단 잡고 난 다음에는, 그 CPU의 캐시들은 효과적으로 성능을 끌어올려주는데, 적어도 커다란 크리티컬 섹션에 대해서는 그렇습니다.

Quick Quiz 7.17: 락을 잡은 쓰레드는 어떻게 다른 쓰레드로부터 간섭을 받을 수도 있을까요? ■

7.2 Types of Locks

놀랍도록 많은 종류의 락들이 존재하는데, 이 짧은 챕터가 모두 다룰 수 없을 정도로 많습니다. 다음의 섹션들에서는 배타적 락들 (Section 7.2.1) reader-writer 락들 (Section 7.2.2), 다양한 역할의 락들 (Section 7.2.3), 그리고 범위적 락킹 (Section 7.2.4)에 대해 알아봅니다.

7.2.1 Exclusive Locks

배타적 락들은 이름이 말하는 그대로입니다: 한번에 하나의 쓰레드만이 그 락을 쥐고 있을 수 있습니다. 따라서 그런 락을 쥐고 있는 쓰레드는 그 락에 의해 보호되는 모든 데이터에 대한 배타적인 접근을 할 수 있게 되고, 따라서 그런 이름을 갖게 되었습니다.

물론, 이 모든 이야기는 이 락은 그 락에 의해 보호된다고 여겨지는 데이터에 대한 모든 접근에 걸쳐서 잡혀 있게 된다는 가정을 하고 있습니다. 모든 필요한 코드 수행 경로에 이 락이 잡혀져 있도록 보장하는 막대한 책임은, 비록 그 책임의 수행을 도와주는 도구들도 있기는 하지만, 결국은 그 코드의 개발자에게 있습니다.

Quick Quiz 7.18: 배타적 락을 잡고나서 곧바로 풀어버리는, 즉, 텅 빈 크리티컬 섹션과 같은 것을 갖는 행위가 어떤 의미를 가질 수 있을까요? ■

7.2.2 Reader-Writer Locks

Reader-writer 락 [CHP71]는 그 락이 보호하는 데이터를 읽는 쓰레드는 다른 쓰레드와 동시에도 얼마든지 잡을 수 있게 허용하는 한편 그 데이터를 쓰는 쓰레드는 단 하나의 쓰레드만이 그 락을 잡을 수 있도록 합니다. 이렇게 되면, 이론상으로는, reader-writer 락들은 주로 읽히기만 하고 쓰여지는 일은 매우 드문 데이터에 대해서는 훌륭한 확장성을 보여야만 합니다. 실전에서는, 실제 확장성은 해당 reader-writer 락의 구현 내용에 따라서 달라질 겁니다.

고전적인 reader-writer 락 구현은 어토믹하게 조정되는 여러개의 카운터들과 플래그들을 사용했습니다. 이런 종류의 구현들은 짧은 크리티컬 섹션으로 인해 배타적 락킹이 받는 것과 같은 문제로 어려움을 겪었습니다: 락을 획득하고 해제하는데 드는 오버헤드가 간단한, 실제 할일만 하는 인스트럭션의 오버헤드에 비해 수백배 가까이 크게 되는 현상입니다. 물론, 이 크리티컬 섹션이 충분히 길다면 그 락을 획득하고 해제하는데 드는 오버헤드는 무시해도 좋은 정도가 됩니다. 하지만, 한번에 하나의 쓰레드만이 그 락을 조정할 수 있으므로, CPU의 수가 들어남에 따라 필요시되는 크리티컬 섹션의 크기는 커집니다.

쓰레드별 배타적 락을 사용해서 읽기를 하는 쓰레드에 훨씬 더 바람직한 형태로 reader-writer 락을 설계하는

	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

것도 가능합니다 [HW92]. 읽기를 하기 위해서는, 쓰레드는 자신의 락만을 획득합니다. 쓰기를 하기 위해서는, 쓰레드는 모든 락들을 획득해야만 합니다. 쓰기를 하려는 쓰레드가 존재하지 않는 상황이라면, 각 읽기를 하는 쓰레드는 어토믹 인스트럭션과 메모리 배리어 오버헤드만을 만들어내고, 캐시 미스도 내지 않는데, 이는 락킹 기능에 있어 상당히 좋은 상황입니다. 불행히도, 쓰기를 하는 쓰레드는 캐시 미스는 물론이고 어토믹 인스트럭션과 메모리 배리어 오버헤드까지 모두 만들어내게 됩니다—그 오버헤드는 또한 쓰레드들의 수만큼 배가 되지요.

요약하자면, reader-writer 락들은 여러 상황에 상당히 유용할 수 있습니다만, 그 구현은 각 종류마다 그自身的 단점을 가지고 있습니다. 규범적인 reader-writer 락킹의 사용은 상당히 큰 읽기 쓰레드 쪽의 크리티컬 섹션을 가지고 있을 때에만 이루어지는데 그 크리티컬 섹션의 크기는 수백 마이크로세컨드부터 심지어 밀리세컨드까지 될 수도 있습니다.

7.2.3 Beyond Reader-Writer Locks

Reader-writer 락과 배타적 락은 각자 허용하는 범위에 대한 정책에서 차이를 보입니다: 배타적 락은 하나의 쓰레드만 락을 잡고 있기를 허용하는 반면, reader-writer 락은 읽기 권한으로 락을 잡는 쓰레드는 얼마든지 락을 잡고 있을 수 있게 허용합니다 (하짐나 쓰기 권한의 경우는 하나만 가능하죠). 매우 많은 락 허용 범위 정책이 있을 수 있는데, 그 중 하나는 VAX/VMS 분산 락 매니저 (DLM) [ST87] 으로, Table 7.1에 표시되어 있습니다. 비어있는 셀들은 호환 가능한 모드들을 의미하며 “X”로 표시된 셀은 호환 불가한 모드들을 가리킵니다.

VAX/VMS DLM은 여섯개의 모드를 사용합니다. 비교를 위해 다른 락들의 모드를 이야기해보면, 배타적 락들은 두개의 모드를 (not held 와 held), reader-writer

락은 세개의 모드 (not held, read held, write held) 를 갖는다고 이야기 할 수 있습니다.

첫번째 모드는 null, 또는 not held 라 불립니다. 이 모드는 모든 다른 모드들과 호환되는데, 즉 다음과 같은 해석이 가능합니다: 한 쓰레드가 락을 잡고 있지 않다면, 그 쓰레드는 다른 어떤 쓰레드가 락을 획득하려 할 때 어떤 방해도 하지 않습니다.

두번째 모드는 concurrent read 인데, exclusive 를 제외한 다른 모든 모드들과 호환됩니다. concurrent-read 모드는 데이터 구조에의 업데이트가 동시적으로 진행되도록 허용하는 가운데 데이터의 대략적 통계치를 구하기 위해 사용될 수도 있습니다.

세번째 모드는 concurrent write 로, null, concurrent read, 그리고 concurrent write 와 호환됩니다. concurrent-write 모드는 데이터의 읽기와 수정을 동시적으로 진행될 수 있도록 허용하는 가운데 대략적인 통계치를 업데이트 하기 위해 사용될 수도 있습니다.

네번째 모드는 protected read 로, null, concurrent read, 그리고 protected read 와 호환되는 모드입니다. protected-read 모드는 동시적인 읽기는 허용하되 업데이트는 허용하지 않는 가운데 데이터 구조체의 일관성을 갖는 스냅샷을 만들거나 할 때 사용될 수도 있습니다.

다섯번째 모드는 protected write 인데, 이 모드는 null 과 concurrent read 모드와 호환됩니다. protected-write 모드는 protected read 모드의 읽기 작업에는 영향을 받을 수 있지만 concurrent read 모드의 읽기 작업에는 문제가 없는 데이터 구조체에의 업데이트를 하는데 사용될 수도 있습니다.

마지막으로 여섯번째 모드는 exclusive 모드인데, 이 모드는 null 모드와만 호환됩니다. exclusive 모드는 모든 다른 액세스들을 제외시켜야만 할 때 사용될 수 있습니다.

배타적 락과 reader-writer 락은 VAX/VMS DLM 으로 모방될 수 있다는 사실은 흥미로운 일입니다. 배타적 락은 null 과 exclusive 모드만을 사용하는 반면 reader-writer 락은 null, protected-read, 그리고 protected-write 모드를 사용할 것입니다.

Quick Quiz 7.19: VAX/VMS DLM 이 reader-writer 락을 다른 방법으로 모방해낼 수도 있을까요? ■

VAX/VMS DLM 정책은 분산 데이터베이스 분야의 상용 제품에 널리 사용되고 있긴 합니다만, 공유 메모리 어플리케이션에서는 그렇게 꽉꽉 사용되고 있지는 않은 것으로 보입니다. 이에 대한 그럴싸한 이유 중 하나는 분산 데이터베이스에서의, 공유 메모리 시스템에서 비해 엄청나게 큰 통신 오버헤드가 VAX/VMS DLM 의 좀 더 복잡해지는 관리 정책의 오버헤드를 숨겨줄 수 있다는 것입니다.

더도 아니고 덜도 아니고, VAX/VLS DLM 은 락킹

뒤의 개념이 얼마나 유연해질 수 있는지를 보여주는 하나의 흥미로운 예입니다. VAX/VLS DLM은 또한 최신 DMBs 들에서 사용되는, 서른개가 넘는 락킹 모드를 가지기도 해서 VAX/VMS 의 여섯개 모드에 비해 훨씬 복잡한 락킹 개념에 대한 매우 간단한 소개 역할을 하기도 합니다.

7.2.4 Scoped Locking

앞서 이야기된 락킹 기능들은 따라서 명시적인 획득과 해제 기능을 필요로 하게 되는데, 예를 들면 `spin_lock()` 과 `spin_unlock()` 같은 것들이 각각 획득과 해제를 위해 사용됩니다. 다른 방법은 객체 지향적인 “자원 할당이 곧 초기화”(RAII: Resource Allocation Is Initialization) 패턴 [ES90] 을 사용하는 것입니다.⁵ 이 패턴은 C++ 같은 언어의 `auto` 변수들에 자주 사용되는 것으로, 어떤 객체의 범위에 진입할 때에는 연관된 생성자(*constructor*)가 호출되고, 그 객체의 범위에서 빠져나올 때에는 연관된 소멸자(*destructor*)가 호출되는 방식입니다. 이 패턴은 생성자가 락을 획득하고 소멸자는 락을 놓는 형태로 락킹에 적용될 수 있습니다.

이런 접근법은 상당히 유용할 수 있는데, 실제로 1990년도에 저는 이것이야말로 필요한 락킹의 종류라고 확신을 설득되었던 바 있습니다.⁶ RAII 락킹의 배우 훌륭한 특성 하나는 그 범위를 빠져나가는 모든 코드 수행경로 각각에 대해서 조심스럽게 락을 해제할 필요가 없다는 것으로, 이로 인해 문제가 될 여지가 있는 여러 버그들이 제거되어버립니다.

하지만, RAII 락킹은 또한 어두운 부분을 가지고 있습니다. RAII 는 락 획득과 해제의 캡슐화를 매우 어렵게 해주는데, 반복자에서의 경우가 한 예가 될 것입니다. 많은 반복자 구현에서, 사람들은 반복자의 “start” 함수에서 락을 잡아 놓고 이 반복자의 “stop” 함수에서 그 락을 풀고 싶어 할 것입니다. RAII 락킹은 그러지 않고 락 획득과 해제가 같은 수준의 범위에서 이뤄지도록 함으로써 그러한 종류의 캡슐화를 어렵게하고 심지어는 불가능하게 하기도 합니다.

RAII 락킹은 또한 크리티컬 섹션들을 겹치게 배치하는 것을 불가능하게 하는데, 범위가 내포되어야 한다는 사실 때문입니다. 이런 불가능은 여러개의 유용한 구조를 어렵거나 아예 불가능하게 만들어 버리는데, 예를 들면 어떤 이벤트를 단언하려는 여러개의 동시적인 시도들 사이에서 중재를 하는, 락을 사용하는 트리 자료구조 같은 것입니다. 임의의 수많은 동시적인 시도들 가운데

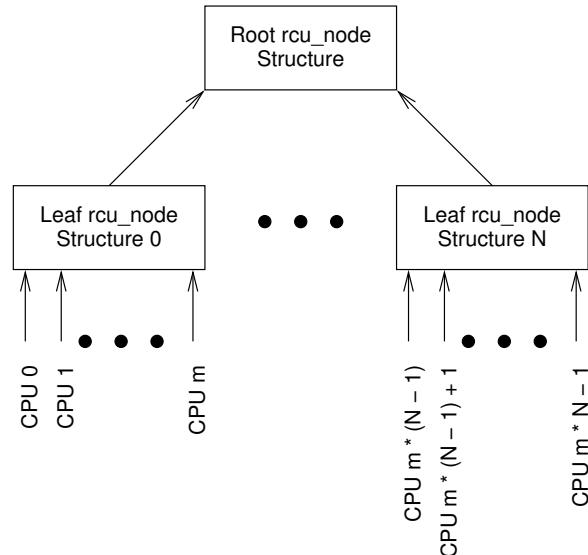


Figure 7.14: Locking Hierarchy

오직 하나의 시도만이 성공할 수 있어야 하고, 그 성공한 시도 이외의 것들을 위한 최선의 전략은 가능한한 빨리, 그리고 가능한한 별다른 고통 없이 실패에 이르는 것입니다. 그러지 않는다면, 락 경쟁정도가 커다란 규모의 시스템에서는 고통스러울 정도가 될 것입니다 (“커다란 규모”는 수백개의 CPU 들을 의미합니다).

예로 들 수 있는 데이터 구조체들 (리눅스 커널의 RCU 구현에서 가져온 것들입니다) 이 Figure 7.14 에 보여져 있습니다. 여기서, 각 CPU 는 leaf `rcu_node` 구조체를 할당받아 가지고 있고, 각 `rcu_node` 구조체는 자신의 부모로의 포인터 (기묘하게도, `->parent` 라 이름붙여져 있습니다) 를 가지고 있어서 이 구조는 트리의 루트 `rcu_node` 구조체까지 이어지는데, 이 루트는 `->parent` 포인터의 값으로 `NULL` 을 갖습니다. 하나의 부모 `rcu_node` 구조체당 가질 수 있는 자식들의 수는 다양할 수 있는데, 일반적으로는 32 또는 64 가 사용됩니다. 각각의 `rcu_node` 구조체는 또한 `->fqsllock` 이라 이름 붙여진 락을 갖습니다.

일반적인 접근 방법은 토너먼트 로, 한 CPU 는 조건적으로 자신에게 할당된 leaf `rcu_node` 구조체의 `->fqsllock` 을 잡고, 그 락을 잡는데 성공했다면, 그 구조체의 부모의 락을 잡으려 시도하고, 자식의 락은 해제하는 방법입니다. 또한, 각 레벨에서, 이 CPU 는 글로벌 `gp_flags` 변수를 체크하는데, 만약 이 변수가 어떤 다른 CPU 가 그 이벤트를 단정했다면, 이 첫번째 CPU 는 더이상의 경쟁을 그만둡니다. 이 획득한 후 해제하기 시퀀스는 `gp_flags` 변수가 누군가가 토너먼트에 승리했다는 것을 알리게 되거나, `->fqsllock` 획득에 실

⁵ http://www.stroustrup.com/bs_faq2.html#finally에 더 분명하게 설명되어 있습니다만.

⁶ 이후의 Sequent Computer Systems 에서의 병렬성 관련한 업무 경험을 통해 저는 곧잘된 개념으로부터 오해를 풀 수 있었습니다.

```

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->fqlock));
10    if (rnp_old != NULL)
11        raw_spin_unlock(&rnp_old->fqlock);
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->fqlock);
22 }

```

Figure 7.15: Conditional Locking to Reduce Contention

패했거나, 또는 root rcu_node 구조체의 \rightarrow fqlock이 획득될 때까지 반복됩니다.

이를 구현하는 간략화된 코드가 Figure 7.15에 보여집니다. 이 함수의 목적은 동시에 `do_force_quiescent_state()` 함수를 실행해야 할 필요성을 감지한 CPU들 사이의 중재를 하는 것입니다. 어떤 때에든, `do_force_quiescent_state()` 함수의 하나의 실행 흐름만이 실행되고 있어야만 의미가 지켜지며, 따라서 이 함수에의 여러 동시에 호출자들이 존재한다면, 그들 중 하나만이 정말로 `do_force_quiescent_state()` 함수를 호출하고, 나머지들은 (가능한 빠르고 고통 없이) 포기하고 떠나도록 해야만 합니다.

이런 결론에 따라, line 7-15의 루프에서의 실행 흐름은 `rcu_node` 계층에서 한단계 위의 레벨로 올라가려는 시도를 합니다. `gp_flags` 변수가 이미 설정되었거나 (line 8) 현재의 `rcu_node` 구조체의 \rightarrow fqlock을 획득하려는 시도가 실패한다면 (line 9), 지역변수 `ret`은 1로 설정됩니다. 만약 line 10이 지역변수 `rnp_old`가 NULL이 아님을 확인한다면, 그 말은 `rnp_old`의 \rightarrow fqlock을 잡았다는 의미이므로, line 11에서 그 락을 해제합니다 (하지만 이는 그 부모 `rcu_node` 구조체의 \rightarrow fqlock을 잡은 후의 일입니다). 만약 line 12에서 line 8이나 line 9에서 경쟁을 그만둘 이유를 발견했음을 확인하면 line 13에서 호출자에게 리턴해 버립니다. 그렇지 않다면, 현재의 `rcu_node` 구조체의 \rightarrow fqlock을 잡은 것이므로, 이 루프의 다음 실행경로를 위한 준비과정으로 line 14에서 이 구조체로의 포인터를 지역 변수 `rnp_old`에 집어넣습니다.

만약 코드 수행 제어가 line 16 까지 도달한다면 토너먼트에 우승했고, 지금 root `rcu_node` 구조체의 \rightarrow

```

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

```

Figure 7.16: Sample Lock Based on Atomic Exchange

fqlock을 잡고 있다는 이야기입니다. Line 16이 여전히 `gp_flags` 전역 변수의 값이 0임을 확인하면, line 17에서 `gp_flags`를 1로 설정하고 line 18에서 `do_force_quiescent_state()`를 호출한 후, line 19에서 `gp_flags`를 0으로 되돌려 놓습니다. 어떤 경우였든, line 21에서는 root `rcu_node` 구조체의 \rightarrow fqlock을 해제합니다.

Quick Quiz 7.20: Figure 7.15의 코드는 황당할 정도로 복잡하군요! 왜 그냥 조건적으로 하나의 글로벌 락을 잡지 않나요? ■

Quick Quiz 7.21: 잠깐만요! Figure 7.15의 line 16에서 토너먼트에 “승리” 했다면, `do_force_quiescent_state()`의 모든 일을 해야만 하잖아요. 어떻게 이걸 승리라고 할 수 있어요, 진짜로? ■

이 함수는 드물지 않게 사용되는 계층적 락킹의 패턴을 보여주고 있습니다. 이 패턴은 앞서 이야기한 iterator 캡슐화와 마찬가지로 RAII 락킹을 사용해서 구현하기 상당히 어렵고, 따라서 예측가능한 미래를 위해서는 lock/unlock 기능들이 필요하게 될 것입니다.

7.3 Locking Implementation Issues

개발자들은 거의 항상 시스템에 의해서 제공되는, 예를 들면 POSIX pthread mutex lock [Ope97, But97]와 같은 락킹 도구들을 사용하도록 서비스 받을 것입니다. 더도 아니고 덜도 아니고, 일부 샘플의 구현을 알아보는 것은 도움이 될 수 있는데, 극단적인 워크로드와 환경에서 발생하는 구현상의 어려움 등을 알아볼 수 있을 것이기 때문입니다.

7.3.1 Sample Exclusive-Locking Implementation Based on Atomic Exchange

이 섹션은 Figure 7.16에 보인 구현을 알아봅니다. 이 락을 위한 데이터 구조체는 line 1에 보여졌듯이 그저 하나의 `int` 인데, 그 외에도 다른 정수형 타입이어도 상관없습니다. 이 락의 최초 값은 0으로, line 2에 보여졌듯이 “락이 걸리지 않았음”을 의미합니다.

Quick Quiz 7.22: 왜 C 언어에서 기본적으로 제공하는 0으로의 초기화 메커니즘을 사용하지 않고 Figure 7.16의 line 2에 보여진 것처럼 명시적으로 초기화를 하는 거죠? ■

Lock 획득은 line 4-9의 `xchg_lock()` 함수에 의해 수행됩니다. 이 함수는 중첩된 루프를 사용하는데, 바깥의 루프를 통해서는 반복적으로 어토믹하게 락의 값을 1 (“락이 걸렸음”을 의미)과 바꿉니다. 만약 이전의 값이 이미 값 1이었다면 (다시 말해, 누군가가 이미 락을 잡고 있다면), 안의 루프 (line 7-8)에서는 그 락이 풀려서 다시 잡으려 시도할 수 있을 때까지 루프를 돌면서 기다리는데, 기다림이 끝나면 바깥 루프에서 다시 한번 락을 잡기 위한 시도를 할 수 있게 됩니다.

Quick Quiz 7.23: 왜 Figure 7.16의 line 7-8의 안쪽 루프를 귀찮게 수행해야 하나요? 왜 그냥 간단하게 line 6에서 계속해서 어토믹한 값 교환 오퍼레이션을 수행하지 않는거죠? ■

락의 해제는 line 12-15의 `xchg_unlock()` 함수에 의해 이루어집니다. Line 14는 어토믹하게 락의 값을 0 (“락이 걸리지 않았음”)으로 교체해서 그 락이 풀려 있는 상태임을 알립니다.

Quick Quiz 7.24: Figure 7.16의 line 14에서는 왜 그냥 간단하게 락에 0 값을 저장하지 않는거죠? ■

이 락은 test-and-set 락 [SR84]의 한가지 간단한 예입니다만, 실제 상용 제품에서도 매우 비슷한 메커니즘이 순수한 스핀락을 위해 상당히 많이 사용됩니다.

7.3.2 Other Exclusive-Locking Implementations

어토믹 인스트럭션들을 활용해서 구현 가능한 수많은 락킹 방법들이 존재하는데, 그 중 많은 것들이 Mellor-Crummey 와 Scott에 의해 검토되었습니다 [MCS91]. 예를 들어, 앞의 섹션에서 보았던 어토믹 교환 오퍼레이션에 기반한 test-and-set 락은 경쟁 정도가 낮을 때 잘 동작하고 메모리 사용량이 적다는 장점을 갖습니다. 또한 당장 락을 받아도 사용할 수 없는 쓰레드에게 락을 넘겨주거나 하지 않습니다만, 그 결과로 비공정성 문제 가 생길 수 있으며 심지어는 높은 경쟁 수준 아래에서는 starvation 상황도 나타날 수 있습니다.

반면, 리눅스 커널에서 사용되는 티켓 락 [MCS91]의 경우에는 높은 경쟁 상황에서도 비공정성 문제를 없앱니다만, first-in-first-out 정책은 락을 받아도 당장 그 락을 사용할 수 없는 예를 들면 다른 쓰레드에게 pre-emption 당했거나 인터럽트 당했거나 또는 그 외에도 어떤 이유등이 있는 쓰레드에게 락을 넘겨줄 수 있습니다. 하지만, 이런 preemption과 인터럽트의 가능성에 대해 지나치게 걱정하지는 않는 것이 중요한데, 이 preemption과 인터럽트는 대부분의 경우 락이 획득된 뒤에나 일어날 수 있기 때문입니다.⁷

test-and-set 락과 티켓 락을 포함해서, 기다리는 쓰레드가 하나의 메모리 위치를 보며 루프를 도는 모든 락킹 구현은 높은 락 경쟁 수준에서는 성능에 문제를 겪게 됩니다. 문제는 락을 해제하는 쓰레드는 연관된 메모리 위치의 값을 업데이트 해야 한다는 점입니다. 경쟁 수준이 낮은 경우에는 이건 문제가 되지 않습니다: 연관된 캐시 라인은 여전히 락을 잡은 CPU의 캐시에 있고 곧바로 쓸 수 있는 상태일 확률이 큽니다. 반면, 경쟁 수준이 높은 경우에는, 락을 획득하려 하는 각 쓰레드는 해당 캐시 라인의 read-only 복사본을 가지고 있을 것이고, 락을 쥐고 있는 쓰레드는 그 락을 해제하는 업데이트를 진행하기에 앞서 모든 read-only 복사본들을 무효화 시켜야 합니다. 일반적으로, 더 많은 CPU들과 쓰레드들이 존재할수록 높은 경쟁 수준 아래에서 락을 해제하는데에 발생하는 오버헤드는 커집니다.

이런 잘못된 확장성은 여러개의 queued-lock 구현들 [And90, GT90, MCS91, WKS94, Cra93, MLH94, TS93]을 만드는 동기가 되었습니다. Queued lock은 높은 캐시 무효화 오버헤드를 각 쓰레드에게 대기열의 원소 하나를 할당하는 방식으로 제거합니다. 이런 대기열의 원소들은 서로 연결되어서 해당 락이 기다리는 쓰레드들에게 넘겨질 순서를 관장하는 하나의 대기열을 만들게 됩니다. 여기서 핵심은 각 쓰레드가 자신에게 할당된 대기열 원소를 바라보면서 루프를 돌기 때문에, 락을 잡고 있는 쓰레드는 다음 쓰레드의 CPU의 캐시에 있는 원소 하나만 무효화 시키면 된다는 것입니다. 이런 구조는 높은 수준의 락 경쟁상황에서 락을 넘기는 일의 오버헤드를 대폭 줄여줍니다.

좀 더 최근의 queued-lock 구현들은 시스템의 구조 또한 신경을 쓰는 방식으로 만들어져서 락을 좀 더 지역적으로 가까운 쪽에 우선적으로 넘겨주되 여전히 starvation을 막는 방법도 지킵니다 [SSVM02, RH03, RH02, JMRR02, MCM02]. 이런 것들 중 다수가 디스크 I/O의 스케줄링에 전통적으로 사용되었던 엘레베이터 알고리즘과 유사한 것으로 이해될 수 있습니다.

⁷ 한편, 높은 락 경쟁 정도를 처리하는 최선의 방법은 일단 그런 상황이 생기지 않게 하는 것입니다! 하지만, 높은 락 경쟁 상황이 생겨날 수 있는 악마들 가운데 떨 중요한 것일 뿐인 상황도 존재하며, 어떤 경우가 되었든, 높은 수준의 경쟁에 대처하는 방법을 연구해 보는 것은 심적으로 좋은 연습이 됩니다.

불행히도, 높은 경쟁 수준에서 queued lock의 효율성을 높여주는 스케줄링 로직은 낮은 경쟁 수준에서의 오버헤드도 증가시킬 수 있습니다. Beng-Hong Lim과 Anat Agarwal은 그래서 낮은 경쟁 수준에서는 간단한 test-and-set lock을 사용하고 높은 수준의 경쟁 상황에서는 queued lock을 사용하도록 락을 교환하는 방식으로 test-and-set 락과 queued lock과 결합시켜서 [LA94] 낮은 수준의 경쟁 상황에서는 낮은 오버헤드를 가지면서 높은 수준의 경쟁에서는 높은 처리량과 공정성을 가질 수 있도록 했습니다. Browning 등은 비슷한 접근을 취했지만, 분리된 플래그의 사용을 막아서 test-and-set의 빠른 수행 경로가 간단한 test-and-set 락에 사용되는 것과 동일한 인스트럭션들을 사용할 수 있도록 했습니다 [BMMM05]. 이런 접근법은 상용 제품들에 사용되었습니다.

높은 수준의 경쟁 상황에서 나타나는 또 다른 문제는 락을 잡고 있는 쓰레드의 수행이 지연되는 경우로, 특히 이 지연이 preemption에 의한 경우가 되는데, 이 경우에는 낮은 우선순위의 쓰레드가 락을 잡지만 중간 정도 우선순위의 CPU를 많이 사용하는 쓰레드에 의해 CPU 사용을 빼앗겨 높은 우선순위의 프로세스가 락을 잡을 수 있게 되기 전까지 기다려야 하는 우선순위 역전 문제를 초래하게 됩니다. 이로 인한 결과는 CPU를 많이 사용하는 중간 우선순위 프로세스가 높은 우선순위의 프로세스가 수행되는 것을 막고 있게 되는 현상입니다. 한 가지 해결책은 우선순위 상속 [LR80]으로, 이 방법에 대해 여전히 남아있는 논쟁 [Yod04a, Loc02]에도 불구하고 real-time 컴퓨팅에서 널리 사용된 방식입니다 [SRL90a, Cor06b].

우선순위 역전 문제를 없애기 위한 또 다른 방법은 락이 잡혀 있는 동안의 preemption을 막는 것입니다. 락이 잡혀 있는 동안의 preemption을 막는 것은 또한 처리량 역시 증가시키기 때문에, 대부분의 독점 UNIX 커널들은 어떤 형태의 스케줄러를 의식하는 동기화 메커니즘 [KWS97]을 제공하는데, 특정한 덩치 큰 데이터베이스 제조사들의 노력이 큰 이유입니다. 이런 메커니즘들은 보통 preemption이 적절치 못할 것이라는 정보를 힌트와 같은 형태로 받습니다. 이런 힌트들은 종종 특정한 기계의 레지스터의 특정 bit을 1로 만드는 형태로 주어지는데, 이 메커니즘을 위해 락 획득마다 추가되는 오버헤드를 매우 낮게 유지시켜줍니다. 반면에, 리눅스는 이런 힌트를 받지 않고, 대신 futex [FRK02, Mol06, Ros06, Dre11]라 불리는 메커니즘을 사용해 비슷한 결과를 만들어냅니다.

흥미롭게도, 락을 구현하는데 어토믹 인스트럭션들은 반드시 필요한 것만은 아닙니다 [Dij65, Lam74]. 간단한 읽기와 쓰기만을 가지고 락킹을 구현하는 과정을 둘러싼 문제들에 대한 훌륭한 소개를 Herlihy와 Shavit의 교재에서 찾을 수 있습니다 [HS08]. 여기서도 되돌이

되는 핵심 포인트는 그것들에 대한 조심스러운 연구는 재미도 있고 배우게 되는 점도 많겠지만, 그런 구현들에 대한 현재 실용적인 사용 예는 적다는 점입니다. 더도 아니고 덜도 아니고, 아래에 설명되는 예외를 포함해서, 그런 연구는 독자 여러분의 연습으로 남겨두겠습니다.

Gamsa 등 [GKAS99, Section 5.3] CPU들 사이를 순환하는 토큰을 사용한, 토큰에 기반한 메커니즘을 설명합니다. 이 토큰이 특정 CPU에 도달하면, 그 CPU는 그 토큰에 의해 보호되는 모든 것에 배타적인 접근 권한을 갖게 됩니다. 이 토큰 기반의 메커니즘을 구현하는데 사용될 수 있는 수많은 설계가 존재하는데, 예를 들면 다음과 같습니다:

1. CPU 별로 플래그를 가지고 있는데, 각 플래그는 하나의 CPU를 위한 것 제외하고는 모두 0으로 초기화되어 있습니다. 한 CPU의 플래그가 0이 아니라면, 그 CPU는 그 토큰을 쥐게 된 것입니다. 그 CPU가 그 토큰의 사용을 마무리하면, 자신의 플래그를 0으로 설정하고 다음 CPU의 플래그를 1로 (또는 어떤 다른 0이 아닌 값으로) 설정합니다.
2. CPU 별 카운터를 가지고 있는데, 각 카운터는 연관된 CPU의 숫자로 설정되는데, 그 숫자는 시스템의 전체 CPU의 갯수가 N 이라고 할 때 0부터 $N - 1$ 사이의 값을 갖게 됩니다. 한 CPU의 카운터가 다음 CPU의 것보다 높은 값을 갖는다면 (카운터가 오버플로우되어 값이 초기화되는 것을 고려해야 합니다) 첫번째 CPU가 그 토큰을 잡은 것입니다. 그 토큰을 사용하는게 끝났다면, 토큰을 사용했던 CPU는 다음 CPU의 카운터를 자신의 카운터보다 1 큰 값으로 설정합니다.

Quick Quiz 7.25: 카운터의 값이 오버플로우되어 초기화되는 경우를 고려하면서 어떻게 하나의 카운터가 다른 카운터보다 큰지를 판단할 수 있나요? ■

Quick Quiz 7.26: 카운터 사용과 플래그 사용 중 뭐가 더 나을까요? ■

이 락은 CPU가 락을 곧바로 얻을 수 없을 수도 있는데, 심지어 다른 CPU가 락을 당장 사용하고 있지 않다 하더라도 그렇다는 점에서 일반적이지 않습니다. 대신, CPU는 토큰이 자신에게 올 때까지 기다려야만 합니다. 이는 CPU들이 주기적으로 크리티컬 섹션에의 접근을 해야 하지만 토큰의 순환율이 변화되는 것을 받아들일 수 있는 경우에는 유용합니다. Gamsa 등 [GKAS99]은 이걸 사용해서 read-copy update (Section 9.5을 참고하세요)의 또 다른 변종을 구현했는데, 이는 또한 메모리 할당자들에 의해 사용되는 CPU 별 캐시 비우기 작업이나 CPU 별 데이터 구조체의 garbage collection, 또는 CPU 별 데이터를 공유된 저장 영역 (또는 대용량 저장장치라던지, 필요에 따라)으로 옮기는 작업과 같은,

```

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

주기적인 CPU 별 작업을 보호하는데에도 사용될 수 있습니다 [MS93].

갈수록 많은 사람들이 병렬 하드웨어에 친숙해지고 더 많은 코드를 병렬화 하는 것을 보아, 더 특수한 목적의 락킹 도구들이 나타날 것을 예상해 볼 수 있습니다. 더도 아니고 덜도 아니고, 독자 여러분은 이 중요한 안전성 팁을 조심스럽게 고려해야 합니다: 언제든 인간적으로 사용이 가능하다면 표준적인 동기화 도구들을 사용하세요. 자신을 위한 동기화 도구를 직접 만드는 것에 비해 표준 동기화 도구들을 사용하는 것의 커다란 장점은 표준 기능들은 일반적으로 버그가 훨씬 적다는 것입니다.⁸

7.4 Lock-Based Existence Guarantees

병렬 프로그래밍에서의 핵심적인 난제는 존재 보장 [GKAS99]을 제공하는 것으로, 어떤 객체에 접근하여 시도하는 것은 그 객체가 이 접근 시도 동안 존재하는지 여부에 의존적입니다. 어떤 경우에는 존재 보장이 명시적이지 않기도 합니다:

1. 기본 모듈의 글로벌 변수와 static 지역 변수들은 어플리케이션이 돌아가는 동안은 존재할 것입니다.
2. 로드된 모듈의 글로벌 변수들과 static 지역 변수들은 그 모듈이 로드된 상태를 유지하는 동안은 존재할 것입니다.

⁸ 그리고, 맞아요, 저는 적어도 저의 직접 만드는 동기화 도구에 대해서는 공유를 했어요. 하지만, 제 머리카락은 그런 일을 시작하기 전에 비해 훨씬 하얗게 세어버렸습니다. 우연의 일치일까요? 그렇지도 모르죠. 하지만, 독자 여러분은 정말로 머리카락이 빨리 하얗게 되는 리스크를 지고 싶으신가요?

3. 하나의 모듈은 자신의 함수들 중 최소한 하나는 실행중인 실행 흐름을 가지고 있되 동안은 로드된 상태를 유지할 것입니다.
4. 어떤 함수의 실행중인 흐름의 스택 위에 할당되는 변수들은 그 함수의 실행 흐름이 리턴을 하기 전까지는 존재할 것입니다.
5. 어떤 함수 위에서 코드를 수행 중이거나 (직접적 이든 간접적이든) 그 함수로부터 호출되었다면, 그 함수는 실행중인 흐름을 가지고 있는 것입니다.

비록 명시적이지 않은 존재 보장과 연관된 버그들이 실제로 만들어질 수 있긴 하지만 이런 명시적이지 않은 존재 보장은 직관적인 이야기입니다.

Quick Quiz 7.27: 어떻게 비명시적인 존재 보장에 의존하는게 버그를 만들어낼 수 있나요? ■

하지만 그보다 더 흥미로운—그리고 문제를 일으키는—보장은 heap 메모리에 연관된 것입니다: 동적으로 할당된 데이터 구조체는 그 메모리가 해제되기 전까지 존재할 것입니다. 여기서 해결해야 할 문제는 구조체에 대한 동시적인 접근들과 그 구조체가 존재하는 메모리의 해제를 동기화 시키는 것입니다. 이를 해결하는 방법 가운데 하나는 락킹과 같은 명시적 보장입니다. 만약 어떤 구조체가 어떤 주어진 락을 잡은 채로만 메모리 해제될 수 있다고 한다면, 그 락을 잡는 행위는 그 구조체의 존재를 보장합니다.

하지만 이 보장은 또다시 그 락 자체의 존재 여부에 의존적입니다. 이 락의 존재를 보장하기 위한 한가지 직선적인 방법은 그 락을 글로벌 변수로 위치시키는 것입니다만, 글로벌한 락킹은 제약된 확장성이라는 단점을 갖습니다. 데이터 구조체의 크기가 커질수록 향상되는 확장성을 제공하는 한가지 방법은 락을 구조체의 각 원소마다에 위치시키는 것입니다. 불행히도 데이터 원소 자체 내에 그 데이터 원소를 지키기 위한 락을 위치시키는 것은 Figure 7.17 미묘한 경쟁 상황을 일으키기 쉽습니다.

Quick Quiz 7.28: 우리가 삭제해야 하는 원소가 Figure 7.17 의 line 8 의 리스트의 첫번째 원소가 아니면 어떻게 하죠? ■

Quick Quiz 7.29: Figure 7.17에서 어떤 경쟁 조건이 발생할 수 있나요? ■

이 예제를 고치는 한가지 방법은 해싱을 사용한 글로벌 락들의 집합을 사용하는 것으로, Figure 7.18에 보여진 것처럼 각각의 해시 bucket이 자신의 락을 가지고 있는 형태입니다. 이 방법은 그 데이터 원소로의 포인터를 얻어오기 (line 10에서) 전에 올바른 락을 얻어올 수 있도록 합니다 (line 9에서). 이 접근법은 이 그림에 보여진 해시 테이블처럼 하나의 분할 가능한 데이터 구조체에 대해서는 잘 동작하지만 주어진 데이터 원소가 여러 해시 테이블의 멤버이거나 트리나 그래프와 같

```

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

은 보다 복잡한 데이터 구조체의 멤버라면 문제가 생길 수도 있습니다. 사실, 이런 문제들은 해결될 수 있는데, 그런 해결책들이 락에 기반한 소프트웨어 트랜잭션 메모리 구현의 기반을 이룹니다 [ST95, DSS06]. 하지만, Chapter 9에서 더 간단한—그리고 더 빠른—존재 보장을 제공하는 방법을 알아보겠습니다.

7.5 Locking: Hero or Villain?

실제 삶에서도 종종 그런 경우가 있듯, 락킹은 그것이 어떻게 사용되는지에 따라 그리고 당장 해결해야 할 문제가 무엇인지에 따라 영웅이 될 수도 있지만 악당이 될 수도 있습니다. 제 경험상으로는, 전체 어플리케이션을 작성하는 사람들은 락킹 사용에 있어 행복해하고, 병렬 라이브러리를 작성하는 사람들은 덜 행복해하며, 존재하는 순차적 방식의 라이브러리를 병렬화 하는 사람들은 매우 행복하지 못합니다. 다음의 섹션들은 이런 시점의 차이에 대한 이유 몇가지를 이야기해 봅니다.

7.5.1 Locking For Applications: Hero!

전체 어플리케이션 (또는 전체 커널)을 작성할 때, 개발자들은 동기화 설계를 포함해 전체 설계를 제어할 수 있습니다. 그 설계가 Chapter 6에서 이야기한 것처럼 패티셔닝을 잘 사용하고 있다면, 락킹은 극단적일 정도로 효과적인 동기화 메커니즘이 될 수 있는데, 상용 제품 품질의 병렬 소프트웨어에서의 많은 락킹의 사용이 이를 입증합니다.

그런 소프트웨어가 자신의 동기화 설계를 보통 락킹으로 사용한다고는 하지만, 더도 아니고 덜도 아니고, 그런 소프트웨어는 거의 항상 다른 동기화 메커니즘 또한 사용하는데, 여기에는 특별한 카운팅 알고리즘

(Chapter 5), 데이터 소유권 (Chapter 8), 레퍼런스 카운팅 (Section 9.2), 순서적 락킹 (Section 9.4), 그리고 read-copy update (Section 9.5) 등이 포함됩니다. 또한, 실제 코드를 짜는 사람들은 다양한 도구들을 사용하는데 이 도구들이 돋는 일은 테드락 감지 [Cor06a], 락 획득/해제 벨러싱 [Cor04b], 캐시 미스 분석 [The11], 그리고 하드웨어 카운터 기반의 프로파일링 [EGMdB11, The12], 그리고 그 외에도 여러가지입니다.

잘 짜여진 설계와 여러 동기화 메커니즘들의 좋은 조합, 훌륭한 도구의 사용 아래에서는 락킹은 어플리케이션과 커널을 위해선 상당히 잘 동작합니다.

7.5.2 Locking For Parallel Libraries: Just Another Tool

어플리케이션과 커널들의 경우와 달리, 라이브러리의 설계자는 라이브러리와 함께 동작하게 될 코드의 락킹 설계에 대해서 알 수가 없습니다. 사실, 그 코드는 몇년 이후에야 쓰여질 수도 있습니다. 따라서 라이브러리의 설계자는 마음대로 제어할 수 있는 부분이 더 적고 라이브러리의 동기화 설계를 만들 때 더 많은 주의를 기울여야만 합니다.

Deadlock은 물론 특별히 신경써야 하는 부분이고, 따라서 Section 7.1.1에서 언급한 기술들이 적용되어야 합니다. 유명한 deadlock 방지 방법 가운데 하나는 라이브러리의 락들은 라이브러리를 둘러싼 프로그램의 락킹 계층의 독립적인 부집합이라 보장을 하는 것입니다. 하지만, 이는 보이기보다 좀 더 어려울 수 있습니다.

그런 복잡한 문제들 중 하나는 Section 7.1.1.2에서 이야기 되었는데, `qsort()`의 비교 함수를 인자로 사용해서 라이브러리 함수가 어플리케이션 코드를 호출할 때가 이 경우가 될 수 있겠습니다. 또 다른 복잡한 문제는 시그널 핸들러와의 상호작용입니다. 어떤 어플리케이션의 시그널 핸들러가 라이브러리 함수 수행 중 받은 시그널에 의해 호출되는 경우, 그 라이브러리 함수가 그 시그널 핸들러를 직접 호출한다면 분명 deadlock이 발생할 수 있습니다. 마지막으로, 또하나의 복잡한 문제가 일어날 수 있는 경우는 `fork()/exec()` 사이에서 사용될 수 있는 라이브러리 함수인데, 예를 들면 `system()` 함수의 사용으로 인한 경우입니다. 이런 경우, 라이브러리 함수가 `fork()` 시점에 어떤 락을 잡고 있었다면, 자식 프로세스는 그 락을 잡은 채 수행을 시작합니다. 그런데 그 락을 놓아줄 쓰레드는 자식 프로세스가 아니라 부모 프로세스에서 돌아가고 있으므로, 만약 자식이 라이브러리 함수를 또다시 호출하게 되면 deadlock이 발생할 것입니다.

다음의 전략들은 이런 경우들에서 deadlock 문제를 해결하기 위해 사용될 수 있을 것입니다:

1. 콜백도 시그널도 사용하지 말 것.

2. 콜백이나 시그널 핸들러 안에서 락을 잡지 말 것.
3. 호출자가 동기화를 제어하도록 할 것.
4. 락킹을 호출자에게 넘길 수 있도록 라이브러리 API를 조정할 것.
5. 명시적으로 콜백 deadlock 을 없앨 것.
6. 명시적으로 시그널 핸들러 deadlock 을 없앨 것.

이 전략들 각각이 다음 섹션들에서 설명됩니다.

7.5.2.1 Use Neither Callbacks Nor Signals

라이브러리 함수가 콜백을 하지 않고 어플리케이션 전체가 시그널을 막는다면, 그 라이브러리 함수에 의해 잡히는 모든 락은 락킹 계층 tree 의 leaf 들이 될 겁니다. 이런 구성은 Section 7.1.1.1에서 이야기했듯 데드락을 막습니다. 비록 이 전략은 적용 가능한 곳에서는 매우 잘 동작하지만, 시그널 핸들러를 사용해야만 하는 어플리케이션들도 존재하게 마련이고, 콜백을 필요로 하는 라이브러리 함수들 (Section 7.1.1.2에서 이야기한 `qsort()` 함수와 같은) 도 일부 있게 마련입니다.

다음 섹션에서 이야기하는 전략이 이런 경우에 많이 사용될 수 있을 겁니다.

7.5.2.2 Avoid Locking in Callbacks and Signal Handlers

어떤 콜백도 시그널 핸들러도 락을 잡지 않는다면, 이 것들은 데드락 사이클에 포함될 수가 없는데, 이 사실은 다시 한번 일직선적인 락킹 계층에서 라이브러리 함수들을 락킹 계층 tree 의 leaf 로 만들어주는 방법을 고려해 보게 합니다. 이 전략은 콜백은 단순히 자신에게 넘겨진 두개의 값을 비교하기만 하는 `qsort` 의 사용에 있어 매우 잘 동작합니다. 이 전략은 또한 많은 시그널 핸들러들에 대해서도 놀라우리만큼 잘 동작하는데, 특히 시그널 핸들러 안에서 락을 잡는 행위는 일반적으로 좋지 않은 일이라는 사실 [Gro01]⁹ 을 놓고 보면 더욱 그렇습니다만 어플리케이션이 시그널 핸들러 안에서 복잡한 데이터 구조체를 조작해야 할 필요가 있다면 잘 동작하지 못할 수도 있습니다.

복잡한 데이터 구조체를 조작해야 하는 상황에서도 시그널 핸들러에서 락을 잡는 향위를 막을 수 있는 몇 가지 방법은 다음과 같습니다:

1. Section 14.3.1에서 이야기된 non-blocking 동기화를 사용하는 간단한 데이터 구조체들을 사용하세요.

⁹ 하지만 표준의 말들은 똑똑한 코더들이 어토믹 오퍼레이션을 가지고 자신만의 락킹 도구들을 만드는 걸 막지는 못합니다.

2. 데이터 구조체가 non-blocking 동기화를 합리적으로 사용하기 어려울 정도로 너무 복잡하다면, non-blocking 한 집어넣기 오퍼레이션을 제공하는 대기열을 하나 만드세요. 시그널 핸들러에서는 그 복잡한 데이터 구조체를 직접 제어하는 대신, 필요한 변경사항을 의미하는 원소를 이 대기열에 집어넣으세요. 별도의 쓰레드는 이 큐에서 원소를 꺼내서 일반적인 락킹을 사용해서 그 요청된 변경을 대신 해줍니다. 이미 사용 가능한 동시적 대기열의 구현들이 여럿 있습니다 [KLP12, Des09, MS96].

이 전략은 임시의 매뉴얼이나 콜백들과 시그널 핸들러들의 자동화된 검사기능 (이편이 더 좋습니다) 아래에서만 사용되어야 합니다. 이런 검사기능을 수행할 때에는 현명한 코더들이 (지혜롭지 못하게도) 만들어낸, 어토믹 오퍼레이션을 사용해 만들어진 그들만의 락을 사용할 수도 있다는 점에 조심해야만 합니다.

7.5.2.3 Caller Controls Synchronization

호출자가 동기화를 제어하게 합시다. 이 방법은 라이브러리 함수들이 각각 호출자가 볼 수 있는 데이터 구조체 인스턴스들 가운데 독립적이어서 각자 개별적으로 동기화가 될 수 있는 것들을 가지고 동작할 때 매우 잘 동작합니다. 예를 들어, 라이브러리 함수가 탐색 tree 를 가지고 동작한다면, 그리고 어플리케이션은 여러개의 독립적인 탐색 tree 들을 가지고 있어야 한다면, 어플리케이션은 각 tree 별로 락을 두어 관리할 수 있을 겁니다. 이렇게 되면 어플리케이션은 필요한대로 락을 잡고 풀어줘서 라이브러리가 병렬성에 대해 전혀 신경쓰지 않아도 되도록 할 수 있을 겁니다. 대신에 Section 7.5.1에서 이야기 했듯 락킹이 매우 잘 동작하도록 어플리케이션이 병렬성을 제어하게 되는 셈입니다.

하지만, 이 전략은 라이브러리가 내부적인 동시성을 필요로 하는 데이터 구조체를 구현하고 있다면 실패할 수 있는데, 예를 들면 해시 테이블이나 병렬적 정렬입니다. 이런 경우에, 라이브러리는 반드시 자신의 동기화 제어를 해야만 합니다.

7.5.2.4 Parameterize Library Synchronization

여기서의 아이디어는 라이브러리의 API에 인자를 추가해서 어떤 락들을 잡아야 하는지, 또는 어떻게 잡아야 하며 어떻게 해제해야 하는지, 또는 둘 다를 알릴 수 있게 하자는 겁니다. 이 전략은 어플리케이션이 (락들로의 포인터를 넘겨줌으로써) 어떤 락을 잡아야 하며 어떻게 그것들을 잡아야 하는지 (락 획득과 해제 작업을 하는 함수로의 포인터를 넘김으로써) 알림으로써 데드락을 방지하기 위한 전체 작업을 하게 됩니다만, 라이브러리 함수는 락들을 어디서 잡고 해제할지 직접 결정함으로써 자신의 동시성을 제어할 수 있도록 합니다.

더 구체적으로는, 이 전략은 락 획득과 해제 함수들이 필요에 따라 시그널들을 발생되지 않도록 막음으로써 라이브러리 코드는 어떤 락들에 대해 어떤 시그널들이 막혀야 하는지 알 필요가 없게 해줍니다. 이 전략에 의해 주의해야 할 부분들이 분리되는 효과는 상당히 효과적입니다만, 어떤 경우에는 뒤의 섹션들에서 설명할 전략이 더 잘 동작할 수 있습니다.

그렇다고는 하나, Section 7.1.1.4에서 이야기 되었듯, 락으로의 명시적인 포인터를 외부의 API에 넘겨주는 것은 매우 조심히 숙고되어야 합니다. 이 방법이 가끔은 해야 마땅한 옳은 일이긴 하지만, 다른 대안의 설계는 없는지 먼저 알아보아야만 합니다.

7.5.2.5 Explicitly Avoid Callback Deadlocks

이 전략 아래의 기본적인 규칙은 Section 7.1.1.2에서 이야기된 바 있습니다: “모르는 코드를 실행하기 전에 모든 락을 놓기.” 이 방법은 어플리케이션이 라이브러리의 락킹 계층에 대해 몰라도 되게 해주기 때문에 일반적으로 최선의 방법입니다: 라이브러리는 어플리케이션의 전체 락킹 계층에 대해 leaf 또는 고립된 subtree로 남아 있습니다.

모르는 코드를 실행하기 전에 모든 락을 놓기가 불가능한 경우에는 Section 7.1.1.3에서 이야기한 layer를 갖는 락킹 계층들이 잘 동작할 수 있습니다. 예를 들어, 모르는 코드가 시그널 핸들러라면, 이는 해당 라이브러리 함수가 모든 락 획득을 가로질러 시그널을 블록하게 될 것임을 의미하는데, 이는 매우 느리고 복잡한 일이 될 것입니다. 따라서, 시그널 핸들러가 (아마도 지혜롭지 못하게도) 락을 획득하는 경우라면 다음 섹션에서 이야기하는 전략이 도움이 될 수 있을 겁니다.

7.5.2.6 Explicitly Avoid Signal-Handler Deadlocks

시그널 핸들러 데드락은 다음과 같이 명시적으로 예방될 수 있습니다:

1. 어플리케이션이 라이브러리 함수를 시그널 핸들러에서 실행한다면, 그 시그널은 시그널 핸들러 밖에서 해당 라이브러리 함수가 호출될 때마다 블락되어야만 합니다.
2. 어플리케이션이 어떤 시그널 핸들러 안에서 잡은 락을 쥐고 있는 채로 라이브러리 함수를 실행한다면, 그 시그널은 그 라이브러리 함수가 시그널 핸들러 밖에서 호출될 때마다 블락되어야 합니다.

이런 규칙들은 리눅스 커널의 lockdep 락 의존성 체크 도구 [Cor06a]와 비슷한 도구들을 사용해서 강제될 수 있습니다. Lockdep의 강력한 점 중 하나는 사람의 직관에 의해 잘못된 실수를 저지르지 않는다는 점입니다 [Ros11].

7.5.2.7 Library Functions Used Between fork() and exec()

앞에서 이야기 했듯이, 라이브러리 함수를 수행하는 쓰레드가 또 다른 어떤 쓰레드가 `fork()`를 수행할 때 락을 쥐고 있었다면, 부모 프로세스의 메모리가 자식 프로세스를 만들기 위해 복사된다는 사실은 이 락이 자식 프로세스의 컨텍스트에는 잡힌 채로 자식 프로세스가 태어나게 됨을 의미합니다. 이 락을 해제하는 쓰레드는 부모 프로세스에서 동작하고 있지 자식 프로세스에서 동작하고 있지 않으므로, 자식 프로세스의 이 락의 복사본은 결코 해제되지 않을 것입니다. 따라서, 자식 프로세스 쪽에서 같은 라이브러리 함수를 실행하려는 시도는 모두 `deadlock`을 초래하고 말 것입니다.

이 문제를 해결하는 한가지 방법은 이 라이브러리 함수가 그 락의 소유자가 여전히 수행중인지 보고, 만약 그렇지 않다면 그 락을 다시 초기화시킴으로써 “깨버리고” 다시 잡는 것입니다. 하지만, 이 방법에는 두개의 취약점이 존재합니다:

1. 그 락에 의해 보호되던 데이터 구조체들은 어떤 중간 상태에 빠져 있어서 낙관적으로 락을 깨는 행위는 임의의 메모리 오염을 초래할 수 있습니다.
2. 자식 프로세스가 추가적인 쓰레드들을 생성하면, 두 쓰레드들은 그 락을 동시에 깰 수 있고, 이렇게 되면 두 쓰레드 모두 자신이 락을 잡고 있다고 생각하게 될 것입니다. 이는 또다시 임의의 메모리 오염 상황을 초래할 수 있습니다.

이런 상황을 처리하는 걸 돋기 위해 `atfork()` 함수가 존재합니다. 아이디어는 세개의 함수들을 등록해 두는데, 이 함수들 중 하나는 `fork()` 전에 부모 프로세스에 의해서 호출되고, 또 하나는 `fork()` 후에 부모 프로세스에 의해서 호출되고, 마지막 하나는 `fork()` 후에 자식 프로세스에 의해 호출됩니다. 이런 상황에서는 이 세개의 시점에서 적절한 처리를 함으로써 이 상황을 해결할 수 있습니다.

하지만, `atfork()` 핸들러들의 코딩은 일반적으로 상당히 섬세하게 되어야 함에 주의하시기 바랍니다. `atfork()` 방식이 가장 잘 동작할 수 있는 경우들은 사용되는 데이터 구조체들이 자식 프로세스에 의해 간단히 다시 초기화 될 수 있는 경우들입니다.

7.5.2.8 Parallel Libraries: Discussion

어떤 전략을 사용하는가에 관계없이, 라이브러리의 API는 사용하는 전략과 호출자가 그 전략과 어떻게 상호작용해야 하는지에 대해 분명한 설명을 가지고 있어야만 합니다. 요약하자면, 락킹을 사용해서 병렬 라이브러리를 구성하는 것은 가능한 일이긴 합니다만, 병렬 어플리케이션을 구성하는 것만큼 쉽지는 않습니다.

7.5.3 Locking For Parallelizing Sequential Libraries: Villain!

이미 접할 수 있는 낯은 가격의 멀티코어 시스템들이 늘어남에 따라, 싱글 쓰레드 기반만을 상정하고 설계된 기존의 라이브러리들을 병렬화 하는게 일반적인 작업이 되었습니다. 병렬성에 무관심하게 이루어지는 이런 모든 훈한 작업들은 병렬 프로그래밍 관점에서는 심각한 결점을 포함하는 라이브러리 API를 만들어지게 할 수 있습니다. 결점이 될 수 있는 것들은 다음과 같습니다:

1. 명시적이지 않은 파티셔닝의 금지.
2. 락킹을 필요로 하는 콜백 함수.
3. 객체 지향 스파게티 코드.

이런 결점들과 락킹에 대한 결론을 다음의 섹션들에서 이야기 해보겠습니다.

7.5.3.1 Partitioning Prohibited

싱글 쓰레드로 해시 테이블을 구현하고 있다고 생각해 봅시다. 해시 테이블의 모든 아이템의 정확한 갯수를 유지하는 것은 쉽고 빠르게 할 수 있을 것이고, 또한 이 정확한 갯수를 매 원소 추가와 삭제 작업때마다 가져다 주는 것도 쉽고 빠르게 할 수 있을 것입니다. 그런데 왜 병렬화가 어려울까요?

한가지 이유는 정확한 카운터는 Chapter 5에서 봤듯이 멀티코어 시스템에서는 성능도 확장성도 그렇게 좋지 못하다는 점입니다. 결국, 이 해시 테이블의 병렬화된 구현은 성능도 확장성도 그다지 좋지 않을 것입니다.

이 문제를 어떻게 하면 좋을까요? 한가지 방법은 Chapter 5에서 이야기한 알고리즘들 가운데 하나를 사용해서 대략적인 수치를 주는 것입니다. 또 다른 방법은 원소 갯수 카운트 기능을 버려버리는 겁니다.

어떤 방법을 사용하든, 해시테이블에 원소를 추가하거나 삭제할 때마다 정확한 원소 갯수를 알아야 하는 이유를 알기 위해 해시 테이블의 사용 패턴을 잘 알아볼 필요가 있을 겁니다. 여기 몇가지 가능한 경우를 열거해 보면:

1. 해시 테이블의 크기를 재조정 해야할 때를 결정하기 위해. 이 경우, 대략적인 카운트도 상당히 잘 동작할 것입니다. 또한 크기 재조정 작업을 가장 긴 체인의 길이에 따라서 시작하도록 하는 것도 효율적일텐데, 이는 체인 별로 잘 파티셔닝 된 형태로 계산되고 관리될 수 있을 것입니다.

2. 전체 해시 테이블을 돌아다니는데 필요한 시간을 계산하기 위해서. 대략적인 카운트는 이 경우에도 역시 잘 동작할 것입니다.
3. 예를 들면 해시 테이블로 집어넣고 해시테이블로부터 빼내는 항목이 없어진 경우의 확인 등을 위한 검사의 목적을 위해서. 이 경우는 분명히 정확한 카운트를 필요로 합니다. 하지만, 이런 사용은 기본적으로 검사를 위한 것이기 때문에 해시 체인들의 길이를 유지하면서 가끔 가다 원소 추가와 삭제 작업을 막아 두고 그 합을 구하는 것만으로도 충분 할 것입니다.

이제 병렬 라이브러리 API 자체에 존재하는 성능과 확장성에 대한 일부 제약에 대한 강한 이론적 토대가 존재함이 밝혀졌습니다 [AGH⁺11a, AGH⁺11b, McK11b]. 병렬 라이브러리를 설계하는 사람들은 모두 그런 제약에 대해 깊이 알아볼 필요가 있습니다.

락킹을 동시성에 친화적이지 않은 API 라 문제라고 비난하는 것은 너무 쉬운 일이지만 그렇게 하는게 현실적으로 큰 도움이 되지는 않습니다. 반면에, (대략) 1985년에 그 설계 선택을 내려야만 했던 불운한 개발자에게 공감을 하는 소수의 선택을 할 수도 있을 것입니다. 그 당시에는 병렬성의 필요성을 예전하는 개발자는 매우 드물고 용기있는 개발자였을 것이고, 심지어 실제로 좋은 병렬성 친화적인 API를 만드는데에는 용기와 행운의 조합이 필요했을 것입니다.

시간은 흐르게 마련이고, 코드도 그와 더불어 변화해 갑니다. 그렇다고는 하나, 유명한 라이브러리라면 수많은 사용자가 존재할 수도 있고, 그런 경우에 호환성을 잃는 변경을 그 라이브러리의 API에 가하는 것은 매우 명청한 짓일 겁니다. 이미 존재하고 매우 많이 사용되고 있는 순차적으로만 동작하는 API를 보완하기 위한 병렬성 친화적인 API를 추가하는 것이 이런 상황에서는 최선의 선택일 겁니다.

더도 아니고 덜도 아니고, 인간사가 그렇듯이, 우리의 불운한 개발자는 자신의 (이해할 수 있기는 하지만) 잘못된 API 설계 선택보다는 락킹 자체를 비난할 가능성이 더 클 것임을 예상할 수 있습니다.

7.5.3.2 Deadlock-Prone Callbacks

Section 7.1.1.2, 7.1.1.3, 그리고 7.5.2에서는 아무런 고려 없이 사용되는 콜백이 어떻게 락킹의 비극을 초래할 수 있는지를 알아보았습니다. 이 섹션들은 또한 이런 문제를 없애기 위해 라이브러리 함수를 어떻게 설계해야 하는지에 대해서도 설명했습니다만, 병렬 프로그래밍에 대한 경험이 없는 1990년대의 프로그래머가 그런 설계 원칙을 따랐기를 기대하는 것은 비현실적입니다. 따라서, 이미 존재하는 콜백을 많이 사용하는 싱글 쓰

레드 기반 라이브러리를 병렬화 하려는 사람은 락킹의 악당들을 마주칠 가능성이 큽니다.

콜백을 많이 사용하는 라이브러리가 이미 여러 곳에서 사용되고 있다면, 앞 섹션에서 이야기 했듯 이미 존재하는 사용자들이 그들의 코드를 점진적으로 바꿔갈 수 있도록 라이브러리에 병렬성 친화적인 API를 추가하는 것이 현명할 것입니다. 대안적으로, 어떤 사람들은 이런 경우에 트랜잭션 메모리의 사용을 추천합니다. 아직 독자 여러분께 트랜잭션 메모리에 대해 설명드리지 않았지만, Section 17.2에서는 그 강점과 약점에 대해 논의해 봅니다. 하드웨어 트랜잭션 메모리 (Section 17.3에서 논의됩니다)는 실제 하드웨어 트랜잭션 메모리 구현이 일의 진행을 보장하지 않는다면 (대부분의 구현이 그런 보장을 하지 않습니다) 이 경우에 도움이 되지 않음을 알아둘 필요가 있습니다. 상당히 실용적인 것으로 나타나는 (과대광고 되지 않았다면) 다른 대안으로는 Sections 7.1.1.5, 7.1.1.6, 그리고 Chapter 8와 9에서 논의된 방법들이 있습니다.

7.5.3.3 Object-Oriented Spaghetti Code

객체 지향 프로그래밍은 1980년대 또는 1990년대의 언젠가에 주류가 되었고, 그 결과로 상용 제품에는 수많은 양의 객체 지향 코드가 존재하게 되었고, 대부분이 싱글 쓰레드 기반입니다. 객체 지향성은 가치있는 소프트웨어 테크닉이 될 수 있지만, 생각 없이 사용된 객체들은 객체 지향적 스파게티 코드를 만들어 낼 수 있습니다. 객체 지향 스파게티 코드에서, 실행 흐름 제어는 필수적으로 무작위한 형태로 객체와 객체 사이에서 흘러다니게 되어서 코드를 더더욱 이해하기 어렵게 만들고 락킹 계층을 수용하기가 불가능하게 될 수도 있습니다.

그런 코드는 어떤 경우에든 정리되어야 한다고 주장하는 사람도 많겠지만, 실제로 그렇게 정리를 하는 일은 말하는 것보다 훨씬 어렵습니다. 그런 코드를 병렬화하는 작업을 맡게 되었다면, Section 7.1.1.5, 와 7.1.1.6, 그리고 Chapter 8와 9에서 논의되고 설명된 기법들을 사용함으로써 락킹을 원망할 기회를 줄일 수 있습니다. 이런 상황은 트랜잭션 메모리에 대한 아이디어를 떠올리게 한 사용 예인 것으로 여겨지며, 따라서 한번 시도해 보는 것도 좋을 것입니다. 그렇다고는 하나, 동기화 메커니즘의 선택은 Chapter 3에서 이야기된 하드웨어의 특성에 따라서 만들어져야만 합니다. 무엇보다, 동기화 메커니즘의 오버헤드가 보호되는 오퍼레이션들의 오버헤드보다 열배 이상 높다면, 그 결과는 그다지 보기 좋지 않을 겁니다.

그리고 그런 것들이 이런 상황에서 이런 질문을 해보는게 가치있게 할겁니다: 코드가 순차적인 형태로 남아 있어야 하는가? 예를 들어, 병렬성은 쓰레드 수준이 아니라 프로세스 수준에서 사용될 수도 있습니다. 일반적으로, 특정 작업이 극단적으로 어려움이 증명되었다면,

가끔은 그 특정한 작업을 해결하기 위한 대안들만이 아니라 그 작업을 해야하게 만드는 문제 자체를 해결하는 대안적 방법들도 생각해 볼 필요가 있습니다.

7.6 Summary

락킹은 아마도 가장 많이 사용되고 일반적으로 가장 유용한 동기화 도구입니다. 하지만, 락킹은 어플리케이션이나 라이브러리에 시작할 때부터 고려되어서 설계되었을 때 가장 잘 동작합니다. 기존에 존재하던 싱글 쓰레드 기반의 거대한 코드를 하루만에 병렬적으로 동작하도록 만들어야 한다면, 락킹은 병렬 프로그래밍 도구상자의 유일한 도구여서는 안될 겁니다. 다음의 몇개 챕터들에서는 다른 도구들을 알아보고, 그것들이 락킹과 또 다른 것들과 함께 어떻게 사용되는게 최선인지에 대해서 알아봅니다.

Chapter 8

Data Ownership

락킹에 따라오는 동기화 오버헤드를 없애는 가장 간단한 방법들 중 하나는 데이터를 쓰레드들 사이로 (또는, 커널의 경우라면, CPU 들 사이로) 포장을 해서 데이터의 한 조각은 하나의 쓰레드에 의해서만 접근되고 수정되도록 하는 것입니다. 흥미롭게도, 데이터 소유권은 병렬 설계 기법의 “큰 세가지 전략”을 모두 충족합니다: 이 기법은 쓰레드별로 (또는, 경우에 따라서는 CPU 들) 데이터를 쪼개고, 모든 지역적 오퍼레이션들을 몰아서 처리하고, 동기화 오퍼레이션의 제거는 그 극단적인 논리에 의해 발생합니다. 따라서 데이터 소유권이 굉장히 많은 곳에서 사용된다는 점은 놀랄 만한 일이 아니고, 실제로 초심자들도 거의 본능적으로 사용하는 사용 패턴입니다. 사실, 그 사용처는 너무 많아서 이 챕터는 새로운 예제를 소개하지는 않을 것이고, 대신 앞의 챕터들에서 보았던 예제들을 참조하겠습니다.

Quick Quiz 8.1: 어떤 형태의 데이터 소유권이 C 나 C++ 를 사용해서 (예를 들어, pthreads 를 사용해서) 공유 메모리 병렬 프로그램을 만들 때 방지하기가 극단적으로 어려울까요? ■

데이터 소유권을 적용하는 다양한 방법이 존재합니다. Section 8.1 에서는 각 쓰레드가 자신의 개인적 주소 공간을 갖는, 데이터 소유권의 논리적 극단을 보여드립니다. Section 8.2 에서는 반대의 극단을 소개하는데, 데이터가 공유되지만 다른 쓰레드들은 데이터에의 다른 접근 권한을 갖는 경우입니다. Section 8.3 에서는 함수 전달을 설명하는데, 다른 쓰레드들이 특정 쓰레드가 소유하는 데이터에 간접적인 액세스를 하도록 허용하는 방법입니다. Section 8.4 에서는 어떻게 특정 쓰레드들이 특정 함수와 그에 연관된 데이터들의 소유권을 할당받을 수 있는지 설명합니다. Section 8.5 에서는 공유 데이터를 사용하는 알고리즘을 데이터 소유권을 사용하도록 변환함으로써 성능을 개선시키는 것에 대해 논의해 봅니다. 마지막으로, Section 8.6 에서는 데이터 소유권을 일등시민으로 두는 몇 가지 소프트웨어 환경을 나열해 봅니다.

It is mine, I tell you. My own. My precious. Yes, my precious.

*Gollum in “The Fellowship of the Ring”,
J.R.R. Tolkien*

8.1 Multiple Processes

Section 4.1 에서 다음의 예를 소개한 바 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
```

이 예는 compute_it 프로그램의 두개의 인스턴스를 메모리를 공유하지 않는 별개의 프로세스들을 통해 병렬로 수행합니다. 따라서, 해당 프로세스의 모든 데이터는 그 프로세스에 소유되어 있어서, 위 예제의 모든 데이터는 소유되어 있습니다. 이 방법은 거의 모든 동기화 오버헤드를 제거합니다. 결과로 만들어지는 극단적인 단순함과 최적의 성능의 조합은 상당히 매력적입니다.

Quick Quiz 8.2: Section 8.1 에서 보인 예제에 남아 있는 동기화 작업은 무엇이 있을까요? ■

Quick Quiz 8.3: Section 8.1 에서 보여진 예제에 어떤 공유된 데이터가 있나요? ■

sh 에서 했던 것과 똑같은 패턴을 C 로도 구현할 수 있는데, Figures 4.2 and 4.3 에 그 구현이 그려져 있습니다.

다음 섹션은 공유 메모리 병렬 프로그램에서의 데이터 소유권의 사용에 대해 논의해 봅니다.

8.2 Partial Data Ownership and pthreads

Chapter 5 는 데이터 소유권을 많이 사용하고 있습니다만, 한가지 새로운 방식을 가지고 있습니다. 쓰레드들은 다른 쓰레드들에 의해 소유되는 데이터를 수정할 수 없도록 되어있지만, 그것들을 읽을 수는 있습니다. 한마디로, 여기서의 공유메모리 사용은 소유권과 접근권한들에의 더 많은, 묘한 개념들을 허용합니다.

예를 들어, page 45 의 Figure 5.9 에 있는 쓰레드별 통계적 카운터 구현을 생각해 봅시다. 여기서, `inc_count()` 는 연관된 쓰레드의 `counter` 인스턴스를 수정할 뿐입니다만, `read_count()` 는 모든 쓰레드의 `counter` 인스턴스들을 접근하되 수정하지는 않습니다.

Quick Quiz 8.4: 각각의 쓰레드가 자신의 쓰레드별 변수만을 읽을 수 있지만 다른 쓰레드의 인스턴스들에는 쓰기를 할 수도 있는 부분적 데이터 소유권도 말이 될까요? ■

순수한 데이터 소유권은 일반적이고 할 뿐더러 유용한데, page 83 부터 시작되는 Section 6.4.3 에서 이야기된 쓰레드별 메모리 할당자 캐시가 한 예입니다. 이 알고리즘에서, 각 쓰레드의 캐시는 그 쓰레드에게 완전히 소유되어 있습니다.

8.3 Function Shipping

앞의 섹션에서는 쓰레드들이 다른 쓰레드들의 데이터에 접근할 수 있는 약한 형태의 데이터 소유권에 대해 이야기했습니다. 이런 형태는 함수에 필요한 데이터를 가져다 주는 것으로 생각될 수 있습니다. 대안적인 접근법은 데이터에 함수를 가져다 주는 것입니다.

그런 접근법이 page 56 에서 시작하는 Section 5.4.3 중 page 57 의 Figure 5.24 에 있는 `flush_local_count_sig()` 와 `flush_local_count()` 함수로 그려져 있습니다.

`flush_local_count_sig()` 함수는 전달되는 함수처럼 동작하는 시그널 핸들러입니다. `flush_local_count()` 의 `pthread_kill()` 함수는 시그널을 보내고—함수를 전달하는 행위—전달된 함수가 실행되기를 기다립니다. 이 전달된 함수는 일반적이지 않은, 동시적으로 수행되는 `add_count()` 나 `sub_count()` 함수들과의 상호작용의 필요로 인해 추가된 복잡성을 갖습니다 (page 58 의 Figure 5.25 와 page 58 의 Figure 5.26 를 참고하세요).

Quick Quiz 8.5: 함수를 전달하는데 POSIX 시그널 외에 어떤 다른 메커니즘이 사용될 수 있을까요? ■

8.4 Designated Thread

앞의 섹션들에서는 각 쓰레드가 데이터 복사본을 또는 데이터의 특정 부분을 소유하는 여러 가지 방법들을 설명했습니다. 반대로, 이 섹션에서는 특별한 선택된 쓰레드가 프로그램이 일을 하는데 필요한 데이터를 소유하는 함수적 분해 접근법을 설명합니다. Section 5.2.3 에서 설명했던 결과적으로 일관적인 카운터 구현이 예를 제공합니다. 이 구현은 Figure 5.8 의 line 15-32에서

보이는 `eventual()` 함수를 수행하는 정해진 쓰레드를 가지고 있습니다. 이 `eventual()` 쓰레드는 주기적으로 쓰레드별 카운트를 글로벌 카운터로 가져와서 글로벌 카운터로의 접근이 이름이 말하는 바와 같이 결과적으로는 실제 값에 수렴하게 합니다.

Quick Quiz 8.6: 하지만 Figure 5.8 의 line 15-32에 보여진 `eventual()` 함수의 어떤 데이터로 실제로는 `eventual()` 쓰레드에게 소유되어 있지 않습니다! 이게 어떻게 데이터 소유권인가요??? ■

8.5 Privatization

공유 메모리 병렬 프로그램의 성능과 확장성을 개선하는 한가지 방법은 공유된 데이터를 특정 쓰레드에 의해 소유된 사유 데이터로 변환하는 것입니다.

이런 방법의 좋은 예가 Section 6.1.1 의 Quick Quiz 중 하나의 답변에 있는데, 식사하는 철학자 문제를 데이터 사유화를 통해 교재의 표준적인 해결책보다 훨씬 좋은 성능과 확장성으로 해결하는 해결책입니다. 원래의 문제는 다섯명의 철학자들이 테이블에 앉아있고 인접한 두 철학자 사이에 하나의 포크만이 놓여 있어서 최대 두명의 철학자들만이 동시에 식사를 할 수 있게 되어 있었습니다.

우린 여기에 다섯개의 포크를 추가함으로써 각각의 철학자가 그 또는 그녀 자신만의 포크 한쌍을 가질 수 있게 하는것으로 간단히 문제에 사유화를 적용할 수 있습니다. 이 방법은 모든 다섯명의 철학자들이 동시에 식사를 할 수 있도록 하고, 또한 그런 종류의 문제의 확산을 주목할만하게 감소시킬 수 있습니다.

다른 경우에는, 사유화는 비용을 의미합니다. 예를 들어, page 48 의 Figure 5.12 에 나온 간단한 리미트 카운터를 생각해 봅시다. 이는 쓰레드들이 서로의 데이터를 읽을 수 있지만 자신의 데이터만 업데이트 할 수 있도록 허용되는 알고리즘의 한가지 예입니다. 해당 알고리즘을 간단히 살펴본 결과 쓰레드간의 접근은 `read_count()` 의 합산을 위한 루프 뿐임을 알 수 있습니다. 이 루프가 사라진다면, 더 효과적인 순수한 데이터 소유권으로 옮겨갈 수 있습니다만, `read_count()` 의 덜 정확한 결과값을 비용으로 지불하게 될겁니다.

Quick Quiz 8.7: 여전히 쓰레드별 데이터의 완전한 사유화를 유지하면서 좋은 정확도를 얻을 수도 있을까요? ■

요약하자면, 사유화는 병렬 프로그래머의 도구상자에 있는 강력한 도구입니다만 충분한 고려 없이 사용되어선 안됩니다. 모든 다른 동기화 도구들처럼, 이 방법 역시 성능과 확장성을 떨어뜨리고 복잡도를 높일 수 있는 잠재적 가능성이 존재합니다.

8.6 Other Uses of Data Ownership

데이터 소유권은 데이터가 쓰레드간 액세스나 업데이트의 필요가 적거나 없을 수 있도록 분할 될 수 있을 때에 가장 잘 동작합니다. 다행히도, 이런 상황은 합당하게도 흔하고, 다양한 병렬 프로그래밍 환경에 존재합니다.

데이터 소유권의 예에는 다음의 것들도 포함됩니다:

1. MPI [MPI08] 와 BOINC [UoC08] 같은 모든 메세지 전달 환경.
2. Map-reduce [Jac08].
3. RPC, 웹 서비스, 그리고 백엔드 데이터베이스 서버를 갖는 수많은 시스템들과 같은 클라이언트-서버 시스템들.
4. 아무것도 공유하지 않는 데이터베이스 시스템들.
5. 프로세스별로 분리된 주소공간을 갖는 Fork-join 시스템들.
6. Erlang 언어와 같은 프로세스 기반 병렬성.
7. 쓰레드 환경에서 C 언어의 스택 위에 할당되는 auto 변수들과 같은 사유 변수들.

데이터 소유권은 아마도 존재하는 것들 중 가장 감사받지 못하고 있는 동기화 메커니즘이입니다. 제대로 사용된다면, 이것은 적수를 찾기 어려울 만큼의 간단함과 성능, 그리고 확장성을 제공합니다. 아마도 이것의 간단함이 이것이 누리기 마땅한 존중을 받게 합니다. 바라건대 데이터 소유권의 섬세함과 힘에 대한 감사함이 늘어나는 것은 더 많은 존중을 이끌어내어서 더 훌륭한 성능과 확장성에 줄어든 복잡성을 이끌어낼 수 있게 할 것입니다.

Chapter 9

Deferred Processing

일을 뒤로 미루는 전략은 기록된 역사의 시작 전까지 이루어져 있습니다. 이것은 자주 미루기나 완전한 게으름으로 여겨져 비웃음을 받아왔습니다. 하지만, 지난 수십년 간 사람들은 병렬 알고리즘들의 단순화와 능률화에 있어서의 이 전략의 가치를 깨달았습니다 [KL80, Mas92]. 이걸 믿든 믿지 않든, 병렬 프로그래밍에서의 “게으름”은 종종 근면성에 비해 성능과 확장성이 좋습니다! 이런 성능과 확장성에서의 장점은 일을 뒤로 미루는 것은 종종 동기화 기능들을 약화시키는게 가능하게 하고, 따라서 동기화 오버헤드를 줄이게 된다는 사실에서 기인합니다. 일을 뒤로 미루는 일반적인 전략은 레퍼런스 카운팅 (Section 9.2), 해저드 포인터 (Section 9.3), 순차적 락킹 (Section 9.4), 그리고 RCU (Section 9.5) 등을 포함합니다. 마지막으로, Section 9.6에서는 이 챕터에서 다루어진 일 뒤로 미루기 전략들 가운데 어떻게 선택을 해야 하는지 이야기하고, Section 9.7에서는 업데이트의 역할에 대해 논합니다.

하지만 먼저 이런 방법들을 비교하고 대비하는데에 사용될 예제 알고리즘을 소개하겠습니다.

9.1 Running Example

이 챕터는 이런 접근법들의 가치를 보이고 또 그것들을 서로 비교할 수 있도록 하기 위해 단순화된 패킷 라우팅 알고리즘을 사용할 겁니다. 라우팅 알고리즘은 운영체제 커널에서 각각의 바깥으로 나가는 TCP/IP 패킷들을 알맞는 네트워크 인터페이스로 전달하는데에 사용됩니다. 이 특정한 알고리즘은 고전적인 1980년대의 packet-train-optimized 알고리즘으로 BSD UNIX [Jac88]에서 사용되었으며, 단순한 링크드 리스트로 구성되었습니다.¹ 최신 라우팅 알고리즘들은 더 복잡한 데이터 구조를 사용합니다만, Chapter 5에서와 같이, 극단적으로 간단한 알고리즘이 극단적으로 이해

¹ 달리 말하자면, 이건 OpenBSD, NetBSD 도 아니고 심지어 FreeBSD 도 아니었고 Pre-BSD 였습니다.

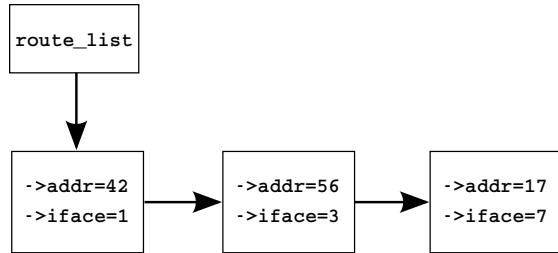


Figure 9.1: Pre-BSD Packet Routing List

하기 쉬운 구성에서의 병렬성에 특정된 문제들을 밝히는데 도움을 줄 것입니다.

우리는 더 나아가서 출발지와 목적지 IP 주소와 포트를 네가지 정보로 구성되는 검색 키를 간단한 정수로 교체함으로써 알고리즘을 더욱 단순화할 겁니다. 검색되고 리턴되는 값 또한 간단한 정수로 바꿔질 것이어서, 데이터 구조는 Figure 9.1에서와 같이 될건데, 이 그림대로라면 address 42의 패킷을 interface 1로, address 56의 패킷을 interface 2로, 그리고 address 17의 패킷을 interface 7로 전달할 겁니다. 외부의 패킷 네트워크는 안정적일 것을 가정하면, 이 리스트는 매우 자주 검색되고 아주 가끔씩만 업데이트 될 것입니다. Chapter 3에서 우리는, 빛의 제한된 속도와 물질의 원자성의 자연적 법칙과 같은 불편한 물리 법칙을 회피하는 가장 좋은 방법은 데이터를 쪼개거나 읽기가 대부분인 공유에 기대는 것임을 배웠습니다. 이 챕터에서, 우리는 이 Pre-BSD 패킷 라우팅 리스트를 사용해 읽기가 대부분인 상황을 위한 동기화 기법을 평가해 보도록 하겠습니다.

Figure 9.2는 Figure 9.1에 연관되는 간단한 싱글 쓰레드 구현을 보입니다. Line 1-5는 route_entry 구조체를 정의하고 line 6는 route_list 헤더를 정의합니다. Line 8-21은 route_lookup()을 정의하는데, 이 함수는 순차적으로 route_list를 검색하고

```

1 struct route_entry {
2     struct cds_list_head re_next;
3     unsigned long addr;
4     unsigned long iface;
5 };
6 CDS_LIST_HEAD(route_list);
7
8 unsigned long route_lookup(unsigned long addr)
9 {
10    struct route_entry *rep;
11    unsigned long ret;
12
13    cds_list_for_each_entry(rep,
14                            &route_list, re_next) {
15        if (rep->addr == addr) {
16            ret = rep->iface;
17            return ret;
18        }
19    }
20    return ULONG_MAX;
21 }
22
23 int route_add(unsigned long addr,
24                 unsigned long interface)
25 {
26    struct route_entry *rep;
27
28    rep = malloc(sizeof(*rep));
29    if (!rep)
30        return -ENOMEM;
31    rep->addr = addr;
32    rep->iface = interface;
33    cds_list_add(&rep->re_next, &route_list);
34    return 0;
35 }
36
37 int route_del(unsigned long addr)
38 {
39    struct route_entry *rep;
40
41    cds_list_for_each_entry(rep,
42                            &route_list, re_next) {
43        if (rep->addr == addr) {
44            cds_list_del(&rep->re_next);
45            free(rep);
46            return 0;
47        }
48    }
49    return -ENOENT;
50 }

```

Figure 9.2: Sequential Pre-BSD Routing Table

검색에 성공하면 연관되는 `->iface` 를 리턴하고, 검색에 실패하면 `ULONG_MAX` 를 리턴합니다. Line 23-35 는 `route_add()` 를 정의하는데, 이 함수는 `route_entry` 구조체를 메모리 할당받고, 초기화 한 후, 리스트에 추가하는데 메모리 할당에 실패한 경우에는 `-ENOMEM` 을 리턴합니다. 마지막으로, line 37-50 은 `route_del()` 을 정의하는데, 이 함수는 특정 `route_entry` 구조체를 존재한다면 제거하고 그렇지 않다면 `-ENOENT` 를 리턴합니다.

이 싱글쓰레드 구현은 이 챕터 안의 다양한 동시성을 사용한 구현의 하나의 프로토 타입 역할을 하고, 또한 이상적인 성능과 확장성의 평가를 위한 역할도 합니다.

```

1 struct route_entry { /* BUGGY!!! */
2     atomic_t re_refcnt;
3     struct route_entry *re_next;
4     unsigned long addr;
5     unsigned long iface;
6     int re_freed;
7 };
8 struct route_entry route_list;
9 DEFINE_SPINLOCK(routelock);
10
11 static void re_free(struct route_entry *rep)
12 {
13     ACCESS_ONCE(rep->re_freed) = 1;
14     free(rep);
15 }
16
17 unsigned long route_lookup(unsigned long addr)
18 {
19     int old;
20     int new;
21     struct route_entry *rep;
22     struct route_entry **repp;
23     unsigned long ret;
24
25     retry:
26     repp = &route_list.re_next;
27     rep = NULL;
28     do {
29         if (rep &&
30             atomic_dec_and_test(&rep->re_refcnt))
31             re_free(rep);
32         rep = ACCESS_ONCE(*repp);
33         if (rep == NULL)
34             return ULONG_MAX;
35         do {
36             if (ACCESS_ONCE(rep->re_freed))
37                 abort();
38             old = atomic_read(&rep->re_refcnt);
39             if (old <= 0)
40                 goto retry;
41             new = old + 1;
42         } while (atomic_cmpxchg(&rep->re_refcnt,
43                                old, new) != old);
44         repp = &rep->re_next;
45     } while (rep->addr != addr);
46     ret = rep->iface;
47     if (atomic_dec_and_test(&rep->re_refcnt))
48         re_free(rep);
49     return ret;
50 }

```

Figure 9.3: Reference-Counted Pre-BSD Routing Table Lookup (BUGGY!!!)

9.2 Reference Counting

레퍼런스 카운팅은 특정 오브젝트가 너무 빨리 해제되는 것을 막기 위해 오브젝트로의 레퍼런스들의 갯수를 추적합니다. 이런 방법은 1960년대 초로 거슬러 올라갈 만큼 매우 긴 역사를 가지고 있습니다 [Wei63].² 따라서 레퍼런스 카운팅은 동시적인 Pre-BSD 라우팅 구현의

² Weizenbaum 은 레퍼런스 카운팅을 이미 잘 알려진 것처럼 이야기 했는데, 따라서 역사는 1950년대로, 심지어는 1940년대까지도 거슬러 올라갈 수 있습니다. 그리고 심지어 더 갈수도 있겠죠. 커다란 기계를 고치고 관리하는 사람들은 각 일꾼들이 자물쇠를 갖는 방법으로 기계적인 레퍼런스 카운팅 테크닉을 사용해 왔습니다.

```

1 int route_add(unsigned long addr, /* BUGGY!!! */
2             unsigned long interface)
3 {
4     struct route_entry *rep;
5
6     rep = malloc(sizeof(*rep));
7     if (!rep)
8         return -ENOMEM;
9     atomic_set(&rep->re_refcnt, 1);
10    rep->addr = addr;
11    rep->iface = interface;
12    spin_lock(&routelock);
13    rep->re_next = route_list.re_next;
14    rep->re_freed = 0;
15    route_list.re_next = rep;
16    spin_unlock(&routelock);
17    return 0;
18 }
19
20 int route_del(unsigned long addr)
21 {
22     struct route_entry *rep;
23     struct route_entry **repp;
24
25     spin_lock(&routelock);
26     repp = &route_list.re_next;
27     for (;;) {
28         rep = *repp;
29         if (rep == NULL)
30             break;
31         if (rep->addr == addr) {
32             *repp = rep->re_next;
33             spin_unlock(&routelock);
34             if (atomic_dec_and_test(&rep->re_refcnt))
35                 re_free(rep);
36             return 0;
37         }
38         repp = &rep->re_next;
39     }
40     spin_unlock(&routelock);
41     return -ENOENT;
42 }

```

Figure 9.4: Reference-Counted Pre-BSD Routing Table Add/Delete (BUGGY!!!)

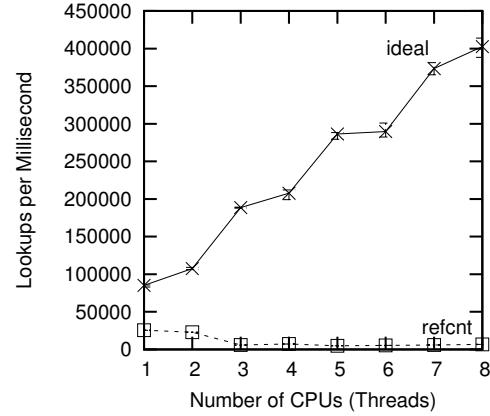


Figure 9.5: Pre-BSD Routing Table Protected by Reference Counting

훌륭한 후보입니다.

그런 생각 아래, Figure 9.3 는 데이터 구조들과 route_lookup() 함수를 보이고 있고, Figure 9.4 는 route_add() 와 route_del() 함수들을 보이고 있습니다(모두 route_refcnt.c 안에 있습니다). 이 알고리즘들은 Figure 9.2 에 보인 순차적 알고리즘과 상당히 비슷하므로, 차이점들만을 이야기 하겠습니다.

Figure 9.3 부터 시작해서, line 2 는 실제 레퍼런스 카운터를 더하고 있고, line 6 에서는 해제 후 사용을 체크하는 필드인 \rightarrow re_freed 를 더하며, line 9 에서는 동시의 업데이트들을 동기화 시키는데 사용될 routelock 을 추가하며, line 11-15 는 re_free() 를 더하는데, 이 함수는 \rightarrow re_freed 를 설정해서 route_lookup() 이 해제 후 사용 버그를 체크할 수 있게 합니다. route_lookup() 내에서, line 29-31 은 앞의 원소의 레퍼런스 카운트를 내려놓고 그 카운트가 0이 되었다면 해제시키며, line 35-43 은 새로운 원소로의 레퍼런스를 얻어오는데, line 36 과 37 은 해제 후 사용 체크를 수행합니다.

Quick Quiz 9.1: 해제 후 사용 체크를 왜 신경쓰죠?

Figure 9.4 에서, line 12, 16, 25, 33, 그리고 40 은 동시에의 업데이트 작업들을 동기화 시키기 위한 락킹을 보이고 있습니다. Line 14 는 해제 후 사용 체크 필드인 \rightarrow re_freed 를 초기화 하고, 만약 레퍼런스 카운트의 새 값이 0이라면 line 34-35 에서 마침내 re_free() 를 호출합니다.

Quick Quiz 9.2: Figure 9.4 의 route_del() 은 해제될 원소를 찾는 과정을 보호하기 위해 레퍼런스 카운트를 사용하지 않는 이유가 뭐죠? ■

Figure 9.5 는 싱글 소켓, 4 코어, 하이퍼쓰레드 가능한 2.5GHz x86 시스템에서 돌아가는 10개 원소 리스트에

서의 read-only 워크로드가 레퍼런스 카운팅을 사용했을 때의 성능과 확장성을 보이고 있습니다. “ideal”은 Figure 9.2에서 보인, 이게 읽기만 하는 워크로드여서 동작할 수 있는 순차적 코드를 돌리는 것으로 만들어졌습니다. “refcnt” 선이 x 축에 붙어버리는 모습으로 볼 수 있듯 레퍼런스 카운팅 성능은 저참하고 그 확장성은 그보다도 더합니다. Chapter 3를 생각하면 이는 놀라운 일은 아닙니다: 레퍼런스 카운트 획득과 해제는 read-only 워크로드에서도 빈번한 공유 메모리 쓰기를 추가했고, 따라서 물리 법칙의 상당한 보복을 만들어냈습니다. 이 세상의 바람직한 생각은 최신 디지털 전자 기술에서 빛의 속도를 늘리거나 원자의 크기를 줄이는 것이 아님을 생각할 때 당연한 일입니다.

Quick Quiz 9.3: Figure 9.5에서 “ideal” 선은 왜 똑바로 증가하지 않고 계단 형태로 증가하죠? ■

Quick Quiz 9.4: 요즘과 같은 시대에, Figure 9.5는 왜 겨우 8 CPU까지만 사용하는 거죠??? ■

이게 끝이 아닙니다.

반복적으로 `route_add()` 와 `route_del()` 을 호출하는 여러개의 업데이트 쓰레드들을 수행시키게 되면 Figure 9.3의 line 37의 해제 후 사용 버그를 알리는 `abort()` 문이 곧 수행될 겁니다. 이는 곧 레퍼런스 카운터가 확장성과 성능을 심각하게 저하시킬 뿐만 아니라, 필요한 보호를 제대로 제공하는데에도 실패함을 의미합니다.

Figure 9.1에 보인 리스트에서 해제 후 사용 버그를 이끌어내는 이벤트 시퀀스 중 하나는 아래와 같습니다:

1. Thread A가 address 42를 탐색해서 Figure 9.3 `route_lookup()`의 line 33에 도달합니다. 달리 말해, Thread A는 첫번째 원소로의 포인터를 갖고 있지만, 아직 그로의 레퍼런스를 획득하진 못한 상태입니다.
2. Thread B가 address 42의 route 원소를 지우려고 Figure 9.4의 `route_del()`을 호출합니다. 이는 성공적으로 완료되며, 이 원소의 `->re_refcnt` 필드는 그 값이 1이므로, `->re_freed` 필드를 설정하고 원소를 해제하기 위해 `re_free()`를 호출합니다.
3. Thread A는 `route_lookup()`의 수행을 계속 합니다. `rep`로의 포인터는 `NULL`이 아니지만, line 36은 자신의 `->re_freed` 필드가 0이 아님을 보게 되어서 line 37에서 `abort()`를 호출하게 됩니다.

문제는 이 레퍼런스 카운트가 보호되어야 할 오브젝트 안에 위치해 있다는 것인데, 이는 레퍼런스 카운트 자체가 획득되는 시점에서는 어떤 보호도 없음을 의미합니다! 이는 Gamsa 등 [GKAS99]에 의해 이야기되

었던 락킹에서의 문제의 레퍼런스 카운팅에서의 비슷한 경우입니다. Route 원소별로 레퍼런스 카운트 획득을 보호하기 위한 글로벌 락이나 레퍼런스 카운트를 생성해 볼 수도 있겠지만, 이는 상당한 경쟁 문제를 초래할 수 있습니다. 동시성 있는 환경에서 안전한 레퍼런스 카운트 획득을 위한 알고리즘들이 존재하긴 합니다만 [Val95], 그것들은 굉장히 복잡할 뿐더러 에러를 만들기가 쉬운데다가 [MS95] 저참한 성능과 확장성을 제공합니다 [HMBW07].

한마디로, 동시성은 분명히 레퍼런스 카운팅의 유용성을 저하시켰습니다!

Quick Quiz 9.5: 동시성이 “분명히 레퍼런스 카운팅의 유용성을 저하시켰다”면, 리눅스 커널은 왜 그렇게 레퍼런스 카운터를 많이 사용하는거죠? ■

그렇다곤 하지만, 가끔은 문제를 해결하기 위해선 완전히 다른 방법으로 문제를 바라볼 필요가 있습니다. 다음 섹션에서는 상당한 성능과 확장성을 제공하는, 안에서 바깥으로의 레퍼런스 카운트에서 생각해 볼수 있는 것들을 논해 봅니다.

9.3 Hazard Pointers

동시적으로 수행되는 레퍼런스 카운팅에서의 문제를 해결하는 한가지 방법은 레퍼런스 카운터들을 뒤집어서 구현하는 것으로, 데이터 원소에 저장되어 있는 정수를 증가시키는 게 아니라, CPU 별 (또는 쓰레드 별) 리스트들에 그 데이터 원소로의 포인터를 저장해 두는 것입니다. 이런 리스트의 각 원소들은 해저드 포인터 [Mic04]라고 불립니다.³ 주어진 데이터 원소의 “가상 레퍼런스 카운터”의 값은 그 원소를 레퍼런스하고 있는 해저드 포인터들의 갯수를 세는 것으로 얻어질 수 있습니다. 따라서, 그 원소가 읽기하는 쓰레드들에 의해 접근할 수 없게 된다면, 그리고 더이상 그 원소를 레퍼런스하고 있는 해저드 포인터가 더이상 존재하지 않는다면, 그 원소는 안전하게 메모리 해제될 수 있습니다.

물론, 이 말은 해저드 포인터 획득은 동시의 삭제들에 의한 정리 과정의 경주들을 막기 위해 매우 조심스럽게 행해져야만 합니다. 그런 한가지 구현이 Figure 9.6에 보여져 있는데, line 1-13에서 `hp_store()`를 보이고 line 15-20에서 `hp_erase()`를 보이고 있습니다. `smp_mb()` 기능은 Section 14.2에서 자세히 설명될 겁니다만, 이 간단한 개략적 설명의 목표를 위해서는 무시되어도 될 겁니다.

`hp_store()` 함수는 동시의 수정을 체크하면서 `p`에 의해 레퍼런스되는 포인터가 있는 데이터 원소를 위한 해저드 포인터를 `hp`에 저장합니다. 동시의 수정이

³ 그와 독립적으로 다른 사람들에 의해 개발된 것도 있습니다 [HLM02].

```

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 9.6: Hazard-Pointer Storage and Erasure

이뤄졌다면, `hp_store()` 는 해저드 포인터를 저장하는 것을 거부하고 0을 리턴함으로써 호출자는 다시 처음부터 데이터 접근을 다시 시작해야 함을 알립니다. 그렇지 않다면, `hp_store()` 는 해당 데이터 원소를 위한 해저드 포인터를 성공적으로 기록했음을 알리기 위해 1을 리턴합니다.

Quick Quiz 9.6: Figure 9.6 의 `hp_store()` 는 왜 데이터 원소로의 접근을 두번이나 간접적으로 하는거죠? 왜 `void *` 가 아니라 `void **` 인 건가요? ■

Quick Quiz 9.7: `hp_store()` 의 호출자는 실패했을 때 왜 데이터 접근을 처음부터 다시 시작해야 하는거죠? 데이터 구조체가 매우 크다면 좀 비효율적이지 않나요? ■

Quick Quiz 9.8: 해저드 포인터들에 대한 논문들은 각각의 포인터의 아래 비트들을 지워진 원소들을 마크하기 위해 사용한다고 하는데, `HAZPTR_POISON` 은 뭔가요? ■

해저드 포인터들을 사용하는 알고리즘들은 데이터 구조체를 지나가는 중 어떤 단계에서든 재시작할 수 있으므로, 그런 알고리즘들은 모든 필요한 해저드 포인터들을 얻어오는 작업이 끝나기 전까지는 이 데이터 구조체에 어떤 변경을 가하지 않도록 특별한 주의를 반드시 기울여야만 합니다.

Quick Quiz 9.9: 하지만 해저드 포인터들에 있는 이런 제약사항들은 다른 형태의 레퍼런스 카운팅에도 똑같이 적용되는 거 아닌가요? ■

이런 제약사항들은 읽기를 하는 쓰레드들에는 커다란 이득으로 귀결되는데, 해저드 포인터들은 각 CPU/쓰레드에 지역적으로 저장되기 때문으로, 이에 의해 데이터 구조체들을 횡단하는 작업은 완전히 읽기만 하면서 행해질 수 있다는 사실 덕입니다. page 63 의 Figure 5.29 를 다시 인용하자면, 해저드 포인터들은 CPU 캐시들이

```

1 struct route_entry {
2     struct hzptr_head hh;
3     struct route_entry *re_next;
4     unsigned long addr;
5     unsigned long iface;
6     int re_freed;
7 };
8 struct route_entry route_list;
9 DEFINE_SPINLOCK(routelock);
10 hazard_pointer __thread *my_hzptr;
11
12 unsigned long route_lookup(unsigned long addr)
13 {
14     int offset = 0;
15     struct route_entry *rep;
16     struct route_entry **repp;
17
18     retry:
19     repp = &route_list.re_next;
20     do {
21         rep = ACCESS_ONCE(*repp);
22         if (rep == NULL)
23             return ULONG_MAX;
24         if (rep == (struct route_entry *)HAZPTR_POISON)
25             goto retry;
26         my_hzptr[offset].p = &rep->hh;
27         offset = !offset;
28         smp_mb();
29         if (ACCESS_ONCE(*repp) != rep)
30             goto retry;
31         repp = &rep->re_next;
32     } while (rep->addr != addr);
33     if (ACCESS_ONCE(rep->re_freed))
34         abort();
35     return rep->iface;
36 }

```

Figure 9.7: Hazard-Pointer Pre-BSD Routing Table Lookup

리소스 복사를 할 수 있게 해서 병렬 액세스 컨트롤 메커니즘을 약화시키는 걸 가능하게 하고, 따라서 성능과 확장성을 높여줍니다. 다른 메커니즘들과의 성능 비교는 Chapter 10 와 다른 출간물들 [HMBW07, McK13, Mic04] 에서 얻을 수 있을 겁니다.

해저드 포인터들은 레퍼런스 카운터들보다 훨씬 더 확장성 있습니다만 여전히 읽기를 하는 쓰레드들이 공유 메모리에 쓰기를 하게 합니다. 다음 섹션의 주제인 시퀀스 카운터들은 읽기 쪽의 쓰기를 완전히 막습니다.

Pre-BSD 라우팅 예제는 해저드 포인터를 사용할 수 있는데, Figure 9.7 에서 그런 데이터 구조들과 `route_lookup()` 를 보이고 있고, Figure 9.8 에서 `route_add()` 와 `route_del()` 을 보이고 있습니다(`route_hzptr.c`). 레퍼런스 카운팅에서와 마찬가지로, 해저드 풍미 구현은 page 120 의 Figure 9.2 에서 보인 순차적 알고리즘과 상당히 유사하므로 차이점만 이야기하겠습니다.

Figure 9.7 에서 시작해, line 2 에서는 해저드 포인터가 해제되길 지연시키는 오브젝트들을 넣어두기 위한 `->hh` 필드를 보이며, line 6 는 해제 후 사용 버그를 발견하기 위한 `->re_freed` 필드를 보이며 line 24-30

```

1 int route_add(unsigned long addr,
2                 unsigned long interface)
3 {
4     struct route_entry *rep;
5
6     rep = malloc(sizeof(*rep));
7     if (!rep)
8         return -ENOMEM;
9     rep->addr = addr;
10    rep->iface = interface;
11    rep->re_freed = 0;
12    spin_lock(&routelock);
13    rep->re_next = route_list.re_next;
14    route_list.re_next = rep;
15    spin_unlock(&routelock);
16    return 0;
17 }
18
19 int route_del(unsigned long addr)
20 {
21     struct route_entry *rep;
22     struct route_entry **repp;
23
24     spin_lock(&routelock);
25     rep = &route_list.re_next;
26     for (;;) {
27         rep = *repp;
28         if (rep == NULL)
29             break;
30         if (rep->addr == addr) {
31             *repp = rep->re_next;
32             rep->re_next =
33                 (struct route_entry *)HAZPTR_POISON;
34             spin_unlock(&routelock);
35             hazptr_free_later(&rep->hh);
36             return 0;
37         }
38         rep = &rep->re_next;
39     }
40     spin_unlock(&routelock);
41     return -ENOENT;
42 }

```

Figure 9.8: Hazard-Pointer Pre-BSD Routing Table Add/Delete

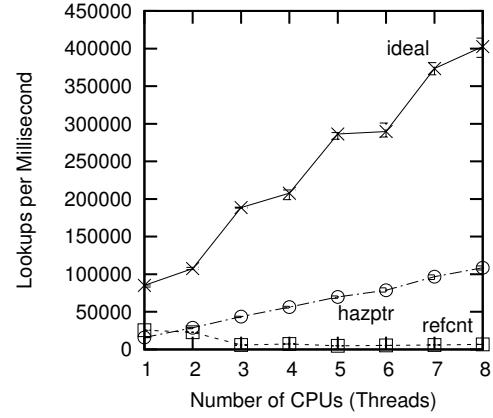


Figure 9.9: Pre-BSD Routing Table Protected by Hazard Pointers

에서는 해저드 포인터를 획득하려 시도하고, 실패하면 line 18의 `retry` 라벨로 브랜치하게 됩니다.

Figure 9.8에서, line 11에서는 `->re_freed`를 초기화시키고, line 32와 33에서는 새로 제거된 오브젝트의 `->re_next`를 파괴하고, line 35는 그 오브젝트를 해당 해저드 포인터들의 `hazptr_free_later()` 함수에 넘기는데, 이 함수는 해당 오브젝트가 해제되기 안전해질 때 해제시킬 겁니다. 스피너들은 Figure 9.4에서와 동일하게 동작합니다.

Figure 9.9는 해저드 포인터로 보호되는 Pre-BSD 라우팅 알고리즘의 Figure 9.5에서와 동일한 read-only 워크로드에서의 성능을 보입니다. 해저드 포인터가 레퍼런스 카운팅에 비해 훨씬 잘 확장되긴 하지만, 해저드 포인터는 여전히 읽기 쓰레드가 공유 메모리에 쓰기를 할 것을 필요로 하며 (대신 훨씬 개선된 레퍼런스의 로컬리티를 제공합니다), 각 오브젝트 방문마다 메모리 배리어와 재시도 여부 검사를 필요로 합니다. 그로 인해 해저드 포인터의 성능은 이상적인 것에 비하면 훨씬 떨어집니다. 반면에, 해저드 포인터는 동시의 업데이트가 존재할 때에도 올바르게 동작합니다.

Quick Quiz 9.10: 논문 “Structured Deferral: Synchronization via Procrastination” [McK13]는 해저드 포인터가 이상적인 경우에 가까운 성능을 보인다는 걸 보였습니다. Figure 9.9에선 무슨 일이 일어난거죠???

다음 섹션은 해저드 포인터에서 더 개선을 하기 위해서 read-side에서의 쓰기도 오브젝트별 메모리 배리어도 없애주는 시퀀스 락을 사용해 보겠습니다.

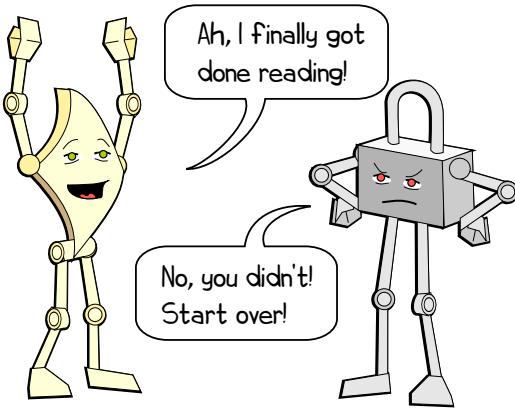


Figure 9.10: Reader And Uncooperative Sequence Lock

9.4 Sequence Locks

시퀀스 락은 읽기를 하는 쓰레드들에게 일관적인 상태로 보여야만 하는 읽기가 대부분 이루어지는 데이터를 위해 리눅스 커널에서 사용됩니다. 하지만, reader-writer 락킹과 달리, 읽기를 하는 쓰레드들은 쓰기를 하는 쓰레드들을 배타시키지 않습니다. 대신에, 해저드 포인터처럼, 시퀀스 락들은 읽기를 하는 쓰레드들이 동시에 수행중인 쓰기를 하는 쓰레드들로부터의 동작이 탐지되면 오퍼레이션을 재시도 하도록 강제합니다. Figure 9.10에 보여지는 것처럼, 읽기를 하는 쓰레드들이 재시도를 하는 경우는 매우 드문 경우일 수 있도록 시퀀스 락을 사용해 코드를 설계하는 것이 중요합니다.

Quick Quiz 9.11: 왜 이 시퀀스 락에 대한 토론은 Chapter 7에서, *locking*의 하나로써 다루어지지 않았던 거죠? ■

시퀀스 락킹에서의 핵심 컴포넌트는 시퀀스 넘버로, 이 수는 업데이트를 하는 쓰레드가 존재하지 않는다면 짹수를 가지고 진행 중인 업데이트가 있다면 훌수를 갖습니다. 읽기를 하는 쓰레드들은 각각의 액세스 전과

```

1 do {
2     seq = read_seqbegin(&test_seqlock);
3     /* read-side access. */
4 } while (read_seqretry(&test_seqlock, seq));

```

Figure 9.11: Sequence-Locking Reader

```

1 write_seqlock(&test_seqlock);
2 /* Update */
3 write_sequnlock(&test_seqlock);

```

Figure 9.12: Sequence-Locking Writer

후로 이 값을 스냅샷을 뜰 수 있습니다. 만약 두 스냅샷 중 하나라도 훌수이거나, 두 스냅샷이 서로 다르다면, 동시에 업데이트가 있었다는 것이고, 따라서 읽기를 한 쓰레드는 해당 액세스의 결과를 버리고 다시 읽기를 시도해야 합니다. 시퀀스 락으로 보호되는 데이터에 대해 읽기를 하기 위해서는 `read_seqbegin()` 과 `read_seqretry()` 함수들이 Figure 9.11과 같이 사용되어야 합니다. 쓰기를 하는 쓰레드들은 각각의 업데이트 전과 후에 그 값을 증가시켜야 하는데, 한번에 하나의 읽기 쓰레드의 수행만이 허가되어 있습니다. 쓰기를 하는 쓰레드들은 시퀀스 락으로 보호되는 데이터를 업데이트 하기 위해서는 `write_seqlock()` 과 `write_sequnlock()` 함수를 Figure 9.12에 보여진 것처럼 사용합니다.

따라서 시퀀스 락으로 보호되는 데이터는 상당히 많은 수의 동시에 수행되는 읽기를 하는 쓰레드들을 가질 수 있습니다만, 쓰기를 하는 쓰레드는 한번에 하나씩만 가능합니다. 시퀀스 락킹은 시간 계측에 사용되는 값의 측정을 보호하기 위해 사용됩니다. 시퀀스 락킹은 또한 동시적인 이름 바꾸기 오퍼레이션들을 파악해내기 위한 경로명 탐색에도 사용됩니다.

시퀀스 락의 간단한 구현이 Figure 9.13(`seqlock.h`)에 있습니다. Line 1-4 `seqlock_t` 데이터 구조체가 있는데, 쓰기를 하는 쓰레드들을 직렬화 시키기 위한 락과 시퀀스 넘버를 가지고 있습니다. Line 6-10에서는 `seqlock_init()`를 보이는데, 이 함수는 이름이 의미하듯이 `seqlock_t`의 초기화를 합니다.

Line 12-19는 `read_seqbegin()` 함수로, 시퀀스 락의 read-side 크리티컬 섹션을 시작합니다. Line 16에서 시퀀스 카운터의 스냅샷을 하나 만들고, line 17에서 이 이스냅샷의 만들어진 순서가 호출자의 크리티컬 섹션보다 전이 되도록 순서를 세웁니다. 마지막으로, line 18에서 스냅샷의 값을 (least-significant bit를 비운 상태로) 리턴하는데, 이 값은 호출자가 뒤의 `read_seqretry()` 호출에 넘겨줄 값입니다.

Quick Quiz 9.12: Figure 9.13의 `read_seqbegin()`은 왜 내부적으로 가장 낮은 자리의 비트를 검사하고 재시도를 하지 않고 어차피 망할 읽기를 시작하는 건가요? ■

Line 21-29는 `read_seqretry()` 함수인데, 이 함수는 연관된 `read_seqbegin()`의 실행 아래로부터 쓰기를 하는 쓰레드가 존재하지 않았다면 `true`를 리턴합니다. Line 26에서는 호출자의 앞의 크리티컬 섹션을 line 27에서의 시퀀스 카운터의 새로운 스냅샷을 얻어오는 작업 이전에 완료되도록 순서를 맞춥니다. 마지막으로, line 28에서는 시퀀스 카운터가 변하지 않았음을 검사하는데, 달리 말하자면 그동안 쓰기가 행해지지 않았음을 확인하고, 그렇다면 `true`를 리턴합니다.

Quick Quiz 9.13: Figure 9.13의 line 26에서의 `smp_`

```

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     s = ACCESS_ONCE(slp->seq);
17     smp_mb();
18     return s & ~0x1UL;
19 }
20
21 static int read_seqretry(seqlock_t *slp,
22                          unsigned long oldseq)
23 {
24     unsigned long s;
25
26     smp_mb();
27     s = ACCESS_ONCE(slp->seq);
28     return s != oldseq;
29 }
30
31 static void write_seqlock(seqlock_t *slp)
32 {
33     spin_lock(&slp->lock);
34     ++slp->seq;
35     smp_mb();
36 }
37
38 static void write_sequnlock(seqlock_t *slp)
39 {
40     smp_mb();
41     ++slp->seq;
42     spin_unlock(&slp->lock);
43 }

```

Figure 9.13: Sequence-Locking Implementation

mb() 는 왜 필요한 건가요? ■

Quick Quiz 9.14: Figure 9.13 의 코드는 완화된 형태의 메모리 배리어를 사용할 수는 없을까요? ■

Quick Quiz 9.15: 시퀀스 락킹 아래서, 업데이트 쓰레드들이 읽기 쓰레드들이 진행 못하게 하는 걸 막는 건 무엇일까요? ■

Line 31-36 는 write_seqlock() 함수로, 단순히 락을 획득하고, 시퀀스 넘버를 증가시키고, 이 값 증가 연산이 호출자의 크리티컬 섹션 전으로 순서맞춰짐을 분명히 하도록 메모리 배리어를 실행합니다. Line 38-43 은 write_sequnlock() 함수를 보여주는데, 이 함수는 호출자의 크리티컬 섹션이 line 44 에서의 시퀀스 넘버 값 증가 전으로 순서맞춰지는 것을 분명히 하도록 메모리 배리어를 실행하고 락을 해제합니다.

Quick Quiz 9.16: 다른 뭔가가 쓰기 쓰레드들을 직렬화 시켜서 락이 필요치 않다면 어떻게 되죠? ■

Quick Quiz 9.17: Figure 9.13 의 line 2 의 seq 는 왜

```

1  struct route_entry {
2      struct route_entry *re_next;
3      unsigned long addr;
4      unsigned long iface;
5      int re_freed;
6  };
7  struct route_entry route_list;
8  DEFINE_SEQ_LOCK(sl);
9
10 unsigned long route_lookup(unsigned long addr)
11 {
12     struct route_entry *rep;
13     struct route_entry **repp;
14     unsigned long ret;
15     unsigned long s;
16
17     retry:
18     s = read_seqbegin(&sl);
19     repp = &route_list.re_next;
20     do {
21         rep = ACCESS_ONCE(*repp);
22         if (rep == NULL) {
23             if (read_seqretry(&sl, s))
24                 goto retry;
25             return ULONG_MAX;
26         }
27         repp = &rep->re_next;
28     } while (rep->addr != addr);
29     if (ACCESS_ONCE(rep->re_freed))
30         abort();
31     ret = rep->iface;
32     if (read_seqretry(&sl, s))
33         goto retry;
34     return ret;
35 }

```

Figure 9.14: Sequence-Locked Pre-BSD Routing Table Lookup (BUGGY!!!)

unsigned 가 아니라 unsigned long 인가요? 무엇보다, unsigned 가 리눅스 커널에서 충분히 좋은 것이라면 모두에게도 충분히 좋지 않을까요? ■

시퀀스 락킹이 Pre-BSD 라우팅 테이블에 적용되면 어떻게 될까요? Figure 9.14 는 데이터 구조들과 route_lookup() 을 보이고, Figure 9.15 는 route_add() 와 route_del() 을 보이고 있습니다 (route_seqlock.c). 이 구현은 역시 앞 섹션의 같은 것들과 비슷하므로 차이점들만을 이야기 하겠습니다.

Figure 9.14 에서, line 5 에는 ->re_freed 가 있는데, line 29 와 30 에서 체크됩니다. Line 8 에는 시퀀스 락이 있는데, route_lookup() 에 의해 line 18, 23, 32 에서 사용되며 line 24 와 33 은 line 17 의 retry 라벨로 수행을 되돌립니다. 이로 인해 업데이트와 동시에 수행되는 텁색은 재시도를 하게 됩니다.

Figure 9.15 에서, line 12, 15, 24, 그리고 40 은 시퀀스 락을 잡고 풀고 있는데, line 11, and 34 는 ->re_freed 를 처리합니다. 따라서 이 구현은 상당히 간단합니다.

Figure 9.16 에서 볼 수 있듯이, 이 구현은 또한 read-only 워크로드에서 상당히 좋은 성능을 보입니다. 이상적인 성능에 비하면 여전히 멀었지만요.

안타깝지만, 이 구현 역시 해제 후 사용 문제

```

1 int route_add(unsigned long addr,
2                 unsigned long interface)
3 {
4     struct route_entry *rep;
5
6     rep = malloc(sizeof(*rep));
7     if (!rep)
8         return -ENOMEM;
9     rep->addr = addr;
10    rep->iface = interface;
11    rep->re_freed = 0;
12    write_seqlock(&sl);
13    rep->re_next = route_list.re_next;
14    route_list.re_next = rep;
15    write_sequnlock(&sl);
16    return 0;
17 }
18
19 int route_del(unsigned long addr)
20 {
21     struct route_entry *rep;
22     struct route_entry **repp;
23
24     write_seqlock(&sl);
25     repp = &route_list.re_next;
26     for (;;) {
27         rep = *repp;
28         if (rep == NULL)
29             break;
30         if (rep->addr == addr) {
31             *repp = rep->re_next;
32             write_sequnlock(&sl);
33             smp_mb();
34             rep->re_freed = 1;
35             free(rep);
36             return 0;
37         }
38         repp = &rep->re_next;
39     }
40     write_sequnlock(&sl);
41     return -ENOENT;
42 }

```

Figure 9.15: Sequence-Locked Pre-BSD Routing Table Add/Delete (BUGGY!!!)

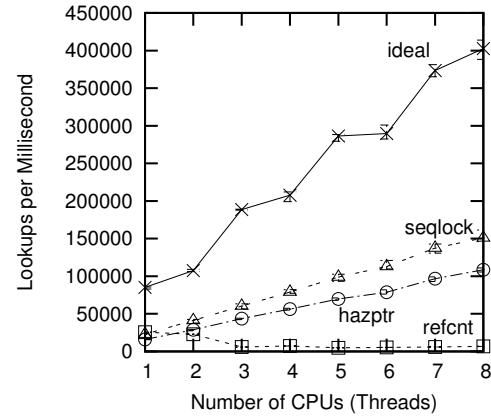


Figure 9.16: Pre-BSD Routing Table Protected by Sequence Locking

를 가지고 있습니다. 문제는 읽기 쓰레드가 `read_seqretry()`를 하기 전에 이미 해제된 구조체에 접근할 수 있기 때문에 segmentation violation 을 낼 수 있다는 점입니다.

Quick Quiz 9.18: 이 버그는 고쳐질 수 있을까요? 달리 말해서, 동시의 삽입, 삭제, 탐색을 지원하는 링크드 리스트를 보호하는 동기화 메커니즘으로 시퀀스락 하나만 사용할 수 있을까요? ■

시퀀스락의 read-side 크리티컬 섹션도, write-side 크리티컬 섹션도 트랜잭션으로 생각될 수 있고, 따라서 시퀀스락은 제한적인 형태의 트랜잭션 메모리로 생각될 수 있겠는데, 트랜잭션 메모리에 대해서는 Section 17.2 에서 이야기 하겠습니다. 시퀀스락의 제약점들은: (1) 시퀀스락은 업데이트를 제약하고 (2) 시퀀스락은 업데이트 쓰레드에 의해 해제되었을 수 있는 오브젝트로의 포인터의 횡단을 허용하지 않습니다. 이런 제약점들은 물론 트랜잭션 메모리를 사용해 극복될 수 있습니다만, 다른 동기화 도구들을 시퀀스락과 함께 사용해서도 극복될 수 있습니다.

시퀀스락들은 쓰기를 하는 쓰레드들이 읽기를 하는 쓰레드들을 뒤로 미뤄지도록 할 수 있지만, 그 반대는 불가능합니다. 이는 공정하지 않고 쓰기가 대부분인 워크로드에서는 스타베이션을 낼 수도 있습니다. 반면, 쓰기 쓰레드가 없다면, 시퀀스락을 사용하는 읽기 쓰레드들은 합리적인 수준으로 빠르고 선형적으로 확장이 가능할 것입니다. 두 가지 모두 최선인 경우를 원하는 게 사람입니다: 읽는 쪽의 실패나 스타베이션의 가능성 없는 빠른 읽기. 또한, 포인터들에 대한 제약점들 역시 극복된다면 좋을 겁니다. 다음 섹션은 이런 속성을 가진 동기화 메커니즘을 소개합니다.

9.5 Read-Copy Update (RCU)

앞의 섹션들에서 다루어진 모든 메커니즘들은 특정 동작들을 그것이 안전하게 행해질 수 있을 때까지 유예하는 여러개의 방법들 가운데 하나를 사용했습니다. Section 9.2에서 다룬 레퍼런스 카운터는 읽기 쓰레드들을 방해할 수 있는 행위들을 유예하기 위해 명시적 카운터를 사용하는데, 이는 읽기 쪽의 혼잡과 그로 말미암은 낮은 확장성을 유발합니다. Section 9.3에서 다루어진 해저드 포인터는 per-thread 포인터 리스트 안의 묵시적 카운터를 사용합니다. 이는 읽기 쪽의 혼잡을 줄입니다만, 읽기 쪽 기능에 전체 메모리 배리어를 필요로 합니다. Section 9.4에서 보인 시퀀스 락 또한 읽기 쪽의 혼잡을 막습니다만, 포인터 traversal을 보호해주지는 않고, 해저드 포인터처럼 읽기 쪽 기능에 전체 메모리 배리어를 필요로 합니다. 이런 방법들의 단점들은 더 좋은 방법은 없는지에 질문을 이끌어냅니다.

이 섹션은 딜레이가 공유된 데이터에의 비싼 업데이트보다는 소스코드에 의해 인식되도록 하는 API를 제공하는 *read-copy update* (RCU)를 소개합니다. 이 섹션의 나머지 부분들에서는 여러개의 다른 관점에서 RCU를 알아봅니다. Section 9.5.1에서는 RCU에 대한 고전적인 소개를 제공하고, Section 9.5.2에서는 기본적인 RCU 컨셉을 다루며, Section 9.5.3에서는 RCU의 일부 공통적인 사용예를 소개하고, Section 9.5.4에서는 리눅스 커널 API를 보이고, Section 9.5.5에서는 사용자 레벨 RCU의 일련의 “장난감” 구현들을 다루며, 마지막으로 Section 9.5.6에서 일부 RCU 연습을 제공합니다.

9.5.1 Introduction to RCU

앞의 섹션들에서 이야기된 방법들은 어느정도 확장성 있긴 했지만 모두 Pre-BSD 라우팅 테이블을 위한 성능에 있어서는 이상적이지 못했습니다. Pre-BSD 루트 오버헤드가 싱글 쓰레드 탐색에서와 동일하도록 별별로 수행되는 탐색들이 싱글 쓰레드에서의 탐색과 동일한 어셈블리어 인스트럭션 시퀀스를 수행한다면 좋을 겁니다. 이는 좋은 목표가 될 수 있지만, 그러기 위해서는 많은 심각한 구현 단계에서의 질문을 이끌어냅니다. 하지만 이걸 시도하면 어떤 일이 벌어질지 알아보고 삽입과 삭제를 구분해서 다뤄봅시다.

인입을 위한 고전적인 방법이 Figure 9.17에 표현되어 있습니다. 첫번째 열은 기본 상태를 보이는데, `gptr`은 NULL의 값을 갖습니다. 두번째 열에서는 하나의 구조체를 메모리 할당하는데, 초기화 되지 않은 부분들은 물음표로 표시되어 있습니다. 세번째 열에서는 이 구조체를 초기화 시킵니다. 다음으로, `gptr`이 이 새로운 원소를 가리키도록 그 값을 할당합니다.⁴ 최근의 범용

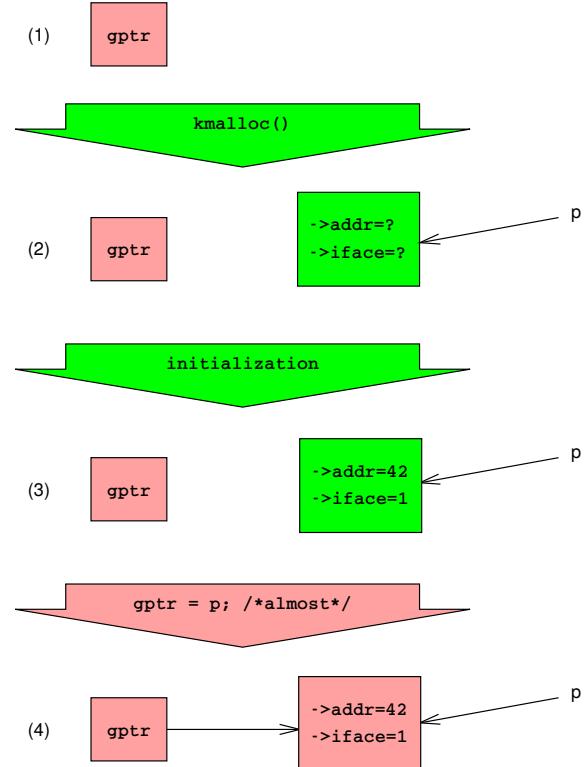


Figure 9.17: Insertion With Concurrent Readers

시스템들에서, 이 값 할당은 어토믹해서 동시에 수행되는, 읽기를 하는 쓰레드들은 NULL 포인터 또는 새로운 구조체 `p`로의 포인터 둘 중 하나만을 보게 하지, 두 값들 중 일부들이 합쳐져 있는 값은 보지 못하게 됩니다. 따라서, 각각의 읽기를 하는 쓰레드는 기본값인 NULL을 보거나 기본값이 아닌 값을 보게 되거나일 것이며 어느 쪽이든 읽기는 일관적인 결과만을 보게 됨이 보장됩니다. 이뿐만이 아니라, 읽기를 수행하는 쓰레드들은 어떤 다른 비싼 동기화 기능을 사용할 필요가 없어서, 이 방법은 리얼타임 쪽의 사용에 꽤 적합할 겁니다.⁵

하지만 동시에 읽기 쓰레드들에 의해 레퍼런스 되고 있는 데이터는 언젠가는 없어져야 할 겁니다. Figure 9.18와 같이 링크드 리스트에서 원소들을 삭제하는, 더 복잡한 예제를 봅시다. 이 리스트는 초기에 원소 A, B, 그리고 C를 가지고 있으며, 여기서 원소 B를 삭제해야

있기 때문에 단순한 값 할당은 충분하지 못합니다. 이런 문제에 대해서는 Section 9.5.2에서 다루게 될 겁니다.

⁴ 많은 컴퓨터 시스템들에서, 컴파일러와 CPU가 간섭을 끼칠 수

⁵ 다시 말하지만, 많은 컴퓨터 시스템들에서, 컴파일러와, DEC Alpha 시스템에서라면 CPU가, 간섭을 행하는 것을 막기 위해 추가 작업이 필요합니다. 이에 대해서는 Section 9.5.2에서 이야기합니다.

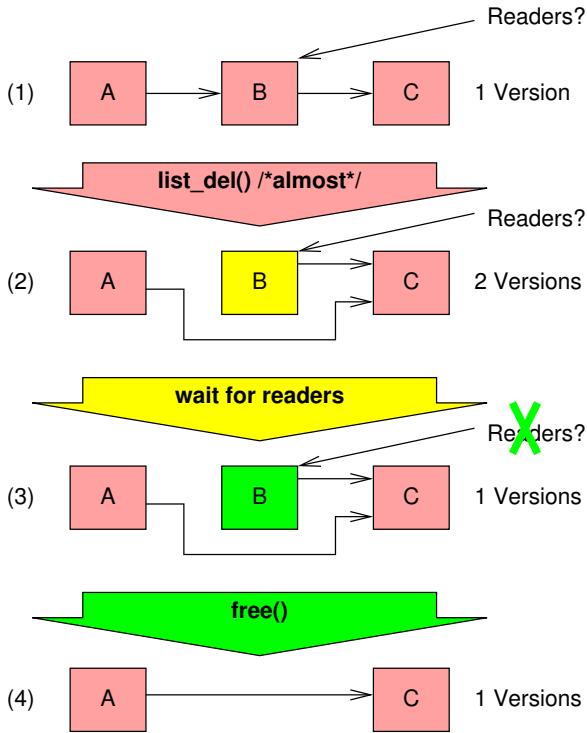


Figure 9.18: Deletion From Linked List With Concurrent Readers

합니다. 먼저, `list_del()`을 사용해 삭제를 진행하는데,⁶ 모든 새로운 읽기 쓰레드들은 원소 B를 리스트에서 삭제된 것으로 보게 될 겁니다. 하지만, 이 원소를 여전히 보고 있는 오래된 읽기 쓰레드들도 있을 수 있습니다. 이 오래된 읽기 쓰레드들이 종료되고 나면, 원소 B를 안전히 메모리 해제시켜서 그림의 아래쪽에 그려진 상태를 만들 수 있을 겁니다.

하지만, 그 종료 시점을 어떻게 알까요?

레퍼런스 카운팅 방법의 경우는, Chapter 5의 Figure 5.3 가 락킹과 시퀀스 락킹처럼 긴 딜레이를 가져올 수 있음을 보이므로 어렵습니다.

읽기 쓰레드들이 그 존재를 알리기 위한 일을 전히 안하는, 극단적인 상황을 생각해 봅시다. 이는 읽기 쓰레드들에 최적의 성능을 가능하게 하겠지만 (아무것도 안해도 되니까요), 업데이트 쓰레드는 어떻게 모든 예전 읽기 쓰레드들의 종료를 알지에 대한 질문이 남습니다. 여기에 합리적 답을 하기 위해서는 추가적인 제약이 분명 필요합니다.

일부 운영체제에 적합한 제약은 CPU를 빼

⁶ 역시 앞서 말했듯 이는 추상화된 예이고, Section 9.5.2에서 이에 관련해서 더 이야기 합니다.

앗기지 않는 상황 (non-preemptible)입니다. CPU를 빼앗길 수 없는 환경에서 쓰레드는 명시적이고 자발적으로 블락킹 되기 전까지는 수행을 계속합니다. 즉, 블락킹 없이 반복되는 무한루프는 CPU를 무한루프 이외의 목적으로는 사용될 수 없게 하다는 의미입니다.⁷ CPU를 빼앗길 수 없다는 특성은 또한 쓰레드들이 스핀락을 잡고 있는 동안은 블락킹 되지 않아야 할 것을 필요로 합니다. 이런 금지사항이 없다면, 블락된 쓰레드에 의해 잡혀 있는 스핀락을 획득하려 시도하며 루프를 도는 쓰레드들에 의해 모든 CPU가 소모되게 될 수도 있습니다. 루프를 도는 쓰레드들은 락을 잡기 전까지는 자신들의 CPU들을 놓지 않을 것인데, 락을 잡고 있는 쓰레드는 이 루프를 돌고 있는 쓰레드들이 CPU를 놓기 전까지는 그 락을 놓을 수가 없습니다. 이는 고전적인 deadlock 상황입니다.

이와 똑같은 제약사항을 링크드 리스트를 획단하며 읽기를 하는 쓰레드들에도 가해봅시다: 그런 쓰레드들은 리스트 획단이 완료되기 전까지는 블락되는 것이 허용되지 않습니다. 업데이트 쓰레드가 `list_del()`을 실행 완료한 직후인 Figure 9.18의 두번째 줄로 돌아가서, CPU 0 가 컨텍스트 스위칭을 한다고 생각해 봅시다. 읽기 쓰레드들은 링크드 리스트 획단 중에 블락되는 것은 허용되지 않으므로, CPU 0에서 수행되던 모든 시간상 앞의 읽기 쓰레드들은 완료되었음이 보장됩니다. 이 이야기를 다른 CPU 들에도 적용해 보자면, 각 CPU가 일단 컨텍스트 스위칭이 수행됨을 확인했다면, 모든 시간상 앞의 읽기 쓰레드들은 완료되었고, 더이상 원소 B를 레퍼런스 하고 있는 읽기 쓰레드는 더이상 없다고 보장된다고 볼 수 있습니다. 그렇다면 업데이트 쓰레드는 안전하게 원소 B를 메모리 해제해서 Figure 9.18의 가장 아래의 상태를 만들어낼 수 있습니다.

이런 방법은 *quiescent state based reclamati*on (QSBR) [HMB06] 이라 명명되어 있습니다. 하나의 QSBR 방법이 Figure 9.19, 에 그림의 꼭대기부터 바닥까지로 시간의 흐름에 따라 그려져 있습니다.

이런 방법의 상품 품질 구현은 상당히 복잡할 수 있지만, 장난감 수준 구현은 상당히 간단합니다:

```
1 for_each_online_cpu(cpu)
2   run_on(cpu);
```

이 `for_each_online_cpu()` 함수는 모든 CPU 들에 루프를 돌고, `run_on()` 함수는 현재 쓰레드가 특정 CPU에서 수행되게 해서 목적지 CPU가 컨텍스트 스위칭을 수행하게 만듭니다. 따라서, 일단 한번 `for_each_online_cpu()` 가 완료되면, 각 CPU는 컨텍스트 스위치를 수행한 것이고, 따라서 모든 시간상 앞의 읽기 쓰레드들은 완료되었음이 보장됩니다.

⁷ 반면, CPU를 빼앗길 수 있는 환경에서의 무한루프는 여전히 CPU 시간을 낭비하고 있긴 하지만, 이 CPU는 다른 일을 할 수 있을 겁니다.

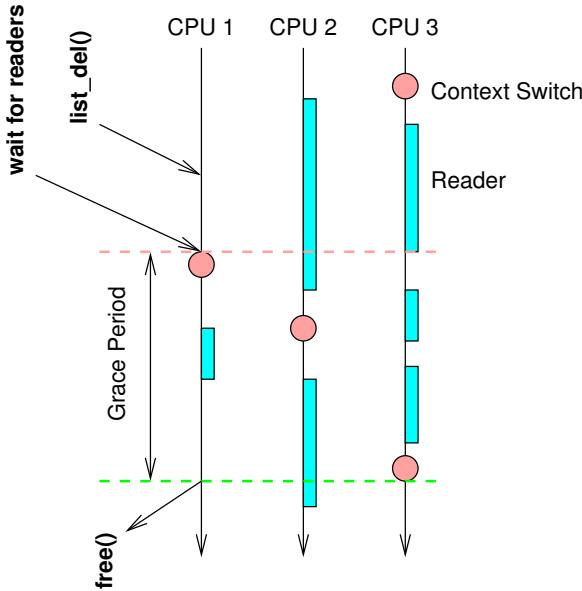


Figure 9.19: RCU QSBR: Waiting for Pre-Existing Readers

이 방법은 상품 수준의 품질이 아님을 알아 두시기 바랍니다. 여러개의 일반적이지 않은 문제 경우의 처리와 여러개의 강력한 최적화의 필요성은 상품 수준 품질의 구현은 상당한 추가적 복잡도를 의미합니다. 또한, CPU 강탈이 가능한 환경에서의 RCU 구현은 읽기 쓰레드들이 실제로 뭔가를 할것을 필요로 합니다. 하지만, 이 간단한 CPU 강탈 불가한 상황의 접근법은 개념적으로는 완벽하고, 다음 섹션에서 다루어질 RCU의 기본을 이해하기 위한 좋은 초기 토대가 될겁니다.

9.5.2 RCU Fundamentals

Read-copy update (RCU)는 2002년 10월에 리눅스 커널에 추가된, 하나의 동기화 메커니즘입니다. RCU는 읽기 작업들이 업데이트 작업들과 동시에 일어날 수 있도록 함으로써 확장성 개선을 달성합니다. 기존에 일반적으로 사용되어온, 동시에 수행되는 쓰레드들에 대해 그것들이 읽기를 하는지 업데이트를 하는지와 상관없이 상호 배제를 보장하는 락킹 기능들 또는 동시에의 읽기 작업은 허용하지만 업데이트가 함께 수행되는 것은 막는 reader-writer 락들과는 대조적으로 RCU는 하나의 업데이트 쓰레드와 여러 읽기 쓰레드들 사이의 동시성을 지원합니다. RCU는 오브젝트들의 여러 버전들을 유지하고 그것들이 이전부터 존재해온 모든 읽기쪽 크리티컬 섹션들이 완료되기 전까지는 메모리에서 해제하지 않음으로써 읽기 작업들이 일관적임을 보장합니다. RCU는 오브젝트의 새 버전을 공개하고 읽는데,

```

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 9.20: Data Structure Publication (Unsafe)

그리고 예전 버전들의 정리 작업을 뒤로 미루어 한번에 처리하는데에 효과적이고 확장성 있는 메커니즘을 정의하고 사용합니다. 이런 메커니즘들은 작업을 읽기와 업데이트쪽 수행경로로 분산시키되 읽기 쪽 수행경로가 극단적으로 빠르게 하는데에 해저드 포인터와 유사한 복사와 규칙 완화를 골자로 하는 최적화 기술을 사용하지만, 읽기 쪽의 재시도는 필요 없게 합니다. 일부 경우에는 (CPU를 뺏기지 않는 커널들), RCU의 읽기 쪽 기능들은 아예 오버헤드가 없습니다.

Quick Quiz 9.19: 하지만 Section 9.4의 시퀀스 락 역시 읽기 쓰레드들과 업데이트 쓰레드들이 동시에 일을 할 수 있도록 하지 않던가요? ■

이는 “RCU는 정확히 무엇인가?” 하는 질문과, 아마도 “RCU는 어떻게 동작할 수 있는가?” 하는 질문을 이끌어낼 수 있을 겁니다 (또는, 드물지 않게, RCU는 동작할 수 없을 것이라는 단정을). 이 문서는 이런 질문들을 기본적 관점에서부터 다룹니다; 뒤의 일부는 RCU를 사용법과 API 관점에서 살펴봅니다. 마지막 부분은 또한 참고할 문서 목록을 포함합니다.

RCU는 세개의 기본적 메커니즘으로 만들어지는데, 첫번째는 항목 삽입에 사용되고, 두번째 것은 항목 삭제에, 그리고 세번째 것은 읽기 쓰레드들이 동시에의 항목 추가와 삭제에 문제 없이 동작하도록 하는데 사용됩니다. Section 9.5.2.1은 항목 추가를 위한 publish-subscribe 메커니즘을 설명하고, Section 9.5.2.2에서는 먼저 시작된 RCU 읽기 쓰레드들을 어떻게 기다려서 항목 삭제가 가능하게 하는지 설명하며, Section 9.5.2.3에서는 최근에 업데이트된 오브젝트들의 여러 버전들을 어떻게 관리해서 동시에의 항목 추가와 삭제를 가능하게 하는지 설명합니다. 마지막으로, Section 9.5.2.4에서는 RCU 기본사항을 요약합니다.

9.5.2.1 Publish-Subscribe Mechanism

RCU의 핵심 요소 중 하나는 데이터가 동시에 수정되고 있는데도 불구하고 안전하게 그 데이터를 읽을 수 있는 능력입니다. 동시에의 항목 삽입에 이런 능력을 제공하

기 위해, RCU 는 공개-구독 (publish-subscribe) 메커니즘이라 생각될 수 있는 방법을 상요합니다. 예를 들어, 초기에 NULL 인 전역 포인터 `gp` 가 새로 할당되고 초기화된 데이터 구조체로의 포인터로 수정되려 한다고 생각해 봅시다. Figure 9.20 에 보이는 코드 조각 (추가로 적절한 락킹을 포함해서) 이 이 목적으로 사용될 수 있을 것입니다.

안타깝게도, 컴파일러와 CPU 가 마지막 네개의 할당 문이 순서대로 수행하도록 강제하는 것이 전혀 없습니다. `gp` 로의 할당이 `p` 필드들의 초기화 전에 일어난다면, 동시 수행중인 읽기 작업들은 이 초기화되지 않은 값들을 볼 수 있을 겁니다. 이것들이 순서를 지키도록 하기 위해 메모리 배리어들이 필요합니다만, 메모리 배리어들은 사용하기가 어렵기로 악명높습니다. 따라서 그것들을 공개 의미를 갖는 `rcu_assign_pointer()` 기능에 집어넣습니다. 그렇게 되면 마지막 네줄은 다음과 같이 될겁니다:

```
1 p->a = 1;
2 p->b = 2;
3 p->c = 3;
4 rCU_assign_pointer(gp, p);
```

이 `rcu_assign_pointer()` 는 새 구조체를 공개하고, 컴파일러와 CPU 가 `gp` 로의 할당이 `p` 로 참조되는 필드들의 할당 후에 수행하도록 강제할 겁니다.

하지만, 업데이트 작업에 순서를 맞추는 것만으로는 충분치 않은데, 읽기 작업도 적절하게 순서가 맞춰져야 하기 때문입니다. 다음과 같은 코드 조각의 예를 생각해 봅시다:

```
1 p = gp;
2 if (p != NULL) {
3     do_something_with(p->a, p->b, p->c);
4 }
```

이 코드 조각은 잘못된 순서에 문제가 없을 것처럼 보이지만, 안타깝게도 DEC Alpha CPU [McK05a, McK05b] 와 값을 예측하는 컴파일러 최적화는, 모든 안믿든, `p->a`, `p->b`, 그리고 `p->c` 가 `p` 의 값 전에 메모리로부터 가져와질 수 있게 할 수 있습니다. 이런 현상은 컴파일러가 `p` 의 값을 추측하고 `p->a`, `p->b`, 그리고 `p->c` 값을 가져온 후에 그 추측이 맞았는지 보기 위해 `p` 의 실제 값을 가져오는, 컴파일러의 값 추측 최적화의 경우에서 보기 가장 쉬울 겁니다. 이런 종류의 최적화는 상당히 공격적이고 미친 행위 같지만, 프로파일 기반의 최적화의 문맥에서는 실제로 일어나는 일입니다.

분명히, 우리는 컴파일러와 CPU 로부터 이런 종류의 야바위질을 막아야 합니다. `rcu_dereference()` 기능은 이런 목적을 위해 필요한 어떤 메모리 배리어 인스트럭션과 컴파일러 지시어들을 사용합니다.⁸

⁸ 리눅스 커널에서, `rcu_dereference()` 는 volatile 캐스팅으로

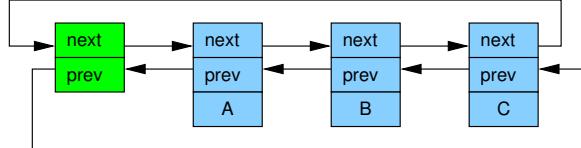


Figure 9.21: Linux Circular Linked List

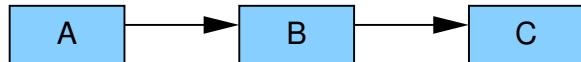


Figure 9.22: Linux Linked List Abbreviated

```
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();
```

따라서 `rcu_dereference()` 함수는 특정 포인터로 주어지는 값에 대한 구독으로, 뒤따르는 `dereference` 오퍼레이션들은 해당 포인터를 공개한, 연관된 `rcu_assign_pointer()` 오퍼레이션 전에 발생한 초기화 작업의 결과들을 보게 될 것이 보장된다고 이해될 수 있습니다. `rcu_read_lock()` 과 `rcu_read_unlock()` 함수 호출들은 반드시 필요합니다: 이것들은 RCU read-side 크리티컬 섹션을 정의합니다. 이것들의 목적인 Section 9.5.2.2 에서 설명됩니다만, 이것들은 결코 스피닝하거나 블락킹 하지 않고, `list_add_rcu()` 가 동시에 수행되는 것을 막지도 않습니다. 사실, `CONFIG_PREEMPT` 옵션이 켜져있지 않은 커널에서 이것들은 아무 코드도 생성하지 않습니다.

이론상으로는 `rcu_assign_pointer()` 와 `rcu_dereference()` 는 상상할 수 있는 RCU 로 보호되는 데이터 구조는 얼마든지 만들 수 있지만, 실제로는 고차원의 방법을 사용하는게 나은 경우가 많이 있습니다. 그런 이유로, `rcu_assign_pointer()` 와 `rcu_dereference()` 함수들이 리눅스에 있는 리스트 조정 API 의 특별한 RCU 사용 버전에 내장되어 있습니다. 리눅스는 이중 링크드 리스트의 두가지 버전을 가지고 있는데, 순환 형태의 `struct list_head` 와 선형의 `struct hlist_head/struct hlist_node` 쌍입니다.

로 구현되고, DEC Alpha 에서는 메모리 배리어 인스트럭션으로 구현됩니다. C11 과 C++11 표준에서는 `memory_order_consume` 이 `rcu_dereference()` 지원을 제공하기 위한 의도로 만들어졌습니다만, 이를 `native`로 구현한 컴파일러는 아직 없습니다. (컴파일러들은 대신 `memory_order_consume` 을 `memory_order_acquire` 로 강화시켜서, 약한 순서 규칙의 시스템에서는 필요없는 메모리 배리어 인스트럭션을 만들합니다.)

```

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 9.23: RCU Data Structure Publication

다. 앞의 버전은 Figure 9.21에 그려져 있는데, (왼쪽의) 초록 상자는 리스트 헤더를 나타내고 (오른쪽의 세개의) 파란 박스들은 리스트의 원소들을 의미합니다. 이 방법은 다루기가 힘들기 때문에 Figure 9.22에 보인 것처럼 헤더 없이 (파란) 원소들만을 보이는 형태로 간략화해서 나타낼 수 있습니다.

이 링크드 리스트에 포인터 공개 예제의 기법을 적용하는 것은 Figure 9.23에 보여진 코드와 같은 형태로 귀결될 겁니다.

Line 15는 여러개의 `list_add_rcu()`가 동시에 수행되는 것을 막기 위해 어떤 다른 동기화 메커니즘(가장 흔하게는 어떤 종류의 락)을 사용해야만 합니다. 하지만, 그런 동기화는 이 `list_add()`의 수행을 RCU 읽기 작업들과 동시에 수행되는 것을 못하게 하지는 않습니다.

RCU로 보호되는 리스트를 구독하는 행위는 간단합니다:

```

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();

```

앞의 `list_add_rcu()` 함수는 원소를 공개하고, 특정 리스트의 헤드에 집어넣고, 이에 연관된 `list_for_each_entry_rcu()` 호출이 정상적으로 같은 원소를 구독하게 될것을 보장합니다.

Quick Quiz 9.20: `list_add_rcu()`와 정확히 똑같은 시간에 `list_for_each_entry_rcu()`이 수행되면 segfault가 날 수 있을 것 같은데, 이걸 무엇이 방지해 주나요? ■

리눅스의 다른 이중 링크드 리스트인 `hlist`는 선형 리스트인데, 이는 헤더로의 포인터만이 필요하지 Figure 9.24에 보여진 순환 형태의 리스트처럼 두개의 포인터가 필요하진 않습니다. 따라서, `hlist`의 사용은 해시 버킷 배열들이나 커다란 해시 테이블에서는 메모리 사용량을 반으로 줄일 수 있습니다. 앞에서와 같이, 이

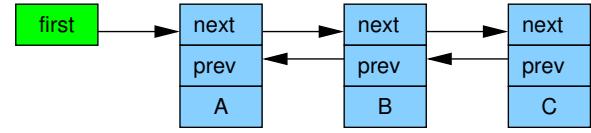


Figure 9.24: Linux Linear Linked List

```

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 9.25: RCU hlist Publication

형태는 다루기가 까다로우므로, `hlist`들은 Figure 9.22에 보인 것과 같은 형태로 간략화 될 겁니다.

새로운 원소를 RCU로 보호되는 `hlist`에 공개하는 건 as shown in Figure 9.25에 보여진 것처럼, 순환형 리스트에서 했던 것과 상당히 유사합니다.

앞에서와 같이, line 15는 예를 들면 락과 같은, 어떤 종류의 동기화 메커니즘으로 보호되어야만 합니다.

RCU로 보호되는 `hlist`를 구독하는 행위 역시 순환형 리스트에서와 비슷합니다:

```

1 rCU_read_lock();
2 hlist_for_each_entry_rcu(p, head, list) {
3     do_something_with(p->a, p->b, p->c);
4 }
5 rCU_read_unlock();

```

RCU 공개와 구독 기능들의 집합이 Table 9.1에 “구독취소” 또는 철회를 위한 추가적인 기능들과 함께 표시되어 있습니다.

`list_replace_rcu()`, `list_del_rcu()`, `hlist_replace_rcu()`, and `hlist_del_rcu()` API들이 복잡도를 더함을 알아두세요. 교체되거나 삭제된 데이터 원소를 메모리에서 해제하는데 안전한 시점은 언제일까요? 자세히 들어가서, 모든 읽기 작업들이 특정 데이터 원소로의 레퍼런스들을 해제한 시점을 어떻게 하면 알 수 있을까요?

이 질문들은 다음의 섹션에서 다루어집니다.

Category	Publish	Retract	Subscribe
Pointers	<code>rcu_assign_pointer()</code>	<code>rcu_assign_pointer(..., NULL)</code>	<code>rcu_dereference()</code>
Lists	<code>list_add_rcu()</code> <code>list_add_tail_rcu()</code> <code>list_replace_rcu()</code>	<code>list_del_rcu()</code>	<code>list_for_each_entry_rcu()</code>
Hlists	<code>hlist_add_after_rcu()</code> <code>hlist_add_before_rcu()</code> <code>hlist_add_head_rcu()</code> <code>hlist_replace_rcu()</code>	<code>hlist_del_rcu()</code>	<code>hlist_for_each_entry_rcu()</code>

Table 9.1: RCU Publish and Subscribe Primitives

9.5.2.2 Wait For Pre-Existing RCU Readers to Complete

가장 기본적인 형태에서, RCU 는 일들이 끝나기를 기다리는 방법입니다. 물론, RCU 외에도 일들이 끝나길 기다리는 훌륭한 방법들이 여럿 있는데, 레퍼런스 카운트, reader-writer lock, 이벤트 등등이 포함됩니다. RCU 의 커다란 장점은 각각의 (대략) 20,000 개의 서로 다른 일들을 명시적으로 그 모든 것들을 각각 정보를 쫓아가지 않고, 성능 하락, 확장성 제한, 복잡한 데드락 시나리오, 그리고 명시적으로 정보 쫓는 방법에서 필연적인 메모리 누수 문제들을 걱정할 필요 없이 기다릴 수 있다는 겁니다.

RCU 의 경우에, 기다려고 있는 일들은 “RCU read-side 크리티컬 섹션” 이라 불립니다. 하나의 RCU read-side 크리티컬 섹션은 `rcu_read_lock()` 함수로 시작되고, 그에 연관되는 `rcu_read_unlock()` 함수로 종료됩니다. RCU read-side 크리티컬 섹션들은 중첩될 수 있고, 어떤 코드든 그 코드가 명시적으로 블락하거나 잡들지 않는 한 (SRCU [McK06b] 라고 불리는 특별한 형태의 RCU 가 SRCU read-side 크리티컬 섹션 내에서의 일반적인 잡들기를 가능하게 하긴 하지만), 그 안에 얼마든지 들어갈 수 있습니다. 이런 규칙에 동의한다면, 코드에서 원하는 부분이라면 어떤 부분이든 완료되기 를 기다리는데에 RCU 를 사용할 수 있습니다.

RCU 는 언제 이런 기다리는 중인 일들이 종료되었는지를 간접적으로 판단해내는 것으로 이 기능을 구현합니다 [McK07f, McK07a].

자세히 말하자면, Figure 9.26 에 보여진 것처럼, RCU 는 전부터 존재했던 RCU read-side 크리티컬 섹션들이 그 크리티컬 섹션들에서 수행되는 메모리 오퍼레이션 등이 완전히 끝나기를 기다리는 한가지 방법입니다. 하지만, 주어진 grace period 의 시작 후에 시작된 RCU read-side 크리티컬 섹션들은 그 grace period 의 종료 이후까지 수행될 수도 있음을 기억해 두십시오.

다음의 슈도코드는 RCU 를 이용해 읽기 작업들을 기다리는 알고리즘들의 기본적 형태를 보입니다:

1. 링크드 리스트에서 한 원소를 바꿔치기 하거나 하는 식으로 변경을 만듭니다.

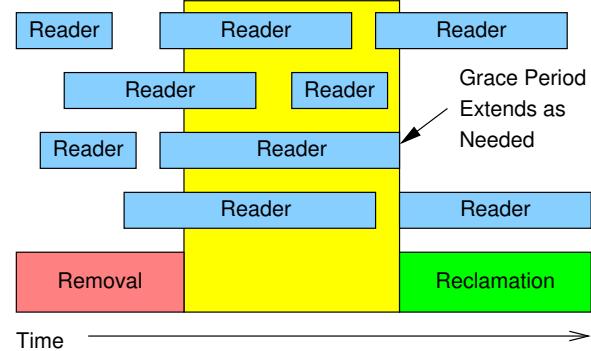


Figure 9.26: Readers and RCU Grace Period

2. 전부터 존재했던 RCU read-side 크리티컬 섹션들이 완전히 종료되길 기다립니다 (예를 들어, `synchronize_rcu()` 기능이나 그 비동기 버전으로 다음 grace period 가 끝난 후에 특정 함수를 호출해주는 `call_rcu()` 를 이용해서). 여기서의 핵심은 뒤이어지는 RCU read-side 크리티컬 섹션들은 이제는 제거된 원소로의 레퍼런스를 얻을 수 없다는 점입니다.
3. 예를 들어 앞서 교체된 원소를 메모리에서 해제하는 식으로 정리를 합니다.

Figure 9.27 에 보여진 코드 조각은 Section 9.5.2.1 에서 가져와진 것으로, 이 프로세스를 보여주는데, 필드 `a` 는 검색을 위한 키로 사용됩니다.

Line 19, 20, 21 은 앞에서 이야기한 세개의 스텝을 보입니다. Line 16-19 이 RCU (“read-copy update”) 에 그 이름을 줍니다: 동시에 `read` 를 허용하면서, line 16 은 `copy` 를 하고 line 17-19 에서 실제 `update` 를 합니다.

Section 9.5.1 에서 이야기된 것처럼, `synchronize_rcu()` 기능은 매우 간단할 수 있습니다 (“장난감” RCU 구현을 더 보기 위해선 Section 9.5.5 을 참고하시기 바랍니다). 하지만, 상품 수준의 구현들은 복잡하고 희귀한 경우들을 처리해야 하고 강력한 최적화를 포함해

```

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 9.27: Canonical RCU Replacement Example

야 하는데, 둘 다 상당한 복잡도가 생기게 하고 맙니다. `synchronize_rcu()`의 간단한 개념적 구현이 있음을 알게 된 건 좋지만, 다른 질문들이 남습니다. 예를 들어, RCU 읽기 작업들은 동시에 업데이트 되는 리스트를 돌아다니면서 정확히 뭘 보게 되는 걸까요? 이 질문을 다음 섹션에서 다루도록 합니다.

9.5.2.3 Maintain Multiple Versions of Recently Updated Objects

이 섹션은 RCU 가 동기화로부터 자유로운 읽기 작업들을 허용하기 위해 리스트의 여러 버전들을 어떻게 관리하는지 보입니다. 주어진 읽기 쓰레드에 의해 레퍼런스 될 수도 있는 원소가 해당 읽기 쓰레드가 자신의 RCU read-side 크리티컬 섹션을 유지하고 있는 동안에도 손상되지 않은채로 어떻게 유지될 수 있는지를 두개의 예제로 보일 겁니다. 첫번째 예제는 리스트 원소의 삭제를 보이고, 두번째 예제는 원소의 교체를 보이도록 하겠습니다.

Example 1: Maintaining Multiple Versions During Deletion 이제 Section 9.5.1에서의 원소 삭제 예제를 다시 들여다 보되, 이번에는 그 아래의 RCU에 있는 기본적인 개념에 대한 확실한 이해와 함께입니다. 이 새로운 버전의 삭제 예제를 시작하기 위해, Figure 9.27의 line 11-21을 다음과 같이 보이게 수정할 겁니다:

```

1 p = search(head, key);
2 if (p != NULL) {
3     list_del_rcu(&p->list);
4     synchronize_rcu();
5     kfree(p);
6 }

```

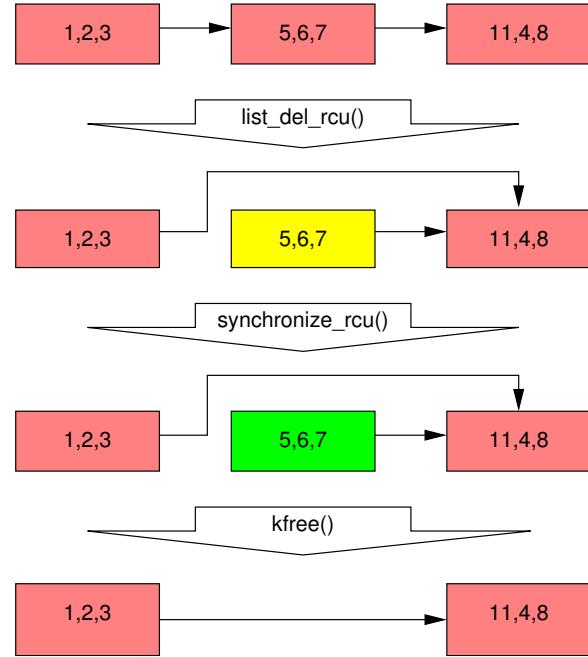


Figure 9.28: RCU Deletion From Linked List

이 코드는 Figure 9.28에 보인 것처럼 리스트를 업데이트 할 겁니다. 각 원소의 세개의 숫자는 필드 a, b, c의 값들을 각각 나타냅니다. 빨간색으로 칠해진 원소들은 RCU 읽기 쓰레드들이 그것들로의 레퍼런스를 가질 수 있음을 나타내는데, 따라서 다이어그램의 꼭대기에 있는 최초의 상태에서는 모든 원소들이 빨간색으로 칠해져 있습니다. 뒤로의 포인터들과 리스트의 tail로부터 head로의 링크는 가독성을 위해 삭제해 둔점을 알아 두시기 바랍니다.

Line 3에서의 `list_del_rcu()`가 완료된 후에, 5, 6, 7 원소는 Figure 9.28의 두번째 줄에 보여진 것처럼 리스트에서 삭제되어 있습니다. 읽기 쓰레드들은 업데이트 쓰레드들과 직접적으로 동기화를 하지 않으므로, 읽기 쓰레드들은 동시에 이 리스트를 스캔하고 있을 수 있습니다. 이런 동시의 읽기 쓰레드들은 이번에 삭제된 원소를 탐색에 따라서는 볼 수도, 보지 않을 수도 있습니다. 하지만, 이번에 삭제된 원소로의 포인터를 가져온 후에 한동안 지연된 (ex: 인터럽트나 ECC 메모리 에러, 또는 CONFIG_PREEMPT_RT 커널에서라면 preemption으로 인해서) 읽기 쓰레드들은 이 삭제로부터 상당한 시간이 흐른 후에도 이 리스트의 과거 버전을 보게 될 수도 있습니다. 따라서, 이 리스트의 두개의 버전이 있는 셈인데, 하나는 5, 6, 7 원소를 가지고 있고 또 다른 하나는 가지고 있지 않습니다. 이 그림에서

두번째 줄의 5, 6, 7 원소는 노란색으로 칠해져 있는데, 오래된 읽기 쓰레드들은 여전히 레퍼런스를 가지고 있을 수 있지만, 새로 시작된 읽기 쓰레드들은 그로의 레퍼런스를 얻을 수 없음을 의미합니다.

읽기 쓰레드들은 각자의 RCU read-side 크리티컬 섹션들에서 빠져나온 후에는 element 5, 6, 7로의 레퍼런스를 얻을 수 없음을 기억하기 바랍니다. 따라서, 일단 line 4에서의 `synchronize_rcu()`가 완료되면, 모든 앞서 존재하던 읽기 쓰레드들은 완료된 것이 보장되므로, 이 원소를 레퍼런스 하는 읽기 쓰레드들은 존재할 수 없어지는데 Figure 9.28의 세번째 줄에 녹색으로 색칠됨으로써 이 상황이 나타내어져 있습니다.

이 시점에서, 5, 6, 7 원소는 Figure 9.28의 마지막 줄에 나타나 있듯이 안전하게 메모리 해제될 수 있습니다. 이 시점에서, 원소 5, 6, 7의 삭제가 완료되었습니다. 다음 섹션에서는 교체를 다룹니다.

Example 2: Maintaining Multiple Versions During Replacement 교체의 예제 설명을 위해, 여기 Figure 9.27 예제의 마지막 몇줄의 코드를 붙여넣습니다:

```

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);

```

리스트의 초기의 상태는 p 포인터를 포함해서 삭제 예제와 동일한데, Figure 9.29의 첫번째 줄에 보여져 있습니다.

앞에서와 마찬가지로, 각 원소의 세개의 숫자는 필드 a, b, c를 각각 나타냅니다. 빨간 원소들은 읽기 쓰레드들에 의해 레퍼런스 될 수도 있고, 읽기 쓰레드들은 업데이트 쓰레드들과 직접 동기화를 하지 않기 때문에 읽기 쓰레드들은 이 전체 교체 프로세스와 동시에 수행될 수 있습니다. 이번에도 뒤쪽으로의 포인터들과 리스트의 tail에서 head로의 링크는 가독성을 위해 제거되었음을 참고 바랍니다.

아래의 글은 5, 6, 7 원소를 5, 2, 3으로 모든 읽기 쓰레드가 이 두 값들 중 하나만 보도록 하면서 어떻게 교체해야 하는지 설명합니다.

Line 1은 교체할 원소를 `kmalloc()` 해서 Figure 9.29의 두번째 줄에 보여진 것과 같은 상태가 만들어지게 합니다. 이 시점에서는, 어떤 읽기 쓰레드도 이 새로 만들어진 원소로의 레퍼런스를 가질 수 없고 (녹색 색깔로 이를 나타냅니다), 이 원소는 아직 초기화되지 않았습니다 (물음표로 나타내어집니다).

Line 2에서는 예전 원소의 값을 새 원소로 복사해서 Figure 9.29의 세번째 줄에 보여진 상태를 만듭니다. 이 새로 만들어진 원소는 여전히 읽기 쓰레드들에 의해 레

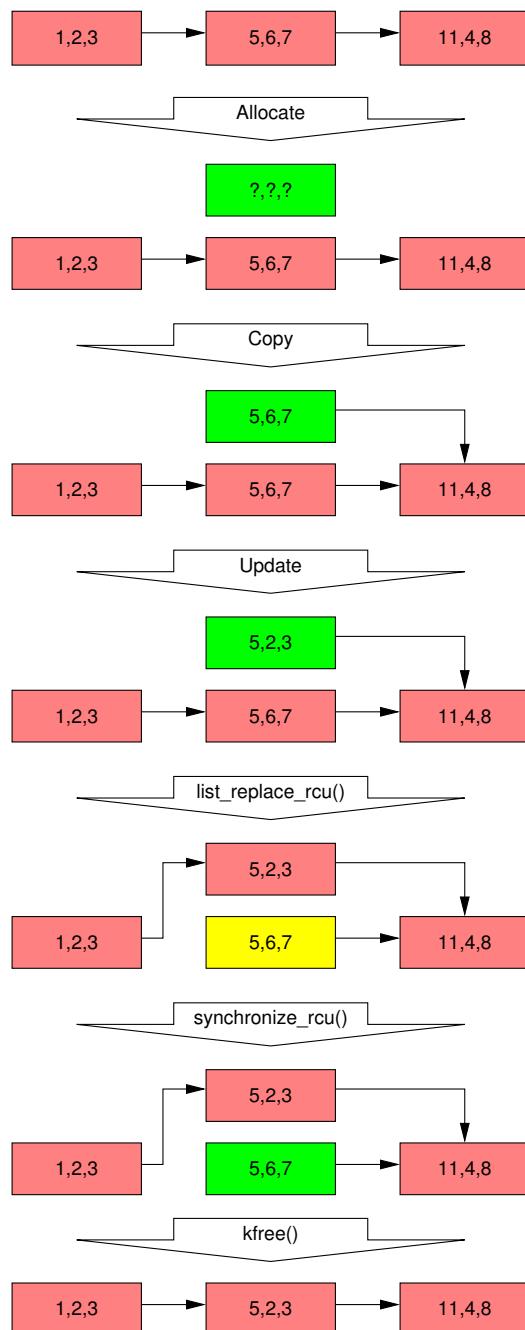


Figure 9.29: RCU Replacement in Linked List

퍼런스될 수는 없지만, 이제 초기화는 되었습니다.

Line 3 는 Figure 9.29 의 네번째 줄에 보여진 것처럼 $q \rightarrow b$ 를 값 “2”로, 그리고 line 4 는 $q \rightarrow c$ 를 값 “3”으로 바꿉니다.

이제, line 5 는 교체를 행해서 새 원소가 읽혀질 수 있게 만드는데, 따라서 Figure 9.29 의 다섯번째 줄에 빨간색으로 칠해져 있습니다. 이 시점에서는 앞에서와 같이, 두가지 버전의 리스트가 존재합니다. 전부터 있던 읽기 쓰레드들은 5, 6, 7 원소를 볼 수 있지만 (따라서 노란색으로 칠해져 있습니다), 새 읽기 쓰레드들은 그대신 5, 2, 3 원소를 볼겁니다. 하지만 어떤 읽기 쓰레드들이든 제대로 형태를 갖춘 리스트만을 볼 것이 보장됩니다.

Line 6 의 `synchronize_rcu()` 가 리턴한 후에는, grace period 는 지나갔고, 따라서 `list_replace_rcu()` 전에 시작된 모든 읽기는 완료된 상태입니다. 특히, 5, 6, 7 원소로의 레퍼런스를 가지고 있는 모든 읽기 쓰레드들은 각자의 RCU read-side 크리티컬 섹션들을 빠져나왔음이 보장되고, 따라서 레퍼런스를 계속해서 잡고 있을 수 없습니다. 따라서, 과거의 원소로의 레퍼런스를 여전히 잡고 있는 읽기 쓰레드는 더이상 있을 수 없는데, 이 상황이 Figure 9.29 의 여섯번째 줄에 초록색으로 나타내어져 있습니다. 읽기 쓰레드들의 관점에서 리스트의 단일 버전만이 존재하는데, 예전 원소는 새 원소로 바뀌어 있습니다.

Line 7에서의 `kfree()` 후에 이 리스트는 Figure 9.29 의 마지막 줄에 보여진 것처럼 됩니다.

RCU 는 교체의 경우에 의해서 이름지어졌다는 사실에도 불구하고, 리눅스 커널 안에서의 RCU 사용 예의 대다수는 Section 9.5.2.3. 의 간단한 삭제의 케이스에 기반해 있습니다.

Discussion 이 예제들은 모든 업데이트 오퍼레이션들이 뮤텍스가 잡혀 있다고 가정을 하고 있는데, 이 말은 한번에 리스트의 버전은 최대 두개까지만 존재할 수 있음을 의미합니다.

Quick Quiz 9.21: 리스트의 버전이 두개보다 많을 수 있도록 하기 위해서는 삭제 예제를 어떻게 수정해야 할까요? ■

Quick Quiz 9.22: 어떤 시점에 하나의 리스트는 RCU 버전들을 몇개까지 가질 수 있을까요? ■

이 일련의 이벤트들은 RCU 업데이트들이 어떻게 여러 버전들을 사용해 동시에 수행중인 읽기 작업들에도 불구하고 안전하게 변경을 처리하는지 보입니다. 물론, 일부 알고리즘들은 여러 버전들을 잘 처리하지 못합니다. 그런 알고리즘을 RCU 에 적용하는 테크닉 [McK04] 들이 있지만, 이것들은 이 섹션의 범위를 넘어섭니다.

Mechanism RCU Replaces	Section
Reader-writer locking	Section 9.5.3.2
Restricted reference-counting mechanism	Section 9.5.3.3
Bulk reference-counting mechanism	Section 9.5.3.4
Poor man's garbage collector	Section 9.5.3.5
Existence Guarantees	Section 9.5.3.6
Type-Safe Memory	Section 9.5.3.7
Wait for things to finish	Section 9.5.3.8

Table 9.2: RCU Usage

9.5.2.4 Summary of RCU Fundamentals

이 섹션에서는 RCU 기반 알고리즘들의 세가지 기본 컴포넌트들을 설명했습니다:

1. 새로운 데이터의 추가를 위한 공개-구독 메커니즘,
2. 전부터 존재했던 RCU 읽기 쓰레드들이 종료되기 위해 기다리는 방법, 그리고
3. 동시에 수행중인 RCU 읽기 쓰레드들에 피해를 주거나 지나치게 대기하도록 만들지 않고 변화를 가할 수 있도록 여러 버전들을 관리하는 방법.

Quick Quiz 9.23: `rcu_read_lock()` 과 `rcu_read_unlock()` 함수는 스핀하지도 블락하지도 않는 데 어떻게 RCU 업데이트 쓰레드들이 RCU 읽기 쓰레드들을 대기시킬 수가 있나요? ■

이 세개의 RCU 컴포넌트들은 데이터가 동시에 수행되는 읽기 쓰레드들에 상관 없이 업데이트 되도록 하고, 놀랍도록 다양한, 다른 종류의 RCU 기반의 알고리즘들을 구현하는 다른 방법들로 조합될 수 있는데, 그 중 일부는 다음 섹션에서 설명하겠습니다.

9.5.3 RCU Usage

이 섹션은 “왜 RCU 인가?”라는 질문에 대해 RCU 가 사용될 수 있는 경우에 대한 관점에서 답변해 보겠습니다. RCU 는 일부 존재하는 메커니즘을 대체하는데에 가장 자주 사용되기 때문에, Table 9.2 에 보여진 것처럼 그런 메커니즘들과의 관계에 대한 점을 중심적으로 알아보겠습니다. 이 테이블에 나열된 섹션들을 뒤이어서, Section 9.5.3.9 에서는 요약을 제공합니다.

9.5.3.1 RCU for Pre-BSD Routing

Figure 9.30 와 9.31 는 RCU 로 보호되는 Pre-BSD 라우팅 테이블을 위한 코드를 보이고 있습니다 (`route_rcu.c`). 앞의 것은 데이터 구조들과 `route_lookup()` 을, 뒤의 것은 `route_add()` 와 `route_del()` 을 위한 코드입니다.

```

1 struct route_entry {
2     struct rCU_head rh;
3     struct cds_list_head re_next;
4     unsigned long addr;
5     unsigned long iface;
6     int re_freed;
7 };
8 CDS_LIST_HEAD(route_list);
9 DEFINE_SPINLOCK(routelock);
10
11 unsigned long route_lookup(unsigned long addr)
12 {
13     struct route_entry *rep;
14     unsigned long ret;
15
16     rCU_read_lock();
17     cds_list_for_each_entry_rcu(rep, &route_list,
18                                 re_next) {
19         if (rep->addr == addr) {
20             ret = rep->iface;
21             if (ACCESS_ONCE(rep->re_freed))
22                 abort();
23             rCU_read_unlock();
24             return ret;
25         }
26     }
27     rCU_read_unlock();
28     return ULONG_MAX;
29 }

```

Figure 9.30: RCU Pre-BSD Routing Table Lookup

Figure 9.30에서, line 2는 RCU reclamation에 사용되는 `->rh` 필드를 보이고, line 6는 해제 후 사용 검사를 위한 `->re_freed` 필드를 보이며, line 16, 17, 23, 그리고 27은 RCU read-side 보호 코드를 보이고, line 21과 22에서 해제 후 사용 여부 검사를 합니다. Figure 9.31에서, line 12, 14, 31, 36, 그리고 41에서 update-side 락킹을 하며, line 13과 35에서 RCU update-side 보호를 하고, line 37에서 grace period가 하나 지나간 후에 `route_cb()`를 호출되게 하고, line 18-25에서 `route_cb()`를 정의합니다. 이는 똑바로 동작하는 구현을 위해 추가되는 최소한의 코드입니다.

Figure 9.32는 read-only 워크로드에서의 성능을 보입니다. RCU는 상당히 잘 확장되어서, 이상적 성능에 가까운 성능을 보입니다. 하지만, 이 데이터는 userspace RCU의 RCU_SIGNAL 사용 버전 [Des09, MDJ13c]을 사용해서 생성된 것으로, `rCU_read_lock()`과 `rCU_read_unlock()`이 약간의 코드를 생성합니다. `rCU_read_lock()`과 `rCU_read_unlock()`에 아무 코드도 생성하지 않는 QSBR 버전의 RCU를 사용한다면 어떻게 될까요? (Section 9.5.1를 참고하고, RCU QSBR에 대한 논의를 위해선 Figure 9.19를 참고하세요.)

그에 대한 답이 RCU QSBR 결과를 RCU와 ideal 사이에 보여주는 Figure 9.33에 있습니다. RCU QSBR은 바랬던 대로 이상적인 동기화를 아예 하지 않는 워크로드와 거의 동일한 성능과 확장성을 갖습니다.

Quick Quiz 9.24: RCU QSBR은 왜 이상적 결과와

```

1 int route_add(unsigned long addr,
2                 unsigned long interface)
3 {
4     struct route_entry *rep;
5
6     rep = malloc(sizeof(*rep));
7     if (!rep)
8         return -ENOMEM;
9     rep->addr = addr;
10    rep->iface = interface;
11    rep->re_freed = 0;
12    spin_lock(&routelock);
13    cds_list_add_rcu(&rep->re_next, &route_list);
14    spin_unlock(&routelock);
15    return 0;
16 }
17
18 static void route_cb(struct rCU_head *rhp)
19 {
20     struct route_entry *rep;
21
22     rep = container_of(rhp, struct route_entry, rh);
23     ACCESS_ONCE(rep->re_freed) = 1;
24     free(rep);
25 }
26
27 int route_del(unsigned long addr)
28 {
29     struct route_entry *rep;
30
31     spin_lock(&routelock);
32     cds_list_for_each_entry(rep, &route_list,
33                             re_next) {
34         if (rep->addr == addr) {
35             cds_list_del_rcu(&rep->re_next);
36             spin_unlock(&routelock);
37             call_rcu(&rep->rh, route_cb);
38             return 0;
39         }
40     }
41     spin_unlock(&routelock);
42     return -ENOENT;
43 }

```

Figure 9.31: RCU Pre-BSD Routing Table Add/Delete

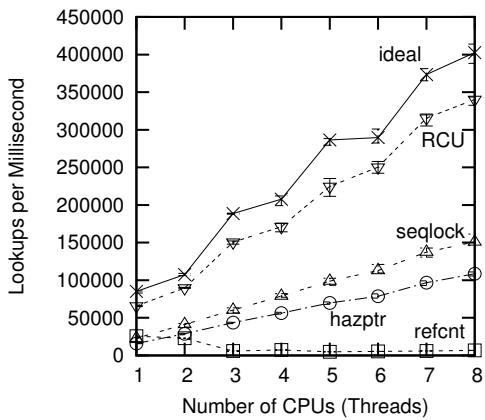


Figure 9.32: Pre-BSD Routing Table Protected by RCU

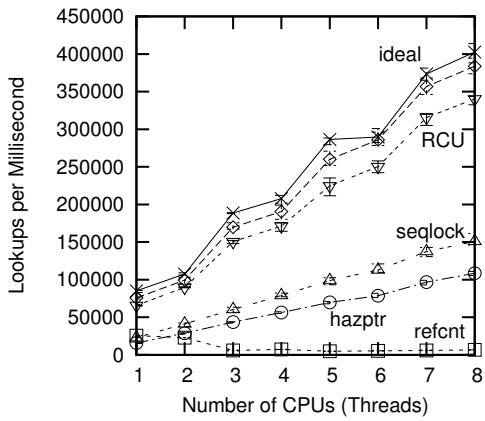


Figure 9.33: Pre-BSD Routing Table Protected by RCU QSBR

동일한 결과를 보이지 않는 거죠? ■

Quick Quiz 9.25: RCU QSBR의 read-side 성능이 이렇게 좋은데, 왜 다른 종류의 userspace RCU를 신경 써야 하죠? ■

9.5.3.2 RCU is a Reader-Writer Lock Replacement

리눅스 커널에서의 가장 흔한 RCU의 사용은 읽기기 대부분인 상황들에서의 reader-writer 락킹의 대체입니다. 더도 아니고 덜도 아니고, 이런 RCU의 사용은 제제는 처음부터 곧장 그래야 할 것처럼 보이진 않았는데, 실제로 저는 1990년대 초기에 범용의 RCU 구현을 만들기 전에 경량의 reader-writer 락 [HW92]⁹을 구현하려 했습니다. 제가 해당 경량 reader-writer 락을 위해 상상했던

⁹ 2.4 리눅스 커널의 brlock과 더 최신의 리눅스 커널 버전들의 lglock과 유사합니다

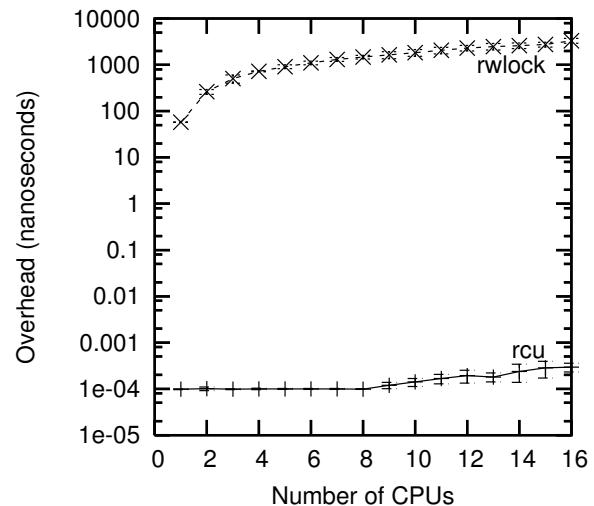


Figure 9.34: Performance Advantage of RCU Over Reader-Writer Locking

모든 각각의 사용 케이스들은 대신 RCU로 구현되었습니다. 사실, 그건 그 경량 reader-writer 락이 처음으로 사용되기보다 3년도 더 전의 일입니다. 이봐요, 과연 제가 절바보라고 생각했을꺼라고 생각해요?

RCU와 reader-writer 락킹 사이의 핵심적인 유사점은 둘 다 병렬로 수행될 수 있는 read-side 크리티컬 섹션들을 가지고 있다는 점입니다. 사실, 어떤 경우에 있어서는, RCU API로 연관된 reader-writer 락 API 멤버들을 기계적으로 대체하는 것이 가능합니다. 하지만 먼저, 왜 그려려 하나요?

RCU의 장점들은 성능, 데드락에의 내성, 그리고 리얼타임 대기시간을 포함합니다. 물론, RCU에도 한계들이 있는데, 읽기 쓰레드들과 업데이트 쓰레드들이 동시에 수행될 수 없고, 낮은 중요도의 RCU 읽기 쓰레드들이 grace period가 지나가길 기다리고 있는 높은 중요도의 쓰레드들을 블락시킬 수 있으며, 이 grace-period 대기시간은 수 밀리세컨드를 넘길 수 있다는 점등이 포함됩니다. 이런 장점들과 제한점들을 다음 섹션들에서 이야기 하겠습니다.

Performance RCU의 reader-writer 락킹에 대비한 읽기 작업 성능의 이점이 Figure 9.34에 그려져 있습니다.

Quick Quiz 9.26: 이게 뭐죠? 3GHz에서의 클락 시간이 300 피코세컨드가 넘는데 대체 어떻게 RCU는 100 펜토세컨드의 오버헤드를 갖는다고 제가 믿을 수 있을 거라고 생각하세요? ■

reader-writer 락킹은 단일 CPU 위에서 RCU보다 열 배가량 느리고 16개의 CPU 위에서는 100배가 더 느리

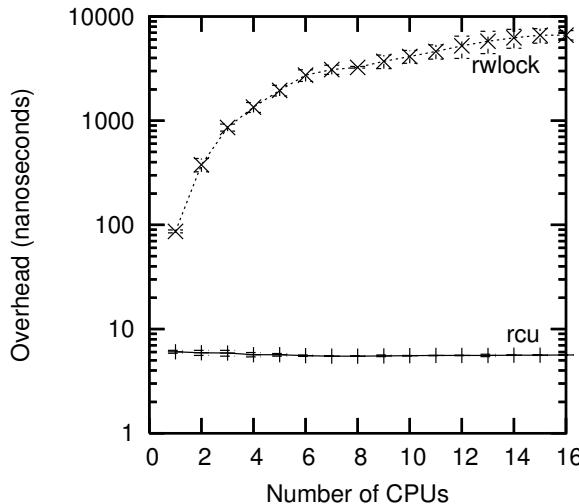


Figure 9.35: Performance Advantage of Preemptible RCU Over Reader-Writer Locking

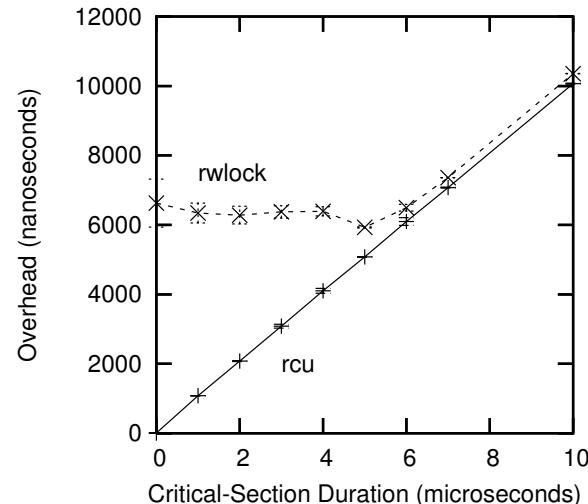


Figure 9.36: Comparison of RCU to Reader-Writer Locking as Function of Critical-Section Duration

다는 점을 알아두세요. 반면, RCU 는 상당히 잘 확장됩니다. 두 케이스 모두, 여러바들은 양쪽으로 표준편차 만큼의 크기를 갖습니다.

더 완화된 그림은 CONFIG_PREEMPT 커널에서 볼 수 있을텐데, 그렇다 하더라도 Figure 9.35에서 볼 수 있듯이 RCU 는 여전히 reader-writer 락킹을 10배에서 1000 배까지 압도하는 모습을 보입니다. 많은 수의 CPU 들에서의 reader-writer 락킹의 오버헤드의 성능이 높은 가변성을 보세요. 그림의 여러바들은 양쪽으로 표준편차 만큼의 크기를 갖습니다.

물론, Figure 9.35에 보인 reader-writer 락킹의 낮은 성능은 비현실적이게 짧은 크리티컬 섹션들로 인해 상당히 과장되었습니다. RCU의 성능상 이점은 16-CPU 환경에서 y 축은 read-side 기능들의 오버헤드와 크리티컬 섹션의 오버헤드의 합을 나타내는 Figure 9.36에서 보인 것처럼, 크리티컬 섹션의 오버헤드가 늘어날수록 덜 뚜렷해집니다.

Quick Quiz 9.27: 크리티컬 섹션 오버헤드가 늘어나면 왜 rwlock 의 오버헤드와 가변성이 모두 줄어드는 거죠? ■

하지만, 이 관측은 여러 시스템 콜들이 (그리고 따라서 그것들이 포함하고 있는 모든 RCU read-side 크리티컬 섹션들도) 수 마이크로세컨드 내에 완료될 수 있다는 사실로 좀 완화되어야만 합니다.

또한, 다음 섹션에서 이야기 되겠지만, RCU read-side 기능들은 거의 전부 테드락에 내성을 가지고 있습니다.

Deadlock Immunity RCU 가 읽기가 대부분인 워크로드들에게 상당한 성능 이득을 제공하긴 하지만, 사실 RCU 를 만들게 된 첫번째 목적은 read-side 의 테드락에 대한 내성입니다. 이 내성은 RCU 의 read-side 기능들은 블락도, 스피닝도, 심지어 뒤로 돌아가기도 하지 않으며, 따라서 그것들의 수행 시간이 결정론적이라는 사실에서 기인합니다. 따라서 이것들이 테드락 사이클에 연관되는 것은 불가능합니다.

Quick Quiz 9.28: 이 테드락 내성에 어떤 예외가 있을까요, 그리고 만약 그렇다면, 어떤 일련의 이벤트들이 테드락을 이끌어 낼 수 있을까요? ■

RCU의 read-side 테드락 내성의 흥미로운 결론은 무조건적으로 RCU 읽기 쓰레드를 RCU 업데이트 쓰레드로 업그레이드 시키는게 가능하다는 것입니다. Reader-writer 락킹을 사용해서 그런 업그레이드를 하려 하면 테드락이 날 것입니다. RCU read-to-update 업그레이드를 하는 예제 코드 조각은 다음과 같습니다:

```

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();

```

do_update() 는 락의 보호 하에 실행되었다는 점, 그리고 RCU read-side 보호 하에 실행되었다는 점에 주의하세요.

RCU 의 데드락 내성의 또 다른 흥미로운 결론은 우선순위 역전 문제의 커다란 클래스에의 내성입니다. 예를 들어, 낮은 우선순위 RCU 읽기 쓰레드들은 높은 우선순위의 RCU 업데이트 쓰레드의 update-side 락을 잡는 걸 막을 수 없습니다. 유사하게, 낮은 우선순위 RCU 업데이트 쓰레드는 높은 우선순위 RCU 읽기 쓰레드들이 RCU read-side 크리티컬 섹션에 들어가는 걸 막을 수 없습니다.

Quick Quiz 9.29: 데드락과 우선순위 역전 모두에 내성이 있다고요??? 사실이라기엔 너무 좋은 이야기 같은데요. 제가 이게 가능하다는 걸 어떻게 믿을 수 있을까요? ■

Realtime Latency RCU read-side 기능들은 스피닝도 블락도 하지 않으므로, 훌륭한 리얼타임 대기시간들을 제공합니다. 추가적으로, 앞에서도 이야기했듯, 이 말은 RCU read-side 기능들과 락들에 관련된 우선순위 역전 문제에도 내성이 있음을 뜻합니다.

하지만, RCU 는 좀 더 미묘한 우선순위 역전 시나리오에는 취약한데, 예를 들어, -rt 커널에서 어떤 RCU grace period 가 종료되기를 기다리느라 블락되어 있는 높은 우선순위의 프로세스는 낮은 우선순위의 RCU 읽기 쓰레드들에 의해 블락될 수 있습니다. 이 문제는 RCU priority boosting [McK07c, GMTW08] 에 의해 해결될 수 있습니다.

RCU Readers and Updaters Run Concurrently
RCU 읽기 쓰레드들은 스피닝도 블락도 하지 않고, 업데이트 쓰레드들은 rollback이나 abort 비슷한 것을 하지 않기 때문에, RCU 읽기 쓰레드들과 업데이트 쓰레드들은 동시에 수행될 수 있습니다. 이는 RCU 읽기 쓰레드들은 낮은 데이터에 접근할 수 있고, 비일관적인 상태를 보게 될수 있어서, reader-writer 락킹에서 RCU 로의 변환이 간단하지 않을 것임을 의미합니다.

하지만, 놀랍도록 많은 상황에서 비일관성과 낮은 데이터는 문제가 되지 않습니다. 그런 고전적인 예는 네트워킹 라우팅 테이블입니다. 라우팅 업데이트는 시스템에 가해지는데 상당한 시간(몇초에서 십자어 몇분까지)을 필요로 할 수 있기 때문에, 시스템은 업데이트가 도착했을 때 상당한 시간동안은 패킷들을 잘못된 방향으로 보낼 수도 있습니다. 몇 밀리세컨드동안 잘못된 방향으로 업데이트를 보내는 건 일반적으로 문제가 되지 않습니다. 더욱이, RCU 업데이트 쓰레드들은 RCU 읽기 쓰레드들이 끝나기를 기다리지 않고 변경을 가하기 때문에, RCU 읽기 쓰레드들은 이 변경을 reader-writer 락킹의 읽기 쓰레드들보다 빠르게 볼 수 있게 되는데, Figure 9.37 에 이 점이 그려져 있습니다.

일단 업데이트가 도착하면, rwlock 쓰기 쓰레드는 마지막 읽기 쓰레드가 완료되기 전까지, 뒤따르는 읽기

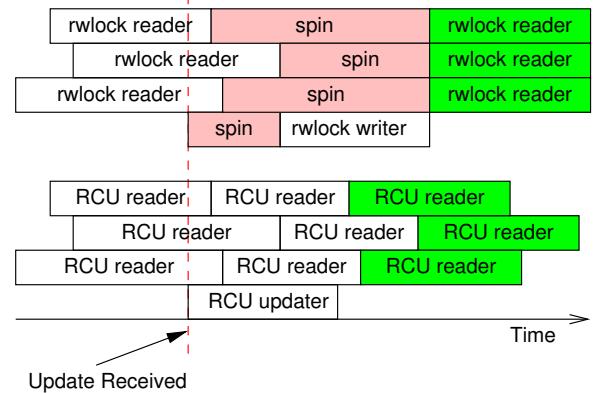


Figure 9.37: Response Time of RCU vs. Reader-Writer Locking

쓰레드들은 쓰기가 완료되기 전까지 진행될 수 없습니다. 하지만, 뒤의 읽기 쓰레드들은 새로운 값을 볼것이 보장되는데, 가장 오른쪽 박스들의 녹색 색깔로 표시되어 있습니다. 반면, RCU 읽기 쓰레드들과 업데이트 쓰레드들은 서로를 막지 않아, RCU 읽기 쓰레드들이 업데이트된 값을 더 빨리 볼 수 있게 합니다. 물론, RCU 업데이트 쓰레드와 겹쳐 수행되기 때문에, 업데이트 전에 시작된 세개의 읽기 쓰레드들을 포함해 모든 RCU 읽기 쓰레드들은 업데이트된 값을 보게 될수도 있을겁니다. 오로지 가장 오른쪽의 녹색으로 칠해진 RCU 읽기 쓰레드들만이 업데이트된 값을 볼 것이 보장됩니다.

Reader-writer 락킹과 RCU 는 단순히 다른 보장을 제공합니다. Reader-writer 락킹에서는, 쓰기 쓰레드보다 늦게 시작을 늦게 한 읽기 쓰레드는 모두 새 값을 볼 것이 보장되고, 쓰기 쓰레드가 스피닝 하는중에 시작되려 시도하는 읽기 쓰레드는 rwlock 의 읽기/쓰기 우선권 구현에 따라 새 값을 볼수도 못볼수도 있습니다. 반면에, RCU 에서는 업데이트 쓰레드가 완료된 후에 시작된 읽기 쓰레드는 모두 새 값을 볼것이 보장되고, 업데이트가 시작된 후에 완료된 모든 읽기 쓰레드는 타이밍에 따라 새 값을 볼수도 못볼수도 있습니다.

여기서의 핵심은, reader-writer 락킹이 컴퓨터 시스템에 국한해서는 실제로 일관성을 보장하지만, 이 일관성이 그 바깥 세계에서는 더 증가된 비일관성이라는 비용이 되는 상황이 존재한다는 것입니다. 달리 말해, reader-writer 락킹은 바깥 세계의 관점에서는 조용히 망가져버린 데이터의 비용으로 내부의 일관성을 얻습니다.

더도 아니고 덜도 아니고, 시스템에 국한된 비일관성과 망가진 데이터가 허용될 수 없는 상황은 존재합니다. 다행히도, 비일관성과 망가진 데이터를 방지하는

방법들이 여럿 [McK04, ACMS03] 있고, 일부 방법은 Section 9.2에서 논의한 레퍼런스 카운팅에 기반합니다.

Low-Priority RCU Readers Can Block High-Priority Reclaimers 리얼타임 RCU [GMTW08], SRCU [McK06b], 또는 QRCU [McK07e] (Section 12.1.4를 참고하세요)에서, preemption 당한 읽기 쓰레드는 grace period가 종료되는 걸 막을 것인데, 높은 우선순위 태스크가 해당 grace period가 완료되기를 기다리느라 블락되어 있더라도 그려할 것입니다. 리얼타임 RCU는 call_rcu()를 synchronize_rcu()로 대체하거나 2008년 초인 지금 시점까지는 아직 실험적 단계인 RCU priority boosting [McK07c, GMTW08]을 사용해서 이 문제를 해결 할 수 있습니다. SRCU와 QRCU에 priority boosting을 더하는게 필요할 수도 있겠지만, 그전에 분명한 실제 세계에서의 필요성이 보여져야 합니다.

RCU Grace Periods Extend for Many Milliseconds QRCU와 Section 9.5.5에 설명된 몇개의 “장난감” RCU 구현들의 예외가 있지만, RCU grace period들은 여러 밀리세컨드까지 늘어납니다. 사용 가능한 비동기적인 인터페이스 (call_rcu()와 call_rcu_bh())를 사용하는 방법을 포함해, 그런 긴 딜레이를 위험하지 않게 만드는 몇가지 테크닉들이 있긴 하지만, 이 상황은 RCU가 읽기가 대부분인 상황에서만 사용되어야 한다는 규칙의 주요 이유입니다.

Comparison of Reader-Writer Locking and RCU Code 적합한 경우에 reader-writer 락킹에서 RCU로의 변경은 위키피디아에서 가져온 [MPA⁺06] Figure 9.38, 9.39, 그리고 9.40에서처럼 매우 간단합니다.

Reader-writer 락킹을 RCU로 바꾸는 더 정교한 경우들은 이 문서의 범위 밖입니다.

9.5.3.3 RCU is a Restricted Reference-Counting Mechanism

RCU read-side 크리티컬 섹션이 진행중인 동안은 grace period들이 완료될 수 없기 때문에, RCU read-side 기능들은 제한적인 레퍼런스 카운팅 메커니즘으로 사용될 수도 있습니다. 예를 들어, 다음과 같은 코드 조각을 생각해 봅시다:

```
1 rCU_read_lock(); /* acquire reference. */
2 p = rCU_dereference(head);
3 /* do something with p. */
4 rCU_read_unlock(); /* release reference. */
```

이 rCU_read_lock()는 p로의 레퍼런스를 얻어오는 것으로 볼 수 있는데, rCU_dereference()를 통한 p로의 할당 후 시작되는 grace period는 연관되는 rCU_read_unlock() 전까지는 끝나지 않을 것이기 때문입니다. 이 레퍼런스 카운팅 방식은 RCU read-side 크리티컬 섹션 안에서는 블락이 허용되지 않고, 다른 태스크로 RCU read-side 크리티컬 섹션을 넘겨줄 수도 없다는 점에서 제한적입니다.

그런 제한에도 불구하고, 다음의 코드는 안전하게 p를 삭제할 수 있습니다:

```
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);
```

head로의 할당은 뒤이은 p로의 레퍼런스 획득을 방지하고, synchronize_rcu()는 앞서 획득된 레퍼런스들이 해제되기를 기다립니다.

Quick Quiz 9.30: 근데 잠깐만요! 이건 RCU를 reader-writer 락킹 대용으로 하려 할 때 쓸법한 코드와 완전 똑같잖아요! 뭐가 새로운 거죠? ■

물론, RCU는 Section 13.2에서 논의된 것처럼, 전통적인 레퍼런스 카운팅과 함께 조합될 수도 있습니다.

하지만 왜 이런 신경을 쓰는 걸까요? 다시 말하지만, 딥 중 하나는 성능으로, 16-CPU 3GHz Intel x86 시스템에서의 데이터가 Figure 9.41에 있습니다.

Quick Quiz 9.31: 6 CPU 근처에서 refcnt 오버헤드가 확 떨어지는건 왜그렇죠? ■

그리고, reader-writer 락킹 때처럼, RCU의 성능상 이득은 Figure 9.42에 16-CPU 시스템에서의 결과로 보여지듯 짧은 길이의 크리티컬 섹션들에서 가장 효과를 발합니다. 또한, reader-writer 락킹에서처럼, 많은 시스템 콜들은 (그리고 그것들을 포함하는 RCU read-side 크리티컬 섹션들은) 수 마이크로세컨드 내에 끝납니다.

하지만, RCU에 따라오는 이 제약들은 상당히 성가십니다. 예를 들어, 많은 경우에 RCU read-side 크리티컬 섹션 내에서 잠들기가 금지된다는 점은 목표 달성을 불가하게 할 수 있습니다. 다음 섹션에서는 적어도 일부 경우에는 전통적인 레퍼런스 카운팅의 복잡도를 낮추면서도 이 문제를 해결하는 방법들을 알아보겠습니다.

9.5.3.4 RCU is a Bulk Reference-Counting Mechanism

앞의 섹션에서 이야기 했듯, 전통적인 레퍼런스 카운터들은 보통 특정 데이터 구조 또는 데이터 구조체들의 특정 그룹과 연관되어 있습니다. 하지만, 매우 다양한 데이터 구조체들에 하나의 글로벌한 레퍼런스 카운터를 사용하는 것은 이 레퍼런스 카운트를 담고 있는 캐시

```

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
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 9.38: 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
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 9.39: 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
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 9.40: Converting Reader-Writer Locking to RCU: Deletion

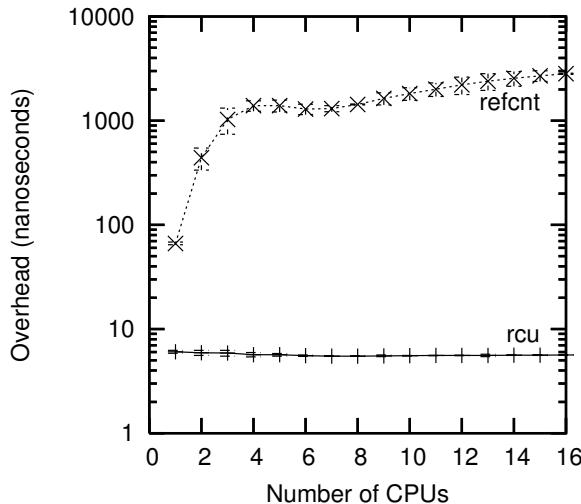


Figure 9.41: Performance of RCU vs. Reference Counting

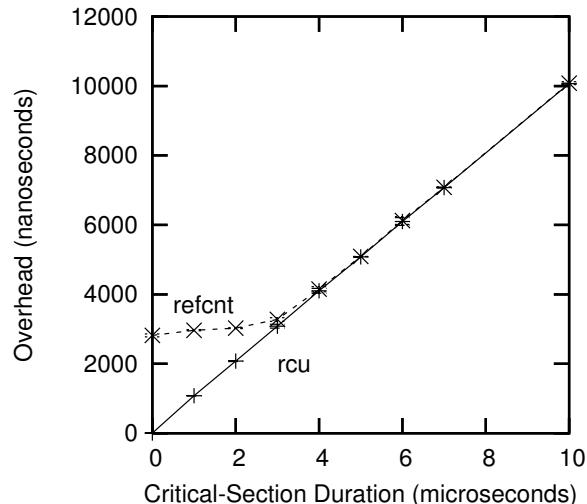


Figure 9.42: Response Time of RCU vs. Reference Counting

라인 바운싱 (cache line bouncing) 을 초래합니다. 그런 캐시 라인 바운싱은 성능을 심각하게 저하시킬 수 있습니다.

반면, RCU 의 가벼운 read-side 기능들은 무시해도 좋을 만한 성능 저하만을 가지면서 굉장히 빈번한 read-side 실행이 가능하게 해서 적거나 아예 없는 퍼포먼스 저하를 갖는 “벌크 레퍼런스 카운팅” 메커니즘으로 RCU 가 사용될 수 있습니다. 블록을 하는 코드 섹션들을 가로질러 하나의 태스크가 하나의 레퍼런스를 쥐고 있어야만 하는 상황에는 Sleepable RCU (SRCU) [McK06b] 가 사용될 수 있을 겁니다. 이는 레퍼런스가 한 태스크에서 다른 태스크로 “전달” 되는, 예를 들어 레퍼런스가 I/O 의 시작에 획득되어지고 연관된 완료 인터럽트 핸들러에서 해제되는, 희귀하지 않은 상황들을 처리하지는 못할 겁니다. (원칙적으로, 이는 SRCU 구현으로 처리될 수도 있지만, 실제로 이게 좋은 트레이드오프인지는 아직 명확하지 않습니다.)

물론, SRCU 는 그 자신의 제약을 가지는데, `srcu_read_lock()` 의 리턴값은 연관된 `srcu_read_unlock()` 으로 전달되고, 어떤 SRCU 기능도 하드웨어 인터럽트 핸들러나 non-maskable interrupt (NMI) 핸들러 안에서 실행될 수 없다는 것입니다. 이런 제약들에 의해 얼마나 많은 문제들이 존재하게 되는지, 그리고 그것들이 어떻게 가장 잘 처리될 수 있는지에 대한 판단은 아직 진행 중입니다.

9.5.3.5 RCU is a Poor Man’s Garbage Collector

RCU 를 처음 공부하는 사람들이 갖는, 희귀하지 않은 감탄 하나는 “RCU 는 가비지 컬렉터 같은 것이다!” 입니다. 이 감탄은 커다란 진실을 품고 있기는 하지만, 또한 오해가 될 수도 있습니다.

RCU 와 자동화된 가비지 컬렉터들 (GC) 사이의 관계를 생각하는 가장 좋은 방법은 RCU 는 컬렉션의 타이밍이 자동으로 정해진다는 점에서 RCU 가 GC 와 닮았다는 점입니다만, RCU 는 GC 와 다음과 같이 다릅니다: (1) 프로그래머는 언제 특정 데이터 구조체가 정리되어도 좋은지를 일일이 알려야만 하고, (2) 프로그래머는 레퍼런스들이 합법적으로 붙잡힐 수 있는 RCU read-side 크리티컬 섹션들을 일일이 표시해야만 합니다.

이런 차이점들에도 불구하고, 유사한 부분들 역시 상당히 깊은 영역까지 들어가는 편이고, RCU 에 대한 최소한 하나의 이론적인 분석이 될 수도 있습니다. 더 나아가서, 제가 신경썼던 첫번째 RCU 같은 메커니즘은 grace period 를 처리하기 위해 가비지 컬렉터를 사용했습니다. 하지만 더도 아니고 덜도 아니고, RCU 를 생각하는 더 나은 방법은 다음 섹션에 설명되어 있습니다.

9.5.3.6 RCU is a Way of Providing Existence Guarantees

Gamsa 등 [GKAS99] 은 존재 보장에 대해 논하고 어떻게 RCU 를 닮은 메커니즘이 이런 존재 보장을 제공하기 위해 사용되는지 설명 (해당 PDF 의 7 페이지의

```

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 9.43: Existence Guarantees Enable Per-Element Locking

section 5 를 보세요) 했고, Section 7.4 에서는 존재 보장을 락킹을 사용해서 어떻게 하는지를 그렇게 하는 것의 단점과 함께 이야기했습니다. 만약 RCU 로 보호되는 데이터 원소가 RCU read-side 크리티컬 섹션 내에서 접근된다면, 그 데이터 항목은 그 RCU read-side 크리티컬 섹션 기간 동안은 존재하는 상태로 유지되는 것이 보장됩니다.

Figure 9.43 는 RCU 기반의 존재 보장이 어떻게 원소별 락킹을 가능하게 하는지를 해시 테이블에서 원소를 하나 삭제하는 함수를 통해 보이고 있습니다. Line 6 는 해시 함수를 수행하고, line 7 에서 RCU read-side 크리티컬 섹션에 들어갑니다. Line 9 에서 연관된 해시 테이블의 버킷이 비어있거나 버킷에 있는 항목이 우리가 삭제하려 하는 것이 아니라고 판단되면 line 10 에서 RCU read-side 크리티컬 섹션을 빠져나오고 line 11 에서 실패했음을 알립니다.

Quick Quiz 9.32: 우리가 삭제하려는 원소가 Figure 9.43 의 line 9 의 리스트의 첫번째 원소가 아니라면 어찌하죠? ■

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 9.33: Figure 9.43 의 line 17 에서 락을 놓기 전에 line 15 에서 RCU read-side 크리티컬 섹션을 빠지는게 왜 안전한거죠? ■

Quick Quiz 9.34: Figure 9.43 의 line 23 에서 RCU read-side 크리티컬 섹션을 빠져나가는 건 왜 line 22 에서 락을 내려놓기 전에 될 수 없나요? ■

기민한 독자들은 이게 Section 9.5.3.8 에서 다뤄진, “RCU 는 일들이 끝나길 기다리는 한가지 방법이다” 테마의 사소한 변종이라는 걸 눈치챘을 겁니다. 그런 독자들은 또한 Section 7.4 에서 이야기한 락 기반의 존재 보장에 비해서 얻어지는 테드락 내성의 장점도 알 것입니다.

9.5.3.7 RCU is a Way of Providing Type-Safe Memory

여러 lockless 알고리즘들은 데이터 원소가 그것을 레퍼런스 하고 있는 RCU read-side 크리티컬 섹션동안 그 아이덴티티를 유지하고 있을 것을 필요로 하지 않습니다—다만 그 데이터가 같은 타입을 유지한다는 가정 아래의 이야기입니다. 달리 말하자면, 이런 lockless 알고리즘들은 데이터 원소가 레퍼런스 되고 있는 와중에도 메모리 해제되고 같은 타입의 구조체로 재할당 되는 상황을 처리할 수 있지만 타입의 변화는 없어야만 합니다. 이런, 학술적 문맥에서는 “type-safe memory” 라 불리는 [GC96] 보장사항은 앞 섹션에서 이야기한 존재 보장 보다 더 완화된 형태이고 따라서 이걸 가지고 일을 처리하기는 약간 더 어렵습니다. 리눅스 커널 안의 Type-safe 메모리 알고리즘들은 slab 캐시들을 사용하는데, 특히 이런 캐시들을 SLAB_DESTROY_BY_RCU 로 표시해서 사용되지 않는 slab 을 시스템 메모리로 반환할 때 RCU 가 사용되게 합니다. 이런 RCU 의 사용은 그런 slab 의 사용중인 원소들은 그 slab 안에 남아 있을 것임이 보장되어서 앞서 존재한 RCU read-side 크리티컬 섹션들의 기간동안은 그 타입을 유지하게 됩니다.

Quick Quiz 9.35: 하지만 여러 쓰레드들에 상당히 긴 RCU read-side 크리티컬 섹션들이 존재해서 어떤 특정한 시점에든 시스템의 최소 하나의 쓰레드는 RCU read-side 크리티컬 섹션을 수행하고 있으면 어떡하죠? 그게 어떤 데이터가 SLAB_DESTROY_BY_RCU 슬랩에서 시스템으로 반환되는걸 막아서 OOM 이벤트를 유발하지는 않을까요? ■

이런 알고리즘들은 새로 레퍼런스된 데이터 구조체가 정말로 요청된 타입이란 것을 분명히 하기 위해 검증 단계를 일반적으로 갖습니다 [LS86, Section 2.5]. 이런 검증은 데이터 구조체의 특정 부분이 메모리 해제-재할당 프로세스에서 건들여지지 않았을 것을 필요로 합니다. 그런 검증은 일반적으로 잘 되기가 매우 어렵고, 애매하고 어려운 버그들을 숨길 수 있습니다.

따라서, type-safety 기반의 락을 사용하지 않은 알고리즘들은 매우 드물고 어려운 상황에서는 큰 도움이 될 수 있지만, 가능하다면 존재 보장을 사용해야 합니다. 더 간단하게 거의 항상 더 낫습니다!

9.5.3.8 RCU is a Way of Waiting for Things to Finish

Section 9.5.2에서 이야기했듯이, RCU의 중요한 요소는, RCU 읽기 쓰레드들이 끝나기를 기다리는 방법입니다. RCU의 큰 장점 가운데 하나는 수천개의 서로 다른 것들이 각자 끝나기를 명시적으로 그들을 각각의 정보를 추적할 필요없이, 그리고 명시적인 정보 추적 방식에서 심각한 성능 저하, 확장성 제한, 복잡한 데드락 시나리오, 그리고 메모리 누수 문제를 걱정할 필요 없이 기다릴 수 있다는 것입니다.

이 섹션에서는 read-side 쪽의 (하드웨어 오퍼레이션들과 인터럽트를 불가능하게 하는 기능들을 사용해서 preemption을 불가능하게 하는 기능을 포함하는) synchronize_sched() 비슷한 기능이 락킹을 사용한다면 상당히 어려울 non-maskable interrupt (NMI) 핸들러들과의 상호작용을 어떻게 할 수 있게 하는지 알아보겠습니다. 이 방법은 “Pure RCU” [McK04] 라 불렸으며, 리눅스 커널의 여러 곳에서 사용됩니다.

그런 “Pure RCU” 디자인의 기본적인 형태는 다음과 같습니다:

- 예를 들어 OS가 NMI에 반응하는 것과 같은 방식으로 변경을 만듭니다.
- 모든 앞서 존재한 read-side 크리티컬 섹션들이 완전히 종료되기를 기다립니다 (예를 들어, synchronize_sched() 기능을 사용해서). 여기서의 핵심은 뒤따르는 RCU read-side 크리티컬 섹션들은 만들어진 변경을 볼 수 있게 보장된다는 것입니다.
- 예를 들어 변경이 성공적으로 만들어졌음을 알리는 상태를 반환하는 식으로 정리를 합니다.

이 섹션의 나머지 부분은 리눅스 커널에서 가져온 예제 코드를 선보입니다. 이 예제에서, timer_stop 함수는 모든 NMI 노티피케이션들이 연관된 리소스들을 반환하기 전에 완료되었음을 보장하기 위해 synchronize_sched()를 사용합니다. 이 코드의 간략화된 버전이 Figure 9.44에 있습니다.

Line 1-4는 크기와 애매한 원소들의 배열을 갖는 profile_buffer 구조체를 정의합니다. Line 5는 profile buffer로의 포인터를 정의하는데, 아마 어디선가 동적으로 할당된 메모리의 영역을 가리키게 될겁니다.

```

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 9.44: Using RCU to Wait for NMIs to Finish

Line 7-16은 NMI 핸들러 안에서 호출되는 nmi_profile() 함수를 정의합니다. 그런 것들이 그러하듯, 이 함수는 preemption 당할 수 없고, 평범한 인터럽트 핸들러에 인터럽트될 수도 없습니다만, 캐시 미스, ECC 에러, 그리고 같은 코어에 위치한 다른 하드웨어 쓰레드에 의한 cycle stealing에는 취약합니다. Line 9는 DEC Alpha에서도 메모리 순서를 강제하기 위해 rcu_dereference() 함수를 이용해 profile buffer로의 지역 포인터를 가져오고, line 11과 12에서 현재 할당된 profile buffer가 없다면 이 함수에서 빠져나가며, line 13과 14에서는 pcvalue 인자가 범위 밖이라면 이 함수에서 빠져나갑니다. 그렇지 않다면, line 15에서 pcvalue 인자로 색인된 profile-buffer 원소의 값을 증가시킵니다. 베파와 그 크기를 함께 저장하는 것은 설명 커다란 베파가 갑자기 작은 것으로 바뀌더라도 크기 검사로 베파를 맞출 수 있음을 보장한다는 것을 기억해 두세요.

Line 18-27은 nmi_stop()을 정의하는데, 이 함수의 호출자는 알아서 상호 배제를 지켜야 합니다 (예를 들면, 올바른 락을 잡는식으로). Line 20은 profile buffer로의 포인터를 가져오고, line 22와 23은 베파가 없다면 이 함수를 빠져나갑니다. 그렇지 않다면, line 24는 profile-buffer 포인터를 NULL로 만들고 (완화된 순서 규칙의 기계에서 메모리 순서를 유지하기 위해 rcu_assign_pointer()를 사용합니다) line 25에서 RCU Sched grace period가 지나가기를 기다리는데, 정확히는 NMI 핸들러를 포함해서 모든 preemption 불가능 영역의 코드들이 완료되길 기다립니다. 일단 line 26으로

수행이 이어지면, 예전 버퍼로의 포인터를 가진 `nmi_profile()` 인스턴스는 모두 종료되었음이 보장됩니다. 따라서 이 버퍼를 메모리 해제해도 안전하며, 여기서는 `kfree()` 함수를 사용했습니다.

Quick Quiz 9.36: `nmi_profile()` 함수가 preemption 당할 수 있다고 해봅시다. 이 예제가 제대로 동작하도록 하기 위해 뭘 바꿔야 할까요? ■

짧게 말해서, RCU 는 동적으로 profile buffer 들 사이를 옮겨다니는 것을 쉽게 해줍니다(그냥 효율적이라도 어토믹 오퍼레이션들을 시도 해보거나, 그냥 락킹을 사용할 수도 있습니다!). 하지만, RCU 는 일반적으로 앞의 섹션들에서 봤듯이 더 높은 수준의 추상화에서 사용됩니다.

9.5.3.9 RCU Usage Summary

핵심적으로, RCU 는 다음을 제공하는 API 이상도 이하도 아닙니다:

1. 데이터 추가를 위한 publish-subscribe 메커니즘,
2. 앞서 존재한 RCU 읽기 쓰레드들이 끝나길 기다리는 방법, 그리고
3. 동시의 RCU 읽기 쓰레드들에 해를 끼치거나 자연시키지 않고 변경을 가할 수 있도록 여러 버전들을 관리하는 규칙.

그렇다면 하나, RCU 위에 앞의 섹션들에서 선보인 reader-writer 락킹, 레퍼런스 카운팅, 그리고 존재 보장 등의 더 높은 수준의 것을 만드는 것은 가능합니다. 더 나아가서, 저는 리눅스 커뮤니티가 다른 동기화 기능들과 함께, RCU 의 새롭고 흥미로운 사용처들을 찾아나갈 것이라 믿습니다.

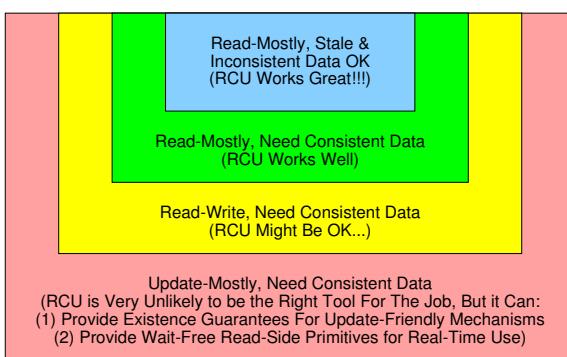


Figure 9.45: RCU Areas of Applicability

그전까지는, Figure 9.45 이 RCU 가 가장 도움되는 곳들에 대한 대략적인 규칙을 보입니다.

그림의 꼭대기의 파란 박스가 보이듯, RCU 는 낡고 비일관적인 데이터가 허용되고 읽기가 대부분인 데이터를 가지고 있을 때 가장 잘 동작합니다(하지만 낡고 비일관적인 데이터에 대한 더 많은 정보를 위해 아래를 읽으세요). 리눅스 커널에서의 이런 케이스의 표준적 예제는 라우팅 테이블입니다. 라우팅 경로 업데이트가 인터넷을 통해 전파되기까지는 수초에서 심지어 분단위까지도 걸릴 수 있기 때문에, 시스템은 상당히 가끔 패킷들을 잘못된 방향으로 보낼 겁니다. 일부 패킷들을 잘못된 방향으로 보내는 작은 가능성을 수 밀리세컨드 동안 갖는 것은 결코 문제가 되지 않습니다.

일관적인 데이터가 필요한 읽기가 대부분인 워크로드라면, RCU 는 초록색의 “read-mostly, need consistent data” 박스로 보인 것처럼 잘 동작합니다. 이런 케이스의 한가지 예는 리눅스 커널의 사용자 레벨 System-V 세마포어 ID 들부터 연관된 커널 내부 데이터 구조체들로의 매핑이 되겠습니다. 세마포어들은 그것들이 생성되고 사라지는 것보다 훨씬 더 빈번하게 사용되고, 따라서 이 매핑은 읽기가 대부분입니다. 하지만, 이미 삭제된 세마포어를 가지고 세마포어 오퍼레이션을 수행하는 건 예러를 내기 쉬울 겁니다. 이런 일관성의 필요는 커널 내 세마포어 데이터 구조체 내에 있는 락과 세마포어가 지워질 때 세워지는 “deleted” 플래그를 사용해 처리됩니다. 어떤 사용자 ID 가 “deleted” 플래그가 세워진 채 커널 내부 데이터 구조체에 매핑되면, 그 데이터 구조체는 무시되어서 그 사용자 ID 는 무효로 처리됩니다.

이런 방법은 읽기 쓰레드들이 세마포어 자체를 나타내는 데이터 구조체의 락을 잡아야 하게 하지만, 데이터 구조체의 매핑에는 락킹이 필요없게 해줍니다. 읽기 쓰레드들은 따라서 락을 사용하지 않고 ID 에서 데이터 구조체로의 매핑을 위해 사용된 트리를 횡단할 수 있는데, 이는 훨씬 향상된 성능, 확장성, 그리고 real-time 반응속도를 가져옵니다.

노란색의 “read-write” 박스로 보인 것처럼, RCU 는 일관적인 데이터가 필요한 read-write 워크로드에서도 유용하지만, 보통은 다른 동기화 도구들과 함께 사용될 때에 그렇습니다. 예를 들어, 최근의 리눅스 커널들의 디렉토리 엔트리 캐시는 시퀀스 락, CPU 별 락, 그리고 데이터 구조체별 락과 함께 RCU 를 사용해서 일반적인 경우의 pathname 들의 횡단을 락을 사용하지 않고 가능하게 합니다. 비록 RCU 가 이 read-write 케이스에는 많은 이득을 가져다 줄 수 있지만, 그런 사용은 읽기가 대부분인 경우에 비해 많은 경우 더 복잡합니다.

마지막으로, 그림 바닥의 빨간 박스가 나타내듯이, 일관적인 데이터를 필요로 하고 업데이트가 대부분인 워크로드는 일부 예외가 있긴 하지만 [DMS⁺12] RCU 를 사용하기 좋은 곳이 아닐 확률이 높습니다. 또한, Section 9.5.3.7 에서 이야기했듯, 리눅스 커널 내에서

SLAB_DESTROY_BY_RCU 슬랩 얼로케이터 플래그는 RCU 읽기 쓰레드들에게 type-safe 메모리를 제공하는데, 이는 non-blocking 동기화와 다른 락을 사용하지 않은 알고리즘들을 매우 간단하게 만들어줄 수 있습니다.

짧게 말해서, RCU 는 데이터 추가를 위한 publish-subscribe 메커니즘, 앞서 존재한 RCU 읽기 쓰레드들이 끝나길 기다리는 방법, 그리고 동시에 RCU 읽기 쓰레드들에 해를 끼치거나 지연시키지 않고 업데이트를 할 수 있도록 여러 버전들을 관리하는 규칙을 포함한 API입니다. 이 RCU API 는 읽기가 대부분인 상황에 가장 잘 맞는데, 넓고 비일관적인 데이터가 허용되는 어플리케이션에서 특히 적합합니다.

9.5.4 RCU Linux-Kernel API

이 섹션은 RCU를 그것의 리눅스 커널 API의 관점에서 바라봅니다. Section 9.5.4.1 은 RCU 의 wait-to-finish API들을, 그리고 Section 9.5.4.2 에서는 RCU 의 publish-subscribe 와 version-maintenance API들을 소개합니다. 마지막으로, Section 9.5.4.4 에서는 결론을 정리합니다.

9.5.4.1 RCU has a Family of Wait-to-Finish APIs

“RCU는 무엇인가”에 대한 가장 직접적인 답변은 RCU 는 리눅스 커널에서 사용되는 API 라는 것으로, 각각 잠들 수 없는 버전과 잠들 수 있는 버전 API 들의 RCU 읽기 쓰레드 기다리기 부분을 보이는 Table 9.3, 그리고 그 API 의 publish-subscribe 부분을 보이는 Table 9.4 에 요약되어 있습니다.

RCU 가 처음이라면 Table 9.3 의 행들 중 하나에만 집중해 볼 것을 고려해 볼만 한데, 각각의 행은 리눅스 커널의 RCU API 패밀리 중 하나의 멤버를 요약하고 있습니다. 예를 들어, 리눅스 커널에서 RCU 가 어떻게 사용되는지를 이해하고자 하는게 주된 목표라면, “RCU Classic” 가장 자주 사용되므로 여기서부터 시작하는게 좋을 것입니다. 반면에, 자신의 이익을 위해 RCU 를 이해하고자 한다면 “SRCU” 가 가장 간단한 API 를 제공합니다. 나중에도 언제든 다른 행을 볼 수 있습니다.

이미 RCU 에 친숙하다면, 이 표들은 유용한 레퍼런스로 사용될 수 있을 겁니다.

Quick Quiz 9.37: Table 9.3 의 일부 셀들은 왜 느낌표 (“!”) 를 가지고 있나요? ■

이 “RCU Classic” 행은 RCU read-side 크리티컬 섹션들은 `rcu_read_lock()` 과 `rcu_read_unlock()` 으로 구분지어지고 중첩될수도 있는, 최초의 RCU 구현에 해당합니다. 여기에 연관되는 동기적인 업데이트 쪽 기능들인 `synchronize_rcu()` 와 그것과 같은 의미인 `synchronize_net()` 은 동시에 실행중인 RCU read-side 크리티컬 섹션들이 모두 완료되기를 기다립니다. 이 기다림의 길이는 “grace period” 라고 알려져 있

습니다. 비동기적인 업데이트 쪽 기능인 `call_rcu()` 는 뒤따르는 grace period 후에 특정 함수를 특정 인자와 함께 호출해 줍니다. 예를 들어, `call_rcu(p, f);` 는 다음의 grace period 후에 “RCU callback” `f(p)` 의 호출이 이뤄지게 합니다. `call_rcu()` 를 사용하는 리눅스 커널 모듈을 언로딩 한다던가 해서 모든 RCU callback 들이 완료되기를 기다려야만 하는 상황도 존재합니다 [McK07d]. `rcu_barrier()` 기능이 그 일을 합니다. 더 최신의 계층적 RCU [McK08a] 구현 또한 “RCU Classic” 시맨틱을 고수함을 알아두세요.

마지막으로, RCU 는 Section 9.5.3.7 에서 설명한 것처럼 type-safe 메모리 [GC96] 를 제공하는데 사용될 수도 있습니다. RCU 의 문맥에서, type-safe 메모리는 주어진 데이터 원소가 그것에 접근하는 모든 RCU read-side 크리티컬 섹션 사이에서 그 타입이 바뀌지 않는다는 것을 보장합니다. RCU 기반의 type-safe 메모리를 사용하기 위해서는 `SLAB_DESTROY_BY_RCU` 를 `kmem_cache_create()` 에 넘겨야 합니다. `SLAB_DESTROY_BY_RCU` 는 `kmem_cache_alloc()` 이 `kmem_cache_free()` 로 자유가 된 메모리를 즉시 재할당 하는 것을 막는 일은 결코 하지 않음을 알아두는 게 중요합니다! 사실, `rcu_dereference` 로 리턴된, `SLAB_DESTROY_RCU` 로 보호되는 데이터 구조체는 상당히 여러번 메모리 해제되고 재할당 될 수 있는데, 심지어 `rcu_read_lock()` 으로 보호되고 있을 때도 그려합니다. 대신, `SLAB_DESTROY_BY_RCU` 는 RCU grace period 가 끝나기 전까지는 `kmem_cache_free()` 가 완전히 해제된 데이터 구조체들의 slab 을 시스템에 반납하는 것을 방지해 줍니다. 짧게 말해서, 비록 데이터 원소가 굉장히 자주 해제되고 재할당될 수 있지만, 최소한 그것의 타입은 똑같이 남아있을 겁니다.

Quick Quiz 9.38: 많은 수의 RCU read-side 크리티컬 섹션들이 `synchronize_rcu()` 실행을 무기한 블록시키는 걸 어떻게 막을 수 있나요? ■

Quick Quiz 9.39: `synchronize_rcu()` API 는 전부터 존재한 인터럽트 핸들러들이 모두 완료되길 기다리죠, 맞죠? ■

“RCU BH” 행에서, `rcu_read_lock_bh()` 와 `rcu_read_unlock_bh()` 는 크리티컬 섹션을 구분짓고, `synchronize_rcu_bh()` 는 하나의 grace period 를 기다리며, `call_rcu_bh()` 는 다음 grace period 후에 특정 함수를 특정 인자와 함께 호출해 줍니다.

Quick Quiz 9.40: 이것들을 섞어서 활용하면 어떻게 되나요? 예를 들어, `rcu_read_lock()` 과 `rcu_read_unlock()` 을 RCU read-side 크리티컬 섹션을 구분하는데 사용하지만 `call_rcu_bh()` 를 RCU callback 을 위해 사용한다고 하면요? ■

Quick Quiz 9.41: 하드웨어 인터럽트 핸들러들은 묵시적인 `rcu_read_lock_bh()` 의 보호 아래 있다고

Attribute	RCU Classic	RCU BH	RCU Sched	Realtime RCU	SRCU
Purpose	Original	Prevent DDoS attacks	Wait for preemptable regions, hardirqs, & NMIs	Realtime response	Sleeping readers
Availability	2.5.43	2.6.9	2.6.12	2.6.26	2.6.19
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>	<code>srcu_read_lock()</code> <code>srcu_read_unlock()</code>
Update-side primitives (synchronous)	<code>synchronize_rcu()</code> <code>synchronize_net()</code>	<code>synchronize_rcu_bh()</code>	<code>synchronize_sched()</code>	<code>synchronize_rcu()</code> <code>synchronize_net()</code>	<code>synchronize_srcu()</code>
Update-side primitives (asynchronous/callback)	<code>call_rcu()!</code>	<code>call_rcu_bh()</code>	<code>call_rcu_sched()</code>	<code>call_rcu()</code>	<code>call_srcu()</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>	N/A
Type-safe memory	<code>SLAB_DESTROY_BY_RCU</code>	<code>SLAB_DESTROY_BY_RCU</code>	<code>SLAB_DESTROY_BY_RCU</code>	<code>No synchronize_srcu()</code>	<code>No synchronize_srcu()</code>
Read side constraints	No blocking	No bottom-half (BH) enabling	Only preemption and lock acquisition	with same <code>srcu_struct</code>	Simple instructions, irq disable/enable, memory barriers
Read side overhead	Precipitously disable/enable (free on non-PREEMPT)	Precipitously disable/enable (free on non-PREEMPT)	Simple instructions, irq disable/enable	N/A	
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	Precipitous RCU	SRCU	
PREEMPT-RT implementation	Preemptible RCU	Realtime RCU	Forced Schedule on all CPUs	Realtime RCU	SRCU

Table 9.3: RCU Wait-to-Finish APIs

생각되어도 되겠죠? ■

“RCU Sched” 행에서, preemption 을 불가능하게 하는 모든 동작은 RCU read-side 크리티컬 섹션처럼 동작되고, `synchronize_sched()` 는 연관된 RCU grace period 를 기다립니다. 이 RCU API 패밀리는 2.6.12 커널에서 들어왔는데, 이것이 과거의 `synchronize_kernel()` API 를 (RCU Classic 을 위한) 지금의 `synchronize_rcu()` 와 (RCU Sched 를 위한) `synchronize_sched()` 로 나눴습니다. RCU Sched 는 처음부터 비동기적인 `call_rcu_sched()` 인터페이스를 가지고 있진 않았다가 2.6.26 에서 추가되었음을 알아두세요. 리눅스 커뮤니티의 어떤 의미에서의 minimalist 철학에 의해, API 들은 필요한 경우에 기반해서 추가됩니다.

Quick Quiz 9.42: RCU Classic 과 RCU Sched 를 섞어서 사용하면 어떻게 되나요? ■

Quick Quiz 9.43: 일반적으로, 모든 전부터 존재한 인터럽트 핸들러들을 기다리는데에 `synchronize_sched()` 에 의존해선 안됩니다, 맞죠? ■

“Realtime RCU” 행은 RCU Classic 과 똑같은 API 를 가지고 있는데, 차이점은 RCU read-side 크리티컬 섹션들이 preemption 당할 수 있고 spinlock 을 획득하는 사이 블락될 수 있다는 점 뿐입니다. Realtime RCU 의 설계는 다른곳에도 설명되어 있습니다 [McK07a].

Table 9.3 의 “SRCU” 행은 RCU read-side 크리티컬 섹션들 안에서 일반적인 잠들기를 허용하는 특수한 RCU API 를 보입니다 [McK06b]. 물론, SRCU read-side 크리티컬 섹션 안에서의 `synchronize_srcu()` 사용은 스스로의 deadlock 을 유발할 수 있으므로, 반드시 피해져야 합니다. SRCU 의 앞의 RCU 구현들과의 차이점은 각각의 별개의 SRCU 사용처마다 호출자가 `srcu_struct` 를 할당해야 한다는 겁니다. 이 방법은 SRCU read-side 크리티컬 섹션들이 연관되지 않은 `synchronize_srcu()` 실행을 블록하는 것을 방지합니다. 또한, 이 RCU 변종에서는 `srcu_read_lock()` 이 연관된 `srcu_read_unlock()` 에 전달되어야 하는 값을 리턴합니다.

Quick Quiz 9.44: `call_srcu()` 사용에 있어 조심해야 하는 이유는 무엇일까요? ■

Quick Quiz 9.45: 어떤 조건에서 `synchronize_srcu()` 가 SRCU read-side 크리티컬 섹션 내에서 안전하게 사용될 수 있을까요? ■

리눅스 커널은 분명히 놀랍도록 많은 RCU API 와 구현을 가지고 있습니다. 이 수를 줄이려는 희망이 있는데, 리눅스 커널의 특정 빌드는 현재 세개의 API 를 뒤에 최대 네개의 구현들을 가지고 있다는 사실 (RCU Classic 과 Realtime RCU 는 같은 API 를 공유합니다) 이 그 증거입니다. 하지만, 충분한 조사와 분석이 필요할텐데, 많은 락킹 API 들 가운데 하나를 제거하려면

필요한 것처럼 말입니다.

이 다양한 RCU API 들은 RCU read-side 크리티컬 섹션들이 반드시 제공해야 하는 forward-progress 보장사항으로 차별화 되고 그 범위로도 차별화되는데, 다음과 같습니다:

1. RCU BH: read-side 크리티컬 섹션들은 NMI 와 인터럽트 핸들러들을 제외한 모든 것에 대해 forward progress 를 보장해야만 합니다만 software-interrupt (`softirq`) 핸들러들은 제외입니다. RCU BH 는 범위내에서 글로벌 합니다.
2. RCU Sched: read-side 크리티컬 섹션들은 NMI 와 `softirq` 핸들러들을 포함한 irq 핸들러들을 제외한 모든 것들에 forward progress 를 보장해야 합니다. RCU Sched 는 범위내에서 글로벌합니다.
3. RCU (Classic 과 Real-time 둘 다): read-side 크리티컬 섹션들은 NMI 핸들러, irq 핸들러, `softirq` 핸들러, 그리고 (real-time 의 경우) 더 높은 우선순위의 real-time task 를 제외한 모든 것에 forward progress 를 보장해야 합니다. RCU 는 범위내에서 글로벌합니다.
4. SRCU: read-side 크리티컬 섹션들은 다른 태스크 가 연관된 grace period 가 완료되기를 기다리고 있지 않다면 forward progress 를 보장하지 않아도 됩니다. 이런 상황에서 이 read-side 크리티컬 섹션들은 수초 이내에는 완료되어야 합니다 (그리고 더 빠르면 더 좋습니다).¹⁰ SRCU 의 범위는 각각 연관된 `srcu_struct` 의 사용에 따라 정의됩니다.

달리 말하자면, SRCU 는 개발자가 그 범위를 제한할 수 있도록 하는 것으로 극단적으로 약한 forward-progress 보장사항의 문제를 보완합니다.

9.5.4.2 RCU has Publish-Subscribe and Version-Maintenance APIs

다행히도, 다음의 표에 보여진 RCU publish-subscribe 와 version-maintenance 기능들은 앞서 언급된 RCU 의 변종들 모두에 적용됩니다. 이 공통성은 어떤 경우들에 더 많은 코드가 공유될 수 있게 해서, 그렇지 않다면 일어날 수 있는 API 충돌을 분명히 줄여줍니다. RCU publish-subscribe API 들의 원래 목적은 메모리 배리어들을 이 API 안에 묻어버려서 리눅스 커널 프로그래머들이 리눅스가 지원하는 각각의 20 종류가 넘는 CPU

¹⁰ 단순히 forward-progress guarantee 가 없다고 말하는 대신 이런 명확한 설명을 하도록 재촉해준 James Bottomley 에게 감사의 말씀을 드립니다.

패밀리들 [Spr01] 의 메모리 순서 모델의 전문가가 되지 않더라도 RCU 를 사용할 수 있게 하려는 것이었습니다.

카테고리들 중 처음 두개의 카테고리들은 순환형의 이중 링크드 리스트인 리눅스 struct list_head 리스트들에 동작합니다. list_for_each_entry_rcu() 함수는 RCU 로 보호되는 리스트를 type-safe 하게 횡단하는데 새로운 리스트 원소가 횡단과 동시에 삽입되는 상황을 위한 메모리 순서의 강제 역시 합니다. Alpha 외의 플랫폼들에서, 이 함수는 list_for_each_entry() 에 비해 성능 하락을 주긴 하지만 그 정도는 적거나 아예 없습니다. list_add_rcu(), list_add_tail_rcu(), 그리고 list_replace_rcu() 함수들은 RCU 아닌 비슷한 것들과 유사합니다만 완화된 순서의 기계들에서는 추가적인 메모리 배리어로의 오버헤드를 갖습니다. list_del_rcu() 함수는 또한 RCU 아닌 비슷한 것과 유사합니다만, list_del() 이 그래야 하는 것처럼 prev 와 next 포인터들을 모두 파괴하는게 아니라 prev 포인터만 파괴하기 때문에 신기하게도 매우 조금 빠릅니다. 마지막으로, list_splice_init_rcu() 함수는 역시 RCU 아닌 비슷한 것들과 유사합니다만 grace-period 대기시간을 갖습니다. 이 grace period 의 목적은 RCU 읽기 쓰레드들이 원본 리스트의 횡단을 그것이 리스트 헤더로부터 분리되기를 완료하기 전까지 안전하게 마치도록 하는 것입니다 – 이에 실패하는 것은 그런 읽기 쓰레드들이 그 횡단을 마무리하지 못하게 할 수도 있습니다.

Quick Quiz 9.46: list_del_rcu() 는 왜 next 와 prev 두 포인터를 모두 파괴하지 않는거죠? ■

다음의 두 카테고리들은 리눅스의 선형 링크드 리스트인 struct hlist_head 에 대해 동작합니다. struct list_head 에 비해 struct hlist_head 의 장점은 하나의 포인터를 갖는 리스트 헤더만이 필요해서 커다란 해시 테이블에서는 상당한 양의 메모리를 아낄 수 있다는 점입니다. 이 표에서의 struct hlist_head 의 기능들과 RCU 를 사용하지 않는 비슷한 것들과의 관계는 struct list_head 기능들이 그러한 것과 거의 같습니다.

마지막의 두 카테고리들은 포인터에 직접적으로 동작하는데, RCU 로 보호되는 배열들이나 tree 들과 같은, RCU 로 보호되지만 리스트가 아닌 데이터 구조체들에 사용하기에 유용합니다. rcu_assign_pointer() 함수는 어떤 앞의 초기화는 완화된 순서 규칙의 기계들에서도 이 포인터로의 할당 전으로 순서지어지게 합니다. 유사하게, rcu_dereference() 함수는 Alpha CPU 에서는, 뒤따르는 해당 포인터를 디레퍼런스 하는 코드가 연관된 rcu_assign_pointer() 앞의 초기화 코드의 효과를 볼 수 있도록 합니다. Alpha 외의 CPU 에서는 rcu_dereference() 는 어떤 포인터 디레퍼

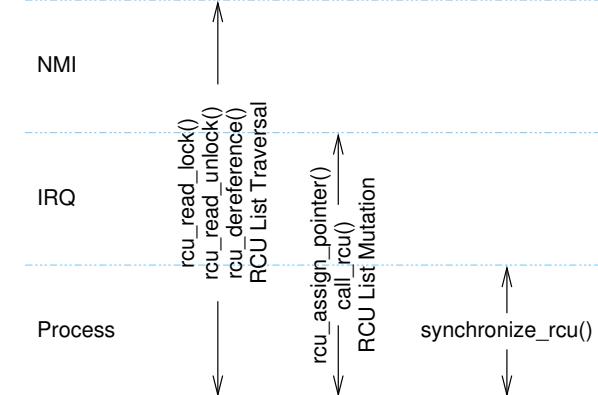


Figure 9.46: RCU API Usage Constraints

런스들이 RCU 로 보호되고 있는지를 표합니다.

Quick Quiz 9.47: 일반적으로, rcu_dereference() 에 사용되는 모든 포인터는 반드시 항상 rcu_assign_pointer() 로 업데이트 되어야만 합니다. 이 규칙에 예외는 뭐가 있을까요? ■

Quick Quiz 9.48: 이런 횡단과 업데이트 기능들이 어떤 RCU API 패밀리 멤버들과 함께 사용된다 하더라도 어떤 문제는 없나요? ■

9.5.4.3 Where Can RCU's APIs Be Used?

Figure 9.46 는 커널 내의 어떤 환경에서 어떤 API 들이 사용될 수 있는지 보입니다. RCU read-side 기능들은 NMI를 포함해 어떤 환경에서도 사용될 수 있고, RCU 의 변경 기능들과 비동기적인 grace-period 기능들은 NMI 를 제외한 모든 환경에서 사용될 수 있으며, RCU 의 동기적인 grace-period 기능들은 프로세스 컨텍스트에서만 사용될 수 있습니다. RCU 리스트 횡단 기능들은 list_for_each_entry_rcu(), hlist_for_each_entry_rcu() 등을 포함합니다. 비슷하게, RCU list 변경 기능들은 list_add_rcu(), hlist_del_rcu() 등이 포함됩니다.

다른 종류의 RCU로부터의 기능들은 대체될 수도 있는데, 예를 들어 srcu_read_lock() 은 rcu_read_lock() 이 사용될 수 있는 모든 컨텍스트에서 사용될 수 있습니다.

9.5.4.4 So, What is RCU Really?

그 핵심에 있어서, RCU 는 더도 덜도 아니고 삽입의 공개와 구독을 지원하고 모든 RCU 읽기 쓰레드들이 완료되길 기다리며, 여러 버전들을 관리하는 API 입니다. 그렇다면 하나, RCU 위에 reader-writer 락킹, 레퍼런스

Category	Primitives	Availability	Overhead
List traversal	<code>list_for_each_entry_rcu()</code>	2.5.59	Simple instructions (memory barrier on Alpha)
List update	<code>list_add_rcu()</code> <code>list_add_tail_rcu()</code> <code>list_del_rcu()</code> <code>list_replace_rcu()</code> <code>list_splice_init_rcu()</code>	2.5.44 2.5.44 2.5.44 2.6.9 2.6.21	Memory barrier Memory barrier Simple instructions Memory barrier Grace-period latency
Hlist traversal	<code>hlist_for_each_entry_rcu()</code> <code>hlist_add_after_rcu()</code> <code>hlist_add_before_rcu()</code> <code>hlist_add_head_rcu()</code> <code>hlist_del_rcu()</code> <code>hlist_replace_rcu()</code>	2.6.8 2.6.14 2.6.14 2.5.64 2.5.64 2.6.15	Simple instructions (memory barrier on Alpha) Memory barrier Memory barrier Memory barrier Simple instructions Memory barrier
Pointer traversal	<code>rcu_dereference()</code>	2.6.9	Simple instructions (memory barrier on Alpha)
Pointer update	<code>rcu_assign_pointer()</code>	2.6.10	Memory barrier

Table 9.4: RCU Publish-Subscribe and Version Maintenance APIs

카운팅, 그리고 Section 9.5.3에서 열거한 존재 보장과 같은 고차원의 것들을 만드는 것이 가능합니다. 더 나아가서, 리눅스 커뮤니티가 RCU의 흥미롭고 새로운 사용처를 찾을 것이라 믿어 의심치 않습니다, 커널의 여러 동기화 기능들에 대해서 그랬듯이요.

물론, 더 완벽한 RCU에 대한 이해는 이 API들을 가지고 할 수 있는 일에 대한 것들도 포함할 것입니다.

하지만, 많은 경우에 RCU에 대한 완벽한 이해는 RCU 구현의 예를 필요로 할겁니다. 따라서 다음 섹션은 복잡도와 기능성을 늘려가는 일련의 “장난감” RCU 구현들을 소개합니다.

9.5.5 “Toy” RCU Implementations

이 섹션의 장난감 RCU 구현은 높은 성능, 실용성, 또는 어떤 종류의 상품에의 사용을 위해 설계되지 않았고¹¹ 그저 분명함을 위해 설계되었습니다. 이 장난감 RCU 구현을 쉽게 이해하기 위해서는 더도 아니고 덜도 아니고, Chapter 2, 3, 4, 그리고 6는 물론이고 Chapter 9의 앞부분들에 대해서도 깊이 이해하고 있어야 합니다.

이 섹션은 존재 보장 문제를 풀어가는 관점에서 세련됨을 증가시켜가는 방향으로 일련의 RCU 구현들을 제공합니다. Section 9.5.5.1은 간단한 락킹에 기반한 초보적인 RCU 구현을 보이고, Sections 9.5.5.2에서 9.5.5.9까지는 락킹, 레퍼런스 카운터, 그리고 free-running 카운터들에 기반한 간단한 RCU 구현을 보입니다. 마지막으로, Section 9.5.5.2부터 9.5.5.9까지는 요약과 바래봄직한 RCU 속성들의 리스트를 제공합니다.

9.5.5.1 Lock-Based RCU

아마도 가장 간단한 RCU 구현은 Figure 9.47 (`rcu_lock.h`와 `rcu_lock.c`)에 보여진 것처럼 락킹을 사용할 겁니다. 이 구현에서 `rcu_read_lock()`은 글로벌 스플래시를 잡고, `rcu_read_unlock()`은 그걸 놓으며, `synchronize_rcu()`는 그걸 잡고는 바로 놓습니다.

`synchronize_rcu()`는 그 락을 잡기 (그리고 놓기) 전까지는 리턴하지 않기 때문에, 모든 앞의 RCU read-side 크리티컬 섹션들이 완료되기 전까지는 리턴할 수 없어서, RCU 시멘틱을 모두 구현하고 있습니다. 물론, 한번에 하나의 RCU 읽기 쓰레드만이 자신의 read-side 크리티컬 섹션을 수행할 수 있어서 RCU의 목적 중 거의 모든 것을 이루지 못합니다. 또한, `rcu_read_lock()`과 `rcu_read_unlock()`에서의 락 오퍼레이션들은 매우 무거운 일이어서, 하나의 Power5 CPU에서 100 나노세컨드 정도의 read-side 오버헤드는 64-CPU 시스템에서는 17 마이크로세컨드 까지 증가합니다. 더 나쁜건, 이것과 같은 락 오퍼레이션들은 `rcu_read_lock()`이 데드락 사이클에 연관될 수 있게 한다는 것입니다. 더 나아가서, 재귀적인 락이 없다 해도, RCU read-side 크리티컬 섹션들은 중첩될수가 없고, 마지막으로, 동시의 RCU 업데이트들이 공통의 grace period에 의해 원론적으로는 가능하지만, 이 구현은 grace period들을 직렬화 시켜서 grace-period 공유를 불가능하게 합니다.

Quick Quiz 9.49: Figure 9.47에서의 RCU 구현의 데드락이 다른 RCU 구현에서의 데드락이 될 수 없는 이유는 뭘까요? ■

Quick Quiz 9.50: 왜 Figure 9.47의 RCU 구현에서는 RCU 읽기 쓰레드들이 병렬로 수행될 수 있도록 간단하

¹¹ 하지만, 상품 품질의 사용자 레벨 RCU 구현은 구할 수 있습니다 [Des09].

게 reader-writer 락을 사용하지 않았나요? ■

이 구현이 실제 상품 구성에서 유용할 거라 생각하기는 어렵습니다만, 거의 모든 사용자 레벨 어플리케이션들에 구현할 수 있다는 장점을 가지고 있습니다. 나아가서, CPU 별로 하나의 락을 갖거나 reader-writer 락을 사용하는 비슷한 구현들은 2.4 리눅스 커널이라는 제품에서 사용된 바 있습니다.

이런 CPU 별로 하나의 락을 사용하는 전략의 수정된, 쓰레드별로 락을 하나씩 갖는 버전은 다음 섹션에서 설명됩니다.

9.5.5.2 Per-Thread Lock-Based RCU

Figure 9.48 (`rcu_lock_percpu.h` 와 `rcu_lock_percpu.c`) 는 쓰레드별로 하나의 락 두기에 기반한 구현을 보입니다. `rcu_read_lock()` 과 `rcu_read_unlock()` 함수들은 각각 현재 쓰레드의 락을 잡고 풁니다. `synchronize_rcu()` 함수는 각 쓰레드의 락들을 한번에 모두 잡고 풁니다. 따라서, `synchronize_rcu()` 가 시작될 때 수행중인 모든 RCU read-side 크리티컬 섹션들은 `synchronize_rcu()` 가 리턴하기 전에 완료되어야만 합니다.

이 구현은 동시의 RCU 읽기 쓰레드들을 가능하게 하고 하나의 글로벌 락을 사용할 때 생길 수 있는 데드락 조건을 예방하는 장점을 갖습니다. 더 나아가서, read-side 오버헤드는 비록 대략 140 나노세컨드 정도로 높긴 하지만 CPU 들의 수에 관계 없이 140 나노세컨드 정도로 유지됩니다. 하지만, 업데이트 쪽의 오버헤드는 하나의 Power5 CPU에서 600 나노세컨드부터 64 CPU에서의 100 마이크로세컨드 정도까지 움직입니다.

Quick Quiz 9.51: Figure 9.48 의 line 15-18 의 루프에서는 락들을 일단 모두 다 잡고나서는 한번에 모두 풀어주는게 더 깔끔하지 않나요? 무엇보다, 이렇게 바꾸면 어떤 읽기 쓰레드도 존재하지 않는 시점이 생기게 되어서 모든 것들이 매우 간단해질 겁니다. ■

Quick Quiz 9.52: Figure 9.48 에 보여진 구현은 dead-

```

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 9.47: Lock-Based RCU Implementation

```

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 9.48: 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 9.49: RCU Implementation Using Single Global Reference Counter

lock 으로부터 자유로울까요? 그렇다면, 또는 그렇지 않다면, 왜죠? ■

Quick Quiz 9.53: Figure 9.48 에 보인 RCU 알고리즘의, 예를 들어 POSIX pthread 처럼 여러곳에서 사용 가능한 기능들만을 사용하고 있는 점도 장점 아닌가요? ■

이 방법은 일부 환경에서는 유용할 수 있는데, 리눅스 2.4 커널에서도 비슷한 방법이 사용되었습니다 [MM00].

이어서 소개될 카운터 기반의 RCU 구현은 락 기반 구현의 일부 한계들을 해결합니다.

9.5.5.3 Simple Counter-Based RCU

약간 더 세련된 RCU 구현이 Figure 9.49 (`rcu_rcg.h` 와 `rcu_rcg.c`)에 보여져 있습니다. 이 구현은 line 1에 정의된 글로벌 레퍼런스 카운터 `rcu_refcnt`를 사용합니다. `rcu_read_lock()` 기능은 어토믹하게 이 카운터를 증가시키고, RCU read-side 크리티컬 섹션 이 이 어토믹 값 증가 뒤로 순서지어지는 것을 보장시킵니다. 유사하게, `rcu_read_unlock()`은 RCU read-side 크리티컬 섹션을 가두기 위해 메모리 배리어를 실행하고 어토믹하게 그 카운터를 감소시킵니다. `synchronize_rcu()` 기능은 레퍼런스 카운터에 감싸인 채 이 레퍼런스 카운터가 0이 되기를 기다립니다. Line 19의 `poll()`은 단순히 순수한 지연을 제공하고, 순수한 RCU 시맨틱의 관점에서는 허용될 수 있습니다. 다시, 일단 `synchronize_rcu()` 가 리턴하면 모든 앞의 RCU read-side 크리티컬 섹션들은 완료되었을 것이 보장됩니다.

Section 9.5.5.1에서 보여진 락 기반의 구현과 상반되는 장점으로, 이 구현은 RCU read-side 크리티컬 섹션들의 병렬 수행을 가능하게 합니다. Section 9.5.5.2에서 보여진 쓰레드별 락 기반의 구현과 상반되는 장점으로, 이 구현은 또한 그것들이 중첩될 수 있게 합니다. 또한, `rcu_read_lock()` 기능은 결코 스팬하지도 블록하지도 않기 때문에 데드락 사이클에 연관될 수 없습니다.

Quick Quiz 9.54: 하지만 `synchronize_rcu()` 을 감싸고 락을 잡고 같은 락을 RCU read-side 크리티컬 섹션 내에서 잡으면 어떻게 되죠? ■

하지만, 이 구현은 여전히 일부 심각한 한계점을 가지고 있습니다. 첫째로, `rcu_read_lock()` 과 `rcu_read_unlock()` 안의 어토믹 오퍼레이션들은 여전히 상당히 무겁고, read-side 오버헤드는 하나의 Power5 CPU에서의 100 나노세컨드부터 64-CPU 시스템에서의 약 40 마이크로세컨드의 범위를 갖습니다. 이는 RCU read-side 크리티컬 섹션들은 실제 read-side 병렬성을 갖기 위해서는 상당히 길어야 함을 의미합니다. 반면에 읽기 쓰레드들이 존재하지 않는다면 grace period는 40 나노세컨드 만에 끝나는데, 이는 리눅스 커널의 제품 품질의 구현보다 수십 수백배는 빠른 겁니다.

Quick Quiz 9.55: `synchronize_rcu()` 가 10-밀리세컨드 지연을 포함하고 있는데 어떻게 grace period가 40 나노세컨드 만에 끝날수가 있나요? ■

둘째로, 많은 동시의 `rcu_read_lock()` 과 `rcu_read_unlock()` 오퍼레이션들이 있다면, `rcu_refcnt`에 굉장한 메모리 경쟁이 발생할 거고, 이는 비싼 캐시 미스를 유발할 겁니다. 이 두개의 한계점들은 RCU의 주 목적인 read-side 동기화 기능에 낮은 오버헤드 제공하기를 매우 어렵게 만듭니다.

마지막으로, 긴 read-side 크리티컬 섹션들을 갖는 많은 수의 RCU 읽기 쓰레드들은 글로벌 카운터가 결코 0

```
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 9.50: RCU Global Reference-Count Pair 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 = 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 9.51: RCU Read-Side Using Global Reference-Count Pair

이 되지 못하게 해서 `synchronize_rcu()` 가 영원히 완료되지 못하게 만들 수도 있습니다. 이는 RCU 업데이트의 starvation을 유발할 수 있는데 이는 당연하게도 제품 환경에서는 받아들여질 수 없는 특성입니다.

Quick Quiz 9.56: Figure 9.49의 RCU 구현은 동시에 `synchronize_rcu()` 가 너무 오래 기다리고 있을 때에는 왜 간단히 `rcu_read_lock()` 을 기다리게 만들지 않는거죠? 그렇게 하면 `synchronize_rcu()` 의 starvation을 막을 수 있지 않나요? ■

따라서, 이 구현은 락 기반의 메커니즘보다는 예를 들어 고도의 스트레스 디버깅 환경에서 적합한 RCU 구현과 같은 잠재성이 약간 있긴 하지만, 제품 환경에서 유용할 거라고 생각하기는 여전히 어렵습니다. 다음 섹션은 쓰기 쓰레드들에 좀 더 신경 쓴 레퍼런스 카운팅 방법들을 설명합니다.

9.5.5.4 Starvation-Free Counter-Based RCU

Figure 9.51 (`rcu_rcgp.h`)는 Figure 9.50에 보여진 한쌍의 레퍼런스 카운터 (`rcu_refcnt[]`)와 그 한쌍 중 하나의 카운터를 고르는데 사용되는 글로벌 인덱스

(rcu_idx), 쓰레드별 중첩 수준 카운터 rcu_nesting, 쓰레드별 글로벌 인덱스의 스냅샷 (rcu_rad_idx), 그리고 하나의 글로벌 락을 사용하는 RCU 구현의 read-side 기능들을 보이고 있습니다.

Design 두개의 원소를 갖는 rcu_refcnt[] 배열이 야말로 starvation 으로부터 자유를 가져다 주는 그 무엇입니다. 핵심은 synchronize_rcu() 는 앞서 존재한 읽기 쓰레드들을 기다리는 것만이 요구된다는 점입니다. 만약 새로운 읽기 쓰레드가 이미 수행을 시작한 특정 synchronize_rcu() 인스턴스 뒤에 시작된다면, 그 synchronize_rcu() 인스턴스는 이 새로운 읽기 쓰레드를 기다려야 할 필요가 없습니다. 어떤 시점이든, 특정 읽기 쓰레드가 rCU_read_lock() 을 통해 자신의 RCU read-side 크리티컬 섹션을 들어갈 때에, 이 읽기 쓰레드는 rCU_idx 변수로 가리키는 rCU_refcnt[] 배열의 원소의 값을 증가시킵니다. 같은 읽기 쓰레드가 rCU_read_unlock() 을 통해 자신의 RCU read-side 크리티컬 섹션을 나갈 때에는 이 읽기 쓰레드는 rCU_idx 값에 가해졌을 수 있는 모든 뒤의 변경들을 무시한 채 자신이 증가시켰던 원소의 값을 감소시킵니다.

이 구성은 synchronize_rcu() 가 rCU_idx 의 값을 rCU_idx = !rcu_idx 식으로 보정함으로써 starvation 을 막을 수 있음을 의미합니다. rCU_idx 의 예전 값이 0이었고, 따라서 새로운 값은 1이 될 것이라고 가정해 봅시다. 이 값 보정 오퍼레이션 후에 도착하는 새로운 읽기 쓰레드들은 rCU_idx[1] 을 증가시킬 것이고, 그동안 앞서 rCU_idx[0] 를 증가시켰던 과거의 읽기 쓰레드들은 각자의 RCU read-side 크리티컬 섹션들을 나갈 때마다 rCU_idx[0] 을 감소시킬 겁니다. 이는 rCU_idx[0] 의 값은 더이상 증가하지 않을것이고, 따라서 단조롭게 감소를 하게 될것이라는 의미입니다.¹² 이는 모든 synchronize_rcu() 가 해야할 일은 rCU_refcnt[0] 가 0이 될 때까지 기다려야 하는 것뿐이란 의미입니다.

이 배경지식과 함께라면, 실제 기능들의 구현을 들여다 볼 준비가 되었습니다.

Implementation rCU_read_lock() 기능은 rCU_idx 로 인덱스되는 rCU_refcnt[] 배열의 멤버의 값을 어토믹하게 증가시키고, 이 인덱스의 스냅샷을 쓰레드별 변수인 rCU_read_idx 안에 보관합니다. rCU_read_unlock() 함수는 연관된 rCU_read_lock() 에서 값을 증가시켰던 카운터의 값을 어토믹하게 감소시킵니다. 하지만, rCU_idx 의 하나의 값만이 쓰레드

```

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 9.52: RCU Update Using Global Reference-Count Pair

별로 기억되기 때문에 read-side 크리티컬 섹션의 중첩을 허용하게 위해서는 추가적인 방법들이 필요합니다. 이런 추가적인 방법들은 중첩 정보를 쓰아가는 데에 쓰레드별 rCU_nesting 변수를 사용합니다.

이것들이 모두 동작하도록 하기 위해, Figure 9.51 의 rCU_read_lock() 의 line 6 에서는 현재 쓰레드의 rCU_nesting 인스턴스를 가져오고 line 7 에서는 지금 중첩된 rCU_read_lock() 중 가장 바깥에 있음을 확인한 후 line 8-10 에서는 rCU_idx 의 현재 값을 가져오고 이 값을 이 쓰레드의 rCU_read_idx 에 저장한 후 rCU_refcnt 의 선택된 원소의 값을 어토믹하게 증가시킵니다. rCU_nesting 의 값과는 무관하게 line 12 에서는 그 값을 증가시킵니다. Line 13 에서는 메모리 배리어를 실행해서 RCU read-side 크리티컬 섹션이 rCU_read_lock() 코드 앞으로 뛰어나오지 않게 막아줍니다.

비슷하게, rCU_read_unlock() 함수는 RCU read-side 크리티컬 섹션이 rCU_read_unlock() 코드 뒤로 빠져나가지 않도록 line 21 에서 메모리 배리어를 실행합니다. Line 22 에서는 이 쓰레드의 rCU_nesting 인스턴스를 가져오고 line 23 에서 지금 중첩된 rCU_read_unlock() 가운데 가장 바깥임을 확인하고, line 24 와 25 에서 이 쓰레드의 (중첩된 rCU_read_lock() 중 가장 바깥의 것에서 저장된) rCU_read_idx 인스턴스를 가져온 후 rCU_refcnt 의 선택된 원소의 값을 감소시킵니다. 중첩 단계와 관계없이, line 27 에서는 이 쓰레드의 rCU_nesting 인스턴스의 값을 감소시킵니다.

Figure 9.52 (rcu_rcpg.c) 는 이에 연관되는 synchronize_rcu() 구현을 보입니다. Line 6 과 19 는 두개 이상의 synchronize_rcu() 인스턴스가 동

¹² 이 “단조로운 감소” 문장이 무시하는 race condition 이 있습니다. 이 race condition 은 synchronize_rcu() 코드와 함께 다루겠습니다. 그전까지는, 의심을 면제주시길 바랍니다.

시에 수행되는 것을 막기 위해 `rcu_gp_lock` 을 각각 잡고 풁니다. Line 7-8 은 각각 `rcu_idx` 의 값을 가져오고 뒤따르는 `rcu_read_lock()` 이 앞의 인스턴스들과 다른 `rcu_idx` 원소를 사용하도록 보정합니다. Line 10-12 는 `rcu_idx` 로 가리켜지는 원소의 값이 0이 될때까지 기다리는데 앞서 line 9 에서의 메모리 배리어를 통해 `rcu_idx` 의 체크가 `rcu_idx` 의 보정 앞으로 튀어오르지 않게 합니다. Line 13-18 은 이 과정을 반복하고, line 20 은 어떤 뒤따르는 오퍼레이션들이 `rcu_refcnt` 의 검사를 앞질러 튀어오르지 않게 합니다.

Quick Quiz 9.57: Figure 9.52 의 `synchronize_rcu()` 의 line 5 의 메모리 배리어는 바로 뒤에 스판락 획득이 있는데도 왜 필요한거죠? ■

Quick Quiz 9.58: Figure 9.52 에서 카운터는 왜 두번 뒤집히는거죠? 한번의 뒤집고 기다리는 사이클만으로도 충분하지 않나요? ■

This implementation avoids the update-starvation issues that could occur in the single-counter implementation shown in Figure 9.49.

Discussion 여전히 심각한 한계점들이 있습니다. 첫째로, `rcu_read_lock()` 과 `rcu_read_unlock()` 의 어토믹 오퍼레이션들은 여전히 상당히 무겁습니다. 사실, 이것들은 Figure 9.49 의 하나의 카운터 사용 버전보다 더 복잡해서 read-side 기능들은 하나의 Power5 CPU 에서 약 150 나노세컨드를, 그리고 64-CPU 시스템에서는 약 40 마이크로세컨드를 소모합니다. 업데이트 쪽의 `synchronize_rcu()` 기능은 이보다도 비싸서, 하나의 Power5 CPU 에서는 200 나노세컨드 정도, 64-CPU 시스템에서는 40 마이크로세컨드 정도를 소모합니다. 이 말은 정말 read-side 병렬성을 얻기 위해서는 RCU read-side 크리티컬 섹션들이 매우 길어야만 한다는 뜻입니다.

둘째로, 많은 동시적인 `rcu_read_lock()` 과 `rcu_read_unlock()` 오퍼레이션들이 존재한다면, `rcu_refcnt` 원소들에의 극심한 경쟁이 만들어질 것이고, 이는 비싼 캐시 미스들을 유발할 것입니다. 이것은 더 나아가 병렬적인 read-side 액세스를 위해 필요시되는 RCU read-side 크리티컬 섹션 길이를 더욱 늘릴 겁니다. 이 두가지 한계점들은 대부분의 상황에서 RCU의 목적을 달성하기 어렵게 합니다.

셋째로, `rcu_idx` 를 두번이나 뒤집어야 하는 필요성은 업데이트 쪽에 상당한 오버헤드를 의미하는데, 특히 쓰레드의 수가 클때에 더욱 그러합니다.

마지막으로, 동시의 RCU 업데이트들이 동일한 grace period 로 처리될 수 있음에도 불구하고 이 구현은 grace period 들을 직렬화 시켜서 grace-period 공유를 불가능하게 합니다.

Quick Quiz 9.59: 어토믹 값 증가와 값 감소가 그렇게

```
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 9.53: RCU Per-Thread Reference-Count Pair 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 = 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 9.54: RCU Read-Side Using Per-Thread Reference-Count Pair

비싸다고 하면, Figure 9.51 의 line 10 에서는 어토믹하지 않은 값 증가를, line 25 에서는 어토믹하지 않은 값 감소를 하지 그래요? ■

이런 한계점들에도 불구하고, 누군가는 이 RCU 변종이 적은 수의 타이트하게 연결된 CPU 들 위에서는 더 복잡한 구현들과의 API 호환성을 유지하는 메모리 절약 구현으로는 사용될 수도 있을거라 생각할 겁니다. 하지만, 그건 적은 CPU 들 위로는 확장되지 못할 겁니다.

다음 섹션은 훨씬 개선된 read-side 성능과 확장성을 제공하는 또 다른 레퍼런스 카운팅 기반 방법을 설명합니다.

9.5.5.5 Scalable Counter-Based RCU

Figure 9.54 (`rcu_rcpl.h`) 는 쓰레드별 레퍼런스 카운터 한쌍을 사용하는 RCU 구현의 read-side 기능들을 보입니다. 이 구현은 Figure 9.51 에서 보인 구현과 상당히 비슷한데, 차이점은 `rcu_refcnt` 가 (Figure 9.53 에 보인 것과 같이) 쓰레드별 배열이라는 것 뿐입니다. 앞 섹션에서의 알고리즘처럼, 이 두개의 배열의 원소를 사용하는 것은 읽기 쓰레드들이 업데이트 쓰레드들을

```

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 9.55: RCU Update Using Per-Thread Reference-Count Pair

starvation에 빠뜨리는 것을 막습니다. 쓰레드별 `rcu_refcnt` 배열의 한가지 장점은 `rcu_read_lock()`과 `rcu_read_unlock()` 기능들이 더이상 어토믹 오퍼레이션들을 사용하지 않는다는 것입니다.

Quick Quiz 9.60: 집어쳐요! `rcu_read_lock()` 안에 `atomic_read()` 가 뺀히 보인다구요!!! 왜 `rcu_read_lock()` 이 어토믹 오퍼레이션을 포함하고 있지 않은 척 하시는거죠??? ■

Figure 9.55 (`rcu_rcpl.c`) 는 `synchronize_rcu()`의 구현을 `flip_counter_and_wait()`과 이름지어진 함수와 함께 보이고 있습니다. `synchronize_rcu()` 함수는 Figure 9.52에 보인 것과 유사합니다만, 반복된 카운터 뒤집기는 line 22와 23에서의 새로운 함수로의 호출 두개로 뒤바뀌었습니다.

이 새로운 `flip_counter_and_wait()` 함수는 `rcu_idx` 변수를 line 5에서 업데이트하고 line 6에서 메모리 배리어를 실행한 후 line 7-11에서 각 쓰레드의 기존 `rcu_refcnt` 원소가 0이 되기를 기다리며 스핀하게 됩니다. 일단 모든 그런 원소들이 0이 된다면, 이 함수는 line 12에서 다른 메모리 배리어를 치고 리턴합니다.

이 RCU 구현은 그 소프트웨어 환경에 중요한 새로운 요구사항을 내포하고 있는데, 이는 (1) 쓰레드별 변수를 선언할 수 있어야 하고, (2) 이 쓰레드별 변수들은 다른 쓰레드들에서도 접근할 수 있어야 하며, (3) 모든 쓰레드의 것들을 하나하나 열거하기가 가능해야 한다는 것입니다. 이런 요구사항들은 거의 모든 소프트웨어 환경

```

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 9.56: RCU Read-Side Using Per-Thread Reference-Count Pair and Shared Update Data

에서 충족됩니다만 종종 전체 쓰레드의 갯수의 상한선이 고정되는 결과가 나오곤 합니다. 더 복잡한 구현은 그런 상한선을 제거할 수 있는데, 예를 들면 확장 가능한 해시 테이블을 사용하거나 하는 형태입니다. 그런 구현은 동적으로 쓰레드들의 정보를 추적할 수 있을텐데, 예를 들면 각 쓰레드들의 정보를 `rcu_read_lock()` 호출 시에 추가하는 형태일 겁니다.

Quick Quiz 9.61: 좋아요, 우리가 N 쓰레드들을 가지고 있다면 우리는 $2N$ 10 밀리세컨드의 기다리는 호출 (`flip_counter_and_wait()` 당 한 세트) 을 할 수 있을텐데, 우리가 각 쓰레드를 위해 오직 한번 기다린다고 가정해도 그렇습니다. ■

이 구현은 여전히 일부 한계점을 가지고 있습니다. 먼저, `rcu_idx`를 두번 뒤집어야 하는 필요성은 업데이트 쪽에 상당한 오버헤드를 내포하고 있는데, 특히 많은 수의 쓰레드가 있을 때 그렇습니다.

두번째로 `synchronize_rcu()`는 이제 쓰레드들의 수가 늘어남에 따라 선형적으로 늘어나는 여러 변수들을 조사해야 하는데, 이는 쓰레드의 수가 많은 아플리케이션에 상당한 오버헤드가 있을 것을 암시합니다.

세번째로, 앞에서와 마찬가지로 동시의 RCU 업데이트들은 원론적으로는 공통의 grace period를 사용할 수 있음에도 이 구현은 grace period들을 직렬화 시켜서 grace period 공유를 막고 있습니다.

마지막으로, 글에서 이야기했듯, 쓰레드별 변수의 필요성과 쓰레드들을 모두 봐야 한다는 필요성은 일부 소프트웨어 환경에서는 문제가 될 수 있습니다.

그렇다고는 하나, read-side 기능들은 매우 잘 확장되는데, 단일 CPU에서 돌아가든 64-CPU Power5 시스템에서 돌아가든 상관없이 약 115 나노세컨드를 필요로 합니다. 앞에서 이야기한 바와 같이, `synchronize_rcu()` 기능은 확장되지 못하는데, 그 오버헤드는 단일 Power5 CPU에서의 약 1 마이크로세컨드를, 그리고 64-CPU 시스템에서는 200 마이크로세컨드를 보입니다. 이 구현은 생각건대 제품 품질의 사용자 수준 RCU 구현에 대한 토대가 될 수도 있을 것입니다.

다음 섹션에서는 더 효과적인 동시의 RCU 업데이트들을 가능하게 하는 알고리즘 하나를 설명합니다.

```

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 9.57: RCU Read-Side Using Per-Thread Reference-Count Pair and Shared Update

9.5.5.6 Scalable Counter-Based RCU With Shared Grace Periods

Figure 9.57 (rcu_rcpls.h) 는 기존과 같이 쓰레드별 레퍼런스 카운트 쌍을 사용하는, 하지만 업데이트들이 grace period 들을 공유할 수 있게 하는 RCU 구현을 위해 read-side 쪽의 기능들을 보입니다. Figure 9.54로 보인 앞의 구현과 주요한 차이점은 rcu_idx 가 이제는 자유롭게 수를 셀 수 있는 long 이어서, Figure 9.57의 line 8 은 아래쪽의 비트만을 집어내서 사용해야만 합니다. 또한 atomic_read() 와 atomic_set() 의 사용에서 ACCESS_ONCE() 로 바꿨습니다. Figure 9.56 에 보인 것처럼, 데이터 또한 상당히 비슷한데 rcu_idx 는 atomic_t 가 아니라 long 으로 바뀌었습니다.

Figure 9.58 (rcu_rcpls.c) 는 synchronize_rcu() 와 거기에 도움을 주는 flip_counter_and_wait() 함수를 보입니다. 이것들은 Figure 9.55 에 있던 것들과 비슷합니다. flip_counter_and_wait() 의 차이점에는 다음과 같은 것들이 있습니다:

1. Line 6 는 atomic_set() 대신 ACCESS_ONCE() 를 사용하고 보정을 하는 대신 값 증가를 시킵니다.
2. 새로운 line 7 은 카운터의 가장 아래 비트를 꺼내서 사용합니다.

synchronize_rcu() 의 변경은 더 많습니다:

```

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 9.58: RCU Shared Update Using Per-Thread Reference-Count Pair

1. 락을 잡기 전의 `rcu_idx`의 값을 Line 23에서 저장해 두는, 새로운 로컬 변수 `oldctr`이 존재합니다.
2. Line 26에서는 `atomic_read()` 대신에 `ACCESS_ONCE()`를 사용합니다.
3. Line 27-30은 락을 잡는 동안 다른 쓰레드에 의해 카운터 뒤집기가 최소 세번 이상 이루어졌는지를 확인해보고, 만약 그렇다면 락을 놓고 메모리 배리어를 수행한 후 리턴합니다. 이 경우, 카운터가 0이 될 때까지 두번의 완전한 기다림이 있었던 것이고, 따라서 다른 쓰레드들은 이미 모든 필요한 일들을 했습니다.
4. Line 33-34에서, `flip_counter_and_wait()`는 락이 잡히는 동안 두번 미만의 카운터 뒤집기가 이루어졌을 때에만 `flip_counter_and_wait()`를 수행합니다. 한편으로는, 두번의 카운터 뒤집기가 있었다면, 일부 다른 쓰레드는 모든 카운터가 0으로 갈 때까지 완전한 기다림을 수행했으므로, 한번만 더 하면 됩니다.

이 방법에서는 임의의 많은 수의 쓰레드들이 각 쓰레드별로 CPU 하나를 가지고 `synchronize_rcu()`를 동시에 수행하게 되면, 카운터가 0이 되기를 총 세번만 기다리게 될 것입니다.

이 개선에도 불구하고, 이 RCU 구현은 여전히 몇 가지 한계점들을 가지고 있습니다. 첫째로, 앞에서와 같이 `rcu_idx`를 두번 뒤집어야 하는 필요성은 업데이트 쪽에 상당한 오버헤드를 암시하는데, 많은 수의 쓰레드들이 존재할 때 특히 그렇습니다.

둘째로, 각 업데이트 쓰레드는 여전히 `rcu_gp_lock`을 잡는데, 할 일이 없을 때조차도 그렇습니다. 이는 수많은 동시의 업데이트들이 존재한다면 상당한 확장성 한계점으로 작용할 수 있습니다. 리눅스 커널의 제품 품질의 RCU 리얼타임 구현 [McK07a]에서 사용되었던 것처럼, 이 문제를 막는 방법들은 존재합니다.

세번째로, 이 구현은 쓰레드별 변수와 쓰레드들을 모두 돌아봐야 하는 기능을 필요로 하는데, 이는 역시 일부 소프트웨어 환경에서는 문제가 될 수 있습니다.

마지막으로, 32-bit 머신에서는 특정 업데이트 쓰레드가 `rcu_idx` 카운터가 오버헤드 되도록 긴 시간동안 `preemption` 당할 수 있습니다. 이는 그런 쓰레드가 불필요한 카운터 뒤집기를 하도록 하게 만들 수도 있습니다. 하지만, 각각의 `grace period`가 1 마이크로세컨드 만을 가진다 하더라도, 문제의 쓰레드는 한시간 이상을 `preemption` 당할 수 있어서 추가적인 카운터 뒤집기만 걱정해도 될 가능성이 큽니다.

Section 9.5.5.3에서 설명한 구현에서와 같이, 이 `read-side` 기능들은 상당히 잘 확장되는데, CPU의 수에 관계없이 대략 115 나노세컨드의 오버헤드를 갖습니다.

```

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 9.59: Data for Free-Running Counter Using RCU

`synchronize_rcu()` 기능은 여전히 비싼 편인데, 1 마이크로세컨드에서 16 마이크로세컨드 사이의 오버헤드를 갖습니다. 이는 Section 9.5.5.5의 구현에서 가졌던 대략 200 마이크로세컨드의 오버헤드에 비하면 더도 덜도 아니고 훨씬 저렴한 비용입니다. 따라서, 그 제한점들에도 불구하고, 이 RCU 구현은 실제 삶에서 접하는 활용처의 제품에서 사용될 수도 있을 거라고 생각될 수 있을 겁니다.

Quick Quiz 9.62: 이 모든 장난감 RCU 구현들은 `rcu_read_lock()`과 `rcu_read_unlock()` 안에 어토믹 오퍼레이션들을 가지고 있거나 쓰레드의 수에 따라 선형적으로 증가하는 `synchronize_rcu()`의 오버헤드를 갖습니다. 어떤 환경에서라면 RCU 구현이 이 세개의 기능들이 모두 결정적인 ($O(1)$)의 오버헤드와 대기시간을 갖는 가벼운 구현을 가질 수 있을까요?

Figure 9.57로 돌아가서, 우리는 하나의 전역 변수 접근과 `thread-local` 변수들로의 네번보다 적지 않은 액세스들이 존재함을 볼 수 있습니다. POSIX 쓰레드를 구현하는 시스템에서의 상대적으로 비싼 비용의 `thread-local` 액세스들을 놓고 보면, 이 세개의 `thread-local` 변수들을 하나의 구조체로 만들어서 `rcu_read_lock()`과 `rcu_read_unlock()`의 `thread-local` 데이터로의 접근을 하나의 `thread-local-storage`로의 접근으로 바꾸고 싶을 수 있을 겁니다. 하지만, 그보다도 더 나은 접근은 다음 섹션에서 하듯이, `thread-local` 액세스의 수를 하나로 바꾸는 것일 겁니다.

9.5.5.7 RCU Based on Free-Running Counter

Figure 9.60 (`rcu.h` and `rcu.c`)는 Figure 9.59에 보여진 데이터와 함께 짹수 값만을 갖는 글로벌한 자유롭게 동작하는 카운터에 기반한 RCU 구현을 보입니다. 이렇게 만들어진 `rcu_read_lock()` 구현은 상당히 단순합니다. Line 3과 4에서는 단순히 글로벌한 자유롭게 동작하는 `rcu_gp_ctr` 변수에 1을 더한 값으로 만들어진 훌수를 쓰레드별 변수인 `rcu_reader_gp`에 저장합니다. Line 5는 뒤따르는 RCU read-side 크리티컬 섹션의 내용이 “빼져나오는 것”을 방지하기 위해 메모리 배리어를 실행합니다.

`rcu_read_unlock()` 구현 역시 비슷합니다. Line 10에서는 이번에도 앞의 RCU read-side 크리티컬 섹션이 “빼져나오는 것”을 방지하기 위해 메모리 배리어를 실행합니다. Line 11과 12는 `rcu_gp_ctr`

```

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 9.60: Free-Running Counter Using RCU

글로벌 변수의 값을 쓰레드별 변수인 `rcu_reader_gp`에 복사해서 이 쓰레드별 변수의 값을 짹수로 만들었으나, 인해 동시에 `synchronize_rcu()` 인스턴스가 이를 무시해도 좋을 것임을 알립니다.

Quick Quiz 9.63: 짹수만으로도 해당 테스크를 무시해도 좋다고 `synchronize_rcu()`에 말하기에 충분하다면 Figure 9.60의 line 10과 11은 왜 단순히 `rcu_reader_gp`에 0을 할당하지 않나요? ■

따라서, `synchronize_rcu()`는 이 모든 쓰레드별 `rcu_reader_gp` 변수들이 짹수를 가지게 될 때까지 기다릴 수도 있습니다. 하지만, 그보다 훨씬 나은 방법이 있는데 `synchronize_rcu()`는 전부터 존재한 RCU read-side 크리티컬 섹션들만 기다리면 그만이기 때문입니다. Line 19에서는 앞의 RCU로 보호되는 데이터 구조의 조정이 line 21의 값 증가 뒤로 (CPU에 의해 서든 컴파일러에 의해서든) 재배치 되는 것을 막기 위해 메모리 배리어를 실행합니다. Line 20에서는 여러 개의 `synchronize_rcu()` 인스턴스들이 동시에 수행되는 것을 막기 위해 `rcu_gp_lock`을 획득합니다 (그리고 line 30에서 이를 해제합니다). Line 21은 글로벌 변수인 `rcu_gp_ctr`의 값을 2 만큼 증가시켜서 모든 전부티 존재해온 RCU read-side 크리티컬 섹션들이 갖는 그들과 연관된 쓰레드별 변수 `rcu_reader_gp`는 `rcu_gp_ctr` 보다 작은 값을 가지게 될 것입니다. 짹수 값의 `rcu_reader_gp`를 갖는 쓰레드들은 RCU read-side

크리티컬 섹션에 들어가 있지 않다는 점을 다시 상기해서, line 23-29는 `rcu_reader_gp` 값들을 그 모든 것들이 짹수이거나 (line 24) 글로벌한 `rcu_gp_ctr` 보다 큰 동안 (line 25-26) 스캔합니다. Line 27에서는 앞서 존재해온 RCU read-side 크리티컬 섹션을 기다리기 위해 짧은 시간동안 블록하지만 만약 grace-period 대기시간이 중요하다면 스핀 루프로 교체될 수 있습니다. 마지막으로, line 31에서의 메모리 배리어는 뒤따르는 모든 오브젝트 해체가 루프의 앞으로 재배치 되지 않도록 보장합니다.

Quick Quiz 9.64: Figure 9.60의 line 19와 31에서의 메모리 배리어들은 왜 필요한 건가요? line 20과 30에서의 락킹이 충분한데 메모리 배리어는 피할 수 있지 않나요? ■

이 전략은 훨씬 나은 read-side 성능을 달성할 수 있게 하는데, Power5 CPU들의 숫자에 관계 없이 대략 63 나노세컨드의 오버헤드를 갖습니다. 업데이트는 더 많은 오버헤드를 갖는데, 하나의 Power5 CPU에서 500 나노세컨드로부터 64 개의 CPU에서 100 마이크로세컨드 까지 같습니다.

Quick Quiz 9.65: Section 9.5.6에서 설명한 update-side batching ◎ Figure 9.60에 보인 구현에 적용될 수도 있지 않을까요? ■

이 구현은 앞서 언급한 많은 update-side 오버헤드 외에도 몇 가지 심각한 한계점들로 인해 문제가 될 수 있습니다. 먼저, 더이상 중첩된 RCU read-side 크리티컬 섹션들이 불가능한데, 이에 대해선 다음 섹션에서 다루도록 합니다. 둘째로, 읽기 쓰레드가 `rcu_gp_ctr`에서 값을 읽어왔지만 `rcu_reader_gp`에 값을 저장하기 전에 Figure 9.60의 line 3에서 `preemption` 당하고 `rcu_gp_ctr` 카운터가 가질 수 있는 값들보다는 적게, 하지만 그 절반보다는 많이 값 증가를 반복한다면, `synchronize_rcu()`는 이 뒤의 RCU read-side 크리티컬 섹션을 무시하게 될겁니다. 마지막으로, 이 구현은 쓰레드별 변수들을 유지할 수 있고 모든 쓰레드들을 검사할 수 있는 소프트웨어 환경에 들어가 있을 것을 필요로 합니다.

Quick Quiz 9.66: Figure 9.60의 line 3-4에서 읽기 쓰레드들이 `preemption` 당할 수 있다는 사실은 진짜 문제일까요? 달리 말하자면, 정말 실패를 만들 수 있는 실제의 이벤트들이 존재하나요? 그렇지 않다면, 왜죠? 그렇다면, 그 이벤트들은 어떤 것이고, 그 실패는 어떻게 해결될 수 있을까요? ■

9.5.5.8 Nestable RCU Based on Free-Running Counter

Figure 9.62 (`rcu_nest.h` and `rcu_nest.c`)는 하나의 글로벌한 자유롭게 동작하는 카운터에 기반하지만 RCU read-side 크리티컬 섹션들의 중첩을 허용

```

1 #define 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 9.61: Data for Nestable RCU Using a Free-Running Counter

하는 RCU 구현을 보입니다. 이 중첩 가능성은 Figure 9.61에 보인 정의들을 사용해서 글로벌한 `rcu_gp_ctr`의 아래쪽 비트들을 중첩을 세는데 사용하도록 전용화 함으로써 이루어집니다. 이는 Section 9.5.5.7에서 보인, 아래쪽의 하나의 비트를 중첩 단계를 세는데에 사용한 것으로 생각될 수 있는 방법의 일반화된 방법입니다. 이를 위해 두개의 C 프리프로세서 매크로인 `RCU_GP_CTR_NEST_MASK`와 `RCU_GP_CTR_BOTTOM_BIT`이 사용됩니다. 이 매크로들은 연관되어 있습니다: `RCU_GP_CTR_NEST_MASK=RCU_GP_CTR_BOTTOM_BIT-1`. `RCU_GP_CTR_BOTTOM_BIT` 매크로는 중첩 단계를 세기 위해 전용화된 비트들의 바로 앞에 자리잡은 하나의 비트를 가지고 있으며, `RCU_GP_CTR_NEST_MASK`는 `rcu_gp_ctr`에서 중첩 단계를 세기 위해 사용되는 지역에 모두 1 비트를 채워 가지고 있습니다. 분명하게, 두개의 C 프리프로세서 매크로들은 요구되는 최대의 RCU read-side 크리티컬 섹션들의 중첩을 허용하기 충분한 만큼의 카운터의 아래쪽 비트들을 가지고 있어야 하고, 이 구현은 대부분의 어플리케이션들에서는 충분할 만한 크기인 최대 127 단계의 RCU read-side 크리티컬 섹션 중첩을 위해 일곱개의 비트를 준비해 두고 있습니다.

그렇게 해서 만들어진 `rcu_read_lock()` 구현은 여전히 합리적인 수준으로 간단합니다. Line 6에서는 이 쓰래드의 `rcu_reader_gp` 인스턴스로의 포인터를 지역 변수인 `rrgp`로 가져와서 비싼 `pthreads` `thread-local-state` API의 호출 횟수를 최소화 합니다. Line 7에서는 `rcu_reader_gp`의 현재 값을 또 다른 지역 변수인 `tmp`에 기록하고, line 8에서 그 아래쪽 비트들이 0인지 확인하는데, 이는 이 쓰래드가 가장 바깥의 `rcu_read_lock()`을 수행중인지 확인하게 됩니다. 만약 그렇다면, line 9에서는 line 7에서 앞서 가져왔던 현재의 값을 더이상 유효하지 않을 수 있으므로 글로벌 변수인 `rcu_gp_ctr`를 `tmp`에 넣습니다. 어떤 경우든, line 10에서는 중첩 단계를 증가시키는데, 이는 카운터의 아래쪽 7개의 비트들에 저장됩니다. Line 11에서는 업데이트된 카운터를 이 쓰래드의 `rcu_reader_gp` 인스턴스에 업데이트하고, 마지막으로 line 12에서는 RCU read-side 크리티컬 섹션이 앞의 `rcu_read_lock()` 안으로 빠져나오지 않도록 메모리 배리어를 실행합니다.

```

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) +=
30         RCU_GP_CTR_BOTTOM_BIT;
31     smp_mb();
32     for_each_thread(t) {
33         while (rcu_gp_ongoing(t) &&
34             ((per_thread(rcu_reader_gp, t) -
35             rcu_gp_ctr) < 0)) {
36             poll(NULL, 0, 10);
37         }
38     }
39     spin_unlock(&rcu_gp_lock);
40     smp_mb();
41 }

```

Figure 9.62: Nestable RCU Using a Free-Running Counter

달리 말하면, 이 `rcu_read_lock()` 구현은 현재의 `rcu_read_lock()`의 실행이 RCU read-side 크리티컬 섹션 안에서 중첩되어 있지 않다면 글로벌 변수 `rcu_gp_ctr`의 복사본을 가져오고, 그렇지 않다면 현재 쓰래드의 `rcu_reader_gp`의 값을 가져옵니다. 어떤 경우든, 추가적인 중첩 단계를 기록하기 위해 뭐가 가져와졌든 가져온 값을 증가시키고 그 결과를 현재 쓰래드의 `rcu_reader_gp` 인스턴스에 저장합니다.

흥미롭게도, 이 `rcu_read_lock()`의 차이점에도 불구하고, `rcu_read_unlock()`의 구현은 Section 9.5.5.7에 보인 것과 상당히 유사합니다. Line 19에서는 RCU read-side 크리티컬 섹션이 뒤따르는 `rcu_read_unlock()`으로 빠져나오는 걸 막기 위해 메모리 배리어를 실행하고, line 20에서 이 쓰래드의 `rcu_reader_gp`의 인스턴스를 값 감소시키는데, 이는 `rcu_reader_gp`의 아래쪽 비트들에 저장되어 있는 중첩 단계 카운트를 감소시키는 효과를 갖습니다. 이 기능의 디버깅 버전은 (값을 감소시키기 전에!) 이 아래

```

1 DEFINE_SPINLOCK(rcu_gp_lock);
2 long rcu_gp_ctrl = 0;
3 DEFINE_PER_THREAD(long, rcu_reader_qs_gp);

```

Figure 9.63: Data for Quiescent-State-Based RCU

쪽 비트들이 0이 아닌지 검사할 수도 있을 겁니다.

`synchronize_rcu()` 의 구현은 Section 9.5.5.7에 보인 것과 상당히 유사합니다. 그 사이엔 두가지 차이가 있습니다. 첫번째는 line 29 와 30에서 상수 “2”가 아니라 `RCU_GP_CTR_BOTTOM_BIT` 을 `rcu_gp_ctrl`에 더 한다는 것이고, 두번째는 line 33에서의 비교가 조건에 관계없이 아래쪽 비트를 검사하는게 아니라 `RCU_GP_CTR_BOTTOM_BIT` 을 검사하는 별도의 함수로 추상화 되었다는 점입니다.

이 방법은 Section 9.5.5.7에 보인 것과 거의 유사한 `read-side` 성능을 달성하는데, Power5 CPU의 수와 관계 없이 대략 65 나노세컨드의 오버헤드를 보입니다. 업데이트는 이번에도 더 많은 오버헤드를 갖는데, 하나의 Power5 CPU에서 약 600 나노세컨드부터 64 CPU에서 약 100 마이크로세컨드를 갖습니다.

Quick Quiz 9.67: 이 복잡한 비트 조정을 하는 대신에 앞의 섹션에서 그랬듯이 별도의 쓰레드별 중첩 수준 변수를 갖지 않는 건가요? ■

이 구현은 RCU `read-side` 크리티컬 섹션들의 중첩이 이제는 가능하다는 점을 제외하고는 Section 9.5.5.7에서의 것과 동일한 한계점들로 문제가 될 수 있습니다. 또한, 32 비트 시스템에서라면, 이 전략은 `global rcu_gp_ctrl` 변수를 오버플로우 시키는데 필요한 시간을 더 줄여버릴 수 있습니다. 다음 섹션은 오버플로우가 일어나는데 필요한 시간을 훨씬 증가시키면서도 `read-side` 오버헤드는 훨씬 줄이는 한가지 방법을 보입니다.

Quick Quiz 9.68: Figure 9.62에 보여진 알고리즘에서, 어떻게 하면 전역 변수인 `rcu_gp_ctrl` 가 오버플로우 되는데 걸리는 시간을 두배로 늘릴 수 있을까요? ■

Quick Quiz 9.69: 다시, Figure 9.62에 보여진 알고리즘에서, 카운터 오버플로우는 치명적인가요? 그 이유는 무엇이죠? 만약 치명적이라면, 그걸 고치기 위해 뭘 할 수 있을까요? ■

9.5.5.9 RCU Based on Quiescent States

Figure 9.64 (`rcu_qs.h`)는 조용한 상태에 기반한 사용자 레벨 RCU 구현을 만드는데에 Figure 9.63에 보인 데이터와 함께 사용되는 `read-side` 기능들을 보입니다. 그림의 line 1-7에서 볼 수 있듯이, `rcu_read_lock()` 과 `rcu_read_unlock()` 기능들은 아무일도 하지 않고, 따라서 인라인 함수로 바뀌고 최적화 단계에서 아예 사라져서 버릴 거라 예상할 수 있고, 리눅스 커널의

```

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_ctrl) + 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_ctrl);
22     smp_mb();
23 }
24
25 static void rcu_thread_online(void)
26 {
27     rcu_quiescent_state();
28 }

```

Figure 9.64: Quiescent-State-Based RCU Read Side

서버 빌드에서는 그렇습니다. 이는 조용한-상태-기반의 RCU 구현들은 RCU `read-side` 크리티컬 섹션들의 양을 앞서 언급한 조용한 상태들을 사용해서 어림잡는다는 사실 때문입니다. 이러한 각각의 조용한 상태들은 `rcu_quiescent_state()` 호출을 포함하는데, 이 함수는 이 그림의 line 9-15에 보여져 있습니다. 확장된 조용한 상태에 들어가는 쓰레드는 (예를 들어, 블록되었을 때) 대신 `rcu_thread_offline()` (line 17-23)을 호출하고 그로부터 빠져나올 때에는 `rcu_thread_online()` (line 25-28)을 호출할 수 있습니다. 그런식으로, `rcu_quiescent_state()`는 `rcu_read_lock()`과 유사하고 `rcu_thread_offline()`은 `rcu_read_unlock()`과 유사합니다. 또한, `rcu_quiescent_state()`는 `rcu_thread_online()` 뒤에 곧바로 `rcu_thread_offline()`이 이어지는 것으로 생각될 수 있습니다.¹³ RCU `read-side` 크리티컬 섹션 내에서 `rcu_quiescent_state()`, `rcu_thread_offline()`, 또는 `rcu_thread_online()`을 호출하는 건 금지된 행위입니다.

`rcu_quiescent_state()`에서, line 11에서는 이 `quiescent state` (조용한 상태) 앞의 (RCU `read-side` 크리티컬 섹션들을 포함해서) 어떤 코드가 `quiescent state` 앞으로 재배치 되는 것을 막기 위해 메모리 배리어를 칩니다. Line 12-13에서는 글로벌 변수 `rcu_gp_ctrl`

¹³ 비록 이 그림의 코드가 `rcu_quiescent_state()`가 `rcu_thread_online()` 뒤에 곧바로 `rcu_thread_offline()`이 따라오는 것과 같지만, 이 관계는 성능 최적화로 인해 불투명해집니다.

의 복사본을 가져오는데, `rcu_gp_ctrl` 가 두번 이상 가져와지는 최적화가 만들어지거나 하지 않게 `ACCESS_ONCE()` 를 사용합니다. 그리고 나서 가져온 값에 1을 더하고 쓰레드별 변수인 `rcu_reader_qs_gp` 에 저장해서 모든 동시에 수행되는 `synchronize_rcu()` 인스턴스들은 훌수를 볼 수 있게 해서, 새로운 RCU read-side 크리티컬 섹션이 시작되었음을 알 수 있게 합니다. 예전의 RCU read-side 크리티컬 섹션들을 기다리는 `synchronize_rcu()` 의 인스턴스들은 따라서 이것을 무시해도 좋을 것임을 알 수 있게 됩니다. 마지막으로, line 14 에서는 메모리 배리어를 실행해서 (RCU read-side 크리티컬 섹션을 포함해서) 뒤따르는 코드가 line 12-13 과 재배치 되는 일을 방지합니다.

Quick Quiz 9.70: Figure 9.64 의 line 14 에 보인 추가적인 메모리 배리어는 `rcu_quiescent_state` 의 오버헤드를 많이 늘리지 않을까요? ■

어떤 어플리케이션들은 RCU 를 가끔씩만 쓰지만, 사용할 때에는 매우 자주 사용할 수도 있습니다. 그런 어플리케이션들은 RCU 를 사용하기 시작할 때 `rcu_thread_online()` 을 사용하고 더이상 RCU 를 사용하지 않을 때에 `rcu_thread_offline()` 을 사용할 수 있을 겁니다. `rcu_thread_offline()` 호출과 뒤따르는 `rcu_thread_online()` 사이의 시간은 하나의 확장된 quiescent state 여서, RCU 는 이 시간동안 명시적으로 quiescent state 가 등록되지는 않을 것이라고 예상할 수 있습니다.

`rcu_thread_offline()` 함수는 쓰레드별 변수 `rcu_reader_qs_gp` 변수를 `rcu_gp_ctrl` 의 현재 값으로 설정하는데, 이 값은 짹수 값을 가질 겁니다. 따라서 동시에 수행되는 `synchronize_rcu()` 인스턴스들은 이 쓰레드를 무시해도 됨을 알 것입니다.

Quick Quiz 9.71: Figure 9.64 의 line 19 와 22 의 메모리 배리어들은 왜 필요한 건가요? ■

`rcu_thread_online()` 은 단순히 `rcu_quiescent_state()` 를 호출해서 연장된 quiescent state 의 종료를 표시합니다.

Figure 9.65 (`rcu_qs.c`) 는 `synchronize_rcu()` 의 구현을 보이는데, 이는 앞 섹션들에서의 것과 상당히 유사합니다.

이 구현은 `rcu_read_lock()`-`rcu_read_unlock()` 왕복에 대략 50 피코세컨드 라는 엄청나게 빠른 read-side 기능들을 갖습니다. `synchronize_rcu()` 의 오버헤드는 단일 CPU 의 Power5 시스템에서 600 나노세컨드부터 64-CPU 시스템에서 100 마이크로세컨드 까지의 오버헤드를 갖습니다.

Quick Quiz 9.72: 분명히 해두겠는데, 2008 년의 Power 시스템들의 클락 주파수는 상당히 높았지만, 5GHz 클락 주파수도 루프가 50 피코세컨드만에 실행되게 하는데에는 부족해요! 무슨 일이 벌어진 거죠? ■

```

1 void synchronize_rcu(void)
2 {
3     int t;
4
5     smp_mb();
6     spin_lock(&rcu_gp_lock);
7     rcu_gp_ctrl += 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_ctrl) < 0)) {
13             poll(NULL, 0, 10);
14         }
15     }
16     spin_unlock(&rcu_gp_lock);
17     smp_mb();
18 }
```

Figure 9.65: RCU Update Side Using Quiescent States

하지만, 이 구현은 각 쓰레드가 주기적으로 `rcu_quiescent_state()` 를 호출하거나 연장된 quiescent state 를 위해 `rcu_thread_offline()` 을 호출해 줄것을 필요로 합니다. 이 함수들을 주기적으로 실행해야 하는 필요성은 이 구현을 특정 타입의 라이브러리 함수들과 같은 일부 환경에서는 사용하기 어렵게 할 수 있습니다.

Quick Quiz 9.73: 코드가 라이브러리에 있다는 사실이 왜 Figure 9.64 와 9.65 에 보여진 RCU 구현의 사용에 어려움을 가져올 수 있는 건가요? ■

Quick Quiz 9.74: 하지만 락을 `synchronize_rcu()` 전후에 걸쳐 잡고, 같은 락을 RCU read-side 크리티컬 섹션에서 잡으면 어떻게 되나요? 이건 데드락이 되어야 할텐데, 하지만 어떻게 어떤 코드도 만들지 않는 기능이 데드락 사이클에 참여될 수가 있죠? ■

또한, 이 구현은 동시에 `synchronize_rcu()` 호출이 grace period 를 공유할 수 있도록 허용하지 않습니다. 그렇다면 하나, 이 버전의 RCU 에 기초해서 제품 품질의 RCU 구현을 쉽게 생각해 볼 수 있을 겁니다.

9.5.5.10 Summary of Toy RCU Implementations

여기까지 잘 도착했다면, 축하합니다! 이제 독자 여러분은 RCU 자체에 대해서만이 아니라 그걸 둘러싸고 있는 소프트웨어 환경들과 어플리케이션들의 요구사항들에 대해서도 분명한 이해를 가지게 되었을 겁니다. 이보다도 더 깊은 이해를 원하는 분들은 제품 품질의 RCU 구현에 대한 설명을 읽어보시기 바랍니다 [DMS¹², McK07a, McK08a, McK09c].

앞의 섹션들에서는 다양한 RCU 기능들의 바랄법한 속성들을 열거해 보았습니다. 새로운 RCU 구현을 만들고자 하는 분들이 쉽게 참조해 볼 수 있도록 다음의 리스트를 제공합니다.

1. Read-side 기능들 (`rcu_read_lock()` 과 `rcu`

`read_unlock()` 과 같은 것들)과 grace-period 기능들 (`synchronize_rcu()` 와 `call_rcu()` 와 같은 것들)이 반드시 있어야만 하는데, 그것들은 grace period 의 시작 시점에 존재한 read-side 크리티컬 섹션은 모두 그 grace period 의 종료 전까지는 완료되어야 합니다.

2. RCU read-side 기능들은 최소한의 오버헤드만을 가져야 합니다. 구체적으로는, 캐시 미스, 어토믹 인스트럭션, 메모리 배리어, 그리고 브랜칭과 같은 비싼 오퍼레이션들은 방지되어야 합니다.
3. 리얼타임 쪽에서의 사용을 위해선 RCU read-side 기능들은 $O(1)$ 의 계산 복잡도를 가져야 합니다. (이는 읽기 쓰레드들은 업데이트 쓰레드들과 동사에 수행됨을 암시합니다.)
4. RCU read-side 기능들은 모든 컨텍스트에서 사용이 가능해야 합니다 (리눅스 커널에서는, idle 루프를 제외한 모든 곳에서 사용이 가능합니다). 중요한 특수 케이스는 RCU read-side 기능들이 RCU read-side 크리티컬 섹션 내부에서 사용되는 것으로, 달리 말하면, RCU read-side 크리티컬 섹션들은 중첩될 수 있어야 합니다.
5. RCU read-side 기능들은 무조건적으로 수행되어서 실패를 리턴하거나 하지 않아야 합니다. 이 속성은 굉장히 중요한데, 실패여부 검사는 복잡성을 증가시키고 테스트와 검증을 번거롭게 만들기 때문입니다.
6. Quiescent state (그리고 grace period) 를 제외한 모든 오퍼레이션은 RCU read-side 크리티컬 섹션 내에서 사용 가능해야 합니다. 구체적으로는, I/O 와 같이 취소 불가능한 오퍼레이션들이 사용 가능해야 합니다.
7. RCU 로 보호되는 데이터 구조체는 RCU read-side 크리티컬 섹션에서 실행 중일 때에도 업데이트 할 수 있어야 합니다.
8. RCU read-side 기능들도 update-side 기능들도 메모리 할당자 설계와 구현과는 독립적이어야 하는데, 달리 말해서, 같은 RCU 구현은 특정 데이터 구조에 대해 그 데이터 원소들이 어떻게 할당되고 해제되는가에 관계없이 그 데이터 구조를 보호할 수 있어야 합니다.
9. RCU grace period 는 RCU read-side 크리티컬 섹션들 바깥에서 종료되는 쓰레드들에 의해 블록되지 않아야 합니다. (하지만 대부분의 quiescent-state 기반의 구현은 이 필요성을 위반합니다.)

Quick Quiz 9.75: Grace period 가 RCU read-side 크리티컬 섹션들로 통제된다면, RCU로 보호되는 데이터 구조체는 어떻게 RCU read-side 크리티컬 섹션 내에서 업데이트 될 수 있을까요? ■

9.5.6 RCU Exercises

이 섹션은 RCU 를 이 책에서 앞서 보인 여러 예제들에 적용하는 내용에 대한 여러개의 Quick Quizz 들로 구성되어 있습니다. 각 Quick Quiz 에의 답은 힌트를 일부 제공하고, 또한 그 해법이 자세히 설명되어 있는 뒤의 섹션으로의 포인터를 담고 있습니다. `rcu_read_lock()`, `rcu_read_unlock()`, `rcu_dereference()`, `rcu_assign_pointer()`, 그리고 `synchronize_rcu()` 기능들만으로도 이 문제들에 충분해야 할 겁니다.

Quick Quiz 9.76: Figure 5.9 (`count_end.c`) 에 보인 통계적 카운터의 구현은 `read_count()` 안에서 합계를 구하는 것을 보호하기 위해 글로벌 락을 사용했는데 이는 부족한 성능과 음의 확장성을 일으켰습니다. `read_count()` 가 훌륭한 성능과 좋은 확장성을 제공할 수 있게 하기 위해 RCU 를 어떻게 적용해 볼 수 있을까요. (`read_count()` 의 확장성은 모든 쓰레드의 카운터들을 스캔해야 한다는 필요성으로 인해 제한되어진다는 점을 명심하세요.) ■

Quick Quiz 9.77: Section 5.5 는 디바이스들을 제거하기 위해 I/O 액세스들을 카운트하는 일을 처리하는 한쌍의 코드 조각들을 보였습니다. 이 코드 조각들은 reader-writer 락을 잡아야 하는 이유로 (I/O를 시작하는) 빠른 수행 경로의 높은 오버헤드 문제를 겪었습니다. 여기에 훌륭한 성능과 확장성을 가져오기 위해 RCU 를 어떻게 사용할 수 있을까요? (I/O 액세스를 하는 일반적인 경우의 첫번째 코드 조각의 성능이 디바이스 제거 코드 조각의 것보다 훨씬 더 중요함을 명심하세요.) ■

9.6 Which to Choose?

Table 9.5 는 이 챕터에서 소개한 네개의 미뤄두고 처리하기 테크닉들 중 무엇을 선택해야 할지를 돋는 대략적 경험적 법칙을 제공합니다.

“Existence Guarantee” 열에서 보인 것처럼, 링크된 데이터 원소들에 대한 존재 보장이 필요하다면 레퍼런스 카운팅, 해저드 포인터, 또는 RCU 를 사용해야 합니다. 시퀀스 락은 존재 보장을 제공하지 않고, 업데이트의 발견과 업데이트를 마주한 read-side 크리티컬 섹션의 재시도 기능을 제공합니다.

물론, “Updates and Readers Progress Concurrently” 열에서 보인 것처럼, 이런 업데이트의 발견은 시퀀스

	Reference Counting	Hazard Pointers	Sequence Locks	RCU
Existence Guarantees	Complex	Yes	No	Yes
Updates and Readers Progress Concurrently	Yes	Yes	No	Yes
Contention Among Readers	High	None	None	None
Reader Per-Critical-Section Overhead	N/A	N/A	Two <code>smp_mb()</code>	Ranges from none to two <code>smp_mb()</code>
Reader Per-Object Traversal Overhead	Read-modify-write atomic operations, memory-barrier instructions, and cache misses	<code>smp_mb()</code>	None, but unsafe	None (volatile accesses)
Reader Forward Progress Guarantee	Lock free	Lock free	Blocking	Bounded wait free
Reader Reference Acquisition	Can fail (conditional)	Can fail (conditional)	Unsafe	Cannot fail (unconditional)
Memory Footprint	Bounded	Bounded	Bounded	Unbounded
Reclamation Forward Progress	Lock free	Lock free	N/A	Blocking
Automatic Reclamation	Yes	No	N/A	No
Lines of Code	94	79	79	73

Table 9.5: Which Deferred Technique to Choose?

락킹이 업데이트 쓰레드와 읽기 쓰레드가 동시에 진행 할 수는 없게 만듭니다. 무엇보다, 그런 진행을 방지하는 것은 시퀀스 락킹을 사용하는 첫번째 이유입니다! 이런 상황은 존재 보장과 업데이트 발견을 제공하기 위해서는 시퀀스 락킹을 레퍼런스 카운팅, 해저드 포인터, 또는 RCU 와 함께 사용해야 함을 가리킵니다. 실제로, 리눅스 커널은 경로 탐색을 할 때에 RCU 와 시퀀스 락킹을 함께 사용합니다.

“Contention Among Readers”, “Reader Per-Critical-Section Overhead”, 그리고 “Reader Per-Object Traversal Overhead” 열은 이 테크닉들의 대략적인 read-side 오버헤드를 보입니다. 레퍼런스 카운팅의 오버헤드는 읽기 쓰레드간의 완전히 순서 맞춰진 read-modify-write 어토믹 오퍼레이션이 각각의 모든 오브젝트 방문에 필요하기 때문에 상당히 클 수 있습니다. 해저드 포인터는 마주치는 각각의 데이터 원소에 메모리 배리어 오버헤드를 끼치고 시퀀스 락은 크리티컬 섹션을 실행하고자 하는 각 시도마다 두개의 메모리 배리어 오버헤드를 만듭니다. RCU 구현의 오버헤드는 아예 없는 경우부터 각각의 read-side 크리티컬 섹션에서의 한쌍의 메모리 배리어까지 다양하고, 따라서 RCU 는 최고의 성능을 제공하는데, 많은 데이터 원소들을 마주치게 되는 read-side 크리티컬 섹션들에 대해서는 특히 그렇습니다.

“Reader Forward Progress Guarantee” 열은 RCU 만이 bounded wait-free forward-progress 보장을 가짐을 보이는데, 이는 유한한 갯수의 인스트럭션들을 수행하는 것으로 유한한 방문을 할 수 있음을 의미합니다.

“Reader Reference Acquisition” 열은 RCU 만이 무조

건적으로 레퍼런스를 얻어오는 것이 가능함을 알립니다. 시퀀스 락의 항목은 “Unsafe” 라 표기되어 있는데, 다시 말하지만 시퀀스 락은 레퍼런스를 얻는게 아니라 업데이트를 발견하기 때문입니다.

레퍼런스 카운팅과 해저드 포인터 둘 다 특정 레퍼런스 획득에 실패했을 때에는 획득을 처음부터 다시 시작할 것이 요구됩니다. 이를 자세히 보기 위해, 오브젝트 A, B, C, 그리고 D 가 순서대로 담겨 있는 링크드리스트에 다음과 같은 이벤트들이 발생한다고 생각해 봅시다:

1. 한 읽기 쓰레드가 오브젝트 B 로의 레퍼런스를 얻습니다.
2. 업데이트 쓰레드가 오브젝트 B 를 제거하지만, 읽기 쓰레드가 레퍼런스를 잡고 있으므로 해제하지는 않습니다. 이 리스트는 이제 오브젝트 A, C, 그리고 D 를 가지고 있고, 오브젝트 B 의 `->next` 포인터는 `HAZPTR_POISON` 으로 설정되어 있습니다.
3. 앞의 업데이트 쓰레드는 오브젝트 C 를 제거하고, 이에 의해 리스트는 오브젝트 A 와 D 만 가지고 있게 됩니다. 오브젝트 C 에는 레퍼런스가 잡혀 있지 않으므로, 곧바로 해제됩니다.
4. 앞의 읽기 쓰레드는 이제는 삭제된 오브젝트 B 의 다음 오브젝트로 넘어가려 하지만, poison 값을 가지고 있는 `->next` 포인터는 이를 못하게 합니다. 이는 좋은 일인데, 이렇게 되지 않으면 오브젝트 B

의 `->next` 포인터는 이미 해제된 메모리 영역을 가리킬 수도 있기 때문입니다.

5. 이 읽기 쓰레드는 따라서 리스트의 헤드부터 횡단을 다시 시작해야 합니다.

따라서, 레퍼런스를 획득하는데 실패했다면, 해저드 포인터나 레퍼런스 카운터를 사용하는 횡단은 처음부터 그 횡단을 재시작해야 합니다. 예를 들어 링크드 리스트를 담고 있는 트리와 같이 중첩된 링크드 데이터 구조의 경우 이 횡단은 가장 바깥의 데이터 구조로부터 재시작되어야 합니다. 이런 상황은 RCU에 훨씬 사용하기 쉬운 이점을 가져다 줍니다.

하지만, RCU의 사용성 이득은 공짜로 오는 것은 아닙니다. “Memory Footprint” 열에서 이를 볼 수 있습니다. RCU의 무조건적 레퍼런스 획득 지원은 곧 어떤 RCU 읽기 쓰레드에게 보이는 오브젝트는 그 읽기 쓰레드가 완료되기 전까지는 해제시킬 수가 없음을 의미합니다. 따라서 RCU는 무한한 메모리 사용량 가능성을 갖는 데, 업데이트가 인공적으로 조절되지 않는 한은 그렇습니다. 반면에, 레퍼런스 카운팅과 해저드 포인터는 정말로 동시의 읽기 쓰레드들이 레퍼런스하고 있는 데이터 원소들만을 유지할 겁니다.

이런 메모리 사용량과 획득 실패 사이의 미묘한 긴장감은 리눅스 커널 안에서는 일부 경우 RCU와 레퍼런스 카운터를 함께 사용하는 것으로 해결되기도 합니다. RCU는 잠깐 사용되는 레퍼런스들에 사용되는데, 이는 RCU read-side 크리티컬 섹션들이 짧을 수 있음을 의미합니다. 이런 짧은 RCU read-side 크리티컬 섹션들은 곧 연관된 RCU grace period를 역시 짧을 수 있어서, 메모리 사용량을 제한할 수 있음을 의미합니다. 긴 시간 사용될 수 있는 레퍼런스를 필요로 하는 일부 데이터 원소들을 위해서는 레퍼런스 카운팅이 사용됩니다. 이 말이 의미하는 바는 레퍼런스 획득 실패의 복잡도를 처리하는 건 그런 일부 데이터 원소들에서만 필요시 된다는 뜻입니다: 대량의 레퍼런스 획득은 RCU 덕분에 고려되지 않습니다. 레퍼런스 카운팅을 다른 동기화 메커니즘과 결합하는 방법에 대한 더 많은 정보를 위해선 Section 13.2을 보시기 바랍니다.

“Reclamation Forward Progress” 열은 해저드 포인터가 non-blocking 업데이트를 제공할 수 있음을 이야기합니다 [Mic04, HLM02]. 레퍼런스 카운팅은 구현에 따라서 그럴 수도 그려지 않을 수도 있습니다. 하지만, 시퀀스 락킹은 update-side 락 때문에 non-blocking 업데이트를 제공할 수 없습니다. RCU 업데이트 쓰레드들은 읽기 쓰레드를 기다려야만 하는데, 이 역시 non-blocking 업데이트의 규칙을 완전히 벗어납니다. 하지만, 블록킹 오퍼레이션은 메모리를 해제하기 위한 기다림 뿐인 상황이 존재하는데, 많은 경우에 이런 상황은 non-blocking 만큼이나 좋은 상황입니다 [DMS⁺12].

“Automatic Reclamation” 열에 보여진 것처럼, 레퍼런스 카운팅만이 메모리 해제를 자동화 할 수 있는데, non-cyclic 데이터 구조들에서만 그렇습니다.

마지막으로, “Lines of Code” 열은 Pre-BSD 라우팅 테이블 구현의 크기를 보이는데, 상대적인 사용의 편리성에 대한 대략적 정보를 제공합니다. 그렇다면 하나, 레퍼런스 카운팅과 시퀀스 락킹 구현은 버그가 존재하며, 정확히 동작하는 레퍼런스 카운팅 구현은 더 복잡할 것으로 여겨짐 [Val95, MS95]을 알아둘 필요가 있습니다. 그런 부분을 위해, 올바른 시퀀스 락킹 구현은 추가적인 또 다른 동기화 메커니즘을 필요로 하는데, 예를 들어 해저드 포인터나 RCU를 사용할 수 있는데, 시퀀스 락킹은 동시에 업데이트를 발견하고 다른 메커니즘은 안전한 레퍼런스 획득을 제공할 수 있을 겁니다.

이런 테크닉들을 조합해서 또는 각각 사용하는 경험이 더 쌓여감에 따라서 이 섹션에 놓인 경험적 법칙은 수정될 수 있을 겁니다. 하지만, 이 섹션은 현재로써는 최선의 결과를 반영하고 있습니다.

9.7 What About Updates?

이 챕터에서 이야기된 미뤄두고 처리하기 테크닉들은 대부분 읽기가 대부분인 환경에는 곧바로 적용할 수 있는데, 이는 곧 “그렇지만 업데이트는 어떻게 하지?”라는 질문을 갖게 합니다. 무엇보다, 읽기 쓰레드들의 성능과 확장성을 증가시키는 것은 잘 되었지만, 쓰기 쓰레드들에도 훌륭한 성능과 확장성을 원하는 건 자연스러운 일입니다.

이미 읽기 쓰레드들에게 높은 성능과 확장성을 가져다주는 상황을 이미 봤는데, Chapter 5에서 이야기된 카운팅 알고리즘들입니다. 이 알고리즘들은 부분적으로 분할된 데이터 구조들을 사용해서 업데이트들이 지역적으로 일어날 수 있게 하면서도 더 비싼 읽기들은 전체 데이터 구조들을 가로질러 더하기를 해야만 하게 했습니다. Silas Boyd-Wickhize는 이런 방향을 OpLog를 만들도록 일반화 시켰는데, 그는 이를 리눅스 커널의 경로 탐색, VM 역 매핑, 그리고 `stat()` 시스템콜에 적용시켰습니다 [BW14].

“Disruptor”라고 불리는 또 다른 방법은 입력 데이터의 커다란 스트림을 처리하는 어플리케이션을 위해 설계되었습니다. 이 방법은 single-producer-single-consumer FIFO 큐를 위한 것으로, 동기화의 필요성을 최소화 시킵니다 [Sut13]. 자바 어플리케이션들에서는 Disruptor는 또한 가비지 컬렉터의 사용을 최소화 시키기도 합니다.

그리고 물론, 가능하다면, 완전히 분할되었거나 “파편화된 (sharded)” 시스템들은 Chapter 6에서 이야기한 것처럼 굉장한 성능과 확장성을 제공합니다.

다음 챕터는 여러 종류의 데이터 구조의 맥락에서 업데이트들을 살펴보겠습니다.

Chapter 10

Data Structures

데이터로의 효율적인 접근은 중요해서 알고리즘에 대한 논의는 관련된 데이터 구조의 시간 복잡도를 포함합니다 [CLRS01]. 하지만, 병렬 프로그램에서는 시간 복잡도의 측정은 동시성에의 영향을 포함해야 합니다. 이런 영향은 Chapter 3에서 보인 것처럼 지배적일 정도로 클 수 있는데, 이는 동시적인 데이터 구조의 설계는 순차적 시간 복잡도에 그러한 것만큼 동시성에도 신경을 써야 합니다. 달리 말하자면, 좋은 병렬 프로그래머가 데이터 구조 관계에서 걱정해야 할 중요한 부분 하나는 동시성에 관련된 부분입니다.

Section 10.1은 이 챕터에서 소개되는 데이터 구조들을 평가하는데 사용될, 모티베이션을 줄 어플리케이션을 보입니다.

Chapter 6에서 논의된 대로, 높은 확장성을 얻는 방법은 파티셔닝입니다. 이는 분할 가능한 데이터 구조를 위한 방법을 이야기하게 되는데, 이 주제는 Section 10.2에서 다룹니다. Chapter 9는 일부 동작들을 미뤄두는게 어떻게 성능과 확장성을 모두 크게 개선할 수 있는지 설명했습니다. 특히 Section 9.5에서는 성능과 확장성을 쫓음에 있어서 미뤄두기가 어떻게 대단한 효과를 보이는지를 보였는데, 이 주제는 Section 10.3에서 다릅니다.

모든 데이터 구조가 분할 가능하지는 않습니다. Section 10.4는 다소 파티셔닝이 불가능한 데이터 구조를 알아봅니다. 이 섹션은 이걸 어떻게 읽기가 대부분이고 파티셔닝 가능한 영역으로 쪼개서 빼르고 확장성 있는 구현을 가능하게 하는지 보입니다.

이 챕터는 사용되어온 모든 동시적 데이터 구조의 자세한 부분들까지 이야기할 수는 없으므로 Section 10.5에서는 대부분의 유명하고 중요한 것들에 대한 간략한 조사 내용을 제공합니다. 최고의 성능과 확장성은 만들어진 내용에서의 세세한 최적화보다는 설계를 만들어내게 되긴 합니다만, 분명한 최고의 달성을 가능한 성능과 확장성을 얻는데에 세세한 최적화가 중요한 위치를 차지하는건 분명합니다. 따라서 이 주제를 Section 10.6에서 다루도록 합니다.

Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

Linus Torvalds

마지막으로, Section 10.7에서는 이 챕터의 요약을 제공합니다.

10.1 Motivating Application

성능을 평가하는데에는 Schrödinger의 Zoo 어플리케이션을 사용하도록 하겠습니다 [McK13]. Schrödinger는 많은 수의 동물들을 가지고 있는 동물원을 하나 가지고 있고, 그는 이 동물들을 in-memory 데이터베이스를 사용해 관리하려 할텐데, 각각의 동물원의 동물은 이 데이터베이스에 데이터 항목으로 표현됩니다. 각 동물은 키로 사용되는 유일한 이름을 가지고, 각 동물마다 관리되는 다양한 데이터를 갖습니다.

태어나는 것, 포획, 그리고 구매는 데이터 삽입이 되고, 사망, 방출, 그리고 판매는 삭제가 됩니다. Schrödinger의 동물원은 쥐와 곤충 등을 포함해 많은 수의 단명하는 동물들을 가지고 있으므로 이 데이터베이스는 높은 비율의 업데이트를 지원해야 합니다.

Schrödinger의 동물들에 관심 있는 사람들은 그것들에 대해 궤리를 던질 수 있습니다만, Schrödinger는 그의 고양이에의 굉장히 높은 비율의 궤리가 존재함을 알렸고, 따라서 그는 그의 쥐들은 이 데이터베이스를 죽을 때에나 사용하게 될거라 생각합니다. 이 말은 Schrödinger의 어플리케이션은 하나의 데이터 항목으로의 높은 비율의 궤리를 지원할 수 있어야 한다는 의미입니다.

다양한 데이터 구조들이 소개되는 중에도 이 어플리케이션을 마음속에 담아 두시기 바랍니다.

10.2 Partitionable Data Structures

오늘날 사용되는 데이터 구조들은 굉장히 많아서, 그것들을 다루는 교재들도 여럿 있습니다. 이 작은 섹션은 하나의 데이터 구조인 해시 테이블에 집중해 봅니다. 이렇게 집중을 해보는 것은 어떻게 동시성이 데이터 구조

```

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 hashtable {
12     unsigned long ht_nbuckets;
13     struct ht_bucket ht_bkt[0];
14 };

```

Figure 10.1: Hash-Table Data Structures

와 상호작용하는지에 대한 깊은 이해를 가능하게 하고, 또한 실제 상황에서 굉장히 많이 사용되는 데이터 구조에 집중할 수 있게 합니다. Section 10.2.1 에서는 이 디자인을 전체적으로 보고, Section 10.2.2 에서는 그 구현을 제공합니다. 마지막으로, Section 10.2.3 에서는 그 결과로 나오는 성능과 확장성을 알아봅니다.

10.2.1 Hash-Table Design

Chapter 6 에서는 쓸만한 성능과 확장성을 얻기 위해서는 파티셔닝을 적용해야 함을 알아보았고, 따라서 파티셔닝 적용 가능성은 데이터 구조를 선택할 때 첫번째로 고려할 기준이어야만 할 겁니다. 이 기준은 병렬성을 위해 많이 사용되는 해시 테이블에서는 잘 만족될 겁니다. 해시 테이블은 개념적으로 간단하고, *hash bucket* 들의 배열로 구성됩니다. 하나의 *hash function* 은 주어진 원소의 *key*로부터 이 원소가 저장되게 될 *hash bucket* 으로의 매팅을 담당합니다. 각각의 *hash bucket* 은 따라서 원소들의 링크드 리스트를 갖게되는데, 이는 *hash chain* 이라 불립니다. 제대로 구성된다면, 이런 *hash chain* 들은 상당히 짧을 것이어서 해시 테이블이 주어진 키를 가지고 해당 원소에 접근하는 것은 매우 효율적이게 될 겁니다.

Quick Quiz 10.1: 하지만 많은 종류의 해시 테이블들이 존재하고, 여기 설명된 체인 사용 (chained) 해시 테이블은 그 중 하나의 종류일 뿐입니다. 왜 체인 사용 해시 테이블에 집중하는 걸까요? ■

또한, 각각의 *bucket* 은 각자의 락을 가질 수도 있어서, 해시 테이블의 서로 다른 *bucket* 의 원소들은 완전히 독립적으로 더해지고 삭제되고 탐색될 수 있습니다. 많은 수의 원소들을 담고 있는 커다란 해시 테이블은 따라서 훌륭한 확장성을 제공합니다.

10.2.2 Hash-Table Implementation

Figure 10.1 (*hash_bkt.c*) 는 *chaing* 과 *bucket* 별 락킹을 사용하는 간단한 고정 크기 해시 테이블에 사용되는

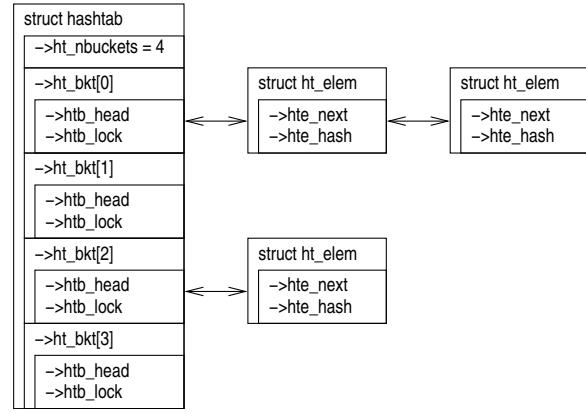


Figure 10.2: Hash-Table Data-Structure Diagram

```

1 #define HASH2BKT(htp, h) \
2     (&(htp)->ht_bkt[h % (htp)->ht_nbuckets])
3
4 static void hashtable_lock(struct hashtable *htp,
5                             unsigned long hash)
6 {
7     spin_lock(&HASH2BKT(htp, hash)->htb_lock);
8 }
9
10 static void hashtable_unlock(struct hashtable *htp,
11                             unsigned long hash)
12 {
13     spin_unlock(&HASH2BKT(htp, hash)->htb_lock);
14 }

```

Figure 10.3: Hash-Table Mapping and Locking

데이터 구조들을 보이고 있고, Figure 10.2 는 그것들이 어떻게 함께 동작하는지 보이고 있습니다. (Figure 10.1 의 line 11-14 의) *hashtable* 구조체는 (Figure 10.1 의 line 6-9 의) *ht_bucket* 구조체들을 네개 가지고 있으며, *->ht_nbuckets* 필드로 이 *bucket* 들의 갯수를 조절합니다. 그런 각각의 *bucket* 은 리스트 헤더 *->htb_head* 와 락 *->htb_lock* 을 갖습니다. 이 리스트 헤더들은 (Figure 10.1 의 line 1-4 의) *->ht_elem* 구조체들을 *->hte_next* 필드를 이용해 연결하고, 각각의 *ht_elem* 구조체는 또한 연관된 원소의 해시 값을 *->hte_hash* 필드에 저장해 둡니다. *ht_elem* 구조체는 이 해시 테이블에 위치한 더 큰 구조체의 안에 포함될 수도 있는데, 이런 경우의 이 큰 구조체는 복잡한 키를 가질 겁니다.

Figure 10.3에 보인 다이어그램에서 *bucket 0* 는 두개의 원소를 가지고 있고 *bucket 2* 는 하나의 원소를 가지고 있습니다.

Figure 10.3는 매팅과 락킹 함수들을 보입니다. Line 1 과 2 는 *HASH2BKT()* 매크로를 보이는데, 이 매크로는 해시 값으로부터 연관된 *ht_bucket* 구조체로의 매팅을 합니다. 이 매크로는 간단한 모듈로 연산을 사용합니

```

1 struct ht_elem *
2 hashtab_lookup(struct hashtab *http,
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(http, 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

다: 더 적극적인 해성이 필요하다면, 이 함수의 호출자는 키로부터 해시 값을 매핑할 때 이를 구현해야 합니다. 남아있는 두 함수들은 특정 해시 값에 연관된 \rightarrow htb_lock 을 각각 획득하고 해제합니다.

Figure 10.4 는 hashtab_lookup() 을 보이는데, 이 함수는 특정 해시 값과 키를 가지고 있는 원소가 존재한다면 그 원소로의 포인터를, 그렇지 않다면 NULL 을 리턴합니다. 이 함수는 해시 값과 그 키로의 포인터를 받는데 이는 이 함수의 사용자들이 임의의 키와 임의의 해시 함수를 사용할 수 있게 하며, qsort() 에서와 비슷하게 cmp() 를 통해 넘겨지는 키 비교 함수를 사용할 수 있게 하기 때문입니다. Line 11 은 해시 값으로부터 그에 연관된 bucket 으로의 포인터를 매핑해 줍니다. Line 12-19 의 루프를 통한 각각의 수행은 bucket 의 해시 체인의 원소 하나를 조사합니다. Line 15 에서는 해시 값이 들어맞는지 보고, 그렇지 않다면 line 16 에서 다음 원소로 넘어갑니다. Line 17 에서는 실제 키가 맞는지 확인해보고, 그렇다면 line 18 에서 그 원소로의 포인터를 리턴합니다. 어떤 원소도 들어맞지 않는다면, line 20 에서 NULL 을 리턴합니다.

Quick Quiz 10.2: 하지만 Figure 10.4 의 line 15-18 에서의 두번의 비교는 키가 unsigned long 에 들어맞는 경우라면 비효율적이지 않나요? ■

Figure 10.5 는 hashtab_add() 와 hashtab_del() 함수들을 보이는데, 이 함수들은 각각 해시 테이블로부터 원소를 더하고 삭제합니다.

hashtab_add() 함수는 단순히 원소의 해시 값을 line 6 에서 설정하고 line 7 과 8 에서 연관된 bucket 에 이 원소를 집어넣습니다. hashtab_del() 함수는 hash chain 리스트의 이중 연결의 원리 덕에 단순히 무엇이든 원소가 속해 있는 hash chain 으로부터 자신을 제거합니다. 이 두 함수들을 수행하기 전에, 호출자는 다

```

1 void
2 hashtab_add(struct hashtab *http,
3              unsigned long hash,
4              struct ht_elem *htep)
5 {
6     htep->hte_hash = hash;
7     cds_list_add(&htep->hte_next,
8                  &HASH2BKT(http, 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

```

1 struct hashtab *
2 hashtab_alloc(unsigned long nbuckets)
3 {
4     struct hashtab *http;
5     int i;
6
7     http = malloc(sizeof(*http) +
8                   nbuckets *
9                   sizeof(struct ht_bucket));
10    if (http == NULL)
11        return NULL;
12    http->htb_nbuckets = nbuckets;
13    for (i = 0; i < nbuckets; i++) {
14        CDS_INIT_LIST_HEAD(&http->ht_bkt[i].htb_head);
15        spin_lock_init(&http->ht_bkt[i].htb_lock);
16    }
17    return http;
18 }
19
20 void hashtab_free(struct hashtab *http)
21 {
22     free(http);
23 }

```

Figure 10.6: Hash-Table Allocation and Free

른 어떤 쓰레드도 같은 bucket 을 접근하거나 수정하고 있지 않음을 분명히 해야 하는데, 예를 들어 hashtab_lock() 을 미리 호출할 수 있을 겁니다.

Figure 10.6 는 hashtab_alloc() 과 hashtab_free() 를 보이는데, 이 함수들은 각각 해시 테이블 할당과 해제를 합니다. 할당은 line 7-9 에서의 필요한 메모리 할당으로 시작합니다. Line 10 에서 메모리가 무족하다는 걸 알게 된다면 line 11 에서 호출자에게 NULL 을 리턴합니다. 그렇지 않다면, line 12 에서는 bucket 들의 갯수를 초기화하고, line 13-16 의 루프는 bucket 각각을 초기화하는데, line 14 에서의 chain 리스트 헤더와 line 15 에서의 락 초기화를 포함합니다. 마지막으로, line 17 에서는 이 새로 할당된 해시 테이블로의 포인터를 리턴합니다. Line 20-23 에서의 hashtab_free() 함수는 단순합니다.

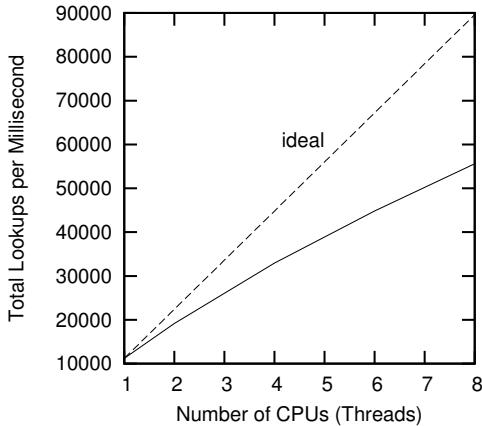


Figure 10.7: Read-Only Hash-Table Performance For Schrödinger's Zoo

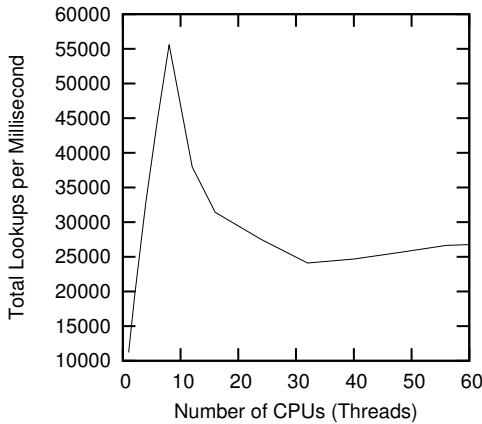


Figure 10.8: Read-Only Hash-Table Performance For Schrödinger's Zoo, 60 CPUs

10.2.3 Hash-Table Performance

여덟개의 2GHz Intel® Xeon® CPU 를 사용한 시스템에서 bucket 별 락을 사용하는 해시 테이블을 1024개의 bucket 을 사용한 성능 결과가 Figure 10.7 에 있습니다. 이 성능은 거의 선형적으로 증가합니다만, 8개의 CPU 만 사용함에도 이상적인 성능 기준의 절반을 넘지 못합니다. 이것은 락 획득과 해제가 하나의 CPU 에서는 캐시 미스를 내지 않지만, 두개 이상의 CPU 에서는 캐시 미스를 내기 때문입니다.

그리고 CPU 의 수가 커져갈수록 상황은 더 나빠져 가는데, 이를 Figure 10.8 가 보이고 있습니다. 여기선 이상적 성능을 위한 선을 추가적으로 봉리 필요도 없습니다: 9개 이상의 CPU 에서의 성능은 끔찍합니다. 이는 적당한 수의 CPU 이상으로 CPU 를 늘리는 것의 위험

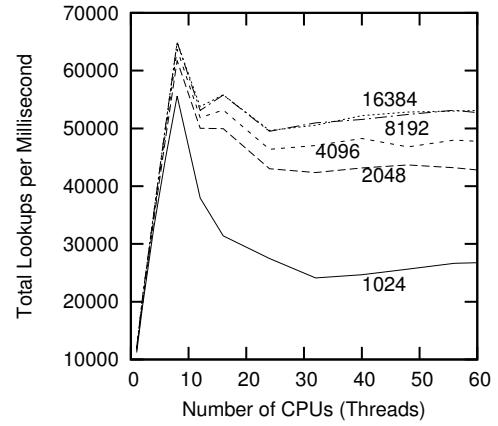


Figure 10.9: Read-Only Hash-Table Performance For Schrödinger's Zoo, Varying Buckets

성을 분명하게 강조합니다.

물론, 이렇게 성능이 떨어지는 것은 bucket 의 수가 부족해서일 수도 있습니다. 일단, 우리는 각 bucket 을 전체 cache line 에 맞아 떨어지도록 패딩을 넣지 않았고, 따라서 하나의 캐시 라인에 여러 bucket 들이 존재할 겁니다. 이로 인해 9개 CPU 에서부터 캐시 쓰래싱이 시작되었을 수 있습니다. 당연하게도 이는 bucket 의 수를 늘리는 것으로 테스트 해볼 수 있을 겁니다.

Quick Quiz 10.3: 단순히 bucket 들의 수를 늘리는 대신에 이미 있는 bucket 들을 캐시에 정렬 시키는게 더 낫지 않을까요? ■

하지만, Figure 10.9, 에서 볼 수 있듯이, bucket 들의 수를 늘리는 것이 실제로 성능을 어느정도 증가시키기는 하지만, 확장성은 여전히 끔찍합니다. 특히, 아홉개 이상의 CPU 들에서는 여전히 현격한 성능 저하를 볼 수 있습니다. 더 나아가서, 8,192 개 bucket 에서 16,384 개의 bucket 으로 넘어가는것부터는 더이상 성능이 늘어나지도 않습니다. 분명 뭔가가 잘못되어 있습니다.

문제는 이 시스템이 Table 10.1 에 보인 것처럼 CPU 0-7 과 32-39 는 첫번째 소켓에 위치하는 멀티 소켓 시스템이라는 점입니다. 따라서, 처음 여덟개 CPU 에 국한되어 돌아가는 테스트는 상당히 잘 동작하지만, 소켓 0 의 CPU 0-7 과 소켓 1 의 CPU 8 이 관여되는 테스트에서는 데이터를 소켓 너머로 주고받는 오버헤드가 생겨납니다. 이는 Section 3.2.1 에서 보인 것처럼, 성능을 상당히 저하시킬 수 있습니다. 짧게 말해서, 커다란 멀티 소켓 시스템에서는 완전한 파티셔닝에 더해서 메모리 참조의 높은 로컬리티를 필요로 합니다.

Quick Quiz 10.4: Schrödinger 의 동물원 어플리케이션의 소켓을 넘어가면서 보이는 음의 확장성을 가지고 생각해보면, 어플리케이션의 복사본을 여럿 만들어서 각각의 복사본이 전체 동물의 부분집합만을 가지고 하

Socket	Core							
0	0	1	2	3	4	5	6	7
	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

나의 소켓 위에서만 각자 돌도록 하는 건 어떨까요? ■
 지금까지 논의된 Schrödinger's-zoo 수행의 한가지 핵심적 특성은 이것들이 모두 읽기만 했다는 것입니다. 이는 락 희들이 초래한 캐시 미스들로 인해 발생한 성능 하락을 더욱 심하게 만듭니다. 이 아래에 깔려 있는 해시 테이블 자체에 대해 업데이트를 하지 않더라도, 메모리에 쓰기를 하기 위한 비용은 여전히 지불하고 있는 것입니다. 물론, 이 해시 테이블이 절대 업데이트 되지 않는다면, 상호 배타성을 완전히 제거할 수 있을 겁니다. 이 방법은 상당히 단순하고, 따라서 독자들의 연습으로 남겨두겠습니다. 하지만 결국은 일어나는 업데이트가 있다 하더라도, 쓰기를 막는 것은 캐시 미스를 막을 것이고, 이는 읽기가 대부분인 데이터의 복사본을 모든 캐시에 만들어두는 것으로 메모리 참조의 로컬리티를 높여줄 수 있을 것입니다.

따라서 다음 섹션에서는 읽기가 대부분이지만 업데이트가 가끔씩이지만 언제든 일어날 수 있는 경우를 위한 최적화들을 알아보겠습니다.

10.3 Read-Mostly Data Structures

파티셔닝이 적용된 데이터 구조가 훌륭한 확장성을 제공할 수 있긴 하지만, NUMA 효과는 성능과 확장성 모두를 심각하게 저하시킬 수 있습니다. 더불어, 읽기 쓰레드들이 쓰기 쓰레드들과 배타적으로 동작해야 한다는 점이 읽기가 대부분인 상황에서의 성능을 저하시킬 수 있습니다. 하지만, Section 9.5에서 소개된 바 있는 RCU를 사용해서 성능과 확장성을 모두 달성할 수 있습니다. 비슷한 결과를 해저드 포인터를 이용해서 얻을 수도 있는데(hazptr.c), 이는 이 섹션의 성능 결과에서 보여질 겁니다 [McK13].

10.3.1 RCU-Protected Hash Table Implementation

Bucket 별 락킹을 사용하는 RCU로 보호되는 해시 테이블을에서, 업데이트 쓰레드들은 Section 10.2에서 설명된 것과 똑같이 락킹을 사용하지만 읽기 쓰레드들

```

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 *hेतp,
6                             void *key))
7 {
8     struct ht_bucket *htb;
9     struct ht_elem *hेतp;
10
11    htb = HASH2BKT(htp, hash);
12    cds_list_for_each_entry_rcu(hेतp,
13                                &htb->htb_head,
14                                hte_next) {
15        if (hेतp->hte_hash != hash)
16            continue;
17        if (cmp(hेतp, key))
18            return hेतp;
19    }
20    return NULL;
21 }

```

Figure 10.11: RCU-Protected Hash-Table Lookup

은 RCU를 사용합니다. 데이터 구조는 Figure 10.1에서 보인 것과 똑같이 남아 있고, HASH2BKT(), hashtab_lock(), 그리고 hashtab_unlock() 함수들은 Figure 10.3에서 보인 것과 똑같습니다. 하지만, 읽기 쓰레드들은 Figure 10.10에 보인 것과 같이 by hashtab_lock_lookup() 와 hashtab_unlock_lookup()로 둘러싸인 더 가벼운 동시성 제어 방법을 사용할 겁니다.

Figure 10.11은 bucket 별 락킹을 사용하여 RCU로 보호되는 해시 테이블에 사용되는 hashtab_lookup() 함수를 보입니다. 이 함수는 cds_list_for_each_entry_rcu()가 cds_list_for_each_entry_rcu()로 바뀌었다는 점을 제외하고는 Figure 10.4에서 보인 버전과 동일합니다. 이것들 둘 다 htb->htb_head로 레퍼런스되는 해시 체인을 따라간다는 점에서는 동일합니다만 cds_list_for_each_entry_rcu()는 추가적으로, 동시의 삽입이 존재하는 경우에 올바른 메모리 액세스 순서를 강제합니다: 순수한 bucket 별 락킹 사용 구현과 달리, RCU로 보호되는 구현에서는 검색 작업이 삽입이나 삭제 작업과 동시에 수행되는 것이 허용되고, RCU를 신경쓰는 기능인 cds_list_for_

```

1 void
2 hashtable_add(struct hashtable *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 hashtable_del(struct ht_elem *htep)
12 {
13     cds_list_del_rcu(&htep->hte_next);
14 }

```

Figure 10.12: RCU-Protected Hash-Table Modification

each_entry_rcu() 와 같은 것들은 이 추가적 동시성을 올바로 제어할 것이 요구됩니다. 또한 hashtable_lookup() 의 호출자는 RCU read-side 크리티컬 섹션 안에 있어야 한다는 점을 알아둬야 하는데, 예를 들어 호출자는 hashtable_lookup() 을 호출하기 전에 hashtable_lock_lookup() 을 호출해야만 합니다(그리고 나중에는 당연히 hashtable_unlock_lookup() 을 호출해야겠죠).

Quick Quiz 10.5: 하지만 해시 테이블의 원소가 검색과 동시에 삭제될 수가 있다면, 그 말은 검색 기능은 검색된 직후에 삭제된 원소로의 레퍼런스를 리턴할 수도 있다는 의미 아닌가요? ■

Figure 10.12 는 hashtable_add() 와 hashtable_del() 함수를 보이는데, 둘 다 Figure 10.5 에서 보인 RCU 사용하지 않는 해시 테이블에서의 같은 기능을 하는 것들과 상당히 비슷합니다. hashtable_add() 함수는 검색되고 있는 도중에 동시에 해시 테이블에 더해지는 원소의 경우에 대해 올바른 순서를 강제하기 위해서 cds_list_add() 대신에 cds_list_add_rcu() 함수를 사용합니다. hashtable_del() 함수는 삭제되기 직전에 검색되는 원소의 경우를 위해서 cds_list_del_init() 대신에 cds_list_del_rcu() 함수를 사용합니다. cds_list_del_init() 와 달리, cds_list_del_rcu() 는 다음 원소로의 포인터를 온전하게 남겨두어서 hashtable_lookup() 이 삭제된 원소의 다음 원소로도 갈 수 있게 해줍니다.

물론, hashtable_del() 을 호출한 후에, 호출자는 이번에 삭제한 원소의 메모리를 해제하거나 재사용하기 전에 (synchronize_rcu() 를 호출하거나 하는 식으로) RCU grace period 를 하나 기다려야만 합니다.

10.3.2 RCU-Protected Hash Table Performance

Figure 10.13 는 RCU 로 보호되는 버전과 해저드 포인터로 보호되는 버전의 해시 테이블들의 읽기만 일어나는 상황에서의 성능을 앞 섹션에서 본 bucket 별 락킹 사

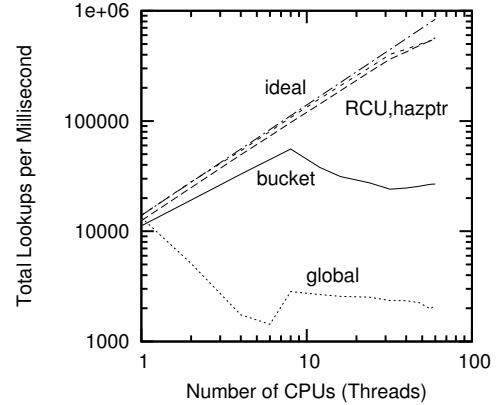


Figure 10.13: Read-Only RCU-Protected Hash-Table Performance For Schrödinger's Zoo

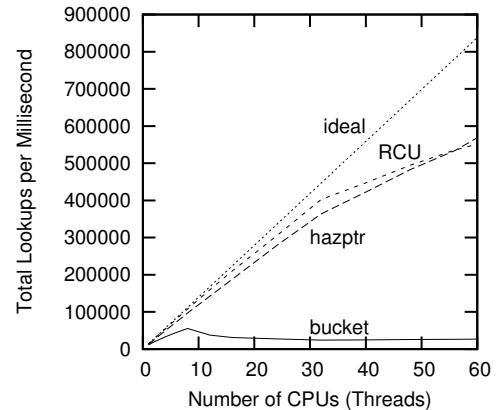


Figure 10.14: Read-Only RCU-Protected Hash-Table Performance For Schrödinger's Zoo, Linear Scale

용 구현 버전과 비교해 보이고 있습니다. 볼 수 있듯이, RCU 와 해저드 포인터 둘 다 큰 수의 쓰레드들에서는 NUMA 영향을 받기는 하지만 이상적인 경우에 가까운 성능과 확장성을 보이고 있습니다. 테이블 전체를 하나의 락으로 보호하는 구현의 결과도 보이고 있는데, 기대된 대로 그런 상황에서의 성능 결과는 bucket 별 락킹 구현 버전에 비해서 조차도 나쁨을 볼 수 있습니다. RCU 는 해저드 포인터보다 아주 약간 좋은 성능을 보이기는 하지만, 그 차이는 이 로그스케일 그림에서는 거의 보이지도 않을 정도입니다.

Figure 10.14 는 같은 데이터를 linear scale 로 보입니다. 여기서는 테이블 전체를 하나의 락으로 보호하는 구현의 경우는 x 축에 붙어버려 제대로 볼 수가 없지만, RCU 와 해저드 포인터 사이의 상대적 성능 차이를 좀 더 분간할 수 있습니다. 둘 다 32 CPU 를 넘어가면서 성

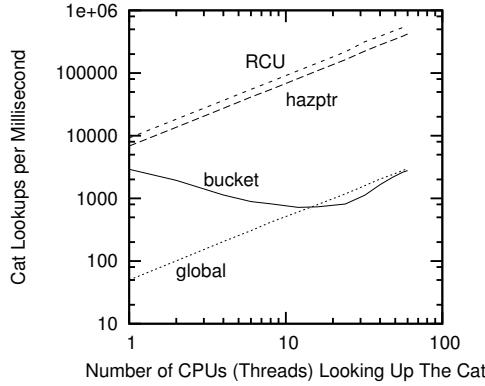


Figure 10.15: Read-Side Cat-Only RCU-Protected Hash-Table Performance For Schrödinger's Zoo at 60 CPUs

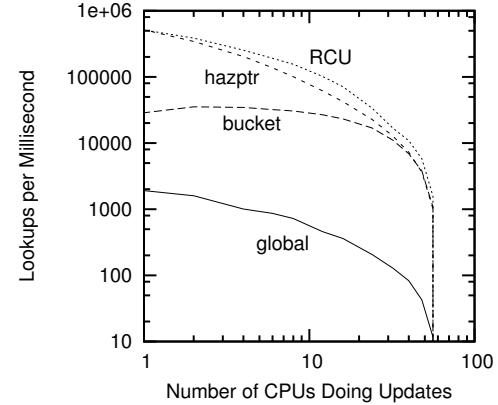


Figure 10.16: Read-Side RCU-Protected Hash-Table Performance For Schrödinger's Zoo at 60 CPUs

능 증가정도가 떨어지기 시작하는 걸 볼 수 있는데, 이는 하드웨어 멀티쓰레딩 때문입니다. 32개 이하의 CPU에서, 각각의 쓰레드는 각각의 코어를 갖고 있습니다. 이 수전에서, RCU 는 해저드 포인터보다 좋은 성능을 보이는 데 이는 해저드 포인터의 read-side 메모리 배리어가 해당 코어 내에서의 시간을 잡아먹기 때문입니다. 짧게 말해, RCU 는 하나의 하드웨어 쓰레드를 가지고 있을 때에는 해저드 포인터보다 코어를 더 잘 활용합니다.

이 상황은 32개보다 많은 CPU 들의 상황에서 변합니다. RCU 는 각 코어의 반이 넘는 리소스를 하나의 하드웨어 쓰레드에서 사용하므로, RCU 각 코어에 두개의 하드웨어 쓰레드가 존재하게 되면 그 성능 이득이 상대적으로 적어지게 됩니다. 해저드 포인터의 성능 그림에서의 경사도 역시 32 CPU 를 넘어서면서 줄어듭니다만, RCU 에 비해서는 좀 덜 극적으로 줄어들고 있는데, 이는 두번째 하드웨어 쓰레드는 첫번째 하드웨어 쓰레드가 메모리 배리어 대기시간으로 멈춰서 있는 사이의 시간을 활용할 수 있기 때문입니다. 뒤의 섹션에서 보게 되겠지만, 하드웨어 포인터의 두번째 하드웨어 쓰레드 사항에서의 이득은 워크로드의 특성에 종속적입니다.

앞서 언급했듯, Schrödinger 는 그의 고양이의 인기에 놀랐고 [Sch35], 이 인기도를 그의 설계에 반영해야 할 필요가 있음을 깨달았습니다. Figure 10.15 는 60-CPU 가 동작하는 상황에서 고양이를 검색하는 CPU 의 숫자를 바꿔가면서 성능 결과를 비교해 봅니다. RCU 와 해저드 포인터 모두 이 상황에 잘 동작합니다만, bucket 락킹은 오히려 성능이 떨어지는데, 결국에는 심지어 하나의 락을 사용하는 경우보다도 나쁜 결과를 보입니다. 만약 모든 CPU 가 고양이만을 검색하고 있다면 그 고양이의 bucket 에 연관된 락에 모두가 매이게 될테고 이는 하나의 락을 사용하는 것과 같으므로 그다지 놀라운 일도 아닙니다.

이 고양이만을 위한 벤치마크는 완전히 파티셔닝된 샤딩 방법에 존재할 수 있는 잠재적 문제를 보이고 있습니다. 이 고양이의 파티션에 연관된 CPU 만이 그 고양이에 접근할 수 있게 되어서, 고양이만 검색될 때의 성능을 제한하게 됩니다. 물론, 수많은 어플리케이션들은 로드 오퍼레이션을 다양하게 흘뿌리는 특성을 가지고 있고, 그런 어플리케이션들에서 샤딩 방법은 상당히 잘 동작할 것입니다. 하지만, 샤딩은 “핫 스팟” 을 아주 잘 처리하지는 못하고, Schrödiger 의 고양이 예제는 그런 하나의 케이스를 보이고 있습니다.

물론, 데이터를 읽기만 할 셈이라면, 애초에 동시성 제어가 필요하지도 않을 겁니다. Figure 10.16 는 따라서, 업데이트의 영향을 보입니다. 이 그래프의 왼쪽 끝에서는 60 개의 모든 CPU 들이 검색만을 하고, 오른쪽 끝에서는 모든 60개의 CPU 들이 업데이트만을 합니다. 네개의 구현 모두, 밀리세컨드당 검색의 수는 업데이트를 하는 CPU 들의 수가 늘어날수록 감소되고, 60개의 모든 CPU 들이 업데이트를 하게 될 때에는 밀리세컨드당 검색의 수는 0이 되어버립니다. RCU 는 해저드 포인터에 비해 상대적으로 좋은 결과를 보이는데 해저드 포인터의 read-side 메모리 배리어는 업데이트가 증가함에 따라 더 커다란 오버헤드를 내기 때문입니다. 따라서 최신의 하드웨어는 메모리 배리어 수행을 많이 최적화 해서 읽기만 할 때의 메모리 배리어 오버헤드를 많이 줄일 것으로 보입니다.

Figure 10.16 는 검색을 하는 와중에 업데이트의 비율에 따른 영향을 보았다면, Figure 10.17 는 업데이트 자신에 대해 증가하는 업데이트 비율의 영향을 보이고 있습니다. 해저드 포인터와 RCU 는 시작점부터 상당히 좋은 성능을 가지고 시작하는데, bucket 락킹과 달리, 읽기 쓰레드들은 업데이트 쓰레드들을 막지 않기 때문입니다. 하지만, 업데이트를 하는 CPU 의 수가 늘어남에

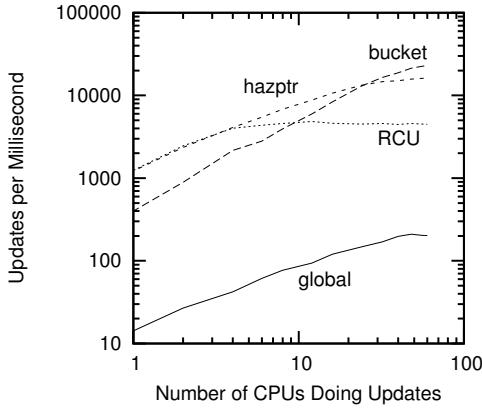


Figure 10.17: Update-Side RCU-Protected Hash-Table Performance For Schrödinger's Zoo at 60 CPUs

따라, 업데이트 쪽 오버헤드는 그 존재를 드러내기 시작해서, RCU 에서 먼저 드러내고, 뒤이어 해저드 포인터에서도 보입니다. 물론, 이 세개의 구현 모두 하나의 락을 사용하는 경우에 비해서는 훨씬 좋은 성능을 보입니다.

물론, 검색 성능에서의 차이는 업데이트 비율의 차이에 의해 영향을 받았을 가능성이 큽니다. 이를 확인할 수 있는 한가지 방법은 인공적으로 bucket 별 락킹과 해저드 포인터에서의 업데이트 비율을 RCU 에서의 그것에 맞도록 바꿔보는 것입니다. 그렇게 하는 것은 bucket 별 락킹에서의 검색 성능을 많이 끌어올리지는 못하고, 해저드 포인터와 RCU 사이의 차이를 줄이지도 못합니다. 하지만, 해저드 포인터의 읽기 쪽 메모리 배리어를 제거하는 것은(그렇게 해서 해저드 포인터의 불안전한 구현을 만드는 것은) 해저드 포인터와 RCU 사이의 성능 차이를 거의 없애버립니다. 비록 이 불안전한 해저드 포인터 구현은 벤치마킹 목적에 따라서는 사용될 수도 있겠지만, 제품 단계에서의 사용은 결코 추천되지 못할 것입니다.

Quick Quiz 10.6: 8개 CPU에서 60 CPU 상황을 쓰레드 수를 늘리는 것으로 시뮬레이션 해보는건 상당히 위험함이 Section 10.2.3에서 드러났었습니다. 하지만 60 CPU를 가지고 더 많은 CPU 상황을 시뮬레이션 해보는건 왜 안전할 수 있을까요? ■

10.3.3 RCU-Protected Hash Table Discussion

RCU 와 해저드 포인터를 사용한 구현들로부터 얻어지는 한가지 결론은 두개의 동시에 수행되는 읽기 쓰레드들은 한 고양이의 상태에 서로 다른 의견을 가질 수 있다는 겁니다. 예를 들어, 한 읽기 쓰레드는 고양이가



Figure 10.18: Even Veterinarians Disagree!

삭제되기 직전에 그 고양이의 데이터로의 포인터를 얻어오지만, 또다른 읽기 쓰레드는 똑같은 포인터를 삭제 직후에 얻어올 수도 있습니다. 첫번째 읽기 쓰레드는 고양이가 살아있다고 믿겠지만, 두번째 읽기 쓰레드는 고양이가 죽었다고 생각할 겁니다.

물론, 이 상황은 Schrödinger 의 고양이에 적합합니다만, 이는 평범한 양자가 아닌 고양이에도 상당히 합리적인 일임이 드러납니다.

동물이 정확히 언제 태어나고 죽었는지 판단하는건 불가능하기 때문입니다.

이를 분명히 하기 위해, 고양이의 죽음을 맥박으로 판단한다고 생각해 봅시다. 이는 죽음을 판단하기 위해선 마지막 맥박으로부터 얼마나 오래 기다려야 하는지에 대한 질문을 만듭니다. 딱 1 밀리세컨드만 기다리는건 분명 웃기는 짓인데, 그렇게 되면 건강하게 살아있는 고양이도 1초 동안에도 여러번 죽은 것으로 판단될 것이기 때문입니다—그리고선 부활하겠죠—. 한달을 기다리는 것도 똑같이 웃긴 짓일텐데, 그렇게 되면 가여운 고양이의 죽음이 냄새로도 분명히 알 수 있게 될테니까요.

동물의 심장은 수초동안 멈췄다가 다시 도착할 수 있으므로, 죽음의 인식과 잘못된 알람의 가능성 사이의 트레이드오프가 있습니다. 두명의 전문가도 마지막 맥박으로부터 얼마나 기다려야 죽음을 판단할 수 있는지에 대해선 의견이 갈릴 수 있습니다. 예를 들어, 한 전문가는 마지막 맥박으로부터 30초 후에 죽음을 판단할 수 있는 반면, 다른 전문가는 1분을 기다려서야 인식할 수 있습니다. 이런 경우, 두명의 전문가는 Figure 10.18에 그린 것처럼 마지막 맥박으로부터 30초가 지난 후 30초 동안은 그 고양이의 상태에 대해 이견을 가질 것입니다.

물론, Heisenberg 는 이런 종류의 불확실성과 함께 살아가야 한다고 가르쳤고 [Hei27], 이는 컴퓨팅 하드웨어와 소프트웨어 역시 비슷하게 동작하기 때문에 좋은

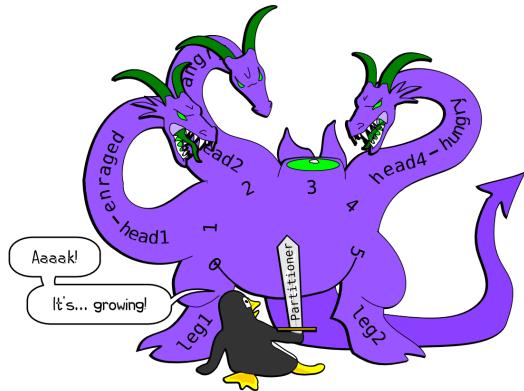


Figure 10.19: Partitioning Problems

일입니다. 예를 들어, 컴퓨팅 하드웨어의 일부 부분이 고장났는지 어떻게 아시나요? 종종 시간적으로 응답을 하지 않을 때가 있죠. 고양이의 맥박처럼, 이는 하드웨어가 고장난 건지 아닌지에 대한 불확실성의 기간을 만들어냅니다.

더 나아가서, 대부분의 컴퓨팅 시스템은 바깥 세상의 것들과 상호 동작하도록 만들어졌습니다. 따라서 이 바깥 세상의 것들과의 일관성은 무엇보다 중요합니다. 하지만, page 140 의 Figure 9.37에서 봤듯이 증가되는 내부의 일관성은 외부의 일관성의 비용을 비싸게 만들 수 있습니다. RCU 와 해저드 포인터와 같은 테크닉들은 향상된 외부 일관성을 얻기 위해 내부 일관성을 약간 포기합니다.

요약해서, 내부의 일관성은 모든 문제 상황에서 자연적인 부분은 아니고, 종종 성능, 확장성, 외부 일관성, 그리고 그런 것들 모두의 측면에서 커다란 비용을 만듭니다.

10.4 Non-Partitionable Data Structures

고정크기 해시 테이블은 파티셔닝을 완전히 적용 가능하지만, 크기 조정이 가능한 해시 테이블은 크기를 키우거나 줄일 때, Figure 10.19에 보인 것처럼 파티셔닝에 있어서의 문제를 갖습니다. 하지만, 다음 섹션에서 설명하듯이 고성능의 확장성 있는 RCU로 보호되는 해시 테이블을 만드는 것은 가능한 것으로 드러났습니다.

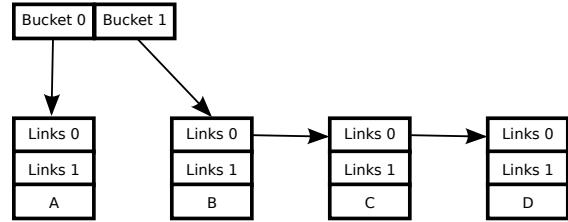


Figure 10.20: Growing a Double-List Hash Table, State (a)

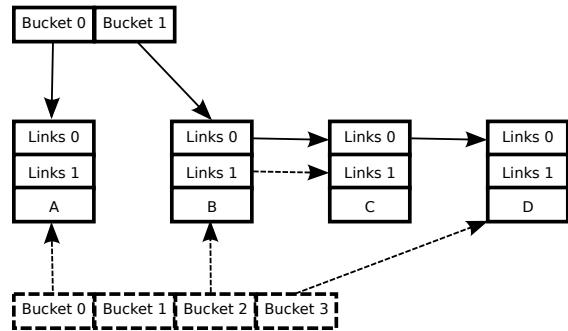


Figure 10.21: Growing a Double-List Hash Table, State (b)

10.4.1 Resizable Hash Table Design

행복하게도 2000년 초의 상황과 대조적으로, 이제는 확장성 있고 RCU로 보호되는 해시 테이블은 세개 이상의 종류가 있습니다. 첫번째의 (그리고 가장 간단한) 것은 Herbert Xu [Xu10]에 의해 리눅스 커널을 위해 개발된 것이고, 다음 섹션에서 설명됩니다. 다른 두 가지는 Section 10.4.4에서 간단히 다루겠습니다.

첫번째 해시테이블의 구현의 기저에 깔린 핵심은 각 데이터 원소가 두개의 리스트 포인터들을 가질 수 있는데, 하나는 현재 RCU 읽기 쓰레드들 (과 non-RCU 업데이트 쓰레드들)에 의해 사용되는 것이고 다른 하나는 새로운, 크기가 재조정된 해시 테이블을 구성하는데 사용된다는 것입니다. 이 방增资은 검색, 삽입, 그리고 삭제 작업이 크기 재조정 작업과 동시에 수행될 수 있게 합니다.

크기 조정 오퍼레이션은 Figure 10.20에 보인 최초의 두개 bucket 상태로부터 시작해서 Figures 10.20-10.23에 보인 것처럼 진행되는데, 시간의 흐름에 따라 그림에서 그림으로 이동합니다. 최초의 상태는 원소들을 bucket 안에 연결하기 위해 zero-index 링크를 사용합니다. 네개의 새로운 bucket 배열이 할당되고, one-index 링크들이 원소들을 이 네개의 새로운 bucket 안으로 연결

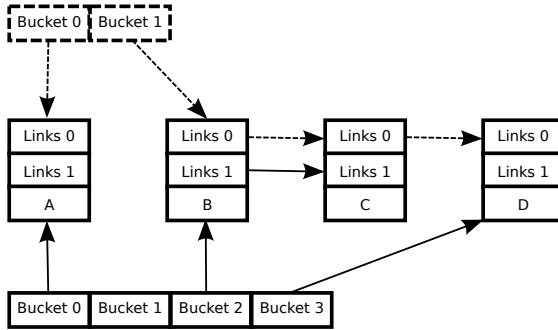


Figure 10.22: Growing a Double-List Hash Table, State (c)

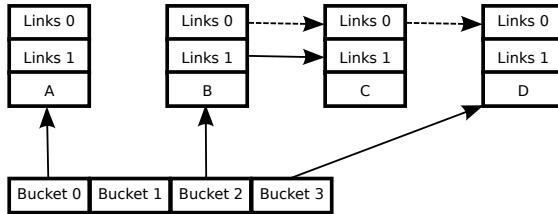


Figure 10.23: Growing a Double-List Hash Table, State (d)

되는데 사용됩니다. 이는 Figure 10.21에 보인 state (b) 상황을 만드는데, 읽기 쓰레드들은 여전히 원래의 두개 bucket 배열을 사용하고 있습니다.

새로운 네개의 bucket 배열은 읽기 쓰레드들에게 노출되고, 모든 읽기 쓰레드들을 기다리기 위해 grace-period를 기다리게 되어서 state (c)에 도달하게 되는데, Figure 10.22에 그 상태가 그려져 있습니다. 이 상태에서, 모든 읽기 쓰레드들은 새로운 네개 bucket 배열을 사용하게 되는데, 이 말은 기존의 두개 bucket 배열은 해제되어도 좋다는 뜻이므로 Figure 10.23에 보여진 state (d)로의 이전을 하게 됩니다.

이 디자인은 상대적으로 간단한 구현을 이끄는데, 이 구현에 대해서는 다음 섹션에서 다룹니다.

10.4.2 Resizable Hash Table Implementation

크기 재조절은 전통적인 우회로 추가 방법으로 이루어지는데, 여기서 이 우회로는 Figure 10.24의 line 12-25에 보이는 ht 구조체입니다. Line 27-30에 보인 hashtable 구조체는 현재의 ht 구조체로의 포인터와 해시 테이블 크기 재조정을 하려는 동시적 시도를 직렬화하는데 사용되는 스플릿만을 담고 있습니다. 전통적인

```

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_cur;
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 hashtable {
28     struct ht *ht_cur;
29     spinlock_t ht_lock;
30 };

```

Figure 10.24: Resizable Hash-Table Data Structures

lock 또는 아토믹 오퍼레이션 기반의 구현을 사용하려 했다면, 이 hashtable 구조체는 성능 측면에서도 확장 성 측면에서도 심각한 병목지점이 될 수 있었을 겁니다. 하지만, 크기 재조정 작업은 상대적으로 가끔씩만 이뤄지므로 RCU를 잘 사용해 볼 수 있습니다.

ht 구조체는 line 13의 `->ht_nbuckets` 필드를 통해 해시 테이블의 크기를 나타냅니다. 이 크기는 bucket 배열을 담고 있는 똑같은 구조체 (line 24의 `->ht_bkt[]`)에도 저장되는데 이 크기와 배열 사이의 미스 매치를 피하기 위해서입니다. Line 14에서의 `->ht_resize_cur` 필드는 현재 크기 재조정 작업이 진행중이지 않다면 -1이 되는데, 크기 조정 작업이 진행 중이라면 line 15의 `->ht_new` 필드로 레퍼런스 될, 새로운 해시 테이블에 추가될 원소의 bucket의 인덱스를 가리키게 됩니다. 진행 중인 크기 조정 작업이 없다면, `->ht_new`는 NULL입니다. 따라서, 크기 조정 작업은 새로운 ht 구조체를 메모리 할당하고 `->ht_new` 포인터로 그것을 레퍼런스 한 후, `->ht_resize_cur`를 기존 테이블의 bucket들을 거쳐 증가시켜갑니다. 새로운 테이블에 모든 원소가 추가되었다면, 새로운 테이블은 hashtable 구조체의 `->ht_cur` 필드로 연결됩니다. 기존의 읽기 쓰레드들이 모두 완료되면, 기존의 해시 테이블의 ht 구조체는 메모리에서 해제될 수 있을 겁니다.

Line 16에 있는 `->ht_idx` 필드는 이 두개의 리스트 포인터들 중 이 해시 테이블을 통해 무엇이 사용되고 있는가를 알리고, line 3의 ht_elem 구조체의 `->hte_`

```

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_csr) {
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

next [] 배열의 인덱스로 사용됩니다.

Line 17-23의 `->ht_hash_private`, `->ht_cmp()`, `->ht_gethash()`, 그리고 `->ht_getkey()` 필드는 원소별 키와 해시 함수를 정의합니다. `->ht_hash_private`은 해시 함수가 혼란스러울 수 있게 해서 [McK90a, McK90b, McK91] 해시 함수에 사용되는 패러미터들에 대한 통계적 추측을 기반으로 하는 denial-of-service 공격을 막는데 사용될 수 있게 합니다. `->ht_cmp()` 함수는 특정 키를 특정 원소와 비교하고, `ht_gethash()`는 특정 키의 해시를 계산하며, `->ht_getkey()`는 키를 감싸고 있는 데이터 원소로부터 키를 꺼낼 수 있게 합니다.

`ht_bucket` 구조체는 앞에서와 동일하고, `ht_elem` 구조체는 앞에서의 하나의 리스트 포인터와 달리 두개의 리스트 포인터의 원소 배열을 제공한다는 점만 다릅니다.

고정크기 해시 테이블에서의 `bucket` 선택은 상당히 간단합니다: 단순히 해시 값을 연관된 `bucket` 인덱스로 변환하면 됩니다. 반면에, 크기 조정이 가능하다면, 과거의 `bucket`과 새로운 `bucket` 중 무엇을 선택할지 결정해야 합니다. 만약 과거 테이블에서 선택될 `bucket`이 이미 새로운 테이블에 분산되었다면, 새로운 테이블에서의 `bucket`이 선택되어야 합니다. 거꾸로, 과거 테이블에서 선택될 `bucket`이 아직 분산되지 않았다면, 그 `bucket`은 과거 테이블에서 선택되어야 합니다.

`Bucket` 선택은 Figure 10.25에 `ht_get_bucket_single()`를 line 1-8에, 그리고 `ht_get_bucket()`를 line 10-24에 보이고 있습니다. `ht_get_bucket_single()` 함수는 특정 해시 테이블의 특정 키에 연관된 `bucket`으로의 레퍼런스를 리턴하는데, 이때에는 크

기 재조정을 고려하지 않습니다. 또한 해당 키에 연관된 해시 값을 line 5와 6에서 패러미터 `b`로 레퍼런스되는 영역에 저장합니다. 그리고 나서 line 7은 연관된 `bucket`으로의 레퍼런스를 리턴합니다.

`ht_get_bucket()` 함수는 해시 테이블 선택을 처리하는데, line 16에서 `ht_get_bucket_single()`을 호출해서 현재 해시 테이블에서 해시 값에 연관된 `bucket`을 선택하는데, 이 해시 값은 패러미터 `b`에 저장합니다. Line 17에서 이 테이블이 크기 재조정 중이고 line 16의 `bucket`이 이미 새로운 해시 테이블로 분산되었다는 점을 알게 되면, line 18에서 새 해시 테이블을 선택하고 line 19에서 이 새로운 해시 테이블에서의 해시 값에 연관된 `bucket`을 선택하고, 이때에도 해시 값은 패러미터 `b`에 저장합니다.

Quick Quiz 10.7: Figure 10.25의 코드는 해시 값을 두번 계산하네요! 왜 이렇게 비효율적일까요? ■

Line 21에서 패러미터 `i`가 NULL이 아니란 걸 알게 되면, line 22에서 선택된 해시 테이블로의 인덱스를 저장합니다. 마지막으로, line 23에서는 선택된 해시 `bucket`으로의 레퍼런스를 리턴합니다.

Quick Quiz 10.8: Figure 10.25의 코드는 선택된 `bucket`에서 진행중일 수도 있는 크기 재조정 프로세스로부터의 보호는 어떻게 하나요? ■

이와 같은 `ht_get_bucket_single()`와 `ht_get_bucket()`의 구현은 크기 재조정 작업과 동시에 탐색과 수정이 가능하게 할 겁니다.

Read-side 동시성 제어는 Figure 10.10에서 보인 것처럼 RCU로 제공됩니다만, update-side 동시성 제어 함수인 `hashtab_lock_mode()`와 `hashtab_unlock_mod()`는 Figure 10.26에 보인 것처럼 동시의 크기 재조정 오퍼레이션 존재 가능성을 처리해야 합니다.

이 그림의 line 1-19에 `hashtab_lock_mod()`가 있습니다. Line 9는 데이터 구조가 획단을 하는 도중에 해제되는 일을 막기 위해 RCU read-side 크리티컬 섹션에 들어가고, line 10에서 현재 해시 테이블로의 레퍼런스를 얻어온 후, line 11에서 이 해시 테이블에서 현재 키에 연관되는 `bucket`으로의 레퍼런스를 얻어옵니다. Line 12는 이 `bucket`의 락을 획득하는데, 이 락은 동시의 크기 재조정 오퍼레이션이 그 `bucket`을 분산시키는 것을 막는데, 물론 크기 재조정 오퍼레이션이 이미 이 `bucket`을 분산시켰다면 아무 영향이 없을 것임에도 불구하고 그렇습니다. Line 13에서는 동시의 크기 재조정 오퍼레이션이 이미 이 `bucket`을 새 해시 테이블에 분산시켰는지 확인해보고, 그렇지 않다면 line 14에서 선택된 `bucket`의 락을 잡은 채로 (그리고 RCU read-side 크리티컬 섹션 안에 들어와 있는 채로) 리턴합니다.

그렇지 않고, 동시의 크기 재조정 오퍼레이션이 이미 이 `bucket`을 분산시킨 것으로, line 15에서 새로운 해시 테이블로 넘어가고 line 16에서 키에 연관되는 `bucket`

```

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_new = 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

을 선택합니다. 마지막으로, line 17에서 해당 bucket의 락을 잡고 line 18에서 기존 해시 테이블의 bucket을 위한 락을 놓아줍니다. 다시 한번, hashtab_lock_mod()은 RCU read-side 크리티컬 섹션 안에 들어와 있는 채로 빠져나가집니다.

Quick Quiz 10.9: Figures 10.25 와 10.26 의 코드는 업데이트를 위해 해시 값은 계산하고 bucket 선택 로직을 두번 수행하네요! 왜 이렇게 비효율적인거죠? ■

hashtab_unlock_mod() 함수는 hashtab_lock_mod()로 잡아두었던 락을 놓아줍니다. Line 28에서 현재 해시 테이블을 고르고, line 29에서 키에 연관된 bucket—당연히 이 bucket은 새로운 해시 테이블에 속해 있을 수도 있습니다—의 레퍼런스를 증가시키기 위해 ht_get_bucket()을 수행합니다. Line 30은 이 bucket의 락을 놓고 마지막으로 line 31에서 RCU read-side 크리티컬 섹션을 빠져나갑니다.

Quick Quiz 10.10: 크기 재조정 작업이 이루어지는 사이에 한 쓰레드가 새로운 해시 테이블에 원소를 넣는다고 생각해 봅시다. 뒤따르는 크기 재조정 작업이 이 삽입 작업 전에 완료됨으로 인해 이 삽입 작업이 없던 것처럼 되어버리는 문제는 어떻게 방지되고 있나요? ■

이제 bucket 선택과 동시성 제어를 다뤘으니, 이 크기 재조정 가능한 해시 테이블의 탐색과 업데이트를 할 준비가 되었습니다. hashtab_lookup(),

```

1 struct ht_elem *
2 hashtab_lookup(struct hashtab *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 hashtab_add(struct hashtab *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->htc_next[i],
36                      &htbp->htb_head);
37 }
38
39 void
40 hashtab_del(struct hashtab *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->htc_next[i]);
52 }

```

Figure 10.27: Resizable Hash-Table Access Functions

hashtab_add(), 그리고 hashtab_del() 함수들이 Figure 10.27에 보여져 있습니다.

이 그림의 line 1-21의 hashtab_lookup() 함수는 탐색을 합니다. Line 11에서는 현재 해시 테이블을 가져오고 line 12에서 특정 키에 연관된 bucket으로의 레퍼런스를 얻어옵니다. 이 bucket은 크기 재조정 오퍼레이션이 원하는 데이터 원소를 담고 있는 예전 해시 테이블의 bucket에서 진행되었을 때에는 새로운 크기 재조정된 해시 테이블에 위치해 있을 겁니다. Line 12는 또한 각 원소의 포인터 쌍들 중 올바른 것을 고르기 위한 인덱스 역시 리턴한다는 점을 알아 두세요. Line 13-

19의 루프는 bucket을 찾는데, line 16에서는 매치되는지를 판단하고, line 18에서 데이터 원소로의 포인터를 리턴하는데, 매치되지 않는다면 line 20에서 실패를 알리는 NULL을 리턴합니다.

Quick Quiz 10.11: Figure 10.27의 hashtable_lookup() 함수에서, 코드는 탐색될 원소가 이미 동시에 크기 재조정 작업에 의해 분산되었다면 새로운 해시 테이블에 있는 올바른 bucket을 찾아옵니다. 이건 RCU로 보호되는 탐색에서는 좀 낭비가 있는 것 같은데요. 이 경우에는 왜 그냥 기존 해시 테이블에서 작업을 끝내지 않는 거죠? ■

그림의 line 23-37의 hashtable_add() 함수는 해시 테이블에 새로운 데이터 원소를 넣습니다. Line 32-34는 앞에서와 같이 키에 연관되는 bucket으로의 포인터(그리고 인덱스)를 얻어오고, line 35는 새로운 원소를 테이블에 집어넣습니다. 호출자는 동시성을 잘 제어할 것이 요구되는데, 예를 들어 hashtable_add() 호출 전에 hashtable_lock_mod()를 호출하고, 후에 hashtable_unlock_mod()를 호출해야 하는 것 등입니다. 이런 두개의 동시성 제어 함수들은 동시에 크기 재조정 작업들과 올바르게 순서를 맞출 겁니다: 만약 이 크기 재조정 작업이 이 데이터 원소가 넣어질 bucket에 대해서 이미 진행되었다면, 이 원소는 새로운 테이블에 추가될 겁니다.

그림의 line 39-52의 hashtable_del() 함수는 해시 테이블에서 존재하는 원소를 하나 제거합니다. Line 48-50에서는 앞에서와 같이 bucket과 인덱스를 제공하고, line 51에서 특정 원소를 제거합니다. hashtable_add()에서와 같이, 호출자는 동시성 제어를 할 책임을 갖게 되고 이 동시성 제어는 동시에 크기 재조정 작업과의 동기화를 처리합니다.

Quick Quiz 10.12: Figure 10.27의 hashtable_del() 함수는 원소를 항상 기존 해시 테이블에서 제거하지는 않는데요. 이 말은 읽기 쓰레드들이 이 새로 제거된 원소를 해제된 후에도 볼 수 있다는 의미 아닌가요? ■

실제로 자신의 크기를 재조정 하는 일은 hashtable_resize로 수행되는데, page 180의 Figure 10.28에 보여져 있습니다. Line 17은 가장 높은 레벨의 ->ht_lock을 조건적으로 얻어오는데, 만약 락을 잡는데 실패했다면 line 18에서 크기 재조정이 이미 진행중임을 알리기 위해 -EBUSY를 리턴합니다. 그렇지 않다면, line 19에서 현재 해시 테이블로의 레퍼런스를 가져오고, line 21-24에서 원하는 크기의 새로운 해시 테이블을 할당받습니다. 새로운 hash/key 함수들이 명시된다면, 이것들이 새로운 테이블을 위해 사용되고, 그렇지 않다면 기존 테이블의 것들을 사용합니다. Line 25에서 메모리 할당 실패를 확인하게 된다면 line 26에서 ->ht_lock을 놓고 line 27에서 실패했음을 알리도록 리턴합니다.

Line 29는 bucket 분산 프로세스를 새로운 해시 테이블로의 레퍼런스를 기존 테이블의 ->ht_new에 넣음으로써 시작합니다. Line 30에서는 새로운 테이블의 존재를 알지 못하는 모든 읽기 쓰레드들이 크기 재조정 작업이 진행되기 전에 완료될 것을 보장합니다. Line 31에서는 현재 테이블의 인덱스를 얻어오고 그 역을 새로운 해시 테이블에 저장함으로써 두개의 해시 테이블이 서로의 링크드 리스트를 덮어쓰는 것을 막는 것을 보장합니다.

Line 33-44의 루프의 각 패스에서는 기존 해시 테이블의 bucket의 내용물 중 하나를 새로운 해시 테이블로 옮깁니다. Line 34에서는 기존 테이블의 현재 bucket으로의 레퍼런스를 가져오고, line 35에서 해당 bucket의 스펀락을 잡은 후 line 36에서 이 bucket이 분산되었음을 알리기 위해 ->ht_resize_cur를 업데이트합니다.

Quick Quiz 10.13: Figure 10.27의 hashtable_resize() 함수에서, line 29에서의 ->ht_new로의 업데이트가 line 36에서의 ->ht_resize_cur로의 업데이트 전에 일어난 것으로 hashtable_lookup(), hashtable_add(), 그리고 hashtable_del()의 관점에 보일 것을 무엇이 보장하죠? ■

Line 37-42의 루프의 각 패스에서는 데이터 원소 하나를 현재의 기존 테이블의 bucket에서 연관된 새로운 테이블의 bucket으로 옮기는는데, 이 때 새로운 테이블의 bucket의 락을 잡고 있습니다. 마지막으로, line 43에서는 기존 테이블 bucket의 락을 놓게 됩니다.

일단 모든 기존 테이블의 bucket들이 새로운 테이블로 분산되면 line 45가 수행됩니다. Line 45는 새로 생성된 테이블을 현재의 것으로 보이게 만들고, line 46에서는 (여전히 기존 테이블을 레퍼런스하고 있을) 기존 읽기 쓰레드들이 완료되기를 기다립니다. 그리고 나서 line 47에서는 크기 재조정 직렬화 락을 놓고, line 48에서 기존 hash table을 해제하고, line 48에서 마침내 성공했음을 리턴합니다.

10.4.3 Resizable Hash Table Discussion

Figure 10.29는 크기 재조정 해시 테이블을 고정 크기 버전과 2048, 16,384, 그리고 131,702 개 원소에 대해 비교해 본 결과입니다. 이 그림은 각 원소 수마다 세개씩의 선을 보여주는데, 하나는 고정크기 1024-bucket 해시 테이블의 것이고, 하나는 고정크기 2048-bucket 해시 테이블, 나머지 하나는 1024 개와 2048 개 bucket 사이를 오갈 수 있는 크기 재조정 가능한 해시테이블 버전으로, 각각의 크기 재조정 작업 사이에는 1 밀리세컨드 멈춥니다.

가장 위의 세개의 선들은 2048 개 원소의 해시테이블의 것입니다. 위쪽 선은 2048개 bucket을 사용하는

```

1 int hashtable_resize(struct hashtable *http_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 *http;
8     struct ht *http_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(&http_master->ht_lock))
18        return -EBUSY;
19    http = http_master->ht_cur;
20    http_new = ht_alloc(nbuckets,
21                       hash_private ? hash_private : http->ht_hash_private,
22                       cmp ? cmp : http->ht_cmp,
23                       gethash ? gethash : http->ht_gethash,
24                       getkey ? getkey : http->ht_getkey);
25    if (http_new == NULL) {
26        spin_unlock(&http_master->ht_lock);
27        return -ENOMEM;
28    }
29    http->ht_new = http_new;
30    synchronize_rcu();
31    idx = http->ht_idx;
32    http_new->ht_idx = !idx;
33    for (i = 0; i < http->ht_nbuckets; i++) {
34        htp = &http->ht_bkt[i];
35        spin_lock(&htp->htb_lock);
36        http->ht_resize_cur = i;
37        cds_list_for_each_entry(htep, &htp->htb_head, hte_next[idx]) {
38            htp_new = ht_get_bucket_single(http_new, http_new->ht_getkey(htep), &b);
39            spin_lock(&htp_new->htb_lock);
40            cds_list_add_rcu(&htep->hte_next[!idx], &htp_new->htb_head);
41            spin_unlock(&htp_new->htb_lock);
42        }
43        spin_unlock(&htp->htb_lock);
44    }
45    rcu_assign_pointer(http_master->ht_cur, http_new);
46    synchronize_rcu();
47    spin_unlock(&http_master->ht_lock);
48    free(htp);
49    return 0;
50 }

```

Figure 10.28: Resizable Hash-Table Resizing

고정 크기 해시테이블의 것이고, 중간의 것은 1024개 bucket 을 사용하는 고정크기 해시 테이블, 그리고 아래의 것은 크기 재조정 가능한 해시 테이블입니다. 이 경우, 짧은 해시 체인들이 평범한 탐색 오버헤드를 매우 낮게 만들어서 크기 재조정 오버헤드가 지배적이게 만들었습니다. 더도아니고 덜도 아니고, 더 큰 고정 크기 해시 테이블을 상당한 성능 이득을 가졌고, 따라서 크기 재조정은 크기 재조정 작업 사이에 충분한 시간이 주어진다면 상당한 이득을 보일 겁니다: 1 밀리세컨드는 분명 너무 짧은 시간입니다.

중간의 세개 선들은 16,384 개 원소를 가진 해시 테이블의 결과입니다. 이번에도 위쪽의 선은 2048개 bucket 을 가진 고정 크기 해시 테이블을 위한 것입니다만, 중

간의 선은 크기 재조정 해시 테이블의 것이고 아래의 것은 1024개 bucket 의 고정크기 해시 테이블입니다. 하지만, 크기 재조정 버전과 1024개 bucket 버전 해시 테이블 사이의 성능 차이는 상당히 적은 것을 볼 수 있습니다. 원소의 수를 (따라서 해시 체인의 길이도) 8배로 증가시킨 것의 한가지 결과는, 끊임없는 크기 재조정은 이제 너무 작은 해시 테이블을 유지하는 것보다 더 나쁘지는 않다는 것입니다.

아래쪽의 세개 선은 131,072 원소를 갖는 해시 테이블입니다. 위쪽 선은 2048 개 bucket 의 고정크기 해시 테이블이고, 중간의 것은 크기 재조정 가능 해시 테이블, 그리고 아래의 것은 1024개 bucket 사용 해시 테이블입니다. 이 경우, 더 길어진 해시체인은 높은 탐색 오

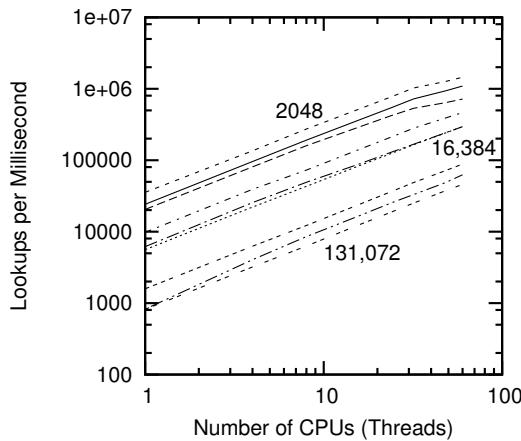


Figure 10.29: Overhead of Resizing Hash Tables

버헤드를 가져왔고, 따라서 이 템색 오버헤드가 해시 테이블의 크기 재조정을 넘어섰습니다. 하지만, 세 개 해시 테이블 모두 131,072 원소 성능 수준은 2048개 원소에서의 수준보다 10배 이상 나빠졌는데, 이는 해시 테이블 크기를 64배 늘리는 것이 최고의 방법일 것임을 의미합니다.

이 데이터의 핵심은 RCU로 보호되는 크기 재조정 가능한 해시 테이블은 고정 크기 버전 만큼이나 성능을 보이고 확장될 수 있다는 겁니다. 실제 크기 재조정 작업 동안의 성능은 물론 안좋은데, 각 원소의 포인터들에의 업데이트가 일으키는 캐시 미스 때문일 것이고 이 영향은 해시 테이블 bucket 리스트가 짧을 때 두드러질 것입니다. 이는 해시 테이블은 상당한 크기에서 크기 재조정되어야 하고 너무 빈번한 크기 재조정 작업으로 인한 성능 저하를 막기 위해 기록에 따른 조정(hysteresis)을 가져야 할 겁니다. 메모리가 충분히 큰 환경이라면, 해시 테이블 크기는 줄어들기보다는 더 공격적으로 증가될 수 있을 겁니다.

또 다른 중요한 점은 `hashtab` 구조체가 분할될 수는 없지만 이것도 읽기가 대부분이므로 RCU를 사용해 볼 만 합니다. 이 크기 재조정 가능한 해시 테이블의 성능과 확장성이 RCU로 보호되는 고정크기 해시 테이블에 근접한다는 점을 놓고 보면, 이 방법이 상당히 성공적이라고 결론내릴 수 있을 겁니다.

마지막으로, 삽입, 삭제, 그리고 템색은 크기 재조정 작업과 동시에 이뤄질 수 있다는 점이 중요합니다. 이 동시성은 커다란 해시 테이블을 크기 재조정할 때 매우 중요한데, 특히 상당한 응답시간 제약을 가진 어플리케이션에서 그려합니다.

물론, `ht_elem` 구조체의 포인터 집합 쌍은 약간의 메모리 오버헤드를 내포하는데, 이에 대해서는 다음 섹션에서 이야기 합니다.

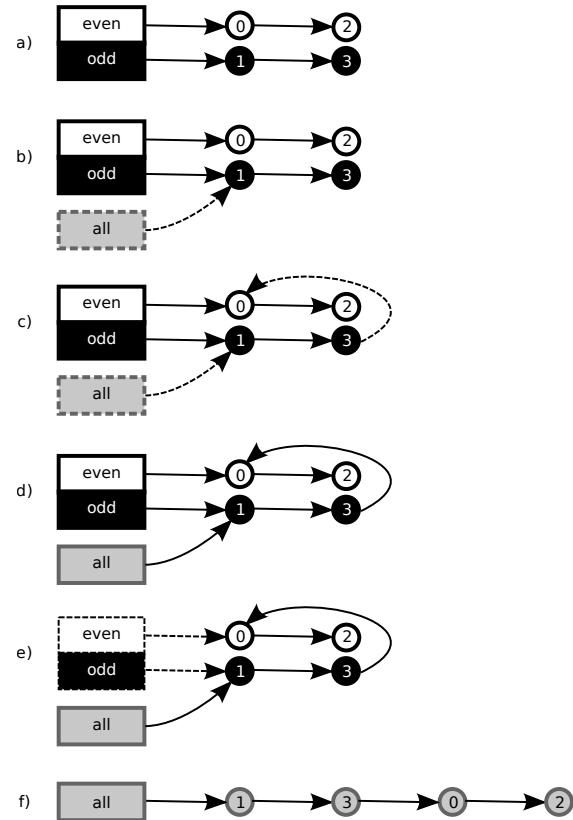


Figure 10.30: Shrinking a Relativistic Hash Table

10.4.4 Other Resizable Hash Tables

이 섹션의 앞에서 이야기한 크기 재조정 가능한 해시 테이블의 한 가지 한계는 메모리 사용량입니다. 각각의 데이터 원소는 한 개가 아니라 두 개의 링크드 리스트 포인터 쌍을 가지고 있습니다. 하나의 쌍만 가지고 일을 해낼 수 있는 RCU로 보호되는 크기 재조정 가능한 해시 테이블을 만들 수는 없는 걸까요?

그에 대한 답은 “가능하다”로 드러났습니다. Josh Triplett 등은 연관된 해시 체인을 점진적으로 쪼개고 조합해서 읽기 쓰레드들은 크기 재조정 작업 중간의 모든 순간에 올바른 해시 체인을 바라보게 되는 상재론적 해시 테이블 [TMW11]을 만들었습니다. 이 점진적인 쪼개기와 조합하기는 읽기 쓰레드들이 다른 해시 체인에 있어야 하는 데이터 원소를 보게 되는 것은 문제가 되지 않는다는 사실에 기인합니다: 이런 일이 일어난다면, 읽기 쓰레드는 키가 맞지 않으므로 해당 데이터 원소를 무시해 버리게 됩니다.

상재론적 해시 테이블을 절반으로 축소시키는 과정은 Figure 10.30에 그려져 있는데, 이 경우에는 두 개의

bucket 을 가지고 있는 해시 테이블을 한개 bucket 의 해시 테이블, 달리 말해 하나의 선형 리스트로 축소시키고 있습니다. 이 과정은 기존의 커다란 해시 테이블의 한쌍의 bucket 들을 새로운 작은 해시 테이블의 하나의 bucket 으로 합치는 것으로 동작합니다. 이 프로세스가 잘 동작하려면, 두 테이블의 해시 함수에 대해 제약을 걸어야 하는 것이 분명합니다. 그런 제약 가운데 하나는 두 테이블 모두 같은 해시 함수를 사용해야 하지만, 큰 테이블에서 작은 테이블로 축소시킬 때에는 뒤쪽의 비트 하나는 버려야 한다는 것입니다. 예를 들어, 기존의 두개 bucket 을 사용하는 해시 테이블은 값의 두개의 꼭대기 비트를 사용하는 반면, 새로운 한개 bucket 의 해시 테이블은 그 값의 꼭대기 비트 하나를 사용할 겁니다. 이런 방식으로, 기존의 커다란 테이블의 인접한 짹수와 홀수 bucket 들은 새로운 작은 해시 테이블의 하나의 bucket 으로 합쳐질 수 있고, 그동안 하나의 해시 값이 이 하나의 bucket 의 원소들을 다룰 수 있습니다.

최초의 상태가 그림의 꼭대기에 보여져 있는데, 아래쪽으로 갈수록 시간이 흐르게 되고, 최초의 상황 (a)에서 시작합니다. 축소 과정은 새로운, bucket 들의 더 작은 배열을 할당하는 것으로 시작하고, 이 새로운 작은 배열의 각각의 bucket 이 기존의 커다란 해시 테이블의 연관된 bucket 들 중 하나의 첫번째 원소를 레퍼런스하도록 해서 상황 (b)가 되게 합니다.

이제 두개의 해시 체인들은 함께 연결되어서 상태 (c)가 됩니다. 이 상태에서, 짹수로 해시값의 원소를 보는 읽기 쓰레드들은 아무 변화도 보지 못하게 되고, 원소 1과 3 을 보는 읽기 쓰레드들은 역시 변화를 보지 못합니다. 하지만, 그와 다른 홀수 해시값의 원소를 찾는 읽기 쓰레드들은 원소 0 과 2 도 지나가게 될겁니다. 어떤 홀수 해시값도 이 두개의 원소들과 같지 않을 것이므로 이는 문제가 되지 않습니다. 이로 인한 약간의 성능 저하가 있지만, 반면에, 이는 새로운 작은 해시 테이블이 자리를 잡게 되면 겪게 될 성능 저하와 완전히 똑같은 정도입니다.

다음으로, 새로운 작은 해시 테이블은 읽기 쓰레드들에게 접근 가능하게 되어서 state (d) 가 됩니다. 오래된 읽기 쓰레드들은 여전히 기존의 커다란 해시 테이블을 횡단하고 있을 수 있으므로, 이 상태는 두개의 해시 테이블을 모두 사용중인 것으로 합니다.

다음 할 일은 모든 기존부터 존재한 읽기 쓰레드들이 완료되길 기다리는 것으로, state (e) 에 도달하게 됩니다. 이 상태에서, 모든 읽기 쓰레드들은 새로운 작은 해시 테이블을 사용하고 있으므로, 기존의 커다란 해시 테이블의 bucket 들은 해제될 수 있어서, 마지막 상태 (f) 로 이르게 됩니다.

상대론적 해시 테이블의 크기를 키우는 건 축소 프로세스의 거꾸로입니다만, 더 많은 grace-period 단계를 필요로하게 되는데, Figure 10.31 에 그려져 있습니다.

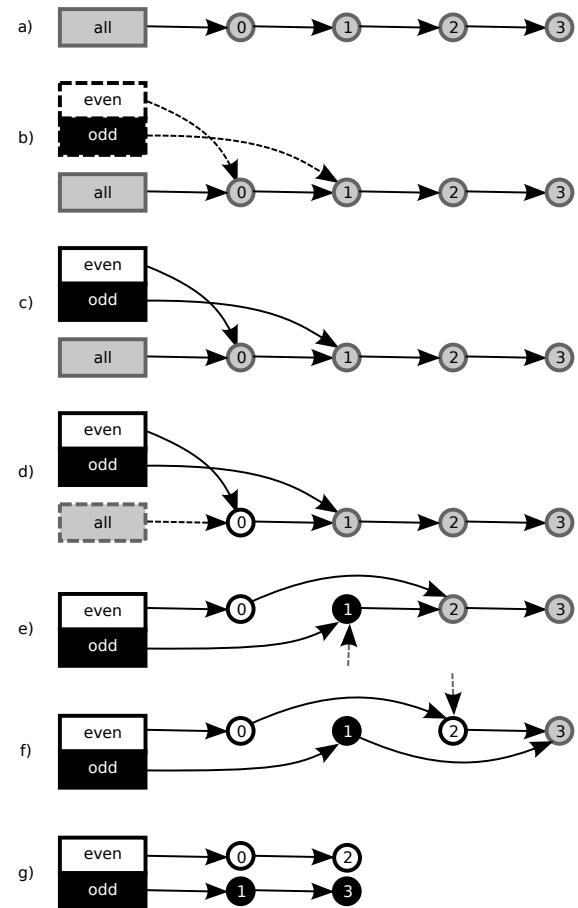


Figure 10.31: Growing a Relativistic Hash Table

최초 상태 (a) 는 그림의 맨 꼭대기에 있고, 아래로 시간에 따라 진행됩니다.

새로운 커다란 두개의 bucket 을 사용하는 해시 테이블을 할당하는 것으로 시작해서, state (b)로 이르게 됩니다. 이 새로운 bucket 들 각각은 그 bucket 을 향하게 될 첫번째 원소를 레퍼런스함을 알아두세요. 이 새로운 bucket 들은 읽기 쓰레드들에게 공개되고, 이로 인해 state (c)로 이릅니다. 하나의 grace-period 가 지나고, 모든 읽기 쓰레드들은 새로운 커다란 해시 테이블을 사용하게 되어서 state (d) 에 이릅니다. 이 상태에서, 짹수 해시값의 원소를 찾는 읽기 쓰레드들만이 하얀 색으로 칠해진 원소 0 을 경유하게 됩니다.

이 시점에서, 기존의 작은 해시 bucket 들은 해제될 수 있지만, 많은 구현들은 아이템들의 리스트를 새로운 bucket 으로 “풀어 놓는” 작업의 진도 상황을 보기 위해 이 기존 bucket 들을 사용하고는 합니다. 그런 원소들의 첫번째 연속적 수행에서의 마지막 짹수 해시값 원소는

다음으로의 포인터가 다음의 짹수 해시값 원소를 가리키도록 업데이트 됩니다. 다음 grace period 후에는, 상태 (e) 가 됩니다. 수직의 화살표는 다음으로 풀리게 될 원소를 가리키고, 원소 1 은 이제 홀수 해시값의 원소를 찾는 읽기 쓰레드들만이 다다르게 될 것을 가리키기 위해 검정색으로 칠해져 있습니다.

다음으로, 그런 원소들의 첫번째 연속적 수행에서의 마지막 홀수 해시값 원소가 다음 원소로의 포인터를 다음의 홀수 해시값 원소를 가리키도록 업데이트 합니다. 다음의 grace period 가 지난후, 상태 (f) 로 다다릅니다. 마지막 풀어놓기 오퍼레이션 (하나의 grace period 를 포함해서) 은 마지막 상태 (g) 로 이르게 합니다.

짧게 말해서, 상대론적 해시 테이블은 원소별 리스트 포인터의 수를 크기 재조정 동안 일어나는 추가적인 grace period 의 비용으로 대신합니다. 삽입, 삭제, 그리고 탐색은 크기 재조정 작업과 동시에 이루어질 수 있으므로, 이런 추가적인 grace period 들은 일반적으로 문제가 되지 않습니다.

원소별 메모리 오버헤드가 한쌍의 포인터에서 하나의 포인터로 줄어들 수 있으면서도 여전히 $O(1)$ 삭제를 유지할 수 있음이 드러났습니다. 이는 split-order 리스트 [SS06] 를 RCU 보호로 증강시킴으로써 가능합니다 [Des09, MDJ13a]. 이런 해시 테이블 안의 데이터 원소는 하나의 정렬된 링크드 리스트로 정렬되고, 각각의 해시 bucket 은 그 bucket 내의 첫번째 원소를 레퍼런스 하게 됩니다. 원소들은 다음 원소로의 포인터 필드의 아래쪽 비트를 설정하는 것으로 지워지고, 이 원소들은 그것들을 마주하게 되는 나중의 횟단에 의해 리스트에서 제거됩니다.

이 RCU 로 보호되는 split-order 리스트는 복잡하지만, lock-free 진행 보장을 모든 삽입, 삭제, 그리고 탐색 작업에 제공합니다. 그런 보장사항은 리얼타임 어플리케이션들에서는 중요할 수 있습니다. 최신 버전의 userspace RCU 라이브러리 [Des09] 에서 이 구현을 사용할 수 있습니다.

10.5 Other Data Structures

앞의 섹션들은 파티셔닝 가능성 (Section 10.2), 읽기가 대부분인 액세스 패턴의 효과적인 처리 (Section 10.3), 또는 읽기가 대부분인 상황에서의 테크닉을 파티셔닝이 불가능함을 해결하기 위해 적용하기 (Section 10.4) 로 동시성을 향상시킨 데이터 구조들에 대해 주목해 보았습니다. 이 섹션은 다른 데이터 구조들에 대한 간략한 리뷰를 합니다.

해시 테이블의 병렬 사용에서의 가장 훌륭한 이점은 완전히 파티셔닝 가능하다는 점인데, 적어도 크기 재조정이 되지 않는 동안은 그렇습니다. 파티셔닝 가능성과 크기 독립성을 둘 다 보호하기 위한 한가지 방법은 trie

라고도 불리는 래딕스 트리(radix tree) 의 사용입니다. Trie 들은 탐색 키를 분할하는데, 각각의 다음 키 부분을 다음 레벨의 trie 를 탐색하도록 사용하는 식입니다. 그렇게 되면, 하나의 trie 는 중첩된 해시 테이블들의 한집합으로 생각될 수 있으므로, 필요한 파티셔닝 가능성을 제공합니다. Trie 들의 한가지 단점은 드문드문한 키 스페이스는 메모리의 비효율적인 사용을 초래할 수 있다는 점입니다. 이런 단점을 보완할 수 있는 몇가지 압축 기법들이 존재하는데, 탐색 전에 키 값을 더 작은 키 스페이스로 해싱을 하는 방법 [ON06] 등이 있습니다. 래딕스 트리는 실제로 많이 사용되는데, 리눅스 커널도 포함됩니다 [Pig06].

해시 테이블과 trie 모두에서의 한가지 중요한 특수 케이스는 아마도 가장 오래된 데이터 구조체이고 배열과 여러 차원의 대응되는 개념인, 행렬입니다. 행렬의 완전히 파티셔닝 가능하다는 본질은 동시적 수학 알고리즘에서 상당히 많이 활용되었습니다.

스스로 밸러스를 잡는 트리들은 순차적 코드에서 상당히 많이 사용되었는데, AVL 트리와 red-black 트리는 아마도 가장 널리 알려진 예일 겁니다 [CLRS01]. AVL 트리들을 병렬화 하려 한 기존의 시도들은 복잡하고 그다지 효율적이지 못했습니다만 [Ell80], 최근의 red-black 트리에서의 작업은 읽기 쓰레드에게 RCU 를 사용하고 읽기와 업데이트를 각각 보호하는데에 해시를 사용한 락들의 배열¹ 을 사용해서 더 나은 성능과 확장성을 제공합니다 [HW11, HW13]. Red-black 트리는 적극적으로 밸런스를 재조정함으로 드러났는데, 이는 순차적 프로그램에서는 잘 동작하지만 병렬적 사용에서는 꼭 그렇지만은 않습니다. 따라서 최근의 작업은 RCU 로 보호되는 “bonsai tree” 를 만들어냈는데, 이 데이터 구조는 멀 적극적으로 밸러스 재조정을 해서 [CKZ12] 최적의 트리 깊이와 더 효과적인 동시의 업데이트 사이의 트레이드 오프를 합니다.

동시의 스kip 리스트들 역시 RCU 읽기 쓰레드들에게 좋은데, 실제로 RCU 를 사용하는 테크닉의 사용을 초기의 학계에서 선보였습니다 [Pug90].

동시적 double-ended 큐들은 Section 6.1.2 에서 다루었고, 동시적 스택과 큐들은 긴 역사를 가지고 있습니다만 [Tre86], 일반적으로 인상적인 성능과 확장성을 보이진 못함에도 불구하고 그렇습니다. 이것들은 동시성 라이브러리들의 혼란 가능 이상도 이하도 아닙니다 [MDJ13b]. 연구자들은 최근에 스택과 큐의 순서 규칙을 완화시키는 것을 제안했는데 [Sha11], 일부 작업은 완화된 순서 규칙의 큐가 실제로는 엄격한 FIFO 큐보다 더 나은 순서 특성을 실제로 가짐을 보이기도 했습니다 [HKLP12, KLP12, HHK⁺13].

¹ 개발자가 공유되지 않는 액세스를 직접 표시하는 소프트웨어 트랜잭션 메모리의 하나인 swissTM [DFGG11] 으로 가장 해서

동시적 데이터 구조에 대한 지속적 작업은 놀라운 특성을 가진 기발한 알고리즘들을 만들어낼 것으로 보입니다.

10.6 Micro-Optimization

이 섹션에 보인 데이터 구조들은 간단히 코딩 되었고, 시스템의 캐시 구조에 대한 적용도 없었습니다. 또한, 많은 구현들이 키에서 해시로의 변환과 다른 빈번한 작업들을 위해 함수로의 포인터들을 사용했습니다. 이런 방법이 간단하고 포팅 가능성을 높여주지만, 많은 경우에 이는 어느정도 성능을 포기하게 합니다.

다음 섹션들에서는 특수화, 메모리 절약, 그리고 하드웨어 고려에 대해 다뤄봅니다. 이 짧은 섹션들이 분명한 이 주제에 대한 전문적 정리된 내용이라고 생각하는 실수를 하지 마세요. 이 내용들은 특정 CPU에서의 최적화에 관해 쓰여진 것이므로 오늘날 흔히 사용되는 CPU들과는 별개입니다.

10.6.1 Specialization

Section 10.4에 선보여진 크기 재조정 가능한 해시 테이블은 카에 불투명한 타입을 사용했습니다. 이는 커다란 유연성을 가능하게 해서, 어떤 종류의 키도 사용될 수 있게 했습니다만, 또한 함수로의 포인터를 사용한 함수 호출로 인해 상당한 오버헤드를 만들었습니다. 최신의 하드웨어는 이런 오버헤드를 최소화하기 위해 세련된 브랜치 예측 테크닉들을 사용합니다만, 다른 한편으로 실제 세상의 소프트웨어는 오늘날의 커다란 하드웨어 브랜치 예측 테이블로도 수용할 수 없을 만큼 큰 경우가 많습니다. 이는 특히 함수 포인터를 통한 호출의 경우에 그러한데, 브랜치 예측 하드웨어는 브랜치가 취해졌는지 안취해졌는지에 대한 정보에 더해서 포인터를 기록해야 하는 경우이기 때문입니다.

이 오버헤드는 해시 테이블 구현을 특정 키 타입과 해시 함수로 특수화 시킴으로써 제거될 수 있습니다. 그렇게 하는 것은 page 176의 Figure 10.24에서 보인 ht 구조체에서 `->ht_cmp()`, `->ht_gethash()`, 그리고 `->ht_getkey()` 함수 포인터들을 제거합니다. 이는 또한 연관된 포인터들을 통한 함수 호출도 제거하는데, 이는 컴파일러가 이 고정된 함수들을 인라인 시킬 수 있게 해서 호출 명령의 오버헤드만이 아니라 인자 정리 오버헤드 역시 제거할 것입니다.

또한, 이 크기 재조정 가능한 해시 테이블은 동시성 제어로부터 bucket 선택을 분리하는 API에 걸맞도록 설계되었습니다. 이는 하나의 테스트가 이 챕터의 모든 해시 테이블의 구현을 테스트할 수 있도록 해주지만, 이는 또한 많은 연산작업들이 해시 값을 계산하고 만들어질 수 있는 크기 재조정 작업과 한번이 아니라

두번 상호작용해야 함을 의미합니다. 성능이 중요한 환경에서라면 `hashtab_lock_mod()` 함수 또한 선택된 bucket으로의 레퍼런스를 리턴해서 이어지는 `ht_get_bucket()` 호출을 제거할 것입니다.

Quick Quiz 10.14: `hashtorture.h`의 코드는 `hashtab_lock_mode()` 가 `ht_get_bucket()` 의 기능을 포섭하도록 수정될 수 없나요? ■

Quick Quiz 10.15: 이런 특수화는 정말로 얼마나 성능을 구하나요? 이게 정말 가치가 있나요? ■

그렇다고는 해도, 제가 처음 프로그램을 배우기 시작한 1970년대 초반에 사용할 수 있던 것들에 비해 최신 하드웨어의 커다란 장점중 하나는 그렇게까지나 특수화가 필요하지는 않다는 것입니다. 이는 4 킬로바이트 주소공간을 사용하던 시절에 그랬던 것에 비해 훨씬 많은 생산성을 가능하게 합니다.

10.6.2 Bits and Bytes

이 챕터에서 논의한 해시 테이블들은 메모리 절약을 위한 시도는 거의 하지 않도록 만들어졌습니다. 예를 들어, page 176의 Figure 10.24에 있는 ht 구조체의 `->ht_idx` 필드는 항상 0 또는 1의 값을 갖는데도 32비트의 메모리를 모두 가집니다. 이건 예를 들어 `->ht_resize_key` 필드에서 비트 하나를 훔쳐다 사용하는 식으로 제거될 수 있을 겁니다. 이 방법은 `->ht_resize_key` 필드가 메모리의 모든 바이트를 가리킬 수 있을 만큼 커다랗고 `ht_bucket` 구조체는 1 바이트 보다 크기 때문에 `->ht_resize_key` 필드는 몇 비트는 아낄 수 있을 것이기 때문에 동작 가능합니다.

이런 부류의 bit-packing 트릭은 리눅스에서의 page 구조체와 같이 많이 복제되는 데이터 구조체들에서 빈번하게 사용됩니다. 하지만, 이 크기 재조정한 해시 테이블의 ht 구조체는 그렇게까지 많이 복제되지는 않습니다. 대신에 우리가 집중해야 할 것은 `ht_bucket` 구조체입니다. `ht_bucket` 구조체의 크기를 줄일 수 있는 두개의 커다란 기회가 존재합니다: (1) `->htb_lock` 필드를 `->htb_head` 포인터들 중 하나의 아래쪽 비트에 위치시키는 것과 (2) 필요한 포인터들의 갯수를 줄이는 것입니다.

첫번째 기회는 `include/linux/bit_spinlock.h`로 제공되는 리눅스 커널의 비트 스피너를 사용할 수도 있을 겁니다. 이것들은 리눅스 커널의 공간 크기에 민감한 데이터 구조체들에 사용됩니다만 그 단점도 존재합니다:

1. 기존의 스피너 기능들에 비해 상당히 느립니다.
2. 리눅스 커널의 데드락 파악 도구인 `lockdep` [Cor06a]과 함께 사용될 수 없습니다.

3. 락 소유권을 기록하지 않아서 디버깅을 복잡하게 만듭니다.
4. -rt 커널에서 우선권 상승 기능과 함께 동작하지 않는데, 이는 비트 스피드을 잡을 때에는 프리엠션 기능이 꺼져야 해서 리얼타임 대기시간을 나쁘게 만들 수 있습니다.

이런 단점들에도 불구하고, 비트 스피드은 메모리가 가장 중요할 때에 상당히 유용합니다.

두 번째 기회에 대한 한 가지 측면은 Section 10.4.4에서 다루어졌는데, Section 10.4에서 보여진 크기 재조정 가능 해시 테이블에서는 두 개의 집합이 필요했던 자리에 한 개의 bucket 리스트 포인터를 집합만을 필요로 했습니다. 또 다른 방법은 이 챕터에서 사용된 양방향 링크드 리스트 대신에 단방향으로 링크된 bucket 리스트를 사용하는 것이 되겠습니다. 이 방법의 한 가지 단점은 삭제 작업이 추가적 오버헤드를 가져오게 될 것인데, 차후의 삭제를 위한 바깥으로 향하는 포인터를 마크하거나 삭제될 원소를 위해 bucket 리스트를 담색하는 작업으로 인한 오버헤드가 될 것입니다.

요약해서, 최소한의 메모리 오버헤드와 성능과 단순성 등 사이에는 트레이드오프가 존재합니다. 다행히도, 최신 시스템에서는 사용 가능한 비교적 커다란 메모리들은 성능과 단순성을 메모리 오버헤드보다 우선시 할 수 있도록 해줍니다. 하지만, 오늘날의 커다란 메모리 시스템들에서도² 가끔은 메모리 오버헤드를 줄이기 위한 극단적인 조사가 필요합니다.

10.6.3 Hardware Considerations

최신 컴퓨터들은 일반적으로 CPU 와 메인 메모리 사이에서 데이터를 32바이트에서 256바이트 사이의 고정된 크기의 블록으로 읽습니다. 이 블록들은 캐시 라인이라 불리는데, Section 3.2에서 논의된 것처럼, 높은 성능과 확장성에 있어 매우 중요합니다. 성능과 확장성을 모두 죽여버리는 오래된 방법 하나는 호환불가한 변수들을 같은 캐시라인에 집어 넣는 것입니다. 예를 들어, 크기 재조정 가능한 해시 테이블 데이터 원소가 ht_elem 구조체를 상당히 빠르게 증가되는 카운터와 같은 캐시라인에 위치했다고 생각해 보세요. 같은 카운터 증가는 해당 캐시라인이 카운터 증가를 하는 CPU에게만 보여질 것입니다. 만약 다른 CPU가 해당 원소를 담고 있는 bucket 리스트를 횡단하려 하면, 이는 비싼 캐시 미스를 일으켜서 성능과 확장성을 모두 떨어뜨릴 겁니다.

64-바이트 캐시 라인의 시스템에서 이 문제를 해결하는 방법 가운데 하나가 Figure 10.32에 보여져 있습니다. 여기서 gcc aligned 속성은 ->counter 와 -> ht_elem 구조체가 서로 다른 캐시 라인에 위치하게 함

² 기가바이트 단위 메모리의 스마트폰, 있나요?

```
1 struct hash_elem {
2     struct ht_elem e;
3     long __attribute__ ((aligned(64))) counter;
4 };
```

Figure 10.32: Alignment for 64-Byte Cache Lines

니다. 이는 같은 카운터 증가에도 불구하고 CPU들이 bucket 리스트를 빠른 속도로 횡단할 수 있게 해줍니다.

물론, 이는 “캐시 라인의 크기가 64 바이트라는 걸 어떻게 알았을까?”라는 질문을 떠올리게 합니다. 리눅스 시스템에서, 이 정보는 /sys/devices/system/cpu/cpu*/cache 디렉토리에서 얻을 수 있고, 설치 과정에서 해당 시스템의 하드웨어 구조에 적합하도록 어플리케이션을 다시 빌드하게 만들 수도 있습니다. 더 나아가서, 설정 프로그램을 리눅스에서만 돌릴 생각이라 해도, 그런 스스로 수정을 하는 설치 방법은 실증을 필요로 합니다.

다행히도, 1995년의 논문 [GKPS95]에는 합리적인 수준으로 잘 동작하는, 경험에 의거한 규칙들이 몇 가지 있습니다.³ 규칙들의 그룹들 중 첫 번째 그룹은 구조체를 적합한 캐시의 기하학적 구조에 맞춰 재배치 하는 것에 관한 것입니다:

1. 읽기가 대부분인 데이터를 빈번하게 업데이트되는 데이터와 분리시키세요. 예를 들어, 읽기가 대부분인 데이터를 구조체의 앞에 위치시키고 빈번하게 업데이트되는 데이터는 끝에 위치시켜야 합니다. 가능하다면, 가끔만 접근되는 데이터를 그 사이에 위치시키세요.
2. 만약 구조체가 여러 그룹의 필드를 가져서 각각의 그룹이 독립적인 코드 수행 경로에서 업데이트된다면, 각각의 그룹들을 분리시키세요. 여기서도 역시, 가끔만 접근되는 데이터를 그룹들 사이에 위치시키는 게 효과를 발휘할 수 있습니다. 일부 경우에 있어서는 각각의 그룹을 원본 구조체에서 래퍼런스될 수 있는 별개의 구조체들에 위치시키는 것도 효과를 발휘할 수 있습니다.
3. 가능하다면, 업데이트가 대부분인 데이터를 CPU, 쓰레드, 또는 태스크와 연관지으세요. Chapter 5에서의 카운터 구현에서 이 경험적 규칙의 효과적인 예를 본 바 있습니다.
4. 가능하다면 Chapter 8에서 논의된 것처럼 데이터를 CPU 별로, 쓰레드 별로, 또는 태스크 별로 분리시켜야 합니다.

³ 이런 규칙들 여럿은 여기서는 Orran Krieger의 허락 하에 의역되고 확장되었습니다.

최근에는 메모리 액세스 기록에 기반한 구조체 필드 재배열의 자동화를 위한 노력들이 있었습니다 [GDZE10]. 이는 멀티쓰레드 소프트웨어에서 훌륭한 성능과 확장성을 위해 필요로 되는 고통스러운 작업의 고통을 경감시켜줄 수 있을 겁니다.

추가적인 경험적 법칙들은 락에 대해 다릅니다:

1. 자주 수정되는 데이터를 보호하며 상당히 경쟁을 받게 되는 락은 다음의 방법 중 하나를 따라야 합니다:
 - (a) 락을 그것이 보호하는 데이터와 다른 캐시라인에 위치시키는 것.
 - (b) 높은 경쟁수위에 맞춰 만들어진, queued 락과 같은 락을 사용할 것.
 - (c) 락 경쟁 수위를 줄이기 위해 설계를 다시 할 것. (이 방법이 최고입니다만 상당한 작업을 필요로 합니다.)
2. 경쟁을 하지 않는 락들은 그것들이 보호하는 데이터와 같은 캐시 라인에 위치시키세요. 이 방법은 락을 현재 CPU로 들고오는 캐시미스가 데이터도 함께 가져오도록 함을 의미합니다.
3. 읽기가 대부분인 데이터를 RCU로 보호하고, 만약 RCU가 사용될 수 없고 크리티컬 섹션이 굉장히 길다면, reader-writer 락을 사용하세요.

물론, 이것들은 절대적 규칙이 아니라 경험적 규칙입니다. 특정 상황에서 무엇이 가장 적합한 것인지를 알아내기 위해서는 일부 실험이 필요합니다.

10.7 Summary

이 챕터는 기본적으로 해시 테이블에 집중했고, 완전히 분할 가능하지는 않은 크기 재조정 가능한 해시 테이블에 대해서도 알아봤습니다. Section 10.5에서는 일부 해시 테이블 외의 데이터 구조에 대해 빠르게 대략적으로 알아봤습니다. 더도 아니고 덜도 아니고, 이 해시 테이블들의 소개는 고성능의 자료 액세스를 둘러싼 다음과 같은 많은 문제들에 대한 훌륭한 소개입니다:

1. 완전히 파티셔닝 가능한 데이터 구조는 예를 들어 하나의 socket의 시스템과 같은 작은 시스템에서 잘 동작합니다.
2. 더 큰 시스템은 완전한 파티셔닝 가능성은 물론이고 레퍼런스의 지역성(locality)를 필요로 합니다.

3. 해저드 포인터와 RCU 같은, 읽기가 대부분인 상황을 위한 기법들은 읽기가 대부분인 워크로드에서의 레퍼런스에 훌륭한 지역성을 제공하고, 따라서 커다란 시스템들에서도 훌륭한 성능과 확장성을 제공합니다.
4. 읽기가 대부분인 상황을 위한 기법들은 또한, 크기 재조정 가능한 해시 테이블들과 같은, 일부의 파티셔닝이 불가능한 데이터 구조에서도 잘 동작합니다.
5. 데이터 구조를 특정 워크로드에 특수화 시킴으로써 추가적인 성능과 확장성을 얻을 수 있습니다. 예를 들어, 일반적인 키를 32-bit 정수로 교체하는 방법으로요.
6. 이식성과 높은 성능은 일반적으로 서로 상충하지만, 이 두개의 요구사항들 사이에서 좋은 밸런스를 잡을 수 있는 일부 데이터 구조 레이아웃 기법들이 존재합니다.

그렇다고는 하나, 성능과 확장성은 안정성 없이는 사용되기 어려운데, 따라서 다음 챕터에서는 겸중에 대해 다릅니다.

If it is not tested, it doesn't work.

Unknown

Chapter 11

Validation

전 처음부터 결함으로 이상하게 동작하는 병렬 프로그램을 몇개 만든 적 있는데, 그건 제가 지난 20년간 많은 수의 병렬 프로그램들을 작성했기 때문일 뿐입니다. 그리고 저는 처음부터 잘 동작할 거라고 생각했지만 실제로는 처음부터 문제가 있어서 저를 바보로 만들었던 많은 병렬 프로그램들을 만든 적 있습니다.

그래서 저는 저는 제 병렬 프로그램들을 위한 검증의 필요성을 강하게 느꼈습니다. 병렬 검증 뒤의 기본적인 속임수는 다른 소프트웨어 검증과 마찬가지로, 컴퓨터는 뭐가 잘못된 것인지 앓을 깨닫는 것입니다. 따라서 컴퓨터가 그걸 당신에게 이야기하도록 하는게 당신이 할 일입니다. 그러므로 이 챕터는 기계를 심문하는 방법에 대한 짧은 수업으로 생각해 볼 수 있습니다.¹

더 긴 수업은 검증에 대한 많은 최신의 책들은 물론, 오래되었지만 상당히 가치있는 것 [Mye79] 으로부터도 얻을 수 있을 겁니다. 검증은 모든 형태의 소프트웨어에 걸쳐 상당히 중요한 주제이고, 따라서 그것 자체만으로도 상당한 공부를 할 가치가 있습니다. 하지만, 이 책은 기본적으로 동시성에 대한 것이므로, 이 챕터는 이 치명적이고 중요한 주제에 대해서 곁핥기보다는 조금 더 다릅니다.

Section 11.1에서는 디버깅의 철학을 소개합니다. Section 11.2 은 트레이싱에 대해 논해보고, discusses tracing, Section 11.3에서는 단정을 논하며, Section 11.4에서는 정적 분석을 논합니다. Section 11.5 은 횡당하리만치 많은 10,000 개의 눈이 코드를 보고 있지는 않을 때에 도움이 될 수 있는 비정규적인 코드 리뷰 접근법을 설명합니다. Section 11.6 은 병렬 소프트웨어의 검증을 위한 확률의 사용을 간단히 알아봅니다. 성능과 확장성은 병렬 프로그래밍에 있어서의 첫번째 요구사항이므로, Section 11.7 에서는 이 주제를 다뤄봅니다. 마지막으로, Section 11.8 에서는 간단한 요약과 피해야 할 통계적 함정들의 짧은 목록을 제공합니다.

¹ 하지만 손가락 고문 도구와 물고문 도구들은 집에 놔두셔도 됩니다. 이 챕터는, 적어도 우리가 알기로는 대부분의 컴퓨터는 고통도 모르고 의사하지도 않는다는 점에서 훨씬 더 세련되고 효과적인 방법들을 다룰겁니다.

But never forget that the two best debugging tools are a solid design and a good night's sleep!

11.1 Introduction

Section 11.1.1 에서는 베그의 근원에 대해서 이야기를 나눠보고, Section 11.1.2 에서는 소프트웨어를 검증할 때 필요한 마음들을 간단히 알아봅니다. Section 11.1.3 에서는 언제 검증을 시작해야 하는지 이야기해 보고, Section 11.1.4 에서 놀랍도록 효과적인 오픈소스 방식의 코드 리뷰와 커뮤니티 테스트에 대해 설명합니다.

11.1.1 Where Do Bugs Come From?

베그들은 개발자들로부터 옵니다. 기본적인 문제는 인류의 뇌는 컴퓨터 소프트웨어와 함께 진화해 오지 않았다는 점입니다. 그보다는, 인류의 뇌는 다른 인류의 뇌와 짐승의 뇌와 함께 진화해 왔습니다. 이런 역사 때문에, 다음과 같은 컴퓨터의 세가지 특성들은 종종 사람의 직관에 충격적으로 다가옵니다.

- 수십년간의 연구가 인공 지능의 제단 위에서 희생되어 왔음에도 불구하고 컴퓨터들은 일반적으로 상식이 부족합니다.
- 컴퓨터들은 일반적으로 사용자의 의도를 이해하지 못하는데, 더 정규적으로 표현하자면, 컴퓨터들은 마음을 이해하는 능력이 떨어집니다.
- 컴퓨터들은 단편적인 계획을 가지고는 어떤 유용한 일도 하지 못해서, 모든 각각의 성립 가능한 시나리오의 모든 자세한 내용들이 설명되어야만 합니다.

앞의 두가지는 쟁점을 갖지 않을게 분명한데, 그런 점들은 여러개의 실패한 제품들, 아마도 가장 유명한 걸로는 Clippy 와 Microsoft Bob 과 같은 예로 설명되기 때문에

입니다. 사용자와 사람의 모습으로 관계를 가지려 시도 함으로써, 이 두개의 제품들은 상식과 마음 이해 능력이 있을 걸로 기대되었는데, 그것들은 결국 이뤄지지 못했음을 그것들 스스로 증명했습니다. 최근들어 스마트폰들에서 나타나기 시작한 소프트웨어 조수들은 아마도 더 나은 결과를 보일 겁니다. 그렇다면 하나, 그것들을 개발하는 일을 하는 개발자들은 여전히 기존 방법으로 개발을 하고 있습니다: 이 조수들은 최종 사용자들에게는 도움을 주겠지만, 그 개발자 스스로에게는 그렇게 많이 도움을 주지 못할 겁니다.

인간의 단편적 계획에 대한 사랑에 대해서는 더 많은 설명이 있어야 하는데, 이게 고전적인 양날의 검이라는 점에서 특히 그렇습니다. 이 단편적 계획에 대한 사랑은 계획을 실제로 수행하는 사람이 (1) 상식과 (2) 그 계획 뒤에 있는 의도에 대한 충분한 이해를 가지고 있을 것이라는 가정 때문입니다. 이 가정은 계획을 짜는 사람과 계획을 수행하는 사람이 똑같은 사람인 경우에 특히나 상식적일 겁니다: 이 경우, 해당 계획은 문제가 생길 때마다 거의 무의식적으로 수정될 겁니다. 따라서, 단편적 계획에 대한 사랑은 인류에 대해서는 잘 동작했는데, 계획할 수 없는 것을 계획하려 시도하는 동안 굽어죽기보다는 음식을 가져올 확률이 큰 무작위적 행동을 취하는게 낫기 때문인 점도 있습니다. 하지만, 이 과거의 단편적 계획의 유용함의 삶은 미래의 컴퓨터에 저장된 프로그램에서의 유용함을 보장하지 않습니다.

더 나아가서, 단편적 계획을 따라야 하는 필요성은 인간의 마음에 중요한 영향을 끼쳤는데, 인류의 역사의 대부분에 있어서, 삶은 어렵고 위험했다는 사실 때문입니다. 높은 확률로 날카로운 이빨과 발톱의 공격을 맞닥뜨릴 수 있는 단편적 계획을 수행하기 위해서는 거의 미친 듯한 수준의 낙관론이 필요함은 전혀 놀랍지 않은 일입니다—그 수준의 낙관론은 사실 대부분의 인간에게 심어져 있죠. 이 미친듯한 수준의 낙관론은 사소한 프로그램의 코딩과 함께 이뤄지는 인터뷰 테크닉의 효율성 (그리고 논쟁)이 증명하듯이, 프로그래밍 능력의 자기 평가로까지 확장됩니다. 사실, 인간의 미친듯하지는 못한 수준의 낙관론에 대한 임상병리학적인 용어는 “임상적 우울증”입니다. 그런 사람은 일반적으로 그들의 일상에 극단적인 기능성 장애를 갖게 되는데, 이는 미친듯한 수준의 낙관론이 평범하고 건강한 삶을 위해 얼마나 중요한지를 반증합니다. 당신이 미친듯 낙관적이지 않다면, 가치가 있지만 어려운 프로젝트를 시작하시는 않을 가능성이 있습니다.²

Quick Quiz 11.1: 컴퓨팅에 있어서 단편적 계획을 따르는게 특히 중요한건 언제인가요? ■

² 이 경험적 법칙에는 유명한 예외들이 존재합니다. 예외들 가운데 하나의 부류는 그들의 우울증으로부터 임시로라도 달아나기 위해 어렵거나 위험에 따르는 프로젝트들을 하는 사람들입니다. 또 다른 부류는 없을 게 없는 사람들입니다: 이 프로젝트는 말 그대로 삶과 죽음의 문제입니다.

중요한 특수 케이스는 가치있지만 그걸 구현하는데 필요한 시간을 정당화 할만큼은 가치있지 않은 프로젝트가 되겠습니다. 이 특수 케이스는 상당히 혼한 케이스이고 이런 경우에 초기부터 생기는 문제는 결정권자들이 그 프로젝트를 정말로 구현하는데 노력할 의지가 없다는 점입니다. 개발자들에게 있어 자연스러운 반응은 그 프로젝트의 시작을 허락받기 위해 비현실적으로 낙관적인 예측을 만드는 것입니다. 만약 그 기관(오픈 소스 또는 독점적인)이 충분히 강하다면, 그 결과로 스케줄이 늦춰지고 예산을 초과하는 일에도 살아남아서 그 프로젝트가 결국 빛을 볼 날이 올겁니다. 하지만, 만약 그 기관이 충분히 강하지 못하고, 그 예측은 실은 쓰레기였다는 것이 분명해졌음에도 결정권자가 그 프로젝트를 취소하지 못한다면 그 프로젝트는 기관을 없애 버릴 수도 있을 겁니다. 이는 또다른 기관이 그 프로젝트를 가져가서 그걸 완료시키거나, 취소하거나, 또는 그 것에 의해 사라져버리거나 하는 결과를 초래할 겁니다. 어떤 프로젝트는 여러 기관을 없애버린 후에야 성공할 수도 있습니다. 어떤 사람은 연쇄 기관 살해마 프로젝트의 최종적 성공을 이끌어낸 기관이 적당한 겸손함을 유지해서 그 기관은 다음 프로젝트로 사라지지 않게 되길 바랄 수도 있습니다.

미친듯한 수준의 낙관론은 중요하지만, 버그의 (그리고 기관의 실패의) 핵심 원인입니다 따라서 질문은 “버그들의 울부짖음을 조용하게 할 수 있기 충분하게 현실성을 가지면서도 커다란 프로젝트를 시작하는데 필요한 낙관을 가질 수 있을까요?”입니다. 다음 섹션에서는 이 수수께끼를 풀어봅니다.

11.1.2 Required Mindset

어떤 검증을 위한 노력을 시작하려 하면, 다음의 정의들을 마음에 새겨둬야 합니다:

1. 버그가 없는 프로그램은 간단한 프로그램들 뿐입니다.
2. 안정적인 프로그램은 알려진 버그들이 없습니다.

이 정의들로부터, 모든 안정적이고 간단하지 않은 프로그램은 알고 있지 않은 버그가 최소 하나는 존재할 것이라는 결론이 논리적으로 따라나옵니다. 따라서, 간단하지 않은 프로그램에서 버그를 찾지 못하는 검증 시도는 그것 자체로 실패한 것입니다. 따라서 좋은 검증은 분해의 연습입니다. 이 말은 당신이 뭔가를 분해하는 것을 즐기는 유형의 사람이라면, 검증은 당시에게 걸맞는 유형의 일이라는 것을 의미합니다.

Quick Quiz 11.2: 다음과 같은 `time` 커맨드의 출력을 처리하는 스크립트를 작성하고 있다고 해봅시다:

```
real    0m0.132s
user    0m0.040s
sys     0m0.008s
```

해당 스크립트는 에러를 처리하고 에러일 수 있는 time 출력을 받았을 경우 적절한 진단 내용을 제공하기 위해 자신에게 주어진 입력을 체크해야 합니다. 싱글 쓰레드 프로그램들로 생성된 time 출력의 사용에 대해 이 프로그램을 테스트하기 위해 어떠한 입력의 테스트를 제공해야 할까요? ■

하지만 당신은 슈퍼 프로그래머여서 당신이 짠 코드는 항상 처음부터 영원히 완벽한 것일 수도 있겠죠. 만약 그렇다면, 축하합니다! 이 챕터를 그냥 넘겨버리셔도 좋아요. 하지만 저는 당신이 제 비판주의를 용서해주시기 바랍니다. 보세요, 저는 처음부터 완벽한 코드를 짤 수 있다고 주장하는 매우 많은 사람들을 만났고, 앞서 언급한 낙관론과 지나친 자신감을 생각해 보면 너무 놀랍지는 않은, 이것을 정말로 어느정도 해낼 수 있는 사람들을 알고 있습니다. 그리고 당신이 정말로 슈퍼 프로그래머라 하더라도, 언젠가는 당신보다 덜 훌륭한 사람의 작업물을 디버깅하고 있는 자신을 보게 될거예요.

그외의 우리들을 위한 한가지 방법은 우리의 일반적인 상태를 미친듯한 낙관과 (물론, 난 그걸 프로그램할 수 있어!) 상당한 비판 (동작할 것처럼 보이긴 해, 하지만 난 거기 어딘가에 더 많은 버그들이 숨어 있을 것인 걸 분명 알아!) 사이에서 교대하는 것입니다. 당신이 뭔가를 분해하는 걸 즐긴다면 이게 도움이 될겁니다. 당신은 그렇지 않다면, 또는 당신의 물건 분해에서의 즐거움은 다른 사람의 물건을 분해하는 것으로 제한되어 있다면, 당신의 코드를 분해하는 것을 좋아하는 누군가를 찾아서 그들이 당신의 테스트를 돋게 하세요.

또 다른 도움이 될 수 있는 마음의 프레임은 다른 사람이 당신의 코드에 있는 버그를 찾는 걸 싫어하는 것입니다. 이 증오는 버그를 다른 누군가가 아니라 당신이 찾을 가능성을 높이기 위한 이유로 당신의 코드를 고문할 동기를 얻는데 도움이 될겁니다.

마지막 마음의 프레임은 다른 누군가의 삶이 당신의 코드가 올바른지에 달려 있을 수 있는 가능성을 고려하는 것입니다. 이것 또한 버그가 어디있는지 밝혀내기 위해 당신의 코드를 고문하기 위한 동기가 될 수 있을겁니다.

이런 다양한 마음의 프레임들은 서로 다른 마음의 프레임을 가진 사람들이 다양한 수준의 낙관을 가지고 프로젝트에 기여할 수 있는 가능성의 문을 열겁니다. 이는 적절히 조직된다면 잘 동작할 수 있습니다.

어떤 사람들에게 격렬한 검증은 Figure 11.1에 보여진 것 같은 고문의 형태로 보여질 수도 있습니다.³ 그

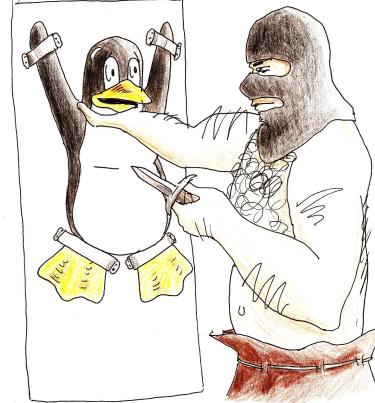


Figure 11.1: Validation and the Geneva Convention

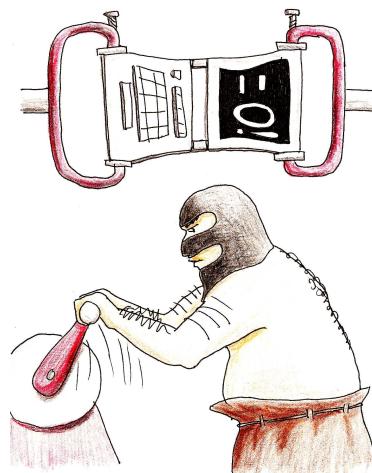


Figure 11.2: Rationalizing Validation

런 사람들은 Figure 11.2에 그려진 것처럼 Tux 그림과 달리, 실제로는 애니메이션이 아닌 물건들을 고문하고 있음을 마음에 되새기는게 도움이 될겁니다.

하지만, 이라고 마는건 프로젝트의 진행 과정 중 정확히 언제 검증이 시작되어야 하는지에 대한 의문을 남겨 두게 되는데, 이 주제는 다음 섹션에서 다루어집니다.

11.1.3 When Should Validation Start?

검증은 프로젝트가 시작된 것과 같은 시점에서부터 시작되어야 합니다.

이 점을 알기 위해, 버그를 추적하는 것은 작은 프로

³ 더 시니컬한 사람은 이런 사람들이 단순히 그 검증을 통해 그들이 고쳐야 할 버그를 찾는 것을 걱정하는 것 뿐 아니라는 질문을

할수도 있겠죠.

그램에서보다 큰 프로그램에서 훨씬 더 어렵단 점을 생각해 보세요. 따라서, 버그들을 추적하는데 필요한 시간과 노력을 최소화하기 위해서, 코드의 작은 단위들을 테스트 해야 합니다. 모든 버그들을 이런 방식으로 찾지는 않겠지만, 상당한 양을 이렇게 찾을 것이고, 이런 방식으로 버그들을 찾아내고 찾아낸 버그들을 고치는 게 훨씬 더 쉬울 겁니다. 이 수준에서의 테스트는 전체 설계에 있는 더 커다란 결점을 알릴 수도 있어서, 설계로 인해 문자 그대로 상당히 고장나 있는 코드를 쓰느라 낭비하는 시간을 최소화 시켜줄 겁니다.

하지만 당신의 설계를 검증하기 전에 왜 코드가 만들 어지기를 기다려야 할까요?⁴ Chapter 3 와 4 를 읽으면 일부 유감스럽게도 흔한 설계상의 잘못들을 피하는데 필요한 정보들을 얻을 수 있을 겁니다만, 당신의 설계를 동료와 토론하거나 심지어 단순히 그걸 글자로 써보는 것만으로도 추가적인 결함을 찾아낼 수 있을 겁니다.

하지만, 설계를 마칠 때까지 검증의 시작을 기다리는 것은 너무나도 대부분의 경우 너무 오래 기다리고 있는 것입니다. 당신의 자연적인 수준의 낙관은 당신이 요구사항들을 완전히 이해하기 전에 설계를 시작하도록 하는 일은 없었을까요? 이 질문에 대한 대답은 거의 항상 “그렇다”입니다. 잘못된 요구사항을 막는 한가지 좋은 방법은 당신의 사용자들에 대해서 아는 것입니다. 정말로 그들을 위해 봉사를 잘 하려 한다면, 그들과 함께 살아야만 합니다.

Quick Quiz 11.3: 지금 저에게 코딩을 시작하기도 전에 검증을 시작하라고 말씀하시는 거예요??? 그건 마치 아무것도 시작하지 않는 훌륭한 방법처럼 들리네요!!!

세상에 없던 새로운 종류의 프로젝트들은 검증에 있어 다른 전략을 필요로 하는데, 예를 들면, 빠른 프로토타입 만들기입니다. 여기서, 처음의 몇개 프로토타입들의 목표는 이 프로젝트가 어떻게 구현되는지를 배우는 것이지, 처음부터 올바른 구현을 만드는게 아닙니다. 하지만, 근본적으로 다른 전략을 취하기보다는, 검증을 누락시켜선 안된다는 것을 명심하는게 중요합니다.

이제 우리는 당신이 프로젝트를 시작할 때부터 검증을 시작해야 하는걸로 합의했으니, 다음 섹션들에서는 그 가치가 이미 증명된 여러가지 검증 기술들과 방법들을 알아보겠습니다.

11.1.4 The Open Source Way

오픈소스 프로그래밍 방법론은 상당히 효율적이며 증명되었고, 격렬한 코드 리뷰와 테스트 제도를 포함하고 있습니다.

⁴ “일단 코드를 짜야 하고, 그러고 나면 생각을 할 수 있다는 보상을 얻는다”는 오래된 말은 통하지 않습니다.

전 개인적으로 오픈소스 커뮤니티의 격렬한 코드리뷰의 효율성을 증명할 수 있습니다. 제가 리눅스 커널을 위해 준비했던 첫번째 패치들은 분산 파일시스템에 관한 것이었는데, 이 파일시스템에서 한 노드의 사용자는 다른 노드의 사용자가 메모리에 매핑해둔 파일의 한 지역에 쓰기를 할 수 있었습니다. 이 경우, 이 파일시스템이 쓰기 작업 중에도 일관성을 유지할 수 있도록 하기 위해, 관련된 페이지들을 해당 매핑으로부터 무효화 시킬 필요가 있습니다. 저는 첫번째 시도를 패치로 코딩했고, 오픈 소스의 “빨리 공유하고, 자주 공유할 것”이라는 행동 원리를 명심하고 있었기에, 그 패치를 공유했습니다. 그러고나서 저는 제가 그걸 어떻게 테스트해야 하나 고민했습니다.

하지만 제가 전체적인 테스트 전략을 결정하기도 전에 저는 제가 공유한 내용에 대해 몇개의 버그를 지적하는 답변을 받았습니다. 전 그 버그들을 고쳐서 패치를 다시 공유했고, 다시 제 테스트 전략에 대해 생각하는 단계로 돌아왔습니다. 하지만, 제가 테스트 코드를 작성할 시간을 갖기도 전에, 저는 더 많은 버그들을 지적하는, 제가 재공유한 패치에 대한 답변을 받았습니다. 이 프로세스는 그 자체로 여러번 반복되었고, 전 제가 그 패치를 정말로 테스트할 기회를 갖기는 했었는지 확신하지 못하겠습니다.

이 경험은 오픈소스가 정말로 말하는 것이 무엇인지 알려줬습니다: 충분한 눈이 있다면, 모든 버그들은 쉽게 파악될 수 있다 [Ray99].

하지만, 당신이 어떤 코드나 패치를 공유하려 한다면, 몇가지 질문을 해보는게 좋을 겁니다:

1. 얼마나 많은 눈이 당신의 코드를 정말로 보게 될까요?
2. 얼마나 많은 눈이 당신의 버그를 정말로 찾기에 충분할 정도로 경험이 많고 현명할까요?
3. 정확히 언제 그들이 당신의 코드를 볼까요?

전 운이 좋았습니다: 제 패치로 제공되는 기능을 필요로 하는 누군가가 거기에 있었는데, 그는 분산 파일시스템들에 긴 경험을 가지고 있었고, 제 패치를 거의 곧바로 보았습니다. 아무도 제 패치를 보지 않았다면, 어떤 리뷰도 없었을 것이고, 따라서 버그를 찾지도 못했을 겁니다. 만약 제 패치를 보는 사람들이 분산 파일시스템들에 대한 경험이 부족했다면, 그들이 모든 버그들을 찾지는 못했을 겁니다. 그들이 코드를 보기 전에 몇달이나 심지어 몇년간을 기다려야 했다면, 전 그 패치가 어떻게 동작해야 했는지도 잊어버려서 그 버그들을 고치기도 훨씬 더 어려웠을 겁니다.

하지만, 우린 오픈소스 개발의 두번째 교리인 격렬한 테스트를 잊지 말아야만 합니다. 예를 들어, 엄청나게

많은 사람들이 리눅스 커널을 테스트 합니다. 어떤 사람들은 패치들을 제출되자마자 테스트하는데, 당신의 것도 그 대상이 될 수 있습니다. 어떤 사람들은 -next 트리를 테스트하는데, 이는 도움이 되지만, 당신이 패치를 작성한 시점부터 그에 -next 트리에 들어가게 되는 시점 까지는 몇주에서 몇달까지의 지연이 있을 수도 있는데, 이는 그 패치가 당신의 기억 속에 여전히 아주 생생하게 남아있지는 않을 가능성이 있는 시간입니다. 어떤 사람들은 메인테이너 트리들을 테스트하는데, 이때에도 비슷한 지연시간이 있곤 합니다.

일부 적은 사람들은 메인라인 또는 마스터 소스 트리(리눅스 커널의 경우라면 Linus 의 트리)에 커밋되기 전까지는 코드를 테스트 하지 않습니다. 당신의 메인테이너가 테스트 되기 전까지는 당신의 패치를 받아주지 않는다면, 이는 당신에게 데드락 상황을 보이게 합니다: 당신의 패치는 테스트 되기 전까지는 받아들여지지 않을 텐데, 이 패치는 또한 받아들여지기 전까지는 테스트 되지 않을 겁니다. 더도 아니고 덜도 아니고, 많은 사람들과 단체들이 코드가 리눅스 배포판에 들어가기 전까지는 그 코드를 테스트 하지 않기 때문에, 메인라인 코드를 테스트하는 사람들은 여전히 상대적으로 공격적입니다.

그리고 누군가가 당신의 패치를 테스트한다고 하더라도, 그들이 당신의 버그들을 찾아내는데 필요한 하드웨어와 소프트웨어 구성과 워크로드를 수행할 것인지에 대해서는 보장이 없습니다.

따라서, 오픈소스 프로젝트를 위한 코드를 작성할 때 라 할지라도, 당신은 당신 스스로의 테스트 장비를 개발하고 수행할 준비를 할 필요가 있습니다. 테스트 개발은 실제 가치에 비해 덜 중요하게 평가되었지만 매우 중요한 기술이므로, 당신이 사용할 수 있는, 이미 존재하는 테스트 장비들의 장점을 모두 취할 수 있도록 하십시오. 테스트 개발이 그렇게 중요하므로, 우리는 그에 대한 더이상의 토론은 그 주제 전용의 책들에게 떠넘기도록 하겠습니다. 따라서 다음의 섹션들은 당신이 이미 좋은 테스트 장비를 갖추고 있다는 가정 하에 당신의 코드 안의 버그들을 찾는 방법들을 다루겠습니다.

11.2 Tracing

다른 모든게 실패한다면, `printf()` 를 추가하세요! 또는, 사용자 모드 C-언어 어플리케이션을 작업 중이라면 `printf()` 를요.

논리적 근거는 간단합니다: 어떻게 실행이 코드의 특정 지점까지 가게 되었는지 알아낼 수가 없다면, 무슨 일이 벌어졌는지 알아내기 위해 코드의 앞부분에 프린트문을 여기저기 집어넣으세요. 비슷한 효과를 (유저 어플리케이션을 위한) `gdb` 나 (리눅스 커널 디버깅을 위한) `kgdb` 와 같은 디버거를 이용해서 더 편리하고 유연

성 있게 얻을 수도 있습니다. 훨씬 더 세련된 도구들도 존재하는데, 그 중 최신의 일부는 문제가 발생한 시점부터 뒤로 시간을 되돌릴 수 있는 기능을 제공합니다.

이런 간단한 테스트 도구들은 모두 가치가 있는데, 일반적인 시스템들이 64K 보다 큰 메모리와 4MHz 보다 빠른 속도로 동작하는 지금에 있어선 특히 그렇습니다. 이런 도구들에 대해서는 더 많은 글들이 있으므로, 이 챕터는 그에 대해 조금만 더 이야기 하겠습니다.

하지만, 이런 도구들은 모두, 디버깅하려는 일이 고성능의 병렬 알고리즘의 빠른 수행경로에 뭐가 잘못되어 있는지를 알려고 하는 것일 경우에 심각한 단점을 가지고 있는데, 달리 말해 대부분의 경우 큰 오버헤드를 갖습니다. 이런 목적을 위한 특수한 추적 기술들이 있는데, 일반적으로 실행시간 데이터 수집의 오버헤드를 최소화 하기 위해 데이터 소유권 테크닉 (Chapter 8을 참고하세요) 을 사용합니다. 리눅스 커널에서의 한가지 예는 데이터가 극단적으로 낮은 오버헤드만을 가지고 수집될 수 있도록 하기 위해 CPU 별 버퍼를 사용하는 “trace events” [Ros10b, Ros10c, Ros10d, Ros10a] 입니다. 그렇다 하더라도, 추적 기능을 활성화 시키는 것은 어떤 경우에는 버그를 숨기기 충분할 만큼 타이밍을 바꿀 수 있어서, Section 11.6 과 Section 11.6.4 에서 특별히 다루어질 *heisenbugs* 를 초래할 수 있습니다.

유저스페이스 코드에는, 여러분을 도울 수많은 도구들이 있습니다. 이를 알아보는데 있어 시작하기 좋은 것은 Brendan Gregg 의 블로그입니다.⁵

Heisenbug 를 막는다 하더라도, 다른 위험들이 기다리고 있습니다. 예를 들어, 비록 기계는 모든 것을 알고 있지만, 그것이 알고 있는 것은 거의 항상 당신의 머리가 쥐고 있을 수 있는 것보다 많은 정보입니다. 이런 이유로, 높은 품질의 테스트 장비들은 일반적으로 대량의 출력을 분석할 수 있도록 세련된 스크립트들을 함께 제공합니다. 하지만 조심하세요—스크립트들은 놀라운 일을 알려줘야만 하는 건 아닙니다. 제 `rcutorture` 스크립트는 그런 점에서의 한 경우입니다: 그 스크립트들의 초기 버전들은 RCU grace period 들이 불명확하게 늦춰지며 동작 (stall) 함에도 문제가 없다고 판단하고는 했습니다. 이는 물론 해당 스크립트들이 RCU grace-period stall 문제를 파악할 수 있도록 수정되었습니다만, 이로 인해 해당 스크립트는 제가 파악할 수 있게 하자고 생각하는 문제들만을 파악한다는 사실은 바뀌지 않습니다. 그 스크립트들은 유용합니다만, 가끔 `rcutorture` 출력을 일일이 읽는 행위를 대체할 수는 없습니다. 하지만 여러분이 훌륭한 설계를 갖지 않는다면, 여러분은 여러분의 스크립트가 무엇을 체크해야 하는지도 모를 수 있음을 알아두시기 바랍니다!

추적 기능과 특히 `printf()` 호출에 있어서의 또 다른 문제는 그 오버헤드가 제품에 사용되기에 너무 크

⁵ <http://www.brendangregg.com/blog/>

다는 것입니다. 그런 경우들 중 일부에 있어서는, 단정문들이 도움이 될 수 있습니다.

11.3 Assertions

단정문들은 일반적으로 다음과 같은 식으로 구현됩니다:

```
1 if (something_bad_is_happening())
2   complain();
```

이 패턴은 종종 C 전처리기 매크로나 프로그래밍 언어의 내재 기능으로 캡슐화 되어지는데, 예를 들어 리눅스 커널의 경우, 이는 `WARN_ON(something_bad_is_happening())` 으로 표현될 겁니다. 물론, 만약 `something_bad_is_happening()` 이 상당히 자주 벌어진다면 그로 말미암은 출력은 다른 문제들의 보고를 이해하기 어렵게 만들 수 있는데, 이런 경우에는 `WARN_ON_ONCE(something_bad_is_happening())` 이 더 적절할 겁니다.

Quick Quiz 11.4: `WARN_ON_ONCE()` 는 어떻게 구현할 수 있을까요? ■

병렬 코드에서, 일어날 수도 있는 한가지 매우 나쁜 일은 특정한 락이 잡힌 상태에서 호출될 것으로 기대되는 어떤 함수가 그 락이 잡히지 않은 채로 호출되는 경우입니다. 그런 함수들은 어떤 경우에는 “호출자는 반드시 이 함수를 호출할 때 `foo_lock` 을 잡아야만 함”과 같은 이야기를 하는 헤더 코멘트를 가질 수도 있습니다만, 그런 코멘트는 누군가가 그걸 정말로 읽지 않는다면 어떤 좋은 일도 하지 않습니다. `lock_is_held(&foo_lock)` 과 같이 실행 가능한 이야기가 훨씬 많은 무게를 갖습니다.

리눅스 커널의 lockdep 기능 [Cor06a, Ros11] 은 이 과정을 좀 더 크게 취하는데, 잠재적인 데드락을 보고하는 것 뿐만 아니라 함수들이 올바른 락들을 잡았는지 검증할 수 있게 도와줍니다. 물론, 이 추가적인 기능은 상당한 오버헤드를 만들어내는데, 그로인해 lockdep은 제품에서의 사용에는 적합하지는 않습니다.

그래서 검사가 필요하지만 실행시간에서의 검사로 인한 오버헤드는 용인될 수 없는 경우들에는 무엇을 해 볼 수 있을까요? 한가지 방법은 정적 분석인데, 다음 섹션에서 이에 대해 논해 보겠습니다.

11.4 Static Analysis

정적 분석인 첫번째 프로그램이 두번째 프로그램을 입력으로 받아서 두번째 프로그램 안에 있는 에러들과 취약점들을 보고해 주는 검증 테크닉입니다. 흥미롭게도, 거의 모든 프로그램들이 컴파일러와 인터프리터들을

통해 정적 분석을 합니다. 이런 도구들은 물론 완벽과는 거리가 멍니다만, 그것들의 에러를 찾아내는 기능은 과거의 수십년간 몹시 개선되었는데, 일부분은 그 분석을 진행할 메모리의 크기가 64K 바이트보다 훨씬 더 커진 것도 한 이유입니다.

원래의 UNIX `lint` 도구 [Joh77] 는 상당히 유용합니다만, 그것의 기능들 가운데 대부분의 것들은 C 컴파일러 안에 포함되었습니다. 더도 아니고 덜도 아니고 `lint` 와 유사한, 개발되고 있고 사용되고 있는 도구들이 오늘날 존재합니다.

`Sparse` 정적 분석 도구 [Cor04b] 는 리눅스 커널의 높은 수준의 문제들을 찾아내는데, 다음과 같은 문제들을 포함합니다:

1. 유저 스페이스 구조체로의 포인터의 잘못된 사용.
2. 너무 긴 상수로부터의 값 할당.
3. 텅 빈 `switch` 문.
4. 잘못 매치된 락 획득과 해제 도구들.
5. Per-CPU 도구들의 잘못된 사용.
6. RCU 포인터가 아닌 곳에서의 RCU 사용과 그 반대 경우.

컴파일러들이 자체적인 정적 분석 기능들을 계속해서 늘려갈 것 같긴 하지만, `sparse` 정적 분석 도구는 컴파일러 밖에서의 정적 분석의 효과를 보여주고 있는데, 특히 어플리케이션에 특정한 버그들을 찾는데 그려합니다.

11.5 Code Review

다양한 코드 리뷰 활동들은 정적 분석의 특수한 경우들입니다만, 사람이 분석을 한다는 차이가 있습니다. 이 섹션은 검사, 가상 리허설, 그리고 자가 검사에 대해서 다룹니다.

11.5.1 Inspection

전통적으로, 정식적인 코드 검사는 정식적으로 정의된 역할들의 얼굴을 맞대는 모임으로 이루어졌습니다: 중재자, 개발자, 그리고 한두명의 참가자. 여기서 개발자는 코드를 읽어내려가면서 그 코드가 하는 일이 무엇이며 왜 동작하는지 설명을 합니다. 한두명의 참가자들은 질문을 하고 문제를 제기하며, 중재자가 하는 일은 모든 충돌을 처리하고 기록을 하는 것입니다. 이 과정은 버그를 찾아내는 데에 상당히 효과적일 수 있는데, 모든 참가자가 이 코드에 친숙하다면 특히 그려합니다.

하지만, 이 얼굴을 맞대는 정식적 과정은 글로벌한 리눅스 커널 커뮤니티에서는 잘 동작하지 못할 수 있습니다. IRC 세션을 사용하면 어쩌면 잘 동작할 수도 있겠지만요. 대신에, 개인들은 코드를 개별적으로 리뷰하고 이메일이나 IRC를 통해 코멘트를 제공합니다. 기록을 하는 행위는 이메일 기록이나 IRC 로그를 통해 제공되며, 중재자들은 그들의 서비스들을 적절하게 자발적으로 봉사합니다. 간헐적인 격론을 주고 받으면서, 이 프로세스는 합리적인 수준으로 잘 동작하는데, 모든 참가자들이 처리해야 하는 코드에 모두 친숙하다면 더욱 그러합니다.⁶

리눅스 커널 커뮤니티의 리뷰 프로세스는 개선될 여지도 많습니다:

1. 가끔은 효과적인 리뷰를 하기에 충분한 시간과 전문성을 가진 사람이 부족할 때가 있습니다.
2. 모든 리뷰 과정의 토론들이 기록된다고는 하지만, 이 기록들은 통찰이 잊혀지거나 사람들이 그 토론을 열어보는데에 종종 실패하는 것과 같은 형태로 “누실” 되기도 합니다.
3. 가끔은 격렬한 토론으로 인한 싸움이 벌어졌을 때에 이를 해결하기가 어려울 수도 있는데, 싸우는 사람들이 합치될 수 없는 목표, 경험, 그리고 어휘를 사용할 때에 특히 그렇습니다.

따라서, 리뷰를 할 때에는 커밋 로그, 버그 레포트, 그리고 LWN 기사에 있는 적절한 문서들을 참고하는게 좋습니다.

11.5.2 Walkthroughs

전통적인 walkthrough (가상 리허설)는 정식적인 검사와 비슷하지만, 사람들이 코드를 가지고 특정 테스트 케이스에 대해서 “컴퓨터인 척 한다”는 점이 다릅니다. 일반적인 walkthrough 팀은 중재자, 비서 (찾아낸 버그를 기록합니다), 테스트 전문가 (테스트 케이스를 만들 어냅니다), 그리고 한두명의 추가적인 사람들을 포함합니다. 이것들은 굉장히 시간을 소모하지만 상당히 효과적일 수 있습니다.

제가 정식적인 walkthrough에 참여한지 수십년이 되었고, 저는 오늘날의 walkthrough는 단일단계 디버거들을 사용할 수도 있을 거라고 생각합니다. 어떤 사람은 다음과 같이 특히나 가학적인 과정을 떠올릴 수 있을겁니다:

⁶ 그렇다고 하나, 전통적인 정식적 검사에 비한 리눅스 커널 커뮤니티 방법의 장점 가운데 하나는 코드에 친숙하지 않은, 따라서 코드에 익숙한 사람들에게 품어져 있는 옳지 않은 가정에 의해 눈이 가려지지 않은 사람들의 기여의 가능성성이 훨씬 더 높다는 것입니다.

1. 테스터는 테스트 케이스를 제공합니다.
2. 중재자는 특정된 테스트 케이스를 입력으로 해서 디버거 위에서 코드를 동작시킵니다.
3. 각각의 명령문이 실행되기 전에, 개발자는 해당 명령문의 결과가 무엇일지를 예상하고 왜 그 결과가 옳은지를 설명합니다.
4. 만약 결과가 개발자에 의해 예상되었던 것과 다르다면, 이는 잠재적인 버그의 증거로 택해집니다.
5. 병렬 코드에서는 “동시성 업자 (concurrency shark)”가 어떤 코드가 이 코드와 동시에 실행될 수 있을 것인지, 그리고 왜 그런 동시성이 해롭지 않은지 질문합니다.

가학적입니다, 분명히. 효과적일까요? 아마도요. 참가자들이 요구사항, 소프트웨어 도구들, 데이터 구조들, 그리고 알고리즘들에 대해 잘 이해하고 있다면, 그 walkthrough는 상당히 효과적일 수 있습니다. 만약 그렇지 않다면, walkthrough는 시간 낭비인 경우가 많습니다.

11.5.3 Self-Inspection

개발자들이 모두 자신의 코드를 검사하는데 그렇게 효과적이지는 않은 게 일반적이지만, 합리적인 대안이 없는 상황들도 많이 존재합니다. 예를 들어, 해당 개발자가 그 코드를 볼 수 있도록 허가된 유일한 사람일 수 있고, 다른 능력 있는 개발자들은 모두 너무 바빠서 일수도 있고, 또는 문제의 해당 코드가 충분히 기묘하게 생겨서 프로토타입을 선보이기 전까지는 어떤 사람도 그걸 심각하게 바라보도록 설득할수가 없어서일 수도 있습니다. 이런 경우들에, 다음과 같은 방법은 상당히 도움이 되는데, 특히 복잡한 병렬 코드에서 그렇습니다:

1. 요구사항, 데이터 구조를 위한 다이어그램, 그리고 설계 선택 사항들에 대한 합리적 이유등을 가지고 설계 문서를 작성합니다.
2. 전문가에게 자문을 구하고 필요하다면 설계 문서를 업데이트 합니다.
3. 코드를 종이 위에 펜으로 써가면서 에러들을 고칩니다. 앞서 존재한 거의 동일한 코드 시퀀스를 참고하려는 유혹을 견뎌내고, 작성한 코드의 사본을 만듭니다.
4. 거기에 에러가 있었다면, 깨끗한 종이에 그 코드를 똑같이 쓰면서 에러들을 고쳐갑니다. 마지막 두 복사본이 동일할 때까지 이를 반복합니다.

5. 모든 분명치 않은 코드에 대해 정확성을 증명합니다.
6. 가능하다면, 코드 조각들을 바닥에서부터 테스트 합니다.
7. 모든 코드가 합쳐지면, 기능성과 스트레스 테스트를 전부 합니다.
8. 코드가 모든 테스트를 통과한다면, 코드 레벨 문서를 작성하는데, 이는 앞서 이야기한 설계 문서의 확장판이 될수도 있습니다.

제가 새로운 RCU 코드를 위해 성실하게 이 과정을 따라갔을 때, 과정의 마지막에 이르러서는 몇개의 버그들만이 존재했습니다. 몇개의 두드러지는 (그리고 당황스러운) 예외가 있지만 [McK11a], 전 남들보다 앞서서 이런 버그들을 잡아내고는 합니다. 그렇다고는 하나, 리눅스 커널 사용자의 수와 다양성이 증가함에 따라 이는 점점 어려워지고 있습니다.

Quick Quiz 11.5: 어떤 사람이 존재하는 코드를 종이에 펜으로 사본을 만들려 하겠어요??? 그건 그냥 필사 과정에서의 에러의 가능성만 증가시키는 거 아닌가요?

Quick Quiz 11.6: 이 과정은 우스꽝스러울 정도로 지나치게 공업화 되어 있어요! 어떻게 당신은 합리적인 양의 소프트웨어들이 이런 방식으로 작성되었을 거라고 생각할 수 있는거죠? ■

앞의 방법은 새로운 코드에 있어서는 잘 동작하되만, 이미 작성한 코드에 대해서 검사가 필요하다면 어떻게 해야 할까요? 물론 기존의 코드에 앞의 방법을 당신이 투입된 특수한 경우 [FPB79]에 대해 적용할 수 있겠습니다만, 덜 절망적인 상황에서는 다음과 같은 방법도 도움이 될 수 있을 겁니다:

1. 당시이 가장 좋아하는 문서화 도구 (L^AT_EX, HTML, OpenOffice, 또는 ASCII)를 사용해서 문제가 되는 코드의 설계를 높은 수준에서 묘사하세요. 데이터 구조들과 이 구조들이 어떻게 업데이트 되는지를 묘사하기 위해 많은 다이어그램들을 사용하세요.
2. 해당 코드의 복사본을 만들고 모든 코멘트들을 지워버리세요.
3. 코드가 무슨 일을 하는지를 한줄 한줄씩 문서화하세요.
4. 발견되는 대로 버그들을 고치세요.

이게 잘 동작하는 이유는 코드를 자세하게 설명하는 것은 버그를 찾아내는 훌륭한 방법이기 때문입니다 [Mye79]. 이 두번째 방법은 어떤 다른 사람의 코드를

이해하는데에도 좋은 방법이긴 합니다만, 많은 경우에 첫번째 단계만으로도 충분합니다.

다른 사람에 의한 리뷰와 검사가 더 효율적이고 효과적이긴 하지만, 앞의 방법들은 어떤 이유가 되었든 다른 사람들이 관여할 수가 없는 상황에서 상당히 도움이 됩니다.

이 시점에서, 당신은 이 모든 지루한 종이와 함께 하는 일들을 하지 않고서 병렬 코드를 작성하는 방법에 대해 궁금할 겁니다. 여기 그걸 달성하기 위한, 오랜 시간이 증명한 몇가지 방법이 있습니다:

1. 사용 가능한 병렬 라이브러리 함수의 사용을 통해 확장되는 순차적 프로그램을 작성하세요.
2. 맵리듀스, BOINC 또는 웹 어플리케이션서버와 같은 병렬 프레임워크를 위한 순차적 플러그인들을 작성하세요.
3. 문제가 완전히 분할 가능한 병렬 설계와 같은 멀진 일을 하고, 상호간 통신 없이 병렬로 수행되는 순차적 프로그램(들)을 구현하세요.
4. 어플리케이션 영역들 중 도구들이 자동으로 문제를 분해하고 병렬화 할 수 있는, (선형 대수와 같은) 하나만하세요.
5. 극단적으로 올바른 방법으로만 병렬 프로그래밍 도구들을 사용해서 그 결과 작성된 코드가 올바르단 것이 쉽게 보일 수 있게하세요. 하지만 조심하세요: 항상 더 좋은 성능과 확장성을 위해서 그 규칙들을 “아주 조금만” 깨고 싶은 욕구가 생길겁니다. 그 규칙들을 깨는 건 일반적으로 문제를 야기합니다. 이 섹션에서 설명한 종이를 가지고 하는 일들을 조심스럽게 하지 않았다면 말이죠.

하지만 슬픈 사실은 당신이 이 종이를 가지고 하는 일을 했거나 종이를 가지고 하는 일을 회피할 수 있는 더 또는 덜 안전한 방법들을 사용했다고 하더라도 버그는 존재할 거라는 겁니다. 모든 걸 제쳐두고, 더 많은 사용자들과 훨씬 많은 사용자의 다양성은 더 많은 버그들을 더 빨리 노출시킬 것인데, 그런 사용자들이 최초의 개발자들이 고려하지 못했던 일들을 하게 되면 특히 그렇습니다. 다음 섹션은 병렬 소프트웨어를 검증하면 너무도 자주 발생하는 확률적인 버그들을 다루는 방법에 대해 설명하겠습니다.

11.6 Probability and Heisenbugs

그러니까, 당신의 병렬 프로그램은 실패합니다. 가끔요.

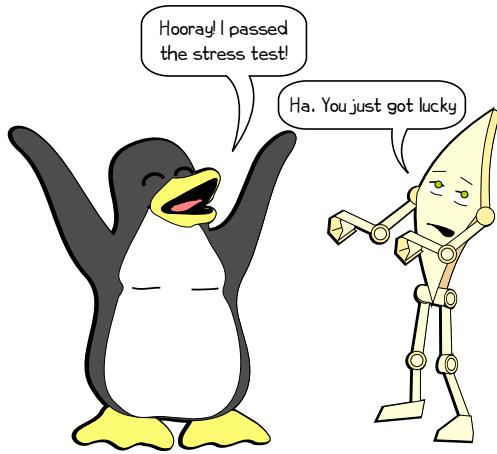


Figure 11.3: Passed on Merits? Or Dumb Luck?

하지만 당신은 그 문제들을 찾아내기 위해 앞의 섹션에서 얻은 방법들을 사용했고 이제 그것들을 수정했어요! 축하합니다!!!

이제 질문은, 한편으로는 그 버그가 발생할 확률만을 줄였거나, 관련된 여러개의 버그들 중 하나만을 고쳤거나, 또는 관계없고 효과없는 변경만을 가한게 아니라 정말로 그 버그를 고쳤다는 점을 분명히 하기 위해서 얼마나 많은 테스트를 해야 하는가입니다. 짧게 말해서, Figure 11.3로 암시되는 영원한 질문에 대한 답은 뭘까요?

불행히도, 정직한 답은 절대적인 정확성을 얻기 위해서는 무한한 테스트가 필요하다는 것입니다.

Quick Quiz 11.7: 당신의 쓰레기 봉투에 굉장히 많은 시스템들이 있다고 생각해 봅시다. 예를 들어, 현재의 클라우드 가격 정책대로라면, 당신은 합리적으로 낮은 가격에 수많은 CPU 시간을 구입할 수 있습니다. 모든 실용적인 목적에 있어서의 정확성을 충분히 가져가기 위해서 이 전략을 사용하지 않나요? ■

하지만 절대적인 정확성을 포기하고 높은 가능성에 치중하도록 하려 한다고 생각해 봅시다. 그렇다면 우린 이 문제를 해결하는데에 강력한 통계적 도구들을 사용할 수 있습니다. 하지만, 이 섹션은 간단한 통계 도구들에 집중해 봅니다. 이런 도구들은 굉장히 도움이 됩니다만, 이 섹션을 읽는 것이 훌륭한 통계 수업들을 듣는 것을 대체할 수는 없다는 점을 알아두시기 바랍니다.⁷

간단한 통계적 도구들을 알아보는 걸 시작하기 위해, 우리가 별개적인 테스트를 할 것인지 연속적인 테스트

⁷ 전 그려기를 강력하게 추천합니다. 제가 들었던 몇 개의 통계 수업들은 제가 그것들을 공부하기 위해 쏟았던 시간에 비례해서 가치를 제공했습니다.

를 할 것인지 결정해야 합니다. 별개적 테스트는 잘 정의된 개별 테스트 수행들을 갖춥니다. 예를 들어, 리눅스 커널 패치의 부팅 테스트는 별개적 테스트의 한 예입니다. 커널을 부팅해 보고, 그게 부팅되거나 아니거나입니다. 비록 당신은 당신의 커널을 부팅 테스트 하는데에 한시간을 소비할 수도 있긴 하지만, 대부분의 경우에 있어서 커널을 부팅하기 위해 시도한 시도의 횟수와 부팅이 성공한 횟수가 테스트에 소비한 시간의 길이보다 더 흥미로운 것이 될 겁니다. 기능 테스트는 별개적이게 되는 경향이 있습니다.

한편으로는, 만약 제 패치가 RCU에 관련되어 있다면, 저는 RCU를 테스트하는 커널 모듈인 `rcutorture`를 돌려볼겁니다. 하나의 별개 테스트의 성공적인 마지막에는 로그인 프롬프트 시그널을 띠워주는 커널 부팅과 달리, `rcutorture`는 커널이 크래시 나거나 당신이 멈추라고 하기 전까지는 RCU를 고문하는 것을 지속할 겁니다. 따라서 `rcutorture` 테스트에 걸리는 시간이 (일반적으로) 당신이 그걸 시작하고 멈춘 시간보다 더 중요하게 됩니다. 따라서, `rcutorture`는 많은 스트레스 테스트들을 포함하는 카테고리인 연속적 테스트의 한 예입니다.

별개 테스트와 연속적 테스트를 운영하는 통계는 몇몇 부분에서 다릅니다. 하지만, 별개 테스트들을 위한 통계는 연속적 테스트를 위한 것보다 더 간단하고 친숙하며, 뿐만 아니라 별개 테스트들을 위한 통계는 종종 (약간의 정확성 손실과 함께) 연속적 테스트를 위한 서비스에 사용되기도 합니다. 따라서 우리는 별개 테스트들부터 시작하겠습니다.

11.6.1 Statistics for Discrete Testing

버그가 한번의 프로그램 수행 사이에 10%의 발생 확률을 가지고 있고 우리는 프로그램을 다섯번 수행한다고 생각해 봅시다. 최소 한번의 수행은 실패할 확률을 어떻게 계산하면 될까요? 그런 한가지 방법은 다음과 같습니다:

1. 하나의 수행이 성공할 확률을 계산하는데, 이건 90%입니다.
2. 모든 다섯번의 수행이 성공할 확률을 계산하는데, 이는 0.9의 5승이어서, 약 59%입니다.
3. 두개의 가능성만이 존재합니다: 다섯번의 수행이 모두 성공하거나, 최소 한번은 실패하거나. 따라서, 최소 한번은 실패할 확률은 100%에서 59%를 제외한 나머지인 41%입니다.

하지만, 많은 사람들이 여러 단계보다는 하나의 공식을 사용하는게 더 쉽다고 생각하므로, 설령 당신은 앞의 단계들이 더 편하다고 해도, 한번 해봅시다! 공식을 더 선호하는 사람들을 위해, 한번의 실패할 확률을 f 라고

해봅시다. 그럼 한번의 성공을 할 확률은 $1 - f$ 이고, 따라서 n 개의 테스트들이 모두 성공할 확률은:

$$S_n = (1 - f)^n \quad (11.1)$$

실패할 확률은 $1 - S_n$, 또는:

$$F_n = 1 - (1 - f)^n \quad (11.2)$$

Quick Quiz 11.8: 뭐라구요??? 제가 앞의, 각각 10% 실패 확률을 갖는 다섯번의 테스트의 예를 이 공식에 집어넣어보면 59,050%를 얻게 되는데 이건 말이 안되잖아요!!! ■

따라서 특정 테스트가 10%의 확률로 실패하고 있다 고 해봅시다. 당신의 수정이 정말로 일을 개선시켰다고 99% 확신할 수 있으려면 얼마나 많은 테스트를 수행시 켜 봐야 할까요?

이걸 다르게 질문해 보면 “실패를 일으킬 확률을 99% 까지 올리려면 얼마나 많은 테스트를 돌려보아야 할까요?”입니다. 일단, 최소 하나의 실패를 볼 수 있는 확률이 99% 까지 되도록 테스트를 충분히 많이 돌린다면, 그리고 거기에 실패가 한번도 없다면, 이는 단지 1% 확률의 행운 덕입니다. 그리고 우리가 Equation 11.2 에 $f = 0.1$ 을 집어넣고 n 을 변경시켜보면, 원래의 10% 라는 테스트당 실패 확률에 대해서 43번의 수행이 99.03% 의 최소 한번은 테스트가 실패할 확률을 가져오고, 44 번의 수행은 99.03% 의 최소 한번은 실패할 확률을 가져옵니다. 따라서 우리가 수정을 한 후에 44번 테스트를 진행하고 실패를 한번도 보지 못한다면, 우리의 수정이 정말로 실제 개선을 만들어냈을 확률이 99% 인 것입니다.

하지만 계속해서 숫자들을 Equation 11.2 에 집어넣 어보는 것은 지켜울 수 있으므로, n 을 풀어봅시다:

$$F_n = 1 - (1 - f)^n \quad (11.3)$$

$$1 - F_n = (1 - f)^n \quad (11.4)$$

$$\log(1 - F_n) = n \log(1 - f) \quad (11.5)$$

마지막으로 요구되는 테스트의 횟수는 다음과 같 이 구해집니다:

$$n = \frac{\log(1 - F_n)}{\log(1 - f)} \quad (11.6)$$

Equation 11.6 에 $f = 0.1$ 와 $F_n = 0.99$ 를 대입해 보면 43.7 이 나오는데, 이는 우리의 수정이 실제 개선을 만들어냈음을 99% 확신하려면 44번의 연속적인 성공적 테스트가 필요함을 의미합니다. 이는 안심스럽게도 앞의 방법에서 얻어진 숫자와 맞아떨어집니다.

Quick Quiz 11.9: Equation 11.6 에서, 로그의 밑은 10 인가요, 2인가요, 또는 e 인가요? ■

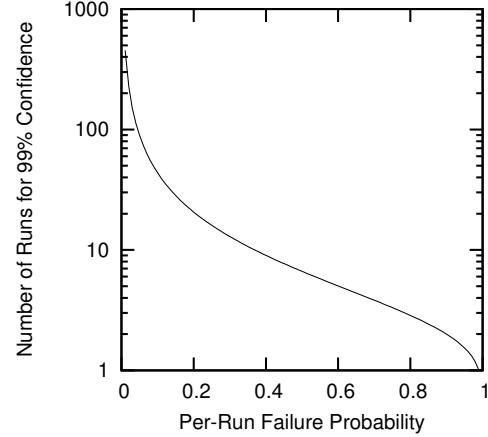


Figure 11.4: Number of Tests Required for 99 Percent Confidence Given Failure Rate

Figure 11.4 는 이 함수를 그래프로 그려보입니다. 놀 랍지 않게, 각각의 테스트가 덜 빈번하게 실패할수록, 그 버그가 고쳐졌음을 99% 확신하기 위해선 더 많은 테스트 수행이 필요합니다. 테스트를 실패하게 하는 버그가 오직 1%의 확률만으로 나타난다면, 애끓는 심정의 458 번의 테스트 수행이 필요합니다. 실패 확률이 줄어들수록, 필요한 테스트 수행 횟수가 증가해서, 실패 확률을 0으로 하려면 무한번의 테스트 수행이 필요해집니다.

이 이야기의 교훈은 가끔씩만 일어나는 버그를 발견했다면, 실패 확률을 훨씬 높이도록 잘 타겟팅 된 테스트를 사용한다면 테스트 업무가 훨씬 쉬워질 것이라고 합니다. 예를 들어, 타겟이 된 테스트가 실패 확률을 1% 에서 30%로 늘린다면, 99% 확신을 갖기 위해 필요한 테스트 수행 횟수는 458 회에서 단지 13회까지로 떨어질 겁니다.

하지만 이 열세번의 테스트 수행은 당신의 수정 사항이 “어떤 개선”을 만들어냈다는 데에 99%의 확신만을 제공합니다. 당신이 원하는건 그게 아니라 당신의 수정 사항이 실패 확률을 10배 가량 줄인 것이라는 99%의 확신이라고 생각해 봅시다. 얼마나 많은 실패하지 않는 테스트 수행이 필요할까요?

30% 실패 확률에서 10배 개선된다면 3%의 실패 확률입니다. 이 숫자들을 Equation 11.6 에 집어넣으면 다음과 같은 결과가 나옵니다:

$$n = \frac{\log(1 - 0.99)}{\log(1 - 0.03)} = 151.2 \quad (11.7)$$

따라서 우리의 10배 개선을 위해서는 대략 10배 이상의 테스트가 필요합니다. 확신은 불가능하고, 높은 확률은 상당히 비쌉니다. 테스트들이 수행은 더 빠르게 되고

실패는 더 잘 일어나도록 분명하게 만드는 것이 높은 수준의 안정성을 갖는 소프트웨어를 개발하는데 있어 핵심의 기술들입니다. 이런 기술들은 Section 11.6.4에서 다루겠습니다.

11.6.2 Abusing Statistics for Discrete Testing

하지만 당신이 열번 수행하면 세번은 실패하는 연속적인 테스트를 가지고 있고, 그 실패를 유발했다고 당신이 생각하는 버그를 고쳤다고 생각해 봅시다. 당신이 실패의 가능성을 줄였다는 점을 99% 확신하려면 얼마나 많은 횟수의 실패 없는 테스트를 수행해야 할까요?

통계를 극단적으로 해치지 않으면서, 우리는 간단히 한시간동안의 수행을 30%의 실패 가능성을 가진 별개의 테스트로 재정의 할 수 있을겁니다. 그렇다면 앞의 섹션에서의 결과는 테스트가 13시간동안 실패 없이 돌아간다면, 우리의 수정사항은 실제로 프로그램의 안정성을 개선했을 가능성이 99%라고 이야기하게 됩니다.

교리적인 통계학자는 이런 접근을 허락하지 않을겁니다만, 슬픈 사실은 이런 종류의 통계 방법론의 오용으로 인해 발생하는 에러들은 당신의 프로그램의 실패 확률의 측정에서 피할 수 없는 에러들보다는 훨씬 작다는 것입니다. 더도 아니고 덜도 아니고, 다음 섹션에서는 약간 덜 사기적인 방법을 설명합니다.

11.6.3 Statistics for Continuous Testing

실패 확률을 위한 기본적인 공식은 Poisson 분포입니다:

$$F_m = \frac{\lambda^m}{m!} e^{-\lambda} \quad (11.8)$$

여기서 F_m 은 테스트에서 m 실패의 확률이고 λ 는 단위시간 동안 예상되는 실패 비율입니다. 더 엄격한 유도는 더 나은 확률에 대한 교재들에서 찾아볼 수 있을텐데, 예를 들어 Feller의 고전인 “An Introduction to Probability Theory and Its Application” [Fel50]가 있으며, 더 직관적인 접근법은 이 책의 첫번째 수정본 [McK14a]에서 찾아볼 수 있을겁니다.

Section 11.6.2에서 예제를 Poisson 분포를 이용해서 다시 한번 작업해 봅시다. 이 예제가 시간당 30% 실패 확률을 갖는 테스트에 관련된 거라는 점, 그리고 질문은 의심되는 수정사항에 대해 그 수정사항이 정말로 실패 확률을 줄였다고 99% 확신하기 위해서는 테스트가 얼마나 오래 실패 없이 돌아갈 수 있어야 하는가라는 점을 다시 상기합시다. 이 경우에, λ 는 0이므로, Equation 11.8은 다음과 같이 간략화됩니다:

$$F_0 = e^{-\lambda} \quad (11.9)$$

이걸 풀기 위해서는 F_0 를 0.01로 설정해야 하며, 여기서 λ 를 구해 보면:

$$\lambda = -\log 0.01 = 4.6 \quad (11.10)$$

우리가 시간당 0.3 실패를 가지므로, 요구되는 시간은 $4.6/0.3 = 14.3$ 으로, Section 11.6.2에서 사용한 방법으로 계산한 13시간과 10% 오차 내에 있습니다. 일반적으로는 실패 확률이 10% 내라는 것은 알지 못할 것이라는 점을 놓고 보면, 이는 Section 11.6.2에서의 방법이 상당히 많은 상황에서 Poisson 분포를 대체할 수 있는 훌륭하고 충분한 대체제임을 말합니다.

더 일반적으로 말해서, 단위시간당 n 실패를 갖는다면, 그리고 어떤 수정사항이 실패 확률을 줄였다고 P% 확신하려면 다음의 공식을 사용할 수 있습니다:

$$T = -\frac{1}{n} \log \frac{100 - P}{100} \quad (11.11)$$

Quick Quiz 11.10: 어떤 버그가 평균적으로 시간당 세번의 테스트 실패를 유발한다고 생각해 봅시다. 수정사항이 실패의 확률을 상당히 줄였음을 99.9% 증명할 수 있는 증거를 위해서는 테스트는 에러 없이 얼마나 오래 돌아가야만 할까요? ■

앞에서와 같이, 버그가 덜 빈번하게 발생하고 필요한 확신의 수준이 클수록 에러 없이 테스트가 수행되어야 하는 시간이 길어집니다.

테스트가 매 시간마다 한번씩 실패하지만, 버그 수정후, 24시간의 테스트에서 두번만 테스트가 수행중 실패했다고 생각해 봅시다. 해당 버그로 이끌어진 실패는 무작위적으로 발생한다고 하면, 두번째 테스트 수행에서의 적은 수의 실패들은 무작위적 기회로 발생한 것일 확률은 얼마일까요? 달리 말해서, 해당 수정이 정말로 버그에 대해서 어떤 영향을 끼쳤다고 얼마나 확신해야 하는 걸까요? 이 확률은 다음과 같이 Equation 11.8을 합하는 것으로 구할 수 있을겁니다:

$$F_0 + F_1 + \dots + F_{m-1} + F_m = \sum_{i=0}^m \frac{\lambda^i}{i!} e^{-\lambda} \quad (11.12)$$

이는 Poisson 누적분포지수 (cumulative distribution function) 인데, 더 간단하게는 다음과 같이 쓰일 수 있습니다:

$$F_{i \leq m} = \sum_{i=0}^m \frac{\lambda^i}{i!} e^{-\lambda} \quad (11.13)$$

여기서 m 은 긴 시간동안의 테스트에서의 에러의 갯수 (이 경우, 2)이고, λ 는 긴 시간동안의 테스트에서 발생하리라 예측된 에러들의 갯수입니다 (이 경우, 24). $m = 2$ 와 $\lambda = 24$ 를 이 수식에 대입하면 약 1.2×10^{-8}

으로 2 언저리의 확률이 나오는데, 달리 말해서, 해당 수정 사항이 정말로 해당 버그에 어떤 관계를 맺었음을 말하는 높은 수준의 증거를 가질 수 있는 겁니다.⁸

Quick Quiz 11.11: 이 모든 계승식 (factorial) 들과 지수식 (exponential) 들을 더하고 있는건 진짜 고통스럽습니다. 좀 더 편한 길은 없나요? ■

Quick Quiz 11.12: 하지만 잠시만요!!! 실패들의 어떤 횟수가 있어야 한다는 (실패가 없을 수 있다는 가능성은 포함해서) 점을 놓고 보면, Equation 11.13에 보여진 더하기는 m 을 1로 놓으면 무한대가 되지 않나요? ■

이 Poisson 분포는 테스트 결과를 분석하기 위한 강력한 도구입니다만 이 마지막 예에서는 24시간동안의 테스트 수행에서의 두개의 실패들이 남아있다는게 사실입니다. 그런 낮은 실패 확률은 매우 긴 테스트 수행을 필요로하게 됩니다. 다음 섹션에서는 이 상황을 개선하기 위한 반직관적인 방법들을 논해봅니다.

11.6.4 Hunting Heisenbugs

이런 생각을 해보는게 heisenbug의 설명에 도움이 됩니다: 추적 기능과 단정문을 추가하는 것은 버그의 발생 가능성을 줄여버릴 수 있습니다. 그리고 이게 바로 극단적으로 가벼운 추적기능과 단정문 메커니즘이 그렇게 중요한 이유입니다.

그 이름인 “heisenbug”는 특정 입자의 위치와 속도를 어떤 특정 시점에 정량화 하는 것은 불가능하다는, 양자 물리학의 하이젠버그 불확정성 원리 [Hei27] (Heisenberg Uncertainty Principle)에서 나왔습니다. 해당 입자의 위치를 좀 더 정확하게 측정하려는 모든 시도는 그 것의 속도에 대한 불확정성을 더욱 증가시키는 결과를 낳습니다. 비슷한 효과가 heisenbug에 대해서도 발생합니다: heisenbug를 추적하려는 시도들은 그 버그의 증상을 철저히 바꿔버리거나 완전히 사라져 버리기조차하게 합니다.

물리학 방면에서 이 문제의 이름에 대한 영감을 받았다면, 그에 대한 해결책을 위해서도 물리학 방면을 찾아보는것이 논리적입니다. 다행히도, 입자 물리학은 그에 대한 일을 하고 있습니다: Heisenbug를 섬멸하기 위해 anti-heisenbug를 창조하면 어떨까요?

이 섹션은 그러기 위한 몇가지 방법들을 설명합니다:

1. 경주조건을 일으키기 쉬운 영역들에 딜레이를 추가하기.
2. 워크로드의 강도를 증가시키기.
3. 의심스러운 서브시스템들을 격리시켜서 테스트하기.

⁸ 물론, 이 결과가 남은 두개의 실패들을 만들어낸 버그(들)을 찾아내고 고쳐야 한다는 사실에 대해 변명거리가 되지 않습니다!

4. 일반적이지 않은 이벤트들을 시뮬레이션 해보기.

5. 실수에 가까운 것들의 수를 세어보기.

특정 heisenbug를 위한 anti-heisenbug를 만들어내는건 과학이라기보다는 예술에 가깝습니다만, 다음 섹션들은 anti-heisenbug의 관련된 것들을 만들어내는데 몇가지 팁을 줄겁니다.

11.6.4.1 Add Delay

Section 5.1의 카운트가 누실되는 코드를 생각해 봅시다. 여기에 `printf()` 문을 집어넣는 행위는 누실되는 카운트 엄청나게 줄이거나 심지어 없애버릴 수도 있을 겁니다. 하지만, 읽어오고-더하고-저장하는 시퀀스를 읽어오고-더하고-기다리고-저장하는 시퀀스로 바꾸는 것은 누실되는 카운트의 발생률을 엄청나게 증가시킬 겁니다 (해보세요!). 경주 상황에 관련되어 있는 버그를 일단 발견했다면, 이런 식으로 딜레이를 추가하는 것으로 anti-heisenbug를 만들어내는게 많은 경우에 가능합니다.

물론, 이는 첫째로 경주 상황을 어떻게 찾아야 하는지에 대한 질문을 이끌어 냅니다. 이건 약간 어둠의 예술입니다만, 그것들을 찾기 위해 할 수 있는 것들이 몇 가지 있습니다.

한가지 방법은, 경주 조건은 많은 경우에 그 경주에 관련된 데이터를 오염시키는 결과를 이끌어 냅을 인식하는 것입니다. 따라서 모든 오염된 데이터의 동기화를 두번씩 체크하는 것은 좋은 습관입니다. 설령 당장은 그 경주 조건을 인식할 수 없다고 해도, 오염된 데이터로의 액세스 전과 후에 딜레이를 추가하는 것은 실패 확률을 바꿀 것입니다. 딜레이들을 잘 조직된 형태(ex: 이진 텁색)로 넣고 빼는 것으로, 경주 조건의 동작에 대해 좀 더 알 수 있을 겁니다.

Quick Quiz 11.13: 만약 그 오염이 어떤 관계없는 포인터에 영향을 끼쳐서 그 포인터를 따라가서 오염을 일으키는 경우에 이 방법은 어떻게 도움을 줄 수 있을까요??? ■

또다른 중요한 방법은 소프트웨어와 하드웨어 구성을 다양하게 해가면서 실패 확률의 통계적으로 심각한 차이를 찾아보는 것입니다. 그리고 나면 실패 확률에 가장 커다란 차이를 만들어낸 소프트웨어나 하드웨어 구성 변경점에 의해 암시되는 코드를 더 집중적으로 보는 것입니다. 예를 들어, 그 코드를 격리시켜서 테스트해보는 것도 도움이 될 수 있을 겁니다.

소프트웨어 구성의 한가지 중요한 측면은 변경의 역사인데, 이게 바로 `git bisect`이 유용한 이유입니다. 변경 역사의 이등분된 한쪽은 heisenbug의 원리에 의해 매우 가치있는 단서를 제공할 수 있습니다.

Quick Quiz 11.14: 하지만 제가 그렇게 이등분을 통

한 탐색을 해봤는데, 결국 나온 범인은 거대한 커밋이었어요. 이제 뭘 해야 하죠? ■

어떻게든 의심스러운 코드 영역을 찾아냈다면, 이제 당신은 실패의 확률을 증가시키기 위해서 딜레이들을 넣을 수 있습니다. 앞서서 우리가 본 것처럼, 실패의 확률을 증가시키는 것은 관련된 수정사항에 대한 높은 신뢰를 얻기 쉽게 해줍니다.

하지만, 일반적인 디버깅 테크닉들을 사용해서 그런 문제를 추적하는 것은 어떤 경우에는 상당히 복잡합니다. 뒤따르는 섹션들은 몇가지 다른 대안들을 제시합니다.

11.6.4.2 Increase Workload Intensity

특정 테스트 셋이 특정 서브시스템에는 상대적으로 적은 스트레스를 가해서 타이밍에 대한 작은 변화가 heisenbug 를 사라지게 할 수 있게 되는 경우는 흔합니다. 이런 경우를 위한 anti-heisenbug 를 만들어내는 한가지 방법은 워크로드의 강도를 증가시키는 것으로, 이는 버그가 나타날 확률을 올리는 기회를 갖습니다. 만약 확률이 충분히 증가한다면, 버그가 사라지게 하지 않으면서도 추적 기능과 같은 가벼운 디버깅 기능들을 추가할 수도 있을 겁니다.

워크로드의 강도를 어떻게 증가시킬 수 있을까요? 이는 프로그램에 따라 다르긴 합니다만, 여기 몇가지 시도해볼만한 것들이 있습니다:

1. CPU 들을 추가합니다.
2. 프로그램이 네트워킹을 사용한다면, 더 많은 네트워크 어댑터들과 원격의 시스템들을 더 많이 추가하거나 더 빠른 것으로 바꾸세요.
3. 프로그램이 문제가 발생할 때 무거운 I/O 를 하고 있다면, (1) 더 많은 저장장치를 추가하거나, (2) 디스크를 SSD 로 대체하거나 하는 식으로 더 빠른 저장장치를 사용하거나, (3) 대용량 저장장치를 메인 메모리로 대체하기 위해 RAM 기반 파일시스템을 사용하세요.
4. 예를 들어 병렬 행렬 곱셈을 하고 있다면 행렬의 크기를 바꾸는 식으로 문제의 크기를 바꾸세요. 더 큰 문제들은 더 많은 복잡도를 가져올 수 있습니다만, 더 작은 문제들은 종종 경쟁의 수준을 증가시킵니다. 더 크게 해야 할지 작게 해야 할지 모르겠다면, 그냥 둘 다 해보세요.

하지만, 버그가 특정 서브시스템 안에 있고, 프로그램의 구조가 그 서브시스템에 가해질 수 있는 스트레스의 양을 제약하고 있는 경우가 종종 있습니다. 다음 섹션은 이 상황을 다뤄봅니다.

11.6.4.3 Isolate Suspicious Subsystems

의심되는 서브시스템에 많은 스트레스를 가하는 것이 어렵거나 불가능한 구조로 프로그램이 짜여져 있는 경우에 유용한 anti-heisenbug 는 서브시스템을 격리시켜서 테스트하는 스트레스 테스트입니다. 리눅스 커널의 rcutorture 모듈은 RCU 에 대해 정확히 이 방법을 취하고 있습니다: 제품 환경에서 가능한 것보다 많은 스트레스를 RCU 에게 가함으로써, RCU 버그들이 제품 사용 상황에서보다 rcutorture 테스트 중에 발생할 확률이 더 높아집니다.⁹

사실, 병렬 프로그램을 만들때에는 컴포넌트들을 개별적으로 스트레스 테스트 하는게 현명합니다. 그런 컴포넌트 단계의 스트레스 테스트들을 만들어내는 것은 시간 낭비처럼 보일 수도 있습니다만 약간의 컴포넌트 단계 테스트가 수많은 양의 시스템 단계 디버깅을 줄일 수 있습니다.

11.6.4.4 Simulate Unusual Events

Heisenbug 들은 어떨 때에는 메모리 할당 실패, 조건적 락 획득 실패, CPU-hotplug 오퍼레이션, 타임아웃, 패킷 로스, 등등의 일반적이지 않은 이벤트로 인해 발생하기도 합니다. 이런 종류의 heisenbug 를 위한 anti-heisenbug 를 구성하는 한가지 방법은 가짜 실패를 만들어내는 겁니다.

예를 들어, `malloc()` 을 곧바로 호출하는 대신에, 그냥 무조건적으로 `NULL` 을 리턴하거나 정말로 `malloc()` 을 호출하고 그 결과로 나온 포인터를 리턴하는데 두가지 경우를 선택하는데에 무작위적 숫자를 사용하는 wrapper 함수를 호출하는 겁니다. 가짜 실패를 만들어내는 것은 병렬 프로그램들은 물론 순차적 프로그램들에서도 견고성을 만들어내기 위한 좋은 방법입니다.

Quick Quiz 11.15: 이미 존재하는 조건적 락킹 도구들은 이런 가짜 실패 기능을 제공하지 않는 이유가 뭔가요? ■

11.6.4.5 Count Near Misses

버그들은 대부분의 경우 그렇거나 아니거나 것들이어서 버그는 발생하거나 그렇지 않거나일뿐, 중간의 것은 없습니다. 하지만, 버그가 실패를 만들어내지는 않지만 거의 명백한 경우인 *near miss* 를 정의하는게 가능한 경우가 간혹 있습니다. 예를 들어, 당신의 코드가 로봇을 걷게 한다고 생각해 봅시다. 이 로봇이 넘어지는 일은 당신의 프로그램 안에 버그가 있음을 나타낼 것입니다만, 비틀거리고서는 회복되는 것은 *near miss* 를 나타낼

⁹ 슬프게도 확률이 1까지 되지는 못하긴 하지만 말이죠.

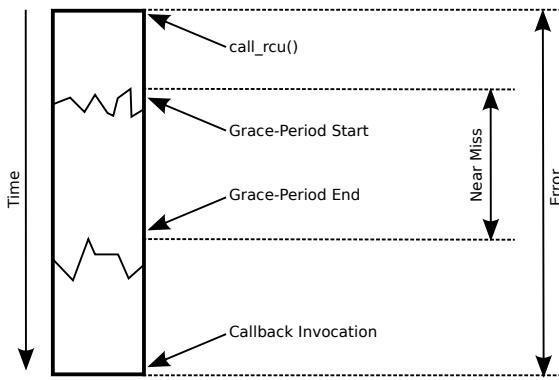


Figure 11.5: RCU Errors and Near Misses

수도 있습니다. 만약 로봇이 한시간에 한번만 넘어진다면, 하지만 몇분마다 비틀거린다면, 넘어지는 횟수에 대해서 비틀거리는 횟수를 셈으로써 디버깅 진도를 좀 더 빠르게 만들 수 있을 겁니다.

동시성 있는 프로그램에서는 타임스탬프를 찍는 일이 가끔은 near miss 들을 파악해 내는데 사용될 수 있습니다. 예를 들어, 락킹 도구들은 상당한 딜레이들을 일으키는데, 같은 락에 대해서 서로 다른 획득을 통해 보호될 것으로 생각되는 한쌍의 오퍼레이션들 사이에 너무 짧은 딜레이가 존재한다면, 이 너무 짧은 딜레이는 near miss 로 카운트될 수도 있을 겁니다.¹⁰

예를 들어, RCU 의 우선순위 올리기에 있는 낮은 확률의 버그는 rcutorture 의 집중된 테스트에서 대략 100 시간에 한번 일어났습니다. 그 버그가 일어날 확률이 상당히 감소되었음을 99% 확신하기 위해서는 거의 500 시간의 실패 없는 테스트 수행이 필요하기 때문에, 그 실패를 찾아내기 위한 git bisect 과정은 고통스러울 정도로 느릴겁니다—또는 극단적으로 거대한 테스트 환경을 필요로 할겁니다. 다행히도, 테스트된 해당 RCU 오퍼레이션은 RCU grace period 를 위한 대기만이 아니라, 시작하기 위한 그 grace period 에 대한 앞선 대기와 그 RCU grace period 가 완료된 후에 호출될 RCU callback 을 위한 뒤이은 대기도 존재했습니다. rcutorture 에러와 near miss 사이의 차이가 Figure 11.5 에 보여져 있습니다. 완전히 본격적인 에러로 구분되기 위해서, RCU read-side 크리티컬 섹션은 grace period 를 시작한 call_rcu() 로부터 앞의 grace period 의 남아있는 부분과 call_rcu() 로 시작된 grace period 전체 (지그재그로 그어진 선으로 표시된 영역), 그리고 그 grace period 의 끝으로부터 callback 의 호출 사이의 딜레이까지 확장되는데, 이게 “Error” 화살표로

¹⁰ 물론, 이 경우, lock_held() 기능과 같은 뭐가 됐든 당신의 환경 상에서 사용 가능한 것을 사용하는게 나을 수도 있을 겁니다. 만약 lock_held() 기능이 없다면, 하나 만들어서 쓰세요!

표시되어 있습니다. 하지만, 정식적인 RCU 의 정의는 RCU read-side 크리티컬 섹션이 하나의 grace period 사이로 확장되는 것을 금지하는데, 이는 “Near Miss” 화살표로 나타내어져 있습니다. 이는 near miss 들을 에러 조건으로 사용할 것을 제안하지만, 다른 CPU 들은 해당 grace period 가 정확히 언제 시작되고 종료되었는지에 대해 지그재그로 그어진 선으로 보여지듯이 다른 의견을 가질 수 있기 때문에, 이는 문제가 될 수 있습니다.¹¹ 따라서 이 near miss 들을 에러 컨디션으로 사용하는 것은 거짓 양성 결과를 만들어낼 수 있는데, 이는 자동화된 rcutorture 테스트에서는 제거되어야 합니다.

순수히 바보같은 운으로 인해, rcutorture 는 grace period 의 near-miss 버전에 민감한 통계를 포함하게 되었습니다. 앞서 이야기한대로, 이런 통계들은 RCU 의 상태 변수들에 대한 동기화 되지 않은 액세스로 인해서 거짓 양성 결과에 영향을 받기 쉽습니다만, 이런 거짓 양성 결과는 IBM 메인프레임이나 x86 과 같은 강한 순서 규칙의 시스템들에서는 상당히 희귀하게 발생하는 것으로 드러났는데, 1000 시간의 테스트 가운데 한번 보다도 낮은 확률로 일어납니다.

이런 near miss 들은 대략 한시간에 한번 꼴로 발생하는는데, 이는 실제 에러들보다 100배 이상 빈번하게 발생하는 겁니다. 이런 near miss 들의 사용은 버그의 원인을 일주일 안되는 시간 내에 파악해내고 수정된 내용의 효과를 하루 내에 확인하는 것을 가능하게 했습니다. 반면에, 진짜 에러만을 보기 위해서 near miss 들을 제외시키는 행위는 수개월의 디버그와 검증 시간을 필요로 했습니다.

Near-miss 카운팅을 정리하자면, 일반적인 방법은 빈번하지 않게 발생하는 실패의 횟수를 세는 것을 더 빈번하게 발생하며 그 실패들과 연과노디어 있을 것으로 여겨지는 near miss 의 횟수를 세는 것으로 대체하는 것입니다. 이런 near-miss 들은 진짜 실패의 heisenbug 에 대한 anti-heisenbug 로 여겨질 수 있는데, 더 빈번하게 발생하는 near-miss 들은 예를 들어 디버깅 코드를 추가하기 위해 만드는 것과 같은 코드의 변경이 있는 상황에서도 그 발생률이 더 일관될 것이기 때문입니다.

여태까지, 우리는 병렬 프로그램의 기능성에서의 버그들에 대해서만 집중해 왔습니다. 하지만, 성능이 병렬 프로그램에서의 첫번째 필요사항이기 때문에 (그렇지 않다면, 순차적 프로그램을 짜지 그려세요?), 다음 섹션에서는 성능상의 버그들을 찾아내는 방법을 알아봅니다.

¹¹ 아무일도 하지 않고 있는 CPU 들은 최근의 수백개의 grace period 들에 대해서도 전혀 알지 못하고 있을 수도 있기 때문에, 이 지그재그로 그어진 라인들은 상당히 과소평가 되어 있습니다.

11.7 Performance Estimation

병렬 프로그램은 보통 성능과 확장성에 대한 요구를 받게 됩니다. 무엇보다, 만약 성능이 문제가 아니라면, 순차적 프로그램을 사용하는게 낫겠죠? 궁극의 성능과 선형적 확장성은 필요치 않을 수도 있습니다만, 순차적으로 같은 일을 하는 최적화된 프로그램보다 느리게 동작하는 병렬 프로그램은 많이 사용되지 않을 겁니다. 그리고 마이크로세컨드도 문제가 되고 나노세컨드라도 필요한 경우들이 정말로 존재합니다. 따라서, 병렬 프로그램에 있어서, 불충분한 성능은 올바르지 못하게 동작하는 것만큼이나 커다란 버그입니다.

Quick Quiz 11.16: 그건 웃긴 이야기네요!!! 어쨌든, 좀 늦더라도 올바른 답을 얻는게 잘못된 답을 얻는 것 보다는 낫지 않겠어요??? ■

Quick Quiz 11.17: 하지만, 어플리케이션을 병렬화하기 위해 필요한 모든 어려운 일을 해내기로 했다면 왜 그걸 올바르게 하지 않는거죠? 왜 최적의 성능과 선형적인 확장성보다 못한 것에 안주하는 겁니까? ■

따라서 병렬 프로그램을 검증하는 일은 그 성능을 검증하는 것을 포함합니다. 하지만 성능을 검증하는 일은 수행시킬 워크로드와 그 프로그램을 평가할 성능 지표를 가지고 있음을 의미합니다. 이런 필요한 것들은 성능 벤치마크로 얻어지기도 하는데, 다음 섹션에서 이에 대해 논해 봅니다.

11.7.1 Benchmarking

오래된 말로 “거짓말이 있어, 빌어먹을 거짓말, 통계, 그리고 벤치마크들.” 란 말이 있죠. 하지만, 벤치마크들은 상당히 많이 사용되고 있고, 따라서 그것들을 지나치게 멸시하는 것은 도움이 되지 않습니다.

벤치마크들은 임시 변통의 움직임들부터 국제적인 표준까지 그 범위가 다양합니다만, 그것들의 정형성의 수준과는 관계없이, 벤치마크들은 네개의 주요 목적을 처리합니다:

1. 비교되는 구현들을 공정하게 비교하기 위한 프레임워크를 제공합니다.
2. 사용자에게 가치있는 방향으로 구현을 개선하기 위한 경쟁적 에너지에 집중합니다.
3. 벤치마크되는 구현의 예제적 사용으로서의 역할을 합니다.
4. 당신의 소프트웨어의 강점을 경쟁자가 제공하는 것에 대비해서 강조하는 마케팅 도구의 역할을 합니다.

물론, 완전하게 공정한 프레임워크는 의도된 어플리케이션 그 자체밖에 없습니다. 그런데 벤치마킹에서의 공정성에 대해서 신경쓰는 사람이라면 단순히 그 어플리케이션 자체를 벤치마크로 사용하는 대신에 불완전한 벤치마크들을 만들어낼 생각을 하는 걸까요?

실제 어플리케이션을 수행하는 것은 그게 실용적인 곳에서라면 실제로 그게 최고의 방법입니다. 불행히도, 그건 다음과 같은 이유로 실용적이지 못한 경우가 많습니다:

1. 그 어플리케이션은 독점된 것이고, 당신이 그 의도된 어플리케이션을 수행할 권한을 갖고 있지 못할 수 있습니다.
2. 그 어플리케이션은 당신이 접근할 수 없는 더 많은 하드웨어를 필요로 할 수도 있습니다.
3. 그 어플리케이션은 당신이 예를 들어 개인정보 규제와 같은 이유로 법적으로 접근할 수 없는 데이터를 사용할 수도 있습니다.

이런 경우에, 그 어플리케이션을 모사하는 벤치마크를 만드는 것이 이런 문제들을 극복하는데 도움이 될 수 있습니다. 세심하게 구성된 벤치마크는 성능, 확장성, 에너지 효율성, 그리고 그 외에도 많은 것들을 촉진시키는데 도움이 될 수 있습니다.

11.7.2 Profiling

소프트웨어의 매우 작은 부분이 성능과 확장성 한계의 주요한 책임을 가지고 있는 경우가 많습니다. 하지만, 개발자들이 실제 병목지점을 손으로 찾아내기는 악명 높도록 불가능합니다. 예를 들어, 커널 버퍼 할당자의 경우에 있어서, 모든 관심은 밀집도가 높은 배열의 탐색에 집중되었습니다만 이는 할당자의 실행 시간 중 몇 퍼센트만을 차지하는 것으로 드러났습니다. 논리 분석기를 통해 수집된 수행 프로파일은 정말로 문제의 주요한 부분에 책임이 있었던 캐시 미스에 관심을 집중시켰습니다 [MS93].

성능과 확장성 버그를 추적하는데 있어 옛날 방식이지만 상당히 효과적인 한가지 방법은 프로그램을 디버거와 물려서 수행시키면서 주기적으로 프로그램을 인터럽트 하고서 매 인터럽트마다 모든 쓰레드의 스택을 기록하는 것입니다. 여기서의 이론은 무언가가 프로그램을 느리게 만들고 있다면, 그것은 쓰레드의 수행에 보일 것이라는 것입니다.

그렇다면 하지만, 당신의 관심을 가장 필요한 곳에 집중할 수 있도록 도와주는데 있어 일반적으로는 훨씬 나은 일을 해주는 많은 도구들이 있습니다. 대중적인 두 가지 선택이 있는데 `gprof` 와 `perf`입니다. 단일 프로세스 프로그램에 `perf`를 사용하여 한다면 그 프로그램을 수행하는 커맨드 앞에 `perf record`를 붙여서

수행하고, 그 커맨드가 완료된 후에, `perf report` 를 입력하세요. 멀티 쓰레드 프로그램의 성능 디버깅을 위한 도구를 위한 수많은 작업들이 있었는데, 이것들은 이 중요한 일을 좀 더 쉽게 만들었습니다. 다시한번 말하지만, Brendan Gregg 의 블로그가 시작점으로 좋습니다.¹²

11.7.3 Differential Profiling

확장성 문제는 프로그램을 매우 큰 시스템들에서 수행해보기 전까지는 분명하지 않을 겁니다. 하지만, 훨씬 작은 시스템들에서 프로그램을 수행할 때라고 해도 임박한 확장성 문제들을 찾아내는 것이 가능할 때도 있습니다. 이를 위한 한가지 테크닉은 차이점 프로파일링 (*differential profiling*) 이라고 불립니다.

아이디어는 워크로드를 두개의 서로 다른 조건들의 집합에서 수행해 보는 겁니다. 예를 들어, 워크로드를 두개의 CPU 위에서 돌려볼 수 있고, 그리고나서는 또다시 네개의 CPU 위에서 해보는 겁니다. 또는 시스템에 가하는 부하의 정도, 네트워크 어댑터들의 갯수, 대용량 저장장치의 갯수, 등을 다양하게 변화시켜볼 수 있습니다. 그리고나선 두 수행의 프로파일들을 모으고, 수학적으로 관련된 프로파일 측정결과를 조합해 보는 겁니다. 예를 들어, 주요한 관심이 확장성이라면, 연관된 측정의 비율을 취하고, 그 비율들을 숫자상으로 내림차순으로 정렬해 볼 수 있을 겁니다. 가장 의심스러운 확장성에 대한 용의자는 정렬된 리스트의 꼭대기에 위치하게 될 겁니다 [McK95, McK99].

`perf` 와 같은 일부 도구들은 내장된 차이점 프로파일링 지원 기능을 가지고 있습니다.

11.7.4 Microbenchmarking

마이크로벤치마킹은 소프트웨어의 더 큰 부분에 포함되기에 어떤 알고리즘이나 데이터 구조가 더 적합할지를 결정하는데에 있어 더 깊은 평가를 위해 유용할 수 있습니다.

마이크로 벤치마킹을 하는 혼한 방법 한가지는 시각을 측정하고, 테스트 되는 코드를 몇번 반복해서 수행시키고, 다시 시각을 측정하는 것입니다. 두개의 시각 사이의 차이를 반복 횟수로 나누면 테스트 되는 코드를 수행하는데 필요한 시간의 측정값이 나옵니다.

불행히도, 측정을 위한 이 방법은 여러개의 에러가 끼어들 수 있게 하는데, 다음과 같은 것들이 있습니다:

1. 이 측정은 시각 측정의 오버헤드가 포함될 수 있습니다. 이런 에러는 수행 반복 횟수를 증가시킴으로써 임의의 작은 값으로 줄일 수 있습니다.

2. 처음 몇번의 테스트 수행 반복은 캐시 미스나 (더 나쁜) 페이지 폴트를 유발할 수도 있는데, 이는 측정된 값을 부풀릴 수도 있을 겁니다. 이런 에러는 역시 수행 반복의 수를 늘림으로써 줄일 수 있으며, 측정 기간이 시작되기 전에 몇번의 웜업 수행 반복을 수행시킴으로써 완전히 없앨 수 있는 경우도 많습니다.
3. 예를 들어 임의의 메모리 에러와 같은 간접의 일부 종류들은 테스트의 수행 반복의 여러 세트를 수행 시킴으로써 조치가 될 수 있을 정도로 드물게 발생합니다. 만약 간접의 정도가 통계적으로 심각하다면, 모든 유난스럽게 다른 성능 결과들을 통계적으로 제거시킬 수 있습니다.
4. 모든 테스트 반복 수행이 시스템의 다른 활동에 의해 간섭을 받을 수 있습니다. 간섭의 근원은 다른 어플리케이션들, 시스템 유트리티들과 데몬, 디바이스 인터럽트, 펌웨어 인터럽트 (시스템 관리 인터럽트인 SMI 포함), 가상화, 메모리 에러, 그리고 그 외에도 다른 여러가지가 있을 수 있습니다. 이런 간섭 유발사건들이 무작위적으로 발생한다고 가정하면, 그것들의 효과는 반복 횟수를 줄임으로써 최소화 시킬 수 있습니다.

간섭 유발 사건들 중 첫번째와 네번째 것은 서로 상충되는 조언을 하고 있는데, 이건 우리가 실제 세계에 살고 있다는 한가지 증거입니다. 이 섹션의 나머지 부분에서는 이런 상충되는 것을 어떻게 해결해야 할지 알아봅니다.

Quick Quiz 11.18: 하지만, 예를 들어 캐시와 메모리 배치 사이의 간섭에 의한 것과 같은 다른 에러 유발 원인들은 어떻게 하죠? ■

다음의 섹션들은 측정 에러들을 처리하는 방법들을 이야기 해보는데, Section 11.7.5에서는 일부 형태의 간섭들을 막기 위해서 사용될 수 있는 격리화 테크닉들을 다루고, Section 11.7.6는 간섭으로 인해 오염되었을 수 있는 측정 데이터를 버릴 수 있도록 간섭을 파악하는 방법들을 다룹니다.

11.7.5 Isolation

리눅스 커널은 특정 CPU 들을 바깥의 간섭으로부터 격리시키는 몇가지 방법들을 제공합니다.

먼저, 다른 프로세스, 쓰레드, 그리고 태스크들에 의해 가해질 수 있는 간섭을 봅시다. POSIX `sched_setaffinity()` 시스템 콜은 많은 태스크들을 특정 CPU 들로부터 몰아내고 당신의 테스트들을 같은 그룹의 CPU 들로 가둬두는데 사용될 수 있습니다. 리눅스에 존재하는 사용자 레벨 `taskset` 커맨드도 같은 목적으로 사용될 수도 있습니다만 `sched_setaffinity()`

¹² <http://www.brendangregg.com/blog/>

와 taskset 은 모두 상승되는 권한을 필요로 하긴 합니다. 리눅스에 존재하는 control groups (cgroups) 가 같은 목적으로 사용될 수도 있습니다. 이 방법은 간섭을 줄이는데 있어 상당히 효과적일 수 있고, 많은 경우에는 이걸로 충분합니다. 하지만, 이 방법도 한계가 있는데, 예를 들어, 태스크들을 잡아두기 위해 자주 사용되는 per-CPU 커널 쓰레드들에 대해서는 아무일도 해주지 못합니다.

per-CPU 커널 쓰레드들로부터의 간섭을 피하는 한가지 방법은, 예를 들어 POSIX sched_setscheduler() 시스템 콜을 이용하거나 해서 테스트를 높은 real-time 우선순위로 수행시키는 것입니다. 하지만, 만약 이렇게 할 경우, 당신은 묵시적으로 무한 루프를 막는다는 책임을 갖게 되는 것인데, 그러지 않는다면 그 테스트는 커널의 한 부분이 동작을 못하게 만들어 버릴 것이기 때문입니다.¹³

이런 방법들은 프로세스, 쓰레드, 그리고 태스크들로부터의 간섭을 상당히 줄이고, 심지어 없애버릴 수도 있습니다. 하지만, 적어도 쓰레드 인터럽트의 부재 시에는 디바이스 인터럽트로부터의 간섭을 막는데에는 아무 일도 하지 않습니다. 리눅스는 인터럽트 벡터당 하나씩 숫자로 된 디렉토리를 담고 있는 /proc/irq 디렉토리를 통해 쓰레드 인터럽트를 일부 제어할 수 있게 해줍니다. 각각의 숫자 이름의 디렉토리는 smp_affinity 와 smp_affinity_list 를 담고 있습니다. 충분한 권한을 가지고 있다면, 특정 CPU 들로의 인터럽트를 제한하기 위해 이 파일들에 값을 써넣을 수 있습니다. 예를 들어, “sudo echo 3 > /proc/irq/23/smp_affinity” 는 vector 23 의 인터럽트들을 CPU 0 과 1에 국한시킬 겁니다. 같은 결과가 “sudo echo 0-1 > /proc/irq/23/smp_affinity_list” 를 통해 얻어질 수도 있습니다. 시스템의 인터럽트 벡터들의 리스트, 각 CPU 에 의해 얼마나 많이 처리되는데, 그리고 어떤 기기가 각각의 인터럽트 벡터를 사용하는지에 대한 정보를 얻기 위해 “cat /proc/interrupts” 를 사용할 수도 있습니다.

비슷한 커맨드를 시스템 상의 모든 인터럽트 벡터들에 대해 수행시키면 인터럽트들이 CPU 0 과 1 에 모두 국한되어서, 나머지 CPU 들을 간섭으로부터 자유로워지게 만들 겁니다. 또는 간섭으로부터 거의 자유이긴 한데, 어쨌던지요. 스케줄링 클락 인터럽트가 유저 모드로 돌아가고 있는 각각의 CPU 에서 발생함이 드러났습니다.¹⁴ 또한 인터럽트들을 가둔 CPU 들이 그 부하를 충분히 처리할 수 있음이 분명하도록 신경을 써 주어야

¹³ 이건 스파이더맨 원칙의 한 예입니다: “With great power comes great responsibility.”

¹⁴ Frederic Weisbecker 는 하나의 실행 가능한 태스크만을 가지고 있는 CPU 들에 대해서는 스케줄링 클락 인터럽트가 꺼지도록 하는 adaptive-ticks 프로젝트를 작업하고 있습니다만, 2013년 초까지는 이 작업은 여전히 진행중입니다.

만 합니다.

하지만 이는 같은 운영체제 위에서 돌아가고 있는 프로세스와 인터럽트들만을 처리합니다. 예컨대, KVM 을 통해 수행되는 리눅스와 같이 그 자체로 하이퍼바이저 위에서 돌아가고 있는 게스트 OS 안에서 테스트를 돌린다고 생각해 봅시다. 이론적으로는 같은 테크닉을 게스트 OS 수준에서 할 수 있듯이 하이퍼바이저에게 가능할 수도 있겠지만, 하이퍼바이저 단계의 오퍼레이션들은 권한을 가진 개인에게만으로 제약되어 있는 경우가 상당히 흔합니다. 또한, 이런 테크닉들 중 어느 하나도 펌웨어 수준의 간섭에 대해서는 아무 일도 하지 못합니다.

Quick Quiz 11.19: 테스트 되는 코드를 격리시키기 위해 제안된 이 테크닉들은 특히나 그 코드가 커다란 어플리케이션에서 돌아가고 있다면 그 코드의 성능에 영향을 끼치지 않을까요? ■

이런 고통스운 상황에 놓여있다면, 간섭을 방지하려고 하는 대신에, 다음 섹션에 설명된 것처럼 간섭을 찾아내야만 합니다.

11.7.6 Detecting Interference

간섭을 예방할 수 없다면, 사실에 기반해서 간섭을 찾아내고 그 간섭으로 영향을 받은 테스트 결과를 제거할 수 있을 겁니다. Section 11.7.6.1 에서는 추가적인 측정을 통한 잘못된 결과값 제거 방법을 알아보고, while Section 11.7.6.2 에서는 통계 기반의 결과 제거 방법을 알아봅니다.

11.7.6.1 Detecting Interference Via Measurement

리눅스를 포함한 많은 시스템들에서는 뒤늦게라도 어떤 형태의 간섭이 일어났었는지를 판단할 수 있는 방법을 제공합니다. 예를 들어, 테스트가 프로세스 기반의 간섭을 만났었다면, 테스트 도중에 컨텍스트 스위치가 일어났을 겁니다. 리눅스 기반의 시스템들에서, 컨텍스트 스위치는 /proc/<PID>/sched 안의 nr_switches 필드에 보여질 겁니다. 비슷하게, 인터럽트 기반의 간섭은 /proc/interrupts 파일을 통해 판단될 수 있습니다.

파일을 열고 읽는 것은 낮은 오버헤드를 위한 방법이 아니고, Figure 11.6 에 보여진 것처럼 getrusage() 시스템 콜을 사용해서 주어진 쓰레드의 컨텍스트 스위치 횟수를 얻는 것이 가능합니다. 마이너 페이지 폴트와 메이저 페이지 폴트를 파악하기 위해서 이와 같은 시스템 콜 (ru_minfl, ru_majfl) 을 사용할 수 있습니다.

안타깝게도, 메모리 어려와 펌웨어 간섭을 파악하는 것은 가상화로 인한 간섭을 파악하는게 그런 것처럼 상당히 실제 시스템에 의존적입니다. 비록 간섭을 파악하는 것보다는 없애는게 낫고, 파악이 통계보다는 낫겠지

```

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

만, 다음 섹션에서 다루게 될 주제인 통계에 기대야만하는 경우도 있습니다.

11.7.6.2 Detecting Interference Via Statistics

모든 통계 분석은 데이터에 대한 가정을 기반으로 가지고 있고, 성능 마이크로 벤치마크는 많은 경우 다음과 같은 가정을 지원합니다:

- 작은 측정이 커다란 측정보다 더 정확할 것이다.
- 좋은 데이터의 측정에 대한 불확실성은 알려져 있다.
- 테스트 수행 결과중에는 합리적인 양의 좋은 데이터를 가지고 있을 것이다.

더 작은 측정이 더 커다란 측정에 비해서 더 정확할 수 있다는 사실은 측정을 크기가 증가하는 순서대로 정렬하는 것이 생산적일 것임을 암시합니다.¹⁵ 측정의 불확실성이 알려졌다는 사실은 이 불확실성에 기반한 측정을 받아들일 수 있도록 합니다: 만약 간섭의 효과가 이 불확실성에 비해서 커다랗다면, 이는 나쁜 데이터를 버리는 것을 덜어줄 겁니다. 마지막으로, 일부 분량(예를 들어, 3분의 1)은 좋은 데이터일 것이라 가정될 수 있다는 사실은 정렬된 리스트에서 첫번째 부분을 별다른 검사 없이 받아들이는 것을 가능하게 할 것이고, 이 데이터는 측정된 데이터의 자연적인 분포도를 가정된 측정의 오류를 넘어서서 추정하는 데에 사용될 수 있을 것입니다.

¹⁵ 오랜 격언을 바꿔 말하자면, “정렬부터 하고 질문은 그다음에 하라.”

이 방법은 정렬된 리스트의 앞쪽의 특정 갯수의 원소들을 취하고, 이것들을 일반적인 원소간의 차이를 추정하는데에 사용하는 것으로, 이렇게 추정된 값은 받아들여질 수 있는 값의 최대 한계치를 얻기 위해서 리스트의 원소의 수를 곱하게 될 수도 있을 겁니다. 이 알고리즘은 이제 리스트의 다음 원소들을 반복적으로 받아들일 수 있을지 고려하게 됩니다. 만약 다음 원소의 값이 앞서 구한 최대 한계치 아래라면, 그리고 이 원소와 그 앞의 원소 사이의 거리가 앞서 받아들여진 리스트의 원소들 사이의 평균적 원소간 거리보다 너무 크지 않다면 그 원소는 받아들여지고 이 과정이 반복됩니다. 만약 그렇지 않다면, 리스트의 나머지 값들은 버려집니다.

Figure 11.7 는 이 방법을 구현한 sh/awk 스크립트를 보입니다. 입력값은 x축의 값에 이어 임의의 긴 y 축 값이고, 출력은 각각의 입력 라인에 대한 한 라인으로, 다음과 같은 필드를 갖습니다:

- x축 값.
- 선택된 데이터의 평균.
- 선택된 데이터의 최소값.
- 선택된 데이터의 최대값.
- 선택된 데이터 항목들의 갯수.
- 입력 데이터 항목들의 갯수.

이 스크립트는 다음과 같은 선택적 인자들을 갖습니다:

- divisor: 리스트를 나누게 될 세그먼트들의 갯수로, 예를 들어 4의 값을 갖게 되면 데이터 원소들의 앞쪽 4분의 1이 좋은 데이터라고 가정됩니다. 기본값은 3입니다.
- relerr: 상대적 측정 에러. 이 스크립트는 이 에러보다 적은 차이값을 갖는 값들은 모두 같은 의도로 만들어진 것이고 같은 목적을 위한 것이라고 가정합니다. 기본값은 0.01로, 이는 1% 와 동일합니다.
- trendbreak: 데이터의 추세를 깨놓는 것으로 간주되는 원소간 거리의 비율. 예를 들어, 받아들여진 데이터의 평균 구간이 1.5라면, 그리고 만약 추세를 깨놓는 것으로 간주되는 비율이 2.0이라면, 다음 데이터 값이 마지막 값과 3.0 이상 차이나면, 이는 추세를 깨놓는 것으로 간주됩니다. (물론, 상대적 에러가 3.0 보다 크지 않다면, 이 “깨짐”은 무시됩니다.)

```

1 divisor=3
2 relerr=0.01
3 trendbreak=0.01
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 11.7 의 Line 1-3 는 패러미터들의 기본값을 설정하고, line 4-21 은 이 패러미터들에 대한 커맨드 라인을 통한 재설정을 처리합니다. Line 23 과 24 의 awk 호출은 divisor, relerr, 그리고 trendbreak 의 shift에서의 것에 대한 상대역을 설정합니다. 일반적은 awk 매너로, line 25-52 는 각각의 입력 라인에 대해서 실행됩니다. Line 24 와 26 의 루프는 입력값의 y축 값들을 line 27 에서 오름차순으로 정렬되는 d 배열에 복사합니다. Line 28 은 divisor 를 적용하고 반올림해서 절대적으로 믿을 수 있는 y축 값들의 수를 계산합니다.

Line 29-33 은 y 축 값들의 하한과 상한으로 사용될

값인 maxdelta 를 계산합니다. 여기까지 해서, line 29 와 30 은 신뢰되는 구간의 값들의 차이들을 divisor 로 곱하는데, 이 값은 신뢰되는 구간에서의 값들의 차이를 y 축 값들 전체 집합에 적용하게 됩니다. 하지만, 이 값은 상대적 에러에 비해서는 상당히 작을 수 있으므로, line 31 에서는 절대적 에러값 ($d[i] * relerr$) 을 구하고 이를 데이터의 신뢰되는 구간에서의 차이값 delta 에 더합니다. Line 32 와 33 은 이 두 값들의 최대값을 계산합니다.

Line 34-43 의 루프의 각 패스는 다른 데이터 값들을 좋은 데이터 집합에 더하려 시도합니다. Line 35-39 는 추세를 깨는 차이값을 계산하고, line 36 에서는 아직 추세를 계산하기에 충분한 값을 가지고 있지 않다면 이 한계점을 폐기하고, line 38 와 39 에서는 trendbreak 를 좋은 데이터 집합의 데이터 값들의 쌍 사이의 차이값의 평균에 곱합니다. 만약 line 40 에서 후보 데이터 값은 최대 상한값 (maxdelta) 의 최소 하한을 넘기게 되었음이 파악되었다면, 그리고 line 41 에서 후보 데이터 값과 그 앞의 값 사이의 차이가 추세를 깨는 차이값 (maxdiff) 를 넘어선다고 판단된다면, line 42 에서는 루프를 빠져나갑니다: 이제 모든 좋은 데이터를 얻었습니다.

Line 44-52 는 이 데이터 셋의 통계를 출력합니다.

Quick Quiz 11.20: 이 방법은 좀 이상하군요! 왜 우리가 통계 수업시간에 배웠던 것처럼 평균과 표준편차를 사용하지 않죠? ■

Quick Quiz 11.21: 하지만 신뢰되는 데이터 그룹의 y 축 값들이 모조리 0이면 어떡하죠? 그러면 스크립트는 0이 아닌 값을 제거하지 않을까요? ■

통계적 간섭 파악 방법은 상당히 유용할 수 있긴 하지만, 이는 마지막 방법으로만 사용되어야 합니다. 첫 번째로는 간섭을 제거하려 하는 것이 (Section 11.7.5), 그럴 수가 없다면 측정을 통해 간섭을 파악하는 것이 (Section 11.7.6.1) 훨씬 낫습니다.

11.8 Summary

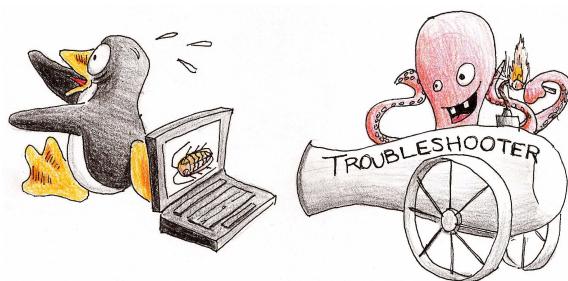


Figure 11.8: Choose Validation Methods Wisely

검증은 결코 엄밀한 과학이 될수는 없겠지만, 조직된 방법은 해야하는 일에 걸맞는 검증 도구를 선택하는 걸 도와서 Figure 11.8에 재밌게 그려진 것과 같은 상황을 피할 수 있기 할 것이기에 조직된 방법을 통해서 많은 것을 얻을 수 있을 겁니다.

선택의 핵심은 통계입니다. 이 챕터에서 소개된 방법들은 거의 모든 경우에 잘 동작하지만, 이것들은 한계들을 가지고 있습니다. 이 한계들은 Halting Problem [Tur37, Pul00]에 의해 일반적으로 불가능한 것을 하려 하기 때문에 근본적으로 존재합니다. 다행히도 우리에게는 주어진 프로그램이 동작을 정지할 것인지 아닌지 만이 아니라 Section 11.7에서 이야기된 것처럼 그것이 동작을 정지하기 전까지 얼마나 오래 돌아갈 수 있을 것인지만 알아내도 되는 수많은 특수한 경우들이 있습니다. 그뿐만 아니라, 주어진 프로그램이 올바르게 동작할 것인지 아닌지를 알아야 하는 경우에는, Section 11.6에서 논의된 것처럼 전체 시간 중 얼만큼의 부분에서 올바르게 동작하는지 추측을 해볼 수 있는 경우가 많습니다.

이런 추측에 의존할 것을 생각하지 않는건 만용을 부리려는것 이상도 이하도 아닙니다. 무엇보다, 이는 코드와 데이터 구조에 있는 거대한 양의 복잡도를 하나의 독립적 숫자로 요약하고 있는 것입니다. 놀랄만큼 많은 경우에 있어서 그런 만용과 같은 객기를 가지고도 일을 할 수 있긴 하지만, 요약되어지는 코드와 데이터는 결국은 상당한 문제들을 일으킬 것이라고 생각하는 편이 합리적입니다.

생길 수 있는 한가지 문제는 다양성으로, 반복된 테스트 수행들은 상당히 다른 결과들을 내놓을 수도 있습니다. 이는 많은 경우에 평균과 표준편차를 관리함으로써 처리되어집니다만, 커다랗고 복잡한 프로그램의 동작을 두개의 숫자만으로 요약하려 하는 것은 그 동작을 오로지 하나의 숫자로 요약하려 하는 것만큼이나 용감한 짓입니다. 컴퓨터 프로그래밍에 있어서, 놀랄만한 것은 평균이나 평균과 표준편차를 사용하는 것만으로도 충분한 경우가 찾다는 것입니다만, 거기에는 보장이 없습니다.

다양한 결과가 나타나는 한가지 이유는 당황스러운 사실들입니다. 예를 들어, 링크드 리스트 탐색에 의해 소모되는 CPU 시간은 리스트의 길이에 의존적일 것입니다. 전혀 다른 리스트 길이를 가진 테스트 수행 결과들을 가지고 평균을 내는 것은 유용하지 못할 것이고, 그 평균값에 표준편차를 더하는 것이 더 좋은 결과를 이끌어내지는 못할 겁니다. 해야할 올바른 일은 리스트 길이에 대한 상수를 가지고 있거나 리스트 길이에 따른 CPU 시간의 관계를 측정하는 식으로 리스트의 길이를 제어하는 것일 겁니다.

물론, 이 충고는 당신이 당황스러운 사실들에 대해서 인지하고 있다는 가정을 가지고 있고, Murphy는 당신

이 그렇지 않을 수 있다고 이야기 합니다. 전 (커질 때에 상당한 전력을 소모해버려서 컴퓨터에 공급되는 전압을 순간적으로 너무 낮아지게 만들어서, 가끔은 문제를 일으켰던) 에어컨, (성능에 있어서 이상한 다양성을 초래했던) 캐시 상태, (디스크 애러, 패킷 로스, 중첩된 이더넷 MAC 어드레스들을 포함한) I/O 애러들, 그리고 심지어 (돌고래가 없다면 상당히 정확한 음파를 통한 위치 선정과 운항을 위해 사용될 수 있는 전파 디지털 송수신기들과 놀고 싶은 마음을 자제할 수 없는) 돌고래들만큼이나 다양한 당황스러운 사실들을 가진 프로젝트들에 연관되어 있습니다. 그리고 이는 숙면이 왜 효과적인 디버깅 도구인지에 대한 이유 중 하나일 뿐입니다.

짧게 말해서, 검증은 항상 시스템의 행동에 대한 어떤 측정을 필요로 합니다. 이 측정은 시스템에 대한 상당한 요약이 되어야 하기 때문에, 이는 잘못된 방향으로 움직일 수 있습니다. 따라서 말 그대로, “조심하세요. 거기 있는건 하나의 진짜 세계입니다.”

하지만 당신이 2013년에 있어서 전세계에 약 10억개의 인스턴스들이 존재하는 리눅스 커널을 위한 작업을 하고 있다고 생각해 보세요. 그런 경우, 백만년에 한번 마주할 수 있는 버그는 전체 설치된 인스턴스에서는 하루에만 세번 마주할 수 있을 겁니다. 한시간동안 수행하면 이 버그를 50% 확률로 발견할 수 있는 테스트는 버그를 마주칠 확률을 십억배 이상 높여야 할텐데, 이는 오늘날의 테스트 방법론으로는 상당히 어려운 도전입니다. 어떤 경우에는 그런 경우에 좋은 효과를 줄 수 있는 한가지 중요한 도구는 정형 검증으로, 다음 챕터의 주제입니다.

Beware of bugs in the above code; I have only proved it correct, not tried it.

Donald Knuth

Chapter 12

Formal Verification

병렬 알고리즘들은 작성하기 어렵고, 디버깅하기는 그보다도 더 어렵습니다. 테스트는 필수이긴 하지만 race condition은 극단적으로 낮은 발현 확률을 가질 수 있기 때문에, 그것만으로는 충분하지 않습니다. 올바름의 증명은 가치있을 수 있습니다만, 그것도 결국은 원래의 알고리즘이 그렇듯이 사람에 의한 에러에 취약합니다. 또한, 올바름의 증명은 가정 자체에 있는 에러, 요구사항의 한계, 아래에 있는 소프트웨어나 하드웨어 기능들에 대한 잘못된 이해, 또는 증명을 할 생각을 하지 못한 에러들은 찾지 못할 것입니다. 이는 정형적 방법들은 테스트를 대체할 수는 없음을 의미합니다만, 정형적 방법들은 검증 도구상자의 또 하나의 가치있는 한가지 이상도 이하도 아닙니다.

어떻게든 모든 race condition들을 찾아낼 수 있는 도구를 갖는다면 매우 도움이 될겁니다. 그런 도구들이 여럿 존재하는데, 예를 들어서 Section 12.1은 범용 상태 공간 탐색 도구들인 Promela와 Spin을 소개하고, Section 12.2는 비슷하게 특수 목적의 ppcmem와 cppmem 도구들을 소개하며, Section 12.3은 자명한 방법의 예를 보고, Section 12.4에서는 간단히 SAT solver들을 살펴보며, 마지막으로 Section 12.5에서는 병렬 알고리즘을 검증하기 위해 정형 검증을 사용하는 것에 대해 요약합니다.

12.1 General-Purpose State-Space Search

이 섹션은 많은 종류의 멀티 쓰레드 기반 코드의 전체 상태 공간을 실행하는데에 사용될 수 있는 범용의 Promela와 spin 도구들을 알아봅니다. 이것들은 또한 데이터 통신 프로토콜들을 증명하는데에도 유용합니다. Section 12.1.1은 두개의 웹업을 위한 어토믹하지 않은 버전과 어토믹한 버전의 값 증가를 증명하는 예제를 포함해서 Promela와 spin에 대해 소개합니다. Section 12.1.2은 커맨드 라인에서의 사용 예와 Promela

와 C의 문법 비교를 포함해서 Promela를 설명합니다. Section 12.1.3에서는 락킹을 증명하는데에 Promela가 어떻게 사용될 수 있는지 보이고, 12.1.4는 일반적이지 않은 “QRCU”라는 이름의 RCU 구현을 증명하는데에 Promela를 사용해 보며, 마지막으로 Section 12.1.5에서는 RCU의 dyntick 구현에 Promela를 적용해 봅니다.

12.1.1 Promela and Spin

Promela는 증명 프로토콜들을 위해 설계된 언어입니다만, 작은 병렬 알고리즘들을 검증하는데에도 사용될 수 있습니다. 당신은 당신의 알고리즘과 정확성 제약을 C 같은 언어인 Promela로 다시 코딩하고, 그다음에 Spin을 사용해 그걸 C 프로그램으로 변환하고 나면 그걸 컴파일하고 실행해 볼 수 있습니다. 그 결과 나오는 프로그램은 당신의 알고리즘의 전체 상태 공간 탐색을 포함하고 있게 되어서, 당신이 당신의 Promela 프로그램에 넣어둔 단정들에 대한 반례들을 찾아주거나 검증하게 됩니다.

이 전체 상태 공간은 매우 강력할 수 있습니다만, 양 날의 검이 될 수 있기도 합니다. 당신의 알고리즘이 너무 복잡하거나 당신의 Promela 구현이 주의깊지 않다면, 메모리에 들어갈 수 있는 것보다 더 많은 상태들이 존재할 수도 있습니다. 더 나아가서, 충분한 메모리를 가졌다 하더라도, 상태 공간 탐색은 예상된 전체 시간 보다 더 긴 시간동안 수행될 수 있습니다. 따라서, 이 도구는 조그맣지만 복잡한 병렬 알고리즘들을 위해 사용하세요. 낙관적으로 이걸 (전체 리눅스 커널은 말할 것도 없고) 보통 크기의 알고리즘들에 적용하는 것도 나쁜 결과로 끝나게 될겁니다.

Promela와 Spin은 <http://spinroot.com/spin/whatispin.html>에서 다운로드 받을 수 있습니다.

앞의 사이트는 또한 Gerard Holzmann의 Promela와 Spin에 대한 훌륭한 책 [Hol03]으로의 링크를 제공하며, <http://www.spinroot.com/spin/Man/index.html>에서 시작하는 검색 가능한 온라인

```

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

레퍼런스들도 제공합니다.

이 문서의 뒷부분은 병렬 알고리즘들을 디버깅 하는 데에 Promela 를 어떻게 사용하는지를 간단한 예제로 시작해서 더 복잡한 경우들로 나아가면서 설명합니다.

12.1.1.1 Promela Warm-Up: Non-Atomic Increment

Figure 12.1 는 어토믹하지 않은 값 증가로 인해 발생하는, 교재에도 있는 race condition 을 보이고 있습니다. Line 1 은 수행할 프로세스들의 수를 정의하고 (우린 상태 공간에의 영향을 보기 위해 이 값을 바꿔볼 겁니다), line 3 은 카운터를 정의하고, line 4 는 line 29-39 에 있는 단정문을 구현하는데 사용될 겁니다.

Line 6-13 은 카운터를 어토믹하지 않게 증가시키는 프로세스를 정의합니다. 인자 me 는 프로세스의 번호로, 코드의 뒤에 있는 초기화 블락에서 설정됩니다. 간단한 Promela 구문들은 모두 어토믹한 것으로 가정되기 때문에, 우리는 이 값 증가를 line 10-11 의 두개의 문장으로 조개야만 합니다. Line 12 에서의 값 할당은 프로세스

의 완료를 표시합니다. Spin 시스템은 모든 가능한 상태들의 시퀀스들을 포함해 상태 공간을 모두 탐색하기 때문에, 전통적인 테스트에서라면 사용되었을 수 있는 루프는 필요치 않습니다.

Line 15-40 은 초기화 블락으로, 제일 처음에 수행됩니다. Line 19-28 은 정말로 초기화를 하고, line 29-39 는 단정을 수행합니다. 이 두 부분은 불필요하게 상태 공간을 증가시키는 걸 막기 위해 모두 어토믹 블락으로 되어 있습니다: 이것들은 테스트 하려는 알고리즘의 부분이 아니기 때문에, 그것들을 어토믹으로 만듬으로써 검증 범위를 줄이는 것입니다.

Line 21-27 의 do-od 구조는 Promela 루프를 구현하는데, case 라벨에 expression 들을 허용하는 switch 문을 담고 있는 C for (;;) 루프로 생각될 수 있습니다. (::: 접두사를 갖는) 조건 블락들은 비결정론적으로 스캔됩니다만, 이 경우에는 한번에 하나의 조건만이 참이 될 것입니다. Line 22-25 에 있는 do-od 의 첫번째 블락은 i-번째 카운터 증가 프로세스의 progress 셀을 초기화하고, i-번째 카운터 증가 프로세스를 수행시키고, 변수 i 를 증가시킵니다. line 26 에 있는 do-od 의 두번째 블락은 이 프로세스들이 모두 시작되면 루프를 빠져나옵니다.

Line 29-39 의 어토믹 블락 또한 프로그레스 카운터를 더하는, 비슷한 do-od 루프를 담고 있습니다. Line 38 의 assert () 문은 모든 프로세스가 완료되었는지, 그렇다면 모든 카운트가 정확히 기록되었는지 검증합니다.

독자 여러분은 이 프로그램들을 다음과 같이 빌드하고 실행해 볼 수 있습니다:

```

spin -a increment.spin # Translate the model to C
cc -DSAFETY -o pan pan.c # Compile the model
./pan # Run the model

```

이 수행의 결과로 나올 수 있는 출력이 Figure 12.2 에 보여져 있습니다. 첫번째 줄은 우리의 단정이 깨졌음을 이야기 합니다 (어토믹하지 않은 값 증가로 인해 예상되었던 대로입니다!). 두번째 줄은 어떻게 이 단정이 깨졌는지에 대한 설명을 trail 파일에 썼음을 이야기 합니다. “Warning” 줄은 우리의 모델에 있어서 모든 것이 좋지 않았음을 반복합니다. 두번째 문단은 진행된 상태 탐색의 타입을 설명하는데, 이 경우에는 단정 위반과 무효한 종료 상태들이었습니다. 세번째 문단은 상태 크기에 대한 통계를 보여줍니다: 이 작은 모델은 45개의 상태만을 가졌습니다. 마지막 줄은 메모리 사용량을 보입니다.

trail 파일 안의 정보는 다음의 커맨드를 통해 사람이 읽을 수 있는 형태로 만들어질 수 있습니다:

```
spin -t -p increment.spin
```

이는 Figure 12.3 에 보여진 것과 같은 결과를 보일 겁

```

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

니다. 보여지듯이, 초기화 블락의 첫번째 부분이 두개의 카운터 증가 프로세스들을 생성했고, 두 프로세스는 모두 카운터 값을 가져간 후에 값을 증가하고 다시 저 장시켰으며, 그중 하나의 카운트를 읽었습니다. 그리고 나서, 전체 상태가 표시된 후에 단정문이 판정되었습니다.

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    +
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

이 예를 값 증가 프로세스의 코드를 Figure 12.4에 보인 것처럼 고치기는 쉽습니다. Promela statement들은 어토믹하기 때문에, 단순히 두개의 statement들을 `counter = counter + 1`로 바꿀 수도 있겠습니다. 어떤 경우든, 이 수정된 모델을 돌려보면 Figure 12.5에 보여진 것처럼 에러 없는 상태 공간 횡단을 보여줍니다.

Table 12.1은 (NUMPROCS로 재정의된) 모델링된 카운터 증가 프로세스의 갯수에 대해 상태의 갯수와 소비된 메모리의 양을 보입니다:

따라서 불필요하게 커다란 모델들을 수행해 보는 것은 약간 권장되지 않습니다, 비록 652MB는 현대의 데스크탑과 랩탑 기계의 한계 내에 있긴 하지만요.

이 예제를 두고서, Promela 모델들을 분석하는데 사용되는 커맨드들을 깊게 알아보고 더 자세한 예제들을 봅시다.

# 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

소스 파일 `qrcu.spin`을 가지고서, 다음과 같은 커맨드들을 사용할 수 있습니다:

- `spin -a qrcu.spin` 상태 머신을 완전히 탐색하는 `pan.c` 파일을 만들어냅니다.
- `cc -DSAFETY -o pan pan.c` 생성된 상태 머신 탐색을 컴파일 합니다. `-DSAFETY`는 단정문만을 (그리고 `never` statement들을) 가지고 있다면 적절한 최적화를 만들어냅니다. 만약 당신이 `liveness`, `fairness`, 또는 `forward-progress` 체크를 가지고 있다면, `-DSAFETY` 없이 컴파일을 해야 할 수도 있습니다. 당신이 그것을 사용할 수 있으면서 `-DSAFETY`를 사용하지 않는다면, 프로그램은 당신에게 그걸 알릴 것입니다.
- `-DSAFETY`로 만들어지는 최적화들은 일들의 속도를 엄청나게 높여줄 것이므로, 사용할 수 있다면 사용해야 합니다. `-DSAFETY`를 사용할 수 없는 환경 가운데 예를 들어보자면 `-DNP`를 통해 (“non-progress cycles”라고도 알려져 있는) `livelock`을 체크할 때입니다.

- `./pan` 이 커맨드는 정말로 상태 공간을 탐색하게 합니다. 매우 작은 상태 머신을 가지고도 상태의 갯수는 수천개에 달할 수 있으므로, 커다란 메모리를 가진 기계가 필요할 겁니다. 예를 들어, `qrcu.spin`을 3개의 읽기 쓰레드와 2개의 업데이트 쓰레드와 함께 동작시키려면 2.7GB의 메모리가 필요합니다.

당신의 기계가 충분한 메모리를 가지고 있는지 분명치 않다면, 한 윈도우에서는 `top`을 동작시키고 다른 윈도우에서 `./pan`을 수행하세요. 만약 필요하다면 곧바로 실행을 멈출 수 있게 할 수 있도록 포커스는 `./pan` 윈도우에 두세요. CPU 시간이 100% 아래로 떨어지기 시작하면, `./pan`의 수행을 중지시키세요. 당신이 `./pan`을 수행중인 윈도우로부터 포커스를 옮겨 두었다면, 윈도우 시스템이 당신을 위해 뭔가 하기 위해 충분한 메모리를 가질 수 있을 때까지 긴 시간을 기다려야 할 겁니다.

출력을 캡쳐해 두는 것을 잊지 마세요, 특히나 당신이 원격의 기계에서 작업하고 있다면요.

만약 당신의 모델이 forward-progress 체크를 포함하고 있다면, ./pan 에 커맨드 라인 인자로 -f 를 줌으로써 “weak fairness” 를 활성화 시켜야 할 수 있을 겁니다. 당신의 forward-progress 체크가 accept 라벨과 관련되어 있다면, -a 인자 또한 필요할 겁니다.

- spin -t -p qrcu.spin 에러를 만나게 된 수행 시에 나온 결과 파일인 trail 파일을 받아서 그 에러를 마주하게 된 과정의 시퀀스를 출력합니다. -g 플래그는 또한 변경된 전역 변수들의 값들 또한 포함시킬 것이고, -l 플래그는 변경된 지역 변수들의 값들도 포함시킬 겁니다.

12.1.2.1 Promela Peculiarities

모든 컴퓨터 언어들은 유사한 점들을 가지고 있음에도 불구하고, C, C++, 또는 Java 를 가지고 코딩하던 사람들에게 Promela 는 조금 놀라운 것들을 제공할 겁니다.

1. C 에서, “;” 는 statement 의 종료를 알립니다. Promela 에서는 statement 들을 분리합니다. 다행히도, Sping 의 더 최신 버전들은 “여분의” 세미콜론들에 훨씬 더 너그러워졌습니다.
2. Promela 에서 루프를 만드는 데 사용되는 do 문은 조건들을 갖습니다. 이 do 문은 if-then-else 로 구성된 루프 문과 상당히 닮아 있습니다.
3. C 의 switch 문에서, 맞는 케이스가 존재하지 않는다면, 전체 statement 가 건너뛰어 집니다. Promela 의 같은 기능에서, 잘못 사용된 if 문에서, 맞아 떨어지는 조건이 없다면, 알아들을 수 있는 관련된 에러 메세지 없이 에러를 내게 됩니다. 따라서, 에러 출력이 문제 없는 코드 라인을 가리킨다면, if 나 do 문에서 해당되지 않는 조건을 남겨둔 건 아닌지 확인해 보시기 바랍니다.
4. C 에서 스트레스 테스트를 만들 때, 어떤 사람은 의심되는 오퍼레이션들을 서로에 대해 반복적으로 경주시키곤 할겁니다. Promela 에서, 어떤 사람은 그 대신에 하나의 경주만을 만들텐데, Promela 는 그 한번의 경주로부터 가능한 모든 결과를 탐색할 것이기 때문입니다. 어떤 경우에는 Promela 에서 루프를 돌 필요가 있는데, 예를 들어 여러 오퍼레이션들이 겹치지만, 그렇게 하는게 당신의 상태 공간을 굉장히 증가시키는 경우가 그런 경우입니다.

5. C 에서, 하기 가장 쉬운 일은 루프의 진행 정도를 추적하고 종료하기 위해 루프 카운터를 사용하는 것입니다. Promela 에서, 루프 카운터들은 역 병처럼 방지되어야만 하는데, 그것들은 상태 공간을 폭발적으로 증가시키기 때문입니다. 다른 한편, Promela 에서 무한 루프들은 변수들 가운데 단조적으로 증가하거나 감소하는 것들이 없다면 문제가 없습니다—Promela 는 루프에서 얼마나 수행이 돌아가면 정말로 영향을 끼칠 것인지를 알아챌 것입니다, 자동적으로 그 지점 뒤의 실행을 없애버릴 겁니다.

6. C 고문 테스트 코드에서는 태스크별 제어 변수들을 두는 것이 종종 현명합니다. 그것들은 읽기에 편하고, 테스트 코드를 디버깅 하는데에 매우 도움이 됩니다. Promela 에서 태스크별 제어 변수는 다른 대안이 없을 때에만 사용되어야 합니다. 이를 자세히 보기 위해, 다섯개의 태스크 검증을 해야 하는데 태스크 각각 작업 완료를 나타내는 하나의 비트를 갖는다고 생각해 봅시다. 이는 32개의 상태들을 만들어냅니다. 반면에, 하나의 간단한 카운터만을 사용한다면 6개의 상태만을 가질 것이어서, 다섯 배가 넘는 상태 갯수의 감소를 이루어냅니다. 이 다섯배는 문제처럼 보이지 않을 수도 있는데, 검증 프로그램이 1억 5천만개의 상태들을 10GB 가 넘는 메모리를 소모해 가면서 처리하느라 고생하고 있지 않을 때에는 그럴 겁니다!

7. C 고문 테스트 코드와 Promela 둘 다에서 가장 어려운 일들 중 하나는 좋은 단정문들을 만들어내는 것입니다. Promela 는 또한 never 가 모든 코드 라인들 사이에 복사되어 있는 단정문과 같은 것들에 대해서 주의를 내도록 하는 것도 가능하게 합니다.

8. 분할하고 지배하기는 Promela 에서 상태 공간을 제어하기에 굉장히 도움이 됩니다. 커다란 모델을 두개의 대략적으로 절반씩을 갖는 것들로 분할하는 것은 각각의 절반이 상태 공간의 루트 값만큼의 양을 갖는 결과를 만들 겁니다. 예를 들어, 백만개의 상태가 결합된 모델은 두개의 천개 상태 모델들로 나뉘어질 수 있을 겁니다. Promela 가 두개의 더 작은 모델들을 더 적은 메모리를 가지고 더 빨리 처리할 뿐만 아니라, 두개의 작은 알고리즘들이 사람이 이해하기에 더 쉽습니다.

12.1.2.2 Promela Coding Tricks

Promela 는 프로토콜들을 분석하기 위해 설계되었으므로, 별별 프로그램에 사용하는건 약간 오용하는 것입니다. 다음의 트릭들은 Promela 를 안전하게 오용하는데 도움을 줄 겁니다:

```

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. 메모리 재배치. 전역 변수 x 와 y 를 지역 변수 r1 과 r2 에 복사하는 두개의 statement 가 있는데, 이것들은 그 순서가 중요한데 (ex: 락으로 보호되지 않음), 메모리 배리어를 사용하지 않았다고 생각해 봅시다. 이는 Promela 에서 다음과 같이 모델링될 수 있습니다:

```

1 if
2 :: 1 -> r1 = x;
3 r2 = y
4 :: 1 -> r2 = y;
5 r1 = x
6 fi

```

if 문의 두갈래 경우들은 비결정적으로 선택될 것인데, 둘 다 선택되는 것이 가능하기 때문입니다. 전체 상태 공간이 탐색될 것이므로, 둘 다의 선택들이 결국은 모든 경우들에 대해 만들어질 것입니다.

물론, 이 트릭은 너무 과하게 사용된다면 상태 공간을 폭증시켜 버릴 수 있습니다. 또한, 이 트릭은 가능한 재배치들을 예측할 것을 필요로 합니다.

2. 상태 감축. 복잡한 단정문들을 가지고 있다면, 그들을 atomic 하에서 수행하세요. 무엇보다, 그들은 알고리즘의 한 부분이 아닙니다. 복잡한 단정문의 한 예는 (뒤에서 더 자세히 다루겠습니다만) Figure 12.6 에 보여져 있습니다.

```

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

이 단정문을 어토믹하지 않게 수행할 이유가 없는 데, 이는 알고리즘의 실제 부분이 아니기 때문입니다. 각각의 statement 가 상태에 영향을 끼치므로, Figure 12.7 에 보여진 것처럼 복잡한 단정문들을 atomic 안에 집어넣음으로써 불필요한 상태들의 수를 줄일 수 있습니다.

3. Promela 는 함수를 제공하지 않습니다. 그대신에 C 전처리기 매크로들을 사용해야만 합니다. 하지만, 조합에 의한 상태 수의 폭증을 막기 위해 그것들은 조심스럽게 사용되어야만 합니다.

이제 더 복잡한 예제들을 볼 준비가 되었습니다.

12.1.3 Promela Example: Locking

락들은 일반적으로 유용하기 때문에, Figure 12.8 에서 보여진 것처럼 여러 Promela 모델들에 include 될 수 있는 lock.h 에서 spin_lock() 과 spin_unlock() 매크로들을 제공합니다. spin_lock() 매크로는 line 3 의 단 하나의 조건 “1” 덕분에 line 2-11 의 무한한 do-od 루프를 가지고 있습니다. 이 루프의 몸통은 하나의 if-fi 문을 담고 있는 하나의 어토믹 블락입니다. 이 if-fi 문은 루프를 돌기보다는 한번의 패스만을 취한다는 점을 제외하고는 do-od 문과 비슷합니다. 락이 line 5 에서 잡혀있지 않다면 line 6 에서 이를 획득하고 line 7 에서 감싸고 있는 do-od 루프를 깨고 나갑니다 (그리고 어토믹 블락에서도 나갑니다). 한편으로는, 만약 락이 line 8 에서 이미 잡혀 있었다면, 아무 일도 하지 않고 (skip), if-fi 문과 어토믹 블락을 빠져나오고 그 바깥 루프의 다음 반복을 진행해서 락이 획득 가능해질 때까지 반복하게 됩니다.

spin_unlock() 매크로는 단순히 이 락을 더이상 잡혀 있지 않았다고 표시합니다.

Promela 는 완전한 순서 규칙을 가정하기 때문에 메모리 배리어들은 필요치 않다는 점을 알아두세요. 어떤 Promela 상태에서도, 모든 프로세스들은 현재 상태와 우리가 현재의 상태에 도달하게 되는 과정에서의 상태

```

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

변화의 순서들에 동의하게 됩니다. 이는 (MIPS 와 PA-RISC 같은) 일부 컴퓨터 시스템들에서 사용되는 “sequentially consistent” 메모리 모델과 비슷합니다. 앞서 언급되었듯이, 그리고 뒤의 예제에서 알아보게 되듯이, 약한 메모리 순서 규칙은 명시적으로 코딩되어야만 합니다.

이 매크로들은 Figure 12.9에 보여진 Promela 코드에 의해 테스트 되었습니다. 이 코드는 line 3에서의 N_LOCKERS 정의로 정의된 락킹 프로세스의 숫자에 의해 값 증가를 테스트하는데 사용되었던 코드와 비슷합니다. 뮤텍스 자체는 line 5에 정의되었고, 락 소유자를 추적하기 위한 배열이 line 6에 정의되어 있으며, line 7은 하나의 프로세스만이 락을 잡고 있음을 증명하기 위한 단정문 코드에 사용됩니다.

락을 잡는 프로세스는 line 9-18에 있는데, line 13에서 락을 획득하고 line 14에서 락을 잡았음을 공표하고 line 15에서 락을 잡지 않고 있다고 이야기한 후, line 16

에서 락을 놓습니다.

Line 20-44의 init 블락은 현재 락 잡는 프로세스의 havelock 배열 원소를 line 26에서 초기화하고, 현재 락 잡는 프로세스를 line 27에서 시작시키며, line 28에서 다음 락 잡는 프로세스로 넘어갑니다. 일단 모든 락 잡는 프로세스들이 시작되면, do-od 루프의 수행은 단정문을 체크하는 line 29로 넘어갑니다. Line 30과 31은 제어 변수들을 초기화하고, line 32-40은 어토믹하게 havelock 배열 원소들의 합을 구하고, line 41에서 단정문을 수행하고, line 42에서 루프를 빠져나옵니다.

우리는 앞의 두개의 코드 조각들을 lock.h 와 lock.spin에 각각 집어넣는 것으로 이 모델을 수행할 수 있게 되며, 다음의 커맨드들을 사용해 돌릴 수 있습니다:

```

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

출력되는 결과는 Figure 12.10에 보여진 것과 비슷할 것입니다. 예상되었듯이, 이 수행은 단정문 실패가 없습니다 (“errors: 0”).

Quick Quiz 12.1: 왜 locker에 미치지 못한 statement가 있는 거죠? 이건 전체 상태-공간 탐색이 아니었나요?

Quick Quiz 12.2: 이 예제에 있어서 Promela 코딩 스타일 문제들은 뭐가 있나요?

12.1.4 Promela Example: QRCU

이 마지막 예제는 synchronize_qrcu()의 빠른 수행경로를 더 빠르게 하기 위해 수정된 Oleg Nesterov의 QRCU [Nes06a, Nes06b]를 위한 실제 세계에서의 Promela 사용을 보입니다.

하지만 먼저, QRCU란 무엇일까요?

QRCU 는 극단적으로 낮은 grace period 대기시간이라는 장점을 더 높은 읽기 오버헤드(전역 변수에의 어토믹한 값 증가와 감소)와 맞바꾸기 하는, SRCU [McK06b] 의 한 변종입니다. 읽기 쓰레드가 없다면, grace period 는 1 마이크로세컨드도 되지 않는 시간에 파악되는데, 이는 대부분의 다른 RCU 구현들의 수 밀리세컨드 grace period 대기시간과 비교됩니다.

1. QRCU 도메인을 정의하는 `qrcu_struct` 가 존재합니다. SRCU 처럼(그리고 다른 RCU 변종들과는 달리) QRCU 의 동작은 글로벌하지 않고, 그 대신에 특정한 `qrcu_struct` 에 집중됩니다.
2. QRCU read-side 크리티컬 섹션들을 구분짓는 `qrcu_read_lock()` 과 `qrcu_read_unlock()` 이 있습니다. 연관되는 `qrcu_struct` 는 이 함수들에 넘겨져야 하고, `rcu_read_lock()` 으로부터의 리턴값은 `rcu_read_unlock()` 으로 넘겨져야만 합니다.

예를 들면 다음과 같습니다:

```
idx = qrcu_read_lock(&my_qrcu_struct);
/* read-side critical section. */
qrcu_read_unlock(&my_qrcu_struct, idx);
```

3. 이전부터 존재해온 QRCU read-side 크리티컬 섹션들이 모두 완료될 때까지 기다리는 `synchronize_qrcu()` 기능이 있습니다만, SRCU 의 `synchronize_srcu()` 처럼, QRCU 의 `synchronize_qrcu()` 는 같은 `qrcu_struct` 를 사용하는 read-side 크리티컬 섹션들만을 기다리면 됩니다.

앞의 예에 이어 예를 들면, `synchronize_qrcu(&your_qrcu_struct)` 는 이전의 QRCU read-side 크리티컬 섹션을 기다릴 필요가 없습니다. 반면에, `synchronize_qrcu(&my_qrcu_struct)` 는 기다려야 할 수 있는데, 같은 `qrcu_struct` 를 공유하기 때문입니다.

QRCU 를 위한 리눅스 커널 패치도 만들어졌습니다만 [McK07b], 2008년 4월 현재까지는 리눅스 커널에 포함되지 않았습니다.

QRCU 를 위한 Promela 코드로 돌아와서, 전역 변수들은 Figure 12.11 에 보인 것과 같습니다. 이 예제는 락킹을 사용하므로, `lock.h` 를 `include` 하고 있습니다. 읽기 쓰레드들의 갯수와 쓰기 쓰레드들의 갯수는 두개의 `#define` 문을 통해 변경될 수 있어서, 두개의 조합 증폭 가능한 방법을 제공합니다. `idx` 변수는 `ctr` 배열의 두 원소들 중 무엇이 읽기 쓰레드들에 의해 사용될 것인지 결정하며, `readerprogress` 변수는 단정문이 언제 모든 읽기 쓰레드들이 종료되었는지를 판단할

```
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

수 있게 합니다(QRCU 업데이트는 모든 앞서 존재한 읽기 쓰레드들이 그들의 QRCU read-side 크리티컬 섹션을 완료하기 전까지는 완료될 수 없기 때문입니다). `readerprogress` 배열의 원소들은 다음과 같이 값을 가져서 연관된 읽기 쓰레드의 상태를 알립니다:

1. 0: 시작되지 않았음.
2. 1: QRCU read-side 크리티컬 섹션 안에 있음.
3. 2: QRCU read-side 크리티컬 섹션을 끝냈음.

마지막으로, `mutex` 변수는 업데이트 쓰레드들의 느린 수행 경로들을 직렬화하는데 사용됩니다.

```
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]-- }
20 }
```

Figure 12.12: QRCU Reader Process

QRCU 읽기 쓰레드들은 Figure 12.12 에 보여진 `qrcu_reader()` 프로세스에 의해 모델링 됩니다. `do-od` 루프가 line 5-16 에 있는데, 하나의 “1” 조건을 line 6 에 가지고 있어서 이를 무한 루프로 만들어 줍니다. Line 7 은 글로벌 인덱스의 현재 값을 가져오고, line 8-15 는 그 값이 0이 아니었다면 이를 원자적으로 증가(`atomic_inc_not_zero()`) 시킵니다(그리고 루프를 나옵니다). Line 17 은 RCU read-side 크리티컬 섹션으로의 진입을 표시하고, line 18 은 이 크리티컬 섹션으로부터 나오는 것을 표시하는데, 두 라인 모두 우리가 뒤에서 마주하게 될 `assert()` 문을 위해서입니다.

Line 19 는 우리가 증가시켰던 같은 카운터를 어토믹하게 감소시키는데, 이렇게 함으로써 RCU read-side 크리티컬 섹션을 빠져나오게 됩니다.

```

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

Figure 12.13에 보인 C 전처리기 매크로는 약한 메모리 순서 규칙을 에뮬레이션하기 위해 두개의 카운터들의 합을 구합니다. Line 2-13은 카운터들 중 하나를 가져오고, line 14는 나머지 하나를 가져와서 그것들을 더 합니다. 어토믹 블락은 하나의 do-od 문으로 구성되어 있습니다. 이 do-od 문 (line 3-12)은 line 4 와 8에 있는 두개의 무조건적인 브랜치들을 가지고 있다는 점에서 일반적이지 않은데, 이는 Promela 가 비결정적으로 두 개의 브랜치 중 하나를 선택하게 합니다 (하지만 다시 말하지만, 전체 상태 공간 탐색은 Promela 가 결국은 모든 가능한 상황의 선택지를 만들게 할 겁니다). 첫번째 브랜치는 0번째 카운터를 가져오고 i 를 1로 설정하고 (line 14가 첫번째 카운터를 가져오도록), 두번째 브랜치는 그 반대 일을 해서, 첫번째 카운터를 가져오고 i 를 0으로 설정합니다 (line 14에서 두번째 카운터를 가져오도록).

Quick Quiz 12.3: o) do-od 문을 좀 더 간단하게 코딩하는 방법은 없을까요? ■

sum_unordered 매크로와 함께, 우리는 이제 Figure 12.14에 보여진 update-side 프로세스로 넘어갈 수가 있게 되었습니다. 이 update-side 프로세스는 line 7-57의 관련된 do-od 루프를 애매하게 반복하게 됩니다. 이 루프를 통하는 각각의 패스는 먼저 line 12-21에서 글로벌한 readerprogress 배열을 로컬의 readerstart 어레이로 스냅샷 땡니다. 이 스냅샷은 line 53에서의 단정문에 사용될 겁니다. Line 23은 sum_unordered 를 호출하고, 빠른 수행경로가 잠재적으로 사용 가능하다면 line 24-27에서 sum_unordered 를 다시 호출합니다.

Line 28-40은 필요하다면 느린 수행 경로의 코드를 수행하는데, line 30과 38에서 update-side 락을 각각 획득하고 해제하며, line 31-33에서 인덱스를 뒤집고, line 34-37에서 모든 이전부터 존재한 읽기 쓰레드들이 완료되길 기다립니다.

```

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             /* Snapshot reader state. */
10            atomic {
11                i = 0;
12                do
13                    :: i < N_QRCU_READERS ->
14                        readerstart[i] = readerprogress[i];
15                        i++;
16                    :: i >= N_QRCU_READERS ->
17                        break
18                    od
19            }
20
21            sum_unordered;
22            if
23                :: sum <= 1 -> sum_unordered
24                :: else -> skip
25            fi;
26            if
27                :: sum > 1 ->
28                    spin_lock(mutex);
29                    atomic { ctr[!idx]++ }
30                    idx = !idx;
31                    atomic { ctr[!idx]-- }
32                    do
33                        :: ctr[!idx] > 0 -> skip
34                        :: ctr[!idx] == 0 -> break
35                    od;
36                    spin_unlock(mutex);
37                :: else -> skip
38            fi;
39
40            /* Verify reader progress. */
41
42            atomic {
43                i = 0;
44                sum = 0;
45                do
46                    :: i < N_QRCU_READERS ->
47                        sum = sum + (readerstart[i] == 1 &&
48                            readerprogress[i] == 1);
49                            i++;
50                        :: i >= N_QRCU_READERS ->
51                            assert(sum == 0);
52                            break
53                        od
54                }
55            od
56        }
57    od
58 }

```

Figure 12.14: QRCU Updater Process

Line 44-56에서는 이제 readerprogress 배열의 현재 값들과 readerstart 배열 안의 앞서 수집된 값들과 비교하며, 이 업데이트 전에 시작된 읽기 쓰레드가 여전히 진행중이라면 단정문이 실패하도록 합니다.

Quick Quiz 12.4: 왜 line 12-21과 line 44-56에 어토믹 블락이 있나요, 그 어토믹 블락들 안의 오퍼레이션들은 현존하는 제품화된 마이크로프로세서 중 어떤 것도 어토믹한 구현을 제공하지 않는는데 말이죠? ■

Quick Quiz 12.5: Line 24-27에서 카운터들을 다시 합해보는 것은 정말로 필요한 건가요? ■

```

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_UPDATORS ->
18            run qrcu_updater(i);
19            i++;
20            :: i >= N_QRCU_UPDATORS -> break
21        od;
22    }
23 }
```

Figure 12.15: QRCU Initialization Process

이제 남은건 Figure 12.15의 초기화 블락 뿐입니다. 이 블락은 단순히 카운터 쌍을 line 5-6에서 초기화하고, line 7-14에서 읽기 프로세스들을 시작시킨 후, line 15-21에서 업데이트 프로세스들을 시작시킵니다. 이는 상태 공간의 크기를 줄이기 위해 모두 하나의 어토믹 블락 안에서 수행됩니다.

12.1.4.1 Running the QRCU Example

○ QRCU 예제를 수행하기 위해서는, 앞 섹션에서의 코드 조각을 `qrcu.spin`이라는 이름의 하나의 파일로 합치고, `spin_lock()`과 `spin_unlock()`의 정의를 `lock.h`라는 이름의 파일로 옮겨야 합니다. 그리고 나서는 다음의 커맨드를 사용해서 QRCU 모델을 빌드하고 수행시킬 수 있습니다:

```
spin -a qrcu.spin
cc -DSAFETY -o pan pan.c
./pan
```

수행 결과 나오는 출력은 이 모델이 Table 12.2에 나온 모든 케이스들을 통과함을 보여줍니다. 이제, 이 케

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

이스를 세개의 읽기 쓰레드들과 세개의 업데이트 쓰레드들과 돌려보는 것도 좋을 것입니다만, 간단한 추정이 이는 최선의 경우에도 수십 테라바이트의 메모리를 필요로 할 것인 점을 이야기 합니다. 그럼, 뭘 해야 할까요? 여기 몇가지 가능한 방법들이 있습니다:

1. 더 적은 수의 읽기 쓰레드들과 업데이트 쓰레드들이 일반적인 경우를 증명하는데 충분한지 보세요.
2. 일일이 정확성의 증명을 구하세요.
3. 더 적합한 도구를 사용하세요.
4. 분할하고 정복하세요.

다음의 섹션은 이런 방법들 각각을 알아봅니다.

12.1.4.2 How Many Readers and Updaters Are Really Needed?

한가지 방법은 `qrcu_updater()`를 위한 Promela 코드를 주의깊게 들여다보고 유일한 전역적 상태 변경은 락 아래 일어남을 알아차리는 것입니다. 따라서, 한번에 하나의 업데이트 쓰레드만이 읽기 쓰레드들이나 다른 업데이트 쓰레드들에게 보일 수 있는 상태 변경을 가하고 있을 수 있습니다. 이는, Promela가 전체 상태 공간 검색을 한다는 사실로 인해, 어떤 시퀀스의 상태 변경들도 하나의 업데이트 쓰레드에 의해 순차적으로 이뤄질 것임을 의미합니다. 따라서, 최대 두개의 업데이트 쓰레드들이 필요합니다: 하나는 상태를 바꾸기 위해, 나머지 하나는 헛갈려 하기 위해.

읽기 쓰레드들과 함께 있는 상황은 좀 덜 분명한데, 각각의 읽기 쓰레드는 하나의 read-side 크리티컬 섹션만을 만들고 종료되기 때문입니다. 빠른 수행경로는 카운터들에서 최대 0과 1 만을 읽을 수 있다는 사실로 인해 유용한 읽기 쓰레드들의 수는 제한되어 있다고 반론을 제기할 수도 있겠습니다. 실제로, 이는 수사에 있어서 알찬 방법으로, 다음 섹션에서 이야기하는 전체 정확성 증명을 이끌게 됩니다.

12.1.4.3 Alternative Approach: Proof of Correctness

비형식적인 증명 [McK07b] 는 다음과 같습니다:

1. `synchronize_qrcu()` 가 너무 일찍 종료되려면, 정의에 의해 `synchronize_qrcu()` 의 전체 수행 사이에 최소 하나의 읽기 쓰레드가 존재해야 합니다.
2. 이 읽기 쓰레드에 연관된 카운터는 이 시간 간격 동안 최소 1이 되었을 겁니다.
3. `synchronize_qruc()` 코드는 최소 하나의 카운터는 최소 1이 되도록 강제합니다.
4. 따라서, 어떤 시점에서든, 카운터들 가운데 하나는 최소 2가 되거나, 두개의 카운터들이 최소 1을 가질 겁니다.
5. 하지만, `synchronize_qrcu()` 빠른 수행 경로 코드는 한번에 하나의 카운터의 값밖에 읽지 못합니다. 따라서 빠른 수행 경로 코드가 첫번째 카운터를 값이 0일 동안 읽어오지만 카운터 뒤집기에서는 경주 상황이 벌어져서 두번째 카운터가 1로 보이는 경우가 있을 수 있습니다.
6. 그런 경주 조건 동안에 존재하는 최대 하나의 읽기 쓰레드가 있을 수 있는데, 그렇지 않다면 그 합은 2 이상이 될 것이기 때문인데, 이는 업데이트 쓰레드가 느린 수행경로를 취하도록 만들 것입니다.
7. 하지만 그 경주 상황이 빠른 수행경로의 첫번째 카운터 읽기에서 발생한다면, 그리고 그 두번째 읽기에서 다시 일어난다면, 두개의 카운터 뒤집기가 있었어야만 합니다.
8. 업데이트 쓰레드는 카운터를 한번만 뒤집으므로, 그리고 업데이트 쪽 락은 두개의 업데이트 쓰레드들이 동시적으로 카운터를 뒤집는 것을 방지하므로, 빠른 수행경로 코드가 뒤집기와 두번 경주상황을 만들 수 있는건 첫번째 업데이트 쓰레드가 완료 했을 때입니다.
9. 하지만 첫번째 업데이트 쓰레드는 모든 앞서 존재한 읽기 쓰레드들이 완료되기 전까지는 완료될 수 없습니다.
10. 따라서, 카운터를 두번 뒤집으며 빠른 수행경로 경주상황이 완료된다면, 모든 앞서 존재한 읽기 쓰레드들은 완료되었어야만 하고, 따라서 빠른 수행경로를 취해도 안전합니다.

물론 모든 병렬 알고리즘에 이런 간단한 증명이 통하지는 않습니다. 그런 경우에 있어서는 더 적합한 도구들을 모을 필요가 있을 겁니다.

12.1.4.4 Alternative Approach: More Capable Tools

Promela 와 Spin 이 상당히 유용하긴 하지만, 훨씬 더 적절한 도구들도 사용할 수 있는데, 특히 하드웨어를 검증할 때 그렇습니다. 이는 당신의 알고리즘이 낮은 단계의 병렬 알고리즘들이 종종 그렇듯이 하드웨어 설계용 VHDL 언어로 변환할 수 있다면, 이 도구들을 코드에 적용해 보는 것도 가능합니다(예를 들어, 최초의 realtime RCU 알고리즘을 위해 이 방법이 사용되었습니다). 하지만, 그런 도구들은 상당히 비싼 비용을 필요로 할 수 있습니다.

흔한 멀티프로세싱의 장점이 결국은 멋진 상태 공간 최소화 기능을 가지고 있는 강력한 프리 소프트웨어 모델 검증기를 만들어낼 수 있겠지만, 이는 현재 여기에 커다란 도움이 되지는 않습니다.

별개로, Spin 은 고정된 양의 메모리를 필요로 하는 대강의 탐색을 지원합니다만, 저는 병렬 알고리즘을 검증할 때 대략적 방법을 신뢰할 수는 없었습니다.

또 다른 방법들은 분할하고 정복하기가 될겁니다.

12.1.4.5 Alternative Approach: Divide and Conquer

커다란 병렬 알고리즘이 개별적으로 증명될 수 있는, 더 작은 조각들로 조각내는 것이 가능한 경우가 종종 있습니다. 예를 들어, 100억개의 상태를 갖는 모델은 두개의 100,000 개의 상태 모델들로 쪼개질 수 있습니다. 이 방법은 Promela 와 같은 도구들이 알고리즘을 검증하는 것을 더 쉽게 해줄 뿐 아니라, 알고리즘들을 더 이해하기 쉽게 만들어 줄 수 있습니다.

12.1.4.6 Is QRCU Really Correct?

QRCU 는 정말로 올바르게 동작할까요? 우리는 Promela 기반의 기계적 증명과 손으로 하는 증명을 했고 둘 모두 그렇다고 이야기했습니다. 하지만, Algave 등 [AKT13] 의 최근 논문은 다르게 이야기합니다(해당 논문 page 12 아래쪽의 Section 5.1 을 보세요). 뭐가 맞을까요?

저는 Algave, Korenig, 그리고 Tautschig 가 문제를 찾아냈던 코드를 들여다볼 수는 없었기 때문에 알수가 없습니다. 해당 저자들은 그런 동시성 벤치마크들이 그들이 연구를 시작하게 된 실제 세계의 예제와 동일할 필요는 없다고 이야기 했지만요. 어떻게 보면, QRCU 는 리눅스 커널에 포함되어지지도 않았고, 제가 알기로는 어떤 다른 제품 소프트웨어에 사용되지도 않았기 때문에 사실 큰 문제가 되지는 않습니다.

하지만, QRCU를 사용할 생각이라면, 조심하시기 바랍니다. QRCU의 정확성 증명은 그것 자체로 올바를 수도, 그렇지 않을 수도 있습니다. 이는 정형적 검증이 완전하게 테스트를 대체할 수는 없을 것이라 생각되는 한가지 이유입니다.

Quick Quiz 12.6: 여기에 설명된 QRCU 알고리즘의 정확성에 대한 두개의 독립적인 증명을 가지고 있고, 올바르지 않음에 대한 증명이 다른 알고리즘은 어떤지를 다루고 있는데, 왜 여전히 의문의 여지가 남는 거죠? ■

12.1.5 Promela Parable: dynticks and Preemptible RCU

2005년 8월부터 시작된 -rt 패치셋 [Mol05]의 RCU 구현과 비슷한, RCU의 preemption 가능한 변종이 2008년 초에 메인라인 리눅스에 realtime 워크로드에 대한 지원과 함께 받아들여졌습니다. Preemption 가능한 RCU는 기존의 RCU 구현들은 RCU read-side 크리티컬 섹션 내에서의 preemption을 불가능하게 했기 때문에 지난친 real-time 대기시간을 초래했기 때문에 real-time 워크로드를 위해 필요했습니다.

하지만, 기존의 -rt 구현의 한가지 단점은 각각의 grace period가 모든 각각의 CPU 위에서, 설령 그 CPU가 저전력의 “dynticks-idle” 상태에 있더라도, 작업을 해야하고, 그로 인해 RCU read-side 크리티컬 섹션들을 수행하기가 불가능하다는 점이었습니다. Dynticks-idle 상태의 아이디어는 idle CPU들은 에너지를 아끼기 위해 물리적으로 성능을 낮춰야 한다는 것입니다. 짧게 말해서, preemption 가능한 RCU는 최신 리눅스 커널에서의 가치있는 에너지 절약 기능을 무효화 시킬 수 있다는 점입니다. Josh Triplett과 Paul McKenney가 CPU들이 RCU graceperiod 사이에도 저전력 상태를 유지할 수 있는 (따라서 리눅스 커널의 에너지 절약 기능을 유지할 수 있는) 방법에 대해서 토론을 가졌습니다만, Steve Rostedt가 새로운 dyntick 구현을 -rt 패치셋의 preemption 가능한 RCU와 결합시키기 전까지는 해결책이 나오지 않았습니다.

이 조합은 Steve의 시스템들 중 하나를 부팅 과정에서 멈춰있게 만들었고, 따라서 10월에, Paul은 preemption 가능한 RCU의 grace-period 처리 부분에 dynticks에 친화적인 수정사항을 코딩했습니다. Steve는 `irq_enter()`와 `irq_exit()` 인터럽트 진입/퇴장 함수들로부터 호출되는 `rcu_irq_enter()`와 `rcu_irq_exit()` 인터페이스들을 코딩했습니다. 이 `rcu_irq_enter()`와 `rcu_irq_exit()` 함수들은 dynticks-idle CPU들이 RCU read-side 크리티컬 섹션들을 포함하고 있는 인터럽트 핸들러에 의해 순간적으로 성능을 올리게 되는 상황을 안정적으로 처리하기 위해 필요했습니다. 이런 변경 사항들과 함께, Steve의 시스템은 안정

적으로 부팅되었습니다만, Paul은 첫번째 시도만에 코드가 올바르게 작성되지는 않았을 것이라는 가정 하에 지속적으로 코드를 검사하기를 계속했습니다.

Paul은 2007년 10월부터 2008년 2월까지 반복적으로 코드를 리뷰했고 거의 매번 최소 하나의 버그는 찾아냈습니다. 한 경우에는, Paul은 심지어 그 버그가 실체가 없는 것이라 것을 깨닫기 전에도 수정사항을 코딩하고 테스트하기도 했고, 사실 모든 경우들에 있어서, “버그”는 실체가 없는 것으로 드러났습니다.

2월 말 즈음에, Paul은 이 게임에 지쳐버렸습니다. 따라서 그는 Section 12에서 이야기한 것처럼 Promela와 spin [Hol03]의 도움을 받기로 했습니다. 다음의 내용은 일곱개의 갈수록 현실적인 Promela 모델들을 보이는데, 그 중 마지막의 것은 상태 공간으로 약 40GB의 메인 메모리를 소비합니다.

더 중요한건, Promela와 Spin은 제게 매우 미묘한 버그를 찾아줬습니다!

Quick Quiz 12.7: 와우, 거 참 대단하네요! 이제, 저는 40GB의 메인 메모리를 가진 기계가 없다면 뭘 해야 하는거죠??? ■

여전히 더 나은 방향은 더 작은 상태 공간을 갖는, 더 간단하고 더 빠른 알고리즘을 사용하는 것일 겁니다. 그보다도 더 나은 방법은 그 정확성이 평상시의 관찰자에 의해서도 분명할 수 있을 만큼 간단한 알고리즘들을 사용하는 것인 것입니다!

Section 12.1.5.1은 preemption 가능한 RCU의 dynticks 인터페이스에 대해 전체적으로 살펴보고, Section 12.1.6와 Section 12.1.6.8은 이로써 (다시) 배운 교훈들을 나열합니다.

12.1.5.1 Introduction to Preemptible RCU and dynticks

Per-CPU `dynticks_progress_counter` 변수가 dyntick과 preemption 가능한 RCU 사이의 인터페이스의 중심 역할을 합니다. 이 변수는 연관된 CPU가 dynticks-idle 모드일 때에는 짹수를 가지게 되고, 그렇지 않을 때에는 훌수를 갖게 됩니다. CPU는 다음의 세가지 이유로 dynticks-idle 모드를 빠져나갑니다:

1. 태스크를 수행하기 시작하기 위해서,
2. 중첩되어 있을 수 있는 인터럽트 핸들러들 가운데 가장 바깥쪽의 것에 들어가기 위해서, 그리고
3. NMI 핸들러에 들어갈 때.

Preemption 가능한 RCU의 grace-period 장치는 언제 dynticks-idle CPU가 안전하게 무시될 수 있을지를 판단하기 위해서 `dynticks_progress_counter` 변수의 값을 샘플링 합니다.

다음의 세개의 섹션들은 태스크 인터페이스, 인터럽트/NMI 인터페이스, 그리고 grace-period 장치에 의한 dynticks_progress_counter 변수의 사용에 대해서 알아봅니다.

12.1.5.2 Task Interface

특정 CPU 가 더이상 수행할 태스크가 존재하지 않아 dynticks-idle 모드로 들어갈 때에, 해당 CPU 는 `rcu_enter_nohz()` 를 호출합니다:

```
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 }
```

이 함수는 단순히 `dynticks_progress_counter` 의 값을 증가시키고 그 결과가 짹수인지를 체크합니다만, 그전에 먼저, `dynticks_progress_counter` 의 새로운 값을 보게될 다른 CPU 가 앞의 RCU read-side 크리티컬 섹션의 완료 역시 보게 될 것을 보장하기 위해 메모리 배리어를 실행합니다.

비슷하게, dynticks-idle 모드에 있는 CPU 가 새로운 수행가능한 태스크를 실행하기를 시작할 준비를 할 때에는, `rcu_exit_nohz()` 를 수행합니다:

```
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 }
```

이 함수는 한번더 `dynticks_progress_counter` 의 값을 증가시킵니다만, 어떤 다른 CPU 가 뒤의 RCU read-side 크리티컬 섹션의 결과를 보게 된다면 그 CPU 는 `dynticks_progress_counter` 의 증가된 값 옆비로 수 있을 것을 보장하기 위해 값 증가에 이어 메모리 배리어를 실행시킵니다. 마지막으로, `rcu_exit_nohz()` 는 값 증가의 결과가 훌수임을 확인합니다.

12.1.5.3 Interrupt Interface

`rcu_irq_enter()` 와 `rcu_irq_exit()` 함수들은 인터럽트/NMI 진입과 종료를 각각 처리합니다. 물론, 중첩된 인터럽트들 또한 적절하게 처리되어야만 합니다. 중첩 인터럽트의 가능성은 인터럽트나 NMI 핸들러로의 진입 시에 (`rcu_irq_enter()` 에서) 값이 증가되고 종료 시에 (`rcu_irq_exit()` 에서) 값이 감소되는 두번째 per-CPU 변수, `rcu_update_flag` 에 의해 처리됩니다. 추가적으로, 앞서서부터 존재해온 `in_interrupt()` 기능은 인터럽트/NMI 가 가장 바깥의 것인지 중첩된 것인지 구별해 내는데에 사용됩니다.

인터럽트 진입은 아래 보여진 `rcu_irq_enter()` 에서 처리됩니다:

```
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 는 현재 CPU 의 숫자를 가져오고, line 5 와 6 는 `rcu_update_flag` 중첩 카운터가 0이 아니라면 그 값을 증가시킵니다. Line 7-9 는 자신이 인터럽트의 가장 바깥 단계인지를 체크하고, 만약 그렇다면 `dynticks_progress_counter` 가 증가되어야 합니다. 그렇다면, line 10 에서 `dynticks_progress_counter` 의 값을 증가시키고, line 11 에서 메모리 배리어를 실행한 후, line 12 에서 `rcu_update_flag` 의 값을 증가시킵니다. `rcu_exit_nohz()` 에서와 마찬가지로, 이 메모리 배리어는 이 인터럽트 핸들러 안에서의 RCU read-side 크리티컬 섹션의 영향을 본 CPU 는 `dynticks_progress_counter` 의 값의 증가 결과 역시 볼 수 있을 것을 보장합니다.

Quick Quiz 12.8: 왜 간단하게 `rcu_update_flag` 의 값을 증가시키고, `rcu_update_flag` 의 기존 값이 0일 경우에만 `dynticks_progress_counter` 의 값을 증가시키는 방식을 사용하지 않는거죠???

Quick Quiz 12.9: 하지만 line 7 이 우리가 가장 바깥의 인터럽트에 있음을 알게 된다면, 우린 항상 `dynticks_progress_counter` 의 값을 증가시켜야 하는 것 아닌가요??

인터럽트 종료는 `rcu_irq_exit()` 에 의해 비슷하게 처리됩니다:

```
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        WARN_ON(per_cpu(dynticks_progress_counter,
12                         cpu) & 0x1);
13    }
14 }
```

Line 3 은 앞에서처럼 현재 CPU 의 번호를 가져옵니다. Line 5 는 `rcu_update_flag` 의 값이 0이 아닌지 확인해보고, 그렇지 않다면 곧바로 (함수의 마지막으로 제어를 넘김으로써) 리턴합니다. 그렇지 않다면,

line 6 부터 12 가 수행됩니다. Line 6 은 `rcu_update_flag` 의 값을 감소시키고, 그 결과가 0이 아니라면 리턴합니다. Line 8 은 우리가 정말로 중첩된 인터럽트들 가운데 가장 마지막 단계를 빠져나가고 있음을 검증하고, line 9 에서 메모리 배리어를 수행한 후, line 10 에서 `dynticks_progress_counter`의 값을 증가시키고, line 11 과 12 에서 이 변수가 이제는 짹수임을 검증합니다. `rcu_enter_nohz()` 에서와 같이, 메모리 배리어는 `dynticks_progress_counter`의 값 증가를 본 다른 CPU 는 해당 인터럽트 핸들러 내에서의 (`rcu_irq_exit()` 호출을 앞서 수행된) RCU read-side 크리티컬 섹션의 결과 역시 보게 될 것을 보장합니다.

이 두 섹션들은 `dynticks_progress_counter` 변수가 태스크들과 인터럽트, NMI 에 의해 `dynticks-idle` 모드로 들어갈 때와 빠져나올 때에 어떻게 관리되는지를 설명했습니다. 다음 섹션은 이 변수가 `preemption` 가능한 RCU 의 grace-period 장치에 의해 어떻게 사용되는지를 설명합니다.

12.1.5.4 Grace-Period Interface

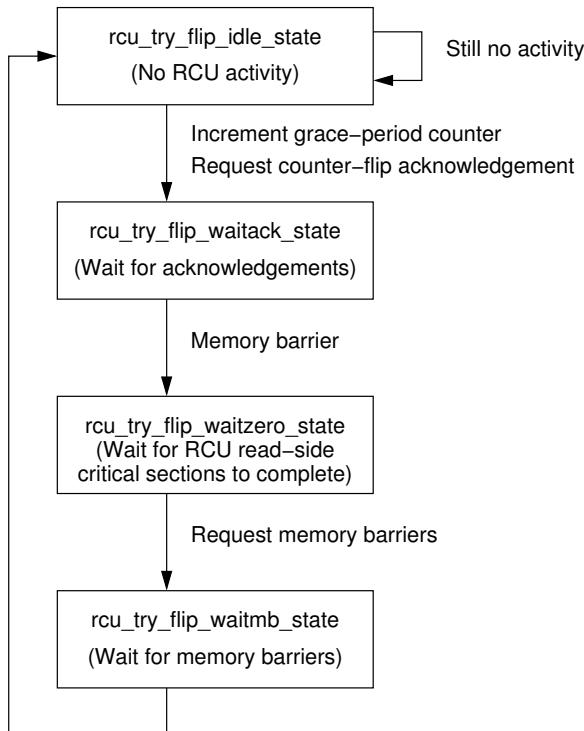


Figure 12.16: Preemptible RCU State Machine

Figure 12.16 에 보인 네개의 `preemption` 가능한 RCU grace-period 상태등 가운데, `rcu_try_flip_`

`waitack_state()` 와 `rcu_try_flip_waitmb_state()` 상태들만이 다른 CPU 들이 응답하길 기다려야 합니다.

물론, 한 CPU 가 `dynticks-idle` 상태에 있다면, 그걸 기다릴 수는 없습니다. 따라서, 이 두개의 상태들 중 하나에 들어가기 직전에, 앞의 상태는 각 CPU 들의 `dynticks_progress_counter` 변수의 값들의 스텝샷을 떠놓고, 그 스텝샷을 또 다른 per-CPU 변수인 `rcu_dyntick_snapshot`에 넣어둡니다. 이는 아래에 보여진 것과 같은 `dyntick_save_progress_counter()` 를 호출함으로써 이뤄집니다:

```

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()` 상태는 아래에 보여진 것과 같은 `rcu_try_flip_waitack_needed()` 를 호출합니다:

```

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 || (snap & 0x1) == 0)
13        return 0;
14    return 1;
15 }
  
```

Line 7 과 8 은 `dynticks_progress_counter`의 현재 버전과 스텝샷 버전을 각각 가져옵니다. Line 9 에서의 메모리 배리어는 뒤의 `rcu_try_flip_waitzero_state()` 에서의 카운터 체크가 이 카운터들을 읽어오는 행위 뒤에 이뤄짐을 분명히 험 험 험 험 험 합니다. Line 10 과 11 은 해당 CPU 가 스텝샷이 찍힌 후로 `dynticks-idle` 상태를 유지했다면 0 을 리턴 (해당 CPU 와의 커뮤니케이션이 필요지 않음을 의미) 합니다. 비슷하게, Line 12 와 13 은 해당 CPU 가 초기에 `dynticks-idle` 상태였거나 `dynticks-idle` 상태를 완전히 지나왔다면 0 을 리턴합니다. 이 두가지 경우 모두에 있어서, 해당 CPU 가 grace-period zk운터의 옛 날 값을 가져왔을 수는 없습니다. 이런 조건들 가운데 하나도 해당되지 않는다면, line 14 에서 1 을 리턴해서 해당 CPU 는 명시적인 응답을 필요로 함을 의미합니다.

`rcu_try_flip_waitmb_state()` 상태는 아래에 보인 것과 같은 `rcu_try_flip_waitmb_needed()` 를 호출합니다:

```

1 static inline int
2 rCU_trY_fLIP_wAItnb_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 }

```

이는 `rcu_trY_fLIP_wAItnb_nEEded()` 와 유사한데, 차이점은 line 12 와 13 뿐으로, dynticks-idle 상태로부터 또는 dynticks-idle 상태로의 상태 전환은 모두 `rcu_trY_fLIP_wAItnb_state()` 상태에 의해 필요 한 메모리 배리어를 실행하기 때문입니다.

이제 우리는 RCU 와 dynticks-idle 상태 사이의 인터페이스에 관여된 모든 코드를 봤습니다. 다음 섹션은 이 코드를 검증하기 위해 Promela 모델을 만들어 봅니다.

Quick Quiz 12.10: 이 섹션에서 보인 모든 코드 가운데 버그들을 찾아냈나요? ■

12.1.6 Validating Preemptible RCU and dynticks

이 섹션은 dynticks 와 RCU 사이의 인터페이스를 위한 Promela 모델을 단계별로 개발해 보며, 뒤따르는 섹션들 각각은 하나의 단계씩을 설명하는데, 프로세스 단계부터 시작해서, 단정문, 인터럽트, 그리고 마지막으로 NMI 를 추가해 봅니다.

12.1.6.1 Basic Model

이 섹션은 프로세스 단계 dynticks 진입/종료 코드와 grace-period 처리를 Promela로 변환해 봅니다 [Hol03]. 우리는 2.6.25-rc4 커널에서의 `rcu_exit_nohz()` 와 `rcu_enter_nohz()` 로 시작해서, 이것들을 dynticks-idle 모드를 들어가고 빠져나오는 과정을 모델링하는 Promela 프로세스를 다음의 루프와 같이 만들 겁니다:

```

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 }

```

Line 6 과 20 은 루프를 정의합니다. Line 7 은 `i` 의 값이 `MAX_DYNTICK_LOOP_NOHZ` 를 넘어서면 루프를 빠져나가게 합니다. Line 8 은 루프의 각 패스가 line 9-19 를 수행하도록 구성합니다. Line 7 과 8 의 조건들은 서로에게 배타적이기 때문에, 일반적인 Promela 의 옳은 조건에 대한 무작위 선택은 무효화 됩니다. Line 9 와 11 은 `rcu_exit_nohz()` 의 어토믹하지 않은 `dynticks_progress_counter` 의 값 증가를 모델링 하며, line 12 는 `WARN_ON()` 을 모델링 합니다. 여기서의 `atomic`은 `WARN_ON()` 이 엄밀히 말해선 알고리즘의 한 부분이 아닌 만큼 단순히 Promela 상태 공간의 크기를 줄여주는 역할을 합니다. Line 14-18 은 비슷하게 `rcu_enter_nohz()` 의 값 증가와 `WARN_ON()` 을 모델링 합니다. 마지막으로, line 19 는 루프 카운터의 값을 증가시킵니다.

따라서 루프의 각 패스는 (예를 들어, 태스크를 시작하려고 해서) dynticks-idle 모드를 빠져나가는, 그리고 나서 다시 (예를 들어, 그 태스크가 블락 당해서) dynticks-idle 모드로 들어가는 CPU 를 모델링합니다.

Quick Quiz 12.11: 왜 `rcu_exit_nohz()` 와 `rcu_enter_nohz()` 사이의 메모리 배리어는 Promela 에 모델링 되지 않은거죠? ■

Quick Quiz 12.12: `rcu_exit_nohz()` 에 이어서 `rcu_enter_nohz()` 가 뒤따르는 경우를 모델링 하는 건 좀 이상하지 않나요? 그보다는 진입 후에 빠져나가는 상황을 모델링하는게 더 자연스럽지 않을까요? ■

다음 단계는 RCU 의 grace-period 처리를 위한 인터페이스의 모델링입니다. 이를 위해, 우리는 2.6.25-rc4 커널의 `dyntick_save_progress_counter()`, `rcu_trY_fLIP_wAItnb_nEEded()`, `rcu_trY_fLIP_wAItnb_state()`, 뿐만 아니라 `rcu_trY_fLIP_wAItnb()` 와 `rcu_trY_fLIP_wAItnb()` 를 모델링 해야 합니다. 다음의 `grace_period()` Promela 프로세스는 이 함수들을 그것들이 하나의 `preemption` 가능한 RCU 의 grace-period 처리 과정 사이에서 호출될 것처럼 모델링 합니다.

```

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 }

```

Line 6-9 은 루프 리미트를 (에러가 났을 경우에 한해 .trail 파일에) 프린트하고 현재 CPU 의 dynticks_progress_counter 변수의 스텝샷을 얻어오는, `rcu_try_flip_idle()` 의 코드 한줄과 `dyntick_save_progress_counter()` 호출을 모델링 합니다. 이 두 라인들은 상태 공간의 크기를 줄이기 위해 어토믹하게 수행됩니다.

Line 10-22 는 `rcu_try_flip_waitack()` 그 함수의 `rcu_try_flip_waitack_needed()` 함수 호출을 모델링합니다. 이 루프는 각 CPU로부터 카운터 뒤집기 응답을 기다리는 grace-period 상태 머신을, 하지만 그 중에서도 dynticks-idle CPU들과 상호작용하는 부분만을 모델링 합니다.

Line 23 은 또다시 CPU 의 dynticks_progress_counter 변수의 스텝샷을 얻어오는 `rcu_try_flip_waitzero()` 의 코드 중 한줄과 그것의 `dyntick_save_progress_counter()` 함수 호출을 모델링 합니다.

마지막으로, line 24-36 은 `rcu_try_flip_waitack()` 과 그 함수의 `rcu_try_flip_waitack_needed()` 호출을 모델링합니다. 이 루프는 각각의 CPU 가 메모리 배리어를 실행하기를 기다리는, 하지만 여기서도 역시 dynticks-idle CPU들과 상호작용하는 부분만을 모델링 합니다.

Quick Quiz 12.13: 잠깐만요! 이 리눅스 커널에서, dynticks_progress_counter 와 `rcu_dyntick_snapshot` 은 per-CPU 변수들입니다. 그런데 왜 그것들이 per-CPU 변수들이 아니라 하나의 글로벌 변수로 모델링 된거죠? ■

최종적인 모델 (`dyntickRCU-base.spin`) 은 `runspin.sh` 스크립트를 통해 돌아갈 때, 691 개의 상태를 생성하고 에러 없이 수행되는데, 이것은 실패를

찾을 수 있는 단정문이 존재하지 않는다는 점을 생각해 보면 놀라울 일이 아닙니다. 따라서 다음 섹션에서는 안전성을 위한 단정문들을 추가해 봅니다.

12.1.6.2 Validating Safety

안전한 RCU 구현에서 grace period 는 그 grace period 의 시작 전에 시작된 모든 RCU 읽기 쓰레드들이 완료되기 전에는 절대 완료될 수 없어야 합니다. 이는 다음과 같은 세개의 상태를 취할 수 있는 `gp_state` 변수를 사용해 모델링 될 수 있습니다:

```

1 #define GP_IDLE      0
2 #define GP_WAITING  1
3 #define GP_DONE     2
4 byte gp_state = GP_DONE;

```

`grace_period()` 프로세스는 이 변수를 아래에 보여진 것과 같이 grace-period 단계를 통해 진행되는 것처럼 설정합니다:

```

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 }

```

Line 6, 10, 25, 26, 29, 그리고 44는 기본적인 RCU 안전성을 검증하기 위해 dyntick_nohz() 프로세스를 허용할 수 있도록 이 변수를(적당한 알고리즘적 오퍼레이션들을 어토믹하게 섞어서) 업데이트 합니다. 이 검증의 형태는 RCU 읽기 쓰레드들이 그럴듯하게 존재하는 시간동안 gp_state 변수의 값이 GP_IDLE에서 GP_DONE으로 바뀔 수 없음을 단정하려는 것입니다.

Quick Quiz 12.14: Line 25 와 26 에 gp_state의 연속된 두개의 변경이 있는데, 어떻게 line 25에서의 변경이 사라지지 않을거라고 확신할 수 있을까요? ■

dyntick_nohz() Promela 프로세스는 이 검증을 아래와 같이 구현합니다:

```

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 }

```

Line 13은 gp_state 변수의 값이 태스크 수행 시작 시점에서 GP_IDLE이라면 old_gp_idle 플래그를 설정하고, gp_state 변수가 태스크 수행 중간에 GP_DONE으로 바뀌었다면 하나의 RCU read-side 크리티컬 섹션이 전체 시간동안 존재할 수 있다는 점에서 비합법적이므로 line 18과 19에서의 단정문이 실패합니다.

결과적으로 만들어지는 모델(dyntickRCU-base-.spin)은 runspin.sh 스크립트를 통해 수행될 때, 964개의 상태들을 생성하고 안심되게도 에러없이 돌아갑니다. 그렇다면, 안전성이 상당히 중요하긴 하지만, 불명확하게 지연되는 grace period를 막는 것 역시 굉장히 중요합니다. 따라서 다음 섹션에서는 liveness의 검증을 다룹니다.

12.1.6.3 Validating Liveness

Liveness는 증명되기에 어려울 수 있지만, 여기 적용할 수 있는 간단한 트릭이 있습니다. 첫번째 단계는 다음의 line 27에서 보이는 것과 같이 dyntick_nohz() 가

dyntick_nohz_done을 통해 할일을 마쳤음을 알리는 것입니다:

```

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 }

```

이 변수를 두면, 불필요한 블록들을 체크하기 위해 다음과 같이 grace_period()에 단정문들을 추가할 수 있습니다:

```

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 }

```

```

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 }

```

우리는 line 10에서 0으로 초기화되는 `shouldexit` 변수를 line 5에서 추가했습니다. Line 17은 `shouldexit` 가 아직 0일 것을 단정하며, line 18에서는 `shouldexit` 를 `dyntick_nohz()`에 의해 관리되는 `dyntick_nohz_done`의 값으로 설정합니다. 따라서 이 단정은 `dyntick_nohz()`가 수행 완료된 후의 카운터가 뒤집히는 응답을 기다리는 과정에서 하나 이상의 패스를 시도하게 되면 실패할 것입니다. 무엇보다, `dyntick_nohz()`가 완료되었다면, 더이상 우리를 해당 루프의 바깥으로 나가게 할 상태 변화는 더이상 없고, 따라서 이 상태를 두번 이상 수행하게 됨은 곧 무한 루프를 의미해서, grace period 가 끝나지 않을 것임을 의미하게 됩니다.

Line 32, 39, 그리고 40은 두번째 (메모리 배리어) 루프를 위해 비슷한 방식으로 수행됩니다.

하지만, 이 모델 (`dyntickRCU-base-sl-busted.spin`) 을 수행하는 것은 실패로 끝나게 되는데, line 23이 잘못된 변수는 짹수라고 체크를 하기 때문입니다. 실패하게 되면, `spin`은 “trail” 파일 (`dyntickRCU-base-sl-busted.spin.trail`) 을 쓰게 되는데, 이 파일은 실패로 이르게 된 상태들의 시퀀스를 기록합니다. `spin`이 이 상태들의 시퀀스를 재추적하게 하고 실행된 statement들과 변수들의 값 (`dyntickRCU-base-sl-busted.spin.trail.txt`) 을 프린트 하려면 `spin -t -p -g -l dyntickRCU-base-sl-busted.spin` 커맨드를 사용하세요. `Spin`은 두 함수를 모두 하나의 파일에서 취할 것이기 때문에 위에 리스트된 코드와 라인 넘버가 맞지 않을 수 있음을 알아 두세요. 하지만, 라인 넘버들은 전체 모델 (`dyntickRCU-base-sl-busted.spin`) 과는 맞아떨어질 겁니다.

`dyntick_nohz()` 프로세스가 step 34에서 (“34:” 를 검색해 보세요) 완료되었지만, `grace_period()` 프로세스는 루프를 빠져나가는데 실패했음을 볼 수 있습니다. `curr`의 값은 6이고 (step 35를 보세요) `snap`의 값은 5 (step 17을 보세요)입니다. 따라서 line 21에서의 첫번째 조건은 `curr != snap`이므로 성립되지 않고, line 23에서의 두번째 조건은 `snap`이 허수이고

`curr`는 `snap` 보다 1만큼 클 뿐이기 때문에 성립되지 않습니다.

따라서 이 두개의 조건들 중 하나는 올바르지 않습니다. `rcu_try_flip_waitack_needed()`의 첫번째 조건에 대한 코멘트를 따르면:

CPU 가 전체 시간동안 `dynticks` 모드에 빠져 있고 인터럽트, NMI, SNMI, 또는 뭐든 받지 않았다면, 해당 CPU 는 `rcu_read_lock()`의 중간에 있을 수 없고, 따라서 그것이 수행하게 되는 다음의 `rcu_read_lock()` 는 카운터의 새로운 값을 사용해야만 한다. 따라서 우리는 안전하게 이 CPU 가 이미 그 카운터에 응답을 한 것처럼 행동을 한다.

첫번째 조건은 실제로 이와 들어맞는데, `curr == snap`이고 `curr`가 짹수라면, 연관된 CPU 는 요구된 대로 `dynticks-idle` 모드에 전체 시간동안 존재한 것이기 때문입니다. 따라서 두번째 조건에 대한 코멘트를 보도록 합시다:

CPU 가 `dynticks` idle 단계를 활성화된 irq 헨들러 없이 거쳐왔거나 진입했다면, 앞에서와 같이, 우린 이 CPU 가 카운터에 응답을 이미 한 것처럼 안전하게 행동할 수 있다.

조건의 첫번째 부분은 올바른데, `curr` 와 `snap` 이 2만큼 차이가 나면 그 사이에 최소 하나의 짹수가 존재할 수 있어서, 하나의 `dynticks-idle` 단계를 완전히 지나왔다고 볼 수 있기 때문입니다. 하지만, 조건의 두번째 부분은 `dynticks-idle` 모드가 시작되었지만, 이 모드가 아직 끝나지 않았음을 의미합니다. 따라서 우리는 짹수가 되어야 할 값으로 `snap`이 아니라 `curr`를 테스트해야합니다.

고쳐진 C 코드는 다음과 같습니다:

```

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 }

```

Line 10-13은 이제 다음과 같이 간단하게 결합될 수 있습니다. 비슷한 단순화가 `rcu_try_flip_waitmb_needed()`에 적용될 수 있습니다.

```

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 }

```

해당 모델에 연관된 수정을 가하면 (dyntickRCU-base-sl.spin) 에러 없이 통과되는 661 개의 상태들을 통한 올바른 검증을 얻을 수 있습니다. 하지만, 첫번째 버전의 liveness 검증은 liveness 검증 자체의 버그로 인해서, 이 버그를 잡는데 실패했음은 알려둘 가치가 있습니다. 이 liveness 증명 버그는 grace_period() 프로세스 안에 무한루프를 넣어둠으로써 발생했습니다!

우린 이제 안전성과 liveness 조건을 모두 성공적으로 검증했습니다만, 동작하고 블락킹 되는 프로세스들에 대해서만 그렇습니다. 인터럽트들 역시 처리해야 할텐데, 이는 다음 섹션에서 처리하도록 하겠습니다.

12.1.6.4 Interrupts

Promela에서 인터럽트를 모델링하는 두가지 방법이 있습니다:

1. C 전처리기 트릭을 사용해서 모든 dynticks_nohz() 프로세스의 statement 사이에 인터럽트 핸들러를 넣거나
2. 인터럽트 핸들러를 별도의 프로세스로 모델링하는 겁니다.

두번째 방법이 더 작은 상태 공간을 갖게 해줄거라 생각할 수 있습니다만, 이는 인터럽트 핸들러가 어떻게든 dynticks_nohz() 와는 어토믹하게 동작해야 하지만 grace_period() 와는 어토믹하지 않게 동작해야 할 것을 필요로 합니다.

다행히도, Promela는 어토믹 statement들을 분기를 가를 수 있게 해줍니다. 이 트릭은 우리가 인터럽트 핸들러에서 플래그를 설정하고, dynticks_nohz() 가 어토믹하게 이 플래그를 체크한 후 그 플래그가 설정되지 않았을 때에만 실행되도록 코드를 수정할 수 있게 합니다. 이는 다음과 같이 라벨과 Promela statement들을 사용하는 C 전처리기 매크로를 사용해 이뤄질 수 있습니다:

```

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     } \

```

이 매크로를 다음과 같이 사용할 수 있을 겁니다:

```

EXECUTE_MAINLINE(stmt1,
                  tmp = dynticks_progress_counter)

```

매그로의 line 2는 특정 statement 라벨을 만듭니다. Line 3-8은 in_dyntick_irq 변수를 테스트 하는 어토믹 블락으로, 이 변수가 값이 설정되어 있다면 (인터럽트 핸들러가 활성 상태임을 의미), 어토믹 블락의 수행을 라벨로 되돌립니다. 그렇지 않다면, line 6에서 요청된 statement를 수행합니다. 전체 효과는 요청된대로 인터럽트가 활성화 될 때마다 메인 수행이 공회전을하게 되는 것입니다.

12.1.6.5 Validating Interrupt Handlers

첫번째 단계는 dyntick_nohz() 를 다음과 같이 EXECUTE_MAINLINE() 의 형태로 바꾸는 겁니다:

```

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 }

```

여러 statement들의 그룹이 EXECUTE_MAINLINE() 으로 전달될 때에는 line 12-15와 같이 해당 그룹 내의 모든 statement들이 어토믹하게 수행됨을 알아둘 필요가 있습니다.

Quick Quiz 12.15: 하지만 EXECUTE_MAINLINE() 내의 statement들이 어토믹 하지 않게 수행되어야 하는 경우라면 어떻게 하겠습니까? ■

Quick Quiz 12.16: 하지만 dynticks_nohz() 프로세스가 “if”나 “do” 문을 가지고 있는데 그 안의 조건절들이 그 분기 본체들은 어토믹하지 않게 수행되는 형태라면 어떻게 되죠? ■

다음 단계는 인터럽트 핸들러를 모델링 하기 위해 dyntick_irq() 프로세스를 작성하는 것입니다:

```

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 }

```

Line 7-48의 루프는 MAX_DYNTICK_LOOP_IRQ 까지의 인터럽트들을 모델링하는데, line 8과 9는 루프 조건을 구성하고 line 46은 제어 변수를 증가시킵니다. Line 10은 dyntick_nohz()에게 인터럽트 핸들러가 돌아가는 중이라고 이야기하고, line 45는 dyntick_nohz()에게 이 핸들러가 완료되었다고 이야기합니다. Line 49는 dyntick_nohz()의 관련된 라인과 같이 liveness 검증을 위해 사용됩니다.

Quick Quiz 12.17: Line 45와 46 (in_dyntick_irq = 0; 와 i++;)은 왜 어토믹하게 수행되는 건가요? ■

Line 11-25는 rcu_irq_enter()를 모델링하고, line 26과 27 dms __irq_enter()의 관련된 부분을

모델링 합니다. Line 28과 29는 dynticks_nohz()의 관련된 부분과 거의 같은 방법으로 안전성을 검증합니다. Line 30과 31은 __irq_exit()의 관련 부분을 모델링하고, line 32-43에서 rcu_irq_exit()을 모델링 합니다.

Quick Quiz 12.18: ⓠ dynticks_irq() 프로세스가 모델링할 수 없는 인터럽트들의 특성은 무엇이 있을까요? ■

이제 grace_period() 프로세스는 다음과 같이 됩니다:

```

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 &&
20                            dyntick_irq_done;
21            curr = dynticks_progress_counter;
22            if
23            :: (curr - snap) >= 2 || (curr & 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                            dyntick_irq_done;
42            curr = dynticks_progress_counter;
43            if
44            :: (curr != snap) || ((curr & 1) == 0) ->
45                break;
46            :: else -> skip;
47            fi;
48        }
49    od;
50    gp_state = GP_DONE;
51 }

```

grace_period()의 구현은 앞의 것과 매우 유사합니다. 변경점은 새로운 인터럽트 카운트 인자를 추가하기 위한 line 10의 추가와 liveness 체크를 위한 새로운 dyntick_irq_done 변수를 추가하는 line 19와 39의 변경, 그리고 line 22와 42에서의 최적화 뿐입니다.

이 모델 (dyntickRCU-irqnn-ssl.spin)은 대략 50만개의 상태들을 에러 없이 성공적으로 검증합니다. 하지만, 이 버전의 모델링은 중첩된 인터럽트들을 처리하지 않습니다. 이 주제는 다음 섹션에서 다루겠습니다.

12.1.6.6 Validating Nested Interrupt Handlers

중첩된 인터럽트 핸들러들은 dyntick_irq()의 루프의 몸체를 다음과 같이 분리시키는 것으로 모델링 될 수 있을 겁니다:

```

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 }

```

이는 dynticks_irq() 프로세스와 유사합니다. 여기선 line 5에서 두번째 카운터 변수 j를 추가해서 i는 인터럽트 핸들러들로의 진입을 셈하고 j는 인터럽트 핸들러로부터 빠져나가는 횟수를 세게 합니다. Line 7에서의 outermost 변수는 안전성 검사를 위해 언제 gp_state 변수가 샘플링 되어야 하는지를 결정합니다. Line 10과 11에서의 루프 종료 체크는 명시된 수의 인터럽트 핸들러들이 진입된 만큼 빠져나와지기도 했음을 요청하기 위해 업데이트 되었고, i의 값 증가는 인터럽트 진입 모델의 마지막인 line 41로 옮겨졌습니다. Line 13-16은 이게 중첩된 인터럽트들의 집합의 가장 바깥인지 여부를 알리기 위해 outermost 변수를 설정하고 dyntick_nohz() 프로세스에 의해 사용되는 in_dyntick_irq 변수를 설정합니다. Line 34-40에서는 gp_state 변수의 상태를 읽어오는데, 가장 바깥의 인터럽트 핸들러 내일 때에만 그렇습니다.

Line 42는 인터럽트 종료 모델링을 위한 do 루프 조건절을 갖습니다: 우리가 진입한 것에 비해 적은 수의 인터럽트들만이 종료된 동안은, 다른 인터럽트를 빠져나가는게 합법적입니다. Line 43-50은 안전성 기준을 체크합니다만, 가장 바깥의 인터럽트 단계에서 빠져나갈 때에만 그렇습니다. 마지막으로, line 65-68은 인터럽트 종료 카운트 j의 값을 증가시키고, 이게 가장 바깥의 인터럽트 단계라면, in_dyntick_irq의 값을 지웁니다.

이 모델 (dyntickRCU-irqnn-ssl.spin)은 약 50만 개를 조금 넘는 수의 상태들에 대한 정확성 검사를 만들어내며, 모두 에러 없이 통과합니다. 하지만, 이 버전의 모델은 NMI를 처리하지 않는데, 이에 대해선 다음 섹션에서 다루겠습니다.

12.1.6.7 Validating NMI Handlers

우리는 NMI는 중첩되지 않는다는 점을 생각하면서 NMI를 위해 인터럽트에 했던 것과 동일한 일반적 방법을 사용합니다. 이는 다음과 같은 dyntick_nmi() 프로세스를 만들어 내게 됩니다:

```

1 proctype dyntick_nmi()
2 {
3     byte tmp;
4     byte i = 0;
5     bit old_gp_idle;

```

```

6    do
7      :: i >= MAX_DYNTICK_LOOP_NMI -> break;
8      :: i < MAX_DYNTICK_LOOP_NMI ->
9        in_dyntick_nmi = 1;
10       if
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 }

```

물론, 우리가 NMI 를 갖는다는 사실은 다른 컴포넌트들에도 조정을 필요로 합니다. 예를 들어, EXECUTE_MAINLINE() 매크로는 이제 다음과 같이 dyntick_nmi_done 변수의 체크를 통해 NMI 핸들러 (in_dyntick_nmi) 는 물론 인터럽트 핸들러 (in_dyntick_irq) 에도 주의를 기울여야 합니다:

```

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   } \

```

우리는 또한, dyntick_irq() 가 dyntick_nmi() 를 배척하도록 하기 위해 in_dyntick_nmi 를 체크하는 EXECUTE_IRQ() 매크로를 만들어야 합니다:

```

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   } \

```

더 나아가서 dyntick_irq() 를 다음과 같이 EXECUTE_IRQ() 로 변환시켜야 합니다:

```

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 }

```

우리가 “if” 문을 open-code 했음을 알아 두시기 바랍니다 (예를 들면, line 17-29). 또한, (line 58 과 같은) 엄격한 로컬 상태를 처리하는 문장들은 dyntick_nmi()를 배제시킬 필요가 없음을 알아두시기 바랍니다.

마지막으로, grace_period()는 몇가지 변경만 있으면 됩니다:

```

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);

```

```

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

```

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 }

```

우리는 line 11에 MAX_DYNTICK_LOOP_NMI 패러미터를 위한 새로운 printf()를 추가했고 dyntick_nmi_done 을 line 22 와 44 의 shouldexit 할당에 추가했습니다.

이 모델(dyntickRCU-irq-nmi-ssl.spin)은 수개의 상태들과 함께 동작하는 정확한 검증을 만들어내며, 에러 없이 통과합니다.

Quick Quiz 12.19: Paul은 항상 그의 코드를 이렇게 고통스럽도록 점진적인 방법으로 작성하나요? ■

12.1.6.8 Lessons (Re)Learned

여기서는 몇가지 (다시) 배운 교훈들을 제공합니다:

```

-     if ((curr - snap) > 2 || (snap & 0x1) == 0)
+     if ((curr - snap) > 2 || (curr & 0x1) == 0)

```

Figure 12.18: Variable-Name-Typo Fix Patch

1. **Promela 와 spin 은 인터럽트/NMI-핸들러 와의 상호작용들을 검증할 수 있습니다.**
2. **코드를 문서화 하는 것은 버그를 찾는데 도움을 줄 수 있습니다.** 이 경우, 문서화는 Figure 12.17 의 패치에 보여진 것처럼 `rcu_enter_nohz()` 와 `rcu_exit_nohz()` 에 있던 잘못 배치된 메모리 배리어들을 찾아냈습니다.
3. **코드를 빨리, 자주, 그리고 없어지기 전까지 검증하십시오.** 이런 노력은 Figure 12.18 의 패치에서 보여진 것과 같이, `rcu_try_flip_waitack_needed()` 안의 테스트하거나 디버깅 하기에 상당히 어려울 수 있는 `qjrmfmf ckwdksoTtmqslek`.
4. **항상 검증 코드를 검증하십시오.** 이를 위한 일반적인 방법은 고의적인 버그를 집어넣고 검증 코드가 이를 잡아내는지 확인하는 것입니다. 물론, 그 검증 코드가 그 버그를 잡아내는데에 실패한다면, 그 버그 자체에 대해서도 검증을 할 필요가 있을 것이고, 그렇게 무한한 반복을 해야 할 수도 있습니다. 하지만, 당신 자신이 그런 무한한 반복에 빠져 있는 걸 깨닫게 된다면, 숙면을 취하는 것이 효과적인 디버깅 테크닉이 될 수 있습니다. 그렇게 되면 분명한 검증을 검증하는 테크닉은 고의적으로 검증되는 코드에 버그를 집어넣는 것이라 것을 알게 될겁니다. 만약 검증 코드가 그것들을 찾는데 실패한다면, 그 검증은 분명히 버그로 가득차 있는 것입니다.
5. **어토믹 인스트럭션들의 사용이 검증을 단순화 시킬 수 있습니다.** 앤타깝게도, `cmpxchg` 어토믹 명령어의 사용은 중요한 irq 빠른 수행 경로를 느리게 수행되도록 만들어서 이 경우에 적절치 못할 수도 있습니다.
6. **복잡한 정식 검증의 필요는 당신의 설계를 다시 한번 생각해 봄이 필요성을 의미합니다.**

이 마지막 핵심에 이르러서, 다음 섹션에서 보여지게 될, dynticks 문제에 대한 훨씬 간단한 해결책이 있음이 드러났습니다.

12.1.6.9 Simplicity Avoids Formal Verification

Preemption 가능한 RCU 의 dynticks 인터페이스의 복잡도는 주로 irq 도 NMI 도 같은 코드 수행 경로와 같은

```

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

상태 변수들을 사용한다는 사실에서 기인합니다. 이는 Manfred Spraul [Spr08]에 의해 간접적으로 제안된대로 계층적 RCU [McK08a]에 가해진 것처럼 별도의 코드 수행 경로와 변수들을 제공하는 방향을 사용할 것으로 귀결됩니다.

12.1.6.10 State Variables for Simplified Dynticks Interface

Figure 12.19 는 새로운 per-CPU 상태 변수들을 보입니다. 이 변수들은 복수의 개별적인 RCU 구현들 (ex: `rcu` 와 `rcu_bh`) 이 편하고 효과적으로 dynticks 상태를 공유할 수 있도록 하기 위해 구조체로 그룹지어졌습니다. 이어서 언급할 것들은 개별적인 per-CPU 변수들로 생각될 수도 있습니다.

`dynticks_nesting`, `dynticks`, 그리고 `dynticks_snap` 변수들은 irq 코드 수행 경로들을 위한 것이고, `dynticks_nmi` 와 `dynticks_nmi_snap` 변수들은 NMI 코드 수행 경로들을 위한 것입니다만, NMI 코드 수행 경로는 `dynticks_nesting` 변수 역시 레퍼런스할 (하지만 수정하지는 않을) 겁니다. 이 변수들은 다음과 같이 사용됩니다:

- **dynticks_nesting:** 이 변수는 연관된 CPU 가 RCU read-side 크리티컬 섹션들을 위해 모니터링되어야 하는 이유들의 수를 카운트 합니다. 만약 CPU 가 dynticks-idle 모드에 있다면, 이 변수는 irq 중첩 단계를 카운트하게 되고, 그렇지 않다면 irq 중첩 단계보다 1 큰 값을 갖게 됩니다.
- **dynticks:** 이 카운터의 값은 연관된 CPU 가 dynticks-idle 모드에 있고 현재 해당 CPU 에서 수행중인 irq 핸들러가 없다면 짹수를, 그렇지 않다면 훌수의 값을 갖게 됩니다. 달리 말하자면, 이 카운터의 값이 훌수라면, 연관된 CPU 는 RCU read-side 크리티컬 섹션 안에 있을 수 있습니다.
- **dynticks_nmi:** 이 카운터의 값은 연관된 CPU 가 NMI 핸들러 안에 있다면 훌수가 되는데, 해당

```

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

NMI 가 이 CPU 가 현재 수행중인 irq 핸들러 없이 dynticks-idle 모드에 빠져 있을 때에만 그렇습니다. 그렇지 않다면, 해당 카운터의 값은 짹수가 됩니다.

- dynticks_snap: 이것은 dynticks 카운터의 스냅샷이 됩니다만, 현재 RCU grace period 가 너무 긴 기간 동안 연장되었을 때에만 그렇습니다.
- dynticks_nmi_snap: 이것은 dynticks_nmi 카운터의 스냅샷이 됩니다만, 역시 현재 RCU grace period 가 너무 오랜 기간 동안 연장되었을 때에만 그렇습니다.

주어진 기간 동안 dynticks 와 dynticks_nmi 가 모두 짹수를 가지고 있었다면, 연관된 CPU 는 해당 기간 동안 quiescent 상태를 지나온 것입니다.

Quick Quiz 12.20: 하지만, 만약 NMI 핸들러가 irq 핸들러가 완료되기 전에 동작하기 시작하면, 그리고 그 NMI 핸들러가 두번째 irq 핸들러가 시작될 때까지 돌아가면 어떤 일이 발생할까요? ■

12.1.6.11 Entering and Leaving Dynticks-Idle Mode

Figure 12.20 는 “nohz” 모드라고도 알려진 dynticks-idle 모드를 들어가고 빠져나오는 rcu_enter_nohz() 와 rcu_exit_nohz() 를 보이고 있습니다. 이 두 함수들은 프로세스 컨텍스트에서 호출됩니다.

```

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

Line 6 는 모든 앞의 (RCU read-side 크리티컬 섹션들로부터의 액세스들을 포함할 수 있는) 메모리 액세스들이 dynticks-idle 모드로의 진입을 표시하는 CPU 들보다 앞서서 다른 CPU 들에 보여질 것을 보장합니다. Line 7 과 12 는 irq 를 각각 비활성화시키고 재활성화 시킵니다. Line 8 은 현재 CPU 의 rcu_dynticks 구조체로의 포인터를 얻어오고, line 9 는 현재 CPU 의 dynticks 카운터의 값을 증가시키는데, 우린 프로세스 컨텍스트에서 dynticks-idle 모드에 진입하고 있으므로 짹수가 되어야 할 겁니다. 마지막으로, line 10 은 이제 0이 되어야 하는 dynticks_nesting 의 값을 감소시킵니다.

rcu_exit_nohz() 함수는 이와 상당히 유사합니다만, dynticks_nesting 을 감소시키는게 아니라 증가시키며 dynticks 의 반대 특성을 체크합니다.

12.1.6.12 NMIs From Dynticks-Idle Mode

Figure 12.21 는 dynticks-idle 모드로부터 NMI 에 들어가는 것과 빠져나오는 것을 RCU 에 알리는 rcu_nmi_enter() 와 rcu_nmi_exit() 함수를 보입니다. 하지만, 해당 NMI 가 irq 핸들러 수행 중에 들어오게 되면, RCU 는 이 CPU 의 RCU read-side 크리티컬 섹션들의 탐색 중일 것이므로, rcu_nmi_enter() 의 line 6 과 7 과 rcu_nmi_exit() 의 line 18 과 19 는 dynticks 가 훌수라면 아무것도 하지않고 리턴합니다. 그렇지 않다면, 두 함수는 dynticks_nmi 의 값을 증가시켜서 rcu_nmi_enter() 는 그 값을 훌수로, rcu_nmi_exit() 는 그 값을 짹수로 만들어둡니다. 두 함수 모두 line 10 과 20 에서 값 증가와 RCU read-side 크리티컬 섹션이 있을 수 있는 곳 사이에 메모리 배리어를 실행합니다.

```

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

12.1.6.13 Interrupts From Dynticks-Idle Mode

Figure 12.22 는 irq 컨텍스트로의 진입과 irq 컨텍스트로부터의 빠져나옴을 RCU 에 알리는 `rcu_irk_enter()` 와 `rcu_irk_exit()` 함수들을 보입니다. `rcu_irk_enter()` 의 Line 6 는 `dynticks_nesting` 의 값을 증가시키고, 이 값이 이미 0이 아니었다면, line 7 에서 조용히 리턴합니다. 그렇지 않다면, line 8 에서 `dynticks` 의 값을 증가시켜서 그 값을 훌수로 만들어서 이 CPU 가 이제 RCU read-side 크리티컬 섹션을 수행할 수 있다는 사실과 일관적이게 만듭니다. 따라서 line 10 은 `dynticks` 의 값 증가가 뒤따르는 irq 핸들러가 실행할 수도 있는 RCU read-side 크리티컬 섹션들보다 먼저 보여지도록 보장하기 위해 메모리 배리어를 실행합니다.

`rcu_irk_exit()` 의 line 18 에서는 `dynticks_nesting` 의 값을 감소시키고, 그 결과가 0이 아니라면, line 19 에서 조용히 리턴합니다. 그렇지 않다면, line 21 에서의 `dynticks` 의 값 증가가 앞의 irq 핸들러가 실행했을 수도 있는 모든 RCU read-side 크리티컬 섹션들 뒤에 보여질 것을 보장하기 위해 메모리 배리어를 수행합니다. Line 22 는 `dynticks` 가 짹수임을 검증해서 어떤 RCU read-side 크리티컬 섹션들도 dynticks-idle 모드에서는 나타날 수 없다는 사실과 일관적이게 합니다. Line 23-25 는 앞의 irq 핸들러들이 RCU 콜백들을 집어넣었는지 확인하고, 만약 그렇다면 재스케줄 API 를 통해 이 CPU 가 dynticks-idle 모드를 빠져나오도록 합니다.

```

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_fqs++;
17    return ret;
18 }

```

Figure 12.23: Saving Dyntick Progress Counters

```

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 || (curr_nmi & 0x1) == 0)) {
16        rdp->dynticks_fqs++;
17        return 1;
18    }
19    return rCU_implicit_offline_qs(rdp);
20 }

```

Figure 12.24: Checking Dyntick Progress Counters

12.1.6.14 Checking For Dynticks Quiescent States

Figure 12.23 는 명시된 CPU 의 `dynticks` 와 `dynticks_nmi` 카운터들의 스냅샷을 가져오는 `dyntick_save_progress_counter()` 를 보입니다. Line 8 과 9 는 이 두개의 변수들을 로컬로 스냅샷을 떠오고, line 10 은 Figure 12.20, 12.21, 그리고 12.22 의 함수들의 메모리 배리어들과 짹을 맞추기 위해 메모리 배리어를 실행합니다. Line 11 과 12 는 나중의 `rcu_implicit_dynticks_qs` 호출을 위한 스냅샷들을 기록해 두고, line 13 과 14 는 dynticks-idle 모드의 이 CPU 가 irq 도 NMI 도 진행중이 아니어서 (달리 말하자면, 두 스냅샷 모두 짹수를 가지고 있어서) 연장된 quiescent state 안에 있는 것인지 체크를 합니다. 만약 그렇다면, line 15 와 16 은 이 이벤트를 세고, 이 CPU 가 quiescent state line 17 에서 true 를 리턴합니다.

Figure 12.24 는 CPU 가 `dynticks_save_progress_counter()` 호출에 이어서 dyntick-idle

모드에 들어갔는지 여부를 체크하는 `rcu_implicit_dynticks_qs()` 함수를 보입니다. Line 9 와 11은 연관된 CPU 의 `dynticks` 와 `dynticks_nmi` 변수들의 새로운 스냅샷을 가져오고, line 10 과 12는 기존에 `dynticks_save_progress_counter()`에 __이해 저장된 스냅샷을 가져옵니다. Line 13은 이어서 Figure 12.20, 12.21, 그리고 12.22의 함수들의 메모리 배리어들과 짝을 맞추기 위해 메모리 배리어를 실행합니다. Line 14-16은 이어서 해당 CPU 가 지금 (`curr` 와 `curr_nmi` 가 짝수를 가져서) quiescent state에 있거나 (`dynticks` 와 `dynticks_nmi` 의 값들이 이미 변해서) 마지막의 `dynticks_save_progress_counter()` 호출 후에 quiescent state를 지나왔는지를 체크합니다. 만약 이 체크가 이 CPU 는 dyntick-idle quiescent state를 지나왔다고 확인한다면, line 17은 그 사실의 횟수를 세고 line 18에서 이에 대한 사실을 알리는 값을 리턴합니다. 어떻게 되었든, line 20에서는 오프라인인 CPU 를 기다리고 있는 RCU 를 초래할 수 있는 경주 상황을 체크합니다.

Quick Quiz 12.21: 이건 여전히 상당히 복잡하네요. 왜 그냥 dyntick-idle 모드에 있는 각 CPU 들에 대해 bit의 값이 설정되어 있는 `cpumask_t` 를 갖고 irq 나 NMI 핸들러에 들어갈 때에는 그 비트의 값을 지우고, 빠져나올 때에는 그 값을 설정하는 식으로 하지 않는거죠?

■

12.1.6.15 Discussion

양간의 시점 변경이 RCU 를 위한 dynticks 이널페이스의 상당한 단순화를 가져왔습니다. 이 단순화를 이끈 핵심 변경은 irq 와 NMI 컨텍스트 사이에 공유되는 것들의 최소화였습니다. 이 단순화된 인터페이스에서의 유일한 공유는 NMI 컨텍스트에서 irq 변수들로의 레퍼런스들 (dynticks 변수) 뿐입니다. NMI 함수들은 이 변수를 절대 업데이트 하지 않아서 그 값은 NMI 핸들러의 수명 동안 상수를 유지하므로 이런 종류의 공유는 친절한 편입니다. 이런 공유의 한계는 Section 12.1.5에 그려진, NMI 가 irq 함수들의 수행 도중의 어느 시점에든 공유된 상태를 바꿀 수 있는 상황과는 반대로, 다행히도 개별적인 함수들이 한번에 하나씩 이해되는걸 가능하게 합니다.

검증은 좋은 일이 될 수 있지만, 단순화는 그보다도 좋습니다.

12.2 Special-Purpose State-Space Search

Promela 와 spin 이 어떤 (작은) 알고리즘이든 굉장히 많은 부분을 검증할 수 있게 해주긴 하지만, 그것들의

커다란 범용성은 어떨때에는 문제가 될수도 있습니다. 예를 들어, Promela 는 메모리 모델들이나 재배치 의미들을 이해하지 못합니다. 따라서 이 섹션은 제품화된 시스템들에서 사용되는 메모리 모델을 이해해서 완화된 순서 규칙의 코드의 검증을 단순화 시켜주는 상태 공간 탐색 도구들을 설명합니다.

예를 들어, Section 12.1.4 는 완화된 메모리 순서 규칙을 처리하기 위해 Promela 를 어떻게 납득시켜야 하는지를 보았습니다. 이런 방법도 잘 동작할수 있지만, 이는 개발자가 해당 시스템의 메모리 모델을 이해하고 있을 것을 필요로 합니다. 안타깝게도, (존재한다 쳐도) 일부 개발자들만이 현대 CPU 들의 복잡한 메모리 모델을 완전하게 이해하고 있습니다.

따라서, 또다른 방법은 Cambridge 대학의 Peter Sewell 과 Susmit Sarkar, INRIA 의 Luc Maranget, Francesco Zappa Nardelli, 그리고 Pankaj Pawan, 그리고 Oxford 대학의 Jade Alglave 가 IBM 의 Derek Williams 와 협력해서 만든 PPCMEM [AMP+11] 과 같은, 이미 이 메모리 순서 규칙을 이해하고 있는 도구들을 활용하는 것입니다. 이 그룹은 Power, ARM, x86 뿐만 아니라 C/C++11 표준 [Bec11] 의 메모리 모델을 형식화 시키고 Power 와 ARM 형식에 기초하여 PPCMEM 도구를 만들었습니다.

Quick Quiz 12.22: 하지만 x86 은 강한 메모리 순서 규칙을 가지고 있어요! 왜 이 메모리 모델을 형식화 시켜야 하는거죠? ■

PPCMEM 도구는 입력으로 리트머스 테스트를 받습니다. 예제 리트머스 테스트가 Section 12.2.1 에 있습니다. Section 12.2.2 은 이 리트머스 테스트를 동일한 C-언어 프로그램으로 관계지어 보고, Section 12.2.3 는 PPCMEM 을 이 리트머스 테스트에 어떻게 적용하는지를 설명하며, Section 12.2.4 이로부터의 의미를 이야기해 봅니다.

12.2.1 Anatomy of a Litmus Test

PPCMEM 을 위한 예제 PowerPC 리트머스 테스트가 Figure 12.25 에 보여져 있습니다. ARM 인터페이스도 정확히 같은 방식으로 동작합니다만, 그러기 위해선 첫 줄의 “PPC” 가 “ARM” 으로 바뀌어야 하고 Power 인스트럭션(instruction) 이 ARM 인스트럭션으로 대체되어야 합니다. 뒤에서 이야기될 웹페이지의 “Change to ARM model” 을 클릭하는 것으로 ARM 인터페이스를 선택할 수 있습니다.

예제에서, line 1 은 시스템의 종류 (“ARM” 또는 “PPC”) 를 밝히고 모델의 이름을 포함합니다. Line 2 는 테스트의 대안적 이름을 위한 공간을 제공하는데, 일반적으로는 앞의 예제에서 보인 것처럼 빈줄로 남겨두게 될 겁니다. 주석은 line 2 와 3 사이에 Ocaml (또는 Pas-

```

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

cal) 문법의 $(\ast \ast)$ 를 이용해 들어갈 수 있습니다.

Line 3-6 는 모든 레지스터들의 초기 값들을 제공합니다; 각각은 $P:R=V$ 의 형태로, P 는 프로세스 식별자이고, R 은 레지스터 식별자이며, V 는 값입니다. 예를 들어, 프로세스 0의 레지스터 $r3$ 는 초기에 2라는 값을 담고 있습니다. 만약 값이 변수 (예제의 x , y , 또는 z) 라면 레지스터는 해당 변수의 주소로 초기화 됩니다. 변수의 값을 초기화 시키는 것도 가능한데, 예를 들어 $x=1$ 은 x 의 값을 1로 초기화 시킵니다. 초기화 되지 않은 변수들은 기본값으로 0을 갖게 되어서, 이 예제에서는 x , y , 그리고 z 는 모두 초기에 0의 값을 갖습니다.

Line 7 은 두 프로세스를 위한 식별자를 제공하므로, line 4 의 $0:r3=2$ 는 $P0:r3=2$ 로 대신 쓰여질 수 있습니다. Line 7 은 있어야만 하며, 식별자들은 Pn 의 형태여야만 하며, n 은 열 수로, 가장 왼쪽의 열을 0으로 세는 것부터 시작합니다. 이는 불필요하게 엄정한 것으로 보여질 수도 있겠습니다만, 이는 실제 사용에 있어 많은 혼란을 방지해 줍니다.

Quick Quiz 12.23: Figure 12.25 의 line 8 은 왜 레지스터들을 초기화 시키죠? 왜 그것들을 line 4 와 5 에서 대신 초기화 시키지 않나요? ■

Line 8-17 은 각각의 프로세스를 위한 코드들입니다. 프로세스는 $P0$ 의 line 11 과 $P1$ 의 line 12-17 처럼 빈 줄을 가질 수도 있습니다. 라벨과 브랜치들 역시 사용 가능한데, line 14 의 브랜치와 line 17 의 라벨로 사용이 보여져 있습니다. 그렇다면 하나, 너무 자유로운 브랜치 사용은 상태 공간의 크기를 폭증시킬 수 있습니다. 루프의 사용은 특히나 상태 공간의 크기를 폭증시키기 위한 좋은 방법입니다.

Line 19-20 은 단정문을 보이는데, 이 경우에는 두 쓰레드가 모두 실행을 완료한 후에 $P0$ 의, 그리고 $P1$ 의 $r3$ 레지스터들이 모두 0을 가지고 있을 수 있는지에 우리가 관심을 갖고 있음을 보입니다. 만약 $P0$ 와 $P1$ 이 모두

```

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

각각의 $r3$ 레지스터에서 0을 보게 된다면 안타깝게도 실패할 수 있는 수많은 경우들이 존재할 수 있기에 이 단정문은 중요합니다.

이것만으로도 간단한 리트머스 테스트들을 구성하는데 충분한 정보가 될 겁니다. 몇가지 추가적인 문서화가 가능합니다만, 이런 추가적인 문서화들은 대부분 테스트를 실제 하드웨어 위에서 수행하기 위한 다른 연구용 도구를 위한 것입니다. 어쩌면 더 중요한 것은, 많은 수의 이미 존재하는 리트머스 테스트들은 온라인 도구를 통해서도 (“Select ARM Test” 와 “Select POWER Test” 벌느들을 통해서) 사용 가능하다는 것입니다. 이런 미리 존재하는 리트머스 테스트들 가운데 하나가 당신의 Power 나 ARM 메모리 순서 규칙에 대한 질문에 대한 답이 될 수도 있을 겁니다.

12.2.2 What Does This Litmus Test Mean?

$P0$ 의 line 8 과 9 는 C 언어의 $x=1$ 과 같은데, line 4 에서 $P0$ 의 레지스터 $r2$ 가 x 의 주소를 갖도록 정의했기 때문입니다. $P0$ 의 line 12 와 13 은 각각 load-linked (ARM 용어로는 “load register exclusive” 이고 Power 용어로는 “load reserve”) 와 store-conditional (ARM 용어로는 “store register exclusive”) 입니다. 이것들이 함께 사용되면, x86 에서는 `lock; cmpxchg` 인스트럭션으로 표현되는 compare-and-swap 시퀀스와 유사한 어토믹 인스트럭션을 형성하게 됩니다. 더 높은 차원의 추상화 단계로 이야기 하면, line 10-15 의 시퀀스는 리눅스 커널의 `atomic_add_return(&z, 0)` 와 같습니다. 마지막으로, line 16 은 C 언어의 $r3=y$ 와 같습니다.

$P1$ 의 line 8 과 9 는 C 언어의 $y=1$ 과 같으며, line 10 은 리눅스 커널의 `smp_mb()` 와 동일한 메모리 배리어이며, line 11 은 C 언어의 $r3=x$ 와 같습니다.

Quick Quiz 12.24: Figure 12.25 의 line 17 의 $Fail:$ 라벨에서는 무슨 일이 벌어지게 되는거죠? ■

이 모든 것들을 한자리에 모아서 만든, 전체 리트머스

```
./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;
0k
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

테스트와 동일한 C-언어 코드가 Figure 12.26에 보여져 있습니다. 핵심은, `atomic_add_return()`이 (리눅스 커널의 요구사항이 그렇듯) 전체 메모리 배리어처럼 동작한다면, 실행이 완료된 후에 `P0()`의, 그리고 `P1()`의 `r3` 가 모두 0일 수는 없다는 것입니다.

다음 섹션은 이 리트머스 테스트를 어떻게 수행할 수 있는지 설명합니다.

12.2.3 Running a Litmus Test

리트머스 테스트들은 <http://www.cl.cam.ac.uk/~pes20/ppcmem/>를 통해서 대화형으로 수행되어 메모리 모델에 대한 이해를 도울 수 있습니다. 하지만, 이 방법은 사용자가 직접 전체 상태 공간 탐색을 해야 할 것을 필요로 합니다. 모든 가능한 이벤트들의 시퀀스를 체크했다고 자신하긴 어려우므로, 이런 목적을 위한 별개의 도구가 제공됩니다 [McK11c].

Figure 12.25에 보인 리트머스 테스트는 `read-modify-write` 인스트럭션들을 포함하고 있기 때문에, 커맨드 라인에 `-model` 인자를 추가해야만 합니다. 해당 리트머스 테스트가 `filename.litmus`에 저장되어 있다면, 이는 Figure 12.27에 보여진 것과 같은 결과를 보일 것인데, 여기서 ...은 많은 양의 빌드 진행상의 출력을 대체하고 있습니다. 마지막 줄의 “Sometimes”는 이를 증명합니다: 해당 단정문의 조건은 항상 그런건 아니지만 일부 수행에서는 일어날 수 있습니다.

이 리눅스 커널 버그를 고치는 방법은 `P0`의 `isync`를 `sync`로 교체하는 것으로, 이렇게 수정된 테스트는 Figure 12.28에 보인 것과 같은 결과를 나오게 합니다. 볼 수 있듯, `0:r3=0; 1:r3=0;`는 상태들의 리스트에 나타나 있지 않고, 마지막 줄은 “Never”라고 이야기 합니다. 따라서, 해당 모델은 문제가 되는 실행 시퀀스는 일어날 수 없음을 예측합니다.

Quick Quiz 12.25: ARM 리눅스 커널도 비슷한 버그를 가지고 있나요? ■

```
./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

12.2.4 PPCMEM Discussion

이 도구들은 ARM과 Power에서 돌아가는 낮은 단계의 병렬 기능들을 사용하는 사람들에게 커다란 도움이 될 것을 약속합니다. 이 도구들은 일부 본질적인 한계점을 가지고 있습니다:

1. 이 도구들은 연구용 프로토타입이고, 그것 자체에 있어서는 기능을 지원하지 않습니다.
2. 이 도구들은 IBM이나 ARM으로부터 각각의 CPU 아키텍쳐들에 대한 공식적 언급을 받은 바가 없습니다. 예를 들어, 두 회사 모두 언제든 이 도구들의 어떤 버전들에 대해서도 버그가 있다고 이야기할 수가 있습니다. 따라서 이 도구들은 실제 하드웨어에서의 조심스러운 스트레스 테스트를 대체제가 될수는 없습니다. 더구나, 이것들의 기반을 형성하고 있는 도구들도 모델들도 모두 활발히 개발되는 중이고 언제든 바뀔 수 있습니다. 한편, 이 모델은 적절한 하드웨어 전문가의 자문에 의해 개발되었기에, 이것은 해당 아키텍쳐들의 든든한 표현이라고 확신을 가져도 좋을 이유가 있는 셈입니다.
3. 이 도구들은 현재 인스트럭션집합 중 부분집합만을 다루고 있습니다. 이 부분집합은 제 목적에 있어서는 충분했습니다만, 여러분의 목적은 다양할 수 있습니다. 구체적으로, 이 도구는 워드 크기 (32 bits)의 액세스들만을 다루며, 이렇게 액세스 되는 워드들은 올바르게 정렬되어 있어야만 합니다. 또한, 이 도구는 ARM 메모리 배리어 인스트럭션의 더 완화된 변종들도, 산술 인스트럭션들도 다루지 않습니다.
4. 이 도구들은 적은 수의 쓰레드에서 돌아가는, 작고 루프를 사용하지 않는 코드 조각들에서 사용될 것으로 제약되어 있습니다. 더 커다란 예제들은 Promela와 spin 같은 비슷한 도구들과 같이 상태 공간의 폭증을 초래합니다.

5. 전체 상태 공간 탐색은 어떻게 각각의 문제되는 상태에 도달했는지를 알려주지 않습니다. 그렇다면 하나, 특정 상태가 정말로 도달할 수 있다는 사실을 깨닫게 되면, 대화형 도구를 사용해서 그 상태를 찾아나서는 것은 일반적으로 어렵지 않습니다.
6. 이 도구들은 복잡한 데이터 구조에는 그다지 좋지 않습니다만, 극단적으로 간단한 링크드 리스트를 “`x=y; y=z; z=42;`”와 같은 형태의 초기화 statement들을 사용해서 생성하고 탐색하는 것은 가능하긴 합니다.
7. 이 도구들은 memory mapped I/O 나 디바이스 레지스터들에 대해서는 처리하지 않습니다. 물론, 그런 것들을 처리하는 것은 그것들이 형식화 될 것을 필요로 하는데, 여기엔 존재하지 않습니다.
8. 이 도구들은 단정문을 통해 여러분이 코딩한 문제들에 대해서만 문제를 찾아줄 겁니다. 이 약점은 모든 형식적 방법들에 공통적인 것이고, 왜 테스트가 여전히 중요한지에 대한 또 다른 이유입니다. 이 챕터의 시작에서 인용한 Donald Knuth의 영원한 말처럼, “다음 코드의 버그들을 경계하세요; 전 그것이 올바르다는 걸 증명했을 뿐이지, 그걸 사용해 보진 않았습니다.”

그렇다면 하나, 이 도구들의 강점 가운데 하나는 해당 아키텍쳐에 의해서 허용되는, 합법적이지만 현재 하드웨어 구현이 부주의한 소프트웨어 개발자들에게 아직 가하지 않은 모든 동작들의 전체 범위를 모델링하도록 설계되었다는 점입니다. 따라서, 이 도구들에서 진료된 알고리즘들은 실제 하드웨어에서 동작할 때에는 추가적인 안전성 마진을 가질 수 있을 확률이 큽니다. 더 나아가서, 실제 하드웨어에서 테스트를 해보는 것은 버그를 찾을 수 있게 해줄 뿐입니다; 그런 테스트로 해당 사용이 올바른지에 대한 증명을 하는 것은 본질적으로 불가능합니다. 이를 이해하기 위해, 연구자들이 그들의 모델의 검증을 위해 실제 하드웨어 위에서 천억개의 테스트를 일일이 수행하고 있는 것을 생각해 보세요. 어떤 경우에 있어서는, 아크텍쳐에 의해 허용되어 있는 행동이 1760억 회의 테스트 수행에도 불구하고 나타나지 않은 적도 있었습니다 [AMP¹¹]. 반면에, 이 전체 상태 공간 탐색은 해당 도구가 코드 조각의 정확성을 증명할 수 있게 해줍니다.

형식적 방법들과 도구들은 테스트의 대체제가 아님을 다시 한번 이야기할 필요가 있습니다. 예를 들어 리눅스 커널과 같은 커다란 안정적인 동시적 소프트웨어 작품을 만들어내는 것은 상당히 어려운 일입니다. 따라서 개발자들은 이 목표를 위해 모든 가능한 도구들을 적용할 준비가 되어 있어야 합니다. 이 챕터에서 보인 도구들은 테스트를 통해 발생시키기 (그리고 추적하기)

상당히 어려운 버그들을 찾아내는 것을 가능하게 해줍니다. 한편, 테스트는 이 챕터에서 소개된 도구들로 처리할 수 있는 것들보다 훨씬 커다란 몸집의 소프트웨어들에 적용될 수 있습니다. 항상 그렇듯이, 작업에 결맞는 도구를 사용하세요!

물론, 여러분의 병렬 코드를 쉽게 분할될 수 있도록 설계하고 (락, 시퀀스 카운터, 어토믹 오퍼레이션, 그리고 RCU 같은) 고차원의 도구들을 사용해서 일을 더 간단하게 되도록 함으로써 이런 단계에서의 일을 할 필요 자체를 없애는 게 항상 최선입니다. 그리고 여러분의 일을 처리하기 위해 반드시 낮은 단계의 메모리 배리어들과 read-modify-write 인스트럭션들을 사용해야만 하는 경우라 할지라도, 이 날카로운 도구들을 더 많이 보수적으로 사용할수록 여러분의 삶이 더 쉬워질 겁니다.

12.3 Axiomatic Approaches

PPCMEM 도구가 Figure 12.29에 보여진, 유명한 “independent reads of independent writes” (IRIW) 리트머스 테스트를 해결할 수 있긴 합니다만, 그러기 위해선 14시간 이상의 CPU 시간과 10기가바이트 이상의 상태 공간을 필요로 합니다. 그렇다면 하나, 이 상황은 이 문제를 풀기 위해선 커다란 레퍼런스 매뉴얼을 뒤져보고, 증명을 시도하고, 전문가와 토론을 하고, 마지막으로 내린 답에 대해서도 확신할 수 없었던, PPCMEM이 나오기 전에 비하면 커다란 개선이 이뤄진 것입니다. 비록 14시간은 긴 시간처럼 보일 수 있겠지만, 이는 수주나 수개월에 비하면 너무나도 짧은 시간입니다.

하지만, 두개의 쓰레드가 두개의 별도의 변수에 값을 쓰고 두개의 다른 쓰레드가 이 두개의 변수로부터 반대 순서로 값을 읽어들일 뿐인 해당 리트머스 테스트의 단순성을 놓고 보면 요구되는 해당 시간은 조금 놀랍습니다. 단정문은 두개의 값을 읽어들이는 쓰레드들이 두개의 값 저장의 순서에 대해 서로 다른 의견을 갖는다면 터집니다. 이 리트머스 테스트는 긴단한데, 표준적 메모리 순서 리트머스 테스트들을 놓고 봐도 그렇습니다.

소모되는 시간과 공간의 양에 대한 한가지 이유는 PPCMEM이 추적 기반의 전체 상태 공간 탐색을 한다는 것으로, 이는 아크텍쳐 단계에서 이벤트들의 모든 가능한 순서와 조합을 만들어내고 수행해 봐야 함을 의미합니다. 이 단계에서, 화려한 이벤트와 액션들의 시퀀스에 연관된 로드와 스토어는 모두 모두 탐색되어야만 하는 매우 커다란 상태 공간을 초래하게 되고, 이는 커다란 메모리와 CPU 소모로 이어지게 됩니다.

물론, 그런 추적들 가운데 많은 것들은 다른 것들과 상당히 유사해서, 비슷한 추적들을 하나로 취급하는 것이 성능을 개선시킬 수도 있을 것임을 시사합니다. 그런 한가지 방법이 Alglave 등의 공리적 집합론 방법 [AMT14]으로, 여기선 메모리 모델을 나타내기 위한

```

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

공리 집합을 만들어내고 리트머스 테스트들을 이 공리들의 집합으로 증명되거나 반증될 수 있는 정리로 변환시킵니다. 이렇게 만들어진, “herd” 라 불리는 도구는 편리하게 PPCMEM 에서와 같은 리트머스 테스트들을 입력으로 받는데, Figure 12.29 에 보인 IRIW 리트머스 테스트도 포함됩니다.

IRIW 를 푸는데에 PPCMEM 이 14 CPU 시간을 필요로 하는 반면, herd 는 17 밀리세컨드만에 IRIW 를 푸는데, 이는 백만배 이상의 속도 향상을 의미합니다. 그렇다면 하나, 문제는 근본적으로 기하급수적이므로, 더 커다란 문제들에 있어서는 herd 역시 기하급수적으로 느려질 것을 예상해야 합니다. 그리고 이는 실제로 일어나는 일인데, 예를 들어 우리가 Figure 12.30 에 보여진 것처럼 쓰기를 하는 CPU 마다 네개의 쓰기를 추가하면, herd 는 50,000 배 이상 느려져서 15 분 이상의 CPU 시간을 필요로하게 됩니다. 쓰레드를 추가하는 것 역시 기하급수적인 속도 저하를 초래합니다 [MS14].

이런 근본적 기하급수적 성질에도 불구하고, PPCMEM 과 herd 는 x86 시스템에서의 queued-lock handoff 를 포함해서 핵심적 병렬 알고리즘을 체크하는데에 상당히 유용한 것으로 증명되었습니다. Herd 의 약점은 Section 12.2.4 에서 이야기한 PPCMEM 의 그것과 유사합니다. PPCMEM 과 herd 가 동의하지 않게 되는 불분명한 (하지만 매우 현실적인) 경우들이 존재하는데 2014년 말 현재까지는 이 이견문제를 해결하는 노력이 진행 중입니다.

장기적으로, 희망은 공리적 방법이 더 높은 단계의 소프트웨어의 것들을 설명하는 공리들을 포함하는 것입니다. 이는 잠재적으로 훨씬 더 커다란 소프트웨어 시스템의 공리적 증명을 가능하게 할 것입니다. 또 다른 대안은 다음 섹션에서 설명하는 것처럼 이진 논리의 공리들을 제공하는 것으로, 다음 섹션에서 다룹니다.

12.4 SAT Solvers

제한된 횟수의 루프와 재귀를 갖는 모든 유한한 프로그램은 입력에 대한 그 프로그램의 단정들을 표현하는 논리 표현식으로 변환될 수 있습니다. 그런 논리 표현식이 있다면, 어떤 입력의 조합이 단정문 가운데 하나를 터지게 만들 수 있는지 알 수 있는지 여부를 아는 것은 매우 흥미로울 겁니다. 만약 입력이 이진 변수들의 조합으로 표시된다면, 이는 곧 satisfiability problem 이라고도 알려진 SAT 입니다. SAT solver 들은 하드웨어의 검증에서 많이 사용되어져 왔으며, 여기서 커다란 진보의 동기를 주었습니다. 1990년대 초의 월드 클래스 SAT solver 는 100 개의 별개의 이진 변수들로 표현된 논리 표현식을 다룰 수 있습니다만, 2010년대 초에 와서는 백만개의 변수로 이루어진 SAT solver 들이 사용 가능합니다 [KS08].

또한, SAT solver 들을 위한 프론트 엔드 프로그램들은 C 코드를 자동으로 논리 표현식으로 변환시킬 수 있으며, 이 때 단정문들을 취하기도 하고 배열 크기 침범 에러들과 같은 에러 조건들을 위한 단정문을 생성하기도 합니다. 한 가지 예는 C bounded model checker, 또는 cbmc 라 알려진 것으로, 많은 리눅스 배포판에 포함되어 있어서 사용 가능합니다. 이 도구는 사용하기가 상당히 간단해서, cbmc test.c 만으로도 test.c 를 검증하기에 충분합니다. 이런 사용상의 편의성은 휴귀 테스트 프레임워크에 포함되고 있는 형식적 검증으로의 문을 열어주기 때문에 매우 중요합니다. 대조적으로, 특정 목적의 언어로의 사소하지 않은 변환을 필요로 하는 전통적인 도구들은 설계 시점에서의 검증으로만 국한되어졌습니다.

근래에 들어서, SAT solver 들은 병렬 코드를 처리하는 형태로도 나타났습니다. 이런 solver 들은 입력이 되는 코드를 single static assignment (SSA) 형태로 변환시키고는 여기서 생겨날 수 있는 모든 액세스 순서들을 생성합니다. 이 방법은 잘 동작할 듯 보입니다만, 실제

```

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

환경에서 얼마나 잘 동작할지는 볼 필요가 있습니다. 예를 들어, 이 테크닉이 어떤 타입과 크기의 프로그램들을 처리할 수 있을지는 분명하지 않습니다. 하지만, SAT solver들이 병렬 코드를 검증하는데 유용해 지기를 빌어볼 만한 이유들이 있습니다.

12.5 Summary

이 챕터에서 설명한 형식적 검증 테크닉은 작은 병렬 알고리즘들을 검증하는데에는 매우 강력한 도구입니다만, 당신의 도구상자에 그것들만 있어선 안됩니다. 지난 수십년간의 형식적 검증에 대한 집중에도 불구하고, 테스트는 커다란 병렬 소프트웨어 시스템을 위한 대표적 검증 도구로 남아있습니다 [Cor06a, Jon11].

물론 이는 항상 그렇지는 않을 수도 있는 것이 사실입니다. 이를 확실히 하기 위해, 2013년 현재에 와서는 십여개가 넘는 리눅스 커널의 인스턴스가 존재한다는 점을 생각해 보세요. 리눅스 커널에 평균적으로 백만년의 수행 중 한번 발생할 수 있는 버그가 있다고 생각해 봅시다. 앞의 챕터의 마지막에서 설명했다시피, 이 버그는 이 전체 설치된 환경에서 하루에 세번씩 나타날 겁니다. 하지만 대부분의 형식적 검증 테크닉은 매우 작은 코드에 대해서만 사용될 수 있다는 사실도 여전합니다. 그런 상황에서 동시적 코드를 짜는 사람들은 뭘 해야 할까요?

한가지 방법은 첫번째 버그, 첫번째로 관련있는 버그, 마지막으로 관련있는 버그, 그리고 마지막 버그를 찾는 것에 대해 생각해 보는 겁니다.

첫번째 버그는 코드 검사난 컴파일러의 조사를 통해 발견되어집니다. 최신의 컴파일러에 의해 제공되는, 지속적으로 세련되어지는 진단 기능이 가벼운 형태의 형

식적 검증으로 여겨질 수도 있긴 하지만, 그것들을 그런 용어로 부르는건 흔하지 않습니다. 이는 부분적으로는 “내가 그걸 사용하고 있다면, 그것은 형식적 검증일 수 없다”라고 말하는, 이상한 실무자의 선입견 때문이기도 하고, 컴파일러 진단기능과 검증에 대한 연구 사이의 서로 다른 철학 때문이기도 합니다.

첫번째로 관련있는 버그는 코드 검사나 컴파일러 진단 기능으로 발견되어질 수도 있지만, 이 두 단계가 오타와 거짓 양성 반응만을 찾게 되는 것도 흔하지 않은 일은 아닙니다. 어떤 방식으로든, 실제 제품에서 마주하게 될 버그들인 관련된 많은 버그들은 많은 경우 테스트를 통해 발견되어질 겁니다.

테스트가 예측된 사용 예에서 만들어졌든 진짜 사용 예에서 만들어졌든, 마지막 연관된 버그가 테스트를 통해 발견되어지는 것은 흔치 않은 일이 아닙니다. 이 상황은 형식적 검증에 대한 완전한 거부의 동기가 될 수도 있겠습니다만, 관계없는 버그들은 black-hat 공격의 덕에 최소한의 알맞은 사이즈에 있어서는 갑자기 관계 있는 것이 되어버리는 안좋은 습관을 가지고 있습니다. 전체 소프트웨어 가운데 그 분포도를 지속적으로 높여가는 보안에 관련된 소프트웨어에 있어서는 마지막 버그를 찾아내고 고쳐내려는 강한 동기가 존재할 수 있습니다. 테스트는 마지막 버그를 찾아내는 것은 명백히 불가능하므로 형식적 검증에게만 가능한 역할이 존재합니다. 형식적 검증이 그 안으로 적용될 수 있다고만 하면 그런 역할이 있다는 이야기입니다. 이 챕터에서 보였듯이, 현재의 형식적 검증 시스템들은 상당히 제한적입니다.

또 다른 방법은 형식적 검증이 많은 경우 테스트보다 적용하기가 어렵다는 점을 고려하는 겁니다. 물론 이는 문화적인 측면에서의 이야기일 수 있고, 형식적 검증이

더 많은 사람들에게 친숙해 질수록 더 쉽게 전파될 것이라는 희망을 가져볼 수도 있습니다. 그렇다고는 하나, 매우 간단한 테스트 사용은 임의의 거대한 소프트웨어 시스템에서 심각한 버그들을 찾아낼 수 있습니다. 반면에, 형식적 검증을 적용하기 위해 필요한 노력은 시스템의 크기가 증가할수록 극적으로 증가합니다.

전 20년이 넘도록 형식적 검증을 형식적 검증이 효력을 발휘하는 곳에서 필요할 때에만 사용했을 뿐인데, 설계 시점에서의, 무엇보다 중요한 소프트웨어 구성물의 작고 복잡한 부분의 검증에 있어 그랬습니다. 무엇보다 중요하지만 더 커다란 소프트웨어 구성물들은 물론 테스트로 검증했습니다.

Quick Quiz 12.26: L4 마이크로커널의 전체 검증을 생각해 보면, 이런 형식적 검증에 대한 제한적인 시각은 약간 시대에 뒤쳐진 것 아닌가요? ■

마지막 방법은 다음의 두개의 정의와 그것들이 암시하는 결론에 대해서 고려해 보는 것입니다:

정의: 버그가 없는 프로그램들은 사소한 프로그램들이 다.

정의: 신뢰할 수 있는 프로그램은 알려진 버그가 없다.

결론: 모든 사소하지 않으며 신뢰할 수 있는 프로그램에는 최소 하나의 아직 알려지지 않은 버그가 있다.

이 시점에서 보면, 모든 검증 영역에서의 진보는 두 가지 영향을 가질 수밖에 없을 겁니다: (1) 사소한 프로그램들의 갯수의 증가 또는 (2) 신뢰할 수 있는 프로그램들의 수의 감소. 물론, 인류의 멀티코어 시스템과 소프트웨어에 대한 증가하는 의존도는 사소한 프로그램들의 갯수의 가파른 증가에 대한 커다란 동기가 될겁니다!

하지만, 만약 여러분의 코드가 너무 복잡해서 여러분이 형식적 검증 도구에 너무 과하게 의존하고 있다는 점을 알게 된다면, 여러분은 여러분의 설계를 다시 세심하게 생각해 봐야 하는데, 당신의 형식적 검증 도구들이 당신의 코드가 특정 목적 언어로 손으로 변환해야 하게 되는 상황이라면 특히 그렇습니다. 예를 들어, Section 12.1.5에서 보인 preemption 가능한 RCU의 복잡한 dynticks 인터페이스 구현에 있어서는 Section 12.1.6.9에서 이야기한대로 훨씬 간단한 대안적 구현이 존재함이 드러났습니다. 다른게 모두 동일하다면, 복잡한 구현을 위한 기계적 증명보다 간단한 구현이 훨씬 낫습니다!

그리고 형식적 검증 테크닉들과 시스템들에 대한 열려있는 도전은 이 요약 내용이 틀리다고 증명하는 것입니다!

Chapter 13

You don't learn how to shoot and then learn how to launch and then learn to do a controlled spin—you learn to launch-shoot-spin.

“Ender’s Shadow”, Orson Scott Card

Putting It All Together

이 챕터는 일부 동시적 프로그래밍의 퍼즐들을 처리하는데 대한 약간의 힌트를 제공하는데, Section 13.1에서 카운터의 난문제로 시작해서, Section 13.3에서 일부 RCU 구조로 이어지고, Section 13.4에서의 해싱 노력으로 마무리 합니다.

13.1 Counter Conundrums

이 섹션은 일부 카운터 문제에 대한 해결책들을 개략적으로 소개합니다.

13.1.1 Counting Updates

Schrödinger (Section 10.1를 참고하세요) 가 각 동물의 업데이트 횟수를 카운트 하고자 하고, 이 업데이트들은 데이터 원소별 락을 통해 동기화 된다고 해봅시다. 이 카운팅은 어떻게 해야 최선일까요?

물론, Chapter 5에서의 카운팅 알고리즘들 중 무엇이든 고려해 볼 수 있을 겁니다만, 이경우에 최적의 방법은 훨씬 간단합니다. 카운터를 각각의 데이터 원소에 위치시키고, 그 원소의 락으로 보호한 채로 카운터를 증가시키세요!

13.1.2 Counting Lookups

Schrödinger 가 각 동물에의 검색의 횟수 역시 세고 싶어하며, 검색은 RCU를 통해 보호되고 있다고 생각해봅시다. 이런 카운팅은 어떻게 하는게 최선일까요?

한 가지 방법은 Section 13.1.1에서 이야기한 원소별 락을 사용해서 검색 카운터를 보호하는 것일 겁니다. 불행히도, 이는 모든 검색이 락을 잡을 것을 필요로 해서, 커다란 시스템에서는 상당한 병목지점이 될겁니다.

또 다른 방법은 카운팅이 “안된다고 하기”로, noatime 마운트 옵션의 예를 따릅니다. 이 방법이 사용 가능한 경우라면, 이게 최고의 방법임이 분명합니다:

무엇보다, 아무것도 하지 않는것보다 빠른 것은 없습니다. 검색 카운터가 면제될 수 없다면, 마저 읽어주세요!

Chapter 5에 나온 모든 카운터들이 사용될 수 있는데, Section 5.2에서 설명한 통계적 카운터가 아마도 가장 흔한 선택이 될겁니다. 하지만, 이는 커다란 메모리 사용량을 초래합니다: 요구되는 카운터들의 갯수는 데이터 원소들의 갯수에 쓰레드의 갯수를 곱한 값이 됩니다.

이런 메모리 오버헤드가 지나치다면, CPU 별 카운터 대신 소켓별 카운터를 유지하고 Figure 10.8.에 보여진 것처럼 해시 테이블 성능 결과를 주목하는 방법이 있습니다. 이는 카운터 증가는 어토믹 오퍼레이션이 될 것을 필요로 할 것인데, 특히 특정 쓰레드가 언제든 다른 CPU로 옮겨갈 수 있는 사용자 모드 수행에서는 특히 그렇습니다.

만약 일부 원소들이 매우 자주 검색된다면, 쓰레드별 로그를 유지하며 특정 원소를 위한 여러 로그 항목들이 병합될 수 있는 식으로 업데이트들을 몰아서 처리하는 방법들이 있습니다. 특정 로그 항목이 충분히 많은 횟수 증가되었거나 충분히 많은 시간이 지났다면, 해당 로그 항목은 연관된 데이터 항목에 적용될 수 있을 겁니다. Silas Boyd-Wickizer 는 이런 노선을 정형화 시켰습니다 [BW14].

13.2 Refurbish Reference Counting

레퍼런스 카운팅이 개념적으로는 간단한 테크닉이지만, 동시적 소프트웨어에 적용되면 그 디테일 안에 많은 악마들이 숨어있습니다. 무엇보다도, 오브젝트가 지나치게 일찍 폐기되는 일이 없는 오브젝트라면, 애초에 레퍼런스 카운터가 필요하지도 않았을 겁니다. 하지만 오브젝트가 폐기될 수 있다면, 레퍼런스를 얻어오는 프로세스 그 자체가 진행되는 동안 폐기되는 것을 무엇으로 막을 수 있을까요?

동시적 소프트웨어에서 사용되기 위해 레퍼런스 카운터를 재정비하는 몇가지 방법들이 있는데, 다음과 같

Acquisition Synchronization	Release Synchronization		
	Locking	Reference Counting	RCU
Locking	-	CAM	CA
Reference Counting	A	AM	A
RCU	CA	MCA	CA

Table 13.1: Reference Counting and Synchronization Mechanisms

은 것들이 포함됩니다:

1. 레퍼런스 카운트를 조정하는 동안에는 바깥에 존재하는 오브젝트의 락을 잡아야만 하기.
2. 오브젝트는 0이 아닌 레퍼런스 카운트와 함께 생성되고, 새로운 레퍼런스들은 레퍼런스 카운터의 현재 값이 0이 아닐 때에만 획득될 수 있게 하기. 어떤 레퍼런스가 특정 오브젝트로의 레퍼런스를 가지고 있지 않다면, 이미 레퍼런스를 가지고 있는 다른 쓰레드의 도움으로 해당 오브젝트로의 레퍼런스를 얻을 수 있음.
3. 해당 오브젝트를 위한 존재 보장이 제공되어서, 해당 오브젝트가 다른 무언가가 레퍼런스를 얻으려 시도하는 동안은 해제되지 않도록 하기. 존재 보장은 자동화된 *garbage collector*를 통해서, 또는 Section 9.5에서 보여지듯이 RCU를 통해서 제공되곤 합니다.
4. 오브젝트를 위한 탑업 안정성을 제공하기. 레퍼런스가 일단 획득되면 추가적인 아이덴티티 체크가 이뤄져야만 합니다. 탑업 안정성 보장은 예를 들어, Section 9.5에서 보여진 것과 같이 리눅스 커널 안에서 `SLAB_DESTROY_BY_RCU` 기능과 같이 특수 목적 메모리 할당자를 통해 제공될 수 있습니다.

물론, 존재 보장을 제공하는 모든 메커니즘은 그 정의에 의해 탑업 안정성 보장도 제공합니다. 따라서 이 섹션은 마지막의 두 단계를 RCU의 전례 하에 레퍼런스 획득 보호에 일반적으로 사용되는 세 개의 카테고리로 그룹지어 보겠습니다.: 레퍼런스 카운팅, 시퀀스 락킹, 그리고 RCU.

Quick Quiz 13.1: 레퍼런스 획득을 단순히 레퍼런스 카운터의 값이 0이 아닌 경우에만 레퍼런스를 획득하는 *compare-and-swap* 오퍼레이션으로 간단하게 만들지는 않는거죠? ■

레퍼런스 카운팅의 핵심 이슈는 레퍼런스의 획득과 오브젝트의 해제 사이의 동기화란 점을 놓고 보면, 우린 Table 13.1에 보인 것과 같은 아홉개의 메커니즘 조합들이 존재 가능합니다. 이 표는 레퍼런스 카운팅 메커니즘들을 다음과 같이 넓은 카테고리들로 나눕니다:

1. 어토믹 오퍼레이션, 메모리 베리어, 정렬 제약도 없는 간단한 카운팅 (“-”).
2. 메모리 베리어 없이 진행되는 어토믹 카운팅 (“A”).
3. 릴리즈 시에만 메모리 베리어를 사용하는 어토믹 카운팅 (“AM”).
4. 어토믹 획득 오퍼레이션과 릴리즈 시에만 필요시 되는 메모리 베리어들과 조합된 체크를 하는 어토믹 카운팅 (“CAM”).
5. 어토믹 획득 오퍼레이션과 조합된 체크를 하는 어토믹 카운팅 (“CA”).
6. 어토믹 획득 오퍼레이션과 레퍼런스 획득 때에도 필요시 되는 메모리 베리어들과 조합된 체크를 사용하는 어토믹 카운팅 (“MCA”).

하지만, 값을 리턴하는 모든 리눅스 커널 어토믹 오퍼레이션들은 메모리 베리어를 포함하도록 정의되어 있으므로,¹ 모든 릴리즈 오퍼레이션들은 메모리 베리어를 포함하고, 모든 체크되는 레퍼런스 획득 오퍼레이션들 또한 메모리 베리어를 포함하게 됩니다. 따라서, “CA”와 “MCA”는 “CAM”과 동일해서, 앞의 네개의 경우들을 위한 섹션들만이 남게 됩니다: “-”, “A”, “AM”, 그리고 “CAM”. 레퍼런스 카운팅을 지원하는 리눅스의 기능들은 Section 13.2.2에 소개되어 있습니다. 뒤의 섹션들은 레퍼런스 획득과 해제가 매우 빈번할 때, 그리고 레퍼런스 카운트가 매우 가끔씩만 0인지 여부를 체크해야 하는 경우에 대해 성능을 개선할 수 있는 최적화 방법들을 인용합니다.

13.2.1 Implementation of Reference-Counting Categories

락킹으로 보호되는 간단한 카운팅 (“-”)이 Section 13.2.1.1에 설명되고, 메모리 베리어 없이 수행되는 어토믹한 카운팅 (“A”)이 Section 13.2.1.2에 설명되고, 레퍼런스 획득 시의 메모리 베리어와 함께 수행되는 어토믹한 카운팅 (“AM”)이 Section 13.2.1.3에 설명되며, 레퍼런스 획득 시의 메모리 베리어와 체크와 함께 수행되는 어토믹한 카운팅 (“CAM”)이 Section 13.2.1.4에서 설명됩니다.

13.2.1.1 Simple Counting

어토믹 오퍼레이션도 메모리 베리어도 사용하지 않는 간단한 카운팅은 레퍼런스 카운터 획득과 해제가 모두

¹ `atomic_read()`와 `ATOMIC_INIT()`는 이 규칙에 예외가 됩니다.

```

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 *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 13.1: Simple Reference-Count API

같은 락으로 보호될 때에 사용될 수 있습니다. 이 경우에 레퍼런스 카운터 자체는 어토믹하지 않게 조정될 것이란 점이 분명한데, 락은 모든 필요한 배타성, 메모리 배리어, 어토믹 인스트럭션, 그리고 컴파일러 최적화에 대한 방지를 제공하기 때문입니다. 이는 해당 락이 레퍼런스 카운트 만이 아니라 다른 오퍼레이션들도 보호해야 하지만 해당 오브젝트로의 레퍼런스가 해당 락을 해제한 후에 잡아야만 할 때에 선택될 수 있는 방법입니다. Figure 13.1은 간단한 어토믹하지 않은 레퍼런스 카운팅을 구현하는데 사용될 수 있는 간단한 API를 보입니다—간단한 레퍼런스 카운팅은 거의 항상 인터페이스 없이 곧바로 코딩되곤 하지만요.

13.2.1.2 Atomic Counting

간단한 어토믹 카운팅은 레퍼런스를 획득하는 모든 CPU는 레퍼런스를 이미 잡고 있어야 하는 경우에 사용될 수 있습니다. 이 스타일은 하나의 CPU가 자신의 사용을 위해 오브젝트를 생성하지만 나중에 태어난 다른 CPU, 테스크, 타이머 핸들러, 또는 I/O 완료 핸들러들이 그 오브젝트에 접근할 수 있도록 해야만 할 때 사용됩니다. 이 오브젝트를 넘겨주는 CPU는 받게 되는 오브젝트를 위해 새로운 레퍼런스를 먼저 획득해야만 합니다. 리눅스 커널에서는, kref 기능이 이런 스타일의 레퍼런스 카운팅을 구현하기 위해 사용되는데, Figure 13.2에 보여져 있습니다.

모든 레퍼런스 카운팅 오퍼레이션들이 락킹으로 보호되는 않는데, 이는 두개의 다른 CPU들이 동시에 레퍼런스 카운트를 조정할 수 있음을 의미하기 때문에

```

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 13.2: Linux Kernel kref API

어토믹 카운팅이 필요해집니다. 평범한 값 증가와 감소 오퍼레이션이 사용된다면, 한쌍의 CPU들이 둘 다 레퍼런스 카운트를 동시에 가져와서, “3”이라는 값을 읽을 수 있을 겁니다. 둘 다 그 값을 증가시키려 한다면, 이들은 모두 “4”를 얻게 될테고, 둘 다 이 값을 카운터에 다시 써넣을 겁니다. 카운터의 새로운 값은 “5”가 되었어야 하므로, 두개의 값 증가 오퍼레이션 중 하나는 없어진 셈입니다. 따라서, 카운터의 값 증가에도 값 감소에도 어토믹 오퍼레이션이 사용되어야만 합니다.

레퍼런스 릴리즈가 락킹이나 RCU로 보호된다면, 메모리 배리어는 필요하지 않을테지만, 다른 이유로 인해서입니다. 락킹의 경우, 락은 모든 필요한 메모리 배리어들을 제공하고 (그리고 컴파일러 최적화를 불능화 시킵니다), 락들은 또한 두개의 레퍼런스 릴리즈가 동시에 수행되는 것을 막습니다. RCU의 경우, 현재 수행중인 RCU read-side 크리티컬 섹션들이 모두 완료되기 전까지는 정리 작업이 유예되어야만 하고, 모든 필요한 메모리 배리어들이나 컴파일러 최적화의 불능화가 RCU 기능으로 제공될 겁니다. 따라서, 두개의 CPU들이 두개의 레퍼런스를 동시에 릴리즈 시킨다면, 실제 정리작업은 두개의 CPU들이 모두 RCU read-side 크리티컬 섹션들을 빠져나갈 때까지 유예될 겁니다.

Quick Quiz 13.2: 한 CPU가 마지막 레퍼런스를 해제한 직후에 다른 CPU가 레퍼런스를 획득하는 경우에 대해서도 보호를 해줘야 하지 않나요? ■

kref 구조체 자체는 하나의 어토믹 데이터 아이템으로 구성되며, Figure 13.2의 line 1-3에 보여져 있습니다.

Line 5-8의 `kref_init()` 함수는 이 카운터를 값 “1”로 초기화 시킵니다. `atomic_set()` 기능은 단순한 값 할당으로, 그 이름은 `atomic_t`의 데이터 타입에서 왔지, 그 동작에서 온것이 아님을 알아두시기 바랍니다. `kref_init()` 함수는 오브젝트 생성 때에, 해당 오브젝트가 어떤 다른 CPU 들에 의해 접근될 수 있게 되기 전에 실행되어야만 합니다.

Line 10-14의 `kref_get()` 함수는 무조건적으로 카운터를 어토믹하게 증가시킵니다. `atomic_inc()` 기능은 모든 플랫폼에서 컴파일러 최적화를 명시적으로 불능화 시켜야만 하지는 않습니다만, `kref` 이 별개의 모듈에 존재하고 리눅스 커널 빌드 프로세스는 모듈간 최적화를 하지 않는다는 사실이 똑같은 효과를 냅니다.

Line 16-28의 `kref_sub()` 함수는 어토믹하게 카운터를 감소시키고, 만약 그 결과가 0이라면, line 24에서 명시된 `release()` 함수를 호출하고 line 25에서 호출자에게 `release()` 가 호출되었음을 알리면서 리턴합니다. 그렇지 않다면, `kref_sub()` 은 0을 리턴해서 호출자에게 `release()` 가 호출되지 않았음을 알립니다.

Quick Quiz 13.3: Figure 13.2의 line 22에서 `atomic_sub_and_test()` 가 호출된 직후에, 어떤 다른 CPU 가 `kref_get()` 을 호출했다고 생각해 봅시다. 이는 이 다른 CPU 가 이제 비합법적으로 해제된 오브젝트로의 레퍼런스를 갖게 된 거 아닌가요? ■

Quick Quiz 13.4: `kref_sub()` 가 0을 리턴해서 `release()` 함수가 호출되지 않았음을 알렸다고 생각해 봅시다. 어떤 조건에서 호출자는 이 오브젝트의 존재의 지속에 의존할 수 있을까요? ■

Quick Quiz 13.5: 왜 그냥 해제 함수로 `kfree()` 를 넘기지 않는거죠? ■

13.2.1.3 Atomic Counting With Release Memory Barrier

이런 스타일의 레퍼런스는 리눅스 커널의 네트워킹 레이어에서 패킷 라우팅에 사용되는 목적지 캐시를 추적하는데에 사용됩니다. 실제 구현은 훨씬 더 관련되어 있습니다; 이 섹션은 이 사용예에 적합하며 Figure 13.3에 보인 `struct dst_entry` 레퍼런스 카운트 핸들링 부분에 집중하고 있습니다.

`dst_clone()` 함수는 호출자가 이미 명시된 `dst_entry`에 대한 레퍼런스를 가지고 있을 때 사용될 수 있는데, 이는 커널 내의 다른 존재에게 넘겨줄 수도 있는 또 다른 레퍼런스를 획득하는 경우입니다. 호출자에 의해 이미 레퍼런스가 하나 잡혀 있기 때문에, `dst_clone()` 은 어떤 메모리 배리어도 실행할 필요가 없습니다. 어떤 다른 존재에게 `dst_entry` 를 넘겨주는 행위는 메모리 배리어를 필요로 할 수도, 필요로 하지 않을 수도 있습니다만, 그런 메모리 배리어가 필요하다면, 그

```

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 13.3: Linux Kernel dst_clone API

메모리 배리어는 `dst_entry` 를 넘기는 메커니즘 내에 내장되어야 합니다.

`dst_release()` 함수는 어떤 환경에서도 호출될 수 있고, 호출자는 `dst_release()` 호출에 앞서 `dst_entry` 구조체의 원소들을 직접 참조할 수도 있습니다. 따라서 `dst_release()` 함수는 line 14에서 메모리 배리어를 포함해서 컴파일러나 CPU 가 접근 순서를 잘못 재배치하지 않도록 합니다.

`dst_clone()` 과 `dst_release()` 를 사용하는 프로그래머는 메모리 배리어에 대해서 신경쓸 필요가 없고, 단지 이 두개의 함수들의 사용 규칙만을 알면 된다는 점을 알아 두시기 바랍니다.

13.2.1.4 Atomic Counting With Check and Release Memory Barrier

호출자가 현재는 레퍼런스를 가지고 있지 않은 오브젝트로의 레퍼런스를 획득해야만 하는 경우를 생각해 봅시다. 최초의 레퍼런스 카운트 획득은 레퍼런스 카운트 해제와 동시에 이뤄질 수 있다는 사실이 복잡도를 더욱 증가시킵니다. 레퍼런스 카운트 해제가 레퍼런스 카운트의 해제 후 값이 0이 되게 됨을 발견했고 그 레퍼런스에 연관된 오브젝트가 해제되어도 안전하다는 신호를 날렸다고 생각해 봅시다. 그런 해제 작업이 개시된 후에 레퍼런스 카운트 획득을 허가할 수 없는 것은 분명하므로, 레퍼런스 획득은 레퍼런스 카운트가 0인지에 대한 검사를 포함해야만 합니다. 이런 검사는 다음에 보인 것과 같이 어토믹한 값 증가 오퍼레이션이 한 부분이 되어야만 합니다.

Quick Quiz 13.6: 레퍼런스 카운트의 값이 0인지에 대한 검사는 왜 간단히 어토믹 값 증가 오퍼레이션을 갖는 “if” 문의 “then” 절에 들어갈 수 없는거죠? ■

리눅스 커널의 `fget()` 과 `fput()` 기능들이 이런 스타일의 레퍼런스 카운팅을 사용합니다. 이 함수들의 간료화된 버전이 Figure 13.4에 보여져 있습니다.

```

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 13.4: Linux Kernel fget/fput API

fget()의 Line 4는, 다른 프로세스들과 공유될 수도 있는 현재 프로세스의 file-descriptor 테이블을 가져옵니다. Line 6는 RCU read-side 크리티컬 섹션을 들어가는 rCU_read_lock()을 호출합니다. 뒤따르는 call_rcu()에 전달되는 콜백 함수의 수행은 매칭되는 rCU_read_unlock()이 호출될 때까지 (이 경우 line 10 또는 14) 연기됩니다. Line 7은 fd 인자에 명시된 파일 디스크립터에 연관되는 file 구조체를 탐색하는데, 이에 대해서는 뒤에서 설명하겠습니다. 명시된 파일 디스크립터에 연관된 열린 파일이 존재한다면 line 9에서는 어토믹하게 레퍼런스 카운트를 획득하려 시도합니다. 만약 그렇게 하는데 실패한다면, line 10-11은 RCU read-side 크리티컬 섹션을 빠져나오고 실패했음을 알립니다. 그렇지 않고 시도가 성공했다면, line 14-15는 이 read-side 크리티컬 섹션을 빠져나오고 해당 file 구조체로의 포인터를 리턴합니다.

fcheck_files() 기능은 fget()을 위한 도움을 주는 함수입니다. 해당 함수는 나중의 디레퍼런싱을 위해 (이는 데이터 종속성이 메모리 순서 규칙을 강제하지 않는 DEC Alpha 와 같은 CPU 들에서는 메모리 배리어를 실행합니다) 안전하게 RCU로 보호되는 포인터를 가져오기 위해 사용됩니다. Line 22는 이 태스크의 현재 file-descriptor 테이블로의 포인터를 가져오기 위해 rCU_dereference()를 사용하고, line 24는 명시된 파일 디스크립터가 범위 안에 있는지 체크합니다. 만약 그렇다면 line 25는 해당 file 구조체로의 포인터를 가져오는데, 여기서도 rCU_dereference() 기능이 사용됩니다. 그리고나서 Line 26은 성공했다면 해당 file 구조체로의 포인터를 리턴하고, 실패했다면 NULL을 리턴하게 됩니다.

fput() 기능은 file 구조체로의 레퍼런스를 해제합니다. Line 31은 어토믹하게 레퍼런스 카운트를 감소시키고, 만약 그 감소 결과가 0이라면 line 32에서 해당 file 구조체를 모든 현재 수행중인 RCU read-side 크리티컬 섹션들이 완료된 후에 (call_rcu()의 두번째 인자로 전달되는 file_free_rcu()를 통해) 해제하기 위해 call_rcu() 기능을 실행시킵니다. 모든 현재 수행중인 RCU read-side 크리티컬 섹션들이 완료되기까지 필요한 시간은 “grace period”라고 불립니다. atomic_dec_and_test() 기능은 메모리 배리어를 포함하고 있음을 알아두시기 바랍니다. 이 메모리 배리어는 이 예제에서는 필요치 않은데, 이 구조체는 RCU read-side 크리티컬 섹션이 완료되기 전까지는 해제될 수 없기 때문입니다만 리눅스에서는 그 결과를 리턴하는 모든 어토믹 오퍼레이션들은 정의에 의해 메모리 배리어를 포함해야만 합니다.

일단 grace period가 완료되면, file_free_rcu() 함수가 line 39에서 file 구조체로의 포인터를 가져오고 line 40에서 해제시킵니다.

이런 방법은 리눅스의 가상 메모리 시스템에서도 사용되는데, 페이지 구조체를 위해선 `get_page_unless_zero()` 와 `put_page_testzero()` 를, 메모리 맴 구조체를 위해서는 `try_to_unuse()` 와 `mmput()` 을 참고하시기 바랍니다.

13.2.2 Linux Primitives Supporting Reference Counting

앞의 예제들에서 사용된 리눅스 커널의 기능들이 다음의 리스트에 요약되어 있습니다.

- `atomic_t` 어토믹하게 조정되는 32-bit 크기의 것을 위한 타입 정의.
- `void atomic_dec(atomic_t *var);` 메모리 배리어를 요청하거나 컴파일러 최적화를 불능화 시키지 않으면 참조된 변수의 값을 어토믹하게 감소.
- `int atomic_dec_and_test(atomic_t *var);` 어토믹하게 참조된 변수의 값을 감소시키고 그 감소 결과값이 0이라면 `true` (0 이 아닌 값) 을 리턴. 이 기능의 앞뒤로 메모리 참조가 옮겨가지 못하도록 메모리 배리어와 컴파일러 최적화 불능화를 수행.
- `void atomic_inc(atomic_t *var);` 메모리 배리어를 요청하거나 컴파일러 최적화를 불능화 시키지 않으면 참조된 변수의 값을 어토믹하게 증가.
- `int atomic_inc_not_zero(atomic_t *var);` 참조된 변수의 값이 0이 아니라면 그 값을 어토믹하게 증가시키고 값 증가가 수행되었다면 `true` (0이 아닌 값) 을 리턴함. 이 기능의 앞뒤로 메모리 참조가 옮겨가지 못하도록 메모리 배리어와 컴파일러 최적화 불능화를 수행.
- `int atomic_read(atomic_t *var);` 참조된 변수의 정수 값을 리턴. 이는 어토믹 오퍼레이션이어야 할 필요가 없고, 메모리 배리어 명령어를 요청할 필요도 없음. “어토믹한 읽기” 라고 생각하기보다는 “어토믹 변수로부터의 평범한 읽기” 라고 생각할 것.
- `void atomic_set(atomic_t *var, int val);` 참조된 어토믹 변수의 값을 “`val`” 로 설정. 이는 어토믹 오퍼레이션이어야 할 필요가 없으며, 메모리 배리어나 컴파일러 최적화의 불능화를 수행할 필요도 없음. 이를 “어토믹한 값 설정”으로 생각하기보다는 “어토믹 변수에의 평범한 값 설정”으로 생각할 것.

- `void call_rcu(struct rcu_head *head, void (*func)(struct rcu_head *head));` 일부 시간이 지나서 현재 수행중인 RCU read-side 크리티컬 섹션들이 완료된 후에 `func(head)` 를 수행하되 `call_rcu()` 기능 자체는 즉시 리턴함. `head` 는 일반적으로 RCU로 보호되는 데이터 구조체이고, `func` 는 일반적으로 이 데이터 구조체를 메모리에서 해제시키는 함수임을 알아둘 것. `call_rcu()` 의 호출과 `func` 호출 사이의 시간 간격은 “grace period” 라 불리움. Grace period 를 포함하는 모든 시간 간격은 그 자체로 grace period 임.
- `type *container_of(p, type, f);` 명시된 타입의 구조체의 필드 `f` 로의 포인터 `p` 를 받고 해당 구조체로의 포인터를 리턴.
- `void rcu_read_lock(void);` RCU read-side 크리티컬 섹션의 시작을 표시.
- `void rcu_read_unlock(void);` RCU read-side 크리티컬 섹션의 종료를 표시. RCU read-side 크리티컬 섹션은 중첩될 수 있음.
- `void smp_mb__before_atomic_dec(void);` 해당 플랫폼의 `atomic_dec()` 기능이 그러지 않는다면 코드를 움직일 수 있는 컴파일러 최적화의 불능화와 메모리 배리어를 요청.
- `struct rcu_head` Grace period 를 기다리는 오브젝트들을 추적하기 위해 RCU 설비에서 사용되는 데이터 구조체. 일반적으로 RCU로 보호되는 데이터 구조체 내에 하나의 필드로 포함되어 있음.

Quick Quiz 13.7: 어토믹하지 않은 `atomic_read()` 와 `atomic_set()` 이라구요? 이건 또 무슨 농담이죠???

13.2.3 Counter Optimizations

카운터 증가와 감소가 흔하게 일어나지만 카운터 값이 0인지에 대한 검사는 가끔만 이루어지는 일부 경우들에 있어서는, Chapter 5 에서 이야기한 것처럼 per-CPU 또는 per-task 카운터들을 두는게 말이 될겁니다. 이런 테크닉이 RCU에 적용된 예를 보기 위해 sleepable read-copy update (SRCU)에 대한 논문 [McK06b] 을 참고하세요. 이 방법은 카운터 값 증가와 감소 기능들에서의 어토믹 명령어나 메모리 배리어의 사용의 필요를 없애 줍니다만, 코드의 위치를 움직이는 컴파일러 최적화의

제거는 여전히 필요합니다. 또한, 합산된 레퍼런스 카운트가 0이 되었는지를 체크하게 되는 `synchronize_srcu()` 와 같은 기능들은 상당히 느려집니다. 이는 이런 테크닉들은 레퍼런스들이 빈번히 획득되고 해제되지만 레퍼런스 카운트가 0이 되었는지에 대한 검사는 가끔만 일어나는 상황을 위해서 설계된 것이란 점을 강조합니다.

하지만, 레퍼런스 카운트가 사용되지 않았다면 읽기 전용이었을 수 있는 데이터 구조에 레퍼런스 카운트의 사용은 (대부분의 경우 어토믹한) 쓰기를 필요로 한다는 점이 문제인 경우가 많습니다. 이런 경우, 레퍼런스 카운트는 읽기를 하는 쓰레드들에게 비싼 캐시 미스를 부과합니다.

따라서 읽기를 하는 쓰레드들이 횡단하게 되는 데이터 구조체에 쓰기를 하지 않아도 되게 하는 동기화 메커니즘들을 찾아볼 가치가 있습니다. 그런 것들 중 하나는 Section 9.3에서 다룬 해저드 포인터이고, 또 다른 하나는 Section 9.5에서 다룬 RCU입니다.

13.3 RCU Rescues

이 섹션은 이 책의 앞부분에서 이야기한 몇 가지 예제들에 RCU를 어떻게 적용하는지를 보입니다. 일부 경우들에 있어서는 RCU는 간단한 코드를 제공하고, 어떤 경우에는 더 나은 성능과 확장성을 제공하며, 또 다른 경우에는 두 가지를 모두 제공합니다.

13.3.1 RCU and Per-Thread-Variable-Based Statistical Counters

Section 5.2.4는 대략적으로 평범한 값 증가 연산 (C++ 오퍼레이터)과 같은—하지만 `inc_count()`를 통해서만 값을 증가시키는—훌륭한 성능과 선형적 확장성을 보이는 통계적 카운터들의 구현을 설명했습니다. 불행히도, `read_count()`를 통해 값을 읽어야 하는 쓰레드들은 글로벌 락을 잡아야만 했고, 따라서 높은 오버헤드를 일으키고 낮은 확장성으로 고통받아야 했습니다. 락 기반의 구현 코드는 Page 45의 Figure 5.9에 보여져 있습니다.

Quick Quiz 13.8: 대체 왜 그런 글로벌 락이 필요했던 거지요? ■

13.3.1.1 Design

원하는건 `inc_count()` 만이 아니라 `read_count()`에서도 훌륭한 성능과 확장성을 얻기 위해 `read_count()`의 쓰레드 횡단을 보호하는 데에 `final_mutex` 대신에 RCU를 사용하는 것입니다. 하지만, 계산된 합계의 정확성을 포기하지도 않고 싶습니다. 자세

히 말하자면, 특정 스레드가 종료될 때에, 우린 종료되는 쓰레드의 카운트를 잊어버릴 수도, 그걸 두번씩 세서도 안됩니다. 그런 에러는 결과의 전체 정확성과 동일한 정도의 비정확성을 초래할 수 있는데, 달리 말하자면 그런 에러는 결과값이 완전히 쓸모없게 만들 수 있습니다. 그리고 사실, `final_mutex`의 목적들 중 하나는 쓰레드들이 `read_count()`의 실행 사이에 들어왔다 나갔다 하지 않음을 분명히 하는 것입니다.

Quick Quiz 13.9: 어쨌든, `read_count()`의 정확성을 대체 뭔가요? ■

따라서, 우리가 `final_mutex`를 없애려 한다면, 우리는 일관성을 보장하기 위한 어떤 다른 방법을 사용해야 합니다. 한가지 방법은 앞서 종료된 쓰레드들 전체를 위한 전체 카운트와 쓰레드별 카운터로의 포인터들의 배열을 하나의 구조체에 넣는 것입니다. `read_count()`에 의해 접근될 수 있는 그런 구조체는 상수가 되므로, `read_count()`가 일관적인 데이터를 보게 될 것을 보장합니다.

13.3.1.2 Implementation

Figure 13.5의 line 1-4는 `countarray` 구조체를 보이고 있는데, 이 구조체는 앞서 종료된 쓰레드들의 카운트를 위한 `->total` 필드와 현재 돌아가고 있는 각각의 쓰레드를 위한 `per-thread counter`로의 포인터들의 배열인 `counterp[]`를 갖습니다. 이 구조체는 `read_count()`의 한 수행이 수행중인 쓰레드의 알려진 집합과 일관적인 전체값을 볼 수 있도록 합니다.

Line 6-8은 `per-thread counter` 변수, 현재의 `countarray` 구조체를 가리키는 `countarrayp` 글로벌 포인터, 그리고 `final_mutex` 스판락의 정의를 담고 있습니다.

Line 10-13은 Figure 5.9로부터 달라지지 않은 `inc_count()`를 보입니다.

Line 15-29는 상당히 많이 바뀐 `read_count()`를 보입니다. Line 21과 27은 `rcu_read_lock()`과 `rcu_read_unlock()`으로 `final_mutex`의 획득과 해제를 대신합니다. Line 22는 현재의 `countarray` 구조체를 로컬 변수 `cap`으로 스냅샷을 뜨기 위해 `rcu_dereference()`를 사용합니다. RCU가 올바르게 사용된다면 이 `countarray` 구조체가 적어도 line 27에서의 현재 RCU read-side 크리티컬 섹션의 종료까지는 유지될 것이 보장될 겁니다. Line 23은 `sum`을 `cap->total`로 초기화 시키는데, 이는 앞서 종료된 쓰레드들의 카운트의 합입니다. Line 24-26은 현재 수행중인 쓰레드들과 연관되어 있는 `per-thread` 카운터들을 합하고, 마지막으로 line 28은 그 합을 리턴합니다.

`countarrayp`의 초기값은 line 31-39의 `count_init()`에 의해 주어집니다. 이 함수는 첫번째 쓰레드가 생성되기 전에 수행되는데, 이 함수의 일은 초

```

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 13.5: RCU and Per-Thread Statistical Counters

기의 구조체를 할당하고 그 값을 0으로 세팅한 후에 countarrayp를 할당하는 것입니다.

Line 41-48은 count_register_thread() 함수를 보아는데, 이 함수는 각 쓰레드가 새로이 생성될 때마다 호출됩니다. Line 43은 현재 쓰레드의 인덱스를 가져오고, line 45는 final_mutex를 획득하며, line 46은 이 쓰레드의 counter로의 포인터를 설치하며, line 47에서 final_mutex를 놓습니다.

Quick Quiz 13.10: 이봐요!!! Figure 13.5의 line 46은 앞에 존재한 countarray 구조체의 값을 수정하잖아요! 이 구조체는 일단 한번 read_count()에 접근 가능하게 되면 상수로 남게 된다고 하지 않았어요???

Line 50-70은 각 쓰레드가 종료하기 직전에 실행되는 count_unregister_thread()를 보입니다. Line 56-60은 새로운 countarray 구조체를 할당하고, line 61에서 final_mutex를 획득하며 line 67에서 이를 해제합니다. Line 62는 현재 countarray의 내용을 새로 할당된 버전에 복사하고, line 63은 종료되는 쓰레드의 counter를 새로운 구조체의 ->total에 더하고, line 66은 countarray 구조체의 새로운 버전을 설치하는데에 rCU_assign_pointer()를 사용합니다. Line 68은 동시적으로 read_count()를 수행하고 있을 수 있는, 따라서 기존의 countarray 구조체로의 레퍼런스를 가지고 있을 수 있는 모든 쓰레드가 이들의 RCU read-side 크리티컬 섹션을 종료해서 그런 레퍼런스를 모두 버려버리도록 grace period가 하나 지나가길 기다립니다. 그리고 나서 line 69는 이제 기존의 countarray 구조체를 안전하게 해제시킬 수 있습니다.

13.3.1.3 Discussion

Quick Quiz 13.11: 우와! Figure 5.9는 라인수가 42밖에 되지 않는데 반해 Figure 13.5는 69라인이나 되는군요. 이 추가적인 복잡도가 정말로 가치가 있는 건가요?

■

RCU의 사용은 종료되는 쓰레드가 다른 쓰레드들이 이 종료되는 쓰레드들의 __thread 변수들을 사용하는 것을 마칠 때까지 기다릴 수 있도록 해줍니다. 이는 read_count() 함수가 락킹을 필요 없게 해서 inc_count()와 read_count() 함수 둘 다에 훌륭한 성능과 확장성을 제공합니다. 하지만, 이 성능과 확장성은 약간의 코드 복잡도의 증가를 비용으로 지불합니다. 컴파일러와 라이브러리를 작성하는 사람들이 안전한 쓰레드간 __thread 변수들로의 접근을 제공하기 위해 유저 레벨 RCU [Des09]를 제공해서 __thread 변수들의 사용자간에 보여지게 되는 복잡도를 줄일 수 있게 됨다면 좋을 겁니다.

13.3.2 RCU and Counters for Removable I/O Devices

Section 5.5 는 제거 가능한 디바이스들로의 I/O 액세스의 카운팅을 다루는 한쌍의 코드 조각을 보였습니다. 이 코드 조각들은 reader-writer 락을 잡아야 하는 필요로 인해 빠른 수행 경로 (I/O 를 시작하는 것) 에서의 높은 오버헤드로 힘들어했습니다.

이 섹션은 RCU 가 이 오버헤드를 어떻게 없앨 수 있는지 보입니다.

I/O 를 수행하는 코드는 원래의 것과 상당히 유사한데, RCU read-side 크리티컬 섹션이 원래 것의 reader-writer 락의 read-side 크리티컬 섹션을 대체합니다:

```

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 }
```

RCU read-side 기능들은 최소한의 오버헤드만을 가지고 있으므로, 원했던대로 빠른 수행 경로의 속도를 높입니다.

디바이스 제거 부분의 업데이트된 코드 조각은 다음과 같습니다:

```

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();
```

여기서 우린 reader-writer 락을 배타적 스피너로 바꾸고 모든 RCU read-side 크리티컬 섹션들이 완료되길 기다리기 위해 synchronize_rcu() 를 추가했습니다. synchronize_rcu() 때문에, 일단 우리가 line 6 에 도달한다면, 우린 모든 남아있는 I/O 들이 처리되었음을 알 수 있습니다.

물론, synchronize_rcu() 의 오버헤드는 클 수 있습니다만, 디바이스의 제거는 상당히 드문 일임을 고려한다면 이는 괜찮은 트레이드오프입니다.

```

1 struct foo {
2   int length;
3   char *a;
4 };
```

Figure 13.6: RCU-Protected Variable-Length Array

```

1 struct foo_a {
2   int length;
3   char a[0];
4 };
5
6 struct foo {
7   struct foo_a *fa;
8 };
```

Figure 13.7: Improved RCU-Protected Variable-Length Array

13.3.3 Array and Length

Figure 13.6 에 보인 것과 같이 RCU 로 보호되는 가변길이의 배열을 가지고 있다고 생각해 봅시다. 배열 $\rightarrow a[]$ 의 길이는 동적으로 언제든 변할 수 있고, 그 길이는 $\rightarrow length$ 를 통해 제공됩니다. 물론, 이는 다음과 같은 경주 조건을 만들어냅니다:

1. 배열의 길이가 초기에는 16 character 만큼의 길이이고, 따라서 $\rightarrow length$ 는 16의 값을 갖습니다.
2. CPU 0 가 $\rightarrow length$ 의 값을 읽어와서 16을 얻게 됩니다.
3. CPU 1 이 이 배열의 길이를 8로 줄이게 되고, 메모리의 8-character 블락 구간으로의 포인터를 $\rightarrow a[]$ 에 할당합니다.
4. CPU 0 가 $\rightarrow a[]$ 의 새로운 포인터를 가져오고, 그 12번째 원소에 새로운 값을 저장합니다. 이 배열은 8 개 character 만을 가지고 있으므로, 이는 SEGV 또는 (더 나쁜 경우인) 메모리 오염을 초래하게 됩니다.

이를 어떻게 막을 수 있을까요?

한가지 방법은 Section 14.2 에서 소개된 메모리 배리어를 조심스럽게 사용하는 것입니다. 이는 제대로 동작합니다만, 읽는 쪽의 오버헤드를 초래하고, 아마도 더 나쁠 수도 있게도, 명시적인 메모리 배리어의 사용을 필요로 합니다.

더 나은 전략은 Figure 13.7 에 보인 것과 같이 해당 배열과 값을 같은 구조체에 놓는 것입니다. 이제 새로운 배열 (foo_a 구조체) 를 할당하는 일은 자동으로 배열 공간을 위한 새로운 공간을 제공합니다. 이는 어떤 CPU 가 $\rightarrow fa$ 로의 레퍼런스를 가져가면 $\rightarrow length$ 는 $\rightarrow a[]$ 의 길이와 들어맞을 것임을 보장합니다.

```

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 13.8: Uncorrelated Measurement Fields

1. 배열의 길이가 초기에는 16 character 길이이고, 따라서 \rightarrow length는 16의 값을 갖습니다.
2. CPU 0 이 \rightarrow fa의 값을 읽어오게 되어서 값 16과 16-byte 배열을 담고 있는 구조체를 가져오게 됩니다.
3. CPU 0 가 \rightarrow fa- \rightarrow length의 값을 읽어와서 16이라는 값을 얻게 됩니다.
4. CPU 1 이 배열의 길이를 8로 줄이고, 메모리의 8-character 블록 구간을 포함하는, 새로운 foo_a 구조체로의 포인터를 \rightarrow fa에 할당합니다.
5. CPU 0 이 \rightarrow a[]로부터 새로운 포인터를 가져가고 새로운 값을 12번째 원소에 저장합니다. 하지만 CPU 0 는 여전히 16-byte 배열을 담고 있는 기존의 foo_a 를 레퍼런스하고 있으므로, 모두 문제 없습니다.

물론, 두 경우 모두, CPU 1 은 기존의 배열을 메모리에서 해제하기 전에 하나의 grace period 를 기다려야만 합니다.

이 방법의 더 일반적인 버전이 다음 섹션에서 제공됩니다.

13.3.4 Correlated Fields

각각의 Schrödinger 의 동물들이 Figure 13.8 에 보여진 데이터 원소로 표현된다고 생각해 봅시다. meas_1, meas_2, 그리고 meas_3 필드는 주기적으로 업데이트되는 연관된 측정치의 집합입니다. 읽기 를 하는 쓰레드들은 이 세개의 값을 하나의 측정치 업데이트로부터 본다는 점이 특히 중요합니다: 만약 한 읽기 쓰레드가 meas_1 의 예전 값을 읽지만 meas_2 와 meas_3 의 새로운 값을 읽게 되다면, 그 읽기 쓰레드는 완전히 혼란에 빠질 겁니다. 읽기 쓰레드들이 이 세개의 값들의 통합된 집합을 볼 수 있도록 보장하려면 어떻게 해야 할까요?

한가지 방법은 새로운 animal 구조체를 할당하고, 기존 구조체를 새로운 구조체로 복사하고, 새로운 구조체의 meas_1, meas_2, 그리고 meas_3 필드들을 업데이트 한 후, 기존의 구조체를 새로운 구조체로 포인터를

```

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 13.9: Correlated Measurement Fields

업데이트하는 방법으로 교체하는 것입니다. 이는 모든 읽기 쓰레드들이 측정 값들의 통합된 집합을 볼 수 있을 것을 보장합니다만, 이는 \rightarrow photo[] 필드 때문에 커다란 구조체의 복사를 필요로하게 됩니다. 이 복사 작업은 받아들일 수 없을 만큼 커다란 오버헤드를 일으킬 수 있습니다.

또다른 방법은 하나의 간접적 단계를 추가하는 것으로, Figure 13.9 보여진 것과 같습니다. 새로운 측정이 취해진다면, 새로운 measurement 구조체를 할당하고, 이를 새로운 측정치로 채우고, animal 구조체의 \rightarrow mp 필드가 이 새로운 measurement 구조체의 포인터를 가리키도록 rcu_assign_pointer() 를 사용해서 업데이트 하는 것입니다. 하나의 grace period 가 지나간 후에, 기존의 measurement 구조체는 메모리에서 해제될 수 있습니다.

Quick Quiz 13.12: 하지만 Figure 13.9 에 보인 방법은 추가적인 캐시 미스를 초래할 수 있어서, 추가적인 읽기 쪽의 오버헤드를 초래할 수 있지 않나요? ■

이 방법은 읽기 쓰레드들이 선택된 필드들에 대해 연관된 값을 최소한의 읽기 쪽 오버헤드만을 가지고 볼 수 있도록 해줍니다.

13.4 Hashing Hassles

이 섹션은 해시 테이블을 다룰 때에 나타날 수 있는 몇 가지 문제들을 알아봅니다. 이런 문제들은 많은 다른 탐색을 하는 구조체들에서도 역시 나타날 수 있음을 알아두시기 바랍니다.

13.4.1 Correlated Data Elements

이 상황은 Section 13.3.4 에서의 것과 비슷합니다: 우린 두개 이상의 원소들의 연관된 시야를 필요로 하는 해시 테이블을 가지고 있습니다. 이 원소들은 함께 업데이트 되고, 첫번째 원소의 기존 버전과 함께 다른 원소들의 새로운 버전을 보지는 않고 싶습니다. 예를 들어, Schrödinger 는 그의 in-memory 데이터베이스에 그의

모든 동물들에 가족을 추가시키려 합니다. Schrödinger는 결혼과 이혼이 순간적으로 일어나지는 않음을 알고 있지만, 그는 전통주의자이기도 합니다. 이와 같이, 그는 그의 데이터베이스가 신부는 이제 결혼했지만, 신랑은 그렇지 않은 경우, 또는 그 반대의 경우를 원하지 않습니다. 달리 말하자면, Schrödinger는 그의 데이터베이스에 *wedlock* (역자 주: 결혼에 관련된 락)에 일관적인 데이터 열람이 가능하기를 원합니다.

한 가지 방법은 시퀀스 락 (Section 9.4를 참고하세요)을 사용해서 *wedlock*에 관계된 *update*들이 *write_seqlock()*의 보호 아래 행해지도록 하고, *wedlock* 일관성을 필요로 하는 *read*들은 *read_seqbegin()* / *read_seqretry()* 루프 아래 행해지도록 하는 겁니다. 시퀀스 락들은 RCU 보호의 대체품이 안미을 알아두시기 바랍니다: 시퀀스 락은 동시적인 수정들에 대해 데이터를 보호하지만, 동시적인 삭제들로부터 데이터를 보호하기 위해선 RCU가 여전히 필요합니다.

이 방법은 연관된 원소들의 갯수가 작고, 이 원소들을 읽는 시간이 짧으며, 업데이트 되는 빈도가 적을 때 상당히 잘 동작합니다. 그렇지 않다면, 업데이트들은 읽기 쓰레드들이 결코 완료될 수 없을 정도로 업데이트들이 자주 일어날 수도 있을 겁니다. Schrödiger는 그의 가장 덜 정상적인 동물들이라 해도 이런 문제가 생길 수 있을 정도로 빠르게 결혼과 이혼을 행하지는 않을 것으로 예상한다고 해도, 그는 이 문제가 다른 상황에서도 역시 일어날 수 있음을 깨닫고 있습니다. 이 읽기 쓰레드 *starvation* 문제를 없애는 한 가지 방법은 읽기 쓰레드들이 너무 많은 재시도를 행하게 된다면 *update-side* 기능들을 사용하는 것입니다만, 이는 성능과 확장성을 떨어뜨릴 수 있습니다.

또한, 만약 *update-side* 기능들이 너무 자주 사용된다면, 락 경쟁으로 인해 낮은 성능과 확장성이 초래될 것입니다. 이를 막을 수 있는 한 가지 방법은 *per-element* 시퀀스 락을 두고, 결혼 상태를 업데이트 할 때 두 배우자의 락을 모두 잡도록 하는 것입니다. 읽기 쓰레드들은 두 멤버에 연관된 결혼 상태의 모든 변경사항에 대한 안정적인 시야를 얻기 위해 두 배우자들의 락들에 대해 재시도 루프를 행할 수 있습니다. 이는 높은 결혼과 이혼 빈도에 의한 경쟁을 없앨 수 있습니다만, 데이터베이스에 대한 한번의 스캔 동안의 모든 결혼 상태들에 대한 안정적인 시야를 얻는 과정이 복잡해집니다.

원소 그룹짓기가 결혼 상태가 그렇기를 바라듯이 잘 정의되어 있고 영속적이라면, 데이터 원소에 주어진 그룹의 멤버들로의 링크를 가리키는 포인터를 추가하는 것도 한 가지 방법입니다. 그렇게 되면 읽기 쓰레드들은 첫번째 멤버와 같은 그룹의 모든 데이터 원소들에 접근하기 위해서 이 포인터들을 따라가면 됩니다.

버전 넘버를 사용하는 다른 방법은 흥미있는 독자들을 위한 뒷으로 남겨두겠습니다.

13.4.2 Update-Friendly Hash-Table Traversal

해시 테이블의 모든 원소들에 대한 통계적 스캐닝이 필요하다고 생각해 봅시다. 예를 들어, Schrödinger는 모든 동물들에 대해 평균적인 신장 대비 체중 비율을 구하고 싶어할 수 있습니다.² 더 나아가서 Schrödinger가 이 통계를 위한 스캐닝이 진행되는 사이에 이 해시 테이블에 추가되고 삭제된 동물들로 인한 작은 에러정도는 무시하고자 한다고 생각해 봅시다. Schrödinger는 동시성을 제어하기 위해 무엇을 해야 할까요?

한 가지 방법은 통계를 위한 스캐닝을 RCU read-side 크리티컬 섹션으로 감싸는 것입니다. 이는 업데이트들이 스캐닝에 과도하게 훼방을 놓지 않으면서 동시적으로 진행되는 것을 가능하게 합니다. 특히, 이 스캐닝은 업데이트를 가로막지 않고 그 반대 역시 마찬가지여서, 설령 매우 높은 업데이트 빈도가 있다 해도 매우 많은 양의 원소들을 담고 있는 해시 테이블들에 대한 스캐닝을 우아하게 지원합니다.

Quick Quiz 13.13: 하지만 이 스캐닝은 크기 재조정이 되고 있는 크기 재조정 가능한 해시 테이블에 대해서는 어떻게 동작하나요? 그런 경우, 기존의 해시테이블도 새로운 해시테이블도 해당 해시 테이블에 모든 원소들을 담고 있다고 보장될 수 없는데요! ■

² 왜 그런 숫자가 필요하죠? 절 때리세요! 하지만 일반적인 그룹 통계는 유용한 경우가 많습니다.

Chapter 14

If a little knowledge is a dangerous thing, just imagine all the havoc you could wreak with a lot of knowledge!

Unknown

Advanced Synchronization

이 섹션에서는 더 약하지만, 비용이 적은 동기화 기능들에 대해 이야기 해 봅니다. 이 약함은 상당히 도움이 되는데, 실제로 어떤 사람들은 약함이 덕목이라 이야기 [Alg13] 하기도 했습니다. 삶의 다른 많은 분야들에서도 그러하듯이, 병렬 프로그래밍에서도 약함이 만병통치약은 아닙니다. 예를 들어, 아직 정리 안된 약화를 생각하기도 전에 Chapter 5 의 마지막에서 이야기 했듯, 철저하게 파티셔닝, 배칭, 그리고 잘 테스트된 약한 API 들을 적용해야 합니다 (Chapter 8 과 Chapter 9 을 참고하세요).

하지만 그들을 모두 적용한 후라면, 이 챕터에서 이야기할 고급 테크닉들을 필요로 할 수도 있을 겁니다. 그러기 위해, Section 14.1 에서 락을 피하기 위해 사용되는 테크닉들을 요약하고, Section 14.2 에서 메모리 배리어를 다룬 후, 마지막으로 Section 14.3 에서 블로킹하지 않는 동기화를 짧게 다룹니다.

14.1 Avoiding Locks

락킹은 제품화 레벨의 병렬성에서 매우 많이 사용되는 방법이지만, 많은 상황에서 락을 사용하지 않는 (lockless) 테크닉을 사용하는 것으로 성능, 확장성, 그리고 실시간 반응성을 모두 크게 개선시킬 수 있습니다. 그런 lockless 테크닉의 한 예는 Section 5.2 에서 보인, 카운터 증가에 락은 물론 어토믹 오퍼레이션, 메모리 배리어, 그리고 심지어 캐시 미스 까지 없었던 통계적 카운터가 될 수 있을 것입니다. 그 외에 우리가 다뤘던 예들은 다음과 같습니다:

1. Chapter 5 에서 다룬 다른 카운팅 알고리즘들의 빠른 수행 경로들.
2. Section 6.4.3 의 리소스 얼로케이터 캐시들의 빠른 수행 경로.
3. Section 6.5 의 미로 풀기 알고리즘.

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

4. Chapter 8 에서 설명한 데이터 소유권 (Data Ownership) 테크닉.
5. Chapter 9 에서 설명한 레퍼런스 카운팅과 RCU 테크닉들.
6. Chapter 10 에서 설명한 루프 코드 경로.
7. Chapter 13 에서 설명한 많은 테크닉들.

한마디로, lockless 테크닉들은 상당히 유용하고 많이 사용되고 있습니다.

하지만, lockless 테크닉들은 `inc_count()`, `memblock_alloc()`, `rcu_read_lock()` 등과 같은 잘 정의된 API 뒤에 숨겨져 있는게 제일 좋습니다. 과한 lockless 테크닉들의 사용은 어려운 버그를 만들어내기 쉽기 때문입니다.

많은 lockless 테크닉들의 핵심 요소는 다음 섹션에서 설명할 메모리 배리어입니다.

14.2 Memory Barriers

인과성과 순서는 매우 직관적이고, 해커들은 이에 대해 일반적인 사람들보다 훨씬 깊은 이해를 갖고 있는 경향이 있습니다. 이 직관들은 순차적 코드이든 락킹과 RCU 같은 표준적 상호 배타 메커니즘을 사용하는 병렬 코드이든 코드를 작성하고 분석하고 디버깅하는데 매우 강력한 도구가 됩니다.

불행히도, 이런 직관들은 공유메모리 안의 데이터 구조들을 위해 명시적 메모리 배리어를 직접 사용하는 코드에서는 완전히 어긋납니다. 예를 들어, Table 14.1의 리트머스 테스트는 해당 단정문이 결코 실패하지 않을 거라 보장되는 듯 보입니다. 일단, $r1 \neq 1$ 이라면, 우린 쓰레드 1의 y 로부터의 로드가 쓰레드 2의 y 로의 스토어 이전에 일어났다고 볼 것이고, 따라서 쓰레드 2의 x 로부터의 로드는 쓰레드 1의 x 로의 스토어 이후에 일어났고, 따라서 해당 단정문에서 필요로 하는 $r2 == 1$ 조건이 참일 것이라 생각될 수 있습니다. 예제는 대칭적이므로, 같은 원리로 $r2 \neq 1$ 은 $r1 == 1$ 을 보장한다고 생각하게 될 것입니다. 안타깝지만, Table 14.1의 메모리 배리어 부재로 인해 이런 생각은 깨져버립니다. 컴파일러도 CPU 도 쓰레드 1과 쓰레드 2의 문장들을 재배열 할 권리가 있으며, 이는 비교적 강한 순서 규칙을 제공하는 x86 같은 시스템에서도 마찬가지입니다.

다음의 섹션들을 통해 정확히 어디서 이 직관들이 깨지는지 알아보고, 이런 문제들을 방지하도록 도움을 줄 수 있는 메모리 배리어의 개념적 모델을 알아봅니다.

Section 14.2.1 는 메모리 접근 순서와 메모리 배리어에 대한 짧은 개론을 제공합니다. 일단 이 배경 지식을 알게 된다면, 다음은 당신의 직관에 문제가 있었음을 인정할 차례입니다. 이 고통스러운 일은 Section 14.2.2에서 직관적으로는 올바른 것처럼 보이지만 실제 하드웨어에서는 비참하게 실패하고 마는 코드 조각들을 보이고 Section 14.2.3에서 여러 값들을 같은 시점에 가질 수 있는 스칼라 변수를 보이는 코드를 제공함으로써 처리합니다. 일단 이 슬픈 과정을 통해 직관을 똑바로 세운 후에는, Section 14.2.4 가 메모리 배리어가 따르는, 우리가 그 위에서부터 일을 시작해야 할 기본적 규칙을 제공합니다. 이런 규칙들은 Section 14.2.5부터 Section 14.2.14에서 좀 더 다듬어 집니다.

14.2.1 Memory Ordering and Memory Barriers

그런데 왜 메모리 배리어들이 거기 있어야만 하는 걸까요? CPU 들이 알아서 순서를 맞출 수 없나요? 그게 우리가 컴퓨터가 알아서 일을 하라고 최일선에 두는 이유 아닌가요?

많은 사람들이 실제로 컴퓨터들이 알아서 일을 할 거라고 예측합니다만, 또한 많은 사람들이 일을 빨리 해야 한다고 주장합니다. 근래의 컴퓨터 시스템 제조사들이 직면한 어려움 가운데 하나는 메인 메모리는 CPU 의 속도를 따라갈 수 없다는 것입니다 – 근래의 CPU 들은 메모리에서 변수 하나를 가져오는데 필요한 시간동안 수백개의 인스트럭션들을 실행할 수 있습니다. 따라서 CPU 들은 Figure 14.1 에 그려진 것처럼, 계속해서 거대한 캐시들을 사용합니다. 특정 CPU 에 의해 매우 자주

사용되는 변수들은 해당 CPU 의 캐시에 남아있는 경향을 보일 것이고, 이로 인해 해당 데이터에 대한 고속의 액세스가 가능해집니다.

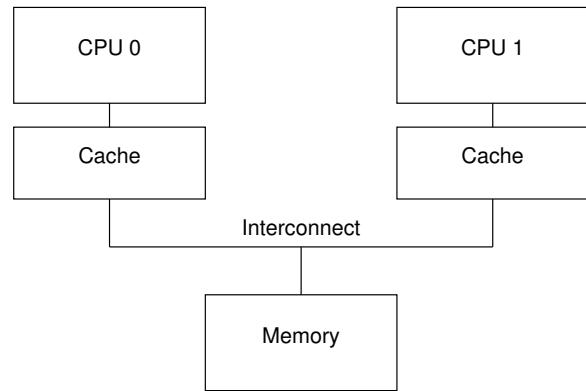


Figure 14.1: Modern Computer System Cache Structure

한 CPU 가 자신의 캐시에 아직 존재하지 않는 데이터에 접근하는 경우엔, 불행히도 데이터를 메모리로부터 가져와야 하는, 비싼 “캐시 미스”를 초래하게 됩니다. 더 안타깝게도, 일반적인 코드의 수행은 상당한 캐시 미스를 초래합니다. 이로 인한 성능 하락을 제한하기 위해, CPU 들은 메모리 참조가 캐시 미스로 인해 메모리에서 데이터를 얻어오는 것을 기다리는 동안 다른 인스트럭션들을 실행할 수 있도록 설계되었습니다. 이는 분명히 인스트럭션들과 메모리 참조가 비순차적으로 실행되도록 해서 Figure 14.2 에 보여지는 것과 같이 상당한 혼란을 가져올 수 있습니다. 컴파일러들과 (락킹과 RCU 같은) 동기화 도구들은 “메모리 배리어들”(예를 들어, 리눅스 커널의 `smp_mb()`)을 이용해 순서가 맞춰지는 듯한 환상을 유지할 책임을 갖습니다. 이런 메모리 배리어들은 ARM, POWER, Itanium, 그리고 Alpha 에서와 같이 명시적인 인스트럭션이 될 수도 있고, 또는 x86 에서처럼 다른 인스트럭션들에 의해 묵시적으로 실행될 수도 있습니다.

표준 동기화 도구들은 순서의 환상을 지키므로, 이 섹션을 읽는 것을 멈추고 그냥 그 도구들을 사용하는 것이 좋을 것입니다.

하지만, 동기화 도구들 자체를 구현해야 한다면, 또는, 그저 메모리 접근 순서와 메모리 배리어들이 어떻게 동작하는지에 흥미가 있다면, 계속 읽으세요!

다음 섹션은 당신이 명시적으로 메모리 배리어를 사용한다면 마주할 수 있는, 직관에 반하는 시나리오들을 이야기 합니다.

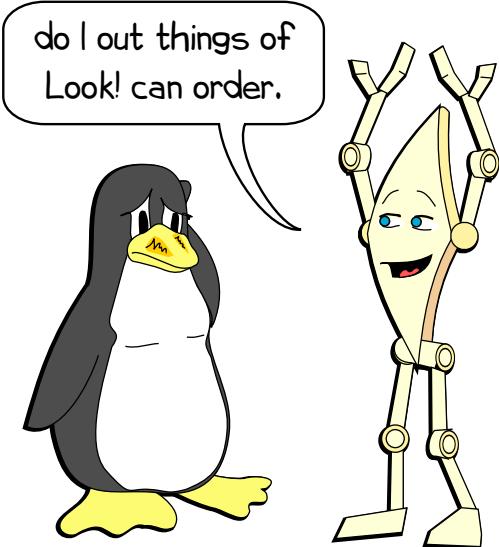


Figure 14.2: CPUs Can Do Things Out of Order

14.2.2 If B Follows A, and C Follows B, Why Doesn't C Follow A?

메모리 순서 규칙과 메모리 배리어는 매우 비직관적일 수 있습니다. 예를 들어, Figure 14.3의 함수들이 변수 A, B, C가 초기값 0을 가진 채 병렬로 수행되는 경우를 생각해보세요:

```

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

직관적으로 볼 때, `thread0()` 는 B의 값 할당을 A에의 값 할당 후에 하고, `thread1()` 은 C에 값을 할당

하기 전, `thread0()` 가 B에 값을 할당할 때까지 기다리며, `thread2()` 는 A의 값을 보기 전에 `thread1()` 이 C에 값을 할당하기까지 기다립니다. 따라서, 역시 직관적으로, line 21의 단정문은 실패할 수 없습니다.

이 말은, 직관적이기는 하지만, 완전히 잘못된 것입니다. 이건 이론적인 단정이 아님에 주의하시기 바랍니다: 실제로 이 코드를 현실의 약한 순서 규칙의 하드웨어 (1.5 GHz 16-CPU POWER 5 시스템)에서 수행하면 천만번의 수행 중 16번이 단정문이 실패했습니다. 분명, 명시적 메모리 배리어를 사용하는 코드를 작성하는 사람은 반드시 극단적인 테스트를 해야만 합니다 – 정확성 증명이 도움이 될 수는 있지만, 메모리 배리어의 본질적으로 비직관적인 동작은 그런 증명이 기반하는 가정을 상당히 제약할 수 있습니다. 여러개의 하드웨어의 존적인 트릭들이 이 수행에서의 실패 가능성을 상당히 증가시키기 때문에 극단적 테스트의 필요성은 결코 가벼워질 수 없습니다.

Quick Quiz 14.1: page 255의 Figure 14.3 코드의 line 21의 단정문이 대체 어떻게 실패할 수 있죠? ■

Quick Quiz 14.2: 좋아요... 그래서 이걸 어떻게 고쳐야 하죠? ■

그러니 이제 어떡해야 할까요? 최선의 선택지는, 가능하다면 모든 필요한 메모리 배리어를 내포하고 있는 현존하는 기능들을 사용하고 이 챕터의 나머지 내용을 그냥 무시하는 것입니다.

물론, 동기화 도구를 직접 구현하고 있다면 그럴 수는 없습니다. 다음의 메모리 순서 규칙과 메모리 배리어에 대한 이야기는 그런 경우를 위한 것입니다.

14.2.3 Variables Can Have More Than One Value

변수가 잘 정의된 글로벌한 순서로 값의 시퀀스를 갖는다고 생각하는건 자연스러운 일입니다. 하지만, 지금은 안타깝더라도 이런 안락한 상상에 작별을 고해야 할 때입니다.

이를 알기 위해, Figure 14.4의 코드 조각을 보기 바랍니다. 이 코드 조각은 여러 CPU 들에 의해 병렬로 실행됩니다. Line 1은 공유된 변수 하나를 자신의 CPU의 ID로 값을 넣고, line 2에서는 몇개의 변수들을 모든 CPU 들에 동기화 되는, 세밀한 하드웨어 “timebase” 카운터를 얻어오는 (안타깝지만, 모든 CPU 아키텍쳐에서 가능한 일은 아닙니다!) `gettb()` 함수로 얻어온 값으로 초기화 하며, line 3-8의 루프에서는 이 CPU가 변수에 할당한 값을 통해 자신이 루프 내에서 사용한 시간의 길이를 기록합니다. 물론, CPU 들 가운데 하나만 루프에 남는데 “승리하고”, 따라서 line 6-8의 체크에 걸리기 전까지는지 루프를 빠져나가지 않을 것입니다.

Quick Quiz 14.3: Figure 14.4의 코드 조각이 가정하

고 있는, 실제 하드웨어에서는 불가능한 일은 무엇인가요? ■

```

1 state.variable = mycpu;
2 lasttb = oldtb = firsttb = gettb();
3 while (state.variable == mycpu) {
4     lasttb = oldtb;
5     oldtb = gettb();
6     if (lasttb - firsttb > 1000)
7         break;
8 }

```

Figure 14.4: Software Logic Analyzer

루프의 종료 전까지, firsttb 는 최초 할당된 값인 타임스탬프를 가지고 있게 되고 lasttb 는 이번 타임스탬프 샘플링 이전 샘플링에서 얻어져 할당되었고 여전히 변수에 담겨 있는 타임스탬프를, 또는, 해당 변수가 루프 진입 전에 바뀐 값 그대로라면, firsttb 와 동일한 값을 가지고 있을 겁니다. 이는 Figure 14.5 에 그려진 것처럼 532 나노세컨드의 시간동안 각 CPU 가 state.variable 을 보는 시각을 그려볼 수 있게 합니다. 이 데이터는 2006년에 각자 2개의 하드웨어 쓰레드를 갖는 8개의 코어를 갖는 1.5GHz POWER5 시스템에서 얻어졌습니다. CPU 1, 2, 3, 4 는 CPU 0 가 테스트를 제어하는 동안 값을 기록했습니다. 타임스탬프 카운터가 값을 증가시키는 시간은 약 5.32 ns 이었으므로, 중간의 캐시 상태를 관찰하는데 충분히 세밀했습니다.

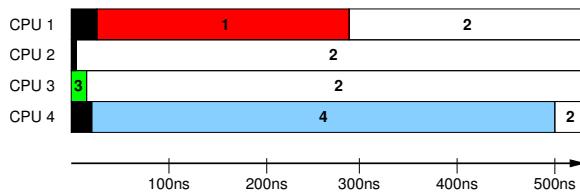


Figure 14.5: A Variable With Multiple Simultaneous Values

각각의 수평선은 각 CPU 의 관찰 결과를 시간에 따라 나타내는데, 왼쪽의 검정색 영역은 해당 CPU 의 첫번째 관측 전의 시간을 나타냅니다. 첫 5ns 동안, CPU 3 만이 해당 변수에 대해 의견을 제시합니다. 다음 10ns 동안, CPU 2 와 3 이 해당 변수의 값에 동의하지 않지만, 곧 이어 해당 값은 “2” 라고 동의하게 되며, 이것이 최종 동의된 값입니다. 하지만, CPU 1 은 이 값이 “1” 이라고 거의 300ns 동안 믿으며, CPU 4 는 거의 500ns 동안이나 해당 값이 “4” 라고 믿습니다.

Quick Quiz 14.4: 어떻게 CPU 들이 하나의 변수에 대해 같은 시간에 그 값을 다르게 볼 수 있을까요? ■

Quick Quiz 14.5: CPU 2 와 3 은 그렇게 빨리 합의에 이르렀는데 CPU 1 과 4 는 그렇게 오래 걸린 이유가 뭐죠? ■

그리고 네개의 CPU 만으로 이루어진 상황에 음모가 있었다고 생각한다면, 같은 상황을 보여주지만 이번에는 15개의 CPU 들이 하나의 공유 변수에 각자의 수를 $t = 0$ 시간에 값 할당하는 상황을 보여주는 Figure 14.5 를 생각해 봅시다. 그림의 가로축은 시간 기반의 턱들로 측정된 시간을 보이는데, 그런 각각의 턱은 약 5.3 나노세컨드 정도씩 유지됩니다. 따라서 전체 이벤트의 흐름은 Figure 14.5 에 기록된 이벤트보다 조금은 더 길게 유지되는데, 이 텀은 CPU 수가 늘어날수록 일관적으로 더 증가합니다. 위의 다이어그램은 전체적인 그림을, 아래쪽의 다이어그램은 처음의 50개 시간 기반 턱들의 관측 결과를 확대해서 보입니다.

다시 말하지만, CPU 0 는 테스트를 관장하므로, 어떤 값도 기록하지 않습니다.

모든 CPU 들이 최종적으로는 마지막 값 9에 합의하게 되는데, 값 15와 12 가 전체 흐름을 주도하기 전입니다. 시간 20 에서는 해당 변수의 값에 대한 14개의 서로 다른 의견이 존재함에 유의하시기 바랍니다. 또한 모든 CPU 들이 Figure 14.7 에 보여지고 있는 방향성 있는 그래프와 일관된 순서의 흐름을 보고 있음을 유의하시기 바랍니다. 더도 아니고 덜도 아니고, 이 표와 그림은 둘 다 메모리 순서에 신경 쓰는 코드에서 올바른 메모리 배리어의 사용의 중요성을 강조하고 있습니다.

이제 우리는 변수의 값과 시간의 흐름에 대한 편안한 직관에 작별을 고해야 하는 정권에 들어왔습니다. 이제 메모리 배리어가 필요해지는 체제입니다.

이 모든 것과 별도로, Chapter 3 와 6 에서 얻은 교훈들을 기억하는 것이 중요합니다. 모든 CPU 들이 같은 변수에 동시적으로 값을 쓰게 하는 것은 결코 병렬 프로그램을 설계하는 방식이 아닙니다, 적어도 성능과 확장성이 전혀 중요하지 않은게 아니라면 말입니다.

14.2.4 What Can You Trust?

당신은 결코 당신의 직관을 믿어선 안됩니다.

당신은 무엇을 믿을 수 있는 걸까요?

당신이 메모리 배리어를 잘 사용할 수 있게 해주는 몇개의 적당히 간단한 규칙들이 있다는 게 밝혀졌습니다. 이 섹션에서는 적어도 이식성 있는 코드의 관점에서 메모리 배리어 이야기의 바탕에 도달해 보고자 하는 사람들을 위해 그런 규칙들을 유도해 봅니다. 실제 유도를 고생스럽게 해보기보다는 그냥 그 규칙들을 간단히 보고 싶다면, 부담 갖지 말고 Section 14.2.6 로 건너뛰어도 좋습니다.

메모리 배리어의 실제 개념은 CPU 에 따라 상당히 다르기 때문에 호환성 있는 코드는 메모리 배리어 개념들 중 최소한의 공통 분모들에만 의존성을 가져야 합니다.

다행히도, 모든 CPU 들이 다음의 규칙들을 갖습니다:

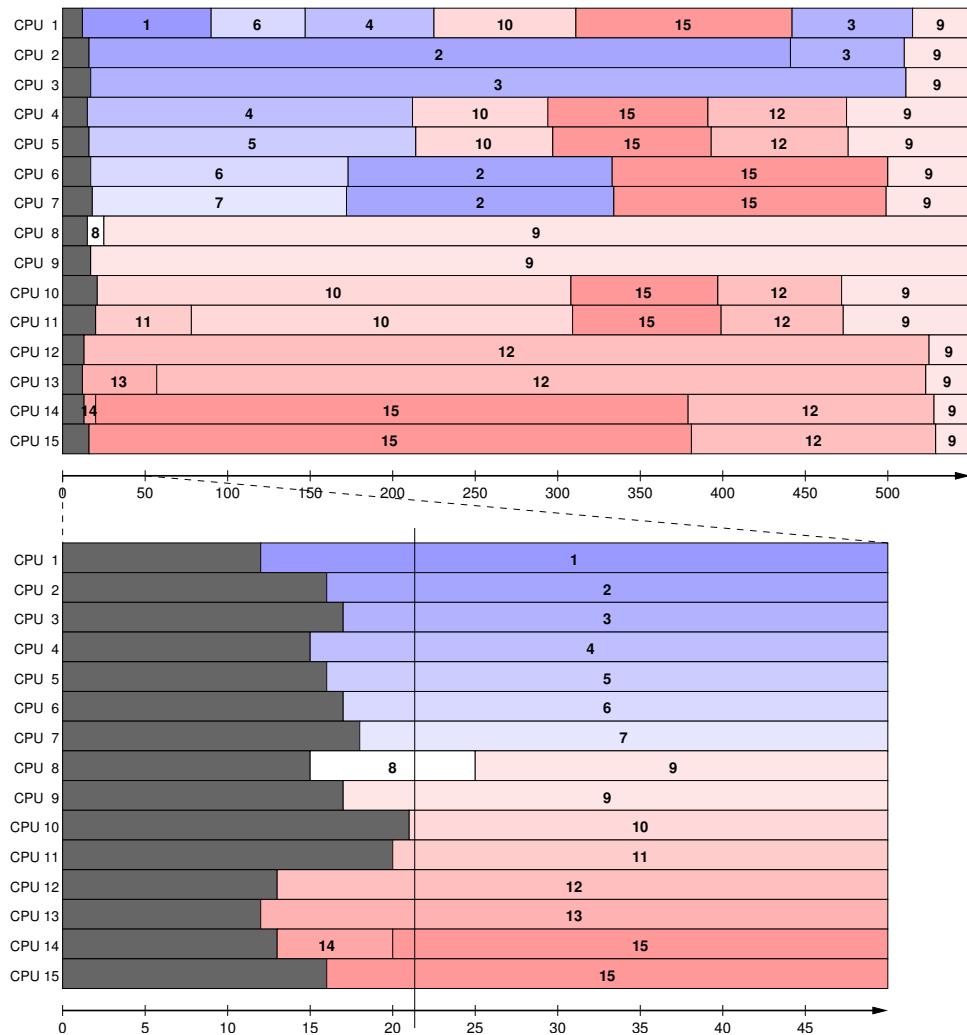


Figure 14.6: A Variable With More Simultaneous Values

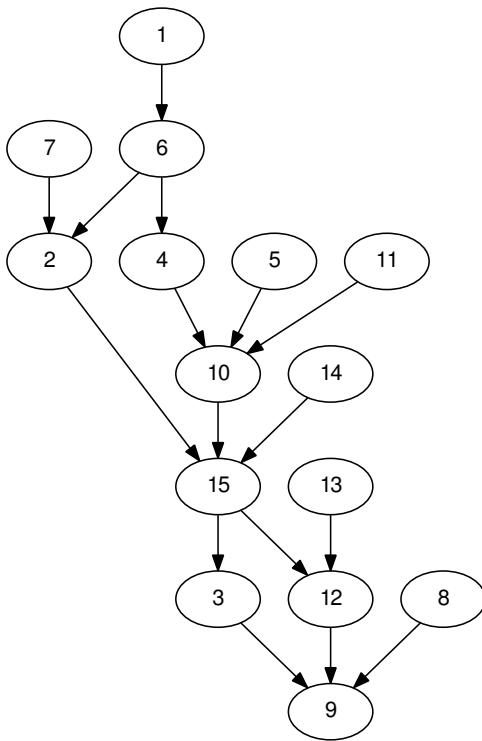


Figure 14.7: Possible Global Orders With More Simultaneous Values

1. 한 CPU 에 의한 모든 액세스들은 그 CPU 에는 프로그램 순서대로 일어나는 것으로 보인다.
2. 모든 CPU 들의 하나의 변수로의 액세스들은 그 변수로의 스토어 오퍼레이션들의 어떤 글로벌 순서 규칙에 맞춰 일관성을 갖는다.
3. 메모리 배리어들은 짹을 맞춰 동작한다.
4. 오퍼레이션들은 배타적 락킹 도구들이 구현되는 곳에 사용될 수 있다.

따라서, 당신이 이식 가능한 코드에서 메모리 배리어를 사용해야 한다면, 이런 특성들에 의존할 수 있습니다.¹ 이런 특성들 각각은 다음의 섹션들에서 설명됩니다.

14.2.4.1 Self-References Are Ordered

하나의 CPU 는 자신의 메모리 액세스들을 마치 인스트럭션들을 재배치나 예측성 수행 없이 한번에 하나의

¹ 또는, 아예 명시적인 메모리 배리어의 사용을 하지 않는게 나을 수도 있습니다. 하지만 그 방법은 다른 섹션의 주제가 될것입니다.

인스트럭션만을 순차적으로 실행하듯이, “프로그램 순서” 대로 일어나는 것처럼 보게 됩니다. 오래된 CPU 들의 경우, 이 제약은 바이너리 호환성을 위해 필요하며, 우리의 타입의 소프트웨어들의 정상성을 부자적인 이유일 뿐입니다. 이 규칙을 제한된 범위 내에서나마 위반한 CPU 들도 일부 있었습니다만, 그런 경우에도 컴파일러에게 순서 규칙이 명시적으로 지켜질 수 있도록 해야 할 의무가 지워져 있었습니다.

어떤 쪽이건, 프로그래머의 관점에서 CPU 는 자신의 액세스들은 프로그램 순서대로 보게 됩니다.

14.2.4.2 Single-Variable Memory Consistency

현재의 상용 컴퓨터 시스템들은 캐시 일관성을 제공하기 때문에, 어떤 CPU 무리가 모두 하나의 변수에 동시에 어토믹하지 않은 스토어를 하게 되면, 모든 CPU 들에 보이는 일련의 값들은 적어도 하나의 글로벌 순서 규칙에 의해 일관적일 것입니다. 예를 들어, Figure 14.5 에 보이는 일련의 액세스들에서, CPU 1 는 {1, 2} 순서로, CPU 2 는 {2} 순서로, CPU 3 는 {3, 2} 순서로, 그리고 CPU 4 는 {4, 2} 순서로 그 값을 보게 됩니다. 이는 글로벌한 순서 {3, 1, 4, 2} 로 일관성을 갖습니다만, 꼭 이 순서가 아니라도 이 네개의 숫자를 가지고 “2”로 끝나는, 다른 다섯개의 순서로 글로벌 순서를 유추하는 것도 가능합니다. 즉, 하나의 변수에서 얻어지는 값의 순서에는 모호함이 있긴 하나, 모든 CPU 의 합의가 이뤄집니다.

반면에, 이 CPU 들이 간단한 유일값의 스토어가 아니라 (리눅스 커널의 `atomic_inc_return()` 과 같은) 어토믹 오퍼레이션들을 사용했다면, CPU 들이 보게 되는 결과들은 하나의 전체적으로 일관된 순서의 값들이 될 것입니다. `atomic_inc_return()` 실행 중 하나가 먼저 일어나서 그 값을 0에서 1로 바꾸고, 다음엔 두번째가 1에서 2로, 그리고 그렇게 진행됩니다. 이 CPU 들은 각자 본 것들을 비교하고 `atomic_inc_return()` 실행의 분명한 순서에의 합의를 얻을 수 있을 것입니다. 이는 앞서 설명한 어토믹하지 않은 스토어로는 불가능한 일인데, 어토믹하지 않은 스토어는 모호성의 가능성으로 인해 이전의 값에 대한 정보를 알려주지 않기 때문입니다.

이 섹션은 모든 CPU 들이 하나의 변수에 접근할 때에만 적용됨에 주의하십시오. 이 단일 변수의 경우, 과감한 컴파일러 최적화가 리눅스 커널의 `ACCESS_ONCE()` 지시어나 C++11 의 완화된 어토믹 오퍼레이션들 [Bec11]를 통해 방지되었다는 최소한의 가정 위에서 캐시 일관성이 글로벌한 순서를 보장합니다. 반면, 만약 여러 변수들이 있다면, 현재의 상용 컴퓨터 시스템들에서는 CPU 들이 일관성 있게 순서에 합의할 수 있도록 하기 위해 메모리 배리어가 필요합니다.

14.2.4.3 Pair-Wise Memory Barriers

짝을 맞추는 메모리 배리어들은 조건적 순서 개념을 제공합니다. 예를 들어, 다음의 오퍼레이션들에 대해, 외부에서 로직 분석을 하는 쪽의 관점에서는 CPU 1 의 A 로의 액세스가 조건 없이 B 로의 액세스를 앞서지는 않습니다 예를 위해 (Appendix B를 참고하세요) (시스템은 그저 이 액세스들이 순서대로 이루어지는 것처럼 동작할 뿐입니다; 실제로 순서대로 이루어지도록 강제될 이유는 없습니다). 하지만, 만약 CPU 2 의 B 로의 액세스가 CPU 1 의 B 로의 액세스의 결과를 보게 된다면, CPU 2 의 A 로의 액세스는 CPU 1 의 A 로의 액세스의 결과 역시 보게 될 것이 보장됩니다. 일부 CPU 의 메모리 배리어들은 실은 좀 더 강력한, 무조건적 순서 개념을 제공하지만, 이식성 있는 코드는 이와 같이 좀 더 약한 조건적 순서 개념만을 사용해야 할 것입니다.

CPU 1	CPU 2
access (A);	access (B);
smp_mb ();	smp_mb ();
access (B);	access (A);

Quick Quiz 14.6: 하지만 메모리 배리어들이 무조건적으로 순서를 강제해 주지 않는다면, 디바이스 드라이버는 대체 어떻게 안정적으로 MMIO 레지스터로의 일련의 로드와 스토어들을 실행할 수 있나요? ■

물론, 액세스들은 로드 또는 스토어야만 하고, 이것들은 서로 다른 특성을 갖습니다. Table 14.2 은 한 쌍의 CPU 에서 모든 가능한 로드와 스토어 조합을 보입니다. 당연하게도, 조건적 순서를 강제하기 위해, 각 CPU 의 오퍼레이션들에 짝을 맞춰 메모리 배리어를 사용해야만 합니다.

14.2.4.4 Pair-Wise Memory Barriers: Portable Combinations

아래에 Table 14.2 에서 이식성 있는 소프트웨어가 사용해야 할 모든 메모리 배리어 조합을 나열합니다.

Pairing 1. 이 조합에서는, 다음과 같이 하나의 CPU 가 메모리 배리어가 중간에 위치한 한 쌍의 로드를 실행하고, 그동안 두 번째 CPU 는 역시 메모리 배리어가 중간에 위치한 한 쌍의 스토어를 실행합니다 (A 와 B 둘 다 처음엔 0 값을 갖습니다):

CPU 1	CPU 2
A=1;	Y=B;
smp_mb ();	smp_mb ();
B=1;	X=A;

두 CPU 모두 이 코드의 수행을 완료한 후에는, 만약 $Y==1$ 이라면 반드시 $X==1$ 이어야 합니다. 이 경우, $Y==1$ 은 CPU 2 의, 자신의 메모리 배리어 이전의 로드가 CPU 1 의 메모리 배리어 후의 스토어를 봤다는 뜻입니다.

입니다. 메모리 배리어 짝맞추기의 원리에 의해, CPU 2 의, 자신의 메모리 배리어 뒤의 로드는 CPU 1 의 메모리 배리어 전의 스토어를 볼 수 있어야 하므로, $X==1$ 입니다.

반면, 만약 $Y==0$ 라면, 메모리 배리어 조건이 잡히지 않으므로, 이 경우 X 는 0이 될 수도, 1이 될 수도 있습니다.

Pairing 2. 이 조합에서, 각 CPU 는 다음과 같이 로드와 메모리 배리어, 그리고 스토어를 순서대로 실행합니다 (A 와 B 는 처음에는 0 값을 갖습니다):

CPU 1	CPU 2
X=A;	Y=B;
smp_mb ();	smp_mb ();
B=1;	A=1;

두 CPU 모두 이 코드의 실행을 완료한 후, 만약 $X==1$ 이라면 $Y==0$ 이어야만 합니다. 이 경우, $X==1$ 이라는 결과는 CPU 1 의, 자신의 메모리 배리어 전의 로드는 CPU 2 의 메모리 배리어 후의 스토어를 봤다는 의미입니다. 메모리 배리어 짝맞추기의 원리에 의해, CPU 1 의, 자신의 메모리 배리어 뒤의 스토어는 CPU 2 의, 자신의 메모리 배리어 앞의 로드의 결과를 봐야 하므로, $Y==0$ 이어야 합니다.

반면, 만약 $X==0$ 라면, 메모리 배리어 조건은 성립되지 않으므로, 이 경우 Y 의 값은 0일 수도, 1일 수도 있습니다.

이 두 CPU 의 코드는 동일하므로, 두 CPU 모두 코드 실행을 마무리 한 후에 $Y==1$ 라면 $X==0$ 일 겁니다.

Pairing 3. 이 조합에서는 다음과 같이 한 CPU 는 로드, 메모리 배리어, 스토어를 순서대로 실행하고, 다른 CPU 는 메모리 배리어를 가운데 두고 두 개의 스토어를 실행합니다 (A 와 B 모두 초기값은 0입니다):

CPU 1	CPU 2
X=A;	B=2;
smp_mb ();	smp_mb ();
B=1;	A=1;

두 CPU 모두 코드 실행을 완료한 후, 만약 $X==1$ 이라면, $B==1$ 이어야 합니다. 이 경우, $X==1$ 은 CPU 1 의, 자신의 메모리 배리어 앞의 로드가 CPU 2 의 메모리 배리어 뒤의 스토어를 봤다는 의미입니다. 메모리 배리어 짝 맞추기의 원리에 의해, CPU 1 의, 자신의 메모리 배리어 뒤의 스토어는 CPU 2 의, 자신의 메모리 배리어 앞의 스토어의 결과를 봐야만 합니다. 이는 CPU 1 의 B 에의 스토어는 CPU 2 의 B 에의 스토어를 덮어써서 $B==1$ 이 됨을 뜻합니다.

반면, 만약 $X==0$ 라면, 메모리 배리어 조건은 성립되지 않고, 따라서 B 의 값은 1도, 2도 될 수 있습니다.

	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

14.2.4.5 Pair-Wise Memory Barriers: Semi-Portable Combinations

Table 14.2에서 가져온 다음의 조합들은 최신 하드웨어에서 사용될 수 있습니다만, 1900년대에 만들어진 일부 시스템에서는 문제가 있을 수 있습니다. 하지만, 2000년 이후 소개된 모든 주류 하드웨어에서는 안전하게 사용될 수 있습니다. 그러니 메모리 배리어가 다루기 어렵다고 생각하신다면, 일부 시스템에는 더 어렵단 점을 기억해 두시기 바랍니다!

Ears to Mouths. 스토어들은 로드들의 결과를 볼 수 없기에(다시 말하지만, MMIO 레지스터는 지금 당장은 잊어주시기 바랍니다), 메모리 배리어 조건이 맞춰졌는지를 항상 알 수는 없습니다. 하지만, 21-세기 하드웨어는 적어도 로드들 중 하나는 연관된 스토어에 의해 저장된 값을 봤다고 보장해줄 수도 있습니다(또는 같은 변수에 대한 뒤의 값을).

Quick Quiz 14.7: ears-to-mouths 시나리오에서 최신 하드웨어가 로드들 중 최소 하나는 다른 쓰레드에 의해 저장된 값을 읽을 거라 보장함을 우리는 어떻게 알죠?

Stores “Pass in the Night”. 다음의 예에서, 두 CPU들이 모두 코드 실행을 끝낸 후, $\{A==1, B==2\}$ 로 결론이 나는 일은 결코 없을 것처럼 보입니다.

CPU 1	CPU 2
$A=1;$ <code>smp_mb();</code> $B=1;$	$B=2;$ <code>smp_mb();</code> $A=2;$

안타깝게도, 이 결론은 21-세기의 시스템들에서는 맞지만, 모든 20-세기 시스템들에서는 맞지 않을 수 있습니다. A를 담고 있는 캐시 라인이 처음에는 CPU 2에 의해 소유되고 있었고, B를 담는 캐시 라인은 처음에는 CPU 1에 의해 소유되고 있었다고 생각해 봅시다. 그리고 나서, 무효화 큐와 스토어 범퍼를 가지고 있는 시스템들에서는, 첫번째 값 할당이 “밤에 지나가는 것 (pass in the night)”이 가능해서, 두번째 값 할당이 실제로는 먼저 일어날 수 있습니다. 이 이상한 효과는 Appendix B에서 설명됩니다.

이와 똑같은 효과는 “ears to mouths” 조합을 포함해, 각 CPU의 메모리 배리어를 스토어가 앞서는 어떤 메모리 배리어 조합에서도 발생할 수 있습니다.

하지만, 21-세기 하드웨어는 이런 순서 규칙을 허용하지 않아서, 이런 조합이 안전하게 사용될 수 있도록 합니다.

14.2.4.6 Pair-Wise Memory Barriers: Dubious Combinations

Table 14.2에서 가져온 다음의 조합들에서 메모리 배리어들은 21-세기 하드웨어에서도 이식성 있는 코드에서는 매우 제한적인 사용만 가능합니다. 하지만 “제한적인 사용”은 “사용 불가”와는 다르니, 뭐가 가능한지 한번 봅시다! 열심인 독자분들은 이게 어떻게 동작하는지 완전히 이해하기 위해 이 조합들 각각을 사용하는 테스트용 프로그램을 작성하고자 할 것입니다.

Ears to Ears. 로드들은 메모리의 상태를 바꾸지 않기 때문에 (MMIO 레지스터는 당장은 무시합시다), 로드

들 중 하나가 다른 로드의 결과를 볼 수는 없습니다. 하지만, 만약 우리가 CPU 2 의 B로부터의 로드가 CPU 1 의 B로부터의 로드보다 새로운 값을 리턴했음을 안다면, 우리는 CPU 2 의 A로부터의 로드가 CPU 1 의 A로부터의 로드와 같거나 그보다 나중의 값을 리턴했음 역시 알 수 있습니다.

Mouth to Mouth, Ear to Ear. 변수들 중 하나는 로드되었고, 다른 하나는 스토어 되었습니다. 로드가 다른 로드의 결과를 보는 것은 불가능하므로 (다시 말하지만, MMIO 레지스터는 무시합니다), 메모리 배리어로 제공되는 조건적 순서를 파악할 수는 없습니다.

하지만, 어떤 스토어가 마지막으로 발생했는지는 알 수 있습니다만, 이를 위해서는 B로부터의 추가적인 로드가 필요합니다. 이 추가적인 B로부터의 로드가 CPU 1 과 CPU 2 가 완료된 후 수행된다면, 그리고 그 수행으로 인해 CPU 2 의 B로의 스토어가 마지막으로 이루어졌음이 밝혀진다면, CPU 2 의 A로부터의 로드는 CPU 1 의 A로부터의 로드와 같거나 나중의 값을 리턴했음 역시 알 수 있습니다.

Only One Store. 하나의 스토어만 있으므로, 변수들 중 하나만이 한 CPU에게 다른 CPU의 액세스의 결과를 볼 수 있게 합니다. 따라서, 메모리 배리어에 의해 제공되는 조건적 순서를 알아챌 방법이 없습니다.

적어도 직접적으로는요. 하지만 Table 14.2의 조합 1에서, CPU 1의 A로부터의 로드가 CPU 2가 A에 저장한 값을 리턴한다고 생각해 봅시다. 그럼 우리는 CPU 1의 B로부터의 로드가 CPU 2의 A로부터의 로드와 같거나 나중의 값을 리턴했음을 알 수 있습니다.

Quick Quiz 14.8: Table 14.2의 다른 “Only one store” 항목은 어떻게 사용될 수 있나요? ■

14.2.4.7 Semantics Sufficient to Implement Locking

우리가 여러개의 변수를 보호하는 (달리 말하자면, 이 변수들은 이 락의 크리티컬 섹션 외에서는 접근되지 않습니다) 배타적 락 (리눅스 커널의 `spinlock_t`나 `pthread` 코드의 `pthread_mutex_t`)을 가지고 있다고 생각해 봅시다. 다음과 같은 특성들이 반드시 지켜져야 합니다:

1. 한 CPU나 쓰레드는 자신의 로드와 스토어들을 그 것들이 프로그램 순서로 이루어진 것처럼 볼 수 있어야만 합니다.
2. 락의 획득과 해제는 하나의 전체적 순서로 이루어 진 것으로 나타나야 합니다.²

² 물론, 이 순서는 매번 같지는 않을 수 있습니다. 하지만, 매번 모든 CPU들과 쓰레드들은 해당 배타적 락의 크리티컬 섹션에 대해

3. 어떤 변수가 현재 수행중인 크리티컬 섹션에서 아직 스토어 되지 않았다고 생각해 봅시다. 그렇다면 해당 크리티컬 섹션에서 수행되는 해당 변수로부터의 어떤 로드도 그 변수에 스토어를 한, 최근의 크리티컬 섹션에서 마지막으로 저장한 값을 가져와야만 합니다.

뒤의 두개 특성 사이의 차이는 약간 사소합니다: 두 번째 것은 락의 획득과 해제가 잘 정의된 순서대로 이루어져야 한다고 하고, 세번째 것은 해당 크리티컬 섹션은 다른 크리티컬 섹션에 곤란을 줄 정도로 “새지” 않아야 한다고 이야기 합니다.

왜 이 특성들이 필요할까요?

첫번째 특성이 지켜지지 않는다고 생각해 봅시다. 그럼 다음 코드의 단정문이 실패합니다!

```
a = 1;
b = 1 + a;
assert(b == 2);
```

Quick Quiz 14.9: 어떻게 page 261의 `b==2` 단정문이 실패할 수 있죠? ■

두번째 특성이 지켜지지 않는다고 생각해 봅시다. 그럼 다음의 코드는 메모리 락을 일으킬 수 있습니다!

```
spin_lock(&mylock);
if (p == NULL)
    p = kmalloc(sizeof(*p), GFP_ATOMIC);
spin_unlock(&mylock);
```

Quick Quiz 14.10: 어떻게 page 261의 코드가 메모리 락을 일으킬 수 있죠? ■

세번째 특성이 지켜지지 않는다고 생각해 봅시다. 그럼 다음 코드의 카운터는 거꾸로 수를 셀 수도 있습니다. 이 세번째 특성은 특히 중요한데, 짹을 맞추는 메모리 배리어들로는 엄격하게 지켜질 수 없기 때문입니다.

```
spin_lock(&mylock);
ctr = ctr + 1;
spin_unlock(&mylock);
```

Quick Quiz 14.11: 어떻게 page 261의 코드가 거꾸로 수를 셀 수 있죠? ■

이 규칙들이 필요함에 납득했다면, 이제 이것들이 어떻게 일반적인 락킹 구현에 동작하는지 알아봅시다.

14.2.5 Review of Locking Implementations

간단한 락과 언락 오퍼레이션 구현의 대략적 수도코드가 아래에 있습니다. `atomic_xchg()` 기능은 어토믹한 교체 작업 전후로 메모리 배리어를 내포하고 있으며, 이 어토믹한 교체 작업 후의 내포된 배리어로 인해 `spin_lock()`에서의 모여시적 메모리 배리어는 필요치 않습니다. 또한, 이름과 달리 `atomic_read()` 와

일관된 순서를 봐야만 합니다.

`atomic_set()` 은 어떤 어토믹 인스트럭션을 실행하지 않고, 대신 그저 단순히 로드와 스토어 오퍼레이션을 행합니다. 언락 오퍼레이션에 대해, 이 수도 코드는 간단한 어토믹하지 않은 스토어와 이어지는 메모리 배리어라는, 여러 리눅스 구현을 따라합니다. 이 최소한의 구현은 Section 14.2.4 에서 이야기한 모든 락킹 특성을 가져야만 합니다.

```

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 }

```

`spin_lock()` 기능은 앞의 `spin_unlock()` 기능이 완료되기 전까지는 수행될 수 없습니다. CPU 1 이 CPU 2 가 획득하려 하는 락을 해제한다면, 오퍼레이션들의 차례는 다음과 같을 것입니다:

CPU 1 (critical section)	CPU 2
smp_mb();	atomic_xchg(&lck->a, 1) ->1
lck->a=0;	lck->a->1
	lck->a->1
	lck->a->0
	(implicit smp_mb() 1)
	atomic_xchg(&lck->a, 1) ->0
	(implicit smp_mb() 2)
	(critical section)

이 특별한 경우, 짹을 맞춘 메모리 배리어만으로도 이 두개의 크리티컬 섹션을 만드는데 충분합니다. CPU 2 의 `atomic_xchg(&lck->a, 1)` 은 CPU 1 의 `lck->a=0` 를 보게 되고, 따라서 CPU 2 의 다음 크리티컬 섹션은 반드시 CPU 1 의 앞선 크리티컬 섹션에서 행한 것들을 볼 수 있게 됩니다. 거꾸로, CPU 1 의 크리티컬 섹션은 CPU 2 의 크리티컬 섹션이 할 일을 볼 수 없습니다.

14.2.6 A Few Simple Rules

아마도 메모리 배리어를 이해하는 가장 쉬운 방법은 몇 개의 간단한 규칙들을 이해하는 것입니다:

1. 각 CPU 는 자신의 액세스들을 순서대로 봅니다.
2. 하나의 공유된 변수가 여러 CPU 들에 의해 로드되고 스토어 되면, 한 CPU 에 보이는 값들의 순서는 다른 CPU 들에 보이는 순서와 일관되고, 각 CPU 들의 순서가 일관되는 값들이 해당 변수에 저장되는 순서가 적어도 하나는 존재할 것입니다.³

³ 한 CPU 의 순서는 물론 완전하지 않을 수 있는데, 예를 들어

3. 만약 한 CPU 가 변수 A 와 B 에의 스토어를 순서 잡고⁴ , 두번째 CPU 가 B 와 A 로부터의 로드를 순서 잡는다면,⁵ 그리고 두번째 CPU 의 B 로부터의 로드가 첫번째 CPU 가 저장한 값을 얻어온다면, 두번째 CPU 의 A 로부터의 로드 역시 첫번째 CPU 가 저장한 값을 가져와야만 합니다.

4. 만약 한 CPU 가 B 로의 스토어 전으로 순서잡아 A 로부터의 로드를 행한다면, 그리고 두번째 CPU 의 A 로의 스토어 이전으로 순서잡아 B 로부터의 로드를 행한다면, 그리고 만약 두번째 CPU 의 B 로부터의 로드가 첫번째 CPU 가 저장한 값을 가져온다면, 첫번째 CPU 의 A 로부터의 로드는 두번째 CPU 가 저장한 값을 가져오지 않아야만 합니다.

5. 만약 한 CPU 가 B 로의 스토어 후로 순서 잡아 A 로부터의 로드를 한다면, 그리고 만약 두번째 CPU 가 A 로의 스토어 후로 순서 잡아 B 로부터의 로드를 한다면, 그리고 만약 첫번째 CPU 의 A 로부터의 로드가 두번째 CPU 가 저장한 값을 가져온다면, 첫번째 CPU 의 B 로의 스토어는 두번째 CPU 의 B 로의 스토어 뒤에 일어나야만 하며, 따라서 첫번째 CPU 가 저장한 값이 존재해야 합니다.⁶

다음 섹션에서 이런 규칙들에 대한 좀 더 실제 동작에 대해 알아봅니다.

14.2.7 Abstract Memory Access Model

Figure 14.8 에 그려진 추상적 모델의 시스템을 고려해 봅시다.

각 CPU 는 메모리 액세스 오퍼레이션들을 생성해내는 프로그램을 실행합니다. 추상적으로 CPU 에서는 메모리 오퍼레이션의 순서 규칙은 매우 완화되어 있고, CPU 는 실제로 프로그램의 인과성이 지켜지는 것으로 보이도록만 한다는 조건 아래, 메모리 오퍼레이션들을 원하는 대로 어떤 순서로든 실행할 수 있습니다. 비슷하게, 컴파일러는 프로그램의 명백한 수행에 영향을 끼치지 않는다는 조건 아래 원하는 대로 어떤 순서로든 생성해내는 인스트럭션을 배열할 수 있습니다.

따라서 앞의 그림에서, 한 CPU 에 의해 수행되는 메모리 오퍼레이션들의 결과는 시스템의 다른 CPU 들에게 해당 CPU 와 나머지들 사이의 인터페이스(점선)을 지나는 오퍼레이션들에 의해 인지됩니다.

한 CPU 가 해당 공유 변수를 로드하지도 스토어 하지도 않았다면, 그 변수의 값에 대해 어떤 의견도 갖지 않을 수 있습니다.

⁴ 예를 들어, A 로의 스토어, 메모리 배리어, 그리고 나서 B 로의 스토어를 순서대로 실행하는 방법으로.

⁵ 예를 들어, B 를 로드하고 메모리 배리어를 친 후 A 로부터의 로드를 수행하는 식으로.

⁶ 또는, 더 경쟁상황 기반으로 이야기 하자면, 첫번째 CPU 의 B 로의 스토어가 “승리”합니다.

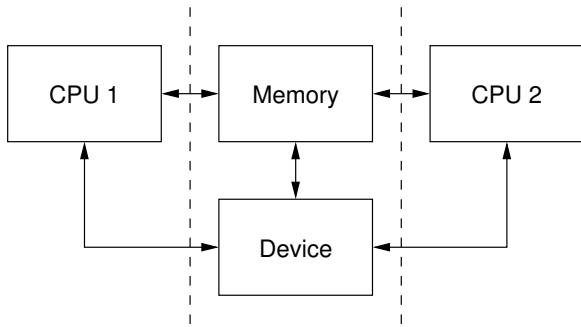


Figure 14.8: Abstract Memory Access Model

예를 들어, 초기값으로 $\{A = 1, B = 2\}$ 를 갖는다는 가정 아래 다음의 코드를 생각해 봅시다:

CPU 1	CPU 2
$A = 3;$	$x = A;$
$B = 4;$	$y = B;$

해당 메모리 시스템에 의해 보여지는 액세스들의 집합은 다음과 같이, 로드를 “ld”로, 스토어를 “st”로 표현하면서 24개의 다른 조합으로 만들어질 수 있습니다:

```

st A=3,    st B=4,    x=ld A→3,    y=ld B→4
st A=3,    st B=4,    y=ld B→4,    x=ld A→3
st A=3,    x=ld A→3,    st B=4,    y=ld B→4
st A=3,    x=ld A→3,    y=ld B→2,    st B=4
st A=3,    y=ld B→2,    st B=4,    x=ld A→3
st A=3,    y=ld B→2,    x=ld A→3,    st B=4
st B=4,    st A=3,    x=ld A→3,    y=ld B→4
st B=4,    ...
```

따라서 수행 결과로 다음과 같이 네개의 서로 다른 값 조합이 만들어질 수 있습니다:

```

x == 1,    y == 2
x == 1,    y == 4
x == 3,    y == 2
x == 3,    y == 4
```

뿐만 아니라, 한 CPU 에 의해 메모리 시스템에 요청된 스토어들은 다른 CPU 에 의해 만들어진 로드 오퍼레이션에 스토어들이 실제 행해진 순서와 다른 순서로 인지될 수도 있습니다.

예를 들기 위해, 초기값 $\{A = 1, B = 2, C = 3, P = \&A, Q = \&C\}$ 의 다음과 같은 이벤트들을 생각해 봅시다:

CPU 1	CPU 2
$B = 4;$	$Q = P;$
$P = \&B$	$D = *Q;$

D 로 할당되는 값은 CPU 2 에 의해 P 로부터 얻어진 주소에 의존적이므로, 여기엔 분명한 데이터 의존성이 존재합니다. 이 코드가 실행 완료된 후, 다음과 같은 결과가 모두 만들어질 수 있습니다:

```

(Q == &A) and (D == 1)
(Q == &B) and (D == 2)
(Q == &B) and (D == 4)
```

CPU 2 는 $*Q$ 의 로드를 요청하기 전에 P 를 Q 에 할당할 것이므로 D 에 C 값을 넣지는 않을 것입니다.

14.2.8 Device Operations

일부 디바이스들은 컨트롤 인터페이스를 메모리의 특정 위치들의 집합으로 제공하는데, 이 컨트롤 레지스터들이 액세스 되는 순서가 매우 중요합니다. 예를 들어, 주소 포트 레지스터 (A) 와 데이터 포트 레지스터 (D) 를 통해 접근되는 내부 레지스터들을 갖는 이더넷 카드를 생각해 봅시다. 내부 레지스터 5를 읽으려면, 다음의 코드가 사용될 수 있을 것입니다:

```

*A = 5;
x = *D;
```

하지만 이는 다음의 두 순서로 만들어질 수 있을 것입니다:

```

STORE *A = 5, x = LOAD *D
x = LOAD *D, STORE *A = 5
```

해당 레지스터를 읽은 뒤에 그 주소를 설정하는 두 번째 순서는 거의 틀림없이 오동작을 일으킬 겁니다.

14.2.9 Guarantees

하나의 CPU 에 대해 기대할 수 있는, 몇개의 최소한의 보장들이 있습니다:

1. 어떤 CPU 에서든, 의존성을 갖는 메모리 액세스들은 해당 CPU 의 관점에서는 순서대로 요청됩니다. 따라서, 다음의 코드에서:

```
Q = P; D = *Q;
```

해당 CPU 는 다음과 같이 메모리 오퍼레이션들을 요청할 겁니다:

```
Q = LOAD P, D = LOAD *Q
```

그리고 항상 이 순서만을 만들 겁니다.

2. 특정 CPU 내에서 로드와 스토어들을 중복되게 하면 그 CPU 내에서는 순서가 맞춰지는 것으로 보일 것입니다. 따라서, 다음의 코드에서:

```
a = *X; *X = b;
```

해당 CPU 는 다음과 같은 순서로만 메모리 오퍼레이션들을 요청할 겁니다:

```
a = LOAD *X, STORE *X = b
```

그리고 다음의 코드에서는:

```
*X = c; d = *X;
```

해당 CPU 는 다음의 순서로만 오퍼레이션을 요청합니다:

```
STORE *X = c, d = LOAD *X
```

(로드들과 스토어들은 그것들이 메모리의 중복된 지역에 가해질 때에 중복됩니다.)

- 하나의 변수로의 일련의 스토어들은 모든 CPU 들에 하나의 순서로 가해진 것으로 보이게 됩니다만, 이 순서는 코드만으로 예측할 수 없고, 실제로 그 순서는 매 실행마다 다를 것입니다.

그리고 반드시 가정되거나 절대로 가정되지 말아야 하는 것들이 있습니다:

- 의존성 없는 로드들과 스토어들은 주어진 순서대로 요청될 것이라고는 절대로 가정하지 말아야 합니다. 무슨 말이냐면, 다음의 코드는:

```
X = *A; Y = *B; *D = Z;
```

다음의 순서들을 만들어 낼 수 있습니다:

```
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
```

- 중복되는 메모리 액세스들은 병합되거나 버려질 수 있음을 반드시 가정해 두어야 합니다. 무슨 말이냐면, 다음의 코드는:

- It *must* be assumed that overlapping memory accesses may be merged or discarded. This means that for:

```
X = *A; Y = *(A + 4);
```

다음의 순서들을 만들어 낼 수 있습니다:

```
X = LOAD *A; Y = LOAD *(A + 4);
Y = LOAD *(A + 4); X = LOAD *A;
{X, Y} = LOAD {*A, *(A + 4)};
```

그리고 다음의 코드에서는:

```
*A = X; *(A + 4) = Y;
```

다음의 순서들을 얻을 수 있습니다:

```
STORE *A = X; STORE *(A + 4) = Y;
STORE *(A + 4) = Y; STORE *A = X;
STORE {*A, *(A + 4)} = {X, Y};
```

마지막으로, 다음의 코드는:

```
*A = X; *A = Y;
```

다음의 순서들을 만들어 낼 수 있습니다:

```
STORE *A = X; STORE *A = Y;
STORE *A = Y;
```

14.2.10 What Are Memory Barriers?

앞에서 봤듯이, 종속성 없는 메모리 오퍼레이션들은 무작위적 순서로 수행됩니다만, 이는 CPU 와 CPU 간 상호작용과 I/O 에 문제가 될 수 있습니다. 따라서 컴파일러와 CPU 에게 그 순서를 강제할 수 있게 개입할 수 있는 수단이 필요합니다.

메모리 배리어들이 그런 개입입니다. 그것들은 배리어의 앞과 뒤 양쪽의 메모리 오퍼레이션들에 대해 부분적인 순서를 만들어 줍니다.

CPU 들과 시스템의 다른 디바이스들은 성능을 끌어올리기 위해 다양한 속임수 - 재배치, 집행 연기, 그리고 메모리 오퍼레이션들의 조합; 투기적 로드; 투기적 브랜치 예측 그리고 다양한 타입의 캐싱 등 - 를 사용할 수 있기 때문에 이런 강제력이 중요합니다. 메모리 배리어들은 이런 속임수들을 무효로 하거나 억제해서 코드가 여러 CPU 들과 디바이스들 사이의 상호 작용을 정상적으로 제어할 수 있게 하기 위해 사용됩니다.

14.2.10.1 Explicit Memory Barriers

메모리 배리어들은 네가지 종류가 있습니다:

- 쓰기 (또는 스토어) 메모리 배리어들,
- 데이터 종속성 배리어들,
- 읽기 (또는 로드) 메모리 배리어들, 그리고
- 범용 메모리 배리어들.

각 타입에 대해 아래에서 설명합니다.

쓰기 메모리 배리어들 쓰기 메모리 배리어는 해당 배리어 앞에서 명기된 모든 STORE 오퍼레이션들이 해당 배리어 뒤에 명기된 모든 STORE 오퍼레이션들보다 먼저 수행된 것으로 시스템의 다른 요소들에 보이도록 하는 것을 보장합니다.

쓰기 메모리 배리어는 스토어들에만 적용되는 부분적 순서세우기입니다; 로드들에 어떤 영향을 끼칠 것은 요구되지 않습니다.

하나의 CPU 는 시간의 흐름에 따라 메모리 시스템에 일련의 스토어 오퍼레이션들을 일으키는 것으로 보여질 수 있습니다. 쓰기 배리어 전의 모든 스토어들은 쓰기 배리어 후의 모든 스토어들 이전에 행해질 겁니다.

쓰기 배리어들은 일반적으로 읽기나 데이터 종속성 배리어들과 짹을 맞춰 사용됨에 유의하세요; Section 14.2.10.6 을 참고하세요.

데이터 종속성 배리어들 데이터 종속성 배리어는 읽기 배리어의 완화된 형태입니다. 두번째 것이 첫번째 것의 결과에 의존하는 것과 같은(예: 첫번째 로드는 두번째 로드가 향하게 될 주소를 얻어옴) 두개의 로드들이 수행되는 경우, 두번째 로드의 타겟이 첫번째 로드에 의해 얻어지는, 해당 주소가 액세스되기 전에 업데이트되었음을 분명히 하기 위해 데이터 종속성 배리어가 필요할 겁니다.

데이터 종속성 배리어는 상호 의존적인 로드들에만 적용되는 부분적 순서 세우기입니다; 어떤 스토어들이나 종속성 없는 로드들이나 중복되는 로드들에 대해서는 어떤 영향을 끼칠 의무가 없습니다.

쓰기 메모리 배리어들에 대해 이야기 했듯, 시스템의 다른 CPU들은 특정 CPU가 원한다면 볼 수 있는 일련의 스토어들을 메모리 시스템에 일으키는 것으로 보여질 수 있습니다. 다른 CPU의 스토어 오퍼레이션을 보기 원하는 CPU가 일으키는 데이터 종속성 배리어는 그것 앞의 모든 로드 오퍼레이션들에 대해, 만약 그 로드가 일련의 다른 CPU로부터의 스토어들 중 하나를 만진다면, 해당 배리어가 완료되는 시점에서는 그 로드에 의한 만점 앞의 모든 스토어들은 이 데이터 종속성 배리어 뒤에 요청되는 모든 로드에 지각될 수 있을 것입니다.

순서 규칙을 보여주는 그림을 위해 Section 14.2.10.7을 참고하세요.

첫번째 로드는 데이터 종속성을 가져야 하지 컨트롤 종속성을 가져야 하는게 아님에 유의하세요. 만약 두번째 로드가 목표로 하는 주소가 첫번째 로드에 의존적이라면, 하지만 그 의존성이 제어에 의해서이지 그 주소 자체를 로드하는게 아니라면, 그것은 컨트롤 종속성이고 이 경우엔 읽기 배리어나 그보다 더한 것이 필요해집니다. 더 많은 내용을 위해 Section 14.2.10.5를 참고하세요.

일반적으로 데이터 종속성 배리어들은 일반적으로 쓰기 배리어들과 짹을 맞춰 사용되어야 합니다; Section 14.2.10.6을 참고하세요.

Read Memory Barriers 읽기 배리어는 데이터 종속성 배리어에 더해서 해당 배리어 앞에 명기된 모든 LOAD 오퍼레이션들이 해당 배리어 뒤에 명기된 모든 LOAD 오퍼레이션들 보다 먼저 행해진 것으로 시스템의 다른 컴포넌트들에 보이게 함을 보장합니다.

읽기 배리어는 로드들에만 적용되는 부분적 순서 세우기입니다; 스토어들에 어떤 영향을 끼칠 의무는 없습니다.

읽기 메모리 배리어들은 데이터 종속성 배리어를 내포하고 있으므로, 그것들을 대체할 수도 있습니다.

읽기 배리어들은 일반적으로 쓰기 배리어들과 짹을 맞춰 사용되어야 합니다; Section 14.2.10.6을 참고하세요.

요.

General Memory Barriers 범용 메모리 배리어는 해당 배리어 앞에서 명기된 모든 LOAD와 STORE 오퍼레이션들이 해당 배리어 뒤에서 명기된 모든 LOAD와 STORE 오퍼레이션들보다 먼저 실행된 것으로 시스템의 다른 컴포넌트들에 보이도록 함을 보장합니다.

범용 메모리 배리어는 로드와 스토어 둘 다에 적용되는 부분적 순서 세우기입니다.

범용 메모리 배리어들은 일기과 쓰기 메모리 배리어 둘 둘 다를 내포하므로, 둘 둘 대체할 수 있습니다.

14.2.10.2 Implicit Memory Barriers

묵시적 메모리 배리어라는 유형들이 존재하는데, 락킹 기능들에 내재되어 있기 때문에 그렇게 불립니다:

1. LOCK 오퍼레이션들 그리고
2. UNLOCK 오퍼레이션들.

LOCK 오퍼레이션들 락 오퍼레이션은 단방향으로 투과될 수 있는 배리어처럼 동작합니다. 이것은 시스템의 나머지 컴포넌트들에게 락 오퍼레이션 뒤의 모든 메모리 오퍼레이션들이 락 오퍼레이션 후에 실행되는 것처럼 보이게 될 것을 보장합니다.

락 오퍼레이션 전의 메모리 오퍼레이션들은 락 오퍼레이션이 완료된 뒤에 실행되는 것처럼 보일 수 있습니다.

락 오퍼레이션은 거의 항상 언락 오퍼레이션과 함께 사용됩니다.

UNLOCK 오퍼레이션 언락 오퍼레이션들 역시 단방향으로 투과될 수 있는 배리어처럼 동작합니다. 이것은 시스템의 나머지 컴포넌트들에게 언락 오퍼레이션 앞의 메모리 오퍼레이션들이 언락 오퍼레이션 전에 행해진 것처럼 보이게 될 것을 보장합니다.

언락 오퍼레이션 뒤의 메모리 오퍼레이션들은 언락 오퍼레이션이 완료되기 전에 수행되는 것처럼 보일 수 있습니다.

락과 언락 오퍼레이션들은 서로에게 명기된 순서대로 보이게 될 것이 보장됩니다.

락과 언락 오퍼레이션들의 사용은 일반적으로 다른 종류의 메모리 배리어의 사용을 막습니다(하지만 Section 14.2.8에서 설명한 예외에 유의하십시오).

Quick Quiz 14.12: 다음의 코드는 변수 “a”와 “b”로의 스토어들의 순서에 어떤 영향을 끼칠까요?

```

a = 1;
b = 1;
<write barrier>

```

14.2.10.3 What May Not Be Assumed About Memory Barriers?

메모리 배리어들이 특정 아키텍쳐 외에서는 보장하지 못하는 것들이 있습니다:

1. 메모리 배리어 앞에 명기된 메모리 액세스들이 메모리 배리어 명령의 완료 이전에 완료된다는 보장은 없습니다; 배리어는 CPU의 액세스 큐에 연관된 탑입의 액세스들만 지나갈 수 없는 선을 긋는다고 볼 수 있습니다.
2. 한 CPU에 메모리 배리어를 가하는 것이 다른 CPU나 시스템의 다른 하드웨어에 직접적인 영향을 준다는 보장은 없습니다. 배리어가 끼치는 간접적인 영향은 두번째 CPU가 첫번째 CPU의 액세스가 일어나는 결과를 보게 되는 순서가 되겠습니다만, 다음 항목을 보세요.
3. 두번째 CPU가 메모리 배리어를 사용한다 해도, 첫번째 CPU 역시 대응하는 메모리 배리어를 사용하지 않는다면 두번째 CPU의 액세스의 결과를 올바른 순서로 보게 된다는 보장은 없습니다 (Section 14.2.10.6를 참고하세요).
4. CPU들 사이의 CPU-외부-하드웨어⁷가 메모리 액세스를 재배치 하지 않는다는 보장은 없습니다. CPU 캐시 일관성 메커니즘은 메모리 배리어의 간접적 영향을 CPU들 사이에 전파하지만, 순서를 지키진 않습니다.

14.2.10.4 Data Dependency Barriers

데이터 의존성 배리어의 사용 필요성은 약간 애매하고, 그 필요성이 항상 분명하지는 않습니다. 이를 자세히 보기 위해, $\{A = 1, B = 2, C = 3, P = \&A, Q = \&C\}$ 초기 조건을 갖는 가정 하에 다음의 이벤트들을 생각해 봅시다:

CPU 1 B = 4; <write barrier> P = &B;	CPU 2 Q = P; D = *Q;
---	----------------------------

⁷ 이건 운영체제 커널의 주요 관심사가 됩니다. 하드웨어 오퍼레이션들과 메모리 순서에 대해 더 많은 정보를 위해선, 리눅스 소스 트리 [Tor03c]에 있는 Documentation 디렉토리의 pci.txt, DMA-API-HOWTO.txt, DMA-API.txt를 참고하세요.

여기엔 분명한 데이터 의존성이 존재하고, 직관적으로 볼 때 이 이벤트들이 종료된 후에, Q는 &A 또는 &B여야 할 것이고, 또한 다음과 같을 것입니다:

```

(Q == &A) implies (D == 1)
(Q == &B) implies (D == 4)

```

그러나, 직관에 반대되지만 CPU 2의 P에 대한 인식은 B의 인식 전에 업데이트 될 수 있고, 따라서 다음의 상황을 이끌어낼 수 있습니다:

```

(Q == &B) and (D == 2) ???

```

이는 일관성이나 인과성의 관리에 실패한 것처럼 보일 수 있지만, 그렇지 않으며, 이 동작은 일부 실제 CPU들 (DEC Alpha와 같은)에서 볼 수 있습니다.

이걸 처리하기 위해, 데이터 의존성 배리어가 다음과 같이 주소의 로드와 데이터의 로드 사이에 삽입되어야만 합니다 (역시 초기값은 $\{A = 1, B = 2, C = 3, P = \&A, Q = \&C\}$ 와 같습니다):

CPU 1 B = 4; <write barrier> P = &B;	CPU 2 Q = P; <data dependency barrier> D = *Q;
---	---

이는 두개의 영향 중 하나를 제거하고, 세번째 가능성을 방지합니다. 이 매우 비직관적인 상황은 분할된 캐시를 갖는 머신에서 쉽게 발생 가능한데, 예를 들어 한 캐시 뱅크가 짹수 번호 캐시 라인을 처리하고 다른 뱅크는 홀수 번호 캐시 라인을 처리한다고 생각해 봅시다. 포인터 P는 홀수 번호 캐시 라인에 저장되어 있고 변수 B는 짹수 번호 캐시 라인에 젖어되어 있습니다. 그리고, 만약 읽는 CPU의 캐시의 짹수 번호 뱅크가 엄청 바쁜데 홀수 번호 뱅크는 쉬고 있었다면, (&B 일) 포인터 P의 값은 새 값을 보지만, 변수 B는 (2 인) 옛 값을 볼 수도 있습니다.

데이터 의존성 배리어가 필요한 또 다른 예는 다음과 같이 숫자가 메모리에서 읽혀지고는 배열에 인덱스를 계산하는데 사용되는 예입니다 (초기 값은 $\{M[0] = 1, M[1] = 2, M[3] = 3, P = 0, Q = 3\}$ 입니다):

CPU 1 M[1] = 4; <write barrier> P = 1;	CPU 2 Q = P; <data dependency barrier> D = M[Q];
---	---

데이터 의존성 배리어는 리눅스 커널의 RCU 시스템에 매우 중요한데, 예를 위해 include/linux/rcupdate.h의 rcu_dereference()를 참고하세요.

요. 이는 RCU 된 포인터의 현재 타겟이 새로운, 수정된 타겟으로 바뀌되어 바뀌는 타겟이 초기화가 덜 된 것으로 보이는 것을 방지할 수 있게 합니다.

더 큰 예를 위해 Section 14.2.13.1 를 참고하세요.

14.2.10.5 Control Dependencies

컨트롤 의존성은 특히나 까다로운데 현재의 컴파일러들은 이를 이해하고 있지 않기 때문입니다. 이 섹션의 규칙과 예제들은 여러분이 여러분의 컴파일러의 무지가 여러분의 코드에 문제를 만드는 것을 막는 것을 돕고자 합니다.

Load-load 컨트롤 의존성은 단순한 데이터 의존성 배리어가 아니라, 전체 읽기 메모리 배리어를 필요로 합니다. 다음의 코드를 생각해 봅시다:

```
1 q = READ_ONCE(x);
2 if (q) {
3     <data dependency barrier>
4     q = READ_ONCE(y);
5 }
```

이는 원했던 효과를 얻지 못할텐데, 여기에는 실제 데이터 의존성은 존재하지 않고, CPU 가 결과를 미리 예측하여 해서는 다른 CPU 들이 “b”로부터의 로드가 “a”로부터의 로드보다 전에 일어난 것처럼 보이게 할 수 있는 컨트롤 의존성이 존재하기 때문입니다. 이런 경우에 정말로 필요한 것은 다음과 같은 코드입니다:

```
1 q = READ_ONCE(x);
2 if (q) {
3     <read barrier>
4     q = READ_ONCE(y);
5 }
```

However, stores are not speculated. This means that ordering *is* provided for load-store control dependencies, as in the following example:

```
1 q = READ_ONCE(x);
2 if (q)
3     WRITE_ONCE(y, 1);
```

Control dependencies pair normally with other types of barriers. That said, please note that neither READ_ONCE() nor WRITE_ONCE() are optional! Without the READ_ONCE(), the compiler might combine the load from x with other loads from x. Without the WRITE_ONCE(), the compiler might combine the store to y with other stores to y. Either can result in highly counterintuitive effects on ordering.

Worse yet, if the compiler is able to prove (say) that the value of variable x is always non-zero, it would be well within its rights to optimize the original example by eliminating the “if” statement as follows:

```
1 q = READ_ONCE(x);
2 WRITE_ONCE(y, 1); /* BUG: CPU can reorder!!! */
```

It is tempting to try to enforce ordering on identical stores on both branches of the “if” statement as follows:

```
1 q = READ_ONCE(x);
2 if (q) {
3     barrier();
4     WRITE_ONCE(y, 1);
5     do_something();
6 } else {
7     barrier();
8     WRITE_ONCE(y, 1);
9     do_something_else();
10 }
```

Unfortunately, current compilers will transform this as follows at high optimization levels:

```
1 q = READ_ONCE(x);
2 barrier();
3 WRITE_ONCE(y, 1); /* BUG: No ordering!!! */
4 if (q) {
5     do_something();
6 } else {
7     do_something_else();
8 }
```

Now there is no conditional between the load from x and the store to y, which means that the CPU is within its rights to reorder them: The conditional is absolutely required, and must be present in the assembly code even after all compiler optimizations have been applied. Therefore, if you need ordering in this example, you need explicit memory barriers, for example, a release store:

```
1 q = READ_ONCE(x);
2 if (q) {
3     smp_store_release(&y, 1);
4     do_something();
5 } else {
6     smp_store_release(&y, 1);
7     do_something_else();
8 }
```

The initial READ_ONCE() is still required to prevent the compiler from proving the value of x.

In addition, you need to be careful what you do with the local variable q, otherwise the compiler might be able to guess the value and again remove the needed conditional. For example:

```
1 q = READ_ONCE(x);
2 if (q % MAX) {
3     WRITE_ONCE(y, 1);
4     do_something();
5 } else {
6     WRITE_ONCE(y, 2);
7     do_something_else();
8 }
```

If MAX is defined to be 1, then the compiler knows that (q%MAX) is equal to zero, in which case the compiler

is within its rights to transform the above code into the following:

```
1 q = READ_ONCE(x);
2 WRITE_ONCE(y, 2);
3 do_something_else();
```

Given this transformation, the CPU is not required to respect the ordering between the load from variable `x` and the store to variable `y`. It is tempting to add a `barrier()` to constrain the compiler, but this does not help. The conditional is gone, and the barrier won't bring it back. Therefore, if you are relying on this ordering, you should make sure that `MAX` is greater than one, perhaps as follows:

```
1 q = READ_ONCE(x);
2 BUILD_BUG_ON(MAX <= 1);
3 if (q % MAX) {
4     WRITE_ONCE(y, 1);
5     do_something();
6 } else {
7     WRITE_ONCE(y, 2);
8     do_something_else();
9 }
```

Please note once again that the stores to `y` differ. If they were identical, as noted earlier, the compiler could pull this store outside of the “`if`” statement.

You must also avoid excessive reliance on boolean short-circuit evaluation. Consider this example:

```
1 q = READ_ONCE(x);
2 if (q || 1 > 0)
3     WRITE_ONCE(y, 1);
```

Because the first condition cannot fault and the second condition is always true, the compiler can transform this example as following, defeating control dependency:

```
1 q = READ_ONCE(x);
2 WRITE_ONCE(y, 1);
```

This example underscores the need to ensure that the compiler cannot out-guess your code. More generally, although `READ_ONCE()` does force the compiler to actually emit code for a given load, it does not force the compiler to use the results.

In addition, control dependencies apply only to the then-clause and else-clause of the `if`-statement in question. In particular, it does not necessarily apply to code following the `if`-statement: >>>> paul/master

```
1 q = READ_ONCE(x);
2 if (q) {
3     WRITE_ONCE(y, 1);
4 } else {
5     WRITE_ONCE(y, 2);
6 }
7 WRITE_ONCE(z, 1); /* BUG: No ordering. */
```

It is tempting to argue that there in fact is ordering

because the compiler cannot reorder volatile accesses and also cannot reorder the writes to `y` with the condition. Unfortunately for this line of reasoning, the compiler might compile the two writes to `y` as conditional-move instructions, as in this fanciful pseudo-assembly language:

```
1 ld r1,x
2 cmp r1,$0
3 cmov,ne r4,$1
4 cmov,eq r4,$2
5 st r4,y
6 st $1,z
```

A weakly ordered CPU would have no dependency of any sort between the load from `x` and the store to `z`. The control dependencies would extend only to the pair of `cmove` instructions and the store depending on them. In short, control dependencies apply only to the stores in the “then” and “else” of the “`if`” in question (including functions invoked by those two clauses), not to code following that “`if`”.

Finally, control dependencies do *not* provide transitivity. This is demonstrated by two related examples, with the initial values of `x` and `y` both being zero:

CPU 0	CPU 1
<code>r1 = READ_ONCE(x);</code> <code>if (r1 > 0)</code> <code> WRITE_ONCE(y, 1);</code>	<code>r2 = READ_ONCE(y);</code> <code>if (r2 > 0)</code> <code> WRITE_ONCE(x, 1);</code>
<code>assert(!(r1 == 1 && r2 == 1));</code>	

The above two-CPU example will never trigger the `assert()`. However, if control dependencies guaranteed transitivity (which they do not), then adding the following CPU would guarantee a related assertion:

CPU 2
<code>WRITE_ONCE(y, 1);</code> <code>assert(!(r1 == 1 && r2 == 1 && x == 1));</code>

But because control dependencies do *not* provide transitivity, the above assertion can fail after the combined three-CPU example completes. If you need the three-CPU example to provide ordering, you will need `smp_mb()` between the loads and stores in the CPU 0 and CPU 1 code fragments, that is, just before or just after the “`if`” statements. Furthermore, the original two-CPU example is very fragile and should be avoided.

The two-CPU example is known as LB (load buffering) and the three-CPU example as WWC [MSS12].

The following list of rules summarizes the lessons of this section:

1. Compilers do not understand control dependencies, so it is your job to make sure that the compiler cannot break your code.

2. Control dependencies can order prior loads against later stores. However, they do *not* guarantee any other sort of ordering: Not prior loads against later loads, nor prior stores against later anything. If you need these other forms of ordering, use `smp_rmb()`, `smp_wmb()`, or, in the case of prior stores and later loads, `smp_mb()`.
3. If both legs of the “`if`” statement begin with identical stores to the same variable, then those stores must be ordered, either by preceding both of them with `smp_mb()` or by using `smp_store_release()` to carry out the stores. Please note that it is *not* sufficient to use `barrier()` at beginning of each leg of the “`if`” statement because, as shown by the example above, optimizing compilers can destroy the control dependency while respecting the letter of the `barrier()` law.
4. Control dependencies require at least one run-time conditional between the prior load and the subsequent store, and this conditional must involve the prior load. If the compiler is able to optimize the conditional away, it will have also optimized away the ordering. Careful use of `READ_ONCE()` and `WRITE_ONCE()` can help to preserve the needed conditional.
5. Control dependencies require that the compiler avoid reordering the dependency into nonexistence. Careful use of `READ_ONCE()`, `atomic_read()`, or `atomic64_read()` can help to preserve your control dependency.
6. Control dependencies apply only to the “`then`” and “`else`” of the “`if`” containing the control dependency, including any functions that these two clauses call. Control dependencies do *not* apply to code following the “`if`” containing the control dependency.
7. Control dependencies pair normally with other types of barriers.
8. Control dependencies do *not* provide transitivity. If you need transitivity, use `smp_mb()`.

14.2.10.6 SMP Barrier Pairing

CPU 사이의 상호작용을 다룰 때에는 특정한 타입의 메모리 배리어가 항상 짹을 맞춰 사용되어야 합니다. 올바른 짹맞추기가 없다면 거의 항상 에러가 나타날 것입니다.

쓰기 배리어는 항상 데이터 종속성 배리어나 읽기 배리어와 짹을 맞춰야 합니다만, 범용 배리어와도 가능합니다. 유사하게 읽기 배리어나 데이터 종속성 배리어는 항상 최소한 쓰기 배리어와는 짹을 맞춰 사용되어야 하며, 여기서도 마찬가지로, 범용 배리어도 가능합니다:

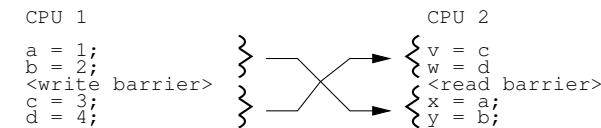
CPU 1	CPU 2
<pre>A = 1; <write barrier> B = 2;</pre>	<pre>X = B; <read barrier> Y = A;</pre>

Or:

CPU 1	CPU 2
<pre>A = 1; <write barrier> B = &A;</pre>	<pre>X = B; <data dependency barrier> Y = *X;</pre>

어떻게 해서든, 읽기 배리어는 설명 좀 더 약한 타입이라도 있어야만 합니다.⁸

쓰기 배리어 전의 스토어들은 일반적으로 읽기 배리어나 데이터 종속성 배리어 뒤의 로드와 맞춰질 것으로 기대되며, 반대도 마찬가지입니다:



14.2.10.7 Examples of Memory Barrier Pairings

첫째로, 쓰기 배리어들은 스토어 오퍼레이션들에의 부분적 순서 세우기로 동작합니다. 다음의 이벤트들을 생각해 봅시다:

```

STORE A = 1
STORE B = 2
STORE C = 3
<write barrier>
STORE D = 4
STORE E = 5

```

이 이벤트들은 Figure 14.9에서 보이듯, 시스템의 나머지 부분들이 순서없는 이벤트 집합 $\{A=1, B=2, C=3\}$ 이 순서 없는 이벤트 집합 $\{D=4, E=5\}$ 보다 먼저 일어나는 것으로 보게 되는 순서로 메모리 일관성 시스템에 들어갑니다.

둘째로, 데이터 종속성 배리어들은 데이터 종속적 로드 오퍼레이션들에의 부분적 순서세우기로 동작합니다. 초기값 $\{B = 7, X = 9, Y = 8, C = \&Y\}$ 을 갖는 다음의 이벤트들을 생각해 봅시다:

⁸ “약한”이라는 말로 의미하고자 하는 바는 “더 적은 순서 보장 사항을 제공하는”입니다. 약한 배리어는 일반적으로 또한 더 강한 배리어에 비해 적은 오버헤드를 갖습니다.

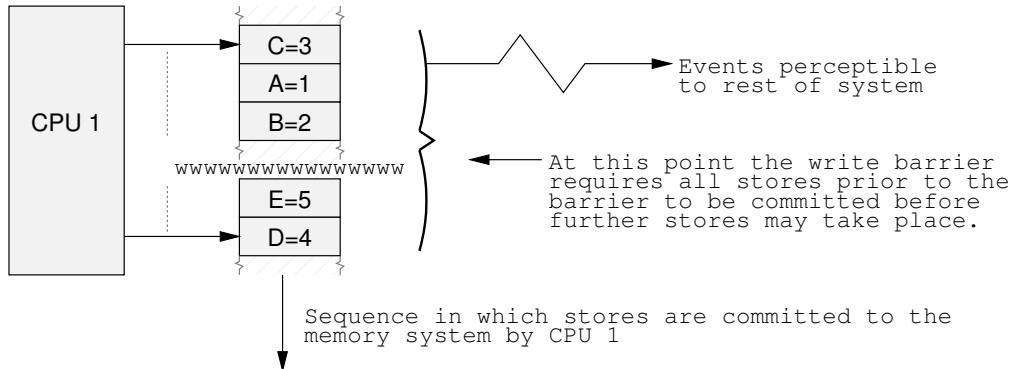


Figure 14.9: Write Barrier Ordering Semantics

CPU 1	CPU 2
$A = 1;$ $B = 2;$ <write barrier> $C = \&B;$ $D = 4;$	LOAD X LOAD C (gets $\&B$) LOAD *C (reads B)

개입이 없다면, CPU 2 는 Figure 14.10 에 보여진 것처럼, CPU 1 의 이벤트들을 CPU 1 이 사용한 쓰기 배리어에도 불구하고 어떤 효율적인 무작위적 순서로 인지하게 될 수 있습니다.

앞의 예에서, CPU 2 는 (B 가 될) *C 의 로드가 C 의 LOAD 뒤에 옴에도 불구하고 B 를 7 로 인지하게 됩니다.

하지만, 만약 CPU 2 의 C 의 로드와 *C (즉, B) 의 로드 사이에 데이터 종속성 배리어가 위치하게 된다면, 이번에도 초기값은 $\{B = 7, X = 9, Y = 8, C = \&Y\}$ 라는 가정 하에:

CPU 1	CPU 2
$A = 1;$ $B = 2;$ <write barrier> $C = \&B;$ $D = 4;$	LOAD X LOAD C (gets $\&B$) <data dependency barrier> LOAD *C (reads B)

순서는 Figure 14.11 처럼 직관적으로 예상한 대로 될 것입니다.

그리고 셋째로, 읽기 배리어는 로드 오퍼레이션들에 의 부문적 순서세우기로 동작합니다. 초기값이 $\{A = 0, B = 9\}$ 라는 가정 하에 아래의 이벤트들을 생각해 봅시다:

CPU 1	CPU 2
$A = 1;$ <write barrier> $B = 2;$	LOAD B LOAD A

개입이 없다면, CPU 1 이 수행한 쓰기 배리어에도

불구하고, Figure 14.12 에 보여지는 것처럼 CPU 2 는 CPU 1 의 이벤트들을 어떤 효율적인 무작위적 순서로 인지할 수도 있습니다.

하지만, 다시 한번 초기값이 $\{A = 0, B = 9\}$ 라는 가정 아래 만약 CPU 2 의 B 의 로드와 A 의 로드 사이에 읽기 배리어가 위치한다면:

CPU 1	CPU 2
$A = 1;$ <write barrier> $B = 2;$	LOAD B <read barrier> LOAD A

Figure 14.13 에 보이듯이, CPU 1 의 쓰기 배리어에 의해 만들어진 부분적 순서가 CPU 2 에 올바르게 인지됩니다.

이 상황을 좀 더 완벽하게 그리기 위해, 이번에도 같은 초기값 $\{A = 0, B = 9\}$ 을 갖는다는 가정 아래, 코드가 읽기 배리어 전후로 A 의 로드를 한다면 어떻게 될지 생각해 봅시다:

CPU 1	CPU 2
$A = 1;$ <write barrier> $B = 2;$	LOAD B LOAD A (1 st) <read barrier> LOAD A (2 nd)

두개의 A 의 로드들이 모두 B 의 로드 뒤에 일어나지 만, Figure 14.14 에 보이듯이 그것들은 서로 다른 값을 가져올 수 있습니다.

물론, CPU 1 의 A 에의 업데이트가 읽기 배리어 완료 전에 CPU 2 에게 인지될 수도 있는데, Figure 14.15 가 이를 보이고 있습니다.

보장되는 것은, 만약 B 의 로드가 $B == 2$ 를 내놓는다면, A 에의 두번째 로드는 항상 $A == 1$ 을 내놓을 것이라는 겁니다. A 에의 첫번째 로드에는 그런 보장이

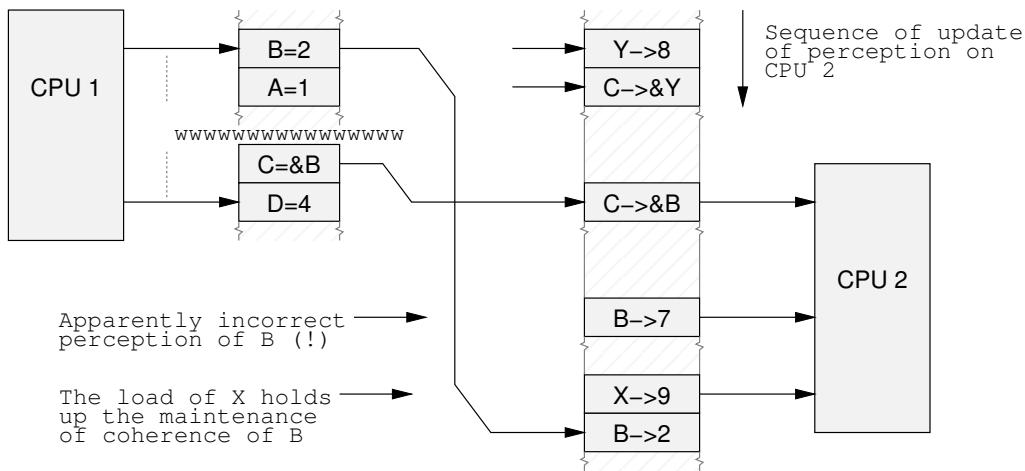


Figure 14.10: Data Dependency Barrier Omitted

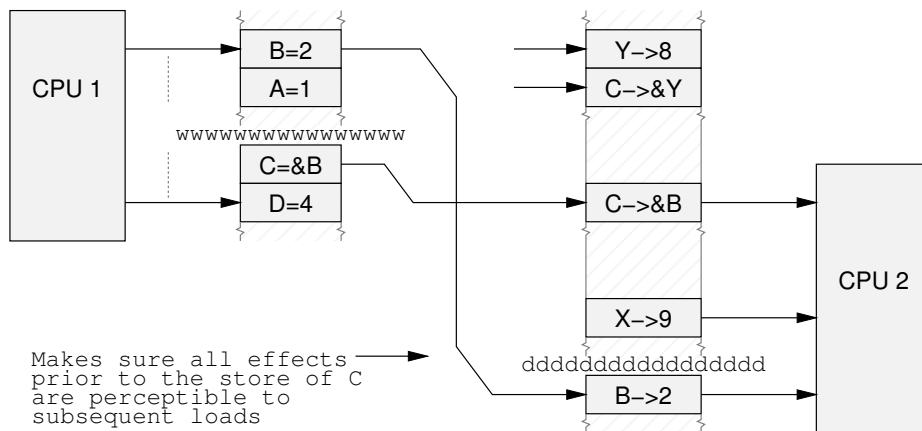


Figure 14.11: Data Dependency Barrier Supplied

없습니다; 그것은 $A == 0$ 나 $A == 1$ 중 하나를 내놓을 것입니다.

14.2.10.8 Read Memory Barriers vs. Load Speculation

많은 CPU 들이 로드 오퍼레이션들을 추측합니다: 즉, 어떤 아이템을 메모리에서 로드해야 할지, 그리고 언제 다른 로드를 위해 버스를 사용하지 않는 시간을 찾아내서는 그 로드 오퍼레이션을 미리 해버립니다 — 아직 그 인스트럭션을 실행할 차례에 도착하지 않았더라도 말이죠. 나중에, 이 동작은 실제 로드 인스트럭션이 곧 바로 완료되게 해주는데, 해당 CPU 는 이미 그 값을 쥐고 있기 때문입니다.

(브랜치로 우회되는 로드라던지로 인해) 해당 CPU

가 실제로는 그 값을 필요로 하지 않음이 밝혀질 수도 있는데, 이 경우에는 해당 값을 그냥 버리거나 나중의 사용을 위해 캐시해 둘 수 있습니다. 예를 들어, 다음과 같은 경우를 생각해 봅시다:

CPU 1	CPU 2
LOAD B	
DIVIDE	
DIVIDE	
LOAD A	

일부 CPU 들에서, 나누기 인스트럭션 (DIVIDE) 들은 완료되기까지 긴 시간을 요하는데, 따라서 CPU 2 의 버스는 그동안 아무 일도 하지 않고 있을 것입니다. 따라서, CPU 2 는 나누기가 완료되기 전에 A 의 로드를 추측적으로 행할 수도 있습니다. (원컨대) 드물게도 이 나누기들 중 하나에서 예외가 발생하는 경우, 이 추측적

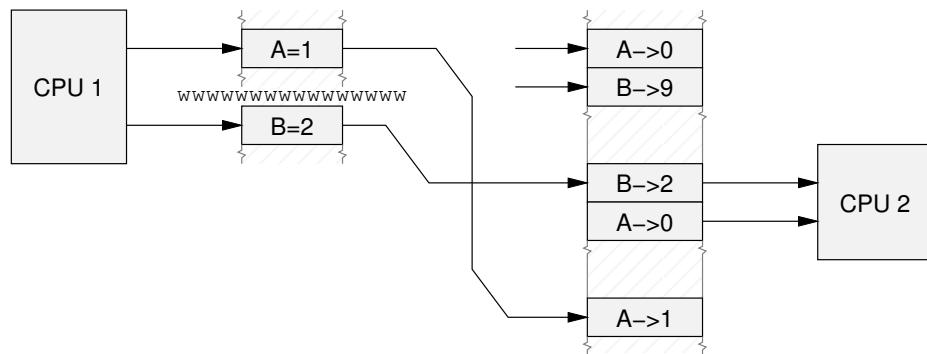


Figure 14.12: Read Barrier Needed

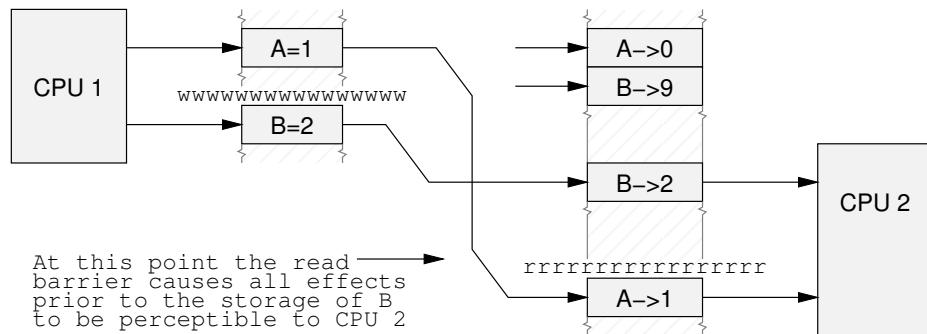


Figure 14.13: Read Barrier Supplied

로드는 쓸모없어질 것입니다만, (역시, 원컨대) 일반적인 경우에는 로드를 나누기와 겹쳐서 실행하는 행위는 해당 로드가 Figure 14.16에 보여진 것처럼 더 빨리 완료되게 해 줄 것입니다.

읽기 배리어나 데이터 의존성 배리어를 두번째 로드 앞에 놓게 된다면:

CPU 1	CPU 2
	LOAD B
	DIVIDE
	DIVIDE
	<read barrier>
	LOAD A

추측적으로 얻어진 모든 값은 사용된 배리어의 종류에 따라 다른 정도로 다시 고려될 것입니다. 해당 추측적으로 얻어진 메모리 위치에 다른 변화가 없었다면, 추측된 해당 값은 Figure 14.17에 보여진 것처럼 그냥 사용될 것입니다. 반면, 어떤 다른 CPU에서 A에 업데이트나 무효화를 행했다면, Figure 14.18에 보여진 것처럼, 추측은 취소되고 A의 값은 다시 로드될 것입니다.

14.2.11 Locking Constraints

앞서 이야기했듯, 락킹 기능들은 묵시적 메모리 배리어들을 내장하고 있습니다. 이런 묵시적 메모리 배리어들은 다음과 같은 보장사항들을 제공합니다:

1. LOCK 오퍼레이션 보장사항:

- LOCK 후에 나오는 메모리 오퍼레이션들은 해당 LOCK 오퍼레이션이 완료된 후에 완료됩니다.
- LOCK 전에 나오는 메모리 오퍼레이션들은 LOCK 오퍼레이션이 완료된 후에 완료될 수도 있습니다.

2. UNLOCK 오퍼레이션 보장사항:

- UNLOCK 전에 나온 메모리 오퍼레이션들은 해당 UNLOCK 오퍼레이션이 완료되기 전에 완료됩니다.
- UNLOCK 후에 나온 메모리 오퍼레이션들은 해당 UNLOCK 오퍼레이션이 완료되기 전에 완료될 수도 있습니다.

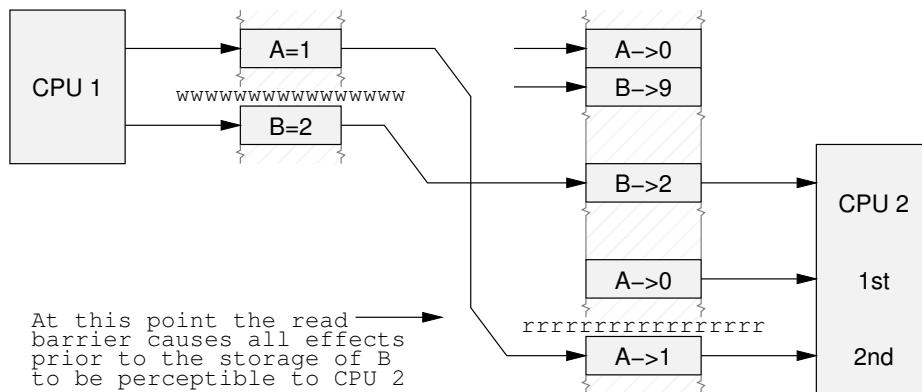


Figure 14.14: Read Barrier Supplied, Double Load

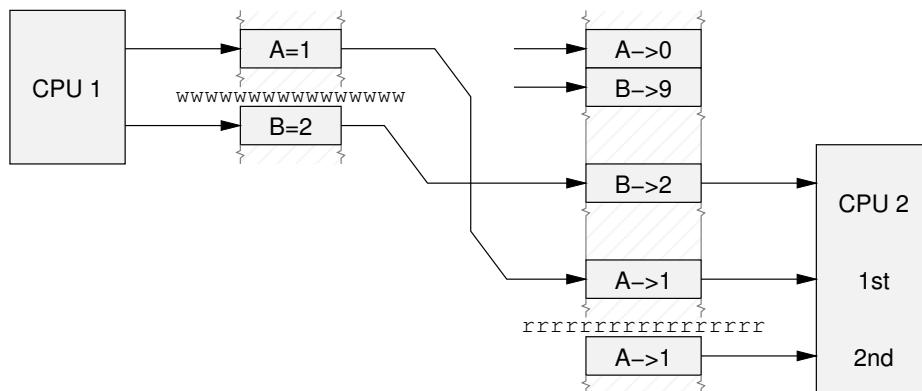


Figure 14.15: Read Barrier Supplied, Take Two

3. LOCK vs LOCK 보장사항:

- 다른 LOCK 오퍼레이션 전에 나온 LOCK 오퍼레이션들은 모두 해당 LOCK 오퍼레이션 전에 완료됩니다.

4. LOCK vs UNLOCK 보장사항:

- 하나의 UNLOCK 오퍼레이션 전에 나온 모든 LOCK 오퍼레이션들은 해당 UNLOCK 오퍼레이션 전에 완료됩니다.
- LOCK 오퍼레이션 전에 나온 모든 UNLOCK 오퍼레이션들은 해당 LOCK 오퍼레이션 전에 완료됩니다.

5. 실패한 조건적 LOCK 보장사항:

- LOCK 오퍼레이션의 일부 변종들은 실패할 수 있는데, 당장 락을 획득할 수 없어서 일 수 도 있고, 락이 획득 가능해질 때까지 기다리 며 잠들어 있는 동안 예외나 블락되지 않은

시그널을 받아서 일 수도 있습니다. 실패한 락은 어떤 배리어도 내포하지 않습니다.

14.2.12 Memory-Barrier Examples

14.2.12.1 Locking Examples

LOCK 다음 UNLOCK: LOCK 다음 UNLOCK 이 따 라오는 코드는 전체 메모리 배리어를 갖는다고 볼 수 없는데, LOCK 앞의 액세스는 LOCK 뒤에 일어날 수 있고, UNLOCK 뒤의 액세스는 UNLOCK 앞에 수행될 수 있으며, 이들은 서로를 지나갈 수 있기 때문입니다. 예를 들어, 다음의 코드는:

```

1 *A = a;
2 LOCK
3 UNLOCK
4 *B = b;

```

다음의 순서로 실행될 수 있습니다:

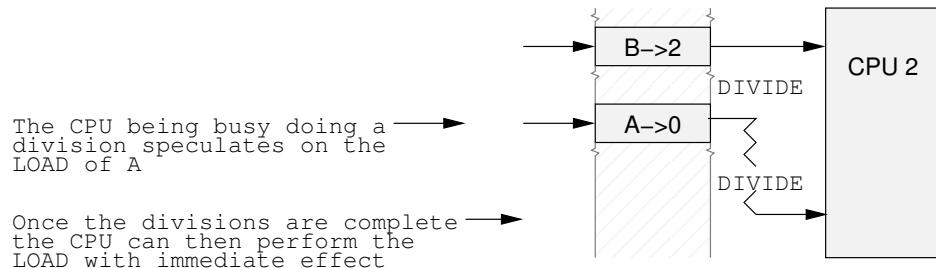


Figure 14.16: Speculative Load

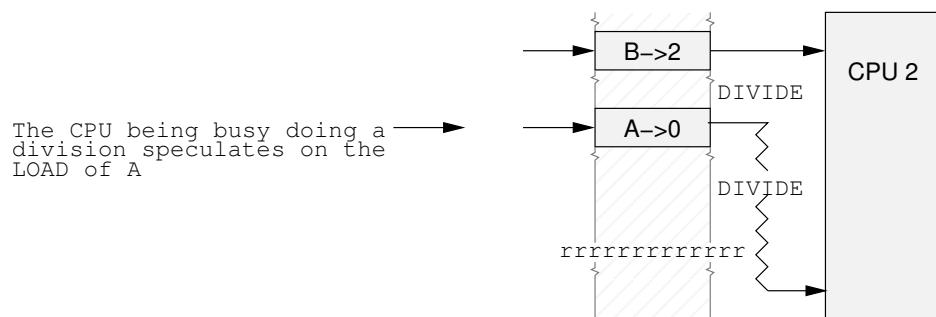


Figure 14.17: Speculative Load and Barrier

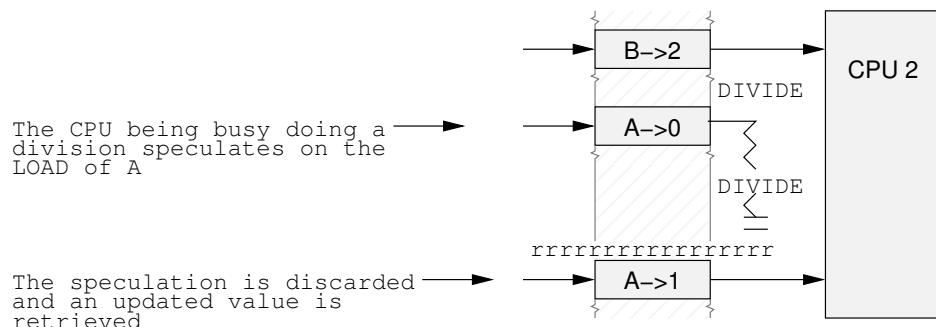


Figure 14.18: Speculative Load Cancelled by Barrier

```

2 LOCK
4 *B = b;
1 *A = a;
3 UNLOCK

```

다시 말하지만, LOCK 과 UNLOCK 은 각각 앞의 오퍼레이션과 뒤의 오퍼레이션을 크리티컬 섹션에 “홀려 들어올” 수 있게 허용함을 항상 기억하시기 바랍니다.

Quick Quiz 14.13: LOCK-UNLOCK 오퍼레이션들을 어떻게 조합해야 전체 메모리 배리어처럼 동작할까요? ■

Quick Quiz 14.14: (만약 있다면) 어떤 CPU 들이 이런 투과성의 락킹 기능들을 만들 수 있는 메모리 배리어

인스트럭션들을 가지고 있을까요? ■

LOCK 기반의 크리티컬 섹션들: LOCK-UNLOCK 짹은 전체 메모리 배리어로 동작하지 않지만, 이 오퍼레이션들은 메모리 순서에 영향을 끼칠 수 있습니다.

다음의 코드를 생각해 봅시다:

```

1 *A = a;
2 *B = b;
3 LOCK
4 *C = c;
5 *D = d;
6 UNLOCK
7 *E = e;
8 *F = f;

```

이것은 다음과 같이 합법적인 순서로 실행될 수 있을 것으로, 같은 줄의 오퍼레이션들은 CPU 가 해당 오퍼레이션들을 동시적으로 수행함을 의미합니다:

```

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; } *E;

Table 14.3: Lock-Based Critical Sections

Quick Quiz 14.15: Table 14.3에서 중괄호로 그룹지어진 오퍼레이션들은 동시에 수행된다고 보면, 표의 어떤 열들이 “A”에서 “F”까지의 변수들과 LOCK/UNLOCK 오퍼레이션들의 합법적인 재배치일까요? (코드의 순서는 A, B, LOCK, C, D, UNLOCK, E, F입니다.) 합법이면 왜 합법이고 아니라면 왜 아니죠? ■

여러 락들을 사용한 순서: 여러 락들을 포함하고 있는 코드는 여전히 그 락들로부터의 순서 제약을 바라보게 됩니다만, 어떤 락이 어떤 락인지 주의 깊게 신경써야 합니다. 예를 들어, Table 14.4에 있는, “M”과 “Q”라 이름붙여진 한쌍의 락을 사용하는 코드를 보기 바랍니다.

CPU 1	CPU 2
A = a;	E = e;
LOCK M;	LOCK Q;
B = b;	F = f;
C = c;	G = g;
UNLOCK M;	UNLOCK Q;
D = d;	H = h;

Table 14.4: Ordering With Multiple Locks

이 예에서는, 앞의 섹션에서 이야기했듯 해당 락들이 각자에게 주는 제약 외에는, 변수 “A”부터 “H”까지의 값 할당이 어떤 순서로 이뤄질지에 대해서는 아무런 보장이 없습니다.

Quick Quiz 14.16: Table 14.4에서의 제약은 뭐죠? ■

하나의 락을 사용하는 여러 CPU들의 순서: Table 14.4처럼 두개의 서로 다른 락들이 아니라, Table 14.5처럼 두개의 CPU들이 같은 락을 획득하는 경우를 생각해 봅시다.

CPU 1	CPU 2
A = a;	E = e;
LOCK M;	LOCK M;
B = b;	F = f;
C = c;	G = g;
UNLOCK M;	UNLOCK M;
D = d;	H = h;

Table 14.5: Ordering With Multiple CPUs on One Lock

이 경우, CPU 1은 CPU 2보다 먼저 M을 획득하거나, 반대일 수 있습니다. 첫번째 경우, A, B, C에의 값 할당은 F, G, H에의 값 할당보다 먼저 이루어질 것입니다. 반대의 경우, 만약 CPU 2가 해당 락을 먼저 획득하면, E, F, G에의 값 할당이 B, C, D보다 먼저 이루어질 것입니다.

14.2.13 The Effects of the CPU Cache

지각되는 메모리 오퍼레이션들의 순서는 CPU들과 메모리 사이에 위치하는 캐시들에게 메모리 일관성과 순서를 관리하는 캐시 일관성 프로토콜에 의해 영향을 받습니다. 소프트웨어 관점에서는, 이런 캐시들은 그저 메모리로 보여집니다. 메모리 배리어들은 Figure 14.19의 점선 위치에서 CPU가 자신의 값을 메모리에 올바른 순서로 제공하도록 보장하고, 다른 CPU들이 만든 변화를 올바른 순서로 볼 수 있도록 보장하도록 동작하는 것으로 생각될 수 있습니다.

이 캐시들은 한 CPU의 메모리 액세스를 시스템의 나머지 것들로부터 “숨길” 수 있지만, 캐시 일관성 프로토콜은 캐시라인들을 필요시 옮기고 무효화 시켜가며 모든 다른 CPU들이 이 숨겨진 액세스들의 효과를 볼 수 있도록 보장합니다. 또한, CPU 코어는 프로그램 인과성과 메모리 순서가 관리됨을 지킨다는 제약 아래에서는 인스트럭션들을 어떤 순서로든 실행할 수 있습니다. 이런 인스트럭션들 중 일부는 해당 CPU의 메모리 액세스 큐에 들어가는 메모리 액세스들을 생성할 수도 있습니다만, 그 실행은 CPU가 내부 자원을 모두 채우기 전까지는 계속되지 못할 것이며, 그렇지 않은 경우 면제 큐에 들어온 다른 메모리 액세스가 완료되기 전까지는 기다려야만 합니다.

14.2.13.1 Cache Coherency

캐시 일관성 프로토콜은 한 CPU가 자신의 액세스들을 순서대로 볼 수 있고, 모든 CPU들이 하나의 캐시라인에 담긴 하나의 변수에의 수정의 순서에 동의하도록

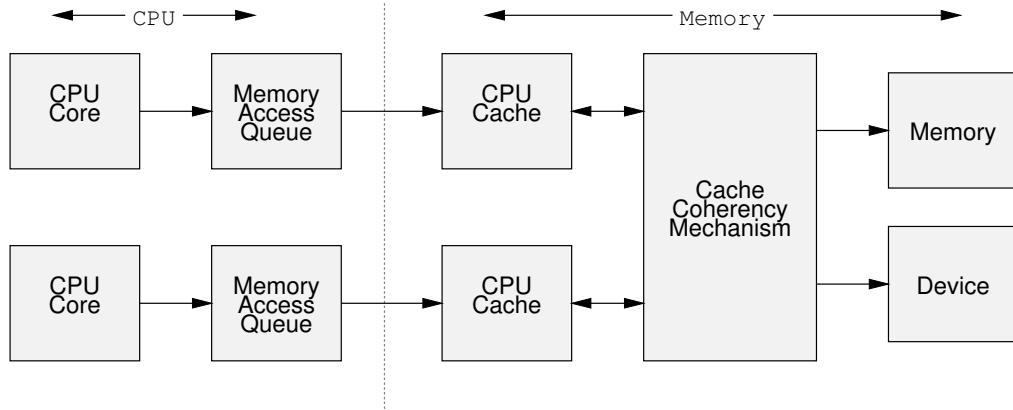


Figure 14.19: Memory Architecture

하지만, 다른 변수에의 설정들이 모든 CPU 들에 같은 순서로 보일 것이라는 보장은 하지 않습니다 — 일부 컴퓨터 시스템들은 그런 보장을 하지만, 이식성 있는 소프트웨어는 그런 사실에 기반해선 안됩니다.

왜 재배치가 일어날 수 있는지 알아보기 위해, Figure 14.20에 보인 것과 같이, 두개의 CPU 를 가지며 각 CPU 는 분할된 캐시를 갖는 시스템을 생각해 봅시다. 이 시스템은 다음과 같은 특성을 갖습니다:

1. 홀수번 캐시 라인은 캐시 A, 캐시 C, 메모리, 또는 이것들의 조합에 존재할 수 있습니다.
2. 짝수번 캐시 라인은 캐시 B, 캐시 D, 메모리, 또는 이것들의 조합에 존재할 수 있습니다.
3. 한 CPU 코어가 자신의 캐시 중 하나에게 질의를 던지는 동안,⁹ 다른 캐시가 가만히 있어야 할 필요는 없습니다. 이 다른 캐시는 그대신 무효화 요청에 응답하거나 dirty 캐시 라인을 메모리에 write back하거나 CPU 의 메모리 액세스 큐의 원소들을 처리하거나 그 외에도 이것 저것 할 수 있습니다.
4. 각각의 캐시는 요청된 일관성과 순서 규칙을 유지하기 위해 해당 캐시에 적용되어야 할 오퍼레이션들을 담는 큐를 갖습니다.
5. 이 큐들은 이 큐들의 원소들에 의해 영향 받는 캐시 라인의 로드와 스토어에 의해 비워져야 할 필요는 없습니다.

요약하자면, 만약 캐시 A 가 바쁘지만 캐시 B 는 할 일이 없다면, CPU 1 의 홀수번 캐시라인으로의 스토어는 CPU 2 의 짝수번 캐시라인으로의 스토어에 비해

늦게 처리될 수 있습니다. 너무 극단적인 경우가 아니라면, CPU 2 는 CPU 1 의 오퍼레이션들을 비순차적으로 보게 될 것입니다.

하드웨어와 소프트웨어에서의 메모리 순서에 대해서 많은 자세한 내용을 위해선 Appendix B 를 참고하세요.

14.2.14 Where Are Memory Barriers Needed?

메모리 배리어들은 두 CPU 들 간, 또는 CPU 와 디바이스 간 상호작용이 있을 수 있을 때에만 필요합니다. 만약 코드의 어떤 곳에도 그런 상호작용이 없음이 보장된다면, 메모리 배리어들은 그 코드에는 필요치 않습니다.

이것들은 최소한의 보장이란 점을 유의해 두십시오. 다른 아키텍쳐들은 Appendix B 에서 이야기했듯, 더 많은 보장사항들을 제공할 것입니다만, 해당 아키텍쳐에서만 동작하도록 특별히 설계된 코드 외의 부분에서는 그런 것들을 기대해서는 안됩니다.

하지만, 어토믹 오퍼레이션들을 구현하는, 락킹 기능이나 어토믹 자료구조 조작과 순회 기능들은 보통 각자의 정의 내에 필요한 모든 메모리 배리어들을 포함하고 있습니다. 하지만, 리눅스 커널의 `atomic_inc()` 같은 예외가 존재하고, 그리고 어쩌면 당신의 소프트웨어 환경의 실제 구현에도 예외가 존재할 수도 있으므로, 문서를 반드시 숙지하기 바랍니다.

마지막 조언: 메모리 배리어 기능을 직접 사용하는 것은 마지막 수단이 되어야만 합니다. 대부분의 경우에는 메모리 배리어를 알아서 사용하는, 이미 존재하는 도구들이 더 낫습니다.

⁹ 하지만 “슈퍼스칼라” 시스템에서는, CPU 는 한번에 두 캐시에 대해 각 캐시의 절반씩을 액세스 할 수도 있으며, 이 두 캐시의 각 절반씩에 여러개의 동시적 액세스를 수행할 수도 있습니다.

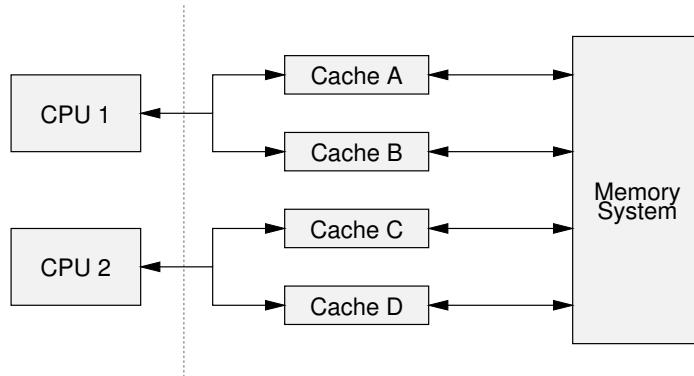


Figure 14.20: Split Caches

14.3 Non-Blocking Synchronization

non-blocking synchronization (NBS)라는 용어는 진행 보장사항 (forward-progress guarantee)들을 달리 하는, 6개의 linearizable 알고리즘 클래스들을 말합니다. 이 진행 보장사항들은 리얼타임 프로그래밍 (Real-time programming)의 근간을 형성하는 그것들과는 직교적입니다:

1. 리얼타임 진행 보장사항들은 그것들과 관련된 분명한 시간을 갖는데, 예를 들어, “스케줄링 대기시간은 100 마이크로세컨드 보다 작아야 한다.” 같은 것입니다. 반면, 가장 인기있는 형태의 NBS는 분명한 최대 제한 없이 유한한 시간 내에 진행이 이뤄질 것만을 보장합니다.
2. 리얼타임 진행 보장사항들은 간혹 확률적인데, 소프트 리얼타임 보장사항은 “최소 99.9%의 경우 스케줄링 대기시간은 100 마이크로세컨드 이내여야 한다.”와 같은 식입니다. 반면, NBS의 진행 보장사항은 전통적으로 무조건적입니다.
3. 리얼타임 진행 보장사항은 종종 환경 제약에 조건적인데, 예를 들어, 각 CPU가 최소한 어떤 특정한 양의 시간을 아무일도 하지 않고 보내거나, I/O의 비율이 특정 최대치 미만인 경우 가장 높은 우선순위의 작업들만이 그 보장을 받게 됩니다. 반면, NBS의 진행 보장사항은 보통 무조건적입니다.¹⁰
4. 리얼타임 진행 보장사항은 보통 소프트웨어 버그가 없을 때에만 적용됩니다. 반면, 대부분의 NBS

보장사항은 멈춰버리는 버그가 있더라도 적용 가능합니다.¹¹

5. NBS 진행 보장사항은 linearizability를 내포합니다. 반면, 리얼타임 진행 보장사항은 linearizability와 같은 순서 제약과 무관합니다.

이런 차이에도 불구하고, 여러 NBS 알고리즘들은 리얼타임 프로그램에 상당히 유용합니다.

현재 NBS 계층 [ACHS13]에는 7개의 단계가 있는데, 간략히 설명하자면 다음과 같습니다:

1. *Bounded wait-free synchronization*: 모든 쓰레드는 특정한 유한한 시간 내에 진행을 만들어낸다 [Her91]. (이 레벨은 대부분의 사람들로부터 불가능할 것으로 여겨지는데, Alitarh 등 [ACHS13]이 이내지 못한 이유일 수도 있습니다.)
2. *Wait-free synchronization*: 모든 쓰레드는 유한한 시간 내에 진행을 만들어낸다 [Her93].
3. *Lock-free synchronization*: 최소한 한 쓰레드는 유한한 시간 내에 진행을 만들어낸다 [Her93].
4. *Obstruction-free synchronization*: 경쟁이 없다면 모든 쓰레드가 유한한 시간 내에 진행을 만들어낸다 [HLM03].
5. *Clash-free synchronization*: 경쟁이 없다면 최소한 하나의 쓰레드는 유한한 시간 내에 진행을 만들어낸다 [ACHS13].
6. *Starvation-free synchronization*: 실패가 없다면 모든 쓰레드가 유한한 시간 내에 진행을 만들어낸다 [ACHS13].

¹⁰ 아래에서 보게 되겠지만, 최근의 일부 NBS는 이 보장사항을 완화시켰습니다.

¹¹ 다시 말하지만, 최근의 일부 NBS는 이 보장사항을 완화했습니다.

7. *Deadlock-free synchronization*: 실패가 없다면 적어도 하나의 쓰레드는 유한한 시간 내에 진행을 만들 어낸다 [ACHS13].

NBS 클래스 1, 2 그리고 3은 1990년대 초, 클래스 4는 2000년대 초, 그리고 클래스 5는 2013년에 처음 입안되었습니다. 마지막 두개의 클래스들은 수십년동안 비공식적으로 사용되어왔습니다만 2013년에 들어 다시 입안되었습니다.

이론적으로는 어떤 병렬 알고리즘도 wait-free 형태로 변형될 수 있습니다만, 흔히 사용되는 NBS 알고리즘들의 부분집합은 상대적으로 작은 편입니다. 이것들 중 일부가 다음 섹션에서 설명됩니다.

14.3.1 Simple NBS

아마도 가장 단순한 NBS 알고리즘은 `fetch-and-add(atomic_add_return())` 기능을 이용한 정수 카운터의 어토믹한 업데이트일 것입니다.

또다른 간단한 NBS 알고리즘은 한 배열 안의 정수들의 집합입니다. 여기서 배열 인덱스는 해당 집합의 멤버일 값을 가리키고 배열의 원소들은 해당 값이 실제로 집합의 멤버인지 아닌지를 알립니다. NBS 알고리즘을 위한 Linearizability 규범은 이 배열에의 읽기와 쓰기가 어토믹 인스트럭션을 사용하거나 메모리 배리어를 수반할 것을 요구합니다만 linearizability가 그다지 중요하지 않은, 그렇게 희귀하지도 않은 경우들에서는 간단한 `volatile` 로드와 스토어만으로도 충분한데, `ACCESS_ONCE()`의 사용이 한 예일 것입니다.

NBS 집합은 비트맵을 이용해 구현될 수도 있을 텐데, 해당 집합의 멤버인 값은 하나의 비트와 연관될 것입니다. 읽기와 업데이트는 일반적으로 어토믹한 비트 조작 인스트럭션들로 이루어져야만 합니다만 `compare-and-swap` (`cmpxchg()` 또는 CAS) 인스트럭션 또한 사용될 수 있습니다.

Section 5.2에서 이야기 되었던 통계적 카운터 알고리즘은 그 합이 정확하지는 않고 대략적이라는 분명한 트릭을 사용할 때라면 wait-free로 간주될 수 있습니다.¹² `read_count()` 함수가 카운터들의 합을 구하는 데 갖는 시간의 길이의 기능인 충분히 큰 에러 한계들을 감안하면, 어떤 linearizable하지 않은 동작이 일어났음을 증명하는 것은 불가능합니다. 이는 분명히 (약간 인위적이라면) 이 통계적 카운터 알고리즘을 wait-free로 분류합니다. 이 알고리즘은 아마도 리눅스 커널에서 가장 많이 사용하는 NBS 알고리즘입니다.

또다른 흔한 NBS 알고리즘은 하나의 어토믹한 큐로, 원소들의 추가는 어토믹한 교체 인스트럭션 [MS98b]

```

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 14.21: NBS Enqueue Algorithm

을 사용하고, 이어서 새 원소의 `->next` 포인터를 저장하는데, 이는 유저스페이스-RCU 라이브러리 구현 [Des09] 인 Figure 14.21에 나와 있습니다. Line 9는 `tail` 포인터가 새 원소를 가리키도록 업데이트하면서 그 앞의 것으로의 레퍼런스를 리턴하는데, 이 값은 로컬 변수 `old_tail`에 저장됩니다. Line 10은 이제 앞의 원소의 `->next` 포인터가 새로 추가된 원소를 가리키도록 업데이트 하며, 마지막으로 line 11에서 해당 큐가 원래 비어있었는지 여부를 리턴합니다.

하나의 원소를 꺼내기 위해선 상호 배타성이 필요하지만 (따라서 꺼내기 작업은 블락됩니다), 큐의 전체 컨텐츠를 제거하는 일은 블락킹하지 않게도 할 수 있습니다. 불가능한 것은 어떤 주어진 원소를 블락킹하지 않는 방법으로 꺼내는 것입니다: 원소를 넣는 쪽은 line 9와 10에서 실패할 것이고, 따라서 요청된 원소는 부분적으로만 추가되었을 것입니다. 이로 인해 원소를 집어넣는 작업은 NBS지만 꺼내는 것은 블락킹되는 반면 NBS인 알고리즘으로 귀결됩니다. 이 알고리즘은 실제 상황에서 사용되지는 않는데, 대부분의 상품화된 소프트웨어는 임의의 fail-stop 에러들을 견뎌내도록 요구되지는 않기 때문입니다.

14.3.2 NBS Discussion

완전히 블락킹 없는 큐 [MS96]를 만드는 건 가능합니다만, 그런 큐들은 앞의 반만 NBS인 알고리즘에 비해 훨씬 복잡합니다. 여기서 얻을 수 있는 교훈은 당신에게 정말로 요구되는 것이 무엇인지 주의깊게 생각해 봐야 한다는 것입니다. 무의미한 요구사항을 완화시키는 것은 간단성과 성능에서 커다란 개선을 가져올 수 있습니다.

최근의 연구는 요구사항들을 완화시키기 위한 또다

¹² 인용이 필요합니다. 전 이 트릭을 Mark Moir로부터 구두로 들었습니다.

른 중요한 방법을 이야기합니다. 공정한 (fair) 스케줄링을 제공하는 시스템들은 wait-free 동기화의 장점 대부분을 심지어 블락킹 하지 않는 동기화만 제공하는 알고리즘을 사용할 때에도 얻을 수 있다고 이론 [ACHS13]과 실제 [AB13]에서 모두 이야기 합니다. 상품화된 단계에서 사용되는 매우 많은 스케줄러들이 실제로 공정성 (fairness)을 제공하므로, wait-free 동기화를 제공하는 더 복잡한 알고리즘들은 일반적으로 더 간단하고 많은 경우 더 빠른 블락킹하지 않는 동기화 알고리즘들에 비해 실제 환경에서 더 나은 점을 제공하지 못하곤 합니다.

흥미롭게도, 공정한 스케줄링은 실제 상황에서 고려되는 이익적 제약들 중 하나일 뿐입니다. 다른 제약의 집합들은 블락킹 알고리즘들이 결정론적 리얼타임 반응을 달성할 수 있게 할 수 있습니다. 예를 들어, 그 획득이 주어진 우선순위에 따라 FIFO 순서로 이루어지게 하는 공정한 (fair) 락, (우선순위 상속 [TS95, WTS96]이나 우선순위 한도 문제와 같은) 우선순위 역전 문제를 회피하는 방법, 제한된 숫자의 쓰레드들, 제한된 크리티컬 섹션들, 제한된 부하량, 그리고 fail-stop 베그들의 회피가 주어진다면, 락 기반의 어플리케이션들은 결정론적 반응 시간을 제공할 수 있습니다 [Bra11]. 물론 이런 시도는 블락킹과 wait-free 동기화 사이의 차이점을 흐리게 만드는데, 좋은 현상입니다. 이론적인 뼈대가 자랄수록, 소프트웨어가 실제로 현장에서 어떻게 구성되는지 설명하는 능력도 증가할 것입니다.

Chapter 15

The difference between you and me is that I was right in time.

Konrad Adenauer

Parallel Real-Time Computing

컴퓨팅에서 중요한 발전중인 영역은 병렬 리얼타임 컴퓨팅입니다. Section 15.1은 “리얼타임 컴퓨팅”에 대해서 흔한 발언들부터 도 의미있는 영역들까지의 다양한 정의를 알아봅니다. Section 15.2은 리얼타임 응답이 필요한 어플리케이션의 종류들에 대해서 알아봅니다. Section 15.3에서는 병렬 리얼타임 컴퓨팅이 우리에게 다가와 있음을 이야기하고, 언제, 그리고 왜 병렬 리얼타임 컴퓨팅이 유용하게 되는지에 대해서 이야기 합니다. Section 15.4에서는 어떻게 병렬 리얼타임 시스템들이 구현될 것인가에 대한 간략한 그림을 제공하며, 마지막으로 Section 15.5에서는 여러분의 어플리케이션이 리얼타임 장비들을 필요로 하는지에 대한 결정을 어떻게 해야 하는지 간단히 설명합니다.

15.1 What is Real-Time Computing?

리얼타임 컴퓨팅을 분류하는 한가지 전통적인 방법은 *hard real time*과 *soft real time*의 카테고리로 나누는 것으로, 여기서 마초같은 *hard real-time* 어플리케이션들은 데드라인을 절대로 어기지 않지만, 약한 *soft real-time* 어플리케이션들은 데드라인을 자주, 그리고 종종 어길 수 있습니다.

15.1.1 Soft Real Time

*Soft real time*에 대한 이 정의에 있어서 문제를 알아보기는 쉽습니다. 한가지 들어보면, 그 정의에 의해서, 모든 소프트웨어의 부분은 *soft real-time* 어플리케이션이라고 말할 수 있습니다: “제 어플리케이션은 백만개 점의 휴리에 변환을 0.5 피코세컨드 만에 계산합니다.” “말도 안돼요!!! 이 시스템의 클락 사이클은 300 피코세컨드보다 크다구요!” “아, 하지만 이건 *soft real-time* 어플리케이션이잖아요!” “*Soft real time*”이라는 용어가

어떻게든 사용된다면, 어떤 제한점이 분명하게 요구됩니다.

따라서 우리는 주어진 *soft real-time* 어플리케이션은 그 응답시간 요구사항이 최소한 얼마간의 시간 내여야만 한다고 말할 수 있을텐데, 예를 들어 우린 그 프로그램이 99.9%의 경우는 20 마이크로세컨드 내에 수행되어야만 한다고 말할 수 있을겁니다.

물론 이는 그 어플리케이션이 그 자신의 응답시간 요구사항을 지키지 못했을 경우에는 어떤 일이 벌어지는지에 대한 의문을 생기게 만들습니다. 그에 대한 대답은 어플리케이션에 따라 다양할 수 있습니다만, 한가지 가능한 답은 제어되고 있는 그 시스템은 가끔의 느린 응답 동작을 문제없이 처리할 수 있을만큼 충분한 안정성과 관성을 갖는다는 것입니다. 또 다른 가능한 답은 해당 어플리케이션이 그 결과를 계산해내는데에 두가지 방법을 가지고 있어서, 하나는 빠르고 결정론적이지만 부정확한 방법이고 또 다른 하나는 매우 정확하지만 예측 불가능한 계산시간을 갖는 것입니다. 합리적인 한가지 방법은 두개지 방법을 병렬적으로 시작하고 정밀한 방법이 시간내에 완료되는데에 실패한다면, 그 수행을 중단하고 빠르지만 부정확한 방법의 결과를 답변으로 사용하는 것입니다. 빠르지만 부정확한 방법으로 사용될 수 있는 한가지는 현재 시간 간격 동안 제어 동작을 취하지 않는 것이고, 또 다른 하나는 앞의 시간 간격동안 취해진 것과 똑같은 제어 동작을 취하는 것입니다.

짧게 말해서, 그것이 정확히 얼마나 *soft* 한지에 대한 어떤 측정을 제공하지 않은 채로는 *soft real time*에 대해서 이야기 하는 것은 의미가 없습니다.

15.1.2 Hard Real Time

대조적으로, *hard real time*의 정의는 상당히 명확합니다. 무엇보다, 주어진 시스템은 항상 그 데드라인을 지키거나 그렇지 않거나 둘 중 하나입니다. 불행하게도, 이 정의의 엄격한 응용은 어떤 *hard real-time* 쓰템도 존재할 수 없을 것을 의미합니다. 이에 대한 이유가 Fig-



Figure 15.1: Real-Time Response Guarantee, Meet Hammer

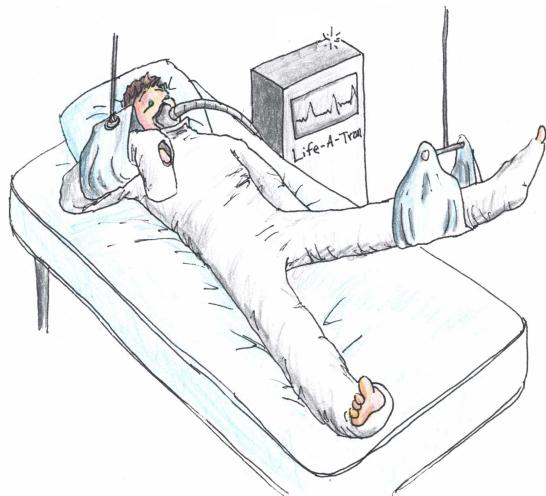


Figure 15.2: Real-Time Response: Hardware Matters

Figure 15.1에 그려져 있습니다. 추가된 여분의 것을 포함해에서도 더 튼튼한 시스템을 만들 수 있는게 사실입니다. 하지만 제가 항상 더 큰 망치를 가지고 올 수 있는 것도 사실이지요.

그렇다면 다시, 단순한 하드웨어 문제가 아니라 커다란 쇠로 된 하드웨어 문제가 분명한 데에서 소프트웨어를 문제시 하는건 공평치 않은 일일 수 있습니다.¹ 이는 우리가 hard real-time 소프트웨어를 하드웨어의 고장이 없을 경우에 한정해서 항상 데드라인을 지키는 소프트웨어로 정의할 것을 제안합니다. 불행히도, Figure 15.2

¹ 또는, 근대의 망치를 놓고 보면, 커다란 강철 문제.

에 보였듯이 고장이 항상 선택적일 수는 없습니다. 해당 그림에 그려진 불쌍한 신사가 우리가 “당신의 비극적인 죽음이 데드라인을 놓쳐서 초래된다면, 그건 소프트웨어 문제로 인한 것은 아닐 것이니 안심하십시오!”라고 말하는데에 안심할 수는 없을 것임을 간단히 예상할 수 있습니다. Hard real-time 응답은 전체 시스템의 속성인지, 소프트웨어의 것만이 아닙니다.

하지만 우리가 완벽을 기할 수는 없다면, 앞서 언급한 soft real-time 방법에서와 비슷하게 공지를 해줄 수는 있을 겁니다. 그런 경우 Figure 15.2의 Life-a-Tron은 데드라인을 놓치게 될 것 같다면 병원 스템에게 알림을 보낼 수 있을 겁니다.

안타깝게도, 이 방법은 Figure 15.3에 보인 것과 같이 하찮은 해결책을 가지고 있습니다. 항상 즉각적으로 그 데드라인을 지킬 수 없을 것 같다고 알림을 내놓는 시스템은 법적으로는 문제가 없지만, 전혀 쓸모 없습니다. 해당 시스템이 전체 시간중 어느 정도의 부분에 있어서는 데드라인을 지킨다거나, 또는 특정한 횟수의 연속된 오퍼레이션들에 대해서는 데드라인을 놓치지 않는다는 요구사항이 분명히 필요합니다.

Hard real time에도 soft real time에도 짧은 말로 설명되는 방법은 취할 수가 없습니다. 따라서 다음 섹션은 더 현실적인 접근방법을 취합니다.

15.1.3 Real-World Real Time

“Hard real-time 시스템들은 항상 데드라인을 지킵니다!” 와 같은 문장은 사람의 마음을 끌 수 있고 기억하기 쉬움에 의심의 여지가 없지만, 실제 세계의 real-time 시스템들을 위해선 뭔가 다른 것들이 필요합니다. 그렇게 이끌어진 명제는 기억하기에 더 어려울 수 있지만, 환경, 워크로드, 그리고 real-time 어플리케이션 자체에 대한 제약을 이끌어냄으로써 real-time 시스템의 구성을 간단화 시킬 수 있습니다.

15.1.3.1 Environmental Constraints

환경에의 제약은 “hard real time”에 의해 상정되는 응답시간에 대한 끝없는 약속에의 반론을 제기합니다. 이 제약은 허용 가능한 운영 온도, 공기 청정도, 전자기파 방사량의 수준과 타입, 그리고, Figure 15.1의 요점에 맞춰서, 충격과 진동의 수준을 명시할 수도 있습니다.

물론, 일부 제약들은 다른 것들보다 지켜지기가 더 쉽습니다. 수많은 사람들이 생필품처럼 팔리는 컴퓨터 컴퓨터들이 종종 얼음이 얼만한 온도에서는 동작하지 못한다는 점을 알고 있는데, 이는 일부 온도 조절 필요성을 제시합니다.

옛 대학교 친구 한명이 한번은 다소 격렬한 염소 화합물을 가진 대기에서 real-time 시스템을 운영해야 하는



Figure 15.3: Real-Time Response: Notification Insufficient

어려움을 겪은 적이 있는데, 그는 이 어려움을 지혜롭게도 하드웨어 설계를 하게 되는 그의 학교 친구들에게 넘겨버렸습니다. 구체적으로, 제 대학교 친구는 컴퓨터를 곧바로 둘러싸는 환경에 대한 대기질 상의 화합물에 대한, 하드웨어 설계자들이 물리적 봉인을 사용함으로써 자주하게 되는 제약을 부과했습니다.

또 다른 옛 대학교 친구는 진공에서 강한 흐름을 그리며 티타늄 주괴를 투겨대는 컴퓨터로 제어되는 시스템을 가지고 작업을 했습니다. 시간에 따라, 그 흐름은 티타늄의 주괴가 그리는 경로가 지겹다고 생각하고는 땅쪽으로 떨어지는 더 짧고 재미있는 경로를 선택할 수 있었습니다. 우리가 우리의 물리 수업 시간에 배웠듯이, 전자의 흐름의 급작스런 변화는 전자기파를 생성하고, 더 커다란 흐름에서의 더 커다란 변화는 강한 힘의 전자기파를 생성합니다. 그리고 이 경우에 초래된 전자기파동은 400 미터 멀리 떨어진 작은 “러버덕 같은” 안테나의 전위의 4분의 1을 유발할 수 있기에 충분했습니다. 이는 역제곱 법칙에 의해 근처의 전도체들은 큰 전압을 볼 수 있음을 의미합니다. 이는 이 투겨대기 작업을 제어하는 컴퓨터를 구성하는 전도체를 포함했습니다. 구체적으로, 그 컴퓨터의 리셋 전선에 가해진 전압은 정말로 컴퓨터를 리셋시키기에, 관련된 모두를 놀래키기에 충분했습니다. 이 경우, 이 어려움은 또한 하드웨어로 처리되었는데, 정교한 봉인과 9600 baud라는 제가 처음 들어볼 만큼 낮은 bitrate의 광섬유 네트워크를 사용했습니다. 그렇다면 하나, 덜 화려한 전자기파 환경은 에러 감지와 수정 코드를 사용하는 소프트웨어를 통해 제어될 수 있습니다. 그렇다면 하나, 에러 감지와 수정 코드가 실패 확률을 줄일 수는 있지만, 그것들은 실패

확률을 아예 없애버릴 수는 없으므로 hard real-time 응답을 달성하는데에 또 다른 문제를 형성하게 됨을 기억해 두는 것이 중요합니다.

또한 최소한의 전력 수준이 필요시 되는 환경도 존재하는데, 예를 들면, 시스템을 이끄는 전력과 시스템이 모니터되거나 제어되려는 바깥 세계와 통신하게 되는 기기의 것에의 전력이 되겠습니다.

Quick Quiz 15.1: 배터리로 동작하는 시스템의 경우에는 어떻죠? 그들은 그 시스템으로 흘러들어오는 에너지를 전혀 필요로 하지 않아요. ■

많은 수의 시스템들이 상당한 수준의 충격과 진동에도 동작할 수 있도록 만들어지는데, 예를 들면 엔진 제어 시스템이 그렇습니다. 지속적인 진동과 급작스런 충격의 경우로 와보면 더 가혹한 요구사항이 발견될 수도 있습니다. 예를 들어, 제 학부시절의 연구중에, 저는 구식의 Athena 탄도학 컴퓨터를 보게 되었는데, 그 컴퓨터는 수류탄이 근처에 떨어지더라도 평범하게 동작을 계속할 수 있도록 설계되었습니다.² 그리고 마침내, 비행기에 사용되는 “블랙 박스”들은 폭발 전에도, 폭발 중에도, 폭발 후에도 동작을 계속해야만 하게 되었습니다.

물론, 환경적 충격과 공격에 대해 더 안전하게 하드웨어를 만드는 것은 가능합니다. 여러개의 독창적인 기계적 충격 제거 기기들은 충격과 진동의 효과를 줄일 수 있고, 여러겹의 봉인은 낮은 전력의 전자기파의 효과를 줄일 수 있으며, 에러 수정 코딩은 높은 전력의 방사의

² 10여년 후, 일부 타입의 컴퓨터 시스템들에 대한 적합성 테스트는 커다란 폭발을 포함했고, 어떤 타입의 통신 네트워크들은 정확히 “탄도학 무선 통신 방해”라고 불리우는 것들을 다뤄야만 했습니다.

효과를 줄일 수 있고, 다양한 포장과 봉인 기술이 공기 청정 수준으로 인한 영향을 줄일 수 있고, 여러 온도 조절 장치 시스템들이 온도의 효과에 대응할 수 있습니다. 극단적인 경우에, 세개의 모듈 중복은 시스템의 하나의 부분의 실패 하나가 모든 시스템이 옳지 않은 동작을 하는 결과를 초래할 확률을 낮출 수 있습니다. 하지만, 이 모든 방법들은 하나의 공통점을 갖습니다: 그것들이 실패의 확률을 낮출 수는 있지만, 아예 그 확률을 0으로 만들 수는 없다는 것입니다.

이런 가혹한 환경적 조건들이 많은 경우에 더 강력하고 안정적인 하드웨어를 사용해서 처리되곤 하지만 다음 두개의 섹션에서의 워크로드와 어플리케이션 제약은 종종 소프트웨어로 해결되곤 합니다.

15.1.3.2 Workload Constraints

사람에게와 마찬가지로, real-time 시스템에 부담을 지나치게 주는 방법으로 데드라인을 지키지 못하게 하는 것이 가능한 경우가 종종 있습니다. 예를 들어, 만약 시스템이 너무 자주 인터럽트 당한다면, 이 시스템은 real-time 어플리케이션을 다루기에 충분한 CPU 대역폭을 갖지 못할 수 있습니다. 이 문제에 대한 하드웨어를 통한 해결책은 시스템에 인터럽트가 전달되는 정도를 제한하는 것입니다. 가능한 소프트웨어를 통한 해결책은 인터럽트가 너무 자주 들어온다면 일정 시간동안 인터럽트를 불능화 시키는 것, 너무 자주 인터럽트를 생성하는 디바이스를 리셋하는 것, 또는 심지어 인터럽트를 아예 금지시키고 폴링 방식을 사용하는 것 등이 있을 수 있습니다.

지나치게 부담을 주는 행위는 queueing effect로 인해 응답시간을 나쁘게 만들 수 있어서, real-time 시스템들에 있어서 CPU 대역폭을 실제보다 많이 사용할 수 있도록 하는 것은 흔치 않은 일이며, 그로 인해 수행 중인 시스템은 (말해보자면) 80% 정도의 idle 시간을 갖습니다. 이 방법은 저장장치와 네트워킹 디바이스에도 역시 적용됩니다. 어떤 경우에 있어서는 real-time 어플리케이션의 높은 우선순위의 부분들만을 위한 별개의 저장장치와 네트워킹 하드웨어를 전용으로 갖춰 두기도 합니다. real-time 시스템에서는 처리량보다 응답시간이 더 중요하기 때문에, 이런 하드웨어가 대부분의 시간을 idle 상태로 유지하는 것은 물론 드문 일이 아닙니다.

Quick Quiz 15.2: 하지만 queueing theory에서의 결과를 놓고 생각해 보면, 낮은 리소스 활용률은 평균 응답시간만을 개선할 뿐이지 최악의 경우의 응답시간은 개선하지 못하지 않을까요? 그리고 최악의 경우에서의 응답시간이야말로 대부분의 real-time 시스템들이 정말로 신경쓰는 것이 아니던가요? ■

물론, 충분히 낮은 리소스 활용률을 유지하는 것은 설계와 구현 단계에서의 많은 연습을 필요로 합니다. 데드라인을 어기게 만들 수 있는 작은 기능적 잘못 같은

것은 없어야 합니다.

15.1.3.3 Application Constraints

최대 시간이 정해진 응답 시간을 제공하기는 일부 오퍼레이션이 다른 것에 비해 쉽습니다. 예를 들어, 인터럽트와 wake-up 오퍼레이션들에 대한 응답시간 명세서를 보는 일은 흔하지만, (예를 들어) 파일 시스템 `umount` 오퍼레이션에 대한 응답시간 명세서는 상당히 드뭅니다. 이에 대한 이유중 하나는 파일시스템 `umount` 오퍼레이션은 파일시스템의 메모리 안에 있는 데이터를 모두 대용량 저장장치로 비워 내려야만 하기 때문에, 해야 하는 일의 양을 제한하기는 상당히 어렵기 때문입니다.

이는 real-time 어플리케이션들이 제한된 대기시간을 합리적으로 제공해줄 수 있는 오퍼레이션들의 사용만으로 국한되어야 함을 의미합니다. 다른 오퍼레이션들은 해당 어플리케이션의 real-time 이 아닌 영역으로 들어가거나 아예 사라져야만 합니다.

이 어플리케이션의 real-time 이 아닌 영역에 대해서도 제한이 있을 수 있습니다. 예를 들어, real-time 이 아닌 어플리케이션은 real-time 영역에 의해서 사용되는 CPU를 사용해도 되는 걸까요? 해당 어플리케이션의 real-time 부분이 일반적이지 않게도 바쁠 수 있는 시간이 존재할까요, 만약 그렇다면, 이 어플리케이션의 real-time 이 아닌 부분은 그런 시간 동안에도 계속 수행될 수 있을까요? 마지막으로, 해당 어플리케이션의 일만큼의 real-time 부분까지는 real-time 이 아닌 부분의 처리량을 저하시켜도 좋은 것으로 허용되는 걸까요?

15.1.3.4 Real-World Real-Time Specifications

앞의 섹션들에서 보았듯이, 실제 세상에서의 real-time 명세는 환경, 워크로드 그리고 어플리케이션 자체에 있어서의 제약을 포함해야 합니다. 또한, 어플리케이션의 real-time 부분이 사용될 수 있는 경우를 위해선, 그런 오퍼레이션들을 구현하는 하드웨어와 소프트웨어에 제약이 존재해야만 합니다.

그런 오퍼레이션들에 있어서, 이런 제약들은 최대 응답시간(그리고 최소 응답시간을 포함할 수도 있습니다)과 그 응답시간을 지킬 수 있는 확률을 포함할 수 있습니다. 100% 확률은 연관된 오퍼레이션이 hard real-time 서비스를 제공해야 함을 의미합니다.

어떤 경우들에 있어서는, 응답시간과 그것이 지켜질 확률에 대한 요구사항이 문제시 되는 오퍼레이션의 인자에 따라 다양하게 달라질 수도 있습니다. 예를 들어, 로컬 LAN을 통한 네트워크 오퍼레이션은 (대략) 100-마이크로세컨드 내에 완료될 확률이 대륙간 WAN을 통한 네트워크 오퍼레이션이 100-마이크로세컨드 내에 완료될 확률보다는 훨씬 클 것입니다. 더 나아가서, 구리선이나 광섬유로 구성된 LAN에서의 네트워크 오퍼

레이션은 시간을 소모하는 재전송 없이 완료될 확률이 매우 높은 반면, 취약한 WIFI 네트워크를 통한 네트워킹 오퍼레이션은 데드라인을 놓칠 확률이 매우 클 것입니다. 유사하게, 밀접하게 엮인 solid-state disk (SSD)에서의 읽기 오퍼레이션은 구식의 USB로 연결된 녹슬은 disk drive에서의 같은 읽기에 비해 훨씬 빠를 것으로 예상할 수 있습니다.³

어떤 real-time 어플리케이션들은 다른 오퍼레이션 단계를 통해 수행될 수 있습니다. 예를 들어, 돌아가는 통나무로부터 (“베니어”라고 불리는) 얇은 나무판을 벗겨내는 합판 선반을 제어하는 real-time 시스템은 반드시: (1) 통나무를 선반에 가져오고, (2) 통나무에 있는 가장 큰 원기둥이 칼날에 노출되도록 통나무를 선반의 고정시키는 목재 위에 위치시키고, (3) 통나무를 돌리기 시작한 후, (4) 통나무로부터 베니어판을 벗겨낼 수 있도록 칼날의 위치를 지속적으로 조절하고, (5) 벗겨내기엔 너무 작게 된 통나무의 중심 부분을 제거하고, (6) 다음 통나무를 기다립니다. 이 여섯 단계의 오퍼레이션 각각은 역시 그 자체의 데드라인과 환경적 제약을 가질 수 있는데, 예를 들어 단계 4의 데드라인은 단계 6의 그것보다 훨씬 가혹해서 초단위가 아니라 밀리세컨드 단위일 것을 예상할 수 있을 겁니다. 따라서 낮은 우선순위의 작업은 단계 4에서 보다는 단계 6에서 수행될 것으로 예상할 수 있을 겁니다. 그렇다면 하나, 단계 4의 더 가혹한 요구사항을 지켜주기 위해선 세심한 하드웨어, 드라이버의 선택과 소프트웨어 구성이 더 필요할 겁니다.

이런 단계에 따른 단계 방법의 장점은 응답시간 예산이 나누어질 수 있어서, 어플리케이션의 다양한 컴포넌트들이 각각 자신의 응답시간 예산을 가진채로 독립적으로 개발될 수 있다는 점입니다. 물론, 모든 다른 종류의 예산과 마찬가지로, 어떤 컴포넌트가 전체 예산 중 얼만큼의 부분을 차지하게 될 것인지에 대해서는 가끔 충돌이 있을 수 있고, 다른 종류의 예산과 마찬가지로 강한 리더쉽과 공유된 목표에 대한 감이 이런 충돌을 빠르게 해결하는데에 도움이 될 수 있습니다. 그리고, 또다시 다른 종류의 기술적 예산과 마찬가지로, 응답시간에 대한 적절한 포커스를 보장하고 응답시간 문제에 대한 경고를 빨리 내기 위해서는 강력한 검증 노력이 필요합니다. 성공적인 검증 시도는 거의 항상, 이론가를 만족시킬수는 없을수도 있지만 실제 일이 수행되는데에 도움을 주는 덕목을 가진 훌륭한 테스트 모음을 포함할 것입니다. 실제로, 2015년 초에 있어서, 대부분의 실세계의 real-time 시스템은 형식적 검증보다는 적합성 테스트를 사용합니다.

그렇다고는 하나, real-time 시스템을 검증하기 위한

테스트 집합의 넓은 사용은 매우 현실적인 단점을 갖는데, real-time 소프트웨어는 특정 하드웨어와 소프트웨어 구성 위에서만 검증된다는 점입니다. 또 다른 하드웨어와 환경구성을 추가하는 것은 추가적인 비용과 시간 소모를 요하는 테스트를 필요로 합니다. 형식적 검증이 이 상황을 바꿀 수 있기에 충분할 만큼 진보될 수도 있겠습니다만, 2015년 초에 있어서는, 더 많은 진보가 필요한 상황입니다.

Quick Quiz 15.3: 형식적 검증은 수십년간 집중된 연구로 인해 이미 상당히 실용 가능한 수준입니다. 추가적인 진보가 정말로 필요한 건가요. 아니면 이건 그저 실무자의 게으르기를 계속하고 형식적 검증의 대단한 위력을 무시하려는 변명일 뿐인가요? ■

어플리케이션의 real-time 부분에서의 응답시간 요구사항에 더해서, 어플리케이션의 real-time이 아닌 부분을 위한 성능과 확장성 요구사항도 있을 수 있습니다. 이런 추가적 요구사항들은 궁극의 real-time 응답시간들은 평균 성능과 확장성을 떨어뜨림으로써 얻어질 수 있다는 사실을 반영합니다.

소프트웨어 엔지니어링 요구사항 또한 중요할 수 있는데, 커다란 팀에 의해 개발되고 유지되어야 하는 커다란 어플리케이션에서는 특히나 그렇습니다. 이런 요구사항들은 모듈성의 증가와 실패의 격리화를 선호하게 되는 경우가 많습니다.

이건 제품 단계의 real-time 시스템을 위한 데드라인과 환경적 제약들을 명세하기 위해 필요한 작업들의 간략한 개요에 지나지 않습니다. 이런 개요가 real-time 컴퓨팅에 대한 말로 듣기 좋은 이야기 기반의 접근법의 부적합성을 잘 보여주기를 바랍니다.

15.2 Who Needs Real-Time Computing?

모든 컴퓨팅이 사실은 real-time 컴퓨팅이라고 주장하는 것도 가능할 겁니다. 한가지 적당히 극단적인 예를 들자면, 여러분이 생일선물을 온라인에서 구매한다면, 여러분은 그 선물이 받을 사람의 생일 전에 도착하기를 바랄 겁니다. 그리고 사실 turn-of-the-millenium 웹 서비스들에서조차도 1초 미만의 응답 제약이 관측되었고 [Boh01], 시간의 경과에 따라서 그 요구사항이 완화되지도 않았습니다 [DHJ⁺07]. real-time이 아닌 시스템이나 어플리케이션에서 단순하게 응답시간 요구사항이 이뤄질 수는 없는 그런 real-time 어플리케이션들에 집중하는게 어느정도 유용할 겁니다. 물론, 하드웨어의 비용이 낮아지고 대역폭과 메모리 크기가 증가할수록 real-time과 real-time이 아닌 것 사이의 간격은 바뀌어 갈 것입니다만, 그런 진보는 결코는 나쁜 일이 아닙니다.

³ 중요한 보안 팁: USB 디바이스에서의 최악의 경우의 응답시간은 극단적일 정도로 길 수 있습니다. 따라서 Real-time 시스템들은 critical path에서 USB 디바이스를 떨어뜨려 놓도록 신경써야 합니다.

다.

Quick Quiz 15.4: Real-time 과 real-time 이 아닌 것을 무엇이 “real-time 이 아닌 시스템과 어플리케이션에 의해서 간단히 이뤄질 수 있는지”로 구분하는 것은 우스운 일입니다! 그런 구별을 위한 이론적 기초는 전혀 존재치 않아요!!! 더 나은 것을 할 수는 없을까요??? ■

Real-time 컴퓨팅은 제조부터 항공 전자공학; 가장 스펙타클한 예를 들면 커다란, 지구에 달라붙어 있는 망원경이 별빛을 없애기 위해 사용되는 적응 제어 광학과 같은 과학 어플리케이션들; 앞서 언급된 항공 전자공학을 포함한 군사 어플리케이션; 그리고 기회를 인식하기 위한 첫번째 계산이 대부분의 초래되는 이익을 거둬들일 수 있는 금융 서비스 어플리케이션들까지 다양한 산업 제어 어플리케이션들에서 사용됩니다. 이 네가지 영역들은 “생산의 탐색”, “삶의 탐색”, “죽음의 탐색”, 그리고 “돈의 탐색”으로 성격지어질 수 있을 겁니다.

돈은 물질이 아니므로 계산 외의 반응속도는 상당히 작은 금융 서비스 어플리케이션은 다른 세개 카테고리의 어플리케이션들과 약간 다릅니다. 반대로, 다른 세개의 카테고리들에서 근본적으로 존재하는 기계적 자연들은 어플리케이션의 실시간 응답시간의 축소가 작은 이익 또는 이익을 아예 제공하지 않게 되는 수학적 간접성을 제공합니다. 이 말은 금융 서비스 어플리케이션들은 다른 real-time 정보 처리 어플리케이션들과 달리 가장 낮은 응답시간을 갖는 어플리케이션이 일반적으로 승리하는 경주에 직면하게 됨을 의미합니다. 결과적으로 응답시간 요구사항은 여전히 Section 15.1.3.4에 설명된 것과 같이 명시될 수 있으며, 이런 요구사항들의 일반적이지 않은 특성은 금융과 정보 처리 어플리케이션을 “real time” 보다는 “low latency” 어플리케이션으로 여겨지게 합니다.

우리가 그것을 정확히 뭐라 부를 것인지와는 관계 없이, real-time 컴퓨팅을 위한 상당한 필요가 존재합니다 [Pet06, Inm07].

15.3 Who Needs Parallel Real-Time Computing?

누가 병렬 real-time 컴퓨팅을 정말 필요로 하는지는 덜 분명합니다만, 그와 상관 없이 낮은 비용의 멀티코어 시스템의 진보가 그 자신을 전면에 대주시켰습니다. 안타깝게도, real-time 컴퓨팅을 위한 전통적인 수학적 기본 사항들은 그 규칙을 증명하는 일부 예외 [Bra11] 와 함께 단일 CPU 시스템을 가정하고 있습니다. 그렇다면 하지만, 최신 컴퓨팅 하드웨어가 real-time 수학적 사항들에 들어맞도록 하는 방법들이 두 가지 존재하고, 일부 Linux 커널 해커들은 학계에서 이런 변환을 만들도록 장려해왔습니다 [Gle10].

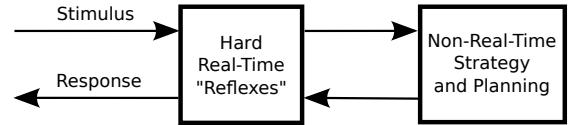


Figure 15.4: Real-Time Reflexes

한가지 방법은 Figure 15.4에 보여진 것처럼, real-time 응답으로부터 실시간이 아닌 전략짜기와 계획짜기까지 다양한 생물학적 신경 시스템을 많은 real-time 시스템들이 반영하고 있다는 사실을 깨닫는 것입니다. 센서들의 상태를 읽고 작동기를 제어하는 hard real-time 응답들은 하나의 CPU 위에서 real-time으로 수행되는 동안, 어플리케이션의 real-time이 아닌 전략짜기와 계획짜기 부분은 남아있는 여러 CPU에서 수행됩니다. 전략짜기와 계획짜기 활동은 통계적 분석, 주기적 교정, 사용자 인터페이스, 공급망 활동, 그리고 준비활동 등을 포함할 수 있습니다. 컴퓨팅 부하를 많이 주는 준비 활동의 한 예를 들어보자면, Section 15.1.3.4에서 이야기한 합판 깎아내기 어플리케이션을 다시 생각해 봅시다. 하나의 CPU가 하나의 통나무를 깎아내는데 필요한 고속의 real-time 계산에 사용되고 있는 동안, 다른 CPU들은 고품질의 합판의 많은 수량을 얻기 위해서 다음 통나무를 어떻게 위치시켜야 할지를 결정하기 위해서 다음 통나무의 크기와 모양을 분석할 수 있을 겁니다. 많은 어플리케이션들이 real-time과 real-time이 아닌 컴퓨트들을 가지고 있음이 드러났으므로 [BMP08], 이 방법은 많은 경우에 전통적인 real-time 분석이 최신의 멀티코어 하드웨어와 결합될 수 있도록 하곤 합니다.

또 다른 사소한 방법은 하나의 하드웨어 쓰레드만을 켜고 나머지는 모두 꺼서 타협된 유니프로세서 real-time 컴퓨팅의 수학으로 되돌아가는 것입니다. 하지만, 이 방법은 잠재적 비용과 에너지 효율성의 장점을 포기하게 됩니다. 그렇다면 하나, 이런 장점을 취하는 것은 Chapter 3에서 다룬 병렬 성능 문제들을 극복해낼 것을 필요로 하고, 평균적인 경우에만 아니라 최악의 경우에 대해서 그렇습니다.

따라서 병렬 real-time 시스템을 구현하는 것은 상당히 어려운 일입니다. 이 어려운 목표를 달성하는 것은 다음 섹션에서 그 방법의 윤곽을 그려봅니다.

15.4 Implementing Parallel Real-Time Systems

우리는 event-driven과 polling이라는 real-time 시스템의 두 가지 주요 스타일들을 알아보겠습니다. Event-driven real-time 시스템은 대부분의 시간을 idle로 유지

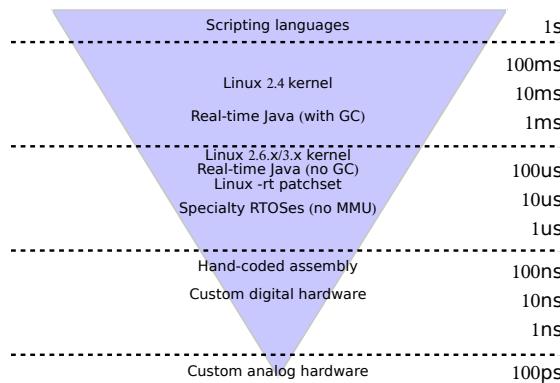


Figure 15.5: Real-Time Response Regimes

하고, 운영체제를 통해 어플리케이션으로 전달되어진 이벤트에 실시간으로 응답합니다. 대안적으로, 시스템은 idle 상태를 유지하는 대신 real-time 이 아닌 워크로드를 백그라운드로 수행시킬 수 있습니다. Polling real-time 시스템은 CPU에 성능이 결정되는 real-time 쓰레드를 갖고서, 이 쓰레드로 딱맞는 루프 내에서 입력이 있는지 알아보고 있을 경우 출력을 업데이트합니다. 이 딱맞는 입력 조사 루프는 사용자 모드 어플리케이션의 주소 공간에 매핑된 하드웨어 레지스터로부터 읽고 그 레지스터로 쓰기를 하는 방식으로 완전히 사용자 모드에서 수행되는 경우가 많습니다. 대안적으로, 일부 어플리케이션들은 입력 조사 루프를 커널에 넣어두는데, 예를 들면 로딩할 수 있는 커널 모듈을 사용하는 방식입니다.

어떤 스타일을 선택하는가와 상관 없이, real-time 시스템을 구현하는데 사용되는 방법은 예를 들면 Figure 15.5에 보인 것과 같이 테드라인에 종속적일 겁니다. 이 그림의 위에서부터 시작해 보자면, 여러분이 1초가 넘는 반응 시간을 가져도 괜찮다면, 여러분의 real-time 어플리케이션을 구현하는데에 스크립트 언어를 사용할 수도 있을 겁니다—그리고 스크립트 언어는 실제로 놀라울 만큼 자주 사용되는데, 제가 이 방법을 추천해야 할 정도는 아닙니다. 요구되는 응답시간이 수십 밀리세컨드를 넘는다면, 오래된 2.4 버전의 리눅스 커널이 사용될 수도 있을 텐데, 이 역시 제가 추천해야 할 정도는 아닙니다. 특수한 real-time Java 구현은 가비지 컬렉터의 사용에도 불구하고 수 밀리세컨드의 실시간 응답시간을 제공합니다. 리눅스 2.6.x와 3.x 커널은 주의깊게 구성되고 튜닝되어서 real-time에 친화적인 하드웨어에서 수행된다면 수백 마이크로세컨드의 실시간 응답시간을 제공합니다. 특수한 real-time Java 구현은 가비지 컬렉터의 사용이 주의깊게 막아진다면 100마이크로세컨드 아래의 실시간 응답시간을 제공할 수 있

습니다. (하지만 가비지 컬렉터를 막는 행위는 Java의 많은 표준 라이브러리의 사용을 막게 되어서 Java의 생산성에서의 장점을 없애버린다는 점을 알아두시기 바랍니다.) -rt 패치셋을 포함한 리눅스 커널은 20마이크로세컨드 아래의 응답시간을 제공하고, 메모리 변환 없이 동작하는 특수한 real-time 운영체제 (RTOSes)들은 10마이크로세컨드 아래의 응답시간을 제공합니다. 마이크로세컨드 아래의 응답시간을 달성하는 것은 손으로 짜여진 어셈블리 코드나 심지어 특수 목적의 하드웨어를 사용할 것을 필요로 합니다.

물론, 이 스택의 전체에 걸쳐 주의깊은 구성과 튜닝이 필요합니다. 상세히 말하자면, 하드웨어나 펌웨어가 real-time 응답을 제공하는데에 실패한다면, 소프트웨어가 그 일어버린 시간을 만회하기 위해 할 수 있는 일은 없습니다. 그리고 고성능 하드웨어는 가끔 더 나은 처리량을 얻기 위해 최악의 경우의 동작을 희생합니다. 실제로, 인터럽트가 불능화된 상태에서 돌아가는 짧은 루프에서의 타이밍은 고성능 무작위수 생성기의 기본을 제공할 수 있습니다 [MOZ09]. 더 나아가서, 일부 펌웨어는 다양한 필수 태스크들을 이끌어가기 위해 cycle-stealing을 하고, 일부 경우에는 희생자 CPU의 하드웨어 클락을 재프로그래밍 함으로써 그 궤도를 덮으려 시도합니다. 물론, cycle stealing은 가상 환경에서는 예상된 행동입니다만, 사람은 가상 환경에서 real-time 응답을 위해 일하지 않습니다 [Gle12, Kis14]. 따라서 여러분의 하드웨어의, 그리고 펌웨어의 real-time 기능을 평가하는게 상당히 중요합니다. 그런 평가를 진행해 주는 기관들이 존재하는데, Open Source Automation Development Lab (OSADL)도 그 중 하나입니다.

하지만 주어진 유능한 real-time 하드웨어와 펌웨어를 두고서, 이 스택의 다음 위쪽 계층은 다음 섹션에서 다루어질 운영체제입니다.

15.4.1 Implementing Parallel Real-Time Operating Systems

Real-time 시스템을 구현하는데 사용될 수 있는 여러 가지 전략들이 있습니다. 한가지 방법은 Figure 15.6에 보인 것처럼 특수 목적 real-time 운영체제 (RTOS) 위에 범용 real-time이 아닌 OS를 포팅하는 것입니다. 초록색의 “Linux Process” 상자들은 리눅스 커널에서 돌아가는 real-time이 아닌 프로세스들을 의미하며, 노란색의 “RTOS Process” 상자들은 RTOS 위에서 돌아가는 real-time 프로세스들을 의미합니다.

이 방법은 리눅스 커널이 real-time 기능을 얻게 되기 전까지 매우 널리 사용되던 방법이고, 오늘날에도 여전히 사용되고 있습니다 [xen14, Yod04b]. 하지만, 이 방법은 어플리케이션이 RTOS에서 수행되는 부분과 리눅스에서 수행되는 또 다른 부분으로 나뉘어질 것을 필

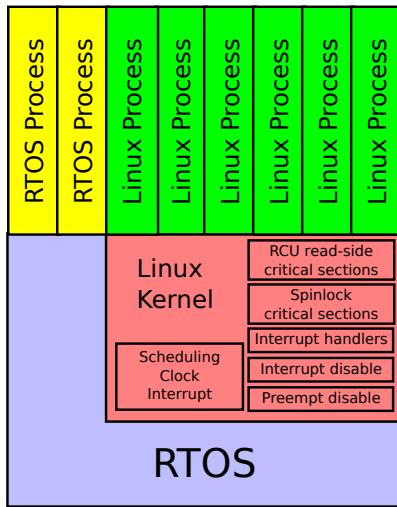


Figure 15.6: Linux Ported to RTOS

요로 합니다. 예를 들어 RTOS로부터의 POSIX 시스템 콜을 리눅스에서 수행되는 유ти리티 쓰레드로 포워딩 시키거나 하는 식으로 이 두개의 환경이 비슷해 보이게 만드는게 가능하긴 하지만, 언제나 그러기 어려운 부분들이 존재합니다.

또한, 이 RTOS는 하드웨어와 리눅스 커널 둘 다와 연결되어야만 하고, 따라서 하드웨어와 커널 둘 다에서 발생되는 변경에 대해서 상당한 관리를 필요로 합니다. 더 나아가서, 이런 RTOS는 각각 그 스스로의 시스템 콜 인터페이스와 시스템 라이브러리 집합을 갖추고 있는 경우가 많아서 에코시스템과 개발자들을 분할해 놓을 수 있습니다. 실제로, 이런 문제들이 RTOS들과 리눅스의 조합을 이끌게 된 것으로 보이는데, 이 방법은 RTOS의 전체 real-time 기능으로의 접근을 허락하는 동시에 리눅스의 풍부하고 다양한 오픈 소스 생태계로의 어플리케이션의 real-time 이 아닌 완전한 접근을 가능하게 하기 때문입니다.

RTOS들을 리눅스 커널과 짹짓는 것은 리눅스 커널이 최소한의 real-time 기능들만을 가지고 있던 때에는 현명하고 유용한 단기적 방법이었긴 합니다만, 이는 또한 리눅스 커널에 real-time 기능을 추가하는 동기가 되었습니다. 이 목표를 향한 진행이 Figure 15.7에 보여져 있습니다. 위쪽의 행은 preemption이 불능화 되어져서 핵심적으로는 real-time 기능을 가지고 있지 않은 상태의 리눅스 커널을 다이어그램으로 그리고 있습니다. 중간의 행은 preemption이 활성화 된 채로 메인라인 리눅스 커널의 real-time 기능들이 들어나는 것을 보이는 다이어그램들을 보이고 있습니다. 마지막으로, 가장 아래의 행은 -rt 패치셋이 적용되어서 real-time 기능을 최대

화 시킨 리눅스 커널의 다이어그램을 보이고 있습니다. -rt 패치셋으로부터의 기능들이 메인라인에 추가됨으로써, 메인라인 리눅스 커널의 기능들이 시간에 따라 증가되었습니다. 더도 아니고 덜도 아니고, 가장 큰 노력을 요하는 real-time 어플리케이션들은 -rt 패치셋을 사용 하길 유지하고 있습니다.

Figure 15.7의 꼭대기에 보여진 non-preemptible 커널은 CONFIG_PREEMPT=n과 함께 빌드되어져서, 이 리눅스 커널에서의 수행은 preemption 당할 수 없습니다. 이는 곧 이 커널의 real-time 응답 시간의 최대 시간은 리눅스 커널의 가장 긴 코드 수행 경로에 따라 정해진다는 것을 의미하는데, 이 시간은 실제로입니다. 하지만, 사용자 모드 수행은 preemption 당할 수 있고, 따라서 우상단에 보여진 real-time 리눅스 프로세스들 가운데 하나는 non-real-time 리눅스 프로세스들 가운데 어느 하나든 사용자 모드에서 수행 중일 때 해당 non-real-time 리눅스 프로세스를 preemption 할 수 있습니다.

Figure 15.7의 가운데 행에 보여진 preemption 가능한 커널들은 CONFIG_PREEMPT=y 설정과 함께 빌드되어지며, 따라서 해당 리눅스 커널 내의 프로세스 수준 코드는 대부분 preemption 될 수 있습니다. 물론 이는 real-time 응답 시간을 상당히 개선시킵니다만, 이 그림의 가운데 행의 가장 왼쪽 다이어그램 안의 빨간색 박스로 보여지듯이 RCU read-side 크리티컬 섹션, 스팬락 크리티컬 섹션, 인터럽트 핸들러, 인터럽트가 불능화된 코드 영역, 그리고 preemption 불능화된 코드 영역에서는 여전히 preemption이 불능화 되어 있습니다. Preemption 가능한 RCU의 진화는 가운데 다이어그램에서 보여지듯이 RCU read-side 크리티컬 섹션이 preemption 될 수 있게 했고, 쓰레드화 된 인터럽트 핸들러들은 가장 오른쪽 다이어그램에 보여지듯이 디바이스 인터럽트 핸들러들이 플리엠션될 수 있게 만들었습니다. 물론, 이 시간 동안에 다른 real-time 기능들을 처리하는 기능들이 추가되었습니다만, 이는 이 다이어그램에 쉽게 표시될 수 없었습니다. 그에 대해서는 대신 Section 15.4.1.1에서 이야기 하겠습니다.

마지막 방법은 Figure 15.8에 보인 것처럼, real-time 프로세스로부터 모든 것을 차우는 것으로, 이 프로세스가 필요로 하는 모든 CPU로부터 모든 다른 프로세스 처리를 없애버리는 것입니다. 이는 3.10 리눅스 커널에서 CONFIG_NO_HZ_FULL Kconfig 인자로 구현되었습니다 [Wei12]. 이 방법은 예를 들면 수행중인 커널 daemon들과 같은 백그라운드 처리를 위해서 최소 하나의 최소 작업을 수행할 CPU를 필요로 한다는 점을 알아둘 필요가 있습니다. 하지만, 최소 작업을 수행할 CPU가 아닌 CPU에 하나의 수행 가능한 task만이 존재할 때에는 해당 CPU에서 scheduling-clock 인터럽트가 꺼져서 간섭과 OS jitter의 중요 원인을 하나 없애

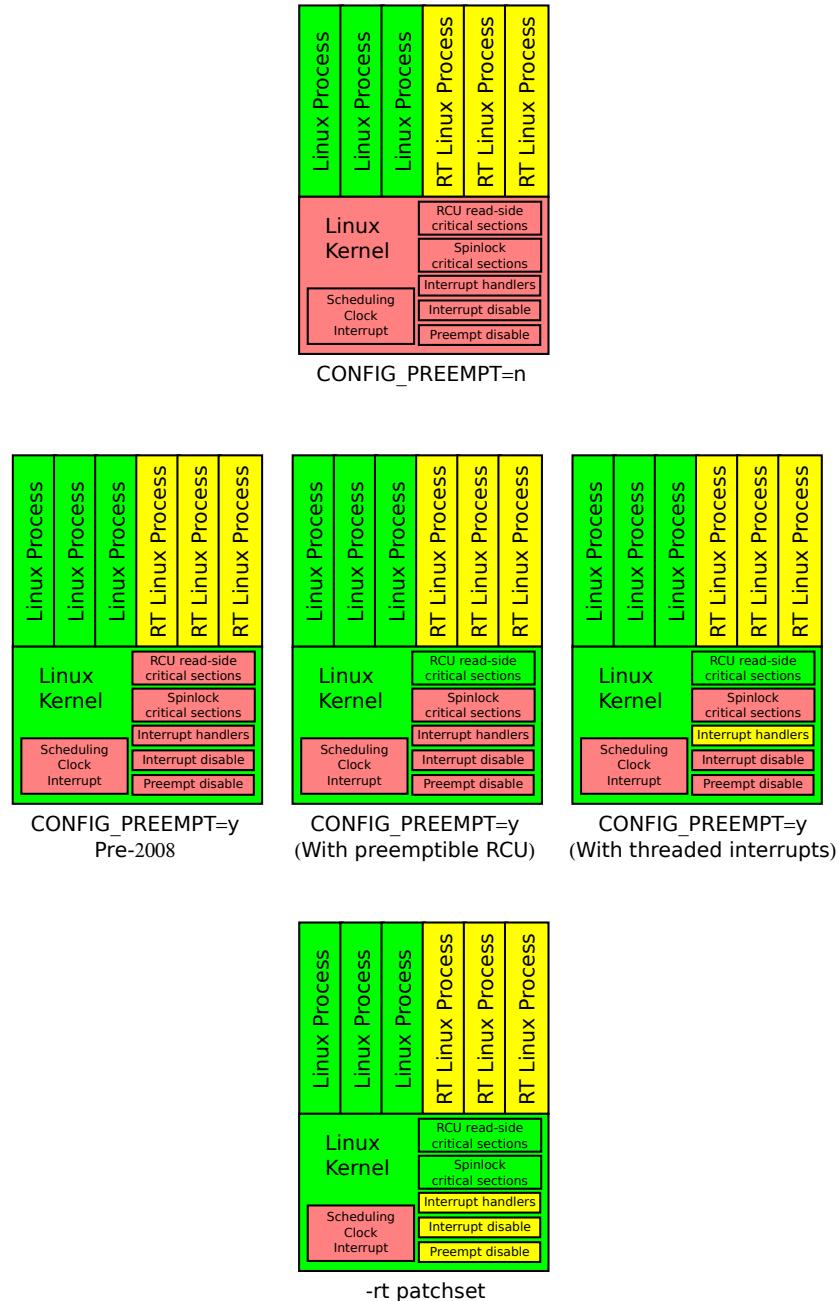


Figure 15.7: Linux-Kernel Real-Time Implementations

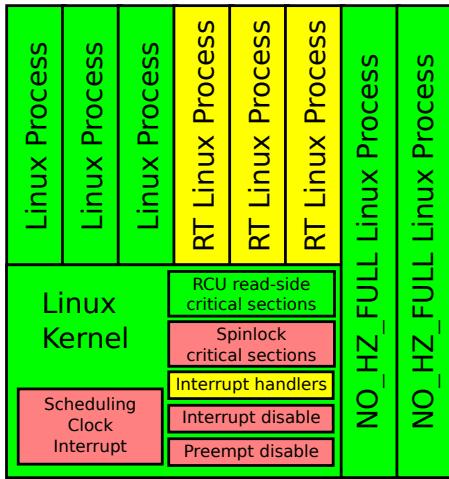


Figure 15.8: CPU Isolation

버립니다.⁴ 몇가지 예외와 함께, 이 커널은 해당의 최소 작업을 수행할 CPU로부터 다른 처리를 없애버리도록 강제하지는 않습니다만, 그대신 해당 CPU에 하나의 수행 가능한 task 만이 존재할 때 더 나은 성능을 제공합니다. 제대로 구성된다면, 간단하지 않은 일을 하는 CONFIG_NO_HZ_FULL은 real-time 쓰레드들에게 거의 bare-metal 시스템의 그것에 필적하는 것에 가까운 수준의 성능을 제공합니다.

이런 방법들 가운데 무엇이 real-time 시스템을 위해 최선인가에 대해서는 물론 상당한 언쟁이 있어왔습니다. 이 언쟁은 상당한 시간동안 진행되어오고 있습니다 [Cor04a, Cor04c]. 대부분의 경우와 같이, 그에 대한 답은 다음 섹션들에서 이야기되듯이 “다른것에 종속적이다” 일 것으로 보입니다. Section 15.4.1.1은 event-drive real-time 시스템들을 알아보고, Section 15.4.1.2는 CPU에 성능이 국한되는 polling 루프를 사용하는 real-time 시스템을 알아봅니다.

15.4.1.1 Event-Driven Real-Time Support

Event-driven real-time 어플리케이션을 위해 필요한 운영 체제의 지원은 상당히 비용이 비쌉니다만, 이 섹션에서는 그 중 몇가지 항목들에만 주목할텐데, 타이머, 쓰레드 인터럽트, 우선순위 상속, preemption 가능한 RCU, 그리고 preemption 가능한 스핀락이 그것입니다.

Timers는 real-time 운영을 위해 상당히 중요한 계분명합니다. 무엇보다도, 무언가가 특정 시간 동안 되었음을 말할 수가 없다면, 어떻게 그 시간 내로 응답을 해줄 수 있겠습니까? Real-time이 아닌 시스템에서조차도, 많은 수의 타이머들이 생성되고, 따라서 그것들은 상당히 효율적으로 다루어져야만 합니다. 사용 예로는 (거의 항상 그것들이 올릴 기회가 오기 전에 취소되어 버리는) TCP 연결을 위한 재전송 타이머,⁵ (드물게 취소되는 sleep(1)과 같이) 시간이 지정된 딜레이, 그리고 (올릴 기회가 오기 전에 취소되는 경우가 자주 있는) poll() 시스템 콜을 위한 타임아웃 등이 있을 겁니다. 따라서 그런 타이머를 위한 좋은 데이터 구조는 추가 기능과 삭제 기능이 빠르고 예약이 걸린 타이머의 갯수에 대해 $O(1)$ 의 시간 비용을 갖는 우선순위 대기열이 될 겁니다.

이런 목적의 고전적인 데이터 구조는 리눅스 커널에서 *timer wheel*이라 불리는 *calendar queue*입니다. 이 오래된 데이터 구조는 discrete-event 시뮬레이션에서 많이 사용되기도 합니다. 여기서의 아이디어는 시간이 양자화 된다는 것으로, 예를 들어 리눅스 커널에서는 시간 분량의 길이는 scheduling-clock 인터럽트의 기간입니다. 특정 시간은 정수로 표현될 수 있고, 어떤 양자화 기준으로 완전치 않은 시점에서 하나의 타이머를 등록하려는 모든 시도는 양자화된 완전한 시간 분량에 가깝도록 조정될 겁니다.

한가지 간단한 구현은 시간의 아래쪽 비트로 인덱스 되는 하나의 배열을 할당하는 것이 될 겁니다. 이는 이론적으로는 동작하지만, 실제 시스템에서는 거의 항상 취소되어버리는 긴 시간의 타임아웃 (timeout)을 굉장히 많이 발생시킵니다 (예를 들어, TCP 세션들에서 45분짜리의 keepalive 타임아웃). 이런 긴 시간을 필요로 하는 타임아웃은 대부분의 시간이 아직 만료되지 않은 타임아웃을 넘기는데에 소모되기 때문에 작은 크기의 배열에서는 문제를 일으킵니다. 한편으로는, 임의의 많은 수의 긴 시간의 타임아웃들을 수용할 수 있을 만큼 충분히 큰 배열은 너무 많은 양의 메모리를 소비하게 될 수 있는데, 특히 성능과 확장성을 고려하는 곳에서는 모든 CPU 각각에 대해 그런 배열을 필요로 한다는 점에서 특히 그렇습니다.

이런 충돌을 해결하는 한가지 일반적인 방법은 계층을 사용해서 여러개의 배열들을 제공하는 것입니다. 이 계층의 가장 낮은 단계에서는 각각의 배열 원소가 시간의 한 단위를 나타냅니다. 두 번째 단계에서, 각각의 배열 원소는 시간 단위 N 개를 나타내는데, N 은 각 배열의 원소의 갯수입니다. 세 번째 단계에서, 각각의 배열 원소는 시간 단위 N^2 개를 나타내고, 그런식으로 계층의 윗단계로 진행됩니다. 이 방법은 Figure 15.9에 그려진 것처럼 비현실적일정도로 작은 8-비트 클락에 대

⁴ 1초에 한번 발생하는 잔여의 scheduling-clock 인터럽트는 프로세스 통계 등을 위해 남습니다. 나중에 할 일 목표에는 이런 것들을 처리하고 이 잔여의 인터럽트를 없애는 것을 포함합니다.

⁵ 합리적으로 낮은 폐킷 로스 발생률을 가정한다면요!

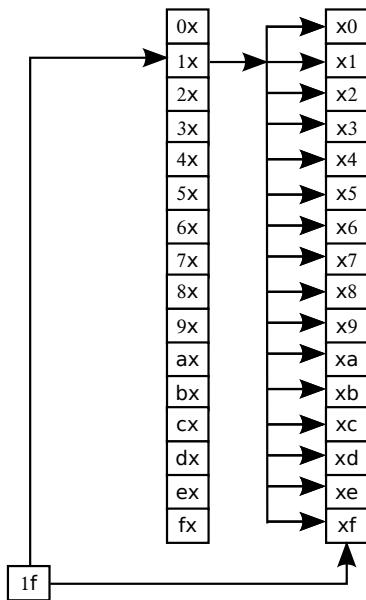


Figure 15.9: Timer Wheel

해 개별적인 배열들이 서로 다른 비트로 인덱스 될 수 있도록 합니다. 여기서, 각각의 배열은 16개의 원소를 가져서 시간의 아래쪽 네개의 비트 (현재 0xf)는 아래 단계의 (가장 오른쪽의) 배열을 인덱스 하고, 다음의 네 개의 비트 (현재 0x1)는 그 다음 위 단계를 인덱스 하게 됩니다. 따라서, 각각 16 개의 원소를 가져서 총 32개의 원소를 갖는 두개의 배열을 가지게 되는데 이는 단일 배열을 사용한다면 256 개의 원소를 갖는 배열을 사용했어야 하는 것에 비하면 굉장히 작은 것입니다.

이 방법은 처리량 기반의 시스템에서는 상당히 잘 동작합니다. 각각의 타이머의 동작은 작은 수의 상수로 $O(1)$ 의 비용을 갖고, 각각의 타이머 원소는 최대 $m+1$ 회 접근되는데, 여기서 m 는 단계의 수입니다.

안타깝게도, time wheel 은 real-time 시스템에서는 두가지 이유로 인해 잘 동작하지 못합니다. 첫번째 이유는 타이머의 정확성과 타이머의 오버헤드 사이에 존재하는 가혹한 트레이드오프로, Figures 15.10 와 15.11 에 그려져 있습니다. Figure 15.10 에서, 타이머 처리는 1 밀리세컨드당 1회만 일어나는데, 이는 많은 (하지만 모든 것은 아닌!) 워크로드들에 대해 충분히 받아들여 질 수 있을 만큼 낮은 오버헤드를 유지하게 됩니다만, 이는 또한 타임아웃이 1 밀리세컨드 단위보다 작은 단위로 설정될 수는 없음을 의미합니다. 다른 한편으로, Figure 15.11 는 타이머 처리가 10 마이크로세컨드마다 발생하게 됨을 보이는데, 여기서는 대부분의 (하지만 모든 것은 아닌!) 워크로드에 대해 받아들여지기 충



Figure 15.10: Timer Wheel at 1kHz



Figure 15.11: Timer Wheel at 100kHz

분할 만큼 세밀한 시간 단위를 제공하지만, 또한 시스템이 타이머를 처리하는 것 이외에는 다른 일을 할수가 없을 만큼 자주 타이머 처리가 이루어짐을 의미합니다.

두번째 이유는 타이머를 위쪽 단계부터 아래쪽 단계 까지 봐야한다는 필요성입니다. Figure 15.9 로 다시 돌아가서, 위단계의 (왼쪽의) 배열의 1x 원소에 넣어진 모든 타이머는 아래 단계의 (오른쪽의) 배열로 이동되어서 그것들의 시간이 도착했을 때에 호출될 수도 있게 해야 합니다. 안타깝게도, 여기에는 아래 단계로 이동하기 위해 기다리고 있는 많은 수의 타임아웃들이 존재할 수 있는데, 특히 많은 수의 단계를 갖는 타이머에서는 더욱 그럴 것입니다. 통계의 힘은 이 단계 이동이 처리량 기반의 시스템에 있어서는 문제가 되지 않도록 합니

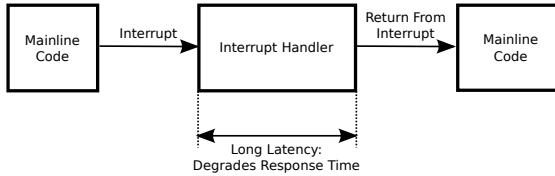


Figure 15.12: Non-Threaded Interrupt Handler

다만, 단계 이동이 real-time 시스템에서는 응답시간의 저하라는 문제를 초래할 수 있습니다.

물론, real-time 시스템은 그냥 다른 데이터 구조를 선택할 수도 있는데, 예를 들면 추가와 삭제 오퍼레이션의 최대 성능 한계 $O(1)$ 이라는 조건을 포기하고 데이터 구조 관리 오퍼레이션에 대해서 $O(\log n)$ 한계를 얻을 수 있는 heap이나 tree의 형태가 될 수 있을 겁니다. 이는 특정 목적의 RTOS 들에 대해서는 좋은 선택이 될 수 있습니다만, 주로 굉장히 많은 수의 타이머들을 지원해야 하는 리눅스와 같은 범용 목적의 시스템들에서는 비효율적입니다.

리눅스 커널의 -rt 패치셋에서 선택된 해결책은 나중의 동작을 스케줄하는 타이머와 TCP 패킷 로스와 같이 발생 확률이 낮은 에러의 에러 처리를 스케줄하는데에 사용되는 타이머를 서로 다르게 하는 것입니다. 여기서의 핵심적인 발견은 에러 처리는 일반적으로 특별히 시간이 중요하지 않고, 따라서 time wheel의 밀리세컨드 단위 관리도 충분하고 좋다는 것입니다. 또 다른 하나의 핵심 발견은 에러 처리용 타임아웃들은 일반적으로 매우 빨리 취소되는데, 그 빠른 정도는 아래 단계로의 전파 전인 경우도 갖다는 것입니다. 마지막 발견은 시스템은 발생시키는 타이머 이벤트 보다도 훨씬 더 많은 에러 처리 타임아웃들을 가지고 있고, 따라서 $O(\log n)$ 데이터 구조는 타이머 이벤트에 받아들여질만한 성능을 제공해야 한다는 것입니다.

짧게 정리해서, 리눅스 커널의 -rt 패치셋은 에러 처리 타임아웃에 대해서는 timer wheel을 사용하고 타이머 이벤트에는 tree를 사용해서 각 카테고리에 필요시 되는 품질의 서비스를 제공합니다.

Thread interrupts는 Figure 15.12에 보여진 것과 같이 느려진 real-time 응답시간의 중요한 원인이 되는, 오래 수행되는 인터럽트 핸들러를 해결하기 위해 사용됩니다. 이런 응답시간들은 하나의 인터럽트에 많은 수의 이벤트를 전달할 수 있는 기기들에서 특히나 문제가 될 수 있는데, 이 인터럽트 핸들러는 이 이벤트들 전체를 처리하는 연장된 시간 기간동안 동작하게 될 것임을 의미합니다. 더 나쁜건 여전히 수행 중인 인터럽트 핸들러에 새로운 이벤트를 전달할 수 있는 기기들로, 그런 인터럽트 핸들러는 정해지지 않은 시간동안 수행될 수

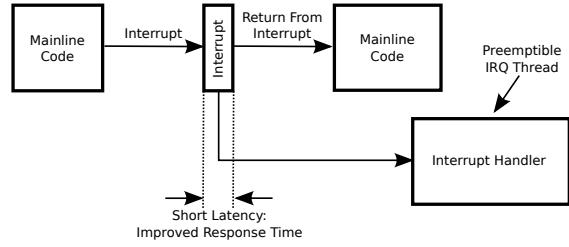


Figure 15.13: Threaded Interrupt Handler

도 있고, 그로 인해 무한정 real-time 응답시간을 느리게 만들 수 있습니다.

이 문제를 해결하는 한가지 방법은 Figure 15.13에 보여진 쓰레드화 된 인터럽트의 사용입니다. 인터럽트 핸들러는 설정될 수 있는 우선순위로 수행되는, preemption 될 수 있는 IRQ 쓰레드의 컨텍스트에서 수행됩니다. 이렇게 되면 이 기기의 인터럽트 핸들러는 IRQ 쓰레드가 이 새로운 이벤트를 알아챌 수 있을 때까지의 짧은 시간동안만 수행됩니다. 그림에서 보였듯이, 쓰레드화된 인터럽트는 real-time 응답시간을 굉장히 개선할 수 있는데, IRQ 쓰레드의 컨텍스트에서 수행되는 인터럽트 핸들러는 더 높은 우선순위의 real-time 쓰레드에 의해 preemption 당할 수 있기 때문인 게 한 부분적인 이유입니다.

하지만, 공짜 점심같은 건 없고, 쓰레드화된 인터럽트에도 단점이 존재합니다. 한가지 단점은 길어진 인터럽트 응답시간입니다. 곧바로 인터럽트 핸들러를 수행하는 대신에, 핸들러의 수행은 IRQ 쓰레드가 수행될 수 있도록 되기까지 지연됩니다. 물론, 이는 인터럽트를 발생시키는 기기가 real-time 어플리케이션의 중요한 수행 경로에 있지 않다면 문제가 되지 않습니다.

또 다른 단점은 잘못 작성된 높은 우선순위의 real-time 코드가 인터럽트 핸들러를 starvation 상태에 빠뜨릴 수 있다는 것으로, 예를 들어 네트워킹 코드가 수행되지 못하게 만들어 버려서 이 문제를 디버깅하기 매우 어렵게 만들 수 있습니다. 따라서 개발자들은 높은 우선순위의 real-time 코드를 작성할 때에는 많은 주의를 기울여야 합니다. 이는 스파이더맨 철학을 떠올리게 합니다: With great power comes great responsibility.

Priority inheritance는 다른 것들보다도 preemption 당할 수 있는 인터럽트 핸들러에 의해 락이 획득된다면 발생할 수 있는 우선순위 역전 문제를 해결하는데에 사용됩니다 [SRL90b]. 낮은 우선순위의 쓰레드가 락을 잡았지만, 최소 CPU 당 하나씩의 갯수를 갖는 중간 우선순위의 쓰레드들에게 preemption 당한다고 생각해 보세요. 만약 인터럽트가 발생한다면, 높은 우선순위의 IRQ 쓰레드는 중간 우선순위 쓰레드들 가운데 하나를



Figure 15.14: Priority Inversion and User Input

preemption 시킬 겁니다만, 낮은 우선순위의 쓰레드가 잡은 락을 잡기로 한 동안만 그렇습니다. 불행히도, 이 낮은 우선순위의 쓰레드는 수행이 시작되기 전까지는 락을 놓을 수가 없는데, 중간 우선순위 쓰레드들이 낮은 우선순위 쓰레드의 수행이 불가능하게 하고 있습니다. 따라서 높은 우선순위의 IRQ 쓰레드는 중간 우선순위 쓰레드들 가운데 하나가 자신의 CPU를 놓기 전까지는 그 락을 잡을 수가 없습니다. 짧게 말해서, 이 중간 웃너 순위 쓰레드가 간접적으로 높은 우선순위의 IRQ 쓰레드의 수행을 막고 있는 셈인데, 이는 고전적인 우선순위 역전의 한 예입니다.

쓰레드화된 인터럽트가 아닌 경우에는 낮은 우선순위의 쓰레드는 락을 잡고 있는 동안 인터럽트를 불능화 시켜야만 해서 중간 우선순위 쓰레드가 낮은 우선순위 쓰레드를 preemption 하는 일을 방지하므로, 이 우선순위 역전 현상은 발생하지 않음을 알아두세요.

이에 대한 우선순위 상속이라는 해결책에서, 락을 잡으려 하는 높은 우선순위 쓰레드는 자신의 우선순위를 락을 잡고 있는 낮은 우선순위의 쓰레드에게 그 락이 해제될 때까지 넘겨주게 되고, 그런 식으로 긴 시간동안의 우선순위 역전을 방지합니다.

물론, 우선순위 상속은 나름대로의 한계를 가지고 있습니다. 예를 들어, 여러분이 여러분의 어플리케이션이 우선순위 역전 문제를 완전히 막을 수 있도록 설계할 수 있다면, 여러분은 어떤 더 나은 응답시간을 얻을 것입니다 [Yod04b]. 우선순위 상속이 최악의 경우의 응답시간에 두번의 컨택스트 스위치를 추가한다는 점을 놓고 보면 이는 놀라운 일은 아닙니다. 그렇다고 하나, 우선순위 상속은 정해지지 않은 시간동안의 수행 연기를 제한된 응답시간의 증가와 교환할 수 있게 하고, 우선순위 상속의 소프트웨어 엔지니어링적 이득은 많은 어플리케이션에서 그 응답시간 비용보다 높을 겁니다.

또 다른 한계점은 특정 운영 체제의 컨택스트 내에서, 락 기반의 우선순위 역전 문제만을 해결한다는 점입니다. 이런 이유로 해결할 수 없는 우선순위 역전 문제의

한 예는 높은 우선순위 쓰레드가 낮은 우선순위의 프로세스에 의해 만들어질 메세지를 소켓에서 받으려 기다리고 있는데, 이 낮은 우선순위 프로세스들은 중간 우선순위의 CPU를 많이 사용하는 프로세스들에게 preemption 당한 경우입니다. 또한, 사용자 입력에 우선순위 상속을 적용할 때의 또 다른 잠재적 단점이 Figure 15.14에 그려져 있습니다.

마지막 한계는 reader-writer 락킹과 관계되어 있습니다. 우리가 매우 많은 수의, 예를 들어 수천개의 낮은 우선순위 쓰레드들을 가지고 있으며, 각각의 쓰레드는 특정 reader-writer 락을 읽기 권한으로 잡고 있다고 생각해 봅시다. 이 쓰레드들이 모두 최소 CPU 당 하나의 중간 우선순위 쓰레드가 존재하며, 이 중간 우선순위 쓰레드들로 인해 preemption 당했다고 생각해 봅시다. 마지막으로, 높은 우선순위 쓰레드가 깨어나고 이 reader-writer 락을 쓰기 권한을 잡으려 한다고 생각해 봅시다. 우리가 읽기 권한으로 이 락을 잡고 있는 쓰레드들의 우선순위를 얼마나 필사적으로 올려주는가에 관계 없이, 이 높은 우선순위 쓰레드가 쓰기 권한을 얻게 되기까지는 긴 시간이 될 겁니다.

이 reader-writer 락 우선순위 역전 문제에 대해서는 몇 가지 가능한 해결책들이 존재합니다:

- 해당 reader-writer 락에 대해 한번에 하나의 읽기 권한의 락 획득만을 가능하게 합니다. (이는 리눅스 커널의 -rt 패치셋에서 전통적으로 택해진 방법입니다.)
- 해당 reader-writer 락에 대해서 한번에 N 개의 읽기 권한 락 획득만을 가능하게 하는데, 이때 N 은 CPU의 갯수입니다.
- 해당 reader-writer 락에 대해서 한번에 N 개의 읽기 권한 락 획득만을 가능하게 하는데, 이 때 N 은 어떤 방식으로든 개발자에 의해 선택된 숫자입니다. 리눅스 커널의 -rt 패치셋에서도 언젠가는 이 방법을 취하게 될 좋은 기회가 존재할 겁니다.
- 높은 우선순위 쓰레드가 보다 낮은 우선순위로 수행되고 있는 쓰레드에 의해 읽기 권한으로 잡힌 reader-writer 락에 대해서 쓰기 권한을 획득하려는 것을 방지합니다. (이는 priority ceiling 프로토콜의 한 변종입니다 [SRL90b].)

Quick Quiz 15.5: 하지만 reader-writer 락에 대해서 한번에 하나의 read 권한 획득만을 허용한다면, 그건 배타적 락과 똑같은 거 아닌가요???

어떤 경우들에 있어, reader-writer 락 우선순위 역전 문제는 reader-writer 락을 RCU로 변환하는 것으로 방지하는 게 가능한데, 이는 다음 섹션에서 짧게 이야기 합니다.

```

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

Preemptible RCU 는 Section 9.5에서 이야기 된 것처럼, reader-writer 락킹의 대체로 사용될 수 있습니다 [MW07, MBWW12, McK14b]. Preemptible RCU 가 사용될 수 있는 곳에서, preemptible RCU 는 읽기 쓰레드들과 업데이트 쓰레드들이 동시적으로 수행될 수 있도록 해줌으로써, 낮은 우선순위의 읽기 쓰레드들이 높은 우선순위의 업데이트 쓰레드들 위에 어떤 종류의 우선순위 역전 시나리오를 가하는 것을 막아줍니다. 하지만, 이게 유용하게 사용되려면, 오랫동안 이어지는 RCU read-side 크리티컬 섹션을 preempt 하는게 가능하도록 할 필요가 있습니다 [GMTW08]. 그러지 않는다면, 길게 이어지는 RCU read-side 크리티컬 섹션들은 지나치게 긴 real-time 응답시간을 초래할 것입니다.

그래서 Preemptible RCU 구현 하나가 리눅스 커널에 추가되었습니다. 이 구현은 현재 RCU read-side 크리티컬 섹션 내에서 preemption 당한 태스크들의 리스트를 유지하면서 커널 내의 모든 각각의 task 의 상태를 개별적으로 추적해야 하는 필요성을 막습니다. 하나의 grace period 는 다음과 같이 종료될 수 있습니다: (1) 일단 모든 CPU 들이 현재의 grace period 시작 전부터 효과를 발휘하고 있던 모든 RCU read-side 크리티컬 섹션들을 완료했을 때, 그리고 (2) 모든 그런 이전부터 존재했던 크리티컬 섹션의 중간에 preemption 당했던 모든 task 들이 그들의 리스트로부터 그 자신을 제거했을 때. 이 구현의 간단화된 버전이 Figure 15.15에 보여져 있습니다. __rcu_read_lock() 함수는 line 1-5에, 그리고 __rcu_read_unlock() 함수가 line 7-22에 있습니다.

__rcu_read_lock() 의 line 3 은 rCU_read_lock() 함수의 중첩된 호출 횟수의 task 별 카운트를 증가시키며, line 4는 컴파일러가 RCU read-side 크리티

컬 섹션의 뒤따르는 코드가 앞의 rCU_read_lock() 보다 앞으로 컴파일러가 재배치 하는 것을 막습니다.

__rcu_read_unlock() 의 line 11에서는 중첩 수준 카운트가 1인지 확인해보는데, 달리 말하자면, 이 함수 호출이 rCU_read_unlock() 호출의 중첩된 단계 중 가장 바깥의 단계인지를 확인합니다. 그렇지 않다면, line 12에서는 이 카운트를 감소시키고, 제어는 호출자에게로 되돌아갑니다. 그렇지 않다면, 이게 rCU_read_unlock() 의 가장 바깥으로, line 14-20을 통해 제어되는 크리티컬 섹션의 마무리를 필요로 합니다.

Line 14는 컴파일러가 크리티컬 섹션 내의 코드가 rCU_read_unlock() 를 구성하는 코드와 재배치하는 것을 막습니다. Line 15는 중첩 카운트를 인터럽트 핸들러 내에 포함된 RCU read-side 크리티컬 섹션들과의 경주 상태를 막기 위해 충분히 작은 음수로 설정하고 [McK11a], line 16은 컴파일러가 이 할당문을 line 17의 특수 경우 체크와 재배치 시키는 것을 막습니다. Line 17이 특수 처리가 필요하다고 판단하면, line 18은 이 특수 처리를 하기 위해 rCU_read_unlock_special() 를 수행시킵니다.

필요한 특수 처리에는 여러 종류가 있습니다만, 우리는 RCU read-side 크리티컬 섹션이 preempt 되었을 때에 필요한 것들에만 집중하도록 하겠습니다. 이 경우, 해당 task 는 자신의 RCU read-side 크리티컬 섹션 중에 처음 preempt 되었을 때에 자신이 추가되었던 리스트로부터 자신을 제거해야만 합니다. 하지만, 이런 리스트들은 lock 으로 보호된다는 점을 알 필요가 있는데, 이는 rCU_read_unlock() 이 더이상 lockless 하지 않음을 의미합니다. 하지만, 가장 높은 우선순위의 쓰레드는 preempt 당하지 않을 것이고, 따라서, 그런 가장 높은 우선순위의 쓰레드들의 rCU_read_unlock() 은 어떤 락도 잡지 않을 겁니다. 또한, 만약 조심스럽게 구현되었다면, 락킹은 real-time 소프트웨어의 동기화를 위해 사용될 수 있습니다 [Bra11].

특수 처리가 필요하든 그렇지 않든, line 19는 컴파일러가 line 17에서의 체크와 중첩 카운트를 0으로 만드는 line 20을 재배치 하는 것을 막습니다.

Quick Quiz 15.6: Figure 15.15의 line 17에서의 t->rcu_read_unlock_special.s 의 로드 직후에 preemption 이 발생했다고 생각해 봅시다. 그렇게 되면 해당 task 가 rCU_read_unlock_special() 를 수행시키지 못하게 되어서, 자기 자신을 현재 grace period 를 막고 있는 task 들의 리스트로부터 삭제하는 것을 막아서, 그 grace period 가 무한정 길어질 수 있도록 할 수 있지 않을까요? ■

이 preemptible RCU 구현은 읽기가 대부분인 데이터 구조에게 많은 수의 읽기 쓰레드들의 우선순위 증가에 피할 수 없는 지연 없이 real-time 응답시간을 가능하게 합니다.

Preemptible spinlocks 는 리눅스 커널에서의 긴 시간 동안 잡히는 spinlock 기반의 크리티컬 섹션 때문에 -rt 패치셋에서의 중요한 부분입니다. 이 기능은 아직 메인라인에 들어가지는 못했습니다: 컨셉적으로 spinlock 을 위한 sleeplock 의 간단한 대체품이긴 하지만, 상대적으로 논란의 여지가 있는 것으로 증명되었습니다.⁶ 하지만, preemptible spinlock 은 real-time 응답시간을 수십 마이크로세컨드 아래로 내릴 필요가 있습니다.

물론, 예를 들면 최근에는 데드라인 스케줄링과 같은, 세계급의 real-time 응답시간을 이루는게 굉장히 중요한 다른 리눅스 커널 컴포넌트가 많이 존재합니다만, 이 섹션에서 언급된 것들은 -rt 패치셋과 합쳐진 리눅스 커널에서는 잘 동작하는 것처럼 보입니다.

15.4.1.2 Polling-Loop Real-Time Support

첫인상으로 보면, polling loop 의 사용은 모든 존재 가능한 운영 체제 간섭 문제들을 없앨 것처럼 보일 수 있습니다. 무엇보다도, 만약 주어진 CPU 가 커널에 들어가질 않는다면, 해당 커널은 완전히 그 그림 바깥에 있는 것이죠. 그리고 커널을 간섭하지 못하도록 두는 전통적인 방법은 단순히 커널을 갖지 않는 것이고, 많은 real-time 어플리케이션들이 실제로 bare metal (순수한 기계 그 자체)에서 수행되는데, 자세히 말해보자면 8-bit 마이크로컨트롤러 위에서 수행됩니다.

어떤 분은 하나의 CPU 에 성능이 종속적인 사용자 모드 쓰레드를 특정 CPU 위에서 수행하고 모든 간섭의 원인을 회피함으로써 현대의 운영체제 커널에서 bare-metal 성능을 얻기를 희망할 수 있을 겁니다. 비록 현실은 그보다 더 복잡하긴 하지만, Frederic Weisbecker 에 의해 진행되었고 3.10 버전의 리눅스 커널에 받아들여진 NO_HZ_FULL 구현 [Cor13b] 덕에 정말 그렇게 하는게 가능해지고 있습니다. 더도 아니고 덜도 아니고, 그런 환경을 올바르게 구성하기 위해서는 여러 OS jitter 의 가능한 원인들을 제어할 필요가 있으므로 상당한 주의가 필요합니다. 아래의 토론은 디바이스 인터럽트, 커널 쓰레드와 daemon들, 스케줄러 real-time throttling (이건 가능이지, 버그가 아닙니다!), 타이머, non-real-time 디바이스 드라이버, 커널 내의 global 동기화, scheduling-clock 인터럽트, 페이지 폴트, 그리고 non-real-time 하드웨어와 펌웨어 등을 포함하는 OS jitter 의 몇 가지 원인들의 제어를 다룹니다.

인터럽트는 OS jitter 의 많은 양의 대단한 원인입니다. 안타깝게도, 대부분의 경우에 인터럽트는 시스템이 바깥의 세계와 통신할 수 있도록 하기 위해 반드시 필

⁶ 또한, -rt 패치셋의 개발은 최근 몇년간 느려졌는데, 아마도 이미 메인라인 리눅스 커널에 존재하는 real-time 기능들이 상당히 많은 사용에 있어 충분하기 때문일 수 있습니다 [Edg13, Edg14]. 하지만, OSADL (<http://osadl.org/>) 는 -rt 패치셋에 남아있는 코드를 메인라인으로 옮기기 위한 자금을 모으기 위해 일하고 있습니다.

요합니다. 이 OS jitter 와 바깥 세계와의 연락을 유지하는 사이에서의 충돌을 해결하는 한가지 방법은 작은 수의 최소한의 일을 처리할 CPU 를 예약해 두고, 모든 인터럽트를 그 CPU 들에서 처리하도록 강제하는 것입니다. 리눅스 소스 트리의 Documentation/IRQ-affinity.txt 파일은 어떻게 디바이스 인터럽트를 특정 CPU 로 향하도록 할 수 있는지 설명하는데, 현재 2015년 초에 있어서는 다음과 같이 됩니다:

```
echo 0f > /proc/irq/44/smp_affinity
```

이 커맨드는 인터럽트 #44 를 CPU 0-3 으로 몰아넣을 겁니다. Scheduling-clock 인터럽트는 특수한 처리를 필요로 하며, 이 섹션의 뒤쪽에서 그에 대해 설명할 것임을 알아두시기 바랍니다.

OS jitter 의 두번째 원인은 커널 쓰레드와 daemon 들입니다. RCU 의 grace-period kthread (rcu_bh, rcu_preempt, 그리고 rcu_sched) 와 같은 개별적인 커널 쓰레드는 taskset 커맨드, sched_setaffinity() 시스템콜, 또는 cgroups 를 사용해 어떤 CPU 를 원하는대로 수행되는 CPU 를 지정당할 수 있습니다.

Per-CPU kthread 들은 종종 더 어려운 과제가 되는데, 어떤 때에는 하드웨어 구성과 워크로드 구성을 강제합니다. 이런 kthread 들로부터의 OS jitter 를 방지하는 것은 특정한 타입의 하드웨어가 real-time 시스템으로 포함되지 않도록 해서 모든 인터럽트와 I/O 초기화가 최소한의 일만을 하는 CPU 에서 처리되도록 하거나, 일거리들을 일을 하는 CPU 로부터 다른 곳으로 넘겨지도록 하기 위해 특수한 커널 Kconfig 나 boot 패러미터가 선택되도록 하거나, 일하는 CPU 들이 커널에 들어가지 않도록 할 것을 필요로 합니다. 구체적인 per-kthread 에 대한 조언은 리눅스 커널 소스의 Documentation 디렉토리 내의 kernel-per-CPU-kthreads.txt 에서 찾을 수 있을 겁니다.

리눅스 커널에서 real-time 우선순위로 돌아가는 CPU 에 성능이 종속적인 쓰레드에 대한 OS jitter 의 세번째 원인은 스케줄러 자신입니다. 이것은 설령 여러분의 real-time 어플리케이션에 무한루프 버그가 존재한다고 할지라도 중요한 non-realtime 일이 매 초마다 최소한 50 밀리세컨드는 할당받을 수 있도록 보장할 수 있도록 설계된, 의도적인 디버깅 기능입니다. 하지만, 여러분이 polling-loop-style real-time 어플리케이션을 수행할 때에는, 이 디버깅 기능을 꺼버릴 필요가 있을 겁니다. 이는 다음과 같이 행해질 수 있습니다:

```
echo -1 > /proc/sys/kernel/sched_rt_runtime_us
```

이 커맨드를 수행시키기 위해서는 여러분은 물론 root 로 수행시켜야 하고, Spiderman 철학을 충분히 고려해야 할 필요가 있습니다. 위험을 최소화 시키기 위한 한가지 방법은 이 단락의 앞부분에서와 마찬가지로 인터럽트와 kernel thread/daemon 들을 수행중인,

CPU 에 성능이 종속적인 real-time 쓰레드들이 수행중인 CPU 로부터 다른 곳으로 옮기는 것입니다. 또한, 여러분은 Documentation/scheduler 디렉토리 안의 것들을 충분히 읽어야 합니다. 구체적으로는 sched-rt-group.txt 파일 안의 것이 중요한데, 여러분이 CONFIG_RT_GROUP_SCHED Kconfig 패러미터로 활성화된 cgroups real-time 기능을 사용하고 있다면 특히 그런데, 그 경우에는 Documentation/cgroups 디렉토리 안의 것들도 역시 읽어야 합니다.

OS jitter 의 네번째 원인은 타이머들입니다. 대부분의 경우, 특정 CPU 를 커널 바깥에 있도록 유지시키는 것은 타이머가 해당 CPU 위에 스케줄 되는 것을 막습니다. 한가지 중요한 예외는 순환하는 타이머로, 특정 타이머 처리 부분이 같은 타이머의 다음 발생을 일으키는 경우입니다. 그런 타이머가 어떤 이유로든 특정 CPU 에서 시작되게 된다면, 그 타이머는 그 CPU 에서 주기적으로 수행되기를 계속할 것이어서 무기한적으로 OS jitter 를 일으킬 수 있습니다. 순환하는 타이머를 연기시키는 조작하지만 효율적인 방법은 CPU hotplug 를 사용해서 CPU 에 성능이 종속적인 real-time 어플리케이션 쓰레드들이 수행되는 CPU 들을 모두 꺼다가 다시 켜고 여러분의 real-time 어플리케이션을 시작하는 것입니다.

OS jitter 의 다섯번째 원인은 real-time 으로 사용될 것으로 의도되지 않은 디바이스 드라이버들로부터 제공됩니다. 오래된 규범적인 예제를 들어보자면, 2005년에 VGA 드라이버는 인터럽트를 불능화 시킨 채 프레임 버퍼를 0 으로 가득채움으로써 화면을 빈 화면으로 만들 수 있었는데, 이는 수십 밀리세컨드의 OS jitter 를 초래했습니다. 디바이스 드라이버가 포함시킨 OS jitter 를 막는 한가지 방법은 real-time 시스템에서 많이 사용되는, 그리고 따라서 그것의 real-time 버그들이 고쳐진 디바이스를 조심스럽게 고르는 것입니다. 또 다른 방법은 그런 디바이스들의 인터럽트와 그 디바이스를 사용하는 모든 코드를 특정한 최소한의 일만 하는 CPU 들에게 국한시키는 것입니다. 세번째 방법은 그 디바이스의 real-time 워크로드를 지원할 수 있는 능력을 테스트하고 모든 real-time 버그들을 고치는 것입니다.⁷

OS jitter 의 여섯번째 원인은 커널 내부의 전체 시스템 동기화 알고리즘에 의해 제공되는데, 가장 명백한 건 글로벌 TBL-flush 알고리즘입니다. 이는 memory-unmapping 오퍼레이션을 막고, 명시적으로 커널 내에서의 unmapping 오퍼레이션들을 막는 것으로 막아질 수 있습니다. 2015년 초인 오늘날, 커널 내의 unmapping 오퍼레이션들을 막는 방법은 커널 모듈의 unloading 을

막는 것입니다.

OS jitter 의 일곱번째 원인은 scheduling-clock 인터럽트와 RCU 콜백 수행에서부터 제공되어집니다. 이것들은 여러분의 커널을 NO_HZ_FULL Kconfig 패러미터를 활성화 시킨채로 빌드하고, nohz_full= 패러미터가 real-time 쓰레드들을 수행하게 될 CPU 들의 리스트를 명시하도록 한 채로 부팅함으로써 막아질 수 있습니다. 예를 들어, nohz_full=2-7 은 CPU 2, 3, 4, 5, 6, 그리고 7 을 일하는 CPU 들로 만들어서, CPU 0 과 1 을 최소한의 일만 하는 CPU 로 남겨둘 겁니다. 일하는 CPU 들은 한개가 넘는 runnable task 가 각각의 CPU 위에 존재하지 않는 한 scheduling-clock 인터럽트를 발생시키지 않을 것이고, 각각의 일하는 CPU 의 RCU 콜백들은 최소한의 일만 하는 CPU 들 가운데 하나에서 수행될 겁니다. CPU 에 하나의 runnable task 만이 존재한다는 이유로 scheduling-clock 인터럽트를 금지시킨 CPU 는 adaptive ticks mode 에 있다고 불리웁니다.

nohz_full= 부팅 패러미터에 대한 대안으로, 여러분은 여러분의 커널을 NO_HZ_FULL_ALL 로 빌드할 수 있는데, 이는 CPU 0 를 최소한의 일만 하는 CPU 로 두고 모든 다른 CPU 들을 일을 하는 CPU 로 만들게 될 겁니다. 어떤 방식이던, 시스템의 나머지 부분드리 만들어내는 필수적인 최소한의 일들이 처리되기에 충분한 최소한의 일만 하는 CPU 들이 할당되었을 수 있을 것을 보장할 필요가 있는데, 이는 충분한 벤치마크와 튜닝을 필요로 합니다.

물론, 공짜 점심은 없고, NO_HZ_FULL 역시 예외는 아닙니다. 앞서 언급되었듯이, NO_HZ_FULL 은 kernel/user 전환의 비용을 더 비싸게 하는데, 다른 프로세스 처리의 필요성과 (RCU 와 같은) 커널 서브시스템에게 전환을 알려야 하는 필요성 때문입니다. 이 방법은 또한 POSIX CPU 타이머가 활성화 된 채 수행되고 있는 프로세스를 수행하는 CPU 들이 adaptive-ticks 모드로 빠지는 것을 방지합니다. 추가적인 한계점, 트레이드오프, 그리고 설정에 대한 조언은 Documentation/timers/NO_HZ.txt 에서 볼 수 있습니다.

OS jitter 의 여덟번째 원인은 page fault 입니다. 대부분의 리눅스 구현은 메모리 보호에 MMU 를 사용하기 때문에, 이런 시스템 위에서 수행되는 real-time 어플리케이션들은 page fault 를 일으킬 수 있습니다. 여러분의 어플리케이션의 페이지를 메모리에 뮐어두기 위해서는 mlock() 과 mlockall() 시스템 콜을 사용해서 major page fault 를 막으세요. 물론, 너무 많은 메모리를 뮐어두는 것은 시스템이 다른 일을 할 수 있도록 하는 것을 막을 수도 있기 때문에 Spiderman 철학이 여기에도 적용됩니다.

OS jitter 의 아홉번째 원인은 불행하게도 하드웨어와 펌웨어입니다. 따라서 real-time 에서의 사용을 위해 설계된 시스템을 사용하는게 중요합니다. OSADL 은 시

⁷ 여러분께서 이 방법을 취한다면, 다른 사람들도 혜택을 얻을 수 있도록 그 수정사항을 업스트림으로 보내주시기 바랍니다. 여러분의 어플리케이션을 나중 버전의 리눅스 커널에 포팅해야 하게 된다면, 여러분이 그런 “다른 사람들”의 한명이 될 것이란 점을 명심해 주십시오.

```

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

스템의 장시간 테스트를 수행하므로, 그들의 웹사이트 (<http://osadl.org/>)를 참고하는게 도움이 될겁니다.

안타깝게도, 이 OS jitter 원인 리스트는 완벽할 수가 없는데, 이 리스트는 새로운 버전의 커널에 따라서 바뀔 것이기 때문입니다. CPU N 이 CPU에 성능 종속적인 사용자 모드 쓰레드를 수행한다고 할 때, Figure 15.16에 보인 커맨드들이 이 CPU가 커널에 들어간 모든 시간의 리스트를 생성할 겁니다. 물론, line 5의 N 은 정보를 얻고자 하는 CPU의 숫자로 대체되어야 하고, line 2의 1은 커널 내의 추가적인 함수 호출 수준을 보기 위해 선 증가되어야 합니다. 결과로 나오는 기록은 OS jitter의 원인을 쫓아가는데에 도움이 될 수 있을 겁니다.

볼 수 있듯이, CPU에 성능 종속적인 real-time 쓰레드를 리눅스와 같은 범용 목적의 OS 위에서 수행하면서 bare-metal 성능을 얻는 것은 자세한 부분에 고통스럽게 정도로 신경쓸 것을 필요로 합니다. 물론 자동화가 도움이 될 수 있고, 일부 자동화가 적용되었습니다만, 상대적으로 적은 수의 사용자를 놓고 보면, 자동화는 상대적으로 느리게 나타날 것으로 예상될 수 있습니다. 더도 아니고 덜도 아니고, 범용 목적의 운영체제를 돌리면서 bare-metal에 가까운 성능을 얻는 능력은 real-time 시스템의 일부 타입의 구성을 더 쉽게 할 것을 약속합니다.

15.4.2 Implementing Parallel Real-Time Applications

Real-time 어플리케이션을 개발하는 것은 넓은 범위의 주제이고, 이 섹션은 그 중 일부 영역에 대해서만 다룹니다. 그렇게 되어서, Section 15.4.2.1은 real-time 어플리케이션에서 흔히 사용되는 일부 소프트웨어 컴포넌트들을 알아보고, Section 15.4.2.2는 polling-loop 기반 어플리케이션들이 어떻게 구현될 수 있는지에 대해서 간략한 개괄을 알아보고, Section 15.4.2.3은 스트리밍 어플리케이션들에 대해 비슷한 개관을 제공하며, Section 15.4.2.4은 이벤트 기반 어플리케이션을 간단히 다룹니다.

15.4.2.1 Real-Time Components

모든 엔지니어링의 영역들에서 그러하듯이, 생산성과 신뢰성에는 튼튼한 컴포넌트들이 필수적입니다. 이 섹

션은 real-time 소프트웨어 컴포넌트들에 대한 전체 카탈로그는 아니고—그런 카탈로그는 한권의 책을 꽉 채울 겁니다—사용할 수 있는 컴포넌트들의 타입들에 대한 간략한 개관입니다.

Real-time 소프트웨어 컴포넌트들을 보기 위한 자연스런 장소는 wait-free 동기화 [Her91]를 제공하는 알고리즘들일 것이고, 실제로 lockless 알고리즘들은 real-time 컴퓨팅에 있어 매우 중요합니다. 하지만, wait-free 동기화는 한정된 시간 안에서의 진행을 보장할 뿐이고, real-time 컴퓨팅은 제한된 시간 내에서의 진행에 대한 훨씬 더 엄중한 보장을 제공하는 알고리즘을 필요로 합니다. 무엇보다도, 백년도 한정된 시간입니다만 여러분의 데드라인이 밀리세컨드 단위로 측정된다면 별 도움이 되지 않을 겁니다.

더도 아니고 덜도 아니고, atomic test and set, atomic exchange, atomic fetch-and-add, 순환형 배열에 기반한 single-producer/single-consumer FIFO queues, 그리고 다양한 쓰레드별 분할 알고리즘 등을 포함하는, 제한된 응답 시간을 제공하는 중요한 wait-free 알고리즘들이 일부 존재합니다. 또한, 최근의 연구는 lock-free 보장을 포함하는 알고리즘은⁸ 확률적인 fair scheduler와 fail-stop 버그로부터 자유로울 것을 가정하는 실제 사례에서는 wait-free와 동일한 응답시간을 제공한다는 발견을 확인했습니다 [ACHS13].⁹ 이는 곧 lock-free 스택과 큐들은 real-time 영역에의 사용이 적합함을 의미합니다.

Quick Quiz 15.7: 하지만 fail-stop 버그에도 불구하고 올바르게 동작하는게 가치있는 fault-tolerance 속성 가운데 하나 아닌가요? ■

이론적으로는 그러함에도, 실전에서는 real-time 프로그램에 락킹이 자주 사용되곤 합니다. 하지만, 더 엄격한 제약 하에서는 락 기반의 알고리즘들 역시 제한된 응답시간을 제공할 수 있습니다 [Bra11]. 이런 제약들은 다음과 같은 것들을 포함합니다:

1. Fair scheduler. 고정된 우선순위 scheduler의 일반적인 경우에, 제한된 길이의 응답시간은 가장 높은 우선순위 쓰레드들에게만 제공됩니다.
2. 워크로드를 지원하기에 충분한 대역폭. 이 제약을 뒷받침하는 하나의 구현 상의 규칙 하나는 “평범한 운영 중에는 모든 CPU의 idle 시간이 최소한 50%는 된다.” 또는, 더 형식적으로 말하자면, “제시된 부하는 워크로드가 언제든 schedule 될 수 있기에 충분할 만큼 낮아야 한다” 일 것입니다.

⁸ Wait-free 알고리즘은 제한된 시간 안에 모든 쓰레드가 진행을 이를 것을 보장하는 반면, lock-free 알고리즘은 제한된 시간 안에 최소한 하나의 쓰레드는 진행을 이를 것을 보장합니다.

⁹ 이 논문은 또한 real-time 실전을 향한 이론적 한단계 진보인 제한된 최소한의 진행의 개념을 소개합니다.

3. Fail-stop 버그의 부재.
4. 제한된 획득, 이관, 해제 응답시간을 갖는 FIFO 락킹 기능들. 다시 말하지만, 같은 우선순위 내에서는 FIFO로 동작하는 일반적인 락킹 기능들에 있어서, 제한된 크기의 응답시간은 가장 높은 우선순위의 쓰레드에게만 제공됩니다.
5. 무제한적인 우선순위 역전 현상을 막을 수 있는 어떤 방법. 이 섹션의 앞에서 언급된 priority-ceiling과 priority-inheritance 방법으로도 충분할 겁니다.
6. 락 획득의 제한된 중첩 정도. 우린 무제한적인 수의 락을 가질 수 있습니다만, 하나의 쓰레드가 한번에 그 중 몇개의 것 (이상적으로는 딱 하나) 이상을 잡으려 하지는 않을 때에만 그렇습니다.
7. 제한된 수의 쓰레드. 앞의 제약들과의 조합으로, 이 제약은 어떤 락을 잡기 위해 기다리고 있는 쓰레드의 수는 제한되어질 것임을 의미합니다.
8. 모든 크리티컬 섹션이든 제한된 시간을 가질 것. 어떤 락에든 제한된 수의 쓰레드만이 기다리고 있고 제한된 크리티컬 섹션 길이만이 존재한다고 하면, 대기 시간의 최대 길이는 제한될 겁니다.

Quick Quiz 15.8: 전 잘 모르겠지만 이 리스트 앞에 “포함한다”는 단어를 봤어요. 다른 제약도 존재하는 건가요? ■

이 결론은 real-time 소프트웨어에 사용되기 위한 상당히 많은 알고리즘과 데이터 구조들의 보고를 엽니다—그리고 오랫동안의 real-time 관습을 검증합니다.

물론, 세심하고 간단한 어플리케이션 설계 역시 굉장히 중요합니다. 세계 최고의 real-time 컴포넌트도 형편없이 만들어진 설계를 좋게 만들지는 못합니다. 병렬 real-time 어플리케이션을 위해서, 동기화 오버헤드는 설계의 핵심 컴포넌트가 되어야만 하는게 분명합니다.

15.4.2.2 Polling-Loop Applications

많은 real-time 어플리케이션들이 센서 데이터를 읽고 제어 법칙을 계산하고, 제어를 위한 출력을 내놓는 하나의 CPU에서 돌아가는 루프로 구성됩니다. 만약 센서 데이터를 제공하고 제어를 위한 출력을 받는 하드웨어 레지스터가 어플리케이션의 주소 공간으로 매핑된다면, 이 루프는 시스템콜로 접근, 조작될 수 있습니다. 하지만 Spiderman 철학을 기억하세요: 거대한 힘에는 거대한 책임이 따라오고, 이 경우에는 하드웨어 레지스터에 잘못된 참조를 함으로써 하드웨어를 벽돌로 만들어 버리는 걸 막을 책임이 따라옵니다.

이런 구성은 종종 bare metal 위에서 운영체제의 도움 (또는 간섭) 없이 이루어집니다. 하지만, 하드웨어 기능을 늘리고 자동화 수준을 늘리는 것은 소프트웨어 기능을 늘리는 동기를 제공하는데, 예를 들어 사용자 인터페이스, 기록 남기기, 보고서 만들기, 모두 운영체제에 의해 도움받을 수 있습니다.

범용 목적 운영체제의 전체 기능과 능력으로의 접근을 가지면서도 bare metal에서 수행하는 것의 이득을 대부분 가져갈 수 있는 한가지 방법은 Section 15.4.1.2에서 설명한 리눅스 커널의 NO_HZ_FULL 기능을 사용하는 것입니다. 이 기능은 리눅스 커널의 버전 3.10에서부터 사용 가능해졌습니다.

15.4.2.3 Streaming Applications

빅데이터 real-time 어플리케이션의 대중적인 것들은 다양한 출처로부터 입력을 받아들이고, 그것을 내부적으로 처리하고, 알림과 요약된 결과를 출력합니다. 이런 *streaming application*은 많은 경우 상당히 병렬적으로, 다른 출처로부터의 정보들을 동시적으로 처리합니다.

Streaming application을 구현하는 한가지 방법은 다른 처리 단계들을 연결하는데에 고밀도의 배열로 이루어진 순환형 FIFO를 사용하는 것입니다 [Sut13]. 그런 각각의 FIO는 그리로 집어넣을 원소를 생성하는 하나의 생성자 쓰레드와 그로부터 원소를 꺼내서 소비하는 (아마도 다른) 하나의 소비자 쓰레드만을 갖습니다. 합쳐지는 곳 (fan-in)과 분산되는 곳 (fan-out)은 자료구조보다는 쓰레드를 사용하며, 따라서 일부 FIFO들의 출력이 합쳐져야 한다면, 다른 쓰레드 하나가 그것들로부터 입력을 받아서 이 별개의 쓰레드가 단하나의 생산자가 되는 또 다른 FIFO에 출력을 내보내게 됩니다. 유사하게, 만약 특정 FIFO의 출력이 나뉘어져야 한다면, 별개의 하나의 쓰레드가 이 FIFO로부터 입력을 받아와서 여러개의 FIFO에 필요한 대로 출력을 내보내게 됩니다.

이 방법은 제한적으로 보일 수도 있습니다만, 이 방법은 최소한의 동기화 오버헤드를 가지고 쓰레드간의 통신이 가능하게 하며, 빠른 응답시간 제약을 지켜야 할 때에 최소한의 동기화 오버헤드는 중요합니다. 이는 각각의 단계에서 처리해야 하는 일의 양이 작아서 동기화 오버헤드가 처리 오버헤드보다 중요할 때에 특히나 그러합니다.

각각의 개별적인 쓰레드는 CPU에 성능이 제한되어 있을 수 있는데, 이 경우 Section 15.4.2.2에서의 조언이 적용될 수 있을 겁니다. 반면에, 만약 각각의 개별적인 쓰레드가 각자의 입력 FIFO로부터의 데이터를 기다리면서 멈춰서 있다면, 다음 섹션의 조언이 적용될 수 있을 겁니다.

```

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

15.4.2.4 Event-Driven Applications

우린 중간 크기의 산업용 엔진으로의 연로 투입을 event-driven 어플리케이션의 한가지 간단한 예로 사용해 보도록 하겠습니다. 일반적인 운영 조건 상에서, 이 엔진은 연료가 상자점(엔진의 타이밍 측정이 만들어지는 기준점)을 둘러싼 1도의 간격 내에 투입될 것을 필요로 합니다. 우리가 1,500-RPM 회전율을 가정한다면, 우리는 초당 25 회의 회전, 초당 9,000 도의 회전을 갖게 되는 셈이며, 이는 곧 1도 회전당 111 마이크로세컨드로 이야기할 수 있습니다. 따라서 우리는 약 100 마이크로 세컨드의 시간 간격 내에서 연료 투입을 스케줄 해야 합니다.

여러분이 엔진을 만들고 있긴 하지만, 연로 투입을 초기화 하기 위해 시간 기준의 대기가 사용되었다고 생각해 보면, 전 여러분이 회전 센서를 제공하기 바랍니다. 우린 시간 기준의 대기 기능의 기능성을 테스트 해야하는데, 아마도 Figure 15.17에 보인 것과 같은 테스트 프로그램을 사용할 수 있을 겁니다. 안타깝게도, 이 프로그램을 돌리면, 설령 -rt 커널을 사용한다 해도 우린 받아들이기 어려운 timer jitter를 겪게 될 겁니다.

여기서의 한가지 문제는 CLOCK_REALTIME이 충분히 이상하게도, real-time에서의 사용을 위한 의도로 만들어진 게 아니란 점입니다. 그게 아니라, 이는 프로세스나 쓰레드에 의해 소비된 CPU 시간의 양에 반대되는 의미로써의 “realtime”을 의미합니다. Real-time에서의 사용을 위해서라면, 여러분은 이대신에 CLOCK_MONOTONIC을 사용해야 합니다. 하지만, 이렇게 변경하더라도, 결과는 여전히 받아들일만하지 못할 겁니다.

또 다른 문제는 이 쓰레드는 sched_setscheduler() 시스템콜을 사용해서 real-time 우선순위로 격상되어야 한다는 점입니다. 하지만 이 변경을 하더라도 충분치 않은데, 여전히 우린 page fault를 만날 수 있기 때문입니다. 우린 또한 어플리케이션의 메모리를 고정시켜서 page fault를 막기 위해 mlockall() 시스템콜을 사용해야 합니다. 이 변경들을 모두 가하면, 그 결과는 드디어 받아들일만 할 겁니다.

다른 환경에서는 추가적인 조정이 필요할 수 있습니다. 시간에 민감한 쓰레드들을 각자의 CPU로 볼도록 만들어야 할 수도 있고, 인터럽트를 그런 CPU들로부터 떨어뜨려야 할 수도 있습니다. 조심스럽게 하드웨어와 드라이버를 골라야 할 수도 있고, 커널 설정을 잘 설정해야 할 가능성이 높습니다.

이 예제에서 볼 수 있듯이, real-time 컴퓨팅은 상당히 어려울 수 있습니다.

15.4.3 The Role of RCU

기온, 습도, 기압에 의해 변화될 수 있는 점진적 변화를 겪는 데이터에 접근할 필요가 있는 병렬 real-time 프로그램을 작성하고 있다고 생각해 봅시다. 이 프로그램에서의 real-time 응답 제약은 매우 가혹해서 스핀닝을 하거나 블록되는 것을 허용할 수 없고, 따라서 락킹을 상요할 수도 없고, 재시도 루프를 사용하는 것도 허용될 수 없어서 시퀀스 락이나 해저드 포인터를 사용할 수도 없습니다. 다행히도, 기온과 기압은 일반적으로 제어되고, 따라서 기본의 하드 코딩 된 데이터 집합만으로도 보통은 충분합니다.

하지만, 온도, 습도, 그리고 기압은 때때로 기본값으로부터 너무 크게 변화되기도 하고, 그런 경우에는 기본값들을 대체할 수 있는 데이터를 제공해야 합니다. 기온, 습도, 그리고 기압은 점진적으로 변화되기 때문에, 업데이트된 값을 제공하는 건 몇분 내로 이뤄져야만 하긴 하지만 긴급한 성격의 문제는 아닙니다. 해당 프로그램은 cur_cal이라 가상으로 이름지어진, 평소에는 정적으로 할당되고 기본의 측정값들이 가상의 a, b, c란 이름의 필드들에 초기값으로 저장되어 있는 default_cal을 가리키는 글로벌 포인터를 사용할 겁니다. 그렇지 않다면, cur_cal은 동적으로 할당되고 현재의 측정값들을 제공하는 구조체를 가리키게 됩니다.

Figure 15.18는 이 문제를 해결하는데 RCU가 어떻게 사용될 수 있는지 보입니다. 탐색들은 line 9-15의 calc_control()로 보여진 것처럼 결정적이며, real-time 요구사항을 일관적으로 지킵니다. 업데이트들은 line 17-35의 update_cal()에 보여진 것처럼 좀 더 복잡합니다.

Quick Quiz 15.9: Real-time 시스템들은 많은 경우 safety-critical 어플리케이션들에 사용된다는 점을 놓고 보면, 그리고 수행시간 메모리 할당은 많은 safety-critical 상황에서 숨겨진다는 점을 생각해보면, malloc() 호출은 뭐때때 있는거죠???

Quick Quiz 15.10: update_cal()을 보호하기 위해 어떤 동기화가 필요하지 않나요?

이 예는 RCU가 어떻게 real-time 프로그램들에 결정론적 read-side 데이터 구조를 제공할 수 있는지를 보입니다.

```

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

15.5 Real Time vs. Real Fast: How to Choose?

Real-time 과 real-fast 컴퓨팅 사이에서의 선택은 어려울 수 있습니다. Real-time 시스템은 종종 real-time 이 아닌 컴퓨팅 쪽에 처리량을 떨어뜨릴 수 있기 때문에, 필요치 않은 곳에 real-time 을 사용하는 것은 Figure 15.19 에 보여진 것처럼 문제가 될 수 있습니다. 그 정도는 적어도 여러분의 상사에게 미안함을 느끼게 만들기에는 거의 충분합니다!

경험에 따른 한가지 규칙은 여러분의 선택을 돋기 위해 다음의 네가지 질문을 사용합니다:

1. 평균적인 긴 시간동안의 처리량만이 유일한 목표인가?
2. 커다란 부하가 응답시간을 악화시키는게 받아들일 만한가?
3. mlockall() 시스템콜의 사용을 필요로 할만큼 높은 메모리 부하가 존재하는가?
4. 여러분의 어플리케이션의 기본적인 일감 항목이



Figure 15.19: The Dark Side of Real-Time Computing



Figure 15.20: The Dark Side of Real-Fast Computing

완료되는데에 100 밀리세컨드 이상을 필요로 하는가?

이 질문들에 대한 대답들 중 하나라도 “그렇다”라면, 여러분은 real-time 보다는 real-fast 를 선택해야 하고, 그렇지 않다면, real-time 이 필요할 겁니다.

현명한 선택을 하시고, real-time 을 선택했다면, 여러분의 하드웨어, 펌웨어, 그리고 운영체제가 그 일을 수행하기에 적당함을 분명히 하세요!

Chapter 16

Ease of Use

16.1 What is Easy?

“쉽다”는 건 상대적인 용어입니다. 예를 들어, 많은 사람들은 15시간의 비행을 약간 괴로운 체험일 것이라 여깁니다—그들이 대체할 수 있는 이동수단, 특히 수영과 같은 것을 고려하는 것을 멈추지 않는다면 말이죠. 이는 사용하기 편리한 API를 만드는 것은 여러분의 의도된 사용자들을 상당히 잘 알아야 함을 의미합니다.

다음의 질문은 이 지점을 잘 설명합니다: “오늘날 살아가는 사람들 가운데 무작위적으로 선택된 사람을 하나 놓고 생각할 때, 어떤 변경이 그의 또는 그녀의 삶을 개선시킬까요?”

모든 사람의 삶을 도울 것이라 보장되는 하나의 변화는 존재하지 않습니다. 무엇보다도, 굉장히 다양한 부류의 사람들이 연관된 다양한 범위의 필요, 원하는 것, 소망, 그리고 열망을 가지고 존재하고 있습니다. 굶어 죽어가고 있는 사람은 음식을 필요로 할 것입니다만, 병적으로 비만인 사람에게 음식을 더 주는 것은 그의 죽음을 앞당길 수도 있습니다. 많은 젊은 사람들에 의해서 열광적으로 소망되는 높은 수준의 짜릿함은 심장마비로부터 회복되고 있는 누군가에게는 치명적일 수 있습니다. 한 사람의 성공에 중요한 정보는 너무 많은 정보의 부하로 고통받고 있는 누군가에게는 실패를 줄 수도 있습니다. 짧게 말해서, 여러분이 전혀 모르는 누군가를 돋기 위한 목적의 소프트웨어 프로젝트를 위해 일하고 있다면, 여러분은 그 누군가가 여러분의 노력에 덜 감명받더라도 놀라지 않아야 합니다.

여러분이 정말로 어떤 집단의 사람들을 돋고자 한다면, 그들과 일정한 기간의 시간동안 그들과 가까이 작업을 하는 것을 대체할 수는 없습니다. 더도 아니고 덜도 아니고, 여러분의 사용자들이 여러분의 소프트웨어로 인해 행복해지는 정도를 늘리기 위해서 여러분이 할 수 있는 몇 가지 간단한 것들이 존재하고, 그 중 일부는 다음 섹션에서 다뤄집니다.

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.

With apologies to any Kathleen Turner fans who might still be alive.

16.2 Rusty Scale for API Design

이 섹션은 Rusty Russell의 2003 Ottawa Linux Symposium 키노트 [Rus03, Slides 39-57]에서 가져와졌습니다. Rusty의 주장의 핵심은 목표는 API를 단순히 사용하기 편하게 만드는 게 아니라, API가 잘못 사용되기 어렵게 해야 한다는 것입니다. 그런 결론 하에서, Rusty는 잘못 사용하기 어렵게 하는 속성을 중요한 순서로 내림차순으로 정리한 “Rusty Scale”을 제안했습니다.

다음의 리스트는 Rusty Scale을 리눅스 커널에 일반화 시킵니다:

1. 잘못 사용되는 것은 불가능합니다. 이건 모든 API 디자이너들이 노력해야 할 표준이긴 합니다만, 가상의 `dwim()`¹ 커맨드가 이를 안지킨, 잘못된 예에 가깝습니다.
2. 컴파일러나 링커는 여러분을 잘못되게 두지 않습니다.
3. 컴파일러나 링커는 여러분이 잘못한 경우에는 경고를 합니다.
4. 가장 간단한 사용법이 올바른 사용법입니다.
5. 이름은 여러분이 그걸 어떻게 사용해야 하는지 알려줘야 합니다.
6. 올바르게 사용하지 않는다면 그것은 수행시간에 항상 깨져야 합니다.
7. 일반적인 관례를 따라야 하며, 그렇게 하면 여러분은 올바른 동작을 얻게 될겁니다. 메모리 할당을 잘못하기는 쉽지만, 수많은 프로젝트들이 적어도 대부분의 경우에는 메모리 할당을 올바르게 관리합니다. `malloc()`을 Valgrind [The11]와 함께 사용하는 행위는 `malloc()`을 거의 “올바르게 사용

¹ `dwim()` 함수는 “do what I mean (내가 의미하는 걸 하라)”의 약자입니다.

- 해야 하며, 그러지 않는다면 수행시간에 항상 깨지는” 지점까지 옮겨 놓을 겁니다.
8. 문서를 읽으면 여러분은 올바른 동작을 얻을 것입니다.
 9. 구현을 들여다보면 여러분은 올바른 동작을 얻을 것입니다.
 10. 올바른 메일링 리스트 기록을 읽으면 여러분은 올바른 동작을 얻을 것입니다.
 11. 올바른 메일링 리스트 기록을 읽으면 여러분은 잘못된 동작을 얻을 것입니다.
 12. 구현을 들여다보면 여러분은 잘못된 동작을 얻을 것입니다. `rcu_read_lock()` 의 `CONFIG_PREEMPT` 없을 때의 원래 구현 [McK07a]은 이 요점의 유명하지 않은, 하나의 예입니다.
 13. 문서를 읽으면 여러분은 잘못된 동작을 얻을 것입니다. 예를 들어, DEC Alpha `wmb` 인스트럭션의 문서 [SW95]는 이 인스트럭션이 실제로 하는 것에 비해 더 많이 엄격한 메모리 순서 의미를 갖는다고 개발자들이 생각하게끔 만들었습니다. 나중의 문서는 이 점에 대해서 설명해서 [Com01], `wmb` 인스트럭션을 “문서를 읽으면 여러분은 올바른 동작을 얻을 것입니다” 지점으로 옮겼습니다.
 14. 일반적인 관계를 따르면 여러분은 잘못된 동작을 얻을 것입니다. `printf()` 명령은 이런 요점의 한 예인데, 개발자들은 거의 항상 `printf()` 의 여러 리턴을 체크하지 않기 때문입니다.
 15. 올바르게 수행한다면 그것은 항상 수행시점에서 깨질 겁니다.
 16. 이름은 여러분이 그것을 어떻게 사용해선 안되는지 설명해 줘야 합니다.
 17. 분명한 사용은 잘못된 것입니다. 리눅스 커널의 `smp_mb()` 함수는 이 요점의 한 예입니다. 많은 개발자들이 이 함수는 실제로 그것이 가지고 있는 것에 비해 강한 순서 규칙을 가지고 있을 것이라 가정합니다. Section 14.2는 이런 실수를 막기 위해 필요한 정보를 담고 있으며, 리눅스 커널 소스 트리의 Documentation 디렉토리 역시 그런 정보를 담고 있습니다.
 18. 컴파일러나 링커는 여러분이 올바른 일을 한다면 경고를 해줄 겁니다.
 19. 컴파일러나 링커는 여러분이 올바른 일을 하도록 놔두지 않을 겁니다.
 20. 올바른 일을 하는 것은 중요합니다. `gets()` 함수는 이 요점의 하나의 유명한 예입니다. 실제로, `gets()`는 무조건적인 버퍼 오버플로우 보안 구멍으로 설명될 수 있을 겁니다.

16.3 Shaving the Mandelbrot Set

유용한 프로그램들은 (Figure 16.1에 보인) Mandelbrot 집합과 깔끔하게 떨어지는 부드러운 외곽을 갖지 않는다는 점에서 닮았습니다—만약 그런 외곽을 갖는다면, halting problem이 풀릴 수 있을 겁니다. 하지만 우린 진짜 사람이 사용할 수 있는 API가 필요하지, 각각의 모든 잠재적 사용에 Ph.D. 학위 논문이 필요한 것들을 원하는게 아닙니다. 따라서, 우리는 “Mandelbrot 집합을 깎아내서”,² API의 사용을 잠재적 사용의 전체 집합 중 쉽게 설명될 수 있는 부분집합으로 제한시켜야 합니다.

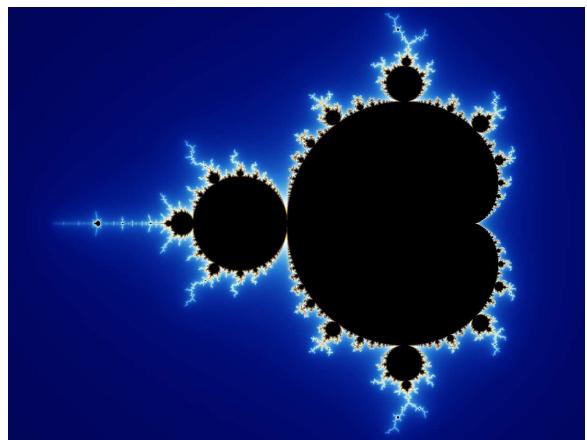


Figure 16.1: Mandelbrot Set (Courtesy of Wikipedia)

그런 깎아냄은 비생산적으로 보일 수도 있습니다. 무엇보다, 한 알고리즘이 동작한다면, 왜 그걸 사용하지 말아야 할까요?

최소한 일부의 깎아내기는 분명히 필요한 이유를 보기 위해서, 데드락을 예방하는, 하지만 가능한 가장 나쁜 방법을 사용하는 락킹 설계를 생각해 봅시다. 이 설계는 순환식 이중 링크드 리스트를 사용하는데, 이 리스트는 시스템 상의 각각의 쓰레드를 위해 하나의 헤더 원소와 함께 하나의 원소를 유지합니다. 새로운 쓰레드가 시작되면, 그 부모 쓰레드는 이 리스트에 새로운 원소를 집어넣어야만 하는데, 이는 어떠한 종류의 동기화를 필요로 합니다.

² Josh Triplett 덕에.

이 리스트를 보호하는 한가지 방법은 글로벌 락을 사용하는 것입니다. 하지만, 이는 쓰레드들이 빈번하게 생성되고 소멸된다면 병목지점이 될 수도 있습니다.³ 또 다른 방법은 해시 테이블을 사용하고 개별적인 hash bucket들을 락으로 관리하는 것입니다만, 이는 리스트를 원소 순서대로 스캔할 때에는 나쁜 성능으로 동작할 겁니다.

세번째 방법은 개별의 리스트 원소들을 락으로 보호하고, 원소의 삽입 시에 해당 원소의 앞쪽 원소와 뒤쪽 원소 둘 다의 락을 잡는 겁니다. 두 락이 모두 잡혀야 하므로, 그것들을 어떤 순서로 잡을 것인지를 우리는 결정해야 합니다. 그런 두개의 일반적인 방법은 락들을 주소순서로 잡거나 그것들이 리스트에 보여지는 순서대로 잡아서, 락으로 잡겨지는 두개의 원소 중 하나가 잡겨질 때 그 헤더가 항상 먼저 얻어지도록 하는 것입니다. 하지만, 이 방법들 모두 특별한 경우 검사와 분기를 필요로 합니다.

여기서 깎아내져야 하는 해결책은 락들을 무조건적으로 리스트 순서로 잡는 것입니다. 하지만 데드락은 어떡하죠?

데드락은 일어날 수 없습니다.

이를 알아보기 위해, 리스트의 원소를 헤더를 0으로, (순환형 리스트이므로, 헤더의 앞쪽 원소인) 리스트의 마지막 원소를 N 으로 하는식으로 번호를 매깁시다. 비슷하게, 쓰레드들을 0부터 $N - 1$ 까지로 번호매깁시다. 만약 각각의 쓰레드가 원소들 가운데 연속적인 두개의 원소들의 락을 잡으려 한다면, 최소한 하나의 쓰레드는 두 락들을 모두 잡을 수 있도록 보장될 겁니다.

왜죠?

리스트의 모든 것들에 닿을 만큼 쓰레드들의 수가 충분히 많지 않기 때문입니다. 쓰레드 0 가 원소 0 의 락을 잡는다고 생각해 봅시다. 이 쓰레드가 블락되기 위해서는, 어떤 다른 쓰레드가 이미 원소 1 의 락을 잡았어야 하는데, 쓰레드 1이 그리고 있다고 가정해 봅시다. 비슷하게, 쓰레드 1 이 블락되기 위해서는 어떤 다른 쓰레드가 원소 2 의 락을 잡고 있어야 하는데, 이런식으로 쓰레드 $N - 1$ 까지 연속되는데, 이 쓰레드는 원소 $N - 1$ 의 락을 획득합니다. 쓰레드 $N - 1$ 이 블락되기 위해서는 어떤 다른 쓰레드가 원소 N 의 락을 잡고 있어야 합니다. 하지만 더이상 쓰레드가 존재하지 않고, 따라서 쓰레드 $N - 1$ 은 블락될 수가 없습니다. 따라서, 데드락은 일어나지 않습니다.

그러니 우리가 왜 이 마음에 드는 작은 알고리즘의 사용을 막아야 합니까?

여러분이 정말로 그걸 사용하고자 한다면, 우리는 여러분을 막을 수 없는게 사실입니다. 하지만, 우리는 우

리가 신경쓰는 어떤 프로젝트에도 그런 코드가 들어오지 못하도록 추천할 수 있습니다.

하지만, 여러분이 이 알고리즘을 사용하기 전에, 다음의 Quick Quiz 를 한번 생각해 주시기 바랍니다.

Quick Quiz 16.1: 원소를 지울 때에도 비슷한 알고리즘을 사용할 수 있습니까? ■

이 알고리즘은 극도로 특수화 되어 있으며 (이 알고리즘은 특수한 크기의 리스트에 대해서만 동작합니다), 또한 상당히 취약한 게 사실입니다. 리스트에 노드를 하나 추가하는데 의도치 않게 실패하게 하는 어떤 버그든 데드락을 초래할 수 있습니다. 사실, 노드를 약간이라도 너무 늦게 추가하는 것만으로도 데드락을 초래할 수 있습니다.

또한, 앞에서 설명한 다른 알고리즘들이 “좋고 충분합니다”. 예를 들어, 단순히 락들을 주소 순서대로 잡는 것은 상당히 간단하고 빠르면서도 어떤 크기의 리스트든 사용할 수 있게 합니다. 다만 비어있는 리스트와 하나의 원소만을 포함하고 있는 리스트의 특수한 경우들에 대해서만 조심하세요!

Quick Quiz 16.2: 그러면요! 이처럼 깎여져나갈 가치가 있는 알고리즘을 가지고 나타난 누군가를 포용해 본 적 있나요??? ■

정리하자면, 우린 그저 동작한다는 이유만으로 알고리즘을 사용하지는 않습니다. 그대신에 우리는 배울 가치가 충분한 알고리즘들만 사용하도록 우리 자신을 제한합니다. 더 어렵고 더 복잡한 알고리즘일수록, 그걸 배우고 그 버그를 고치는데 드는 고통이 가치가 있을 수 있도록 더 일반적으로 유용해야만 합니다.

Quick Quiz 16.3: 이 규칙에 대한 예외를 대 보세요. ■

예외는 차치하고, 우리는 우리의 프로그램들이 Figure 16.2 에 보여진 것처럼 유지가능한 상태를 유지할 수 있도록 소프트웨어를 깎아내기를 계속해야만 합니다.

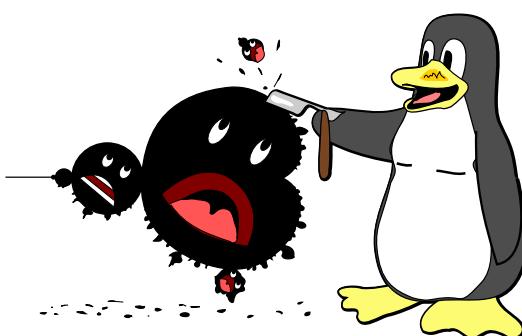


Figure 16.2: Shaving the Mandelbrot Set

³ 운영체제에 대한 많은 배경지식을 가지고 있는 여러분이라면, 이 말을 불신하는 걸 잠시 멈춰주세요. 이 말에 대한 불신을 잠시 멈추는게 불가능하다면, 우리에게 이보다 더 나은 예를 보내주세요.

Prediction is very difficult, especially about the future.

Niels Bohr

Chapter 17

Conflicting Visions of the Future

이 챕터는 병렬 프로그래밍의 미래에 대한 일부 서로 다른 전망들을 보입니다. 이것들 중 어느것이 현실이 될지, 정확히는 이것들 중 하나라도 현실이 될지는 명확치가 않습니다. 이것들은 더도 아니고 덜도 아니고 각자를 좋아하는 지지자들이 있기 때문에 중요하며, 충분히 많은 사람들이 이중 무언가를 열렬하게 믿는다면, 여러분은 그것의 지지자들의 생각, 말, 그리고 행위에서의 영향의 형태로 최소한 그것의 존재의 그림자를 다뤄야 할 겁니다. 그와는 별개로, 이 전망들 중 두개 이상의 것들이 실제 현실이 될 수도 있습니다. 하지만 대부분은 가짜입니다. 뭐가 진짜고 뭐가 가짜일지 말할 수 있다면 여러분은 부자가 될겁니다 [Spi77].

따라서, 뒤의 섹션들은 트랜잭션 메모리, 하드웨어 트랜잭션 메모리, 그리고 병렬 함수형 프로그래밍에 대한 개괄을 제공합니다. 하지만 먼저, 예언에 대한 경고적인 이야기를 2000년대 초로부터 가져와 봅니다.

17.1 The Future of CPU Technology Ain't What it Used to Be

과거란 많은 기간의 경험을 거친 렌즈를 통해 보기에는 항상 매우 간단하고 순수해 보입니다. 그리고 2000년대 초는 Moore's Law 가 그땐 전통적이었던 CPU 클락 주파수의 증가를 가져오던 현상이 깨지기 시작하는 시점에 임박했던, 순수했던 시대였습니다. 아, 그때도 기술의 한계에 대한 가끔의 경고는 있었습니다만 그런 경고들은 수십년째 이어져 오고 있었습니다. 그걸 마음에 둔 채로, 다음의 시나리오들을 고려해 봅시다:

1. 유니프로세서 Über Alles (Figure 17.1),
 2. 멀티쓰래드 매니아 (Figure 17.2),
 3. 더 많은 같은것들 (Figure 17.3), 그리고
 4. 메모리 장벽에 부딪치는 것들 (Figure 17.4).
- 다음의 섹션들은 이 시나리오들 각각을 다룹니다.

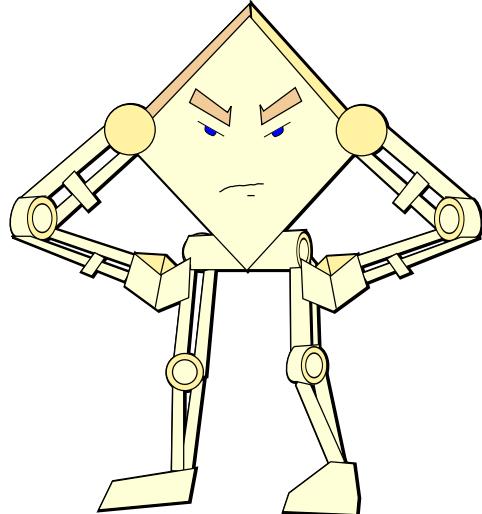


Figure 17.1: Uniprocessor Über Alles

17.1.1 Uniprocessor Über Alles

2004년에 이야기한 것 [McK04] 처럼:

이 시나리오에서, Moore's-Law 를 통한 CPU 클락 속도의 증가와 수평적으로 확장되는 컴퓨팅의 계속된 발전의 조합은 SMP 시스템들을 무의미하게 만듭니다. 따라서 이 시나리오는 “Uniprocessor Über Alles”, 말 그대로 다른 모든것보다 나은 유니프로세서라고 불립니다.

이런 유니프로세서 시스템들은 인스트럭션 오버헤드만이 문제가 될텐데, 메모리 배리어, cache thrashing, 그리고 cache contention 은 단일 CPU 시스템에서는 문제가 없기 때문입니다.

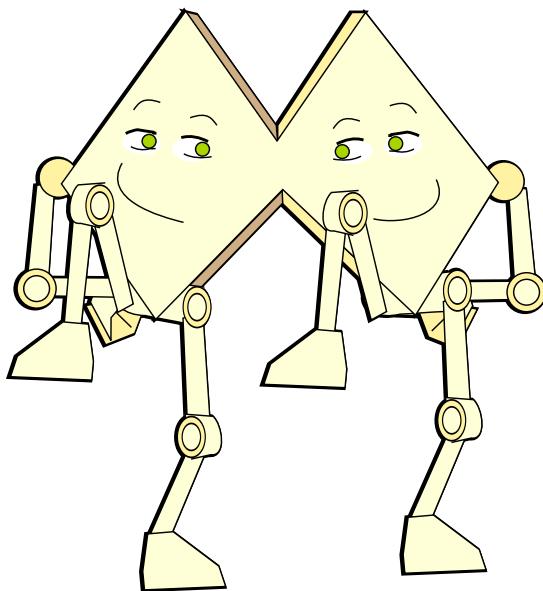


Figure 17.2: Multithreaded Mania

니다. 이 시나리오 상에서, RCU 는 NMI 들과의 상호작용과 같은 간단한 부분에서만 유용 할 것입니다. 이미 RCU 를 구현한 운영체제는 그대로 RCU 를 가지고 있어도 되겠지만, RCU 가 존재하지 않는 운영체제가 RCU 를 적용해야 할지는 분명치 않습니다.

하지만, 최근의 멀티쓰레드 사용 CPU 의 발전은 이 시나리오가 이뤄질 가능성은 적다고 이야기 합니다.

실제로 그렇게 되진 않을 겁니다! 하지만 더 커다란 소프트웨어 커뮤니티는 그들이 병렬성을 포용해야 한다는 사실을 받아들이기를 꺼려했는데, 이는 이 커뮤니티가 Moore's Law 로 인한 CPU 코어 클락 주파수 상승의 “공짜 점심” 이 정말로 끝났다는 결론을 내리기 전이었습니다. 잊지 마세요: 믿음은 감정이지, 이성적이고 기술적인 생각 과정의 결과가 아닐 수 있습니다!

17.1.2 Multithreaded Mania

역시 2004년의 이야기 [McK04] 에서:

Uniprocessor Über Alles 의 텔 그단적인 변종은 하드웨어 멀티쓰레딩을 제공하는 유니프로세서들을 포함하고, 멀티쓰레딩 기능을 제공하는 CPU 들은 오늘날 많은 데스크탑과 랩

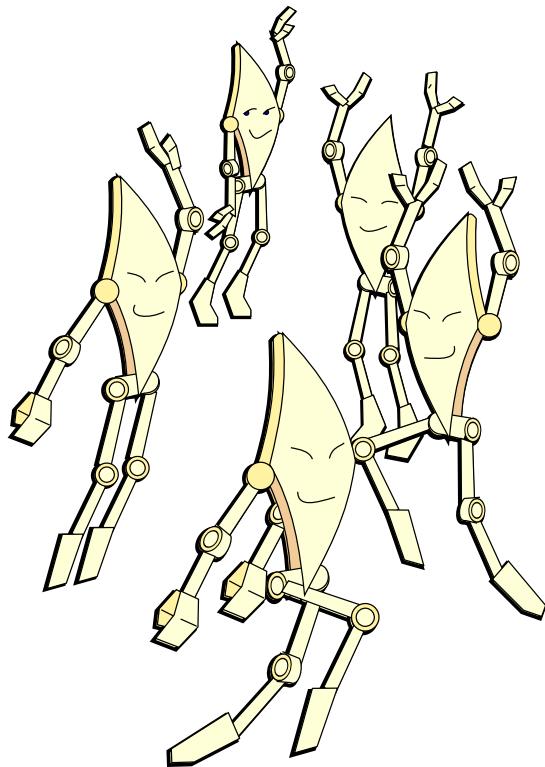


Figure 17.3: More of the Same

탑 컴퓨터 시스템들에서 표준이 되어있습니다. 멀티쓰레드 기능을 가장 적극적으로 제공하는 CPU 들은 모든 레벨의 캐시 구조를 공유하고, 그로 인해 CPU 에서 CPU 로의 메모리 접근 응답시간을 없애버려, 전통적인 동기화 메커니즘에서의 성능 페널티를 거의 없애버립니다. 하지만, 멀티쓰레딩 기능을 제공하는 CPU 는 메모리 배리어들로 인해 발생하는 경쟁과 파이프라인 stall 들로 인한 오버헤드로 부담을 겪을 겁니다. 더 나아가서, 모든 하드웨어 쓰레드들이 모든 단계의 캐시를 공유하기 때문에, 하나의 하드웨어 쓰레드에서 사용 가능한 캐시는 동일한 단일 쓰레드를 사용하는 CPU 의 것의 일부분만이 될 것이어서, 많은 캐시 사용량을 갖는 어플리케이션은 성능이 덜어질 수 있을 겁니다. 또한, 제한된 양의 캐시만이 사용 가능하다는 점이 RCU 기반 알고리즘들을 그것들의 grace-period 가 가져온 추가적인 메모리 사용으로 인한 성능 페널티를 겪게 할 가능성도 있습니다.

하지만, 그런 성능 저하를 막기 위해서는, 멀



Figure 17.4: Crash Dummies Slamming into the Memory Wall

티쓰레드 기능을 제공하는 CPU들과 multi-CPU 칩들이 적어도 하드웨어 쓰레드별 기본 위에서 캐시의 어느 단계에서는 파티션을 가져야 합니다. 이는 각각의 하드웨어 쓰레드가 사용 가능한 캐시의 양은 증가시킵니다만, 다시 하나의 하드웨어 쓰레드에서 다른 쓰레드로 전달되는 캐시라인을 위한 메모리 응답시간의 증가를 다시금 가져옵니다.

그리고 우리 모두 이 이야기가 하나의 소켓에 꽂혀진 하나의 디아 위의 멀티쓰레드 기능을 제공하는 여러개의 코어의 형태로, 어떻게 진행되었는지 압니다. 이제 질문은 미래의 공유 메모리 시스템들은 하나의 소켓에 들어맞을 것인지를 됩니다.

17.1.3 More of the Same

다시 한번 2004년의 이야기 [McK04]로부터:

More-of-the-Same 시나리오는 메모리 응답시간이 지금 날과 거의 같은 수준으로 머물 것이라는 가정 하에 세워집니다.

이 시나리오는 실제로는 변화를 나타내는데, 같은 현상이 반복된다면, 접합부의 성능이 Moore's-Law로 인한 CPU 코어 성능의 증가에도 유지되기 때문입니다. 이 시나리오에서, 파이프라인 stall, 메모리 응답시간, 그리고 경쟁으로 인한 오버헤드는 여전히 심각한 정도로 유지되고, RCU는 오늘날 그런 것처럼 높은 수준의 응용력을 유지하게 됩니다.

그리고 그 변화는 Moore's Law가 여전히 제공하고 있는대로, 높아지는 수준의 접속도입니다. 하지만 더 긴

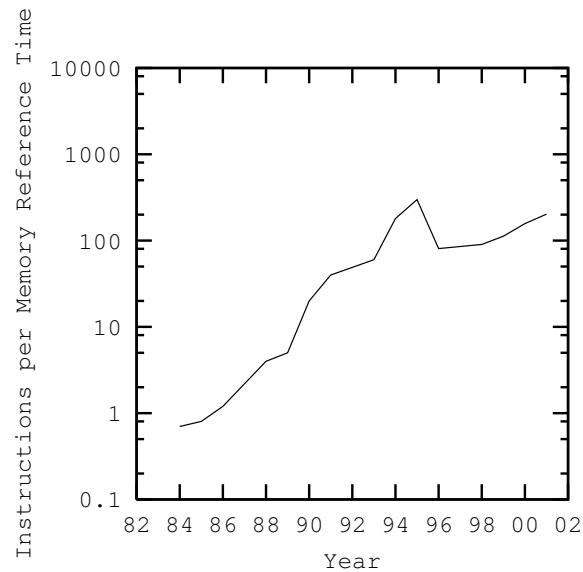


Figure 17.5: Instructions per Local Memory Reference for Sequent Computers

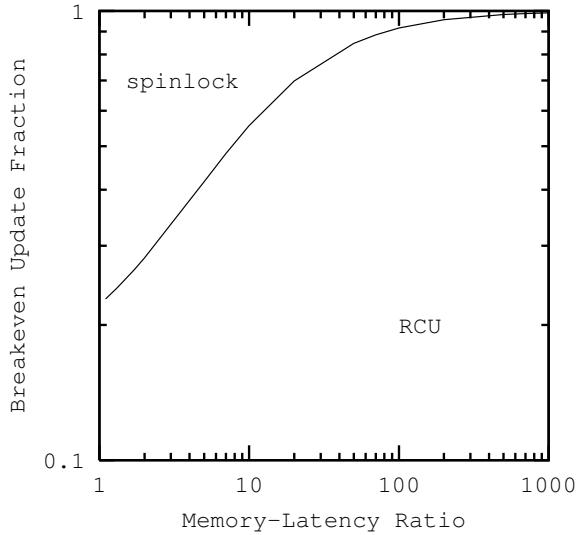
시간의 관점에서 본다면, 무엇이 될까요? 더 많은 디아당 CPU의 갯수? 아니면 I/O, 캐시, 그리고 메모리?

서버들은 앞의 것들을 선택한 것 같고, 그사이 하나의 칩 위에 올라가는 임베디드 시스템들 (SoCs)은 뒤의 것을 계속 선택해 갈 것 같습니다.

17.1.4 Crash Dummies Slamming into the Memory Wall

그리고 2004년의 이야기 [McK04]로부터 하나 더 인용해서:

Figure 17.5에 보인 메모리 응답시간 트렌드가 계속된다면, 메모리 응답시간은 인스트럭션 수행 오버헤드에 비해 커지기를 계속할 것입니다. RCU를 많이 사용하는 리눅스와 같은 시스템들은 RCU의 추가적인 사용이 Figure 17.6에 보인 것처럼 더 이득이 될것을 알게 될겁니다. 이 그림에서 볼 수 있는 것처럼, RCU가 많이 사용된다면, 증가되는 메모리 응답시간은 RCU에게 다른 동기화 메커니즘 대비 더 나은 이득을 주게 될겁니다. 반면에, RCU를 적게 사용하는 시스템들은 Figure 17.7에 보인 것처럼 더 많은 읽기 비율을 필요로 하게 될겁니다. 이 그림에서 볼 수 있듯이, RCU가 적게 사용된다면, 증가되는 메모리 응답시간 비율은 RCU를 다른 동기화 메커니즘들 대비 단점이 많아지게 합니다.

Figure 17.6: Breakevens vs. r, λ Large, Four CPUs

리눅스는 높은 부하 아래에서 grace period 당 1,600 개가 넘는 콜백들을 보았기 때문에, 리눅스는 앞의 카테고리에 속한다고 말하기 충분할 것 같습니다.

한편, 이 구절은 RCU 가 상당한 업데이트 위주의 워크로드에서 겪을 수 있는 cache warmth 문제를 설명하지 못하는데, 한편으로는 그런 워크로드에서 RCU 가 사용되지는 않을 것이기 때문입니다. 결과적으로, 이런 cache-warmth 문제가 문제시 되는 여러 경우에는 시퀀스 락킹이 그렇듯이 SLAB_DESTROY_BY_RCU 가 사용되도록 되었습니다. 한편으로는, 이 구절은 또한 RCU 가 스케줄링 응답시간을 줄이거나 보안 기능을 위해 사용되지는 못할 것이란 점을 설명하지도 않습니다.

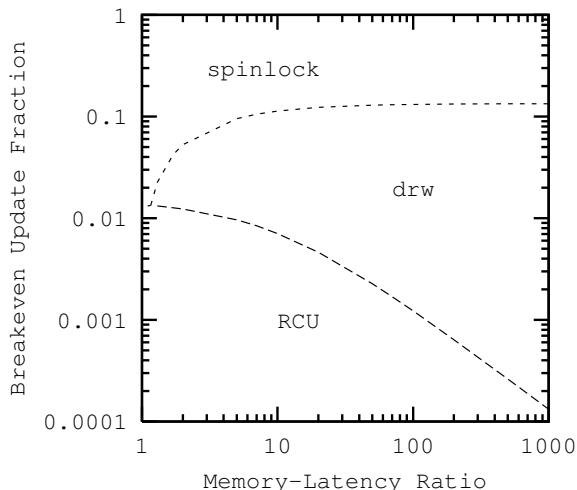
짧게 말해서, 이 챕터의 뒷부분에서 설명하는 것들을 포함해 전조들에 주의하시기 바랍니다.

17.2 Transactional Memory

데이터베이스 바깥의 영역에서 트랜잭션을 사용하자는 아이디어는 수십년 전으로 거슬러 오르는데 [Lom77], 이 아이디어는 데이터베이스에서와 데이터베이스 외에서의 트랜잭션에 대해 데이터베이스 외에서의 트랜잭션은 데이터베이스 트랜잭션을 정의하는 성질인 “ACID”에서 “D”를 빼냅니다. 하드웨어에서 메모리 기반의 트랜잭션, 또는 “트랜잭셔널 메모리”(TM)을 지원하려는 아이디어는 최근의 것입니다만 [HM93], 안타깝게도 실제 많이 사용되는 하드웨어에서 그런 트랜잭션의 지원은 그와 비슷한 것들에 대한 제안은 계속 있어왔지만 [SSHT93], 곧바로 이루어지지는 않았습니다. 그로부터 그렇게 오래지 않아, Shavit과 Touitou 는 일반적으로 사용되는 하드웨어에서 동작할 수 있는, 메모리 접근 순서를 조정해서 소프트웨어만으로 이루어진 트랜잭셔널 메모리 구현(STM)을 제안했습니다. 이 제안은 여러 해동안 절질 끌려졌는데, 어쩌면 연구자 커뮤니티들의 관심은 non-blocking 동기화(Section 14.3 을 참고하세요)에 열중되었기 때문일 수도 있습니다.

하지만 세기가 변하면서, TM 은 일부 주의의 목소리를 받기 시작했고 [BLM05, MMW07], 2000년부터 2009년 사이의 중반 쯤에는 몇가지 경고의 목소리에도 불구하고 [Her05, Gro07], 그 관심의 수준이 “열광적”이라 할 수 있게 되었습니다.

TM에 대한 기본 아이디어는 한 섹션의 코드를 어토믹하게 수행해서 다른 쓰레드가 그 중간의 상태를 볼 수 없게 하자는 것입니다. 그렇게, TM의 의미는 간단히 각각의 트랜잭션을 반복적으로 획득할 수 있는 글로벌 락의 획득과 해제로 감싸는 것으로, 성능과 확장성이 처참해지길 하겠지만, 구현될 수 있습니다. 하드

Figure 17.7: Breakevens vs. r, λ Small, Four CPUs

웨어에서든 소프트웨어에서든 TM 구현에서는 피할 수 없는 복잡성은 동시적인 트랜잭션들이 안전하게 병렬적으로 수행될 수 있는지에 대한 파악입니다. 이 파악은 동적으로 이루어지기 때문에, 충돌하는 트랜잭션들은 취소되거나 “롤백”될 수 있고, 일부 구현에서는, 이 실패한 모드가 프로그래머에게 보여질 수 있습니다.

트랜잭션 룰백은 트랜잭션 크기가 줄어듦에 따라 발생률이 줄어들 것이기 때문에, TM은 스택, 큐, 해시 테이블, 탐색 트리들과 같은 데에 사용되는 링크드 리스트 제어와 같이 작은 크기의 메모리 기반 오퍼레이션들에 매력적일 수 있습니다. 하지만, I/O와 프로세스 생성과 같은 메모리 기반이 아닌 오퍼레이션들을 포함하는 것들과 같은, 커다란 트랜잭션들에 TM을 적용하기는 어렵습니다. 다음 섹션들은 “Transactional Memory Everywhere” [McK09d]의 커다란 비전을 위해 존재하는 현재의 극복해야 할 사항들을 알아봅니다. Section 17.2.1은 바깥의 것들과 상호동작하는 데에 나타나는 문제점들을 알아보고, Section 17.2.2는 프로세스 수정 기능들과의 상호동작을 보며, Section 17.2.3은 다른 동기화 기능들과의 상호 동작들을 살펴보며, 마지막으로 Section 17.2.4에서 일부 토의와 함께 마무리 합니다.

17.2.1 Outside World

Donald Knuth의 말을 인용하면:

많은 컴퓨터 사용자들이 입력과 출력은 “진짜 프로그래밍”의 실제 부분은 아니고, 그것들은 그저 기계로 정보를 넣고 그로부터 정보를 뽑아내기 위해 (불행히도) 반드시 해야만 하는 것들이라고 느낍니다.

우리가 입력과 출력이 “진짜 프로그래밍”이라 믿든 아니라 믿든, 대부분의 컴퓨터 시스템들에 있어서, 바깥 세상과의 상호작용이 첫번째 중요도의 요구사항임은 사실입니다. 따라서 이 섹션은 트랜잭션의, I/O 오퍼레이션을 통해서든, 시간 딜레이를 통해서든, 또는 영구적 저장장치를 이용해서든 이루어지는, 상호작용을 위한 기능들에 대해 비평해 봅니다.

17.2.1.1 I/O Operations

누군가는 I/O 오퍼레이션을 락 기반의 크리티컬 섹션 안에서 할 수도, 적어도 원칙적으로는, RCU read-side 크리티컬 섹션 안에서 할 수도 있습니다. 트랜잭션 안에서 I/O 오퍼레이션을 수행하려 하면 어떻게 될까요?

여기 내포된 문제는 트랜잭션은 룰백될 수 있는데, 예를 들어 *conflict*이거나 하는 경우입니다. 대략적으로 말해서, 이는 특정 트랜잭션 내에서 일어날 수 있는 모든 오퍼레이션들이 복구될 수 있어서 해당 오퍼레이션을 두 번 행하는 것은 한번만 행한 것과 똑같은 효과를

내야 할 것을 필요로 합니다. 안타깝게도, I/O는 일반적으로는 근본적으로 돌이킬 수 없는 오퍼레이션이어서, 일반적인 I/O 오퍼레이션을 트랜잭션 내에 넣기는 어렵게 합니다. 사실, 일반적인 I/O는 돌이킬 수 없습니다: 일단 여러분이 핵탄두를 발사시키는 버튼을 눌렀다면, 되돌릴 수는 없습니다.

여기 트랜잭션 안에서 I/O를 다루기 위한 몇 가지 선택사항들이 있습니다:

1. 트랜잭션 내에서의 I/O를 메모리 상의 버퍼에 버퍼링 되는 I/O로 제약시킵니다. 이렇게 되면 이 버퍼들은 다른 메모리 위치들이 포함될 수 있는 것과 같은 방식으로 트랜잭션 내에 포함되어질 수 있을 겁니다. 이는 선택될 수 있는 메커니즘으로 보이며, 이는 실제로 stream I/O나 대용량 I/O와 같은, 많은 흔한 상황들에서 잘 동작합니다. 하지만, 복수의 레코드에 기반한 출력 스트림들이 복수의 프로세스들로부터 하나의 파일에 합쳐지는, 예컨대 “a+” 옵션과 함께 사용된 `fopen()`이나 `O_APPEND` 플래그와 함께 사용된 `open()`과 같은 경우에는 특수한 처리가 필요합니다. 또한, 다음 섹션에서 보게 되겠지만, 일반적인 네트워킹 오퍼레이션들은 버퍼링을 통해 처리될 수 없습니다.
2. 트랜잭션 안에서의 I/O를 금지시켜서 I/O 오퍼레이션을 행하려는 모든 시도는 그를 둘러싼 트랜잭션을 `abort` 시키게 만듭니다(그리고 복수의 중첩된 트랜잭션들까지도요). 이 방법은 버퍼링 되지 않은 I/O를 위한 관습적인 TM 전략처럼 보이지만, TM이 I/O를 허용할 수 있는 다른 동기화 기능들을 포함할 것을 필요로 합니다.
3. 트랜잭션 안에서의 I/O를 금지시키지만, 이 금지를 강제시키는데에 컴파일러의 협조를 받습니다.
4. 한번에 단 하나의 되돌려질 수 있는 트랜잭션 [SMS08] 만이 수행될 수 있게 허가해서, 되돌려질 수 있는 트랜잭션들은 I/O 오퍼레이션을 포함할 수 있도록 허가합니다.¹ 이는 일반적으로 동작합니다만, I/O 오퍼레이션들의 성능과 확장성을 상당히 제한합니다. 확장성과 성능이 병렬화의 첫 번째 목표라는 점을 놓고 보면, 이 방법의 일반성은 약간 스스로를 제한시키는 것으로 보입니다. 더 나쁜 것은, 되돌릴 수 있는 성질을 I/O 오퍼레이션을 막는데에 사용하는 것은 직접적인 트랜잭션 `abort` 오퍼레이션의 사용을 금지하는 것으로 보입니다.² 마지막으로, 특정 데이터 아이템을 조정하는 도돌려질 수 있는 트랜잭션이 존재한다면, 똑같은 데

¹ 이전의 문헌에서, 되돌려질 수 있는 트랜잭션들은 *inevitable* 트랜잭션이라 칭해졌습니다.

² 이 문제는 Michael Factor에 의해 지적되었습니다.

이터 아이템을 조정하는 다른 모든 트랜잭션들은 non-blocking semantic 을 가질 수가 없습니다.

5. I/O 오퍼레이션들이 트랜잭션의 하층에 들어가질 수 있는 새로운 하드웨어와 프로토콜을 만듭니다. 인풋 오퍼레이션의 경우, 하드웨어는 그 오퍼레이션의 결과를 올바르게 예측할 수 있어야 할 것이고, 그 예측이 틀렸을 경우에는 그 트랜잭션을 abort 시킬 수 있어야 할 겁니다.

I/O 오퍼레이션은 TM 의 잘 알려진 약한 부분들이고, 트랜잭션에서 I/O 를 지원하기 위한 이 문제가 합리적이고 일반적인 해결책을 가지고 있는지는 확실치 않은데, 적어도 “합리적” 이란 말이 사용 가능한 성능과 확장성을 포함한다면 그렇습니다. 더도 아니고 덜도 아니고, 이 문제에 대한 계속된 시간과 관심이 이 문제에 대한 추가적인 진보를 만들어낼 겁니다.

17.2.1.2 RPC Operations

누군가는 RPC 들을 락 기반의 크리티컬 섹션에서도, RCU read-side 크리티컬 섹션 내에서도 수행할 수 있습니다. 여러분이 RPC 를 트랜잭션 내에서 수행하려면 어떻게 될까요?

RPC 요청과 그에 대한 응답이 해당 트랜잭션 내에 포함된다면, 그리고 트랜잭션의 일부 부분이 그 응답으로 리턴되는 결과에 의존적이라면, 버퍼링을 사용하는 I/O 의 경우에 사용되었던 메모리 버퍼를 사용한 트릭은 사용할 수 없습니다. 이런 버퍼링 전략을 사용하려는 모든 시도는 트랜잭션을 deadlock 에 바지게 할 것인데, 요청은 그 트랜잭션이 성공할 것이라 보장되기 전까지는 보내질 수가 없지만, 트랜잭션의 성공 여부는 응답이 도착하기 전까지는 알 수 없을 것이기 때문에, 다음과 같은 경우가 그 예가 됩니다:

```

1 begin_trans();
2 rpc_request();
3 i = rpc_response();
4 a[i]++;
5 end_trans();

```

이 트랜잭션의 메모리 사용량은 RPC 응답이 도착하기 전까지는 결정될 수 없고, 이 트랜잭션의 메모리 사용량이 결정되기 전까지는, 이 트랜잭션이 커밋되어도 되는지 여부를 결정할 수가 없습니다. 따라서 트랜잭션의 의미론에 있어 일관적인 유일한 동작은 무조건적으로 이 트랜잭션을 abort 시키는 것으로, 이 말은, 곧 도움이 되지 않는다는 말입니다.

여기 TM 에서 사용할 수 있는 몇가지 선택들이 있습니다:

1. 트랜잭션 내에서의 RPC 를 금지시켜서, RPC 오퍼레이션을 수행하려 하는 모든 시도는 그를 둘러싼

트랜잭션을 (그리고 아마도 복수개의 중첩된 트랜잭션들도) abort 시키도록 합니다. 대안적으로, 컴파일러가 RPC 없는 트랜잭션들을 강제하도록 도움을 줄 수 있게 합니다. 이 방법은 동작합니다만, TM 이 다른 동기화 도구들과 상호작용할 것을 필요로 합니다.

2. 한번에 하나의 되돌려질 수 있는 특수한 트랜잭션 [SMS08] 만을 허용해서, 이 되돌려질 수 있는 트랜잭션은 RPC 오퍼레이션을 포함할 수 있도록 합니다. 이 방법은 일반적으로 동작합니다만, RPC 오퍼레이션들의 확장성과 성능을 상당히 제한하게 됩니다. 확장성과 성능이 병렬화의 첫번째 목표임을 상기해보면, 이 방법의 일반성은 약간 제한적인 것으로 보입니다. 더 나아가서, RPC 오퍼레이션을 받아들일 수 있는, 되돌려질 수 있는 트랜잭션들의 사용은 일단 RPC 오퍼레이션이 시작되면 손으로 작성된 트랜잭션 abort 오퍼레이션들을 배제시킵니다. 마지막으로, 특정 데이터 아이템을 조정하는 되돌려질 수 있는 트랜잭션이 존재한다면, 같은 데이터 아이템을 조정하는 모든 다른 트랜잭션들은 non-blocking semantic 을 가질 수 없습니다.
3. 트랜잭션의 성공이 RPC 응답이 도착하기 전에 결정될 수 있는 특수한 경우를 정의하고, RPC 요청이 보내지기 직전에 이것들을 자동적으로 되돌려질 수 있는 트랜잭션들로 변환시킵니다. 물론, 만약 여러 동시적 트랜잭션들이 이 방식으로 RPC 호출을 시도한다면, 이것들 중 하나만을 제외하고는 모두 롤백시켜야 할 것으로, 결과적으로 성능과 확장성이 떨어질 겁니다. 이 방법은 더도 아니고 덜도 아니고, RPC 로 마무리되는, 오랫동안 수행되는 트랜잭션들이 있을 때에는 가치가 있을 겁니다. 이 방법은 여전히 손으로 직접 하는 트랜잭션 abort 오퍼레이션들과의 문제가 존재합니다.
4. RPC 응답이 트랜잭션의 바깥으로 옮겨질 수 있는 특수한 경우들을 정의하고, 버퍼링을 사용한 I/O 에서 사용된 것과 비슷한 기법을 사용합니다.
5. 트랜잭션적인 구조가 RPC 클라이언트만이 아니라 서버도 포함하도록 확장합니다. 이건 이론적으로는 가능하며, 분산 데이터베이스들을 통해 보여졌습니다. 하지만, 분산 데이터베이스 기법을 통해 요구되는 성능과 확장성 요구사항들이 맞춰질 수 있을지는 불명확한데, 메모리 기반의 TM 은 느린 디스크 드라이브에서 나오는 느린 응답시간을 감출 수 없을 것이기 때문입니다. 물론, 솔리드 스테이트 디스크 (SSD) 의 발전과 함께, 데이터 베이스들이 그 자신의 응답시간들을 디스크 드라이브의

응답시간들에 숨겨둘 수 있을지도 불명확해지고 있습니다.

앞의 섹션에서 이야기 된 것처럼, I/O 는 TM 의 알려진 약점이고, RPC 는 그저 I/O 의 특별히 문제가 되는 경우일 뿐입니다.

17.2.1.3 Time Delays

트랜잭션 턱한 접근과의 상호작용에 관련된 중요한 특수 케이스 하나는 트랜잭션 내에서의 명시적인 시간 딜레이와 관련됩니다. 물론, 트랜잭션 내에서의 시간 딜레이이라는 이 생각은 TM 의 원자성에 위배됩니다만, 어떤 사람들은 이런 종류의 일은 완화된 원자성이 모두 그러한 것이라고 주장할 수 있습니다. 나아가서, memory-mapped I/O 와의 올바른 상호작용은 가끔 주의깊게 제어되는 타이밍을 필요로 하며, 어플리케이션들은 다양한 목적을 위해 시간 딜레이를 자주 사용합니다.

그러니, TM 은 트랜잭션 내에서의 시간 딜레이에 대해 뭘 해야 할까요?

1. 트랜잭션 내에서의 시간 딜레이를 무시합니다. 이 방법은 우아한 모습으로 보이지만, 다른 너무 많은 “우아한” 해결책들처럼, 기존 코드에 사용되는 순간 실패하게 됩니다. 크리티컬 섹션 내에서 중요한 시간 딜레이를 가지고 있을 수도 있는 그런 코드는 트랜잭션화 되는 과정에서 실패할 겁니다.
2. 시간 딜레이 오퍼레이션을 마주하는 순간 트랜잭션을 `abort` 시킵니다. 이 방법은 매력적이지만, 안타깝게도 시간 딜레이 오퍼레이션을 자동적으로 탐지하는게 항상 가능하지는 않습니다. 어떤 짧은 루프는 정말 중요한 무언가를 계산하는 루프일까요, 아니면 그저 시간이 지나가길 기다리는 걸까요?
3. 트랜잭션 내에서의 시간 딜레이를 금지시키기 위해 컴파일러의 도움을 받습니다.
4. 시간 딜레이가 평범하게 수행되도록 합니다. 안타깝게도, 일부 TM 구현은 커밋 시점에서야 수정사항을 외부에 노출시켜서, 많은 경우에는 시간 딜레이의 목적을 달성 불가능하게 할겁니다.

단 하나의 올바른 답이 있는지 여부는 분명치 않습니다. 완화된 원자성을 갖춰서 변경 사항을 트랜잭션 내에서 곧바로 외부에 노출시키는 (`abort` 시에는 이 변경들을 되돌리는) TM 구현은 마지막 대안에 의해 잘 처리될 수 있을 겁니다. 이런 경우라 하더라도, 트랜잭션의 다른 끝의 (또는, 심지어 하드웨어의) 코드는 `abort` 된 트랜잭션을 처리하기 위해 상당한 재설계가 필요할 겁니다. 이런 재설계의 필요성은 트랜잭션 메모리를 기존 코드에 적용하기를 더 어렵게 할겁니다.

17.2.1.4 Persistence

락킹 기능들에는 많은 다른 타입들이 존재합니다. 한가지 흥미로운 차이는 지속성으로, 달리 말하자면, 락이 그 락을 사용하는 프로세스의 주소 공간에 대한 의존성이 없이 존재할 수 있는가 여부입니다.

지속성 없는 락은 `pthread_mutex_lock()`, `pthread_rwlock_rdlock()`, 그리고 대부분의 커널 레벨 락킹 기능들을 포함합니다. 만약 지속성 없는 락의 데이터 구조를 담고 있는 메모리 위치가 사라지면, 락 역시 그렇게 됩니다. `pthread_mutex_lock()` 의 일반적인 사용에 있어, 이는 프로세스가 종료될 때, 그것의 모든 락들이 사라짐을 의미합니다. 이 속성은 프로그램 종료 시점에 락 정리를 대수롭지 않게 하기 위해 사용될 수도 있습니다만, 관계없는 어플리케이션들 사이에서 락을 공유하기는, 그런 공유를 위해 선 어플리케이션 사이에 메모리를 공유할 것을 필요로 하기에, 어렵게 합니다.

지속성 있는 락들은 관련없는 어플리케이션들 사이에 메모리를 공유해야할 필요를 없앱니다. 지속성 있는 락킹 API 들은 `flock` 계열인 `lockf()`, System V 세마포어, `open()`에 사용되는 `O_CREAT` 플래그 등을 포함합니다. 이 지속성 있는 API 들은 다양한 어플리케이션들을 구축하는 커다란 규모의 오퍼레이션들을 보호하는데에 사용될 수 있고, `O_CREAT`의 경우에는 심지어 운영체제 리부팅 뒤에도 살아남습니다. 필요하다면, 락들은 분산된 락 매니저와 분산 파일 시스템들을 통해서 여러 컴퓨터 시스템들에까지 미칠 수 있습니다—그리고 이 컴퓨터 시스템들의 어떤 모든 것들의 리부팅 전후에도 지속됩니다.

지속성 있는 락들은 어떤 어플리케이션을 통해서도 사용될 수 있는데, 복수의 언어와 소프트웨어 환경을 사용해 작성된 어플리케이션도 포함됩니다. 사실, 하나의 지속성 있는 락이 C 언어로 쓰여진 어플리케이션에 의해 획득되고는 Python 으로 쓰여진 어플리케이션에 의해 해제될 수도 있는 것입니다.

TM 에는 이와 비슷한 지속성 있는 기능들이 어떻게 주어질 수 있을까요?

1. 지속성 있는 트랜잭션들을 그것들을 지원하기 위해 설계된, SQL 과 같은 특수 목적 환경으로 제한 시킵니다. 이는 데이터베이스 시스템의 수십년의 역사로 보건대 분명히 잘 동작합니다만, 지속성 있는 락들에 의해 제공되는 것만큼의 유연성을 제공하지는 않습니다.
2. 일부 저장 장치나 파일시스템들에서 제공되는 스냅샷 기능들을 사용합니다. 안타깝게도, 이는 네트워크 통신을 처리하지도, 스냅샷 기능을 제공하지 않는, 예를 들어 메모리 스틱과 같은 장치로의 I/O 는 처리하지도 못합니다.

3. 타임머신을 만듭니다.

물론, 이것이 트랜잭션 메모리 라 불린다는 사실이 한숨을 돌리게 할텐데, 이 이름 자체가 지속성 있는 트랜잭션의 컨셉과 들어맞지 않기 때문입니다. 이는 가능성을 트랜잭션 메모리의 피할 수 없는 한계점을 보이는 중요한 테스트 케이스로 고려할 만큼, 딱 그만큼의 가치만 있습니다.

17.2.2 Process Modification

프로세스들은 영원하지 않습니다: 프로세스들은 생성되고 소멸되며, 메모리 매핑은 수정되고, 동적 라이브러리들과 링크되고, 디버깅 됩니다. 이 세션들은 어떻게 트랜잭션 메모리가 항상 변화되는 실행 환경을 처리할 수 있는지 알아봅니다.

17.2.2.1 Multithreaded Transactions

락을 친채, 또는, 필요하다면 RCU read-side 크리티컬 섹션 내에서 프로세스나 쓰레드를 생성하는 것은 완전히 합법적인 일입니다. 이건 합법적일 뿐만 아니라, 다음 코드 조각에서 볼 수 있듯이 상당히 간단하기도 합니다.

```
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(...);
```

이 슈도코드는 CPU 당 하나의 쓰레드를 생성하기 위해 `pthread_create()`를 사용하고, 각각의 쓰레드가 완료되기를 기다리기 위해 `pthread_join()`을 사용하는데, 모두 `pthread_mutex_lock()`의 보호 아래에서 행해집니다. 원하는 결과는 락 기반의 크리티컬 섹션을 병렬적으로 수행하는 것이고, `fork()` 와 `wait()`를 사용해에서도 비슷한 효과를 얻을 수 있을 겁니다. 물론, 이 크리티컬 섹션은 쓰레드 생성 오버헤드를 정당화하기 충분할 만큼 커야 할 것입니다만, 제품 소프트웨어에서 커다란 크리티컬 섹션들의 예는 많이 존재합니다.

TM은 트랜잭션 내에서의 쓰레드 생성에 대해서 무엇을 해줄까요?

1. 트랜잭션 내에서의 `pthread_create()`를 불법적인 것으로 규정해서, `pthread_create()` 사용의 경우 트랜잭션이 `abort` 되거나 (이게 선호됩니다) 정해지지 않은 동작을하도록 만듭니다. 대안적으로는, 컴파일러가 `pthread_create()` 가 없는 트랜잭션만을 강제하도록 도움을 받습니다.

2. 트랜잭션 내에서 `pthread_create()` 가 수행될 수 있도록 하되, 그 부모 쓰레드만이 트랜잭션의 일부로 여겨지도록 합니다. 이 방법은 이미 존재하는 TM 구현들과 합리적으로 호환될 수 있을 것처럼 보입니다만, 부주의한 사람에게는 함정이 될 것으로 보입니다. 이 방법은 몇 가지 더 질문을 떠오르게 하는데, 자식 쓰레드로의 접근의 충돌을 어떻게 처리할 것인가와 같은 것들입니다.

3. `pthread_create()` 를 함수 호출로 변환시킵니다. 이 방법 역시 매력적인 골칫거리가 되는데, 자식 쓰레드들이 서로 통신하는 드물지 않은 경우들을 처리하지 않기 때문입니다. 이 방법은 또한 부모 쓰레드가 트랜잭션을 커밋하기 전에 자식 쓰레드들을 기다리지 않는다면 어떤 일이 일어나게 될 것인지와 같은 흥미로운 질문 역시 물려일으킵니다. 더 흥미롭게도, 부모가 트랜잭션 내에 포함되어 있는 변수의 값에 기초해서 조건적으로 `pthread_join()` 을 수행한다면 무슨 일이 일어날까요? 이 질문들에 대한 답은 락킹의 경우에는 꽤 간단합니다. TM에서의 이 질문들에 대한 답은 독자 여러분의 몫으로 남겨두겠습니다.

데이터베이스 쪽에서 트랜잭션들의 병렬적 수행은 일반적이기 때문에, 현재의 TM 제안들은 그것들을 위해 제공된게 아니라 점은 놀라울 수도 있습니다. 한편으로는, 앞의 예제들은 간단한 교재에서의 예제에서는 일반적으로 찾을 수 없는, 락킹의 복잡한 사용 예여서, 그것들의 부작위함이 예상될 수도 있습니다. 그렇다고 하나, 일부 TM 연구자들이 트랜잭션 내에서의 `fork/join` 병렬성을 위해 노력하고 있다는 소문이 돌리고 있으므로, 이 주제는 조만간 더 완벽하게 다루어질 수도 있습니다.

17.2.2.2 The exec() System Call

락을 잡은채로, 또는 RCU read-side 크리티컬 섹션 내에서도 `exec()` 시스템 콜을 호출할 수 있습니다. 그 호출의 정확한 의미는 기능의 타입에 따라 정해집니다.

지속성 없는 기능들 (`pthread_mutex_lock()`, `pthread_rwlock_rdlock()`, 그리고 RCU 등)의 경우에는, `exec()` 가 성공한다면, 모든 잡혀 있던 락들을 포함해서 전체 주소 공간이 사라집니다. 물론, `exec()` 가 실패한다면, 이 주소 공간은 여전히 살아있게 되어서, 모든 연관된 락들 역시 여전히 살아 있게 됩니다. 약간 이상할 수 있겠지만, 합리적으로 잘 정의된 의미입니다.

다른 한편, 지속성이 있는 기능들 (`flock` 부류들, `lockf()`, System V 세마포어, `O_CREAT` 플래그와 함께 수행되는 `open()` 등)은 `exec()` 의 성공이나 실패

여부와 관계 없이 살아남아서, `exec()` 된 프로그램은 그것들을 놓아줘야 할 겁니다.

Quick Quiz 17.1: 메모리의 `mmap()` 리전 내의 데이터 구조체로 표현되는 지속성 없는 기능들은 어떨까요? 그런 기능들로 만들어진 크리tical 섹션 내에서의 `exec()` 가 존재한다면 어떤 일이 벌어질까요? ■

트랜잭션 안에서 `exec()` 시스템 콜을 수행하려 한다면 어떤 일이 벌어질까요?

1. 트랜잭션 내에서의 `exec()` 를 금지시켜서, `exec()` 가 호출되면 그를 둘렀나 트랜잭션들이 `abort` 되게 합니다. 이는 괜찮은 정의입니다만, TM 외의 동기화 기능들이 `exec()` 와의 관계 안에서도 잘 동작할 수 있을 것을 필요로 합니다.
2. 트랜잭션 내에서의 `exec()` 를 금지시키며, 컴파일러가 이 금지를 강제하도록 합니다. 이 방법을 취해서, 함수들이 `transaction_safe` 와 `transaction_unsafe attribute` 로 장식되도록 하는, C++ 에서의 TM 에 대한 작성중인 명세가 있습니다.³ 이 방법은 실행시간 중에 트랜잭션을 `abort` 하는 것에 비해 몇가지 장점이 있습니다만, 역시 `exec()` 와 관련해서 사용되어야 하는 TM 이 아닌 동기화 기능들을 필요로 합니다.
3. 트랜잭션을 지속성 없는 락킹 기능들과 비슷한 방식을 취급해서, `exec()` 가 실패한다면 트랜잭션이 성공하고, `exec()` 가 성공하면 말없이 트랜잭션을 커밋해 버립니다. 일부 변수들이 `mmap()` 으로 매핑된 메모리에 존재하는 경우에 대해서는(따라서 성공적인 `exec()` 시스템 콜 뒤에는 살아남들 변수들) 독자 여러분의 뜻으로 남겨두겠습니다.
4. `exec()` 시스템 콜이 성공할 것 같다면 트랜잭션을 (그리고 `exec()` 시스템 콜을) `abort` 시키고, `exec()` 시스템 콜이 실패할 것 같다면 트랜잭션을 지속시킵니다. 이는 어떤 의미에서는 “올바른” 방법입니다만, 성이 차지 않는 결과를 위해서는 상당한 작업을 필요로 할 겁니다.

아마도 `exec()` 시스템 콜은 보편적인 TM 적용성에 대한 가장 이상한 반론의 예 중 하나일 텐데, 어떤 방법이 말이 되는지가 전혀 분명치 않고, 누구가는 이건 그저 실제 삶에서 중역들과 일하는 위험의 반영일 뿐이라고 주장할 수도 있을 겁니다. 그렇다고 하나, 트랜잭션 내에서의 `exec()` 를 금지시키는 두가지 방법은 그나마 가장 논리적일 겁니다.

비슷한 문제들이 `exit()` 과 `kill()` 시스템 콜에 대해서도 존재합니다.

³ 이 명세서를 짚어준 Mark Moir 와, 그보다 앞의 버전을 알려준 Michael Wong 에게 감사를 드립니다.

17.2.2.3 Dynamic Linking and Loading

락 기반의 크리티컬 섹션과 RCU read-side 크리티컬 섹션 모두 C/C++ 공유 라이브러리나 Java 클래스 라이브러리와 같은, 동적으로 링크되고 로드되는 함수들을 수행시키는 코드를 합법적으로 담고 있을 수 있습니다. 물론, 이 라이브러리에 들어있는 코드에 대해서는 그 정의에 따라, 컴파일 타임에는 알 수 없습니다. 그러나, 동적으로 로드된 함수가 트랜잭션 안에서 수행된다면 어떤 일이 벌어질까요?

이 질문은 두개의 부분으로 구성됩니다: (1) 트랜잭션 안에서 어떻게 함수를 동적으로 링크하고 로드하는지와 (b) 이 함수 안의 알 수 없는 코드에 대해서 무엇을 해야 할까요? 공정해지기 위해, 항목 (b) 는 적어도 이론상으로는 락킹과 RCU 모두에게 일부 도전적 과제를 갖게 합니다. 예를 들어서, 동적으로 링크된 함수는 락킹에 있어 데드락을 가져올 수도 있고 (예외에 의해) RCU read-side 크리티컬 섹션 안에 quiescent state 를 가져올 수도 있습니다. 차이점은, 락킹과 RCU 크리티컬 섹션 안에서 허용되는 오퍼레이션들의 클래스는 여전히 잘 이해되어 있지만, TM 의 경우에 대해서는 상당한 불확실성이 존재한다는 점입니다. 사실, 서로 다른 구현의 TM 은 서로 다른 제한을 갖게 될 것으로 보입니다.

그러니 동적으로 링크되고 로드되는 라이브러리 함수에 대해서 TM 은 무엇을 해야 할까요? 실제 코드를 로드하는 (a) 부분의 선택사항은 다음과 같은 것들을 포함합니다:

1. 동적 링킹과 로딩을 페이지 풀트와 비슷한 형태로 취급해서, 함수가 로드되고 링크되면 해당 프로세스의 트랜잭션을 `abort` 시킵니다. 만약 그 트랜잭션이 `abort` 되면, 재시도는 그 함수가 이미 존재함을 보게 될거고, 해당 트랜잭션은 이제 정상적으로 진행될 것으로 예상될 수 있습니다.
2. 트랜잭션 안에서의 함수의 동적 링킹과 로딩을 금지시킵니다.

아직 로드되지 않은 함수 안에서의 TM 에 친화적이지 않은 오퍼레이션들을 파악해낼 수 없는 특성에 대한 (b) 부분에 대한 선택 사항은 다음과 같은 것들이 있을 수 있습니다:

1. 그냥 코드를 수행합니다: 해당 함수 안에 TM 에 친화적이지 않은 오퍼레이션들이 존재한다면, 그냥 트랜잭션을 `abort` 시킵니다. 불행히도, 이 방법은 컴파일러가 특정 트랜잭션들이 안전하게 구성되어 있는지 여부를 알 수 없게 합니다. 그에 상관없이 이 조합이 가능하게 할 수 있는 방법 한가지는 되돌이킬 수 있는 트랜잭션의 사용입니다만, 현재의 구현들은 한번에 하나의 되돌이킬 수 있는 트랜잭션의 수행만을 허용해서, 성능과 확장성을 상당히

떨어뜨릴 수 있습니다. 되돌이킬 수 있는 트랜잭션들은 직접적인 트랜잭션 abort 오퍼레이션의 사용을 제외시킬 것으로 보여집니다. 마지막으로, 특정 데이터 아이템을 조정하는 되돌이킬 수 있는 트랜잭션이 하나 존재한다면, 똑같은 데이터 아이템을 조정하는 모든 다른 트랜잭션은 non-blocking 시맨틱을 가질 수 없습니다.

- 함수 선언 부분을 어떤 함수들이 TM 친화적인지 알리도록 장식합니다. 이렇게 되면, 이 장식들은 컴파일러의 타입 시스템에 의해 강제될 수 있습니다. 물론, 많은 언어들에 있어서, 이는 연관된 시간 딜레이와 함께 언어의 확장이 제시되고, 표준화되고, 구현될 것을 필요로 합니다. 그렇다고는 하나, 그런 표준화를 위한 노력이 이미 진행 중입니다 [ATS09].
- 앞에서와 같이, 트랜잭션 안에서의 함수의 동적 링킹과 로딩을 불허합니다.

물론 I/O 오퍼레이션은 TM의 알려진 약점이고, 동적 링킹과 로딩은 I/O의 또 다른 특수 케이스 중 하나로 여겨질 수 있습니다. 더도 아니고 덜도 아니고, TM의 제안자들은 이 문제를 해결하거나, TM이 병렬 프로그래머의 도구상자의 여러 도구들 가운데 하나일 뿐이라고 스스로를 인정하게 해야 합니다. (공평을 위해 말하자면, 많은 수의 TM 제안자들이 그들 스스로는 TM 외의 것들도 갖춘 세계에 존재하는 존재 가운데 하나일 뿐임을 인정해 왔습니다.)

17.2.2.4 Memory-Mapping Operations

락 기반의 크리티컬 섹션 내에서 (`mmap()`, `shmat()`, 그리고 `munmap()` [Gro01] 과 같은) 메모리 매핑 오퍼레이션들을 사용하는건 완전히 합법적이고, 적어도 원칙적으로는, RCU read-side 크리티컬 섹션 내에서의 사용도 드립습니다. 그런 오퍼레이션을 트랜잭션 내에서 수행하려 시도하면 무슨 일이 벌어질까요? 더 나아가서, 다시 매핑된 메모리 영역이 현재 쓰레드의 트랜잭션에 사용되는 변수를 담고 있다면 어떻게 될까요? 그리고 이 메모리 영역이 다른 스레드의 트랜잭션에 사용되는 변수를 담고 있다면 어떻게 될까요?

대부분의 락킹 기능들이 락 변수들을 다시 매핑하는 행위의 결과를 정의하지 않는다는 점을 놓고 보면 TM 시스템의 메타데이터가 다시 매핑되는 케이스의 경우는 고려하지 않아도 될 것입니다.

TM에서 사용 가능한 메모리 매핑에서의 선택지들이 여기 있습니다:

- 트랜잭션 안에서의 메모리 재 매핑은 불법으로 간주되어서, 모든 메모리 재 매핑을 둘러싼 트랜잭션이 abort 됩니다. 이는 일을 단순화 시킵니다만, TM

이 크리티컬 섹션 내에서 재 매핑을 처리할 수 있는 동기화 기능들과 상호작용할 수 있을 것을 필요로 합니다.

- 트랜잭션 안에서의 메모리 재 매핑은 불법으로 간주되고, 컴파일러가 이 금지사항을 강제할 수 있도록 돕습니다.
- 트랜잭션 내에서의 메모리 매핑은 합법입니다만, 매핑된 영역 안에 변수를 가지고 있는 모든 다른 트랜잭션들은 abort 됩니다.
- 트랜잭션 내에서의 메모리 매핑은 합법입니다만, 매핑되는 영역이 현재 트랜잭션의 메모리 사용 영역과 겹친다면 그 매핑 오퍼레이션은 실패합니다.
- 트랜잭션 안에서든 밖에서든 행해지는 모든 메모리 매핑 오퍼레이션은 시스템의 모든 트랜잭션의 메모리 사용 범위와 겹쳐지는지를 체크합니다. 만약 겹친다면, 해당 메모리 매핑 오퍼레이션은 실패합니다.
- 시스템의 어떤 트랜잭션의 메모리 사용 영역과 겹치는 메모리 매핑 오퍼레이션의 영향은 TM conflict manager에 의해 결정되는데, 메모리 매핑 오퍼레이션을 실패시킬지 모든 충돌하는 트랜잭션들을 abort 시킬지를 동적으로 결정합니다.

`munmap()`은 메모리의 관련된 영역을 매핑되지 않은 채로 놔두어서, 추가적인 재미있는 영향을 가질 수 있음을 알아둘만 합니다.⁴

17.2.2.5 Debugging

브레이크포인트와 같은 일반적인 디버깅 오퍼레이션들은 락 기반의 크리티컬 섹션과 RCU read-side 크리티컬 섹션 안에서는 평범하게 동작합니다. 하지만, 초기의 트랜잭션을 메모리 하드웨어 구현 [DLMN09]에서는 트랜잭션 안에서의 exception은 그 트랜잭션을 abort 시켰는데, 이는 브레이크포인트는 그를 둘러싼 트랜잭션들을 abort 시켰음을 의미합니다.

트랜잭션은 어떻게 디버깅 될 수 있을까요?

- 브레이크포인트를 담고 있는 트랜잭션에 대해서 소프트웨어 에뮬레이션 테크닉을 사용합니다. 물론, 모든 트랜잭션의 범위 내에서 브레이크포인트가 설정될 때마다 모든 트랜잭션을 에뮬레이션 해야 할 필요가 있을 겁니다. 만약 런타임 시스템이 특정 브레이크포인트가 트랜잭션의 범위 안에 있는지 아닌지 여부를 결정할 수가 없다면, 안전한

⁴ 매핑과 매핑 해제 사이의 이 차이점은 Josh Triplett에 의해 이야기 되었습니다.

쪽으로 있기 위해 모든 트랜잭션을 에뮬레이션 해야 할 필요가 있을 겁니다. 하지만, 이 방법은 상당한 오버헤드를 가져오게 될 것이어서, 쪽고 있는 버그를 찾기 어렵게 만들 겁니다.

2. 브레이크포인트 exception 을 처리할 수 있는 하드웨어 TM 구현만을 사용합니다. 불행히도, 지금 (2008년 9월) 글을 쓰는 시점에서는, 모든 그런 구현들은 연구용 프로토타입들 뿐입니다.
3. 하드웨어 TM 구현들의 더 간단한 것들보다는 (매우 간략히 말해서) 더 exception 을 잘 처리해주는 소프트웨어 TM 구현만을 사용합니다. 물론, 소프트웨어 TM은 하드웨어 TM 보다 높은 오버헤드를 갖는 경향이 있으므로, 이 방법이 모든 상황에서 적합하지는 않을 겁니다.
4. 더 주의깊게 프로그램을 짜서, 트랜잭션 안에서는 버그가 존재하지 않는 것을 첫번째 목적으로 합니다. 이걸 어떻게 할 수 있는지 알아낸다면, 부디 모두에게 그 비밀을 공유해 주세요!

트랜잭션 메모리가 다른 동기화 메커니즘들에 비해서 더 나은 생산성을 가져다 줄 것임을 믿을 만한 몇 가지 이유가 있습니다만, 전통적인 디버깅 테크닉들이 트랜잭션에 적용될 수가 없다면 이런 개선점들은 사라질 수도 있을 듯 합니다. 특히나 트랜잭션 메모리에 익숙지 않은 사람이 커다란 트랜잭션을 트랜잭션 메모리로 사용하려 한다면 특히나 그렇게 될 가능성이 큽니다. 대조적으로, 거친 “탑 클래스의” 프로그래머들은, 특히나 작은 트랜잭션에 대해서는 그런 디버깅의 필요 자체가 없게 할 수도 있을 겁니다.

따라서, 트랜잭션 메모리가 익숙지 않은 프로그래머들에게도 생산성을 약속해 줄 수 있으려면, 디버깅 문제는 풀릴 필요가 있습니다.

17.2.3 Synchronization

어느날 트랜잭션 메모리가 모두에게의 모든 것이 된다면, 다른 동기화 메커니즘을 사용할 필요가 없어질 겁니다. 그렇게 되기 전까지는, 트랜잭션 메모리가 할 수 없는 것을 할 수 있거나 주어진 상황에서 보다 자유롭게 동작할 수 있는 동기화 메커니즘과 함께 사용되어야 할 겁니다. 다음 섹션들은 이 영역에 존재하는 현재의 challenge 들을 정리해 봅니다.

17.2.3.1 Locking

락을 잡은채로 다른 락을 잡는 것은 일반적인 일인데, 최소한, 데드락을 막기 위해 일반적으로 잘 알려진 소프트웨어 엔지니어링 테크닉이 사용된다면 이는 잘 동작합니다. RCU read-side 크리티컬 섹션 안에서 락을

잡는 것은 흔하지 않은데, RCU read-side 기능들은 락 기반의 데드락 사이클에 포함될 수가 없기 때문에 이는 데드락에 대한 고려를 줄여줍니다. 하지만 트랜잭션 안에서 락을 잡으려 한다면 무슨 일이 일어날까요?

이론적으로, 그에 대한 답은 간단합니다: 해당 락을 나타내는 데이터 구조를 트랜잭션의 한 부분으로 잡으면, 모든 것이 잘 동작합니다. 실제로는, TM 시스템의 상세한 구현에 따라서 여러개의 명확치 않은 복잡한 문제 [VGS08] 가 존재할 수 있습니다. 이런 복잡한 문제들은 해결될 수 있긴 하지만, 트랜잭션의 바깥에서 잡는 락들에 대해서는 45%의 오버헤드 증가, 트랜잭션 안에서 잡는 락들에 대해서는 300%의 오버헤드 증가를 그 비용으로 갖게 됩니다. 적은 양의 락킹을 포함하고 있는 트랜잭션을 사용하는 프로그램에서 이런 오버헤드는 받아들일 만도 하게지만, 때때로 트랜잭션을 사용하고자 하는 락 기반의 제품 수준 프로그램에서는 받아들여질 수 없을 겁니다.

1. 락킹에 친화적인 TM 구현만을 사용합니다. 불행히도, 락킹에 친화적이지 않은 구현들은 성공적인 트랜잭션을 위한 낮은 오버헤드와 극단적으로 큰 트랜잭션을 수용할 수 있는 능력과 같은 장점을 일부 갖습니다.
2. TM 을 락 기반의 프로그램에 사용할 때 “조금만” TM 을 사용해서 락킹에 친화적인 TM 구현의 한계 점들을 받아들입니다.
3. 락킹 기반의 기존 시스템을 전부 무시해서, 모든 것을 트랜잭션 기준으로 다시 구현합니다. 이 방법은 지지하는 사람이 부족하지 않습니다만, 이는 이 시리즈에서 이야기된 모든 문제들이 해결되기 를 필요로 합니다. 이 문제들을 해결하는데 걸리는 시간 동안, 경쟁상대인 동기화 메커니즘들 역시 개선될 수 있을 겁니다.
4. TxLinux [RHP⁺07] 그룹에 의해서 행해진 것처럼, TM 을 락킹 기반의 시스템에서의 최적화로만 사용합니다. 이 방법은 잘 동작할 듯 들리지만, (데드락을 막기 위한 필요와 같은) 락킹 설계 문제들을 그대로 남겨둡니다.
5. 락킹 기능들로 추가되는 오버헤드를 줄이려 노력합니다.

많은 사람에게 놀랄 수 있는 TM 과 락킹 사이를 잇는 문제가 존재할 수 있다는 사실은 실제 제품 소프트웨어에서 새로운 메커니즘과 기능들을 사용해야 함을 강조합니다. 다행히도, 오픈 소스의 진보는 연구자들을 포함해서 모두에게 많은 그런 소프트웨어가 사용 가능함을 의미합니다.

17.2.3.2 Reader-Writer Locking

락을 잡은 채로 reader-writer 락의 읽기 권한 획득을 하는 것은 잘 동작하며, 적어도 데드락을 막기 위해 잘 알려진 소프트웨어 엔지니어링 테크닉이 사용되는 곳에서는 흔한 일입니다. RCU read-side 크리티컬 섹션 아래서 reader-writer 락의 읽기 권한 획득을 하는 것 역시 동작하며, RCU read-side 기능들은 락 기반의 데드락 사이클에 포함될 수가 없기 때문에 데드락에 대한 주의를 완화시켜줍니다. 하지만 트랜잭션 안에서 reader-writer 락의 읽기 권한을 획득하려 하면 무슨 일이 벌어질까요?

안타깝게도, 트랜잭션 안에서 전통적인 카운터 기반의 reader-writer 락의 읽기 권한을 획득하려는 시도는 reader-writer 락의 목적을 왜해시킬 수 있습니다. 이를 보기 위해, 같은 reader-writer 락의 읽기 권한을 동시에 획득하려 하는 한쌍의 트랜잭션을 생각해 봅시다. 읽기 권한 획득은 reader-writer 락의 데이터 구조의 수정에 연관되기 때문에, 충돌이 날 것이고, 이는 두 트랜잭션 가운데 하나를 roll back 시킬 겁니다. 이 동작은 reader-writer 락의 목적인 동시적인 읽기 쓰레드들을 허용하는 것과 전혀 일관적이지 못합니다.

TM에서 해볼 수 있는 선택사항 몇 가지가 다음과 같습니다:

1. Per-CPU 또는 per-thread reader-writer 락킹 [HW92]을 사용해서, 특정 CPU(또는, 쓰레드)가 락의 읽기 권한 획득 시에 로컬 데이터만을 조정하도록 합니다. 이는 동시에 락의 읽기 권한을 획득하려는 두개의 트랜잭션이 충돌하는 현상을 막아서 원래 의도대로 동시에 둘 다 진행되도록 합니다. 안타깝게도, (1) per-CPU/thread 락킹에서의 쓰기 권한 획득 오버헤드는 상당히 크며, (2) per-CPU/thread 락킹의 메모리 오버헤드가 금지될 수 있으며, (3) 이 변환은 여러분이 문제가 되는 소스 코드에 접근 권한이 있을 때에만 사용할 수 있는 방법입니다. 더 최근의, 다른 확장성 있는 reader-writer 락 [LLO09]은 이런 문제들을 일부 또는 전부 없앨 수도 있습니다.
2. TM을 락 기반의 프로그램에 접목할 때 “작은 부분”에서만 TM을 사용해서 트랜잭션 안에서 reader-writer 락의 일기 권한 획득을 막습니다.
3. 락킹 기반의 기존 시스템을 완전히 분리시켜서, 모든 것을 트랜잭션으로 다시 구현합니다. 이 방법은 지지자가 부족하지는 않지만, 이는 이 글에서 설명된 모든 문제들이 해결될 것을 필요로 합니다. 이 문제들을 해결하는 시간이 지나는 동안, 다른 경쟁상대 동기화 메커니즘들이 더 나아질 가능성도 물론 존재합니다.

4. TxLinux [RHP⁰⁷] 그룹에 의해서 그랬던 것처럼 TM을 완전히 락 기반의 시스템의 최적화에만 사용합니다. 이 방법은 잘 동작할 듯 들리지만, 락킹 설계의 (데드락 방지 등과 같은) 제약점들을 그대로 둡니다. 더 나아가서, 이 방법은 여러 트랜잭션들이 같은 락에 대해 읽기 권한을 획득하려 할 때에 불필요한 트랜잭션 롤백을 초래할 수 있습니다.

물론, 배타적 락킹에 대해서도 그랬던 것처럼, TM을 reader-writer 락킹에 조합하는 것을 둘러싼 또 다른 명확치 않은 이슈들이 있을 수 있습니다.

17.2.3.3 RCU

Read-copy update (RCU)는 리눅스 커널에서 주로 사용되기 때문에, RCU와 TM을 조합하려는 학계에서의 시도는 없었을 거라고 생각할 수 있을 겁니다.⁵ 하지만, 오스틴 텍사스 대학교의 TxLinux 그룹은 다른 선택이 없었습니다 [RHP⁰⁷]. 그들은 RCU를 사용하는 리눅스 2.6 커널에 TM을 적용했기 때문에 TM이 RCU 업데이트를 위한 락킹의 자리를 대신하는 방식으로 TM과 RCU를 조합해야만 했습니다. 해당 논문은 RCU 구현의 락들(예: `rcu_ctrlblk.lock`)이 트랜잭션으로 변환되었다고 이야기 하긴 했지만, 안타깝게도 RCU 기반의 업데이트에 사용된 락들(예: `dcache_lock`)에는 무슨 일이 있었는지는 이야기 되지 않았습니다.

RCU는 읽기 쓰레드들과 업데이트 쓰레드들이 동시에 수 행될 수 있게 하며, 더 나아가서 RCU 읽기 쓰레드들은 업데이트가 진행되고 있는 데이터에도 접근할 수 있도록 한다는 점을 알아두는게 중요합니다. 물론, 성능, 확장성, real-time 응답성에서의 이득이 될 수 있는, 이런 RCU의 속성은 TM의 원자성 속성을 무시하고 동작합니다.

그러니 TM 기반의 업데이트가 동시에 RCU 읽기 쓰레드들과 상호작용할 때에는 어떻게 해야 할까요? 몇 가지 가능한 것들은 다음과 같습니다:

1. RCU 읽기 쓰레드들이 충돌하는 동시에 TM 업데이트들을 abort 시킵니다. 이는 실제로 TxLinux 프로젝트에서 취해진 방법입니다. 이 방법은 RCU semantic을 유지하고, 또한 RCU의 read-side 성능, 확장성, 그리고 real-time 응답 속성을 유지합니다만, 불행히도, 불필요하게 충돌되는 업데이트들을 abort 시키는 부작용을 갖게 됩니다. 최악의 경우에는 길게 유지되는 RCU 읽기 쓰레드들은 모든 업데이트 쓰레드들을 진행하지 못하게 만들 잠재성을 가지고 있어서 이는 이론적으로 시스템이 멎게 되는 결과를 초래할 수 있습니다. 또한, 모든 TM 구

⁵ 하지만, user-space RCU [Des09, DMS¹²]의 발전으로 커널 안에서만 사용될 것이라는 변명은 힘을 잃습니다.

현이 이런 방법에 필요한 강한 원자성을 제공하는 것은 아닙니다.

2. 충돌하는 TM 업데이트들과 동시에 수행되는 RCU 읽기 쓰레드들은 충돌하는 어떤 RCU load로부터든 과거의 (이전 트랜잭션의) 값을 가져오도록 합니다. 이는 RCU semantic과 성능을 유지하며, RCU 업데이트 쪽의 starvation을 방지합니다. 하지만, 모든 TM 구현이 수행중인 트랜잭션에 의해 임시적으로 업데이트된 변수들의 기존 값들에 대한 빠른 접근을 제공할 수 있는 건 아닙니다. 구체적으로, 기존 값들을 로그에 유지하는 (그렇게 함으로써 훌륭한 TM 커밋 성능을 제공하는) 로그 기반의 TM 구현은 이런 방법에 대해서는 즐거울 수 없을 겁니다. RCU 가 커다란 길이의 트랜잭션 구현 안에서 기존 값을 접근하는 것을 가능하게 하기 위해 아마도 `rcu_dereference()` 기능이 사용될 수 있을 겁니다만, 성능은 여전히 문제가 될 수 있을 겁니다. 더도 아니고 덜도 아니고, 이런 방법으로 쉽고 효율적으로 RCU 와 조합될 수 있는 유명한 TM 구현들이 존재합니다 [PW07, HW11, HW13].
3. RCU 읽기 쓰레드가 현재 수행중인 트랜잭션과 충돌하는 액세스를 수행한다면, 그 RCU 액세스는 충돌하는 트랜잭션이 커밋하거나 abort 되기 전까지 딜레이 되도록 합니다. 이 방법은 RCU semantic을 유지합니다만, RCU의 성능이나 real-time 응답은 유지하지 않으며, 특히나 긴시간 수행되는 트랜잭션의 존재 시에는 더욱 그렇습니다. 또한, 모든 TM 구현에서 충돌되는 액세스들을 딜레이 시키는 게 가능하지는 않습니다. 그렇다고는 하나, 이 방법은 작은 트랜잭션들만을 지원하는 하드웨어 TM 구현들에는 확실히 합리적일 것으로 보입니다.
4. RCU 읽기 쓰레드들을 트랜잭션으로 변환시킵니다. 이 방법은 RCU 가 모든 TM 구현과 완전히 호환될 것을 보장합니다만, RCU read-side 크리티컬 섹션에서의 TM의 롤백이 가능해서 RCU의 real-time 응답 보장을 지켜지지 못하게 하고, 따라서 RCU의 read-side 성능을 떨어뜨립니다. 더 나아가서, 이 방법은 RCU read-side 크리티컬 섹션들 가운데 하나라도 사용되는 TM 구현이 처리할 수 없는 오퍼레이션들을 포함하고 있는 경우에 대해서는 적절치 못합니다.
5. RCU의 많은 update-side 사용들이 새로운 데이터 구조를 공개하기 위해 하나의 포인터를 수정합니다. 이런 일부 경우들에 있어서, RCU는 이어서 롤백될 트랜잭션에 의해 포인터 업데이트를 안전하게 볼 수 있는데, 해당 트랜잭션이 메모리 순서 규칙을 지켜주는 한, 그리고 이 롤백 프로세스가 연관

된 구조체를 메모리에서 해제시키는데에 `call_rcu()`를 사용하는 한은 그렇습니다. 안타깝게도, 모든 TM 구현들이 트랜잭션 안에서의 메모리 배리어를 따르지는 않습니다. 명백하게, 이 생각은 트랜잭션들은 원자적일 것으로 여겨지기 때문에, 트랜잭션 안에서의 액세스들의 순서는 문제가 되지 않을 것으로 여겨진다는 것입니다.

6. RCU 업데이트에서의 TM의 사용을 금지시킵니다. 이는 잘 동작할 것으로 보장됩니다만, 약간 제한적일 것으로 보입니다.

추가적인 방법들 역시 모습을 드러내게 될 것으로 보이는데, 특히나 user-level RCU 구현의 발전에 따라서 특히 그렇습니다.⁶

17.2.3.4 Extra-Transactional Accesses

락 기반의 크리티컬 섹션 내에서, 동시에 액세스 되는 변수를 조정하는 것은 완전히 합법적이며, 심지어 그 락의 크리티컬 밖에서 수정되고 있는 변수에 액세스 하는 것도 합법적인데, 그런 혼란 예 중 하나는 통계적 카운터일 것입니다. 같은 일이 RCU read-side 크리티컬 섹션 안에서도 비슷하며, 이는 혼란 일입니다.

제품화된 데이터베이스 시스템에서도 혼란 “dirty reads” 라 불리는 이런 메커니즘을 놓고 보면, TM의 제안자로부터 extra-transactional access들이 상당한 관심을 받은 것은 놀랄만한 일이 아닌데, 완화된, 그리고 강화된 원자성 [BLM06]이 이런 의미에서의 하나의 케이스입니다.

여기 TM에 사용 가능한 extra-transactional 선택 사항들이 몇 가지 있습니다:

1. Extra-transactional 액세스로 인한 충돌은 항상 트랜잭션을 abort 시킵니다. 이는 강한 원자성입니다.
2. Extra-transactional 액세스로 인한 충돌은 무시되어서, 트랜잭션 끝의 충돌만이 트랜잭션들을 abort 시킵니다. 이는 완화된 원자성입니다.
3. 트랜잭션은 메모리를 할당할 때나 락 기반의 크리티컬 섹션과 상호작용할 때와 같이 특수한 경우에 트랜잭션에서 할 수 없는 동작도 할 수 있도록 허용됩니다.
4. 복수개의 트랜잭션에 의해 하나의 변수에 동시에 수행될 수 있는 일부 오퍼레이션(예를 들면, 더하기)이 가능하게 하는 하드웨어 확장장치를 만듭니다.

⁶ 여러개의 앞의 대안들을 가져와준 TxLinux 그룹, Maged Michael, 그리고 Josh Triplett에게 감사를.

5. 트랜잭션 메모리에 완화된 semantic 을 가져옵니다. 한가지 방법은 Section 17.2.3.3 에서 설명된 RCU 와의 조합으로, Gramoli 와 Guerraoui 가 예를 들면 커다란 “신축성 있는” 트랜잭션들을 작은 트랜잭션들로의 제한적 분할로 (맥빠지는 성능과 확장성에도 불구하고) 충돌 가능성성을 줄이는 식의 많은 수의 완화된 트랜잭션 방법을 연구했습니다 [GG14].

트랜잭션들은 어떤 다른 동기화 메커니즘과의 상호 작용의 필요 없이 혼자 존재할 수 있도록 만들어진 것으로 보입니다. 만약 그렇다면, 트랜잭션들을 트랜잭션으로 인한 것이 아닌 액세스와 조합할 때에 많은 혼란과 복잡성이 등장하는 것은 놀라운 일이 아닙니다. 하지만 트랜잭션들이 격리된 데이터 구조로의 적은 업데이트로만 국한되어 있는 것이 아니라면, 또는 기존에 존재하던 커다란 병렬 코드와 상호작용하지 않는 새로운 프로그램에만 국한되어 있는 게 아니라면, 트랜잭션은 커다란 수준의 실용적인 효과를 내기 위해서는 트랜잭션이 아닌 액세스와 조합되어야만 합니다.

17.2.4 Discussion

보편적인 TM 적용에 대한 문제들은 다음과 같은 결론을 이끌어냅니다:

1. TM 의 흥미로운 속성 가운데 하나는 트랜잭션들은 롤백과 재시도에 걸리기 쉽다는 사실입니다. 이 속성은 버퍼링 되지 않는 I/O, RPC, 메모리 매핑 오퍼레이션, 시간 딜레이, 그리고 `exec()` 시스템콜과 같이 되돌이킬 수 없는 오퍼레이션들에 대한 TM 의 어려움을 암시합니다. 이 속성은 또한 안타깝게도, 개발자에게 보여질 수 있는 형태로 존재하는, 실패의 가능성으로 인해 존재하는 동기화 도구들에 내재된 복잡성을 가져오는 결론을 이끌어냅니다.
2. Shpeisman 등 [SATG⁺09] 에 의해 이야기된 TM 의 또다른 흥미로운 속성은, TM 은 자신이 보호하는 데이터와 동기화가 뒤엉켜진다는 것입니다. 이 속성은 I/O, 메모리 매핑 오퍼레이션들, 트랜잭션 이외의 접근들, 그리고 디버깅 브레이크포인트들과 연관되는 TM 의 문제들을 암시합니다. 대조적으로, 락킹과 RCU 를 포함하는 전통적인 동기화 기능들은 동기화 기능들과 그들이 보호하는 데이터 사이의 분명한 구분을 유지합니다.
3. TM 분야의 많은 사용자들의 이야기된 목표들 가운데 하나는 커다란 순차적 프로그램의 병렬화를

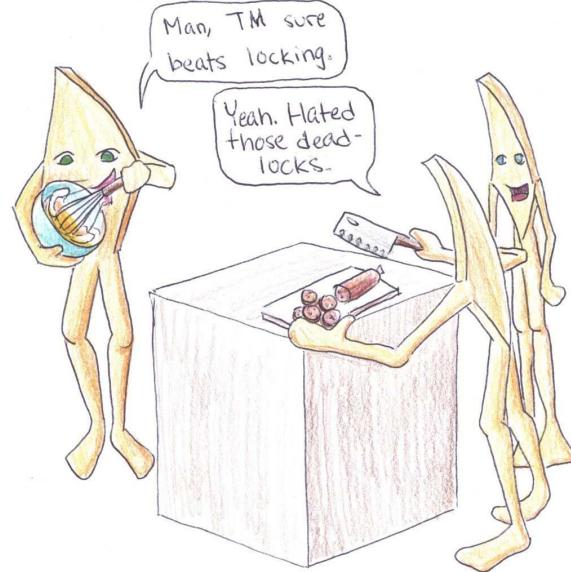


Figure 17.8: The STM Vision

쉽게 하는 것입니다. 그렇게 되면, 개별적인 트랜잭션들은 흔히 순차적으로 수행될 것으로 기대되어서, TM 의 멀티쓰레드로 수행되는 트랜잭션들에 대한 문제들을 상당수 설명할 수도 있을 겁니다.

TM 연구자들과 개발자들은 이런 모든 것에 대해 무엇을 해야 할까요?

한가지 방법은 TM 에 대한 집중은 작게 하고, 다른 동기화 기능들에 대비해 상당한 장점을 잠재적으로 하드웨어가 줄 수 있는 상황에 집중하는 것입니다. 이는 실제로 Sun 이 Rock 연구용 CPU 에서 취한 방법입니다 [DLMN09]. 일부 TM 연구자들은 이 방법에 동의하는 것으로 보입니다만, 다른 사람들은 TM 에 그보다 훨씬 많은 기대를 갖고 있습니다.

물론, TM 이 커다란 문제들에서 취해질 수 있게 되는 것도 상당히 가능할 것이고, 이 섹션은 TM 이 이 높은 목표를 달성하기 위해서는 해결해야 할 문제들을 일부 나열했습니다.

물론, 관련된 모든 사람들은 이를 배우는 경험으로 다뤄야 합니다. TM 연구자들은 커다란 소프트웨어 시스템을 전통적인 동기화 기능들로 성공적으로 만들어낸 실무자들로부터 배우기 위해 많은 거래를 하게 될 것으로 보입니다.

그리고 반대도 마찬가지죠.

하지만, STM 의 현재 상황은 일련의 만화로 요약될 수 있을 것 같습니다. 먼저, Figure 17.8 는 STM 의 전망을 보이고 있습니다. 항상 그렇듯, 실제는 Figures 17.9,



Figure 17.9: The STM Reality: Conflicts

17.10, 그리고 17.11에 그려진 것처럼 좀 더 미묘한 차이가 있습니다.

최근의 구할 수 있는 하드웨어에서의 발전은 뒤의 섹션에서 설명될 HTM의 변종으로의 문을 열었습니다.

17.3 Hardware Transactional Memory

2012년 초에 이르러, 하드웨어 트랜잭션 메모리 (HTM)이 상품으로 구입할 수 있는 흔한 컴퓨터 시스템들에도 도입되기 시작하고 있습니다. 이 섹션은 병렬 프로그래머의 도구상자에 이 하드웨어 트랜잭션 메모리가 위치할 자리를 알아보기 위한 시도를 해 봅니다.

개념적 관점에서, HTM은 명시된 명령문의 그룹 (하나의 “트랜잭션”)을 다른 프로세서에서 수행되는 모든 다른 트랜잭션들의 시야에 원자적으로 보이도록 그 효과를 만들기 위해 프로세서 캐시와 예측적 수행을 사용합니다. 이 트랜잭션은 begin-transaction 기계 인스트럭션을 통해 초기화 되고 commit-transaction 기계 인스트럭션을 통해 완료됩니다. 일반적으로 abort-transaction 기계 인스트럭션도 존재하는데, 이는 (begin-transaction 인스트럭션과 그를 뒤따른 인스트럭션들이 수행되지 않은 것처럼) 예측적 수행 내용을 짓이기고 failure han-



Figure 17.10: The STM Reality: Irrevocable Operations

dler의 수행을 시작합니다. 이 failure handler의 위치는 begin-transaction 인스트럭션에 의해서 명시적인 failure-handler의 주소를 통해서든 또는 해당 인스트럭션 자체의 조건적 코드에 의해서든 주어집니다. 각각의 트랜잭션은 모든 다른 트랜잭션에 대해 어토믹하게 수행됩니다.

HTM은 여러가지 중요한 장점을 갖는데, 데이터 구조의 자동적인 동적 분할, 동기화 기능의 캐시 미스의 감소, 그리고 상당수의 실용적 어플리케이션의 지원 등이 이런 장점을 포함됩니다.

하지만, 항상 작은 문자로 인쇄된 부분을 읽어야 하고, HTM 역시 예외가 아닙니다. 이 섹션의 중요한 요점은 어떤 조건 하에서 HTM의 장점들이 그것의 작은 문자로 인쇄된 내용에 감추어져 있는 복잡성을 앞서는 가를 결정하는 것입니다. 이런 결론 아래, Section 17.3.1은 HTM의 장점을 설명하고 Section 17.3.2은 그 단점들을 설명합니다. 이는 이전의 논문들 [MMW07, MMTW10]에서 취해진 것과 같은 방법입니다만, 전체적으로 TM보다는 HTM에 초점을 맞춥니다.⁷

이어서 Section 17.3.3은 리눅스 커널에서 (그리고 일부 user-space 어플리케이션에서) 사용되는 동기화 도구들의 조합과 관련해서 HTM의 단점을 설명합니다. Section 17.3.4은 병렬 프로그래머의 도구상자에서 어디에 HTM이 가장 잘 어울릴지를 알아보고, Section 17.3.5

⁷ 그리고 저는 다른 저자들인 Maged Michael, Josh Triplett, Jonathan Walpole, 그리고 Andi Kleen과의 자극적인 토론들에 기꺼이 감사를 드립니다.



Figure 17.11: The STM Reality: Realtime Response

은 HTM 의 범위와 매력을 상당히 증가시킬 수 있을 몇 가지 사건들을 나열합니다. 마지막으로, Section 17.3.6에서는 결론을 내립니다.

17.3.1 HTM Benefits WRT to Locking

HTM 의 주된 장점들은 (1) 다른 동기화 기능들에 의해 종종 일어나는 캐시 미스의 제거, (2) 동적으로 데이터 구조를 파티셔닝 하는 능력, (3) 상당한 수의 실용적 어플리케이션이 존재한다는 사실입니다. 저는 두 가지 이유로 TM 의 전통과 달리 사용의 편의성을 별도로 열거하지 않습니다. 첫째로, 사용의 편의성은 이 섹션이 초점을 맞추고 있는 HTM 의 주요 장점으로부터 기인합니다. 둘째, 날 프로그래밍 재능을 위한 테스트를 하려는 시도를 둘러싼 [Bow06, DBA09], 그리고 심지어 취직을 위한 면접에서의 작은 프로그래밍 연습문제의 사용을 둘러싼 [Bra07] 상당한 논쟁이 존재했습니다. 이는 우리가 무엇이 프로그래밍을 쉽게 하고 어렵게 하는지에 대한 진정한 이해를 가지고 있지 못함을 의미합니다. 따라서, 이 섹션은 앞서 나열한 세개의 장점에 대해서만, 각각 뒤의 섹션들에서 초점을 맞추도록 하겠습니다.

17.3.1.1 Avoiding Synchronization Cache Misses

대부분의 동기화 메커니즘들은 어토믹 인스트럭션으로 조정되는 데이터 구조에 기반합니다. 일반적으로 이

어토믹 인스트럭션들은 먼저 연관된 캐시 라인이 수행되고 있는 CPU 에 의해 소유되도록 하기 때문에, 다른 CPU 에서 같은 동기화 기능 인스턴스의 수행을 뒤따라 하게 되면, 캐시 미스를 초래하게 됩니다. 이러한 통신으로 인한 캐시 미스 이벤트들은 전통적인 동기화 메커니즘들의 성능과 확장성을 상당히 떠나게 합니다 [ABD⁺97, Section 4.2.3].

반면에, HTM 은 CPU 의 캐시를 사용해서 동기화를 하므로 동기화 데이터 구조의 필요와 그로 말미암은 캐시 미스가 없습니다. HTM 의 장점은 락 데이터 구조가 별개의 캐시 라인에 위치해 있을 때에 극대화 되는데, 크리티컬 섹션을 HTM 트랜잭션으로 변환함으로써 전체 캐시 미스로 인한 크리티컬 섹션의 오버헤드를 줄여주는 경우가 그런 경우입니다. 이런 이득은 짧은 크리티컬 섹션을 갖는 흔한 경우에 특히 클 수 있으며, 최소한 생략된 락이 그 락에 의해 보호되는, 자주 쓰여지는 변수와 캐시 라인을 공유하지 않을 때의 상황에서는 그렇습니다.

Quick Quiz 17.2: 해당 락 변수와 캐시 라인을 공유하는, 자주 쓰여지는 변수가 왜 문제가 될까요? ■

17.3.1.2 Dynamic Partitioning of Data Structures

일부 전통적 동기화 메커니즘의 사용에 대한 주요한 방해는 정적으로 데이터 구조를 분할해야 하는 필요성입니다. 간단하게 분할될 수 있는 데이터 구조들이 여럿 존재하는데, 유명한 예로 해시 테이블이 존재하는데, 여기서는 각각의 해시 체인이 하나의 파티션을 구성하게 됩니다. 각각의 해시 체인을 위한 락을 할당하는 것으로 해시 테이블을 해당 체인에 국한된 오퍼레이션들로 병렬화 시키게 됩니다.⁸ 배열, radix tree, 그리고 일부 다른 데이터 구조들에 대해서도 파티셔닝은 비슷하게 간단합니다.

하지만, 많은 종류의 tree 와 graph 에 있어 파티셔닝은 상당히 어렵고, 그 결과는 종종 복잡합니다 [Ell80]. 일반적인 데이터 구조를 파티셔닝 하는데에 two-phased 락킹과 해시된 락 배열들을 사용하는 것도 가능하지만, 다른 방법들이 더 선호되었는데 [Mil06], 이것들은 Section 17.3.3 에서 논의될 겁니다. 동기화 캐시 미스의 제거를 놓고 볼 때, HTM 은 최소한 상대적으로 적은 업데이트를 가정하면 커다란 파티셔닝 불가능한 데이터 구조를 위한 그럴싸한 방법입니다.

Quick Quiz 17.3: HTM 성능과 확장성에 상대적으로 적은 업데이트가 중요한 이유가 뭐죠? ■

⁸ 그리고 이 방법을 여러 해시 체인에 접근하는 오퍼레이션들로 그런 오퍼레이션들이 연관된 체인들을 위한 락들을 해시 순서대로 모두 잡도록 하는 것으로 쉽게 확장할 수 있습니다.

17.3.1.3 Practical Value

HTM 의 실질적 가치에 대한 몇몇 증거들이 Sun Rock [DLMN09] 와 Azul Vega [Cli09] 을 포함한 여러 하드웨어 플랫폼들에서 보여졌습니다. 실질적 이점들이 더 최신의 IBM Blue Gene/Q, Intel Haswell TSX, 그리고 AMD ASF 시스템들에서도 나올 것이라 가정하는 것은 합리적입니다.

예상되는 실질적 이점들은 다음과 같습니다:

1. In-memory 데이터 액세스와 업데이트를 위한 락 생략 [MT01, RG02].
2. 커다란 파티셔닝 불가능한 데이터 구조로의 동시 적인 액세스와 약간의 무작위적 업데이트들.

하지만, HTM 은 또한 실질적인 한계점들도 가지고 있는데, 이에 대해서는 다음 섹션에서 이야기 하겠습니다.

17.3.2 HTM Weaknesses WRT Locking

HTM 의 컨셉은 상당히 간단합니다: 메모리로의 액세스와 업데이트가 그룹 단위로 어토믹하게 일어난다는 것입니다. 하지만, 많은 간단한 아이디어들이 그러하듯이, 이를 실제 세계의 실제 시스템에 적용할 때에서야 복잡성이 나타납니다. 이 복잡성들은 다음과 같은 것들입니다:

1. 트랜잭션 크기 한계.
2. Conflict 처리.
3. Abort 와 롤백.
4. 진행 보장의 부재.
5. 되돌이킬 수 없는 오퍼레이션들.
6. Semantic 상의 차이점들.

이 각각의 복잡성들은 다음의 섹션들에서 다루어지고, 그 뒤를 이어 요약을 합니다.

17.3.2.1 Transaction-Size Limitations

현재 HTM 구현들의 트랜잭션 크기 한계점은 그 트랜잭션에 영향받는 데이터를 쥐고 있기 위해 프로세서의 캐시를 사용한다는 데서 나옵니다. 이는 특정 CPU 가 트랜잭션을 자신의 캐시 안에 국한된 채로 수행시켜서 해당 트랜잭션이 다른 CPU 들에게 어토믹하게 보이도록 하는 것을 가능하게 하지만, 이는 또한 캐시에 들어가지 않는 모든 트랜잭션은 abort 될것을 의미하기도

합니다. 더 나아가서, 인터럽트, 시스템 콜, exception, trap, 그리고 컨텍스트 스위치와 같이 수행 문맥을 바꾸는 이벤트들은 해당 CPU 에서 수행중인 트랜잭션을 모두 abort 시키거나 다른 수행 컨텍스트에 의한 캐시 사용량으로 인해 트랜잭션의 크기를 제한해야만 합니다.

물론, 최신 CPU 들은 커다란 캐시를 갖는 경향이 있고, 많은 트랜잭션들에 필요한 데이터는 1 메가바이트 캐시 안에도 잘 들어갈 겁니다. 불행히도, 캐시에 있어서, 크기만이 모든 문제는 아닙니다. 문제는, 대부분의 캐시들은 하드웨어로 구현된 해시 테이블로 생각될 수 있다는 것입니다. 하지만, 하드웨어 캐시는 (일반적으로 *set* 이라 불리는) 버킷들을 연결시키지 않고, *set* 당 고정된 수의 캐시라인들을 제공합니다. 특정 캐시에서 각각의 *set*에 제공되는 원소들의 수는 해당 캐시의 *associativity* 라고 불립니다.

Cache associativity 는 다양하지만, 제가 지금 타자를 치고 있는 랩탑의 8-way associativity level-0 캐시는 흔하지 않습니다. 이게 의미하는바는 특정 트랜잭션이 9개의 캐시 라인을 건드리게 되고, 그 9개의 캐시 라인들이 모두 같은 *set*으로 매핑된다면, 그 트랜잭션은 해당 캐시에 얼마나 많은 메가바이트의 용량들이 남아있는가에 관계없이 성공할 수 없다는 것입니다. 그렇습니다, 특정 데이터 구조에서 무작위적으로 골라진 데이터 원소들에 있어서, 그 트랜잭션이 커밋에 성공할 확률은 매우 높긴 합니다만, 어떤 보장사항도 없습니다.

이 한계점을 경감시키기 위한 일부 연구들이 있었습니다. Fully associative *victim cache* 는 associativity 한계를 경감시킬 수 있습니다만, *victim cache*의 성능과 에너지 효율성에 대한 많은 한계가 현재 존재합니다. 그렇다고는 하나, 수정되지 않은 캐시 라인들을 위한 HTM *victim cache* 는 주소만을 가지고 있을 수도 있으므로, 상당히 작을 수 있습니다: 데이터 주소 자체는 충돌되는 쓰기를 파악하기에 충분하며 [RD12] 데이터 자체는 메모리에 쓰여지거나 다른 캐시에 shadow 될수 있습니다.

Unbounded transactional memory (UTM) 방법 [AAKL06, MBM⁺06] 은 DRAM 을 극단적으로 커다란 *victim cache* 로 사용합니다만, 그런 방법을 제품 품질의 캐시 일관성 메커니즘과 결합하는 것은 여전히 해결되지 않은 문제입니다. 또한, DRAM 을 *victim cache* 로 사용하는 것은 성능과 에너지 효율성의 저하를 가져올 수 있는데, *victim cache* 가 fully associative 하다면 특히 그렇습니다. 마지막으로, UTM 의 “unbounded” 라는 측면은 DRAM 이 모두 *victim cache* 로 사용될 수 있다는 가정을 합니다만, 실제로는 커다랗긴 하지만 고정된 양의, 해당 CPU 에 주어진 DRAM 의 용량만으로 해당 CPU 의 트랜잭션의 크기가 제한될 겁니다. 다른 방법들은 하드웨어 트랜잭션 캐시 메모리와 소프트웨어 트랜잭션 캐시 메모리의 조합을

사용하고 [KCH⁺06], HTM 의 fallback 메커니즘으로 STM 을 사용하는 방법을 생각해 볼 수도 있을 겁니다.

하지만, 제가 알기로는, 현재로써 사용 가능한 시스템들은 이런 연구 아이디어들을 구현한 바가 없습니다.

17.3.2.2 Conflict Handling

첫번째 문제는 *conflict* 의 가능성입니다. 예를 들어, transaction A 와 B 가 다음과 같이 정의되었다고 생각해 봅시다:

Transaction A

$x = 1;$
 $y = 3;$

Transaction B

$y = 2;$
 $x = 4;$

각각의 트랜잭션이 각자의 프로세서 위에서 동시에 수행된다고 생각해 봅시다. 만약 transaction A 가 x 에 쓰기를 하는 동시에 transaction B 가 y 에 쓰기를 한다면, 두 트랜잭션 모두 진행될 수 없습니다. 이를 보기 위해, transaction A 가 y 로의 쓰기를 수행한다고 생각해 봅시다. 그럼 transaction A 는 transaction B 와 섞여들어 가게 되는데, 이는 트랜잭션이 상대방의 시점에 어토믹하게 수행되어야 한다는 트랜잭션의 요구사항을 위반하는 것입니다. Transaction B 가 x 로의 저장을 수행하게 허락한다면, 이는 비슷하게 어토믹 수행 요구사항을 위반하는 것입니다. 이 상황은 *conflict* 라 명명되는데, 두개의 동시에 수행되는 트랜잭션들이 똑같은 변수에 접근하게 되며 그 접근들 가운데 최소한 하나는 쓰기인 경우에 발생합니다. 따라서 시스템은 수행이 진행될 수 있도록 하기 위해 이 트랜잭션들 가운데 하나나 두개 모두를 *abort* 시킬 의무를 갖습니다. 정확히 어떤 트랜잭션을 *abort* 시킬 것인가에 대한 선택은 Ph.D. 학위논문을 만들 능력이 있을 만큼 흥미로운 주제인데, 그런 예 [ATC⁺11]도 있으니, 보시기 바랍니다.⁹ 이 섹션의 목적을 위해서, 우린 시스템이 무작위적 선택을 한다고 가정하겠습니다.

또하나의 문제는 *conflict* 파악으로, 적어도 가장 간단한 경우에 있어서는 비교적 간단한 편입니다. 프로세서가 트랜잭션을 수행할 때, 프로세서는 그 트랜잭션에 의해 접근되는 모든 캐시라인을 표시해 둡니다. 만약 이 프로세서의 캐시가 현재 트랜잭션에 의해 접근된 것으로 표시된 캐시라인에 대한 요청을 받게 되면, 잠재적 *conflict* 이 일어난 것입니다. 더 세련된 시스템들은 현재 프로세서의 트랜잭션이 그 요청을 보낸 프로세서의 트랜잭션을 앞서도록 순서잡을 것이고, 이 프로세스의 최적화는 또한 Ph.D. 학위논문을 쓰기 위한 능력을 얻을 수 있게 해줄 겁니다. 하지만 이 섹션은 매우 간단한

conflict 파악 전략을 가정합니다.

하지만, HTM 이 효율적으로 동작하려면 *conflict* 의 가능성이 충분히 낮아야 하는데, 이는 데이터 구조가 충분히 낮은 *conflict* 가능성을 유지하도록 구성되어야 할것을 필요로 합니다. 예를 들어, 간단한 삽입, 삭제, 그리고 탐색 오퍼레이션을 제공하는 red-black 트리는 이런 경우에 적합합니다만, 트리의 모든 원소의 정확한 수를 유지해야 하는 red-black 트리는 그렇지 않습니다.¹⁰ 또다른 예로, 트리의 모든 원소를 하나의 트랜잭션에서 열거하는 red-black 트리는 높은 *conflict* 확률을 가질 것이고, 성능과 확장성을 떨어뜨릴 겁니다. 결과적으로, 많은 순차적 프로그램들은 HTM 이 효과적으로 동작하도록 하기 위해 일부 재구성을 필요로 할 겁니다. 몇몇 경우에 있어서, 실무자들은 그런 추가적 단계를 취하거나 (red-black 트리의 경우에 있어서, radix 트리나 해시 테이블과 같은 파티셔닝 가능한 데이터 구조로의 전환 같은) 그냥 락킹을 사용하는 것을 선호할 수 있는데, HTM 이 모든 관련된 구조에서 사용 가능하기 충분한 시간이 오기 전까지는 특히 그럴 수 있습니다 [Cli09].

Quick Quiz 17.4: 동기화 메커니즘의 선택에 관계 없이 어떻게 red-black 트리가 트리 내의 모든 원소의 열거를 효율적으로 할 수 있을까요???

더 나아가서, *conflict* 이 일어날 수 있다는 사실은 다음 섹션에 이야기되는 것처럼 *failure* 처리를 어떻게 할 것인지 그림을 가져올 수 있게 해줍니다.

17.3.2.3 Aborts and Rollbacks

모든 트랜잭션이 언제든 *abort* 될 수 있으므로, 트랜잭션은 롤백될 수 없는 명령을 포함해선 안된다는 점이 중요합니다. 이 말은 트랜잭션은 I/O, 시스템콜, 또는 디버깅 브레이크포인트 (HTM 트랜잭션에서의 디버거에서의 single step 수행이 안됩니다!!!) 를 가질 수 없음을 의미합니다. 대신, 트랜잭션은 스스로를 평범한 캐시된 메모리에만 접근하도록 국한시켜야만 합니다. 더 나아가서, 일부 시스템에서는, 인터럽트, exception, trap, TLB 미스, 그리고 다른 이벤트들 역시 트랜잭션을 *abort* 시킵니다. 잘못된 여러 조건의 처리로 초래된 많은 수의 버그들을 생각해보면, 사용성을 위해 *abort* 와 *rollback* 이 어떤 효과를 갖는지 알아보는게 좋을 겁니다.

Quick Quiz 17.5: 하지만 왜 디버거는 트랜잭션의 앞의 인스턴스의 스텝들을 다시 추적하기 위해 재시도에 의존하면서 브레이크포인트를 트랜잭션의 성공되는 명령문 줄에 설정해 두는 것으로 single stepping 을 흉내낼 수 없나요??

물론, *abort* 와 롤백은 HTM 이 real-time 시스템에 유용할 수 있는 것인지라는 질문을 떠올리게 합니다.

⁹ “Toxic Transactions” 라는 제목의 Liu 와 Spear 의 논문 [LS11]은 이런 면에서 특히 유명합니다.

¹⁰ 이 수를 업데이트 해야할 필요성이 트리로의 삽입과 트리로부터의 삭제가 서로 *conflict* 를 발생시키게 해서 strong non-commutativity 를 초래할 겁니다 [AGH⁺11a, AGH⁺11b, McK11b].

HTM의 성능적 이득이 abort와 롤백의 비용을 넘을까요, 그리고 그렇다면 어떤 조건 하에서 그럴까요? 트랜잭션은 우선순위 향상 기능을 사용할 수 있을까요? 아니면 높은 우선순위 쓰레드를 위한 트랜잭션은 낮은 우선순위 쓰레드들을 우선적으로 abort 시켜야 할까요?

만약 그렇다면, 하드웨어는 어떻게 효율적으로 우선순위를 알 수 있을까요? 실제 세계에서의 HTM의 사용에 대한 환경은 협소한데, 연구자들이 트랜잭션들이 비 real-time 환경에서 잘 동작하기에는 충분한 것들보다 더 많은 문제들을 찾고 있기 때문일 수도 있습니다.

현재의 HTM 구현들은 결정론적으로 특정 트랜잭션을 abort 시킬 수도 있기 때문에, 소프트웨어는 fallback 코드를 제공해야만 합니다. 이 fallback 코드는 예를 들면 락킹과 같은, 어떤 다른 형태의 동기화를 사용해야만 합니다. 만약 이 fallback이 빈번하게 사용된다면, 데드락의 가능성을 포함한 모든 락킹의 제한점들이 다시 나타납니다. 물론, 이 fallback이 자주 사용되지 않아서 더 간단하고 디드락이 나타나기 쉽지 않은 설계가 사용될 수 있게 되기를 바랄 수 있습니다. 하지만 이는 시스템은 락 기반의 fallback에서 트랜잭션으로 어떻게 전환할 것인지에 대한 질문을 떠올리게 합니다.¹¹ 한가지 방법은 test-and-test-and-set 방법 [MT02]의 사용으로, 모두가 락이 해제될 때까지 기다림으로써 시스템이 트랜잭션적으로 깨끗한 백지 상태에서 시작할 수 있도록 하는 것입니다. 하지만, 이는 상당한 spinning을 초래할 수 있는데, 이는 락을 전 쓰레드가 블락되어 있거나 preemption 당했다면 현명하지 못한 행위일 수 있습니다. 또 다른 방법은 트랜잭션이 락을 전 쓰레드와 병렬적으로 수행될 수 있도록 하는 것 [MT02]입니다만, 이 방법은 원자성을 유지하는데에 어려움을 낳으며, 특히 그 쓰레드가 락을 잡고 있는 이유가 연관된 트랜잭션이 캐시에 들어가지 않기 때문이라면 더 그렇습니다.

마지막으로, abort와 롤백 가능성을 처리하는 것은 개발자에게 모든 가능한 여러 조건들의 조합을 올바르게 처리해야 한다는 추가적인 부담을 지우는 것으로 보일 수 있습니다.

HTM의 사용자들이 fallback 코드 수행경로와 fallback에서 트랜잭션 코드로의 전환 모두에 상당한 검증을 위한 노력을 기울여야 함은 분명합니다.

17.3.2.4 Lack of Forward-Progress Guarantees

트랜잭션 크기, conflict, 그리고 abort/rollback이 모두 트랜잭션을 abort되게 만들 수 있다고는 하지만, 충분히 작고 짧은 기간동안 수행되는 트랜잭션은 결국은 성공할 것이라 보장된다고 희망할 수 있을 것입니다. 이는 compare-and-swap (CAS)와 load-linked/store-

¹¹ 어플리케이션이 fallback 모드에서 멈춰서게 되는 가능성은 Dave Dice가 만드는데 기여한, “lemming effect”라고 명명되었습니다.

conditional (LL/SC) 오퍼레이션들이 이 인스트럭션들을 어토믹 오퍼레이션을 구현하는데에 사용하는 코드에서 무조건적으로 재시도 되는 것과 똑같이 트랜잭션이 무조건적으로 재시도 되도록 허용하도록 할 수 있을 것입니다.

불행히도, 대부분의 현재 사용 가능한 HTM 구현들은 어떤 종류의 진행 보장도 제공하지 않는데, 이는 HTM은 시스템의 데드락을 막는데에 사용될 수 없음을 의미합니다.¹² 미래의 HTM 구현은 어떤 진행 보장을 제공할 수도 있을 겁니다. 그러기 전까지는, HTM은 real-time 어플리케이션은 상당한 주의 아래 사용되어야만 합니다.¹³

2013년에 있어 이 우울한 그림에 대한 한가지 예외는 특수한 제약된 트랜잭션 [JSG12]을 시작하는데 사용되는 별도의 인스트럭션을 제공하는 IBM 메인프레임의 차기 버전들입니다. 이름에서 추측할 수 있듯이, 그런 트랜잭션은 다음과 같은 제약 아래에 살아남을 수 있어야만 합니다:

- 각각의 트랜잭션의 데이터 사용량은 4개의 32-byte 메모리 블락 안에 들어갈 수 있어야만 합니다.
- 각각의 트랜잭션은 최대 32개의 어셈블리 인스트럭션을 수행하도록 허용됩니다.
- 트랜잭션은 뒤로 돌아가는 브랜치 (e.x: 루프를 가질 수 없음)를 가질 수 없습니다.
- 각각의 트랜잭션의 코드는 256 바이트의 메모리로 제한됩니다.
- 특정 트랜잭션의 데이터 사용량의 한 부분이 4K 페이지 안에 위치한다면, 그 4K 페이지는 트랜잭션의 인스트럭션을 담을 수 없습니다.

이런 제약은 가혹합니다만, 이는 더도 아니고 덜도 아니고 스택, 큐, 해시 테이블, 등을 포함해서 다양한 데이터 구조의 업데이트가 구현될 수 있도록 합니다. 이런 오퍼레이션들은 결국은 완료될 것이 보장되고, 따라서 데드락과 라이브락 조건으로부터 자유롭습니다.

시간의 흐름에 따라 하드웨어의 진행 보장이 얼마나 지우너될 것인지 보는 것은 꽤 흥미로울 것입니다.

17.3.2.5 Irrevocable Operations

Abort와 롤백의 또 다른 결론은 HTM 트랜잭션은 되돌이켜질 수 없는 오퍼레이션들을 담을 수 없다는 것입니다.

¹² HTM은 데드락의 가능성을 줄이는데 사용될 수는 있습니다만, fallback 코드가 수행될 가능성이 존재하는 한, 데드락의 가능성은 존재합니다.

¹³ 2012년 중반까지는, 트랜잭션 메모리의 real-time 특성에 대해서는 놀라만큼 적은 작업만이 있었습니다.

다. 현재의 HTM 구현들은 일반적으로 트랜잭션 안에서의 모든 액세스가 캐시될 수 있는 메모리 안으로만 국한되도록 하고 (따라서 MMIO 액세스를 금지하고) 인터럽트, trap, 그리고 exception 시에는 트랜잭션을 어보팅시키는 것으로 (따라서 시스템콜을 금지시키는 것으로) 이 제한을 강제합니다.

HTM 트랜잭션은 buffered I/O 역시 버퍼의 fill/flush 오퍼레이션이 트랜잭션 외에서 일어나는 한은 담겨질 수 있음을 알아두시기 바랍니다. 이게 동작하는 이유는 버퍼에 데이터를 넣고 빼는 것은 되돌이켜질 수 있기 때문입니다: 실제 버퍼 fill/flush 오퍼레이션들만이 되돌이켜질 수 없습니다. 물론, 이 buffered-I/O 방법은 I/O 를 트랜잭션의 흔적에 포함시키는 효과를 내서, 트랜잭션의 크기를 키우고 그로 인해 트랜잭션 실패 확률을 증가시킵니다.

17.3.2.6 Semantic Differences

HTM 이 많은 경우에 락킹의 대체제로 사용될 수 있기는 하지만 (그래서 transactional lock elision [DHL⁺08] 이란 명칭이 있습니다), semantic 상에 약간의 차이가 있습니다. 트랜잭션으로 수행되면 deadlock이나 livelock 을 초래할 수 있는, 락 기반의 크리티컬 섹션과 연관된 특히 골치아픈 예가 Blundell 에 의해 알려졌습니다만 [BLM06], 훨씬 간단한 예는 텅 빈 크리티컬 섹션입니다.

락 기반의 프로그램에서, 텅 빈 크리티컬 섹션은 기존에 그 락을 쥐고 있던 모든 프로세스들이 지금은 그것을 해제한 상태일 것을 보장합니다. 이 idiom 은 2.4 리눅스 커널의 네트워킹 스택에서 설정 변경을 조정하기 위해 사용되었습니다. 하지만 이 텅 빈 크리티컬 섹션이 트랜잭션으로 변환된다면, 그 결과는 no-op 입니다. 모든 앞의 크리티컬 섹션들이 종료되었을 것이라는 보장은 사라집니다. 달리 말해서, transactional lock elision 은 락킹의 데이터 보호 semantic 을 유지하지만 락킹의 시간에 기반한 메세지 semantic 은 잊어버립니다.

Quick Quiz 17.6: 하지만 누가 텅 빈 락 기반의 크리티컬 섹션을 필요로 하나요??? ■

Quick Quiz 17.7: 락 기반의 텅 빈 크리티컬 섹션들을 생략하지 않는 방법으로 간단하게 락킹의 시간 기반 메세징 semantic 을 transactional lock elision 에서 처리할 수는 없을까요? ■

Quick Quiz 17.8: 최신 하드웨어 [MOZ09] 에서, 누가 병렬 소프트웨어가 타이밍에 의존해서 동작할 거라고 기대할 수 있겠습니까? ■

락킹과 트랜잭션 사이의 중요한 semantic 상의 차이 하나는 락 기반의 real-time 프로그램에서 우선순위 역전을 막기 위해 상요되는 priority boosting 입니다. 우선순위 역전이 일어날 수 있는 한가지 경우는 락을 쥐고 있는 낮은 우선순위의 쓰레드가 중간 우선순위의 CPU

```

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

를 많이 사용하는 쓰레드에 의해 preemption 당하는 경우입니다. 만약 CPU 마다 그런 중간 우선순위 쓰레드가 최소 하나씩은 있다면, 낮은 우선순위 쓰레드는 수행될 기회를 결코 얻지 못할 겁니다. 만약 높은 우선순위 쓰레드가 이제 그 락을 얻으려 시도하면, 이 쓰레드는 블록될 겁니다. 이 쓰레드는 낮은 우선순위 쓰레드가 락을 놓기 전까지는 그 락을 얻지 못할 것이고, 이 낮은 우선순위 쓰레드는 수행될 기회를 얻기 전까지는 그 락을 해제하지를 못할 것이며, 낮은 우선순위 쓰레드는 중간 우선순위 쓰레드가 CPU 를 놓기 전까지는 수행될 기회를 얻지를 못할 겁니다. 따라서, 중간 우선순위 쓰레드는 실질적으로 높은 우선순위 프로세스를 블록하고 있는 셈인데, 이게 바로 “우선순위 역전”이라 불리는 이유입니다.

우선순위 역전을 막기 위한 한가지 방법은 *priority inheritance* 로, 락에 의해 블록된 높은 우선순위 쓰레드가 임시적으로 자신의 우선순위를 락을쥔 쓰레드에게 넘겨주는 것인데, *priority boosting* 이라고도 불립니다. 하지만, *priority boosting* 은 우선순위 역전을 막는 것 외에도 Figure 17.12 에 보여진 것처럼도 사용될 수 있습니다. 이 그림의 line 1-12 는 매 밀리세컨드마다 수행되어야 하는 낮은 우선순위 프로세스를 보이고 있고, line 14-24 는 *boostee()* 가 주기적으로 필요한 만큼 수행되는 것을 보장하기 위해 *priority boosting* 을 사용하는 높은 우선순위 프로세스를 보입니다.

boostee() 함수는 두개의 *boost_lock[]* 락들 가운데 하나를 항상 쥐고 있어서 *booster()* 의 line 20-21 이 우선순위를 필요한 만큼 증폭시킬 수 있게 함으로써 이를 가능하게 합니다.

Quick Quiz 17.9: 하지만 Figure 17.12 의

`boostee()` 함수는 그 락을 반대 순서로 잡고 있어요! 이는 deadlock 을 초래할 수 있지 않을까요?

■ 이 구조는 `boostee()` 가 시스템이 바빠지기 전에 line 5에서 첫번째 락을 잡을 것을 필요로 합니다만, 이는 최신 하드웨어에서조차도 쉽게 가능합니다.

안타깝게도, 이 구조는 transactional lock elision 의 존재 하에서는 깨질 수 있습니다. `boostee()` 함수의 겹쳐지는 크리티컬 섹션들은 잠시후든 나중이든 abort 될 하나의 무한한 트랜잭션이 되어버리는데, 예를 들면 `boostee()` 함수를 수행하는 쓰레드가 preemption 당하는 첫번째 시점이 되겠습니다. 이 시점에서, `boostee()` 는 락킹으로 물러나게 됩니다만, 그 낮은 우선순위와 초기화 단계가 완료되었다는 사실 때문에 (이게 바로 `boostee()` 가 preemption 을 당한 이유입니다), 이 쓰레드는 다시 수행될 기회를 얻지 못하게 됩니다.

그리고 만약 `boostee()` 쓰레드가 락을 잡고 있지 않다면, `booster()` 쓰레드의 Figure 17.12 line 20 과 21에서의 텅빈 크리티컬 섹션은 아무런 효과를 갖지 못하는 텅빈 트랜잭션이 되어서, `boostee()` 는 결코 수행되지 않을 겁니다. 이 예는 트랜잭션 메모리의 rollback-and-retry 시맨틱의 미묘한 결론을 그리고 있습니다.

경험이 추가적인 묘한 semantic 상의 차이를 더 드러낼 것인데, HTM 기반의 lock elision 의 커다란 프로그램으로의 적용은 주의 하에 이루어져야 합니다. 그렇다면 하나, 적용이 되는 곳이라면, HTM 기반 lock elision 은 해당 락 변수에 연관된 캐시 미스들을 제거할 수 있어서 2015년 초에 있어서 실제 세계의 커다란 소프트웨어 시스템들에서는 수십 퍼센트의 성능 향상을 가져옵니다. 따라서 우리는 이 기술을 지원하는 하드웨어에서의 이 테크닉의 상당한 사용을 기대합니다.

Quick Quiz 17.10: 그래서 많은 사람들이 락킹을 대신하는 작업을 시작하고는 대부분은 락킹을 최적화하는 것으로 결론을 내리나요???? ■

17.3.2.7 Summary

HTM 이 강력한 사용 예들을 갖는 것처럼 보이긴 하지만, 현재의 구현들은 주의깊은 처리를 필요로 하는, 트랜잭션 크기, conflict 처리의 복잡성, abort-and-rollback 이슈, 그리고 semantic 상의 차이와 같은 심각한 한계들을 가지고 있습니다. HTM 의 현재 상황이 락킹과 비교해서 Table 17.1에 요약되어 있습니다. 보여지듯이, HTM 의 현재 상황이 락킹의 일부 심각한 단점들을 경감시키긴 하지만,¹⁴ HTM 역시 그 자체의 상당

¹⁴ 공정성을 위해 말해두자면, 락킹의 단점들은 잘 알려져 있고 널리 사용되고 있는, deadlock detector [Cor06a], 락킹에 적용된 데이터 구조의 풍부함, 그리고 Section 17.3.3에서 이야기한 것과 같이 결합

히 많은 단점들을 포함하고 있습니다. 이러한 단점들은 TM 커뮤니티의 리더들에 의해서도 인정된 바 있습니다 [MS12].¹⁵

또한, 이게 전부가 아닙니다. 락킹은 일반적으로 그 자체만으로 사용되지 않고, 보통 레퍼런스 카운팅, 어토믹 오퍼레이션, non-blocking 데이터 구조, 해저드 포인터 [Mic04, HLM02], 그리고 read-copy update (RCU) [MS98a, MAK⁺01, HMBW07, McK12a] 등과 같은 다른 동기화 메커니즘들과 결합되어 사용됩니다. 다음 섹션은 그러한 결합이 이 수식을 어떻게 바꾸어놓는지 봅니다.

17.3.3 HTM Weaknesses WRT to Locking When Augmented

실무자들은 락킹의 일부 단점들을 막기 위해 오랫동안 레퍼런스 카운팅, 어토믹 오퍼레이션, non-blocking 데이터 구조, 해저드 포인터, 그리고 RCU 를 사용해 왔습니다. 예를 들어, deadlock 은 많은 경우에 레퍼런스 카운트, 해저드 포인터, 또는 RCU 를 데이터 구조를 보호하는데 사용함으로써 막아질 수 있고, 특히나 read-only 크리티컬 섹션에서는 그렇습니다 [Mic04, HLM02, DMS⁺12, GMTW08, HMBW07]. 이 방법은 또한 Chapter 10에서 봤던 것처럼 데이터 구조를 분할할 필요를 줄여줍니다. RCU 는 더 나아가서 contention에서 자유로운 wait-free read-side 기능들을 제공합니다 [DMS⁺12]. 이런 점을 Table 17.1에 추가하면 Table 17.2에 보인, augmented locking 과 HTM 사이의 업데이트된 비교가 나옵니다. 두개의 표간의 차이점을 요약해보면 다음과 같습니다:

1. Non-blocking read-side 메커니즘의 사용은 deadlock 문제를 줄여줍니다.
2. 해저드 포인터와 RCU 와 같은 Read-side 메커니즘들은 분할이 불가능한 데이터에서 효과적으로 동작할 수 있습니다.
3. 해저드 포인터와 RCU 는 서로간에 또는 업데이트 쓰레드와 충돌하지 않아서, 읽기가 대부분인 워크로드에서는 훌륭한 성능과 확장성을 제공합니다.

되어 사용되어온 긴 역사를 포함한, 엔지니어링 단계에서의 해결책들이 있음을 강조해둘 필요가 있습니다. 한가지 더 말하자면, 락킹이 정말로 많은 학계의 논문들을 살짝만 보아도 믿어질 만큼 끔찍한 것이었다면, 그 수많은 락 기반의 (FOSS 와 독점의) 병렬 프로그램들은 대체 어디서 나왔을까요?

¹⁵ 또한, 2011년 초에, 저는 트랜잭션 메모리를 둘러싼 일부 가정에 대한 비평을 하도록 초대된 바 있습니다 [McK11d]. 제가 발표를 하기 위해 시차에 매우 시달렸기 때문에 저를 편하게 해주기 위해서 였을지는 몰라도, 청중은 놀라우리만큼 적대적이지 않았습니다.

	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.	
	⇓ Locking primitives typically result in expensive cache misses and memory-barrier instructions.		- Partitioning required for fallbacks (less important for rare fallbacks).	
	+ Contention effects are focused on acquisition and release, so that the critical section runs at full speed.		- Transactions begin/end instructions typically do not result in cache misses, but do have memory-ordering consequences.	
	+ Privatization operations are simple, intuitive, performant, and scalable.		- Contention aborts conflicting transactions, even if they have been running for a long time.	
	+ Commodity hardware suffices.		- Privatized data contributes to transaction size.	
Hardware Support	+ Performance is insensitive to cache-geometry details.		- New hardware required (and is starting to become available).	
	+ APIs exist, large body of code and experience, debuggers operate naturally.		- Performance depends critically on cache geometry.	
Software Support	+ Long experience of successful interaction.		- APIs emerging, little experience outside of DBMS, breakpoints mid-transaction can be problematic.	
Interaction With Other Mechanisms	+ Yes.		⇓ 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. 해저드 포인터와 RCU 는 진행 보장을 제공합니다 (각각 lock freedom 과 wait-freedom 을 제공합니다).
5. 해저드 포인터와 RCU 에서의 privatization 오퍼레이션들은 간단합니다.

물론, 다음 섹션에서 논의하듯이 HTM 을 다른 기능들과 결합하는 것도 가능합니다.

17.3.4 Where Does HTM Best Fit In?

HTM 의 적용 영역이 page 146 의 Figure 9.45 에 보여진 RCU 처럼 그려지기 전인 것 같긴 하지만, 그런 방향으로 움직이기 시작하지 않을 이유는 없습니다.

HTM 은 커다란 멀티프로세서에서 돌아가는 상대적으로 커다란 in-memory 데이터 구조의 겹치지 않는 영역에 대한 상대적으로 작은 변경에 연관된 업데이트 위주의 워크로드에 들어맞을 텐데, 이런 워크로드는 현재 HTM 구현의 크기 제약을 맞출 수 있고 conflict 과 그로 인한 abort 와 롤백의 확률을 최소화 시켜줄 것이기 때문입니다. 이 시나리오는 현재의 동기화 도구들을 가지고는 처리하기가 상대적으로 어려운 시나리오이기도 합니다.

HTM 과 함께 락킹을 사용하는 것은 돌이킬 수 없는 오퍼레이션들에 대한 HTM 의 어려움을 극복해줄 수 있을 것으로 보이며, RCU 나 해저드 포인터의 사용은 데이터 구조의 커다란 부분을 획단하는, read-only 오퍼레이션들에 대해서 HTM 의 트랜잭션 크기 제한을 해결해 줄 수 있을 것입니다. 현재의 HTM 구현들은 RCU 나 해저드 포인터 읽기 쓰레드와 conflict 나는 업데이트 트랜잭션을 무조건적으로 abort 시키고 있지만, 미래의 HTM 구현들은 이런 동기화 메커니즘들과 더 부드럽게 작용할 것입니다. 그전까지는, 커다란 RCU 나 해저드 포인터 read-side 크리티컬 섹션과 conflict 나는 업데이트의 확률은 동일한 read-only 트랜잭션과 conflict 날 확률에 비해 훨씬 작아야 합니다.¹⁶ 더도 아니고 덜도 아니고, RCU 나 해저드 포인터 읽기 쓰레드들의 느린 흐름이 연관된 conflict 의 느린 흐름으로 인해 업데이트 쓰레드를 starve 시킬 수도 있습니다. 이런 취약점은 앞의 트랜잭션의 로드된 메모리 로케이션의 복사본에 대한 트랜잭션 외적인 읽기 방법을 제공하는 것으로 (상당한 하드웨어 비용과 복잡성을 요하겠지만) 제거될 수도 있습니다.

HTM 트랜잭션들이 fallback 을 가져야만 한다는 사실로 인해 어떤 경우에는 데이터 구조들의 정적 분할

가능성을 강제해야만 할 수도 있습니다. 이 제약점은 미래의 HTM 구현들이 진행 보장을 제공한다면 어떤 경우에는 fallback 코드의 필요성을 제거할 것이므로 없어질 수 있을 것인데, 이는 HTM 이 높은 conflict 확률 아래의 환경에서도 효과적으로 사용될 수 있을 것입니다.

요약하자면, HTM 이 중요한 사용과 응용 분야를 가질 수 있을 것이지만, 병렬 프로그래머의 도구상자의 또 다른 도구일 뿐이지, 도구상자 전체의 대체제는 아닙니다.

17.3.5 Potential Game Changers

HTM 의 필요성을 상당히 증가시킬 Game changer 들은 다음과 같은 것들이 있습니다:

1. 진행 보장.
2. 트랜잭션 크기 증가.
3. 개선된 디버깅 지원.
4. 완화된 원자성.

이것들은 다음 섹션들에서 확장되어 설명됩니다.

17.3.5.1 Forward-Progress Guarantees

Section 17.3.2.4 에서 논의한 것과 같이, 현재의 HTM 구현들은 진행 보장을 갖지 않아서 HTM 의 실패된 트랜잭션을 처리하기 위한 fallback 소프트웨어를 필요로 합니다. 물론, 보장을 추가하는 건 쉽습니다만, 그걸 제공하는건 항상 쉽지는 않습니다. HTM 의 경우에, 보장을 제공하는데 걸리는 문제는 캐시 크기와 캐시 associativity, TLB 크기와 TLB associativity, 트랜잭션의 시간적 길이와 인터럽트 빈도, 그리고 스케줄러 구현 등이 포함됩니다.

캐시 크기와 캐시 associativity 는 Section 17.3.2.1 에서 현재의 한계점을 우회하기 위한 일부 연구 작업들과 함께 논의된 바 있습니다. 하지만, HTM 의 진행 보장은 크기에 대해 한계와 함께 제공될 겁니다. 그런데 현재의 HTM 구현들은, 예를 들자면 캐시의 associativity 의 한계에 맞춰진 작은 트랜잭션들에 대해서는 진행 보장을 제공하지 않는 걸까요? 한가지 잠재적인 이유는 하드웨어 failure 를 처리해야 하는 필요성일 수 있습니다. 예를 들어, 문제가 생긴 캐시의 SRAM 셀은 해당 문제가 생긴 셀을 비활성화 시키는 것으로 처리될 수 있는데, 이는 캐시의 associativity 를 줄이게 되고 따라서 진행 보장이 제공될 수 있느니 트랜잭션의 최대 크기 역시 줄이게 됩니다. 이는 단순히 보장된 트랜잭션 크기를 줄일 뿐이란 점을 생각해 보면, 실제로는 다른 이유들도 있을 것으로 보입니다. 제품 수준의 하드웨어에서

¹⁶ NoSQL 데이터베이스들이 데이터베이스 어플리케이션의 엄격한 트랜잭션에의 의존도를 완화시키고 있는 상황에서 shared-memory 시스템에서는 엄격한 트랜잭션 메커니즘이 떠오르는 것은 상당히 아이라고 합니다.

진행 보장을 제공하는 것은 아마도 소프트웨어에서 진행 보장을 제공하는데 걸리는 어려움보다도 더 어려울 수 있습니다. 따라서 문제를 더 쉽게 풀기 위해, 문제를 소프트웨어에서 하드웨어로 옮기는 게 필요합니다.

물리적으로 tag 되고 index 되는 캐시에 있어서는 트랜잭션이 캐시 안에 들어간다는 것만으로는 충분치 않습니다. 여기서는 주소변환이 TLB에 들어 맞기도 해야 합니다. 따라서 모든 진행 보장은 TLB 크기와 TLB associativity도 신경을 써야 합니다.

현재의 HTM 구현들에서는 인터럽트, trap, 그리고 exception이 트랜잭션을 abort 시킨다는 점을 생각해 보면, 특정 트랜잭션의 수행 시간 길이가 인터럽트 간의 예상되는 시간 간격보다 짧아야 할 필요가 있습니다. 특정 트랜잭션이 얼마나 적은 데이터만 건드리는가와는 관계 없이, 너무 오랫동안 수행된다면, 해당 트랜잭션은 abort 될겁니다. 따라서, 모든 진행 보장은 트랜잭션 크기만이 아니라 트랜잭션 수행시간에 대해서도 조정되어야만 합니다.

진행 보장은 conflict 되는 여러개의 트랜잭션들 가운데 어떤 트랜잭션이 aobrt되어야 하는지 결정하는 능력에도 특히 의존적입니다. 각각 앞의 트랜잭션을 aobrt시키고 자기 자신도 뒤의 트랜잭션에 의해서 aobrt되어서 어떤 트랜잭션도 실질적으로 커밋하지 못하는 무한한 트랜잭션의 연속을 쉽게 생각해 볼 수 있습니다. Conflict 처리의 복잡성은 제안된 바 있는 많은 수의 HTM conflict 해결 정책들 [ATC⁺11, LS11]로 알 수 있습니다. 트랜잭션 외적인 액세스들로 인해 추가적인 복잡성들이 나타나는데, Blundell에 의해 알려졌습니다 [BLM06]. 이런 모든 문제들에 대해서 트랜잭션 외적인 액세스들을 탓하기는 쉽습니다만, 이런 생각의 어리석음은 각각의 트랜잭션 외적인 액세스를 그 자신만의 하나의 액세스로 구성된 트랜잭션으로 교체함으로써 쉽게 드러납니다. 이는 그 자체로 문제인 액세스 패턴이지, 그것들이 트랜잭션 안에서 일어나느냐 아니냐의 문제가 아닙니다.

마지막으로, 트랜잭션을 위한 모든 진행 보장은 해당 트랜잭션을 수행하는 쓰레드가 커밋을 성공하기에 충분할 만큼 길게 수행될 수 있도록 해줄 수 있는 스케줄러에 의존적입니다.

따라서 진행 보장을 제공하는 HTM 제작사들에게는 상당한 문제가 존재합니다. 하지만, 그것들을 해결하면 얻을 수 있는 효과는 대단합니다. 그렇게 되면 HTM 트랜잭션은 소프트웨어 fallback이 더 이상 필요없어지는 데, 이는 HTM은 마침내 TM의 deadlock 제거의 약속을 선사하게 됨을 의미합니다.

그리고 2012년 말, IBM Mainframe은 일반적인 죄선의 HTM 구현에 대해서 *constrained transaction*을 포함하는 HTM 구현을 발표했습니다 [JSG12]. *constrained transaction*은 best-effort 트랜잭션을 시작하는데 사용

되는 tbegin 명령과 달리 tbeginc 명령으로 시작됩니다. Constrained transaction은 항상 (결국은) 성공할 것이 보장되어 있으며, 따라서 만약 어떤 트랜잭션이 abort된다면, (best-effort 트랜잭션이 그렇듯이) fallback path로 분기되기보다는 하드웨어가 그 트랜잭션을 tbeginc 명령으로부터 재시작 시킵니다.

이 Mainframe 구조는 이 진행 보장을 위해서 상당한 측정을 해야 했습니다. 특정 constrained transaction이 반복적으로 실패한다면, 이 CPU는 branch prediction을 비활성화하고, in-order execution을 강제하고, 심지어 pipelining을 비활성화 시킬 수도 있습니다. 만약 반복된 failure가 높은 contention 때문이라면, CPU는 speculative fetch를 비활성화하고, 무작위적 delay를 집어넣고, 심지어 충돌하는 CPU들의 수행을 직렬화 시킬 수조차 있습니다. “흥미로운” 진행 보장 시나리오는 두개의 CPU만이 관여될 수도, 백개의 CPU들이 관여될 수도 있습니다. 아마도 이런 상당한 측정은 왜 다른 CPU들이 constrained transaction을 제공하는 것을 그렇게 자제했는지에 대한 이해를 일부 제공합니다.

그 이름이 이야기 하듯이, constrained transaction은 실제로 상당히 제약되어 있습니다:

1. 최대 데이터 사용량은 메모리의 4개 블럭으로 제한되는데, 각각의 블럭은 32 바이트 이상이 될 수 없습니다.
2. 최대 코드 크기는 256 바이트입니다.
3. 만약 특정 4K 페이지가 constrained transaction의 코드를 담고 있다면, 해당 페이지는 그 트랜잭션의 데이터를 담고 있을 수 없습니다.
4. 실행될 수 있는 assembly 인스트럭션의 최대 갯수는 32입니다.
5. 뒤 방향으로의 분기는 금지됩니다.

더도 아니고 덜도 아니고, 이러한 제약들은 링크드리스트, 스택, 큐, 그리고 배열과 같은 여러개의 중요한 데이터 구조들을 지원합니다. 따라서 constrained HTM은 병렬 프로그래머의 도구상자에서 중요한 도구가 될 수 있을 것으로 보입니다.

17.3.5.2 Transaction-Size Increases

진행 보장은 중요하지만, 우리가 봤듯이, 트랜잭션 크기와 시간에 기반한 조건적 보장이 될 겁니다. 작은 크기의 트랜잭션에 대한 보장도 상당히 유용함을 알아둘 것이 필요합니다. 예를 들어, 두개의 캐시 라인 크기에 대한 보장은 스택, 큐, dequeue에 충분합니다. 하지만,

더 큰 데이터 구조는 더 큰 트랜잭션에 대한 보장을 필요로 하는데, 예를 들어 tree 를 순서대로 횡단하는데에는 tree 의 노드의 수만큼의 크기에 대한 보장이 필요합니다.

따라서, 보장의 크기를 늘리는 것은 HTM 의 유용성 역시 늘려주고, 따라서 CPU 들이 HTM 을 제공하거나 훌륭하고 충분한 우회적 해결방법을 제공할 필요를 증가시킵니다.

17.3.5.3 Improved Debugging Support

트랜잭션 크기에 대한 또 다른 억제 요소는 트랜잭션을 디버깅 해야할 필요입니다. 현재 메커니즘에서의 문제는 single-step exception 이 자신을 둘러싼 트랜잭션을 abort 시킨다는 것입니다. 이 문제를 위한 우회적 해결 방법으로 프로세서 에뮬레이션 (느려요!), HTM 을 STM 으로의 대체 (느리고 시맨틱이 약간 다릅니다!), 진행을 에뮬레이션 하기 위한 반복적 재시도를 사용한 playback 테크닉 (이상한 failure 모드가 존재합니다!), 그리고 HTM 트랜잭션에서의 디버깅 지원 (복잡해요!) 등의 여러가지 방버들이 있습니다.

HTM 제조사들 가운데 누군가는 브레이크포인트, single stepping, 그리고 print 명령을 포함한 고전적인 디버깅 방법을 트랜잭션 안에서 사용할 수 있는 간단한 방법을 가능하게 하는 HTM 시스템을 제공해야 하며, 이는 HTM 을 더욱 강력하게 만들어줄 겁니다. 2013년에 이르러, 일부 트랜잭션 메모리 연구자들은 이 문제를 인식하고 있으며 하드웨어가 돋는 디버깅 기능들에 관한 제안도 있습니다 [GKP13]. 물론, 이 제안은 그런 기능들을 실제로 가지고 있고 사용할 수 있는 하드웨어에 의존적입니다.

17.3.5.4 Weak Atomicity

HTM 이 가까운 미래에 어떤 종류의 크기 제한을 갖게 될 것이라는 점을 놓고 보면, HTM 은 다른 메커니즘들과 부드럽게 연동될 수 있어야 할 겁니다. 해저드 포인터와 RCU 같은 읽기가 대부분인 경우를 위한 메커니즘들과 HTM 의 연동 능력은 트랜잭션 외적인 읽기가 같은 목적지에 쓰기를 하는 트랜잭션을 무조건적으로 abort 시키지 않는다면 개선될 수 있을 겁니다—대신, 해당 읽기는 간단히 해당 트랜잭션 전의 값을 얻어올 수 있을 겁니다. 이런 방식으로, 해저드 포인터와 RCU 는 HTM 이 커다란 데이터 구조를 처리하고 conflict 확률을 줄이는데 사용될 수 있을 겁니다.

하지만, 이는 간단하지 않습니다. 이 방법을 구현하는 가장 간단한 방법은 각각의 캐시라인과 bus 에 추가적인 상태를 가질 것을 필요로 하는데, 이는 추가적 비용을 필요로 합니다. 이 비용과 함께 생기는 장점은 커다란 영역을 접근하는 읽기 쓰레드들이 연속된 conflict

로 인해 업데이트 쓰레드들이 starve 하게 되는 문제 없이 수행될 수 있게 해준다는 것입니다.

17.3.6 Conclusions

현재의 HTM 구현들은 실질적인 이득을 가져다 줄 것으로 보이긴 하지만, 또한 상당한 단점들을 가지고 있기도 합니다. 가장 심각한 단점은 제한적인 트랜잭션 사이즈, conflict 처리, abort 와 롤백의 필요, 진행 보장의 부재, 되돌이켜질 수 없는 오퍼레이션들의 처리 불가능성, 그리고 락킹과의 미묘한 semantic 상의 차이입니다.

이러한 단점들 가운데 일부는 미래의 구현들에서는 줄어들 수 있겠습니다만, 기존에 언급된 바 [MMW07, MMTW10] 와 같이 많은 다른 종류의 동기화 메커니즘들과 함께 동작할 수 있어야 할 필요성은 계속될 것으로 보입니다.

요약해서, 현재의 HTM 구현들은 병렬 프로그래머의 도구박스에 들어오면 좋은, 그리고 유용한 추가적 도구가 되겠고, 그것들을 사용하기 위해서는 많은 흥미롭고 도전적인 작업들을 필요로 합니다. 하지만, 그것들은 모든 병렬 프로그래밍에서의 문제들을 모두 한번에 처리해줄 마법봉으로 여겨질 수는 없습니다.

17.4 Functional Programming for Parallelism

1980 년대 초에 제가 처음으로 함수형 프로그래밍 수업을 들었을 때, 교수님은 side-effect-free 한 함수형 프로그래밍 스타일은 사소한 병렬화와 분석에 잘 맞는다고 이야기했습니다. 30년이 지나서도 이 말은 여전히 남아있습니다만, 프로그램은 상태도 I/O 도 갖지 않아야만 한다고 이야기한 이 교수님의 또하나의 말과는 달리 병렬 함수형 언어를 사용하는 주류 업체는 적습니다. Erlang 과 같은 함수형 언어들의 틈새 사용 예가 존재하고, 일부 다른 함수형 언어들에 멀티쓰레드 지원이 추가되었습니다만, 주류 업체의 언어 사용은 (일반적으로 OpenMP, MPI, 또는 Fortran 의 경우, coarrays 와 연동되는) C, C++, Java, 그리고 Fortran 과 같은 절차형 언어의 것으로 남아있습니다.

이 상황은 기본적으로 “분석이 목표라면, 분석을 하기 전에 절차형 언어를 함수형 언어로 변환하는게 어떨까?” 하는 질문을 이끌어냅니다. 이 접근방법에 대해서는 물론 여러가지 반대의견들이 있는데, 여기선 세개만 나열해 보자면 다음과 같습니다:

1. 절차적 언어들은 종종 글로벌 변수들을 많이 사용하곤 하는데, 이 변수들은 다른 함수들, 또는, 더 나쁘게도, 여러 쓰레드들에서 접근될 수 있습니다.

Haskell 의 *monad* 는 단일 쓰레드의 글로벌 상태를 다루기 위해 만들어졌고, 글로벌 상태로의 복수 쓰레드에서의 접근은 함수적 모델에 대한 추가적 위반을 필요로 함을 알아두세요.

2. 멀티쓰레드를 지원하는 절차적 언어들은 종종 락, 어토믹 오퍼레이션, 트랜잭션과 같은, 함수형 모델에 대한 위반을 추가하는 동기화 기능들을 종종 사용합니다.
3. 절차적 언어들은, 예를 들면 하나의 함수에의 같은 호출에 두개의 서로 다른 인자에 같은 구조체로의 포인터를 넘김으로써 함수 인자들을 *alias* 할 수 있습니다. 이는 함수가 알지 못한채로 그 구조체를 두개의 서로 다른 (그리고 겹칠 수 있는) 코드 흐름을 통해 수정하는 결과를 초래할 수 있는데, 이는 분석을 상당히 복잡하게 만듭니다.

물론, 글로벌 상태, 동기화 기능들, aliasing 의 중요성으로 인해, 영리한 함수형 프로그래밍 전문가들은 함수형 프로그래밍 모델을 그것들에 조화시키기 위한 방법들을 여럿 제안했고, monads 는 그런 것들 중 하나에 불과합니다.

또 다른 접근방법은 병렬 절차적 프로그램을 함수형 프로그램으로 변환하고, 그 결과 나온 프로그램을 분석하는데에 함수형 프로그래밍 도구들을 사용하는 것입니다. 하지만 모든 실제 컴퓨팅은 유한한 시간 간격 동안 유한한 입력과 함께 동작하는 커다란 finite-state machine 이라는 점을 놓고 보면, 이보다 훨씬 잘 할 수 있을 법 합니다. 이는, 모든 실제 프로그램은 비록 실용적이지 못할만큼 커다란 것이라 할지라도 하나의 수식으로 변환될 수 있음을 의미합니다 [DHK12].

하지만, 여러개의 병렬 알고리즘의 낮은 단계의 알맹이들이 현대의 컴퓨터들의 메모리에 들어가기에 알맞을 만큼 충분히 작은 크기의 수식으로 변환될 수 있습니다. 만약 그런 수식이 단정문과 결합된다면, 해당 단정문이 들어맞는지 알아보는 것은 충족 가능성 문제가 됩니다. 충족 가능성 문제는 NP-complete 하긴 하지만, 이 문제들은 전체 상태 공간을 만들어내는데 필요한 것 보다는 훨씬 적은 시간 안에 풀이될 수 있는 경우가 많습니다. 또한, 이 풀이에 걸리는 시간은 그 아래 깔려있는 메모리 모델과는 독립적인 것으로 나타나서, 완화된 순서 규칙의 시스템에서 수행되는 알고리즘은 순차적으로 일관적인 시스템에서만큼이나 빠르게 검사될 수 있습니다 [AKT13].

일반적인 접근방법은 프로그램을 single-static-assignment (SSA) 형태로 변환시켜서 하나의 변수로의 각각의 값 할당이 그 변수의 별개의 버전을 만들도록 하는 것입니다. 이는 모든 동작중인 쓰레드로부터의 값 할당에 적용되어서, 그로부터 말미암은 표현은 검사하고자 하는 코드의 모든 가능한 수행경로를 담고

있게 됩니다. 단정문의 추가는 입력과 초기 값들의 어떤 조합이든 단정문이 틀리게 되는 경우를 만들 수 있는지에 대한 질문, 즉 앞에서 이야기한 충족 가능성 여부에 대한 질문을 수반 합니다.

여기서 있을법한 반대의견 중 하나는, 임의의 루프 구조에 대해서는 제대로 처리를 하지 못한다는 점입니다. 하지만, 많은 경우에 이는 루프를 유한한 횟수만큼 풀어놓는 것으로 처리될 수 있습니다. 또한, 아마도 일부 루프는 귀납법으로는 비난을 면치 못하게 될것이 증명될 겁니다.

또하나의 있을법한 반대의견은 스피너는 임의의 긴 루프에 관계되고, 유한 횟수 루프를 풀어놓는 방법은 해당 스피너의 모든 동작을 담지는 못할 것이라는 점입니다. 이 반대의견은 쉽게 극복될 수 있음이 드러났습니다. 전체 스피너를 모델링하는 대신에, 락을 얻으려 시도하며, 만약 즉시 락을 얻지 못한다면 abort 하도록 하는 trylock 을 모델링 하는 것입니다. 이렇게 되면 단정문은 락이 곧바로 얻을 수 없어서 abort 되는 스피너에 대해서는 단정문 실패가 되지 않도록 수정되어야 합니다. 논리수식은 시간과는 무관하기 때문에, 모든 가능한 동시성 동작들은 이 방법을 통해 담아질 수 있을 겁니다.

마지막 반대의견은 이 테크닉은 리눅스 커널을 만드는 수백만 줄의 코드로 이루어진 것과 같은 실제 전체 크기의 소프트웨어 작품을 다루기에는 적합치 않을 것이라는 것입니다. 이건 그럴 수도 있습니다만, 리눅스 커널 내의 그보다 훨씬 작은 병렬 기능들 각각을 제대로 검증하는 것도 상당히 가치있는 것이라는 사실은 그대로 남아있습니다. 그리고 실제로 연구자들은 이 방법을 리눅스 커널의 RCU 구현을 포함한 (RCU 의 덜 심오한 속성들 중 하나를 검증하는 것이기는 하지만) 단순하지 않은 실제 세계의 코드에 적용해 보려 노력하고 있습니다.

이 테크닉이 얼마나 넓게 적용될 수 있을 것인지를 볼 필요가 있습니다만, 이는 formal verification 분야의 더 흥미로운 혁신들 중 하나입니다. 그리고 이는 모든 프로그램을 함수적 형태로 작성하라는 고전적 방법보다는 훨씬 받아들일 만 할 것입니다.

Ask me no questions, and I'll tell you no fibs.

“She Stoops to Conquer”, Oliver Goldsmith

Appendix A

Important Questions

다음 섹션들은 SMP 프로그래밍에 관련된 몇 가지 중요한 질문들에 대해 논합니다. 또한, 각각의 섹션은 여러분의 목표가 여러분의 SMP 코드가 가능한한 빠르고 문제 없이 돌아가기 일 뿐—그건 그렇고, 훌륭한 목표죠!—이라면, 관련된 질문들에 대해 걱정하는 것을 어떻게 방지하는지를 보입니다.

이 질문들에 대한 대답들은 비록 많은 경우 싱글 쓰레드 환경에서에 비해 상당히 덜 직관적인 경우가 많습니다만, 이해하기에 어렵지는 않습니다. 여러분이 귀납법을 깨우쳤다면, 압도적일 만큼 어려운 것은 없을 겁니다.

A.1 What Does “After” Mean?

“After”는 직관적이지만 놀라우리만큼 어려운 개념입니다. 한가지 중요한 반직관적 문제는 코드가 언제든 열만큼이든 지연되어서 수행될 수 있다는 점입니다. 타임스탬프 “t”와 정수 필드 “a”, “b”, 그리고 “c”를 포함하는 글로벌 구조체를 이용해서 통신을 하는 생산자와 소비자 구조를 생각해 봅시다. 생산자는 Figure A.1에 보여진 것처럼 (1970년으로부터의 현재 시각까지 지난 초를 10진수로 나타내는) 현재 시각을 기록하고 “a”, “b”, 그리고 “c”를 업데이트 하는 루프를 돋습니다. 소비자는 Figure A.2에 보여진 것처럼 역시 현재 시각을 기록하지만 생성자의 타임스탬프와 “a”, “b”, 그리고 “c” 필드의 값을 복사해 옵니다. 프로그램 수행 종료 시점에서, 소비자는 이례적인 기록들을 출력하는데, 예를 들면 시간이 뒤로 돌아간 것으로 보이는 경우입니다.

Quick Quiz A.1: 이 예제에서 어떤 SMP 코딩 에러가 보이거나요? 전체 코드를 보기 위해선 `time.c` 파일을 보세요. ■

생성자가 타임스탬프나 값들을 저장하는데에 그렇게 많은 시간이 걸리지 않을 것이기에 생성자와 소비자 타임스탬프 간의 차이는 상당히 작을 것이라고 예상하는 사람들이 있겠습니다. 듀얼코어 1GHz x86에서의 출

```
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 = gettimeofday();
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

력 결과가 Table A.1에 보여져 있습니다. 여기서, “seq” 행은 루프를 수행한 횟수이고, “time” 행은 변칙적 행위가 나타난 시각을 초로 나타낸 것이고, “delta” 행은 소비자의 타임스탬프가 생성자의 것보다 얼마나 뒤의 것이었는지 (즉, 이 값이 음수라면 소비자가 생성자보다 타임스탬프를 먼저 받았음을 말합니다), 그리고 “a”, “b”, 그리고 “c”로 나타내어진 행들은 이 변수들이 소비자에 의해 수집된 앞의 스냅샷에 비해 얼마나 증가되었는지를 보입니다.

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

왜 시간이 거꾸로 가는 걸까요? 팔호 안의 숫자는 마이크로세컨드 단위의 차이로써, 큰 수는 10マイ크로세

```

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

컨드를 넘기고, 한번은 100 마이크로세컨드를 넘기기 조차 했습니다! 이 CPU는 그동안 100,000 개의 인스트럭션을 수행할 수도 있음을 알아두시기 바랍니다.

한가지 가능한 이유는 다음과 같은 이벤트 시퀀스로 설명됩니다:

1. 소비자가 타임스탬프를 읽어옵니다 (Figure A.2, line 13).
2. 소비자가 preemption 당합니다.
3. 임의의 시간이 흐릅니다.
4. 생성자가 타임스탬프를 읽어옵니다 (Figure A.1, line 10).
5. 소비자가 수행을 다시 재개하고, 생성자의 타임스탬프를 읽어옵니다 (Figure A.2, line 14).

이 시나리오 상에서, 생성자의 타임스탬프는 소비자의 타임스탬프보다 얼마큼이든 뒤의 것일 수 있습니다.

여러분은 여러분의 SMP 코드가 “after”의 의미로 고민하는 것을 어떻게 막으시나요?

그냥 SMP 기능들을 설계된 대로 사용하세요.

이 예제에서, 가장 간단한 수정방법은 락킹을 사용하는 것으로, 예를 들어 생성자는 Figure A.1의 line 10 앞에서 락을 잡고 소비자는 Figure A.2의 line 13 앞에서 락을 잡도록 합니다. 또한, 이 락은 Figure A.1의 line 13 뒤에서 해제되고 Figure A.2의 line 17 뒤에서 해제되어야만 합니다. 이 락들은 Figure A.1의 line 10-13과 Figure A.2의 line 13-17의 코드 조각들이 서로를 배제하도록 해주는데, 달리 말하자면 서로에 대해서 어토믹하게 수행되게 된다는 말입니다. 이는 Figure A.3로 표현되어 있습니다: 이 락킹은 모든 상자 안의 코드가 시간상으로 겹쳐지는 것을 방지해 줘서, 소비자의 타임스탬프는 앞의 생성자의 타임스탬프 뒤에 수집될 수 있도록 해줍니다. 이 그림에 있는 각각의 상자 안의 코드 조각들은 “크리티컬 섹션”이라 명명됩니다; 한 시점에는 오로지 하나의 크리티컬 섹션만이 수행됩니다.

이렇게 락킹을 추가하면 Table A.2에 보여진 것과 같은 출력 결과가 나옵니다. 여기선 시간이 뒤로 간 경우는 없었고, 그대신 소비자에 의해 읽어진 연속된 읽기로 읽혀진 값들 사이에는 1,000 카운트 이상의 차이들이 생겼습니다.

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: 소비자의 연속적인 읽기들 사이에 어떻게 그렇게 큰 간격이 존재하게 된걸까요? 전체 코드를 위해선 `timelocked.c` 파일을 보세요. ■

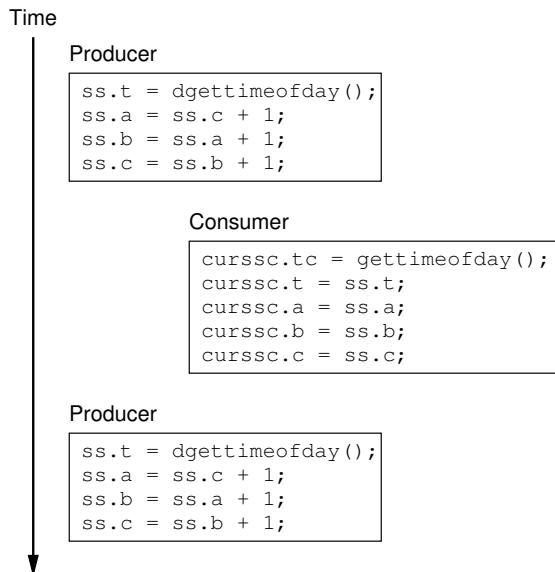


Figure A.3: Effect of Locking on Snapshot Collection

요약하자면, 여러분이 배타적 락을 획득한다면, 여러분은 그 락을 잡은 채 하는 모든 일이 그 락을 앞서 잡고서 행한 모든 것보다 뒤에 행해진 것으로 나타남을 알게 됩니다. 어떤 CPU가 메모리 배리어를 수행했는지 안했는가로 걱정할 필요가 없고, CPU나 컴파일러가 오퍼레이션들을 재배치 했는지에 대해 걱정하지 않아도 됩니다—삶은 간단합니다. 물론, 이 락킹이 이 두개의 코드 조각을 동시적으로 수행되지 못하도록 막는 것은 프로그램이 멀티프로세서에서 성능을 높일 가능성을 막는 데, 즉 “안전하지만 느린” 상황을 초래할 수도 있는 것입니다. Chapter 6는 많은 상황에서 성능과 확장성을 높일 수 있는 방법들을 설명합니다.

하지만, 대부분의 경우에 있어서, 어떤 주어진 코드 조각의 전과 후에 어떤 일이 일어나는지 걱정된다면, 표준 기능들의 사용을 더 잘 하기 위해 이를 힌트로 삼아야 합니다. 이 기능들이 여러분이 걱정을 하지 않아도 되도록 하도록 해주세요.

A.2 What is the Difference Between “Concurrent” and “Parallel”?

고전적 컴퓨팅 관점에서, “concurrent”과 “parallel”은 분명한 유의어입니다. 하지만, 많은 사람들이 이 두개의 단어 사이의 차이점을 이야기 하는 것을 멈추지 않았고, 이제 이 차이점들은 두개의 서로 다른 관점에서 이해될 수 있음이 들어났습니다.

첫번째 관점은 “parallel”을 “data parallel”의 약자로

취급하고, “concurrent”를 그 외의 대부분의 것으로 취급합니다. 이 관점에서, parallel computing에서는 전체 문제의 각각의 조각이 서로 다른 조각 사이의 통신 없이 완전히 독립적으로 진행될 수 있습니다. 이 경우, 조각들 사이의 협동은 거의 필요없거나 아예 필요 없습니다. 반면에, concurrent computing은 경쟁되는 락, 트랜잭션, 또는 다른 형태의 동기화 메커니즘들과 같은 형태의 강한 상호 작용성을 가질 수 있습니다.

Quick Quiz A.3: RCU read-side 기능만을 유일한 동기화 수단으로 사용하는 프로그램의 한 부분을 생각해 봅시다. 이는 parallelism 또는 concurrency인가요? ■

이는 물론, 그런 차이점이 뭐가 문제인가 하는 질문을 떠오르게 하는데, 이 질문은 스케줄러에 대한 두번째 관점을 가져올 수 있게 해줍니다. 스케줄러는 상당한 복잡성과 가능성을 갖는데, 경험에 의거한 법칙으로 이야기 하자면, 병렬의 프로세스들이 더 긴밀하고 불규칙하게 통신할수록 스케줄러에게 더 높은 수준의 세련됨이 필요해집니다. 따라서, parallel computing의 상호의존성의 제거는 parallel-computing 프로그램들은 최소한의 기능만 갖춘 스케줄러에서도 잘 동작함을 의미합니다. 실제로, 완전한 parallel-computing 프로그램은 임의의 단위로 분할되고 하나의 유니프로세서에 배치된 채로도 성공적으로 동작합니다.¹ 반면에, concurrent-computing 프로그램은 스케줄러에 상당히 미묘한 점들을 필요로 할겁니다.

우린 그저 스케줄러로부터의 합리적인 수준의 사항만을 알려주도록 요구해서, parallelism과 concurrency 사이의 차이점을 무시할 수 있어야 한다는 반론이 있을 수 있을 겁니다. 이는 많은 경우에 좋은 전략이지만, 스케줄러가 합리적으로 제공할 수 있는 정보의 수준을 효율성, 성능, 그리고 확장성이 크게 제한하는 중요한 경우들이 존재합니다. 그런 중요한 예제 가운데 하나는 스케줄러가 하드웨어로 구현된 경우로, SIMD나 GPGPU가 대부분 그렇습니다. 또 다른 예는 일의 단위가 상당히 짧은 워크로드로, 소프트웨어 기반의 스케줄러라 하더라도 간단함과 효율성 사이에서 어려운 선택을 해야만 하는 경우가 생깁니다.

이제, 이 두번째 관점은 워크로드가 현재 사용 가능한 스케줄러와 들어맞도록, 즉 parallel 워크로드는 간단한 스케줄러에서 동작할 수 있고 concurrent 워크로드는 더 세련된 스케줄러를 필요로 하도록 만드는 것으로 생각될 수 있습니다.

불행히도, 이 관점은 첫번째 관점으로부터 나온 의존성 기반의 차이점과 항상 들어맞지는 않습니다. 예를 들어, 상당히 독립적인 락 기반의, CPU 당 쓰레드 하나의 워크로드는 스케줄러의 결정이 필요치 않기 때문에 간단한 스케줄러 위에서도 돌아갈 수 있습니다. 실제로,

¹ 그렇습니다, 이는 parallel-computing 프로그램들은 순차적 수행에 잘 어울림을 의미합니다. 왜 물어보셨죠?

이런 종류의 일부 워크로드는 순차적 기계 위에서 조차도 한 부분 다음 다른 부분이 돌아가는 식으로 수행될 수 있습니다. 따라서, 그러한 워크로드는 첫번째 관점에 의해서는 “concurrent” 라 표시되지만 두번째 관점을 취하면 “parallel” 이라 표시될 수 있을 것입니다.

Quick Quiz A.4: 두번째 (스케줄러 기반의) 관점의 어떤 부분에서 락 기반의 CPU 당 하나의 쓰레드를 사용하는 워크로드가 “concurrent” 로 여겨질 수 있을까요?

■ 이는 문제가 되지 않습니다. 인류가 만든 어떤 규칙도 객관적 진실성에 대한 중요성을 갖지 않는데, 이는 멀티프로세서 프로그램을 “concurrent” 와 “parallel”로 카테고리를 나누는 것에 대해서도 마찬가지입니다.

이 카테고리화에서의 실패는 그런 규칙들이 쓸모없음을 의미하지 않으며, 오히려 여러분이 그것들을 새로운 상황에 적용하려 할 때에는 충분히 비판적인 프레임을 취해야 함을 의미합니다. 항상 그렇듯이, 그런 규칙은 적용될 수 있을 때에만 사용하고 그렇지 않을 때에는 무시하세요.

실제로, parallel, concurrent, map-reduce, task-based, 기타 등등 외에도 추가적인 카테고리들이 생겨날 수 있습니다. 누군가는 시간의 평가를 기다리고 있을 수도 있을 텐데, 올바른 예측이길 바랍니다!

A.3 What Time Is It?

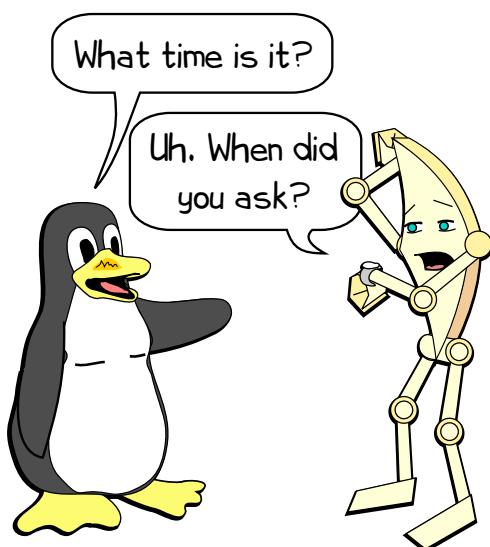


Figure A.4: What Time Is It?

멀티코어 컴퓨터에서 시간을 관리하는데 있어서의

핵심 문제가 Figure A.4에 그려져 있습니다. 한가지 문제는 시간을 읽는데에도 시간이 걸린다는 점입니다. 하나님의 인스트럭션을 통해 하드웨어 시계를 읽을 것이고, 이 읽기 오퍼레이션을 완료하기 위해서 코어 바깥으로 (더 나쁜 경우라면, socket 바깥으로) 나가야 할 겁니다. 또한, 읽어온 값에 대해서 어떤 계산을 해야 할 수도 있는데, 예를 들어, 요구된 형태로 변환을 하기 위해, 또는 네트워크 타임 프로토콜 (NTP)을 통한 조정을 가하기 위해, 등등의 이유로 인해서입니다. 그러니 최종적으로 리턴되는 시간은, 시간을 얻어도기 위해 걸린 시간의 시작점에서의 시각일까요, 끝에서의 시각일까요, 또는 그사이 어딘가일까요?

더 나쁜건, 시간을 읽는 쓰레드가 인터럽트 당하거나 preemption 당할 수도 있습니다. 더 나아가서, 시간을 읽어오고 나서 실제로 그렇게 읽어온 시간을 사용하기 전에 또 다른 계산 과정이 들어갈 수도 있습니다. 이런 가능성들이 불확실한 시간 간격을 더 늘립니다.

이에 대한 한가지 방법은 시간을 두번 읽고, 이렇게 읽어온 두가지 값의 평균을 사용하는 것입니다. 두번의 읽어온 값 사이의 차이는 사이에 끼인 오퍼레이션이 발생 시킨 시간의 불확실성의 정도입니다.

물론, 많은 경우에, 정확한 시각이 필요치는 않습니다. 예를 들어, 사람이 사용하기 위한 시각을 프린트하는 경우에 있어서는, 우리는 내부 하드웨어와 소프트웨어의 비규칙적인 딜레이를 처리하는데에 있어, 사람의 느린 반응성에 의존할 수 있습니다. 비슷하게, 어떤 서버가 클라이언트로의 응답 시간을 측정해야 한다면, 요청의 도착 시각과 응답의 전송 사이의 시각은 똑같이 팬참을 겁니다.

Appendix B

Why Memory Barriers?

그래서 무엇이 CPU 설계자들을 불쌍하고 의심할 줄 모르는 SMP 소프트웨어 설계자들에게 메모리 배리어를 주게 만들었을까요?

한마디로, 메모리 참조 순서 재배치는 성능을 훨씬 좋게 만들고, 따라서 올바른 동작 여부가 메모리 참조 순서에 의존적인 동기화와 같은 작업에는 순서를 강제하기 위해 메모리 배리어가 필요해졌습니다.

이 질문에 더 자세한 답변을 얻기 위해선 어떻게 CPU 캐시들이 동작하는지, 특히 캐시가 정말 잘 동작하게 하기 위해 필요한게 무엇인지에 대한 깊은 이해가 필요합니다. 다음의 섹션들은:

1. 캐시의 구조를 보이고,
2. 캐시 일관성 프로토콜이 어떻게 CPU 들이 메모리의 각 위치의 값들에 대해 합의하며, 마지막으로,
3. 어떻게 스토어 베퍼들과 인밸리데이트 큐들이 캐시와 캐시 일관성 프로토콜이 높은 성능을 얻을 수 있게 돋는지 알아봅니다.

우린 메모리 배리어들이 좋은 성능과 확장성을 위한 필 요약이고, CPU 들이 그들 사이의 접합부보다도, 그들이 접근하려 시도하는 메모리 보다도 훨씬 빠르다는 사실로부터 기인했음을 보게 될 것입니다.

B.1 Cache Structure

현대의 CPU 들은 현대의 메모리 시스템들보다 훨씬 빠릅니다. 2006 년의 CPU 는 나노세컨드당 열개의 인스트럭션들을 수행할 수 있습니다만, 메인 메모리에서 데이터 아이템 하나를 가져오는데엔 수십 나노세컨드를 필요로 할겁니다. 이 속도의 간극 — 100 배가 넘는 — 이 현대 CPU 에서 볼 수 있는 수 메가바이트의 캐시를 있게 했습니다. 이런 캐시들은 Figure B.1 에 보여졌듯,

CPU 들과 연관되어지고 수 사이클만에 접근될 수 있습니다.¹

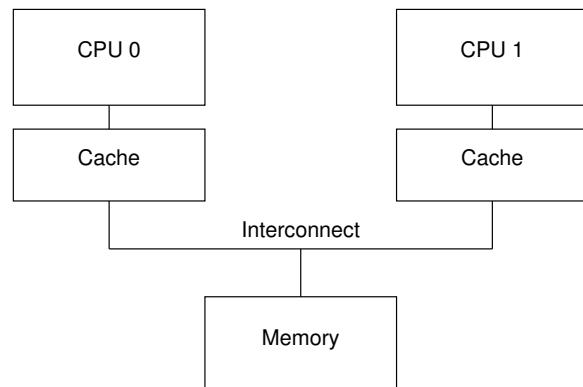


Figure B.1: Modern Computer System Cache Structure

CPU 의 캐시와 메모리 사이의 데이터 흐름은 “캐시 라인” 이라 불리는, 일반적으로 16 과 256 사이의 2의 거듭제곱 바이트 크기인, 고정된 길이의 블록 단위로 이루어집니다. 한 데이터 아이템이 한 CPU 에 의해 처음 액세스 되면, 그 아이템은 해당 CPU 의 캐시에 없을 것이고, 이는 곧 “캐시 미스” (또는, 보다 분명히 말하면, “스타트업” 또는 “웜업” 캐시 미스) 를 의미합니다. 이 캐시 미스는 CPU 는 해당 아이템이 메모리로부터 얻어져 오는 동안 수백 사이클을 기다려야 (또는 “스톨” 되어야) 함을 의미합니다. 하지만, 해당 아이템은 해당 CPU 의 캐시 위에 로드될 것이고, 따라서 다음의 액세스는 해당 아이템을 캐시에서 찾아낼 것이고, 따라서 최대 속도로 처리될 것입니다.

¹ CPU 에 가깝게 작고 한 사이클의 액세스 타임을 갖는 1단계 캐시를, 그 다음엔 더 크고, 약 10 사이클 정도의 긴 액세스 타임을 갖는 2단계 캐시를 두는 식으로 여러 단계의 캐시를 사용하는건 표준적인 일입니다. 고성능 CPU 는 종종 세단계 또는 네단계까지도 캐시를 둡니다.

	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 B.2: CPU Cache Structure

시간이 지난 후, 이 CPU의 캐시는 꽉 찰 것이고, 이후의 미스들은 새로 가져온 아이템들을 위한 공간을 만들기 위해 캐시로부터 아이템을 빼울 것을 필요로 할 수 있습니다. 그런 캐시 미스는 캐시의 제한된 용량으로 인해 발생하기 때문에 “용량 미스”라고 불리웁니다. 하지만, 대부분의 캐시들은 아직 용량이 꽉 차지 않았다 해도 새 아이템의 공간을 만들기 위해 오래된 아이템을 비우도록 강제되기도 합니다. 이는 커다란 캐시들은 Figure B.2 체이닝 없이 고정된 크기의 해시 버켓 (CPU 설계자들이 부르는 용어로는 “set”)들을 사용하는 하드웨어 해시 테이블로 구현되어 있기 때문입니다.

이 캐시는 16개의 “set”들과 두개의 “way”를 가져서 총 32개의 “라인”을 가지며, 각 엔트리는 256 바이트의 “캐시 라인” 하나를 담는, 256 바이트 정렬 블록의 메모리입니다. 이 캐시 라인 크기는 좀 작지만, 16진수 계산을 훨씬 간단하게 해줄 겁니다. 하드웨어 용어로, 이것은 two-way set-associative 캐시라고 불리우며, 16개의 버켓을 가지고, 각 버켓은 최대 두개의 원소를 가질 수 있는 해시 체인인 소프트웨어 해시 테이블로 비유될 수 있습니다. 사이즈 (이 경우 32 개의 캐시 라인들) 와 associativity (이 경우 2)는 함께 캐시의 “기하도형적 배열 (geometry)”이라 불립니다. 이 캐시는 하드웨어로 구현되었기 때문에, 해시 함수는 엄청 간단합니다: 메모리 어드레스에서 네개의 비트를 뽑아냅니다.

Figure B.2에서 각 박스는 256 바이트 캐시 라인을 담는 캐시 엔트리를 나타냅니다. 하지만, 캐시 엔트리는 그림의 빈 박스처럼 비어있을 수도 있습니다. 그 외의 박스들은 각 엔트리가 담고 있는 캐시 라인의 메모리 주소들을 표시하고 있습니다. 캐시 라인들은 256 바이트로 정렬되어야 하기 때문에, 각 주소의 하위 8 비트는 0입니다, 그리고 하드웨어 해시 함수는 그 다음 4개 상위 비트를 해시 라인 넘버로 매치시킵니다.

그림에 그려진 상황은 프로그램의 코드가 메모리 주소로 0x43210E00부터 0x43210EFF 사이에 위치하고, 이 프로그램이 0x12345000부터 0x12345EFF 까지의 데이터를 순차적으로 접근했다면 나타날 것입니다. 이 프로그램이 이제 0x12345F00을 접근하려 한다고 생각해 봅시다. 이 위치는 0xF 열로 해시되고, 이 라인의 두 way는 모두 비어있으므로, 이로 인한 256 바이트 라인은 캐시에 들어올 수 있습니다. 만약 프로그램이 0x1233000 위치를 액세스 하려 한다면, 0x0 열로 해시되고, 이로 인한 256 바이트 캐시 라인은 way 1에 들어올 수 있습니다. 하지만, 만약 프로그램이 0x1233E00 위치에 액세스 한다면, 0xE 열로 해시되는데, 여기 존재하는 것들 중 하나는 새로 들어올 캐시 라인을 위한 공간을 만들기 위해 비워져야 합니다. 만약 이렇게 비워진 캐시 라인이 나중에 액세스 된다면, 캐시 미스가 날겁니다. 그러한 캐시 미스를 “associativity miss”라고 합니다.

지금까지는 CPU가 데이터 아이템을 읽는 경우만 생각해봤습니다. 쓰기를 하면 어떻게 될까요? 주어진 데이터 아이템의 값에 대해 모든 CPU가 동의를 하는 것이 중요하기에, CPU는 어떤 데이터 아이템에 값을 쓰기 전에, 먼저 그 아이템을 다른 CPU의 캐시에서 삭제하거나, “무효화 (invalidate)” 시켜야만 합니다. 무효화 작업이 완료되면, 이 CPU는 안전하게 해당 데이터 아이템을 수정할 수 있습니다. 만약 해당 데이터 아이템이 이 CPU의 캐시에 존재했다면, 그러나 읽기 전용이었다면, 이 작업은 “write miss”라고 합니다. CPU가 주어진 데이터 아이템을 다른 CPU의 캐시들로부터 무효화시키는데 성공하면, 이 CPU는 해당 데이터 아이템을 반복적으로 쓸 수 (그리고 읽을 수도) 있습니다.

나중에, 다른 CPU가 해당 데이터 아이템에 접근하려 하면, 이번엔 첫번째 CPU가 그 아이템에 쓰기를 하려고 그 아이템을 무효화 시켰기 때문에 캐시 미스가 납니다. 이런 캐시 미스는 일반적으로 일부 CPU들이 데이터 아이템을 커뮤니케이션에 사용하려 해서 (예를 들어, 락은 CPU 들 사이에서 상호 배제적 알고리즘을 위한 커뮤니케이션에 사용되는 데이터 아이템입니다) 발생하기 때문에, “커뮤니케이션 미스”라고 합니다.

분명히, 모든 CPU들이 해당 데이터에 일관된 시야를 유지하도록 보장하는데 많은 주의를 기울여야만 합니다. 이 데이터 가져오기, 무효화 하기, 쓰기 작업을 통해, 데이터를 잃어버리거나 (더 나쁘게도) 다른 CPU들이 같은 데이터 아이템에 대해 각자의 캐시에 다른 값을 갖게 되는 경우를 상상해 볼 수 있습니다. 이런 문제는 다음 섹션에서 다루는 “캐시 일관성 프로토콜”에 의해 방지됩니다.

B.2 Cache-Coherence Protocols

캐시 일관성 프로토콜은 비일관적인 상황이나 데이터 유실을 막기 위해 캐시 라인 상태를 관리합니다. 이런 프로토콜은 열개 이상의 상태를 가져서 매우 복잡할 수 있습니다만,² 우리의 목적을 위해서는 MESI 캐시 일관성 프로토콜의 네가지 상태에 대해서만 신경쓰면 됩니다.

B.2.1 MESI States

MESI 는 한 캐시 라인이 이 프로토콜을 통해 가질 수 있는 네가지 상태인 “modified”, “exclusive”, “shared”, 그리고 “invalid” 의 약자입니다. 따라서 이 프로토콜을 사용하는 캐시는 캐시라인마다 해당 라인의 물리 주소와 데이터 이외에도 두 비트의 상태 “tag” 를 갖습니다.

“modified” 상태의 라인은 연관된 CPU 가 최근에 메모리 스토어를 했고, 연관된 메모리는 다른 CPU 의 캐시에 존재하지 않음이 보장됩니다. 따라서 “modified” 상태의 캐시 라인들은 해당 CPU 에 의해 “소유된 상태” 라고도 이야기 합니다. 이 캐시는 최신 상태의 데이터 카피만을 가지고 있으므로, 이 캐시는 메모리의 예전 데이터를 최신 데이터로 갱신하거나 다른 캐시에게 넘기거나 할 의무를 가지고 있으며, 그 의무는 이 라인이 다른 데이터를 가지게 되기 전에 반드시 수행되어야 합니다.

“exclusive” 상태는 “modified” 상태와 유사합니다만, 해당 캐시 라인은 연관된 CPU 에 의해 수정되지 않은 상태라는 점이 유일한 차이점으로, 메모리에 있는 캐시 라인의 데이터의 복사본이 최신 버전이란 뜻입니다. 하지만, CPU 는 다른 CPU 의 눈치를 보지 않고 언제든 이 라인에 언제든 스토어를 할 수 있으므로, “exclusive” 상태의 라인은 연관된 CPU 에 소유되어 있다고 이야기 할 수 있습니다. 그렇다면 하나, 메모리에 있는 값이 최신 상태이므로, 이 캐시는 데이터를 메모리에 다시 돌려넣거나 다른 CPU 에게 넘기지 않고 버릴 수 있습니다.

“shared” 상태의 라인은 적어도 한개의 다른 CPU 의 캐시에는 복사되어 있으며, 따라서 이 PU 는 먼저 다른 CPU 에게 이야기를 하지 않고서는 해당 라인에 스토어를 할 수 없습니다. “exclusive” 상태처럼, 메모리에 있는 값은 최신 버전이므로, 이 캐시는 메모리에 값을 다시 돌려놓거나 다른 CPU 에게 넘기지 않고 그 데이터를 버릴 수 있습니다.

“invalid” 상태의 라인은 비어있는데, 달리 말하자면, 아무 값도 들고 있지 않습니다. 새로운 데이터는 캐시에 들어올 때, 가능하다면 “invalid” 상태의 캐시 라인에

위치합니다. 다른 상태에 있는 라인을 교체하는 것은 교체된 라인이 미래에 다시 참조될 때 비싼 캐시 미스가 발생할 수 있기 때문에 이런 방법이 선호됩니다.

캐시 라인의 데이터에 대해 모든 CPU 가 일관적인 뷰를 유지해야 하기 때문에, 이 캐시 일관성 프로토콜은 시스템 사이의 캐시 라인의 이동을 조정하기 위해 메세지를 제공합니다.

B.2.2 MESI Protocol Messages

앞 섹션에서 이야기한 많은 작업들은 CPU 간의 통신을 필요로 합니다. 만약 CPU 들이 하나의 공유된 버스만을 사용한다면, 다음과 같은 메세지를 만으로도 충분합니다:

- Read: “Read” 메세지는 읽혀질 캐시 라인의 물리 주소를 담습니다.
- Read Response: “Read response” 메세지는 앞의 “read” 메세지에서 요청받은 데이터를 담습니다. 이 “read response” 메세지는 메모리에서 또는 캐시들 중 하나에서 얻어와질 수도 있습니다. 예를 들어, 캐시들 가운데 하나가 현재 필요로 하는 데이터를 “modified” 상태로 가지고 있다면, 그 캐시가 “read response” 메세지를 보내야 합니다.
- Invalidate: “Invalidate” 메세지는 무효화할 캐시 라인의 물리 주소를 담습니다. 모든 다른 캐시들은 그 데이터를 각자의 캐시에서 제거하고 답장을 해야 합니다.
- Invalidate Acknowledge: “Invalidate” 메세지를 받은 CPU 는 반드시 요청된 데이터를 자신의 CPU 로부터 제거한 후 “invalidate acknowledge” 메세지를 보내야 합니다.
- Read Invalidate: “Read Invalidate” 메세지는 읽음과 동시에 다른 캐시들로부터는 제거할 캐시 라인의 물리 주소를 담습니다. 따라서, 이름으로부터 알 수 있듯 이것은 “read” 와 “invalidate” 의 조합인 셈입니다. “read invalidate” 메세지는 “read response” 와 “invalidate acknowledge” 메세지 둘 다 응답으로 받습니다.
- Writeback: “writeback” 메세지는 메모리에 도로 쓰여질 (그리고 그 과정에서 다른 CPU 의 캐시에 “snooped” 를 보낼 수 있을 겁니다) 데이터와 주소를 모두 담습니다. 이 메세지는 캐시들이 “modified” 상태의 라인을 다른 데이터를 위한 공간을 만들어야 하거나 할 때 제거할 수 있도록 합니다.

² SGI Origin2000 의 9개 상태, Sequent (이제는 IBM) NUMA-Q 의 26개 상태 그림을 보기 위해 Culler et al. [CS99] 670 페이지와 671 페이지를 참고하세요. 두 그림 모두 실제보다는 단순합니다.

Quick Quiz B.1: Writeback 메세지는 어디서 와서 어디로 가나요? ■

흥미롭게도, 공유 메모리 멀티프로세서 시스템은 내부적으로는, 실제로 메세지 전달 컴퓨터입니다. 이는 분산 공유 메모리를 사용하는 SMP 머신 클러스터들은 두 개의 서로 다른 단계의 시스템 구조 간에 메세지 패싱을 구현하고 있다는 의미입니다.

Quick Quiz B.2: 두개의 CPU 들이 같은 캐시 라인을 동시에 무효화 하려고 하면 어떻게 되나요? ■

Quick Quiz B.3: 커다란 멀티프로세서에서 “invalidate” 메세지가 생기면, 모든 CPU 가 “invalidate acknowledge” 응답을 보내야만 합니다. 그로 인한 “invalidate acknowledge” 의 “폭풍우” 가 시스템 버스를 완전히 뒤덮지 않을까요? ■

Quick Quiz B.4: SMP 머신들이 실제로 메세지 전달을 어떻게든 사용한다면, 애초에 왜 SMP 에 신경을 쓰는거죠? ■

B.2.3 MESI State Diagram

캐시 라인은 Figure B.3 에 보여진 것처럼 프로토콜 메세지가 보내지고 받아짐에 따라 그 상태가 바뀝니다.

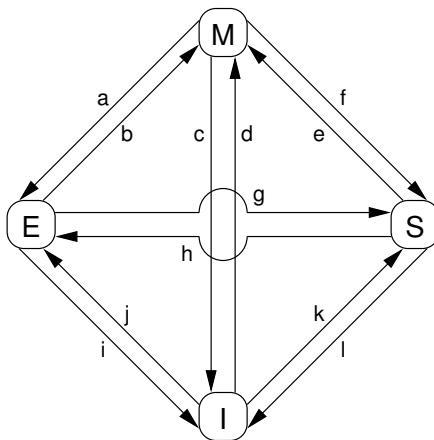


Figure B.3: MESI Cache-Coherency State Diagram

이 그림의 상태 변화들은 다음과 같습니다:

- 상태 변화(a): 캐시 라인이 메모리로 메모리에 writeback 되지만, 해당 CPU 는 그 데이터를 자신의 캐시에 유지하고 더나아가 그 데이터를 수정할 권리도 유지합니다. 이 상태 변경은 “writeback” 메세지를 필요로 합니다.
- 상태 변화 (b): CPU 가 이미 독점적 액세스 권한을 이미 가지고 있는 캐시 라인에 값을 씁니다. 이 상

태 변경은 어떤 메세지의 전송이나 수신을 필요로 하지 않습니다.

- 상태 변화 (c): 자신이 수정한 캐시 라인으로의 “read invalidate” 메세지를 CPU 가 받습니다. 메세지를 받은 CPU 는 자신의 복사본을 무효화시키고, “read response” 와 “invalidate acknowledge” 메세지 둘 다를 보내서 메세지를 보내온 CPU 에게 데이터를 전송하고 자신이 더이상 그 복사본을 들고 있지 않음을 알려야 합니다.
- 상태 변화 (d): CPU 가 자신의 캐시에 존재하지 않는 한 데이터 아이템에 어토믹 read-modify-write 오퍼레이션을 수행합니다. 이 CPU 는 “read invalidate” 메세지를 보내서 “read response” 를 받습니다. CPU 가 “invalidate acknowledge” 메세지까지 받으면 그 상태 변화가 완료됩니다.
- 상태 변화 (e): CPU 가 자신의 캐시에 read-only 상태로 존재하던 데이터 아이템에 어토믹 read-modify-write 오퍼레이션을 수행합니다. 상태 변화를 완료하기 위해 CPU 는 “invalidate” 메세지들을 보내고, “invalidate acknowledge” 응답이 모두 도착하길 기다려야 합니다.
- 상태 변화 (f): 어떤 다른 CPU 가 해당 캐시라인을 읽고, 그 오퍼레이션은 메모리에 writeback 중일 수도 있는 read-only 상태의 복사본을 가지고 있는 CPU 의 캐시에서 값을 얻어갑니다. 이 상태 변화는 “read” 메세지의 수신으로 시작되고 이 CPU 는 요청된 데이터를 담고 있는 “read response” 메세지로 응답을 보냅니다.
- 상태 변화 (g): 어떤 다른 CPU 가 이 캐시 라인의 데이터 아이템을 읽고, 그 오퍼레이션이 해당 CPU 의 캐시나 메모리에서 그 값을 얻어갑니다. 어떤 경우든, 이 CPU 는 read-only 복사본을 가지고 있습니다. 이 상태 변화는 “read” 메세지의 수신과 함께 시작되고, 이 CPU 는 요청된 데이터를 담고 있는 “read response” 메세지로 응답을 보냅니다.
- 상태 변화 (h): 이 CPU 는 자신이 곧 이 캐시 라인에 어떤 데이터를 쓰게 될 것임을 깨닫고, 따라서 “invalidate” 메세지를 보냅니다. 해당 CPU 는 모든 “invalidate acknowledge” 응답을 모두 받기 전까지 상태 변화를 마무리 할 수 없습니다. 또 다른 경우로, 모든 다른 CPU 들이 이 캐시 라인을 각자의 캐시에서 (다른 캐시라인을 위한 공간을 만들기 위해) “writeback” 메세지로 비워버려서 이 CPU 가 그 캐시 라인을 캐시하고 있는 마지막 CPU 가 될 수도 있습니다.

- 상태 변화 (i): 어떤 다른 CPU 가 이 CPU 의 캐시에만 있는 캐시 라인의 데이터 아이템에 어토믹 read-modify-write 오퍼레이션을 수행합니다. 이 상태 변경은 “read invalidate” 메세지를 수신하는 것으로 시작되고, 이 CPU 는 응답으로 “read response” 와 “invalidate acknowledge” 메세지를 모두 보냅니다.
- 상태 변화 (j): 이 CPU 가 자신의 캐시에 존재하지 않는 캐시 라인에 데이터 아이템을 저장하고, 따라서 “read invalidate” 메세지를 보냅니다. 이 CPU 는 “read response” 와 모든 “invalidate acknowledge” 메세지들을 받기 전까지 상태 변환은 끝나지 않습니다. 이 캐시라인은 이후 실제 저장이 완료되는 대로 transition (b) 를 통해 “modified” 상태로 변경될 것입니다.
- 상태 변화 (k): 이 CPU 는 자신의 캐시에 존재하지 않는 캐시라인에 데이터 아이템을 읽습니다. 이 CPU 는 “read” 메세지를 보내고, 그에 상응하는 “read response” 를 받는대로 상태 변화를 완료합니다.
- 상태 변화 (l): 어떤 다른 CPU 가 이 캐시 라인의 데이터 아이템에 쓰기를 합니다만, 이 캐시라인은 (예를 들어 현재 CPU 의 캐시 같은) 다른 CPU 의 캐시에도 존재하기 때문에 read-only 상태입니다. 이 상태 변경은 “nvalidat” 메세지의 수신과 함께 시작되고, 이 CPU 는 “invalidate acknowledge” 메세지로 응답을 보냅니다.

Quick Quiz B.5: 앞에 설명된 지연되는 상태 변경들은 하드웨어에서 어떻게 처리하나요? ■

B.2.4 MESI Protocol Example

이제 네 개의 CPU 시스템의 한개짜리 라인을 가지는 각 캐시 사이에 이동하게 될, 최초에는 메모리 어드레스 0 에 존재하는 캐시 라인의 데이터의 시각에서 바라봐 봅시다. Table B.1 이 첫째 행은 오퍼레이션들의 시퀀스를, 두번째 행은 그 오퍼레이션을 수행하는 CPU 를 그리고 세번째 행은 수행되고 있는 오퍼레이션을, 네번째 행은 각 CPU 의 캐시 라인(MESI 상태에 의해 따라오는 메모리 주소)의 상태를, 그리고 마지막의 두 행은 연관된 메모리 내용이 최신인지(“V”) 아닌지(“T”) 표시하면서 이런 데이터 흐름을 보입니다.

최초에, 해당 데이터가 위치하게 될 CPU 캐시 라인은 “invalid” 상태로, 데이터는 메모리에 유효한 채로 존재합니다. CPU 0 가 이 주소 0 의 데이터를 읽어오면, 캐시라인은 CPU 0 의 캐시 안에서 “shared” 상태가 되고 메모리에의 값도 여전히 유효합니다. CPU 3 역시

주소 0 의 이 데이터를 읽어오고, 이로 인해 캐시 라인은 두 CPU 의 캐시 안에서 모두 “shared” 상태가 되며, 여전히 메모리의 값도 유효합니다. 다음으로 CPU 0 가 다른 캐시 라인(주소 8) 의 데이터를 읽어오면, 주소 0 의 데이터는 캐시에서 무효화가 되도록 할 것이고, 그 공간을 주소 8 의 데이터가 차지하게 됩니다. CPU 2 가 이제 주소 0 을 읽게 되는데, 이 CPU 는 곧 거기에 쓰기를 해야 함을 깨닫고, 따라서 독점적 복사본을 얻기 위해 “read invalidate” 메세지를 날려서 (메모리의 값은 여전히 최신 버전이지만) CPU 3 의 캐시에서 해당 캐시 라인을 무효화 시킵니다. 다음으로 CPU 2 는 예상했던 쓰기를 하게 되어, 상태를 “modified” 로 바꿉니다. 메모리에 있는 해당 데이터의 복사본은 이제 과거의 것이 되었습니다. CPU 1 은 어토믹한 증가를 시키기 위해 CPU 2 의 캐시에 있는 데이터를 읽어오고 무효화 시키도록 “read invalidate” 를 이용하게 되는데, 이로 인해 CPU 1 의 복사본은 “modified” 상태가 됩니다 (그리고 메모리의 복사본은 여전히 과거 값입니다). 마지막으로 CPU 1 이 주소 8 의 캐시 라인을 읽게 되는데, 여기선 주소 0 의 데이터를 메모리에 되돌려 놓기 위해 “writeback” 메세지를 사용하게 됩니다.

몇몇 다른 CPU 의 캐시에 값을 남겨둔 채라는 것을 기억해 두세요.

Quick Quiz B.6: 어떤 오퍼레이션들의 시퀀스가 CPU 의 캐시를 모두 “invalid” 상태로 돌려 놓을까요? ■

B.3 Stores Result in Unnecessary Stalls

Figure B.1 에 보인 캐시 구조는 하나의 CPU 로부터 하나의 데이터 아이템으로의 반복적인 읽기와 쓰기에는 좋은 성능을 보이지만, 하나의 캐시 라인에의 첫번째 쓰기 성능은 매우 떨어집니다. 이를 분명히 하기 위해, CPU 0 에서 CPU 1 의 캐시에 있는 캐시라인으로의 쓰기 상황을 보여주는 Figure B.4 를 생각해 보세요. CPU 0 는 쓰기를 하기 전에, 해당 캐시 라인이 도착하기를 기다려야만 하기 때문에, CPU 0 는 더 긴 시간을 멈춰 있어야만 합니다.³

하지만 CPU 0 를 그렇게나 길게 멈춰있게 둘 진정한 이유는 없습니다—무엇보다, CPU 1 이 보내는 캐시라인에 어떤 값이 들어있을지와는 관계 없이, CPU 0 는 어떤 조건 없이 그걸 덮어써 버릴테니까요.

³ 캐시 라인을 하나의 CPU 의 캐시에서 다른 CPU 의 캐시로 전달하는데 필요한 시간은 일반적으로 간단한 레지스터에서 레지스터로 행해지는 인스트럭션에 요구되는 시간의 수십배는 됩니다.

Sequence #	CPU #	Operation	CPU Cache				Memory
			0	1	2	3	
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 B.1: Cache Coherence Example

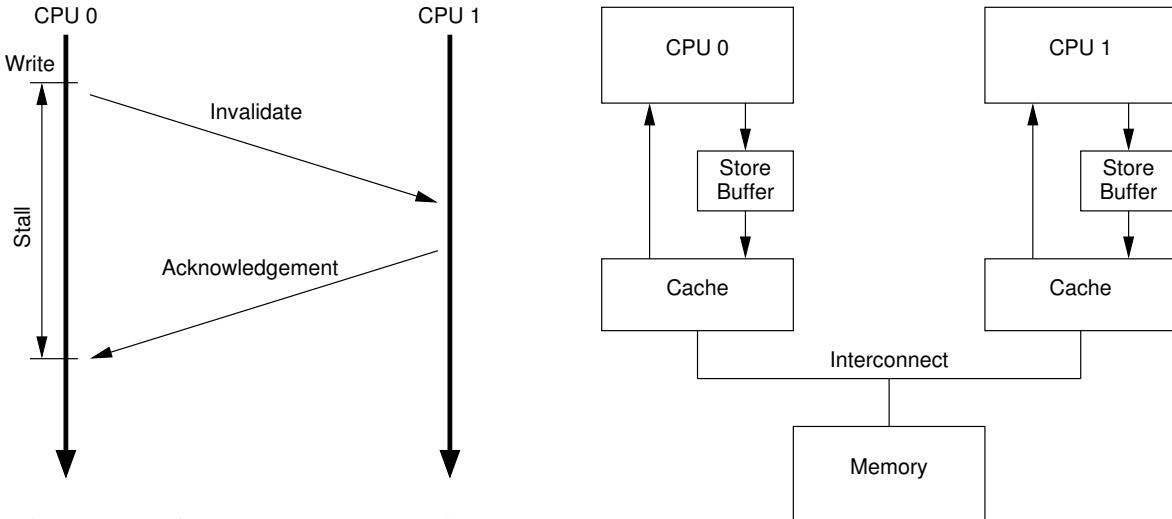


Figure B.4: Writes See Unnecessary Stalls

Figure B.5: Caches With Store Buffers

B.3.1 Store Buffers

이런 불필요한 쓰기 지연을 예방하는 한가지 방법은 각 CPU 와 각 CPU 의 캐시 사이에 Figure B.5 에 보이는 것처럼 “스토어 버퍼”를 추가하는 것입니다. 이렇게 스 토어 버퍼가 추가되면, CPU 0 는 단순히 자신의 쓰기를 자신의 스托어 버퍼에 저장해 두고 실행을 계속할 수 있습니다. 해당 캐시 라인이 마침내 CPU 1 에서 CPU 0 으로 이동하면, 해당 데이터는 스托어 버퍼에서 해당 캐시 라인으로 이동하게 될겁니다.

Quick Quiz B.7: 하지만 스托어 버퍼의 주목적이 멀티프로세서 캐시 일관성 프로토콜의 응답 지연시간을 숨기기 위해서라면, 왜 유니프로세서들도 스托어 버퍼를 가지고 있는거죠? ■

이런 스托어 버퍼는 한 CPU 에 지역적이거나, 하드웨어 멀티쓰레딩을 지원하는 시스템에서는 한 코어에 지역적입니다. 어떤 방식이든, 한 CPU 는 자신에게 할당된 스托어 버퍼에만 액세스 할 수 있습니다. 예를 들

어, Figure B.5 에서 CPU 0 은 CPU 1 의 스托어 버퍼에 액세스 할 수 없고 반대로 마찬가지입니다. 이 제약이 고려사항들을 분리시킴으로써 하드웨어를 단순화 시킵니다: 스托어 버퍼는 반복적인 쓰기에 대해 성능을 향상시키며, CPU 들 (또느 코어들) 사이의 통신에 대한 책임은 캐시 일관성 프로토콜이 완전히 가지고 있습니다. 하지만, 이런 제약이 있다고 해도, 다음 두 섹션에서 이야기할, 여전히 처리되어야만 하는 복잡한 문제들이 존재합니다.

B.3.2 Store Forwarding

첫번째 문제인 스스로에 대한 일관성의 파괴 문제를 보기 위해, 변수 “a” 와 “b” 는 0 값을 가지며, 변수 “a” 를 담는 캐시 라인은 CPU 1 에 소유되어 있고 “b” 를 담는 캐시라인은 CPU 0 이 가지고 있는 초기 상태를 가정해 두고 다음의 코드를 생각해 봅시다.

```

1  a = 1;
2  b = a + 1;
3  assert(b == 2);

```

누군가는 이 단정은 실패하지 않을 거라고 생각할 겁니다. 하지만, Figure B.5에서 보여진 매우 단순한 구조를 사용할 만큼 명청하다면 깜짝 놀랄 겁니다. 그런 시스템은 다음과 같은 이벤트의 시퀀스를 발생시킬 가능성이 있습니다:

1. CPU 0 는 $a = 1$ 을 수행하기 시작합니다.
2. CPU 0 는 “a” 가 캐시에 있는지 보고, 자신의 캐시에 없음을 깨닫습니다.
3. 따라서 CPU 0 는 “a” 를 담고 있는 캐시 라인에 배타적 소유권을 얻기 위해 “read invalidate” 메세지를 보냅니다.
4. CPU 0 는 “a” 로의 스토어를 자신의 스토어 버퍼에 기록합니다.
5. CPU 1 은 “read invalidate” 메세지를 받고, 해당 캐시라인을 보내고 해당 캐시라인을 자신의 캐시에서 지우는 것으로 응답합니다.
6. CPU 0 는 $b = a + 1$ 을 수행합니다.
7. CPU 0 는 CPU 1 으로부터 여전히 “a” 의 값이 0인 해당 캐시 라인을 받습니다.
8. CPU 0 는 자신의 캐시로부터 “a” 를 읽어내고 값 0 을 얻습니다.
9. CPU 0 는 자신의 스토어 버퍼의 항목을 새로 도착한 캐시 라인에 적용해서, 자신의 캐시에 있는 “a” 의 값을 1로 만듭니다.
10. CPU 0 는 앞서 읽혀진 “a” 의 값 0에 1을 더하고, 이를 “b” 를 담고 있는 캐시 라인 (우리는 이게 이미 CPU 0 에 소유되어 있다 가정했습니다) 에 저장합니다.
11. CPU 0 는 `assert(b == 2)` 를 수행하고, 실패합니다.

문제는 우리가 “a” 의 두개의 복사본을 하나는 캐시에, 또 다른 하나는 스토어 버퍼에 가지고 있다는 것입니다.

이 예를 각각의 CPU 는 항상 그것들이 프로그램 순서대로 행하는 것처럼 각자의 오퍼레이션들을 바라봐야 한다는 매우 중요한 보장사항을 깨뜨렸습니다. 이 보장사항을 깨뜨리는 것은 소프트웨어에게 매우 비직관적 인 일이어서 하드웨어 쪽 사람들은 연민을 가지게 되었고 각 CPU 가 로드 오퍼레이션을 수행할 때 Figure B.6

에 보여지는 것처럼 자신의 캐시만이 아니라 스토어 버퍼 역시 참고 (또는 “기웃거리게(snoop)”) 하는 “store forwarding” 을 구현했습니다. 달리 말하자면, 주어진 CPU 의 스토어는 이후에 이어지는 자신의 로드에 캐시를 통하지 않고 곧바로 포워딩 됩니다.

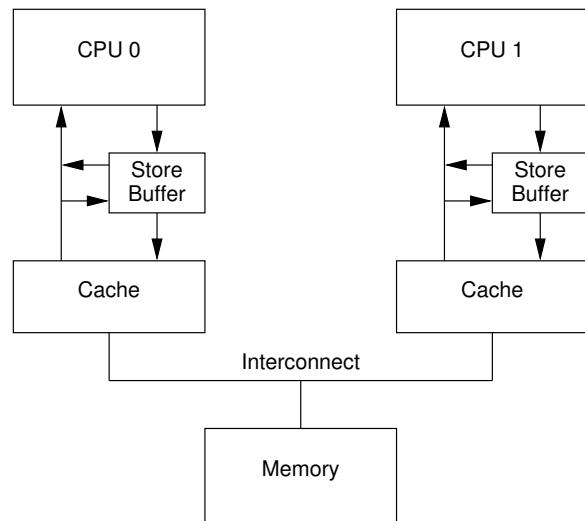


Figure B.6: Caches With Store Forwarding

스토어 포워딩이 존재하면, 앞의 시퀀스에서의 아이템 8 은 “a” 의 올바른 값 1을 스토어 버퍼에서 찾고, 따라서 “b” 의 최종값은 우리가 원했던 대로 2가 될 것입니다.

B.3.3 Store Buffers and Memory Barriers

글로벌 메모리 순서의 위반이라는 두번째 문제를 보기 위해, 변수 “a” 와 “b” 가 초기값 0을 가지고 있다는 가정 하에 다음의 코드를 봅시다:

```

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 }

```

CPU 0 는 `foo()` 를, 그리고 CPU 1 은 `bar()` 를 수행한다고 해봅시다. 그리고 “a” 를 담는 캐시라인은 CPU 1 의 캐시에, 그리고 “b” 를 담는 캐시라인은 CPU 0 에 소

유되어 있습니다. 그러면 다음과 같은 오퍼레이션들의 시퀀스가 생길 수 있습니다:

1. CPU 0 는 `a = 1` 을 실행합니다. 해당 캐시 라인은 CPU 0 의 캐시에 없으므로, CPU 0 는 “a” 의 새로운 값을 스토어 버퍼에 넣고 “read invalidate” 메세지를 던집니다.
2. CPU 1 은 `while (b == 0) continue` 를 수행합니다만, “b” 를 담는 캐시 라인이 자신의 캐시에 없습니다. 따라서 이 CPU 는 “read” 메세지를 보냅니다.
3. CPU 0 이 `b = 1` 을 수행합니다. 이 CPU 는 이 캐시 라인을 이미 가지고 있으므로(달리 말하자면, 해당 캐시 라인은 이미 “modified” 또는 “exclusive” 상태입니다), “b” 의 새 값을 캐시 라인에 저장합니다.
4. CPU 0 이 “read” 메세지를 받고, CPU 1 에게 업데이트 되어 있는 “b” 값을 담은 캐시 라인을 보내며, 또한 자신의 캐시 위의 해당 라인을 “shared” 로 표시합니다.
5. CPU 1 은 “b” 를 담고 있는 캐시라인을 받고 자신의 캐시에 집어넣습니다.
6. CPU 1 은 이제 “b” 의 값이 1임을 확인했으나 `while (b == 0) continue` 의 수행을 끝낼 수 있게 되었고, 다음 문장을 수행합니다.
7. CPU 1 은 `assert (a == 1)` 을 수행하게 되는데, CPU 1 은 “a” 의 예전 값을 들고 있으므로, 이 단정문은 실패합니다.
8. CPU 0 은 “a” 를 담는 캐시 라인을 받고 바로 이때에서야 스토어 버퍼에 머물러 있던 스토어 오퍼레이션을 행해서 CPU 1 의 실패한 단정문이 가능하게 합니다.

Quick Quiz B.8: 앞의 step 1에서, 왜 CPU 0 는 그냥 “invalidate” 가 아니라 “read invalidate” 를 보내는거죠?

하드웨어 설계자들은 이 문제를 직접 도와줄 수는 없는데, CPU 들은 어떤 변수들이 연관되어 있는지, 어떻게 변수들이 연관될 수 있을지 알수가 없기 때문입니다. 따라서, 하드웨어 설계자들은 소프트웨어가 CPU 에게 그런 연관관계를 알려줄 수 있도록 메모리 배리어 인스트럭션들을 제공합니다. 앞의 프로그램 조각은 다음과 같이 메모리 배리어를 포함하도록 수정되어야 합니다:

```

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 }
```

메모리 배리어 `smp_mb()` 는 해당 CPU 가 자신의 스토어 버퍼를 이후의 스토어 오퍼레이션이 대상 변수의 캐시 라인에 값을 써넣기 전에 비우도록 합니다. 해당 CPU 는 단순히 이어서 진행을 하기 전에 스토어 버퍼가 비워질 때까지 멈춰 있을 수도 있고, 다음의 스토어 오퍼레이션들을 이미 스토어 버퍼에 채워져 있던 내용들이 모두 적용될 때까지 스토어 버퍼 안에 들고 있을 수도 있습니다.

이 두번째 방법을 사용하게 되면 오퍼레이션들의 시퀀스는 다음과 같아질 겁니다:

1. CPU 0 는 `a = 1` 을 실행합니다. 해당 캐시 라인은 CPU 0 의 캐시에 없으므로, CPU 0 는 “a” 의 새로운 값을 스토어 버퍼에 넣고 “read invalidate” 메세지를 던집니다.
2. CPU 1 은 `while (b == 0) continue` 를 수행합니다만, “b” 를 담는 캐시 라인이 자신의 캐시에 없습니다. 따라서 이 CPU 는 “read” 메세지를 보냅니다.
3. CPU 0 가 `smp_mb()` 를 수행하고, 현재 스토어 버퍼에 있는 모든 항목(구체적으로는, `a = 1`)을 마크해 둡니다.
4. CPU 0 가 `b = 1` 을 수행합니다. 이 CPU 는 이 캐시 라인을 이미 가지고 있습니다만(달리 말하자면, 해당 캐시 라인은 이미 “modified” 또는 “exclusive” 상태입니다), 스토어 버퍼에 마크된 항목들이 있습니다. 따라서, “b” 의 새 값을 해당 캐시 라인에 저장하는 대신, CPU 0 는 이 동작을 스토어 버퍼에 (하지만 마크되지 않은 자리에) 집어넣습니다.
5. CPU 0 는 “read” 메세지를 받고, “b” 의 원래 값이 저장되어 있는 캐시 라인을 CPU 1 에게 보냅니다. 또한 자신의 이 캐시라인의 복사본을 “shared” 로 바꿔줍니다.
6. CPU 1 은 “b” 를 담고 있는 캐시 라인을 받고 자신의 캐시에 집어넣습니다.

7. CPU 1 은 이제 “b” 의 값을 읽을 수 있습니다만, “b” 의 값이 여전히 0임을 확인하므로, 해당 while 문을 계속 반복 수행하게 됩니다. “b” 의 새 값은 안전하게 CPU 0 의 스토어 버퍼 안에 숨겨져 있습니다.
8. CPU 1 은 “read invalidate” 메세지를 받고, “a” 를 담고 있는 해당 캐시 라인을 CPU 0 에게 보내고 자신의 캐시에서 해당 캐시 라인을 무효화 시킵니다.
9. CPU 0 은 “a” 를 담고 있는 캐시 라인을 받고 스토어 버퍼에 들어가 있던 스토어 오퍼레이션을 행하고 이 캐시 라인을 “modified” 상태로 바꿉니다.
10. “a” 로의 스토어는 스토어 버퍼 내에 `smp_mb()` 에 의해 마크되어 있던 유일한 항목이므로, CPU 0 은 “b” 에의 새 값의 저장도 할 수 있습니다 — “b” 를 담고 있는 캐시 라인이 이제는 “shared” 상태라는 사실은 제외하고.
11. 따라서 CPU 0 는 CPU 1 에게 “invalidate” 메세지를 보냅니다.
12. CPU 1 은 “invalidate” 메세지를 받고, 자신의 캐시로부터 “b” 를 담고 있는 캐시 라인을 무효화 시키고, “acknowledgement” 메세지를 CPU 0 에게 보냅니다.
13. CPU 1 은 `while (b == 0) continue` 를 수행합니다만, “b” 를 담고 있는 캐시 라인이 자신의 캐시에 없습니다. 따라서 “read” 메세지를 CPU 0 에게 보냅니다.
14. CPU 0 는 “read” 메세지를 받고, “b” 의 새 값을 담고 있는 캐시 라인을 CPU 1 에게 보냅니다. 또한 자신이 가지고 있는 이 캐시 라인의 복사본을 “shared” 상태로 바꿉니다.
15. CPU 1 이 “b” 를 담고 있는 캐시 라인을 받고 자신의 캐시에 올려놓습니다.
16. CPU 1 은 이제 “b” 의 값을 읽을 수 있고, “b” 의 값이 1임을 확인하기 되므로 while 루프를 종료하고 다음 문장으로 실행을 넘깁니다.
17. CPU 1 은 `assert (a == 1)` 을 수행하는데, “a” 를 담는 캐시 라인이 자신의 캐시에 없습니다. 이 캐시 라인을 CPU 0 로부터 다시 받으면, 최신 버전의 값을 갖는 “a” 를 가지고 동작하게 되므로, 단정문은 통과됩니다.

볼 수 있듯, 이 작업은 적지 않은 양의 부기 (bookkeeping) 를 필요로 합니다. 몇몇 내용은 직관적으로 간단해 보이겠지만, “a 의 값을 로드하기” 와 같은 명령도 실리콘에서는 수많은 복잡한 단계를 야기할 수 있습니다.

B.4 Store Sequences Result in Unnecessary Stalls

불행히도, 각각의 스토어 버퍼는 비교적 작아야 해서, 적당한 스토어들만 수행하는 CPU 도 스토어 버퍼를 가득 채울 수 있습니다 (예를 들어, 그 모든 것들이 캐시 미스를 초래한다면). 이렇게 되면, CPU 는 계속해서 수행을 하기 전에 스토어 버퍼가 비워지도록 또다시 무효화가 완료되길 기다려야만 합니다. 메모리 배리어 이후에도, 모든 뒤이어 요청되는 스토어 인스트럭션들이 해당 스토어들이 캐시 미스를 내는지 안내는지와 관계 없이 무효화가 완료되길 기다리는 똑같은 경우가 발생할 수 있습니다.

이런 상황은 무효화 응답 메세지가 좀 더 빨리 도착하게 함으로써 개선될 수 있습니다. 이를 가능하게 하는 한가지 방법은 CPU 별 무효화 메세지의 큐, “invalidate queue” 를 사용하는 것입니다.

B.4.1 Invalidate Queues

무효화 응답 메세지가 그렇게 오랜 시간을 소모하는 한 가지 이유는 응답 메세지들은 연관된 캐시 라인이 실제로 무효화 되었음을 보장해야 하는데, 예를 들어 CPU 가 해당 캐시에 있는 데이터의 로드와 스토어를 열심히 하고 있거나 해서 캐시가 바빠 무효화가 지연될 수도 있다는 것입니다. 또한, 짧은 시간 동안 많은 무효화 메세지가 도착하면, 해당 CPU 는 이것들을 처리하는데 바빠서, 다른 CPU 들을 모조리 멈춰있게 만들 수도 있습니다.

하지만, 해당 CPU 는 응답을 보내기 전에 정말로 캐시 라인들을 무효화 해야만 할 필요는 없습니다. 대신 CPU 가 나중에 해당 캐시 라인과 관련된 메세지를 보내기 전까지는 그 메세지가 처리되어야만 한다는 점만 기억해두고 무효화 메세지를 큐에 집어넣어 둘 수도 있습니다.

B.4.2 Invalidate Queues and Invalidate Acknowledge

Figure B.7 는 무효화 큐를 가진 시스템을 그리고 있습니다. 무효화 큐를 가진 CPU 는 관련된 캐시 라인이 실제로 무효화 되는 걸 기다리지 않고 무효화 메세지를 큐에 넣자마자 응답 메세지를 보낼 수 있습니다. 물론, 해당 CPU 는 무효화 메세지들을 보낼 준비를 할 때 자신의 무효화 큐를 확인해 봐야 합니다 — 관련된 캐시 라인을 위한 항목이 무효화 큐에 존재한다면, 해당 CPU 는 무효화 메세지를 곧바로 보낼 수 없습니다; 대신 무효화 큐의 해당 항목이 처리되길 기다려야 합니다.

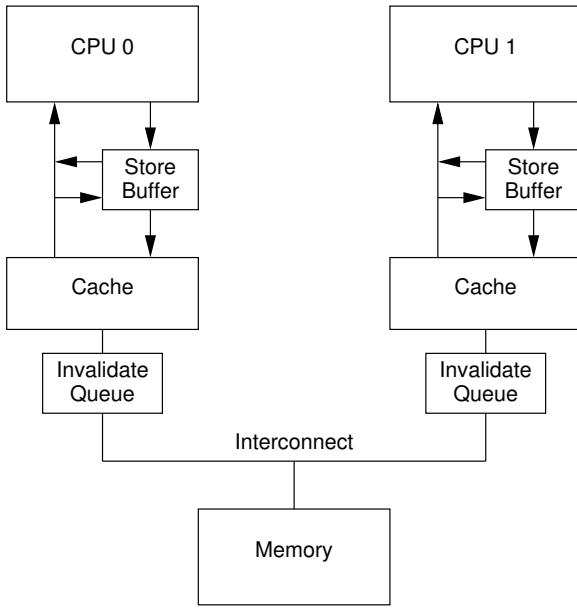


Figure B.7: Caches With Invalidate Queues

한 항목을 무효화 큐에 집어넣는 것은 결국은 해당 CPU 가 해당 항목과 관련된 캐시 라인에 대한 어떤 MESI 프로토콜 메세지를 나중에 보내기 전까지는 해당 항목을 처리하겠다는 약속입니다. 관련된 데이터 구조에 너무 많은 요청이 들어오지 않는다면 해당 CPU 가 그 약속으로 인해 어려움을 겪는 일은 거의 없을 겁니다.

하지만, 무효화 메세지가 무효화 큐에 들어가 있을 수 있다는 사실은 다음 섹션에서 이야기할 추가적인 메모리 오퍼레이션의 잘못된 재배치 문제를 야기합니다.

B.4.3 Invalidate Queues and Memory Barriers

CPU 들이 무효화 요청을 큐에 집어넣기만 하고 곧바로 응답하는 경우를 생각해 봅시다. 이 방법은 스토어를 하는 CPU 들에게 보이는 캐시 무효화 대기시간을 최소화 해주지만, 다음의 예에서 보여지듯이 메모리 배리어들을 패배시킬 수가 있습니다.

“a” 와 “b” 가 초기값 0을 가지고, “a” 는 read-only 로 복사되어 있고 (MESI “shared” 상태), “b” 는 CPU 0 에 의해 소유되어 있는 (MESI “exclusive” 또는 “modified” 상태) 라고 가정합시다. 그리고 CPU 0 는 다음의 코드에서 `foo()` 함수를 실행하고, 그동안 CPU 1 은 `bar()` 함수를 실행하고 있다고 생각해 봅시다:

```

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 }

```

그럼 다음과 같은 오퍼레이션들의 시퀀스가 가능합니다:

1. CPU 0 이 `a = 1` 을 실행합니다. 관련된 캐시 라인 이 CPU 0 의 캐시에 read-only 상태이므로, CPU 0 는 “a” 의 새 값을 스토어 버퍼에 넣고 해당 캐시 라인을 CPU 1 의 캐시로부터 없애기 위해 “invalidate” 메세지를 전송합니다.
2. CPU 1 이 `while (b == 0) continue` 를 실행합니다만, “b” 를 담고 있는 캐시라인이 자신의 캐시에 없습니다. 따라서 “read” 메세지를 보냅니다.
3. CPU 1 이 CPU 0 의 “invalidate” 메세지를 받고, 큐에 넣은 후, 곧바로 그에 응답합니다.
4. CPU 0 은 CPU 1 로부터 응답을 받고, 따라서 line 4 의 `smp_mb()` 를 수행할 수 있게 되어 “a” 의 값을 스토어 버퍼에서 자신의 캐시 라인으로 옮깁니다.
5. CPU 0 가 `b = 1` 을 수행합니다. 이미 이 캐시 라인을 소유하고 있으므로 (달리 말하자면, 해당 캐시 라인은 이미 “modified” 또는 “exclusive” 상태입니다), “b” 의 새 값을 자신의 캐시 라인에 써넣습니다.
6. CPU 0 는 “read” 메세지를 받고, 이제 막 업데이트된 “b” 의 값을 담고 있는 캐시 라인을 CPU 1 에게 전송하고, 자신의 캐시에서 해당 캐시 라인을 “shared” 로 표시합니다.
7. CPU 1 이 “b” 의 값을 포함하는 캐시 라인을 받고 자신의 캐시에 올립니다.
8. CPU 1 은 이제 `while (b == 0) continue` 의 수행을 끝내고, “b” 의 값이 1임을 확인하므로, 다음 문장으로 수행을 넘깁니다.
9. CPU 1 은 `assert(a == 1)` 을 수행하지만, “a” 의 이전 값이 여전히 CPU 1 의 캐시에 있으므로, 이 단정문은 실패합니다.

10. 단정문이 실패했음에도, CPU 1은 큐에 들어가 있는 “invalidate” 메세지를 처리하고, (뻔뻔스럽게) 자신의 캐시에서 “a”를 담고 있는 캐시 라인을 무효화 시킵니다.

Quick Quiz B.9: Section B.4.3의 첫번째 시나리오의 스텝 1에서, 왜 “read invalidate”가 아니라 “invalidate”를 보내는 거죠? CPU 0은 “a” 외에도 이 캐시 라인을 공유하는 다른 변수들의 값을 필요로 할수도 있지 않나요? ■

메모리 배리어를 무시되게 한다면 무효화 응답을 빠르게 하는 것은 별 의미가 없어집니다. 하지만, 메모리 배리어 인스트럭션은 한 CPU가 메모리 배리어를 수행할 때, 무효화 큐에 있는 모든 항목을 표시해 두고 다음에 요청되는 로드 오퍼레이션들은 모든 표시된 항목들이 해당 CPU의 캐시에 적용될 때까지 기다리게 하도록 무효화 큐와 상호작용 할 수 있습니다. 따라서, 우린 bar 함수에 다음과 같이 메모리 배리어를 추가할 수 있습니다:

```

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 B.10: 뭐라구요??? CPU는 while 루프가 끝나기 전까지는 assert()를 수행할 수 없는데 왜 여기에 메모리 배리어가 필요해요? ■

변경된 코드에서 오퍼레이션들의 시퀀스는 다음과 같이 될겁니다:

- CPU 0이 a = 1을 실행합니다. 관련된 캐시 라인은 CPU 0의 캐시에 read-only 상태이므로, CPU 0은 “a”的 새 값을 스토어 버퍼에 넣고 해당 캐시 라인을 CPU 1의 캐시로부터 없애기 위해 “invalidate” 메세지를 전송합니다.
- CPU 1이 while (b == 0) continue를 실행합니다만, “b”를 담고 있는 캐시라인이 자신의 캐시에 없습니다. 따라서 “read” 메세지를 보냅니다.
- CPU 1이 CPU 0의 “invalidate” 메세지를 받고, 큐에 넣은 후, 곧바로 그에 응답합니다.

- CPU 0은 CPU 1로부터 응답을 받고, 따라서 line 4의 smp_mb()를 수행할 수 있게 되어 “a”的 값을 스토어 버퍼에서 자신의 캐시 라인으로 옮깁니다.
- CPU 0이 b = 1을 수행합니다. 이미 이 캐시 라인을 소유하고 있으므로 (달리 말하자면, 해당 캐시 라인은 이미 “modified” 또는 “exclusive” 상태입니다), “b”的 새 값을 자신의 캐시 라인에 써넣습니다.
- CPU 0은 “read” 메세지를 받고, 이제 막 업데이트된 “b”的 값을 담고 있는 캐시 라인을 CPU 1에게 전송하고, 자신의 캐시에서 해당 캐시 라인을 “shared”로 표시합니다.
- CPU 1이 “b”的 값을 포함하는 캐시 라인을 받고 자신의 캐시에 올립니다.
- CPU 1은 이제 while (b == 0) continue의 수행을 끝내고, “b”的 값이 1임을 확인하므로, 다음 문장의, 메모리 배리어를 수행하게 됩니다.
- CPU 1은 이제 모든 무효화 큐에 들어가 있던 메세지들이 처리될 때까지 기다려야만 합니다.
- CPU 1은 이제 큐에 들어가 있던 “invalidate” 메세지를 수행하게 되어, 자신의 캐시에서 “a”를 담고 있는 캐시 라인을 무효화 시킵니다.
- CPU 1은 assert(a == 1)을 수행하는데, “a”를 담고 있는 캐시 라인이 더이상 CPU 1의 캐시에 존재하지 않으므로, “read” 메세지를 보냅니다.
- CPU 0은 이 “read” 메세지에 “a”的 새로운 값을 담고 있는 캐시 라인으로 응답합니다.
- CPU 1은 “a”的 값 1을 담고 있는 이 캐시 라인을 받고, 따라서 단정문은 성공합니다.

훨씬 많이 주고받게 된 MESI 메세지들과 함께, CPU 을바른답에도 달했습니다. 이 섹션은 왜 CPU 설계자들이 캐시 일관성 최적화에 매우 조심스러워야만 하는지 이야기 했습니다.

B.5 Read and Write Memory Barriers

앞의 섹션에서, 메모리 배리어들은 스토어 버퍼와 무효화 큐 둘 다의 항목들을 표시하는데 사용되었습니다. 하지만 우리의 코드에서, foo()는 무효화 큐에 대해 별다른 일을 할 이유가 없었고, bar()는 스토어 버퍼에 추가적인 일을 할 이유가 없었습니다.

따라서 많은 CPU 들이 이 두가지 일 중 하나만 하느 좀 더 약한 메모리 배리어 인스트럭션을 제공합니다. 간단히 말해서, “읽기 메모리 배리어”는 무효화 큐만을, 그리고 “쓰기 메모리 배리어”는 스도어 베피만을 막아야, 전체 메모리 배리어는 둘 다 합니다.

이로 인해 읽기 메모리 배리어는 자신을 수행한 CPU에서의 로드 오퍼레이션들에 대해서만 순서를 맞춰줘서, 읽기 메모리 배리어 앞에 있었던 로드 오퍼레이션들은 읽기 메모리 배리어 뒤에 따라오는 로드 오퍼레이션들 이전에 완료된 것으로 보이게 하는 효과를 냅니다. 유사하게, 쓰기 메모리 배리어는 자신을 수행한 CPU에서 스도어 오퍼레이션들만을 순서를 맞춰줘서, 쓰기 메모리 배리어 앞의 모든 스도어 오퍼레이션들이 쓰기 메모리 배리어를 뒤따르는 스도어 오퍼레이션들 이전에 끝난 것처럼 보이게 합니다. 전체 메모리 배리어는 로드와 스도어 둘 다를 순서세웁니다만, 아까도 이야기 했듯 메모리 배리어를 실행하는 CPU 위에서만 그렇습니다.

`foo` 와 `bar` 를 읽기 메모리 배리어와 쓰기 메모리 배리어를 사용하도록 바꾸면 다음과 같을 것입니다:

```

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 }
```

일부 컴퓨터는 이보다도 다양한 메모리 배리어를 가지고 있기도 합니다만, 이 세개의 변종만 이해해도 일반적인 메모리 배리어에 대해 잘 소개 할 수 있을 것입니다.

B.6 Example Memory-Barrier Sequences

이 섹션은 괜찮아 보이지만 약간씩 잘못된 메모리 배리어의 사용 예를 제공합니다. 대부분은 많은 상황에서 올바르게 동작할 것이고, 일부는 특정 CPU에서라면 항상 잘 동작하겠지만, 모든 CPU에서 안정적으로 동작하는 코드를 만들고자 한다면 이런 잘못된 사용은 없어야만 합니다. 이 약간씩 잘못된 부분을 잘 보기 위해, 먼저 순서세우기에 적대적인 아키텍쳐에 집중을 해 볼 필요가

있습니다.

B.6.1 Ordering-Hostile Architecture

지난 수십년간 여러개의 순서세우기에 적대적인 컴퓨터 시스템들이 만들어졌습니다만, 적대성의 본질은 항상 매우 교묘해서 그것을 이해하는데에는 특정 하드웨어에 대한 자세한 지식이 필요했습니다. 특정 하드웨어 벤더를 고르고 독자 여러분을 자세한 기술 사양으로 끌어들이는 대신, 실존하지는 않지만 최대한 메모리 순서에 적대적인 컴퓨터 아키텍쳐를 설계해 봄시다.⁴

이 하드웨어는 다음과 같은 순서 규칙 [McK05a, McK05b]을 지켜야만 합니다:

1. 각 CPU 는 항상 자기 자신의 메모리 액세스를 프로그램 순서로 이루어진 것으로 인지해야 합니다.
2. CPU 들은 스도어 오퍼레이션들만으로 구성된 오펠에시녀들을 두 오퍼레이션들이 서로 다른 위치에 대해 행해진다면 순서를 바꿀 것입니다.
3. 읽기 메모리 배리어 (`smp_rmb()`) 앞에 있는 한 CPU 의 모든 로드 오퍼레이션들은 모든 CPU 들에 그 읽기 메모리 배리어 뒤에 따라오는 로드 오퍼레이션들보다 앞서 수행된 것으로 인지될 것입니다.
4. 쓰기 메모리 배리어 (`smp_wmb()`)를 앞서는 한 CPU 의 모든 스도어 오퍼레이션들은 모든 CPU 들에 그 쓰기 메모리 배리어 뒤에 따라오는 스도어 오퍼레이션들보다 앞서 수행된 것으로 인지될 것입니다.
5. 전체 메모리 배리어 (`smp_mb()`)를 앞서는 한 CPU 의 모든 액세스들(로드 오퍼레이션들과 스도어 오퍼레이션들)은 그 메모리 배리어를 뒤따르는 어떤 다른 액세스들보다 먼저 수행된 것으로 모든 CPU 들에 인지될 것입니다.

Quick Quiz B.11: 각 CPU 는 자신의 메모리 액세스를 순서대로 보게 된다는 보장 사항은 각 유저 레벨 쓰레드 역시 자신의 메모리 액세스를 순서대로 볼 것이라고 보장하나요? 그렇다면 왜 그렇고, 아니라면 왜 아니죠? ■

주어진 노드의 CPU 들에게 interconnect 밴드위쓰를 공정하게 할당해 줄 수 있도록 각 노드의 interconnect interface 마다 CPU 별 큐를 제공하는, Figure B.8 와

⁴ 실제 하드웨어 아키텍쳐를 자세히 알아보는 것을 선호하는 독자 여러분들은 CPU 벤더들의 매뉴얼들 [SW95, Adv02, Int02b, IBM94, LHF05, SPA94, Int04b, Int04a, Int04c], Gharachorloo 의 박사 학위 논문 [Gha95], Peter Sewell 의 작업물 [Sew], 또는 Sorin, Hill, 그리고 Wood 의 훌륭한 하드웨어에 기반한 초급 독본 [SHW11] 을 참고해 보시기 바랍니다.

같이 커다란 non-uniform 캐시 아키텍쳐 (NUCA) 시스템을 생각해 보세요. 한 CPU 의 액세스들은 그 CPU에서 수행한 메모리 배리어에 의해 의도한대로 순서가 세워지지만, 두 개의 CPU 째의 액세스들의 상대적인 순서는 이후 보게 되듯 상당히 자유롭게 재배치 될 수 있습니다.⁵

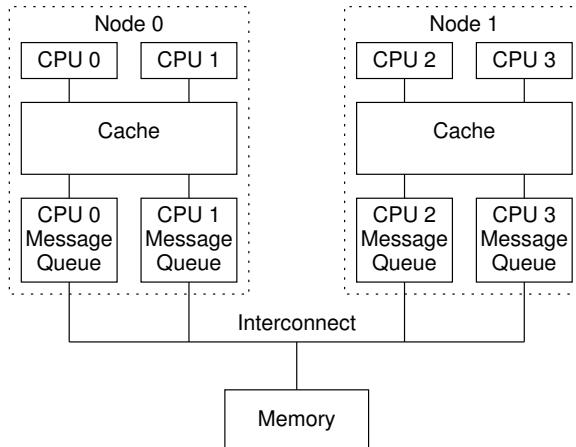


Figure B.8: Example Ordering-Hostile Architecture

B.6.2 Example 1

Table B.2 는 각각 CPU 0, 1, 2 에서 동시에 수행되는 코드를 보이고 있습니다. “a”, “b”, “c” 는 처음엔 0 값을 갖습니다.

CPU 0 가 최근에 많은 캐시 미스를 겪었고, 따라서 메세지 큐가 꽉 찼지만, CPU 1 은 캐시 안에서 대부분의 수행을 마무리 지었고, 따라서 메세지 큐가 비어있다고 생각해 봅시다. 이제 CPU 0 의 “a” 와 “b” 에의 값 대입은 Node 0 의 캐시에 곧바로 이루어질 (따라서 CPU 1 에게 곧바로 보일 것이고) 테지만, CPU 0 의 메세지 큐는 이미 꽉 차 있으므로 바깥에 그 현상이 보여지려면 큐에 차 있는 작업들이 처리되길 기다려야 할 겁니다. 반면, CPU 1 의 “c” 에의 값 대입은 CPU 1 의 비어있는 큐를 통해 곧바로 이루어질 것입니다. 따라서, CPU 2 는 CPU 1 의 “c” 에의 값 대입을 CPU 0 의 “a” 에의 값 대입 이전에 보게 될 것이고, 메모리 배리어에도 불구하고 단정문이 실패하게 만들 것입니다.

⁵ 진짜 하드웨어 아키텍트들이나 디자이너들은 여기에 강력하게 반발할텐데, 두 CPU 가 모두 액세스 하는 캐시 라인에 관한 메세지를 어떤 큐가 처리해야 하는지에 대한 가능성에 대해, 이 예가 포함하고 있을 수 있는 많은 레이스 상황에 대해 아무것도 말할 수 없기 때문에, 좀 화가 날 수 있기 때문입니다. 제가 말 할 수 있는 건 “더 나은 예를 주세요.” 뿐입니다.

따라서, 컴파일러와 CPU 모두 코드가 단정문을 실패하도록 만들 수 있으므로, 다양한 환경에 포팅을 해야 하는 코드라면 이 단정문 통과에 기대선 안됩니다.

Quick Quiz B.12: CPU 1 의 “while” 과 “c” 값 대입문 사이에 메모리 배리어를 넣는 것으로 이 코드를 고칠 수 있지 않을까요? 고칠 수 있다면 왜이고 고칠 수 없다면 그건 또 왜죠? ■

B.6.3 Example 2

Table B.3 는 각각 CPU 0, 1, 2 에서 동시에 수행되는 코드를 보이고 있습니다. “a”, “b”, “c” 는 처음엔 0 값을 갖습니다.

이번에도, CPU 0 는 최근 많은 캐시 미스를 겪었어서 메세지 큐가 꽉 차 있고, CPU 1 은 캐시 안에서만 수행을 마무리 해왔기에 CPU 1 의 메세지 큐는 비어 있는 상황을 생각해 봅시다. CPU 0 의 “a” 에의 값 대입은 Node 0 의 캐시에 곧바로 반영될 (따라서 곧바로 CPU 1 에게 보이게 됩니다) 것이지만, CPU 0 의 메세지 큐가 가득 차 있으므로 곧바로 외부에 보이지는 못할 것입니다. 반면, CPU 1 의 “b” 에의 값 대입은 CPU 1 의 비어있는 메세지 큐를 곧바로 넘어갑니다. 따라서, CPU 2 는 CPU 1 의 “b” 에의 값 대입을 CPU 0 의 “a” 에의 값 대입 이전에 보게 되고, 메모리 배리어에도 불구하고 단정문의 실패를 야기하게 됩니다.

이론적으로, 포터블한 코드는 이 예제의 코드를 사용해선 안됩니다만, 실제로는 앞에서와 같이 대부분의 주류 컴퓨터 시스템에서 이 코드도 잘 동작합니다.

B.6.4 Example 3

Table B.3 는 각각 CPU 0, 1, 2 에서 동시에 수행되는 코드를 보이고 있습니다. 모든 변수는 처음엔 0 값을 갖습니다.

CPU 1 도 CPU 2 도 CPU 0 의 line 3 에서의 “b” 에의 값 할당을 보기 전까지는 line 5 로 넘어갈 수 없음을 알아 듭시다. CPU 1 과 2 가 line 4 의 메모리 배리어를 실행한다면, 둘 다 CPU 0 의 line 2 의 메모리 배리어 앞의 값 할당을 볼 수 있도록 보장됩니다. 비슷하게, line 8 의 CPU 0 의 메모리 배리어는 line 4 의, CPU 1 과 2 에 의한 메모리 배리어와 짹을 이루므로 CPU 0 는 line 9 의 “e” 의 값 할당을 자신의 “a” 에의 값 할당이 다른 CPU 들 다에게 보이기 전까지는 수행할 수 없을 것입니다. 따라서, CPU 2 의 line 9 에서의 단정문은 실패하지 않을 것이 보장됩니다.

Quick Quiz B.13: Table B.4 에서 CPU 1 과 2 의 line 3-5 가 인터럽트 핸들러 안에서 수행되고, CPU 2 의 line 9 는 프로세스 레벨에서 수행된다고 생각해 봅시다. 코드가 정확히 동작하도록 하는데, 달리 말하자면 단정문이

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 B.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 B.3: Memory Barrier Example 2

	CPU 0	CPU 1	CPU 2
1	a = 1;		
2	smp_wmb();		
3	b = 1;	while (b == 0);	
4		smp_mb();	
5		c = 1;	
6	while (c == 0);		while (b == 0);
7	while (d == 0);		smp_mb();
8	smp_mb();		d = 1;
9	e = 1;		assert(e == 0 a == 1);

Table B.4: Memory Barrier Example 3

실패하지 않도록 하는데 어떤 변경이 필요할까요? 필요하다면 무엇일까요? ■

Quick Quiz B.14: Table B.4 의 예에서 CPU 2 가 `assert (e==0 || c==1)` 을 수행하면 단정문은 실패할 수 있을까요? ■

리눅스 커널의 `synchronize_rcu()` 는 이 예에서 보인 것과 유사한 알고리즘을 사용합니다.

B.7 Memory-Barrier Instructions For Specific CPUs

각 CPU 는 Table B.5 에 보여진 것처럼 각자 자신의 독특한 메모리 배리어 인스트럭션들을 갖는데, 이는 호환성을 갖게 하는데 어려움으로 작용할 수 있습니다. 실제로, `pthread` 와 Java 를 포함한 많은 소프트웨어 환경에서는 메모리 배리어의 직접적인 사용을 금지해 버리고, 프로그래머가 필요한 정도만의 배타적 수행을 위한 도구만을 사용할 수 있도록 제한하고 있습니다. 표를 보면, 첫번째 네개의 행은 해당 열의 CPU 가 로드와 스토어를 네개의 조합으로 재배치를 할 수 있는지 보입니다. 다음의 두 행은 해당 열의 CPU 가 로드와 스토어를 어토믹 인스트럭션과 함께 재배치 할 수 있는지 보입니다.

일곱번째 행, data-dependent reads reordered 는 좀 설명이 필요한데, 뒤의 Alpha CPU 들을 설명하는 섹션에서 자세히 설명합니다. 여기서 간단히 설명하자면 Alpha 는 연결된 데이터 구조의 읽는 쪽은 물론 업데이트 하는 쪽에도 메모리 배리어를 필요로 합니다. 맞습니다, 이는 Alpha 는 포인터로 가리켜진 데이터를 포인터 자체를 가져오기 전에도 가져오는 효과를 낼 수 있고, 이는 좀 이상하게 들리겠지만 사실입니다. 거짓말 같다면 “ask the wizard” 문서 [Com01] 를 참고하세요. 이런 극도로 완화된 메모리 모델의 장점으로 Alpha 는 더 간단한 캐시 하드웨어를 사용할 수 있어서 Alpha 의 전성기 때의 그 높은 클럭 주파수를 가능하게 했습니다.

마지막 두 행은 해당 열의 CPU 가 비일관적인 캐시와 파이프라인을 가질 수 있는지 보입니다. 그런 CPU 들은 스스로를 수정하는 코드를 수행하기 위해서는 특별한 인스트럭션들이 필요합니다.

괄호에 싸인 CPU 이름들은 구조적으로 허용된 모드들을 의미하는데, 실제로는 잘 사용되지 않습니다.

이런, 흔한 메모리 배리어에 대해 “그냥 안된다고 말하기” 방법은 적용될 수 있는 곳에서는 매우 합리적입니다만, 리눅스 커널과 같이 메모리 배리어의 직접적 사용이 필요한 환경도 존재합니다. 따라서, 리눅스는 조심스럽게 선택된 메모리 배리어 기능의 최소한의 공통분모 집합을 제공하는데, 다음과 같습니다:

- `smp_mb()`: 로드 오퍼레이션들과 스토어 오퍼레이션들 모두 순서 맞추는 “메모리 배리어”. 이는 이

	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 B.5: Summary of Memory Ordering

메모리 배리어보다 앞의 로드들과 스토어들은 이 메모리 배리어 뒤의 로드들과 스토어들보다 먼저 메모리에 처리될 것이라 뜻입니다.

- `smp_rmb()` : 로드들만 순서 맞추는 “읽기 메모리 배리어”.
- `smp_wmb()` : 스토어들만 순서 맞추는 “쓰기 메모리 배리어”.
- `smp_read_barrier_depends()` 는 앞의 오퍼레이션들에 의존적인 뒤따르는 오퍼레이션들을 순서맞춰지게 만듭니다. 이 기능은 Alpha 이외의 플랫폼에서는 no-op 입니다.
- `mmiowb()` 는 글로벌 스피너으로 보호되는 MMIO 쓰기들의 순서를 강제합니다. 이 기능은 스피너 안의 메모리 배리어가 이미 MMIO 순서를 강제하는 플랫폼들에서는 no-op입니다. `mmiowb()` 가 no-op 이 아닌 것으로 정의되는 플랫폼들 중 일부로는 (전체는 아닙니다) IA64, FRV, MIPS, 그리고 SH 시스템들이 포함됩니다. 이 기능은 비교적 새로운 기능이라서 비교적 적은 드라이버들만이 이 기능을 사용합니다.

The `smp_mb()`, `smp_rmb()`, and `smp_wmb()` 는 또한 컴파일러가 배리어 전후로 메모리 오퍼레이션 순서를 재배치 하는 효과를 줄 수 있는 최적화를 하지 않도록 강제합니다. `smp_read_barrier_depends()` 기능 역시 비슷한 효과를 갖지만, Alpha CPU 들에서만 그렇습니다. 이런 기능들에 대해 더 자세한 정보를 위해 Section 14.2 를 참고하세요.

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.

이런 기능들은 SMP 커널에서만 코드를 생성해냅니다만, UP 커널에서도 메모리 배리어를 생성해내는 UP 버전(`mb()`, `rmb()`, `wmb()`, 그리고 `read_barrier_depends()`)을 각각 갖습니다. 대부분의 경우에는 `smp_` 버전들이 사용되어야 합니다. 하지만, 이 UP 버전들은 드라이버를 작성할 때 유용한데, MMIO 액세스들은 UP 커널에서도 순서를 맞춰줘야 하기 때문입니다. 메모리 배리어 인스트럭션이 없다면, CPU 도 컴파일러도 이런 액세스들을 제멋대로 재배치할 것이고, 최선의 경우에도 디바이스는 이상하게 동작할 것이고, 어쩌면 커널 크래시를 내거나, 심지어는 하드웨어를 손상시킬 수도 있을 것입니다.

따라서 대부분의 커널 프로그래머들은 이 인터페이스들을 사용한다면 각 CPU 전체의 메모리 배리어 특성을 걱정하지 않아도 됩니다. 특정 CPU 의 아키텍쳐에 특화된 코드에 심화된 작업을 하고 있다면, 물론 이야기가 다릅니다.

게다가, 모든 리눅스의 락킹 기능들 (스핀락, reader-writer 락, 세마포어, RCU, ...) 은 필요한 모든 배리어 기능을 내포하고 있습니다. 따라서 이런 기능을 사용하고 있는 코드로 작업하고 있다면, 리눅스의 메모리 순서 세우기 기능에 대해서는 걱정하지 않아도 됩니다.

그렇다면 하나, 각 CPU 의 메모리 일관성 모델에 대한 깊은 지식은, 말하자면 아키텍쳐에 특화된 코드나 동기화 기능을 작성할 때 등의 상황 시, 디버깅에 매우 큰 도움이 될 수 있습니다.

한편, 어떤 사람들은 얇은 지식은 매우 위험한 것이라 이야기 하곤 합니다. 많은 지식을 가지고 만들 수 있는 피해를 생각해 보세요! 각 CPU 의 메모리 일관성 모델에 대해 더 이해하고 싶은 분들을 위해, 다음 섹션에서는 가장 대중적이고 유명한 CPU 들에 대해 설명합니다. 해당 CPU 의 문서를 읽는걸 아예 대체할 수는 없지만 이 섹션들은 좋은 시작점이 될 것입니다.

B.7.1 Alpha

이미 사망 예고가 떨어진 CPU 에 대해 이야기 하는게 이상해 보일 수 있지만, Alpha 는 가장 완화된 메모리 순서 모델을 가지고 있어서, 메모리 오퍼레이션들을 가장 과감하게 재배치 하기 때문에 한번 볼 가치가 있습니다. 때문에 Alpha 를 포함해 모든 CPU 들에서 동작해야 하는 리눅스 커널의 메모리 순서 기능들은 Alpha 로부터 정의되었습니다. 따라서 Alpha 를 이해하는 것은 리눅스 커널 해커에게 매우 중요합니다.

Alpha 와 다른 CPU 들간의 차이점은 Figure B.9 의 코드에 나타나 있습니다. 이 그림의 line 9 의 `smp_wmb()` 는 line 6-8 의 원소 초기화가 line 10에서 이 원소가 리스트에 추가되기 전에 수행되는 것을 보장해서 lock-free 탐색이 정확히 동작하게 합니다. 다만, 이 보장은 Alpha 를 제외한 모든 CPU 들에만 보장됩니다.

Alpha 는 극단적으로 완화된 메모리 순서 규칙을 가지고 있어서 Figure B.9 line 20 의 코드는 line 6-8 의 초기화 이전의 예전 쓰레기 값을 볼 수도 있습니다.

Figure B.10 는 캐시 라인들이 번갈아가며 다른 파티션들에서 처리되는, 분할된 캐시를 갖는 과감한 병렬 머신에서 이런 일이 어떻게 일어날 수 있는지 보입니다. 리스트 헤더 `head` 가 `cache bank 0`에서 처리되고, 새 원소는 `cache bank 1`에서 처리된다고 생각해 봅시다. Alpha 에서, Figure B.9 line 6-8 에 의해 야기된 캐시 무효화는 `smp_wmb()` 에 의해 line 10 이전에 interconnect 에 도착할 것을 보장합니다만, 새 값이 읽는 CPU 의 코

```

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 B.9: Insert and Lock-Free Search

어에 도착하는 순서에 대해서는 아무런 보장을 하지 않습니다. 예를 들어, 읽는 CPU의 cache bank 1이 매우 바쁜 한편 cache bank 0은 아무 일도 하고 있지 않습니다. 이런 경우 새 원소에 대한 캐시 무효화 결과의 도착이 지연되어 읽는 CPU는 포인터의 새 값은 받지만, 캐시되어 있던, 새 원소의 예전 값을 읽는 결과가 나올 수 있습니다. 여전히 제가 가상의 이야기를 하고 있다고 생각된다면, 더 자세한 정보를 위해 앞서 이야기한 웹사이트를 한번 방문해 보세요.⁶

포인터를 가져오는 것과 포인터를 디레퍼런스 사이에 smp_rmb()를 넣어 문제를 해결할 수 있을 겁니다. 하지만, 이는 읽는 쪽의 데이터의 의존성을 고려하는 (i386, IA64, PPC, 그리고 SPARC 같은) 시스템 들에는 불필요한 오버헤드를 가져올 수 있습니다. 이런 시스템에서의 오버헤드를 없애기 위해 smp_read_barrier_depends() 기능이 리눅스 2.6 커널에 추가되었습니다. 이 기능은 Figure B.11의 line 19에 사용될 수 있을 겁니다.

모든 읽는 CPU들이 쓰는 CPU의 쓰기를 순서대로 읽도록 강제하는 소프트웨어 배리어를 smp_wmb() 자리에 구현할 수도 있을 겁니다. 하지만, 이 방법은 리눅스 커뮤니티에 Alpha처럼 극단적으로 완화된 순서를 갖는 CPU들에서는 지나친 오버헤드를 가져올 것이라 판단되었습니다. 이 소프트웨어 배리어는 프로세서간 인터럽트(IPI)를 모든 CPU들에 보내는 것으로 구현될 수도 있습니다. IPI의 사용 시, CPU는 메모리 배리어

⁶ 물론, 눈치 빠른 독자는 더럽고 비열한 Alpha는 더이상 어디에도 없지만, Section B.6.1의 가상의 아키텍처가 (감사하게도) 이 케이스가 된다는 것을 이미 알아차렸을 겁니다.

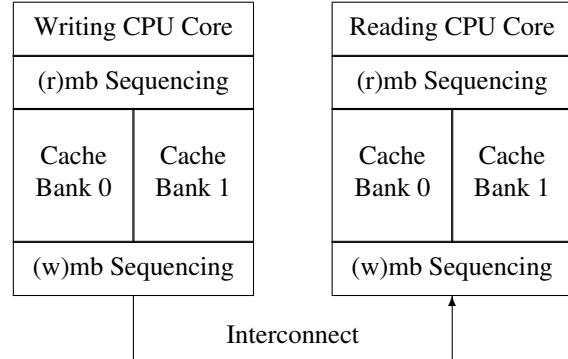


Figure B.10: Why smp_read_barrier_depends() is Required

```

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 B.11: Safe Insert and Lock-Free Search

인스트럭션을 수행할 수도 있어 메모리 배리어 슷다운을 구현할 수도 있습니다. 데드락을 막기 위해 추가적인 로직이 필요합니다. 물론, 데이터의 의존성을 지키는 CPU들은 그런 배리어를 단순히 smp_wmb()로 정의할 것입니다. Alpha는 이미 져버렸기 때문에 이에 대해선 나중에 한번 더 생각해 볼 필요가 있습니다.

리눅스 메모리 배리어 기능들은 그 이름을 Alpha의 인스트럭션들에서 가져왔기 때문에, smp_mb()는 mb, smp_rmb()는 rmb, 그리고 smp_wmb()는 wmb입니다. Alpha는 smp_read_barrier_depends()가 no-

op 이 아니라 `smp_mb()` 인 유일한 CPU 입니다.

Quick Quiz B.15: Alpha 의 `smp_read_barrier_depends()` 는 `smp_rmb()` 가 아니라 `smp_mb()` 인가요? ■

Alpha 에 대해 더 자세한 정보를 위해선, 레퍼런스 매뉴얼 [SW95] 을 참고하세요.

B.7.2 AMD64

AMD64 는 x86 과 호환 가능하며, 문서화된 메모리 모델 [Adv07] 을 실제 구현이 가끔 제공하던 더 엄격한 순서 규칙을 강제하도록 업데이트 했습니다. 리눅스 `smp_mb()` 기능의 AMD64 구현은 `mfence` 이고, `smp_rmb()` 는 `lfence`, 그리고 `smp_wmb()` 는 `msfence` 입니다. 이론상으로 이것들은 완화될 수 있지만 그러기 위해선 SSE 와 3DNOW 인스트럭션들을 반드시 사용해야 합니다.

B.7.3 ARMv7-A/R

ARM 계열 CPU 들은 임베디드 분야, 특히 전력이 제한되는 휴대폰과 같은 분야에서는 극단적일 정도로 인기가 있습니다. ARM 의 멀티프로세서 구현은 아직 5년을 넘지 않았습니다. ARM 의 메모리 모델은 Power 의 것 (Section B.7.7 를 참고하세요) 과 유사합니다만, ARM 은 다른 메모리 배리어 인스트럭션 셋 [ARM10] 을 사용합니다:

1. DMB (데이터 메모리 배리어: data memory barrier) 는 특정 타입의 오퍼레이션들을 같은 타입의 뒤따르는 오퍼레이션들 이전에 완료된 것으로 나타나게 gkqslck. 오퍼레이션의 “타입”은 모든 오퍼레이션들이 될 수도 있고, 쓰기 만으로 제한될 수도 있습니다 (Alpha `wmb` 와 POWER `eieio` 인스트럭션들과 유사합니다). 또한, ARM 은 캐시 일관성이 세개의 영역 중 하나를 고를 수 있게 합니다: 단일 프로세서, 프로세서들 중 일부 (“inner”) 그리고 글로벌 (“outer”).
2. DSB (데이터 동기화 배리어: data synchronization barrier) 는 특정 타입의 오퍼레이션들이 실제로 뒤따르는 (모든 타입의) 오퍼레이션들이 수행되기 이전에 완료되게 합니다. 오퍼레이션들의 “타입”은 DMB 에서와 같습니다. DSB 인스트럭션은 ARM 아키텍쳐 초기 버전에서는 DWB (drain write buffer 또는 drain write barrier) 라 불렸습니다.
3. ISB (인스트럭션 동기화 배리어: instruction synchronization barrier) 는 CPU 파이프라인을 비워버려서 ISB 를 뒤따르는 모든 인스트럭션들은 ISB

가 완료된 이후에야 파이프라인으로 들어오게 됩니다. 예를 들어, (JIT 처럼) 스스로를 수정하는 프로그램을 만들고 있다면, 코드를 생성하고 나서 실행하기 이전에 ISB 를 실행해야 합니다.

이 중 어떤 인스트럭션도 리눅스의 `rmb()` 기능과 의미적으로 일치하지 않기 때문에 DMB 로 구현됩니다. DMB 와 DSB 인스트럭션은 배리어 전후 액세스 순서에 대한 재귀적 정의를 가지고 있기 때문에, POWER 의 누적성과 비슷한 효과를 갖습니다.

ARM 은 또한 컨트롤 의존성을 구현해서, 조건적 브랜치가 어떤 로드에 의존적이라면, 해당 조건적 브랜치 이후에 수행되는 모든 스토어 오퍼레이션은 해당 로드 이후로 순서세워집니다. 하지만, 조건적 브랜치를 뒤따르는 로드들은 해당 브랜치와 해당 로드 사이에 ISB 인스트럭션이 있지 않는 한 순서세워진다고 보장되지 않습니다. 다음의 예를 생각해 봅시다:

```
1 r1 = x;
2 if (r1 == 0)
3   nop();
4 y = 1;
5 r2 = z;
6 ISB();
7 r3 = z;
```

이 예제에서, 로드-스토어 컨트롤 의존성 순서세우기가 line 1 의 `x`로부터의 로드가 line 4 의 `y` 로의 스토어보다 먼저 수행되도록 만듭니다. 하지만, ARM 로드-로드 컨트롤 의존성은 신경쓰지 않으므로, line 1 의 로드는 line 5 의 로드보다 나중에 수행될 수 있습니다. 반면, line 2 의 조건적 브랜치와 line 6 의 ISB 인스트럭션의 조합은 line 7 의 로드가 line 1 의 로드보다 뒤에 수행될 것을 보장합니다. line 3 과 4 사이 어딘가에 ISB 인스트럭션을 추가하는 것은 line 1 과 5 사이의 순서를 지키도록 해줄 것임을 알아 두시기 바랍니다.

B.7.4 IA64

IA64 는 완화된 일관성 모델을 제공하므로, 명시적으로 메모리 배리어 인스트럭션을 사용하지 않으면 IA64 는 메모리 참조들을 재배치할 수 있습니다 [Int02b]. IA64 는 `mf` 라는 이름의 memory-fence 인스트럭션을 가지고 있지만, 로드, 스토어, 그리고 일부 어토믹 인스트럭션들에 덧붙일 수 있는 “half-memory fence” 변경자 역시 가지고 있습니다 [Int02a]. Figure B.12 에 그린 것처럼, `acq` 변경자는 뒤의 메모리 참조 인스트럭션들이 `acq` 앞으로 재배치 되는 것을 막습니다만, 앞의 메모리 참조인스트럭션들이 `acq` 뒤로 재배치 되는 것은 허용합니다. 유사하게, `rel` 변경자는 앞의 메모리 참조가 `rel` 뒤로 재배치 되는 것을 막습니다만, 뒤의 메모리 참조

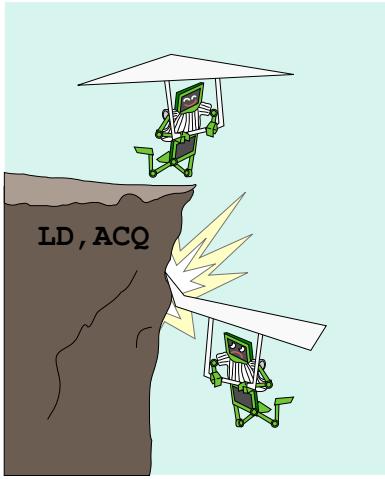


Figure B.12: Half Memory Barrier

인스트럭션들이 rel 앞으로 재배치 되는 것은 허용합니다.

이 half-memory fence 들은 오퍼레이션들을 크리티컬 섹션에 안전하게 집어넣을 수 있으므로 크리티컬 섹션에 유용합니다만, 이 오퍼레이션들이 빠져나오게 되면 매우 위험합니다. IA64 는 이 기능을 제공하는 CPU 들 중 하나⁷ 로써 리눅스의 락 획득과 해제에 연관된 메모리 오퍼레이션 순서 의미를 정의합니다. 크리티컬 섹션은 오퍼레이션들을 집어넣기 안전하지만 바깥으로 빠져나오면 치명적이기에 이 half-memory fence 들은 크리티컬 섹션에 유용합니다.

IA64 mf 인스트럭션이 리눅스 커널의 smp_rmb(), smp_mb(), 그리고 smp_wmb() 기능을 위해 사용됩니다. 아, 그리고 모순되는 루머들이 있긴 하지만, “mf” 는 정말로 “memory fence” 의 약자입니다.

마지막으로, IA64 는 “release” 오퍼레이션과 “mf” 인스트럭션들에 글로벌한 전체 순서를 제공합니다. 이는 한 코드 조각이 한 액세스가 일어난 것으로 본다면, 그 뒤의 코드 조각들도 역시 그 앞의 액세스가 일어난 것으로 보게 되는 타동성의 개념을 제공합니다. 이는 모든 코드 조각들이 올바르게 메모리 배리어들을 사용했다는 가정 하의 이야기입니다.

B.7.5 MIPS

MIPS 메모리 모델 [Ima15, Table 6.6] 은 ARM, IA64, 그리고 Power 와 흡사해서 완화된 순서를 기본으로 합니다만 의존성을 지켜줍니다. MPIS 는 다양한 종류의

메모리 배리어 인스트럭션들을 갖습니다만, ARM64 추가 인스트럭션들처럼 하드웨어 관점이 아니라 리눅스 커널과 C++11 표준 [Smi15] 에서 제공되는 사용예 관점으로 분류되어 있습니다:

SYNC 메모리 참조 외에도 여러 하드웨어 오퍼레이션들을 위한 전체 배리어

SYNC_WMB 리눅스 커널의 smp_wmb() 구현을 위해 사용될 수 있는 쓰기 메모리 배리어.

SYNC_MB 메모리 오퍼레이션들에만 적용되는 전체 메모리 배리어. 이 기능은 리눅스 커널의 smp_mb() 와 C++ atomic_thread_fence(memory_order_seq_cst) 의 구현에 사용될 수 있습니다.

SYNC_ACQUIRE 리눅스 커널의 smp_load_acquire() 와 C++ atomic_load_explicit(..., memory_order_acquire) 구현에 사용될 수 있는 Acquire 메모리 배리어.

SYNC_RELEASE 리눅스 커널의 smp_store_release() 와 C++ atomic_store_explicit(..., memory_order_release) 의 구현에 사용될 수 있는 Release 메모리 배리어.

SYNC_RMB 리눅스 커널의 smp_rmb() 구현에 사용될 수 있는 읽기 메모리 배리어.

MIPS 아키텍쳐에 대한 비공식적 토의들은 MIPS 가 ARM 과 Power 의 그것과 유사한 타동성과 누적성을 가짐을 알려줍니다. 하지만, 다른 MIPS 구현은 다른 메모리 순서 특성을 가질 수 있으므로, 당신이 사용하고 있는 특정 MIPS 구현을 위해서는 문서를 참고하는게 좋습니다.

B.7.6 PA-RISC

PA-RISC 아키텍쳐는 로드와 스토어의 완전한 재배치를 허용하지만, 실제 CPU 들은 순서를 지킵니다 [Kan96]. 이는 리눅스 커널의 메모리 순서 기능들이 어떤 코드도 만들어내지 않을 것임을 의미합니다만, 실제로는 메모리 배리어 전후로 코드를 재배치 할 수 있는 컴파일러 최적화를 막기 위해 gcc 의 memory 한정사를 사용합니다.

B.7.7 POWER / PowerPC

POWER 와 PowerPC® CPU 계열은 다양한 메모리 배리어 인스트럭션들을 가지고 있습니다 [IBM94, LHF05]:

⁷ ARMv8 은 최근 load-acquire 와 store-release 인스트럭션들을 추가했습니다.

1. sync 는 모든 앞의 오퍼레이션들이 뒤의 오퍼레이션이 하나라도 시작하기 전에 완료된 것처럼 보이게 합니다. 따라서 이 인스트럭션은 상당히 비용이 비쌉니다.
2. lwsync (light-weight sync) 는 로드 오퍼레이션들을 뒤의 로드와 스토어 오퍼레이션들에 대해 순서 맞추며, 스토어 오퍼레이션 역시 순서를 맞춰줍니다. 하지만, 스토어 오퍼레이션들을 뒤의 로드들에 대해 순서맞추지는 않습니다. 흥미롭게도, lwsync 인스트럭션은 zSeries, 그리고 우연히도, SPARC TSO 와 같은 순서 규칙을 강제합니다. lwsync 인스트럭션은 load-acquire 와 store-release 오퍼레이션을 구현하는데 사용될 수도 있습니다.
3. eieio (enforce in-order execution of I/O, in case you were wondering) 은 앞의 모든 캐싱 가능한 스토어들을 뒤의 스토어들 이전에 완료된 것으로 나타나게 합니다. 하지만, 캐싱 가능한 메모리로의 스토어들은 캐싱 불가능한 메모리로의 스토어들과는 별개로 순서지어지는데, 이는 eieio 가 MMIO 스토어가 스펀락 해제 뒤로 재배치 되는 현상도 막지 않는다는 뜻입니다.
4. isync 는 뒤의 인스트럭션이 하나라도 실행을 시작하기 이전에 앞의 인스트럭션들이 완료된 것으로 나타나게 만듭니다. 이는 앞의 인스트럭션들은 그들이 만들 수 있는 어떤 트랩도 이미 일어났거나 일어나지 않도록 보장될 만큼 충분히 진행되었어야 하며, 이 인스트럭션들의 어떤 사이드 이펙트들(예를 들어, 페이지 테이블의 변경) 도 뒤의 인스트럭션들에게 보여야 함을 의미합니다.

불행히도, 이 인스트럭션들 중 어떤 것도 모든 스토어들을 순서 맞추지만 sync 인스트럭션처럼 큰 오버헤드의 많은 동작을 요하지는 않는, 리눅스의 wbm() 기능과 일치하지 않습니다. 하지만 선택의 여지가 없습니다: wmb() 와 mb() 의 ppc64 버전은 좀 무겁지만 sync 인스트럭션을 사용하게 되어 있습니다. 하지만, 리눅스의 smp_wmb() 인스트럭션은 (어차피 드라이버가 MMIO 의 순서를 UP 커널에서도 SMP 커널에서도 알아서 순서를 맞춰야 하므로) MMIO 에 사용하지 않기 때문에, 가벼운 eieio 인스트럭션으로 구현되어 있습니다. 이 인스트럭션은 다섯개 모음의 약자라 좀 독특해 보일 수 있습니다. smp_mb() 인스트럭션 또한 sync 인스트럭션으로 정의되어 있습니다만, smp_rmb() 와 rmb() 는 가벼운 lwsync 인스트럭션으로 정의되어 있습니다.

Power 는 타동성을 얻기 위해 사용될 수 있는, “누적성”을 가지고 있습니다. 제대로 사용된다면, 앞의 코드 조각의 결과를 보는 코드는 이 앞의 코드 조각이 봤던

결과 역시 볼 수 있을 것입니다. 더 자세한 건 McKenney 와 Silvera의 작업물 [MS09]에서 볼 수 있습니다.

Power isync 인스트럭션이 ARM ISB 인스트럭션으로 대체된다는 점을 제외하고는, Power 는 ARM 에서 하는 것과 같은 방식으로 컨트롤 의존성을 지켜줍니다.

POWER 아키텍쳐의 많은 멤버들이 일관성을 지키지 않는 인스트럭션 캐시를 가지고 있어서, 인스트럭션이 위치하는 메모리로의 스토어는 인스트럭션 캐시에 제대로 반영되지 않을 수 있습니다. 감사하게도, 오늘날에는 적은 사람들만이 스스로를 수정하는 코드를 작성합니다만, JIT들과 컴파일러들은 항상 그 것을 합니다. 더욱이, 최근에 수행된 프로그램을 다시 컴파일 하는 행위는 CPU 의 관점에서는 스스로를 수정하는 코드처럼 보입니다. icbl (instruction cache block invalidate) 인스트럭션은 인스트럭션 캐시에서 특정 캐시 라인을 무효화 하게 하므로, 이런 상황에 사용될 수 있을 것입니다.

B.7.8 SPARC RMO, PSO, and TSO

SPACE 의 Solaris 는 리눅스가 “sparc” 32 비트 아키텍처용으로 빌드되었을 때 그러하듯, TSO (total-store order) 를 사용합니다. 하지만, 64 비트 리눅스 커널 (“sparc64” 아키텍처)은 SPARC 를 RMO (relaxed-memory order) 모드로 운용합니다 [SPA94]. RMO 로 수행되는 모든 프로그램은 PSO 또는 TSO 로도 수행될 수 있고, 비슷하게 PSO 로도 프로그램은 TSO 로도 수행될 수 있을 겁니다. 앞에서 이야기 했듯 동기화 기능들을 표준적으로 사용하는 프로그램은 메모리 배리어에 대해 걱정할 필요 없음에도 불구하고, 한 공유 메모리 병렬 프로그램을 (RMO 에서 PSO 로 가듯) 다른 방향으로 옮기는 것은 조심스로운 메모리 배리어 추가를 필요로 할 것입니다.

SPARC 는 세부적으로 순서를 조절할 수 있도록 하는 매우 유연한 메모리 배리어 인스트럭션 [SPA94]들을 가지고 있습니다:

- StoreStore: 앞의 스토어들을 뒤의 스토어들 앞으로 순서 맞춥니다. (리눅스 smp_wmb() 를 위해 이게 사용됩니다.)
- LoadStore: 앞의 로드들을 뒤의 스토어들 앞으로 순서 맞춥니다.
- StoreLoad: 앞의 스토어들을 뒤의 로드들 앞으로 순서 맞춥니다.
- LoadLoad: 앞의 로드들을 뒤의 로드들 앞으로 순서 맞춥니다. (리눅스 smp_rmb() 를 위해 이게 사용됩니다.)
- Sync: 뒤의 오퍼레이션을 하나라도 시작하기 전에 앞의 모든 오퍼레이션들을 완료시킵니다.

- **MemIssue:** 앞의 모든 메모리 오퍼레이션들을 뒤의 모든 메모리 오퍼레이션들보다 먼저 완료시키는데, memory-mapped I/O 같은 경우에 중요합니다.
- **Lookaside:** 같은 메모리 위치에 액세스하는 앞의 스토어들과 뒤의 로드들 사이에만 적용된다는 점을 제외하고는 MemIssue 와 동일합니다.

리눅스 `smp_mb()` 기능은 앞의 네개 인스트럭션들을 `membar #LoadLoad | #LoadStore | #StoreStore | #StoreLoad` 처럼 함께 사용해서 메모리 오퍼레이션들이 완전히 순서맞춰지도록 구현합니다.

그런데, 왜 `membar #MemIssue` 가 필요할까요? `membar #StoreLoad`는 뒤의 로드가 값을 스토어 버퍼에서 가져오도록 허용할 수 있는데, 이는 해당 스토어가 읽힐 값에 사이드 이펙트를 주는 MMIO 레지스터로의 쓰기였다면 문제가 될 수 있기 때문입니다. 반면, `membar #MemIssue`는 해당 로드들을 실행하기 전에 해당 스토어 버퍼들이 비워질 때까지 기다리도록 할 것이므로, 해당 로드는 실제로 그 값을 MMIO 레지스터에서 읽어오도록 보장할 것입니다. 드라이버들은 대신 `membar #Sync`를 사용할 수 있습니다만, 상대적으로 비싼 `membar #Sync`의 기능이 모두 필요하지 않은 경우라면 더 가벼운 `membar #MemIssue` 가 선호될 것입니다.

`membar #Lookaside` 는 `membar #MemIssue` 의 경량화 버전으로, MMIO 레지스터에 쓰기를 하는데 그 쓰기가 뒤에 그 레지스터로부터 읽힐 값에 영향을 주는 경우 융요합니다. 하지만, 해당 MMIO 레지스터에의 쓰기가 뒤의 다른 MMIO 레지스터로부터의 읽기기에 영향을 끼친다면 더 무거운 `membar #MemIssue` 가 사용되어야만 합니다.

현재의 정의는 일부 드라이버에서 버그에 취약할 수 있어 보임에도 왜 SPARC 가 `wmb()` 를 `membar #MemIssue` 로, 그리고 `smb_wmb()` 가 `membar #StoreStore` 로 정의하지 않았는지는 불명확합니다. 다만 리눅스가 돌아가는 모든 SPARC CPU 들이 아키텍처에서 허용하는 것보다 더 보수적인 메모리 순서 모델을 구현하고 있어서일 가능성이 많습니다.

SPARC 는 한 인스트럭션의 저장되고 나서 실행되기 전에 `flush` 인스트럭션의 사용을 필요로 합니다 [SPA94]. 이는 해당 위치의 이전 값을 SPARC 의 인스트럭션 캐시로부터 비워내기 위해 필요합니다. `flush` 는 주소를 받고, 인스트럭션 캐시로부터 해당 주소만을 비워냄을 주의하십시오. SMP 시스템에서는 모든 CPU 들의 캐시들이 비워지지만, 밖의 CPU 의 `flush` 가 언제 완료되는지를 알 수 있는 간편한 방법은, 구현 노트에 레퍼런스는 있습니다만, 없습니다.

B.7.9 x86

x86 CPU 들은 “프로세스 순서”를 제공해서 모든 CPU 들이 한 CPU 의 메모리에의 쓰기 순서에 동의하므로 해당 CPU [Int04b]에서 `smp_wmb()` 기능은 no-op 으로 동작합니다. 하지만, 컴파일러가 `smp_wmb()` 를 가로지르는 재배치의 결과를 만드는 최적화를 가하는 것을 막기 위해 컴파일러 지시어가 필요합니다.

반면, x86 CPU 들은 전통적으로 로드들에 대해선 순서에 대한 보장을 제공하지 않았으므로, `smp_mb()` 와 `smp_rmb()` 기능들은 `lock; addl` 로 확장됩니다. 이어토믹 인스트럭션은 로드와 스토어 둘 다에 대한 배리어처럼 동작합니다.

또한, 인텔은 x86 을 위한 메모리 모델 [Int07]을 출판했습니다. 이를 통해 인텔의 실제 CPU 들은 앞의 사양에서 이야기한 것보다 더 강한 순서를 제공하고 있음이 드러났으며, 따라서 이 모델은 이전의 사실상의 행동을 의무화 했습니다. 더 최근에, 인텔은 업데이트된 x86 을 위한 메모리 모델 [Int11, Section8.2] 을 발간했는데, 여기선 스토어들에 대한 전체 전역 순서를 의무화 했습니다만 각각의 CPU 들은 여전히 각자의 스토어들은 전체 전역적 순서가 가리키는 것보다 빨리 일어난 것으로 볼 수 있게 허용하고 있습니다. 스토어 버퍼와 관련된, 중요한 하드웨어 최적화를 위해 전체 스토어 순서에 대한 이 예외가 필요했습니다. 또한, 메모리 순서 규칙은 인과성을 준수해서, CPU 0 가 CPU 1 의 스토어를 봤다면, CPU 0 는 CPU 1 이 그 스토어 이전에 봤던 모든 스토어들을 볼 수 있도록 보장됩니다. 소프트웨어는 이런 하드웨어 최적화를 무효화 하기 위해 어토믹 오퍼레이션을 사용할 수 있는데, 이것이 어토믹 오퍼레이션이 그렇지 않은 것들에 비해 큰 비용을 필요로 하는 경향을 띠는 이유 중 한가지입니다. 이 전체 스토어 순서 (total store order) 는 옛날 프로세서들에서는 보장되지 않습니다.

또한, 주어진 메모리 위치에서 수행되는 어토믹 인스트럭션들은 모두 같은 크기여야 함 [Int11, Section 8.1.2.2] 을 주의해야만 합니다. 예를 들어, 한 CPU 가 어토믹 1 바이트 증가 오퍼레이션을 수행하는 동안 다른 CPU 에서는 같은 위치에 4 바이트 어토믹 증가 오퍼레이션을 수행한다면, 결과는 알 수 없습니다.

하지만, 일부 SSE 인스트럭션들은 완화된 순서 규칙을 가짐을 주의하십시오 (`clflush` 와 임시적이지 않은 `move` 인스트럭션들 [Int04a]). SSE 를 갖는 CPU 들은 `smp_mb()` 를 위해 `mfence` 를, `smp_rmb()` 를 위해 `lfence` 를, 그리고 `smp_wmb()` 를 위해 `sfence` 를 사용할 수 있습니다.

x86 CPU 의 일부 버전들은 순서 외 스토어를 가중하게 하는 `mode bit` 를 가지고 있고, 이런 CPU 들에서는 `smp_wmb()` 가 `lock; addl` 로 정의되어야만 합니다.

최신의 x86 구현은 스스로 수정하는 코드를 특별한 인스트럭션 없이 수용할 수 있지만, 과거와 잠재적인

미래 x86 구현과 호환성을 갖기 위해, CPU는 코드의 수정과 실행 사이에 jump 인스트럭션이나 직렬화 인스트럭션 (예: cpuid)을 실행해야만 합니다 [Int11, Section 8.1.3].

B.7.10 zSeries

zSeries 머신들은 기존에 360, 370, 그리고 390 [Int04c]로 알려진 IBM™ 메인프레임들을 만드는데 사용됩니다. zSeries에 병렬성은 꽤 늦게 도입되었습니다만, 이 메인프레임들이 1960년대 중반에 처음 출시된 걸 생각해보면, 그렇게 늦은 것도 아닙니다. bcr 15,0 인스트럭션은 리눅스 smp_mb(), smp_rmb(), 그리고 smp_wmb() 기능으로 사용됩니다. 이들은 또한 Table B.5에 나온 것과 같이 비교적 강력한 메모리 순서 규칙을 가져서, smp_wmb()가 nop이 되는 것도 가능해야 합니다 (그리고 당신이 이 글을 읽고 있는 시점에서는, 이미 리눅스 커널은 그렇게 되었을 수 있습니다). 앞에서 이야기한 표는 이 상황을 좀 과소평가하도록 이야기하는데, 그렇지 않다면 zSeries 메모리 모델은 순차적 일관성 (sequential consistent)을 지키는 것으로 이야기되는데, 이는 모든 CPU들이 다른 CPU들에서의 연관되지 않은 스토어들에 대해서도 순서를 동의한다는 뜻이기 때문입니다.

대부분의 CPU들처럼, zSeries 아키텍쳐는 인스트럭션 스트림에 대해 캐시 일관성을 제공하지 않고, 따라서 스스로를 수정하는 코드는 직렬화 인스트럭션을 인스트럭션 수정과 그들의 실행 사이에 실행해야만 합니다. 그렇다면 하나, 많은 실제 zSeries 머신들은 스스로를 수정하는 코드를 직렬화 인스트럭션 없이도 잘 실행합니다. zSeries 인스트럭션 집합은 compare-and-swap, 일부 타입의 브랜치들 (예를 들어, 앞에 이야기한 bcr 15,0 인스트럭션), 그리고 test-and-set, 등등의 다양한 직렬화 인스트럭션들을 제공합니다.

B.8 Are Memory Barriers Forever?

최근의 시스템들 중에는 일반적으로 순서 외 (out-of-order) 실행에, 그리고 특히 메모리 참조 재배치에 상당히 덜 과감한 경우가 여럿 있었습니다. 이런 경향은 메모리 배리어가 과거의 것이 되게 하는 지점까지 이어질까요?

긍정적으로 생각하는 측에서는 제안된, 매우 많은 멀티 쓰레드를 지원해서 각 쓰레드는 메모리가 준비되기 까지 기다리게 되고, 그 사이 수십, 수백, 심지어 수천 개의 다른 쓰레드들이 진전을 이루게 되는 하드웨어 아키텍처를 인용할 수 있을 겁니다. 그런 아키텍처에서 쓰레드는 다음 인스트럭션을 수행하기 전, 나머지 오퍼레이션들이 모두 완료될 때까지 그저 기다릴 것이기

때문에 메모리 배리어들이 필요 없을 겁니다. 수천개의 다른 쓰레드가 존재하고 있을 것이기 때문에, 해당 CPU는 완전히 활용되고 있을 것이고, 따라서 CPU는 시간을 허비하지 않습니다.

반대하는 측에서는 극단적으로 적은, 1000개 쓰레드 까지 확장이 가능한 어플리케이션의 수와 일부 어플리케이션에서는 수십 마이크로세컨드에 불과할 정도로 가혹한 실시간 요구사항을 인용할 수 있을 겁니다. 앞서 이야기한 거대한 멀티 쓰레드 시나리오대로라면 실시간 반응성 요구사항은 달성되기 어려우며, 극단적으로 적은 싱글 쓰레드 처리량을 생각하면 더더욱 어려울 것입니다.

또 다른 찬성 의견 측에서는 갈수록 늘어나는, 여전히 순서와 실행의 성능 이점을 거의 다 제공하면서도 완전히 순차적 일관성 (sequential consistency)을 제공하는 것 같은 환상을 제공하는 CPU를 구현하는 정교한, 대기시간을 숨기는 하드웨어 구현 기술들을 인용할 수도 있을 것입니다. 반대 측에서는 갈수록 가혹해져 가는, 배터리 기반 디바이스와 환경적 요구사항 둘다에서 발생하는 전력 효율성 요구사항을 인용할 수 있을 것입니다.

누가 맞을까요? 딱히 단서가 없으니, 두 시나리오 모두에 대비해서 살아가야 합니다.

B.9 Advice to Hardware Designers

하드웨어 설계자들이 소프트웨어쪽 사람들의 삶을 고통스럽게 하기 위해 할 수 있는 여러가지 일들이 있습니다. 여기 우리가 과거에 마주했던 그런 것들 중 몇개를 미래에는 그런 문제가 없어지도록 도움을 주기 위한 바램으로 나열해 봅니다:

1. 캐시 일관성을 무시하는 I/O 디바이스들.

이 유혹적인 잘못된 기능은 메모리로부터의 DMA가 아웃풋 버퍼에 만들어진 최근의 변경을 놓치거나, 인풋 버퍼가 CPU 캐시에 의해 DMA 완료 직후에야 덮어써지게 만들 수 있습니다. 그런 잘못된 동작에서도 시스템이 정상적으로 동작하게 만들려면, 조심스럽게 모든 DMA 버퍼 위치의 CPU 캐시를 그 버퍼가 I/O 디바이스에 넘어가기 전에 비워야만 합니다. 비슷하게, 모든 DMA 버퍼 위치의 CPU 캐시를 해당 버퍼로의 DMA가 완료된 직후 비워야 합니다. 그리고 나서도, 잘못 만들어진 인풋 버퍼로의 읽기 작업조차도 해당 데이터 입력을 망가뜨릴 수 있기 때문에, 포인터 버그를 매우 조심스럽게 방지해야 합니다!

2. 캐시 일관성 데이터를 전달하는데 실패할 수 있는 외부 버스.

이건 앞의 문제보다도 더 심각한 문제로, 여러 디바이스 그룹들—그리고 심지어 메모리 자신도—을 캐시 일관성과 관련해 실패하게 만듭니다. 이야기하기 고통스러운 말이지만, 임베디드 시스템들이 멀티코어 아키텍처로 옮겨감에 따라 이런 문제들이 이 여렷 나타나게 될 것입니다. 부디 이 문제들이 2015년에는 해결되길 바랍니다.

3. 캐시 일관성을 무시하는 디바이스 인터럽트들.

이건 문제 없어 보일 수 있습니다 — 일단, 인터럽트들은 메모리 참조가 아니지 않습니까? 하지만 분할된 캐시를 갖는 CPU에서 인풋 버퍼의 마지막 캐시라인을 매우 바쁜 뱅크 하나가 가지고 있는 상황을 생각해 봅시다. 만약 관련된 I/O 완료 인터럽트가 이 CPU에 도착하면, 이 CPU의 해당 버퍼의 마지막 캐시라인으로의 메모리 참조는 예전 데이터를 반환할 수 있고, 이로 인해 데이터가 망가질 수 있으며, 이 망가진 데이터는 뒤에 만들어질 크래시 덤프에는 보이지도 않을 겁니다. 시스템이 문제가 된 입력 버퍼를 덤프 뜨려 해는 시점에서는 DMA는 완료됐을 것이기 때문입니다.

4. 캐시 일관성을 무시하는 프로세서간 인터럽트 (IPS).

IPI가 그 목표지에 연관된 메세지 버퍼의 캐시 라인들이 모두 메모리에 반영되기 전에 도달한다면 문제가 될 수 있습니다.

5. 캐시 일관성을 추월하는 컨텍스트 스위치.

메모리 액세스들이 너무 비순차적으로 완료될 수 있다면, 컨텍스트 스위치가 문제가 될 수 있습니다. 태스크가 원래 있던 CPU에 보이는 모든 메모리 액세스들이 넘어갈 CPU에 보여지게 되기 전에 이동하게 된다면, 해당 태스크에게는 관련된 변수들이 이전의 값으로 보이게 되어, 대부분의 알고리즘을 혼란스럽게 만들 것입니다.

6. 지나치게 친절한 시뮬레이터들과 에뮬레이터들.

메모리 순서 재배치를 강제하는 시뮬레이터나 에뮬레이터를 만들기는 어려운 관계로, 이런 환경에서 잘 동작하던 소프트웨어는 실제 하드웨어에서 수행되면 불쾌한 놀림을 받을 수 있습니다. 불행히도, 하드웨어는 시뮬레이터나 에뮬레이터보다 일탈적이라는게 여전히 진실입니다만, 이런 상황이 바꿔질 바랍니다.

다시 말하지만, 하드웨어 설계자들이 이런 것들을 좀 바꿔주길 바랍니다!

Appendix C

Read-Copy Update in Linux

이 챕터에서는 2008년 중반 이후부터의 리눅스 커널에서의 RCU의 역사를 설명합니다. 그 전의 RCU의 역사를 대해서는 다른 곳 [McK04, MW08]을 찾아 보시기 바랍니다. Section C.1은 리눅스에서의 RCU 사용의 증가를 개괄적으로 알아보고 Section C.2는 최근의 RCU의 진화에 대한 자세한 설명을 제공합니다.

C.1 RCU Usage Within Linux

리눅스 커널의 RCU 사용은 Figure C.1 [McK06a]를 통해 볼 수 있듯이 시간에 따라 증가되었습니다. RCU는 코드 내에 존재하는 다른 동기화 메커니즘 (예를 들면, 네트워킹 프로토콜 스택의 `brlock` [MM00, Tor03a, Tor03b])들을 대체했으며, 새로운 기능을 구현하는 코드 (예를 들면, SELinux [Mor04]에서의 audit 시스템)와 함께 나타나기도 했습니다. 하지만, RCU는 Figure C.2에 보여지듯이 락킹에 비해서 틈새에만 사용되는 기술로 남아있습니다. 락킹이 커널 해커의 동시성 도구상자의 망치라면, RCU는 아마도 스크류드라이버입니다. 만약 그렇다면, Figure C.3를 통해 볼 수 있듯 매우 급격히 성장하는 스크류드라이버입니다.

C.2 RCU Evolution

이 섹션은 2008년 중반 이후 계속 진행중인 RCU에 대한 경험을 제공합니다.

C.2.1 2.6.27 Linux Kernel

This release added the `call_rcu_sched()`, `rcu_barrier_sched()`, and `rcu_barrier_bh()` RCU API members.

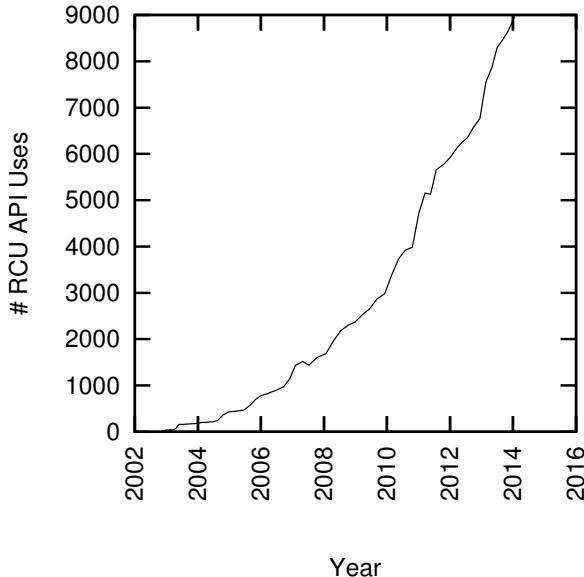


Figure C.1: RCU API Usage in the Linux Kernel

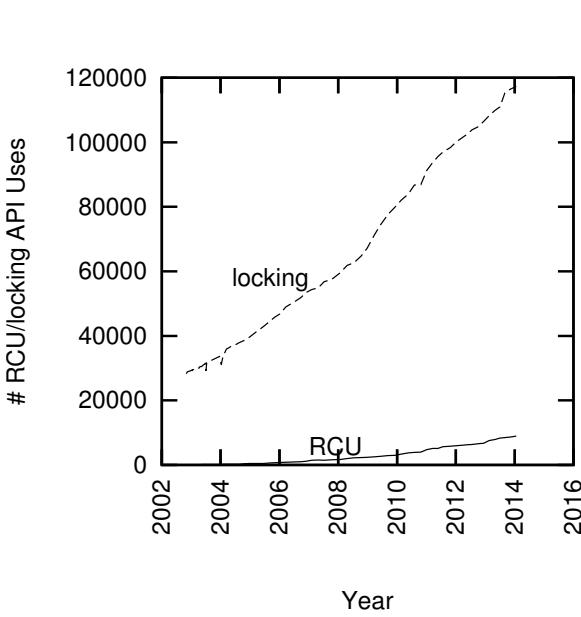


Figure C.2: RCU API Usage in the Linux Kernel vs. Locking

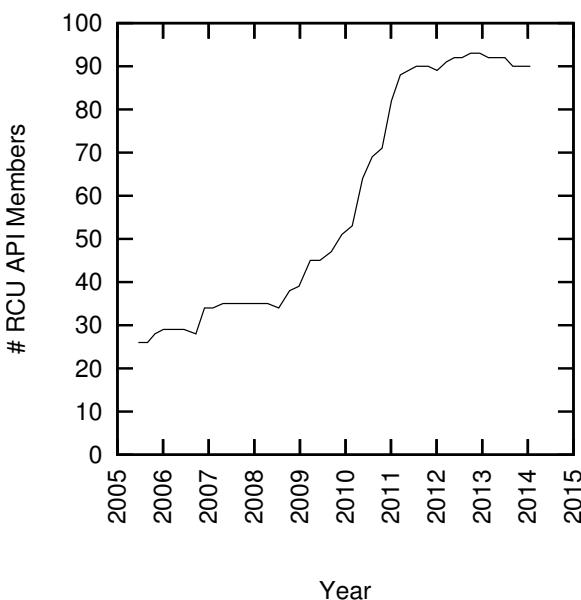


Figure C.3: RCU API Growth Over Time

C.2.2 2.6.28 Linux Kernel

RCU API의 크기를 실제로 줄이는데에 관련된 변경 사항으로 `list_for_each_rcu()` 기능의 제거가 있었습니다. 이 기능은 `list_head` 구조체를 구성하는 포인터쌍을 따라가기보다는 (헷갈릴 수 있지만, 리스트 헤더뿐 아니라 리스트 원소로 동작하는) 구조체들을 따라가는 장점을 가진 `list_for_each_entry_rcu()`로 대체되었습니다. 이 변경은 2.6.28 리눅스 커널에 받아들여졌습니다.

안타깝게도, 2.6.28 리눅스 커널은 또한 `rcu_read_lock_sched()` 와 `rcu_read_unlock_sched()` RCU API 멤버들을 추가했습니다. 이 API들은 가독성을 올리기 위해 추가되었습니다. 과거에는, `synchronize_sched()`에 연관된 RCU read-side 크리티컬 섹션들을 표시하기 위해 인터럽트나 `preemption` 을 불능화 시키는데 사용되는 기능들을 사용했습니다. 하지만, 이는 개발자들이 `preemption`이나 인터럽트를 불능화 시킬 필요가 없어졌지만 RCU 보호를 위한 필요를 알지 못할 때에 버그가 생기도록 이끌었습니다. `rcu_read_lock_sched()`의 사용은 그런 버그를 예방하는데 도움이 될 것입니다.

C.2.3 2.6.29 Linux Kernel

새로운 더욱 확장성 있는 구현, “Tree RCU” 가 `bitmap` 을 `combining tree`로 대체하면서 2.6.29 리눅스 커널에 받아들여졌습니다. 이 구현은 근래의 멀티프로세서들의 계속해서 증가하는 코어의 갯수에서 영감을 받아서 수백개의 CPU 환경을 위해 설계되었습니다. 현재의 구조상의 한계는 262,144 개 CPU 인데, 이는 개발자들이 (순진한 것일 수도 있지만) 한동안은 충분할 것이라 믿는 숫자입니다. 이 구현은 또한 preemptible RCU의 개선된 `dynamic-tick` 인터페이스 인터페이스를 받아들였습니다.

Mathieu Desnoyers 는 리눅스 커널의 tracing code 가 RCU를 사용하도록 하는데 필요한 `rcu_read_lock_sched_notrace()` 와 `rcu_read_unlock_sched_notrace()` 를 추가했습니다. 이 API 들 없이는, RCU read-side 크리티컬 섹션을 주적하려는 시도는 무한 재귀 문제를 일으킵니다.

Eric Dumazet 은 리스트 포인터에 단일 비트 마커들이 저장될 수 있도록 하는 새로운 종류의 RCU로 보호되는 리스트를 추가했습니다. 이 종류의 리스트는 여러개의 락이 없는 알고리즘들을 가능하게 하는데, Maged Michael 에 의해 보고된 것 [Mic04] 등을 포함합니다. Eric 의 작업은 `hlist_nulls_add_head_rcu()`, `hlist_nulls_del_rcu()`, `hlist_nulls_del_init_rcu()`, 그리고 `hlist_nulls_for_each_entry_rcu()` 를 추가합니다. 이는 또한

`hlist_nulls_node`라는 이름의 새로운 구조체를 추가합니다.

또한, 이전 엄격히 말해서 리눅스 커널의 부분은 아니긴 하지만, 동시에 Mathieu Desnoyers는 그의 유저스페이스 RCU 구현 [Des09]을 발표했습니다. 이는 real-time user-level RCU 구현을 향한 중요한 첫걸음입니다.

C.2.4 2.6.31 Linux Kernel

Jiri Pirko는 RCU-subscription 기능인 `rcu_dereference()`를 더 높은 수준의 리스트 접근 기능에 포함시킨 `list_entry_rcu()`와 `list_first_entry_rcu()` 기능을 추가했는데, 이 기능들은 일련의 버그들을 제거해줄 수 있을 겁니다.

또한, “Tree RCU” 구현은 “experimental” 상태로부터 업그레이드 되었습니다.

C.2.5 2.6.32 Linux Kernel

리눅스 커널의 이 버전에서의 가장 큰 변경은 “Classic RCU” 구현의 제거일 것입니다. 이 구현은 “Tree RCU” 구현으로 대체되었습니다.

이 버전은 그 외에도 여러 변경을 가졌는데, 다음과 같은 것들이 포함됩니다:

1. `synchronize_rcu_expedited()`, `synchronize_sched_expedited()`, 그리고 `synchronize_rcu_bh_expedited()` RCU API 멤버들의 추가. 이 기능들은 grace period를 신속히 처리하기 위해 측정을 한다는 점을 제외하고는 non-expedited 버전의 것들과 동일합니다.
2. “Tree RCU” 구현에 preemptible-RCU 기능들을 추가함으로써 리눅스로 돌아가는 커다란 멀티프로세서 머신에서의 real-time response에 대한 문제를 해결.
3. 이 새로운 “Tree Preemptible RCU” 구현은 기존의 preemptible RCU 구현을 구형으로 만들어서 리눅스 커널에서 사라지게 했습니다.

C.2.6 2.6.33 Linux Kernel

이 릴리즈에서의 가장 극적인 추가사항은 Tree RCU에서의 day-one 버그 [McK09a]였을 겁니다. 다른 변경 사항들은 다음과 같습니다:

1. “RCU: The Bloatwatch Edition” [McK09b]라고도 알려진 “Tiny RCU”.

2. `synchronize_srcu_expedited()`의 형태의 expedited SRCU.
3. 앞서 이야기된 버그에 의해 축발된 Tree RCU 동기화의 정리.
4. Tree Preemptible RCU의 expedited 구현의 추가 (이전의 릴리즈에서, “expedited” 지원은 단순히 `synchronize_rcu()`로 매핑되었는데, 이는 의미적으로는 옳은 동작이지만 성능 관점에서는 별로 도움이 되지 않았습니다.)
5. 스트레스 테스트를 향상시킨 네번째 수준의 Tree RCU 추가. 이로 인해, 누군가가 16,777,216개의 CPU를 가진 시스템에서 리눅스를 돌리고자 한다면, RCU를 그런 환경에도 준비가 되었습니다! 대략 1600만개의 per-CPU 데이터 원소들을 스캐닝하는 것에 대한 응답시간 의미 ...

C.2.7 2.6.34 Linux Kernel

이 릴리즈에서 가장 눈에 띄는 추가사항은 `rcu_dereference()`가 올바른 락킹 조건을 체크하도록 하는 `CONFIG_PROVE_RCU` [McK10] 였습니다. 다른 변경들은 다음과 같습니다:

1. 기존 grace period를 강제하고 새로운 grace period를 시작하는 사이의 Tree RCU의 상호작용의 간소화.
2. Free-running 카운터들이 부호 없게 되도록 카운터들을 재작업. (C-컴파일러 해커들이 부호 있는 정수를 오버플로우 시킴으로써 코드를 깨먹을 수 있는 최적화를 이야기하면서 얼굴에 띠웠던 즐거운 표정을 여러분은 상상하지도 못할겁니다!!!)
3. RCU read-side 크리티컬 섹션 안에서 지나치게 많은 시간 동안 preemption 당한 모든 태스크들을 출력하기 위한 Tree Preemptible RCU의 stall 파악 기능 업데이트.
4. Tree RCU의 CPU-stall-detection 코드의 다른 버그 수정과 개선. 이 코드는, 예를 들어, 인터럽트가 불능화된 채 무한루프를 돌거나 하는, 락업된 CPU들을 체크합니다.
5. 배터리로 동작하는 멀티프로세서 시스템에서 마지막 CPU가 idle 상태로 빠질 때 grace period를 가속화 시키기 위한 일부 코드를 프로토타입. 시스템이 모든 CPU들이 꺼질 수 있는 상태로 넘어가는데에 추가적인 몇 밀리세컨드를 RCU가 가져가는데에 대해서 상당히 불쾌해 했던 사람들이 존재합니다!

C.2.8 2.6.35 Linux Kernel

이 릴리즈는 여러개의 버그 수정과 코드 정리를 추가합니다. 여기서의 주요 변경 사항은 Mathieu Desnoyers 의 RCU 콜백의 오용을 위한 검사로, 예를 들면 하나의 grace period 안에서 `rcu_head` 구조체를 `call_rcu()`에 두번 보내거나 하는 경우입니다.

C.2.9 2.6.36 Linux Kernel

Mathieu Desnoyers 의 디버그 목적의 작업물의 핵심은 2.6.36 에 나타났는데, 일부 코드 정리 작업은 다른 maintainer 트리들에 나오는 커밋들과의 종속성 때문에 2.6.37 로 미뤄졌습니다. Arnd Bergmann 의 sparse RCU 검사의 한 핵심적 부분이 2.6.36 에 나타났는데, 나머지는 역시 다른 maintainer 트리에 나오는 커밋과의 종속성 때문에 2.6.37 로 미뤄졌습니다. 마지막으로, Eric Dumazet 으로부터의 패치는 `rcu_dereference_bh()` 의 에러 검사에서의 에러를 고쳤습니다.

C.2.10 2.6.37 Linux Kernel

Mathieu Desnoyers 의 디버그 목적 작업의 마지막 정리 작업이 2.6.37 에 Arnd Bergmann 의 sparse 기반 검사 작업과 함께 나타났습니다. Lai Jiangshan 은 `rcutorture`에 `preemption`에 의한 문제 상황을 추가했고 Tree RCU 의 per-CPU 데이터 처리를 일부 간단화 시켰습니다. Tetsuo Handa 는 RCU lockdep 으로 나온 문제 하나를 고쳤고, Christian Dietrich 는 불필요하게 반복된 `#ifdef` 를 제거했으며, Dongdong Deng 은 일부 Tree RCU 제어 데이터로의 락 없는 액세스를 가능하게 하도록 `ACCESS_ONCE()` 를 추가했습니다.

Paul 의 preemptible Tiny RCU 의 구현 또한 RCU CPU stall-warning 코드, docbook 수정, 중복 코드의 합체, Tree RCU 속도향상, RCU callback flow 에의 queuing 모델 지원에 대한 tracing 추가, 그리고 몇 가지 기타 수정과 정리 작업과 함께 2.6.37 에서 나타났습니다.

C.2.11 2.6.38 Linux Kernel

Lai Jiangshan 은 `synchronize_sched_expedited()` 를 `kernel/sched.c` 에서 그것이 유래한 `kernel/rcutree.c` 와 `kernel/rcutiny.c` 로 옮겼습니다. 그는 또한 `orphan_cbs_list` 를 제거함으로써 CPU-hotplug 오퍼레이션 동안의 RCU-callback 처리를 단순화 해서, offline 이 되는 CPU 에 의해 고아가 되는 RCU 콜백들이 곧바로 해당 offline 화 시키는 동작을 함께 돋는 CPU 들에 의해 곧바로 처리될 수 있도록 했습니다. Tejun Heo 는 `synchronize_sched_expedited()` 의

batching 기능을 개선했는데, 이는 결국 많은 동시적 `synchronize_sched_expedited` 오퍼레이션들을 갖는 워크로드들의 성능과 확장성을 개선했습니다. Frédéric Weisbecker 는 RCU 를 idle 상태일 때 더욱 전력에 효율적이도록 만들어주는 RCU 핵심 코드상의 일부 변경을 제공했습니다. Mariusz Kozlowski 는 `_list_for_each_rcu()` 상의 문법상 에러를 고쳐졌고, 곧 사라졌습니다. (하지만 이 고쳐진 버전은 필요한 경우를 위해 git tree 상에 존재합니다.) Nick Piggin 은 RCU 로 보호되는 bit-locked doubly-linked list 들을 위해 `hlist_bl_set_first_rcu()`, `hlist_bl_first_rcu()`, `hlist_bl_del_init_rcu()`, `hlist_bl_del_rcu()`, `hlist_bl_add_head_rcu()`, 그리고 `hlist_bl_for_each_entry_rcu()` 기능을 추가했습니다. Christoph Lameter 는 변수가 곧바로 접근되는 경우에 사용되기 위한 `_get_cpu_var()` 의 최적화된 변종인 `_this_cpu_read()` 를 구현했습니다.

또한, TINY_RCU 는 priority boosting 을 얻었고, `synchronize_srcu_expedited()` 의 race condition 이 고쳐졌고, `synchronized_srcu_expedited()` 가 동시적인 읽기 쓰레드들이 존재할 때에 expedited 본성을 얻을 수 있도록 수정되었으며, grace-period 시작/종료 검사가 개선되었고, TREE_RCU 의 leaf-level fanout 이 lock-contention 문제들을 고치기 위해 16 으로 제한되었습니다. 이 마지막 변경은 TREE_RCU 와 TREE_PREEMPT_RCU 가 지원할 수 있는 CPU 들의 최대 갯수를 4,194,304 로 줄였는데, 이는 (다시 말하지만, 어쩌면 순진하게도) 충분할 것으로 여겨집니다.

C.2.12 2.6.39 Linux Kernel

Lai Jiangshan 은 task 가 RCU read-side 크리티컬 섹션 내에서 종료되는 경우에 디버깅 상태를 보존하기 위해 `exit_rcu()` 가 `rcu_read_unlock()` 대신 `_rcu_read_unlock()` 을 호출하도록 만들었으며, Jesper Juhl 은 `rcutorture` 내의 중복된 `sched.h` 포함을 제거했고, Amerigo Wang 은 `rcu_fixup_free()` 로부터 의미없는 코드들을 일부 제거했습니다.

또한, MCE 서브시스템에서의 사용을 위해 새로운 `rcu_access_index()` 가 만들어졌습니다.

C.2.13 3.0 Linux Kernel

많은 사람들이 2.6.40 릴리즈가 될 것이라 예상했던 것은 3.0 릴리즈가 되었습니다. 여기서 가장 중요한 RCU 기능은 Tree RCU 를 위한 priority boosting 의 추가였습니다: 계획했던 것보다 여려면에서 중요해서 [McK11a], 3.0-rc7 뒤의 RCU 수정이 되었습니다.

Shaohua Li, Peter Zijlstra, Steven Rostedt 의 RCU, 스케줄러, 그리고 thread 로 돌아가는 인터럽트들 사이의 충돌의 fallout 처리에 대한 많은 도움에 감사를 드리는 바입니다. 또한, RCU CPU stall warning 들은, 여전히 커널 부트 패러미터나 sysfs 로 조정될 수 있는 `rcu_cpu_stall_suppress` 모듈 패러미터로 불능화 될 수 있긴 하지만, 이제 무조건적으로 Tree RCU 안으로 컴파일 됩니다.

Mathieu Desnoyers 는 non-preemptible RCU 구현에 들어오게 되는 `DEBUG_OBJECTS_RCU_HEAD` 체크를 활성화 시켰습니다. Lai Jiangshan 은 fire-and-forget `kfree_rcu()` 를 만들었고 (그리고 이를 커널을 통해 적용했습니다), 또한 `exit_rcu()` 가 task 가 RCU read-side 크리티컬 섹션 내에서 끝나는 경우에 디버깅 상태를 유지할 수 있도록 하기 위해 `rcu_read_unlock()` 대신 `__rcu_read_unlock()` 을 호출하도록 만들었습니다. Eric Dumazet 는 더 나아가서 `TINY_RCU` 를 감소시켰고 Gleb Natapov 는 가상화가 guest OS 와의 문맥 전환 시에 일어나는 quiescent state 들 사이에 RCU 가 신경을 쓸 수 있도록 RCU hook 들을 추가했습니다. Peter Zijlstra 는 RCU kthread 블록킹과 wakeup 을 streamline 시켰습니다.

C.2.14 3.1 Linux Kernel

3.1 버전은 Arun Sharma, Jiri Kosina, Michal Hocko, Peter Zijlstra, 그리고 Jan H. Schönherr 로부터의 마이너한 수정들과 코드 정리들을 포함한, RCU 에 있어서는 조용한 시간이었습니다.

C.2.15 3.2 Linux Kernel

3.2 리눅스 커널은 RCU 의 코드의 꼭대기부터 바닥까지의 검사 첫번째 단계에서 발견된 문제들에 대한 여러 수정사항들을 담고 있습니다. 이 검사의 결과물 가운데 하나는 코드의 irq 불능화된 구간이 preemptible RCU read-side 크리티컬 섹션의 시작이 아니라 끝과 겹치게 되면 데드락이 일어날 수 있다는 것이었습니다. 따라서, 부분적으로 irq 불능화된 코드 구간과 겹칠 수 있는 RCU read-side 크리티컬 섹션들을 만들지 마세요: 대신에, irq 불능화된 코드 섹션을 주어진 RCU read-side 크리티컬 섹션 안에 완전히 집어넣거나 그 반대로 하거나 하세요.

이 릴리즈는 RCU 이벤트 추적 기능을 처음으로 보였습니다. Eric Dumazet 는 RCU 의 kthread 들이 NUMA 시스템에 최적화된 방식으로 위치된 메모리를 가지게 됨을 보장하는 `kthread_create_on_node()` 기능을 적용했습니다. 그는 또한 `rcu_assign_pointer()` 가 무조건적으로 메모리 배리어를 삽입하도록 했는데, 특정 환경에서는 이 메모리 배리

어가 생략되게 했던 기존의 컴파일러의 마술이 그보다 최신의 버전의 컴파일러에서는 통하지 않았기 때문입니다. 따라서, RCU 로 보호되는 포인터에 NULL 을 할당할 때에는, `rcu_assign_pointer()` 보다는 `RCU_INIT_POINTER()` 를 사용하세요.

Shaohua Li 는 RCU 의 per-CPU kthread 들의 불필요한 self-wakeup 을 제거했고, Andi Kleen 은 일부 충돌되는 변수 선언들을 정리했습니다. Mike Galbraith 는 RCU 가 `RCU_BOOST_PRIO` 커널 패러미터를 무시하게 하는 버그를 잡았고, 마지막으로, 계속해서 확장되는 RCU 의 기능들을 잡을 수 있도록 `rcutorture` 가 진척을 만들었습니다.

C.2.16 3.3 Linux Kernel

3.3 리눅스 커널은 RCU 의 idle CPU 가 아니라면 `scheduling-clock ticks` 를 위한 필요를 줄여주는에너지 효율성 개선, uprobes 에 의해 필요시되는 새로운 `srcu_read_lock_raw()` 기능, 여전히 진행중인 RCU 전체 검사로 찾아내어진 문제들에 대한 수정사항들, 그리고 스크립트로 짜여진, 타입이나 userspace 레이아웃의 존재에 의존적이지 않은 KVM 기반의 RCU 테스트를 가능하게 하는 `rcutorture` 개선사항이 포함되었습니다.

또한 Thomas Gleixner 로부터의 `-rt` RCU 패치들 포함했고, 또한 dyntick-idle 모드부터 usermode 수행까지의 응용을 지원하는 Frédéric Weisbecker 로부터의 RCU 기반 시설 관련 여러 패치들을 포함했습니다.

비록 일부 초기 작업들은 RCU-preempt 의 `__rcu_read_lock()` 과 `__rcu_read_unlock()` 가 inline 될 수 있도록 했지만, 그보다 훨씬 많은 작업이 다양한 `include` 파일 문제들을 푸는데 필요했습니다. 마지막으로, Rusty Russell 로부터의 잡다한 수정들이 있었습니다.

`rcu_barrier_expedited()` 에 대한 초기 요청이 있었습니다만, 그 요청을 한 사람은 이 문제를 풀기 위한 다른 방법을 찾았기에, 이는 상대적으로 낮은 우선순위를 가졌습니다.

C.2.17 3.4 Linux Kernel

3.4 커널은 전력 효율성을 위해 급격한 idle 진입/종료 워크로드에 대한 단점을 줄이는 작업을 포함했습니다. 여기서 관리된 트레이드오프는 긴 idle 시간에 대비해서 idle 진입에 대한 작업의 증가이고, 따라서 이 릴리즈에서의 변경사항들은 추가적인 노력이 의미없을 때를 인식하는 데 대해 더 나은 일을 하는데, 예를 들어 CPU 가 워크로드 때문에 idle 을 급격하게 진입했다가 빠져나간다면 CPU 가 더 오래 sleep 상태에 머무르게 하는 `idle-entry` 동작을 취하는데에 약간의 이점이 있습니다.

이 릴리즈는 또한 계속해서 증가하는 사용예인 idle CPU 들에서 RCU 를 호출하는 행위를 처리하는데 사용되기 위한 `RCU_NOIDLE()` 을 추가했습니다. RCU 는 idle CPU 들을 무시하기 때문에, 이런 행위는 상당히 위험합니다. 따라서 이 새로운 `RCU_NONIDLE()` 매크로는 RCU read-side 크리티컬 섹션이 그 일을 할 수 있도록 idle로부터 급격하게 빠져나가도록 합니다.

RCU 의 CPU hotplug 처리가 개선되었고, `rcutorture` 는 RCU CPU stall 경고를 테스트하기 위한 기본적 기능을 일부 얻었으며, 이 stall 경고 자체는 더 많은 정보를 추가하고 sysfs 를 통해 타임아웃을 제어할 수 있는 기능을 추가함으로써 개선되었습니다. `TREE_RCU` 는 `CONFIG_SMP=n` 커널에서는 더이상 사용되지 않습니다; `TINY_RCU` 가 그대신 사용됩니다. 이 릴리즈는 또한, non-preemptible-RCU read-side 크리티컬 섹션 내에서의 잠자는 행위와 RCU read-side 크리티컬 섹션 내에서의 idle loop 진입을 위한 lockdep-RCU 체크의 추가를 포함합니다.

`TINY_RCU` 는 v3.0-rc7 RCU trainwreck [McK11a] 을 위한 `TREE_RCU` 수정사항들을 상속받았습니다. Grace-period 초기화 프로세스는 기존의 single-node 최적화를 제거했으며, offline 이 된 CPU 들에 남아있는 콜백들은 더이상 두번째의 grace period 전체를 거치지 않아도 되게 되었습니다. 더 나아가서, offline CPU 들은 더이상 RCU 콜백들을 호출할 수 없게 되었습니다.

이 외에도 더 많은 전력 효율성 코드의 수정사항들이 idle CPU 위에서 비참하게 살아있을 수도 있는 Time lazy 콜백들의 양을 제한했습니다. 마지막으로, Frédéric Weisbecker, Heiko Carstens, Julia Lawall, Hugh Dickins, Jan Beulich, 그리고 Paul Gortmaker 에 의해서 다수의 수정사항들이 추가되었습니다.

C.2.18 3.5 Linux Kernel

3.5 리눅스 커널은 `CONFIG_RCU_FAST_NO_HZ` 전력 효율성 코드에 타이머 처리와 `RCU_NOIDLE()` 에 의한 idle로부터의 정지에 대한 올바른 처리 등을 포함한, 더 많은 변화를 가져왔습니다.

이는 또한 `rcu_barrier()` 와 그 비슷한 것들에 의한 분열을 콜백들을 아무것도 갖지 않은 CPU 에 넣는 것을 막음으로써 줄였습니다. 이 작업은 또한 `rcu_barrier()` 와 offline 이 된 CPU 들에 의하고아가 된 콜백들 사이의 상호작용을 보다 명시적이게 만들었는데, 이는 일부 교묘한 race condition 들을 막는데 필요했던 것입니다. `__rcu_read_unlock()` 을 inline 하려다 실패한 시도는 그러나 RCU 의 task 종료 처리의 오버헤드를 줄이고 격리화 시키는 효과를 남겼습니다.

이 릴리즈는 Lai Jiangshan 에 의한 SRCU 의 완전한 재작성, 그 외에도 Jan Engelhardt, Michel Machado, 그

리고 Dave Jones 에 의한 수정사항들을 포함했습니다.

C.2.19 3.6 Linux Kernel

3.6 리눅스 커널은 `rcu_node` 트리의 leaf 단계 fanout 을 boot-time 패러미터를 통해 제어 가능하게 함으로써 수천개의 CPU 들을 갖는 시스템에서의 RCU 의 스케줄링 응답시간 효과를 줄이기 위한 변경 사항들 중 첫번째 것들을 포함했습니다. 이 변경은 grace-period 초기화 동안 만져져야 하는 메모리의 양을 4배 줄였고, 그로 인해 응답시간의 효과를 200 마이크로세컨드에서 60-70 마이크로세컨드로 줄였습니다. 이 릴리즈는 또한 `rcu_barrier()` 동시성을 증가시켰습니다.

전통을 따라서, 이 릴리즈는 또한 `CONFIG_RCU_FAST_NO_HZ` 기능을 위한 전력 효율성 상의 변경 사항을 포함했습니다. 마지막으로, 이 릴리즈는 Carsten Emde로부터의 초기화되지 않은 문자열에 대한 수정을 포함해서 여러개의 수정 사항들을 포함했습니다.

C.2.20 3.7 Linux Kernel

3.7 리눅스 커널은 grace-period 초기화 작업을 별개의 kthread 로 옮겼는데, 이 쓰레드는 preemptible 해서, 커다란 시스템에서의 grace-period 초기화 응답시간 문제를 제거할 수 있을 것입니다. 이 릴리즈는 또한, 기존의 `__rcu_barrier()` 의 악성의 `__stop_machine()` 로의 종속성을 제거했습니다. 이는 또한 Frédéric Weisbecker 의 `CONFIG_NO_HZ_FULL` bare-metal 기반기능 [Cor13b] 에 의해 필요한 RCU 기반시설을 일부 포함했으며, 이 RCU 기반시설의 대부분은 실제로 Frédéric 에 의해 작성되었습니다. 마지막으로, 이 릴리즈는 Tejun Heo, Thomas Gleixner, Li Zhong, 그리고 Dimiti Sivanich로부터의 수정사항들과 최적화들을 포함했습니다.

C.2.21 3.8 Linux Kernel

3.8 리눅스 커널은 새로운 `CONFIG_RCU_NOCB_CPU_Kconfig` 패러미터 [Cor12b] 의 형태로 RCU 콜백 오프로딩의 프로토타입 구현을 추가했으며, Paul Gortmaker 는 필요한 수정사항들을 몇 가지 제공했습니다. 이 프로토타입 구현은 CPU 0 가 offload 되지 않을 것을, 결국은 모든 콜백들이 CPU 0 로 처리될 것을 필요로 했습니다. 이는 분명 scalable 하지 않으며, 따라서 더 나은 구현이 나중에 나올 겁니다. RCU CPU stall-warning 메세지는 한번 더 업그레이드 되었고, RCU 의 CPU-hotplug 코드로의 일부 개선이 추가되었습니다.

Lai Jiangshan 은 SRCU 로의 static definition 기능을 추가했고 Michael Wang 은 RCU 의 오래된 debugfs 추적 기능을 재작업했습니다. Antti P. Miettinen 은 모든

RCU synchronous grace-period 기능들이 expedited 모드에서 동작하도록 강제하는 커널 부트 패러미터를 추가했으며, Eric Dumazet 은 RCU 콜백 batch-limit 문제를 수정했습니다.

C.2.22 3.9 Linux Kernel

3.9 리눅스 커널은 콜백들의 그룹들을 연관된 숫자로 태그하는데, 이는 RCU 가 콜백들을 지나치게 promote 시키는 것에 대한 걱정 없이 그들을 최대한 적극적으로 promote 시킬 수 있도록 해줍니다. 또한, 이 릴리즈는 TINY_RCU 를 위한 RCU CPU stall warning 들을 추가했습니다.

Lai Jiangshan 은 일부 SRCU 업데이트를 제공했는데, 이는 SRCU read-side 기능들이 idle 과 offline CPU 들에서 수행될 수 있게 해주었으며, 그외에도 일부 추가적인 수정사항들이 있었습니다. 추가적인 수정사항들이 Sasha Levin, Steven Rostedt, Li Zhong, Cody P. Schafer, 그리고 Josh Triplett 에 의해 제공되었습니다.

C.2.23 3.10 Linux Kernel

3.10 리눅스 커널로, RCU 는 마침내 시스템 레벨에서 측정될 수 있는 전력 효율 향상을 가져다주는 전력 효율성 메커니즘을 갖게 되었습니다 [MER13]. 여기서의 트릭은 CONFIG_RCU_FAST_NO_HZ 가 3.9에서 추가된 callback 태깅 기능을 사용하도록 한겁니다. 이는 idle 로 빠지는 CPU 들이 자기만의 콜백들을 분류하고 숫자를 매기는것만을 필요로 하게 됨을 의미하는데, 이는 기존의 RCU state machine 이 진행되게 하던 시도에 비해서 상당히 저렴한 비용만을 필요로 합니다. 또한, 이 callback 태깅 기능은 CPU 들이 미래의 grace period 에 대한 필요를 알리는 것을 가능하게 하는 개선이 있었는데, 이는 요청을 한 CPU 가 전체 과정 동안 잠들어 있었다는 사실에도 불구하고 CPU 들이 grace period 에 대한 필요를 알릴 수 있도록 했습니다.

또한, CONFIG_RCU_NOOB_CPU 기반기능들은 CPU 0 로의 종속성을 제거하도록 개선되어서, RCU 콜백들이 모든 CPU 들로 offload 될 수 있도록 했습니다.

이 릴리즈는 또한 Steven Rostedt, Eric Dumazet, Sasha Levin, Frédéric Weisbecker, Al Viro, Steven Whitehouse, Srivatsa S. Bhat, Jiang Fang, 그리고 Akinobu Mita 로부터의 수정사항들을 포함했습니다.

C.2.24 3.11 Linux Kernel

3.11 리눅스 커널은 3.9 와 3.10 에서의 callback 태깅 작업의 정리 작업 겨로가를 추가했고 유니프로세서 모드에서는 TREE_PREEMPT_RCU 를 수행하도록

TINY_PREEMPT_RCU 를 제거했습니다. 이 릴리즈는 또한 Paul Gortmaker 와 Kees Cook 으로부터의 수정사항들을 포함했습니다.

C.2.25 3.12 Linux Kernel

3.12 커널은 CONFIG_NO_HZ_FULL 이 효율적으로 언제 전체 시스템이 idle 인지를 판단할 수 있도록 하는데 필요한 기반기능을 제공하는 CONFIG_NO_HZ_FULL_SYSIDLE Kconfig 패러미터를 추가했습니다. CONFIG_NO_HZ_FULL 이 전체 시스템이 idle 상태임을 증명할 수 없다면, CPU 0 가 scheduling-clock 인터럽트를 active 상태로 유지해야 하는데, 이는 배터리 수명에 좋은 일이 아니기 때문에 [Cor13a] 이는 중요합니다.

이는 또한 rcutorture 의 테스트 범위를 synchronous, asynchronous, 그리고 expedited grace-period 기능들을 병렬적으로 테스트 할 수 있게 함으로써 rcutorture 를 개선시켰습니다. 또한 duplicate-callback 테스트를 추가했고 rcutorture 가 CPU-online 오퍼레이션이 실패했을 때 더 많은 정보를 제공하도록 만들었습니다. 마지막으로, 이 릴리즈는 Steven Rostedt, Tejun Heo, 그리고 Borislav Petkov 로부터의 수정사항들을 포함했습니다.

C.2.26 3.13 Linux Kernel

3.13 커널은 CONFIG_RCU_FAST_NOHZ 수행에서의 일부 항상점들을 포함하는데, 그가운데는 콜백들을 강화시키려는 너무 빈번한 시도를 막는 것도 있습니다. 그런 변화가 필요한 이유는, 그와 같이 콜백들이 강화될 수 있도록 허용하는 이벤트들은 일반적으로 수 밀리세컨드에 한번 정도만 발생해서, jiffy 마다 한번 이상의 빈도로 콜백들을 강화하려는 시도는 성능과 전력의 낭비 외에는 아무일도 하지 않는다는 것입니다. 따라서 3.13 커널은 현재 jiffy 내에 이미 콜백을 강화시키려 시도한 적이 없을 경우에만 시도를 합니다.

새로 추가된 `rcu_is_watching()` 함수는 호출자가 RCU read-side 크리티컬 섹션에 진입하기에 안전한가 그렇지 않은가를 판단할 수 있도록 해줍니다. 달리 말하자면, `rcu_is_watching()` 은 해당 CPU 가 idle 이거나 offline 상태가 아니라면 `true` 를 리턴합니다. 또한, 새로 추가된 (Michael S. Tsirkin 에 의해 제공된) `smp_mb__after_srcu_read_unlock()` 인터페이스는 `srcu_read_unlock()` 에서 전체 메모리 배리어를 칠 것을 보장합니다. 비록 `srcu_read_unlock()` 이 현재로써는 이미 전체 메모리 배리어를 제공하지만, 기존의 구현들은 그리하지 않았고 미래의 구현들은 또다시 그리하지 않게 될수도 있다는 점을 알아두시기 바랍니다.

RCU 의 소스 파일들은 3.13 부터 새로운 장소로 옮겨졌는데, kernel 디렉토리로부터 kernel/rcu 디렉

토리로 옮겨졌습니다.

마지막으로, Christoph Lameter 는 RCU 의 per-CPU 변수 API 사용을 업데이트 하는 패치를 하나 제공했고 Kirill Tkhai 는 CONFIG_RCU_NOCB_CPU_ALL 과 함께 만들어진 커널들이 중간중간 비어있는 CPU 숫자 규칙을 가진 시스템에서 돌아가는 경우 부팅 중에 panic 을 발생시킬 수 있던 문제의 수정을 제공했습니다.

C.2.27 3.14 Linux Kernel

3.14 에서 추가된 주요한 내용들은 커널 내의 rcutorture 테스트 스크립트의 개선사항들인데, 이는 테스트 케이스들의 long-overdue 리팩토링을 포함합니다. 이 릴리즈는 또한, 불필요하게 NO_HZ_FULL CPU 들에서 처리되고 있는 코어를 떠맡은 RCU 에 의해 이루어진 OS jitter 의 원인을 제거했습니다. 이 릴리즈는 또한 Fengguang Wu 에 의한 Coccinelle warning 의 수정사항, Lai Jiangshan 에 의한 `rcu_read_unlock_special()` 체크 제거를 위한 첫번째 단계, Chen Gang 으로부터의 일부 버퍼 오버플로우 방지, Teodora Baluta 에 의한 불필요한 `extern` 태그들의 제거, Josh Triplett 에 의한 `rcu_assign_pointer()` 로직의 개선을 포함했습니다.

The Answer to the Ultimate Question of Life, The Universe, and Everything.

“The Hitchhikers Guide to the Galaxy”,
Douglas Adams

Appendix D

Answers to Quick Quizzes

D.1 How To Use This Book

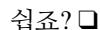
Quick Quiz 1.1:

이 Quick Quiz 들의 답은 어디에 있을까요?



Answer:

페이지 369에서 시작하는 Appendix D.



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 가이 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 가능하겠습니다만, 지금은 일단 실제 교수님들과 병렬 시스템들에 따라 그들이 주는 답의 유용성이 다를 수 있다는 점만 동의하고 넘어갑시다.

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.



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 PDF 에서 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.

2016년 중반부터 퀴즈는 답으로, 답은 퀴즈로 링크가 되어 있습니다. 답으로 이동하기 위해서는 “Quick Quiz” 나 작은 검정 네모를 클릭하세요. 답에서 퀴즈의 시작으로 가기 위해서는 답 시작점이나 작은 검정 네모를, 퀴즈의 끝으로 가려면 작은 하얀 네모를 클릭하세요. □

D.2 Introduction

Quick Quiz 2.1:

여봐요!!! 병렬 프로그래밍은 수십년간 엄청나게 어렵다고 알려졌다구요. 근데 당신은 그게 그렇게 어렵지 않다고 슬쩍 이야기하는 것 같네요. 뭔 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, 데드락부터 레이스 컨디션, 테스팅 커버리지 등등. 하지만 진짜 답은 그건 그렇게까지 어렵지는 않다입니다. 일단, 만약 병렬 프로그래밍이 정말로 그렇게 소름끼치도록 어렵다면, 어떻게 Apache나 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 이야기 한 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 생각하던 시대는 곧 지나갈 겁니다.²



Quick Quiz 2.2:

어떻게 병렬 프로그래밍이 시퀀셜 프로그래밍만큼 쉬운가 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?



¹ 그래요, 이 갑작스런 깨달음은 그가 좀 더 조심히 걷게 만들었습니다. 왜 그런걸 물어요?

² 병렬 프로그래밍은 시퀀셜 프로그래밍보다는 어렵습니다. 예를 들어, 병렬적으로 validation을 하는 것은 더 어렵습니다. 하지만 더이상 미질듯이 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)를 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 설령 그렇다 해도 얼마나 빨리 그리고 얼마나 많이 일을 할 수 있는지는 신경쓸 겁니다. 무엇보다, 가장 유명한 취미가 용 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?

Answer:

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 회생하면 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 별별 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 CPU를 추가해도 성능을 개선되지 않게 할 수 있을까요?



Answer:

잠재적 병목지점이 얼마든지 있습니다:

- 메인 메모리. 싱글 쓰레드가 모든 가용한 메모리를 사용하고 있다면, 추가된 쓰레드는 단순히 명청하게 자신을 페이지 아웃 시키겠죠.
- 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 있을 겁니다.

4. I/O 밴드위쓰. 싱글쓰레드가 I/O에 바운드 되어 있다면, 쓰레드들을 추가하는 것은 그저 그들 모두 관련된 I/O 자원에 줄을 서서 기다리고만 있게 될 겁니다.

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:

- 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.
- I/O 밴드위쓰. 각 쓰레드가 많은 스토리지 I/O나 네트워크 트래픽을 만든다면 너무 많은 수의 쓰레드는 과도한 I/O 큐잉 딜레이를 일으키고, 결국 성능이 또 저하될 것입니다. 일부 네트워크 프로토콜은 너무 많은 쓰레드가 네트워킹 이벤트를 만들어 시간 내에 그 응답을 받지 못할 경우 타임아웃이나 다른 문제상황을 낼 수 있습니다.

- 동기화 오버헤드. 많은 동기화 프로토콜에서 과도한 수의 쓰레드는 지나친 스피닝, 블락킹, 또는 룰백을 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:

- 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.
- 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.
- 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.
- 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.
- 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.
- 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 수행시킨다던지, 많이 사용하는 라이브러리의 병렬적 구현체를 사용한다면지, 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 있습니다만, 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 몇가지 가능한 복잡한 경우를 뺏습니다:

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.

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	2,600	7,220
Global Comms	195,000,000	542,000,000

Table D.1: Performance of Synchronization Mechanisms on 16-CPU 2.8GHz Intel X5550 (Nehalem) System

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 별별 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 CPU 들의 속도를 10 GHz 아래로 제한하고 있는, 에너지 소비 문제를 피해간다 해도 존재합니다.

Table D.1 과 페이지 22 의 Table 3.1 를 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.

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 롤을 가져오세요. 미제 휴지라면, 한 롤의 휴지는 약 350-500 조각의 휴지로 구성됩니다. 한개 조각이 하나의 클락 사이클이라고 생각하고, 휴지를 모두 풀어보세요.

그게 하나의 CAS 캐시 미스를 의미한다고 보면 됩니다.

더 비싼 시스템간 커뮤니케이션 대기시간을 보려면, 몇개의 휴지 롤 (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 튀어나나고, 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 발끝도 따라가지 못합니다. 컨덕터에서 측정된 전자파의 속도는 더도 아니고 덜도 아니고 빛의 속도의 발끝은 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. 이것은 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 그런 시스템을 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 네 개 CPU로 구성된 시스템에서 보다도 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 질 수도 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 잘 돌아갈 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 대부분의 영역은 간단한 공유 메모리 기법을 사용하도록 해줍니다.

□

D.4 Tools of the Trade

Quick Quiz 4.1:

이것들을 도구라고 하셨나요??? 제게 이것들은 도구라 기보다는 낮은 단계의 동기화 기능들처럼 보이는데요!

■

Answer:

그것들은 실제로 낮은 단계의 동기화 기능들이기 때문에 그렇게 보입니다. 하지만 또한, 그것들은 실제로 낮은 단계의 동시성 있는 소프트웨어를 만들기 위한 기본적인 도구들입니다. □

Quick Quiz 4.2:

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 4.3:

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 4.4:

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/O 를 포함합니다.
3. 스크립트에서 믿고 쓸 수 있는 사용 가능한 동기화 기본 도구들 역시 일반적으로 비싼 file I/O 를 포함합니다.
4. 스크립트 언어들은 많은 경우 너무 느립니다만, 더 낮은 단계의 프로그래밍 언어로 쓰여진, 오랫동안 수행되는 프로그램의 수행을 조정할 때에는 많은 경우 유용합니다.

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 4.2 에서 다루는 것과 같은 다른 병렬 프로그래밍 환경을 고려해 봐야 합니다.

**Quick Quiz 4.5:**

왜 `wait()` 함수는 그렇게 복잡해야만 하는거죠? 왜 그냥 셀 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 4.3 에서 본 것처럼, `wait()` 함수를 가지고 `waitall()` 함수를 만드는 건 어렵지 않습니다만 그 반대는 불가능하겠지요. 한번 특정 자식 프로세스에 대한 정보를 잊어버리면, 그건 복구될 수 없습니다.

**Quick Quiz 4.6:**

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

**Answer:**

맞습니다, 그리고 그리고 이 섹션은 나중에 메세징 기능(UNIX 파일, TCP/IP, 그리고 공유 파일 I/O 같은)과 메모리 매핑(`mmap()` 과 `shmget()` 같은) 기능을 포함하도록 확장될 수도 있을 겁니다. 그 전까지는, 이런 기능들에 대해 훨씬 자세하게 설명하는 다른 교재들이 많이 있고, 정말 잘 알고 싶다면 `man` 페이지(여주: UNIX 커맨드 `man`)나, 이런 기능을 사용하며 현존하는 병렬 어플리케이션들, 또는 리눅스 커널 구현의 소스 코드를 참고해도 될 것입니다.

Figure 4.4 의 부모 프로세스는 자식 프로세스가 종료될 때까지 자신의 `printf()` 를 위해 기다리고 있다는 것을 기억해 둘 필요가 있습니다. `printf()` 의 buffered I/O 를 같은 파일에 대해 여러 프로세스에서 동시적으로 사용하는 것은 일반적이지 않고, 그러지 않는 게 최선입니다. 정말로 동시적으로 buffered I/O 를 해야만 한다면, 당신의 OS 의 문서를 보세요. UNIX/Linux 시스템에서는 Stewart Weiss 의 강의 노트가 예제 [Wei13] 와 함께 좋은 소개를 제공합니다.

**Quick Quiz 4.7:**

Figure 4.5 의 `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 4.8:**

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 특허나 문제가 되는 플랫폼들 위에서는 동작하지 않습니다. 예를 들어, 32 비트 포인터와 16 비트 버스를 갖는 임베디드 시스템을 생각해 보세요. 그런 시스템에서는 하나의 포인터에 값을 저장하고 읽어오는 데이터 레이스에서 읽기는 아래쪽 16 비트는 예전 값이고 위쪽 16 비트는 새 값인 값을 읽어올 수도 있을 겁니다.

더도 아니고 덜도 아니고, 리눅스 커널의 경우에 있어서도, 데이터 경주 상황은 상당히 위험하고 가능하다면 막아져야 합니다 [Cor12a].



Quick Quiz 4.9:

제가 여러 쓰레드들이 한번에 같은 락을 쥐고 있게 하고 싶으면 어떻게 하죠?



Answer:

가장 먼저 당신이 해야 할 일은 왜 그려길 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 4.10:

왜 그냥 Figure 4.6 라인 5에서 `lock_reader()` 가 곧 바로 `pthread_mutex_t` 포인터를 받도록 하지 않는 거죠?



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 안 좋고 간단한 포인터 캐스팅에 비해 잘 하기가 어렵습니다.



Quick Quiz 4.11:

`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 애러 체킹을 해주는 함수로 감싸져서 사용되곤 합니다. 뒤에서, 우리는 이들을 리눅스 커널의 `spin_lock()` 과 `spin_unlock()` API 들로 감싸서 사용할 겁니다.



Quick Quiz 4.12:

“`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 4.13:

서로 다른 락을 사용하는건 쓰레드가 서로 상대의 중간 상태를 볼 수 있는 등 혼란스럽게 할 수 있는 것 같은데요. 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 할 겁니다.

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 4.14:

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 4.15:

잠깐만요!!! Figure 4.7 에서는 공유 변수 x 를 초기화하지 않았는데, Figure 4.8 에서는 왜 초기화 해야 했던 거죠?



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 4.16:

여기 저기 모든 곳에서 `READ_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, `READ_ONCE()` 의 사용은 코드를 읽는 사람에게 `goflag` 가 동시적 리드와 업데이트 동작에 연관되어 있음을 분명하게 보여줍니다. 하지만, `READ_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 4.17:

`READ_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 아닌가요? 맞습니다, 그리고 `__sync_fetch_and_add()` 아래에 해당 쓰레드가 시작해야 하는지 보기 전에 자신의 존재를 분명히 알리기 위해 필요한 메모리 배리어가 있습니다.



Quick Quiz 4.18:

예를 들어 `gcc __thread` 스토리지 클래스를 사용해 선언된 쓰레드별 변수에 접근할 때에도 `READ_ONCE()` 가 필요할까요?



Answer:

경우에 따라 다릅니다. 만약 그 쓰레드별 변수가 그 쓰레드에서만 접근되었다면, 그리고 시그널 핸들러에서 접근되지 않았다면, 필요하지 않습니다. 하지만 그렇지 않다면, `READ_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 저장해둠으로써 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 4.19:

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 4.20:

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 기록해두고, 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 Chapter 9에서 다루고 있습니다.



Quick Quiz 4.21:

Figure 4.10에서 100M에서의 경우 이외의 값들은 이상적인 선에서 부드럽게 멀어집니다. 반면, 100M에서의 값은 64 CPU에서 갑자기 이상적인 선으로부터 멀어지는군요. 또, 100M 값과 10M 값 사이의 거리는 10M 값과 1M 값 사이의 거리보다 작아요. 왜 100M 값은 이렇게 남들과 다른거죠?



Answer:

첫번째 단서는 64 CPU는 정확히 기계의 128 CPU의 절반이란 겁니다. 그 차이는 하드웨어 쓰레딩의 영향입니다. 이 시스템은 코어당 2개 하드웨어 쓰레드를 가지며 총 64 코어를 갖습니다. 64 쓰레드보다 적은 쓰레드가 돌 때 까지는 각 쓰레드가 자신의 코어를 하나씩 갖고 동작합니다. 하지만 쓰레드의 수가 64를 넘는 순간, 일부 쓰레드들은 코어를 공유해야만 합니다. 한 코어를 공유하는 쓰레드들은 일부 하드웨어 자원을 공유해야하기 때문에, 한 코어를 공유하는 두 쓰레드의 성능은 각자 코어 하나씩 가지고 도는 두 쓰레드의 것에 비해 낮을 수밖에 없습니다. 따라서 100M 경우의 성능은 리더-라이터 락에 의해 제한되는게 아니라 싱글 코어에서의 하드웨어 쓰레드간의 하드웨어 자원 공유에 의해 제한되는 것입니다.

이건 10M 경우에도 볼 수 있는데, 64 쓰레드까지 일관되게 떨어지던 성능 폭은 이후 급격하게 떨어져서 100M 경우와 비슷하죠. 64 쓰레드까지는 10M 경우도 리더-라이터 락의 확장성에 의해 성능이 제한되지만,

그 이후부터는 싱글 코어에서의 하드웨어 쓰레드간의 하드웨어 자원 공유로 인해 제한되는 것입니다.



Quick Quiz 4.22:

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



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 4.23:

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 4.24:

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



Answer:

안타깝지만, 아닙니다. 극명한 반례를 위해 Chapter 5을 보시기 바랍니다. ■

Quick Quiz 4.25:

리눅스 커널의 `fork()` 와 `wait()` 대체물은 어디 갔죠?

**Answer:**

그런건 없습니다. 리눅스 커널 내에서 실행되는 모든 태스크들은 메모리를 공유합니다. 당신이 거대한 메모리 매핑을 손으로 일일히 할 생각이 아니라면 말이죠.

**Quick Quiz 4.26:**

변수 `counter` 가 `mutex` 의 보호 없이 값 증가된다면 어떤 문제가 있을 수 있나요? ■

Answer:

Load-store 아키텍처의 CPU 들에서라면, `counter` 의 값 증가 연산은 다음과 같은 형태로 컴파일 될 수 있습니다:

```
LOAD counter, r0
INC r0
STORE r0, counter
```

그런 기계에서라면, 두개의 쓰레드가 동시에 `counter` 의 값을 읽어오고, 각자 그 값을 증가시킨 후, 각각 그 결과를 저장할 수 있습니다. 그렇게 되면 `counter` 의 새로운 값은 두개의 쓰레드가 각각 그 값을 증가시켰음에도 기존에 의해 1만큼만 클 것입니다.

**Quick Quiz 4.27:**

Per-thread-variable API 를 제공하지 않는 시스템에서는 어떻게 이를 우회할 수 있을까요? ■

Answer:

한가지 방법은 `smp_thread_id()` 로 인덱스 되는 배열을 생성하는 방법이 있겠고, 또 다른 방법은 `smp_thread_id()` 를 배열 인덱스로 매핑하는 해시 테이블을 상요하는 방법이 있겠습니다—이 API 들이 `pthread` 환경에서 실제로 동작하는 방법입니다.

또 다른 방법은 부모가 각각의 per-thread 변수를 담는 필드로 구성된 구조체를 할당하고, 이를 쓰레드 생성 시에 자식 쓰레드에게 넘겨주는 방법입니다. 하지만, 이 방법은 커다란 시스템에서는 큰 소프트웨어 엔지니어링 비용을 야기할 수 있습니다. 이를 자세히 알아보기 위해선, 커다란 시스템의 모든 전역 변수들이, 그것들이 C static 변수들인지 아닌지에 관계없이 하나의 파일에 선언되어야 하는 경우를 떠올려보세요! □

Quick Quiz 4.28:

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 5.1 에서 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 CPU 를 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 될 겁니다—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 할 겁니다. 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, 읽히지는 않은 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 5.4를 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 x86의 add-to-memory 명령어를 만들지 않나요? 그리고 CPU 캐시는 그걸 어토믹하게 수행하지 않나요? ■

Answer:

++ 연산자는 어토믹할 수도 있지만, 그래야만 한다는 규칙은 없습니다. 그리고 실제로, gcc는 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 이런 유의 카운팅 작업에 보내게 됩니다.

게다가, 이 챕터의 시작에서 이야기했듯, 카운팅은 공유 메모리 병렬 프로그램에서 마주칠 수 있는 문제들을 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 이 후의 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 디자인과 전력을 소모하게 될겁니다.

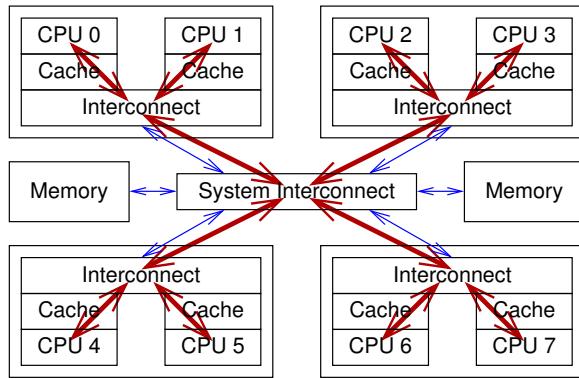


Figure D.1: Data Flow For Global Combining-Tree Atomic Increment

하지만 앞의 두가지 조건이 없다면? 그럼 당신은 Section 5.2에서 논의되는, 실제 상용화된 하드웨어에서 이상적 상황에 근접하는 성능을 보이는 알고리즘을 잘 고려해 봐야 할 겁니다.

앞의 두개 조건이 하나라도 걸려 있다면, 개선된 하드웨어에 약간의 희망이 있습니다. 콤비닝 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 예상 외 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. C는 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()` [Cor12a] 같은 `volatile` 접근법을 사용해서 컴파일러에 제약을 가할 수 있습니다. □

Quick Quiz 5.14:

Figure 5.6의 쓰레드별 `counter` 변수는 어떻게 초기화되나요? ■

Answer:

C 표준은 전역 변수의 초기값은 명시적으로 초기화되지 않는 이상 0이라고 명시합니다. 그리고, 사용자가 통계적 카운터들의 연속되는 값 사이의 차이에 관심 있는 게 아니라면, 초기값은 의미가 없습니다. □

Quick Quiz 5.15:

Figure 5.6의 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 5.6의 `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 $\frac{r\Delta}{N+1}$ 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 쌍여감을 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:

$$r\left(\frac{\Delta}{2} + t\right) \quad (\text{D.4})$$

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 $(1-2f)r$ 로 주어집니다. 이걸 Equation D.3 에 대입하면:

$$\frac{(1-2f)r\Delta}{2} \quad (\text{D.5})$$

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 5.8 의 `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 [Cor12a] 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 절삭될텐데니 별 문제 없습니다. □

Quick Quiz 5.18:

Figure 5.8 의 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 5.8 의 `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 5.8 의 최종적 일관성 알고리즘은 읽기에도 쓰기에도 매우 적은 오버헤드와 극단적인 확장성을 보이는 데, 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.

물론, 다른 트레이드오프를 갖는 것도 가능합니다. 예를 들어, 하나의 싱글쓰레드가 모든 최종적으로 일관적인 카운터들을 처리하도록 만들어질 수도 있는데, 이는 오버헤드를 하나의 CPU로 제한시킵니다만, 카운터들의 수가 늘어남에 따라 `update-to-read` 대기시간을 증가시키는 결과를 초래할 겁니다. 대안으로, 해당 싱글 쓰레드는 자주 업데이트 되는 카운터들을 더 자주 방문하는 식으로 카운터들의 업데이트 비율을 추적할 수 있습니다. 또한, 카운터들을 쥐고 있는 쓰레드들의 수는 전체 CPU 갯수의 일부로 설정될 수도 있으며, 이는 실행시간 도중에 정해질 수도 있습니다. 마지막으로, 각 카운터는 자신의 대기시간을 명시할 수 있고, 각 카운터에 요구된 대기시간을 지켜주기 위해 `deadline-scheduling` 테크닉이 사용될 수도 있습니다.

이외에도 얼마든지 더 많은 트레이드오프를 만들 수 있음이 분명합니다. □

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? ■

Answer:

정말 왜일까요?

gcc 에는 리눅스 커널은 무시할 수 있는 몇 가지 문제들이 존재합니다. 유저 레벨 쓰레드가 종료될 때, 그 쓰레드의 쓰레드별 변수는 모두 사라지는데 이로 인해, 적어도 유저 레벨 RCU(Section 9.5) 가 충분히 개선되기 전까지는, 쓰레드별 변수에의 액세스 문제는 복잡해집니다. 반면, 리눅스 커널에서는 한 CPU 가 오프라인이 되더라도 그 CPU의 CPU 별 변수는 여전히 매핑되어 있고 액세스 가능한 상태로 남습니다.

비슷하게, 새 유저 레벨 쓰레드가 생성되면, 그 쓰레드의 쓰레드별 변수는 갑자기 생겨나야 합니다. 반면,

리눅스 커널에서는 특정 CPU가 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 바운드된다는 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 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? ■

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 9에서 이 중재작업을 그보다도 가볍게 해줄 수 있는 메커니즘을 소개할 겁니다.

다른 방법으로는 쓰레드별 변수 대신 배열을 사용하는 방법이 있겠는데요, Alexey Roystman 이 이야기한대로 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 중요 한 영역이 아닌만큼 좀 더 안전에 치중했습니다. 만약 우리가 이 코드를 수천개의 CPU를 가진 기계에서 테스트 한다면야 이 락을 없애야 할수도 있습니다만 “겨우” 수백개 CPU의 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 CPU가 꺼져 있다 해도 접근 가능해요 — 심지어 해당 CPU가 한번도 켜겼던 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.h`의 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 셀 때, 카운터는 한번에 1씩 증가합니다. 반면, 바이트를 셀 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.

```

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         if (counterp[t] != NULL)
18             sum += *counterp[t];
19     return sum;
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

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 될겁니다. 리더에게도 쓸만한 성능과 확장성을 주면서 쓰기 작업도 여전히 빠르고 확장성 있게 하는 방법은 없을까요? ■

Answer:

글로벌한 추정값을 두는 게 한 방법이 될겁니다. 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 병행적 쓰기 오버헤드와 읽혀지는 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 5.2 에서 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 한계치 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 뭔가요? 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. “인티저 오버플로우.”

앞의 코드를 10의 값을 갖는 `counter` 와 `ULONG_MAX` 값을 갖는 `delta` 에 대해 수행해 보세요. 그리고 나서 Figure 5.12 의 코드로 한번 더 해보세요.

이 예제의 뒷부분은 인티저 오버플로우에 대한 깊은 이해를 필요로 하므로, 인티저 오버플로우 문제를 한번도 겪어본 적 없다면, 몇몇 예제를 가지고 이해해 보려 노력해보세요. 일부 경우에 있어서는 인티저 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 아닌채로 놔두질 않는거죠? ■

Answer:

사실 이전의 버전의 이 코드에서는 그렇게 했습니다. 하지만 더하기와 빼기는 매우 비용이 싼 동작이고, 모든 특수 케이스를 처리하는건 상당히 복잡합니다. 다시 말하지만, 직접 한번 해보세요, 다만 인티저 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 하는 것, 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 0으로 만듭니다.

3. Thread B 가 Thread A 의 `counter` 를 제거합니다.

4. 자신의 `countermax` 가 0이 아님을 확인했던 Thread A 는 `counter` 에 값을 더하고, 이로 인해 `counter` 는 0이 아닌 값을 갖습니다.

다시 말하지만, `countermax` 와 `counter` 를 별개의 변수로 두고서 어토믹하게 조정하는 것도 가능하긴 할 겁니다만, 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_crandmax()` 에 넘기는 코드도 보게 될겁니다. □

Quick Quiz 5.39:

Figure 5.17 의 `merge_crandmax()` 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 웬말이에요? `break` 몰라요??? ■

Answer:

해당 `goto` 를 `break` 로 대체하면 라인 15 에서 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_crandmax()` 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.

2. 만약 `flush_local_count()` 의 `atomic_xchg()` 가 두 빠른 수행경로의 `split_crandmax()` 뒤에, 그러나 빠른 수행경로의 `atomic_cmpxchg()` 보단 앞에 수행된다면, `atomic_cmpxchg()` 를 실패할거고, 빠른 수행경로를 재시작해서 앞의 case 1 의 상황으로 돌아갈 겁니다.

3. 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 스토어를 할 뿐인데, 어떻게 Figure 5.21 의 라인 21 에서의 `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:

REQ와 ACK 상태를 합치는게 나쁜 이유를 들어보자면:

1. 해당 느린 수행 경로는 REQ와 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 REQACK으로 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 14와 16) 중요합니다. 이것들이 없어지면, 컴파일러는 라인 14-17을 다음과 같이 바꿀 수도 있습니다:

```
14     theft = THEFT_READY;
15     if (counting) {
16         theft = THEFT_ACK;
```

느린 수행 경로는 잠깐 들어오는 값인 THEFT_READY를 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 5.24의 코드는 gcc와 POSIX에서 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 5.24의 라인 41에서는 왜 시그널을 다시 보내죠? ■

Answer:

지난 수십년간 많은 운영 체제는 갑자기 시그널을 잊어버리는 특성을 가졌기 때문입니다. 이게 가능인지 버그인지는 논쟁거리이지만, 그건 무의미합니다. 사용자가

보기에는 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 5.2.3에서 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? 뭐하는거예요??? ■

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 해결해야 할 문제들이 또 뭐가 있을 수 있을까요? ■

Answer:

엄청나게 많죠!

일단 몇가지 생각을 시작할 것들은:

1. 디바이스는 여러개가 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/O를 완료하는 쪽에서 디바이스 제거 쓰레드를 깨우는 편이 낫습니다.
3. I/O는 실패할 수 있으므로, `do_io()`는 리턴 값을 가져야 할 겁니다.
4. 디바이스가 고장나면, 마지막 I/O는 성공하지 못할 것입니다. 이런 경우, 예리 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 7.9에 소개되어 있습니다.
7. 폴링 루프는 매우 낮은 에너지 효율성을 초래할 것입니다. 이벤트 기반 설계가 나을 겁니다.

□

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? ■

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 ID와 연관된 `_thread` 변수 사이의 매팅을 유지하는 추가적인 계층을 갖습니다. □

Quick Quiz 5.59:

Table 5.1의 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 인스트럭션 들에 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:

식사하는 철학자들 문제를 위한 더 나은 해결책은 없을까요? ■

Answer:

그런 개선된 해결책이 Figure D.3 에 보여져 있는데, 철학자들에게 추가로 다섯개의 포크를 주는 것입니다. 이제 모든 다섯명의 철학자들이 동시에 식사를 할 수 있고, 따라서 철학자들이 남을 기다려야 할 필요가 없습니다. 또한, 이 방법은 질병 확산 제어를 염청나게 개선시킵니다.

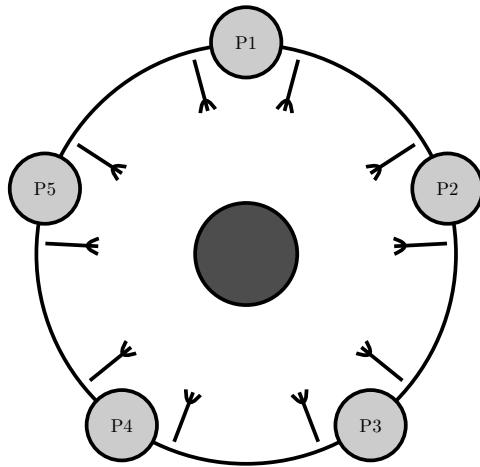


Figure D.3: Dining Philosophers Problem, Fully Partitioned

이 해결책은 누군가에겐 사기처럼 보이겠지만, 그런 “사기”가 많은 동시성 문제에 좋은 해결책을 찾는 핵심입니다. □

Quick Quiz 6.2:

그리고 어떤 관점에서 이 “수평적 병렬성”은 “수평적”이라 이야기 될 수 있는 걸까요? ■

Answer:

Inamn은 프로토콜 스택에 대한 작업을 했는데, 이 프로토콜 스택은 일반적으로는 어플리케이션을 꼭대기에 두고 하드웨어 연결을 바닥에 두는 식으로 수직적으로 그림그려집니다. 데이터는 이 스택의 위아래로 흐르게 됩니다. “수평적 병렬성”은 다른 네트워크로부터의 패킷들을 병렬적으로 처리하게 되는데, 반면 “수직적 병렬성”은 각 패킷들에게 서로 다른 프로토콜 처리 단계를 병렬적으로 줍니다.

“수직적 병렬성”은 또한 “파이프라인”이라고도 불립니다. □

Quick Quiz 6.3:

이 compound double-ended 큐 구현에서, 비어있던 큐가 락을 놓고 다시 잡는 과정 사이 더이상 비어있지 않게 된다면 어떻게 해야 할까요? ■

Answer:

이 경우, 그냥 더이상 비어있지 않은 그 큐의 원소를 디큐하고, 락들을 풀고, 리턴하면 그만입니다. □

Quick Quiz 6.4:

해시를 사용한 double-ended 큐는 좋은 해결책인가요? 그렇다면 왜고 그렇지 않다면 또 왜죠? ■

Answer:

이 질문에 답을 하는 가장 좋은 방법은 `lockhdeq.c`를 여러개의 멀티프로세서 시스템들 위에서 돌려보는 것이고, 가능한한 그렇게 해보시기를 권장합니다. 신경 써야 하는 한가지 이유는, 이 구현에서 각 오퍼레이션은 한개가 아니라 두개의 락들을 획득해야 한다는 점입니다.

일단은 잘 설계된 성능 연구를 보는 것으로 시작을 해야 할 겁니다.³ 순차적 구현과 비교해 보는 것을 잊지 마세요! □

Quick Quiz 6.5:

비어진 큐로 모든 원소들을 옮긴다구요? 대체 어떤 미친 세상에서는 이런 방법이 최적인 거죠??? ■

Answer:

데이터의 흐름이 아주 가끔만 그 방향을 바꿀 때에 최적입니다. 물론 double-ended 큐가 양쪽에서 동시에 비어진다면 매우 잘못된 선택일 겁니다. 이는 당연하게도 또 다른 질문을 일으키는데, 대체 어떤 세상에서는 양쪽에서 동시에 큐를 비게 만드는게 합당한 일이 되느냐는 것이죠. Work-stealing 큐들이 이 질문에의 가능한 답 중 하나일 것입니다. □

Quick Quiz 6.6:

왜 compound parallel double-ended queue 구현은 대칭적으로 만들어질 수 없는거죠? ■

Answer:

식사하는 철학자들 문제에 대한 해결책에서 포크에 숫자를 매기는 것이 그랬듯이, 락 계층구조를 사용해 데드락을 예방해야 하는 관계로 비대칭성이 필요해집니다 (Section 6.1.1를 참고하세요). □

Quick Quiz 6.7:

Figure 6.12의 line 28에서 왜 오른쪽 디큐를 또 시도하는거죠? ■

Answer:

이 쓰레드가 `d->rlock`을 line 25에서 놓고 나서 line 27에서 다시 잡는 사이에 다른 쓰레드가 원소를 집어넣었을 수도 있기 때문에 필요합니다. □

Quick Quiz 6.8:

왼쪽 락은 언젠가는 사용 가능할 겁니다!!! 그런데 Figure 6.12의 line 25에서는 오른쪽 락을 조건없이 놔버려야 하는거죠? ■

³ Dalessandro 등의 연구 [DCW⁺11] 와 Dice 등의 연구 [DLM⁺10] 가 좋은 시작점이 될 것입니다.

Answer:

원쪽 락이 획득 가능할 때에 그걸 얻기 위해 `spin_trylock()` 을 사용할 수도 있을 겁니다. 하지만, 실패할 경우에는 여전히 오른쪽 락을 잠시 내려놓고 두 락을 순서대로 다시 잡아야만 할 겁니다. 이런 변경을 가해보는건 (그리고 그게 가치있는 것인지 결정하는 것은) 독자의 몫으로 남겨두겠습니다. □

Quick Quiz 6.9:

double-ended 큐 문제에 왜 한개가 아니라 두개나 해결책이 있는 거죠? ■

Answer:

실은 최소한 세개입니다. Dominik Dingel 의 세번째 것은 `reader-writer` 락킹을 사용하는데, `lockrwdeq.c` 에서 보실 수 있을 겁니다. □

Quick Quiz 6.10:

직렬화된 double-ended 큐는 해시 기반 double-ended 큐보다 두배나 빠른데, 이런 현상은 제가 해시 테이블의 크기를 미친듯이 크게 만들어줘도 그렇습니다. 왜 그런 거죠? ■

Answer:

해시 기반 double-ended 큐의 락킹 디자인은 각 끝에서 한번에 하나의 쓰레드만을 허용하고, 또한 각 오퍼레이션을 위해 두개의 락 획득을 필요로 합니다. 직렬화된 double-ended 큐 또한 각 끝에서 한번에 하나의 쓰레드만을 허용하지만, 일반적인 경우에는 오퍼레이션당 하나의 락 획득만을 필요로 합니다. 따라서, 직렬화된 double-ended 큐는 해시 기반 double-ended 큐보다 성능이 나을 것으로 여겨집니다.

여러개의 동시적 오퍼레이션들을 가능하게 하는 double-ended 큐를 만들 수 있을 것 같나요? 그렇다면, 어떻게요? 불가능하다면, 왜죠? □

Quick Quiz 6.11:

double-ended 큐들을 위한 동시성 제어에 대한 훌씬 나은 방법이 있나요? ■

Answer:

한가지 방법은 풀어야 하는 문제를 여러개의 double-ended 큐들을 병렬로 이용할 수 있게 바꾸는 것으로, 더 간단한 단일 락 기반 double-ended 큐를 사용하고, 각 double-ended 큐를 일반적인 single-ended 큐의 쌍으로 바꾸는 것이 가능해집니다. 그런 “수평적 확장” 없이는, 성능 향상은 2.0으로 제한됩니다. 반면, 수평적-확장 설계들은 매우 커다란 성능 향상을 가질 수 있고, 큐의 각

끝에서 일하는 여러 쓰레드들이 있다면 특히 매력적인데, 여러 쓰레드가 있는 경우 디큐는 엄격한 순서 보장을 하지 않아도 되기 때문입니다. 무엇보다, 한 쓰레드가 어떤 아이템을 처음으로 제거했다는 사실은 그 아이템을 처음으로 처리할 것이라는 의미를 갖지는 않습니다 [HKLP12]. 그리고 그런 보장이 없다면, 이런 보장을 제공하지 않는데서 오는 성능 이득을 얻을 수도 있을 것입니다.

문제가 여러 큐를 사용하는 형태로 바뀔 수 있느냐 없느냐와 관계없이, 일이 몰아서 처리될 수 있어서 각 인큐와 디큐 오퍼레이션이 더 큰 단위의 일에 연관될 수 있는지 알아보는 것도 해볼 가치가 있습니다. 이 배치 (batching) 전략은 큐 데이터 구조체들로의 경쟁을 줄여줘서, 성능과 확장성 둘 다를 증가시켜주는데, Section 6.3 에서 알아볼 것입니다. 무엇보다, 높은 동기화 오버헤드를 겪어야만 한다면, 돈을 가치있게 쓰고 있는지 분명히 해두기 바랍니다.

다른 연구자들은 큐에서의 제한된 순서 보장의 이점을 살리는 다른 방법들을 연구하고 있습니다 [KLP12]. □

Quick Quiz 6.12:

크리티컬 섹션들과 관련한 이 모든 문제들은 우리가 크리티컬 섹션이 아예 없는 non-blocking 동기화 [Her90]를 사용해야 한다는 의미는 아닌가요? ■

Answer:

Non-blocking 동기화는 일부 상황에서는 매우 유용할 수 있지만, 만병통치약은 아닙니다. 또한, non-blocking 동기화는 Josh Triplett 에 의해 이야기되었듯 실제로는 크리티컬 섹션을 갖습니다. 예를 들어, compare-and-swap 오퍼레이션에 기반한 한 non-blocking 알고리즘에서, 최초의 로드와 이어지는 compare-and-swap 으로 이어지는 코드는 여러면에서 락 기반의 크리티컬 섹션과 유사합니다. □

Quick Quiz 6.13:

구조체가 그것의 락이 잡혀 있는 동안은 메모리 해제 되지 않도록 할 수 있는 방법들은 어떤 것들이 있을까요? ■

Answer:

여기 존재 보장 문제를 위한 해결책 몇가지가 있습니다:

1. 구조체별 락이 잡혀 있는 동안 잡혀 있는 정적으로 할당된 락을 두는 것으로, 계층적 락킹의 한 예입니다 (Section 6.4.2 을 참고하세요). 물론, 이런 목적으로 하나의 글로벌 락을 사용하는 것은 허용 불가능할 정도로 높은 락 경쟁 상황을 가져와서 극적인 성능과 확장성의 하락을 가져올 수 있습니다.

2. 정적으로 할당된 락들의 배열을 두어서, 구조체의 어드레스를 해싱해서 잡아야 할 락을 고르는 것으로, Chapter 7에서 설명된 방식입니다. 주어진 해시 함수가 충분히 높은 성능을 갖는다면, 이는 하나의 글로벌 락이 갖는 확장성의 한계를 해결할 수 있습니다만, 읽기가 대부분인 상황에서는 락 획득 오버헤드가 수용불가할 정도로 성능을 떨어뜨릴 수 있습니다.
3. 가비지 컬렉터를 제공하는 소프트웨어 환경이라면 가비지 컬렉터를 사용해서, 구조체가 참조되어 있는 동안은 메모리 해제되지 않도록 하는 것입니다. 이 방법은 잘 동작하고, 존재-보장의 짐을 (그리고 그외의 것들을) 개발자의 어깨에서 내려놓게 하지만, 프로그램의 가비지 컬렉션의 오버헤드를 갖습니다. 가비지 컬렉션 기술은 지난 수십년간 상당히 진보했지만, 그 오버헤드는 일부 어플리케이션에서는 수용하기 어려울 만큼 높을 수 있습니다. 또한, 일부 어플리케이션들은 개발자가 데이터 구조체의 배치와 위치의 조정에 대해 대부분의 가비지 컬렉션 환경에 비해 많은 연습을 필요로 할 수도 있습니다.
4. 가비지 컬렉터의 특수한 경우로, 글로벌 레퍼런스 카운터를 두거나 레퍼런스 카운터들의 글로벌한 배열을 두는 방법이 있습니다.
5. 안에서-밖으로의 참조 카운트라 생각할 수 있는, 해저드 포인터 [Mic04]들을 사용하는 방법이 있습니다. 해저드 포인터 기반의 알고리즘들은 쓰레드별 포인터들의 리스트를 두어서, 이 리스트들에 있는 포인터의 존재가 연관된 구조체로의 참조처럼 동작하게 합니다. 해저드 포인터들은 흥미로운 연구 방향입니다만, 상품화 단계 (2008년도 시점에선)에선 아직 잘 쓰여지지 않고 있습니다.
6. 트랜잭션 메모리(TM) [HM93, Lom77, ST95]를 사용해서 데이터 구조체로 요청되는 각각의 참조와 수정이 어토믹하게 수행되도록 하는 방법이 있습니다. TM은 최근의 수년간 대단한 흥분을 일으켰고 상품 단계 소프트웨어에서 일부 사용될 것처럼 보였지만, 개발자들은 몇 가지 주의를 기울여야 하며 [BLM05, BLM06, MMW07] 이는 특히 성능에 영향을 주는 코드에선 더더욱 그렇습니다. 특히, 존재 보장은 트랜잭션이 글로벌 레퍼런스부터 업데이트되는 데이터 원소들에 이르기까지 전체를 감싸야 할 것을 필요로 합니다.
7. 극단적으로 가벼운 가비지 컬렉터의 추상화로 생각될 수 있는, RCU를 사용하는 것입니다. 업데이트를 하는 쪽은 RCU로 보호되는 데이터 구조체를

RCU 읽는쪽이 여전히 참조를 하고 있는 동안은 메모리에서 해제시킬 수 없습니다. RCU는 읽기가 대부분인 데이터 구조체에서 상당히 많이 사용되고 있고, Chapter 9에서 이야기될 것입니다.

존재 보장에 대해 더 많은 내용을 위해선 Chapter 7과 9을 참고하세요. □

Quick Quiz 6.14:

싱글쓰레드로 동작하는 64 행 64 열 행렬 곱셈이 어떻게 1.0보다 낮은 효율성을 가질 수 있죠? Figure 6.23의 모든 조합에서의 결과들이 한 쓰레드에서만 돌아갈 때에는 정확히 1.0의 효율성을 보여야 하는 거 아닌가요?

■

Answer:

matmul.c 프로그램은 명시된 수의 워커 쓰레드들을 생성하므로, 하나의 워커 쓰레드만을 생성한 경우에도 쓰레드 생성 오버헤드는 발생할 수 있습니다. 하나의 워커 쓰레드의 경우에 쓰레드 생성 오버헤드를 없애기 위한 변경을 만드는 것은 독자분들의 연습문제로 남겨 두겠습니다. □

Quick Quiz 6.15:

행렬 곱셈에서 데이터 병렬화 기법이 어떻게 도움이 될 수 있나요? 그건 이미 병렬적인 데이터잖아요!!! ■

Answer:

주의를 기울이고 있으신 것 같아 기쁩니다! 이 예는 데이터 병렬성은 매우 좋은 것이긴 하지만, 어떤 그리고 모든 비효율성의 원인을 자동으로 제거하는 마술봉은 아니란 것을 보이기 위한 것임을 보이기 위한 것입니다. 64 쓰레드에게 “한정적인” 상황에서 조차도 전체 성능을 가지고 선형적으로 성능을 확장하는 데에는 설계와 구현의 모든 단계에서의 세심한 주의가 필요합니다.

특히, 분할된 조각의 크기에 매우 세심한 관심을 기울여야만 합니다. 예를 들어, 64 행 64 열 행렬 곱셈 문제를 64 쓰레드에게 쪼갠다면, 각 쓰레드는 64 개의 부동소수점 곱셈 연산만을 하게 될 것입니다. 부동소수점 곱셈 연산의 비용은 쓰레드 생성의 오버헤드에 비하면 매우 작습니다.

교훈: 변화무쌍한 입력을 받을 수 있는 병렬 프로그램을 가지고 있다면, 항상 입력의 크기를 체크하는 과정을 포함시켜서 병렬화 시킬 가치가 없을 정도로 너무 작은 입력 사이즈에 대응하도록 하십시오. 그리고 병렬화에 도움이 되지 않을 정도라면, 쓰레드를 생성하는데 필요한 오버헤드를 감내하기에도 도움이 되지 않을 정도입니다, 그렇죠? □

Quick Quiz 6.16:

계층적 락킹이 잘 동작할 만한 상황은 뭐가 있을까요? ■

Answer:

Figure 6.26의 라인 31에서의 비교 연산이 훨씬 무거운 연산으로 교체된다면, `bp->bucket_lock`을 놓는 작업은 아마도 락 경쟁을 충분히 줄여줘서 추가적인 `cur->node_lock`의 획득과 해제에 드는 오버헤드보다 우세해질 수도 있을 것입니다. □

Quick Quiz 6.17:

Figure 6.32에서, 세개의 샘플들마다 할당 수행 시간 길이가 증가함에 따라 성능이 따라 증가하는 패턴이 존재하는데, 예를 들어, 할당 수행 시간 길이 10, 11, 12의 경우입니다. 왜 그런거죠? ■

Answer:

이는 per-CPU 타겟 값이 3이기 때문입니다. 할당 수행 시간 길이 12의 경우는 글로벌 풀 락을 두번 잡아야 하는데, 반면 할당 수행 시간 길이 13은 글로벌 풀 락을 세번 잡아야 합니다. □

Quick Quiz 6.18:

할당 실패 횟수들은 두개 쓰레드를 사용한 테스트에서 할당 수행 길이가 19 이상일 때 발견되었습니다. 글로벌 풀 사이즈가 40이고 per-thread 타겟 풀 사이즈 s 가 3이고, 쓰레드의 갯수 n 가 2이며, per-thread 풀들은 초기에 비어있고 메모리를 아무도 사용하고 있지 않다고 가정하는 조건 하에서, 할당 실패가 발생할 수 있는 최소의 할당 수행 시간 길이 m 은 무엇일까요? (각 쓰레드는 메모리의 m 블럭을 할당받고 m 블럭의 메모리를 해제하는 것을 반복함을 다시 이야기 합니다.) 그 대신에, n 쓰레드들이 각각 풀 사이즈 s 를 갖는다면, 그리고 각 쓰레드가 먼저 메모리의 m 개 블럭을 할당받고 그 m 블럭들을 해제하는 것을 반복한다고 하는 조건이라면, 글로벌 풀 사이즈는 얼마나 커야 할까요? Note: 올바른 답을 얻기 위해서는 `smpalloc.c` 소스 코드를 자세히 들여다보고, 한단계 한단계씩 들여다봐야 할 겁니다. 분명 경고했어요! ■

Answer:

이 답안은 Alexey Roystan 이 제출한 것으로부터 받아들여졌습니다. 이는 다음의 정의에 기초합니다:

g 전역적으로 사용 가능한 블럭들의 갯수

i 초기화 쓰레드의 per-thread 풀에 남아 있는 블럭들의 수. (이게 코드를 들여다봐야 하는 이유 중 하나입니다!)

m 할당/해제 수행 시간 길이.

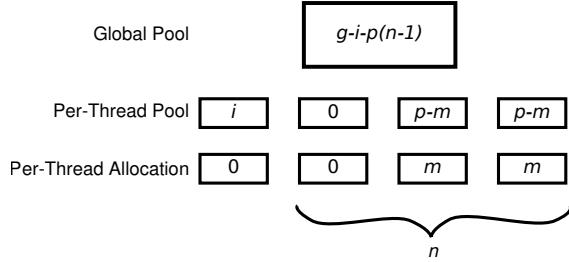


Figure D.4: Allocator Cache Run-Length Analysis

n 초기화 쓰레드를 제외한 쓰레드의 수.

p 실제로 할당된 블럭들과 per-thread 풀에 남아있는 블럭들을 포함해서 쓰레드 당 최대 블럭 소비량.

g, m , 그리고 n 의 값은 문제에서 이미 주어졌습니다. p 의 값은 m 을 s 의 배수로 반올림된 값으로, 다음과 같습니다:

$$p = s \left\lceil \frac{m + s - 1}{s} \right\rceil \quad (D.6)$$

i 의 값은 다음과 같습니다:

$$i = \begin{cases} g \pmod{2s} = 0 : 2s \\ g \pmod{2s} \neq 0 : g \pmod{2s} \end{cases} \quad (D.7)$$

이 값들 사이의 관계는 Figure D.4에 보여진 것과 같습니다. 글로벌 풀은 이 그림의 꼭대기에 그려져 있고, “나머지” 초기화 쓰레드의 쓰레드별 풀과 쓰레드별 할당은 가장 왼쪽의 두개의 박스들로 그려져 있습니다. 이 초기화 쓰레드는 할당된 블럭을 가지고 있지 않습니다만, i 개의 블럭들이 초기화 쓰레드의 쓰레드별 풀에 고립되어 있습니다. 가장 오른쪽의 두개의 박스들은 최대 가능한 수의 블럭을 쥐고 있는 쓰레드들의 쓰레드별 풀들과 쓰레드별 할당들이고, 왼쪽에서 두번째의 박스들은 현재 할당을 시도하고 있는 쓰레드를 가리킵니다.

블럭 전체의 갯수는 g 이고, 쓰레드별 할당과 쓰레드별 풀들을 함께 고려해보면, 우리는 글로벌 풀이 $g - i - p(n - 1)$ 블럭들을 가지고 있음을 알 수 있습니다. 할당을 하는 쓰레드가 할당에 성공하려면, 글로벌 풀에는 최소한 m 개의 블럭들은 남아 있어야 하는데, 달리 말하자면 다음과 같습니다:

$$g - i - p(n - 1) \geq m \quad (D.8)$$

문제에서 $g = 40, s = 3, n = 2$ 였습니다. Equation D.7에 따라 $i = 4$ 이고 Equation D.6에 따라 $m = 18$ 이면 $p = 18$ 이고 $m = 19$ 이면 $p = 21$ 입니다. 이것들을

Equation D.8에 대입해 보면 $m = 18$ 은 넘치지 않는 반면, $m = 19$ 는 넘칠 것을 알 수 있습니다.

i 의 존재는 버그로 여겨질 수도 있을 것입니다. 무엇보다도, 왜 메모리를 초기화 쓰레드의 캐시에 고립된 채로 남겨둡니까? 이를 고치는 한가지 방법은 현재 쓰레드의 풀을 글로벌 풀로 비워버리는 `memblock_flush()` 함수를 제공하는 것이 될 것입니다. 그렇게 되면 이 초기화 쓰레드는 블럭들을 모두 해제한 이후에 이 함수를 호출할 수 있을 것입니다. □

D.7 Locking

Quick Quiz 7.1:

희생양 역할을 한게 어떻게 명예로운 것으로 여겨질 수가 있나요??? ■

Answer:

락킹이 연구 논문에서의 희생양 역할을 맡게 된 이유는 그것이 실제적으로 상당히 많이 사용되기 때문입니다. 그렇지 않고 아무도 락킹에 신경쓰지 않는다면, 대부분의 연구 논문들은 그것에 대해 언급하는 것조차도 고려하지 않을 것입니다. □

Quick Quiz 7.2:

하지만 데드락의 정의는 단지 각 쓰레드가 최소 하나 이상의 락을 잡고 일부 쓰레드에 의해 잡혀 있는 또 다른 락을 기다리고 있는 것이잖아요. 어떻게 거기에 사이클이 존재하는지 알 수 있죠? ■

Answer:

그래프에 사이클이 존재하지 않는다고 생각해 봅시다. 만약 그렇다면 우리는 방향성 비순환 그래프 (DAG: directed acyclic graph)를 가지게 될텐데, 이 그래프는 최소한 하나의 잎사귀 노드 (leaf node)를 가질 겁니다.

이 잎사귀 노드가 락이라면, 우린 어떤 쓰레드에도 잡혀 있지 않은 락을 기다리고 있는 한 쓰레드를 가지고 있다는 이야기가 됩니다 (그리고 이 경우 해당 쓰레드는 즉시 해당 락을 얻게 되겠죠.)

반면에, 만약 이 잎사귀 노드가 쓰레드였다면, 어떤 락도 기다리고 있지 않은 쓰레드가 하나 존재한다는 이야기가 되는데, 이것 역시 데드락의 정의를 위반합니다. (그리고 이 경우, 해당 쓰레드는 수행중이거나 락이 아닌 무언가에 의해 블락되어 있거나 하는 상태일 것입니다.)

따라서, 이 데드락의 정의에 기반하면, 앞서 설명한 데드락 상태를 나타내는 그래프에는 사이클이 존재해야만 합니다. □

Quick Quiz 7.3:

이 규칙에 대한 어떤 예외가 존재해서, 라이브러리 코드가 호출자의 어떤 함수도 실행하지 않는다는 조건 하에서도 라이브러리와 호출자 둘 다의 락을 포함하는 데드락 사이클이 실제로는 존재할 수도 있을 수 있나요? ■

Answer:

실제로 존재합니다! 여기 그런 예외들 중 일부가 있습니다:

1. 라이브러리 함수의 인자 가운데 하나가 이 라이브러리 함수가 획득하는 락으로의 포인터라면, 그리고 만약 그 라이브러리 함수가 그 호출자의 락을 잡은 상태로 자신의 락들을 잡는다면, 호출자의 락과 라이브러리 락들이 함께 들어간 데드락 사이클을 가질 수도 있게 됩니다.
2. 라이브러리 함수들 가운데 하나가 호출자에 의해 잡힌 락으로의 포인터를 리턴하게 된다면, 그리고 만약 그 호출자가 라이브러리의 락을 잡은 채로 자신의 락을 잡는다면, 이번에도 호출자의 락과 라이브러리의 락들이 함께 들어간 데드락 사이클을 가질 수도 있게 됩니다.
3. 라이브러리 함수들 가운데 하나가 락을 하나 잡고 그걸 잡은 채로 리턴하게 된다면, 그리고 만약 그 호출자가 자신의 락들 가운데 하나를 잡게 된다면, 또다시 호출자의 락과 라이브러리의 락이 모두 연관된 데드락 사이클을 생성하는 방법이 됩니다.
4. 호출자가 락을 잡는 시그널 핸들러를 가지고 있다면, 호출자의 락과 라이브러리의 락이 모두 연관된 데드락 사이클이 있을 수 있습니다. 하지만, 이 경우에 라이브러리의 락들은 데드락 사이클에 죄가 없는 방관자일 뿐입니다. 그렇다면 하나, 시그널 핸들러 안에서 락을 획득하는 것은 대부분의 환경에서 하지 말아야 할 일임을 알아두시기 바랍니다—그건 그냥 나쁜 생각일 뿐 아니라, 지지받지 못할 일입니다. □

Quick Quiz 7.4:

하지만 `qsort()`가 비교 함수를 실행하기 전에 그 자신의 락들을 모두 해제해버리면, 다른 `qsort()` 쓰레드들과의 경주 문제를 어떻게 지킬 수 있나요? ■

Answer:

비교되는 데이터 원소들을 사유화 시켜버리거나 (Chapter 8에서 설명한 것과 같이) 레퍼런스 카운팅과 같은 뒤로 미루기 메커니즘을 사용하는 것 (Chapter 9에서 설명한 것과 같이)으로 가능합니다. □

Quick Quiz 7.5:

락으로의 포인터를 함수에 넘기는 게 완벽하게 합리적인 예외를 하나만 들어 보세요. ■

Answer:

락킹 도구들이죠, 당연히! □

Quick Quiz 7.6:

`pthread_cond_wait()` 함수가 먼저 그 뮤텍스를 해제하고 나서 다시 획득한다는 사실은 데드락의 가능성은 제거하는 거 아닌가요? ■

Answer:

전혀 그렇지 않습니다!

mutex_a 를 획득하고, 그 후엔 mutex_b 를 획득하고 나서는 mutex_a 를 `pthread_cond_wait` 에 넘겨버리는 한 프로그램을 생각해 보세요. 이제 `pthread_cond_wait` 는 mutex_a 를 해제합니다만, 리턴하기 전에 그걸 다시 잡습니다. 만약 그 사이에 다른 쓰레드가 mutex_a 를 잡고서 mutex_b 를 사용해 블락된다면, 이 프로그램은 데드락에 빠지고 맙니다. □

Quick Quiz 7.7:

Figure 7.9 에서 Figure 7.10 로의 변경이 어디서나 적용될 수 있는 걸까요? ■

Answer:

절대 그렇지 않습니다!

이 변경은 `layer_2_processing()` 함수가 같은 입력에 대해선 항상 같은 출력을 내놓는다는 가정을 가지고 있는데, `layer_1()` 의 라우팅 결정이 변화되면 같은 패킷에 대해 이 함수가 여러번 수행될 수도 있기 때문입니다. 따라서, 실제 세상에서 이런 변경을 적용하기는 상당히 복잡해질 수 있습니다. □

Quick Quiz 7.8:

그렇지만 Figure 7.10 에서의 추가된 복잡도는 그게 데드락을 막는다는 점을 생각하면 꽤 가치있는 거예요, 맞죠? ■

Answer:

아마도요.

만약 `layer_1()` 에서의 라우팅 결정이 충분히 자주 바뀐다면, 해당 코드는 항상 재시도만 할테고, 진전은 전혀 이루지 못할 겁니다. 이런 현상을 가리켜 어떤 쓰레드도 진행을 못하고 있을 때에는 “livelock” 이라 하고, 일부 쓰레드는 진행을 이루고 있지만 다른 것들은 못하고 있을 때에는 “starvation” 이라 합니다 (Section 7.1.2 을 참고하세요). □

Quick Quiz 7.9:

Section 7.1.1.6 에서 이야기한 “필요한 락들을 먼저 획득하기” 방법을 사용할 때에, 어떻게 하면 livelock 을 막을 수 있을까요? ■

Answer:

추가적인 글로벌 락을 제공하는 겁니다. 어떤 쓰레드가 반복적으로 필요한 락들을 획득하려 노력하고 실패했다면 그 쓰레드는 조건없이 새로운 글로벌 락을 획득하도록 하고 조건없이 모든 필요한 락들을 획득하게 하는 겁니다. (Doug Lea. 에 의해 제안된 방법입니다.) □

Quick Quiz 7.10:

시그널 핸들러 안에서 획득되는 Lock B 를 잡은 채로 시그널들을 블락시키지 않고 시그널 핸들러 밖에서 획득되는 Lock A 를 잡는 행위가 해선 안되는 행위이죠? ■

Answer:

데드락을 유발할 수 있기 때문입니다. Lock A 가 시그널들을 블락시키지 않은 상태로 시그널 핸들러의 밖에서 잡혔다면, 이 락을 잡고 있는 상태로 어떤 시그널은 처리될 수 있을 겁니다. 이제 그 시그널에 연관된 시그널 핸들러는 Lock B 를 잡으려 할텐데, 따라서 Lock B 는 Lock A 를 잡은 채로 획득될 것입니다. 따라서, 질문에서 이야기한 것처럼 우리가 Lock B 를 잡은 채로 Lock A 를 잡으려 한다면 데드락 사이클이 생기게 되는 것입니다.

따라서, 시그널 핸들러 안에서 획득하는 락을 잡고 시그널들을 블락시키지 않은 채로 시그널 핸들러 밖에서 획득되는 락을 잡는 행위는 금지된 행위입니다. □

Quick Quiz 7.11:

시그널 핸들러 안에서 어떻게 시그널들을 블락시킬 수가 있죠? ■

Answer:

그렇게 할 수 있는 가장 간단하고 가장 빠른 방법은 시그널을 설정할 때 `sigaction()` 함수에 전달하는 `struct sigaction` 객체의 `sa_mask` 필드를 사용하는 겁니다. □

Quick Quiz 7.12:

시그널 핸들러들 안에서 락을 잡는 행위가 그렇게 나쁜 생각이라면, 그걸 안전하게 해내기 위한 생각 자체는 대체 왜 하는거죠? ■

Answer:

똑같은 규칙이 운영체제 커널들과 일부 임베디드 어플리케이션들에서 사용되는 인터럽트 핸들러들에도 적용되기 때문입니다.

많은 어플리케이션 환경에서, 시그널 핸들러 안에서 락을 잡으려 하는 행위는 해로운 행위로 여겨져 왔습니다 [Ope97]. 하지만, 그런 분위기가 똑똑한 개발자들이 (보통은 현명하지 못한 짓이지만) 어토믹 오퍼레이션들을 사용해서 직접 만든 락들을 사용해대는 것을 멈추게 하지는 않았습니다. 그리고 어토믹 오퍼레이션들은 많은 경우 시그널 핸들러 안에서 사용되어도 전혀 문제가 되지 않습니다. □

Quick Quiz 7.13:

다양한 객체들 사이로 실행 제어가 완전 자유롭게 오가는 객체 지향 어플리케이션이어서 직관적인 락킹 계층, 레이어, 또는 다른 것들이 존재하지 않는 경우⁴라면 그 어플리케이션은 어떻게 병렬화 시켜야 할까요? ■

Answer:

다양한 방법이 있습니다:

1. 기계적인 또는 전기적인 기기를 위한 좋은 설계를 찾아내기 위해서라든지 하는 경우를 위해 많은 수의 시뮬레이션을 수행하게 되는 시뮬레이션을 통한 모수 검색의 경우라면, 그 시뮬레이션을 단일 쓰레드로 수행되도록 두되, 많은 수의 시뮬레이션의 수행들을 병렬로 돌리는 겁니다. 이 방법은 객체 지향적 설계를 유지하고, 높은 수준으로 병렬성을 끌어올리며, 또한 동기화 오버헤드 역시 없답니다.
2. 객체들을 여러 그룹들로 분류하는데, 한번에 그룹과 그룹 사이의 객체들 사이의 수행은 없도록 분류를 합니다. 그리고 각 그룹별로 락을 사용하도록 연관시킵니다. 이는 Section 7.1.1.7에서 알아본 한번에 한 락만 잡기 전략의 한 예라고 볼 수 있습니다.
3. 객체들을 여러 그룹들로 분류하는데, 쓰레드들은 모두 한 그룹의 객체들에 대해 어떤 그룹 단위의 순서로만 수행을 할 수 있도록 분류합니다. 그리고 각 그룹별로 락을 하나씩 연관시키고, 그룹들 단위의 락킹 계층을 적용시킵니다.
4. 임의로 선택된 계층을 락들에 적용하고 락을 비순서적으로 획득해야 하는 경우에는 Section 7.1.1.5에 이야기 했던 조건적 락킹을 적용합니다.
5. 일련의 오퍼레이션들을 수행하기 전에, 그 오퍼레이션들의 수행을 위해 어떤 락들이 획득되어야 할지 예측하고, 그것들을 정말로 어떤 업데이트를 하기 전에 획득합니다. 그 예측이 틀렸던 것으로 드러난다면, 모든 락들을 내려놓고 경험으로부터 얻은 정보를 바탕으로 수정된 예측을 가지고 다시 락

⁴ “객체 지향 스파게티 코드”라고도 알려져 있죠.

획득을 시도합니다. 이런 접근법은 Section 7.1.1.6에서 이야기 되었습니다.

6. 트랜잭션 메모리를 사용합니다. 이 접근법은 많은 장점과 단점을 가지고 있는데, Section 17.2에서 다루어질 겁니다.
7. 어플리케이션 자체가 좀 더 동시성에 친화적이라도 수정을 합니다. 이는 심지어 어플리케이션이 단일 쓰레드로 수행될 때조차도 빠르게 동작하는 부가적 효과를 만들어낼 수도 있습니다만 어플리케이션을 수정하기가 좀 더 어렵게 만들어 버릴 수도 있습니다.
8. 락킹 이외에 뒤의 챕터들에서 설명할 기법들을 적용해 봅니다. □

Quick Quiz 7.14:

Figure 7.11에 보여진 livelock은 어떻게 예방할 수 있나요? ■

Answer:

Figure 7.10는 일부의 좋은 힌트들을 제공합니다. 많은 경우, livelock들은 락킹 설계를 다시 한번 들여다 봐야 한다는 힌트 역할을 하곤 합니다. 또는, 락킹 설계가 “성장했다면” 첫번째로 들여다봐야 하는 영역이기도 합니다.

그렇다고는 하나, Doug Lea에 의하면 훌륭하고 충분한 접근법은 Section 7.1.1.5에서 설명된 것처럼 조건적 락킹을 하되, Section 7.1.1.6에서 설명된 대로 공유되어 있는 데이터를 수정하기 전에 모든 필요한 락들을 먼저 획득하는 방법과 조합해서 쓰는 것입니다. 주어진 크리티컬 섹션이 너무 많이 재시도를 하게 되면, 조건 없이 글로벌 락을 잡아버리고, 조건 없이 모든 필요한 락들을 잡아버리는 방법입니다. 이 방법은 deadlock과 livelock 두 가지 모두를 예방할 뿐더러, 이야기한 글로벌 락이 너무 자주 잡혀지지는 않는다는 가정 하에서는 합리적인 수준으로 확장성을 갖습니다. □

Quick Quiz 7.15:

Figure 7.12에서 어떤 문제를 발견할 수 있나요? ■

Answer:

여기 두가지가 있습니다:

1. 대부분의 경우에 대기시간 1초는 너무나 긴 시간입니다. 대기 간격은 대략 해당 크리티컬 섹션을 수행하는데 걸리는 시간 정도에서부터 시작해야 하는데, 이는 일반적으로 마이크로세컨드나 밀리세컨드 영역에 있을 것입니다.

2. 해당 코드는 오버플로우를 체크하지 않고 있습니다. 반면에, 이 버그는 그 앞의 버그에 의해 무효가 됩니다: 32 비트로 나타낼 수 있는 최대의 초는 50년이 넘습니다.

□

Quick Quiz 7.16:

락 경쟁정도가 충분히 낮아서 비공정성을 예방할 수 있을 정도가 되는 좋은 병렬 설계를 사용하는 편이 더 낮지 않을까요? ■

Answer:

어떤 면에서는 그렇게 하는게 낫겠습니다만, 가끔은 높은 락 경쟁률을 갖게 되어버리는 설계를 사용하는 편이 더 적합한 상황도 존재합니다.

예를 들어, 매우 드물게 발생하는 에러 조건을 처리하는 시스템을 생각해 봅시다. 이렇게 매우 드물게 발생하는 에러 조건에 대해서는 매우 낮은 성능과 확장성을 갖지만 간단한 에러 처리 설계를 사용하는 게 그런 매우 드물게 발생하는 에러 조건들 가운데 하나가 실제로 발생했을 때에만 효과를 발휘하는 복잡하고 디버깅하기 어려운 설계를 사용하는 것에 비해 최선일 수 있습니다.

그렇다고는 하나, 간단할 뿐만 아니라 에러 조건에 대해서도 효과적인 설계를 만들어내는데 노력을 기울이는 것이 할만한 가치가 있는데, 문제를 조개는 것이 한 예입니다. □

Quick Quiz 7.17:

락을 잡은 쓰레드는 어떻게 다른 쓰레드로부터 간섭을 받을 수도 있을까요? ■

Answer:

만약 어떤 락에 의해 보호되는 데이터가 그 락과 같은 캐시 라인 위에 위치해 있다면, 다른 CPU 들이 그 락을 잡으려 시도하는 행위는 그 락을 잡고 있는 CPU 쪽에 비용이 비싼 캐시 미스를 초래시킬 수 있을 겁니다. 이건 *false sharing* 의 특별한 경우라 할 수 있는데, 다른 락들로 보호되는 두개의 변수가 같은 캐시 라인을 공유하고 있으면 벌어지는 현상이기도 합니다. 반대로, 어떤 락이 그 락을 통해 보호되는 데이터와 다른 캐시 라인에 위치해 있는다면, 그 락을 잡은 CPU 는 그 변수에의 첫 번째 접근 때에만 캐시 미스를 겪게 될 것입니다.

물론, 락과 데이터를 별도의 캐시 라인에 위치시키는 것으로 인한 단점은, 그렇게 설계된 코드로의 변경은 락을 잡기 위한 경쟁이 가열차지 않은 경우에는 단 한번이었을 캐시 미스가 두번의 캐시 미스로 바뀌게 된다는 점입니다. □

Quick Quiz 7.18:

배타적 락을 잡고나서 곧바로 풀어버리는, 즉, 텅 빈 크리티컬 섹션과 같은 것을 갖는 행위가 어떤 의미를 가질 수 있을까요? ■

Answer:

텅 빈 크리티컬 섹션의 사용은 드물긴 하지만, 분명히 존재합니다. 배타적 락의 의미적 기능은 두개의 부분으로 나뉘어진다는게 핵심입니다: (1) 모두가 잘 알고 있는 데이터 보호의 기능과 (2) 주어진 락을 놓는 것으로 같은 락을 획득하기 위해 기다리고 있는 쓰레드에게 던져주는 공지와 같은 메세지 전달의 기능입니다. 텅 빈 크리티컬 섹션은 데이터 보호의 기능 없이 메세지 전달의 기능만을 사용합니다.

이 답의 나머지 부분에서는 텅 빈 크리티컬 섹션의 사용 예를 몇 가지 보여 드릴 겁니다만, 이 예제들은 백마법이나 흑마법이 아닌, “회색 마법”으로 여겨져야만 합니다.⁵ 텅 빈 크리티컬 섹션은 그 자체로는 실전에서는 거의 항상 사용되지 않습니다. 더도 아니고 덜도 아니고, 이 회색 영역 안으로 들어가 봅시다...

텅 빈 크리티컬 섹션이 사용된 역사적 사례 가운데 하나는 리눅스 커널 2.4 버전에서의 네트워킹 스택에서 발견됩니다. 이 사용 패턴은 Section 9.5에서 다뤄지게 될 *read-copy update (RCU)* 의 효과를 비슷하게 따라하기 위한 방법 중 하나라고 이해될 수 있겠습니다.

텅 빈 크리티컬 섹션이라는 관용적 코드는 일부 상황에서 락 경쟁정도를 줄이기 위해 사용될 수도 있습니다. 예를 들어, 멀티쓰레드로 동작하는 사용자 레벨 어플리케이션으로, 각 쓰레드가 전체 일 중에서 쓰레드별 리스트에 매달려서 관리되는 한 단위의 조각을 처리하게 되며, 쓰레드는 다른 쓰레드의 리스트를 건드릴 수 없는 경우를 생각해 봅시다. 이런 상황에서는 업데이트는 앞서 스케줄된 일의 조각들이 모두 완료된 후에 진행되어야 하는 업데이트가 존재할 겁니다. 이런 상황을 처리하는 한가지 방법은 일의 조각들을 각 쓰레드에 스케줄해서, 이 일의 조각들이 모두 완료되면, 업데이트가 진행되도록 하는 것입니다.

일부 어플리케이션에서는 쓰레드는 나타났다가 사라졌다가 할 수 있습니다. 예를 들어, 개별 쓰레드는 어플리케이션의 한 사용자에 연관될 수도 있고, 따라서 그 사용자가 로그아웃하거나 어떤 다른 방법으로 연결이 끊기거나 하게 되면 그 쓰레드 역시 사라질 겁니다. 많은 어플리케이션에서, 쓰레드는 어토믹하게 사라질 수 없습니다: 쓰레드는 사라지기 위해서는 쓰레드 자신 스스로를 특정한 일련의 동작들을 하는 어플리케이션의 다양한 영역들로부터 명시적으로 떨어져야만 합니다. 그런 동작들 중 한 예로는 다른 쓰레드들로부터 들어올

⁵ 이 설명을 위해 도움 준 Alexey Roytman에게 감사를 드립니다.

수 있는 이후의 작업 요청을 거절하는 것이 될 수 있겠고, 또 하나 더 그런 동작의 예를 들어보면 쓰레드 자신의 리스트에 아직 남아있는 일거리들을 처분하는 것이 되겠는데, 예를 들면, 이런 남아있는 일거리들을 글로벌한 일거리 처분 리스트에 집어넣어 다른 남아있는 쓰레드들 가운데 하나에 의해 처분되도록 할 수 있겠습니다. (남아있는 일들 각각을 그냥 처리함으로써 그 쓰레드의 일거리 리스트를 비워버리는 건 어떨까요? 주어진 일거리들은 또 다른 일거리를 만들어낼 수도 있기 때문에, 일거리 보관 리스트는 적당한 시간 내에 비워지지 못할 수도 있습니다.)

이런 어플리케이션이 좋은 확장성과 성능을 가지기 위해선, 좋은 락킹 설계가 필요합니다. 흔하게 사용되는 해결책 중 하나는 쓰레드의 정리 과정을 전부 보호하는 하나의 글로벌 락 (이걸 G라고 해보죠) 을 두고 개별적인 정리 동작들을 보호하는 더 잘게 조개진 크리티컬 섹션을 위한 락을 또 두는 것입니다.

이제, 사라지려 하는 쓰레드는 자신의 일거리 리스트에 있는 일거리들을 처분하기에 앞서서 향후의 작업 요청들을 분명하게 거절해야 하는데, 그러지 않으면 남아있는 일거리들을 처분한 이후에도 추가적인 일거리들이 들어닥쳐서 일거리 처분 과정이 무효가 되게 만들 것이기 때문입니다. 그런 점을 고려해 쓰레드가 사라지는 과정을 간략히 슈도코드로 표현해 보면 다음과 같을 것입니다:

1. 락 G를 잡습니다.
2. 커뮤니케이션을 보호하는 락을 잡습니다.
3. 다른 쓰레드로부터의 향후의 커뮤니케이션을 거절합니다.
4. 커뮤니케이션을 보호하는 락을 풀어줍니다.
5. 글로벌 일거리 처분 리스트를 보호하는 락을 잡습니다.
6. 해당 쓰레드의 일거리 리스트에 남아 있는 모든 일거리들을 글로벌한 처분될 일거리 리스트에 옮깁니다.
7. 글로벌한 처분될 일거리 리스트를 보호하는 락을 풀어줍니다.
8. 락 G를 풀어줍니다.

물론, 먼저 스케줄된 일거리들이 모두 처리되기를 기다리는 쓰레드는 사라지는 쓰레드 역시 염두해 두어야 합니다. 이 상황을 보기 위해서, 어떤 사라지려 하는 쓰레드가 다른 쓰레드들과의 야홍의 커뮤니케이션을 거절한 직후에 모든 앞서 스케줄된 일거리들이 완료되기를 기다리고 있는 쓰레드를 생각해 봅시다. 쓰레드들은

다른 쓰레드의 일거리 리스트에 접근할 수 없다는 점을 생각해보면, 이 쓰레드는 사라지려 하는 쓰레드의 일거리들이 완료되기를 어떻게 기다릴 수 있을까요?

직선적인 한가지 방법은 이 쓰레드가 G를 획득하고 글로벌한 처분될 일거리 리스트를 지키는 락을 잡고 나서, 그 리스트의 일거리들을 자신의 리스트로 가져오는 것입니다. 그리고 나서는 두 락을 모두 놓아주고, 자신의 일거리 리스트의 끝에 일거리 하나를 집어넣고, 각 쓰레드의 일거리 리스트 (자신의 것을 포함해) 안에 들어있는 일거리들이 완료되기를 기다리는 겁니다.

이 방법은 많은 경우에 동작합니다만, 글로벌한 처분될 일거리 리스트로부터 가져오는 각각의 일거리에 대해 가져올 때마다 특별한 처리가 필요하다면 G에의 경쟁도가 지나치게 될 겁니다. 그런 지나친 경쟁을 막기 위한 한가지 방법은 G를 획득한 직후에 놔버리는 것입니다. 그렇게 되면 앞서 스케줄된 모든 일거리들이 완료되길 기다리는 과정은 다음과 같이 될 것입니다:

1. 글로벌 카운터의 값을 1로 두고 조건 변수의 값을 0으로 초기화 시킵니다.
2. 모든 쓰레드들에 메세지를 보내서 쓰레드들이 어토믹하게 글로벌 카운터의 값을 증가시키고 일거리를 대기열에 집어넣게 합니다. 그 일거리는 어토믹하게 글로벌 카운터의 값을 감소시키고 그 감소된 결과값이 0이라면 조건 변수의 값을 1로 만듭니다.
3. G를 잡는데, 이로 인해 사라지려 하는 쓰레드가 있다면 사라지는 작업이 끝날 때까지 기다리게 될 것입니다. 한번에 하나의 쓰레드만 사라질 수 있기 때문에, 모든 남아있는 쓰레드들은 이미 앞의 단계에서 보낸 메세지를 받았을 것입니다.
4. G를 놓습니다.
5. 글로벌한 처분할 일거리 리스트를 보호하는 락을 잡습니다.
6. 글로벌한 처분할 일거리 리스트의 일거리들을 이 쓰레드의 리스트로 옮기고, 옮기는데 필요한 일이 있다면 합니다.
7. 글로벌한 처분할 일거리 리스트를 보호하는 락을 놓아줍니다.
8. 이 쓰레드의 리스트에 추가적으로 일거리 하나를 넣습니다. (앞에서와 마찬가지로, 이 일거리는 자동적으로 글로벌 카운터의 값을 어토믹하게 감소시킬 것이고, 그 결과가 0이라면 조건 변수의 값을 1로 만듭니다.)
9. 조건 변수가 1이 되기를 기다립니다.

이 과정이 완료되면, 모든 앞서 스케줄된 일거리들은 완료되어 있을 것이 보장됩니다. 이와 같이, 텅 빈 크리티컬 섹션들은 락킹을 데이터 보호만이 아니라 메세지 보내는 용도로 사용하는 것입니다. □

Quick Quiz 7.19:

VAX/VMS DLM ① reader-writer 락을 다른 방법으로 모방해낼 수도 있을까요? ■

Answer:

실제로 여러가지 방법이 있습니다. 한가지 방법은 null, protected-read, 그리고 exclusive 모드를 사용하는 것이겠습니다. 또 다른 방법은 null, protected-read, 그리고 concurrent-write 모드를 사용하는 것일 테고요. 세번째 방법은 null, concurrent-read, 그리고 exclusive 모드를 사용하는 것이겠지요. □

Quick Quiz 7.20:

Figure 7.15 의 코드는 황당할 정도로 복잡하군요! 왜 그냥 조건적으로 하나의 글로벌 락을 잡지 않나요? ■

Answer:

조건적으로 하나의 글로벌 락을 잡는 방법은 실제로 잘 동작합니다만, 비교적 작은 수의 CPU들을 사용할 때만 그렇습니다. 수백개의 CPU를 가진 시스템에서 이게 왜 문제가 될 수 있는지 보려면, Figure 5.3를 보시고 8개에서 1,000 개로 CPU들이 늘어나면 그 딜레이가 어떻게 될지 한번 생각해 보세요. □

Quick Quiz 7.21:

잠깐만요! Figure 7.15 의 line 16에서 토너먼트에 “승리” 했다면, do_force_quiescent_state() 의 모든 일을 해야만 하잖아요. 어떻게 이걸 승리라고 할 수 있어요, 진짜로? ■

Answer:

어떻게 그려나구요? 이건 그저 동시성에 있어서, 삶이 그러하듯이, 게임을 진행하기 전에 승리함이 의미하는 바가 정확히 무엇인지 알기 위해 주의를 기울여야 한다는 걸 보여줄 뿐입니다. □

Quick Quiz 7.22:

왜 C 언어에서 기본적으로 제공하는 0으로의 초기화 메커니즘을 사용하지 않고 Figure 7.16 의 line 2에 보여진 것처럼 명시적으로 초기화를 하는 거죠? ■

Answer:

이 기본적인 초기화 메커니즘은 한 함수의 범위 안에서 지정된 auto 변수로 할당된 락에 대해서는 적용되지 않을 것이기 때문입니다. □

Quick Quiz 7.23:

왜 Figure 7.16 의 line 7-8 의 안쪽 루프를 귀찮게 수행해야 하나요? 왜 그냥 간단하게 line 6에서 계속해서 어토믹한 값 교환 오퍼레이션을 수행하지 않는거죠? ■

Answer:

락이 잡혀 있고 여러 쓰레드들이 그 락을 잡으려 시도한다고 생각해 봅시다. 이런 상황에서, 이 쓰레드들이 모두 어토믹 값 교환 오퍼레이션을 하면서 루프를 반복한다면, 이들은 그 락을 포함하고 있는 캐시 라인을 쓰레드들 사이에서 계속 주고받을 것이고, 이는 곧 CPU 사이의 로드를 의미합니다. 반면에, 이 쓰레드들이 line 7-8의 안쪽 루프에서 루프를 돌게 되면 각자의 캐시만을 가지고 루프를 돌게 되어서 CPU 사이의 로드는 위협이 되지 않을 정도가 될 것입니다. □

Quick Quiz 7.24:

Figure 7.16 의 line 14에서는 왜 그냥 간단하게 락에 0 값을 저장하지 않는거죠? ■

Answer:

그렇게 하는 것도 문제 없는 구현이긴 합니다만, 그 저장 작업에 ACCESS_ONCE() 사용되고, 그 뒤에 메모리 배리어가 놓여질 때에만 그렇습니다. 메모리 배리어는 xchg() 오퍼레이션이 사용될 경우에는 필요치 않은데 이 오퍼레이션은 전의 값을 리턴해야 한다는 사실 때문에 자체적으로 전체 메모리 배리어를 내포하고 있기 때문입니다. □

Quick Quiz 7.25:

카운터의 값이 오버플로우 되어 초기화 되는 경우를 고려하면서 어떻게 하나의 카운터가 다른 카운터보다 큰지를 판단할 수 있나요? ■

Answer:

C 언어에서는 다음과 같은 macro 를 사용해서 이 문제를 정확하게 처리할 수 있습니다:

```
#define ULONG_CMP_LT(a, b) \
    (ULONG_MAX / 2 < (a) - (b))
```

부호를 갖는 정수들을 간단히 빼기 연산하는게 나을 것 같아 보일 수 있지만, C 언어에서 부호를 갖는 것들의 오버플로우에 대한 결과는 정의되어 있지 않기 때문에 그 방법은 피해져야만 합니다. 예를 들어, 만약 컴파일러가 하나의 값은 양수이고 다른 하나는 음수란 것을 안다면, 컴파일러는 단순히 양수가 음수보다 크다고 판단할 권리를 갖고 있는데, 양수에서 음수를 빼는 결과가 오버플로우를 일으키고 따라서 음수가 나올 수 있더라도 그러합니다.

컴파일러가 어떻게 두 숫자의 부호를 알 수 있을까요? 앞의 값 할당과 비교로부터 추론해 낼 수 있습니다.

이 경우, CPU 별 카운터들이 부호가 존재한다면, 컴파일러는 항상 그것들이 값을 증가시킨다는 것을 추론할 수 있고, 그렇게 되면 이 값들은 결코 음수가 되지 않는다고 가정할 수 있습니다. 이런 가정은 컴파일러가 유감스러운 코드를 생성하는 결과를 이끌어 낼 수 있을 겁니다 [McK12c, Reg10]. □

Quick Quiz 7.26:

카운터 사용과 플래그 사용 중 뭐가 더 나을까요? □

Answer:

플래그를 사용하는 방법은 일반적으로 더 적은 캐시 미스를 낼 겁니다만, 더 나은 방법은 두 가지 모두 시도해 보고 당장 적용해야 하는 특정한 워크로드에서 뭐가 더 잘 동작하는지 보는 것입니다. □

Quick Quiz 7.27:

어떻게 비명시적인 존재 보장에 의존하는게 버그를 만들어낼 수 있나요? □

Answer:

비명시적 존재 보장의 잘못된 사용에 의해 초래될 수 있는 일부 버그들이 여기 있습니다:

- 어떤 프로그램이 한 글로벌 변수의 주소를 한 파일에 쓰고, 그 후에 똑같은 프로그램의 새로운 실행 흐름이 그 주소를 읽고는 그 주소의 값을 읽어보려 시도합니다. 이는 그 프로그램의 기존 수행에서의 내용을 기반으로 다음 수행에서의 구성을 예측할 수 없게 하는 주소 공간 임의 추출 (address space randomization) 기법으로 인해 실패할 수 있습니다.
- 어떤 모듈은 자신의 변수들 가운데 하나의 주소를 다른 모듈에 위치해 있는 한 포인터에 기록해 두고는 그 포인터가 가리키는 값을 모듈이 내려간 (unloaded) 다음에 읽어보려 시도할 수 있습니다.
- 어떤 함수는 자신의 스택 위에 할당된 변수들 가운데 하나의 주소를 어떤 글로벌 포인터에 기록해 두고는 그 함수가 리턴한 후에 어떤 다른 함수가 그 포인터가 가리키는 값을 읽어보려 시도할 수 있습니다.

그 외에도 여러가지 추가적인 가능성성이 있을 수 있다고 확신합니다. □

Quick Quiz 7.28:

우리가 삭제해야 하는 원소가 Figure 7.17 의 line 8 의 리스트의 첫번째 원소가 아니면 어떻게 하죠? □

Answer:

이건 체이닝을 사용하지 않는 매우 간단한 해시 테이블이므로, 주어진 bucket 의 원소는 첫번째 원소 뿐입니다.

독자 여러분은 이 간단한 예제 해시 테이블에 체이닝을 적용해 보셔도 좋을 겁니다. □

Quick Quiz 7.29:

Figure 7.17에서 어떤 경쟁 조건이 발생할 수 있나요? □

Answer:

다음과 같은 일련의 이벤트들을 생각해 봅시다:

- Thread 0 이 `delete(0)` 을 호출하고, 그림의 line 10에 도달해서 락을 얻습니다.
- Thread 1이 동시에 `delete(0)` 를 호출하고, line 10에 도착합니다만 Thread 0이 이미 락을 잡고 있으니 그 락을 기다리며 루프를 돋니다.
- thread 0이 line 11-14를 수행하고 해시테이블에서 그 원소를 삭제하고 락을 해제하고, 그 원소를 메모리 해제시킵니다.
- Thread 0는 수행을 계속해서 메모리를 할당받는데 이 때 정확히 방금 해제한 메모리 블락을 받습니다.
- Thread 0는 이제 이 메모리 블락을 어떤 다른 타입의 구조체로 초기화 시킵니다.
- Thread 1의 `spin_lock()` 은 스피너이라 생각했던 `p->lock` 이 더이상 스피너가 아니기 때문에 실패합니다.

존재에 대한 보장이 없기 때문에 해당 데이터 원소의 정체성은 어떤 쓰레드가 line 10에서 그 원소의 락을 얻기 위해 시도중인 상황 중에도 바뀔 수가 있는 겁니다! □

D.8 Data Ownership

Quick Quiz 8.1:

어떤 형태의 데이터 소유권이 C나 C++를 사용해서 (예를 들어, pthreads를 사용해서) 공유 메모리 병렬 프로그램을 만들 때 방지하기가 극단적으로 어려울까요? □

Answer:

함수 안에서의 `auto` 변수들의 사용입니다. 기본적으로, 이것들은 현재 함수를 수행하고 있는 쓰레드에 소유됩니다. □

Quick Quiz 8.2:

Section 8.1에서 보인 예제에 남아있는 동기화 작업은 무엇이 있을까요? □

Answer:

sh 의 & 오퍼레이터를 통한 쓰레드들의 생성과 sh 의 wait 명령을 통한 쓰레드 종료 기다리기입니다.

물론, 프로세스들이 예를 들어 `shmget()`이나 `mmap()` 시스템 콜을 이용해 명시적으로 메모리를 공유한다면 이 공유 메모리 영역을 접근하거나 업데이트하는 데에는 명시적인 동기화가 필요할 것입니다. 프로세스들은 또한 다음의 프로세스간 통신 메커니즘들 중 어떤 것이든 사용할 수도 있을 겁니다.

1. System V 세마포어.
2. System V 메세지 큐.
3. UNIX-domain 소켓.
4. TCP/IP, UDP, 그리고 모든 다른 것들의 호스트를 포함한 네트워킹 프로토콜.
5. 파일 락킹.
6. O_CREAT 과 O_EXCL 플래그와 함께 사용되는 `open()` 시스템콜.
7. `rename()` 시스템콜의 사용.

가능한 동기화 메커니즘의 전체 리스트를 만드는 것은 독자의 몫으로 둘텐데, 이는 굉장히 긴 리스트가 될 것입니다. 놀랍도록 많은 수의 예상치 못한 시스템콜들이 동기화 메커니즘으로 나타날 수 있습니다. □

Quick Quiz 8.3:

Section 8.1에서 보여진 예제에 어떤 공유된 데이터가 있나요? ■

Answer:

이건 철학적인 질문입니다.

“아니오”라는 답을 원하는 사람들은 프로세스들이 정의에 의해 메모리를 공유하지 않는다고 주장할 수도 있을 겁니다.

“예”라고 대답하고 싶은 사람들은 공유 메모리를 필요로 하지 않는 많은 동기화 메커니즘들의 리스트를 나열하고 커널은 공유된 상태를 가질 것임을 이야기 할 수 있고, 아마도 심지어는 프로세스 ID (PID) 의 할당은 공유된 데이터에 해당된다고 주장할 수도 있습니다.

그런 주장들은 매우 지적인 행위이고 불행한 학교 친구나 동료들에게서 점수를 딸 수 있는, 지적이라는 느낌을 받을 수 있는 훌륭한 방법이기도 하며, (특히!) 유용한 일이 이루어지기를 막아버리는 행위입니다. □

Quick Quiz 8.4:

각각의 쓰레드가 자신의 쓰레드별 변수만을 읽을 수 있지만 다른 쓰레드의 인스턴스들에는 쓰기를 할 수도 있는 부분적 데이터 소유권도 말이 될까요? ■

Answer:

놀랍게도, 그렇습니다. 한가지 예는 쓰레드들이 다른 쓰레드의 메일함에 메세지를 보내고, 각 쓰레드는 자신이 보낸 메세지를 일단 그 메세지가 읽혀졌다면 지워야 할 책임을 갖는 간단한 메세지 전달 시스템입니다. 그런 알고리즘의 구현은 다른 알고리즘들을 비슷한 소유권 패턴으로 구분해 보는 일과 함께 독자의 몫으로 남겨두겠습니다. □

Quick Quiz 8.5:

함수를 전달하는데 POSIX 시그널 외에 어떤 다른 메커니즘이 사용될 수 있을까요? ■

Answer:

그런 메커니즘에는 상당히 많은 것들이 있는데, 다음의 것들을 포함합니다:

1. System V 메세지 큐.
2. 공유 메모리 디큐 (Section 6.1.2를 참고하세요).
3. 공유 메모리 메일함.
4. UNIX-domain 소켓.
5. RPC, HTTP, XML, SOAP, 그 외 여러가지 높은 레벨의 프로토콜들과 결합되어 사용될 수도 있는 TCP/IP 또는 UDP.

전체 리스트의 완성은 성실한 독자들의 몫으로 남겨둘 텐데, 그 리스트는 굉장히 길 수 있다는 점을 경고해 듭니다. □

Quick Quiz 8.6:

하지만 Figure 5.8의 line 15-32에 보여진 `eventual()` 함수의 어떤 데이터로 실제로는 `eventual()` 쓰레드에게 소유되어 있지 않습니다! 이게 어떻게 데이터 소유권인가요??? ■

Answer:

핵심 문구는 “데이터로의 권한을 소유한다”입니다. 이 경우, 질문되는 권한은 해당 그림의 line 1에 정의된 쓰레드별 `counter` 변수에 접근할 수 있는 권한을 말합니다. 이 상황은 Section 8.2에서 설명된 것과 비슷한 상황입니다.

하지만, 정말로 `eventual()` 쓰레드에 의해 소유되는 데이터도 존재하는데, 해당 그림의 line 17과 18에 정의되어 있는 `t`와 `sum` 변수들입니다.

정해진 쓰레드의 다른 예를 보려면, 리눅스 커널의 커널 쓰레드들을 보면 되는데, 예컨대 `kthread_create()` 와 `kthread_run()` 으로 생성되는 것들을 보시기 바랍니다. □

Quick Quiz 8.7:

여전히 쓰레드별 데이터의 완전한 사유화를 유지하면서 좋은 정확도를 얻을 수도 있을까요? ■

Answer:

네. 한가지 방법은 `read_count()` 가 자신의 쓰레드별 변수를 더하도록 하는 것입니다. 이렇게 되면 완전한 소유권과 성능을 유지하지만 정확도에 있어서는 아주 조금 개선이 될텐데, 특히나 매우 많은 수의 쓰레드들이 존재하는 시스템에서 그럴 겁니다.

또다른 방법은 `read_count()` 가 함수 전달을 사용하는 것으로, 예를 들면 쓰레드별 시그널과 같은 형태가 되겠습니다. 이 방법은 정확도를 매우 많이 개선시킬 것입니다만, `read_count()` 의 성능에 대해선 상당한 비용을 갖게 될겁니다.

하지만, 이런 방법들 모두 카운터 업데이트의 일반적인 경우에 있어서의 캐시 라인 바운싱 (cache-line bouncing) 을 제거해주는 이점을 가질 겁니다. □

D.9 Deferred Processing

Quick Quiz 9.1:

해제 후 사용 체크를 왜 신경쓰죠? ■

Answer:

버그 발견 가능성을 크게 늘리기 위해서입니다. 하나님의 타입의 구조체를 할당받고 해제할 뿐인 작은 `torture-test` 프로그램 (`routetorture.h`) 은 놀랍도록 많은 양의 해제 후 사용 실수들을 제어할 수 있습니다. Page 196 의 Figure 11.4 를 한번 보시고, 버그를 찾을 확률을 높이는 것의 중요성을 더 알기 위해서 Page 198 부터 시작하는 Section 11.6.4 에서의 토론을 살펴보시기 바랍니다. □

Quick Quiz 9.2:

Figure 9.4 의 `route_del()` 은 해제될 원소를 찾는 과정을 보호하기 위해 레퍼런스 카운트를 사용하지 않는 이유가 뭐죠? ■

Answer:

그 과정은 이미 락으로 보호되고 있기 때문에, 추가적인 보호는 필요하지 않기 때문입니다. □

Quick Quiz 9.3:

Figure 9.5 에서 “ideal” 선은 왜 똑바로 증가하지 않고 계단 형태로 증가하죠? ■

Answer:

하이퍼쓰레딩 때문입니다. 이 시스템에서, 코어 위에서의 하드웨어 쓰레드들은 연속되는 CPU 번호를 갖습니다. 또한, 이 특정한 포인터를 따라가는, 낮은 캐시 미스 확률의 워크로드는 하나의 하드웨어 쓰레드가 자신의 코어의 거의 모든 자원을 소모할 수 있도록 하는 것으로 보입니다. 더 많은 계산 작업을 갖는 워크로드는 각 코어의 두번째 하드웨어 쓰레드로부터 더 많은 이득을 얻을 것이라 예상할 수 있을 겁니다. □

Quick Quiz 9.4:

요즘과 같은 시대에, Figure 9.5 는 왜 겨우 8 CPU 까지만 사용하는 거죠??? ■

Answer:

레퍼런스 카운팅의 끔찍한 확장성을 고려해보면, 누가 8개보다 많은 CPU 를 필요로 하겠어요? CPU 네개까지만 해도 요점을 이야기 하는데 충분합니다! 하지만, 더 많은 CPU 를 원하는 분들은 Chapter 10 를 참고하시기 바랍니다. □

Quick Quiz 9.5:

동시성이 “분명히 레퍼런스 카운팅의 유용성을 저하시켰다”면, 리눅스 커널은 왜 그렇게 레퍼런스 카운터를 많이 사용하는거죠? ■

Answer:

그 문장은 “유용성을 저하시켰다”고 했지, “유용성을 없앴다”고는 하지 않았습니다, 그렇죠?

리눅스 커널이 레퍼런스 카운팅의 장점을 높은 동시성이 존재하는 환경에서 가져오기 위해 사용하는 방법들을 이야기하는 Section 13.2 를 보기 바랍니다. □

Quick Quiz 9.6:

Figure 9.6 의 `hp_store()` 는 왜 데이터 원소로의 접근을 두번이나 간접적으로 하는거죠? 왜 `void *` 가 아니라 `void **` 인 건가요? ■

Answer:

`hp_record()` 는 동시의 수정에 대해서 체크를 해봐야 하기 때문입니다. 이 일을 하기 위해서는 그 원소로의 포인터로의 포인터가 필요한데, 그게 있으면 해당 원소로의 포인터로의 수정이 있었는지 검사할 수 있기 때문입니다. □

Quick Quiz 9.7:

`hp_store()` 의 호출자는 실패했을 때 왜 데이터 접근을 처음부터 다시 시작해야 하는거죠? 데이터 구조체가 매우 크다면 좀 비효율적이지 않나요? ■

Answer:

어떻게 보면 좀 비효율적일 수도 있겠습니다만 분명한

사실은 정확성을 위해 그런 처음부터의 재시작이 반드시 필요하다는 점입니다. 이를 확실히 보기 위해, 원소 A, B, 그리고 C 를 가지고 있는 해저드 포인터로 보호되는 링크드 리스트가 다음의 일련의 이벤트들을 받는다고 생각해 봅시다:

1. 쓰레드 0 가 원소 B 로의 해저드 포인터를 저장합니다 (아마도 원소 A 를 거쳐 원소 B 를 찾아왔을 겁니다).
2. 쓰레드 1 이 리스트로부터 원소 B 를 삭제하는데, 이를 위해 원소 B 에서 원소 C 로의 포인터를 이 삭제 작업을 표시하기 위해 특수한 HAZPTR_POISON 값으로 설정합니다. Thread 0 는 원소 B 로의 해저드 포인터를 가지고 있으므로, 아직 정리 될 수 없습니다.
3. 쓰레드 1 이 리스트에서 원소 C 를 삭제합니다. 원소 C 를 레퍼런스 하는 해저드 포인터들이 존재하지 않으므로, 원소 C 는 곧바로 메모리 해제될 수 있습니다.
4. 쓰레드 0 이 이제 없어진 원소 B 의 다음 원소로의 해저드 포인터를 얻으려 합니다만, HAZPTR_POISON 값을 보게 되고, 따라서 0 을 리턴해서, 호출자가 리스트의 시작점부터 탐색을 다시 시작하도록 강제합니다.

따라서 처음부터 다시 탐색하는게 좋은 방법인데, 이렇게 하지 않는다면 쓰레드 0 는 이제는 제거된 원소 C 에 액세스를 시도할 수 있는데, 이는 임의의 끔찍한 메모리 오염을 일으키는 결과를 만들어낼 수 있는데, 특히나 원소 C 를 위해 사용되던 메모리가 어떤 다른 목적으로 재할당 되었다면 특히 그럴 것이기 때문입니다.

그와는 별개로, 해저드 포인터의 재시작은 최소한의 메모리 사용량을 유지할 수 있도록 함을 이해하기 바랍니다. 현재 해저드 포인터로 레퍼런스 되고 있지 않은 오브젝트는 곧바로 해제될 수 있습니다. 대조적으로, Section 9.5 에서는 read-side 재시도를 방지하지만 (read-side 오버헤드도 최소화 시킵니다), 훨씬 큰 메모리 사용량을 갖는 메커니즘을 이야기할 겁니다. □

Quick Quiz 9.8:

해저드 포인터들에 대한 논문들은 각각의 포인터의 아래 비트들을 지원하는 원소들을 마크하기 위해 사용한다고 하는데, HAZPTR_POISON 은 뭔가요? ■

Answer:

출간된 논문의 해저드 포인터 구현들은 그 삽입과 삭제를 위해 non-blocking 동기화 기법들을 사용했습니다.

이런 기법들은 데이터 구조체를 가로지르며 읽기를 하는 쓰레드들이 업데이트를 하는 쓰레드들이 그들의 업데이트를 완료하도록 “도움”을 줄 것을 필요로 하는데, 이 말은 읽기를 하는 쓰레드들은 삭제된 원소의 다음 원소를 봐야만 한다는 뜻입니다.

반면에, 우리는 업데이트 작업들을 동기화 시키는데 락킹을 사용할 것인데, 이렇게 되면 읽기를 하는 쓰레드들이 업데이트를 하는 쓰레드들이 그들의 업데이트를 완료할 수 있도록 돋는 일을 해줄 필요가 없어져서 포인터들의 아래쪽 비트들을 놔둘 수 있게 해줍니다. 이 방법은 읽기를 하는쪽 코드를 좀 더 간단하고 빠르게 해줍니다. □

Quick Quiz 9.9:

하지만 해저드 포인터들에 있는 이런 제약사항들은 다른 형태의 레퍼런스 카운팅에도 똑같이 적용되는 거 아닌가요? ■

Answer:

이런 제약사항은 레퍼런스 획득이 실패할 수 있는 레퍼런스 카운팅 메커니즘들에만 적용됩니다. □

Quick Quiz 9.10:

논문 “Structured Deferral: Synchronization via Procrastination” [McK13] 는 해저드 포인터가 이상적인 경우에 가까운 성능을 보인다는 걸 보였습니다. Figure 9.9 에선 무슨 일이 일어난거죠??? ■

Answer:

첫째로, Figure 9.9 는 1 차원적 y-축을 갖는데 반해 “Structured Deferral” 논문에서의 그래프는 로그스케일 y-축을 갖습니다. 다음으로, 그 논문은 가벼운 일을 하는 해시 테이블을 사용했고, Figure 9.9 의 수행은 10 개 원소의 간단한 링크드 리스트를 사용했는데, 이는, 해저드 포인터가 이 워크로드에서 “Structured Deferral” 논문에서의 것보다 더 커다란 메모리 배리어 페널티를 받았다는 뜻입니다. 마지막으로, 그 논문은 더 커다랗고 오래된 x86 시스템을 쓴 반면, Figure 9.9 의 결과를 생성하는데 사용된 시스템은 더 신형이고 더 작은 시스템이었습니다.

항상 그렇듯, 비용은 경우에 따라 다를 수 있습니다. 이 성능의 차이를 놓고 보면, 해저드 포인터가 (메모리 배리어 오버헤드가 최소한 캐시 미스 페널티와 겹칠 만큼) 매우 커다란 데이터 구조체에서나 해시 테이블과 같이 탐색 작업이 최소한의 해저드 포인터만을 필요로 하는 경우에는 가장 이상적인 성능을 줄 것임이 분명합니다. □

Quick Quiz 9.11:

왜 이 시퀀스 락에 대한 토론은 Chapter 7 에서, *locking* 의 하나로써 다루어지지 않았던 거죠? ■

Answer:

시퀀스 락 메커니즘은 실제로는 시퀀스 카운트와 락킹이라는 두개의 별개의 동기화 메커니즘들의 조합입니다. 사실, 시퀀스 카운트 메커니즘은 리눅스 커널에서 `write_seqcount_begin()` 과 `write_seqcount_end()` 기능들을 사용해 별개로 사용될 수도 있습니다.

하지만, 조합된 `write_seqlock()` 과 `write_sequnlock()` 기능들은 리눅스 커널에서 훨씬 많이 사용됩니다. 더 중요한 건, 더 많은 사람들이 “시퀀스 카운트”라고 말하는 것보다는 “시퀀스 락”이라고 말하는 걸 더 쉽게 이해할 것이라는 점입니다.

그래서 이 섹션은 사람들이 제목으로부터 이게 뭐에 대한 것인지를 이해할 수 있도록 “Sequence Locks”라고 이름지어졌고, “Deferred Processing”으로 표현한 이유는 (1) “시퀀스 락”에서 “시퀀스 카운트”의 강조를 위함과 (2) “시퀀스 락”은 단순한 락보다는 더 많은 의미를 갖기 때문입니다. □

Quick Quiz 9.12:

Figure 9.13 의 `read_seqbegin()` 은 왜 내부적으로 가장 낮은 자리의 비트를 검사하고 재시도를 하지 않고 어차피 망할 읽기를 시작하는 건가요? ■

Answer:

그렇게 하는 것도 합법적인 구현이 될겁니다. 하지만, 워크로드에 읽기가 대부분이라면, 일반적인 경우의 성공적인 읽기의 오버헤드가 증가할 것인데, 이는 생산적이지 못한 일입니다. 하지만, 충분히 많은 업데이트가 존재하고 충분히 높은 읽기의 오버헤드가 존재한다면, 그런 검사를 `read_seqbegin()` 내부에서 하는 것도 괜찮을 겁니다. □

Quick Quiz 9.13:

Figure 9.13 의 line 26 에서의 `smp_mb()` 는 왜 필요한 건가요? ■

Answer:

만약 그게 없다면, 컴파일러 또는 CPU 가 이 `read_seqretry()` 함수 호출 전의 크리티컬 섹션을 이 함수 뒤로 옮겨버릴 수도 있기 때문입니다. 이렇게 되면 시퀀스 락이 크리티컬 섹션을 보호하지 못하게 만들어 버릴 겁니다. `smp_mb()` 기능은 그런 재배치를 막아줍니다. □

Quick Quiz 9.14:

Figure 9.13 의 코드는 완화된 형태의 메모리 배리어를 사용할 수는 없을까요? ■

Answer:

리눅스 커널의 오래된 버전들에서라면, 안됩니다.

최신의 리눅스 커널에서는, line 16 은 `ACCESS_ONCE()` 대신에 `smp_load_acquire()` 를 사용할 수 있고, 그렇게 되면 line 17에서의 `smp_mb()` 는 없어질 수 있게 될겁니다. 비슷하게, line 41 은 `smp_store_release()` 를 사용할 수 있는데, 예를 들면 다음과 같은 겁니다:

`smp_store_release(&slp->seq, ACCESS_ONCE(slp->seq) + 1);`

이는 line 40 의 `smp_mb()` 를 없앨 수 있게 해줄 겁니다. □

Quick Quiz 9.15:

시퀀스 락킹 아래서, 업데이트 쓰레드들이 읽기 쓰레드들이 진행 못하게 하는 걸 막는 건 무엇일까요? ■

Answer:

그런 건 없습니다. 이것은 시퀀스 락킹의 약점들 가운데 하나이고, 그 결과로, 시퀀스 락킹은 읽기가 대부분인 환경에서만 사용되어야 합니다. 만약 읽기를 하는 쪽이 스타베이션에 빠지는 것도 괜찮은 상황이라면, 시퀀스 락킹 업데이트들을 가지고 더 폭넓게 사용해도 좋을 겁니다! □

Quick Quiz 9.16:

다른 뭔가가 쓰기 쓰레드들을 직렬화 시켜서 락이 필요치 않다면 어떻게 되죠? ■

Answer:

그런 경우라면, `->lock` 필드는, 리눅스 커널의 `seqcount_t` 가 그렇듯이, 사라질 수 있을 겁니다. □

Quick Quiz 9.17:

Figure 9.13 의 line 2 의 `seq` 는 왜 `unsigned` 가 아니라 `unsigned long` 인가요? 무엇보다, `unsigned` 가 리눅스 커널에서 충분히 좋은 것이라면 모두에게도 충분히 좋지 않을까요? ■

Answer:

전혀 그렇지 않습니다. 리눅스 커널은 다음과 같은 일련의 이벤트들을 무시하는 것을 가능하게 하는 특별한 속성들을 가지고 있습니다:

1. Thread 0 가 `read_seqbegin()` 을 실행하고, line 16에서 `->seq` 를 얻어오는데, 그 값은 짹수여서, 호출자에게 리턴합니다.
2. Thread 0 가 자신의 `read-side` 크리티컬 섹션을 실행하기 시작합니다만, 곧 오랫동안 cpu 를 빼앗깁니다.

3. 다른 쓰레드들이 반복적으로 `write_seqlock()` 과 `write_sequnlock()` 을 호출하는데, `->seq` 의 값이 오버플로우 되어서 Thread 0 가 얻어온 값과 같아질 때까지 반복합니다.
4. Thread 0 가 실행을 재개하고, 자신의 read-side 크리티컬 섹션을 이제 비일관적인 데이터와 함께 마무리짓습니다.
5. Thread 0 는 `read_seqretry()` 를 호출하는데, 부정확하게도 Thread 0 가 이 시퀀스 락으로 보호되는 데이터의 일관적인 모습만 보았다고 결론내리게 됩니다.

리눅스 커널은 매우 드물게 업데이트 되는 것들에 대해서만 시퀀스 락킹을 사용하는데, 하루의 시각 정보가 그런 경우입니다. 이 정보는 아무리 잡아도 1 밀리세컨드에 한번 업데이트 되며, 따라서 카운터를 오버플로우시키는데에는 7 주일이 필요할 겁니다. 만약 어떤 커널 쓰레드가 7 주일간 cpu 를 빼앗겼다면, 이 리눅스 커널의 soft-lockup 코드가 그동안 2분에 한번씩 경고를 띠웠을 겁니다.

반면, 64-bit 카운터를 사용한다면 설령 업데이트가 매 나노세컨드마다 일어난다고 해도 오버플로우를 하는데 500년 이상이 필요합니다. 따라서, 이 구현은 `->seq` 의 타입에 64 비트 시스템에서는 64 비트인 타입을 사용합니다. □

Quick Quiz 9.18:

이 버그는 고쳐질 수 있을까요? 달리 말해서, 동시에 삽입, 삭제, 탐색을 지원하는 링크드 리스트를 보호하는 동기화 메커니즘으로 시퀀스락 하나만 사용할 수 있을까요? ■

Answer:

이를 가능하게 하는 한가지 방법은 read-only 액세스를 포함해 모든 액세스들을 `write_seqlock()` 과 `write_sequnlock()` 으로 감싸버리는 것입니다. 물론, 이 방법은 모든 read-side 병렬성을 없애버릴 것이고, 커다란 락 컨텐션을 초래하며, 더 나아가서 간단한 락킹처럼 쉽게 구현될 수 있을 겁니다.

Read-side 액세스를 보호하는데 `read_seqbegin()` 과 `read_seqretry()` 를 사용하는 방법을 찾아낸다면, 그 방법이 다음의 이벤트 시퀀스도 잘 처리할 수 있는지 확인해 보시기 바랍니다:

1. CPU 0 가 링크드 리스트를 순회하고, 리스트 원소 A 로의 포인터를 집어냅니다.
2. CPU 1 이 원소 A 를 리스트에서 제거하고 해제합니다.

3. CPU 2 가 관계없는 데이터 구조체를 하나 할당하는데, 이 과정에서 원소 A 가 사용하고 있던 메모리를 받아오게 됩니다. 원소 A 의 `->next` 포인터를 저장하는데 사용되던 영역은 이제 이 관계없는 데이터 구조 내에서 floating-point 수를 저장하는데 사용됩니다.
4. CPU 0 가 원소 A 의 `->next` 포인터였던 것을 보게 되는데, 무작위적인 비트를 보게 되고, 따라서 segmentation fault 가 날 겁니다.

이런 종류의 문제를 해결하기 위해서는 “타입-안전 메모리”가 필요한데, 이에 대해서는 Section 9.5.3.7 에서 다루도록 하겠습니다. 하지만 이 경우, 시퀀스 락에 대해서 또 다른 동기화 메커니즘을 사용해야 할 겁니다!

□

Quick Quiz 9.19:

하지만 Section 9.4 의 시퀀스 락 역시 읽기 쓰레드들과 업데이트 쓰레드들이 동시에 일을 할 수 있도록 하지 않던가요? ■

Answer:

맞는 말이고 아닌 말이기도 합니다. 시퀀스 락의 읽기 쓰레드들은 쓰기 쓰레드들과 동시에 수행될 수 있지만, 이런 상황이 발생할 때마다, `read_seqretry()` 기능이 읽기 쓰레드는 다시 수행을 시도하도록 강제할 겁니다. 이는 시퀀스 락을 사용하명 업데이트 쓰레드와 동시에 수행되는 읽기 쓰레드가 하는 일은 모두 취소되고 다시 수행될 것을 의미합니다. 따라서 시퀀스 락을 사용하는 읽기 작업은 업데이트 작업과 동시에 수행될 수 있지만, 이런 경우에 실제로는 어떤 일도 만들어내지 못합니다.

반면에, RCU 읽기 쓰레드들은 동시에 RCU 업데이트 쓰레드들이 존재하더라도 의미있는 일을 처리해낼 수 있습니다. □

Quick Quiz 9.20:

`list_add_rcu()` 와 정확히 똑같은 시간에 `list_for_each_entry_rcu()` 이 수행되면 segfault 가 날 수 있을 것 같은데, 이걸 무엇이 방지해 주나요? ■

Answer:

리눅스가 돌아가는 모든 시스템에서 포인터로의 로드와 스토어는 모두 어토믹한데, 즉, 포인터로의 스토어가 같은 포인터로부터의 로드와 같은 시점에 일어난다면, 이 로드는 초기값이나 저장된 값을 가져오지, 그 두 값이 섞여진 값을 가져오는 일은 결코 없습니다. 또한,

`list_for_each_entry_rcu()` 는 항상 리스트를 앞으로만 움직이지, 결코 뒤로 움직이지는 않습니다. 따라서, `list_for_each_entry_rcu()` 는 `list_add_rcu()` 를 통해 들어온 원소를 보거나 보지 않을 뿐이지 만 어떤 경우든 적합하게 잘 형성된 리스트를 볼 겁니다. □

Quick Quiz 9.21:

리스트의 버전이 두개보다 많을 수 있도록 하기 위해서는 삭제 예제를 어떻게 수정해야 할까요? ■

Answer:

이를 가능하게 하는 한가지 방법은 Figure D.5 에 보인 것처럼 하는 것입니다.

```

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

이 코드는 여러개의 동시에 수행되는 삭제 작업들이 `synchronize_rcu()` 에서 기다리게 될수도 있음을 의미합니다. □

Quick Quiz 9.22:

어떤 시점에 하나의 리스트는 RCU 버전들을 몇개까지 가질 수 있을까요? ■

Answer:

동기화 설계에 따라 달라집니다. 업데이트를 보호하는 세마포어가 grace period 에 걸쳐 잡혀 있다면, 옛날 버전과 새로운 버전, 최대 두개의 버전이 있을 수 있을 겁니다.

하지만, 검색, 업데이트, 그리고 `list_replace_rcu()` 만이 락으로 보호되고 있어서 Figure D.5 보인 코드에서처럼 `synchronize_rcu()` 가 락의 바깥에 있다고 생각해 봅시다. 더 나아가서 수많은 쓰레드들이 거의 동시에 RCU 교체 기능을 수행했고, 이 읽기 쓰레드들은 또한 데이터 구조를 계속해서 지나다니고 있다고 생각해 봅시다.

그러면 다음과 같은 일련의 이벤트들이 Figure 9.29 의 마지막 상태로부터 발생할 수 있습니다.

- 쓰레드 A 가 리스트를 지나가면서 5,2,3 원소로의 레퍼런스를 얻습니다.
- 쓰레드 B 가 5,2,3 원소를 새로운 5,2,4 원소로 교체하고, 자신의 `synchronize_rcu()` 호출이 리턴하길 기다립니다.
- 쓰레드 C 가 리스트를 지나가면서 5,2,4 원소로의 레퍼런스를 얻습니다.
- 쓰레드 D 가 5,2,4 원소를 새로운 5,2,5 원소로 교체하고, 자신의 `synchronize_rcu()` 호출이 리턴하길 기다립니다.
- 쓰레드 E 가 리스트를 지나가면서 5,2,5 원소로의 레퍼런스를 얻습니다.
- 쓰레드 F 가 5,2,5 원소를 새로운 5,2,6 원소로 교체하고, 자신의 `synchronize_rcu()` 호출이 리턴하길 기다립니다.
- 쓰레드 G 가 리스트를 지나가면서 5,2,6 원소로의 레퍼런스를 얻습니다.
- 그리고 앞의 두 스텝들이 빠르게 계속해서 반복되어서 모든 쓰레드들이 `synchronize_rcu()` 호출이 리턴하길 기다립니다.

따라서, 얼마든지 많은 버전들이 존재할 수 있는데, 다만 메모리 크기와 얼마나 많은 업데이트들이 하나의 grace period 안에 완료될 수 있느냐에 그 수가 제한됩니다. 하지만 그렇게 자주 업데이트 되는 데이터 구조체들은 RCU 를 사용하기에 적합한 후보가 아닐 수 있음을 알아두시기 바랍니다. 그렇다면 해도, RCU 는 필요할 때에는 높은 비율의 업데이트를 처리할 수 있습니다. □

Quick Quiz 9.23:

`rcu_read_lock()` 과 `rcu_read_unlock()` 함수는 스펜하지도 블락하지도 않는데 어떻게 RCU 업데이트 쓰레드들이 RCU 읽기 쓰레드들을 대기시킬 수가 있나요? ■

Answer:

특정 RCU 업데이트에 의해서 가해진 수정사항은 연관된 CPU 가 해당 데이터를 포함하는 캐시 라인들을 무효화 하도록 만들 것이고, 동시에 수행중인 RCU 읽기 쓰레드들을 수행중인 해당 CPU 들이 비싼 캐시 미스를 마주하도록 만들 것입니다. (동시에 수행중인 읽기 쓰레드들에게 비싼 캐시 미스를 만들지 않고 데이터 구조에 변경을 가할 수 있는 알고리즘을 설계할 수 있을까요? 또는 뒤따르는 읽기 쓰레드들에게?) □

Quick Quiz 9.24:

RCU QSBR 은 왜 이상적 결과와 동일한 결과를 보이지 않는 거죠? ■

Answer:

`rcu_dereference()` 도구는 컴파일러의 최적화를 조금 제약할 수 있어서, 아주 약간 느린 코드를 만들어 낼 수 있습니다. 이 영향은 일반적으로는 별로 심각하지 않을 것입니다만, 각각의 탐색이 약 13 나노세컨드를 더 필요로 하게 되는데, 이는 코드 생성의 작은 차이가 느껴지지도 못할 만큼 충분히 짧은 시간입니다. 이 차이는 약 1.5% 부터 약 11.1% 까지 나오는데, RCU QSBR 코드가 “이상적인” 코드는 할 수 없는 동시의 업데이트도 처리할 수 있다는 점을 생각하면 상당히 작은 차이입니다.

C11 `memory_order_consume` 로드 [Smi15] 가 언젠가는 `rcu_dereference()` 에게 더 낮은 비용에 필요한 보호를 해주는 날이 오기를 기대해 볼만 합니다. □

Quick Quiz 9.25:

RCU QSBR 의 read-side 성능이 이렇게 좋은데, 왜 다른 종류의 userspace RCU 를 신경써야 하죠? ■

Answer:

RCU QSBR 은 경우에 따라서는 불가능한 제약을 어플리케이션에 강제하는데, 예를 들어 어플리케이션 내의 각각의 모든 쓰레드가 quiescent state 를 정규적으로 통과해야 하는 식입니다. 무엇보다도, 이는 RCU QSBR 이 라이브러리를 만드는 사람에게는 도움이 될 수 없음을 의미하는데, 그런 사람들은 다른 종류의 userspace RCU 를 사용하는게 좋을 겁니다 [MDJ13c]. □

Quick Quiz 9.26:

이게 뭐죠? 3GHz 에서의 클락 시간이 300 피코세컨드 가 넘는데 대체 어떻게 RCU 는 100 펜토세컨드의 오버헤드를 갖는다고 제가 믿을 수 있을 거라고 생각하세요? ■

Answer:

먼저, 이 측정을 위해 사용된 내부의 루프가 다음과 같다고 생각해 보세요:

```
1 for (i = 0; i < CSCOUNT_SCALE; i++) {
2     rCU_read_lock();
3     rCU_read_unlock();
4 }
```

다음으로, `rcu_read_lock()` 과 `rcu_read_unlock()` 이 실질적으로 수행하는 코드가 다음과 같다고 생각해 봅시다:

```
1 #define rCU_read_lock()    do { } while (0)
2 #define rCU_read_unlock()  do { } while (0)
```

그리고 컴파일러가 이 루프를 다음과 같이 바꿔게 하는 간단한 최적화를 한다고 생각해 보세요:

```
i = CSCOUNT_SCALE;
```

따라서 100 펜토세컨드의 “측정” 은 단순히 고정된 시간 측정 오버헤드를 `rcu_read_lock()` 과 `rcu_read_unlock()` 호출을 담고 있는 루프의 수행 횟수로 나눈 값일 뿐입니다. 그리고 따라서, 이 측정은 실제로 어려인데, 실제로 수십수백배의 어려울입니다. 앞의 `rcu_read_lock()` 와 `rcu_read_unlock()` 의 정의에서 알 수 있듯이, 실제 오버헤드는 정확히 제로입니다.

100 펜토세컨드의 시간 측정이 과하게 추측된 걸로 판명되는 건 모든 케이스가 아닐 것은 분명합니다! □

Quick Quiz 9.27:

크리티컬 섹션 오버헤드가 늘어나면 왜 `rwlock` 의 오버헤드와 가변성이 모두 줄어드는 거죠? ■

Answer:

크리티컬 섹션 오버헤드가 증가하면 그 아래에 위치한 `rwlock_t` 를 얻기 위한 경쟁이 줄어들기 때문입니다. 하지만, 이 `rwlock` 오버헤드는 단일 CPU 에서는 캐시 쓰래싱 오버헤드 때문에 그렇게 크게 떨어지지는 않을 겁니다. □

Quick Quiz 9.28:

이 데드락 내성에 어떤 예외가 있을까요, 그리고 만약 그렇다면, 어떤 일련의 이벤트들이 데드락을 이끌어 낼 수 있을까요? ■

Answer:

RCU read-side 기능들이 연관된 데드락 사이클을 만들 어내는 한 가지 방법은 다음과 같은 (불법적인) 일련의 코드들을 통해 가능합니다:

```
rcu_read_lock();
synchronize_rcu();
rcu_read_unlock();
```

여기서의 `synchronize_rcu()` 는 모든 전부터 존재한 RCU read-side 크리티컬 섹션들이 완료되기 전 까지 리턴할 수 없습니다만, 이 자체가 RCU read-side 크리티컬 섹션 안에 들어가 있고 그 크리티컬 섹션은 `synchronize_rcu()` 가 리턴하기 전까지 끝날 수 없습니다. 결국 이건 고전적인 self-deadlock 의 결과를 이끌어냅니다—읽기 모드로 reader-writer 락을 잡은 채로 쓰기 모드로 락을 잡으려 하면 똑같은 효과가 나타날 겁니다.

o) self-deadlock 시나리오는 RCU QSBR 에는 적용되지 않는다는 점을 알아두셔야 하는데, 이는 synchronize_rcu() 에 의해 수행되는 컨텍스트 스 위치는 이 CPU 에 정지한 상태처럼 행동할 것이어서 grace period 가 완료되게 할 것이기 때문입니다. 하지만, 이는 오히려 더 안좋은 케이스로, 이 RCU read-side 크리티컬 섹션에 의해 사용되던 데이터가 해당 grace period 의 완료로 인해 메모리에서 해제될 수 있기 때문입니다.

짧게 말하자면, RCU read-side 크리티컬 섹션 내에서 동기적인 RCU update-side 기능들을 실행하지 마세요. □

Quick Quiz 9.29:

데드락과 우선순위 역전 모두에 내성이 있다고요??? 사실이라기엔 너무 좋은 이야기 같은데요. 제가 이게 가능하다는걸 어떻게 믿을 수 있을까요? ■

Answer:

정말로 잘 동작합니다. 무엇보다, 이게 동작하지 않는다면, 리눅스 커널은 동작하지 못하겠죠. □

Quick Quiz 9.30:

근데 잠깐만요! 이건 RCU 를 reader-writer 락킹 대용으로 하려 할 때 쓸법한 코드와 완전 똑같잖아요! 뭐가 새로운 거죠? ■

Answer:

이는 장난감 예제의 법칙의 효과입니다: 어떤 특정 포인트 이후로는 해당 코드는 똑같아 보일 겁니다. 이 코드를 어떻게 생각하느냐가 단 하나의 차이입니다. 하지만, 이 차이점은 매우 중요할 수 있습니다. 이 중요성에 대해 한가지만 예를 들어, RCU 를 레퍼런스 카운팅 방법으로 생각하고 있다고 치면, 업데이트들이 RCU read-side 크리티컬 섹션들을 배제할거라 생각하는 실수를 하지 않을 겁니다.

더도 아니고 덜도 아니고 RCU 를 reader-writer 락킹의 대체제로 생각하는 건 종종 쓸모있는데, 예를 들면 reader-writer 락킹을 RCU 로 대체할 때입니다. □

Quick Quiz 9.31:

6 CPU 근처에서 refcnt 오버헤드가 확 떨어지는건 왜그렇죠? ■

Answer:

NUMA 효과 때문일 겁니다. 하지만, 여러 바들로 보여진 것처럼, refcnt 선을 측정하는데 사용된 값들은 상당한 편차를 가지고 있습니다. 사실, 일부 케이스에 있어서 표준편차는 측정된 값의 10% 를 초과합니다. 따라서 해당 오버헤드의 큰 차이는 통계적 탈선일 수도 있습니다. □

Quick Quiz 9.32:

우리가 삭제하려는 원소가 Figure 9.43 의 line 9 의 리스트의 첫번째 원소가 아니라면 어떡하죠? ■

Answer:

Figure 7.17 에서와 마찬가지로, 이건 체이닝을 하지 않는 매우 간단한 해시 테이블이어서 하나의 베킷에 들어갈 수 있는 원소는 첫번째 원소 뿐입니다. 다시 한번 이 해시 테이블을 완전한 체이닝을 사용하도록 바꿔보는 건 독자 여러분의 몫으로 두겠습니다. □

Quick Quiz 9.33:

Figure 9.43 의 line 17 에서 락을 놓기 전에 line 15 에서 RCU read-side 크리티컬 섹션을 빠지는게 왜 안전한거죠? ■

Answer:

첫째로, line 14 에서의 두번째 체크는 어떤 다른 CPU 가 이 원소를 우리가 락을 잡는 동안 삭제했을 수 있기 때문에 필요합니다. 하지만, 우리가 이 락을 잡는 동안 RCU read-side 크리티컬 섹션에 들어와 있다는 사실은 이 원소가 재할당되고 이 해시 테이블에 다시 삽입되지는 못함을 보장합니다. 더 나아가서, 일단 우리가 락을 잡았다면, 그 락 자체는 이 원소의 존재를 보장하게 되고, 따라서 우리는 더이상 RCU read-side 크리티컬 섹션 안에 머무를 필요가 없습니다.

원소의 키를 다시 체크해 볼 필요가 있을까라는 질문에 대한 답은 독자 여러분의 몫으로 남겨두겠습니다. □

Quick Quiz 9.34:

Figure 9.43 의 line 23 에서 RCU read-side 크리티컬 섹션을 빠져나가는 건 왜 line 22 에서 락을 내려놓기 전에 될 수 없나요? ■

Answer:

우리가 이 두 라인의 순서를 뒤집는다고 생각해 봅시다. 그러면 이 코드는 다음의 일련의 이벤트들에 취약해집니다:

1. CPU 0 가 delete() 를 실행하고, 삭제될 원소를 찾아낸 후 line 15 를 수행합니다. 아직 항목의 삭제를 실제로 하지는 않았고, 이제 곧 할 참입니다.
2. CPU 1 이 동시에 delete() 를 실행하고 같은 원소를 삭제하려 합니다. 하지만, CPU 0 는 아직 락을 잡고 있으므로, CPU 1 은 line 13 에서 기다립니다.
3. CPU 0 이 line 16 과 17 을 수행하고 line 18 에서 CPU 1 이 RCU read-side 크리티컬 섹션을 나오기 를 기다립니다.

4. CPU 1 은 이제 락을 획득하지만 CPU 0 가 이미 해당 원소를 삭제했으므로 line 14 에서의 테스트가 실패합니다. CPU 1 은 이제 line 22 (line 23 과 이 Quick Quiz 를 위해 바꾼) 를 수행해서 RCU read-side 크리티컬 섹션을 나옵니다.
5. CPU 0 는 이제 synchronize_rcu() 로부터 리턴하고 따라서 line 19 를 수행해서 이 항목을 메모리 해제하고 재사용 가능 리스트로 옮깁니다.
6. CPU 1 이 이제 이미 재사용이 가능한, 다른 종류의 데이터 구조체에 사용되기 위해 재할당 될 수도 있는 항목을 위한 락을 놓으려 합니다. 이건 치명적인 메모리 오염 에러입니다. □

Quick Quiz 9.35:

하지만 여러 쓰레드들에 상당히 긴 RCU read-side 크리티컬 섹션들이 존재해서 어떤 특정한 시점에든 시스템의 최소 하나의 쓰레드는 RCU read-side 크리티컬 섹션을 수행하고 있으면 어떡하죠? 그게 어떤 데이터가 SLAB_DESTROY_BY_RCU 슬랩에서 시스템으로 반환되는 걸 막아서 OOM 이벤트를 유발하지는 않을까요? ■

Answer:

분명, 최소 하나의 쓰레드는 항상 RCU read-side 크리티컬 섹션에 존재하는, 굉장히 긴 기간이 존재할 수 있습니다. 하지만, Section 9.5.3.7 의 설명에서의 키워드는 “사용중인”과 “전부터 존재한”입니다. 주어진 RCU read-side 크리티컬 섹션은 컨셉적으로 그 크리티컬 섹션의 시작 시점에 사용된 데이터에 한해서만 레퍼런스를 가질 수 있도록 됨을 명심하기 바랍니다. 더 나아가서, 슬랩은 자신의 데이터 원소들이 모두 해제되기 전까지는 시스템에 반환될 수 없고, 사실, RCU grace period 는 그것들이 모두 해제되기 전까지는 시작할 수가 없습니다.

따라서, 슬랩 캐시는 그 슬랩의 마지막 원소가 해제되기 전에 시작한 RCU read-side 크리티컬 섹션들이 완료될 때까지만 기다리면 됩니다. 이 말은 마지막 원소가 해제된 뒤에 시작된 어떤 RCU grace period 도 수행을 계속한다는 뜻입니다—그 슬랩은 그 grace period 가 끝난 후에 시스템에 반환될 것입니다. □

Quick Quiz 9.36:

nmi_profile() 함수가 preemption 당할 수 있다고 해봅시다. 이 예제가 제대로 동작하도록 하기 위해 뭘 바꿔야 할까요? ■

```

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

Answer:

한가지 가능한 방법은 rCU_read_lock() 과 rCU_read_unlock() 을 nmi_profile() 내에서 사용하고, synchronize_sched() 를 synchronize_rcu() 로 바꾸는 것으로, 아마도 Figure D.6 와 같은 모습이 될겁니다.

□

Quick Quiz 9.37:

Table 9.3 의 일부 셀들은 왜 느낌표 (“!”) 를 가지고 있나요? ■

Answer:

느낌표와 함께 표시된 API 멤버들 (rCU_read_lock(), rCU_read_unlock(), 그리고 call_rcu()) 만이 과거 90년대 중반에 Paul E. McKenney 가 신경썼던 리눅스 RCU API 의 멤버들입니다. 이 시절에는, 그는 그가 RCU 에 대해 알아야 할 것들을 모두 알고 있다는 잘못된 인상을 가지고 있었습니다. □

Quick Quiz 9.38:

많은 수의 RCU read-side 크리티컬 섹션들이

`synchronize_rcu()` 실행을 무기한 블록시키는 걸 어떻게 막을 수 있나요? ■

Answer:

RCU read-side 크리티컬 섹션들이 `synchronize_rcu()` 실행을 무기한 블록하는 걸 막을 필요가 전혀 없는데, `synchronize_rcu()` 실행은 전부터 존재한 RCU read-side 크리티컬 섹션들만 기다리면 되기 때문입니다. 따라서 각 RCU read-side 크리티컬 섹션이 한정된 길이를 갖는다면, 아무 문제가 없습니다. □

Quick Quiz 9.39:

`synchronize_rcu()` API는 전부터 존재한 인터럽트 핸들러들이 모두 완료되길 기다리죠, 맞죠? ■

Answer:

전혀 그렇지 않아요! 그리고 preemption 가능한 RCU를 사용하고 있다면 특허나 그렇지 않습니다! 대안으로, `rcu_read_lock()` 과 `rcu_read_unlock()` 를 `synchronize_rcu()` 기다리기를 바라는 인터럽트 핸들러 안에 위치시킬 수 있겠습니다. □

Quick Quiz 9.40:

이것들을 섞어서 활용하면 어떻게 되나요? 예를 들어, `rcu_read_lock()` 과 `rcu_read_unlock()` 을 RCU read-side 크리티컬 섹션을 구분하는데 사용하지만 `call_rcu_bh()` 를 RCU callback 을 위해 사용한다고 하면요? ■

Answer:

`call_rcu_bh()` 가 실행되는 시점에 `rcu_read_lock_bh()` 와 `rcu_read_unlock_bh()` 로 구분된 RCU read-side 크리티컬 섹션들이 존재하지 않는다면 RCU는 이 콜백을 곧바로 수행해도 되는 권한을 갖게 되어서, 해당 RCU read-side 크리티컬 섹션이 사용중인 데이터 구조체를 메모리에서 해제해버릴 수도 있습니다! 이건 단순히 이론적인 가능성이 아닙니다: `rcu_read_lock()` 과 `rcu_read_unlock()` 으로 구분지어졌고 오랫동안 동작중인 RCU read-side 크리티컬 섹션은 이런 실패에 취약합니다.

하지만, `rcu_dereference()` 함수들은 모든 RCU 변종들에 적용됩니다. (변종별 `rcu_dereference()` 를 만들려는 시도도 있었지만, 그건 너무 혼란스러웠습니다.) □

Quick Quiz 9.41:

하드웨어 인터럽트 핸들러들은 묵시적인 `rcu_read_lock_bh()` 의 보호 아래 있다고 생각되어도 되겠죠? ■

Answer:

전혀 그렇지 않아요! 그리고 preemption 가능한 RCU

를 사용중일 때에는 특허나 그렇지 않습니다! “`rcu_bh`”로 보호되는 데이터 구조체를 인터럽트 핸들러 내에서 접근하려 한다면, 명시적으로 `rcu_read_lock_bh()` 와 `rcu_read_unlock_bh()` 를 사용해야 합니다. □

Quick Quiz 9.42:

RCU Classic 과 RCU Sched 를 섞어서 사용하면 어떻게 되나요? ■

Answer:

Non-PREEMPT 나 PREEMPT 커널에서는 이 두개의 일은 우연히 섞이게 되는데, 그런 커널 빌드에서 RCU Classic 과 RCU Sched 는 같은 구현으로 매핑되기 때문입니다. 하지만, 이런 조합은 -rt 패치셋을 사용하는 PREEMPT_RT 빌드에서는 치명적인데 Realtime RCU 의 read-side 크리티컬 섹션들은 preemption 당할 수 있어서 `synchronize_sched()` 가 RCU read-side 크리티컬 섹션이 `rcu_read_unlock()` 호출을 하기 전에 리턴할 수 있기 때문입니다. 이는 데이터 구조체가 그 구조체를 사용하는 read-side 크리티컬 섹션이 끝나기 전에 메모리 해제될 수 있게 해서 커널의 보험 통계적 리스크를 무척 증가시킬 수 있습니다.

실제로, RCU Classic 과 RCU Sched 의 분리는 preemption 가능해야 하는 RCU read-side 크리티컬 섹션들로부터 영감을 받았습니다. □

Quick Quiz 9.43:

일반적으로, 모든 전부터 존재한 인터럽트 핸들러들을 기다리는데에 `synchronize_sched()` 에 의존해선 안됩니다, 맞죠? ■

Answer:

맞습니다! -rt 리눅스는 쓰레드로 도는 인터럽트 핸들러들을 사용하기 때문에, 인터럽트 핸들러 내부에서의 컨텍스트 스위치가 있을 수 있습니다. `synchronize_sched()` 는 각 CPU 가 컨텍스트 스위치를 할 때까지만 기다리기 때문에, 특정 인터럽트 핸들러가 완료되기 전에 리턴을 할 수도 있습니다.

특정 인터럽트 핸들러가 완료되기까지 기다려야 한다면, 그대신 `synchronize_irq()` 를 사용하거나 명시적으로 RCU read-side 크리티컬 섹션들을 기다리기 원하는 인터럽트 핸들러들 안에 위치시켜야 합니다. □

Quick Quiz 9.44:

`call_srcu()` 사용에 있어 조심해야 하는 이유는 무엇일까요? ■

Answer:

하나의 테스크는 SRC 콜백들을 매우 빠르게 등록할 수 있습니다. SRCU 가 읽기 쓰레드들이 임의의 기간동안

블록될 수 있도록 허용함을 생각하면, 이는 임의의 커다란 양의 메모리를 소모할 것을 알 수 있습니다. 반대로, 동기적인 `synchronize_srcu()` 인터페이스에서는 주어진 태스크는 다음 grace period 를 기다리는 걸 시작하기 전에 주어진 grace period 를 기다리는 것을 마무리 해야만 합니다. □

Quick Quiz 9.45:

어떤 조건에서 `synchronize_srcu()` 가 SRCU read-side 크리티컬 섹션 내에서 안전하게 사용될 수 있을까요? ■

Answer:

원론적으로, 특정 `srcu_struct` 와 함께, `synchronize_srcu()` 는 어떤 다른 `srcu_struct` 를 사용하는 SRCU read-side 크리티컬 섹션 안에서 사용될 수 있습니다. 하지만, 실제에서는 이런 짓을 하는건 거의 분명하게 나쁜 생각입니다. 세부적으로는 Figure D.7 에 보인 코드가 여전히 데드락을 일으킬 수 있을 것입니다. □

```

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 9.46:

`list_del_rcu()` 는 왜 `next` 와 `prev` 두 포인터를 모두 파괴하지 않는거죠? ■

Answer:

`next` 포인터를 파괴하는 행위는 이 포인터를 사용해야 하는 동시의 RCU 읽기 쓰레드들과 간섭될 수 있습니다. 하지만, RCU 읽기 쓰레드들은 `prev` 포인터의 사용으로부터 숨겨져 있으므로 이건 안전하게 파괴될 수 있습니다. □

Quick Quiz 9.47:

일반적으로, `rcu_dereference()` 에 사용되는 모든 포인터는 반드시 항상 `rcu_assign_pointer()` 로 업데이트 되어야만 합니다. 이 규칙에 예외는 뭐가 있을까요? ■

Answer:

그런 예외 가운데 하나는 여러 원소가 연결된 데이터 구조체가 다른 CPU 들에서 접근할 수 없는 상태에서 하나의 유닛으로써 초기화되었고 하나의 `rcu_assign_pointer()` 가 이 데이터 구조체로의 글로벌 포인터를 설정하는데 사용된 경우입니다. 비록 그 구조체가 글로벌하게 보여진 후에 일어나는 그런 할당들은 반드시 `rcu_assign_pointer()` 를 사용해야 하지만, 초기화 시점의 포인터 할당은 `rcu_assign_pointer()` 를 사용할 필요가 없습니다.

하지만, 이 초기화 코드가 매우 자주 실행되는 코드 수행경로에 있는게 아니라면, 이론적으로는 불필요할지도 `rcu_assign_pointer()` 를 사용하는게 현명할 것입니다. 어떤 “사소한” 변경이 그 초기화는 혼자서 행하게 된다는 소중한 가정을 무효화 시키기는 매우 쉽습니다. □

Quick Quiz 9.48:

이런 횡단과 업데이트 기능들이 어떤 RCU API 패밀리 멤버들과 함께 사용된다 하더라도 어떤 문제는 없나요? ■

Answer:

어떤 경우에는 “sparse” 와 같은 자동화된 코드 검사기 (또는 사람이) 가 어떤 종류의 RCU read-side 크리티컬 섹션이 어떤 RCU 횡단 기능들과 연관되어 있는지 알기 어려울 수 있습니다. 예를 들어, Figure D.8 에 보여진 것과 같은 코드를 생각해 봅시다.

```

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

`rcu_dereference()` 기능은 RCU Classic 크리티컬 섹션 안에 있을까요 RCU Sched 크리티컬 섹션 안에 있을까요? 이걸 알아내려면 어떻게 해야할까요? □

Quick Quiz 9.49:

Figure 9.47 에서의 RCU 구현의 데드락이 다른 RCU 구현에서의 데드락이 될 수 있는 이유는 뭘까요? ■

Answer:

Figure D.9 의 함수 `foo()` 와 `bar()` 가 다른 CPU 들에서 동시에 호출된다고 생각해 봅시다. `foo()` 는 `my_lock()` 을 line 3에서 잡는데, 그동안 `bar()` 는 line 13

```

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

에서 `rcu_gp_lock` 을 잡습니다. `foo()` 가 line 4 로 진행되면, `bar()` 에게 잡혀있는 `rcu_gp_lock` 을 잡으려 합니다. 그리고는 `bar()` 는 line 14 로 넘어가서 `foo()` 에게 잡혀 있는 `my_lock` 을 잡으려 시도합니다.

각 함수가 이제 서로가 잡고 있는 락을 기다리게 되는 고전적인 deadlock 상황이 됩니다.

다른 RCU 구현들은 `rcu_read_lock()` 안에서 스핀하지도 블록하지도 않기 때문에 데드락이 예방됩니다. □

Quick Quiz 9.50:

왜 Figure 9.47 의 RCU 구현에서는 RCU 읽기 쓰레드들이 병렬로 수행될 수 있도록 간단하게 reader-writer 락을 사용하지 않았나요? □

Answer:

누군가는 실제로 reader-writer 락을 이런식으로 사용할 수도 있겠습니다. 하지만, 교재의 reader-writer 락은 메모리 경쟁에 취약해서 정말로 병렬 수행이 가능해지려면 RCU read-side 크리티컬 섹션들이 상당히 길어져야 할겁니다 [McK03].

한편으로는, `rcu_read_lock()` 에서 읽기 권한을 획득하는 식의 reader-writer 락 사용은 앞서 언급된 데드락 조건은 막을 수 있을 겁니다. □

Quick Quiz 9.51:

Figure 9.48 의 line 15-18 의 루프에서는 락들을 일단 모두 다 잡고나서는 한번에 모두 풀어주는게 더 깔끔하지 않나요? 무엇보다, 이렇게 바꾸면 어떤 읽기 쓰레드도 존재하지 않는 시점이 생기게 되어서 모든 것들이 매우 간단해질 겁니다. □

Answer:

이 변경은 다시 deadlock 을 가능하게 할 것이므로, 안되고, 더 깔끔하지도 않아요. □

Quick Quiz 9.52:

Figure 9.48 에 보여진 구현은 deadlock 으로부터 자유로울까요? 그렇다면, 또는 그렇지 않다면, 왜죠? □

Answer:

데드락 시나리오 중 하나는 한 락이 `synchronize_rcu()` 을 감싸고 잡히고 같은 락이 한 RCU read-side 크리티컬 섹션에 잡힐 때가 될 수 있을 겁니다. 하지만, 이 상황은 어떤 RCU 구현에서도 데드락을 유발할 수 있습니다. 무엇보다, `synchronize_rcu()` 기능은 모든 전부터 존재한 RCU read-side 크리티컬 섹션들이 끝나길 기다려야 하지만, 그런 크리티컬 섹션들 가운데 하나가 `synchronize_rcu()` 를 수행한 쓰레드가 잡고 있는 락에 스핀하고 있다면, RCU 의 정의상 피할 수 없는 데드락을 갖게 됩니다.

또다른 데드락 시나리오는 RCU read-side 크리티컬 섹션들을 중첩시키려 할 때일 겁니다. 이 데드락은 이 구현 특유의 것이고 재귀적 락을 사용하거나 `rcu_read_lock()` 을 통해선 읽기 권한을, `synchronize_rcu()` 를 통해선 쓰기 권한을 획득하는 reader-writer 락을 사용해서 막을 수도 있을 겁니다.

하지만, 앞의 두 경우를 제외하면, 이 RCU 구현은 어떤 데드락 상황도 만들지 않습니다. 이는 어떤 다른 쓰레드의 락을 획득하게 되는 경우는 `synchronize_rcu()` 를 수행할 때 뿐이며, 이 경우에 그 락은 곧 바로 해제되기 때문에 앞의 경우처럼 `synchronize_rcu()` 전후로 잡히는 락과 연관되는 데드락을 제외하고는 데드락 사이클을 예방합니다. □

Quick Quiz 9.53:

Figure 9.48 에 보인 RCU 알고리즘의, 예를 들어 POSIX `pthread` 처럼 여走路에서 사용 가능한 기능들만을 사용하고 있는 점도 장점 아닌가요? □

Answer:

이는 실제로 장점입니다만 `rcu_dereference()` 와 `rcu_assign_pointer()` 는 여전히 필요함을 잊지 말아야 하는데, 이는 `rcu_dereference()` 를 위한 `volatile` 조정과 `rcu_assign_pointer()` 를 위한 메모리 배리어의 필요를 뜻합니다. 물론, 대부분의 Alpha CPU 들은 두 기능 모두에 메모리 배리어가 필요합니다. □

Quick Quiz 9.54:

하지만 `synchronize_rc()` 을 감싸고 락을 잡고 같은 락을 RCU read-side 크리티컬 섹션 내에서 잡으면 어떻게 되죠? □

Answer:

사실, 이것은 모든 합법적인 RCU 구현에서 데드락을 일으킬 겁니다. 하지만 `rcu_read_lock()`은 정말로 데드락 사이클에 참가하고 있는 걸까요? 그렇게 믿는다면, Section 9.5.5.9 의 RCU 구현을 보게 될 때 같은 질문을 해보시기 바랍니다. □

Quick Quiz 9.55:

`synchronize_rcu()` 가 10-밀리세컨드 지연을 포함하고 있는데 어떻게 grace period 가 40 나노세컨드 만에 끝날 수가 있나요? ■

Answer:

이 update 쪽 테스트는 읽기 쓰레드들 없이 수행되었고, 따라서 `poll()` 시스템 콜은 결코 호출되지 않았습니다. 또한, 실제 코드는 이 `poll()` 시스템 콜을 주석처리 해서 이 update-side 코드의 진정한 오버헤드를 측정하기에 더 좋게 되어 있습니다. 이 코드의 모든 제품에서 사용하기에는 `poll()` 시스템 콜을 사용하도록 하는 편이 좋을 겁니다만 다시 말하지만 제품에서 사용하기에는 이 섹션의 뒤에서 이야기될 다른 구현이 더 걸맞을 수도 있습니다. □

Quick Quiz 9.56:

Figure 9.49 의 RCU 구현은 동시의 `synchronize_rcu()` 가 너무 오래 기다리고 있을 때에는 왜 간단히 `rcu_read_lock()` 을 기다리게 만들지 않는 거죠? 그렇게 하면 `synchronize_rcu()` 의 starvation 을 막을 수 있지 않나요? ■

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 9.5.5.4 for a proper cure. □

Quick Quiz 9.57:

Figure 9.52 의 `synchronize_rcu()` 의 line 5 의 메모리 배리어는 바로 뒤에 스핀락 획득이 있는데도 왜 필요한 거죠? ■

Answer:

스핀락 획득은 스핀락의 크리티컬 섹션이 이 획득 전으로 “흘러나오지” 않게 보장할 뿐입니다. 이는 스핀락 획득 앞의 코드가 크리티컬 섹션 안으로 재배치되는 것은 막지 않습니다. 그런 재배치는 RCU 로 보호되는 리스트의 삭제 작업이 `rcu_idx` 보정 뒤로 재배치되도록

해서 새로이 시작하는 RCU read-side 크리티컬 섹션이 최근에 삭제된 데이터 원소를 보게 만들어 버릴 수 있습니다.

독자를 위한 연습문제: Promela/spin 같은 도구를 사용해서 Figure 9.52 의 메모리 배리어들 가운데(존재한다면) 무엇이 정말로 필요한 것인지 알려주세요. 이 도구들의 사용법을 위해선 Section 12 를 참고하세요. 처음으로 옳고 완벽한 답변은 인정을 받을 겁니다. □

Quick Quiz 9.58:

Figure 9.52 에서 카운터는 왜 두 번 뒤집히는 거죠? 한번의 뒤집고 기다리는 사이클만으로도 충분하지 않나요? ■

Answer:

두번의 뒤집기가 모두 필요합니다. 이를 확인하기 위해서는 다음과 같은 일련의 이벤트를 생각해 보세요:

1. Figure 9.51 `rcu_read_lock()` 의 line 8에서 `rcu_idx` 를 가져오고 그 값이 0임을 확인합니다.
2. Figure 9.52 `synchronize_rcu()` 의 line 8에서 `rcu_idx` 를 가져오고 그 값이 0임을 확인합니다.
3. `synchronize_rcu()` 의 line 10-13에서 `rcu_refcnt[0]`의 값이 0임을 확인하고 리턴합니다. (질문은 line 14-20 이 사라지면 어떻게 되는지니까요.)
4. `rcu_read_lock()` 의 line 9 와 10 은 각각 이 쓰레드의 `rcu_read_idx` 에 0을 저장하고, `rcu_refcnt[0]`의 값을 증가시킵니다. 실행은 이제 read-side 크리티컬 섹션의 안으로 들어갑니다.
5. `synchronize_rcu()` 의 또 다른 인스턴스가 다시 `rcu_idx` 를 바꾸는데, 이번에는 그 값을 0 으로 바꿉니다. `rcu_refcnt[1]`의 값이 0이므로, `synchronize_rcu()` 는 곧바로 리턴합니다. (`rcu_read_lock()` 은 `rcu_refcnt[0]` 을 증가시켰지, `rcu_refcnt[1]` 을 증가시키지 않았으니까요!)
6. Step 4 전에 시작된 RCU read-side 크리티컬 섹션이 완료되지 않았음에도 step 5에서 시작한 grace period 가 종료되는 것이 허가되었습니다. 이는 RCU 시멘틱을 위반하는 것이고 업데이트가 RCU read-side 크리티컬 섹션에 여전히 레퍼런스 하고 있는 데이터 원소를 해제시킬 수 있게 할 수 있습니다.

독자를 위한 연습문제: `rcu_read_lock()` 이 line 8 뒤에서 매우 긴 시간(몇시간정도!) preemption 당한다면 어떤 일이 일어날까요? 이 구현은 그런 경우에도 똑바로 동작할까요? 그 답의 이유는 뭐죠? 맞고 완벽한 첫번째 답은 인정을 받을 겁니다. □

Quick Quiz 9.59:

어토믹 값 증가와 값 감소가 그렇게 비싸다고 하면, Figure 9.51 의 line 10 에서는 어토믹하지 않은 값 증가를, line 25 에서는 어토믹하지 않은 값 감소를 하지 그래요? ■

Answer:

어토믹하지 않은 오퍼레이션들을 사용하면 값 증가와 감소가 사라지게 될 수 있고, 이는 이 구현이 실패하게 할 수 있습니다. `rcu_read_lock()` 과 `rcu_read_unlock()`에서 어토믹하지 않은 오퍼레이션들을 사용하는 안전한 방법을 위해선 Section 9.5.5.5 을 참고하세요. □

Quick Quiz 9.60:

집어쳐요! `rcu_read_lock()` 안에 `atomic_read()` 가 빤히 보인다구요!!! 왜 `rcu_read_lock()` 이 어토믹 오퍼레이션을 포함하고 있지 않은 척 하시는거죠??? ■

Answer:

해당 `atomic_read()` 기능은 실제로는 어토믹한 머신 인스트럭션을 실행하지 않고, 그저 `atomic_t`로부터의 평범한 로드를 수행합니다. 이 기능의 목적은 컴파일러의 타입 검사를 쉽게 해주려는 것입니다. 리눅스 커널이 8비트 CPU에서 수행된다면, 16비트 포인터를 저장하는데에 일부 8-bit 시스템에서의 8비트 액세스 두번이 행해지는 “store tearing”을 막을 필요도 있을 겁니다. 하지만 감사하게도, 누구도 리눅스를 8비트 시스템에서 사용하고 있는 것 같지는 않습니다. □

Quick Quiz 9.61:

좋아요, 우리가 N 쓰레드들을 가지고 있다면 우리는 2N 10 밀리세컨드의 기다리는 호출 (`flip_counter_and_wait()` 당 한 세트)을 할 수 있을텐데, 우리가 각 쓰레드를 위해 오직 한번 기다린다고 가정해도 그렇습니다. ■

Answer:

기다리는 것은 오로지 해당 쓰레드가 여전히 앞에서 시작된 RCU read-side 크리티컬 섹션 안에 있을 때 뿐이라는 점을, 그리고 하나의 쓰레드를 기다리는 것은 모든 다른 쓰레드들에게 여전히 수행중일 수도 있는 모든 앞서 시작한 RCU read-side 크리티컬 섹션들을 완료할

기회를 준다는 점을 기억하세요. 따라서 우리가 $2N$ 간격을 기다리게 되는 경우는 모든 앞의 쓰레드들에의 모든 기다림에도 불구하고 직전의 쓰레드가 여전히 앞서 시작된 RCU read-side 크리티컬 섹션 안에 있을 때 뿐입니다. 짧게 말해서, 이 구현은 불필요하게 기다리지는 않을 겁니다.

하지만, RCU 를 사용하는 코드를 스트레스 테스트하고 있다면, 부정확하게 RCU 로 보호되고 있는 데이터 원소로의 레퍼런스를 RCU read-side 크리티컬 섹션 밖에서도 쥐고 있게 되는 버그를 더 잘 잡기 위해서는 `pool()` 구문을 없애고 싶을 수도 있을 겁니다. □

Quick Quiz 9.62:

이 모든 장난감 RCU 구현들은 `rcu_read_lock()` 과 `rcu_read_unlock()` 안에 어토믹 오퍼레이션들을 가지고 있거나 쓰레드의 수에 따라 선형적으로 증가하는 `synchronize_rcu()` 의 오버헤드를 갖습니다. 어떤 환경에서라면 RCU 구현이 이 세개의 기능들이 모두 결정적인 ($O(1)$) 의 오버헤드와 대기시간을 갖는가벼운 구현을 가질 수 있을까요? ■

Answer:

특수한 목적의 RCU 유니프로세서 구현이 이 이상적인 상황을 만들 수 있을 겁니다 [McK09c]. □

Quick Quiz 9.63:

짝수만으로도 해당 태스크를 무시해도 좋다고 `synchronize_rcu()` 에 말하기에 충분하다면 Figure 9.60 의 line 10 과 11 은 왜 단순히 `rcu_reader_gp`에 0을 할당하지 않나요? ■

Answer:

0을 (또는 어떤 다른 짝수의 상수를) 할당하는 것은 실제로 동작할 것입니다만 `rcu_gp_ctrl` 의 값을 할당하는 것은 개발자에게 언제 연관된 쓰레드가 마지막으로 RCU read-side 크리티컬 섹션을 끝냈는지와 같은, 디버깅을 위한 가치 있는 도움을 제공할 수 있습니다. □

Quick Quiz 9.64:

Figure 9.60 의 line 19 와 31 에서의 메모리 배리어들은 왜 필요한 건가요? line 20 과 30 에서의 락킹이 충분한데 메모리 배리어는 피할 수 있지 않나요? ■

Answer:

락킹 기능들은 단지 크리티컬 섹션을 가드는 것만을 보장하기 때문에 이 메모리 배리어들이 필요합니다. 락킹 도구들은 다른 코드가 크리티컬 섹션 안으로 흘러들어오는 것을 막기 위한 어떤 의무도 가지고 있지 않습니다.

따라서 이런 종류의, 컴파일러에 의해서든 CPU에 의해 서든 행해질 수 있는 코드의 움직임을 방지하기 위해 이 두개의 메모리 배리어들이 필요합니다. □

Quick Quiz 9.65:

Section 9.5.5.6에서 설명한 update-side batching이 Figure 9.60에 보인 구현에 적용될 수도 있지 않을까요?
■

Answer:

실제로, 약간의 수정과 함께라면 그럴 수 있습니다. 이 작업은 독자의 연습문제로 남겨두겠습니다. □

Quick Quiz 9.66:

Figure 9.60의 line 3-4에서 읽기 쓰레드들이 preemption 당할 수 있다는 사실은 진짜 문제일까요? 달리 말하자면, 정말 실패를 만들 수 있는 실제의 이벤트들이 존재하나요? 그렇지 않다면, 왜죠? 그렇다면, 그 이벤트들은 어떤 것이고, 그 실패는 어떻게 해결될 수 있을까요? ■

Answer:

진짜 문제입니다. 그리고 실패를 이끌어내는 일련의 이벤트들이 존재하며, 이를 해결하기 위한 방법들이 여럿 있습니다. 더 자세한 내용을 위해서는 Section 9.5.5.8의 끝부분의 Quick Quizz들을 참고하세요. 이에 대한 토론을 거기에 두는 이유는 (1) 당신에게 생각할 시간을 더 주고, (2) 그 섹션에서 추가되는 중첩 지원은 카운터를 오버플로우 시키는데 걸리는 시간을 훨씬 줄여주기 때문입니다. □

Quick Quiz 9.67:

이 복잡한 비트 조정을 하는 대신에 앞의 섹션에서 그랬듯이 별도의 쓰레드별 중첩 수준 변수를 갖지 않는 건가요? ■

Answer:

별도의 쓰레드별 변수의 분명한 단순성은 주의를 돌리는 것일 뿐입니다. 이 방법은 조심스럽게 오퍼레이션들을 순서맞추는 데에서 훨씬 큰 복잡도를 만들어내는데, 특히 시그널 핸들러들이 RCU read-side 크리티컬 섹션들을 가질 수 있게 허용되는 경우에서 특히 그렇습니다. 하지만 그냥 제 말에서 멈추지 말고, 코딩을 직접 해보시고 그렇게 해서 뭐가 나오는지 한번 보시기 바랍니다!
□

Quick Quiz 9.68:

Figure 9.62에 보여진 알고리즘에서, 어떻게 하면 전역 변수인 `rcu_gp_ctrl`가 오버플로우 되는데 걸리는 시간을 두배로 늘릴 수 있을까요? ■

Answer:

한가지 방법은 line 33과 34에서의 비교의 규모를 쓰레드별 `rcu_reader_gp` 변수와 `rcu_gp_ctrl+RCU_GP_CTR_BOTTOM_BIT` 사이의 동일 여부 검사로 바꾸는 게 될겁니다. □

Quick Quiz 9.69:

다시, Figure 9.62에 보여진 알고리즘에서, 카운터 오버플로우는 치명적인가요? 그 이유는 무엇이죠? 만약 치명적이라면, 그걸 고치기 위해 뭘 할 수 있을까요? ■

Answer:

실제로 치명적일 수 있습니다. 이를 보기 위해, 다음의 일련의 이벤트들을 생각해 봅시다:

1. Thread 0 가 `rcu_read_lock()`에 들어와 아직 중첩되지 않은 상태임을 확인하고 글로벌 변수 `rcu_gp_ctrl`의 값을 가져옵니다. Thread 0은 그리고나서 (자신의 쓰레드별 변수 `rcu_reader_gp`에 값을 저장하기 전에) 상당히 긴 시간동안 pre-emption 당합니다.
2. 다른 쓰레드들이 반복적으로 `synchronize_rcu()`를 호출해서 `rcu_gp_ctrl`의 새로운 값이 Thread 0가 가져왔을 때의 값보다 `RCU_GP_CTR_BOTTOM_BIT` 적은 값이 됩니다.
3. Thread 0은 이제 다시 수행을 시작하고, 자신의 쓰레드별 변수인 `rcu_reader_gp`에 값을 저장합니다. 이 값은 전역변수인 `rcu_gp_ctrl`의 현재 값보다 `RCU_GP_CTR_BOTTOM_BIT + 1` 만큼 큽니다.
4. Thread 0가 RCU로 보호되는 데이터 원소 A로의 레퍼런스를 얻습니다.
5. Thread 1이 이제 thread 0가 방금 레퍼런스를 얻어간 원소 A를 제거합니다.
6. Thread 1이 `synchronize_rcu()`를 호출해서 글로벌 변수 `rcu_gp_ctrl`의 값을 `RCU_GP_CTR_BOTTOM_BIT` 만큼 증가시킵니다. 그리고는 모든 쓰레드별 `rcu_reader_gp` 변수의 값을 검사하지만, thread 0의 값은 (부정확하게도) 자신이 thread 1의 `synchronize_rcu()` 호출 뒤에 시작되었다고 알려서 thread 1은 thread 0가 RCU read-side 크리티컬 섹션을 완료하기를 기다리지 않게 됩니다.
7. Thread 1은 thread 0가 여전히 레퍼런스를 가지고 있는 데이터 원소 A를 메모리 해제시켜버리게 됩니다.

이 시나리오는 Section 9.5.5.7에서 보인 구현에서도 역시 일어날 수 있음에 주의하세요.

이 문제를 고치는 한가지 방법은 64-bit 카운터를 사용해서 오버플로우에 필요한 시간이 컴퓨터 시스템의 일반적인 수행시간을 넘어서도록 하는 것입니다. x86 CPU 계열의 최신 CPU들은 64-bit 카운터들을 `cmpxchq64b` 인스트럭션을 통해 어토믹하게 조정할 수 있게 함을 알아두세요.

또다른 방법은 비슷한 효과를 얻기 위해 grace period 가 일어나도록 허용되는 비율에 제한을 두는 것입니다. 예를 들어, `synchronize_rcu()` 는 자신이 호출된 마지막 시간을 기록해 두고, 그 뒤의 모든 수행은 이 시간을 체크하고 원하는 간격을 유지하는데 필요한 만큼 블록될 수 있을 겁니다. 예를 들어, 카운터의 아랫쪽 네 개의 비트들이 중첩을 위해 전용화 되어 있다면, 그리고 grace period 는 초당 열번까지 일어나는 것이 허용되어 있다면, 이 카운터가 오버플로우 되는데에는 300일 이상이 걸릴 겁니다. 하지만, 이 방법은 이 시스템이 CPU에 성능이 제한되는 높은 우선순위의 리얼타임 쓰레드를 300일 이상 돌리게 될 가능성 있는 경우에는 도움이 되지 않습니다. (아마도 희박한 가능성 있지만 먼저 고려해 보는게 최선입니다.)

세번째 방법은 이 경우에는 시스템에서 리얼타임 쓰레드를 사용할 수 없도록 관리하는 것입니다. 이 경우, `preemption` 당한 프로세스는 우선순위를 높여갈 것이고, 따라서 카운터가 오버플로우 날 기회를 얻기 전까지의 수행을 길게 할 것입니다. 물론, 이 전략은 리얼타임 어플리케이션들에는 별로 도움이 되지 않습니다.

마지막 방법은 `rcu_read_lock()` 이 전역 변수 `rcu_gp_ctrl` 의 값을 자신의 쓰레드별 변수 `rcu_reader_gp` 마운터에 저장한 후에 다시 검사해서 전역 변수 `rcu_gp_ctrl` 의 새로운 값이 적절하지 않다면 다시 일을 시도하는 것입니다. 이 방법은 동작하지만, `rcu_read_lock()` 에 예측 불가한 실행 시간을 가져오게 됩니다. 달리 말하면, 당신의 어플리케이션이 카운터가 오버플로우 나기전에 충분한 시간만큼 `preemption` 당하면, 당신은 예측 가능한 실행 시간을 기대할 수 없을 겁니다!

□

Quick Quiz 9.70:

Figure 9.64의 line 14에 보인 추가적인 메모리 배리어는 `rcu_quiescent_state`의 오버헤드를 많이 늘리지 않을까요? ■

Answer:

실제로 그렇습니다! 따라서 이 RCU 구현을 사용하는 어플리케이션은 `rcu_quiescent_state` 를 아껴서 사용하고, 대부분의 경우에는 그대신 `rcu_read_`

`lock()` 과 `rcu_read_unlock()` 을 사용해야 합니다.

하지만, 이 메모리 배리어는 호출자에 의해 실행될 뒤따르는 RCU read-side 크리티컬 섹션 전에 다른 쓰레드들이 line 12-13의 저장 결과들을 볼 수 있도록 하기 위해 반드시 필요합니다. □

Quick Quiz 9.71:

Figure 9.64의 line 19와 22의 메모리 배리어들은 왜 필요한 건가요? ■

Answer:

Line 19의 메모리 배리어는 `rcu_thread_offline()` 에 앞서는 어떤 RCU read-side 크리티컬 섹션들도 컴파일러나 CPU에 의해 line 20-21의 값 할당 뒤로 재배치되지 않게 합니다. Line 22에서의 메모리 배리어는 엄밀하게 말하면 불필요한데, `rcu_thread_offline()` 뒤에 RCU read-side 크리티컬 섹션들을 두는 것은 금지된 행위이기 때문입니다. □

Quick Quiz 9.72:

분명히 해두겠는데, 2008년의 Power 시스템들의 클럭 주파수는 상당히 높았지만, 5GHz 클럭 주파수도 루프가 50 피코세컨드만에 실행되게 하는데에는 부족해요! 무슨 일이 벌어진 거죠? ■

Answer:

측정을 하는 루프는 한쌍의 텅빈 함수들만을 가지고 있으므로, 컴파일러는 이것을 최적화해서 없애버립니다. 이 측정 루프는 매번의 `rcu_quiescent_state()` 호출마다 1,000번 넘어가므로, 이 측정은 한번의 `rcu_quiescent_state()` 호출마다 대략 천번의 오버헤드를 가진 결로 치는 셈입니다. □

Quick Quiz 9.73:

코드가 라이브러리에 있다는 사실이 왜 Figure 9.64와 9.65에 보여진 RCU 구현의 사용에 어려움을 가져올 수 있는 건가요? ■

Answer:

라이브러리 함수는 호출자에 대한 제어가 전혀 존재치 않고 따라서 호출자가 `rcu_quiescent_state()` 를 주기적으로 실행할 것을 강제하지 못합니다. 다른 편으로는, 주어진 RCU로 보호되는 데이터 구조체에 많은 레퍼런스를 가지고 있을 수도 있는 라이브러리 함수는 `rcu_thread_online()` 을 매 원소마다, `rcu_quiescent_state()` 를 주기적으로, 그리고 `rcu_thread_offline()` 을 종료될 때에 호출할 수도 있을 겁니다. □

Quick Quiz 9.74:

하지만 락을 `synchronize_rcu()` 전후에 걸쳐 잡고, 같은 락을 RCU read-side 크리티컬 섹션에서 잡으면 어떻게 되나요? 이건 데드락이 되어야 할텐데, 하지만 어떻게 어떤 코드도 만들지 않는 기능이 데드락 사이클에 참여될 수가 있죠? ■

Answer:

RCU read-side 크리티컬 섹션은 둘러싸는 `rcu_read_lock()` 과 `rcu_read_unlock()` 너머로 확장되어서 앞과 뒤의 `rcu_quiescent_state()`에 닿음을 알아두시기 바랍니다. 이 `rcu_quiescent_state`는 `rcu_read_lock()` 을 곧바로 뒤따르는 `rcu_read_unlock()`으로 생각될 수 있습니다.

비록 그렇다 해도, 실제 데드락 자신은 RCU read-side 크리티컬 섹션 내에서와 `synchronize_rcu()`에서의 락 획득에 연구되지, `rcu_quiescent_state()` 와 연루되지는 않을 겁니다. □

Quick Quiz 9.75:

Grace period 가 RCU read-side 크리티컬 섹션들로 통제된다면, RCU로 보호되는 데이터 구조체는 어떻게 RCU read-side 크리티컬 섹션 내에서 업데이트 될 수 있을까요? ■

Answer:

이 상황이 `call_rcu()` 와 같은 비동기적 grace-period 기능들의 존재 이유 중 하나입니다. 이 기능은 RCU read-side 크리티컬 섹션 내에서 실행될 수도 있고, 이 특정 RCU 콜백은 하나의 grace period 가 종료된 이후에 뒤늦게 실행될 것입니다.

RCU read-side 크리티컬 섹션 내에서 RCU 업데이트를 할 수 있는 능력은 상당히 편리할 수 있고, (허구의) reader-writer 락킹에서의 무조건적인 read-to-write 업그레이드로 비유될 수 있습니다. □

Quick Quiz 9.76:

Figure 5.9 (`count_end.c`) 에 보인 통계적 카운터의 구현은 `read_count()` 안에서 합계를 구하는 것을 보호하기 위해 글로벌 락을 사용했는데 이는 부족한 성능과 음의 확장성을 일으켰습니다. `read_count()` 가 훌륭한 성능과 좋은 확장성을 제공할 수 있게 하기 위해 RCU 를 어떻게 적용해 볼 수 있을까요. (`read_count()` 의 확장성은 모든 쓰레드의 카운터들을 스캔해야 한다는 필요성으로 인해 제한되어진다는 점을 명심하세요.) ■

Answer:

힌트: 글로벌 변수 `finalcount` 와 배열 `counters[]`

를 하나의 RCU 로 보호되는 구조체 안에 넣으세요. 초기화 때, 이 구조체는 할당되고 모두 0과 NULL로 설정될 겁니다.

`inc_count()` 함수는 바뀔 필요가 없을 겁니다.

`read_count()` 함수는 `final_mutex` 를 잡는 대신에 `rcu_read_lock()` 를 사용할 것이고, 현재 구조체로의 레퍼런스를 얻어오는데에 `rcu_dereference()` 를 사용해야 할 겁니다.

`count_register_thread()` 함수는 이 새로 생성된 쓰레드와 연관된 배열 원소를 그 쓰레드의 쓰레드별 `counter` 변수로의 레퍼런스로 설정할 겁니다.

`count_unregister_thread()` 함수는 새로운 구조체를 할당받고, `final_mutex` 를 잡은 후, 기존의 구조체를 새로운 것으로 복사하고, 나가는 쓰레드의 `counter` 변수를 전체 값에 더하고, 이 `counter` 변수로의 포인터를 NULL로 만든 후, 이 새로운 구조체를 과거의 것의 자리에 설치하기 위해 `rcu_assign_pointer()` 를 사용한 후, `final_mutex` 를 풀고, grace period 를 기다린 후, 마지막으로 기존의 구조체를 메모리 해제시켜야 할 것입니다.

이게 정말 동작할까요? 그 이유는 무엇이죠?

더 자세한 내용을 위해서는 page 247 의 Section 13.3.1 를 참고하세요. □

Quick Quiz 9.77:

Section 5.5 는 디바이스들을 제거하기 위해 I/O 액세스들을 카운트하는 일을 처리하는 한쌍의 코드 조각들을 보였습니다. 이 코드 조각들은 reader-writer 락을 잡아야 하는 이유로 (I/O를 시작하는) 빠른 수행 경로의 높은 오버헤드 문제를 겪었습니다. 여기에 훌륭한 성능과 확장성을 가져오기 위해 RCU를 어떻게 사용할 수 있을까요? (I/O 액세스를 하는 일반적인 경우의 첫번째 코드 조각의 성능이 디바이스 제거 코드 조각의 것보다 훨씬 더 중요함을 명심하세요.) ■

Answer:

힌트: 이 reader-writer 락의 read 권한 획득을 RCU read-side 크리티컬 섹션들로 바꾸고, device 제거 코드를 이에 맞게 조정하세요. 이 문제에 대한 하나의 해결책을 위해선 Page 249 의 Section 13.3.2 를 참고하세요. □

D.10 Data Structures

Quick Quiz 10.1:

하지만 많은 종류의 해시 테이블들이 존재하고, 여기 설명된 체인 사용 (chained) 해시 테이블은 그 중 하나의 종류일 뿐입니다. 왜 체인 사용 해시 테이블에 집중하는 걸까요? ■

Answer:

체인 사용 해시 테이블들은 완전히 파티셔닝 적용 가능하고, 따라서 동시적 사용에 잘 맞습니다. 다른 완전히 파티셔닝 적용 가능한 해시 테이블들도 존재하는데, 예를 들면 split-ordered list [SS06] 가 있습니다만, 그런 것들은 훨씬 더 복잡합니다. 따라서 우리는 체인 사용 해시 테이블부터 시작하겠습니다. □

Quick Quiz 10.2:

하지만 Figure 10.4 의 line 15-18 에서의 두번의 비교는 키가 unsigned long 에 들어맞는 경우라면 비효율적이지 않나요? ■

Answer:

실제로 그렇습니다! 하지만, 해시 테이블들은 상당히 자주 키에 unsigned long 에 들어맞아야 할 이유가 없는 문자열과 같은 정보를 저장하기도 합니다. 해시 테이블 구현을 키가 항상 unsigned long 에 들어맞는 경우에 맞춰 간소화 시키는 것은 독자 여러분의 연습으로 두겠습니다. □

Quick Quiz 10.3:

단순히 bucket 들의 수를 늘리는 대신에 이미 있는 bucket 들을 캐시에 정렬 시키는게 더 낫지 않을까요? ■

Answer:

이에 대한 답은 상당히 많은 것들에 의존적입니다. 만약 해시 테이블이 bucket 마다 많은 수의 원소를 가지고 있었다면, bucket 들의 수를 늘리는게 분명 나을 겁니다. 한편으로는, 만약 해시 테이블에 가해지는 로드가 적었다면, 이에 대한 답은 하드웨어, 해시 함수의 효율성, 그리고 워크로드에 의존적일 겁니다. 관심있는 독자들은 실험을 해보시기 바랍니다. □

Quick Quiz 10.4:

Schrödinger 의 동물원 어플리케이션의 소켓을 넘어가면서 보이는 음의 확장성을 가지고 생각해보면, 어플리케이션의 복사본을 여럿 만들어서 각각의 복사본이 전체 동물의 부분집합만을 가지고 하나의 소켓 위에서만 각자 돌도록 하는 건 어떨까요? ■

Answer:

그렇게 해볼 수 있습니다! 사실, 이 아이디어를 커다란

클러스터 시스템으로 확장할 수도 있는데, 각 어플리케이션 복사본을 클러스터의 각 노드에 수행시키는 형식으로 말입니다. 이 방법은 “sharding” 이라 불리며, 실제로 커다란 웹 기반의 상점들에서 상당히 많이 사용되는 방법입니다 [DHJ⁺07].

하지만, 멀티소켓 시스템에서 소켓별로 분할을 하려 한다면, 별개의 더 작고 짧은 단일 소켓 시스템들을 사서 각각의 데이터베이스 족각을 그 시스템들 각각에서 수행시키는게 어떻겠어요? □

Quick Quiz 10.5:

하지만 해시 테이블의 원소가 검색과 동시에 삭제될 수가 있다면, 그 말은 검색 기능은 검색된 직후에 삭제된 원소로의 레퍼런스를 리턴할 수도 있다는 의미 아닌가요? ■

Answer:

네, 그럴 수 있습니다! 이게 왜 hashtable_lookup() 이 RCU read-side 크리티컬 섹션 안에서 호출되어야만 하는지에 대한 이유이고, 왜 hashtable_add() 와 hashtable_del() 이 또한 RCU 를 신경쓰는 리스트 조작 기능들을 사용해야 하는가에 대한 이유입니다. 마지막으로, 이게 바로 hashtable_del() 가 삭제한 원소를 메모리에서 해제시키기 전에 (ex: synchronize_rcu() 를 호출해서) grace period 를 기다려야 하는지에 대한 이유입니다. □

Quick Quiz 10.6:

8개 CPU 에서 60 CPU 상황을 쓰레드 수를 늘리는 것으로 시뮬레이션 해보는건 상당히 위험함이 Section 10.2.3 에서 드러났었습니다. 하지만 60 CPU 를 가지고 더 많은 CPU 상황을 시뮬레이션 해보는건 왜 안전할 수 있을까요? ■

Answer:

그건 안전하지 않고, 유용한 방법은 그런 프로그램은 더 큰 시스템에서 돌리는 것입니다. 그렇다고는 하나, 다른 테스트는 RCU read-side 기능들은 1024 쓰레드까지도 일관된 성능과 확장성을 제공하는 것을 보였습니다. □

Quick Quiz 10.7:

Figure 10.25 의 코드는 해시 값을 두번 계산하네요! 왜 이렇게 비효율적이죠? ■

Answer:

기준의 해시 테이블과 새로운 해시 테이블은 전혀 다른 해시 함수를 사용할 수도 있고, 따라서 기준의 테이블을 위해 계산된 해시 값은 새로운 테이블에서는 적절치 않을 수 있기 때문입니다. □

Quick Quiz 10.8:

Figure 10.25 의 코드는 선택된 bucket에서 진행중일 수도 있는 크기 재조정 프로세스로부터의 보호는 어떻게 하나요? ■

Answer:

그런 보호는 전혀 이루어지지 않고 있습니다. 그건 뒤에서 설명될 업데이트 쪽 동시성 제어 함수에서 할 일입니다. □

Quick Quiz 10.9:

Figures 10.25 와 10.26 의 코드는 업데이트를 위해 해시 값을 계산하고 bucket 선택 로직을 두번 수행하네요! 왜 이렇게 비효율적인거죠? ■

Answer:

이 방법은 `hashtorture.h` 테스트 도구가 재사용될 수 있게 합니다. 그렇다고는 하지만, 제품 품질의 크기 재조정 가능한 해시 테이블은 이런 두번의 계산이 없도록 최적화 되어야 할겁니다. 그런 최적화를 하는건 독자 여러분의 몫으로 남겨둡니다. □

Quick Quiz 10.10:

크기 재조정 작업이 이루어지는 사이에 한 쓰레드가 새로운 해시 테이블에 원소를 넣는다고 생각해 봅시다. 뒤따르는 크기 재조정 작업이 이 삽입 작업 전에 완료됨으로 인해 이 삽입 작업이 없던 것처럼 되어버리는 문제는 어떻게 방지되고 있나요? ■

Answer:

두번째 크기 재조정 작업은 해당 bucket에서 삽입 작업이 이루어지고 있는 장소로 움직일 수 없을텐데, 삽입 작업이 새로운 해시 테이블(이 예에서의 세개의 해시 테이블 중 두번째)의 bucket 중 하나의 락을 잡고 있기 때문입니다. 더 나아가서, 이 삽입 작업은 RCU read-side 크리티컬 섹션 안에 위치하고 있습니다. `hashtab_resize()` 함수에서 봤듯이, 이 말은 첫번째 크기 재조정 작업이 삽입의 read-side 크리티컬 섹션에 완료되기를 `synchronize_rcu()` 를 이용해 기다리고 있을 것을 의미합니다. □

Quick Quiz 10.11:

Figure 10.27 의 `hashtab_lookup()` 함수에서, 코드는 탐색될 원소가 이미 동시의 크기 재조정 작업에 의해 분산되었다면 새로운 해시 테이블에 있는 올바른 bucket을 찾아옵니다. 이건 RCU로 보호되는 탐색에서는 좀 낭비가 있는 것 같은데요. 이 경우에는 왜 그냥 기존 해시 테이블에서 작업을 끝내지 않는 거죠? ■

Answer:

크기 재조정 작업이 시작되었고 기존 테이블의 bucket

들 중 절반만 새로운 테이블로 분산했다고 생각해 보세요. 더 나아가서 한 쓰레드가 이미 분산된 bucket 안에 원소를 하나 넣었고, 이 쓰레드가 이제 방금 새로 추가한 원소를 탐색한다고 생각해 봅시다. 탐색이 무조건적으로 기존 해시 테이블을 탐색한다고 하면, 이 쓰레드는 자신이 집어넣은 원소를 찾는데 실패할텐데, 이는 저한테는 분명한 버그처럼 들리는군요! □

Quick Quiz 10.12:

Figure 10.27 의 `hashtab_del()` 함수는 원소를 항상 기존 해시 테이블에서 제거하지는 않는데요. 이 말은 읽기 쓰레드들이 이 새로 제거된 원소를 해제된 후에도 볼 수 있다는 의미 아닌가요? ■

Answer:

아닙니다. `hashtab_del()` 함수는 이제 제거된 원소를 담고 있는 bucket에 대해 크기 재조정 작업이 이미 완료되었을 때에만 기존 해시 테이블에서 제거합니다. 하지만 이는 새로운 `hashtab_lookup()` 오퍼레이션은 해당 원소를 탐색할 때 새로운 해시 테이블을 사용할 것을 의미합니다. 따라서, `hashtab_del()` 보다 먼저 시작된 `hashtab_lookup()` 오퍼레이션들만이 이번에 제거된 원소를 만날 수 있을 겁니다. 이는 `hashtab_del()` 는 이 불편한 `hashtab_lookup()` 을 막기 위해 하나의 RCU grace period 만을 기다리면 된다는 것을 의미합니다. □

Quick Quiz 10.13:

Figure 10.27 의 `hashtab_resize()` 함수에서, line 29에서의 `->ht_new` 로의 업데이트가 line 36에서의 `->ht_resize_cur` 로의 업데이트 전에 일어난 것으로 `hashtab_lookup()`, `hashtab_add()`, 그리고 `hashtab_del()` 의 관점에 보일 것을 무엇이 보장하죠? ■

Answer:

Figure 10.27 의 line 30에서의 `synchronize_rcu()` 는 전부터 존재한 RCU 읽기 쓰레드들이 line 29에서 새로운 해시 테이블로의 레퍼런스를 설치한 시점과 line 36에서 `->ht_resize_cur`를 업데이트한 시점 사이에서 완료되었을 것을 보장합니다. 이 말은 `->ht_resize_cur`의 음이 아닌 값을 본 모든 읽기 쓰레드들은 `->ht_new` 할당 전에 시작되었을 수 없고, 따라서 새로운 해시 테이블로의 레퍼런스를 볼 수 있어야 합니다. □

Quick Quiz 10.14:

`hashtorture.h` 의 코드는 `hashtab_lock_mode()` 가 `ht_get_bucket()` 의 기능을 포섭하도록 수정될 수 없나요? ■

Answer:

그럴 수도 있을테고, 그렇게 하는게 이 챕터에 선보여진 bucket 별 락킹을 사용하는 해시 테이블에 이득을 가져다 줄 수 있을 겁니다. 이런 변경을 만드는 건 독자 여러분의 몫으로 두겠습니다. □

Quick Quiz 10.15:

이런 특수화는 정말로 얼마나 성능을 구하나요? 이게 정말 가치가 있나요? ■

Answer:

첫번째 질문에 대한 답은 독자의 몫으로 남겨져 있습니다. 크기 재조정 가능한 해시 테이블을 특수화 시켜보고 그 결과로 성능이 얼마나 개선되는지 살펴보세요. 두번째 질문은 일반적으로는 답변될 수가 없습니다만, 대신 특수한 사용 예에 맞춰서 답변될 수 있을 겁니다. 일부 사용 케이스는 성능과 확장성에 극단적으로 민감하고 다른 것들은 그보다 덜할 수 있습니다. □

D.11 Validation

Quick Quiz 11.1:

컴퓨팅에 있어서 단편적 계획을 따르는게 특히 중요한 건 언제인가요? ■

Answer:

얼마든지 많은 상황이 존재할 수 있습니다만, 가장 중요한 상황은 그 프로그램을 개발하기 위해 조립해야 할 것들을 아무도 만든게 없는 상황이 되겠습니다. 이런 경우, 믿을만한 계획을 만들어내는 유일한 방법은 그 프로그램을 구현하고, 계획을 만들고, 이걸 또다시 두번째로 구현하는 것 뿐입니다. 하지만 누가 됐든 처음으로 그 프로그램을 구현하는 사람은 단편적 계획을 따르는 것밖에 다른 선택이 없는데 현실을 알지 못하고 만들어진 자세한 계획은 실제 세계에서는 살아남을 수가 없기 때문입니다.

그리고 아마도 이게 단편적 계획을 따를 수 있을만큼 미친듯이 낙관적인 인류를 진화가 선택한 이유 가운데 하나일 겁니다. □

Quick Quiz 11.2:

다음과 같은 `time` 커맨드의 출력을 처리하는 스크립트를 작성하고 있다고 해봅시다:

```
real      0m0.132s
user      0m0.040s
sys       0m0.008s
```

해당 스크립트는 에러를 처리하고 에러일 수 있는 `time` 출력을 받았을 경우 적절한 진단 내용을 제공하기 위해 자신에게 주어진 입력을 체크해야 합니다. 싱글

쓰레드 프로그램들로 생성된 `time` 출력의 사용에 대해 이 프로그램을 테스트하기 위해 어떠한 입력의 테스트를 제공해야 할까요? ■

Answer:

- 모든 시간이 CPU-bound 프로그램에 의해 user 모드에서 소비되는 경우를 위한 테스트 케이스가 있습니까?
- 모든 시간이 CPU-bound 프로그램에 의해 system 모드에서 소비되는 경우를 위한 테스트 케이스가 있습니까?
- 세개의 시간이 모두 0인 경우를 위한 테스트 케이스가 있습니까?
- “user” 와 “sys” 시간의 합이 “real” 시간보다 큰 경우를 위한 테스트 케이스가 있습니까? (멀티쓰레드 프로그램에 있어서는 이건 물론 완전히 합법적인 결과입니다.)
- 시간들 중 하나가 1초 이상인 경우를 위한 테스트 케이스 집합이 있습니까?
- 시간들 중 하나가 10초 이상인 경우를 위한 테스트 케이스 집합이 있습니까?
- 시간들 중 하나가 0이 아닌 분을 갖는 시간을 갖는 경우를 위한 테스트 케이스 집합이 있습니까? (예를 들어, “15m36.342s”.)
- 시간들 중 하나가 60보다 큰 두번째 값을 갖는 경우를 위한 테스트 케이스 집합이 있습니까?
- 시간들 중 하나가 밀리세컨드에 있어 32 비트를 오버플로우 하는 경우를 위한 테스트 케이스 집합이 있습니까? 64 비트의 밀리세컨드는요?
- 시간들 중 하나가 음수인 경우를 위한 테스트 케이스 집합이 있습니까?
- 시간들 중 하나가 양수의 분 값을 갖지만 음수의 두번째 값을 갖는 경우를 위한 테스트 케이스 집합이 있습니까?
- 시간들 중 하나가 “m”이나 “s”를 누락하고 있는 경우를 위한 테스트 케이스 집합이 있습니까?
- 시간들 중 하나가 숫자가 아닌 경우를 위한 테스트 케이스 집합이 있습니까? (예를 들어, “Go Fish”.)
- 입력의 열들 중 하나가 누락된 경우를 위한 테스트 케이스 집합이 있습니까? (예를 들어, “real” 값과 “sys” 값은 있지만 “user” 값은 없는 경우.)

15. 입력의 열들 중 하나가 중복된 경우를 위한 테스트 케이스 집합이 있습니까? 또는 중복되었지만 그 중복된 내용은 다른 시간 값을 갖는 경우를 위한?
16. 입력의 특정 열이 하나 이상의 시간 값을 갖는 경우를 위한 테스트 케이스 집합이 있습니까? (예를 들어, “real 0m0.132s 0m0.008s”.)
17. 무작위적 문자를 담고 있는 경우를 위한 테스트 케이스 집합이 있습니까?
18. 잘못된 입력에 관련된 모든 테스트 케이스들은 모든 순열을 생성합니까?
19. 각각의 테스트 케이스에 대해서 그 테스트에 예상되는 결과를 가지고 있습니까?

위의 상당한 수의 테스트 케이스들을 위한 테스트 데이터를 만들지 않았다면, 당신은 더 높은 품질의 테스트들을 만들 수 있는 기회를 갖기 위해서는 더 분해적인 자세를 가져야 합니다.

물론, 분해를 효율적으로 하는 한가지 방법은 테스트되어야 하는 소스 코드와 함께 테스트를 생성하는 것인데, 이는 white-box 테스트라고 불립니다 (black-box 테스트와 반대되는 개념입니다). 하지만, 이게 만병통치는 아닙니다: 당신의 생각이 그 프로그램이 뭘 다룰 수 있는지에만 제한되어 있어서 정말로 분해적인 입력을 생성하는데 실패하게 되는 경우를 발견하기는 매우 쉽다는 걸 알게 될겁니다. □

Quick Quiz 11.3:

지금 저에게 코딩을 시작하기도 전에 검증을 시작하고 말씀하시는 거예요??? 그건 마치 아무것도 시작하지 않는 훌륭한 방법처럼 들리네요!!! □

Answer:

그게 당신의 프로젝트라면, 예를 들어 취미라면, 원하는 대로 하세요. 당신이 낭비하는 모든 시간은 당신만의 것이고, 그걸 신경쓸 어떤 사람도 존재하지 않습니다. 그리고 시간이 전혀 낭비되지 않을 수 있는 좋은 기회도 있지요. 예를 들어, 당신이 세상에 없던 최초의 종류의 프로젝트에 착수했다면, 요구사항들은 어떤 방법으로도 알 수 없는 것들일 겁니다. 이 경우, 최선의 방법은 빠르게 여러개의 대략적 해결책의 프로토타입들을 만들어 보고, 그걸 사용해 보고, 뭐가 가장 잘 동작하는지 보는 겁니다.

다른 한편으로, 당신이 이미 존재하는 시스템과 전체적으로 유사한 시스템을 만들도록 돈을 받았다면, 당신은 당신의 사용자인 고용인에게 빚을 지고 있는 것이고 당신의 미래는 빨리 그리고 자주 검증해야 하는 것입니다. □

Quick Quiz 11.4:

WARN_ON_ONCE() 는 어떻게 구현할 수 있을까요? □

Answer:

가끔은 두번이나 세번까지는 경고를 날릴 수 있는 WARN_ON_ONCE() 여도 문제가 없다면, 간단히 초기값 0을 갖는 static 변수를 사용하세요. 조건이 만족된다면, 이 static 변수를 체크하고, 만약 0이 아니라면, 그냥 리턴합니다. 그렇지 않다면, 그 값을 1으로 설정하고, 메세지를 출력한 후, 리턴하세요.

경고 메세지가 굉장히 커다랗거나 해서 메세지가 절대로 두번 이상 나타나선 안된다면 앞의 “그 값을 1로 설정” 부분을 어토믹 교환 오퍼레이션을 사용할 수 있습니다. 경고 메세지는 이 어토믹 교환 오퍼레이션이 0을 리턴한 경우에만 프린트하도록 합니다. □

Quick Quiz 11.5:

어떤 사람이 존재하는 코드를 종이에 펜으로 사본을 만들려 하겠어요??? 그건 그냥 필사 과정에서의 에러의 가능성만 증가시키는 거 아닌가요? □

Answer:

필사 과정에서의 에러가 걱정된다면, 제가 당신에게 `diff`라는 이름의 정말 멋진 도구를 처음으로 소개해 줄 수 있게 해주세요. 또한, 필사 과정을 진행하는 건 상당히 가치있을 수 있습니다:

1. 많은 코드를 복사하게 된다면 당신은 추상화를 함으로써 얻을 수 있는 이득을 얻을 수 없을 겁니다. 코드를 필사하는 행위는 추상화를 위한 커다란 동기 부여를 줄 수 있습니다.
2. 코드를 필사하는 행위는 코드가 정말로 새로운 환경에서도 잘 동작할 것인지 생각해볼 기회를 줍니다. 인터럽트를 불가능하게 해야 한다거나 어떤 락을 잡아야 한다거나와 같은, 명확치 않은 제약이 있나요?
3. 코드를 필사하는 행위는 또한 그 일을 하게 하는 데 어떤 더 나은 방법이 있을지 고민해 볼 시간을 줍니다.

그러니까, 그래요, 코드를 필사하세요! □

Quick Quiz 11.6:

이 과정은 우스꽝스러울 정도로 지나치게 공업화 되어 있어요! 어떻게 당신은 합리적인 양의 소프트웨어들이 이런 방식으로 작성되었을 거라고 생각할 수 있는거죠? □

Answer:

실제로, 코드를 손으로 복사하는 행위를 반복하는 것은 노동집약적이고 느립니다. 하지만, 상당한 스트레스

테스트와 정확성 검증과 조합되어진다면, 궁극적인 성능과 안정성이 필요하고 디버깅이 어려운 복잡한 병렬 코드에 있어서는 이 또한 상당히 효과적입니다. 리눅스 커널 RCU 구현이 그런 케이스입니다.

한편으로는, 당신이 어떤 데이터를 조작하기 위해 간단한 싱글 쓰레드 셀 스크립트를 작성하고 있다면, 다른 방법론이 가장 잘 당신을 도와줄 겁니다. 예를 들어, 당신이 원하는 대로 동작하고 있음을 확신할 수 있도록 하는 테스트 데이터와 함께 커맨드 하나 하나를 하나씩 대화형 셀에 입력하고, 그리고 나서 성공한 커맨드들을 스크립트 안에 복사해서 붙여넣을 수 있을 겁니다. 마지막으로, 전체 스크립트를 테스트하세요.

도움을 주려 하는 친구나 동료가 있다면, 페어 프로그래밍이 잘 될 수 있을텐데, 정식적인 디자인 리뷰와 코드리뷰 프로세스에 있어서도 마찬가지입니다.

그리고 당신이 취미로 코드를 작성하고 있다면, 뭐가 됐든 하고 싶은대로 하세요.

짧게 요약해서, 다른 종류의 소프트웨어는 다른 개발 방법론을 필요로 합니다. □

Quick Quiz 11.7:

당신의 쓰레기 봉투에 굉장히 많은 시스템들이 있다고 생각해 봅시다. 예를 들어, 현재의 클라우드 가격 정책대로라면, 당신은 합리적으로 낮은 가격에 수많은 CPU 시간을 구입할 수 있습니다. 모든 실용적인 목적이 있어서의 정확성을 충분히 가져가기 위해서 이 전략을 사용하지 않나요? ■

Answer:

이 방법은 당신의 검증 무기고에 가치있는 추가물품이 될 수도 있을 겁니다. 하지만 이 방법은 몇 가지 한계점을 가지고 있습니다:

1. 일부 버그들은 극단적으로 낮은 발생 가능성을 가지고 있지만, 더도 아니고 덜도 아니고 고쳐져야 합니다. 예를 들어, 리눅스 커널의 RCU 구현이 평균적으로 100년의 CPU 시간에 딱 한번 발생하는 버그를 가지고 있다고 생각해 봅시다. 100년의 CPU 시간은 가장 짠 클라우드 플랫폼에서라고 해도 상당히 비싼 가격입니다만, 2011년의 세계에서 1억 개의 리눅스 인스턴스 위에서라면 이 버그는 하루에도 2,000 회 이상의 문제를 일으킬 수 있을 것으로 예상할 수도 있습니다.
2. 버그는 당신의 테스트 환경에서는 발생 가능성이 없을 수 있는데, 이는 당신이 테스트를 하는데에 얼마나 많은 시간을 쏟아붓는가와 관계없이 당신은 그 버그를 볼 수 없을 것이란 것을 의미합니다.

물론, 당신의 코드가 충분히 작다면 Section 12에서와 같이, 정식적인 검증 절차가 도움이 될 수 있을 겁니다. 하지만 주의하세요: 당신의 코드의 정식적 검증은 당신의 가정, 요구사항에 대한 잘못된 이해, 당신이 사용하는 소프트웨어나 하드웨어에 대한 잘못된 이해, 또는 당신이 증명을 할 생각을 하지 못한 에러에 대해서는 문제를 찾아내지 못할 겁니다. □

Quick Quiz 11.8:

뭐라구요??? 제가 앞의, 각각 10% 실패 확률을 갖는 다섯번의 테스트의 예를 이 공식에 집어넣어보면 59,050%를 얻게 되는데 이건 말이 안되잖아요!!! ■

Answer:

당신 말이 맞아요, 전혀 말이 안되죠.

확률은 0과 1 사이의 숫자여서 당신은 확률을 구하기 위해서는 퍼센티지를 100으로 나눠야 함을 기억하세요. 따라서 10%의 확률은 0.1로, 공식에 대입한 확률은 0.4095 값을 얻게 되는데, 이는 41%로 반올림 되는데, 이렇게 되면 앞의 결과와 상당히 일관적이지요. □

Quick Quiz 11.9:

Equation 11.6에서, 로그의 밑은 10인가요, 2인가요, 또는 e 인가요? ■

Answer:

그건 상관 없습니다. 어떤 밑의 로그를 취하는가에 관계없이 똑같은 답을 얻을 수 있을텐데 그 결과는 로그 값들 간의 비율이기 때문입니다. 유일한 제약은 분모와 분자에 둘다 같은 밑을 사용해야 한다는 점입니다. □

Quick Quiz 11.10:

어떤 버그가 평균적으로 시간당 세번의 테스트 실패를 유발한다고 생각해 봅시다. 수정사항이 실패의 확률을 상당히 줄였음을 99.9% 증명할 수 있는 증거를 위해서는 테스트는 에러 없이 얼마나 오래 돌아가야만 할까요? ■

Answer:

Equation 11.11에서 n 은 3으로, 그리고 P 를 99.9로 놓으면, 결과는 다음과 같습니다:

$$T = -\frac{1}{3} \log \frac{100 - 99.9}{100} = 2.3 \quad (\text{D.9})$$

테스트가 실패하지 않은채 2.3 시간동안 돌아간다면, 우린 해당 수정사항이 실패 가능성을 줄였음을 99.9% 확신할 수 있습니다. □

Quick Quiz 11.11:

이 모든 계승식 (factorial) 들과 지수식 (exponential) 들을 더하고 있는건 진짜 고통스럽습니다. 좀 더 편한 길은 없나요? ■

Answer:

한가지 방법은 “maxiam” 라는 이름의 오픈 소스 기호논리학 조작 프로그램을 사용하는 것입니다. 많은 Debian 기반 리눅스 배포본의 일부분인 이 프로그램을 일단 설치하면, 이걸 수행하고 `load(distrib)`; 커맨드를 넣고 이어서 `bfloor(cdf_poisson(m, 1))`; 커맨드들의 숫자들을 아무거나 넣을 수 있는데, 여기서 `m`은 m 을 위한 원하는 값을 봐꾸고 1은 원하는 λ 값으로 바꿔야 할 겁니다.

구체적으로는, `bfloor(cdf_poisson(2, 24))`; 커맨드는 1.181617112359357b-8 을 내놓는데, 이는 Equation 11.13 로 구해진 값과 맞아 떨어집니다.

대안적으로는, Section 11.6.2 에서 설명하는 rough-and-ready 방법을 사용할 수 있습니다. □

Quick Quiz 11.12:

하지만 잠시만요!!! 실패들의 어떤 횟수가 있어야 한다는 (실패가 없을 수 있다는 가능성은 포함해서) 점을 놓고 보면, Equation 11.13 에 보여진 더하기는 m 을 1로 놓으면 무한대가 되지 않나요? ■

Answer:

실제로 그래야 합니다. 그리고 그렇지요.

이를 알아보기 위해, $e^{-\lambda}$ 는 i 에 의존되지 않는데, 이는 다음과 같이 더하기에서 빠져나올 수 있음을 의미함을 보세요:

$$e^{-\lambda} \sum_{i=0}^{\infty} \frac{\lambda^i}{i!} \quad (\text{D.10})$$

남은 더하기는 정확히 e^{λ} 를 위한 Taylor 시리즈로, 다음과 같이 됩니다:

$$e^{-\lambda} e^{\lambda} \quad (\text{D.11})$$

이 두개의 지수식들은 서로 상반되므로 무시되어서 정확히 요구된 대로 1 이 됩니다. □

Quick Quiz 11.13:

만약 그 오염이 어떤 관계없는 포인터에 영향을 끼쳐서 그 포인터를 따라가서 오염을 일으키는 경우에 이 방법은 어떻게 도움을 줄 수 있을까요??? ■

Answer:

실제로, 그런 상황이 일어날 수 있습니다. 많은 CPU 들이 그런 관계없는 포인터를 찾아내는 것을 도울 수 있는 하드웨어 디버깅 기능을 가지고 있습니다. 뿐만 아니라,

core dump 를 가지고 있다면, 메모리의 오염된 영역을 가리키는 포인터들을 위해 core dump 를 조사할 수 있습니다. 또한 오염의 데이터 구조를 살펴보고, 그 구조와 맞아떨어지는 포인터들을 체크해 볼수도 있습니다.

또한 한발자국 떨어져서 프로그램을 구성하는 모듈들을 더 혹독하게 테스트 해볼 수 있는데, 이는 오염을 그에 책임이 있는 모듈로 국한시킬 겁니다. 만약 이게 오염을 사라지게 한다면, 각각의 모듈에서 노출된 함수들의 인자 체크를 더 추가해 보는 걸 고려해 보세요.

더도 아니고 덜도 아니고, 이건 어려운 문제인데, 그게 제가 “약간 어둠의 예술” 이라는 말을 사용한 이유입니다. □

Quick Quiz 11.14:

하지만 제가 그렇게 이등분을 통한 탐색을 해봤는데, 결국 나온 범인은 거대한 커밋이었어요. 이제 뭘 해야 하죠? ■

Answer:

커다란 커밋이요? 그건 참 부끄러운 일입니다! 이건 왜 당신이 커밋들을 작게 유지할 것이라 기대되는지에 대한 한가지 이유일 뿐입니다.

그리고 그게 당신의 답입니다: 해당 커밋을 바이트 단위의 조각들로 쪼개고 그 조각들을 다시 이진탐색하세요. 제 경험상으로, 그 커밋을 쪼개는 일 자체는 대부분의 경우에 그 버그를 상당히 분명하게 만들기에 충분합니다. □

Quick Quiz 11.15:

이미 존재하는 조건적 락킹 도구들은 이런 가짜 실패 기능을 제공하지 않는 이유가 뭔가요? ■

Answer:

조건적 락킹 도구들이 진실만을 말한다는 사실에 의존적인 락킹 알고리즘들이 존재합니다. 예를 들어, 만약 조건적 락 실패가 어떤 다른 쓰레드가 이미 어떤 일을 하고 있음을 나타낸다면, 가짜 실패는 그 일은 결코 완료가 되지 못하게 만들 수 있어서, 결과적으로 프로그램 수행이 한 구간에서 나아가지 못하게 될겁니다. □

Quick Quiz 11.16:

그건 웃긴 이야기네요!!! 어쨌든, 좀 늦더라도 올바른 답을 얻는게 잘못된 답을 얻는 것보다는 낫지 않겠어요??? ■

Answer:

이 질문은 아예 답을 계산하지 않는, 그리고 그렇게 함으로써, 또한 그 답을 계산하는데 드는 비용을 고려하게 만드는 선택지를 고려하게 만듭니다. 예를 들어, 짧은 기간의 날씨 예측과 같은, 정확한 모델들이 존재하지만

정말로 그 날씨가 와버리기보다 먼저 그 모델을 수행하기를 원한다면 커다란 (그리고 비싼) 클러스터로 구성된 슈퍼컴퓨터들을 필요로 하는 경우를 생각해 보세요.

그리고 이 경우에, 그 모델이 실제 날씨보다 빠르게 수행되는 것을 방지하는 모든 성능 버그는 모든 날씨 예측을 방지해 버립니다. 그 커다란 클러스터로 구성된 슈퍼컴퓨터들을 구입한 목적은 모두 날씨를 예측하기 위해서였다는 점을 생각해 보면, 그 모델을 날씨보다 빠르게 수행하지 못한다면, 그 모델을 아예 수행하지 않는 편이 나을 겁니다.

안전성에 민감한 리얼타임 컴퓨팅 분야에서는 더 많은 가혹한 예들이 발견될 수도 있을 겁니다. □

Quick Quiz 11.17:

하지만, 어플리케이션을 병렬화하기 위해 필요한 모든 어려운 일을 해내기로 했다면 왜 그걸 올바르게 하지 않는거죠? 왜 최적의 성능과 선형적인 확장성보다 못한 것에 안주하는 겁니까? ■

Answer:

전 정말로 당신의 정신과 포부에 경의를 표합니다만, 당신은 프로그램의 완료에 걸리는 딜레이가 높은 비용이 될수도 있다는 점을 잊고 있습니다. 극단적인 예를 들어보면, 단일 쓰레드로 짜인 어플리케이션에서의 40%의 성능 저하가 매일 한 사람을 죽게 만든다고 생각해 보세요. 더 나아가서 당신이 사람들과 함께 여덟개의 CPU를 가진 시스템에서 순차적 버전에 비해서 50% 빠르게 동작하는 병렬 프로그램을 하루만에 빠르고 지저분하게 만들어 냈지만(hack), 최적의 병렬 프로그램은 네달동안의 고통스러운 설계, 코딩, 디버깅, 그리고 최적화를 필요로 한다고 상상해 보세요.

백명이 넘는 사람들은 그 빠르고 더럽게 만들어진 버전을 더 좋아할 거라고 충분히 말할 수 있습니다. □

Quick Quiz 11.18:

하지만, 예를 들어 캐시와 메모리 배치 사이의 간섭에 의한 것과 같은 다른 에러 유발 원인들은 어떻게 하죠? ■

Answer:

메모리 배치의 변경은 실제로 비현실적으로 수행 시간을 줄여버리는 결과를 초래할 수 있습니다. 예를 들어, 어떤 마이크로벤치마크는 거의 항상 L0 캐시의 associativity를 오버플로우 시키지만, 올바른 메모리 배치에서 만큼은 모두 맞아떨어진다고 생각해보세요. 이게 정말 문제라면, 메모리 배치를 완전히 제어할 수 있도록 마이크로벤치마크를 *huge page*를 사용해서 수행해보는 방안 (또는 커널에서 수행하거나 기계 위에서 곧바로) 을 생각해 보세요. □

Quick Quiz 11.19:

테스트 되는 코드를 격리시키기 위해 제안된 이 테크닉들은 특히나 그 코드가 커다란 어플리케이션에서 돌아가고 있다면 그 코드의 성능에 영향을 끼치지 않을까요? ■

Answer:

실제로 그럴 수도 있습니다. 마이크로 벤치마킹을 위한 과정에서 대부분은 테스트하고자 하는 코드를 그 어플리케이션에서 끄집어 낼테지만요. 더도 아니고 덜도 아니고, 어떤 이유로 테스트 되는 코드를 그 어플리케이션 내에 두어야만 한다면, Section 11.7.6에서 언급된 테크닉들을 사용해야 할 겁니다. □

Quick Quiz 11.20:

이 방법은 좀 이상하군요! 왜 우리가 통계 수업시간에 배웠던 것처럼 평균과 표준편차를 사용하지 않죠? ■

Answer:

평균과 표준편차는 이런 일을 하라고 만들어진게 아니기 때문입니다. 이를 확인하기 위해, 다음의 데이터 집합에 평균과 표준편차를 적용하게 되면, 1%의 측정 에러가 나옵니다:

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

문제는 평균과 표준편차는 측정 에러에 대한 가정을 전혀 하고 있지 않다는 것이고, 따라서 49,500 근처의 값과 49,900 근처의 값들 사이의 차이가 통계적으로 상당하다고 생각하게 될것인데, 실은 추정된 측정 에러의 한계선 안에 있습니다.

물론, 절대적인 차이보다 표준편차를 사용해서 비슷한 효과를 얻을 수 있도록 Figure 11.7 와 비슷한 스크립트를 만드는 것도 가능할 것인데, 이는 흥미 있는 독자 여러분의 몫으로 남겨두겠습니다. 동일한 데이터 값의 문자열들로부터 나타날 수 있는 divide-by-zero 에러를 없앨 수 있도록 조심하세요! □

Quick Quiz 11.21:

하지만 신뢰되는 데이터 그룹의 y축 값들이 모조리 0이면 어떡하죠? 그러면 스크립트는 0이 아닌 값들을 제거하지 않을까요? ■

Answer:

실제로 그럴 겁니다! 하지만 당신의 성능 측정이 자주 정확히 0이란 값을 내놓는다면, 당신의 성능 측정 코드를 좀 더 자세히 들여다 볼 필요가 있을 수 있습니다.

평균과 표준편차에 기반하는 많은 방법들이 이런 종류의 데이터 집합에 대했 비슷한 문제를 가지고 있음을 알아두시기 바랍니다. □

D.12 Formal Verification

Quick Quiz 12.1:

왜 locker 에 미치지 못한 statement 가 있는 거죠? 이건 전체 상태-공간 탐색이 아니었나요? ■

Answer:

locker 프로세스는 무한 루프이므로, 이 프로세스의 종료까지 제어가 끊지를 않습니다. 하지만, 단조적으로 증가되는 변수가 존재하지 않기 때문에, Promela 는 이 무한 루프를 작은 수의 상태만 가지고도 모델링 할수가 있습니다. □

Quick Quiz 12.2:

이 예제에 있어서 Promela 코딩 스타일 문제들은 뭐가 있나요? ■

Answer:

몇가지가 있습니다:

1. sum 의 선언은 init 블락 안으로 옮겨져야 하는데, 그 외의 곳에서는 어디서도 사용되지 않기 때문입니다.
2. 단정문 코드는 초기화 루프 바깥으로 옮겨져야 합니다. 그렇게 되면 초기화 루프는 하나의 어토믹 블락 안에 위치할 수 있어서, 상태 공간을 훨씬 줄일 수 있습니다(얼마나 줄일 수 있을까요?).
3. 단정문 코드를 감싸고 있는 어토믹 블락은 sum 과 j 의 초기화를, 그리고 단정문도 포함하도록 확장되어야 합니다. 이것 역시 상태 공간을 줄일 것입니다(이번에도, 얼마나 줄일 수 있을까요?). □

Quick Quiz 12.3:

이 do-od 문을 좀 더 간단하게 코딩하는 방법은 없을까요? ■

Answer:

있습니다. 이걸 if- fi 로 바꾸고 break 문을 없애버리세요. □

Quick Quiz 12.4:

왜 line 12-21 과 line 44-56 에 어토믹 블락이 있나요,

그 어토믹 블락들 안의 오퍼레이션들은 현존하는 제품화된 마이크로프로세서 중 어떤 것도 어토믹한 구현을 제공하지 않는데도 말이죠? ■

Answer:

그 오퍼레이션들은 단정문을 위한 것들일 뿐이기 때문입니다. 그것들은 알고리즘 자체를 위한 것이 아닙니다. 따라서 그것들을 어토믹으로 표시해도 문제가 되지 않고, 따라서 그것들을 어토믹으로 표시하는 것은 Promela 모델을 통해 탐색되어야 하는 상태 공간을 굉장히 줄여주게 됩니다. □

Quick Quiz 12.5:

Line 24-27에서 카운터들을 다시 합해보는 것은 정말로 필요한 건가요? ■

Answer:

그렇습니다. 이걸 확실히 보기 위해서, 이 라인들을 지우고 모델을 돌려보세요.

대안적으로는, 다음과 같은 단계들을 생각해 보세요:

1. 한 프로세스가 RCU read-side 크리티컬 섹션 안에 있어서, ctr[0] 의 값은 0이고 ctr[1] 은 2입니다.
2. 한 업데이트 쓰레드가 수행을 시작하고, 카운터들의 합이 2임을 보게 되고 따라서 빠른 수행 경로는 실행될 수 없습니다. 따라서 락을 잡습니다.
3. 두 번째 업데이트 쓰레드가 수행을 시작하고, ctr[0] 을 값 0을 가져옵니다.
4. 첫 번째 업데이트 쓰레드가 ctr[0] 에 1을 더하고, 인덱스를 바꾸고 (이제 0이 됩니다), ctr[1] 에서 1을 뺍니다 (이제 1이 됩니다).
5. 두 번째 업데이트 쓰레드가 ctr[1] 의 값을 가져오는데, 지금은 1입니다.
6. 두 번째 업데이트 쓰레드는 이제 원래의 읽기 쓰레드가 아직 완료되지 않았음에도 빠른 수행 경로를 진행해도 된다고 잘못된 결론을 내리게 됩니다. □

Quick Quiz 12.6:

여기에서 설명된 QRCU 알고리즘의 정확성에 대한 두개의 독립적인 증명을 가지고 있고, 올바르지 않음에 대한 증명이 다른 알고리즘은 어떤지를 다루고 있는데, 왜 여전히 의문의 여지가 남는 거죠? ■

Answer:

항상 의문의 여지가 존재합니다. 이 경우에는, 정확성에

대한 두개의 증명이 실제 세계 메모리 모델들을 신경쓰지 않고 있으므로, 이 두개의 증명들은 잘못된 메모리 순서규칙 가정에 기반하고 있을 수 있다는 점을 명심하시기 바랍니다. 더욱이, 두 증명 모두 같은 사람에 의해 만들어졌으므로, 동일한 에러를 포함하고 있을 가능성이 농후합니다. 다시 말하지만, 항상 의심의 여지가 존재합니다. □

Quick Quiz 12.7:

와우, 거 참 대단하네요! 이제, 저는 40GB 의 메인 메모리를 가진 기계가 없다면 뭘 해야 하는거죠??? ■

Answer:

진정하세요, 이 질문에 대한 많은 정당한 답들이 있습니다:

1. 해당 모델을 더 최적화 해서, 메모리 소비량을 줄이세요.
2. 리눅스 커널의 코드에 있는 주석부터 시작해서 연필과 종이 증명 방법을 실행하세요.
3. 비록 코드의 정확성을 증명할 수는 없지만 숨겨진 버그들을 찾아줄 수 있는, 신중히 생각한 고문 테스트들을 고안하세요.
4. 작은 기계들의 클러스터들을 사용해서 모델 체크를 하는 도구들을 만들고 사용하려는 움직임이 일부 있습니다. 하지만, Paul은 사용할 수 있는 일부 커다란 기계들로 인해서 그런 도구들을 직접 사용해 본 적은 없음을 알아두시기 바랍니다.
5. 당신의 문제를 적용하기에 맞는 크기의 메모리 크기를 가진, 구매할 만한 각겨대의 시스템이 나오길 기다리세요.
6. 짧은 시간 동안 커다란 시스템을 대여하기 위해 클라우드 컴퓨팅 서비스들 가운데 하나를 고려해 보세요. □

Quick Quiz 12.8:

왜 간단하게 `rcu_update_flag` 의 값을 증가시키고, `rcu_update_flag` 의 기존 값이 0일 경우에만 `dynticks_progress_counter` 의 값을 증가시키는 방식을 사용하지 않는거죠??? ■

Answer:

이는 NMI의 존재 시에 실패합니다. 이를 자세히 보기 위해, `rcu_irq_enter()` 가 `rcu_update_flag` 를 증가시킨 직후, 하지만 `dynticks_progress_counter` 를 증가시키기 전에 NMI를 받았다고 생각해 봅시다. NMI에 의해 호출된 `rcu_irq_enter()`

의 인스턴스는 `rcu_update_flag` 의 원래 값이 0이 아닌 것으로 보게 될 것이고, 따라서 `dynticks_progress_counter` 의 값을 증가시키지 않을 겁니다. 이는 RCU grace-period 치가 이 CPU에서 NMI 핸들러가 실행되었음을 알 수 있는 증거를 남기지 않을 것이고, 따라서 이 NMI 핸들러 내에서의 모든 RCU red-side 크리티컬 섹션들은 RCU 보호를 깨트릴 겁니다.

그 정의에 의해 마스킹 될 수 없는 NMI 핸들러들의 존재 가능성은 정말로 이 코드를 복잡하게 만듭니다. □

Quick Quiz 12.9:

하지만 line 7 이 우리가 가장 바깥의 인터럽트에 있음을 알게 된다면, 우린 항상 `dynticks_progress_counter` 의 값을 증가시켜야 하는 것 아닌가요? ■

Answer:

수행 중인 태스크를 인터럽트 했다면 그렇지 않습니다! 그런 경우에는, `dynticks_progress_counter` 는 이미 `rcu_exit_nohz()`에 의해 값이 증가되었을 것이고, 이 경우에 대해서는 값을 한번 더 증가시킬 필요가 없을 겁니다. □

Quick Quiz 12.10:

이 섹션에서 보인 모든 코드 가운데 버그들을 찾아냈나요? ■

Answer:

당신이 맞았는지 보기 위해 다음 섹션을 읽어보세요. □

Quick Quiz 12.11:

왜 `rcu_exit_nohz()` 와 `rcu_enter_nohz()` 사이의 메모리 배리어는 Promela에 모델링 되지 않은거죠? ■

Answer:

Promela는 sequential consistency를 가정하므로, 메모리 배리어를 모델링할 필요가 없습니다. 사실, 그대신 page 215의 Figure 12.13처럼 명시적으로 메모리 배리어의 부재를 모델링 해야 합니다. □

Quick Quiz 12.12:

`rcu_exit_nohz()`에 이어서 `rcu_enter_nohz()` 가 뒤따르는 경우를 모델링 하는건 좀 이상하지 않나요? 그보다는 진입 후에 빠져나가는 상황을 모델링하는게 더 자연스럽지 않을까요? ■

Answer:

그게 더 자연스러울 수 있습니다만, 우린 우리가 뒤에서 추가할 liveness 체크를 위해 이 특정한 순서가 필요합니다. □

Quick Quiz 12.13:

잠깐만요! 이 리눅스 커널에서, `dynticks_progress_counter` 와 `rcu_dyntick_snapshot` 은 per-CPU 변수들입니다. 그런데 왜 그것들이 per-CPU 변수들이 아니라 하나의 글로벌 변수로 모델링 된거죠? ■

Answer:

해당 grace-period 코드는 각각의 CPU 의 `dynticks_progress_counter` 와 `rcu_dyntick_snapshot` 을 분리해서 처리하기 때문에, 우린 이 상태를 하나의 CPU 로 뭉칠 수 있습니다. 만약 그 grace-period 코드가 특정 CPU 의 특정 값을 가지고 뭔가 타그별한 일을 하려 했다면, 우린 정말로 여러개의 CPU 들을 모델링 해야 할겁니다. 하지만 다행히도, 우린 하나의 CPU 는 grace-period 처리를 수행하고 있고 다른 하나는 `dynticks-idle` 모드를 들어가고 빠져나오는 두개의 CPU 들에만 우리를 안전히 국한시킬 수 있습니다. □

Quick Quiz 12.14:

Line 25 와 26 에 `gp_state` 의 연속된 두개의 변경이 있는데, 어떻게 line 25 에서의 변경이 사라지지 않을거라고 확신할 수 있을까요? ■

Answer:

Promela 와 spin 이 모든 가능한 상태 변경의 시퀀스를 추적한다는 점을 다시 상기하시기 바랍니다. 따라서, 타이밍은 무의미합니다: Promela/spin 은 어떤 상태 변수가 명시적으로 방해를 하지 않는 한은 그 두개의 상태들 사이의 모든 모델을 적당히 섞어낼 겁니다. □

Quick Quiz 12.15:

하지만 `EXECUTE_MAINLINE()` 내의 statement 들이 어토믹 하지 않게 수행되어야 하는 경우라면 어떻게 하겠습니까? ■

Answer:

그렇기 위한 가장 쉬운 방법은 각각의 statement 를 각자의 `EXECUTE_MAINLINE()` 안에 넣는 것입니다. □

Quick Quiz 12.16:

하지만 `dynticks_nohz()` 프로세스가 “if” 나 “do” 문을 가지고 있는데 그 안의 조건절들이 그 분기 본체들은 어토믹하지 않게 수행되는 형태라면 어떻게 되죠? ■

Answer:

뒤의 섹션에서 보게 되겠지만, 한가지 방법은 명시적인 라벨과 “goto” 문을 사용하는 겁니다. 예를 들면, 아래와 같은 것은:

```
if
:: i == 0 -> a = -1;
:: else -> a = -2;
fi;
```

다음과 같은 식으로 모델링 될 수 있습니다:

```
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;
```

하지만, 이 매크로가 “if” 문의 경우에 있어서도 도움이 될지는 명확하지 않은데, 따라서 이런 종류의 상황들에 대해서는 뒤의 섹션들에서 정리하겠습니다. □

Quick Quiz 12.17:

Line 45 와 46 (`in_dyntick_irq = 0;` 와 `i++;`) 은 왜 어토믹하게 수행되는 건가요? ■

Answer:

이 코드 라인들은 모델을 제어하는 부분에 속하지 모델링되는 코드에 속하지 않으므로, 이것들을 어토믹하지 않게 수행시킬 이유가 없습니다. 이것들을 굳이 어토믹하게 모델링 하게 되는 동기는 상태 공간의 크기를 줄이려는 것입니다. □

Quick Quiz 12.18:

이 `dynticks_irq()` 프로세스가 모델링할 수 없는 인터럽트들의 특성은 무엇이 있을까요? ■

Answer:

그런 특성 가운데 하나는 뒤의 섹션에서 다루게 될, 중첩된 인터럽트들입니다. □

Quick Quiz 12.19:

Paul 은 항상 그의 코드를 이렇게 고통스럽도록 점진적인 방법으로 작성하나요? ■

Answer:

항상 그렇지는 않지만, 훨씬 더 자주 그렇습니다. 이 경우, Paul 은 인터럽트 핸들러를 포함한 가장 작은 코드 조각으로 시작했는데, 인터럽트들을 Promel 에서 어떻게 모델링 하는지 최선일지에 대한 확신이 없었기 때문입니다. 일단 동작하게 하고 나서는, 그는 다른 기능들을 추가해 갔습니다. (하지만 그가 이걸 다시 하게 된다면, 그는 “장난감” 핸들러를 가지고 시작할 겁니다. 예를 들어, 한 변수의 값을 두번 증가시키는 핸들러를 가지고 그 값이 항상 짹수임을 검증하는 메인 코드를 만들 겁니다.)

왜 점진적 방법을 사용하는가? Brian W. Kerninghan에 의한 다음의 것들을 고려해 보세요:

디버깅은 코드를 처음 작성하는 것보다 두배 만큼 어렵습니다. 따라서, 당신이 코드를 할 수 있는 한 현명하게 작성한다면, 그 정의에 따라, 당신은 그걸 디버깅하기에 충분히 똑똑하지 않습니다.

이는 코드의 제품을 최적화하려는 노력은 그것의 최소한 66%의 중점은 디버깅 과정을 최적화하는데에 쏟아져야 하는데, 설명 코딩 자체에 쏟는 시간과 노력을 증가시키더라도 그렇습니다. 점진적 코딩과 테스트는 코딩 자체에 드는 노력을 증가시키는 대신에 디버깅 과정을 최적화하는 한가지 방법입니다. Paul은 코딩과 디버깅을 하는데에 (매주는 말할 것도 없고) 모든 날들을 전념할 호사를 부릴 수 있는 기회는 거의 없기 때문에 이 방법을 사용합니다. □

Quick Quiz 12.20:

하지만, 만약 NMI 핸들러가 irq 핸들러가 완료되기 전에 동작하기 시작하면, 그리고 그 NMI 핸들러가 두번째 irq 핸들러가 시작될 때까지 돌아가면 어떤 일이 발생할까요? ■

Answer:

이는 하나의 CPU에 몰아넣은 상황에선 일어날 수 없는 일입니다. 첫번째 irq 핸들러는 NMI 핸들러가 리턴하기 전까지는 완료될 수 없습니다. 따라서, dynticks와 dynticks_nmi 변수들 각각이 주어진 시간 기간동안 짹수를 가졌다면, 연관된 CPU는 그 기간 중의 어느 시점에는 quiescent 상태에 정말로 있었을 겁니다. □

Quick Quiz 12.21:

이건 여전히 상당히 복잡하네요. 왜 그냥 dyntick-idle 모드에 있는 각 CPU들에 대해 bit의 값이 설정되어 있는 cpumask_t를 갖고 irq나 NMI 핸들러에 들어갈 때에는 그 비트의 값을 지우고, 빠져나올 때에는 그 값을 설정하는 식으로 하지 않는거죠? ■

Answer:

이 방법이 가능적으로는 올바르겠지만, 이는 커다란 머신에서는 지나치게 커다란 irq 들어가고 빠져나올 때의 오버헤드를 초래할 겁니다. 반면에, 이 섹션에서 이야기한 방법은 각각의 CPU가 irq와 NMI에 들어가고 나올 때에 per-CPU 데이터만을 만지게 하므로 훨씬 적은 irq 들어가고 빠져나올 때의 오버헤드를 초래할 것이고, 특히 커다란 기계에선 더욱 그러할 겁니다. □

Quick Quiz 12.22:

하지만 x86은 강한 메모리 순서 규칙을 가지고 있어요! 왜 이 메모리 모델을 형식화 시켜야 하는거죠? ■

Answer:

사실, 학계에서는 x86 메모리 모델을 완화된 것으로 여기는데, 앞의 store가 뒤의 load와 재배치될 수 있기 때문입니다. 학계의 시점에서 볼 때, 강한 메모리 모델은 어떠한 재배치도 절대적으로 허용하지 않아서 모든 쓰래드들이 그들에게 보일 수 있는 모든 오퍼레이션들의 순서에 대해 동의할 수 있는 것입니다. □

Quick Quiz 12.23:

Figure 12.25의 line 8은 왜 레지스터들을 초기화 시키죠? 왜 그것들을 line 4와 5에서 대신 초기화 시키지 않나요? ■

Answer:

두 방식 모두 잘 동작합니다. 하지만, 일반적으로는, 명시적 인스트럭션 사용보다는 초기화를 사용하는게 낫습니다. 이 예제에서의 명시적 인스트럭션들은 그 사용법을 보이기 위해 사용되었습니다. 또한, 이 도구의 웹사이트 (<http://www.cl.cam.ac.uk/~pes20/ppcmem/>)에서 사용 가능한 리트머스 테스트들 가운데 많은 것들은 명시적 초기화 인스트럭션들을 사용하도록 자동으로 생성되었습니다. □

Quick Quiz 12.24:

Figure 12.25의 line 17의 Fail: 라벨에서는 무슨 일이 벌어지게 되는거죠? ■

Answer:

atomic_add_return()의 Powerpc 버전 구현은 stwcx 인스트럭션이 condition-code 레지스터에 있는 0이 아닌 상태값을 설정함으로써 통신을 하며 bne 인스트럭션을 통해 테스트하게 되는 성공 여부가 실패로 드러나면 루프를 돌게 됩니다. 실제 루프를 모델링하게 되는 것은 상태 공간의 폭증을 유발하게 되므로, 그대신에 우리는 Fail: 라벨로의 브랜치를 하고, P0의 r3 레지스터의 값이 초기값 2인 형태로 모델링을 종료하여서, exists 단정문의 실패를 유발하지 않게 합니다.

이 트릭이 일반적으로 적용 가능한 것인지에 대해서는 일부 논쟁이 있습니다만, 전 아직 이게 실패하는 예제를 본적이 없습니다. □

Quick Quiz 12.25:

ARM 리눅스 커널도 비슷한 버그를 가지고 있나요? ■

Answer:

ARM에서 리눅스는 atomic_add_return() 함수의 어셈블리어 구현의 앞과 뒤에 smp_mb()를 넣어두기

때문에 이 특정한 버그는 ARM에는 존재하지 않습니다. PowerPC는 더이상 이 버그를 가지고 있지 않습니다; 이 버그는 오래전에 고쳐졌습니다. 리눅스 커널이 가지고 있을수도 있는 또다른 버그들을 찾는 것은 독자 여러분의 몫으로 남겨두겠습니다. □

Quick Quiz 12.26:

L4 마이크로커널의 전체 검증을 생각해 보면, 이런 형식적 검증에 대한 제한적인 시각은 약간 시대에 뒤쳐진 것 아닌가요? ■

Answer:

안타깝지만, 그렇지 않습니다.

L4 마이크로커널의 첫번째 전체 검증은 많은 수의 Ph.D. 학생들 학생마다 매우 느린 속도로 진행한, 손으로 하는 코드 검증으로 이루어진 역작이었습니다. 이런 수준의 노력은 대부분의 소프트웨어 프로젝트들에는 적용될 수가 없는데, 변화의 비율이 너무 거대하기 때문입니다. 더 나아가서, 비록 L4 마이크로커널이 형식적 검증의 시점에서 보기에는 커다란 소프트웨어 작품이긴 합니다만, LLVM, gcc, 리눅스 커널, Hadoop, MongoDB, 그 외의 커다란 많은 것들 등과 같은 많은 수의 프로젝트들에 비교하면 매우 작은 것입니다.

형식적 검증이 마침내 더 최근의, 커다란 수준의 자동화에 관련된 L4 검증을 포함해서 어떤 전망을 내놓기 시작하고 있기는 하지만, 전망 가능한 미래에 테스트를 완전히 대체할 기회는 현재로써는 보이지 않습니다. 그리고 이 점에 있어서 제가 틀렸다고 증명되면 전 좋아 할 겁니다만, 그런 증명은 실제 소프트웨어를 검증하는 실제 도구를 통한 형태여야 하지, 자극적인 미사여구를 통한 형태가 되어선 안될 겁니다. □

D.13 Putting It All Together

Quick Quiz 13.1:

레퍼런스 획득을 단순히 레퍼런스 카운터의 값이 0이 아닌 경우에만 레퍼런스를 획득하는 compare-and-swap 오퍼레이션으로 간단하게 만들지는 않는거죠? ■

Answer:

이 방법이 마지막 레퍼런스의 해제와 새로운 레퍼런스 획득 사이의 경주를 해결할 수는 있겠지만, 데이터 구조체가 해제되고 전혀 다른 종류의 구조체로 재할당되는 것을 방지하는데 있어서는 어떤 일도 해주지 않습니다. “간단한 compare-and-swap 오퍼레이션”이 다른 종류의 구조체에 적용되면 정의되지 않은 결과를 낼 가능성이 상당히 큽니다.

짧게 말해서, compare-and-swap과 같은 어토믹 오퍼레이션의 사용은 타입 안정성이나 존재 보장을 필요로 합니다. □

Quick Quiz 13.2:

한 CPU가 마지막 레퍼런스를 해제한 직후에 다른 CPU가 레퍼런스를 획득하는 경우에 대해서도 보호를 해줘야 하지 않나요? ■

Answer:

CPU는 합법적으로 다른 레퍼런스를 얻기 위해서는 이미 레퍼런스를 가지고 있어야 하기 때문입니다. 따라서, 한 CPU가 마지막 레퍼런스를 해제했다면, 새로운 레퍼런스를 획득할 수 있는 CPU는 존재할 수가 없습니다. 이와 같은 사실이 Figure 13.2의 line 22의 어토믹하지 않은 검사를 가능하게 합니다. □

Quick Quiz 13.3:

Figure 13.2의 line 22에서 atomic_sub_and_test()가 호출된 직후에, 어떤 다른 CPU가 kref_get()을 호출했다고 생각해 봅시다. 이는 이 다른 CPU가 이제 비합법적으로 해제된 오브젝트로의 레퍼런스를 갖게 된 거 아닌가요? ■

Answer:

그런 일은 이 함수가 올바르게 사용된다면 일어날 수 없는 일입니다. 이미 레퍼런스를 가지고 있지 않다면 kref_get()을 호출하는 것은 비합법적인 일어서 kref_sub()는 카운터의 값을 0으로 감소시킬 수 없었을 겁니다. □

Quick Quiz 13.4:

kref_sub()가 0을 리턴해서 release() 함수가 호출되지 않았음을 알렸다고 생각해 봅시다. 어떤 조건에서 호출자는 이 오브젝트의 존재의 지속에 의존할 수 있을까요? ■

Answer:

호출자는 최소 하나의 레퍼런스가 계속해서 존재할 것임을 알지 못한다면 오브젝트의 존재 지속에 의존할 수 없습니다. 일반적으로, 호출자는 이를 알 방법이 없을 것이고, 따라서 kref_sub() 후에 오브젝트를 참조하는 것을 주의깊게 피해야만 합니다. □

Quick Quiz 13.5:

왜 그냥 해제 함수로 kfree()를 넘기지 않는거죠? ■

Answer:

일반적으로 kref 구조체는 더 커다란 구조체에 포함되어 있게 되므로, kref 필드만이 아니라 전체 구조체를 해제시킬 필요가 있습니다. 이는 일반적으로

`container_of()` 와 `kfree()` 를 호출하는 wrapper 함수를 정의하는 것으로 이뤄질 수 있습니다. □

Quick Quiz 13.6:

레퍼런스 카운트의 값이 0인자에 대한 검사는 왜 간단히 어토믹 값 증가 오퍼레이션을 갖는 “if” 문의 “then” 절에 들어갈 수 없는거죠? ■

Answer:

“if” 조건이 완료되었고 레퍼런스 카운터의 값이 1이라는 걸 알게 되었다고 해봅시다. 이어서 레퍼런스 해제 오퍼레이션이 실행되어서 이 레퍼런스 카운터의 값을 0으로 바꾸었고 오브젝트 해제 오퍼레이션을 시작시킵니다. 하지만 이제 “then” 절은 이 카운터의 값을 다시 1로 증가시킬 수가 있어서 오브젝트가 해제된 후에 사용되는 상황을 가능하게 해버리고 맙니다. □

Quick Quiz 13.7:

어토믹하지 않은 `atomic_read()` 와 `atomic_set()` 이라구요? 이건 또 무슨 농담이죠??? ■

Answer:

그렇게 보일수도 있겠습니다만, 현재 문제시 되는 어토믹 변수에 어떤 다른 CPU 도 접근을 하지 않는 상황에서는 실제 어토믹 인스트럭션의 오버헤드는 낭비가 될 수 있습니다. 다른 CPU 가 접근하지 안는 경우의 두 가지 예는 초기화와 정리 작업입니다. □

Quick Quiz 13.8:

대체 왜 그런 글로벌 락이 필요했던 거지요? ■

Answer:

특정 쓰레드의 `__thread` 변수들은 그 쓰레드가 종료될 때 없어집니다. 따라서 다른 쓰레드의 `__thread` 변수들을 접근하는 모든 오퍼레이션은 쓰레드 종료와 동기화 되어야 할 필요가 있습니다. 그런 동기화가 없다면, 방금 종료된 쓰레드의 `__thread` 변수로의 접근은 `segmentation fault` 를 초래할 겁니다. □

Quick Quiz 13.9:

어쨌든, `read_count()` 의 정확성을 대체 뭔가요? ■

Answer:

Page 45 의 Figure 5.9 를 참고하세요. 동시적인 `inc_count()` 의 실행이 존재하지 않는다면, `read_count()` 는 분명한 결과를 내놓을 것은 분명합니다. 하지만, `inc_count()` 의 동시적인 실행이 존재한다면, 그 합계값은 `read_count()` 가 그 합산을 진행함에 따라 실제로 달라질 것입니다. 그렇다면 하나, 쓰레드의 생성과 종료는 `final_mutex` 에 의해 배제되므로, `counterp` 안의 포인터들은 상수로 유지될 것입니다.

메모리의 즉석 스냅샷을 얻어올 수 있는 가상의 기계를 상상해 봅시다. 이 기계가 `read_count()` 의 실행 시작지점의 스냅샷과 `read_count()` 의 실행 종료 시점의 스냅샷을 만들어낸다고 생각해 봅시다. 그렇다면 `read_count()` 는 이 두 스냅샷들 사이의 어떤 시점에서의 각 쓰레드의 카운터에 접근을 할 것이고, 따라서 이 두개의 스냅샷들에 의해 값이 포괄적으로 한정지어지는 결과를 얻어오게 될 것입니다. 따라서, 전체 합계는 이 두개의 스냅샷으로부터 각각 얻어와질 수 있는 합계들의 쌍에 의해 그 값이 한정지어질 것입니다 (다시 말하지만, 포괄적으로).

따라서 예상되는 에러는 이 두개의 스냅샷으로부터 얻어올 수 있는 두개의 합계의 쌍들 사이의 차이의 절반일 것이고, 이는 `read_count()` 의 실행 시간에 단위 시간당 `inc_count()` 의 예상되는 호출 횟수를 곱한 값의 절반입니다.

또는, 수식을 선호하는 분들을 위해 표시하면:

$$\varepsilon = \frac{T_r R_i}{2} \quad (D.12)$$

로, ε 는 `read_count()` 의 리턴 값에 예측되는 에러이고, T_r 은 `read_count()` 가 실행되는데 걸리는 시간이고, R_i 는 단위 시간당 `inc_count()` 호출 횟수의 비율입니다. (그리고 당연하지만, T_r 과 R_i 는 같은 단위 시간을 사용해야 합니다: 마이크로세컨드와 마이크로세컨드당 호출 횟수, 초와 초당 호출 횟수, 뭐가 됐든, 같은 단위를 사용하기만 한다면.) □

Quick Quiz 13.10:

이봐요!!! Figure 13.5 의 line 46 은 앞서 존재한 `countarray` 구조체의 값을 수정하잖아요! 이 구조체는 일단 한번 `read_count()` 에 접근 가능하게 되면 상수로 남게 된다고 하지 않았어요??? ■

Answer:

실제로 전 그렇게 말했습니다. 그리고 `count_unregister_thread()` 가 현재 그렇듯이 `count_register_thread()` 가 새로운 구조체를 할당하도록 하는 것이 가능할 겁니다.

하지만 그건 불필요한 일입니다. 우리가 메모리의 스냅샷에 기반을 둔 `read_count()` 에 의해 최대값이 제한된다는 유도를 다시 생각해 보세요. 새로운 쓰레드들은 그 값이 0인 `counter` 값과 함께 시작되므로, 이 유도는 우리가 `read_count()` 의 실행 중간에 새로운 쓰레드를 추가한다고 해도 지켜집니다. 따라서, 흥미롭게도, 새로운 쓰레드를 추가할 때에, 이 구현은 새로운 구조체를 할당하는 효과를 실제로는 할당을 하지 않으면서도 얻을 수 있게 됩니다. □

```

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 13.11:

우와! Figure 5.9 는 라인수가 42 밖에 되지 않는데 반해 Figure 13.5 는 69 라인이나 되는군요. 이 추가적인 복잡도가 정말로 가치가 있는 건가요? ■

Answer:

이는 당연하게도 경우에 따라 다르게 결정되어야 합니다. 선형적으로 확장되는 `read_count()` 구현이 필요하다면, Figure 5.9 에 보인 락 기반의 구현은 당신을 위한 동작을 하지 않을 겁니다. 반면에, 만약 `count_read()` 의 호출이 충분히 드물다면, 락 기반 버전이 더 간단하고 따라서 더 나을 수도 있습니다. 대부분의 크기 차이는 구조체 정의, 메모리 할당, 그리고 NULL 리턴 체크로 인한 것이니 하지만 말입니다.

물론, 더 나은 질문은 “왜 이 언어는 `__thread` 변수들로의 쓰레드를 넘어서는 접근을 구현하지 않는거죠?”가 될겁니다. 무엇보다, 그런 구현은 락킹과 RCU 의 사용을 불필요하게 만들 겁니다. 이는 결국 Figure 13.5 에 보여진 구현의 확장성과 성능상의 이득에 더해서 Figure 5.9 에 보여진 것보다도 더 간단한 구현이 가능하게 할 겁니다! □

Quick Quiz 13.12:

하지만 Figure 13.9 에 보인 방법은 추가적인 캐시 미스를 초래할 수 있어서, 추가적인 읽기 쪽의 오버헤드를 초래할 수 있지 않나요? ■

Answer:

실제로 그럴 수 있습니다.

이 캐시 미스 오버헤드를 없애기 위한 한가지 방법이 Figure D.10 에 보여져 있습니다: 간단히 `measurement` 구조체의 인스턴스 하나를 `meas`라는 이름으로 `animal` 구조체 내에 내장시키고, `->mp` 필드를 통해 이 `->meas` 필드를 가리키는 것이죠.

측정치 업데이트는 이제 다음과 같이 진행될 수 있습니다:

1. 새로운 `measurement` 구조체를 할당하고 새로운 측정치를 그 안에 집어넣습니다.

2. `rcu_assign_pointer()` 를 사용해서 `->mp` 가 이 새로운 구조체를 가리키도록 합니다.

3. 하나의 grace period 가 지나가길 기다리는데, 예를 들어 `synchronize_rcu()` 나 `call_rcu()` 를 사용합니다.

4. 새로운 `measurement` 구조체의 측정치들을 내장된 `->meas` 필드로 복사합니다.

5. `rcu_assign_pointer()` 를 사용해서 `->mp` 가 다시 이 기존의 내장된 `->meas` 필드를 가리키도록 합니다.

6. 또 다른 하나의 grace period 가 지나간 후, 새로운 `measurement` 구조체를 메모리에서 해제시킵니다.

이 방법은 일반적인 경우에서의 추가적인 캐시 미스를 제거하기 위해 더 무거운 업데이트 절차를 사용합니다. 이 추가적인 캐시 미스는 업데이트가 실제로 진행 중일 때에만 일어날 겁니다. □

Quick Quiz 13.13:

하지만 이 스캐닝은 크기 재조정이 되고 있는 크기 재조정 가능한 해시 테이블에 대해서는 어떻게 동작하나요? 그런 경우, 기존의 해시테이블도 새로운 해시테이블도 해당 해시 테이블에 모든 원소들을 담고 있다고 보장될 수 없는데요! ■

Answer:

사실입니다, Section 10.4 에서 이야기된 크기 재조정 가능한 해시 테이블들은 크기 재조정 진행 중에 모두 스캔 될 수 없습니다. 이를 위한 한가지 간단한 방법은 `hashtab` 구조체의 `->ht_lock` 을 스캐닝 중에 잡는 것입니다만, 이 방법은 두개 이상의 스캔이 동시에 진행되지 못하도록 합니다.

또 다른 방법은 크기 재조정 진행 중에는 업데이트들이 새로운 해시 테이블만이 아니라 기존의 것도 변형시키도록 하는 것입니다. 이는 스캐닝이 기존의 해시 테이블에서도 모든 원소들을 볼 수 있도록 합니다. 이를 구현하는 것은 독자 여러분의 몫으로 남겨두겠습니다. □

D.14 Advanced Synchronization

Quick Quiz 14.1:

page 255 의 Figure 14.3 코드의 line 21 의 단정문이 대체 어떻게 실패할 수 있죠? ■

Answer:

핵심 포인트는, 직관적 분석이 놓친 것은 C 에의 값 할당 결과가 A 에의 값 할당보다 먼저 `thread2()`에게 전파되는 것을 막는 것이 없다는 점입니다. 이건 이 섹션의 뒤에서 설명됩니다. □

Quick Quiz 14.2:

좋아요... 그래서 이걸 어떻게 고쳐야 하죠? ■

Answer:

가장 쉬운 방법은 line 12 와 line 20 의 `barrier()` 를 `smp_mb()` 로 바꾸는 것입니다.

물론, 일부 하드웨어는 다른 하드웨어보다 너그럽습니다. 예를 들어, x86 에서 page 255 Figure 14.3 의 line 21 의 단정문은 실패하지 않습니다. PowerPC 에서는 line 20 의 `barrier()` 만 `smp_mb()` 로 바꿔도 단정문이 실패하지 않게 할 수 있습니다. □

Quick Quiz 14.3:

Figure 14.4 의 코드 조각이 가정하고 있는, 실제 하드웨어에서는 불가능한 일은 무엇인가요? ■

Answer:

해당 코드는 주어진 CPU 가 자신의 값을 보는 것을 멈추는 순간, 최종의 모든 CPU 가 동의한, 최종값을 볼 것이라 생각합니다. 실제 하드웨어에서는, 일부 CPU 들은 마지막 값에 이르기 이전의 중간 상태 값도 볼 수 있습니다. □

Quick Quiz 14.4:

어떻게 CPU 들이 하나의 변수에 대해 같은 시간에 그 값을 다르게 볼 수 있을까요? ■

Answer:

많은 CPU 들이 최근의 쓰기 값을 기록하며 연관된 캐시 라인이 CPU 로 불려갈 때 적용되는 write buffer 들을 갖습니다. 따라서, 각 CPU 가 한 변수에 대해 동시에 서로 다른 값을 보는 것이 가능합니다 — 그리고 메인 메모리는 또 다른 값을 가지고 있을 수 있습니다. 메모리 배리어가 만들어진 이유들 중 하나는 소프트웨어가 이런 상황을 잘 처리할 수 있도록 하기 위해서입니다. □

Quick Quiz 14.5:

CPU 2 와 3 은 그렇게 빨리 합의에 이르렀는데 CPU 1 과 4 는 그렇게 오래 걸린 이유가 뭐죠? ■

Answer:

CPU 2 와 3 은 같은 코어에서 수행되는 하드웨어 쓰레드여서 같은 캐시 구조를 공유하며, 따라서 매우 적은 통신 대기시간을 갖습니다. 이건 NUMA, 또는, 더 정확히는 NUCA 효과입니다.

이 사실은 CPU 2 와 3 이 동의하지 않는 경우가 왜 발생할 수 있는 건지 궁금하게 만듭니다. 한가지 가능한 이유는 CPU 2 와 3 이 공유된 커다란 캐시 외에도 작은 개별 캐시를 가지고 있을 수도 있다는 점입니다. 또 다른 가능한 이유는 인스트럭션 재배치로, 동의에 걸리는 시간이 매우 짧은 10 나노세컨드라는 점과 해당 코드에 메모리 배리어가 없다는 점이 이 가설을 뒷받침합니다. □

Quick Quiz 14.6:

하지만 메모리 배리어들이 무조건적으로 순서를 강제해 주지 않는다면, 디바이스 드라이버는 대체 어떻게 안정적으로 MMIO 레지스터로의 일련의 로드와 스트어들을 실행할 수 있나요? ■

Answer:

MMIO 레지스터들은 특별한 경우입니다: 그것들은 물리 메모리의 캐시되지 않는 영역에 위치합니다. 메모리 배리어들은 캐시되지 않는 메모리로의 로드와 스트어는 무조건적으로 순서를 강제하는데, 이에 대해선 Section 14.2.8 에서 다루겠습니다. □

Quick Quiz 14.7:

ears-to-mouths 시나리오에서 최신 하드웨어가 로드들 중 최소 하나는 다른 쓰레드에 의해 저장된 값을 읽을 거라 보장함을 우리는 어떻게 알죠? ■

Answer:

그 시나리오는 A 와 B 의 초기값이 0이라는 가정 하에 다음과 같습니다:

CPU 0: A=1; `smp_mb()`; r1=B;

CPU 1: B=1; `smp_mb()`; r2=A;

두 로드 모두 관련된 스트어를 보지 못한다면, 두 CPU 들이 모두 실행을 마쳤을 때, r1 과 r2 둘 다 0일 겁니다. 이제 r1 이 0이라고 생각해 봅시다. 그렇다면 CPU 0 의 B 로부터의 로드가 CPU 1 의 B 로의 스트어 이전에 일어났음을 알 수 있습니다: 그렇지 않다면 r1 이 1일 겁니다. 하지만 CPU 0 의 B 로부터의 로드가 CPU 1 의 B 로의 스트어 이전에 일어났으므로, 메모리 배리어 조합이 CPU 0 의 A 로의 스트어가 CPU 1 의 A 로부터의 로드 이전에 일어났음을 보장하고, 따라서 r2 는 0이 아니라 1일 것임을 보장합니다.

따라서, r1 과 r2 중 하나는 0이 아닐 것이고, 이는 이야기 했듯 최소 하나의 로드는 관련된 스트어를 봤음을 의미합니다. □

Quick Quiz 14.8:

Table 14.2 의 다른 “Only one store” 항목은 어떻게 사용될 수 있나요? ■

Answer:

조합 2 에서는, 만약 CPU 1 의 B 로부터의 로드가 CPU 2 의 B 로의 스토어보다 전의 값을 본다면, 우린 CPU 2 의 A 로부터의 로드가 CPU 1 의 A 로부터의 로드와 같거나 나중의 값을 리턴할 것을 알 수 있습니다.

조합 4 에서는, 만약 CPU 2 의 B 로부터의 로드가 CPU 1 의 B 로의 스토어의 값을 본다면, CPU 2 의 A 로부터의 로드가 CPU 1 의 A 로부터의 로드와 같거나 나중의 값을 리턴할 것을 알 수 있습니다.

조합 8 에서는, CPU 2 의 A 로부터의 로드가 CPU 1 의 A 로 저장한 값을 본다면, CPU 1 의 B 로부터의 로드가 CPU 2 의 A 로부터의 로드와 같거나 나중의 값을 보게 됨을 알 수 있습니다. □

Quick Quiz 14.9:

어떻게 page 261 의 $b==2$ 단정문이 실패할 수 있죠? ■

Answer:

해당 CPU 가 자신의 모든 로드와 스토어들을 순서대로 보도록 요구되지 않는다면, $b=1+a$ 는 변수 “a” 의 예전 버전을 볼 수도 있습니다.

이게 바로 각 CPU 나 쓰레드가 자신의 모든 로드와 스토어들을 프로그램 순서대로 보도록 해야 하는 이유입니다. □

Quick Quiz 14.10:

어떻게 page 261 의 코드가 메모리 럭을 일으킬 수 있죠? ■

Answer:

해당 크리티컬 섹션의 첫번째 수행만이 $p==NULL$ 을 볼 수 있습니다. 하지만,

$tt mylock$ 을 통한 크리티컬 섹션들 간의 글로벌한 순서가 없다면, 어떻게 어떤 특정한 한 하나의 크리티컬 섹션 수행이 첫번째라고 이야기할 수 있겠습니까? 만약 여러개의 다른 크리티컬 섹션 수행이 자신이 첫번째라고 생각한다면, 그들은 모두 $p==NULL$ 을 보게 될 것이고, 따라서 모두 메모리를 할당받을 것입니다. 하나를 제외한 나머지 수행들은 메모리를 누수시켜 버립니다.

이게 하나의 배타적 럭을 사용하는 모든 크리티컬 섹션들이 잘 정의된 순서대로 실행되는 것으로 나타나야 하는 이유입니다. □

Quick Quiz 14.11:

어떻게 page 261 의 코드가 거꾸로 수를 셀 수 있죠? ■

Answer:

해당 카운터가 값 0 으로 시작을 하고, 세개의 크리티컬

섹션이 수행되어 그 값을 3 으로 바꿔 놓았다고 생각해 봅시다. 만약 네번째 해당 크리티컬 섹션 수행이 가장 최근의 이 변수에의 저장된 값을 보도록 강제되지 않는다면, 이 수행 흐름 역시 원본값인 0 을 보게 되고, 따라서 카운터를 1 로 만들어 버리는데, 이건 거꾸로 수를 세는 셈입니다.

이게 특정 크리티컬 섹션에서의 특정 변수로부터의 로드는 앞의 마지막 크리티컬 섹션의 해당 변수에의 스토어 결과를 봐야 하는 이유입니다. □

Quick Quiz 14.12:

다음의 코드는 변수 “a” 와 “b” 로의 스토어들의 순서에 어떤 영향을 끼칠까요? ■

```
a = 1;
b = 1;
<write barrier>
```

Answer:

아무것도요. 이 배리어는 “a” 와 “b” 로의 값 할당이 뒤의 어떤 값 할당보다도 먼저 일어남을 보장할 겁니다만, “a” 와 “b” 로의 값 할당 자체 사이의 순서에 대해서는 아무것도 강제하지 않습니다. □

Quick Quiz 14.13:

LOCK-UNLOCK 오퍼레이션들을 어떻게 조합해야 전체 메모리 배리어처럼 동작할까요? ■

Answer:

등을 맞대는 두개의 LOCK-UNLOCK 오퍼레이션들, 또는, 덜 관습적이지만, UNLOCK 뒤에 LOCK 을 배치하는 경우가 되겠습니다. □

Quick Quiz 14.14:

(만약 있다면) 어떤 CPU 들이 이런 투과성의 럭킹 기능들을 만들 수 있는 메모리 배리어 인스트럭션들을 가지고 있을까요? ■

Answer:

Itanium 이 하나의 예가 될 것입니다. 다른 것들을 찾아보는 것은 독자의 몫으로 두겠습니다. □

Quick Quiz 14.15:

Table 14.3 에서 중괄호로 그룹지어진 오퍼레이션들은 동시에 수행된다고 보면, 표의 어떤 열들이 “A” 에서 “F” 까지의 변수들과 LOCK/UNLOCK 오퍼레이션들의 합법적인 재배치일까요? (코드의 순서는 A, B, LOCK, C, D, UNLOCK, E, F 입니다.) 합법이면 왜 합법이고 아니라면 왜 아니죠? ■

Answer:

- 합법입니다, 순서 그대로 실행되었습니다.
- 합법입니다, 락 획득은 크리티컬 앞의 마지막 값 할당과 동시적으로 수행되었습니다.
- 비합법입니다, “F”에의 값 할당은 LOCK 오퍼레이션 뒤에 이뤄져야만 합니다.
- 비합법입니다, LOCK은 크리티컬 섹션의 어떤 오퍼레이션 보다도 먼저 완료되어야만 합니다. 하지만, UNLOCK은 앞의 오펠에시녀들과 동시에 수행되어도 합법일 수 있습니다.
- 합법입니다, “A”에의 값 할당은 요청된 대로 UNLOCK을 앞섰고, 모든 다른 오퍼레이션들은 순서 그대로입니다.
- 비합법입니다, “C”에의 값 할당은 LOCK 뒤에 이뤄져야만 합니다.
- 비합법입니다, “D”에의 값 할당은 UNLOCK 앞에 이뤄져야만 합니다.
- 합법입니다, 모든 할당이 LOCK과 UNLOCK 기준으로 순서 맞춰져 있습니다.
- 비합법입니다, “A”에의 값 할당은 UNLOCK 전에 이뤄져야만 합니다.

□

Quick Quiz 14.16:

Table 14.4에서의 제약은 뭐죠? ■

Answer:

모든 CPU들은 다음의 순서 제약을 봐야만 합니다:

- LOCK M은 B, C, D 보다 먼저 행해짐.
- UNLOCK M은 A, B, C 보다 뒤에 행해짐.
- LOCK Q는 F, G, H 보다 먼저 행해짐.
- UNLOCK Q는 E, F, G 보다 뒤에 행해짐.

□

D.15 Parallel Real-Time Computing**Quick Quiz 15.1:**

배터리로 동작하는 시스템의 경우에는 어떻죠? 그것들은 그 시스템으로 흘러들어오는 에너지를 전혀 필요로 하지 않아요. ■

Answer:

금방이든 나중이든, 배터리는 재충전되어야 하는데, 그러지 않으면 그 시스템은 동작을 멈추게 될 것이므로 이는 시스템으로 흘러들어오는 에너지를 필요로하게 됩니다. □

Quick Quiz 15.2:

하지만 queueing theory에서의 결과를 놓고 생각해 보면, 낮은 리소스 활용률은 평균 응답 시간만을 개선할 뿐이지 최악의 경우의 응답시간은 개선하지 못하지 않을까요? 그리고 최악의 경우에서의 응답시간이야말로 대부분의 real-time 시스템들이 정말로 신경쓰는 것이 아니던가요? ■

Answer:

맞습니다, 하지만 ...

그런 queueing-theory의 결론들은 리눅스 커널이라면 무한한 수의 task 들로 연관될 수 있는, 무한한 “호출하는 인구”를 가정하고 있습니다. 2016년 중반의 시점에서, 어떤 실제 시스템도 무한한 수의 task 들을 지원하지 않으며, 따라서 무한한 호출하는 인구를 가정한 결과는 무한보다는 적은 응용성을 갖게 될 것이라 기대 될 겁니다.

또다른 queueing-theory의 결론은 유한한 호출 인구 수를 갖는데, 이 경우는 깔끔하게 그 한계가 제한되어 있는 응답 시간을 갖습니다 [HL86]. 이런 결론들은 실제 시스템을 더 잘 모델링 하고, 이런 모델링들은 리소스 활용율의 감소에 따른 평균 응답시간과 최악의 경우 응답시간의 감소를 모두 예측합니다. 이런 결론들은 락킹과 같은 동기화 메커니즘을 사용하는 동시성 시스템을 모델링하는데에도 확장될 수 있습니다 [Bra11].

짧게 요약해서, 정확하게 실제 세계의 real-time 시스템을 묘사하는 queueing-theory의 결론들은 줄어드는 리소스 활용율에 따라 최악의 경우의 응답시간도 줄어듬을 보입니다. □

Quick Quiz 15.3:

형식적 검증은 수십년간 집중된 연구로 인해 이미 상당히 실용 가능한 수준입니다. 추가적인 진보가 정말로 필요한 건가요, 아니면 이건 그저 실무자의 게으르기를 계속하고 형식적 검증의 대단한 위력을 무시하려는 변명일 뿐인가요? ■

Answer:

아마도 이 상황은 이론가들의 실제 소프트웨어의 복잡한 세계로 뛰어들지 않기 위한 변명일 수도 있겠는데요? 더 자세히 말해보자면, 다음과 같은 진보가 필요합니다:

1. 형식적 검증은 더 커다란 소프트웨어 작품들을 처리할 수 있어야만 합니다. 행해진 가장 큰 검증은 약 10,000 라인의 코드를 갖는 시스템에 대한 검증 뿐이었고, 그것들은 real-time 응답시간들보다는 훨씬 간단한 속성들에 대한 검증이었습니다.
2. 하드웨어 제조사들은 정형적인 시간 보장사항에 대한 내용을 제공해야 하게 될겁니다. 이는 하드웨어가 훨씬 더 간단했던 시절에는 일반적인 일이었습니다만, 오늘날의 복잡한 하드웨어는 최악의 경우의 성능에 대해서는 과하게 복잡한 표현을 초래하게 되었습니다. 안타깝게도, 에너지 효율성에 대한 관심은 제조사들을 이보다도 더 복잡한 세계로 떠밀고 있습니다.
3. 타이밍 분석은 개발 방법론과 IDE 들에 합쳐질 필요가 있습니다.

그렇게 말했긴 하지만, 최근의 실제 컴퓨터 시스템의 메모리 모델에 대한 정형화 작업 [AMP⁺11, AKNT13]을 놓고 보면 희망이 있습니다. □

Quick Quiz 15.4:

Real-time 과 real-time 이 아닌 것을 무엇이 “real-time 이 아닌 시스템과 어플리케이션에 의해서 간단히 이뤄질 수 있는지”로 구분하는 것은 우스운 일입니다! 그런 구별을 위한 이론적 기초는 전혀 존재치 않아요!!! 더 나은 것을 할 수는 없을까요??? ■

Answer:

이 구분법은 엄격한 이론적 관점에서는 틀림없이 만족스럽지 못합니다. 하지만 한편으로는, 이것은 개발자가 어떤 어플리케이션이 쉽게 짠 비용으로 real-time 이 아닌 방법을 사용해서 개발될 수 있는지, 또는 더 어렵고 비싼 real-time 접근법이 필요한지를 결정하기 위해 필요한 것입니다. 달리 말하자면, 이론은 상당히 중요합니다만, 우리처럼 일이 제대로 행해지기를 바라는 사람들에게 있어서는, 이론은 실전을 지원할 뿐이지, 그외의 것은 아닙니다. □

Quick Quiz 15.5:

하지만 reader-writer 락에 대해서 한번에 하나의 read 권한 획득만을 허용한다면, 그건 배타적 락과 똑같은 거 아닙니까??? ■

Answer:

실제로 그렇습니다, API 만 제외하고는 말이지요. 그리고 그 API가 중요한데, 그게 리눅스 커널이 -rt 패치셋이 웃기도록 커다란 크기로 커지지 않고도 real-time 기능들을 제공하도록 해주기 때문에 중요합니다.

하지만, 이 방법은 read-side 의 확장성을 분명하고 상당히 제한합니다. 리눅스 커널의 -rt 패치셋은 몇 가지 이유들로 인해 이 한계점들을 가지기로 했습니다: (1) Real-time 시스템은 전통적으로 상대적으로 작은 크기였고, (2) Real-time 시스템은 일반적으로 프로세스 제어에 중점을 두었으므로, I/O 서브시스템에서의 확장성 제한에 영향을 받지 않을 것이며, (3) 많은 리눅스 커널의 reader-writer 락은 RCU 로 변경되었습니다.

그렇다고는 하나, 리눅스 커널이 언젠가는 우선순위 증폭을 위해 reader-writer 락의 read-side 병렬성에 제한을 주는 상황은 가능합니다. □

Quick Quiz 15.6:

Figure 15.15 의 line 17 에서의 `t->rcu_read_unlock.special.s` 의 로드 직후에 `preemption` 이 발생했다고 생각해 봅시다. 그렇게 되면 해당 task 가 `rcu_read_unlock_special()` 를 수행시키지 못하게 되어서, 자기 자신을 현재 grace period 를 막고 있는 task 들의 리스트로부터 삭제하는 것을 막아서, 그 grace period 가 무한정 길어질 수 있도록 할 수 있지 않을까요? ■

Answer:

그건 실제로 문제이고, RCU 의 스케줄러 쪽 후킹을 통해 해결되어졌습니다. 만약 그 후킹된 scheduler 코드가 `t->rcu_read_lock_nesting` 의 값이 음수임을 보게 된다면, 그 코드는 컨텍스트 스위치가 완료되도록 하기 전에 필요하다면 `rcu_read_unlock_special()` 을 호출합니다. □

Quick Quiz 15.7:

하지만 fail-stop 베그에도 불구하고 올바르게 동작하는 게 가치있는 fault-tolerance 속성 가운데 하나 아닌가요?

■

Answer:

그렇고 아닙니다.

Non-blocking 알고리즘들이 fail-stop 베그들의 존재 하에서도 fault tolerance 를 제공할 수 있다는 점에서 그려합니다만, 이는 실질적인 fault tolerance 에는 심하게 충분치 못하다는 점에서 그렇지 않습니다. 예를 들어, 여러분이 wait-free 대기열을 가지고 있고, 방금 하나의 원소를 거기서 꺼낸 쓰레드가 하나 있다고 생각해 보세요. 만약 그 쓰레드가 fail-stop 베그로 죽는다면, 이 쓰레드가 막 꺼낸 원소는 실질적으로는 잃어버리게 됩니다. 진정한 fault tolerance 는 단순한 non-blocking

속성보다 더한 것을 필요로 하고, 그것은 이 책의 범위 밖의 것입니다. □

Quick Quiz 15.8:

전 잘 모르겠지만 이 리스트 앞에 “포함한다”는 단어를 봤어요. 다른 제약도 존재하는 건가요? ■

Answer:

실제로 존재하구요, 많은 것들이 존재합니다. 하지만, 그들은 그들은 특정 상황에 종속적인 경향을 갖고, 그중 많은 것들은 앞서 언급된 제약들의 세련된 형태로 생각될 수 있습니다. 예를 들어, 데이터 구조의 선택에 있어서의 많은 제약들은 “모든 크리티컬 섹션은 제한된 길이의 시간만을 소모해야함” 제약을 지키는데 도움이 될겁니다. □

Quick Quiz 15.9:

Real-time 시스템들은 많은 경우 safety-critical 어플리케이션들에 사용된다는 점을 놓고 보면, 그리고 수행시간 메모리 할당은 많은 safety-critical 상황에서 숨겨진다는 점을 생각해보면, `malloc()` 호출은 뭐때매 있는거죠??? ■

Answer:

2016년 초에 있어, 수행 시간 메모리를 숨기는 상황이 멀티쓰레드 컴퓨팅에서 그렇게까지 신나지는 않습니다. 따라서 수행시간 메모리 할당은 safety criticality에 추가적인 장애는 아닙니다. □

Quick Quiz 15.10:

`update_cal()`을 보호하기 위해 어떤 동기화가 필요하지 않나요? ■

Answer:

실제로 그렇고, 여러분은 이 책의 앞에서 이야기된 기술들을 얼마든지 사용할 수 있습니다. □

D.16 Ease of Use

Quick Quiz 16.1:

원소를 지울 때에도 비슷한 알고리즘을 사용할 수 있습니까? ■

Answer:

네. 하지만, 각각의 쓰레드는 가운데 하나를 삭제하기 위해 연속된 세개의 원소의 락을 잡아야 하므로, N 개의 쓰레드가 존재한다면, 데드락이 없도록 하기 위해서는 리스트에 ($N + 1$ 개가 아니라) $2N + 1$ 개의 원소가 존재해야만 합니다. □

Quick Quiz 16.2:

그려운요! 이처럼 깎여져나갈 가치가 있는 알고리즘을 가지고 나타난 누군가를 포용해 본 적 있나요??? ■

Answer:

그건 Paul 일 수 있을 겁니다.

그는 다섯명의 철학자들이 참가하는 비위생적인 스파게티 저녁식사에 관련된, *Dining Philosopher's Problem*을 생각하고 있었습니다. 다섯개의 접시가 있지만 다섯개의 포크만이 테이블에 있다는 점을 가지고, 그리고 각각의 철학자는 식사를 위해 한번에 두개의 포크가 필요하다는 점을 가지고, 누군가는 데드락을 예방하는 포크 할당 알고리즘을 생각해 낼 수 있을 겁니다. Paul의 응답은 “조용! 그냥 다섯개 더 포크를 가져와!” 였습니다.

이건 그 자체로는 괜찮습니다만, Paul은 이내 이와 똑같은 해결책을 순환형 링크드 리스트에 적용했습니다.

이것 역시 그렇게 나쁘진 않았을 겁니다만, 그는 누군가에게 그에 대해 이야기를 해야만 했습니다! □

Quick Quiz 16.3:

이 규칙에 대한 예외를 대 보세요. ■

Answer:

한가지 예외는 주어진 상황 하에서는 유일하게 동작하는 어렵고 복잡한 알고리즘이 될 수 있을 겁니다. 또 다른 예외는 주어진 상황에서 동작하는 것으로 알려진 것 중에서는 가장 간단한, 어렵고 복잡한 알고리즘이 될 수 있을 겁니다. 하지만, 이런 경우들에 있어서 조차도, 더 간단한 알고리즘을 내놓기 위해 조금 시간을 써보는게 좋을 겁니다! 무엇보다도, 여러분이 어떤 작업을 하는 첫번째 알고리즘을 만들어내게 된다면, 더 간단한 것을 하나 만드는 것은 그렇게까지 어렵지는 않을 겁니다. □

D.17 Conflicting Visions of the Future

Quick Quiz 17.1:

메모리의 `mmap()` 리전 내의 데이터 구조체로 표현되는 지속성 없는 기능들은 어떨까요? 그런 기능들로 만들어진 크리티컬 섹션 내에서의 `exec()`가 존재한다면 어떤 일이 벌어질까요? ■

Answer:

`exec()` 된 프로그램이 그 똑같은 메모리 영역을 매핑한다면, 이 프로그램은 원칙적으로 그 락을 놓아주어야 합니다. 이 방법이 소프트웨어 엔지니어링 관점에서도

말이 되는 소리인지에 대한 판단은 독자 여러분의 몫으로 남겨두도록 하겠습니다. □

Quick Quiz 17.2:

해당 락 변수와 캐시 라인을 공유하는, 자주 쓰여지는 변수가 왜 문제가 될까요? ■

Answer:

락이 그것이 보호하는 변수와 같은 캐시라인에 있다면, 하나의 CPU에 의한 그 변수들로의 쓰기는 모든 다른 CPU들에 있는 그 캐시 라인을 무효화 시킵니다. 이런 무효화는 많은 충돌과 재시도를 만들어내고, 심지어 락킹에 의해 성능과 확장성을 떨어뜨릴 수도 있을 겁니다. □

Quick Quiz 17.3:

HTM 성능과 확장성에 상대적으로 적은 업데이트가 중요한 이유가 뭐죠? ■

Answer:

업데이트가 많을수록, 충돌의 가능성이 커지고, 따라서 재시도의 가능성이 커져서 성능이 하락됩니다. □

Quick Quiz 17.4:

동기화 메커니즘의 선택에 관계 없이 어떻게 red-black 트리가 트리 내의 모든 원소의 열거를 효율적으로 할 수 있을까요??? ■

Answer:

많은 경우에, 이 열거는 정확하지 않아도 좋습니다. 이런 경우들에 있어서, hazard pointer나 RCU가 특정한 삽입이나 삭제와의 낮은 conflict 확률을 유지하면서 임기 쓰레드들을 보호하는데 사용될 수 있습니다. □

Quick Quiz 17.5:

하지만 왜 디버거는 트랜잭션의 앞의 인스턴스의 스텝들을 다시 추적하기 위해 재시도에 의존하면서 브레이크포인트를 트랜잭션의 성공되는 명령문 줄에 설정해 두는 것으로 single stepping을 흉내낼 수 없나요? ■

Answer:

이 방법은 높은 확률로 동작할 수 있을 겁니다만, 대부분의 사용자들에게는 상당히 놀라운 형태로 실패할 수 있습니다. 이를 보기 위해, 다음 트랜잭션을 생각해 봅시다:

```

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();

```

사용자가 line 3에 트랜잭션을 abort시키고 디버거에 들어가게 될 브레이크포인트를 설정했다고 생각해 봅시다. 브레이크포인트가 시작되고 디버거가 모든 쓰레드를 정지시키는 사이에, 어떤 다른 쓰레드가 a의 값을 0으로 설정했다고 생각해 봅시다. 사용자가 이 프로그램을 single-step하면, 짜잔! 프로그램은 이제 then-절 대신 else-절에 들어와 있습니다.

이는 제가 사용성이 좋은 디버거라 부르는 것이 아닙니다. □

Quick Quiz 17.6:

하지만 누가 텅 빈 락 기반의 크리티컬 섹션을 필요로 하나요??? ■

Answer:

Section 7.2.1의 Quick Quiz 7.18의 답을 보시기 바랍니다.

하지만, 진행 보장이 없는 강력한 어토믹 HTM 구현들에 대해서, 텅 빈 크리티컬 섹션에 기반한 메모리 기반의 락킹 설계는 transactional lock elision의 존재에도 올바르게 동작할 거라는 주장이 있습니다. 비록 저는 이 주장에 대한 증명을 보지는 못했습니다만, 이 주장에는 직선적인 합리성이 존재합니다. 주요 아이디어는, 강력한 어토믹 HTM 구현에 있어서, 특정 트랜잭션의 결과는 그 트랜잭션이 성공적으로 완료되기 전까지는 보여지지 않는다는 것입니다. 따라서, 트랜잭션이 시작된 것을 볼 수 있다면, 그것이 이미 완료되었음이 보장되는데, 이 말은 뒤따르는 텅 빈 락 기반의 크리티컬 섹션은 성공적으로 그것을 “기다릴” 것을 의미합니다—무엇보다도, 기다림이 요구되지 않습니다.

이 이야기는 (많은 STM 구현을 포함하는) weakly atomic 시스템에는 적용되지 않고, 통신에 메모리 이외의 수단을 사용하는 락 기반의 프로그램들에는 적용되지 않습니다. 그런 수단으로는 시간의 흐름(예를 들어, hard real-time 시스템)이나 우선순위의 흐름(예를 들어, soft real-time 시스템)이 있습니다.

Priority boosting에 의존하는 락킹 설계는 특히 흥미로운 경우입니다. □

Quick Quiz 17.7:

락 기반의 텅 빈 크리티컬 섹션들을 생략하지 않는 방법으로 간단하게 락킹의 시간 기반 메세징 semantic 을 transactional lock elision에서 처리할 수는 없을까요? ■

Answer:

그럴수도 있습니다만, 이는 불필요하고 불충분할 겁니다.

텅 빈 크리티컬 섹션이 조건적 컴파일에 의한 것이라면 이 방법은 필요가 없습니다. 여기서는, 해당 락의 유일한 목적은 데이터를 보호하는 것임으로, 이를 생략시키는 것이 해야할 옳은 일일 것입니다. 실제로, 락 기반의 텅 빈 크리티컬 섹션을 남겨두는 것은 성능과 확장성을 떨어뜨릴 수 있습니다.

또다른 한편, 락 기반의 비어있지 않은 크리티컬 섹션이 락킹의 데이터 보호화 시간 기반의 메세징 semantic에 의존하고 있을 수도 있습니다. 그런 경우에 transactional lock elision을 사용하는 것은 올바르지 않고, 버그를 초래할 수 있습니다. □

Quick Quiz 17.8:

최신 하드웨어 [MOZ09]에서, 누가 병렬 소프트웨어가 타이밍에 의존해서 동작할 거라고 기대할 수 있겠습니까? ■

Answer:

짧게 답하자면 일반적으로 구할수 있는 하드웨어에서, 짧은 시간 단위의 타이밍에 기반한 동기화 설계들은 무모하고 모든 조건 하에서 올바르게 동작할 거라고 예상될 수 없습니다.

그렇다고는 하나, 훨씬 더 결정론적인, hard real-time에서의 사용을 위해 설계된 시스템들이 있습니다. 여러분이 그런 시스템을 사용하게 되는 (있을 수 없을법한) 상황이라면 여기 시간 기반의 동기화가 동작할 수 있는지 보이는 예가 있습니다. 다시 말하지만, 일반적인 마이크로프로세서는 매우 비결정론적인 성능 특성을 가지고 있으므로, 이를 일반적인 마이크로프로세서 위에서는 시도하지 마세요.

이 예는 하나의 제어 쓰레드와 함께 복수개의 worker 쓰레드를 사용합니다. 각각의 worker 쓰레드는 외부로 나가는 데이터에 연관되고, 각 단위의 일을 수행한 후에 (예를 들어, `clock_gettime()` 시스템콜로 얻어오는) 현재 시간을 per-thread `my_timestamp` 변수에 저장합니다. 이 예의 real-time 특성은 다음과 같은 제한을 갖습니다:

- 특정 worker 쓰레드가 자신의 타임스탬프를 `MAX_LOOP_TIME` 기간이 넘도록 업데이트 하지 못하는 것은 치명적인 에러입니다.

2. 락들은 글로벌 상태에 접근하고 업데이트하는데 절약적으로 사용되어야 합니다.

3. 락들은 각각의 쓰레드 우선순위 내에서는 엄격한 FIFO 순서로 얻어집니다.

Worker 쓰레드들이 일을 받는 걸 완료하면, 이들은 어플리케이션의 다른 부분들로부터 그들을 풀어내고 –1로 초기화 한 per-thread `my_status` 변수의 상태값을 설정해야 합니다. 쓰레드들은 종료되지 않습니다; 그대신 뒤이어 처리해야 할 것들을 처리해주기 위해 쓰레드 풀에 들어갑니다. 제어 쓰레드는 필요한 만큼 worker 쓰레드를 할당 (그리고 재할당) 하고, 또한 쓰레드 상태들의 히스토그램을 관리합니다. 제어 쓰레드는 worker 쓰레드보다 높지는 않은 real-time 우선순위로 동작합니다.

Worker 쓰레드의 코드는 다음과 같습니다:

```

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 /*
13  * Disentangle from application, might
14  * acquire other locks, can take much longer
15  * than MAX_LOOP_TIME, especially if many
16  * threads exit concurrently.
17  */
18 my_status = get_return_status();
19 release_lock(&departing_thread_lock);
20
21 /* thread awaits repurposing. */

```

제어 쓰레드의 코드는 다음과 같습니다:

```

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         }
12     }
13  /* Repurpose threads as needed. */
14 }

```

Line 5는 해당 쓰레드가 종료되었는지를 추론하는데에 시간의 흐름을 사용하고, 만약 그렇다면 line 6-10을 수행합니다. Line 7과 8의 락 기반의 텅 빈 크리티컬 섹션은 종료되는 프로세스의 모든 쓰레드는 완료되었음을 보장합니다 (락은 FIFO 순서로 얻어짐을 기억하세요!).

다시 말하건대, 이것들을 일반적인 마이크로프로세서 위에서 수행하려 시도하지 마세요. 무엇보다도, hard real-time 사용을 위해 설계된 시스템을 사용할 권한을 얻기도 충분히 어렵습니다! □

Quick Quiz 17.9:

하지만 Figure 17.12 의 `boostee()` 함수는 그 락을 반대 순서로 잡고 있어요! 이는 deadlock 을 초래할 수 있지 않을까요? ■

Answer:

Deadlock 은 일어나지 않을 겁니다. Deadlock 이 일어나려면, 두개의 다른 쓰레드가 각각 두개의 락을 반대 순서로 잡아야 하는데, 이 예에서는 그런 일은 없습니다. 하지만, lockdep [Cor06a] 과 같은 deadlock detector 들은 이에 거짓 양성 반응을 보일 수 있습니다. □

Quick Quiz 17.10:

그래서 많은 사람들이 락킹을 대신하는 작업을 시작하고는 대부분은 락킹을 최적화 하는 것으로 결론을 내리나요??? ■

Answer:

그들은 최소한 어떤 유용한 것을 얻습니다! 그리고 시간이 흐름에 따라 HTM 에 추가적인 진보가 있을 수 있습니다. □

D.18 Important Questions

Quick Quiz A.1:

이 예제에서 어떤 SMP 코딩 어러가 보이거나요? 전체 코드를 보기 위해선 `time.c` 파일을 보세요. ■

Answer:

- 루프에서 `barrier()` 나 `volatile` 을 사용하지 않음.
- 업데이트 쪽에서 메모리 배리어를 사용하지 않음.
- 생성자와 소비자 사이의 동기화가 없음. □

Quick Quiz A.2:

소비자의 연속적인 읽기들 사이에 어떻게 그렇게 큰 간격이 존재하게 된걸까요? 전체 코드를 위해선 `timelocked.c` 파일을 보세요. ■

Answer:

1. 소비자는 긴 시간동안 preemption 당했을 수 있습니다.

2. 오랫동안 수행되는 인터럽트가 소비자를 지연시켰을 수 있습니다.

3. 생성자는 소비자가 수행되는 CPU 에 비해 더 빠른 CPU 위에서 수행되었을 수 있습니다 (예를 들어, CPU 들 가운데 하나는 발열 처리나 에너지 소비 제한에 의해 자신의 클럭 주파수를 낮췄을 수 있습니다). □

Quick Quiz A.3:

RCU read-side 기능만을 유일한 동기화 수단으로 사용하는 프로그램의 한 부분을 생각해 봅시다. 이는 parallelism 또는 concurrency 인가요? ■

Answer:

그렇습니다. □

Quick Quiz A.4:

두번째 (스케줄러 기반의) 관점의 어떤 부분에서 락 기반의 CPU 당 하나의 쓰레드를 사용하는 워크로드가 “concurrent” 로 여겨질 수 있을까요? ■

Answer:

해당 워크로드를 임의의 형태로 나누고 끼워넣으려 하는 사람입니다. 물론, 임의의 분할은 락 획득을 연관된 락 해제로부터 떼어내는 일로 귀결될 것인데, 이는 다른 쓰레드가 락을 획득하려 하는 것을 방지하게 될겁니다. 만약 그 락이 순수한 스피너라면, 이는 데드락을 초래 할수도 있습니다. □

D.19 Why Memory Barriers?

Quick Quiz B.1:

Writeback 메세지는 어디서 와서 어디로 가나요? ■

Answer:

Writeback 메세지는 해당 CPU 에서, 또는 일부 설계에서는 해당 CPU 의 캐시의 해당 레벨에서 발생합니다— 또는 심지어 여러 CPU 들 사이에 공유된 캐시에서도. 핵심은 해당 캐시는 현재 데이터 아이템을 위한 공간이 없어서 공간을 만들기 위해 캐시에서 일부 데이터를 제거해야 한다는 겁니다. 다른 캐시나 메모리에 데이터의 복사본이 일부 존재한다면, 그 부분은 writeback 메세지 필요 없이 그냥 버려질 수도 있습니다.

반면, 만약 제거될 데이터의 모든 부분이 수정된 상태여서 최신 버전이 이 캐시에만 존재한다면, 그런 데이터

아이템들은 어딘가 다른곳에 복사되어야만 합니다. 이 복사 과정이 “writeback 메세지”를 통해 이루어집니다.

Writeback 메세지의 목적지는 새 값을 쓸 수 있는 곳이어야 합니다. 이는 메인 메모리가 될 수도 있지만, 다른 캐시일 수도 있습니다. 만약 그게 캐시라면, 그 캐시는 보통 같은 CPU 의 높은 레벨 캐시로, 예를 들어, 레벨-1 캐시는 레벨-2 캐시에 writeback 을 할 수도 있습니다. 하지만, 일부 하드웨어 설계는 CPU 간 writeback 을 허용해서, CPU 0 의 캐시는 CPU 1 에 writeback 메세지를 날릴 수 있느릅니다. 이는 보통 CPU 1 이 어떻게든, 예를 들어, 최근에 읽기 리퀘스트를 했다던지와 같이 그 데이터에 흥미를 표했다면 행해질 수 있습니다.

한마디로, writeback 메세지는 공간이 부족한 시스템의 어떤 부분에서 보내질 수 있고, 그 데이터를 수용할 수 있는 시스템의 다른 부분에서 받게 됩니다. □

Quick Quiz B.2:

두개의 CPU 들이 같은 캐시 라인을 동시에 무효화 하려고 하면 어떻게 되나요? ■

Answer:

그 CPU 들 중 하나가 먼저 공유 버스에의 액세스를 얻고, 그 CPU 가 “이깁니다”. 이기지 못한 CPU 는 해당 캐시라인의 카피를 무효화 시키고 “invalidate acknowledge” 메세지를 이긴 CPU 에게 보내야 합니다.

물론, 진 CPU 는 곧바로 “read invalidate” 요청을 보낼 것이라 예상할 수 있고, 따라서 이긴 CPU 의 승리는 덧없는 것이 될 겁니다. □

Quick Quiz B.3:

커다란 멀티프로세서에서 “invalidate” 메세지가 생기면, 모든 CPU 가 “invalidatge acknowledge” 응답을 보내야만 합니다. 그로 인한 “invalidate acknowledge” 의 “폭풍우” 가 시스템 버스를 완전히 뒤덮지 않을까요? ■

Answer:

커다란 규모의 멀티프로세서가 그렇게 구현되어 있다면 그렇겠죠. 커다란 멀티프로세서들, 특히 NUMA 구조의 경우에는, “dicrectory-based” 라 불리는 캐시 일관성 프로토콜을 사용해서 이런 문제 등의 여러 문제들이 나타나지 않게 합니다. □

Quick Quiz B.4:

SMP 머신들이 실제로 메세지 전달을 어떻게든 사용한다면, 애초에 왜 SMP 에 신경을 쓰는거죠? ■

Answer:

이 주제에 대해서는 지난 수십년 동안 상당한 논쟁이 있었습니다. 한 대답은 캐시 일관성 프로토콜은 상당히 간단해서 하드웨어만으로 구현되어 소프트웨어 메세지 전달로는 얻을 수 없는 대역폭과 대기시간을 얻을

수 있다는 겁니다. 또 다른 대답은 진정한 사실은 커다란 SMP 머신과 작은 SMP 머신의 클러스터의 상대적 가격으로 인한 경제 규모에서 찾을 수 있습니다. 세번째 대답은 SMP 프로그래밍 모델이 분산 시스템의 것보다 쉽다는 겁니다만 HPC 클러스터들과 MPI 의 출현을 가지고 반박할 수도 있겠습니다. 그렇게 논쟁은 계속되는거죠. □

Quick Quiz B.5:

앞에 설명된 지연되는 상태 변경들은 하드웨어에서 어떻게 처리하나요? ■

Answer:

추가적인 상태를 더해서 처리합니다만, 한번에 일부 라인들만 상태 변경을 한다는 사실 때문에 이 추가적인 상태들이 실제로 캐시 라인에 쓰여질 필요는 없습니다. 상태 변경을 지연해야 하는 요구사항은 실제 세계 캐시 일관성 프로토콜을 이 부록에서 설명된 지나치게 간략화된 MESI 프로토콜에 비해 훨씬 복잡하게 만드는 문제 중 하나입니다. Hennessy 와 Patterson 의 컴퓨터 구조에 대한 오래된 소개 문서 [HP95] 에서 이런 문제들을 다룹니다. □

Quick Quiz B.6:

어떤 오퍼레이션들의 시퀀스가 CPU 의 캐시를 모두 “invalid” 상태로 돌려 놓을까요? ■

Answer:

해당 CPU 의 인스트럭션 셋에 특별한 “내 캐시 비우기” 인스트럭션이 있지 않다면 그런 시퀀스는 없습니다. 대부분의 CPU 들은 그런 인스트럭션을 갖습니다. □

Quick Quiz B.7:

하지만 스토어 버퍼의 주목적이 멀티프로세서 캐시 일관성 프로토콜의 응답 지연시간을 숨기기 위해서라면, 왜 유니프로세서들도 스토어 버퍼를 가지고 있는거죠? ■

Answer:

스토어 버퍼의 목적은 멀티프로세서 캐시 일관성 프로토콜의 응답 지연시간을 숨기는 것만이 아니라 일반적인 메모리 지연시간을 숨기려는 것이기 때문입니다. 메모리는 유니프로세서의 캐시보다 훨씬 느리기 때문에, 유니프로세서의 스토어 버퍼는 write-miss 대기시간을 줄이는데 도움을 줄 수 있습니다. □

Quick Quiz B.8:

앞의 step 1에서, 왜 CPU 0 는 그냥 “invalidate” 가 아니라 “read invalidate” 를 보내는거죠? ■

Answer:

해당 캐시 라인은 변수 *a* 외에도 많은 정보를 담고 있기 때문입니다. □

Quick Quiz B.9:

Section B.4.3 의 첫번째 시나리오의 스텝 1에서, 왜 “read invalidate” 가 아니라 “invalidate” 를 보내는 거죠? CPU 0 는 “*a*” 외에도 이 캐시 라인을 공유하는 다른 변수들의 값을 필요로 할수도 있지 않나요? ■

Answer:

CPU 0 는 “*a*” 를 담고 있는 캐시 라인의 read-only 복사본을 가지고 있기 때문에, 해당 캐시 라인을 공유하는 다른 변수들의 값을 이미 알고 있습니다. 따라서 “invalidate” 메세지 만으로도 충분합니다. □

Quick Quiz B.10:

뭐라구요??? CPU 는 while 루프가 끝나기 전까지는 assert() 를 수행할 수 없는데 왜 여기에 메모리 배리어가 필요해요? ■

Answer:

CPU 들은 예측적으로 수행을 할 수 있어서 while 루프가 완료되기 전에 단정문을 수행하는 것과 같은 효과를 일으킬 수 있습니다. 더욱이, 컴파일러들은 일반적으로 현재 수행중인 쓰레드만이 해당 변수의 값을 수정한다고 가정하고, 따라서 이 가정으로 인해 컴파일러는 *a* 읽기를 루프보다 앞으로 당겨올 수 있습니다.

사실, 일부 컴파일러들은 이를 루프를 다음과 같이 무한루프를 감싸는 브랜치로 바꿀수도 있습니다:

```

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 }
```

이런 최적화가 만들어지면, 해당 단정문은 아예 수행이 안될 수도 있습니다. 컴파일러가 당신의 병렬 코드를 없애버리는 현상을 막기 위해서는 volatile 캐스팅이나 (가능하다면) C++ relaxed atomic 들을 사용해야 합니다.

한마디로, 컴파일러들도 CPU 들도 최적화를 공격적으로 하기 때문에, 컴파일러 지시어나 메모리 배리어를 이용해 당신의 제약을 분명하게 그것들에게 알려야 합니다. □

Quick Quiz B.11:

각 CPU 는 자신의 메모리 액세스를 순서대로 보게 된다는 보장 사항은 각 유저 레벨 쓰레드 역시 자신의 메모리 액세스를 순서대로 볼 것이라고 보장하나요? 그렇다면 왜 그렇고, 아니라면 왜 아니죠? ■

Answer:

아니요. 한 쓰레드가 한 CPU 에서 다른 CPU 로 옮겨 가는데, 옮겨져 가는 CPU 는 해당 쓰레드가 옮겨져 온 기존 CPU 에서 한 최근의 메모리 오퍼레이션들을 다른 순서대로 인지할 수 있는 경우를 생각해 보세요. 유저 모드에서의 일관성을 위해선, 커널 해커들이 컨텍스트 스위치 경로에서마다 메모리 배리어를 반드시 사용해 줘야 합니다. 하지만, 이미 컨텍스트 스위치를 안전하게 하기 위해 필요시되는 락킹이 유저 레벨 태스크가 자기 자신의 액세스들을 순서대로 볼 수 있도록 하는데 필요한 메모리 배리어들을 제공해야만 하게 되어 있습니다. 그렇다면 해도, 커널 레벨에서든 유저 레벨에서든 당신이 매우 최적화된 스케줄러를 설계하고 있다면, 부디 이 시나리오를 기억해 두세요! □

Quick Quiz B.12:

CPU 1 의 “while” 과 “*c*” 값 대입문 사이에 메모리 배리어를 넣는 것으로 이 코드를 고칠 수 있지 않을까요? 고칠 수 있다면 왜이고 고칠 수 없다면 그건 또 왜죠? ■

Answer:

고칠 수 없습니다. 그런 메모리 배리어는 그저 CPU 1 에 지역적인 순서만을 강제합니다. CPU 0 의 액세스들과 CPU 1 의 액세스들 사이의 상대적인 순서에 대해선 어떤 영향도 끼치지 않으므로, 단정문은 여전히 실패할 수 있습니다. 하지만, 모든 주류 컴퓨터 시스템들은 “타동성” 을 제공하기 위한 하나 또는 여러개의 메커니즘을 제공해서, 직관적인 인과 순서를 제공합니다: B 가 A 의 액세스들로 인한 현상을 보게 되면, 그리고 C 가 B 의 액세스들로 인한 현상을 보면, C 는 A 의 액세스들로 인한 현상 역시 봐야만 합니다. 짧게 말해서, 하드웨어 설계자들은 소프트웨어 개발자들을 위한 조금의 연민 정도는 가지고 있습니다. □

Quick Quiz B.13:

Table B.4 에서 CPU 1 과 2 의 line 3-5 가 인터럽트 핸들러 안에서 수행되고, CPU 2 의 line 9 는 프로세스 레벨에서 수행된다고 생각해 봅시다. 코드가 정확히 동작하도록 하는데, 달리 말하자면 단정문이 실패하지 않도록

하는데 어떤 변경이 필요할까요? 필요하다면 무엇일까요? ■

Answer:

“e” 의 로드 오퍼레이션이 “a” 의 로드 오퍼레이션 이전에 이루어졌음을 분명히 하도록 단정문이 쓰여져야 합니다. 리눅스 커널에서는 barrier() 를 앞의 예들에서 단정문에 사용되었던 메모리 배리어처럼 사용되는 것으로 이를 달성할 수 있을 것입니다. □

Quick Quiz B.14:

Table B.4 의 예에서 CPU 2 가 assert (e==0 || c==1) 을 수행하면 단정문은 실패할 수 있을까요? ■

Answer:

해당 CPU 가 “타동성” 을 지원하는가에 달려 있습니다. 달리 말하자면, CPU 0 는 CPU 0 의 “c” 로부터의 로드와 “e” 로의 스토어 사이의 메모리 배리어와 함께 CPU 1 의 “c” 에의 스토어 오퍼레이션을 바라본 후에 “e” 에 스토어를 행했습니다. 만약 다른 CPU 가 CPU 0 의 “e” 에의 스토어를 본다면, 그 CPU 는 CPU 1 의 스토어도 볼 것이라고 보장될 수 있을까요?

제가 신경쓰는 모든 CPU 들은 타동성을 제공한다고 이야기합니다. □

Quick Quiz B.15:

Alpha 의 smp_read_barrier_depends () 는 smp_rmb () 가 아니라 smp_mb () 인가요? ■

Answer:

첫째로, Alpha 는 mb 와 wmb 인스트럭션들만 가지고 있고, 따라서 smp_rmb () 는 어떤 경우든 Alpha mb 인스트럭션으로 구현됩니다.

더 중요한 건, smp_read_barrier_depends () 는 뒤의 스토어들을 순서맞춰야 한다는 겁니다. 예를 들어, 다음의 코드를 생각해 보세요:

```

1 p = global_pointer;
2 smp_read_barrier_depends ();
3 if (do_something_with(p->a, p->b) == 0)
4     p->hey_look = 1;

```

여기서 p->a 와 p->b 의 로드만이 아니라 p->hey_look 로의 스토어 역시 순서를 맞춰야 합니다. □

Dictionaries are inherently circular in nature.

*“Self Reference in word definitions”,
David Levary et al.*

Appendix E

Glossary and Bibliography

Associativity: 캐시가 동시에 쥐고 있을 수 있는, 해당 캐시 내에서 동일하게 해시되는 캐시라인들의 갯수. 각각의 해시 값 별로 네개의 캐시 라인을 들고 있을 수 있는 캐시는 “four-way set-associative” 캐시라 명칭되며, 각각의 해시 값 별로 한개의 캐시 라인만을 들고 있을 수 있는 캐시는 “direct-mapped” 캐시라고 불리워집니다. Associativity 가 그 용량과 동일한 캐시는 “fully associative” 캐시라 불리워집니다. Fully associative 캐시는 associativity 미스를 제거할 수 있는 장점이 있습니다만, 하드웨어적 한계 때문에, fully associative 캐시는 일반적으로 크기가 제한됩니다. 최신 마이크로프로세서에 장착되는 커다란 캐시의 associativity 는 일반적으로 two-way에서 eight-way 사이입니다.

Associativity Miss: 연관된 CPU 가 최근에 캐시의 특정 set 이 담고 있을 수 있는 것보다 더 많은, 해당 set 으로 해시되는 데이터를 접근해서 발생하는 캐시 미스. Fully associative 캐시는 associativity 미스를 일으키지 않습니다 (또는, 동일하게, fully associative 캐시에서, associativity 와 capacity 미스는 동일하다고 말할 수 있겠습니다).

Atomic: 특정 오퍼레이션은 그 중간 상태를 관찰할 수 없도록 되어 있다면 “atomic” 하다고 여겨집니다. 예를 들어, 대부분의 CPU 에서, 올바르게 정렬된 포인터로의 store 는 어토믹한데, 다른 CPU 들은 기존 값 또는 새로운 값을 볼 수 있지만, 새로운 값의 일부와 기존 값의 일부가 결합된 값을 볼 수는 없도록 보장되기 때문입니다.

Cache: 근래의 컴퓨터 시스템들에서 CPU 는 빈번하게 사용되는 데이터를 담고 있을 수 있는 캐시를 갖습니다. 이런 캐시는 매우 간단한 해시 함수를 사용하는 하드웨어로 구현된 해시 테이블로 생각될 수 있습니다만, 각각의 해시 bucket (하드웨어 타입에서는 “set”이라 명명됩니다) 은 제한된 수의 데이터 항목만을

담고 있을 수 있습니다. 각각의 캐시의 해시 bucket 이 담고 있을 수 있는 데이터 항목의 갯수는 해당 캐시의 “associativity” 라고 명칭됩니다. 이런 데이터 항목들은 일반적으로 “cache line” 이라 불리우는데, CPU 와 메모리 사이에 돌아다니는 고정된 길이의 데이터 블락이라고 생각될 수 있습니다.

Cache Coherence: 모든 CPU 들이 특정 변수에 대해 일련의 변화되는 값을 보게 되는 대부분의 근래의 SMP 기계들의 속성은 적어도 해당 변수에 대해 하나의 전역적인 값 변화의 순서로 일관성을 갖는다는 겁니다. Cache coherence 또한 특정 변수에 대한 일련의 store 들의 마지막에는 모든 CPU 들이 그 변수의 마지막 값에 대해 동의를 할 것을 보장해 줍니다. Cache coherence 는 하나의 변수에 의해 얻어진 일련의 값들에 대해서만 적용됨을 알아두시기 바랍니다. 반면에, 메모리 일관성 모델은 여러개의 변수들에 대한 load 와 store 가 어떻게 일어난 것으로 보이게 되는지를 설명합니다. 더 자세한 내용을 위해선 Section 14.2.4.2 을 참고하세요.

Cache Coherence Protocol: 메모리 일관성과 순서를 강제하고 다른 CPU 들이 각자의 캐시에 담겨있는 데이터의 일관적이지 않은 상태를 보는 것을 막아주는, 일반적으로 하드웨어로 구현되는 통신 프로토콜.

Cache Geometry: 캐시의 크기와 associativity 는 그것의 geometry 라 불리웁니다. 각각의 캐시는 이차원 배열로, 캐시 라인들의 행 (“set”) 은 같은 해시 값을 갖고, 캐시 라인들의 열 (“ways”) 는 그 안의 모든 캐시 라인이 서로 다른 해시 값을 갖는 것으로 생각될 수 있습니다. 특정 캐시의 associativity 는 그 열의 수이고 (따라서 “way” 라 불립니다—two-way set-associative 캐시는 두개의 “way” 를 갖습니다), 캐시의 크기는 행의 수와 열의 수의 곱입니다.

Cache Line: (1) CPU 와 메모리 사이를 돌아다니는 데

이터의 단위로, 일반적으로 그 크기는 2의 거듭제곱입니다. 일반적인 캐시라인의 크기는 16 바이트에서 256 바이트 사이입니다.

(2) 데이터의 캐시라인 하나 단위를 담아둘 수 있는 CPU 캐시의 물리적 위치.

(3) 데이터의 캐시라인 하나 단위를 담아둘 수 있는, 그러나 또한 캐시 라인 경계로 정렬되어 있는 메모리 상의 물리적 위치. 예를 들어, 256-바이트 캐시라인을 갖는 시스템에서 메모리 상의 캐시 라인의 첫번째 word의 주소는 0x00으로 끝날 겁니다.

Cache Miss: CPU가 필요로 하는 데이터가 해당 CPU의 캐시 안에 존재하지 않으면 하나의 캐시 미스가 발생합니다. 해당 데이터는 여러가지 이유로 캐시에 없을 수 있는데, 다음과 같은 이유들이 있을 수 있습니다: (1) 이 CPU는 기존에 해당 데이터를 액세스한 적이 없거나("startup" 또는 "warmup" 미스), (2) 이 CPU는 최근에 자신의 캐시가 담고 있을 수 있는 것보다 더 많은 데이터에 액세스를 해서, 오래된 데이터 중 일부는 제거되었거나("capacity" 미스), (3) 이 CPU는 특정 set에 해당 set¹이 담아둘 수 있는 것보다 많은 데이터에 액세스를 했거나("associativity" 미스), (4) 이 CPU가 해당 데이터에 액세스한 후에 어떤 다른 CPU가 해당 데이터에 (또는 같은 캐시 라인의 어떤 다른 데이터에) 쓰기를 수행해서 ("communication miss"), 또는 (5) 이 CPU가 해당 캐시라인이 다른 CPU의 캐시에 복사되어 있거나 해서 read-only 인데 쓰기를 수행해서입니다.

Capacity Miss: 연관된 CPU가 최근에 캐시가 담을 수 있는 것보다 많은 데이터를 액세스 해서 발생하는 캐시 미스.

Code Locking: 크리티컬 섹션의 한 집합을 보호하기 위해 "global lock"이 사용되어서 특정 쓰레드의 그런 집합으로의 액세스가 그 쓰레드가 어떤 데이터에 접근하려 하는가에 의해서가 아니라 현재 그 크리티컬 섹션들의 집합을 차지하고 있는 쓰레드들의 집합에 기반해서만 허용되거나 거부되는 간단한 락킹 디자인. Code-locking을 사용하는 프로그램의 확장성은 그 코드에 의해 제한됩니다; 해당 데이터 집합의 크기를 증가시키는 것은 일반적으로 확장성을 증가시키지 못할 겁니다 (사실, 일반적으로는 "lock contention"을 증가시켜서 확장성을 떨어뜨립니다). "Data locking"에 대비됩니다.

¹ 하드웨어 캐시 용어에서, "set"이라는 말은 소프트웨어 캐시를 논할 때의 "bucket" 이란 말과 동일한 의미로 사용됩니다.

Communication Miss: 이 CPU가 해당 캐시라인에 액세스한 후에 어떤 다른 CPU가 해당 캐시라인에 액세스해서 발생하는 캐시미스.

Critical Section: 어떤 동기화 메커니즘으로 보호되어서 그 수행이 그 기능으로 제한되는 코드의 부분. 예를 들어, 크리티컬 섹션들의 한 집합이 같은 global lock으로 보호된다면, 한번에 그 크리티컬 섹션들 가운데 하나만이 수행됩니다. 만약 한 쓰레드가 그런 크리티컬 섹션 가운데 하나를 수행하고 있다면, 모든 다른 쓰레드는 그 첫번째 쓰레드가 해당 집합 내의 크리티컬 섹션들의 수행을 모두 완료하기 전까지는 대기해야만 합니다.

Data Locking: 각각의 데이터 구조 인스턴스가 각자의 락을 가지고 있는 확장성 있는 락킹 디자인. 각각의 쓰레드가 이 데이터 구조의 서로 다른 인스턴스를 사용한다면, 이 모든 쓰레드들은 동시에 크리티컬 섹션을 수행하고 있을 수 있습니다. 데이터 락킹은 증가하는 CPU의 수와 증가하는 데이터 인스턴스의 갯수에 따라 자동적으로 확장되는 장점을 갖습니다. "Code locking"과 대비됩니다.

Direct-Mapped Cache: 하나의 way만을 가지고 있어서, 하나의 해시값마다 하나의 캐시라인만을 담고 있을 수 있는 캐시.

Embarrassingly Parallel: 쓰레드를 추가하는 것이 연산 과정의 전체적인 비용을 크게 증가시키지 않아서 (충분한 CPU가 사용 가능한 상태라는 가정 하에) 쓰레드가 추가됨에 따라서 선형적으로 속도가 증가하게 되는 문제나 알고리즘.

Exclusive Lock: 한번에 하나의 쓰레드만이 해당 락이 보호하고 있는 크리티컬 섹션에 들어갈 수 있도록 하는 상호 배타적인 메커니즘.

False Sharing: 두개의 CPU가 각각 한 쌍의 데이터 항목 중 하나씩에 빈번하게 쓰기를 하지만, 이 한쌍의 데이터 항목이 같은 캐시 라인에 위치해 있다면, 이 캐시 라인은 반복적으로 무효화되어서, 이 두 CPU의 캐시들 사이를 "ping-pong" 하길 반복하게 됩니다. 이는 "cache bouncing"이라고도 불리는 "cache thrashing" 현상의 혼란 원인입니다(리눅스 커뮤니티에서는 앞의 표현이 더 혼합니다). False sharing은 성능과 확장성을 모두 크게 떨어뜨릴 수 있습니다.

Fragmentation: 많은 양의 사용되지 않은 메모리가 존재하지만 비교적 작은 메모리 할당 요청들을 충족해줄 수 있도록 맞춰져 있지 않은 메모리 풀을 단편화(fragment) 되었다고 합니다. 할당된 메모리

블록들 사이에 작은 조각의 메모리 공간이 존재 하도록 메모리 공간이 쪼개졌을 때 external fragmentation 이 발생하며, 특정한 또는 특정한 종류의 메모리 할당 요청이 실제로 필요한 것보다 많은 메모리를 할당받았을 때 internal fragmentation 이 발생합니다.

Fully Associative Cache: Fully associative 캐시는 하나의 set 만을 가지고 있어서, 해당 set 은 그 용량이 허용하는 한 메모리의 어떤 부분집합도 담고 있을 수 있습니다.

Grace Period: Grace period 는 어떤 시간 간격으로, 해당 시간 간격에 앞서 시작된 모든 RCU read-side 크리티컬 섹션은 이 시간 간격의 종료 전에 종료됩니다. 많은 RCU 구현들은 각각의 쓰레드가 최소 하나의 quiescent state 를 지나는 사이의 시간 간격으로 정의합니다. RCU read-side 크리티컬 섹션은 그 정의에 의해 quiescent state 를 포함할 수 없으므로, 이 두개의 정의는 거의 항상 서로 교체될 수 있습니다.

Heisenbug: 추적하기 위해 기록을 추가하거나 print 문을 추가하면 발견되지 않게 되는, 타이밍에 민감한 버그.

Hot Spot: 매우 많이 사용되어서 연관된 락에 대한 높은 수준의 contention 을 초래하는 데이터 구조. 이런 상황의 한 예로는 형편없는 해시 함수를 사용한 해시 테이블이 있을 수 있습니다.

Humiliatingly Parallel: 쓰레드를 추가하면 연산의 전체 비용이 상당히 줄어들어서(충분한 CPU 가 사용 가능한 상태라는 가정 하에) 쓰레드를 추가함에 따라 선형적 속도증가를 초월하는 속도 증가를 초래하는 문제나 알고리즘.

Invalidation: 한 CPU 가 하나의 데이터 항목에 쓰기를 하려 하면, 해당 CPU 는 먼저 이 데이터 아이템이 다른 CPU 의 캐시에는 존재하지 않음을 보장해야 합니다. 만약 필요하다면, 쓰기를 하는 CPU 는 해당 데이터의 사본을 자신의 캐시에 담고 있는 모든 CPU 에게 “invalidation” 메세지를 보내서 다른 CPU 의 캐시로부터 해당 항목을 제거합니다.

IPI: 하나의 CPU 에서 다른 CPU 로 보내어지는 프로세서 간의 인터럽트. IPI 는 리눅스 커널에서 많이 사용되는데, 예를 들어 스케줄러가 CPU 에게 높은 우선순위 프로세스가 현재 수행 가능 상태임을 알리는데 사용됩니다.

IRQ: 인터럽트 요청 (Interrupt request) 으로, 리눅스 커널 커뮤니티에서는 “interrupt” 의 약자로 사용되어서, “irq handler” 와 같은 용례로 사용됩니다.

Linearizable: 일련의 오퍼레이션들은 모든 CPU 들 과 / 또는 쓰레드들의 관찰된 내용과 일관적인 전체적인 해당 오퍼레이션들의 순서가 존재한다면 “linearizable” 하다고 표현합니다. Linearizability 는 많은 연구자들에 의해 중요시됩니다만, 실용적인 면에서는 기대되는 것보다 유용하지 못합니다 [HKLP12].

Lock: 크리티컬 섹션을 보호하는데 사용될 수 있는 소프트웨어 추상화의 하나로, “상호 배타성 (mutual exclusion) 메커니즘” 의 한 예. “배타적 락” 은 한번에 하나의 쓰레드만이 해당 락으로 보호되는 크리티컬 섹션들의 집합에 들어갈 수 있도록 해주며, “reader-writer lock” 은 읽기를 하는 쓰레드라면 몇 개의 쓰레드든지, 쓰기를 하는 쓰레드라면 하나의 쓰레드만이 해당 락으로 보호되는 크리티컬 섹션들의 집합에 들어갈 수 있도록 해줍니다. (분명히 하자면, 주어진 reader-writer 락의 크리티컬 섹션들 가운데 하나라도 쓰기 쓰레드가 존재한다면, 어떤 읽기를 하는 쓰레드도 해당 락의 크리티컬 섹션들 가운데 하나도 들어갈 수 없을 것이고, 그 반대도 마찬가지입니다.)

Lock Contention: 락은 매우 많이 사용되어 해당 락을 얻으려 기다리는 CPU 가 많은 경우 contention 을 겪는다고 표현됩니다. 병렬 알고리즘을 설계하고 병렬 프로그램을 구현할 때에 Lock contention 을 줄이는 것이 많은 경우 중요한 고심거리입니다.

Memory Consistency: 변수들의 그룹에 대한 액세스들의 순서가 어떻게 일어난 것을 보여야만 하는가에 대한 제약을 만들어내는 속성들의 집합. 메모리 일관성 모델은 학계에서 인기있으며 매우 강력한 모델인 sequential consistency 부터 process consistency, release consistency, 그리고 weak consistency 까지 다양합니다.

MESI Protocol: Modified, exclusive, shared, 그리고 invalid (MESI) 상태를 갖는 캐시 일관성 프로토콜로, 이 때문에 이 프로토콜은 해당 캐시의 캐시라인이 가질 수 있는 상태들의 이름을 본따서 그렇게 이름 지어졌습니다. Modified 상태의 캐시라인은 최근에 이 CPU 에 의해 쓰기가 되어진 상태로, 연관된 메모리 영역의 현재 값의 유일한 표현입니다. Exclusive 상태의 캐시 라인은 값이 새로 씌여지지는 않았지만, 해당 캐시라인은 다른 CPU 의 캐시에 복사되지 않았음이 보장되기에 이 CPU 가 언제든

그 위에 쓰기를 할 권리를 가진 상태입니다 (메인 메모리 상의 연관된 영역은 최신 버전이지만 말입니다). Shared 상태의 캐시라인은 어떤 다른 CPU의 캐시에 복사되어 있는 (또는 복사되어 있을 수 있는) 상태로, 이 CPU는 이 캐시 라인에 쓰기를 하기 전에 그런 다른 CPU들과 상호작용을 해야만 합니다. Invalid 상태의 캐시라인은 값을 포함하고 있지 않아서, 메모리로부터 데이터가 로드되어질 수 있는 상태인 해당 캐시의 “빈 공간”을 나타냅니다.

Mutual-Exclusion Mechanism: 쓰레드의 “크리티컬 섹션”과 연관된 데이터로의 액세스를 조절하는 소프트웨어 추상화.

NMI: Non-maskable interrupt. 이름이 이야기하듯, 이는 가려질 수 없는, 매우 높은 우선순위의 인터럽트입니다. 이것들은 프로파일링과 같은 하드웨어의 특수한 목적에 사용됩니다. 프로파일링을 하는 데에 NMI를 사용함으로써 얻을 수 있는 이득은, 인터럽트를 비활성화 시킨 채 수행되는 코드에 대해서도 프로파일링을 할 수 있다는 것입니다.

NUCA: CPU 들의 그룹들이 캐시를 공유하는, Non-uniform cache architecture. 따라서 한 그룹 내의 CPU들은 다른 그룹 내의 CPU들과 할 수 있는 것에 비해 같은 그룹 내의 CPU들과 캐시라인을 더 빠르게 교환할 수 있습니다. 하드웨어 쓰레드를 갖는 CPU로 구성된 시스템에서는 일반적으로 NUCA 아키텍처를 갖게 됩니다.

NUMA: 메모리가 뱅크 단위로 쪼개어져 있고 각각의 뱅크는 “NUMA node”라 불리우는, 특정 CPU들의 그룹에 “가까운” 구조인 Non-uniform memory architecture. NUMA 기계의 한 예는 Sequent의 NUMA-Q 시스템으로, 네개의 CPU들로 구성된 각각의 그룹들이 하나씩 메모리 뱅크를 근처에 두고 있습니다. 특정 그룹 내의 CPU들은 그들의 메모리에 다른 그룹의 메모리보다 훨씬 빨리 액세스 할 수 있습니다.

NUMA Node: 커다란 NUMA 기계 내에서의 가까이 위치한 CPU들과 거기 연관된 메모리의 그룹. 하나의 NUMA 노드는 NUMA 구조를 가질 수도 있음을 알아 두시기 바랍니다.

Pipelined CPU: 많은 장점과 단점을 가지고 있는, 어떤 의미에선 조립 공정 라인과 비슷한, CPU 내부의 인스트럭션들의 내부 흐름인 pipeline을 가진 CPU. 1960년대부터 1980년대 초까지, pipeline을 내장한 CPU는 슈퍼컴퓨터의 영역에만 있었습니다만,

1980년대 후반부터는 (80486과 같은) 마이크로프로세서에서도 나타나기 시작했습니다.

Process Consistency: 각각의 CPU가 자신의 스토어 오퍼레이션은 program order대로 발생한 것으로 보이지만 다른 CPU들은 복수의 CPU들로부터의 액세스를 다른 순서로 발생한 것으로 볼 수 있는 메모리 일관성 모델.

Program Order: 특정 쓰레드의 인스트럭션들이 지금에 와서는 미신적 존재가 되어버린, 다음 인스트럭션의 수행이 진행되기 전에 현재 인스트럭션을 완전히 수행하는, “in-order” CPU에 의해 수행되는 순서. (그런 CPU들이 이제는 고대의 미신과 전설상의 존재로만 남게 된 것은 그것들은 엄청나게 느리기 때문입니다. 이 고대의 공룡들은 Moore's Law가 가져온 CPU 클럭 주파수 향상의 많은 희생자들 가운데 하나입니다. 어떤 사람들은 이 야수들이 지구를 다시 한번 더 정복할 것이라고 이야기 하지만, 그 외의 사람들은 그런 말을 아예 무시합니다.)

Quiescent State: RCU에서, RCU로 보호되는 데이터 구조로의 레퍼런스가 하나도 잡혀 있지 않은, 일반적으로는 RCU read-side 크리티컬 섹션의 바깥의 모든 지점. 모든 쓰레드가 최소한 하나의 quiescent state를 지나는 동안의 모든 시간 간격은 “grace period”라고 불리워집니다.

Read-Copy Update (RCU): Reader-writer 락킹이나 reference counting의 대체물로 생각될 수 있는 하나의 동기화 메커니즘. RCU는 읽기 쓰레드들에게 매우 낮은 오버헤드의 액세스를 제공하는 반면, 쓰기 쓰레드에게는 앞서 존재한 읽기 쓰레드들의 성능 이득을 위해 기존 버전을 유지해야하는 추가적 오버헤드를 일으킵니다. 읽기 쓰레드들은 블락되지도 spin하지도 않고, 따라서 데드락에 참여될 수 없습니다만, 정상적인 데이터를 볼 수 있고 업데이트와 동시에 수행될 수 있습니다. 따라서 RCU는 정상적인 데이터가 (라우팅 테이블에서처럼) 지연되거나 (리눅스 커널의 System V IPC 구현에서처럼) 막아질 수 있는 읽기가 대부분인 환경에 가장 적합합니다.

Read-Side Critical Section: 어떤 reader-writer 동기화 메커니즘의 읽기 권한 획득으로 보호되는 코드의 일부분. 예를 들어, 한 크리티컬 섹션들의 집합이 특정 전역적 reader-writer 락의 읽기 권한 획득으로 보호된다면, 크리티컬 섹션들의 첫번째 집합은 그 락의 read-side 크리티컬 섹션이 됩니다. 수에 상관없이 복수의 쓰레드들이 read-side 크리티컬 섹션을 동시에 수행할 수 있습니다만, 어떤 쓰레드도

write-side 크리티컬 섹션을 수행하고 있지 않은 경우에만 그렇습니다.

Reader-Writer Lock: Reader-writer 락은 수에 관계없이 복수의 읽기를 하는 쓰레드들을, 또는 단 하나의 쓰기 쓰레드들을 해당 락으로 보호되는 크리티컬 섹션들의 집합으로 들어갈 수 있도록 허용하는 mutual-exclusion 메커니즘입니다. 쓰기를 하려는 쓰레드는 모든 앞서 존재한, 읽기를 하는 쓰레드들이 해당 락을 놓을 때까지 기다려야만 하고, 비슷하게, 앞서 존재한 쓰기가 존재한다면, 쓰기를 하려는 모든 쓰레드는 해당 쓰기 쓰레드가 해당 락을 놓기를 기다려야 합니다. Reader-writer 락에 있어서의 핵심 고려사항은 “fairness”입니다: 끝나지 않는 읽기 쓰레드의 연속은 쓰기 쓰레드를 starve 시킬 수 있고, 그 반대로 마찬가지입니다.

Sequential Consistency: 모든 메모리 레퍼런스가 하나의 전역적인 순서에 일관적인 순서로 일어난 것으로 보이도록, 그리고 각각의 CPU의 메모리 레퍼런스가 모든 CPU들에게 프로그램 순서로 일어난 것으로 보이도록 하는 메모리 일관성 모델.

Store Buffer: CPU에서 지연된 쓰기를 기록하며 그사이 연관된 캐시라인들은 해당 CPU에서 사용할 수 있도록 사용되는 CPU 내부의 레지스터들의 작은 집합. “Store queue”라고도 불립니다.

Store Forwarding: 소프트웨어에게 CPU가 메모리 오퍼레이션들을 program order대로 진행한 것으로 보일 것을 보장하기 위해 CPU가 스托어 버퍼와 캐시 사이를 오가도록 하는 하나의 조정.

Super-Scalar CPU: 하나의 scalar (non-vector) CPU는 여러개의 인스트럭션들을 동시에 수행할 수 있습니다. 이는 여러개의 인스트럭션들을 조립 공정의 방식으로 수행하는 pipelined CPU의 진화된 형태입니다—super-scalar CPU에서는, 각각의 pipeline의 stage가 하나의 인스트럭션 이상을 처리할 수 있게 됩니다. 예를 들어, 만약 조건이 정확히 맞아 떨어진다면, 1990년대 중반에 나온 Intel Pentium Pro CPU는 하나의 클락 사이클당 두개의 (그리고 어떤 경우엔 세개의) 인스트럭션들을 처리할 수 있습니다. 따라서, 200MHz Pentium Pro CPU는 초당 최대 4억개의 인스트럭션을 수행 완료하거나 “retire” 시킬 수도 있습니다.

Teachable: 교사가 완전히 이해할 수 있고, 따라서 가리키기 간편한 주제, 컨셉, 방법, 또는 메커니즘.

Transactional Memory (TM): 오퍼레이션들의 어토믹한 시퀀스이며 atomicity, consistency, isolation

을 제공하지만 durability를 제공하지 않는다는 점에서 고전적인 트랜잭션과는 다른 “트랜잭션”을 지원하는 공유메모리 동기화 설계. 트랜잭션 메모리는 하드웨어로 구현 (하드웨어 트랜잭션 메모리 또는 HTM이라 불립) 되거나, 소프트웨어로 구현 (소프트웨어 트랜잭션 메모리 또는 STM이라 불립) 되거나, 또는 하드웨어와 소프트웨어의 조합으로 구현 (“unbounded” 트랜잭션 메모리, 또는 UTM이라 불립) 될 수 있습니다.

Unteachable: 교사가 잘 이해하지 못하고 있고, 따라서 가르치기가 어려운 주제, 컨셉, 방법, 또는 메커니즘.

Vector CPU: 하나의 인스트럭션을 여러개의 데이터 항목에 동시적으로 적용시킬 수 있는 CPU. 1960년대부터 1980년대까지는 슈퍼컴퓨터들만이 vector 기능을 가지고 있었지만, x86 CPU들의 MMX와 PowerPC CPU들의 VMX의 발전은 벡터 처리를 대중들에게 가져다 주었습니다.

Write Miss: 캐시 라인이 다른 CPU의 캐시에 복사되어있거나 하는 이유로 read-only 상태인 캐시라인에 CPU가 쓰기를 시도했기 때문에 발생하는 캐시 미스.

Write-Side Critical Section: 어떤 reader-writer 동기화 메커니즘의 읽기 권한 획득으로 보호되는 코드의 부분. 예를 들어, 한 크리티컬 섹션들의 집합이 특정 global reader-writer 락의 쓰기 권한 획득으로 보호되어 있고 두번째 크리티컬 섹션 집합은 같은 reader-writer 락의 읽기 권한 획득으로 보호되어 있다면, 첫번째 크리티컬 섹션 집합은 해당 락의 write-side 크리티컬 섹션이 됩니다. 한번에 하나의 쓰레드만이 write-side 크리티컬 섹션을 수행할 수 있으며, 이 때 어떤 쓰레드도 동시에 연관된 read-side 크리티컬 섹션들을 수행하고 있지 않아야 합니다.

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-ieeemicro2006.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.
- [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. URL: http://www.bohmann.dk/articles/response_time_still_matters.html [broken, November 2016], July 2001.
- [Bow06] Maggie Bowman. Dividing the sheep from the goats. <http://www.cs.kent.ac.uk/news/2006/RBornat/>, February 2006.
- [Bra07] Reg Braithwaite. Don’t overthink fizzbuzz. <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. URL: <https://www.cs.unc.edu/~anderson/diss/bbbdiss.pdf>.
- [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. <https://pdos.csail.mit.edu/papers/sbw-phd-thesis.pdf>.
- [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. Parnas. 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 real-time Linux. URL: <http://lwn.net/Articles/106010/>, October 2004.
- [Cor04b] Jonathan Corbet. Finding kernel problems automatically. <http://lwn.net/Articles/87538/>, June 2004.
- [Cor04c] Jonathan Corbet. Realtime preemption, part 2. URL: <http://lwn.net/Articles/107269/>, 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.
- [Cor12a] Jon Corbet. ACCESS_ONCE(). <http://lwn.net/Articles/508991/>, August 2012.
- [Cor12b] Jon Corbet. Relocating RCU callbacks. <http://lwn.net/Articles/522262/>, October 2012.
- [Cor13a] Jonathan Corbet. Is the whole system idle? <http://lwn.net/Articles/558284/>, July 2013.
- [Cor13b] 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.
- [Dep06] Department of Computing and Information Systems, University of Melbourne. CSIRAC. <http://www.cis.unimelb.edu.au/about/csirac/>, 2006.
- [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, Swami-nathan 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. URL: <http://lwn.net/Articles/572740/>, November 2013.
- [Edg14] Jake Edge. The future of the realtime patch set. URL: <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.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/WebHomeLecturesParallelProgrammingExercises/2011hw04.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 data layout optimizations for multi-core processors. In *Proceedings of the 5th International Conference on High Performance*

- [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. URL: <http://lwn.net/Articles/397422/>, 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 Kinematik 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.
- [HL86] Frederick S. Hillier and Gerald J. Lieberman. *Introduction to Operations Research*. Holden-Day, 1986.
- [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.
- [HMDZ06] David Howells, Paul E. McKenney, Will Deacon, and Peter Zijlstra. Linux kernel memory barriers. <https://www.kernel.org/>

- [Hoa74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.
- [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.
- [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. URL: <http://www.b-eye-network.com/view/3365>, 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:

- [Jac88] Van Jacobson. Congestion avoidance and control. In *SIGCOMM '88*, pages 314–329, August 1988.
- [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. <http://princeton.kumarbhope.com/papers/PPoPP06/ppopp06.pdf>.
- [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. URL: <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 Kothothanassis, 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.
- [LHF05] Michael Lyons, Bill Hay, and Brad Frey. PowerPC storage model and AIX programming. <http://www.ibm.com/developerworks/systems/articles/powerpc.html>, November 2005.
- [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.
- [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/RCU/r-clock_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, <http://rdrop.com/users/paulmck/techreports/survey.2012.09.17a.pdf>, 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.
- [McK96] 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].
- [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 preemption overview. URL: <http://lwn.net/Articles/146861/>, August 2005.
- [McK06a] Paul E. McKenney. RCU Linux usage. Available: <http://www.rdrop.com/users/paulmck/RCU/linuxusage.html> [Viewed January 14, 2007], October 2006.
- [McK06b] 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/Articles/264090/> [Viewed January 10, 2008], January 2008.
- [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. Hunting heisenbugs. Available: <http://paulmck.livejournal.com/14639.html> [Viewed June 4, 2010], November 2009.
- [McK09b] Paul E. McKenney. RCU: The Bloatwatch edition. Available: <http://lwn.net/Articles/323929/> [Viewed March 20, 2009], March 2009.
- [McK09c] 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.
- [McK09d] Paul E. McKenney. Transactional memory everywhere? <http://paulmck.livejournal.com/tag/transactional%20memory%20everywhere>, September 2009.
- [McK10] Paul E. McKenney. Lockdep-RCU. <https://lwn.net/Articles/371986/>, February 2010.
- [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. 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.
- [McK12b] 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.
- [McK12c] 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.
- [McK14a] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It? (First Edition)*. kernel.org, Corvallis, OR, USA, 2014.
- [McK14b] 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.
- [MDJ13c] Paul E. McKenney, Mathieu Desnoyers, and Lai Jiangshan. User-space RCU. <https://lwn.net/Articles/573424/>, November 2013.
- [MER13] Paul E. McKenney, Dietmar Eggemann, and Robin Randhawa. Improving energy efficiency on asymmetric multiprocessing systems. <https://www.usenix.org/system/files/hotpar13-poster8-mckenney.pdf>, June 2013.
- [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.tu.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. URL: <http://www.kernel.org/pub/linux/kernel/projects/rt/>, 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.
- [Mor04] James Morris. Recent developments in SELinux kernel performance. Available: http://www.livejournal.com/users/james_morris/2153.html [Viewed December 10, 2004], December 2004.
- [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].
- [MS95] Maged M. Michael and Michael L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, December 1995.
- [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.
- [MSS12] Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the ARM and POWER relaxed memory models. <https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf>, October 2012.
- [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.
- [MW08] Paul E. McKenney and Jonathan Walpole. Introducing technology into the Linux kernel: a case study. *SIGOPS Oper. Syst. Rev.*, 42(5):4–17, 2008.
- [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. URL: <http://www.nytimes.com/2006/12/11/technology/11reuters.html?ei=5088&en=e5e9416415a9eeb2&ex=1323493200...>, December 2006.
- [Pig06] Nick Pigglin. [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.lel.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.
- [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/383362/> [Viewed: August 28, 2011], April 2010.
- [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 Rosenburg, 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.
- [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.
- [Tor03a] Linus Torvalds. Linux 2.5.69. Available: <http://marc.theaimsgroup.com/?l=linux-kernel&m=105209603501299&w=2> [Viewed June 23, 2004], May 2003.
- [Tor03b] Linus Torvalds. Linux 2.5.70. Available: <http://marc.theaimsgroup.com/?l=linux-kernel&m=105400162802746&w=2> [Viewed June 23, 2004], May 2003.
- [Tor03c] 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.
- [Val95] John D. Valois. Lock-free linked lists using compare-and-swap. In *Symposium on Principles of Distributed Computing*, pages 165–172, 1995.

- [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.
- [Wei63] J. Weizenbaum. Symmetric list processor. *Commun. ACM*, 6(9):524–536, September 1963.
- [Wei12] Frédéric Weisbecker. Interruption timer périodique. http://www.dailymotion.com/video/xtxtew_interruption-timer-periodique-frederic-weisbecker-kernel-recipes-12_tech, 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. URL: <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. URL: <http://www.yodaiken.com/papers/sync.pdf>, September 2004.

If I have seen further it is by standing on the shoulders of giants.

Isaac Newton, modernized

Appendix F

Credits

F.1 Authors

F.2 Reviewers

- Alan Stern (Section 14.2).
- Andy Whitcroft (Section 9.5.2, Section 9.5.4).
- Artem Bityutskiy (Section 14.2, Appendix B).
- Dave Keck (Appendix B).
- David S. Horner (Section 12.1.5).
- Gautham Shenoy (Section 9.5.2, Section 9.5.4).
- “jarkao2”, AKA LWN guest #41960 (Section 9.5.4).
- Jonathan Walpole (Section 9.5.4).
- Josh Triplett (Chapter 12).
- Michael Factor (Section 17.2).
- Mike Fulton (Section 9.5.2).
- Peter Zijlstra (Section 9.5.3).
- Richard Woodruff (Appendix B).
- Suparna Bhattacharya (Chapter 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 87 originally appeared in 4th USENIX Workshop on Hot Topics on Parallelism [McK12b].
3. Section 9.5.2 (“RCU Fundamentals”) on page 130 originally appeared in Linux Weekly News [MW07].
4. Section 9.5.3 (“RCU Usage”) on page 136 originally appeared in Linux Weekly News [McK08c].
5. Section 9.5.4 (“RCU Linux-Kernel API”) on page 147 originally appeared in Linux Weekly News [McK08b].
6. Chapter 12 (“Formal Verification”) on page 207 originally appeared in Linux Weekly News [McK07e, MR08, McK11c].

7. Section 12.3 (“Axiomatic Approaches”) on page 236 originally appeared in Linux Weekly News [MS14].
8. Section 14.2 (“Linux Kernel Memory Barriers”) on page 253 originally appeared in kernel.org [HMDZ06].
9. Section 14.2 (“Linux Kernel Memory Barriers”) on page 253 originally appeared in kernel.org [HMDZ06].
10. Appendix B.7 (“Memory-Barrier Instructions For Specific CPUs”) on page 351 originally appeared in Linux Journal [McK05a, McK05b].

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 43) by Melissa Broussard.
11. Figure 6.1 (p 65) by Kornilios Kourtis.
12. Figure 6.2 (p 66) by Melissa Broussard.
13. Figure 6.3 (p 66) by Kornilios Kourtis.
14. Figure 6.4 (p 69) by Kornilios Kourtis.
15. Figure 6.18 (p 79) by Melissa Broussard.
16. Figure 6.20 (p 80) by Melissa Broussard.
17. Figure 6.21 (p 80) by Melissa Broussard.
18. Figure 7.1 (p 96) by Melissa Broussard.
19. Figure 7.2 (p 96) by Melissa Broussard.
20. Figure 10.18 (p 174) by Melissa Broussard.
21. Figure 10.19 (p 175) by Melissa Broussard.
22. Figure 11.1 (p 189) by Melissa Broussard.
23. Figure 11.2 (p 189) by Melissa Broussard.
24. Figure 11.3 (p 195) by Melissa Broussard.
25. Figure 11.8 (p 205) by Melissa Broussard.
26. Figure 14.2 (p 255) by Melissa Broussard.
27. Figure 14.6 (p 257) by Akira Yokosawa.
28. Figure 14.8 (p 263) by David Howells.
29. Figure 14.9 (p 270) by David Howells.
30. Figure 14.10 (p 271) by David Howells.
31. Figure 14.11 (p 271) by David Howells.
32. Figure 14.12 (p 272) by David Howells.
33. Figure 14.13 (p 272) by David Howells.
34. Figure 14.14 (p 273) by David Howells.
35. Figure 14.15 (p 273) by David Howells.
36. Figure 14.16 (p 274) by David Howells.
37. Figure 14.17 (p 274) by David Howells.
38. Figure 14.18 (p 274) by David Howells.
39. Figure 14.19 (p 276) by David Howells.
40. Figure 14.20 (p 277) by David Howells.
41. Figure 15.1 (p 282) by Melissa Broussard.
42. Figure 15.2 (p 282) by Melissa Broussard.
43. Figure 15.3 (p 283) by Melissa Broussard.
44. Figure 15.10 (p 291) by Melissa Broussard.
45. Figure 15.11 (p 291) by Melissa Broussard.
46. Figure 15.14 (p 293) by Melissa Broussard.
47. Figure 15.19 (p 300) by Sarah McKenney.
48. Figure 15.20 (p 300) by Sarah McKenney.
49. Figure 16.2 (p 303) by Melissa Broussard.
50. Figure 17.1 (p 305) by Melissa Broussard.

51. Figure 17.2 (p 306) by Melissa Broussard.
52. Figure 17.3 (p 306) by Melissa Broussard.
53. Figure 17.4 (p 307) by Melissa Broussard.
54. Figure 17.8 (p 318) by Melissa Broussard.
55. Figure 17.9 (p 319) by Melissa Broussard.
56. Figure 17.10 (p 319) by Melissa Broussard.
57. Figure 17.11 (p 320) by Melissa Broussard.
58. Figure A.4 (p 336) by Melissa Broussard.
59. Figure B.12 (p 355) by Melissa Brossard.
60. Figure D.3 (p 394) 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, Ravi Arimilli, Cathy May, Derek Williams, H. Peter Anvin, Andy Glew, Leonid Yegoshin, Richard Grisenthwaite, and Will Deacon. 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.