

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

Edited by:

Paul E. McKenney
Facebook
paulmck@kernel.org

February 3, 2024
Commit: Edition.2-3461-gd05732cc (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, z Systems, 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.
- Intel, Itanium, Intel Core, and Intel Xeon are trademarks of Intel Corporation or its subsidiaries in the United States, other countries, or both.
- Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.
- SPARC is a registered trademark of SPARC International, Inc. Products bearing SPARC trademarks are based on an architecture developed by Sun Microsystems, Inc.
- 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 © 2005–2024 by Paul E. McKenney. Each individual contribution is copyright by its contributor at the time of contribution, as recorded in the git archive.

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

² <https://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?	4
2	Introduction	7
2.1	Historic Parallel Programming Difficulties	7
2.2	Parallel Programming Goals	8
2.2.1	Performance	9
2.2.2	Productivity	9
2.2.3	Generality	10
2.3	Alternatives to Parallel Programming	12
2.3.1	Multiple Instances of a Sequential Application	12
2.3.2	Use Existing Parallel Software	12
2.3.3	Performance Optimization	12
2.4	What Makes Parallel Programming Hard?	13
2.4.1	Work Partitioning	14
2.4.2	Parallel Access Control	14
2.4.3	Resource Partitioning and Replication	15
2.4.4	Interacting With Hardware	15
2.4.5	Composite Capabilities	15
2.4.6	How Do Languages and Environments Assist With These Tasks?	15
2.5	Discussion	16
3	Hardware and its Habits	17
3.1	Overview	17
3.1.1	Pipelined CPUs	17
3.1.2	Memory References	19
3.1.3	Atomic Operations	19
3.1.4	Memory Barriers	20
3.1.5	Cache Misses	20
3.1.6	I/O Operations	20
3.2	Overheads	21
3.2.1	Hardware System Architecture	21
3.2.2	Costs of Operations	22
3.2.3	Hardware Optimizations	24

3.3	Hardware Free Lunch?	25
3.3.1	3D Integration	25
3.3.2	Novel Materials and Processes	26
3.3.3	Light, Not Electrons	26
3.3.4	Special-Purpose Accelerators	26
3.3.5	Existing Parallel Software	27
3.4	Software Design Implications	27
4	Tools of the Trade	29
4.1	Scripting Languages	29
4.2	POSIX Multiprocessing	30
4.2.1	POSIX Process Creation and Destruction	30
4.2.2	POSIX Thread Creation and Destruction	31
4.2.3	POSIX Locking	32
4.2.4	POSIX Reader-Writer Locking	34
4.2.5	Atomic Operations (GCC Classic)	36
4.2.6	Atomic Operations (C11)	36
4.2.7	Atomic Operations (Modern GCC)	37
4.2.8	Per-Thread Variables	37
4.3	Alternatives to POSIX Operations	37
4.3.1	Organization and Initialization	37
4.3.2	Thread Creation, Destruction, and Control	37
4.3.3	Locking	39
4.3.4	Accessing Shared Variables	39
4.3.5	Atomic Operations	46
4.3.6	Per-CPU Variables	46
4.4	The Right Tool for the Job: How to Choose?	47
5	Counting	49
5.1	Why Isn't Concurrent Counting Trivial?	49
5.2	Statistical Counters	51
5.2.1	Design	51
5.2.2	Array-Based Implementation	51
5.2.3	Per-Thread-Variable-Based Implementation	52
5.2.4	Eventually Consistent Implementation	54
5.2.5	Discussion	55
5.3	Approximate Limit Counters	55
5.3.1	Design	55
5.3.2	Simple Limit Counter Implementation	56
5.3.3	Simple Limit Counter Discussion	60
5.3.4	Approximate Limit Counter Implementation	60
5.3.5	Approximate Limit Counter Discussion	61
5.4	Exact Limit Counters	61
5.4.1	Atomic Limit Counter Implementation	61
5.4.2	Atomic Limit Counter Discussion	64
5.4.3	Signal-Theft Limit Counter Design	64
5.4.4	Signal-Theft Limit Counter Implementation	65
5.4.5	Signal-Theft Limit Counter Discussion	68
5.4.6	Applying Exact Limit Counters	68
5.5	Parallel Counting Discussion	69

5.5.1	Parallel Counting Performance	69
5.5.2	Parallel Counting Specializations	69
5.5.3	Parallel Counting Lessons	70
6	Partitioning and Synchronization Design	73
6.1	Partitioning Exercises	73
6.1.1	Dining Philosophers Problem	74
6.1.2	Double-Ended Queue	74
6.1.3	Partitioning Example Discussion	80
6.2	Design Criteria	80
6.3	Synchronization Granularity	82
6.3.1	Sequential Program	82
6.3.2	Code Locking	83
6.3.3	Data Locking	84
6.3.4	Data Ownership	85
6.3.5	Locking Granularity and Performance	86
6.4	Parallel Fastpath	88
6.4.1	Reader/Writer Locking	88
6.4.2	Hierarchical Locking	89
6.4.3	Resource Allocator Caches	89
6.5	Beyond Partitioning	92
6.5.1	Work-Queue Parallel Maze Solver	93
6.5.2	Alternative Parallel Maze Solver	94
6.5.3	Performance Comparison I	95
6.5.4	Alternative Sequential Maze Solver	97
6.5.5	Performance Comparison II	97
6.5.6	Future Directions and Conclusions	98
6.6	Partitioning, Parallelism, and Optimization	98
7	Locking	99
7.1	Staying Alive	100
7.1.1	Deadlock	100
7.1.2	Livelock and Starvation	105
7.1.3	Unfairness	106
7.1.4	Inefficiency	106
7.2	Types of Locks	107
7.2.1	Exclusive Locks	107
7.2.2	Reader-Writer Locks	108
7.2.3	Beyond Reader-Writer Locks	108
7.2.4	Scoped Locking	109
7.3	Locking Implementation Issues	111
7.3.1	Sample Exclusive-Locking Implementation Based on Atomic Exchange	111
7.3.2	Other Exclusive-Locking Implementations	111
7.4	Lock-Based Existence Guarantees	113
7.5	Locking: Hero or Villain?	114
7.5.1	Locking For Applications: Hero!	115
7.5.2	Locking For Parallel Libraries: Just Another Tool	115
7.5.3	Locking For Parallelizing Sequential Libraries: Villain!	117
7.6	Summary	119

8 Data Ownership	121
8.1 Multiple Processes	121
8.2 Partial Data Ownership and pthreads	122
8.3 Function Shipping	122
8.4 Designated Thread	122
8.5 Privatization	123
8.6 Other Uses of Data Ownership	123
9 Deferred Processing	125
9.1 Running Example	125
9.2 Reference Counting	126
9.3 Hazard Pointers	128
9.4 Sequence Locks	132
9.5 Read-Copy Update (RCU)	135
9.5.1 Introduction to RCU	135
9.5.2 RCU Fundamentals	141
9.5.3 RCU Linux-Kernel API	147
9.5.4 RCU Usage	156
9.5.5 RCU Related Work	167
9.5.6 RCU Exercises	169
9.6 Which to Choose?	170
9.6.1 Which to Choose? (Overview)	170
9.6.2 Which to Choose? (Details)	170
9.6.3 Which to Choose? (Production Use)	173
9.7 What About Updates?	174
10 Data Structures	175
10.1 Motivating Application	175
10.2 Partitionable Data Structures	175
10.2.1 Hash-Table Design	176
10.2.2 Hash-Table Implementation	176
10.2.3 Hash-Table Performance	178
10.3 Read-Mostly Data Structures	179
10.3.1 RCU-Protected Hash Table Implementation	179
10.3.2 RCU-Protected Hash Table Performance	180
10.3.3 RCU-Protected Hash Table Discussion	183
10.4 Non-Partitionable Data Structures	183
10.4.1 Resizable Hash Table Design	184
10.4.2 Resizable Hash Table Implementation	185
10.4.3 Resizable Hash Table Discussion	189
10.4.4 Other Resizable Hash Tables	190
10.5 Other Data Structures	192
10.6 Micro-Optimization	192
10.6.1 Specialization	192
10.6.2 Bits and Bytes	193
10.6.3 Hardware Considerations	193
10.7 Summary	194

11 Validation	197
11.1 Introduction	197
11.1.1 Where Do Bugs Come From?	197
11.1.2 Required Mindset	198
11.1.3 When Should Validation Start?	200
11.1.4 The Open Source Way	201
11.2 Tracing	202
11.3 Assertions	202
11.4 Static Analysis	203
11.5 Code Review	203
11.5.1 Inspection	203
11.5.2 Walkthroughs	204
11.5.3 Self-Inspection	204
11.6 Probability and Heisenbugs	205
11.6.1 Statistics for Discrete Testing	206
11.6.2 Statistics Abuse for Discrete Testing	207
11.6.3 Statistics for Continuous Testing	207
11.6.4 Hunting Heisenbugs	208
11.7 Performance Estimation	212
11.7.1 Benchmarking	212
11.7.2 Profiling	212
11.7.3 Differential Profiling	213
11.7.4 Microbenchmarking	213
11.7.5 Isolation	214
11.7.6 Detecting Interference	214
11.8 Summary	216
12 Formal Verification	219
12.1 State-Space Search	219
12.1.1 Promela and Spin	219
12.1.2 How to Use Promela	221
12.1.3 Promela Example: Locking	225
12.1.4 Promela Example: QRCU	226
12.1.5 Promela Parable: dynticks and Preemptible RCU	231
12.1.6 Validating Preemptible RCU and dynticks	234
12.2 Special-Purpose State-Space Search	246
12.2.1 Anatomy of a Litmus Test	247
12.2.2 What Does This Litmus Test Mean?	248
12.2.3 Running a Litmus Test	248
12.2.4 PPCMEM Discussion	249
12.3 Axiomatic Approaches	250
12.3.1 Axiomatic Approaches and Locking	251
12.3.2 Axiomatic Approaches and RCU	251
12.4 SAT Solvers	253
12.5 Stateless Model Checkers	254
12.6 Summary	254
12.7 Choosing a Validation Plan	255

13 Putting It All Together	259
13.1 Counter Conundrums	259
13.1.1 Counting Updates	259
13.1.2 Counting Lookups	259
13.2 Refurbish Reference Counting	260
13.2.1 Implementation of Reference-Counting Categories	261
13.2.2 Counter Optimizations	264
13.3 Hazard-Pointer Helpers	264
13.3.1 Scalable Reference Count	264
13.4 Sequence-Locking Specials	264
13.4.1 Correlated Data Elements	264
13.4.2 Upgrade to Writer	265
13.5 RCU Rescues	265
13.5.1 RCU and Per-Thread-Variable-Based Statistical Counters	265
13.5.2 RCU and Counters for Removable I/O Devices	267
13.5.3 Array and Length	268
13.5.4 Correlated Fields	268
13.5.5 Update-Friendly Traversal	269
13.5.6 Scalable Reference Count Two	269
14 Advanced Synchronization	271
14.1 Avoiding Locks	271
14.2 Non-Blocking Synchronization	271
14.2.1 Simple NBS	272
14.2.2 Applicability of NBS Benefits	274
14.2.3 NBS Discussion	275
14.3 Parallel Real-Time Computing	276
14.3.1 What is Real-Time Computing?	276
14.3.2 Who Needs Real-Time?	280
14.3.3 Who Needs Parallel Real-Time?	281
14.3.4 Implementing Parallel Real-Time Systems	281
14.3.5 Implementing Parallel Real-Time Operating Systems	282
14.3.6 Implementing Parallel Real-Time Applications	292
14.3.7 Real Time vs. Real Fast: How to Choose?	295
15 Advanced Synchronization: Memory Ordering	297
15.1 Ordering: Why and How?	297
15.1.1 Why Hardware Misordering?	298
15.1.2 How to Force Ordering?	299
15.1.3 Basic Rules of Thumb	302
15.2 Tricks and Traps	303
15.2.1 Variables With Multiple Values	303
15.2.2 Memory-Reference Reordering	304
15.2.3 Address Dependencies	307
15.2.4 Data Dependencies	308
15.2.5 Control Dependencies	309
15.2.6 Cache Coherence	310
15.2.7 Multicopy Atomicity	310
15.3 Compile-Time Consternation	317
15.3.1 Memory-Reference Restrictions	318

15.3.2 Address- and Data-Dependency Difficulties	319
15.3.3 Control-Dependency Calamities	321
15.4 Higher-Level Primitives	324
15.4.1 Memory Allocation	324
15.4.2 RCU	324
15.5 Hardware Specifics	331
15.5.1 Alpha	333
15.5.2 Armv7-A/R	334
15.5.3 Armv8	335
15.5.4 Itanium	336
15.5.5 MIPS	336
15.5.6 POWER / PowerPC	337
15.5.7 SPARC TSO	338
15.5.8 x86	338
15.5.9 z Systems	339
15.6 Where is Memory Ordering Needed?	339
16 Ease of Use	341
16.1 What is Easy?	341
16.2 Rusty Scale for API Design	341
16.3 Shaving the Mandelbrot Set	342
17 Conflicting Visions of the Future	345
17.1 The Future of CPU Technology Ain't What it Used to Be	345
17.1.1 Uniprocessor Über Alles	345
17.1.2 Multithreaded Mania	346
17.1.3 More of the Same	347
17.1.4 Crash Dummies Slamming into the Memory Wall	347
17.1.5 Astounding Accelerators	349
17.2 Transactional Memory	349
17.2.1 Outside World	349
17.2.2 Process Modification	352
17.2.3 Synchronization	355
17.2.4 Discussion	358
17.3 Hardware Transactional Memory	360
17.3.1 HTM Benefits WRT Locking	361
17.3.2 HTM Weaknesses WRT Locking	362
17.3.3 HTM Weaknesses WRT Locking When Augmented	367
17.3.4 Where Does HTM Best Fit In?	367
17.3.5 Potential Game Changers	367
17.3.6 Conclusions	371
17.4 Formal Regression Testing?	372
17.4.1 Automatic Translation	372
17.4.2 Environment	373
17.4.3 Overhead	373
17.4.4 Locate Bugs	374
17.4.5 Minimal Scaffolding	374
17.4.6 Relevant Bugs	375
17.4.7 Formal Regression Scorecard	376
17.5 Functional Programming for Parallelism	376

17.6 Summary	378
18 Looking Forward and Back	379
A Important Questions	383
A.1 What Does “After” Mean?	383
A.2 What is the Difference Between “Concurrent” and “Parallel”?	385
A.3 What Time Is It?	386
A.4 How Much Ordering?	386
A.4.1 Where is the Defining Data?	387
A.4.2 Consistent Data Used Consistently?	387
A.4.3 Is the Problem Partitionable?	388
A.4.4 None of the Above?	388
B “Toy” RCU Implementations	389
B.1 Lock-Based RCU	389
B.2 Per-Thread Lock-Based RCU	390
B.3 Simple Counter-Based RCU	390
B.4 Starvation-Free Counter-Based RCU	391
B.5 Scalable Counter-Based RCU	393
B.6 Scalable Counter-Based RCU With Shared Grace Periods	394
B.7 RCU Based on Free-Running Counter	396
B.8 Nestable RCU Based on Free-Running Counter	397
B.9 RCU Based on Quiescent States	398
B.10 Summary of Toy RCU Implementations	400
C Why Memory Barriers?	401
C.1 Cache Structure	401
C.2 Cache-Coherence Protocols	403
C.2.1 MESI States	403
C.2.2 MESI Protocol Messages	403
C.2.3 MESI State Diagram	404
C.2.4 MESI Protocol Example	405
C.3 Stores Result in Unnecessary Stalls	405
C.3.1 Store Buffers	406
C.3.2 Store Forwarding	407
C.3.3 Store Buffers and Memory Barriers	407
C.4 Store Sequences Result in Unnecessary Stalls	409
C.4.1 Invalidate Queues	409
C.4.2 Invalidate Queues and Invalidate Acknowledge	410
C.4.3 Invalidate Queues and Memory Barriers	410
C.5 Read and Write Memory Barriers	412
C.6 Example Memory-Barrier Sequences	412
C.6.1 Ordering-Hostile Architecture	412
C.6.2 Example 1	413
C.6.3 Example 2	413
C.6.4 Example 3	413
C.7 Are Memory Barriers Forever?	414
C.8 Advice to Hardware Designers	414

D Style Guide	417
D.1 Paul's Conventions	417
D.2 NIST Style Guide	418
D.2.1 Unit Symbol	418
D.2.2 NIST Guide Yet To Be Followed	419
D.3 L ^A T _E X Conventions	419
D.3.1 Monospace Font	419
D.3.2 Cross-reference	423
D.3.3 Non Breakable Spaces	424
D.3.4 Hyphenation and Dashes	424
D.3.5 Punctuation	425
D.3.6 Floating Object Format	426
D.3.7 Improvement Candidates	426
E Answers to Quick Quizzes	431
E.1 How To Use This Book	431
E.2 Introduction	432
E.3 Hardware and its Habits	436
E.4 Tools of the Trade	439
E.5 Counting	445
E.6 Partitioning and Synchronization Design	459
E.7 Locking	464
E.8 Data Ownership	471
E.9 Deferred Processing	472
E.10 Data Structures	487
E.11 Validation	491
E.12 Formal Verification	498
E.13 Putting It All Together	505
E.14 Advanced Synchronization	507
E.15 Advanced Synchronization: Memory Ordering	509
E.16 Ease of Use	519
E.17 Conflicting Visions of the Future	520
E.18 Important Questions	525
E.19 "Toy" RCU Implementations	526
E.20 Why Memory Barriers?	531
Glossary	535
Bibliography	543
Credits	585
L ^A T _E X Advisor	585
Reviewers	585
Machine Owners	585
Original Publications	585
Figure Credits	586
Other Support	587

Chapter 1

How To Use This Book

이 책의 목적은 여러분이 정신을 잃는 위험 없이 공유 메모리 병렬 시스템을 프로그램하는 것을 돋는 것입니다.¹ 하지만, 이 책의 정보는 완벽한 성당이라기보다는 만들고자 하는 것의 토대 정도로 생각하셔야 합니다. 여러분의 미션은, 만약 여러분이 받아들인다면, 신나는 병렬 프로그래밍 분야에 발전—결국 이 책을 필요 없게 만들 발전—을 더 만드는 것을 돋는 겁니다.

21세기의 병렬 프로그래밍은 더이상 과학, 연구, 거대한 도전적 프로젝트에만 초점을 맞추지 않습니다. 그리고 이건 병렬 프로그래밍이 엔지니어링 분야가 되어 가고 있음을 의미하니 좋은 현상입니다. 따라서, 이 책은 엔지니어링 분야에 적합한 정도로 병렬 프로그래밍 작업들을 다루고 그것들을 어떻게 접근해야 하는지 설명합니다. 놀랍도록 다양한 경우에, 이 작업들은 자동화 될 수 있습니다.

이 책은 성공적 병렬 프로그래밍 프로젝트에 깔려 있는 엔지니어링 규칙을 보이는 것이 새로운 병렬 프로그래밍 해커 세대를 느리고 고통스럽게 오래된 바퀴를 새로 발명해야 하는 것으로부터 자유롭게 하고, 그대신 그들의 에너지와 창조성을 새로운 영역에 집중하는 것을 가능하게 할 것이라는 희망 하에 쓰였습니다. 하지만, 여러분이 이 책에서 얻는 것은 여러분이 그 안에 무엇을 쓸는가에 따라 정해집니다. 단순히 이 책을 읽는 것도 도움될 수 있고, Quick Quizz 들을 푸는 건 더 도움될 겁니다. 하지만, 최선의 결과는 이 책에서 가르치는 기술들을 실제 삶의 문제들에 적용해 보는 데서 나올 겁니다. 항상 그렇듯, 실전이 완벽을 만듭니다.

하지만 여러분이 어떻게 접근하는가와 상관없이, 병렬 프로그래밍이 여러분에게 많은 즐거움, 신남, 그리고 도전을 우리에게 그랬듯 여러분에게도 가져다 주길 진심으로 바랍니다!

If you would only recognize that life is hard, things would be so much easier for you.

Louis D. Brandeis

1.1 Roadmap

Cat: Where are you going?

Alice: Which way should I go?

Cat: That depends on where you are going.

Alice: I don't know.

Cat: Then it doesn't matter which way you go.

Lewis Carroll, Alice in Wonderland

이 책은 아주 적은 영역에만 적용 가능한 최적 알고리즘들의 모음이라기보다는 널리 적용될 수 있고 매우 많이 사용되는 디자인 기술의 안내서입니다. 여러분은 현재 Chapter 1 을 읽고 있는데, 알고 있겠죠. Chapter 2 은 병렬 프로그래밍에 대한 높은 수준에서의 개요를 제공합니다.

Chapter 3 는 공유 메모리 병렬 하드웨어를 소개합니다. 어쨌건, 아래에 깔려있는 하드웨어를 이해하지 않고서는 좋은 병렬 코드를 작성하기가 어렵습니다. 하드웨어는 지속적으로 발전하므로, 이 챕터는 항상 시대에 뒤떨어져 있을 겁니다. 우린 최대한 시대에 맞춰지도록 최선을 다하겠습니다. Chapter 4 는 이어서 일반적인 공유 메모리 병렬 프로그래밍 기초도구에 대한 간략한 개요를 제공합니다.

Chapter 5 은 상상 가능한 가장 간단한 문제 중 하나인 카운팅의 병렬화를 들여다봅니다. 거의 모든 사람이 카운팅을 알고 있기 때문에, 이 챕터는 더 구체적인 컴퓨터 과학 문제들에 방해받지 않고 많은 중요한 병렬 프로그래밍 이슈를 다룰 수 있습니다. 제 생각에 이 챕터는 병렬 프로그래밍 수업에서 많이 사용되었습니다.

Chapter 6 는 Chapter 5 에서 정의된 이슈들을 다루는 다양한 설계 수준에서의 방법들을 소개합니다. 가능할 때에는 병렬성을 설계 수준에서 다루는 게 중요함이 드러났습니다: Dijkstra [Dij68] 의 말을 바꿔 말하자면, “고쳐진 병렬성은 심하게 덜 최적화된 것으로 여겨진다” [McK12c].

¹ 또는, 더 정확하게는, 병렬 프로그래밍이 아닌 프로그래밍이 일으키는 것보다 너무 크지는 않은 정도의 위험으로.

다음의 세 챕터는 동기화를 위한 세 가지 중요한 접근법을 살펴봅니다. Chapter 7은 여전히 제품 품질 병렬 프로그래밍에 널리 사용될 뿐만 아니라 병렬 프로그래밍의 죄악의 악당으로도 널리 여겨지는 락킹을 다룹니다. Chapter 8은 종종 과소평가되지만 놀라울 정도로 널리 사용되며 강력한 데이터 소유권의 개념을 간단히 알아봅니다. 마지막으로, Chapter 9은 다양한 미뤄서 처리하기 (deferred-processing) 메카니즘을 소개하는데, 레퍼런스 카운팅, 해저드 포인터, 시퀀스 락킹, 그리고 RCU가 포함됩니다.

Chapter 10은 앞의 챕터들에서 배운 것들을 해시 테이블에 적용해 봅니다. 해시 테이블은 (보통) 훌륭한 성능과 확장성으로 이어지는 훌륭한 분할 가능성이 덕분에 매우 널리 사용됩니다.

많은 사람들이 그들의 슬픔으로부터 배웠듯이, 검증 없는 병렬 프로그래밍은 비참한 실패로 향하는 확실한 길입니다. 여러분의 프로그램의 안전성을 테스트하는 건 물론 불가능하므로, Chapter 12은 정형적 검증을 위한 몇 가지 실용적 접근법에 대한 간단한 개요를 제공합니다.

Chapter 13은 적당한 크기의 병렬 프로그래밍 문제 몇 가지를 포함합니다. 이 문제들의 난이도는 다양하지만, 앞의 챕터들의 것들을 터득한 사람들에게 적당할 겁니다.

Chapter 14는 고급 동기화 방법들을 알아보는데, non-blocking 동기화와 병렬 리얼타임 컴퓨팅을 포함하며, Chapter 15는 메모리 순서 규칙의 고급 주제를 다룹니다. Chapter 16는 사용하기 쉬운 몇 가지 조언들을 봅니다. Chapter 17에서는 몇 가지 가능할 법한 미래의 방향을 보는데, 공유 메모리 병렬 시스템 설계, 소프트웨어와 하드웨어 기반의 transactional memory, 그리고 병렬화를 위한 함수형 프로그래밍을 포함합니다. 마지막으로, Chapter 18은 이 책의 것들과 그것들의 원조를 리뷰합니다.

이 챕터의 뒤로 다수의 부록이 있습니다. 이 중 가장 대중적인 것은 Appendix C 일 것으로, 메모리 순서 규칙에 대해 더 다룹니다. Appendix E는 악명 높은 Quick Quizz들의 답을 포함하는데, 다음 섹션에서 이에 대해 다룹니다.

1.2 Quick Quizzes

Undertake something difficult, otherwise you will never grow.

Abbreviated from Ronald E. Osburn

“Quick quiz”들이 이 책 여기 저기에 나오며, 그 답은 Appendix E의 page 431에 있습니다. 그 중 일부는

그 quick quiz 가 나온 곳의 것들에 기반하고 있지만, 일부는 그 섹션을 넘어서 생각할 것을 필요로 하며, 일부 경우는 현재 지식을 넘어서야 하기도 합니다. 대부분의 시도처럼, 여러분이 이 책에서 얻는 것은 여러분이 무엇을 투자하는가에 달려있습니다. 따라서, 답을 보기 전에 퀴즈를 풀기 위해 진실된 노력을 한 독자는 그 노력이 병렬 프로그래밍에 대한 나아진 이해와 함께 보상됨을 발견할 겁니다.

Quick Quiz 1.1:

Quick Quiz의 답은 어디 있나요?



Quick Quiz 1.2:

일부 Quick Quiz의 질문은 저자의 시선보다는 독자의 시선에서 쓰여진 것 같습니다. 의도된 바인가요?



Quick Quiz 1.3:

이 Quick quiz는 제게 맞지 않는 것 같아요. 어떡하죠?



요약해서, 이 주제에 대해 깊은 이해가 필요하다면, Quick Quiz에 답변하는데 시간을 좀 투자해야 합니다. 절 틀렸다 하지 마세요, 이것들을 수동적으로 읽는 것은 상당히 가치 있을 수 있지만, 완전한 문제 해결 능력을 얻기 위해선 여러분이 문제 풀이를 연습할 필요가 있습니다.

저는 이를 제 늦깎이 박사과정 코스워에서 어렵게 배웠습니다. 저는 익숙한 주제를 연구하고 있었고, 그 챕터의 연습문제 중 일부만을 당장 풀 수 있음에 놀랐습니다.² 스스로를 그 질문들에 대답하도록 밀어붙이는 것이 그것들에 대한 제 기억을 상당히 증진시켰습니다. 그러니 이 Quick Quiz를 통해 여러분에게 제가 하지도 않은 걸 하라고 하는 게 아닙니다.

마지막으로, 가장 흔한 학습 장애는 여러분이 그걸 이미 이해하고 있다고 생각하는 겁니다. Quick Quiz는 그에 대한 매우 효과적인 치료가 될 수 있습니다.

1.3 Alternatives to This Book

Between two evils I always pick the one I never tried before.

Mae West

Knuth가 어렵게 배웠듯이, 여러분의 책이 유한하길 바란다면 그 책은 특정 주제에 집중되어 있어야 합니다. 이 책은 공유 메모리 병렬 프로그래밍에 집중하고 있으며, 운영체제 커널, 병렬 데이터 관리 시스템, 저수준 라이브러리 등과 같은 소프트웨어 스택의 바닥 근처에 있는

² 그래서 제 교수님이 제가 그 수업을 포기하는 것을 허락하지 않았다고 생각합니다!

소프트웨어를 강조하고 있습니다. 이 책에서 사용하는 프로그래밍 언어는 C입니다.

병렬성의 다른 측면에 관심이 있다면, 다른 책도 도움이 될 겁니다. 다행히, 가능한 대안이 여럿 있습니다:

1. 병렬 프로그래밍에 대한 더 학술적이고 정밀한 취급을 원한다면, Herlihy 와 Shavit 의 책 [HS08, HSLS20] 를 좋아하실 겁니다. 이 책은 하드웨어로부터의 추상화의 높은 수준에서의 흥미로운 저수준 기본기능 조합과 함께 시작하고 락킹과 리스트, 큐, 해시 테이블, 카운터 등의 간단한 자료구조를 다루며 트랜잭션 메모리로 끝을 맺습니다. Michael Scott 의 책 [Sco13] 은 비슷한 주제를 더 소프트웨어 엔지니어링에 중점을 둘러 다루며, 제가 알기로는 RCU 만을 위한 섹션을 가진 최초의 정식적으로 출간된 학계 서적입니다.
2. 프로그래밍 언어 실용성 시점에서 병렬 프로그래밍을 다루고 싶다면 Scott 의 프로그래밍 언어 실용성에 대한 책 [Sco06, Sco15] 의 동시성 챕터에 관심이 가실 수도 있습니다.
3. C++ 위주로 병렬 프로그래밍을 객체 지향 패턴적으로 다루는 데 관심 있다면, Schmidt 의 POSA 시리즈 [SSRB00, BHS07] 의 볼륨 2 와 4 를 시도해 보실 수도 있습니다. 특히 볼륨 4 는 병렬 프로그래밍을 참고 관리 어플리케이션에 적용하는 흥미로운 챕터들을 포함하고 있습니다. 이 예제의 현실성은 “Partitioning the Big Ball of Mud” 라는 제목의 섹션에서 평가되는데, 병렬성에 내재된 문제들은 종종 실제 어플리케이션에서는 눈에 띄지 않는 자리에 있게 된다는 것입니다.
4. 리눅스 커널 디바이스 드라이버를 작업하고자 한다면, Corbet, Rubini, 그리고 Kroah-Hartman 의 “Linux Device Drivers” [CRKH05] 는 필수적일 것이며, Linux Weekly News 웹 사이트 (<https://lwn.net/>) 또한 그럴 것입니다. 리눅스 커널 내부에 대한 일반적 주제에 대해서는 많은 수의 책과 자료들이 있습니다.
5. 여러분의 주요 관심이 과학 기술적 컴퓨팅이라면, 그리고 패턴적 접근을 선호한다면, Mattson 등의 책 [MSM05] 을 시도해 볼 수 있겠습니다. 이 책은 Java, C/C++, OpenMP, 그리고 MPI 를 다룹니다. 이 책의 패턴들은 먼저 설계에, 이어서 구현에 훌륭히 집중되어 있습니다.
6. 여러분의 주요 관심이 과학 기술적 컴퓨팅이고 GPU, CUDA, 그리고 MPI 에 흥미 있다면, Norm Matloff 의 “Programming on Parallel Machines” [Mat17] 을 시도해 볼 수 있겠습니다. 물론,

GPU 제조사들은 많은 추가적 정보를 가지고 있습니다 [AMD20, Zel11, NVi17a, NVi17b].

7. 여러분이 POSIX 쓰레드에 관심 있다면, David R. Butenhof 의 책 [But97] 을 읽어보실 수 있겠습니다. 또한, W. Richard Stevens 의 책 [Ste92, Ste13] 은 UNIX 와 POSIX 를 다루며, Stewart Weiss 의 강의 노트 [Wei13] 는 좋은 예제들과 함께 깊고 접근 가능한 소개를 제공합니다.
8. C++11 에 관심 있으시다면, Anthony Williams 의 “C++ Concurrency in Action: Practical Multithreading” [Wil12, Wil19] 를 좋아하실 수도 있습니다.
9. C++ 에 관심 있지만 Windows 환경을 원한다면, Dr. Dobbs Journal [Sut08] 의 Herb Sutter’s “Effective Concurrency” 시리즈를 읽어보실 수도 있겠습니다. 이 시리즈는 병렬성에 대한 상식적 접근법을 합리적으로 제공합니다.
10. Intel Threading Building Blocks 를 시도해 보고 싶다면, James Reinders 의 책 [Rei07] 이 여러분이 찾는 것일 겁니다.
11. 다양한 종류의 멀티 프로세서 하드웨어 캐시 구조가 어떻게 커널 내부 구현에 영향을 끼치는지 흥미 있다면 Curt Schimmel 의 이 주제에 대한 고전적 접근 [Sch94] 을 읽어 보셔야 합니다.
12. 하드웨어 관점을 보고자 한다면, Hennessy 와 Patterson 의 고전적 교재 [HP17, HP11] 가 읽어볼 만 할 겁니다. 메모리 순서 규칙에 대한 학술 교재를 찾고 있다면 Daniel Sorin 등이 책 [SHW11, NSHW20] 을 강하게 추천합니다. 리눅스 커널 관점에서의 메모리 순서 규칙에 대한 튜토리얼을 위해선 Paolo Bonzini 의 LWN series 가 시작하기에 좋습니다 [Bon21a, Bon21e, Bon21c, Bon21b, Bon21d].
13. 마지막으로, Java 를 사용하는 분들에겐 Doug Lea 의 교재 [Lea97, GPB⁰⁷] 가 도움될 수 있습니다. 하지만, 저수준 소프트웨어, 특히 C 로 쓰여진 소프트웨어를 위한 병렬 설계의 원칙에 관심이 있다면, 계속 읽으세요!

1.4 Sample Source Code

Use the source, Luke!

Unknown Star Wars fan

이 책은 소스 코드의 공정 공유에 대해 이야기 하며, 많은 경우 이 소스 코드는 이 책의 git tree 의 CodeSamples 디

Listing 1.1: Creating an Up-To-Date PDF

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/...
→ paulmck/perfbook.git
cd perfbook
# You may need to install a font. See item 1 in FAQ.txt.
make                # -jN for parallel build
evince perfbook.pdf &  # Two-column version
make perfbook-1c.pdf
evince perfbook-1c.pdf & # One-column version for e-readers
make help            # Display other build options
```

렉토리에 있습니다. 예를 들어, UNIX 시스템에서라면, 아래와 같은 커맨드를 사용할 수 있습니다:

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

이 커맨드는 Appendix B에서 이야기 되는 `rcu_rcplS.c` 파일을 찾아낼 겁니다. 다른 종류의 시스템에는 파일네임으로 파일을 찾는 잘 알려진 다른 방법들이 있을 겁니다.

1.5 Whose Book Is This?

If you become a teacher, by your pupils you'll be taught.

Oscar Hammerstein II

표지에서 이야기 하듯, 이 책의 편집자는 Paul E. McKenney입니다. 하지만, 이 편집자는 `perfbook@vger.kernel.org` 이메일 리스트를 통한 기여를 허용하고 있습니다. 이 기여들은 상당히 다양한 어떤 종류든 될 수 있는데, 텍스트 이메일이나 이 책의 `LATEX` 소스로의 패치, 심지어 `git pull` 리퀘스트 같은 대중적인 방법도 포함됩니다. 여러분에게 가장 잘 맞는 방법을 무엇이든 사용하세요.

패치 또는 `git pull` 리퀘스트를 만들려면 이 책의 `LATEX` 소스 코드가 필요할 텐데, `git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git`에 있습니다. 여러분은 또한 `git`과 `LATEX`이 필요할텐데, 이것들은 대부분의 주류 리눅스 배포판에서 사용 가능합니다. 다른 패키지들도 필요할 수 있는데, 여러분이 사용하는 배포판에 따라 달라집니다. 일부 유명한 배포판을 위한 패키지 목록이 이 책의 `LATEX` 소스의 `FAQ-BUILD.txt` 파일에 있습니다.

이 책의 현재 `LATEX` 소스 트리를 생성하고 보려면 Listing 1.1에 보여진 리눅스 커맨드를 사용하세요. 일부 환경에서는 `perfbook.pdf`를 보여주는 `evince` 커맨드가 예를 들어 `acroread`와 같은 다른 결로 변경되어야

Listing 1.2: Generating an Updated PDF

```
git remote update
git checkout origin/master
make                # -jN for parallel build
evince perfbook.pdf &  # Two-column version
make perfbook-1c.pdf
evince perfbook-1c.pdf & # One-column version for e-readers
```

할수도 있습니다. `git clone` 커맨드는 PDF를 생성하는 처음에만 필요하며, 그 후에는 모든 업데이트를 가져오고 업데이트된 PDF를 생성하기 위해 Listing 1.2의 커맨드를 사용하시면 됩니다. Listing 1.2의 커맨드는 반드시 Listing 1.1의 커맨드로 생성된 `perfbook` 디렉토리에서 실행되어야 합니다.

이 책의 PDF 파일들은 때때로 `https://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html`과 `http://www.rdrop.com/users/paulmck/perfbook/`에 업로드 됩니다.

패치를 기여하고 `git pull` 리퀘스트를 보내는 실제 과정은 리눅스 커널의 그것과 비슷한데, `https://www.kernel.org/doc/html/latest/process/submitting-patches.html`에 문서화되어 있습니다. 중요한 한가지 요구사항은 각 패치 (`git pull` 리퀘스트의 경우라면 각 커밋)는 아래 포맷의 유효한 `Signed-off-by:` 행을 가져야 한다는 것입니다:

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

`Signed-off-by:` 행을 갖는 예제 패치를 `https://lkml.org/lkml/2007/1/15/219`에서 확인하시기 바랍니다.

`Signed-off-by:` 행은 여러분이 아래와 같이 증명한다는, 매우 구체적인 의미를 가짐을 알아두는 게 중요합니다:

- (a) 이 기여는 온전히 또는 부분적으로 나에 의해 만들어졌으며 나는 이 기여를 이 파일에 표시된 오픈소스 라이센스 아래 제출할 권리가 있습니다; 또는
- (b) 이 기여는 제가 알기로는 적절한 오픈소스 라이센스 아래의 기존 작업으로 만들어졌으며 나는 그 라이센스 아래 그 작업물을 온전히든 부분적으로든 내가 만든 수정사항과 함께 해당 파일에 적시된 대로 같은 오픈소스 라이센스 아래 제출할 권리가 있습니다; 또는
- (c) 이 기여는 (a), (b), 또는 (c)를 증명하는 누군가에 의해 나에게 전달되었으며 나는 이를 수정하지 않았습니다.

- (d) 나는 이 프로젝트와 이 기여가 공공의 것이며 이 기여의 기록(내가 그와 함께 제출하는 모든 개인정보와 나의 sign-off 를 포함하여) 이 무한정 관리되고 이 프로젝트 또는 관련된 오픈소스 라이센스에 일관성을 유지한 채 재배포 될 수 있음을 이해하고 동의합니다.

이는 리눅스 커널에서 사용되는 Developer's Certificate of Origin (DCO) 1.1 과 상당히 비슷합니다. 여러분은 실명을 사용해야 합니다: 전 불행히도 가명 또는 익명 기여를 받을 수 없습니다.

이 책의 언어는 미국 영어입니다만, 이 책의 오픈소스라는 본질이 번역을 가능하게 하며, 저 개인적으로 이를 장려합니다. 이 책의 오픈소스 라이센스는 추가적으로 여러분이 원한다면 여러분의 번역을 판매하는 것을 허용합니다. 해당 번역본의 복사본을 (가능하다면 하드카피를) 제게 보내줄 것을 부탁드립니다만, 이는 전문적 예의에 따른 부탁이며, 여러분이 Creative Commons 와 GPL 라이센스 아래 이미 가진 권리에 대한 어떤 선요구 사항은 아닙니다. 현재 진행중인 번역 작업들의 목록을 보기 위해선 소스 트리의 FAQ.txt 파일을 확인하시기 바랍니다. 저는 번역 작업이 최소 한 챕터가 완전히 번역되었다면 “진행중”이라고 판단합니다.

“미국 영어” 규정에는 많은 스타일이 있습니다. 이 특정 책을 위한 스타일은 Appendix D 에 문서화 되어 있습니다.

이 섹션의 시작에서 이야기 되었듯, 전 이 책의 편집자입니다. 하지만, 여러분이 기여를 하기로 선택한다면, 이 책은 여러분의 것이기도 하게 됩니다. 그런 정신으로, 우리의 소개 부분인 Chapter 2 을 드립니다.

Chapter 2

Introduction

병렬 프로그래밍은 해커가 덤빌 수 있는 가장 어려운 영역이라는 평판을 얻었습니다. 논문과 교재들은 데드락, 라이브락, 레이스 컨디션, 비결정성, 암달의 법칙에 의한 확장성 제한, 그리고 지나친 리얼타임 반응시간의 위험을 경고합니다. 그리고 이런 위험들은 진짜입니다; 우리 저자들은 그로 인한 감정적 흥터, 백발, 탈모를 여러해 겪었습니다.

하지만, 처음 소개될 때엔 사용하기 어려웠던 기술들이 언제나 시간의 흐름에 따라 쉬워집니다. 예를 들어, 한때는 희귀했던 자동차 운전이 지금은 많은 나라에서 흔합니다. 이 극적인 변화는 두가지 기본적 이유에서 기인합니다: (1) 자동차가 저렴해지고 쉽게 구매할 수 있게 되어서, 더 많은 사람들이 운전을 배울 기회가 늘었고, (2) 자동 변속, 자동 초크, 자동 시동, 상당히 개선된 안정성, 그리고 다른 기술적 개선사항의 적용 등으로 자동차를 운영하기가 쉬워졌습니다.

컴퓨터를 비롯한 많은 다른 기술들에도 이게 동일하게 적용됩니다. 프로그래밍을 위해 편 칭머신을 사용할 필요가 더이상 없습니다. 스프레드시트는 프로그래머가 아닌 대부분의 사람들도 수십년 전이라면 전문가 팀이 필요했을 것을 컴퓨터에서 얻을 수 있게 합니다. 가장 강력한 예는 웹 서핑과 컨텐츠 작성일 것으로, 이것들은 2000년대 초반 이후로 훈련되지도 교육받지도 않은 사람들이 다양한, 지금은 흔한 소셜 네트워킹 도구들을 이용해서 쉽게 할 수 있게 되었습니다. 1968년만 해도, 그런 컨텐츠 작성은 당시에는 “백악관 정원에 UFO가 착륙하는 듯”[Gri00] 하다고 묘사되는 전위적 연구 프로젝트였습니다[Eng68].

따라서, 여러분이 병렬 프로그래밍이 현재 많은 사람들에게 인식되듯 어려운 영역으로 남을 거라고 주장하고 싶다면, 많은 노력이 있던 영역들에서 수세기 동안 있었던 반례를 기억하고서 증명을 해야 하는 건 여러분의 몫입니다.

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

Unknown

2.1 Historic Parallel Programming Difficulties

Not the power to remember, but its very opposite, the power to forget, is a necessary condition for our existence.

Sholem Asch

그 제목에서 보여지듯이, 이 책은 다른 접근법을 취합니다. 병렬 프로그래밍의 어려움에 대해 불평을 하기보다는, 병렬 프로그래밍이 어려운 이유를 알아보고, 독자들이 이 어려움들을 극복하는 것을 돕습니다. 이어서 보게 되겠지만, 이런 어려움들은 역사적으로 다음의 것들을 포함하는 몇가지 카테고리로 나뉘었습니다:

1. 역사적으로 높은 비용과 상대적으로 희귀한 병렬 시스템의 존재.
2. 일반적인 연구자와 견습자의 병렬 시스템에 대한 경험의 부족.
3. 공개된 병렬 코드의 부재.
4. 병렬 프로그래밍에 대한 널리 알려진 엔지니어링 규칙의 부재.
5. 심지어 강하게 결합된 공유 메모리 컴퓨터에 조차 존재하는 작업 대비 커뮤니케이션의 높은 오버헤드.

이런 역사적 어려움 중 다수는 극복되어 가는 중입니다. 첫째, 지난 수십년간, 병렬 시스템의 가격은 Moore의 법칙 덕분에 집 여러채의 가격에서 간단한 식사 값 정도가 되었습니다. 멀티코어 CPU의 장점을 이야기하는 논문이 1996년부터 [ONH⁺96] 출판되었습니다. IBM은 고성능 POWER 제품군에 2000년에는 동시적 멀티쓰레딩 (simultaneous multi-threading) 을, 2001년에는

멀티코어를 도입했습니다. Intel은 2000년 11월 하이퍼 캐리드를 Pentium 제품군에 도입했으며, AMD와 Intel은 2005년에 듀얼코어 CPU를 소개했습니다. Sun은 2005년 말 멀티코어/멀티캐리드 기능이 도입된 Niagara를 이어 발표했습니다. 실제로, 2008년에 이르러, 싱글 CPU 데스크탑을 찾기가 어려워졌으며, 싱글코어 CPU는 넷북과 임베디드 기기에만 주로 사용되게 되었습니다. 2012년에는 스마트폰조차 멀티 CPU를 사용하기 시작했습니다. 2020년, 안전성이 중요한 소프트웨어 표준들이 동시성을 다루기 시작했습니다.

둘째로, 비용이 낮고 당장 사용 가능한 멀티코어 시스템의 발전은 한때는 희귀했던 병렬 프로그래밍 경험이 지금은 거의 모든 연구자와 견습자에게 가능해졌음을 의미합니다. 사실, 병렬 시스템은 오랫동안 학생들과 취미가들에게 부담이었습니다. 따라서 우린 병렬 시스템을 둘러싼 발명과 혁신의 상당한 수준 증가를 기대할 수 있고, 그렇게 늘어난 친숙도는 한때는 금지된거나 마찬가지로 금전적 부담이 커던 병렬 프로그래밍 분야가 훨씬 친숙해지고 일반적이게 만들 겁니다.

셋째로, 20세기에, 고수준 병렬 소프트웨어를 사용하는 거대 시스템은 거의 항상 독점적 보안을 통해 폐쇄적으로 지켜지고 있었습니다. 행복하게도 대조적으로, 21세기는 많은 오픈소스(따라서 공공적으로 사용이 가능한) 병렬 소프트웨어 프로젝트를 보아왔는데, 리눅스 커널 [Tor03], 데이터베이스 시스템들 [Pos08, MS08], 그리고 메세지 패싱 시스템들 [The08, Uni08a] 등이 포함됩니다. 이 책은 리눅스 커널을 기초로 할겁니다만, 사용자 수준 어플리케이션에 사용 가능한 많은 것들을 제공할 겁니다.

넷째로, 1980년대와 1990년대의 거대 스케일 병렬 프로그래밍 프로젝트들은 거의 모두 독점 프로젝트였지만, 이 프로젝트들은 제품 품질의 병렬 코드를 개발하는데 필요한 엔지니어링 규칙을 이해하는 개발자들로 핵심그룹이 구성된 커뮤니티들의 씨앗을 뿐었습니다. 이 책의 주요 목적은 이 엔지니어링 규칙을 제공하는 겁니다.

불행히도, 다섯번째 어려움, 처리 대비 높은 비용의 커뮤니케이션은 여전히 남아있습니다. 이 어려움은 2000년대에 들어 증가된 주목을 받았습니다. 하지만, Stephen Hawking에 따르면, 빛의 제한된 속도와 물질의 원자적 성질이 이 영역에서의 발전을 제한하고 있습니다 [Gar07, Moo03]. 다행히도, 이 어려움은 1980년대 후반부터 드러나기 시작했고, 따라서 앞서 언급한 엔지니어링 규칙은 이를 처리하기에 실용적이고 효과적인 전략으로 발전했습니다. 또한, 하드웨어 설계자들이 이 문제를 더 주목하고 있으므로, 어쩌면 미래의 하드웨어는 Section 3.3에서 이야기하듯 병렬 소프트웨어에 더 친화적이 될 수도 있습니다.

Quick Quiz 2.1: 이봐요!!! 병렬 프로그래밍은 수십 년동안 엄청나게 어렵다고 알려져왔어요. 당신은 그게 그렇게 어렵지 않다고 하는 것 같군요. 뭔가 게임이라도 하는 건가요?

■ 하지만, 병렬 프로그래밍이 일반적으로 알려진 것만큼 어렵진 않을 수 있다고 쳐도, 많은 경우 순차적 프로그래밍보다는 어렵습니다.

Quick Quiz 2.2: 병렬 프로그래밍이 순차적 프로그래밍만큼 쉬워지는게 가능은 할까요?

■ 따라서 병렬 프로그래밍의 대체안을 생각해 보는게 말이 됩니다. 하지만, 병렬 프로그래밍의 목표를 이해하지 못한 채로 병렬 프로그래밍의 합리적 대체안을 생각하는건 불가능합니다. 이 주제는 다음 섹션에서 다룹니다.

2.2 Parallel Programming Goals

If you don't know where you are going, you will end up somewhere else.

Yogi Berra

병렬 프로그래밍의(순차적 프로그래밍에 비해 더 나아지고자 하는) 세가지 주요 목표는 다음과 같습니다:

1. 성능.
2. 생산성.
3. 범용성.

불행히도, 현재 기술로는, 어떤 병렬 프로그램도 이 세가지 목표 중 두개까지만 달성이 가능합니다. 따라서 이 세개의 목표는 병렬 프로그래밍의 철의 삼각지대를 이루는데, 너무 낙관적인 희망은 모두 슬픔으로 끝나고 마는 삼각지대입니다.¹

Quick Quiz 2.3: 오, 정말로요??? 정확성, 유지가능성, 견고성, 등등은 어쩌고요?

■ **Quick Quiz 2.4:** 그리고 정확성, 유지가능성, 견고성이 그 리스트에 들어가지 못한다면, 생산성과 범용성은 왜 들어가는거죠?

■ **Quick Quiz 2.5:** 병렬 프로그램은 순차적 프로그램보다 정확성을 입증하기가 훨씬 어렵다는 점을 고려하면, 정확성도 정말 이 리스트에 들어가야 하지 않습니까?

¹ 철의 삼각지대라는 이름을 지어준 데 대해 Michael Wong에게 감사를 드립니다.

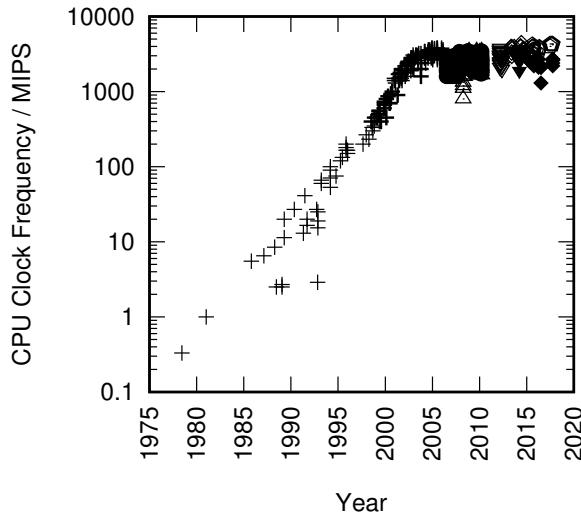


Figure 2.1: MIPS/Clock-Frequency Trend for Intel CPUs

Quick Quiz 2.6: 그냥 재미를 목표로 하는건 어때요?

■ 이 목표들 각각을 다음 섹션들에서 더 설명하겠습니다.

2.2.1 Performance

성능은 대부분의 병렬 프로그래밍의 주요 목표입니다. 어쨌건, 성능이 문제가 아니라면, 왜 여러분에게 호의를 베풀지 않습니까? 그냥 순차적 코드를 작성하고, 행복해지는 것 말이예요. 그건 훨씬 쉬울거고 여러분은 훨씬 더 빠르게 일을 끝낼 수 있을 겁니다.

Quick Quiz 2.7: 병렬 프로그래밍이 성능 외의 것을 위한 경우는 없나요?

■ 여기서 “성능” 이란 단어는 넓은 의미를 갖는데, 예를 들어 확장성 (CPU 당 성능)과 효율성 (watt 당 성능)을 포함함을 알아두시기 바랍니다.

그러나, 성능의 포커스는 하드웨어에서 병렬 소프트웨어로 옮겨졌습니다. 이 포커스의 변화는 Moore의 법칙은 트랜지스터 집적도 증가를 지속하게 함에도, 전통적인 단일 쓰레드 성능 증가는 중단되었다는 사실 때문에 이루어졌습니다. 이는 Figure 2.1²에 그려져 있는데,

² 이 그림은 이론적으로 클락당 한개 이상의 인스트럭션을 처리할 수 있는 신형 CPU 들의 경우 클락 주파수를, 가장 단순한 인스트럭션의 처리에도 여러 클락을 필요로 하는 구형 CPU 들의 경우에는 MIPS (보통 Dhrystone 벤치마크를 통해 얻어진, 초당 몇백만개의 인스트럭션이 처리 가능한가를 나타내는 수)를 보이고 있습니다. 두개의 측정을 사용하는 이유는 신형 CPU 의 클락당 여러 인스트럭션을 처리할 수 있는 능력이 메모리 시스템의 성능으로 인해 제한되곤하기 때문입니다. 뿐만 아니라, 구형 CPU 의 성능 측정에 사용된 벤치마크는 더이상 사용되지 않고, 신형 벤치마크를 구형 CPU 를 탑재한

싱글 쓰레드 코드를 작성하고 CPU 가 성능이 좋아지기 1-2년 기다리는건 더이상 선택지가 아님을 보입니다. 모든 주요 제조사가 멀티코어/멀티쓰레드 시스템으로 방향을 잡은 최근의 트렌드를 놓고 보면, 시스템의 완전한 성능을 내고자 하는 사람에게라면 병렬성이 맞는 방향입니다.

Quick Quiz 2.8: 프로그램을 비효율적인 스크립트 언어에서 C 나 C++ 로 재작성하는 건 어떨까요?

■ 그렇기는 하나, 첫번째 목표는 확장성보다는 성능인데, 선형적 확장성을 얻는 가장 쉬운 방법은 각 CPU 의 성능을 낮추는 것 [Tor01] 이라는 점을 놓고 보면 특히 그렇습니다. 네개의 CPU 를 탑재한 시스템이 있다면, 당신이라면 뭘 선호하겠습니까? 단일 CPU 에서 초당 100개의 트랜잭션을 처리하지만 전혀 멀티 CPU 확장성이 없는 프로그램입니까? 아니면 단일 CPU 에서 초당 10개의 트랜잭션만을 처리하지만 CPU 개수를 늘림에 따라 완전하게 확장 가능한 프로그램입니까? 첫번째 프로그램이 더 나은 선택일 겁니다, 32개 CPU 가 탑재된 시스템이라면 답이 달라질 수도 있겠지만요.

그러나, 여러분이 여러 CPU 를 가졌다는 건 그 자체로 그걸 모두 사용해야만 한다는 이유가 되지 않는 데, 특히 최근의 멀티 CPU 시스템의 가격 하락을 놓고 보면 그렇습니다. 핵심은 병렬 프로그래밍은 기본적으로 성능 최적화이며, 그건 여러 잠재적 최적화 방법 중 하나라는 것입니다. 여러분의 프로그램이 현재 작성된 대로도 충분히 빠르다면, 병렬화를 해서든 다른 잠재적 순차적 최적화 방법들을 동원해서든 최적화를 할 이유가 없습니다.³ 같은 이유로, 순차적 프로그램의 최적화 수단으로 병렬화를 하고자 한다면, 최선의 순차적 알고리즘과 병렬 알고리즘들을 비교해야 합니다. 현재 많은 출판물이 병렬 알고리즘의 성능을 논할 때 순차적 알고리즘의 경우들을 무시하곤 하기 때문에 이 부분에서 주의가 필요합니다.

2.2.2 Productivity

Quick Quiz 2.9: 왜 이건 기술적이지 않은 문제들에 대해 떠드는 거죠??? 단순히 아무 기술적이지 않은 문제가 아니라, 생산성이라구요? 누가 이걸 신경씁니까?

■ 최근 수십년간 생산성은 점점 더 중요해졌습니다. 이를 보기 위해, 초기 컴퓨터의 가격은 엔지니어 연봉이 수천달러이던 시절에 수억 달러였음을 생각해 보십시오. 그런 기계를 위해 열명의 엔지니어 팀을 전담시키는

시스템에서 돌리는 건 어려운데, 부분적으로는 동작하는 구형 CPU 를 찾기도 쉽지 않기 때문입니다.

³ 물론, 여러분이 병렬 소프트웨어를 작성하는게 주요 관심사인 취미 생활자라면 여러분이 관심있는 소프트웨어가 무엇이건 병렬화를 할 충분한 이유가 됩니다.

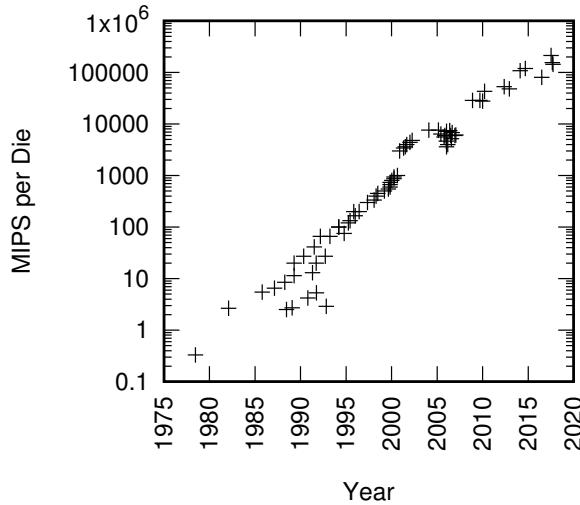


Figure 2.2: MIPS per Die for Intel CPUs

것이 그 성능을 10% 라도 향상시킨다면, 그들의 연봉은 여러번 더 지불될 수 있을 겁니다.

그런 기계 중 하나로 CSIRAC 가 있는데, 가장 오래되었고 여전히 손상되지 않은 stored-program 컴퓨터로, 1949년부터 작동되었습니다 [Mus04, Dep06]. 이 기계는 트랜지스터 시대 전에 만들어졌으므로, 2,000 개의 진공관으로 만들어졌고 1 kHz 클락 주파수로 작동하며, 30 kW 전력을 소비하고, 3 톤의 무게를 가졌습니다. 이 기계가 768 워드 RAM 을 가졌음을 생각해 보면, 이 기계는 오늘날의 거대 규모 소프트웨어 프로젝트가 시달리는 생산성 이슈로 고생하지는 않았을 거라 말할 수 있을 겁니다.

오늘날, 이렇게 작은 컴퓨팅 파워를 가진 기계를 구입하기 상당히 어려울 겁니다. 가장 비슷한 장비는 아마도 유서 깊은 Z80 [Wik08]로 대표되는 8-bit 임베디드 마이크로프로세서가 될 겁니다만, 오래된 Z80 조차도 CSIRAC 보다 1,000 배나 빠른 CPU 클락 주파수를 가졌습니다. Z80 CPU는 8,500 개 트랜지스터를 가졌고, 2008년 기준으로 1,000개씩 구매하면 개당 \$2 US 도 하지 않았습니다. CSIRAC에 대비되기도, Z80에 있어 소프트웨어 개발 비용은 중요치 않습니다.

CSIRAC 와 Z80은 장기적 트렌드 상의 두 지점으로, Figure 2.2 위에서 볼 수 있습니다. 이 그림은 다이당 예상 연산력을 과거 40년의 세월에 걸쳐 그리고 있는데, 40년간 백만배가 넘는 인상적인 향상을 보입니다. 멀티 코어 CPU의 발전은 2003년 이후 마주친 클락 주파수 장벽에도 불구하고 다이당 50개의 하드웨어 쓰레드를 지원함으로써 이 향상을 지속가능하게 했음을 알아두시기 바랍니다.

하드웨어 가격의 가파른 하락의 부정할 수 없는 결론은 소프트웨어 생산성이 갈수록 중요해져 가고 있다

는 겁니다. 단순히 하드웨어를 효율적으로 사용하는 건 더이상 충분치 않습니다: 이제 소프트웨어 개발자들을 극단적으로 효율성 있게 활용할 필요가 있습니다. 이는 순차적 하드웨어에 있어서 오랫동안 그랬습니다만, 병렬 하드웨어는 최근 들어서야 저렴한 일반 상품이 되었습니다. 따라서, 최근에 들어서야 높은 생산성이 병렬 소프트웨어를 만드는 데 있어 크게 중요해 졌습니다.

Quick Quiz 2.10: 병렬 시스템이 그렇게 싸졌는데, 그걸 프로그램하라고 누가 돈을 줄까요?

적어도 한때, 병렬 소프트웨어의 단 한가지 목적은 성능이었습니다. 하지만, 지금 생산성은 더 많은 주목을 받고 있습니다.

2.2.3 Generality

병렬 소프트웨어 개발의 높은 비용을 정당화 하는 한가지 방법은 최대한의 범용성을 갖추는 것입니다. 다른게 모두 똑같다면, 더 범용적인 소프트웨어는 그 비용이 더 많은 사람들에게 나뉘어 질 수 있을 겁니다. 사실, 이 경제성이 범용성의 중요한 특수 케이스인 이식성에 대한 매니악한 주목을 설명합니다.⁴

불행히도, 범용성은 성능, 생산성, 또는 둘 다의 비용을 초래합니다. 예를 들어, 이식성은 적용 레이어를 통해 얻어지는데, 이는 어쩔 수 없이 성능 저하를 일으킵니다. 이를 더 일반적으로 보기 위해, 다음의 유명한 병렬 프로그래밍 환경들을 생각해 봅시다:

C/C++ “락킹과 쓰레드”: POSIX 쓰레드 (pthreads) [Ope97], Windows 쓰레드, 다양한 운영체제 커널 환경을 포함하는 이 카테고리는 (적어도 하나의 SMP 시스템 내에서는) 훌륭한 성능과 좋은 범용성을 제공합니다. 상대적으로 낮은 생산성이 유감입니다.

Java: 범용이고 본질적으로 멀티쓰레드 기반인 이 프로그래밍 환경은 자동화된 가비지 컬렉터와 풍부한 클래스 라이브러리 덕에 C 나 C++ 보다 훨씬 높은 생산성을 제공한다고 널리 믿어집니다. 하지만, 그 성능은 2000년대 초반에 상당히 개선되긴 했지만 C 와 C++에 비해선 훨씬 떨어집니다.

MPI: 이 Message Passing Interface [MPI08]는 세계에서 가장 거대한 과학과 기술 분야 컴퓨팅 클러스터를 굽이며 전대미문의 성능과 확장성을 제공합니다. 이론적으로 이건 범용입니다만 과학과 기술 분야 컴퓨팅에 주로 사용됩니다. 이것의 생산성은 많은 이들에게 C/C++ “락킹과 쓰레드” 보다도 낮다고 여겨집니다.

⁴ 이걸 짚어준 Michael Wong에게 감사의 말씀을 드립니다.

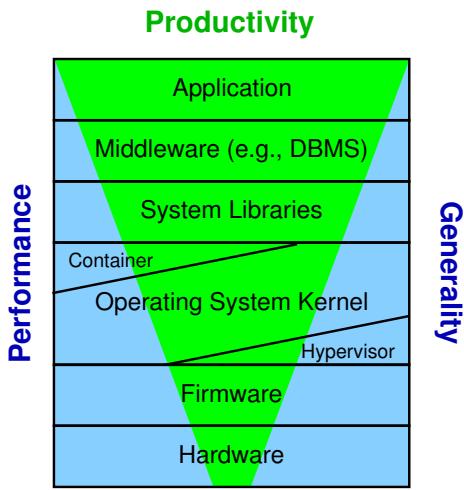


Figure 2.3: Software Layers and Performance, Productivity, and Generality

OpenMP: 이 컴파일러 지시어 집합은 병렬화 하는데 사용될 수 있습니다. 때문에 이 작업에 상당히 특수화 되어 있으며, 이 특수성이 종종 그 성능을 제한합니다. 하지만, MPI나 C/C++ “락킹과 쓰레드” 보다 훨씬 사용하기가 쉽습니다.

SQL: Structured Query Language [Int92]는 관계형 데이터베이스 질의에 특수화 되어 있습니다. 하지만, Transaction Processing Performance Council (TPC) 벤치마크 결과 [Tra01]로 측정된 바에 따르면 그 성능은 상당히 좋습니다. 생산성도 훌륭합니다; 사실, 이 병렬 프로그래밍 환경은 사람들이 병렬 프로그래밍 개념에 대한 지식이 적거나 아예 없더라도 거대한 병렬 시스템을 잘 사용할 수 있게 해줍니다.

세계급의 성능, 생산성, 그리고 범용성을 제공하는 병렬 프로그래밍 환경의 천국은 아직 존재하지 않습니다. 그런 천국이 나타나기 전까지는, 성능, 생산성, 범용성 사이에서 엔지니어링 트레이드오프를 해야 합니다. 그런 트레이드오프 중 하나가 Figure 2.3에 보여져 있는데, 생산성이 시스템 스택의 상층부에서 점점 더 중요해지고 있으며, 반면 성능과 범용성은 하층부에서 점점 더 중요해지고 있음을 보입니다. 하층부에서 발생하는 거대한 개발 비용은 많은 수의 사용자로 나누어지며 (따라서 범용성이 중요합니다), 하층부의 성능 저하는 상층부에서 회복시킬 수 없습니다. 이 스택의 상층부는, 해당 어플리케이션의 사용자는 매우 적을 수도 있으므로, 생산성에 대한 염려가 주요합니다. 이는 스택의 위쪽으로 갈수록 “bloatware”가 되어가는 경향을 설명합니다: 하드웨어를 추가하는 건 종종 개발자를 추

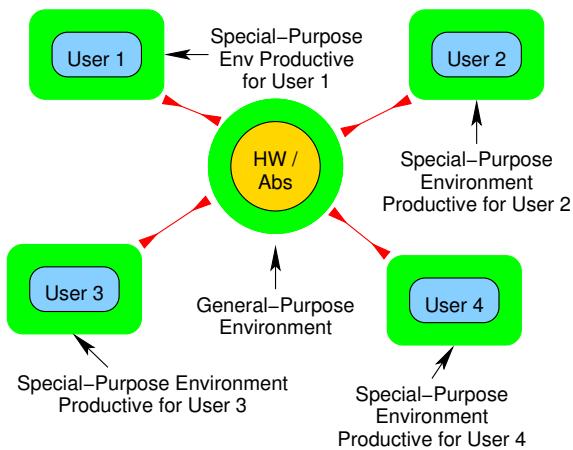


Figure 2.4: Tradeoff Between Productivity and Generality

가하는 것보다 저렴합니다. 이 책은 이 성능과 범용성이 주요 관심사인, 스택의 바닥 근처에서 일하는 개발자를 위해 쓰였습니다.

생산성과 범용성 사이의 트레이드오프는 많은 영역에 수세기동안 존재해 왔음을 알아둘 필요가 있습니다. 한가지만 예를 들어보면, 뭇박기 기계는 망치보다 못을 박는데는 더 생산적이지만, 망치가 상대적으로 뭇박기 외의 더 많은 일에 사용될 수 있습니다. 따라서 비슷한 트레이드오프가 병렬 컴퓨팅 분야에서도 발견되는 건 놀라운 일이 아닙니다. Figure 2.4에 이 점이 간략히 그려져 있습니다. 사용자 1, 2, 3, 그리고 4는 컴퓨터의 도움을 필요로 하는 일이 있습니다. 해당 사용자를 위해 가장 생산적인 언어나 환경은 프로그래밍, 설정, 또는 다른 셋업 없이도 그 사용자의 일을 하는 것일 겁니다.

Quick Quiz 2.11: 이건 이루어질 수 없는 이상이예요! 왜 실제로 이를 수 있는 것들에 집중하지 않는 거죠?

■

불행히도, 사용자 1에 의해 요청된 일을 하는 시스템은 사용자 2의 일을 해주진 않을 가능성이 큽니다. 달리 말하자면, 가장 생산적인 언어와 환경은 도메인에 특화되어 있으며, 따라서 그 정의에 따라 범용성이 떨어집니다.

다른 선택지는 Figure 2.4의 가운데 원으로 보여진 것처럼 주어진 프로그래밍 언어와 환경을 하드웨어 시스템에 맞추거나 (예를 들어, 어셈블리, C, C++, 또는 Java 같은 저수준 언어들) 추상화 계층에 맞추는 (예를 들어, Haskell, Prolog, 또는 Snobol) 겁니다. 이 언어들은 모두 사용자 1, 2, 3, 그리고 4에게 필요한 일에 모두 똑같이 특수화 되지 않았다는 점에서 범용적으로 여겨질 수 있습니다. 더 나쁜 건, 특정 추상화 계층에 맞춰진 언어는 그 추상화가 실제 하드웨어에 효율적으로

매핑되지 않는다면 성능과 확장성 문제를 겪을 가능성 이 높다는 겁니다.

성능, 생산성, 그리고 범용성이라는 철의 삼각지대의 상충되는 세 가지 목표로부터 벗어날 수는 없을까요?

많은 경우에 탈출구가 존재하는데, 예를 들어 다음 세션에서 다루어질 병렬 프로그래밍의 대안을 사용하는 것입니다. 어쨌건, 병렬 프로그래밍은 무척 즐거운 선택이 될 수 있지만, 항상 최고의 도구가 되는건 아닙니다.

2.3 Alternatives to Parallel Programming

Experiment is folly when experience shows the way.

Roger M. Babson

병렬 프로그래밍의 대안을 제대로 고려하려면, 먼저 여러분은 병렬성이 여러분에게 뭘 해줄 거라고 생각하고 있는지 정확히 알아야 합니다. Section 2.2에서 알아봤듯, 병렬 프로그래밍의 주요 목표는 성능, 생산성, 그리고 범용성입니다. 이 책은 소프트웨어 스택의 바닥 근처에 있는 성능이 중요한 코드를 작업하는 개발자들을 위해 쓰여졌기 때문에, 이 섹션의 나머지는 기본적으로 성능 개선에 주목합니다.

하지만 병렬성은 성능을 개선하는 한가지 방법에 불과할 뿐임을 명심해 두는게 중요합니다. 다른 잘 알려진 방법들로는 어려운 순서대로 대략적으로 나열해 보면 다음과 같은 것들이 있습니다:

1. 순차적 어플리케이션의 여러 인스턴스를 돌리기.
2. 이미 존재하는 병렬 소프트웨어를 사용해 어플리케이션을 만들기.
3. 순차적 어플리케이션을 최적화하기.

이 방법들이 다음 섹션에 다뤄집니다.

2.3.1 Multiple Instances of a Sequential Application

순차적 어플리케이션의 인스턴스 여러개를 돌리는 것은 병렬 프로그래밍을 정말로 하지는 않으면서 병렬 프로그래밍을 할 수 있도록 해줍니다. 해당 어플리케이션의 구조에 따라, 이 접근법을 위한 여러 방법이 있습니다.

여러분의 프로그램이 여러개의 다양한 시나리오를 분석한다면, 또는 여러개의 독립적 데이터 셋을 분석한다면, 쉽고도 효과적인 방법 중 하나는 하나의 분석을

하는 하나의 순차적 프로그램을 만든 후, 여러 스크립트 환경 (예를 들어 bash 쉘) 중 아무거나 하나를 사용해 이 순차적 프로그램의 여러 인스턴스를 병렬로 수행하는 것입니다. 어떤 경우에는, 이 접근법은 여러 기계의 클러스터로 쉽게 확장될 수도 있습니다.

이 방법은 사기처럼 보일 수도 있겠고, 어떤 사람들은 실제로 그런 프로그램을 “부끄럽게도 병렬적”이라며 모욕하기도 합니다. 그리고 실제로, 이 접근법은 메모리 소모량 증가, 공통되는 중간 결과를 재계산하기 위해 낭비되는 CPU 사이클, 데이터 복사의 증가 등의 일부 잠재적 단점이 있습니다. 하지만, 이는 많은 경우 극단적으로 생산적이고, 아주 적은 양의 추가적 노력만으로 극단적 성능 향상을 얻게 해줍니다.

2.3.2 Use Existing Parallel Software

관계형 데이터베이스 [Dat82], 웹 어플리케이션 서버, 맵리듀스 환경 등을 포함해 싱글쓰레드 프로그래밍 환경을 제공하는 병렬 소프트웨어 환경은 더이상 부족하지 않습니다. 예를 들어, 흔한 설계 중 하나는 각 사용자를 위해 사용자의 질의들로부터 SQL을 생성하는 프로세스를 유저별로 제공하는 것입니다. 이 유저별 SQL은 이 사용자들의 질의들을 자동으로 동시에 수행하는 흔한 관계형 데이터베이스에서 돌아가게 됩니다. 이 유저별 프로그램들은 해당 유저 인터페이스에만 책임이 있으며, 관계형 데이터베이스가 병렬성과 지속성을 둘러싼 어려운 문제들에 대한 모든 책임을 갖습니다.

또한, 병렬 라이브러리 함수들이 늘어나고 있는데, 특히 숫자 계산을 위한 것들입니다. 더 나은 것이, 일부 라이브러리는 벡터 유닛과 범용 그래픽 처리 장치 (GPGPU) 와 같은 특수목적 하드웨어를 활용합니다.

이 접근법을 취하는 것은 종종 성능을 일부 희생하는데, 주의깊게 손으로 코딩된 완전한 병렬 어플리케이션에 비해서는 그렇습니다. 하지만, 그런 희생은 많은 경우 개발 노력의 큰 감소로 보상됩니다.

Quick Quiz 2.12: 잠깐만요! 이 방법은 개발 노력 을 당신으로부터 당신이 사용하는 병렬 소프트웨어를 개발한 누군가에게 떠넘길 뿐인 것 아닌가요?



2.3.3 Performance Optimization

2000년대 초반까지, CPU 클럭 주파수는 매 18개월마다 두배가 되었습니다. 때문에, 주의 깊게 성능을 최적화하는 것보다 새로운 기능을 만드는게 일반적으로 더 중요했습니다. 이제 무어의 법칙은 트랜지스터 집적도와 트랜지스터당 성능을 모두 증가시키는 대신, “오로지” 트랜지스터 집적도만 증가시키므로, 성능 최적화의 중요성을 다시 생각해 보기 좋은 때일지도 모릅니다. 어쨌건, 새로운 하드웨어 세대는 더이상 대단한 싱글쓰레드

성능 개선을 가져오지 않습니다. 더 나아가서, 많은 성능 최적화는 전력을 아끼기도 합니다.

이런 관점에서, 병렬 프로그래밍은 그저 하나의 성능 최적화일 뿐이지만, 병렬 시스템이 점점 저렴해지고 더 쉽게 접근 가능해져 가고 있기 때문에 훨씬 매력적이 되어가고 있습니다. 하지만, 병렬화를 통해 얻을 수 있는 속도 개선은 CPU 갯수에 대략적으로 제한됨을 명심하기 바랍니다 (하지만 흥미로운 예외 경우를 위해 Section 6.5 을 보기 바랍니다). 반대로, 전통적인 싱글 쓰레드 소프트웨어 최적화로 얻을 수 있는 속도 증가는 훨씬 클 수 있습니다. 예를 들어, 긴 링크드 리스트를 해시 테이블 또는 탐색 트리로 교체하는 것은 수십수 백배 성능을 개선할 수 있습니다. 이 고도로 최적화된 싱글쓰레드 프로그램은 최적화 되지 않은 병렬 버전에 비해 훨씬 빠를 수 있어서, 병렬화를 불필요하게 만들 수도 있습니다. 물론, 고도로 최적화된 병렬 프로그램은 그보다도 나을 겁니다, 추가적인 개발 작업이 필요하겠지만요.

더 나아가서, 다른 프로그램은 다른 성능 병목지점을 가질 수도 있습니다. 예를 들어, 여러분의 프로그램이 디스크 드라이브로부터 데이터를 기다리는데 시간을 대부분 소모하고 있다면, 여러 CPU를 사용하는건 디스크를 기다리는 시간만 증가시킬 뿐일 겁니다. 사실, 그 프로그램이 회전형 디스크에 직렬로 놓여진 거대한 하나의 파일을 읽고 있었다면, 그걸 병렬화 하는건 추가된 탐색 오버헤드 때문에 더 느려진 결과를 낼 겁니다. 여러분은 그 대신 그 데이터의 배치를 변경해서 그 파일을 더 작게 만들고 (따라서 더 빨리 읽을 수 있고), 그 파일을 다른 드라이브들로부터 병렬로 접근될 수 있게 조각들로 쪼개고, 자주 접근되는 데이터를 메인 메모리에 캐쉬하고, 또는, 만약 가능하다면 읽어야만 하는 데이터의 양 자체를 줄여야 합니다.

Quick Quiz 2.13: 어떤 다른 병목지점이 추가된 CPU 가 성능을 개선하는 걸 막을 수 있을까요?



병렬성은 강력한 최적화 기술이 될 수 있지만, 그게 유일한 기술도 아니고 모든 상황에 적합한 것도 아닙니다. 물론, 여러분의 프로그램이 병렬화 하기 더 쉬울수록, 병렬화는 더 매력적인 최적화 수단이 됩니다. 병렬화는 상당히 어렵다는 평판이 있어서, “구체적으로 무엇이 병렬 프로그래밍을 그렇게 어렵게 만드는가?” 라는 질문을 이끌어 냅니다.

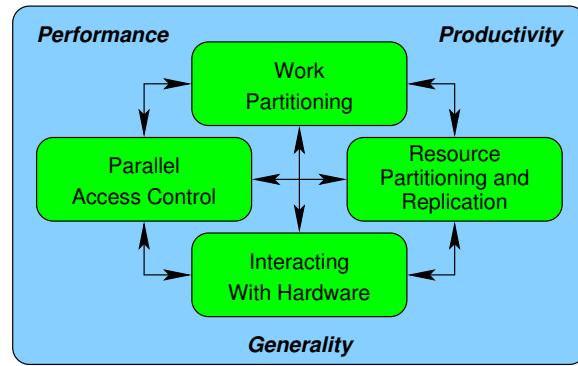


Figure 2.5: Categories of Tasks Required of Parallel Programmers

2.4 What Makes Parallel Programming Hard?

Real difficulties can be overcome; it is only the imaginary ones that are unconquerable.

Theodore N. Vail

병렬 프로그래밍은 그게 병렬 프로그래밍 문제의 기술적 성격 만큼이나 사람에 연관된 것이라는 걸 아는게 중요합니다. 우리는 프로그래밍이라고도 알려진, 인류가 병렬 시스템에게 무슨 일을 해야할지 말할 수 있는 것이 필요합니다. 하지만 병렬 프로그래밍은 쌍방향의 소통이 사용되는데, 프로그램의 성능과 확장성은 기계에서 사람으로의 소통이 됩니다. 짧게 말해서, 사람은 컴퓨터에게 무슨 일을 할지 이야기하는 프로그램을 짜고, 컴퓨터는 이 프로그램을 결과 성능과 확장성으로 비평합니다. 따라서, 추상화나 수학적 분석은 상당히 제한된 도구가 됩니다.

산업혁명에서, 사람과 기계 사이의 인터페이스는 사람족의 것들에 대한 연구로 평가되었고, 이후 time-and-motion 연구라 불리게 되었습니다. 병렬 프로그래밍을 평가하는 사람족의 것들에 대한 연구도 좀 있긴 합니다만 [ENS05, ES05, HCS⁺05, SS94], 이 연구들은 극단적으로 좁은 분야에 주목하고 있고, 따라서 일반적 결과는 전혀 보이지 못합니다. 더욱이, 프로그래머의 개개인의 생산성은 열배 이상 차이남을 놓고 보면, 생산성의 (말하자면) 10% 차이를 발견하는 것을 가능케 하는 연구가 연구비를 지원받는 건 비현실적입니다. 그런 연구가 수백 수천배의 차이를 안정적으로 발견해낼 수 있다고 하면 매우 가치있는 일이겠으나, 가장 인상적인 개선도 10% 개선 정도입니다.

그러니 우린 다른 전략을 취해야 합니다.

그런 전략 가운데 하나는 순차적 프로그래머에겐 요구되지 않지만 병렬 프로그래머에겐 요구되는 것을 유의 깊게 고려해 보는 겁니다. 그러면 특정 프로그래밍 언어나 환경이 개발자가 이 일들을 하는 걸 얼마나 잘 도와주는지 평가할 수 있을 겁니다. 이 일들은 Figure 2.5에 보인 네가지 카테고리에 들어가는데, 이 각각이 다음 섹션들에서 다루어집니다.

2.4.1 Work Partitioning

일 분할하기는 병렬 수행에 절대적으로 필수적입니다: 단 하나의 일 “덩어리” 만이 존재한다면, 그건 한번에 하나의 CPU 를 통해서만 수행될 수 있는데, 이는 곧 순차적 수행의 정의입니다. 하지만, 코드를 분할하는 건 상당한 주의가 필요합니다. 예를 들어, 비동일한 크기로 분할하는 것은 작은 조각들이 완료된 후에는 순차적 수행이 이뤄지는 결과를 낳습니다 [Amd67]. 덜 극단적인 경우에는, 사용 가능한 하드웨어가 모두 사용되도록 함으로써 성능과 확장성을 회복하게 하기 위해 로드밸런싱이 사용될 수 있습니다.

분할하기가 성능과 확장성을 크게 개선할 수 있지만, 이는 또한 복잡성을 늘릴 수 있습니다. 예를 들어, 분할하기는 전역적 에러와 이벤트의 처리를 복잡하게 만들 수 있습니다: 어떤 병렬 프로그램은 그런 전역적 이벤트를 안전하게 처리하기 위해 사소하지 않은 수준의 동기화를 해야만 할 수 있습니다. 더 일반적으로, 각각의 조각은 소통 같은 걸 필요로 합니다: 어쨌건, 어떤 쓰레드가 전혀 소통을 하지 않는다면, 그 쓰레드는 어떤 영향도 끼치지 못할 테고 그러니 수행될 필요도 없습니다. 하지만, 소통은 오버헤드를 일으키므로, 주의 깊지 않게 분할하기를 선택하는 것은 상당한 성능 하락을 초래할 수 있습니다.

더 나아가서, 동시에 수행되는 쓰레드의 수는 많은 경우 제어되어야만 하는데, 그런 쓰레드 각각은 예를 들면 CPU 캐시 공간과 같은 공유 자원을 사용하기 때문입니다. 너무 많은 쓰레드가 동시에 수행될 수 있다면, 이 CPU 캐시가 넘쳐나서, 높은 캐시 미스율을 초래하고, 이는 성능 하락을 의미합니다. 거꾸로, 연산과 I/O 를 겹치게 함으로써 I/O 기기가 완전히 사용되게 하기 위해 많은 수의 쓰레드가 요구되는 경우도 많습니다.

Quick Quiz 2.14: CPU 캐시 용량 외에, 어떤 것이 동시에 수행되는 쓰레드의 수를 제한할 수 있을까요?

마지막으로, 쓰레드들이 동시에 수행될 수 있도록 하는 것은 프로그램의 상태 공간을 상당히 증가시키며, 이는 이 프로그램을 이해하고 디버깅하기 어렵게 만들어 생산성을 하락시킵니다. 다른 모든게 동일하다면, 더 일반적인 구조를 갖는 더 작은 상태 공간이 더 이해하기 쉽습니다만, 이는 그게 기술적이나 수학적 이야기인 만큼 인간적 요소에 대한 이야기입니다. 좋은 병렬 프로그

램 설계는 극단적으로 큰 상태 공간을 가질 수도 있지만, 그 일반적인 구조 덕분에 이해하기가 쉬운 반면, 안 좋은 설계는 상대적으로 작은 상태 공간을 가지고도 불가 해할 수 있습니다. 최선의 설계는 부끄러운 병렬성을 활용하거나, 그 문제를 부끄러운 병렬성 해결책을 갖는 것으로 변형시킵니다. 어느 쪽이든, “부끄러운 병렬성”은 실제로 부자들의 부끄러움입니다. 좋은 설계에 대한 것들 중 현재로써 가장 훌륭한 것은 다음과 같습니다: 상태 공간 크기와 구조에 대한 일반적 평가를 위해선 더 많은 연구가 필요하다.

2.4.2 Parallel Access Control

싱글쓰레드의 순차적 프로그램이 있다면, 그 쓰레드는 이 프로그램의 자원 전체에 대한 접근 권한을 갖습니다. 이 자원은 대부분의 경우 메모리 내에 존재하는 데이터 구조들입니다만, CPU, 메모리 (캐시 포함), I/O 기기, 연산 가속기, 파일, 그외에도 여러가지가 될 수 있습니다.

병렬 접근 제어 문제 중 첫번째는 특정 자원에 대한 접근의 형태가 해당 자원의 위치에 종속적이나는 것입니다. 예를 들어, 많은 메세지 패싱 환경에서, 지역 변수에 대한 접근은 표현 (expression) 과 할당 (assignment) 을 통해 이루어지는 반면, 원격 변수로의 접근은 메세징이 포함되는, 전혀 다른 구문을 사용합니다. POSIX 쓰레드 환경 [Ope97], Structured Query Language (SQL) [Int92], 그리고 Universal Parallel C (UPC) [EGCD03, CBF13] 같은 분할된 전역 어드레스 공간 (PGAS) 환경은 암묵적 접근을 제공하지만, Message Passing Interface (MPI) [MPI08]은 원격 데이터로의 접근은 명시적 메세징을 필요로 하기 때문에 명시적 접근을 제공합니다.

다른 병렬 접근 제어 문제는 여러 쓰레드 간에 자원으로의 접근을 어떻게 조정할 것인가입니다. 이 조정은 다양한 병렬 언어와 환경에서 제공되는 많은 동기화 메커니즘을 통해 이루어지는데, 메세지 패싱, 락킹, 트랜잭션, 레퍼런스 카운팅, 명시적 타이밍, 공유 원자적 변수, 그리고 데이터 소유권 등이 포함됩니다. 많은 전통적 병렬 프로그래밍 이 조정으로부터 발생하는 데드락, 라이브락, 트랜잭션 롤백 등을 걱정해 왔습니다. 이 프레임워크는 이 동기화 메커니즘들의 비교를 포함하도록 더 개선될 수 있겠는데, 예를 들어 락킹 대 트랜잭션 메모리 [MMW07] 같은 것입니다만, 그런 개선은 이 섹션의 범위를 벗어나는 것입니다. (트랜잭션 메모리에 대한 더 많은 정보를 위해선 Section 17.2 과 17.3 를 참고하세요.)

Quick Quiz 2.15: “명시적 타이밍” 이 정확히 뭔가요???



2.4.3 Resource Partitioning and Replication

가장 효과적인 병렬 알고리즘과 시스템은 자원 병렬성을 상당히 노출시키는데, 따라서 보통은 여러분의 쓰기 집약적 자원은 분할을 시키고 자주 접근되며 대부분 읽기만 되는 자원은 복사를 하는 것으로 병렬화를 시작하는 게 현명한 선택입니다. 여기서 말하는 자원은 대부분의 경우 데이터로, 컴퓨터 시스템간, 대용량 저장장치간, NUMA 노드간, CPU 코어간(또는 다이간이나 하드웨어 쓰레드간), 페이지간, 캐시라인간, 동기화 도구의 인스턴스간, 또는 코드의 크리티컬 섹션간으로 분할될 수 있을 수도 있을 겁니다. 예를 들어, 락킹 도구간에 분할하는 것은 “데이터 락킹” [BK85] 이라고 불립니다.

자원 분할은 종종 어플리케이션에 종속적입니다. 예를 들어, 수학 어플리케이션은 종종 행렬을 행으로, 열로, 또는 부분 행렬로 분할하는 경우가 많습니다만, 상업 어플리케이션들은 종종 쓰기 집약적 데이터 구조를 분할시키고 읽기가 대부분인 데이터 구조는 복사를 하곤 합니다. 따라서, 상업 어플리케이션은 특정 고객을 위한 데이터를 거대한 클러스터 중 일부 컴퓨터에 할당할 수 있을 겁니다. 어떤 어플리케이션은 데이터를 정적으로 분할할 수도, 시간에 따라 동적으로 분할할 수도 있을 겁니다.

자원 분할은 굉장히 효과적입니다만, 복잡하게 연결된 데이터 구조의 경우엔 매우 어려울 수 있습니다.

2.4.4 Interacting With Hardware

하드웨어와의 상호작용은 일반적으로 운영체제, 컴파일러, 라이브러리, 또는 다른 소프트웨어 환경 인프라의 영역입니다. 하지만, 최신 하드웨어 기술과 컴포넌트를 가지고 작업하는 개발자들은 그런 하드웨어를 직접적으로 사용해야 할 경우가 많습니다. 또한, 하드웨어로의 직접적 접근은 주어진 시스템의 모든 성능을 마지막 한방울까지 쥐어짤 때 필요할 수 있습니다. 이 경우, 개발자는 목표 하드웨어의 캐시 구조, 시스템 자형, 또는 인터컨넥트 프로토콜에 맞춰 어플리케이션을 재단하거나 구성해야 할 수 있습니다.

어떤 경우, 하드웨어는 앞의 섹션들에서 설명한 것처럼 분할하거나 액세스 제어를 하기 쉽다고 여겨질 수도 있습니다.

2.4.5 Composite Capabilities

이 네 가지 것들이 기본적인 것들이긴 하지만, 훌륭한 엔지니어링 연습은 이것들의 조합을 사용합니다. 예를 들어, 데이터 병렬화 전략은 먼저 분할된 조각간 소통의 필요를 최소화 할 수 있게끔 데이터를 분할하고, 코드

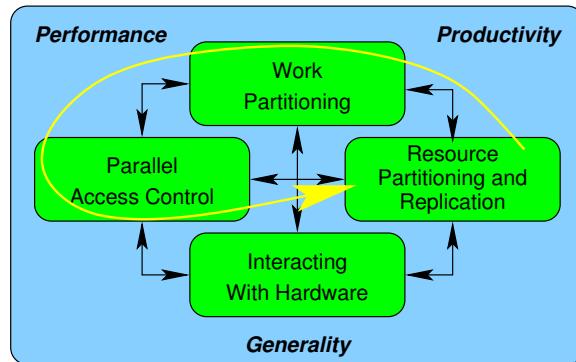


Figure 2.6: Ordering of Parallel-Programming Tasks

역시 그렇게 분할한 후, 마지막으로 데이터 조각과 쓰레드를 쓰레드간 소통은 최소화하면서 처리량은 최대화 할 수 있게끔, Figure 2.6에 보여진 것처럼 매핑합니다. 그러면 개발자는 각 조각을 개별적으로 들여다 볼 수 있어서, 관련된 상태 공간의 크기를 무척 줄고, 결과적으로 생산성이 증대됩니다. 어떤 문제들은 분할이 불가능하다 하더라도, 분할이 가능한 형태로 그것을 잘 변형시키는 것이 성능과 확장성을 모두 크게 증대시킬 수 있게 하는 경우가 있습니다 [Met99].

2.4.6 How Do Languages and Environments Assist With These Tasks?

많은 환경이 개발자에게 이 작업들을 직접 하기를 요구하지만, 상당한 자동화를 이룬 오래된 환경들이 있습니다. 이런 환경들의 전형적인 예는 SQL로, 하나의 거대한 질의문을 자동으로 병렬화 시키고, 개별 질의와 업데이트의 동시 수행 또한 자동화 합니다.

이 네 개의 작업 카테고리는 모든 병렬 프로그램에서 행해져야 합니다만, 이 작업들을 개발자가 일일이 해야만 함을 의미하지는 않습니다. 병렬 시스템이 저렴해지고 쉽게 사용가능해 져가기를 계속할수록 이 네 개의 작업의 자동화가 점차 늘어나는 것을 볼 수 있을 거라 기대할 수 있겠습니다.

Quick Quiz 2.16: 병렬 프로그래밍에 어떤 다른 장애물들도 있을까요?



2.5 Discussion

Until you try, you don't know what you can't do.

Henry James

이 섹션은 병렬 프로그래밍의 어려움에 대한 개론을 목표와 대안과 함께 제공했습니다. 이 개론에 이어서 무엇이 병렬 프로그래밍을 어렵게 만들 수 있는지에 대한 토론을 병렬 프로그래밍의 어려움을 다루기 위한 고수준에서의 전략이 이어졌습니다. 병렬 프로그래밍이 사용이 불가할 정도로 어렵다고 여전히 주장하고자 하는 분들은 병렬 프로그래밍의 더 오래된 안내서를 읽어보셔야 합니다 [Seq88, Bir89, BK85, Imm85]. Andrew Birrell의 다음과 같은 인용문 [Bir89]이 그 점을 특히 잘 이야기 합니다:

동시성 있는 프로그램을 작성하는 것은 익숙지 않고 어렵다는 평판을 받아왔습니다. 나는 둘 다 사실이 아니라 믿습니다. 여러분은 좋은 도구들과 적합한 라이브러리를 제공하는 시스템이 필요하고, 기본적인 조심과 주의가 필요하며, 유용한 기술의 무기고가 필요하고, 일반적인 위험들을 알아야 합니다. 이 논문이 제 믿음을 여러분께 공유하는데 도움이 되길 바랍니다.

이 오래된 안내서의 저자들은 1980년대에 병렬 프로그래밍의 어려움을 직면했습니다. 그러니, 21세기 현재에 와서 병렬 프로그래밍의 도전을 거부할 변명이 없습니다!

이제 우린 우리의 병렬 소프트웨어 아래 위치한 병렬 하드웨어의 속성에 빠져볼 다음 챕터로 넘어갈 준비가 되었습니다.

Premature abstraction is the root of all evil.

A cast of thousands

Chapter 3

Hardware and its Habits

대부분의 사람들이 시스템간에 메세지를 주고받는 것 이 단일 시스템 내에서의 간단한 계산을 수행하는 것보다는 더 비싸다는 것을 직관적으로 이해하고 있습니다. 하지만 단일 공유메모리 시스템 내에서 쓰레드간의 커뮤니케이션 또한 상당히 비쌀 수 있습니다. 이 약간의 페이지는 공유 메모리 병렬 하드웨어 설계의 곁을 훑는 것 이상은 하지 못할 겁니다: 더 많은 자세한 내용을 원하는 독자 분들은 Hennessy 의 최신 판본과 Patterson 의 고전 교과서 [HP17, HP95] 를 읽는 걸로 시작하면 좋을 겁니다.

Quick Quiz 3.1: 병렬 프로그래머는 왜 하드웨어의 저수준 특성을 배우는 걸 신경써야 하죠? 추상화의 움단에 머무르는 게 더 쉽고, 낫고, 더 우아하지 않을까요?

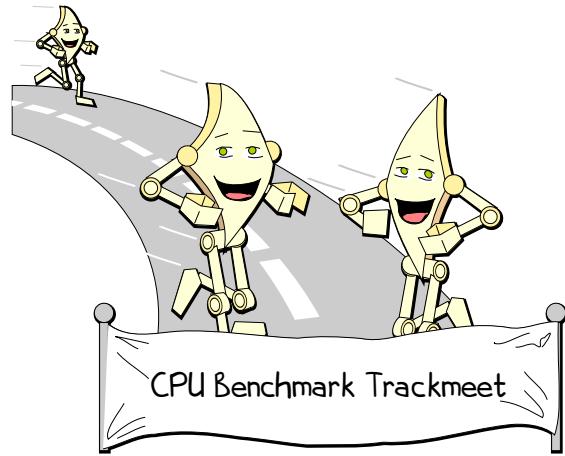


Figure 3.1: CPU Performance at its Best

3.1 Overview

Mechanical Sympathy: Hardware and software working together in harmony.

Martin Thompson

컴퓨터 시스템의 스펙 시트를 부주의하게 읽는 것은 누군가를 CPU 성능은 Figure 3.1 에 그려진 것처럼 가장 빠른 사람이 항상 이기는, 깨끗한 운동장에서의 달리기 경주라고 생각하게 만들 수 있습니다.

Figure 3.1 에 그려진 이상적인 경우를 다루는 일부 CPU 국한적 벤치마크들이 존재하지만, 일반적 프로그램은 평범한 달리기 경주보다는 장애물 달리기 경주를 더 닮았습니다. 이는 CPU 의 내부 구조가 지난 수십년간 Moore의 법칙 덕에 극적으로 변화했기 때문입니다. 이 변화들을 다음 섹션들에서 설명합니다.

3.1.1 Pipelined CPUs

1980년대에 일반적인 마이크로프로세서는 명령을 (instruction) 가져와 (fetch) 해석하고 (decode) 수행했는데 (execute), 하나의 명령을 완료하는데 일반적으로 최소 세개의 클럭 사이클을 필요로 했습니다. 반면에, 1990년대 후반과 2000년대의 CPU 는 CPU 를 거치는 명령과 데이터의 처리 흐름을 최적화 하기 위해 파이프라인; 슈퍼스칼라 기술; 비순차 명령과 데이터 처리; 투기적 실행 기술 등을 사용해 여러 명령을 동시에 수행합니다 일부 코어는 두개 이상의 하드웨어 쓰레드를 갖는데, 이는 *simultaneous multithreading* (SMT) 또는 *하이퍼쓰레딩* (HT) [Fen73] 이라 불리며, 이 쓰레드 각각은 소프트웨어에게 적어도 기능적 관점에서는 개별 CPU 로 보이게 됩니다. 이런 근대의 하드웨어 기능들은 Figure 3.2 에 보여진 것처럼 성능을 상당히 개선할 수 있습니다.

긴 파이프라인을 갖는 CPU 에서 완전한 성능을 이끌어 내는데에는 프로그램의 상당히 예측 가능한 제어

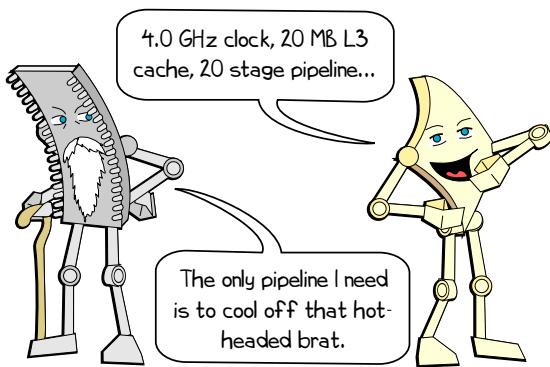


Figure 3.2: CPUs Old and New

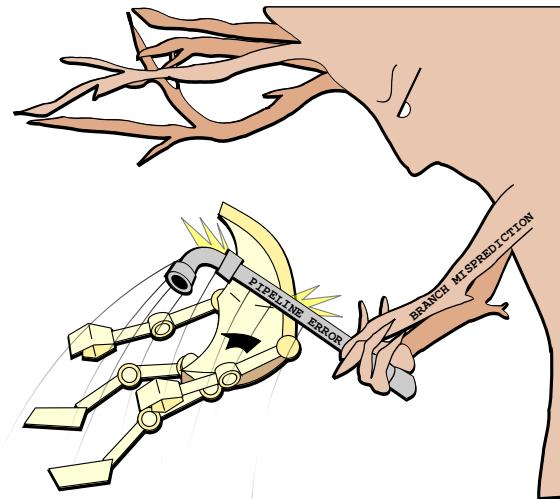


Figure 3.3: CPU Meets a Pipeline Flush

흐름이 필요합니다. 그런 제어 흐름은 예를 들어 거대한 행렬이나 벡터를 가지고 행해지는 계산과 같이 기본적으로 타이트한 반복문을 수행하는 프로그램에서 제공될 수 있습니다. 그럼 CPU는 이 루프의 끝에서의 브랜치가 거의 모든 경우 취해짐을 제대로 예측할 수 있어서, 파이프라인이 꽉 차게 만들고 CPU가 완전한 속도로 수행될 수 있게 할 수 있습니다.

하지만, 브랜치 예측은 항상 쉽지는 않습니다. 예를 들어, 프로그램이 여러 반복문을 포함하며 각각의 반복문은 작은 무작위 횟수만큼 반복되는 경우를 생각해 볼 수 있겠습니다. 또 다른 예로, 자주 호출되는 멤버 함수를 갖는 서로 다른 구현체를 갖는 여러 다른 실제 오브젝트를 참조하는 많은 가상 오브젝트를 가져서 결과적으로 많은 수의 포인터 기반 함수 호출을 하게 되는 고전적 객체 지향 프로그램을 생각해 봅시다. 이런 경우, CPU가 다음 브랜치는 어떻게 될지를 예측하기는 어려우며

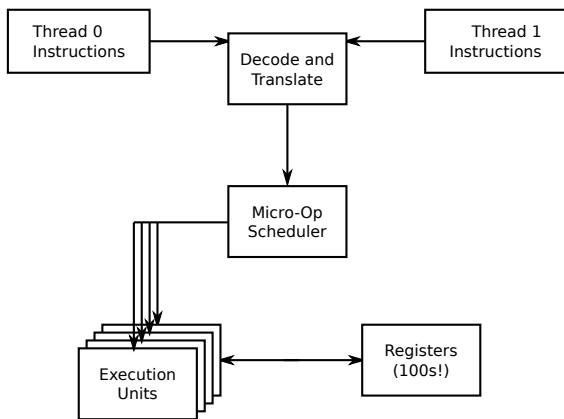


Figure 3.4: Rough View of Modern Micro-Architecture

불가능할 수도 있습니다. 그럼 CPU는 그 브랜치가 어느 방향을 향하게 될지 확신할 수 있을 때까지 수행이 진행될 때까지 기다리거나, 추측을 해보고 투기적 수행을 진행해야 합니다. 예측 가능한 제어 흐름을 갖는 프로그램에 대해서는 추측이 매우 잘 동작하지만, (바이너리 탐색 등의) 예측 불가능한 브랜치들에 대해서는 그 추측이 자주 틀립니다. 잘못된 추측은 그 비용이 비쌀 수 있는데, CPU는 해당 브랜치를 따라가서 투기적으로 수행된 모든 명령의 결과를 버려야 해서 파이프라인 비우기를 해야 하기 때문입니다. 파이프라인 비우기가 너무 자주 행해지면, Figure 3.3에 그려진 것처럼 전체 성능이 무척 감소됩니다.

이는 갈수록 더 흔해져가는 하이퍼쓰레딩 (또는, 여러분이 그 이름을 선호한다면, SMT)에서는 더 나빠지는데, 특히 투기적 수행을 하는 파이프라인 기반의 슈버스칼라 비순차 CPU에서는 더 그렇습니다. 점점 더 흔해지는 이와 같은 경우, 코어를 공유하는 모든 하드웨어 쓰레드는 이 코어의 레지스터, 캐시, 수행 유닛, 등등의 자원을 공유합니다. 명령은 종종 마이크로-오퍼레이션으로 해석되고, 공유된 수행 유닛과 수백개의 하드웨어 레지스터의 사용은 마이크로-오퍼레이션 스케줄러에 의해 조정됩니다. 그런 두개 쓰레드를 제공하는 코어에 대한 디어그램이 Figure 3.4에 그려져 있으며, 더 정확한 (그리고 따라서 더 복잡한) 디어그램은 교재와 학술 논문에 많이 있습니다.¹ 따라서, 한 하드웨어 쓰레드의 수행은 해당 코어를 공유하는 다른 하드웨어 쓰레드의 동작에 의해 자주 방해받을 수 있습니다.

오직 하나의 하드웨어 쓰레드만이 활동중이라도 (예를 들어, 단 하나의 쓰레드만 존재하는 고전적 CPU 설계의 경우), 반직관적인 결과는 상당히 흔합니다. 수행

¹ 2010년대 후반 인텔 코어를 위한 예 하나가 여기 있습니다: [https://en.wikichip.org/wiki/intel/microarchitectures/skylake_\(server\)](https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)).

유닛은 겹치는 능력을 가진 경우가 흔해서, CPU의 수행 유닛 선택이 다음 명령을 위한 해당 수행 유닛에 대한 경쟁이 파이프라인 지연을 야기할 수 있습니다. 이론적으로, 이 경쟁은 회피될 수 있습니다만, 실제로는 CPU는 천리안 없이도 이 선택을 매우 빨리 해야만 합니다. 특히, 타이트한 반복문에 명령을 추가하는 것은 간혹 수행을 더 빠르게 하는 경우도 있습니다.

불행히도, 파이프라인 비우기와 공유 자원 경쟁은 근대의 CPU가 맞닥뜨리는 장애물의 전부가 아닙니다. 다음 섹션은 메모리 참조에서의 문제를 다룹니다.

3.1.2 Memory References

1980년대에는 마이크로프로세서가 메모리로부터 값을 읽어오는데 걸리는 시간이 하나의 명령을 수행하는데 걸리는 시간보다 적은 경우가 많았습니다. 최근에는, 마이크로프로세서는 메모리를 액세스하는데 걸리는 시간 동안 수백, 또는 심지어 수천개의 명령을 수행할 수도 있습니다. 이 격차는 무어의 법칙이 메모리 반응속도의 감소보다 훨씬 큰 폭으로 CPU 성능을 향상시켰다는 사실 때문입니다. 이는 부분적으로는 메모리 크기 증가 비율 때문이기도 합니다. 예를 들어, 일반적인 1970년대 미니컴퓨터는 4 KB (그래요, 메가바이트가 아니라 킬로바이트) 메인 메모리를 가졌으며 액세스하는데 단 하나의 사이클이 필요했습니다.² 오늘날의 CPU 설계자들은 여전히 4 KB 메모리를 단일 사이클에 액세스 할 수 있도록 할 수도 있는데, 수 GHz 클락 주파수의 시스템에서도 그렇습니다. 그리고 실제로 그런 메모리를 자주 구성합니다만, 그들은 이제 그걸 “레벨-0 캐쉬”라 부르며, 그것들은 4 KB 보다도 상당히 클 수 있습니다.

현대 마이크로프로세서에서 찾을 수 있는 대용량 캐쉬가 메모리 접근 반응시간을 해결하는데 상당한 도움이 되지만, 이 캐쉬는 그 반응시간을 숨기는데 성공하기 위해 상당히 예측 가능한 데이터 액세스 패턴을 필요로 합니다. 불행히도, 링크드 리스트를 순회하는 것과 같은 일반적인 오퍼레이션들은 상당히 예측 불가능한 메모리 액세스 패턴을 갖습니다—어쨌건, 그 패턴이 예측 가능하다면, 우리 소프트웨어 측은 포인터를 가지고 고생하지도 않았을 겁니다, 그렇죠? 따라서, Figure 3.5에 보인 것과 같이, 메모리 참조는 현대 CPU에게 상당한 장애물을 야기합니다.

지금까지 우리는 CPU가 싱글쓰레드 코드를 수행할 때 마주칠 수 있는 장애물에 대해 생각해 봤습니다. 멀티쓰레딩은 CPU에 추가적인 장애물을 제공하는데, 다음 섹션들에서 설명합니다.

² 이 단 하나의 사이클이란 게 1.6마이크로세컨드 이상의 시간이었음을 말해두는게 공평하겠습니다.

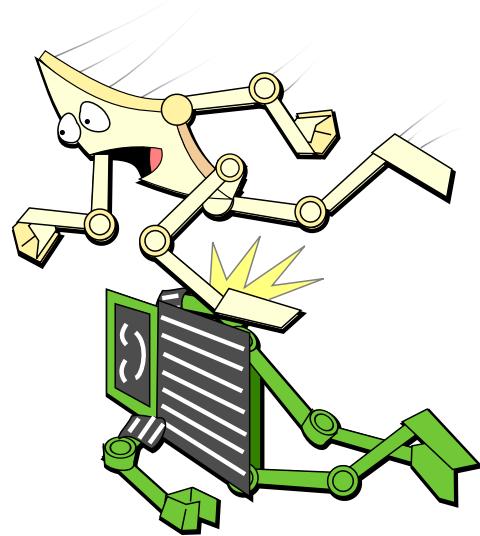


Figure 3.5: CPU Meets a Memory Reference

3.1.3 Atomic Operations

그런 장애물 중 하나는 어토믹 오퍼레이션들입니다. 여기서의 문제는 어토믹 오퍼레이션의 개념 자체가 CPU 파이프라인의 한번에 한 조각씩 조립해 나간다는 원칙과 충돌한다는 것입니다. 하드웨어 설계자들 덕분에, 현대의 CPU는 그런 오퍼레이션들이 실제로는 한번에 한조각씩만 실행됨에도 원자적으로 수행되는 것으로 보이게 하는 상당히 영리한 속임수를 사용하는데, 흔히 사용되는 속임수 중 하나는 원자적으로 처리하려 하는 데이터를 포함하고 있는 전체 캐시라인을 인식하고, 해당 캐시라인은 이 어토믹 오퍼레이션을 수행하고 있는 CPU에게 소유됨을 보장하고, 그런 상태에서만 이 어토믹 오퍼레이션을 진행하는 것입니다. 이 모든 데이터가 이 CPU에 사적으로 소유되기 때문에, 다른 CPU들은 CPU의 한번에 한조각씩 처리하는 파이프라인의 본성에도 불구하고 어토믹 오퍼레이션을 간섭할 수 없습니다. 말할 필요도 없이, 이런 종류의 속임수는 어토믹 오퍼레이션이 올바르게 완료될 수 있도록 하기 위해 이 셋업을 수행하기 위해 파이프라인이 자연되거나 심지어 비워져야만 할 것을 요구할 수도 있습니다.

반면, 비 어토믹 오퍼레이션을 수행할 때에는, CPU는 캐시라인 소유권을 기다릴 필요 없이 데이터가 나타나자마자 캐시 라인으로부터 값을 읽어들일 수 있으며, 수행 결과를 스토어 버퍼에 저장할 수 있습니다. 가끔 캐쉬 응답시간을 숨길 수 있는 다양한 하드웨어 최적화가 존재하긴 하지만, 이로 인한 성능에의 영향은 너무나도 종종 Figure 3.6에 보인 것과 같습니다.

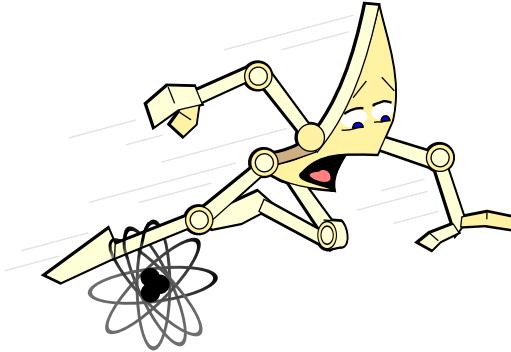


Figure 3.6: CPU Meets an Atomic Operation

불행히도, 어토믹 오퍼레이션은 데이터의 단일 원소에만 적용됩니다. 많은 병렬 알고리즘은 순서 규칙이 다수의 데이터 요소의 업데이트 중에도 유지될 것을 필요로 하기 때문에, 대부분의 CPU는 메모리 배리어를 제공합니다. 이 메모리 배리어는 또한 성능 제약으로 작용하는데, 다음 섹션에서 설명합니다.

Quick Quiz 3.2:

대체 어떤 기계가 다양한 데이터 원소에 대한 어토믹 오퍼레이션을 허용할 수 있죠?



3.1.4 Memory Barriers

메모리 배리어는 Chapter 15 와 Appendix C 에서 더 자세히 다뤄질 겁니다. 그 전까지, 다음의 간단한 락 기반의 크리티컬 섹션을 생각해 봅시다:

```

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

```

CPU 가 이 선언문들을 보여지는 순서대로 수행하도록 강제되지 않는다면, 그 영향으로 변수 “a”는 “mylock”의 보호 없이 증가될 것이어서, 락을 잡는 의미가 없어질 겁니다. 그런 파괴적 순서 재배치를 막기 위해, 락킹 도구들은 명시적이거나 암시적인 메모리 배리어를 내포합니다. 이런 메모리 배리어의 모든 목적은 성능을 증가시키기 위해 CPU 가 할 수도 있는 순서 재배치를 막기 위한 것인데, 메모리 배리어는 Figure 3.7 에 그린 것처럼 항상 성능을 감소시키기 때문입니다.

어토믹 오퍼레이션에서와 같이, CPU 설계자들은 메모리 배리어 오버헤드를 줄이기 위해 노력해왔고, 상당한 발전을 이뤘습니다.



Figure 3.7: CPU Meets a Memory Barrier

3.1.5 Cache Misses

CPU 성능에 대한 또 다른 멀티쓰레딩 장애물은 “캐쉬 미스”입니다. 앞에서도 언급되었듯, 현대의 CPU는 높은 메모리 응답시간 때문에 야기될 수 있는 성능 불이익을 줄이기 위해 큰 캐쉬를 갖습니다. 하지만, 이런 캐쉬는 CPU 간에 자주 공유되는 변수를 위해서는 반-생산적입니다. 이는 어느 CPU 가 해당 변수를 수정하고자 할 때 다른 CPU 가 그것을 최근에 수정했을 가능성이 높기 때문입니다. 이 경우, 그 변수는 지금 수정하고자 하는 CPU 가 아니라 앞서 그것을 고친 CPU 의 캐쉬에 있을 것이며, 이는 비용이 높은 캐쉬 미스를 일으키게 됩니다 (더 자세한 내용을 위해서는 Section C.1 을 읽어주시기 바랍니다). 그런 캐쉬 미스는 Figure 3.8 에 보인 것처럼 CPU 성능에 주요 장애물이 됩니다.

Quick Quiz 3.3: 그래서 CPU 설계자들은 캐쉬 미스의 오버헤드도 많이 줄였나요?



3.1.6 I/O Operations

캐쉬 미스는 CPU 간 I/O 오퍼레이션으로 생각될 수 있으며, 따라서 있을 수 있는, 가장 비용 저렴한 I/O 오퍼레이션 중 하나입니다. 네트워킹, 대용량 저장장치, 또는 (더 나쁜) 사람이 연관되는 I/O 오퍼레이션은 Figure 3.9 에 그려진 것처럼 앞의 섹션에서 이야기된 내부 장애물들보다 훨씬 큰 장애를 일으킵니다.

이는 공유메모리와 분산시스템 병렬성 사이의 차이 중 하나입니다: 공유메모리 병렬 프로그램은 보통 캐쉬

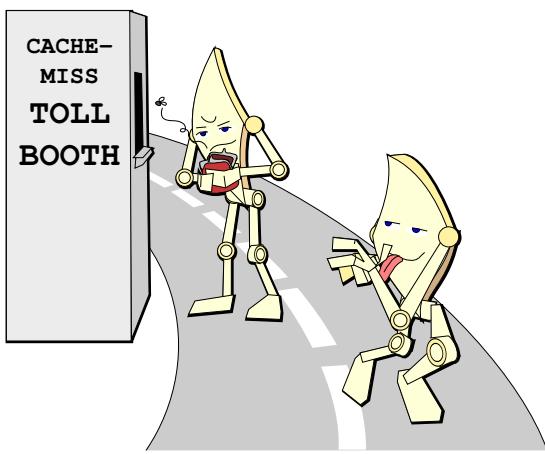


Figure 3.8: CPU Meets a Cache Miss



Figure 3.9: CPU Waits for I/O Completion

미스보다 심한 장애는 다룰 필요가 없지만, 분산 병렬 프로그램은 보통 거대한 네트워크 통신 응답시간을 겪게 됩니다. 두 경우 모두, 발생하는 응답시간은 통신의 비용으로 생각될 수 있습니다—순차적 프로그램에서는 존재하지 않을 비용. 따라서, 통신의 오버헤드와 수행되는 실제 일의 양 사이의 비율이 핵심 설계 인자입니다. 병렬 하드웨어 설계의 주요 목표는 이 비율을 적합한 성능과 확장성 목표를 이루기에 필요한 수준까지 낮추는 것입니다. Chapter 6에서 이야기 되겠지만, 병렬 소프트웨어 설계의 주요 목표는 소통과 캐시 미스 같은 비용이 높은 오퍼레이션들의 빈도를 줄이는 것입니다.

물론, 어떤 오퍼레이션이 장애물이라고 하는 것과 그 오퍼레이션이 심각한 장애물이라고 하는 건 다른 이야기입니다. 이 차이를 다음 섹션들에서 이야기 합니다.

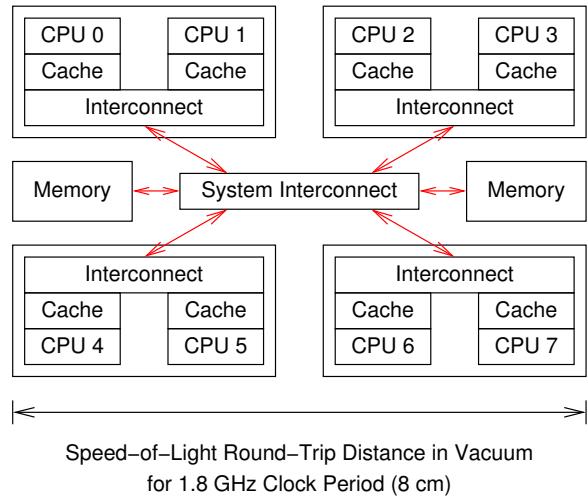


Figure 3.10: System Hardware Architecture

3.2 Overheads

Don't design bridges in ignorance of materials, and don't design low-level software in ignorance of the underlying hardware.

Unknown

이 섹션은 앞의 섹션에 소개된 성능 장애물들의 실제 오버헤드를 보입니다. 하지만, 먼저 하드웨어 시스템 구조의 대략적인 모습을 알아볼 필요가 있는데, 이를 다음 섹션에서 다룹니다.

3.2.1 Hardware System Architecture

Figure 3.10 는 여덟개 코어를 가진 컴퓨터 시스템의 대략적 모습을 보입니다. 각 쌍은 한쌍의 CPU 코어를 가지는데, 각 쌍은 캐시를 가지며, 이 한쌍의 CPU들이 서로간에 통신을 할 수 있게 하는 인터컨넥트 (interconnect) 역시 있습니다. 시스템 인터컨넥트 (system interconnect)는 이 네개의 쌍들이 서로간에, 그리고 메인 메모리와 통신할 수 있게 합니다.

데이터는 “캐시 라인” 단위로 이 시스템 내에서 이동하는데, 이는 2의 n 승의 고정된 크기로 정렬된 메모리 블록이며, 일반적으로 그 크기는 32에서 256 바이트 사이입니다. CPU가 메모리로부터 레지스터로 변수를 로드할 때에는 그 변수를 담고 있는 캐시라인을 자신의 캐시로 먼저 읽어야 합니다. 비슷하게, CPU가 변수를 자신의 레지스터로부터 메모리로 저장할 때에도 해당 변수를 담고 있는 캐시라인을 자신의 캐시로 읽어야

합니다만, 또한 다른 CPU 가 그 캐시라인의 복사본을 가지고 있지 않음을 보장해야 합니다.

예를 들어, CPU 0 이 CPU 7 의 캐시에 있는 캐시라인의 변수에 쓰기를 하려 했다면, 다음과 같은 무척 단순화된 일들이 순차적으로 벌어질 수 있습니다:

1. CPU 0 이 자신의 캐시를 검사하고, 해당 캐시라인이 거기 없음을 알아차립니다. 따라서 이 쓰기를 자신의 스토어 버퍼 (store buffer) 에 기록합니다.
2. 이 캐시라인에 대한 요청이 CPU 0 과 1 의 인터컨넥트로 전달되는데, 이 요청은 CPU 1 의 캐시를 검사하고 거기서도 이 캐시라인을 찾지 못합니다.
3. 이 요청은 이제 시스템 인터컨넥트로 전달되어, 다른 세개의 다이를 모두 검사하고, 이 캐시라인이 CPU 6 와 7 을 가지고 있는 다이 안에 있음을 알게 됩니다.
4. 이 요청은 이제 CPU 6 과 7 의 인터컨넥트로 전달되어, 두 CPU 의 캐시를 검사하고, 이 값이 CPU 7 의 캐시에 있음을 발견하게 됩니다.
5. CPU 7 은 이 캐시라인을 자신의 인터컨넥트로 전달하고, 자신의 캐시로부터 이 캐시라인을 제거합니다.
6. CPU 6 과 7 의 인터컨넥트는 이 캐시라인을 시스템 인터컨넥트로 전달합니다.
7. 시스템 인터컨넥트는 이 캐시라인을 CPU 0 과 1 의 인터컨넥트로 전달합니다.
8. CPU 0 과 1 의 인터컨넥트는 이 캐시라인을 CPU 0 의 캐시로 전달합니다.
9. CPU 0 은 이제 이 쓰기를 완료하고, 새로 도착한 이 캐시라인의 연관된 부분을 스토어 버퍼에 기록된 기존 값으로부터 업데이트합니다.

Quick Quiz 3.4: 이게 단순화된 거라구요? 어떻게 이보다 더 복잡한 일이 가능한가요?



Quick Quiz 3.5: 왜 이 캐시라인을 CPU 7 의 캐시로부터 제거해야 하나요?



이 단순화된 이벤트 나열은 캐시 일관성 프로토콜 (*cache-coherency protocols*) [HP95, CSG99, MHS12, SHW11] 이라 불리는 규칙의 시작에 불과한데, 이는 Appendix C 에서 더 자세하게 다뤄집니다. CAS 오퍼레이션에 의해 일어나는 이벤트들로부터 볼 수 있듯이,

하나의 명령이 상당한 프로토콜 트래픽을 야기할 수 있으며, 이는 여러분의 병렬 프로그램의 성능을 심각하게 하락시킬 수 있습니다.

다행히도, 어떤 변수가 업데이트는 없이 특정 기간동안 빈번하게 읽혀지기만 한다면, 해당 변수는 모든 CPU 의 캐시에 복사될 수 있습니다. 이 복사는 모든 CPU 가 이 읽기가 대부분인 변수로의 매우 빠른 액세스를 할 수 있게 합니다. Chapter 9 이 중요한 하드웨어 기반의 읽기가 대부분인 경우를 위한 최적화의 장점을 온전히 취하기 위한 동기화 메커니즘을 소개합니다.

3.2.2 Costs of Operations

병렬 프로그램에 중요한 일부 일반적 오퍼레이션들의 오버헤드가 Table 3.1 에 표시되어 있습니다. 이 시스템의 클락 기간은 대략 0.5 ns 입니다. 현대의 마이크로프로세서가 클락 기간당 여러 명령을 처리할 수 있는 건 흔하지만, 오퍼레이션의 비용은 “Ratio” 라고 라벨링 된 세번째 행에 클락 기간으로 얼마큼의 비율인지 표시되어 있습니다. 이 표에 대해 말씀드릴 첫번째 것은 그런 비율 중 큰 값이 여럿 있다는 것입니다.

같은 CPU 에서의 compare-and-swap (CAS) 오퍼레이션은 대략 7 나노세컨드를 소비하는데, 이는 클락 기간의 10배가 넘는 기간입니다. CAS 는 하드웨어가 특정 메모리 위치의 내용을 특정 “기준” 값과 비교하고, 그것들이 동일하다면 특정한 “새로운” 값을 저장하는, 즉 CAS 오퍼레이션이 성공하는, 원자적 오퍼레이션입니다. 만약 그 값들이 동일하지 않았다면, 해당 메모리 위치는 그 (예상되지 않았던) 값을 유지하게 되고, CAS 오퍼레이션은 실패한 게 됩니다. 해당 메모리 위치의 값은 이 비교와 저장 사이에 바뀌지 않음을 하드웨어가 보장하므로 이 오퍼레이션은 원자적입니다. CAS 의 기능은 x86 에서는 `lock; cmpxchg` 로 제공됩니다.

“same-CPU” 접두어는 특정 변수에 CAS 오퍼레이션을 수행하는 CPU 가 이 변수를 마지막으로 액세스한 CPU 임을, 따라서 여기 연관된 캐시라인이 이 CPU 의 캐시에 있음을 의미합니다. 비슷하게, same-CPU lock 오퍼레이션 (락 획득과 해제 한쌍의 “왕복” 작업) 은 15 나노세컨드, 달리 말하면 30 클락 사이클을 소비합니다. 이 락 오퍼레이션은 락 데이터 구조에 한번은 획득을 위해, 또한 한번은 해제를 위해, 두번의 어토믹 오퍼레이션을 할 것을 필요로 하기 때문에 CAS 보다 비용이 높습니다.

하나의 코어를 공유하는 하드웨어 쓰레드간의 상호 작용을 하게 되는 in-core 오퍼레이션은 same-CPU 오퍼레이션과 거의 같습니다. 이 두개의 하드웨어 쓰레드는 전체 캐시 구조를 공유함을 생각하면 이는 크게 놀랍진 않을 겁니다.

Blind CAS 의 경우, 소프트웨어는 메모리 위치를 고려 치 않은 채 기존 값을 특정합니다. 이는 락을 획득하고자 하는 시도에 적합합니다. 락이 잡히지 않은 상태가 값

Table 3.1: CPU 0 View of Synchronization Mechanisms on 8-Socket System With Intel Xeon Platinum 8176 CPUs @ 2.10 GHz

Operation	Cost (ns)	Ratio (cost/clock)	CPUs
Clock period	0.5	1.0	
Same-CPU CAS	7.0	14.6	0
Same-CPU lock	15.4	32.3	0
In-core blind CAS	7.2	15.2	224
In-core CAS	18.0	37.7	224
Off-core blind CAS	47.5	99.8	1–27,225–251
Off-core CAS	101.9	214.0	1–27,225–251
Off-socket blind CAS	148.8	312.5	28–111,252–335
Off-socket CAS	442.9	930.1	28–111,252–335
Cross-interconnect blind CAS	336.6	706.8	112–223,336–447
Cross-interconnect CAS	944.8	1,984.2	112–223,336–447
Off-System			
Comms Fabric	5,000	10,500	
Global Comms	195,000,000	409,500,000	

0 으로 표현되고 락이 잡힌 상태는 값 1로 표현된다면, 0 을 기준값으로, 1 을 새 값으로 하는, 이 락에의 CAS 오퍼레이션은 이 락이 아직 잡히지 않은 상태라면 획득하게 됩니다. 핵심은 이 메모리 위치로의 액세스는 오직 하나, 이 CAS 오퍼레이션이 있다는 겁니다.

반면, 이 평범한 CAS 오퍼레이션의 기준값은 앞서 행해진 어떤 로드로부터 얻어집니다. 예를 들어, 어떤 어토믹 증가 오퍼레이션을 구현하기 위해서는, 해당 위치의 현재 값이 읽혀지고, 새 값을 만들기 위해 그 값을 증가시킵니다. 그리고 나서는 이 CAS 오퍼레이션에 앞서 읽혀진 값이 기준값으로, 이 증가된 값은 새 값으로 명세됩니다. 이 값이 이 읽기와 CAS 사이에 변경되지 않았다면, 이는 해당 메모리 위치의 값을 증가시킬 겁니다. 하지만, 이 값이 바뀌었다면, 기존 값이 다음을 알게 되어서, 이 CAS 오퍼레이션이 실패하게 할겁니다. 핵심은 이제 해당 메모리 위치로의 두개의 액세스, 즉 읽기와 CAS 가 존재한다는 겁니다.

따라서, in-core blind CAS 는 대략 7 나노세컨드만 소비하는 반면 in-core CAS 는 약 18 나노세컨드를 소비한다는 게 놀랍지 않습니다. 이 blind 가 아닌 경우의 추가적인 로드는 공짜로 오지 않습니다. 그렇다면 하나, 이 오퍼레이션들의 오버헤드는 단일 CPU 의 CAS 와 락에 각각 비슷합니다.

Quick Quiz 3.6: Table 3.1 는 CPU 0 가 CPU 224 와 코어를 공유한다고 하는데요. 그건 CPU 1 이 되어야 하는거 아닌가요???



다른 코어에 있지만 같은 소켓에 위치한 CPU 들이 연관되는 blind CAS 는 대략 50 나노세컨드, 달리 말하면 약 100 클락 사이클을 소모합니다. 이 캐쉬미스 측정을 위해 사용된 코드는 해당 캐쉬라인을 한쌍의 CPU 사이에서 주고받게 하므로, 이 캐쉬 미스는 메모리에서가 아니라 다른 CPU 의 캐쉬에서 이뤄집니다. 앞서 이야기 되었듯 변수의 기준 값을 보고 새 값을 저장도 해야 하는 non-blind CAS 오퍼레이션은 약 100 나노세컨드, 즉 약 200 클락 사이클을 소모합니다. 이를 좀 더 생각해 봅시다. CAS 오퍼레이션 하나를 위해 필요한 시간동안, 이 CPU 는 200개의 평범한 명령을 수행했을 수도 있었습니다. 이는 fine-grained 락킹만이 아니라 모든 다른 동기화 메커니즘이 잘게 쪼개진 전역적 동의사항에 의존하고 있음을 보일겁니다.

이 한쌍의 CPU 가 서로 다른 소켓에 위치해 있다면, 이 오퍼레이션들은 상당히 더 비싸집니다. 하나의 blind CAS 오퍼레이션은 대략 150 나노세컨드, 즉 300 클락 사이클 이상을 소모합니다. 일반적인 CAS 오퍼레이션은 400 나노세컨드, 즉 거의 1000 클락 사이클 이상을 소모합니다.

더 나쁜 것이, 모든 소켓 쌍이 똑같이 만들어지지는 않습니다. 이 시스템은 네개짜리 소켓 컴포넌트의 한쌍으로 구성되어 있는데, 서로 다른 컴포넌트에 위치한 CPU 사이에는 추가적인 응답시간 페널티가 존재합니다. 이 경우, 하나의 blind CAS 오퍼레이션은 300 나노세컨드 이상, 즉 700 클락 사이클 이상을 소모합니다. 하나의 CAS 오퍼레이션은 거의 1 마이크로세컨드, 즉 거의 2000 클락 사이클을 소모합니다.

Table 3.2: Cache Geometry for 8-Socket System With Intel Xeon Platinum 8176 CPUs @ 2.10 GHz

Level	Scope	Line Size	Sets	Ways	Size
L0	Core	64	64	8	32K
L1	Core	64	64	8	32K
L2	Core	64	1024	16	1024K
L3	Socket	64	57,344	11	39,424K

Quick Quiz 3.7: 분명 하드웨어 설계자들은 이 상황을 개선하려 노력했을 수 있을 거예요! 왜 그들은 이 단일 명령 오퍼레이션의 끔찍한 성능에 만족하고 있는 거죠?

■
불행히도, 코어 내에서와 소켓 내에서의 통신의 빠른 속도는 공짜로 오지 않습니다. 첫째로, 하나의 코어 내에는 두개의 CPU만 있으며 하나의 소켓 내에는 56개의 코어만 있는 반면, 이 시스템 전체에는 448개의 코어가 있습니다. 둘째로, Table 3.2에 보인 것처럼, 코어 내의 캐시는 소켓 내 캐시에 비해 상당히 작으며, 소켓 내 캐시 역시 시스템 전체에 구성된 1.4 TB의 메모리에 비하면 상당히 작습니다. 셋째로, 이 그림에 따르면, 캐시들은 버킷당 아이템의 수가 제한된 하드웨어 해쉬테이블로 구성되어 있습니다. 예를 들어, L3 캐시의 크기 (“Size”)는 대략 40 MB입니다만, 각 버켓 (“Line”)은 11개의 메모리 블락만 (“Ways”) 가질 수 있으며, 각 블락은 최대 64 바이트가 될 수 있습니다 (“Line Size”). 이 말은 이 40 MB 캐시를 오버플로우 시키는데 12 바이트의 메모리만으로도 (물론 주의 깊게 골라진 주소들의 것으로) 충분하다는 말입니다. 반면에, 역시 주의 깊게 골라진 주소들을 사용하면 전체 40 MB를 잘 사용할 수도 있습니다.

참조의 공간적 지역성은 데이터를 메모리 전역에 퍼트리는 것 만큼이나 무척 중요합니다.

I/O 오퍼레이션은 심지어 더 비쌉니다. “Comms Fabric” 행에 보여진 것처럼, InfiniBand나 다른 독점적 인터 커넥트들과 같은 고성능의 (그리고 비싼!) 통신 장비는 단말간 왕복에 대략 5 마이크로세컨드의 응답시간을 갖는데, 이는 만개가 넘는 명령이 수행될 수도 있는 시간입니다. 표준 기반의 통신 네트워크는 종종 일종의 프로토콜 처리를 필요로 해서, 응답시간을 더 늘립니다. 물론, 지리적 거리 역시 응답시간을 늘리는데, 광섬유를 사용해 광속으로 지구를 한바퀴 드는데 걸리는 시간은 “Global Comms” 행에 보인 것처럼 대략 195 밀리세컨드로, 달리 말하면 4억 클락 사이클이 넘습니다.

Quick Quiz 3.8: 이 숫자들은 미친듯이 크군요! 이걸 어떻게 제 머리로 이해할 수 있을까요?

■

3.2.3 Hardware Optimizations

하드웨어가 어떻게 도움을 주는지 묻는건 자연스러운 일입니다, 그리고 그에 대한 대답은 “무척!”입니다.

그런 하드웨어 최적화 중 하나는 큰 캐쉬라인입니다. 이는 특히 소프트웨어가 메모리를 순차적으로 액세스할 때 큰 성능 향상을 제공합니다. 예를 들어, 64 바이트 캐쉬라인이 있고 소프트웨어가 64비트 변수들을 접근한다면, 첫번째 액세스는 빛의 속도에 의한 (그 외의 것들이 없다면) 지연 때문에 여전히 느리겠지만, 뒤따르는 일곱번의 액세스는 무척 빠를 수 있습니다. 하지만, 이 최적화는 이른바 false sharing이라 불리는, 같은 캐쉬라인에 있는 다른 변수들이 다른 CPU에 의해 업데이트되어 높은 캐쉬미스율을 초래하는 어두운 부분을 가지고 있습니다. 소프트웨어는 false sharing을 방지하기 위해 많은 컴파일러에서 사용 가능한 정렬 지시어를 사용할 수 있으며, 그런 지시어를 추가하는 것은 병렬 소프트웨어 튜닝에서 흔한 단계입니다.

관련된 두번째 하드웨어 최적화는 캐쉬 prefetching으로, 연속적인 액세스에 대해 뒤쪽 캐쉬라인을 미리 읽어들여오는 식으로 하드웨어가 반응함으로써 뒤따르는 캐쉬라인들에 대한 빛의 속도에 의한 지연을 막습니다. 물론, 하드웨어는 언제 미리 읽기를 할지 판단하기 위해 간단한 휴리스틱을 사용해야만 하며, 이 휴리스틱은 많은 어플리케이션이 갖는 복잡한 데이터 액세스 패턴에 의해 바보같아질 수 있습니다. 다행히도, 일부 CPU 제품군은 특수한 미리읽기 명령을 제공함으로써 이를 극복 가능하게 합니다. 불행히도, 일반적인 경우에서의 이 명령의 효과는 실망적입니다.

세번째 하드웨어 최적화는 store buffer로, 일련의 스토어 명령이 그 스토어 명령들이 연속적이지 않은 주소를 향하더라도, 그리고 이 스토어 명령들에 필요한 캐쉬라인들이 해당 CPU의 캐쉬에 존재하지 않은 경우에도 빠르게 수행되게 합니다. 이 최적화의 어두운 부분은 메모리 순서 오류인데, Chapter 15를 참고하시기 바랍니다.

네번째 하드웨어 최적화는 투기적 수행(speculative execution)으로, 하드웨어가 메모리 순서 오류를 초래하지 않고 이 스토어 버퍼를 잘 사용할 수 있게 해줍니다. 이 최적화의 어두운 부분은 이 투기적 수행이 뒤틀어지고 수행 결과가 되돌이켜지고 재수행되어야 할 때 발생하는 에너지 비효율성과 성능 하락입니다. 더 나쁜 것이, Spectre와 Meltdown [Hor18]의 발견은 하드웨어의 투기적 수행이 메모리 보호 하드웨어를 깨부수는 사이드 채널 공격을 가능하게 해서 비특권 프로세스가 그들은 액세스하면 안되는 메모리를 읽을 수 있게 할 수 있습니다. 투기적 수행과 클라우드 컴퓨팅의 조합은 상당한 재작업이 필요함은 분명합니다!

다섯번째 하드웨어 최적화는 커다란 캐쉬로, 개별 CPU가 비싼 캐쉬 미스를 일으키지 않고도 큰 데이터셋

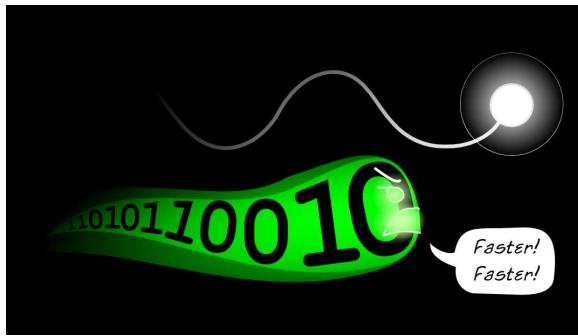


Figure 3.11: Hardware and Software: On Same Side

을 가지고 동작할 수 있게 합니다. 커다란 캐쉬는 에너지 효율성과 캐쉬 미스 응답시간을 나쁘게 만들 수 있지만, 마이크로프로세서 제품들의 지속적인 캐쉬 크기 증가 경향은 이 최적화의 힘을 증명합니다.

마지막 하드웨어 최적화는 읽기가 대부분인 경우를 위한 복사로, 자주 읽히지만 가끔씩만 업데이트 되는 데이터는 모든 CPU 의 캐쉬에 존재하게 하는 겁니다. 이 최적화는 읽기가 대부분인 데이터는 무척 효율적으로 액세스 될 수 있게 하며, Chapter 9 의 주제입니다.

요약하자면, 하드웨어와 소프트웨어 엔지니어들은 정말로 같은 쪽에 있으며, 둘 다 물질의 법칙의 최대의 노력에도 불구하고 컴퓨터를 더 빠르게 하기 위해 Figure 3.11 에 우리의 데이터 흐름이 빛의 속도를 넘기기 위해 노력하는 모습으로 그려진 것처럼 노력하고 있습니다. 다음 섹션은 최근의 연구가 실제 세계에 어떤 모습의 결과를 내게 되느냐에 따라 하드웨어 엔지니어들이 할 수도 (또는 하지 못할 수도) 있는, 추가적인 것들을 이야기 합니다. 이 훌륭한 목표를 향한 소프트웨어의 기여는 이 책의 나머지 챕터들에 있습니다.

3.3 Hardware Free Lunch?

The great trouble today is that there are too many people looking for someone else to do something for them. The solution to most of our troubles is to be found in everyone doing something for themselves.

Henry Ford, updated

지난 몇년간 동시성이 그렇게 많은 주목을 받게 된 주요한 이유는 page 9 의 Figure 2.1 에도 보여져 있는, 무어의 법칙으로 인한 단일쓰레드 성능 증가 (또는 “공짜 점심-free lunch-” [Sut08]) 의 종료입니다. 이 섹션은 하드웨어 설계자들이 이 “공짜 점심”을 다시 가져올 수 있는 몇가지 방법에 대해 간단히 알아봅니다.

하지만, 앞의 섹션은 동시성을 노출시키는데 대한 상당한 하드웨어 상의 장애물 몇가지를 보였습니다. 하드웨어 설계자들이 마주치는 상당한 물리적 한계 중 하나는 제한된 빛의 속도입니다. Page 21 의 Figure 3.10에서도 이야기 되었듯, 빛은 1.8 GHz 클락 기간동안 진공관 내에서 8-센티미터 거리밖에 왕복하지 못합니다. 이는 5 GHz 클락에서는 3 센티미터로 줄어듭니다. 이 거리들 둘 다 현대 컴퓨터 시스템의 크기에 비교해 상대적으로 작습니다.

더 나빠질 여지가 있는 것이, 반도체 내에서의 전자기파는 빛이 진공 속에서 그런 것에 비해 세배에서 서른배 까지 느리게 움직이며, 일반적인 클락 기반의 하드웨어 구조는 이를 더 느리게 만드는데, 예를 들어 하나의 메모리 참조는 이 요청이 시스템의 나머지 부분에 넘겨지기 전까지 로컬 캐쉬 탐색이 완료되길 기다려야 할 수 있습니다. 더 나아가, 예를 들어 CPU 와 메인 메모리 간의 통신의 경우 전자신호를 하나의 반도체에서 다음 것으로 이동시키기 위해 상대적으로 낮은 스피드와 높은 전력의 드라이버가 필요합니다.

Quick Quiz 3.9: 하지만 전자 각각은 그렇게 빠르게 전혀 움직이지 못해요, 컨덕터 (conductor) 내에서도요!!! 세미컨덕터 전압 수준에서의 컨덕터 내 전자의 속도는 초당 오직 밀리미터 수준이라고요. 어떻게 된거죠???

■

하지만 이것들을 개선시킬 기술이 (하드웨어적으로도 소프트웨어적으로도) 몇가지 있습니다.

1. 3D 통합,
2. 첨단의 물질과 프로세스,
3. 빛으로 전자기를 대체하기,
4. 특수 목적 가속기, 그리고
5. 혼존하는 병렬 소프트웨어.

이것들 각각을 다음 섹션들에서 다룹니다.

3.3.1 3D Integration

3차원 통합 (3DI) 는 매우 얇은 실리콘 다이들을 수직으로 쌓아 붙이는 것입니다. 이는 잠재적 이득을 제공하지만, 또한 심각한 제작 상의 도전을 가져옵니다 [Kni08].

3DI 의 가장 중요한 이득은 아마도 Figure 3.12 에 보인 것처럼 시스템 전체를 관통하는 길이의 단축입니다. 3-센티미터 실리콘 다이가 네개의 1.5-센티미터 다이의 더미로 변경되면, 각 층의 두께는 무척 얇다는 점을 고려하면 이론상 시스템 전체 관통 최대 거리가 두배로 줄어들게 됩니다. 또한, 설계와 배치에 충분한 주의를

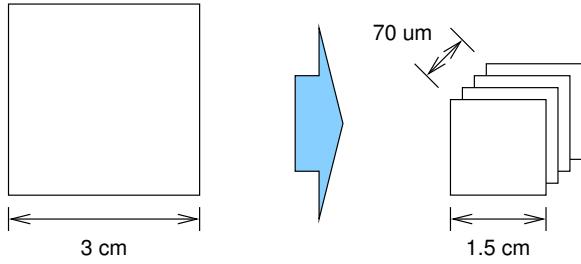


Figure 3.12: Latency Benefit of 3D Integration

기울인다면, 길다란 수평적 (느리고 전력을 많이 소모하는) 전자기적 연결은 더 짧고 전력을 덜 소모하는, 짧은 수직적 전자기적 연결로 교체될 수 있습니다.

하지만, 클락 기반 논리회로 때문에 발생하는 지연은 3D 통합으로도 줄어들지 않으며, 그 예상되는 이득을 그대로 가져가면서 제품단까지 3D 통합이 사용되기 전에 상당한 제조, 테스팅, 전력 공급, 그리고 열 처리 상의 문제가 해결되어야 합니다. 열처리 문제는 다이아몬드 기반의 반도체를 사용해 해결될 수도 있는데, 이는 열 문제에 있어선 좋은 도체이지만 전자 절연체로 써는 별로입니다. 그렇지만, 커다란 하나의 다이아몬드 덩어리를 얻기는 여전히 어려우며, 그걸 웨이퍼로 잘라내는 것의 어려움은 말할 필요도 없습니다. 또한, 이 기술 중 어느 것도 몇몇 사람들이 받은 정도를 넘어서 폭발적 증가를 제공할 수 있지는 않을 것 같습니다. 그렇다면 하나, 최근의 Jim Gray의 “smoking hairy golf balls” [Gra02]로의 여정에는 필요한 단계가 될 수 있습니다.

3.3.2 Novel Materials and Processes

Stephen Hawking은 반도체 제조사들이 두 가지 기본적 문제를 가지고 있다고 이야기 했다고 알려져 있습니다: (1) 한정된 빛의 속도와 (2) 물질의 원자적 본성 [Gar07]. 반도체 제조사들이 이런 한계를 해결하려 노력하고 있을 수 있지만, 이 근본적 한계를 회피하는데 주목하는 연구와 개발 노력도 있습니다.

원자적 본성을 회피하기 위한 한 가지 노력은 “high-K dielectric” 물질이라 불리는 것으로, 커다란 기기들이 있을 수 없을 만큼 작은 기기들의 전자적 특성을 흉내내게 하는 것입니다. 이런 물질들은 심각한 제조상의 도전을 갖지만, 첨단을 조금 더 앞으로 나아가게 하는 것을 도울 수도 있을 겁니다. 또 다른 더 색다른 방법은 여러 비트를 하나의 전자에 저장하는 것으로 특정 전자가 여러 수준의 에너지를 가지고 존재할 수 있다는 사실에 기반합니다. 이 특정 시도가 제품 수준의 반도체 기기에서 안정적으로 동작할 수 있는지는 두고봐야 합니다.

제안된 또다른 방법은 “quantum dot” 방법으로 훨씬 더 작은 크기의 기기가 가능하게 하지만, 여전히 연구 단계에 있습니다.

여기서의 한 가지 도전사항은 최근의 많은 하드웨어 기기 수준 해결책은 어떤 원자가 어디에 위치했는지 매우 꼼꼼하게 제어해야 한다는 것입니다 [Kell17]. 따라서 누구든 원자들을 칩 위의 수십억개의 기기 위에 하나씩 위치시킬 수 있는 좋은 방법을 찾는 사람은 가장 훌륭한 자랑할 권한을 가질 겁니다, 다른게 없다면요!

3.3.3 Light, Not Electrons

빛의 속도가 강한 제약이 되긴 하겠지만, 진실은 반도체 기기들은 빛의 속도가 아니라 전자기의 속도에 의해 제약되며, 반도체 물질 내에서의 전자기파는 진공에서의 빛의 속도의 3%에서 30% 정도로 움직인다는 것입니다. 실리콘 기기들 사이에 구리로 만들어진 연결부위를 사용하는 것은 전자기의 속도를 증가시키는 한 방법이며, 실제 빛의 속도까지 이를 나아가게끔 하는 추가적인 발전도 상당히 가능할 겁니다. 또한, 유리 내에서의 빛의 속도는 진공에서의 빛의 속도의 60%를 넘는다는 사실에 기반한, 칩 내에서 그리고 칩간 연결을 위해 작은 광섬유를 사용하는 실험이 있었습니다. 그런 광섬유 사용을 막는 문제는 전자기와 빛 사이, 그리고 그 반대의 변환에서의 비효율성으로, 이는 전력 소모와 열 처리 문제를 유발합니다.

그러나, 물리 분야에서의 근본적 발전이 있지 않고서는 데이터 흐름의 어떤 폭발적 증가도 진공에서의 빛의 속도에 제한될 겁니다.

3.3.4 Special-Purpose Accelerators

특수한 문제를 위해 일하는 범용 CPU는 종종 상당한 시간과 전력을 해당 문제에 핵심적이지는 않은 부분을 처리하는데 사용합니다. 예를 들어, 두개의 벡터의 내적을 구할 때, 범용 CPU는 일반적으로 (루프 언롤링 - loop unrolling - 이 적용되지 않은) 루프 카운터를 가지고 루프를 돌릴 겁니다. 이 명령들을 디코딩하고 루프 카운터를 증가시키고 이 카운터를 검사하고 이 루프의 시작지점으로 수행 흐름을 되돌리는 것은 어떤 면에서 불필요한 노력입니다: 진짜 목표는 이 두개의 벡터의 연관된 원소들을 곱하는 것입니다. 따라서, 벡터들을 곱하기 위해서만 설계된 하드웨어의 특수한 부분은 더 적은 전력을 사용하며 이 일을 더 빨리 끝낼 수 있습니다.

이는 실제로 많은 상용 마이크로프로세서에 존재하는 벡터 명령의 동기 부여가 되었습니다. 이 명령들은 여러 데이터 아이템을 동시적으로 처리하기 때문에, 내적 계산을 더 적은 명령 디코딩과 루프 오버헤드를 가지고 처리할 수 있게 합니다.

비슷하게, 특수화된 하드웨어는 암호화와 복호화, 압축과 압축 해제, 인코딩과 디코딩, 그리고 그 외에도 많은 다른 작업들을 더 효율적으로 처리할 수 있습니다. 불행히도, 이 효율성은 공짜로 오지 않습니다. 이 특수 하드웨어를 가진 컴퓨터 시스템은 더 많은 트랜지스터를 가질 것인데, 이는 실제 사용되지 않고 있을 때에도 전력을 조금 소비하게 됩니다. 소프트웨어는 이 특수화된 하드웨어의 장점을 취하기 위해 수정되어야만 하며, 이 특수화된 하드웨어는 해당 하드웨어를 위한 설계 비용이 많은 사용자들에게 나뉘어져 그 가격이 구매할 만한 낮은 가격이 될 수 있게끔 충분히 범용적이어야만 합니다. 부분적으로는 이런 종류의 경제적 고려사항 때문에 특수화된 하드웨어는 지금까지는 일부 어플리케이션 영역에서만 사용되어왔는데, 이는 그래픽 처리 (GPU), 배터 처리 (MMS, SSE, 그리고 VMX 명령들), 그리고, 더 적은 정도로, 암호화 정도에만 사용되어왔습니다. 심지어 이 영역에서조차 예상한 만큼의 성능 증가를 얻기는 항상 쉽지만은 않은데, 예를 들어 열처리 문제 등 때문입니다 [Kra17, Lem18, Dow20].

서버와 PC 쪽과 달리, 스마트폰은 다양한 하드웨어 가속기를 오랫동안 사용해 왔습니다. 이 하드웨어 가속기는 종종 첨단 MP3 플레이어가 CPU는 완전히 꺼져있는 상태에서도 음악을 수분간 재생할 수도 있도록 미디어 디코딩에 종종 사용되었습니다. 이런 가속기들의 목표는 전력 효율을 개선시키고 배터리 수명을 늘리기 위함이었습니다: 특수 목적 하드웨어는 범용 CPU 보다 종종 더 효율적으로 연산작업을 합니다. 이는 Section 2.2.3에서 이야기되는 규칙의 또 하나의 예입니다: 범용성은 거의 항상 공짜가 아닙니다.

하지만, Moore's-Law에 의한 싱글쓰레드 성능 증가의 종료를 생각해 보면, 다양한 특수 목적 하드웨어의 출현이 늘어날 것이라 생각해도 안전할 겁니다.

3.3.5 Existing Parallel Software

멀티코어 CPU가 컴퓨터 산업계를 놀라게 한 듯 보이지만, 실제로는 공유 메모리 병렬 컴퓨터 시스템이 판매되기 시작한지 25년도 넘게 지났습니다. 이는 상당한 병렬 소프트웨어가 나타나기에 충분한 시간이며, 실제로 그랬습니다. 병렬 운영체제는 상당히 흔해졌으며, 병렬 쓰레딩 라이브러리, 병렬 관계형 데이터베이스 관리 시스템, 그리고 병렬 수학 소프트웨어도 그렇습니다. 존재하는 병렬 소프트웨어의 사용은 우리가 마주칠 수 있는 모든 병렬 소프트웨어 참사를 해결하는 데에 상당한 진전을 가능하게 합니다.

이런 가장 흔한 예는 병렬 관계형 데이터베이스 관리 시스템일 겁니다. 싱글 쓰레드 프로그램이 중심의 관계형 데이터베이스에 동시적으로 접근하기 위해 고수준의 스크립트 언어로 쓰여있는 경우가 드물지 않습니다. 그 결과 만들어지는 고도로 병렬화된 시스템에서는 테

이터베이스만이 실제로 병렬성을 직접 다루게 됩니다. 동작할 때에는 매우 훌륭한 트릭입니다!

3.4 Software Design Implications

One ship drives east and another west
While the self-same breezes blow;
'Tis the set of the sail and not the gail
That bids them where to go.

Ella Wheeler Wilcox

Table 3.1의 비율값들은 상당히 중요한데, 그것들이 주어진 병렬 어플리케이션의 효율성을 제한하기 때문입니다. 이를 이해하기 위해, 이 병렬 어플리케이션이 쓰레드간 통신을 위해 CAS 오퍼레이션을 사용한다고 생각해 봅시다. 이 CAS 오퍼레이션은 일반적으로 캐쉬 미스를 낼텐데, 쓰레드들이 스스로 모든 걸 하기보다는 서로간에 통신을 한다는 가정 하에 그렇습니다. 나아가서 각 CAS 통신 오퍼레이션에 연관된 단위 작업이 300 ns, 즉 여러 부동소수점 연산을 수행할 수 있는 시간을 소비한다고 생각해 봅시다. 그럼 수행 시간의 절반 가량이 이 CAS 소통 오퍼레이션으로 소모될 겁니다! 이는 결국 그런 병렬 프로그램을 수행하는, 두개의 CPU가 달린 시스템은 순차적으로 구현된 이 프로그램을 한 개의 CPU로 수행하는 것보다 빠르지 않을 겁니다.

하나의 통신 오퍼레이션의 응답시간이 수천 또는 심지어 수백만 부동소수점 연산만큼이나 긴 시간을 소모하는 분산시스템의 경우는 이 상황이 더 나빠집니다. 이는 통신 오퍼레이션이 극단적으로 드물게 수행되어서 대부분의 시간은 진짜 일을 하는데 수행되어야 하는게 얼마나 중요한지를 잘 보입니다.

Quick Quiz 3.10: 분산시스템에서의 통신이 그렇게 무섭도록 비용이 높다면, 왜 누군가는 그런 시스템을 사용하여 하나요?

■

교훈은 분명합니다: 병렬 알고리즘은 이런 하드웨어 특성을 분명히 명심한 채로 설계되어야 합니다. 그러기 위한 한가지 방법은 거의 종속성 없는 쓰레드들을 수행시키는 것입니다. 이 쓰레드들이 어토믹 오퍼레이션을 사용해서든 락이나 명시적 메세지를 사용해서든 덜 통신할수록 이 어플리케이션의 성능과 확장성은 나아질 겁니다. 이 방법은 Chapter 5에서 이야기 되고, Chapter 6에서 둘러본 후, Chapter 8에서 그 논리적 극단을 취해 봅니다.

또 다른 방법은 모든 공유되는 것들에는 읽기가 대부분이게 하는 것으로, 이는 CPU의 캐슁이 읽기가 대부분인 데이터를 복사해 두어서 모든 CPU가 빠른 액세스를

할 수 있게 합니다. 이 방법은 Section 5.2.4 에서 다루어졌으며, Chapter 9 에서 더 자세히 알아봅니다.

요약하자면, 훌륭한 병렬 성능과 확장성을 이루어낸 데이터 구조와 알고리즘의 선택에 신경을 써서든 존재하는 병렬 어플리케이션과 환경을 사용해서든, 또는 해당 문제를 당황스럽도록 병렬적인 형태로 변환시킴을 통해서든 당황스럽도록 병렬적인 알고리즘과 구현을 위해 노력함을 의미합니다.

Quick Quiz 3.11: 좋아요, 우리가 분산 프로그래밍 기술을 공유메모리 병렬 프로그램에 적용해야 할 거라면, 그냥 항상 분산 기술을 사용하고 공유 메모리는 생략하는게 어떤가요?



자, 정리해 봅시다:

1. 좋은 소식은 멀티코어 시스템이 안비싸고 언제든 구할 수 있다는 겁니다.
2. 더 좋은 소식은: 많은 동기화 오퍼레이션의 오버헤드는 2000년대 초반의 병렬 시스템에서 그랬던 것보다 훨씬 낮다는 겁니다.
3. 안좋은 소식은 캐쉬 미스 오버헤드는 여전히 높으며, 거대한 시스템에서 특히 그렇다는 겁니다.

이 책의 뒷부분은 이 나쁜 소식을 다루는 방법들을 설명합니다.

좀 더 자세히 이야기 하자면, Chapter 4 는 병렬 프로그래밍에 사용되는 일부 저수준 도구들을 다루고, Chapter 5 는 병렬 카운팅의 문제들과 해결책을 분석해 보며, Chapter 6 에서는 성능과 확장성을 높이는 설계 철학을 다룹니다.

Chapter 4

Tools of the Trade

이 챕터는 병렬 프로그래밍 트레이드오프에 사용되는 일부 기본적 도구들을 주로 리눅스와 비슷한 운영체제에서 수행되는 어플리케이션에서 사용할 수 있는 것들에 초점을 맞춰 소개합니다. Section 4.1에서 스크립트 언어로 시작해서, Section 4.2에서는 POSIX API에 의해 지원되는 멀티 프로세스 병렬성을 설명하고 POSIX 쓰레드를 다룬 후, Section 4.3에서는 다른 환경에서의 비슷한 것들을 다룬 후, 마지막으로 Section 4.4에서는 일을 해결해 줄 도구를 고르는 걸 돋습니다.

Quick Quiz 4.1: 이것들을 도구라고 부르시나요???
이것들은 제게는 그보다는 저수준 (low-level) 동기화 도구들처럼 보이는데요!

■
이 챕터는 간단한 소개를 제공할 뿐임을 알아두시기 바랍니다. 더 자세한 것들은 참조문헌에서 (그리고 인터넷에서) 얻을 수 있으며, 더 많은 정보들이 뒤의 챕터들에서 제공될 겁니다.

4.1 Scripting Languages

The supreme excellence is simplicity.

Henry Wadsworth Longfellow, simplified

리눅스 셸 스크립트는 병렬성을 다루는 간단하지만 효과적인 방법들을 제공합니다. 예를 들어, 여러분이 `compute_it`이라는 프로그램이 있고 이걸 두개의 다른 인자 집합을 가지고 두번 수행시켜야 한다고 생각해 봅시다. 이걸 UNIX 셸 스크립트를 이용해서 다음과 같이 해낼 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
```

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

Unknown

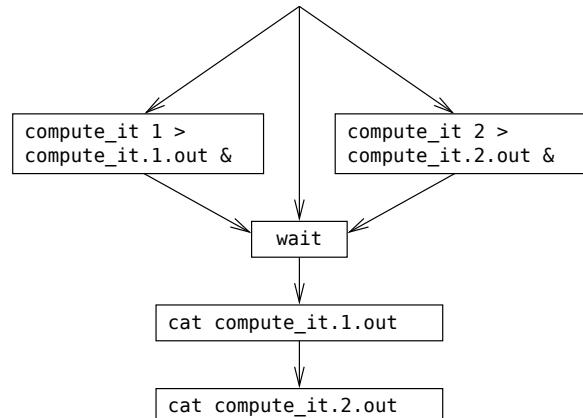


Figure 4.1: Execution Diagram for Parallel Shell Execution

라인 1과 2는 이 프로그램을 두번 실행시키고, 이 것들의 결과물을 두개의 별도의 파일에 저장하는데 & 문자는 셸에게 이 두 프로그램 실행을 백그라운드에서 하도록 지시합니다. 라인 3은 이 두개의 실행이 완료 되기를 기다리고, 라인 4과 5는 그 결과를 출력합니다. 그렇게 진행되는 실행과정이 Figure 4.1에 보여져 있습니다: `compute_it`의 두 실행은 병렬로 진행되며, `wait`은 이 두 실행이 모두 끝난 후 종료되며, 이후에는 두번의 `cat` 실행이 순차적으로 이루어집니다.

Quick Quiz 4.2:

하지만 이 웃긴 셸 스크립트는 진짜 병렬 프로그램이 아니예요! 왜 이런 사소한 걸 신경쓰죠???

■

Quick Quiz 4.3: 병렬 셸 스크립트를 작성할 수 있는 더 간단한 방법이 있을까요? 있다면, 어떻게 하죠? 없다면, 왜 없죠?

■

또 다른 예로, 소프트웨어 빌드 스크립트 언어인 `make`는 이 빌드 과정에 얼만큼의 병렬성이 사용되어야 할지

명세하는 `-j` 옵션을 제공합니다. 따라서, 리눅스 커널을 빌드할 때 `make -j4` 를 타이핑 하는 것은 최대 네개의 빌드 과정이 동시에 수행되도록 합니다.

이 간단한 예제들을 통해 병렬 프로그래밍은 항상 복잡하거나 어려워야 할 필요는 없다는 것을 여러분이 받아들이길 바랍니다.

Quick Quiz 4.4: 하지만 스크립트 기반의 병렬 프로그래밍이 그렇게 쉽다면, 다른 것들에 신경쓰는 이유가 뭘까요?

4.2 POSIX Multiprocessing

A camel is a horse designed by committee.

Unknown

이 섹션은 `pthreads` [Ope97] 를 포함해 POSIX 환경에 대해 간단히 알아보는데, 이 환경은 곧바로 사용 가능하며 널리 구현되어 있기 때문입니다. Section 4.2.1 에서는 POSIX `fork()` 와 관련된 도구들을 훑어보고, Section 4.2.2 에서는 쓰래드 생성과 제거에 대해 다룬 후, Section 4.2.3 에서는 POSIX 락킹에 대한 간단한 개론을 제공하며, 마지막으로 Section 4.2.4 에서는 많은 쓰래드에 의해 읽혀지지만 간혹 가다가 업데이트 되는 데이터에 사용되어야 하는 락에 대해 설명합니다.

4.2.1 POSIX Process Creation and Destruction

프로세스는 `fork()` 기능을 통해 생성되고, `kill()` 기능을 통해 소멸되며, `exit()` 기능을 통해 스스로를 소멸 시킬 수도 있습니다. `fork()` 기능을 실행하는 프로세스는 새로 생성된 프로세스의 “부모” 라 불립니다. 부모는 `wait()` 기능을 통해 자식을 기다릴 수 있습니다.

이 섹션의 예제는 상당히 간단한 것들임을 알아두시기 바랍니다. 이 기능들을 사용하는 실제 세계의 어플리케이션들은 시그널, 파일 디스크립터, 공유 메모리 세그먼트, 그리고 여러 많은 리소스를 다뤄야 할 수 있습니다. 또한, 일부 어플리케이션은 특정 자식이 종료되었을 때 특별한 행동을 취해야 하며, 또한 그 자식이 종료된 이유에 대해서 신경써야 할 수도 있습니다. 이 문제들은 또한 코드의 복잡도를 상당히 높일 수 있습니다. 더 많은 정보를 위해선, 이 주제에 대한 여러 책을 보시기 바랍니다 [Ste92, Wei13].

`fork()` 가 성공하면, 이 함수는 한번은 부모에게 또 한번은 자식에게 두번 리턴합니다. `fork()` 로부터 반환되는 값은 Listing 4.1 (`forkjoin.c`) 에 보인 것과 같이 호출자가 이 차이를 알 수 있게 합니다. 라인 1는

Listing 4.1: Using the `fork()` Primitive

```

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

Listing 4.2: Using the `wait()` Primitive

```

1 static __inline__ 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(EXIT_FAILURE);
13         }
14     }
15 }
```

`fork()` 기능을 실행하고, 그 반환값을 지역 변수인 `pid`에 저장합니다. 라인 2는 `pid` 가 0인지 체크하는데, 이는 이게 자식이라는 의미여서, 이 때에는 라인 3로 수행을 이어갑니다. 앞서 언급되었듯, 이 자식은 `exit()` 기능을 통해 종료될 수도 있습니다. 그렇지 않다면, 이것은 부모여서, 라인 4에서 `fork()` 기능으로부터 어려가 반환되었는지 체크하고, 그렇다면 라인 5-7에서 어려를 출력하고 종료합니다. 그렇지 않다면, `fork()` 는 성공적으로 수행된 것이며, 따라서 이 부모는 변수 `pid` 가 자식의 프로세스 ID 를 포함한 채로 라인 9를 수행하게 됩니다.

이 부모 프로세스는 자식들이 끝나길 기다리기 위해 `wait()` 함수를 사용할 수 있습니다. 하지만, `wait()` 호출은 한번에 단 하나의 자식 프로세스만 기다리기 때문에 이 함수의 사용은 셀 스크립트의 비슷한 것에 비해 약간 복잡합니다. 따라서 Listing 4.2 (`api-pthreads.h`)에 보인 것처럼 셀 스크립트의 `wait` 커맨드와 비슷한 의미를 갖는 `waitall()` 함수와 비슷한 함수로 `wait()` 를 감싸는 게 관습적입니다. 라인 6-14 의 루프를 통하여 각 패스가 자식 프로세스를 기다립니다. 라인 7는 하나의 자식 프로세스가 종료될 때까지 블록되어 기다리고 해당 자식 프로세스의 프로세스 ID 를 리턴하는 `wait()` 함수를 실행합니다. 리턴값이 프로세스 ID 가 아니라 -1라면, 이는 `wait()` 함수가 하나의 자식 프로세스를 기다릴 수 없었음을 알립니다. 만약 그렇다면, 라인 9는 `ECHILD` 어려 케이스를 검사하는데, 이 경우는 더이상 차일드 프로세스가 없음을 알리므로, 라인 10

Listing 4.3: Processes Created Via `fork()` Do Not Share Memory

```

1 int x = 0;
2
3 int main(int argc, char *argv[])
4 {
5     int pid;
6
7     pid = fork();
8     if (pid == 0) { /* child */
9         x = 1;
10    printf("Child process sees x=%d\n", x);
11    exit(EXIT_SUCCESS);
12 }
13 if (pid < 0) { /* parent, upon error */
14     perror("fork");
15     exit(EXIT_FAILURE);
16 }
17 /* parent */
18
19 waitall();
20 printf("Parent process sees x=%d\n", x);
21
22 return EXIT_SUCCESS;
23
24 }
```

에서 루프를 종료합니다. 그렇지 않다면, 라인 11 와 12에서는 에러를 출력하고 종료합니다.

Quick Quiz 4.5: 이 `wait()` 함수는 왜 그리 복잡하죠? 그냥 셀 스크립트의 `wait` 처럼 동작하게 만드는게 어떤가요?

부모와 자식은 메모리를 공유하지 않는다는 것을 알아두는게 무척 중요합니다. Listing 4.3 (`forkjoinvar.c`)에 보여진 프로그램이 이를 나타내는데, 여기선 자식이 전역 변수 `x`를 라인 9에서 1로 설정하고, 라인 10에서 메세지를 프린트 한 후, 라인 11에서 종료됩니다. 부모는 line 20을 이어 수행하는데, 여기서 자식을 기다리고, 라인 21에서 자신의 변수 `x`의 복사본이 여전히 0임을 확인합니다. 따라서 출력은 다음과 같을 겁니다:

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

Quick Quiz 4.6: `fork()` 와 `wait()`에 대해서 이야기할 게 더 많지 않나요?

세밀한 수준의 병렬성은 공유 메모리를 필요로 하며, 이는 Section 4.2.2에서 다루어집니다. 그러나, 공유 메모리 병렬성은 fork-join 병렬성보다 훨씬 더 복잡할 수 있습니다.

4.2.2 POSIX Thread Creation and Destruction

존재하는 프로세스 내에서 쓰레드를 만들기 위해선, 예를 들면 Listing 4.4 (`pcreate.c`)의 라인 16 와 17처럼

Listing 4.4: Threads Created Via `pthread_create()` Share Memory

```

1 int x = 0;
2
3 void *mythread(void *arg)
4 {
5     x = 1;
6     printf("Child process set x=%d\n", x);
7     return NULL;
8 }
9
10 int main(int argc, char *argv[])
11 {
12     int en;
13     pthread_t tid;
14     void *vp;
15
16     if ((en = pthread_create(&tid, NULL,
17                             mythread, NULL)) != 0) {
18         fprintf(stderr, "pthread_create: %s\n", strerror(en));
19         exit(EXIT_FAILURE);
20     }
21
22     /* parent */
23
24     if ((en = pthread_join(tid, &vp)) != 0) {
25         fprintf(stderr, "pthread_join: %s\n", strerror(en));
26         exit(EXIT_FAILURE);
27     }
28     printf("Parent process sees x=%d\n", x);
29
30     return EXIT_SUCCESS;
31 }
```

`pthread_create()` 기능을 실행시켜야 합니다. 첫번째 인자는 새로 생성되는 쓰레드의 ID를 저장하게 되는 `pthread_t`로의 포인터이고, 두번째 NULL 인자는 선택적으로 넣을 수 있는 `pthread_attr_t`로의 포인터이며, 세번째 인자는 이 새로운 쓰레드에 의해 수행될 함수 (이 경우, `mythread()`)이고, 마지막 NULL 인자는 `mythread()`에게 전달될 인자입니다.

이 예에서 `mythread()`는 단순히 리턴하지만, 그대로 `pthread_exit()`을 호출할 수도 있습니다.

Quick Quiz 4.7: Listing 4.4의 `mythread()` 함수가 그냥 리턴해도 된다면, 왜 `pthread_exit()`을 신경쓰나요?

라인 24에 보인 `pthread_join()` 기능은 fork-join의 `wait()` 기능과 유사합니다. 이 함수는 `tid` 변수로 특정된 쓰레드가 `pthread_exit()`을 수행해서 또는 해당 쓰레드의 꼭대기 단계 함수가 리턴해서 수행을 완료할 때까지 기다립니다. 이 쓰레드의 종료 값은 `pthread_join()`의 두번째 인자로 넘어간 포인터에 저장될 겁니다. 이 쓰레드의 종료 값은 해당 쓰레드가 어떻게 종료되느냐에 따라 `pthread_exit()`에 전달된 값이거나 이 쓰레드의 꼭대기 단계 함수에서 리턴된 값이 됩니다.

Listing 4.4에 보인 프로그램은 다음과 같은 실행 결과를 통해 메모리가 이 두 쓰레드에 의해 실제로 공유되고 있음을 보입니다.

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

이 프로그램은 한번에 쓰레드들 중 하나만이 변수 x 에 값을 저장하는 것을 주의 깊게 보장하고 있음을 알아두시기 바랍니다. 한 쓰레드가 특정 변수에 값을 저장하고 있는 동안 어떤 다른 쓰레드가 똑같은 변수의 값을 읽거나 쓰려 하는 모든 상황은 데이터 레이스 (*data race*)라고 호칭됩니다. C 언어는 데이터 레이스의 결과에 대해 어떤 합리적 보장도 하지 않기 때문에, 우린 다음 섹션에서 이야기 될 락킹 기능 등을 사용해서 동시에 안전하게 데이터를 접근하고 수정하는 방법이 필요합니다.

하지만 여러분의 데이터 레이스는 별거 아니다, 그렇게 말할 건가요? 글쎄요, 그럴지도 모르죠. 하지만, 부디 모두에게(여러분 자신도 포함해서) 큰 자비를 베푸시고 Section 4.3.4.1를 매우 주의 깊게 읽기 바랍니다. 컴파일러가 더욱 더 적극적으로 최적화를 할수록, 정말로 별거 아닌 데이터 레이스는 더욱 줄어듭니다.

Quick Quiz 4.8: 데이터 레이스의 존재 하에 C 언어는 어떤 보장도 하지 않는다면, 리눅스 커널은 왜 그렇게 많은 데이터 레이스를 갖는 거죠? 리눅스 커널은 완전히 망가져 있다는 말을 하고 싶은 건가요???

4.2.3 POSIX Locking

POSIX 표준은 프로그래머가 “POSIX 락킹”을 사용해 데이터 레이스를 방지할 수 있게 합니다. POSIX 락킹은 여러 기능을 제공하는데, 가장 기본적인 건 `pthread_mutex_lock()`과 `pthread_mutex_unlock()`입니다. 이 기능들은 `pthread_mutex_t` 타입인 락들을 통해 동작합니다. 이 락들은 `PTHREAD_MUTEX_INITIALIZER`를 통해 정적으로 선언되고 초기화 되거나, `pthread_mutex_init()` 기능을 통해 동적으로 할당되고 초기화될 수 있습니다. 이 섹션의 예제 코드는 앞의 것을 사용합니다.

`pthread_mutex_lock()` 기능은 특정 락을 “획득”하며, `pthread_mutex_unlock()`은 특정 락을 “해제”합니다. 이것들은 “배타적” 락킹 기능들이므로, 한번에 한 쓰레드만이 특정 시간에 특정 락을 “쥐고 있음” 수 있습니다. 예를 들어, 한 쌍의 쓰레드가 동시에 같은 락을 획득하려 시도하면, 이 한쌍 중 하나만이 그 락을 먼저 획득하는게 “허용” 될 것이고, 다른 쓰레드는 이 첫번째 쓰레드가 해당 락을 해제할 때까지 기다려야 합니다. 간단하고 합리적이며 유용한 프로그래밍 모델은 특정 데이터 하이템으로의 접근을 연관된 락을 쥐고 있을 때에만 허용하는 것입니다 [Hoa74].

Quick Quiz 4.9: 동시에 여러 쓰레드가 같은 락을 절 수 있게 하고 싶으면 어떡하죠?

이 배타적 락킹 속성이 Listing 4.5 (`lock.c`)의 코드에 보여져 있습니다. 라인 1은 `lock_a`라는 이름의 POSIX 락을 정의하고 초기화 하며, 라인 2는 비슷하게 `lock_b`라는 이름의 락을 정의하고 초기화 합니다. 라인 4는 공유된 변수 x 를 정의합니다.

라인 6-33는 `arg`로 명시된 락을 전 채로 공유 변수 x 를 반복적으로 읽는 `lock_reader()` 함수를 정의합니다. 라인 12는 `arg`를 `pthread_mutex_lock()`과 `pthread_mutex_unlock()` 기능들에 요구되는 대로 `pthread_mutex_t` 포인터로 캐스팅합니다.

Quick Quiz 4.10: 왜 Listing 4.5의 line 6에 있는 `lock_reader()`의 인자를 `pthread_mutex_t` 포인터로 만들지 않는 거죠?

Quick Quiz 4.11: Listing 4.5의 라인 20와 47의 `READ_ONCE()`와 라인 47의 `WRITE_ONCE()`는 왜 있는 거죠?

라인 14-18는 명시된 `pthread_mutex_t`를 획득하고, 예러를 체크하고 예러가 있다면 프로그램을 종료합니다. 라인 19-26는 x 의 값을 반복적으로 체크하고 그게 바뀌었을 때마다 새 값을 프린트 합니다. 라인 25은 1밀리세컨드 동안 잠을 자는데, 이는 단일 프로세서 기계에서 이 코드가 잘 동작하게끔 합니다. 라인 27-31는 `pthread_mutex_t`를 해제하고, 다시 예러를 체크하고 예러가 있으면 프로그램을 종료합니다. 마지막으로, 라인 32은 `NULL`을 리턴함으로써 `pthread_create()`에 의해 요구된 함수 타입을 맞춥니다.

Quick Quiz 4.12: `pthread_mutex_t`를 획득하고 해제할 때마다 네 줄의 코드를 써야 하는 건 분명 고통스러울 것 같군요! 더 나은 방법은 없을까요?

Listing 4.5의 라인 35-56는 주기적으로 공유 변수 x 를 특정 `pthread_mutex_t`를 전 채로 업데이트하는 `lock_writer()` 함수를 보입니다. `lock_reader()`에서처럼, 라인 39는 `arg`를 `pthread_mutex_t` 포인터로 캐스팅하고, 라인 41-45는 해당 락을 획득하며, 라인 50-54는 이를 해제합니다. 이 락을 잡고 있는 동안, 라인 46-49는 이 공유 변수 x 의 값을 증가시키고, 각 증가 사이에 5밀리세컨드를 잡니다. 마지막으로, 라인 50-54는 이 락을 해제합니다.

Listing 4.6은 `lock_reader()`와 `lock_writer()`를 `lock_a`라는 같은 락을 사용하는 쓰레드로 수행하는 코드 조각을 보입니다. 라인 2-6는 `lock_reader()`를 수행하는 쓰레드를 만들고, 이어서 라인 7-11는 `lock_writer()`를 수행하는 쓰레드를 만듭니다. 라인 12-19

Listing 4.5: Demonstration of Exclusive Locks

```

1 pthread_mutex_t lock_a = PTHREAD_MUTEX_INITIALIZER;
2 pthread_mutex_t lock_b = PTHREAD_MUTEX_INITIALIZER;
3
4 int x = 0;
5
6 void *lock_reader(void *arg)
7 {
8     int en;
9     int i;
10    int newx = -1;
11    int oldx = -1;
12    pthread_mutex_t *pmlp = (pthread_mutex_t *)arg;
13
14    if ((en = pthread_mutex_lock(pmlp)) != 0) {
15        fprintf(stderr, "lock_reader:pthread_mutex_lock: %s\n",
16                strerror(en));
17        exit(EXIT_FAILURE);
18    }
19    for (i = 0; i < 100; i++) {
20        newx = READ_ONCE(x);
21        if (newx != oldx) {
22            printf("lock_reader(): x = %d\n", newx);
23        }
24        oldx = newx;
25        poll(NULL, 0, 1);
26    }
27    if ((en = pthread_mutex_unlock(pmlp)) != 0) {
28        fprintf(stderr, "lock_reader:pthread_mutex_unlock: %s\n",
29                strerror(en));
30        exit(EXIT_FAILURE);
31    }
32    return NULL;
33 }
34
35 void *lock_writer(void *arg)
36 {
37     int en;
38     int i;
39     pthread_mutex_t *pmlp = (pthread_mutex_t *)arg;
40
41     if ((en = pthread_mutex_lock(pmlp)) != 0) {
42         fprintf(stderr, "lock_writer:pthread_mutex_lock: %s\n",
43                 strerror(en));
44         exit(EXIT_FAILURE);
45    }
46    for (i = 0; i < 3; i++) {
47        WRITE_ONCE(x, READ_ONCE(x) + 1);
48        poll(NULL, 0, 5);
49    }
50    if ((en = pthread_mutex_unlock(pmlp)) != 0) {
51        fprintf(stderr, "lock_writer:pthread_mutex_unlock: %s\n",
52                strerror(en));
53        exit(EXIT_FAILURE);
54    }
55    return NULL;
56 }

```

Listing 4.6: Demonstration of Same Exclusive Lock

```

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

```

Listing 4.7: Demonstration of Different Exclusive Locks

```

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

```

는 두 쓰레드가 모두 완료될 때까지 기다립니다. 이 코드 조각의 결과물은 다음과 같습니다:

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

두 쓰레드가 같은 락을 사용하기 때문에, lock_reader() 쓰레드는 lock_writer() 가 락을 잡은 채로 만들어내는 x의 중간 값들은 보지 못합니다.

Quick Quiz 4.13: “x=0”는 listing 4.6의 코드 조각이 낼 수 있는 유일한 결과일까요? 그렇다면, 왜죠? 아니라면, 어떤 다른 결과가 나올 수 있고, 그건 왜일까요?



Listing 4.7 이 비슷한 코드 조각을 보이는데, 이번에는 다른 락들을 사용합니다: lock_reader()는 lock_a 를, lock_writer()는 lock_b 를. 이 코드 조각의 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
```

이 두 쓰레드가 다른 락들을 사용하므로, 서로를 배타하지 않고, 동시에 수행될 수 있습니다. 따라서 `lock_reader()` 함수는 `lock_writer()`가 저장한 `x`의 중간값들을 볼 수 있습니다.

Quick Quiz 4.14: 다른 락을 사용하는 건 서로의 중간 상태를 어떤 락을 가지고 바라보는지에 대해 상당한 혼란을 야기할 수 있습니다. 그러니 이런 종류의 혼란을 막기 위해 잘 쓰여진 병렬 프로그램은 같은 락을 사용하도록 강제되어야 할까요?

Quick Quiz 4.15: Listing 4.7에 보여진 코드에서, `lock_reader()`는 `lock_writer()`가 만들어내는 값을 모두 볼 것이 보장될까요? 그렇다면 왜일까요, 아니라면 또 왜일까요?

Quick Quiz 4.16: 기다려봐요!!! Listing 4.6은 공유 변수 `x`를 초기화 하지 않았는데, 왜 Listing 4.7에서는 초기화 되어야 했죠?

POSIX 배타적 락킹에 대해서는 다룰 게 더 많이 있지만, 이 기능들로도 시작해볼 수 있으며 상당히 많은 상황에서 실제로 충분합니다. 다음 섹션은 POSIX reader-writer 락킹에 대해 간단히 살펴 봅니다.

4.2.4 POSIX Reader-Writer Locking

POSIX API는 `pthread_rwlock_t`로 표현되는 reader-writer 락을 하나 제공합니다. `pthread_mutex_t`에서처럼, `pthread_rwlock_t`는 `PTHREAD_RWLOCK_INITIALIZER`를 통해 정적으로 초기화될 수도 있고 `pthread_rwlock_init()` 기능을 통해 동적으로 초기화 될 수도 있습니다. `pthread_rwlock_rdlock()` 기능은 특정 `pthread_rwlock_t`를 읽기-획득하고, `pthread_rwlock_wrlock()` 기능은 쓰기-획득하며, `pthread_rwlock_unlock()` 기능은 해당 락을 해제합니다. 한번에 단 하나의 쓰레드만이 특정 `pthread_rwlock_t`를 쓰기-획득하고 있을 수 있지만, 해당 락을 쓰기-획득해서 가지고 있는 쓰레드가 현재 존재하지 않는다면 여러 쓰레드가 읽기-획득해서 가지고 있을 수 있습니다.

예상했을 수도 있겠지만, reader-writer 락은 읽기가 대부분인 상황을 위해 설계되었습니다. 배타적 락은 그 정의에 따라 한번에 하나의 쓰레드만이 락을 짚 수 있지만 reader-writer 락은 임의의 큰 수의 읽기 쓰레드들이

Listing 4.8: Measuring Reader-Writer Lock Scalability

```
1 pthread_rwlock_t rwl = PTHREAD_RWLOCK_INITIALIZER;
2 unsigned long holdtime = 0;
3 unsigned long 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 en;
15     int i;
16     long long loopcnt = 0;
17     long me = (long)arg;
18
19     __sync_fetch_and_add(&nreadersrunning, 1);
20     while (READ_ONCE(goflag) == GOFLAG_INIT) {
21         continue;
22     }
23     while (READ_ONCE(goflag) == GOFLAG_RUN) {
24         if ((en = pthread_rwlock_rdlock(&rwl)) != 0) {
25             fprintf(stderr,
26                     "pthread_rwlock_rdlock: %s\n", strerror(en));
27             exit(EXIT_FAILURE);
28         }
29         for (i = 1; i < holdtime; i++) {
30             wait_microseconds(1);
31         }
32         if ((en = pthread_rwlock_unlock(&rwl)) != 0) {
33             fprintf(stderr,
34                     "pthread_rwlock_unlock: %s\n", strerror(en));
35             exit(EXIT_FAILURE);
36         }
37         for (i = 1; i < thinktime; i++) {
38             wait_microseconds(1);
39         }
40         loopcnt++;
41     }
42     readcounts[me] = loopcnt;
43     return NULL;
44 }
```

락을 동시에 쥐고 있을 수 있게 하므로, 이런 상황에서 reader-writer 락은 배타적 락보다 훨씬 높은 확장성을 제공합니다. 하지만, 실전에서는 reader-writer 락에 의해 제공되는 추가적 확장성이 얼마나 되는지 알 필요가 있습니다.

Listing 4.8 (`rwlockscale.c`)은 reader-writer 락의 확장성을 측정하는 한가지 방법을 보입니다. 라인 1은 reader-writer 락의 정의와 초기화를 보이며, 라인 2은 각 쓰레드가 reader-writer 락을 쥐는 시간을 제어하는 `holdtime` 인자를 보이며, 라인 3은 reader-writer 락의 해제와 다음 획득 사이의 시간을 제어하는 `thinktime` 인자를 보이고, 라인 4는 각 읽기 쓰레드가 락을 획득한 횟수를 기록하는 `readcounts` 배열을 정의하며, 라인 5은 언제 모든 읽기 쓰레드가 수행을 시작하는지 결정하는 `nreadersrunning` 변수를 정의합니다.

라인 7-10는 이 테스트의 시작과 끝을 동기화 하는 `goflag`를 정의합니다. 이 변수는 초기에 `GPFLAG_INIT`

로 설정되고, 이후 모든 읽기 쓰레드가 시작된 후에는 `GOFLAG_RUN`이 되며, 이 테스트 수행이 종료된 후에는 `GOFLAG_STOP`이 됩니다.

라인 12-44는 읽기 쓰레드인 `reader()`를 정의합니다. 라인 19는 이 쓰레드가 이제 수행 중임을 표시하기 위해 `nreadersrunning` 변수를 어ト믹하게 증가시키며, 라인 20-22는 이 테스트가 시작하기를 기다립니다. `READ_ONCE()` 기능은 이 컴파일러가 이 루프의 각 패스에서 `goflag`를 읽어오도록 강제합니다—그렇게 강제하지 않으면 컴파일러는 `goflag`의 값이 절대 바뀌지 않을 거라고 생각할 수 있습니다.

Quick Quiz 4.17: 모든 곳에서 `READ_ONCE()`를 사용하는 대신에 간단히 Listing 4.8의 라인 10에서 `goflag`를 `volatile`로 선언하는 것은 어떤가요?

■

Quick Quiz 4.18: `READ_ONCE()`는 컴파일러에만 영향을 끼치고 CPU는 건들지 않습니다. Listing 4.8의 `goflag`의 값의 변화가 빠르게 각 CPU로 전파됨을 보장하기 위해 메모리 배리어를 사용해야 하지 않나요?

■

Quick Quiz 4.19: 예를 들어 GCC의 `__thread` 저장소 클래스를 사용해 선언되는 것 같은 쓰레드별 변수를 접근할 때에도 `READ_ONCE()`를 사용할 필요가 있을까요?

■

라인 23-41의 루프는 성능 테스트를 진행합니다. 라인 24-28는 락을 획득하고, 라인 29-31는 수 마이크로 세컨드 동안 락을 쥐고 있으며, 라인 32-36는 락을 놓고, 라인 37-39는 락을 다시 잡기 전에 수 마이크로세컨드 동안 기다립니다. 라인 40는 이 락 획득 횟수를 셹니다.

라인 42는 이 락 획득 횟수를 `readcounts[]` 배열의 이 쓰레드를 위한 원소로 옮기고, 라인 43은 리턴해서 이 쓰레드를 종료시킵니다.

Figure 4.2는 이 테스트를 224코어 Xeon 시스템에서 코어당 두개의 하드웨어 쓰레드를 사용해 총 448개의 소프트웨어에게 보이는 CPU를 가지는 환경에서 수행되었을 때의 결과를 보입니다. `thinktime` 패러미터는 모든 테스트 동안 0이었으며, `holdtime` 패러미터는 1마이크로세컨드부터 (그래프 상의 “1us”) 10,000마이크로세컨드 (그래프 상의 “10000us”) 까지 적용되었습니다. 그려진 실제 값은 다음과 같습니다:

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

여기서 N 은 쓰레드의 수이며, L_N 은 N 개 쓰레드에 의한 락 획득 횟수이며 L_1 은 단일 쓰레드에 의한 락 획득 횟수입니다. 이상적인 하드웨어와 소프트웨어 확장성 하에서는 이 값은 항상 1.0이어야 합니다.

그림 상에서 볼 수 있듯이, reader-writer 락킹의 확장성은 분명 이상적이지 않은데, 특히 작은 크기의 크리

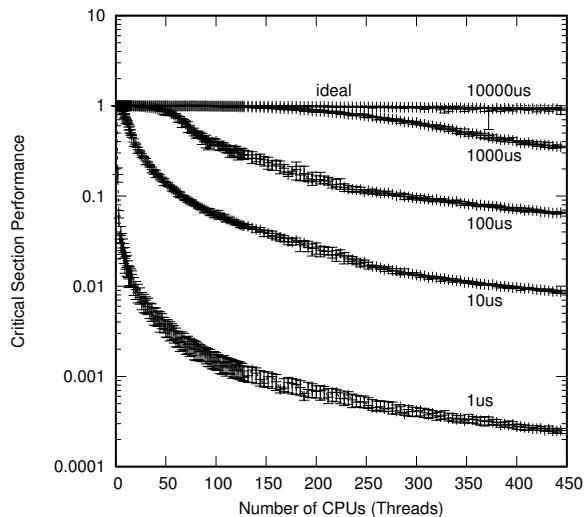


Figure 4.2: Reader-Writer Lock Scalability vs. Microseconds in Critical Section on 8-Socket System With Intel Xeon Platinum 8176 CPUs @ 2.10GHz

티컬 섹션에서 그렇습니다. 왜 읽기 락 획득이 그렇게 느린지 알아보기 위해선, 락을 획득하려는 모든 쓰레드가 `pthread_rwlock_t` 데이터 구조를 업데이트 한다는 걸 생각해 봅시다. 따라서, 만약 모든 448개의 쓰레드가 이 reader-writer 락을 동시에 읽기 모드로 획득하려 한다면, 이들은 `pthread_rwlock_t`를 한번에 하나씩 업데이트 해야만 합니다. 운이 좋은 쓰레드는 거의 곧바로 그렇게 할 수 있겠지만, 가장 운이 나쁜 쓰레드는 다른 447개의 쓰레드가 업데이트를 끝낼 때까지 기다려야만 합니다. 이 상황은 CPU를 추가할 수록 나빠지기만 할 겁니다. 그리고 y 축이 로그스케일이라는 점도 알아 두시기 바랍니다. 10,000마이크로세컨드의 기록이 상당히 이상적인 것으로 보이지만, 실제로는 이상적인 경우에 비해 10% 뒤쳐집니다.

Quick Quiz 4.20: 단일 CPU의 처리량과 비교하는 건 좀 너무한 거 아닌가요?

■

Quick Quiz 4.21: 하지만 1마이크로세컨드는 크리티컬 섹션의 크기로 특별히 작은 건 아닙니다. 예를 들어 몇개의 명령만이 들어있는 것 같은, 훨씬 더 작은 크리티컬 섹션이 필요할 땐 어떠해야 하죠?

■

Quick Quiz 4.22: 여기 사용된 시스템은 몇년 이상 되었고, 새로운 하드웨어는 더 빠를 겁니다. 그러니 누가 reader-writer 락이 느려짐에 대해 걱정하겠습니까?

■

이런 제한에도 불구하고, reader-writer 락킹은 많은 경우에 유용한데, 예를 들어 읽기 쓰레드들이 응답시간이

긴 파일이나 네트워크 I/O 를 처리해야 하는 경우입니다. 일부 대안도 존재하는데, Chapter 5 와 9 에서 보입니다.

4.2.5 Atomic Operations (GCC Classic)

Figure 4.2 는 reader-writer 락킹의 오버헤드는 가장 작은 크리티컬 섹션에서 가장 심각함을 보이며, 따라서 작은 크리티컬 섹션들을 보호하는 어떤 다른 방법이 있다면 좋을 겁니다. 그런 한가지 방법은 어토믹 오퍼레이션입니다. 우린 이미 하나의 어토믹 오퍼레이션을 봤는데, Listing 4.8 의 라인 19 의 `_sync_fetch_and_add()` 기능입니다. 이 기능은 두번째 인자의 값을 첫번째 인자로 참조되는 값에 원자적으로 더하고 기존 값을 리턴합니다 (이 경우엔 무시되었습니다). 한쌍의 쓰레드가 동시에 같은 변수에 대해 `_sync_fetch_and_add()` 를 실행하면 이 변수의 결과값은 두 더하기의 결과를 포함하게 됩니다.

GNU C 컴파일러는 `_sync_fetch_and_sub()`, `_sync_fetch_and_or()`, `_sync_fetch_and_and()`, `_sync_fetch_and_xor()`, 그리고 `_sync_fetch_and_nand()` 를 포함한 수많은 추가적 어토믹 오퍼레이션을 제공하는데, 이것들은 모두 기존 값을 리턴합니다. 그대신 새 값이 필요하다면, `_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()` 기능을 사용할 수 있습니다.

Quick Quiz 4.23: 그런 두 종류의 기능이 정말로 필요한가요?

고전적인 compare-and-swap 오퍼레이션은 `_sync_bool_compare_and_swap()` 와 `_sync_val_compare_and_swap()`, 한쌍의 기능들로 제공됩니다. 이 기능들 둘 모두 새로운 값을 원자적으로 업데이트 합니다만 그 기존 값이 명시된 이전 값과 동일할 때만 그렇습니다. 앞의 첫번째 변종은 이 오퍼레이션이 성공하면 1 을, 실패하면 0 을 리턴하는데, 예를 들어 그 기존 값이 명시된 이전 값과 갖지 않은 경우 실패합니다. 두번째 변종은 해당 위치의 기존값을 리턴하는데, 즉, 이 값이 명시된 이전 값과 동일하다면 이 오퍼레이션은 성공했음을 의미합니다. 모든 단일 위치로의 어토믹 오퍼레이션은 앞의 오퍼레이션들이 종종 더 효율적이지만, compare-and-swap 을 이용해 구현될 수 있다는 점에서 compare-and-swap 오퍼레이션은 “보편적”입니다. 이 compare-and-swap 오퍼레이션은 더 다양한 어토믹 오퍼레이션 집합의 기본으로 사용될 수도 있습니다만, 이것들을 더 다듬는 것은 종종 복잡도, 확장성, 성능 문제로 고통받습니다 [Her90].

Quick Quiz 4.24: 이 어토믹 오퍼레이션들은 밑바닥의 인스트럭션 셋을 이용해 직접적으로 지원되는 단일

Listing 4.9: Compiler Barrier Primitive (for GCC)

```
#define ACCESS_ONCE(x) (*(volatile typeof(x) *)&(x))
#define READ_ONCE(x) \
    ({ typeof(x) __x = ACCESS_ONCE(x); __x; })
#define WRITE_ONCE(x, val) \
    do { ACCESS_ONCE(x) = (val); } while (0)
#define barrier() __asm__ __volatile__("" : : :"memory")
```

어토믹 인스트럭션을 생성하는 경우가 많을 텐데, 그것이 일을 하는데 가장 빠른 방법일까요?

■

`_sync_synchronize()` 기능은 컴파일러와 CPU 모두 오퍼레이션을 재배치하지 못하게 하는 “메모리 배리어” 를 수행하는데, Chapter 15 에서 이야기 합니다. 어떤 경우에는 컴파일러의 오퍼레이션 재배치 기능은 제한하지만 CPU 는 내버려 둘도 충분한데, 그런 때에는 `barrier()` 기능이 사용될 수 있습니다. 어떤 경우에는 컴파일러가 특정 메모리 읽기를 최적화를 위해 없애버리는 것을 막을 필요가 있는데, 그럴 때에는 Listing 4.5 의 라인 20 에서와 같이 `READ_ONCE()` 기능이 사용될 수 있습니다. 비슷하게, `WRITE_ONCE()` 기능은 컴파일러가 특정 메모리 쓰기를 최적화해 없애버리는 걸 막는데 사용될 수 있습니다. 이 마지막 세개의 기능들은 GCC 에서 직접 제공되진 않습니다만 Listing 4.9 에 보인 것처럼 간단히 구현될 수 있으며, 이 세가지가 Section 4.3.4 에서 길게 설명됩니다. 대안적으로, `READ_ONCE(x)` 가 GCC 내재 기능인 `_atomic_load_n(&x, __ATOMIC_RELAXED)` 와 공통되는 부분이 많으며, `WRITE_ONCE()` 는 GCC 내재 기능인 `_atomic_store_n(&x, v, __ATOMIC_RELAXED)` 와 공통되는 부분이 많습니다.

Quick Quiz 4.25: `ACCESS_ONCE()` 에는 무슨 일이 벌어진 건가요?

■

4.2.6 Atomic Operations (C11)

C11 표준은 로드 (`atomic_load()`), 스토어 (`atomic_store()`), 메모리 배리어 (`atomic_thread_fence()` 와 `atomic_signal_fence()`), 그리고 read-modify-write 어토믹 오퍼레이션들을 포함한 어토믹 오퍼레이션들을 추가했습니다. 이 read-modify-write 어토믹 오퍼레이션에는 `atomic_fetch_add()`, `atomic_fetch_sub()`, `atomic_fetch_and()`, `atomic_fetch_xor()`, `atomic_exchange()`, `atomic_compare_exchange_strong()`, 그리고 `atomic_compare_exchange_weak()` 이 포함됩니다. 이것들은 Section 4.2.5 에서 설명한 것들과 비슷하게 동작합니다만 모든 오퍼레이션의 `_explicit` 변종의 추가된 메모리 순서 인자와 함께 동작합니다. 메모리 순서 인자가 없이는 이 모든 어토믹 오퍼레이션이 완전한 순서 규칙 (fully ordered)

아래 동작하며, 이 인자가 주어지면 좀 더 약한 순서 규칙 (weaker ordering) 을 허용합니다. 예를 들어, “`atomic_load_explicit(&a, memory_order_relaxed)`” 는 대략적으로 말해서 리눅스 커널의 “`READ_ONCE()`” 와 비슷합니다.¹

4.2.7 Atomic Operations (Modern GCC)

C11 어토믹 오퍼레이션의 한계점 중 하나는 특수한 어토믹 타입에만 그것들이 적용될 수 있다는 것으로, 이는 문제가 될 수 있습니다. 따라서 GNU C 컴파일러는 어토믹 내재기능들을 제공하는데, `__atomic_load()`, `__atomic_load_n()`, `__atomic_store()`, `__atomic_store_n()`, `__atomic_thread_fence()` 등이 포함됩니다. 이 내재기능들은 C11 의 비슷한 것들과 같은 기능을 제공합니다만, 평범한 어토믹 타입이 아닌 객체들에도 사용될 수 있습니다. 이것들 중 일부는 아래 리스트 중 하나의 메모리 순서 인자를 받을 수 있습니다: `__ATOMIC_RELAXED`, `__ATOMIC_CONSUME`, `__ATOMIC_ACQUIRE`, `__ATOMIC_RELEASE`, `__ATOMIC_ACQ_REL`, 그리고 `__ATOMIC_SEQ_CST`.

4.2.8 Per-Thread Variables

Thread-specific data, thread-local storage, 또는 다른 덜 겹손한 이름으로 불리는 쓰레드별 변수 (per-thread variables) 는 동시성 코드에서 굉장히 자주 사용되는데 Chapter 5 and 8 에서 더 이야기 될 겁니다. POSIX 는 쓰레드별 변수 생성을 (그리고 그에 연관된 키를 리턴하기 위해 `pthread_key_create()` 를, 키에 연관된 쓰레드별 변수의 삭제를 위해 `pthread_key_delete()` 를, 현재 쓰레드의 특정 키에 연관된 변수의 값을 설정하기 위해 `pthread_setspecific()` 를, 이 값을 리턴하기 위해 `pthread_getspecific()` 를 제공합니다.

여러 컴파일러가 (GCC 포함) 해당 변수가 쓰레드별로되어야 함을 나타내기 위해 변수 정의 부분에 사용될 수 있는 `__thread` 지시어를 제공합니다. 그러면 이 변수의 이름은 그 변수의 현재 쓰레드의 값을 평범하게 접근하는데 사용될 수 있습니다. 물론, `__thread` 는 POSIX thread-specific 데이터보다 사용하기가 훨씬 쉽고, 때문에 GCC 또는 `__thread` 를 지원하는 컴파일러로만 빌드되는 코드에서는 더 선호되는 편입니다.

다행히도, C11 표준은 `__thread` 의 자리에 사용될 수 있는 `_Thread_local` 키워드를 도입했습니다. 충분한 시간이 지난 후에는 이 새로운 키워드가 `__thread` 의 좋은 사용성과 POSIX thread-specific data 의 이식성을 결합시킬 겁니다.

¹ 메모리 순서 규칙은 Chapter 15 와 Appendix C 에 더 자세히 설명되어 있습니다.

4.3 Alternatives to POSIX Operations

The strategic marketing paradigm of Open Source is a massively parallel drunkard's walk filtered by a Darwinistic process.

Bruce Perens

불행히도, 쓰레드 오퍼레이션, 락킹 기능, 그리고 어토믹 오퍼레이션은 다양한 표준 위원회들이 그것들에 다가가기 훨씬 전부터 사용되어왔습니다. 그 결과, 이 오퍼레이션들이 어떻게 지원되는지에 대한 상당한 차이들이 존재합니다. 역사적인 이유로, 또는 특정 환경에서의 더 나은 성능을 위해서 이 오퍼레이션들이 어셈블리 언어로 구현되는 경우는 상당히 흔합니다. 예를 들어, GCC 의 `__sync_` 기능군은 모두 완전한 메모리 순서 규칙을 제공하는데, 이는 과거에 많은 개발자들을 완전한 메모리 순서 규칙이 필요하지 않은 상황을 위한 각자의 구현을 만들게 이끌었습니다. 다음 섹션들은 리눅스 커널의 일부 대안과 이 책의 예제 코드에서 사용된 역사적 기능들을 보입니다.

4.3.1 Organization and Initialization

많은 환경이 특별한 초기화 코드를 필요로 하지 않지만, 이 책의 예제 코드는 `pthread_t` 에서 연속된 정수들로의 매핑을 초기화하는 `smp_init()` 를 호출하는 것으로 시작합니다. 유저스페이스 RCU 라이브러리² 역시 비슷하게 `rcu_init()` 호출을 필요로 합니다. 생성자를 지원하는 환경 (GCC의 것 같은) 에서는 이 호출들이 숨겨질 수 있지만, 유저스페이스 RCU 라이브러리에서 지원되는 대부분의 RCU 변종들은 각 쓰레드가 쓰레드 생성 시에 `rcu_register_thread()` 를, 쓰레드 종료 전에 `rcu_unregister_thread()` 를 호출할 것을 필요로 합니다.

리눅스 커널의 경우, 커널은 특수한 초기화 코드 호출을 필요로 하지 않는다고 봐야 할지 또는 커널의 부팅 시의 코드가 사실은 필요한 초기화 코드라고 봐야 할지에 대한 것은 철학적 질문입니다.

4.3.2 Thread Creation, Destruction, and Control

리눅스 커널은 `kthread` 를 추적하기 위해 `struct task_struct` 포인터를, 그것들을 생성하기 위해 `kthread_create()` 를, 명시적으로 멈출 것을 제안하기 위

² RCU 에 대한 더 많은 정보를 위해 Section 9.5 를 보시기 바랍니다.

Listing 4.10: Thread API

```
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)
```

해 (POSIX 에는 비슷한 게 없습니다) `kthread_should_stop()` 을,³ 그것들이 멈추기를 기다리기 위해 `kthread_stop()` 을, 그리고 시간제한을 둔 기다림을 위해 `schedule_timeout_interruptible()` 을 사용합니다. 몇가지 추가적인 `kthread` 관리 API 가 존재합니다만, 이것들만으로도 좋은 시작 내지 검색어가 될 수 있습니다.

CodeSamples API 는 제어의 장소인 “threads” 에 집중합니다.⁴ 그런 쓰레드 각각은 `thread_id_t` 타입의 식별자를 가지며 동시에 수행 되는 두개의 쓰레드가 같은 지시어를 갖지는 못합니다. 쓰레드는 쓰레드별 지역 상태를 제외한 모든 것을 공유하는데⁵ 프로그램 카운터와 스택이 포함됩니다.

Listing 4.10 에 쓰레드 API 가 보여져 있으며, 그 멤버들이 다음 섹션에서 설명됩니다.

4.3.2.1 create_thread()

`create_thread()` 기능은 새로운 쓰레드를 하나 생성하고, `create_thread()` 의 첫번째 인자로 명시된 `func` 함수에서 이 새 쓰레드의 수행을 시작하며, `create_thread()` 의 두번째 인자로 명시된 인자를 넘겨줍니다. 이 새로 생성된 쓰레드는 `func` 로 명시된 시작 함수가 리턴할 때 종료됩니다. `create_thread()` 기능은 새로 생성된 자식 쓰레드에 연관된 `thread_id_t` 를 리턴합니다.

이 기능은 해당 프로그램이 생성한 쓰레드의 수를 암묵적으로 세면서 `NR_THREADS` 보다 많은 수의 쓰레드가 생성되면 프로그램을 종료시킵니다. `NR_THREADS` 는 컴파일 시점에 변경될 수 있는 상수입니다만 일부 시스템은 허용 가능한 쓰레드의 수에 대한 상한선을 가지고 있을 수도 있습니다.

4.3.2.2 smp_thread_id()

`create_thread()` 로부터 리턴된 `thread_id_t` 는 시스템 종속적이므로, `smp_thread_id()` 기능은 이 요청

³ POSIX 환경에서는 `kthread_should_stop()` 의 부재를 해결하기 위해 `pthread_join()` 과 함께 제대로 동기화 되는 boolean 플래그를 사용할 수 있습니다.

⁴ 비슷한 소프트웨어의 것들을 위한 많은 다른 이름들이 있는데 “프로세스”, “태스크”, “파이버”, “이벤트”, “수행 에이전트” 등이 있습니다. 비슷한 설계 원칙이 이것들 모두에 적용됩니다.

⁵ 순환적 정의에서는 이게 어떻게 될까요?

을 한 쓰레드에 연관된 쓰레드 인덱스를 리턴합니다. 이 인덱스는 이 프로그램이 시작된 이후로 존재해온 쓰레드의 최대 갯수보다 작을 것이 보장되어 있으며, 따라서 비트마스킹, 배열 인덱스 등등에 유용합니다.

4.3.2.3 for_each_thread()

`for_each_thread()` 매크로는 존재하는 모든 쓰레드를 순회하는데 생성되었다면 존재 했을 모든 쓰레드가 포함됩니다. 이 매크로는 Section 4.2.8 에서 보이듯이 쓰레드별 변수를 제어하는데 유용합니다.

4.3.2.4 for_each_running_thread()

`for_each_running_thread()` 매크로는 현재 존재하는 쓰레드에 대해서만 순회를 합니다. 필요하다면 쓰레드 생성과 삭제에 대해 동기화 하는 것은 호출하는 쪽의 역할입니다.

4.3.2.5 wait_thread()

`wait_thread()` 기능은 그것에 넘겨지는 `thread_id_t` 로 명시되는 쓰레드의 종료를 기다립니다. 이는 이 명시된 쓰레드의 수행에 대해서는 어떤 영향도 끼치지 않습니다; 대신, 그저 기다립니다. `wait_thread()` 는 연관된 쓰레드가 리턴하는 값을 리턴함을 알아두시기 바랍니다.

4.3.2.6 wait_all_threads()

`wait_all_threads()` 기능은 현재 수행중인 모든 쓰레드의 종료를 기다립니다. 필요하다면 쓰레드 생성과 삭제와 동기화 하는 것은 호출자의 역할입니다. 하지만, 이 기능은 일반적으로 프로그램 수행 종료 시에 정리를 위해 사용되며, 따라서 그런 동기화는 보통은 필요치 않습니다.

4.3.2.7 Example Usage

Listing 4.11 (`threadcreate.c`) 은 헬로월드 같은 자식 쓰레드 예제를 보입니다. 앞서 이야기 되었듯, 각 쓰레드는 스스로의 스택을 할당받으며, 따라서 각 쓰레드는 각자의 사적인 `arg` 인자와 `myarg` 변수를 갖습니다. 각 자식은 종료 전에 간단히 자신의 인자와 `smp_thread_id()` 를 출력합니다. 라인 7 의 `return` 문은 이 쓰레드를 종료시키고, 이 쓰레드를 위한 `wait_thread()` 를 호출한 누군가에게 `NULL` 을 리턴합니다.

이 부모 프로그램이 Listing 4.12 에 보여져 있습니다. 이 프로그램은 라인 6 에서 이 쓰레드 시스템을 초기화 시키기 위해 `smp_init()` 를 호출하고, 라인 8-15 에서 인자들을 분석한 후, 자신의 존재를 라인 16 에서

Listing 4.11: Example Child Thread

```

1 void *thread_test(void *arg)
2 {
3     int myarg = (intptr_t)arg;
4
5     printf("child thread %d: smp_thread_id() = %d\n",
6            myarg, smp_thread_id());
7     return NULL;
8 }
```

Listing 4.12: Example Parent Thread

```

1 int main(int argc, char *argv[])
2 {
3     int i;
4     int nkids = 1;
5
6     smp_init();
7
8     if (argc > 1) {
9         nkids = strtoul(argv[1], NULL, 0);
10        if (nkids > NR_THREADS) {
11            fprintf(stderr, "nkids = %d too large, max = %d\n",
12                    nkids, NR_THREADS);
13            usage(argv[0]);
14        }
15    }
16    printf("Parent thread spawning %d threads.\n", nkids);
17
18    for (i = 0; i < nkids; i++)
19        create_thread(thread_test, (void *)(intptr_t)i);
20
21    wait_all_threads();
22
23    printf("All spawned threads completed.\n");
24
25    exit(0);
26 }
```

Listing 4.13: Locking API

```

void spin_lock_init(spinlock_t *sp);
void spin_lock(spinlock_t *sp);
int spin_trylock(spinlock_t *sp);
void spin_unlock(spinlock_t *sp);
```

알립니다. 명시된 수의 자식 쓰레드를 라인 18-19에서 만들고, 그것들이 종료되기를 라인 21에서 기다립니다. `wait_all_threads()`는 이 쓰레드들의 리턴 값들이 이 경우에는 모두 별 흥미 없는 `NULL`일 것이므로, 무시합니다.

Quick Quiz 4.26: 리눅스 커널의 `fork()` 와 `wait()` 비슷한 것들엔 무슨 일이 일어났나요?

4.3.3.1 spin_lock_init()

`spin_lock_init()` 기능은 명시된 `spinlock_t` 변수를 초기화 하며, 이 변수가 다른 스핀락 기능들에 넘겨지기 전에 호출되어야만 합니다.

4.3.3.2 spin_lock()

`spin_lock()` 기능은 명시된 스핀락을 획득하는데, 필요하다면 이 스핀락이 획득 가능해질 때까지 기다립니다. Pthread 같은 일부 환경에서는 이 기다림이 블록킹을 일으킬 수도 있는데, 리눅스 커널과 같은 다른 환경에서는 CPU-bound spin loop을 일으킬 수도 있습니다.

핵심은 한번에 단 하나의 쓰레드만이 스핀락을 잡을 수 있다는 것입니다.

4.3.3.3 spin_trylock()

`spin_trylock()` 기능은 명시된 스핀락을 획득합니다만, 곧바로 획득이 가능할 때만 그렇습니다. 해당 스핀락을 획득할 수 있었다면 `true`를 리턴하고 그렇지 않다면 `false`를 리턴합니다.

4.3.3.4 spin_unlock()

`spin_unlock()` 기능은 명시된 스핀락을 해제해서 다른 쓰레드가 해당 락을 획득할 수 있게 해줍니다.

4.3.3.5 Example Usage

`mutex` 라 이름지어진 스핀락이 `counter` 변수를 보호하기 위해 아래와 같이 사용될 수 있습니다:

```

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

Quick Quiz 4.27: 변수 `counter`가 `mutex`의 보호 없이 증가되면 어떤 문제가 벌어지나요?



하지만, `spin_lock()` 과 `spin_unlock()` 기능은 성능에 영향을 끼치는데, Chapter 10에서 이에 대해 알아보겠습니다.

4.3.4 Accessing Shared Variables

2011년 전까지 C 표준은 공유된 변수에 동시적으로 `read/write` 액세스를 하는 것에 대한 의미를 정의하지 않았습니다. 하지만, 동시성 C 코드는 최소 4반세기 전부터 쓰여지기 시작했습니다 [BK85, Inm85]. 이는 오늘날의 수염이 하얗게 센 분들은 C11 이전의 면과거에는

4.3.3 Locking

시작하면서 알아보기 좋을 리눅스 커널의 락킹 API 중 일부의 집합이 Listing 4.13에 표시되어 있는데, 각 API 원소는 다음 섹션들에서 설명됩니다. 이 책의 CodeSamples 락킹 API는 리눅스 커널의 그것을 따라갑니다.

Listing 4.14: Living Dangerously Early 1990s Style

```

1 ptr = global_ptr;
2 if (ptr != NULL && ptr < high_address)
3   do_low(ptr);

```

Listing 4.15: C Compilers Can Invent Loads

```

1 if (global_ptr != NULL &&
2     global_ptr < high_address)
3   do_low(global_ptr);

```

어떻게 살았는지 궁금증을 일으킵니다. 이 질문에 대한 짧은 답은 “그들은 위험하게 살았다”입니다.

그들은 최소한 2021년도 컴파일러를 사용했더라도 위험하게 살았을 겁니다. (대충) 1990년대 초, 컴파일러는 지금보다 적은 최적화를 했는데, 이는 부분적으로는 더 적은 컴파일러 작성자가 존재했으며 다른 부분적으로는 당시에 메모리가 상대적으로 더 작았기 때문입니다. 하지만 Listing 4.14에 보여진 것처럼 문제는 일어났는데, 컴파일러는 이 코드를 Listing 4.15의 것으로 바꿀 권리를 가지고 있습니다. 볼 수 있듯이, Listing 4.14의 라인 1에서의 임시적 부분이 최적화로 사라져버리고, 따라서 `global_ptr`는 세번 로드될 겁니다.

Quick Quiz 4.28: Listing 4.14의 `global_ptr`를 세번 로드하는데 문제가 뭐죠?

Section 4.3.4.1은 평범한 액세스가 일으키는 추가적인 문제들을 설명하고, Section 4.3.4.2와 4.3.4.3는 C11 이전 컴파일러에서의 해결책 일부를 설명합니다. 물론, 그게 실용적이라면 Section 4.2.5 또는 (특히나) Section 4.2.6에서 설명된 기능들이 데이터 레이스를 막기 위해, 즉, 특정 변수에 동시적인 여러 액세스가 있다면 그 액세스는 모두 로드라는 것을 보장하기 위해 사용되어야 합니다.

4.3.4.1 Shared-Variable Shenanigans

평범한 로드와 스토어를 하는⁶ 코드가 주어지면, 컴파일러는 그로 인해 영향을 받는 변수들은 다른 쓰레드에 의해 액세스 되거나 수정되지 않는다고 가정할 권리를 갖습니다. 이 가정은 컴파일러가 많은 변경을 할 수 있게 하는데, load tearing, store tearing, load fusing, store fusing, 코드 재배치, invented load, invented store, store-to-load 변경, dead-code 제거, 싱글쓰레드 코드에서라면 문제 없을 만한 모든 것을 포함합니다. 하지만 동시성 코드는 이런 변화들, 달리 말하면 공유 변수 사기질로 인해 망가질 수 있는데, 아래에서 이를 설명합니다.

Load tearing은 컴파일러가 하나의 액세스를 위해 여러 로드 인스트럭션을 사용할 때 발생합니다. 예를

⁶ 즉, C11 어토믹, 인라인 어셈블리, 또는 volatile 액세스가 아닌 일반적인 로드와 스토어.

Listing 4.16: Inviting Load Fusing

```

1 while (!need_to_stop)
2   do_something_quickly();

```

들어, 컴파일러는 이론상 `global_ptr` (Listing 4.14)의 라인 1을 참고하세요로부터의 로드를 1바이트 로드 여러개로 번역해낼 수 있습니다. 만약 어떤 다른 쓰레드가 동시에 `global_ptr`를 NULL로 만든다면, 그 결과는 해당 포인터의 한 바이트를 제외한 나머지는 0이 될 것이어서, “야생의 포인터”를 형성할 것입니다. 그런 야생의 포인터를 사용한 스토어는 메모리의 임의의 지역을 오염시켜서, 드물고 디버깅하기 어려운 크래쉬를 야기할 것입니다.

더 나쁜 것이, (말하자면) 16-비트 포인터를 갖는 8-비트 시스템에서, 컴파일러는 주어진 포인터를 접근하기 위해 8-비트 인스트럭션 한쌍을 사용하는 것밖에 선택이 없을 수도 있습니다. C 표준은 모든 시스템을 지원해야 하므로, 이 표준은 일반적인 경우에서의 load tearing을 제거할 수 없습니다.

Store tearing은 컴파일러가 단일 액세스를 위해 여러 스토어 인스트럭션을 사용할 때 발생합니다. 예를 들어, 한 쓰레드는 4-바이트 정수형 변수에 0x12345678을 저장하고 있는 와중에 다른 쓰레드는 0xabcdef00을 저장하고 있을 수 있습니다. 컴파일러가 각 액세스를 위해 16-비트 스토어를 사용한다면, 그 결과는 0x1234ef00이 될 수도 있는데, 이 정수를 로드하는 코드 입장에선 무척 놀라운 결과가 될 겁니다. 이건 엄밀한 이론적 이슈일 뿐인 것이 아닙니다. 예를 들어, 작은 즉석 인스트럭션 필드를 갖는 CPU들이 있으며, 그런 CPU에서 컴파일러는 64-비트 CPU에서 할지라도 레지스터에 64-비트 상수를 명시적으로 두는 오버헤드를 줄이기 위해 64-비트 스토어를 두개의 32-비트 스토어로 쪼갤 수 있습니다. 이게 현장에서 실제로 발생한 역사적 보고서들이 존재하며 (예를 들어 [KM13]), 최근의 보고도 있습니다 [Dea19].⁷

물론, 32-비트 시스템에서 64-비트 정수형을 사용하는 코드가 있을 수 있다는 점을 놓고 보면 컴파일러는 일반적인 경우에 그냥 일부 스토어를 쪼개는 것밖에 다른 선택지가 없습니다. 하지만 제대로 정렬된 머신 워드 크기 스토어에 대해서는 `WRITE_ONCE()`가 store tearing을 방지할 겁니다.

Load fusing은 컴파일러가 특정 변수에 대한 이전 로드로부터의 결과를 한번 더 로드하는 대신 사용할 때 발생합니다. 이는 싱글쓰레드 코드에서는 잘 동작하는

⁷ 이 조개짐은 제대로 정렬된 머신 워드 크기 액세스에서 조차 발생할 수 있으며, 이 특수한 경우 volatile 스토어들에 대해서까지 그렇습니다. 어떤 사람들은 이 동작인 컴파일러 내의 버그를 의미한다고 주장할 수도 있겠지만, 어떻게 표현하건 컴파일러 작성자의 시점에서의 store tearing의 인식된 가치를 나타내고 있습니다.

Listing 4.17: C Compilers Can Fuse Loads

```

1 if (!need_to_stop)
2   for (;;) {
3     do_something_quickly();
4     do_something_quickly();
5     do_something_quickly();
6     do_something_quickly();
7     do_something_quickly();
8     do_something_quickly();
9     do_something_quickly();
10    do_something_quickly();
11    do_something_quickly();
12    do_something_quickly();
13    do_something_quickly();
14    do_something_quickly();
15    do_something_quickly();
16    do_something_quickly();
17    do_something_quickly();
18    do_something_quickly();
19  }

```

최적화일 뿐만 아니라 멀티쓰레드 코드에서도 종종 별 다른 문제를 야기하지 않습니다. 불행히도, “종종” 이란 단어는 실제로 짜증나는 예외를 숨깁니다.

예를 들어, 어떤 리얼타임 시스템이 `do_something_quickly()`라는 이름의 함수를 `need_to_stop`이라는 변수가 세팅될 때까지 반복적으로 호출되어야 하며, 컴파일러는 `do_something_quickly()`가 `need_to_stop`에 뭔가를 저장하지 않음을 볼 수 있다고 해봅시다. 이를 코딩하는 (안전하지 못한) 방법 하나가 Listing 4.16에 보여져 있습니다. 컴파일러는 루프 종료 시점의 반대방향 브랜칭을 줄이기 위해 합리적으로 이 루프를 16번 정도 언롤링 (unroll) 할 수도 있습니다. 더 나쁜 것은, 컴파일러는 `do_something_quickly()`가 `need_to_stop`에 무언가를 저장하지 않음을 알고 있으므로, 컴파일러는 이 변수를 단 한번만 검사하자는 합리적 결정을 내릴 수도 있어서, Listing 4.17의 코드를 내게 될 수도 있습니다. 일단 들어가면, 라인 2-19의 루프는 다른 쓰레드가 몇번이나 `need_to_stop`에 0이 아닌 값을 썼는가에 관계 없이 결코 끝나지 않습니다. 그 결과는 최선의 경우 놀랄일 것이고, 심각한 물리적 손상을 포함할 수도 있습니다.

컴파일러는 놀랍도록 많은 범위의 코드의 로드를 합칠 수 있습니다. 예를 들어, Listing 4.18,의 `t0()`와 `t1()`은 동시에 수행되며, `do_something()`과 `do_something_else()`는 인라인 함수입니다. 라인 1은 `gp` 포인터를 선언하는데, C는 이를 `NULL`로 초기화 합니다. 어떤 시점에선가, `t0()`의 라인 5는 `gp`에 `NULL`이 아닌 값을 저장합니다. 그사이, `t1()`은 라인 10, 12, 그리고 15에서 세번 `gp`로부터의 로드를 합니다. 라인 13은 `gp` 가 `NULL` 아 아님을 발견했으므로, 어떤 사람은 line 15에서의 역참조가 실패할 수가 없을 거라고 바랄 수도 있겠습니다. 불행히도, 컴파일러는 라인 10과 15에서의 읽기를 하나로 합칠 수가 있는데, 이는 라인 10이 `NULL`, 라인 12는 `&myvar`를 로드했다면, line 15는

Listing 4.18: C Compilers Can Fuse Non-Adjacent Loads

```

1 int *gp;
2
3 void t0(void)
4 {
5   WRITE_ONCE(gp, &myvar);
6 }
7
8 void t1(void)
9 {
10   p1 = gp;
11   do_something(p1);
12   p2 = READ_ONCE(gp);
13   if (p2) {
14     do_something_else();
15   p3 = *gp;
16 }
17 }

```

Listing 4.19: C Compilers Can Fuse Stores

```

1 void shut_it_down(void)
2 {
3   status = SHUTTING_DOWN; /* BUGGY!!! */
4   start_shutdown();
5   while (!other_task_ready) /* BUGGY!!! */
6     continue;
7   finish_shutdown();
8   status = SHUT_DOWN; /* BUGGY!!! */
9   do_something_else();
10 }
11
12 void work_until_shut_down(void)
13 {
14   while (status != SHUTTING_DOWN) /* BUGGY!!! */
15     do_more_work();
16   other_task_ready = 1; /* BUGGY!!! */
17 }

```

`NULL`을 로드하게 되어 실패가 일어날 수 있음을 의미합니다.⁸ 사이에 `READ_ONCE()`를 끼워넣는 것은 세개의 로드가 모두 같은 변수를 향한 것임에도 불구하고 다른 두개의 로드가 합쳐지는 것을 방지하지 않음을 알아두시기 바랍니다.

Quick Quiz 4.29: Listing 4.18 내의 `do_something()`과 `do_something_else()`가 인라인 함수라는 것이 왜 중요한가요?

Store fusing은 특정 변수로의 두개의 연달은 스토어가 중간에 해당 변수로의 로드 없이 이루어짐을 컴파일러가 눈치채면 발생할 수 있습니다. 이 경우, 컴파일러는 첫번째 스토어를 없애버릴 권리를 갖습니다. 이는 싱글 쓰레드 코드에서는 결코 문제가 되지 않습니다, 그리고 실제로 제대로 쓰여진 동시성 코드에서 이는 일반적으로 문제가 아닙니다. 어쨌건, 이 두개의 스토어가 빠른 속도로 수행되었다면, 다른 쓰레드가 첫번째 스토어로 인한 값을 읽을 기회는 아주 희박할 겁니다.

⁸ Will Deacon은 이게 리눅스 커널에서도 일어났음을 보고했습니다.

하지만 예외가 있는데, 그런 예가 Listing 4.19에 보여져 있습니다. `shut_it_down()` 함수는 라인 3와 8에서 공유변수 `status`에 스토어를 하고, 따라서 `start_shutdown()`과 `finish_shutdown()`은 `status`에 액세스 하지 않는다 생각하는데, 컴파일러는 라인 3에서의 `status`로의 스토어를 제거할 수 있습니다. 불행히도, 이는 `work_until_shutdown()`이 라인 14와 15의 루프를 결코 끝내지 않을 것을 의미하며, 따라서 `other_task_ready`에 0이 아닌 값을 저장하지 않을 것이어서, 결국 `shut_it_down()`은 라인 5와 6의 루프를 결코 끝내지 않을 것입니다. 컴파일러가 라인 5에서의 `other_task_ready`로부터의 이어지는 로드를 합쳐버리지 않는다 해도요.

그리고 Listing 4.19에는 코드 재배치를 포함한 더 많은 문제가 있습니다.

Code reordering은 흔한 세부 표현들을 하나로 합치기 위한 일반적인 컴파일 기술이며, 현대 슈퍼스칼라 마이크로프로세서에서 사용 가능한 많은 기능 유닛의 사용률을 개선시킵니다. 이는 또한 Listing 4.19의 코드에 왜 버그가 있는지에 대한 이유이기도 합니다. 예를 들어, 라인 15의 `do_more_work()` 함수가 `other_task_ready`에 액세스 하지 않는다고 생각해 봅시다. 그럼 컴파일러는 라인 16에서의 `other_task_ready`로의 값 할당을 라인 14를 앞서도록 재배치 할 수 있는데, 이는 라인 15에서의 `do_more_work()` 호출이 라인 7에서의 `finish_shutdown()` 호출 이전에 일어났기를 바라는 사람에게는 큰 실망이 될 겁니다.

아랫단의 하드웨어가 액세스 순서를 재배치 할 수 있는데 컴파일러가 액세스 순서를 재배치 하는 걸 방지하는 건 헛된 일처럼 보일 수도 있습니다. 하지만, 현대의 기계들은 *exact exception*과 *exact interrupt*를 가지고 있는데, 이는 어떤 인터럽트나 익셉션도 인스트럭션 흐름 내의 특정 장소에서 일어난 것으로 보일 것임을 의미합니다. 이는 해당 이벤트들의 핸들러는 모든 앞선 인스트럭션의 효과를 보게 될 것을, 그러나 어떤 뒤따르는 인스트럭션의 효과도 보자 않을 것을 의미합니다. 따라서 `READ_ONCE()`와 `WRITE_ONCE()`는 인터럽트된 코드와 인터럽트 핸들러 사이의 통신을 아랫단 하드웨어가 제공하는 순서와 무관하게 제어할 수 있습니다.⁹

Invented loads는 Listing 4.14와 4.15의 코드에서 설명되는데, 컴파일러가 임시 변수를 최적화로 없애버리고, 따라서 의도된 것보다 여러번 공유 변수로부터의 로드를 하는 것입니다.

만들어진 로드는 성능에 장애가 될 수 있습니다. 이 장애는 “뜨거운” 캐쉬라인의 변수로부터의 로드가 `if` 문 바깥에서 일어날 때 발생할 수 있습니다. 이런 최적

⁹ 그렇다고는 하나, 다양한 표준 위원회는 여러분이 `READ_ONCE()`와 `WRITE_ONCE()` 대신 `sig_atomic_t` 타입의 어토믹 변수를 사용할 것을 선호합니다.

Listing 4.20: Inviting an Invented Store

```
1 if (condition)
2   a = 1;
3 else
4   do_a_bunch_of_stuff();
```

Listing 4.21: Compiler Invents an Invited Store

```
1 a = 1;
2 if (!condition) {
3   a = 0;
4   do_a_bunch_of_stuff();
5 }
```

화는 드문 일이 아니며, 상당한 캐쉬 미스 증가를 일으킬 수 있으므로, 성능과 확장성에 상당한 하락을 야기할 수 있습니다.

Invented stores는 다양한 상황에서 발생할 수 있습니다. 예를 들어, Listing 4.19에서 `work_until_shutdown()`의 코드를 생성하는 컴파일러는 `other_task_ready`가 `do_more_work()`에 의해 액세스 되지 않으며 라인 16에서 스토어 된다는 것을 알아챌 수도 있습니다. 만약 `do_more_work()`이 복잡한 인라인 함수였다면, 레지스터 비우기를 해야 할 수도 있으며, 이 경우 `other_task_ready`는 임시 저장소로 사용하기 매력적인 장소입니다. 어쨌건, 거기 액세스가 없는데, 뭐가 문제겠습니까?

물론, 잘못된 시간에 이 변수로 0이 아닌 값을 저장하는 것은 라인 5의 `while` 루프가 예상보다 빨리 종료되게 하여, `finish_shutdown()`이 `do_more_work()`과 동시에 수행되는 것을 허용합니다. 이 `while`의 요점은 그런 동시 수행을 막기 위한 것임을 생각하면, 이는 좋은 일이 아닙니다.

스토어의 대상이 되는 변수를 임시 저장소로 사용하는건 이상하게 보일 수도 있겠지만 이는 표준에 의해 허용된 행위입니다. 그러나, 독자 여러분은 덜 이상한 예를 원할 수도 있을 텐데, 그런 분들을 위해 Listing 4.20과 4.21가 있습니다.

Listing 4.20의 코드를 만들어내는 컴파일러는 `a`의 값이 초기에는 0임을 알수도 있는데, 이는 이 코드를 Listing 4.21의 코드로 변형시킴으로써 브랜치 하나를 최적화해 없애버리고자 하는 강한 욕망을 느끼게 될 겁니다. 여기서, 라인 1는 1을 `a`에 무조건적으로 저장하고, 이어서 라인3에서 `condition`의 값이 0이라면 `a`의 값을 0으로 되돌립니다. 이는 이 `if-then-else`를 `if-then`으로 변형시켜서 하나의 브랜치를 아깁니다.

Quick Quiz 4.30: 이런! 그러니까 컴파일러는 언제든 원하면 평범한 변수로의 스토어를 만들어낼 수 없나요?

■

마지막으로, C11 이전의 컴파일러는 쓰여진 변수들에 인접한 관계 없는 변수들에 쓰기를 만들어낼 수도 있었습니다 [Boe05, Section 4.2]. 이 만들어진 스토어의

Listing 4.22: Inviting a Store-to-Load Conversion

```

1 r1 = p;
2 if (unlikely(r1))
3   do_something_with(r1);
4 barrier();
5 p = NULL;

```

Listing 4.23: Compiler Converts a Store to a Load

```

1 r1 = p;
2 if (unlikely(r1))
3   do_something_with(r1);
4 barrier();
5 if (p != NULL)
6   p = NULL;

```

변종은 데이터 레이스를 만들어내는 컴파일러 최적화에 대한 추방으로 인해 사라졌습니다.

Store-to-load transformation은 컴파일러가 평범한 스토어가 메모리 상의 값을 실제로는 바꾸지 않을 것임을 알 때 일어날 수 있습니다. 예를 들어, Listing 4.22을 생각해 봅시다. 라인 1은 p를 읽어오지만, 라인 2의 “if” 문은 또한 컴파일러에게 개발자는 p가 일반적으로 0일 것이라 생각한다고 이야기 합니다.¹⁰ 라인 4의 barrier() 문은 컴파일러가 p의 값을 잊게 만듭니다만, 원한다면 컴파일러가 이 힌트를 기억하게 할 수도—또는 피드백 방향 최적화를 통해 추가적인 힌트를 줄 수도 있습니다. 그렇게 하는 것은 컴파일러에게 라인 5이 종종 비싸기만 한 no-op 일 뿐임을 알게 해줍니다.

따라서 그런 컴파일러는 Listing 4.23의 라인 5과 6에서 보인 것처럼 NULL 저장을 추가적 검사를 통해 보호할 수도 있습니다. 이 변형은 종종 바람직하지만, 순서를 위해 실제 스토어가 필요했다면 문제가 될 수도 있습니다. 예를 들어, 쓰기 메모리 배리어 (리눅스 커널의 smp_wmb()) 가 스토어를 순서잡아줄 수는 있으나, 로드에 대해서는 그렇지 않습니다. 이 상황은 smp_wmb()를 넘어서 smp_store_release()를 사용할 것을 제안할 수도 있습니다.

Dead-code elimination은 어떤 로드를 통해 얻어진 값이 절대 사용되지 않음을, 또는 어떤 값이 어딘가에 저장되지만 결코 로드 되지는 않음을 컴파일러가 알아챘을 때 발생할 수 있습니다. 이는 물론 공유 변수로의 액세스를 제거할 수 있어서, 결국 메모리 순서 기능을 방해해서, 여러분의 동시성 코드가 놀라운 방식으로 동작하게끔 만들어 버릴 수도 있습니다. 지금까지의 경험은 그런 놀라움들 중 상대적으로 적은 수의 것들만이 즐거운 것임을 이야기 합니다. 저장될 뿐인 변수의 제거는 외부 코드가 심볼 테이블을 통해 해당 변수를 찾아내는 경우에는 특히나 위험합니다: 컴파일러는 그런 외부

¹⁰ unlikely() 함수는 이 힌트를 컴파일러에게 전달하며, 다른 컴파일러는 unlockely()를 구현하는 다른 방법을 제공합니다.

코드 액세스를 무시해야 하며, 따라서 이 외부 코드가 의존하고 있는 변수를 없애버릴 수도 있습니다.

안정적인 동시성 코드는 컴파일러가 숫자, 순서, 그리고 공유 메모리로의 중요한 액세스의 타입을 보존하기 위한 방법이 분명 필요한데, 이 주제는 다음으로 이어지는 Section 4.3.4.2 와 4.3.4.3에서 다룹니다.

4.3.4.2 A Volatile Solution

지금은 비난을 받지만, C11 과 C++11 [Bec11] 의 발전 전에는 volatile 키워드는 병렬 프로그래머의 도구상자에서 필수적인 도구였습니다. 이는 volatile이 정확히 뭘 의미하는지에 대한 질문을 떠올리게하는데, 이 질문은 이 표준 [Smi19]의 더 최신 버전에서조차 충분히 꼼꼼하게 답변되지 않았습니다.¹¹ 이 버전은 “volatile glvalues 를 통한 액세스는 abstract machine 의 규칙에 근거해 엄격하게 평가된다”는 것을, volatile 액세스는 부수작용을 동반함을, 이것은 네개의 forward-progress 를 알리는 것임을, 그리고 이것들의 정확한 의미는 구현에 의해 정의됨을 보장합니다. 가장 깔끔한 가이드는 아마도 이 표준은 아닌 노트를 통해 제공될 수 있을 겁니다:

volatile은 어떤 객체의 값이 구현에 의해 서는 탐지될 수 없는 방법에 의해 변경될 수도 있기 때문에 해당 객체에 영향을 끼치는 적극적인 최적화를 방지하기 위해 구현에게 주는 힌트입니다. 더 나아가, 일부 구현을 위해서는 volatile은 해당 객체에 액세스 하기 위해 특수한 하드웨어 인스트럭션이 필요함을 알릴 수도 있습니다. 구체적 의미를 위해서 6.8.1을 참고하십시오. 일반적으로, volatile의 의미는 그것이 C에서 그러한 것처럼 C++에서도 그러할 것으로 의도됩니다.

이 문장들은 저수준 코드를 작성하는 사람들을 안심시킬텐데 컴파일러 작성자들은 표준이 아닌 노트를 완전히 무시해도 된다는 사실을 제외하면 그렇습니다. 병렬 프로그래머들은 대신 컴파일러 작성자들이 디바이스 드라이버를 망가뜨리는 것을 방지하고자 할 것을 통해 (비록 이는 디바이스 드라이버 개발자와의 “솔직하고 개방된” 토론을 좀 한 후에야 가능하겠지만요), 그리고 디바이스 드라이버는 최소한 다음의 제약을 가지고 있음을 [MWPF18] 통해 스스로를 대신 안심시킬 수도 있습니다:

1. 구현은 정렬된 volatile 액세스를 해당 액세스의 사이즈와 타입을 위한 기계 인스트럭션이 존재할 때

¹¹ JF Bastien은 C++에서 volatile 키워드의 역사와 사용 예에 대해 자세하게 문서화 했습니다 [Bas18].

Listing 4.24: Avoiding Danger, 2018 Style

```

1 ptr = READ_ONCE(global_ptr);
2 if (ptr != NULL && ptr < high_address)
3     do_low(ptr);

```

Listing 4.25: Preventing Load Fusing

```

1 while (!READ_ONCE(need_to_stop))
2     do_something_quickly();

```

분할시켜선 안된다.¹² 동시성 코드는 불필요한 로드와 스토어 분할시키기를 방지하기 위해 이 제약에 의존한다.

- 구현은 volatile 액세스의 의미에 대해 어떤 것도 가정해선 안되며, 값을 리턴하는 어떤 volatile access에 대해서는 리턴될 수도 있는 값의 가능한 집합에 대해서도 그러하다.¹³ 동시성 코드는 다른 프로세서가 동시에 같은 위치에 액세스하고 있을 수 있을 때 적용되어선 않아야 할 최적화를 방지하기 위해 이 제약에 의존한다.
- 정렬된 기계 워드 크기의 섞이지 않은 크기의 volatile access는 그 앞과 뒤의 volatile 어셈블리 코드 순서와 자연적으로 상호작용한다. 일부 기기는 volatile MMIO 액세스와 특수 목적 어셈블리어 인스트럭션의 조합을 통해서 액세스 될 것을 필요로 하기에 이 제약이 필요하다. 동시성 코드는 volatile 액세스와 Section 4.3.4.3에서 이야기 된 다른 방법들의 조합에서의 순서 속성을 이루기 위해 이 제약에 의존한다.

동시성 코드는 또한 모든 액세스가 정렬되었고 기계 워드 크기라는 가정 하에 특정 객체로의 액세스 중 어느 하나가 어토믹이 아니거나 volatile이 아닌 경우 데이터레이스에 의해 야기될 수 있는 정의되지 않은 동작을 막기 위해 앞의 두 제약에 의존할 수 있습니다. 같은 위치로의 섞인 크기의 액세스의 의미는 더 복잡한데, 일단 나중을 위해 설명하지 않겠습니다.

그래서 volatile은 앞의 예제들과 어떻게 엮일 수 있을까요?

Listing 4.14의 line 1에서 READ_ONCE()를 사용하는 것은 만들어진 로드를 방지하여, Listing 4.24에 보인 코드가 나오게 합니다.

Listing 4.25에서 보인 것처럼, READ_ONCE()는 Listing 4.17의 루프 풀기도 방지할 수 있습니다.

¹² 이는 128-bit CAS는 있지만 128-bit 로드와 스토어는 존재하지 않는 CPU에서 128-bit 로드와 스토어에 대해 어떻게 해야 하는지는 명시하지 않고 있음을 알아 두시기 바랍니다.

¹³ 이는 앞에서 이야기된 구현에 의해 정의되는 의미에 의해 합축되어 있습니다.

Listing 4.26: Preventing Store Fusing and Invented Stores

```

1 void shut_it_down(void)
2 {
3     WRITE_ONCE(status, SHUTTING_DOWN); /* BUGGY!!! */
4     start_shutdown();
5     while (!READ_ONCE(other_task_ready)) /* BUGGY!!! */
6         continue;
7     finish_shutdown();
8     WRITE_ONCE(status, SHUT_DOWN); /* BUGGY!!! */
9     do_something_else();
10 }
11
12 void work_until_shut_down(void)
13 {
14     while (READ_ONCE(status) != SHUTTING_DOWN) /* BUGGY!!! */
15         do_more_work();
16     WRITE_ONCE(other_task_ready, 1); /* BUGGY!!! */
17 }

```

Listing 4.27: Disinviting an Invented Store

```

1 if (condition)
2     WRITE_ONCE(a, 1);
3 else
4     do_a_bunch_of_stuff();

```

READ_ONCE()와 WRITE_ONCE()는 Listing 4.19에 보인 store fusing과 invented stores도 방지할 수 있어서, Listing 4.26가 되게 합니다. 하지만, 이는 코드 재배치를 방지하기 위한 일은 하지 않으므로, Section 4.3.4.3에서 배우게 될 추가적인 트릭이 좀 필요합니다.

마지막으로, WRITE_ONCE()는 Listing 4.20에 보인 store invention을 방지하는데 사용될 수 있어서, Listing 4.27에 보이는 코드가 되게 합니다.

요약하자면, volatile 키워드는 로드와 스토어가 기계 워드 크기이고 적절히 정렬되었을 때 load tearing과 store tearing을 방지합니다. 이것은 또한 load fusing, store fusing, invented loads, 그리고 invented stores를 방지합니다. 하지만, 이게 컴파일러가 volatile 액세스를 서로간에 재배치하는 것을 막기는 하나, CPU가 이 액세스들을 재배치하는 것을 막는 일은 전혀 하지 못합니다. 더 나아가서, 컴파일러나 CPU가 volatile이 아닌 액세스를 그것들 간에 또는 volatile 액세스와 재배치하는 것을 막는 일은 전혀 하지 않습니다. 이런 종류의 재배치를 막기 위해서는 다음 섹션에서 이야기 되는 기술이 필요합니다.

4.3.4.3 Assembling the Rest of a Solution

추가적인 순서 규칙은 전통적으로 어셈블리어를 통해 제공되어왔는데, 예를 들면 GCC asm 지시어가 있습니다. 이상하게 들리겠지만, 이 지시어들은 Listing 4.9의 barrier() 매크로에 의해 예시 되듯이 어셈블리어를 포함하지는 않습니다.

barrier() 매크로 내부에서 __asm__이 asm 지시어를 가져오며, __volatile__이 컴파일러가 asm을

Listing 4.28: Preventing C Compilers From Fusing Loads

```

1 while (!need_to_stop) {
2     barrier();
3     do_something_quickly();
4     barrier();
5 }
```

Listing 4.29: Preventing Reordering

```

1 void shut_it_down(void)
2 {
3     WRITE_ONCE(status, SHUTTING_DOWN);
4     smp_mb();
5     start_shutdown();
6     while (!READ_ONCE(other_task_ready))
7         continue;
8     smp_mb();
9     finish_shutdown();
10    smp_mb();
11    WRITE_ONCE(status, SHUT_DOWN);
12    do_something_else();
13 }
14
15 void work_until_shut_down(void)
16 {
17     while (READ_ONCE(status) != SHUTTING_DOWN) {
18         smp_mb();
19         do_more_work();
20     }
21     smp_mb();
22     WRITE_ONCE(other_task_ready, 1);
23 }
```

최적화 해서 없애버리는 걸 방지하고, 빈 문자열은 어떤 실제 명령도 생성될 필요가 없음을 명시하고, 마지막으로 "memory"는 컴파일러에게 이 아무것도 하지 말라는 asm은 메모리를 임의로 바꿀 수도 있음을 알립니다. 이에 대한 응답으로, 컴파일러는 이 `barrier()` 매크로 앞뒤로 메모리 참조들을 옮기는 것을 막을 겁니다. 이는 Listing 4.17에 보인 루프 풀어버리기가 Listing 4.28의 라인 2와 4에 의해 방지될 수 있음을 의미합니다. 이 두 줄의 코드는 컴파일러가 `need_to_stop`으로부터의 로드를 `do_something_quickly()` 내로 또는 그 뒤로 어떤 방향으로든 변경하는 것을 방지합니다.

하지만, 이는 CPU가 메모리 참조를 재배치하는 것을 방지하기 위한 어떤 대책도 내놓지 않습니다. 많은 경우에 이는 문제가 아닌데 하드웨어는 일정한 정도의 재배치만 할 수 있기 때문입니다. 하지만 하드웨어가 제약되어야만 하는 Listing 4.19 같은 경우들도 있습니다. Listing 4.26은 store fusing과 invention을 방지했으며, Listing 4.29은 더 나아가 `smp_mb()`를 라인 4, 8, 10, 18, 그리고 21에 추가함으로써 남아있는 재배치를 방지했습니다. `smp_mb()` 매크로는 Listing 4.9에 보인 `barrier()`와 비슷하지만 빈 문자열이 예를 들어 x86이라면 "mfence", PowerPC라면 "sync"와 같은 실제 명령을 담은 문자열로 대체되어 있습니다.

Quick Quiz 4.31: 하지만 완전한 메모리 배리어는 무척 무겁지 않나요? Listing 4.29에 필요한 순서만을 강제하는 더 가벼운 방법이 있지 않나요?

■

순서 규칙 강제는 일부 read-modify-write 어토믹 오퍼레이션들에 의해서도 제공되는데, 그 중 일부는 Section 4.3.5에서 보였습니다. 일반적인 경우에 메모리 순서 규칙 강제는 Chapter 15에서 이야기 되었듯 상당히 미묘할 수 있습니다. 다음 섹션은 메모리 순서 규칙 강제의 대안, 즉 데이터 레이스를 제한하거나 아예 방지하는 방법을 알아보겠습니다.

4.3.4.4 Avoiding Data Races

"Doctor, it hurts my head when I think about concurrently accessing shared variables!"

"Then stop concurrently accessing shared variables!!!"

이 의사의 조언은 도움이 되지 않는 듯 보일 수도 있겠습니다만, 공유된 변수들을 동시에 액세스하는 걸 막는 하나의 시간으로 검증된 방법은 그 변수들을 일반적인 락을 쥐고 있을 때에만 액세스 하는 것으로, Chapter 7에서 더 이야기 될 겁니다. 또 다른 방법은 특정 CPU나 쓰레드에서만 특정 "공유된" 변수를 액세스 하는 것으로, Chapter 8에서 더 이야기 하겠습니다. 이 두개의 방법을 융합하는 것도 가능한데, 예를 들면 특정 변수는 특정 CPU나 쓰레드에 의해서 어떤 락을 잡고 있을 때에만 변경될 수 있으며, 같은 CPU나 쓰레드에 의해서는 그냥 읽힐 수 있고 다른 CPU나 쓰레드에서는 그 락을 잡고 있을 때에만 읽을 수 있는 식입니다. 이 모든 상황에서, 해당 공유 변수로의 모든 액세스는 평범한 C-언어 액세스일 겁니다.

특정 변수로의 액세스가 평범한 로드와 스토어만으로도 가능한 상황들의 리스트가 여기 있는데, 해당 변수로의 다른 액세스들은 마킹이 (`READ_ONCE()`와 `WRITE_ONCE()` 같은) 필요합니다:

1. 어떤 공유 변수가 특정 소유권을 가진 CPU나 쓰레드에 의해서 변경되지만, 다른 CPU나 쓰레드에 의해 읽혀질 때. 모든 스토어는 `WRITE_ONCE()`를 사용해야 합니다. 그외의 모든 다른 것은 로드를 위해 `READ_ONCE()`를 사용해야만 합니다.
2. 어떤 공유 변수가 특정 락을 잡고서만 변경되지만, 해당 락을 쥐지 않은 다른 코드가 읽을 때. 모든 스토어는 `WRITE_ONCE()`를 사용해야 합니다. 해당 락을 쥐고 있는 CPU나 쓰레드는 평범한 로드를 사용할 수 있습니다. 그외의 모든 다른 것은 로드를 위해 `READ_ONCE()`를 사용해야만 합니다.

3. 어떤 공유 변수가 특정 락을 쥐고 있을 때에 소유권을 가진 CPU나 쓰레드에 의해서만 액세스되지만, 해당 락을 쥐지 않은 다른 CPU나 쓰레드나 코드에 의해서 읽혀질 때. 모든 스토어는 `WRITE_ONCE()`를 사용해야 합니다. 소유권을 가진 CPU나 쓰레드는 평범한 로드를 사용할 수 있으며, 다른 CPU나 쓰레드도 락을 쥐고 있다면 그렇습니다. 그외의 모든 다른 것은 로드를 위해 `READ_ONCE()`를 사용해야만 합니다.
4. 어떤 공유 변수가 특정 CPU나 쓰레드, 그리고 해당 CPU나 쓰레드의 컨텍스트에서 동작하는 시그널이나 인터럽트 핸들러에 의해서만 접근될 때. 이 핸들러는 평범한 로드와 스토어를 사용할 수 있는데, 해당 핸들러가 호출되는 것을 방지한 모든 다른 코드, 즉 시그널과 인터럽트를 막은 코드도 그렇습니다. 모든 다른 코드는 `READ_ONCE()`와 `WRITE_ONCE()`를 사용해야만 합니다.
5. 어떤 공유 변수가 특정 CPU나 쓰레드, 해당 CPU나 쓰레드의 컨텍스트에서 수행되는 시그널이나 인터럽트 핸들러에 의해서만 접근되며, 이 핸들러는 항상 자신이 값을 쓴 변수의 값을 기준의 것으로 리턴하기 전에 항상 복원시킬 때. 이 핸들러는 평범한 로드와 스토어를 사용할 수 있으며, 이 핸들러가 호출되는 것을 막은, 즉 시그널과 인터럽트를 막은 코드도 그렇습니다. 모든 다른 코드는 `READ_ONCE()`와 `WRITE_ONCE()`를 사용해야만 합니다.

Quick Quiz 4.32: 인터럽트나 시그널 핸들러가 그 스스로도 인터럽트 당할 수 있다면 뭘 해야 하나요?

■
대부분의 다른 경우에는 어떤 공유 변수로의 로드와 스토어는 `READ_ONCE()`와 `WRITE_ONCE()` 또는 그보다 강력한 것을 각각 사용해야만 합니다. 하지만 `READ_ONCE()`도 `WRITE_ONCE()`도 컴파일러 외의 것에 대해서는 어떠한 순서 보장도 제공하지 않음을 명심해 두시기 바랍니다. 그런 보장들에 대한 정보를 위해선 Section 4.3.4.3 또는 Chapter 15를 읽어보시기 바랍니다.

데이터 레이스 방지 패턴의 많은 예가 Chapter 5에 보여집니다.

4.3.5 Atomic Operations

리눅스 커널은 다양한 어토믹 오퍼레이션을 제공합니다만, `atomic_t` 타입에 대해 정의된 것들이 시작하기에 좋을 겁니다. 평범한 찢어지지 않는 (non-tearing) 로드와 스토어는 `atomic_read()`와 `atomic_set()`을 통해 각각 제공됩니다. `Acquire load`는 `smp_load_acquire()`를 통해, `release store`는 `smp_store_release()`를 통해 제공됩니다.

Listing 4.30: Per-Thread-Variable API

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

값을 리턴하지 않는 `fetch-and-add` 오퍼레이션은 `atomic_add()`, `atomic_sub()`, `atomic_inc()`, 그리고 `atomic_dec()`를 통해 제공됩니다. 오퍼레이션 결과 값이 0이 되면 이를 알리는 어토믹 값 감소 오퍼레이션은 `atomic_dec_and_test()`와 `atomic_sub_and_test()`를 통해 제공됩니다. 새로운 값을 리턴하는 어토믹 값 증가 오퍼레이션은 `atomic_add_return()`을 통해 제공됩니다. `atomic_add_unless()`와 `atomic_inc_not_zero()`는 둘 다 조건적 어토믹 오퍼레이션을 제공하는데, 해당 어토믹 변수의 원래 값이 명시된 값과 다르면 아무 일도 일어나지 않습니다 (이는 예를 들면 레퍼런스 카운터를 관리하는데 무척 편리합니다).

어토믹 값 교환 오퍼레이션은 `atomic_xchg()`를 통해 제공되고, 그 유명한 `compare-and-swap` (CAS) 오퍼레이션은 `atomic_cmpxchg()`를 통해 제공됩니다. 이것들 둘 다 기존 값을 리턴합니다. 많은 추가적 어토믹 RMW 기능들이 리눅스 커널 내에 존재하는데, 리눅스 소스 트리의 `Documentation/atomic_t.txt` 파일을 읽어보시기 바랍니다.

이 책의 `CodeSamples` API는 리눅스 커널의 것에 가깝습니다.

4.3.6 Per-CPU Variables

리눅스 커널은 per-CPU 변수를 정의하기 위해 `DEFINE_PER_CPU()`를, 해당 per-CPU 변수의 이 CPU의 인스턴스로의 참조를 형성하기 위해 `this_cpu_ptr()`를, 해당 per-CPU 변수의 특정 CPU의 인스턴스로의 액세스를 위해 `per_cpu()`를 사용하며, 다른 특수 목적 per-CPU 오퍼레이션들도 제공합니다.

Listing 4.30은 리눅스 커널의 per-CPU-변수 API의 패턴을 따라한 이 책의 per-thread-variable API를 보입니다. 이 API는 전역 변수의 per-thread 버전을 제공합니다. 이 API는 엄밀하게 말하면 필요치 않지만¹⁴, 리눅스 커널 코드의 유저스페이스로의 비슷한 좋은 것을 제공할 수 있습니다.

Quick Quiz 4.33: Per-thread-변수 API가 존재하지 않는 시스템에선 이걸 어떻게 할 수 있을까요?

¹⁴ 여러분은 이것 대신 `_thread`나 `_Thread_local`을 사용할 수 있습니다.

4.3.6.1 DEFINE_PER_THREAD()

DEFINE_PER_THREAD() 기능은 per-thread 변수 하나를 정의합니다. 불행히도, 리눅스 커널의 DEFINE_PER_THREAD() 기능에서는 가능한 것처럼 초기화를 제공하는 것은 불가능합니다만, 쉬운 런타임 초기화를 가능하게 해주는 init_per_thread() 기능이 있습니다.

4.3.6.2 DECLARE_PER_THREAD()

DECLARE_PER_THREAD() 기능은 C 센스의 정의가 아닌 선언입니다. 따라서, DECLARE_PER_THREAD() 기능은 어떤 다른 파일에서 정의된 per-thread 변수를 접근하기 위해 사용될 수 있습니다.

4.3.6.3 per_thread()

per_thread() 기능은 특정 쓰레드의 변수에 액세스 합니다.

4.3.6.4 __get_thread_var()

__get_thread_var() 기능은 현재 쓰레드의 변수를 액세스 합니다.

4.3.6.5 init_per_thread()

init_per_thread() 기능은 특정 변수의 모든 쓰레드의 인스턴스를 특정 값으로 설정합니다. 리눅스 커널은 이를 링커 스크립트의 현명한 사용과 CPU-online 과정에서의 코드 실행을 통해 평범한 C 초기화로 해냅니다.

4.3.6.6 Usage Example

매우 자주 읽히지만 가끔만 증가되는 카운터가 하나 있다고 생각해 봅시다. Section 5.2에서 명확해 지겠지만, 그런 카운터는 per-thread 변수를 사용해 만드는게 도움 됩니다. 그런 변수는 다음과 같이 정의될 수 있습니다:

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

이 카운터는 아래와 같이 초기화 되어야 합니다:

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

어떤 쓰레드는 이 카운터의 자신의 인스턴스를 다음과 같이 증가시킬 수 있습니다:

```
p_counter = &__get_thread_var(counter);
WRITE_ONCE(*p_counter, *p_counter + 1);
```

이 카운터의 값이 이것의 인스턴스들의 값의 합입니다. 따라서 이 카운터의 값의 한 스냅샷은 다음과 같이 모아질 수 있습니다:

```
for_each_thread(t)
    sum += READ_ONCE(per_thread(counter, t));
```

다시 말하지만, 다른 메커니즘을 통해 비슷한 효과를 얻을 수 있습니다. 하지만 per-thread 변수는 편의성과 고성능을 함께 제공하는데, Section 5.2에서 더 자세히 알아보겠습니다.

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

If you get stuck, change your tools; it may free your thinking.

Paul Arden, abbreviated

간단한 경험에 의한 규칙으로써, 지금 해야 하는 일을 해주는 가장 간단한 도구를 사용하십시오. 할 수 있다면, 순차적 프로그램을 하십시오. 그게 부족하다면, 병렬성을 조절하기 위해 셸 스크립트를 사용해 보세요. 그로 인한 셸 스크립트의 fork()/exec() 오버헤드가 (Intel Core Duo 랩톱에서의 가장 작은 C 프로그램이라면 약 480 마이크로세컨드가 소요됩니다) 너무 크다면, C-언어의 fork() 와 wait() 기능을 시도해 보세요. 그 오버헤드가 (최소한의 차일드 프로세스에 대해 대략 80 마이크로세컨드) 여전히 너무 크다면, 적절한 럭킹과 어토믹 오퍼레이션 기능과 함께 POSIX 쓰레드 기능을 사용해야 할 수도 있습니다. POSIX 쓰레딩 기능의 오버헤드가 (일반적으로 1 마이크로세컨드 미만) 여전히 크다면, Chapter 9에서 소개되는 기능들이 필요할 수도 있습니다. 물론, 실제 오버헤드는 여러분의 하드웨어만이 아니라 여러분이 그 기능들을 사용하는 방식에 의존될 겁니다. 더 나아가서, 프로세스간 커뮤니케이션과 메세지 패싱은 공유 메모리 멀티 쓰레딩 수행의 좋은 대안이 될 수 있으며, 특히 여러분의 코드가 Chapter 6에서 이야기 되는 설계 원칙을 잘 따른다면 그려함을 항상 기억하세요.

Quick Quiz 4.34: 셸은 fork() 대신 vfork()를 사용하지 않을까요?

■ 동시성은 C 언어가 처음으로 동시성 시스템을 만드는데 사용된지 수십년이 지나서 C 표준에 추가되었기 때문에, 공유 변수를 동시에 접근하는 여러 방법이 존재합니다. 다른게 같다면, Section 4.2.6에서 이야기 되는

C11 표준 오퍼레이션이 첫번째 선택지가 되어야 합니다. 특정 공유 변수를 평범한 액세스로도 어토믹하게도 접근할 수 있어야 한다면, Section 4.2.7에서 이야기 된 현대의 GCC 어토믹이 여러분을 위해 잘 동작할 수도 있습니다. 고전의 GCC `__sync` API를 사용하는 오래된 코드베이스 위에서 일하고 있다면, Section 4.2.5와 연관된 GCC 문서들을 읽어야 합니다. 리눅스 커널이나 `volatile` 키워드를 인라인 어셈블리와 결합해 사용하는 비슷한 코드베이스에서 작업 중이라면, 또는 순서를 제공하기 위한 의존성이 필요하다면, Section 4.3.4과 Chapter 15에서 나온 것들을 보시기 바랍니다.

여러분이 어떤 방법을 택하든, 멀티 쓰레드 코드를 무작위로 해킹하는 것은 무척이나 나쁜 생각인데, 특히 공유 메모리 병렬 시스템은 여러분의 인지 능력을 여러분에게 사용함을 놓고 보면 그렇습니다. 여러분이 더 똑똑할 수록, 여러분은 여러분이 문제에 직면해 있다는 것을 알기 전까지 더 깊은 구멍을 스스로에게 파고 있을 겁니다 [Pok16]. 따라서, 뒤따르는 챕터들에서 이야기 하겠지만 개별 기능에 대한 옳은 선택은 물론 올바른 설계 선택을 하는게 필요합니다.

Chapter 5

Counting

카운팅은 아마도 컴퓨터가 할 수 있는 가장 간단하고도 자연스러운 일일 겁니다. 하지만, 거대한 공유 메모리 멀티프로세서에서 효율적이고도 확장성 있게 카운팅을 하는 것은 상당히 어렵습니다. 더 나아가서, 카운팅에 내재하는 개념의 간단함은 우리가 잘 발달된 데이터 구조나 복잡한 동기화 도구들의 방해 없이 동시성의 근본적 문제들을 탐험할 수 있게 해줍니다. 따라서 카운팅은 병렬 프로그래밍으로의 훌륭한 소개를 제공합니다.

이 챕터는 간단하고, 빠르고, 확장성 있는 카운팅 알고리즘들이 존재하는 여러개의 특수한 경우들을 다룹니다. 하지만 먼저, 여러분이 동시적 카운팅에 대해 얼마나 알고 있는지 먼저 알아봅시다.

Quick Quiz 5.1: 효율적이고도 확장성 있는 카운팅이 왜 어려워야 하죠??? 무엇보다도, 컴퓨터는 카운팅이라는 하나의 목적을 위한 특수 하드웨어를 가지고 있잖아요!!!

Quick Quiz 5.2: 네트워크 패킷 카운팅 문제. 여러분이 송수신된 네트워크 패킷의 갯수에 대한 통계를 수집해야 한다고 해봅시다. 패킷은 시스템 상의 어떤 CPU를 통해서든 송수신 될 수도 있습니다. 더 나아가서 여러분의 시스템이 CPU마다 초당 수백만개 이상의 패킷을 처리할 수 있다고, 그리고 그 수를 5초마다 세는 시스템 모니터링 패키지가 있다고 해봅시다. 여러분은 이 카운터를 어떻게 구현하겠습니까?

Quick Quiz 5.3: 대략적 구조체 할당 한계 문제. 어떤 구조체의 할당이 그 구조체의 수가 어떤 한계 (예를 들어 10,000)를 넘어섰을 때 실패하도록 하거나 하기 위해 할당된 구조체의 갯수를 유지해야 한다고 생각해봅시다. 더 나아가서 이 구조체는 수명이 짧아서, 이 한계는 아주 가끔만 넘어서게 되며, “약간 허술한” 대략적 한계가 허용된다고 생각해 봅시다.

Quick Quiz 5.4: 정확한 구조체 할당 한계 문제. 어떤 구조체의 할당이 그 구조체의 수가 정확한 한계 (예를

들어 10,000)를 넘어섰을 때 반드시 실패하도록 하기 위해 할당된 구조체의 갯수를 유지해야 한다고 생각해 봅시다. 더 나아가서 이 구조체는 수명이 짧고, 이 한계는 아주 가끔만 초과되어서, 거의 항상 하나의 구조체만이 실제로 사용되고 있고, 더 나아가서 이 카운터가 정확히 언제 0이 되는지, 예를 들어 그 구조체가 단 하나도 사용되고 있지 않다면 어떤 메모리를 해제하거나 하기 위해 이 카운터가 정확히 언제 0이 되는지 알아야 한다고 해봅시다.

Quick Quiz 5.5: 제거 가능한 I/O 기기 액세스 카운트 문제. 많이 사용되는 제거 가능한 대용량 저장장치의 레퍼런스 카운트를 유지해야 한다고 해봅시다, 여러분이 사용자에게 이 기기를 제거하는게 언제 안전한지 말하기 위해서요. 평범한 경우처럼, 이 사용자는 이 기기를 제거하고자 하는 의도를 알릴 것이고, 시스템은 이 사용자에게 그게 언제 안전한지 알려줘야 합니다.

Section 5.1은 왜 카운팅이 사소하지 않은지 보입니다. Sections 5.2 and 5.3은 각각 네트워크 패킷 카운팅과 대략적 구조체 할당 한계를 분석합니다. Section 5.4는 정확한 구조체 할당 한계를 다룹니다. 마지막으로, Section 5.5는 성능 측정과 기타 토론을 제공합니다.

Sections 5.1 and 5.2는 초반 소개를 담고 있으며, 뒤따르는 섹션들은 좀 더 고급 주제를 다룹니다.

5.1 Why Isn't Concurrent Counting Trivial?

Seek simplicity, and distrust it.

Alfred North Whitehead

일단 뭔가 간단한, 예를 들어 Listing 5.1 (`count_nonatomic.c`)에 보여진 간단한 수식으로 시작해 봅시

Listing 5.1: Just Count!

```

1 unsigned long counter = 0;
2
3 static __inline__ void inc_count(void)
4 {
5     WRITE_ONCE(counter, READ_ONCE(counter) + 1);
6 }
7
8 static __inline__ unsigned long read_count(void)
9 {
10    return READ_ONCE(counter);
11 }

```

Listing 5.2: Just Count Atomically!

```

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

```

다. 여기서, 우린 라인 1에 카운터를 가지고 있고 라인 5에서 이를 증가시키며, 라인 10에서 그 값을 읽습니다. 뭐가 더 간단할 수 있을까요?

Quick Quiz 5.6: 한가지 더 간단할 수 있는 것은 READ_ONCE() 와 WRITE_ONCE() 를 함께 사용하는 대신 더 간단한 `++` 를 사용하는 것일 수 있습니다. 무엇 때문에 그렇게 추가적인 타이핑을 하죠???

이 방법은 여러분이 읽기를 많이 하고 값 증가는 거의 하지 않는다면 무척 빠를 것이고, 작은 시스템에서라면 성능이 훌륭할 겁니다.

여기 하나의 큰 문제가 있습니다: 이 방법은 카운트를 잃을 수 있습니다. 제 여섯개 코어가 있는 x86 랩톱에서, `inc_count()` 를 285,824,000 번 수행시켰습니다만, 이 카운터의 최종값은 35,385,525 뿐이었습니다. 대략적인 정확성이 컴퓨팅에서 큰 위치를 차지하긴 하지만, 87%의 카운트 손실은 약간 지나칩니다.

Quick Quiz 5.7: 하지만 영리한 컴파일러는 Listing 5.1 의 라인 5 이 `++` 연산자와 동일함을 알아차리고 x86 add-to-memory 인스트럭션을 생성하지 않을까요? 그리고 CPU 캐쉬는 이를 어토믹하게 만들지 않을까요?

Quick Quiz 5.8: 실패 횟수의 8-figure accuracy 는 당신이 이걸 진짜로 테스트 했음을 알립니다. 특히나 버그가 검사하는 것만으로 쉽게 보일 수 있는 이런 경우에 이런 사소한 프로그램을 테스트할 필요가 있을까요?

정확한 카운팅을 하는 간단한 방법은 Listing 5.2 (`count_atomic.c`)에 보여진 것처럼 어토믹 오퍼레이

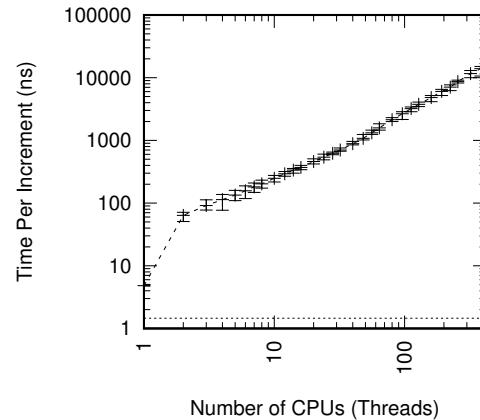


Figure 5.1: Atomic Increment Scalability on x86

션을 사용하는 것입니다. 라인 1은 어토믹 변수를 정의하고, 라인 5는 이를 어토믹하게 증가시키며, 라인 10는 이를 읽습니다. 이것은 어토믹 하므로, 완전한 카운트를 유지합니다. 하지만, 더 느립니다: 제 6개 코어 x86 랩톱에서, 어토믹하지 않은 값 증가 버전에 비해, 단일 쓰레드만 사용될 때 조차도 20배가 넘게 느립니다.¹

이 치참한 성능은 Chapter 3에서의 토의를 생각해 보면 놀랍지 않은 것이고, 어토믹 값 증가의 성능이 CPU와 쓰레드의 수가 증가함에 따라 Figure 5.1에 보여진 것처럼 더 느려지는 것도 놀라운 일이 아닙니다. 이 그림에서, x 축의 가로로 그려진 점선은 완전하게 확장되는 알고리즘이 얻을 수 있을 이상적 성능입니다: 그런 알고리즘이 있다면, 하나의 값 증가는 싱글쓰레드 프로그램에서의 것과 동일한 오버헤드만을 일으킬 겁니다. 단일 전역 변수의 어토믹 값 증가는 분명 이상적이지 않고, 추가되는 CPU에서 수천수만배의 오버헤드를 일으킵니다.

Quick Quiz 5.9: x 축 상의 가로의 점선은 왜 $x = 1$ 에서 대각선에 붙지 않나요?

Quick Quiz 5.10: 하지만 어토믹 값 증가는 여전히 무척 빠릅니다. 그리고 짧은 반복문 내에서 하나의 변수를 값 증가시키는 것은 제게 굉장히 비현실적으로 들리는데, 어쨌건, 대부분의 프로그램의 실행은 진짜 일을 하는데 사용되어야지, 자신이 한 일을 세는데 쓰이면 안됩니다! 왜 제가 이걸 빠르게 하는데 신경을 써야 하죠?

¹ 흥미롭게도, 어토믹하지 않게 카운터를 증가시키는 것은 어토믹하게 이 카운터를 증가시키는 것보다도 빠른 속도로 그 값을 증가시킵니다. 물론, 여러분의 목표가 오직 이 카운터를 빨리 증가시키려는 거라면, 더 쉬운 방법은 그냥 큰 값을 이 카운터에 할당하는 것일 겁니다. 하지만, 더 큰 성능과 확장성을 위해 완화된 정확성의 개념을 주의 깊게 사용하는 알고리즘의 역할도 있을 수 있을 겁니다 [And91, ACMS03, Rin13, Ung11].

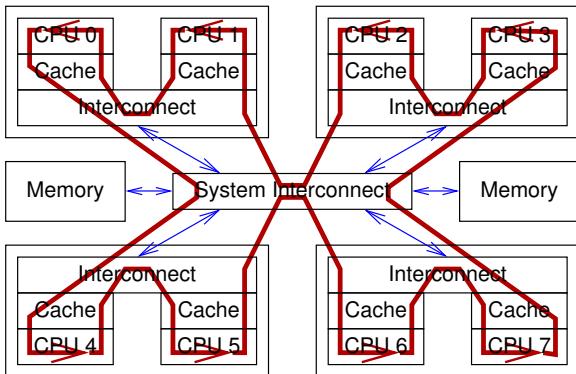


Figure 5.2: Data Flow For Global Atomic Increment

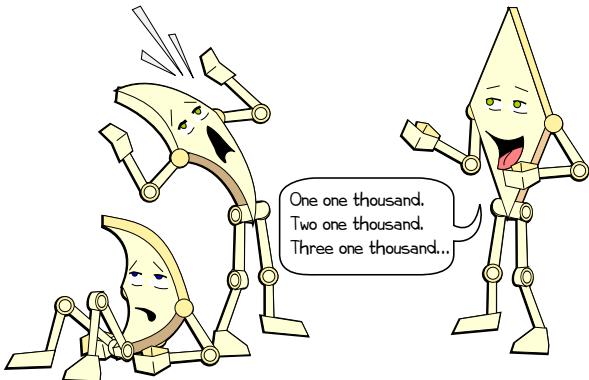


Figure 5.3: Waiting to Count

전역 어토믹 값 증가에 대한 다른 관점을 위해, Figure 5.2 를 보시기 바랍니다. 각 CPU 가 주어진 전역 변수의 값을 증가할 기회를 얻기 위해, 해당 변수를 담고 있는 캐쉬 라인은 빨간 화살표로 보여진 것처럼 모든 CPU 사이를 순환해야만 합니다. 그런 순환은 상당한 시간을 취할 것이어서, Figure 5.1 에 보여진 처참한 성능에 이를 것으로, Figure 5.3 처럼 생각될 수 있겠습니다. 다음 섹션들에서는 이런 순환에서 피할 수 없는 지연을 회피하기 위한 고성능 카운팅에 대해 이야기 해봅니다.

Quick Quiz 5.11: 하지만 왜 CPU 설계자들은 값이 증가되어야 하는 전역 변수를 담고 있는 캐쉬 라인이 순환될 필요를 없애는 추가적인 기능을 제공하지 않는 거죠?



Listing 5.3: Array-Based Per-Thread Statistical Counters

```

1 DEFINE_PER_THREAD(unsigned long, counter);
2
3 static __inline__ void inc_count(void)
4 {
5     unsigned long *p_counter = &__get_thread_var(counter);
6
7     WRITE_ONCE(*p_counter, *p_counter + 1);
8 }
9
10 static __inline__ unsigned long read_count(void)
11 {
12     int t;
13     unsigned long sum = 0;
14
15     for_each_thread(t)
16         sum += READ_ONCE(per_thread(counter, t));
17
18     return sum;
19 }
```

5.2 Statistical Counters

Facts are stubborn things, but statistics are pliable.

Mark Twain

이 섹션은 카운트가 무척 자주 업데이트 되고 그 값은 가끔만 읽혀지는, 통계적 카운터라는 흔한 구체적 케이스를 다룹니다. 이는 Quick Quiz 5.2 에서 암시된 네트워크 패킷 카운팅 문제를 푸는데 사용될 겁니다.

5.2.1 Design

통계적 카운팅은 일반적으로 쓰레드당 (또는 커널에서 돌아가는 경우라면 CPU 당) 카운터를 제공해서 page 46 의 Section 4.3.6 에서 앞서 보여진 것처럼 각 쓰레드가 자신의 카운터를 증가시키도록 합니다. 카운터들의 합쳐진 값은 간단히 이 쓰레드들의 카운터를 모두 합해서 구해지는데, 더하기의 상호성과 결합성에 의존합니다. 이는 page 85 의 Section 6.3.4 에서 소개된 데이터 소유권 패턴의 한 예입니다.

Quick Quiz 5.12: 하지만 C 의 “정수형” 이 크기 제한이 있다는 사실이 이를 복잡하게 만들지는 않나요?



5.2.2 Array-Based Implementation

쓰레드당 변수를 제공하는 한가지 방법은 (거짓 공유를 방지하기 위해 캐쉬라인에 맞춰 정렬되고 패딩 되어 있다는 가정 하에) 쓰레드당 하나의 원소를 갖는 배열을 할당하는 것입니다.

Quick Quiz 5.13: 배열이요??? 하지만 그럼 쓰레드의 수가 제한되지 않나요?



그런 배열은 Listing 5.3 (count_stat.c)에 보여진 per-thread 기능으로 감싸질 수 있습니다. 라인 1은 창의 적이게도 counter 라고 이름지어진, unsigned long 타입의 쓰레드당 카운터의 집합을 담는 배열을 정의합니다.

라인 3-8는 `__get_thread_var()` 기능을 현재 수행 중인 쓰레드의 counter 배열 내 원소 위치를 알아내기 위해 사용해서 카운터를 증가시키는 함수를 보입니다. 이 원소는 이 연관된 쓰레드에 의해서만 증가되므로, 어 토믹하지 않은 값 증가로도 충분합니다. 하지만, 이 코드는 위험한 컴파일러 최적화를 방지하기 위해 `WRITE_ONCE()`를 사용합니다. 한가지 예만 들자면, 이 컴파일러는 저장되어질 위치를 임시의 저장소로 사용할 권리를 갖고 있으므로, 이 위치에 어떤 의도와 목적으로든 쓰레기를 이 요청된 스토어 이전에 이 위치에 저장할 수 있습니다. 이는 당연하게도 이 카운트를 읽으려는 모든 시도를 혼란하게 만들 수 있습니다. `WRITE_ONCE()`의 사용은 이 최적화와 다른 것들을 방지해 줍니다.

Quick Quiz 5.14: GCC는 이것 외에 어떤 못된 최적화를 할 수 있나요?

라인 10-18는 이 카운터의 합쳐진 값을 읽어들이는데, 현재 수행 중인 쓰레드의 리스트를 순회하는데에 `for_each_thread()` 기능을 사용하고, 특정 쓰레드의 카운터를 읽어들이기 위해 `per_thread()` 기능을 사용합니다. 이 코드는 또한 컴파일러가 이 로드를 무시하는 최적화를 하지 못하게 `READ_ONCE()`를 사용합니다. 한가지만 예를 들자면, `read_count()`로의 이어지는 두개의 호출은 인라인될 수도 있고, 대담한 최적화는 같은 위치가 더하기 되었으며 따라서 이것들을 한번만 더하기하고 그 결과 값을 두번 사용하는게 더 간단하고 나을 수 있을 거라는 잘못된 결론을 내릴 수 있습니다. 이런 종류의 최적화는 나중의 `read_count()` 호출이 다른 쓰레드의 활동을 계산할 거라 예상하는 사람들을 좌절하게 만들 수도 있습니다. `READ_ONCE()`의 사용은 이 최적화와 비슷한 것들을 방지합니다.

Quick Quiz 5.15: Listing 5.3의 counter per-thread 변수는 어떻게 초기화 되나요?

Quick Quiz 5.16: Listing 5.3의 코드는 복수의 카운터를 어떻게 허용할까요?

이 방법은 `inc_count()`를 수행하는 업데이트 쓰레드의 수의 증가와 함께 선형적으로 확장됩니다. Figure 5.4의 각 CPU에 초록 화살표로 보여진 것처럼, 이에 대한 이유는 각 CPU가 비싼 시스템 간 통신 없이 자신의 쓰레드의 변수를 값 증가시키는데 빠른 진행을 만든다는 것입니다. 그것으로서, 이 섹션은 이 챕터의 시작에서 이야기 된 네트워크 패킷 카운팅 문제를 해결합니다.

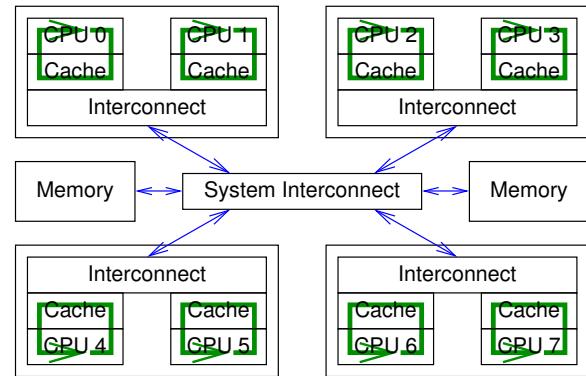


Figure 5.4: Data Flow For Per-Thread Increment

Quick Quiz 5.17: 이 읽기 오ペ레이션은 쓰레드당 값 을 합하는데 시간을 요하며, 이 시간 동안 이 카운터의 값은 변화될 수 있습니다. 이는 Listing 5.3의 `read_count()`를 통해 리턴되는 값은 정확하지 않을 것을 의미합니다. 이 카운터는 단위 시간당 r 카운트의 속도로 증가된다고, 그리고 `read_count()`의 수행은 단위 시간을 소모한다고 가정해 봅시다. 리턴되는 값에 예상되는 에러는 얼마정도일까요?

하지만, 많은 구현이 임의적 배열 크기 제한으로부터 자유로우며 훨씬 가격이 싼 쓰레드당 데이터 메커니즘을 제공합니다. 이게 다음 섹션의 주제입니다.

5.2.3 Per-Thread-Variable-Based Implementation

GCC는 쓰레드별 저장소를 제공하는 `__thread` 저장소 클래스를 제공합니다. 이는 잘 확장되고 임의의 쓰레드 수 제한을 회피할 뿐만 아니라 간단한 어토믹하지 않은 값 증가에 비해 적거나 아예 없는 성능 페널티를 갖는 통계적 카운터를 구현하는데 Listing 5.4 (count_end.c)에 보여진 것처럼 사용될 수 있습니다.

라인 1-4는 필요한 변수들을 정의합니다: `counter`는 쓰레드별 카운터 변수이고, `counters[]` 배열은 쓰레드들이 서로의 카운터를 접근할 수 있게 하며, `finalcount`는 각 쓰레드가 종료될 때마다 전체 카운터를 담게 되며, `final_mutex`은 카운터의 전체 값을 계산하는 쓰레드와 종료되는 쓰레드의 순서를 조정하는 데에 사용됩니다.

Quick Quiz 5.18: Listing 5.4의 명시적인 `counters` 배열은 쓰레드 수에 대한 임의의 제한을 다시 암시하지 않나요? 왜 GCC는 쓰레드들이 서로의 쓰레드별 변수를 쉽게 접근할 수 있게끔 리눅스 커널의 `per_cpu()`

Listing 5.4: Per-Thread Statistical Counters

```

1 unsigned long __thread counter = 0;
2 unsigned long *counterp[NR_THREADS] = { NULL };
3 unsigned long finalcount = 0;
4 DEFINE_SPINLOCK(final_mutex);
5
6 static __inline__ void inc_count(void)
7 {
8     WRITE_ONCE(counter, counter + 1);
9 }
10
11 static __inline__ unsigned long read_count(void)
12 {
13     int t;
14     unsigned long sum;
15
16     spin_lock(&final_mutex);
17     sum = finalcount;
18     for_each_thread(t)
19         if (counterp[t] != NULL)
20             sum += READ_ONCE(*counterp[t]);
21     spin_unlock(&final_mutex);
22     return sum;
23 }
24
25 void count_register_thread(unsigned long *p)
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 }

```

기능과 비슷한 `per_thread()` 인터페이스를 제공하지 않나요?

업데이트 쓰레드에 의해 사용되는 `inc_count()` 함수는 라인 6~9에서 볼 수 있듯이 상당히 간단합니다.

읽기 쓰레드에 의해 사용되는 `read_count()` 함수는 약간 복잡합니다. 라인 16는 종료되는 쓰레드를 제외시키기 위해 락을 잡고, 라인 21에서 해제합니다. 라인 17은 이미 종료된 쓰레드에 의해 계산된 카운트의 합을 초기화시키고, 라인 18~20는 현재 수행 중인 쓰레드에 의해 계산되는 카운트를 합합니다. 마지막으로, 라인 22은 이 합을 리턴합니다.

Quick Quiz 5.19: Listing 5.4의 라인 19에서의 NULL 체크는 추가적인 브랜치 예측 실패를 일으키지 않나요? 영구히 0 값을 갖는 변수 집합을 갖고 사용되지 않은 카운터 포인터를 NULL로 만드는 대신 그 변수를 가리키게 하는건 어떤가요?

Quick Quiz 5.20: Listing 5.4의 `read_count()`에서의 합산을 보호하는 `lock` 같이 무거운 게 도대체 왜 필요한거죠?

라인 25~32는 각 쓰레드에 의해 이 카운터를 처음 사용하기 전에 반드시 호출되어야 하는 `count_register_thread()` 함수를 보입니다. 이 함수는 단순히 `counterp[]` 배열 내 이 쓰레드의 원소가 자신의 쓰레드별 `counter` 변수를 가리키도록 합니다.

Quick Quiz 5.21: Listing 5.4의 `count_register_thread()`에서 대체 왜 락을 잡아야만 하죠? 이것은 어떤 다른 쓰레드도 수정하지 않는 위치에 적절히 정렬되어 저장된 하나의 기계 단어이니, 어쨌건 어토믹할 거예요, 그렇죠?

라인 34~42는 앞서 `count_register_thread()`를 호출한 쓰레드들은 종료되기 전에 반드시 호출해야 하는 `count_unregister_thread()` 함수를 보입니다. 라인 38는 락을 잡고, 라인 41에서는 해제해서, `read_count()`로의 호출은 물론 `count_unregister_thread()`로의 다른 호출들도 배제시킵니다. 라인 39는 이 쓰레드의 `counter`를 전역 변수인 `finalcount`에 더하며, 이어서 라인 40은 자신의 `counterp[]` 배열 내 원소를 NULL로 만듭니다. 이어지는 `read_count()` 호출은 이 종료되는 쓰레드의 카운트를 전역 변수 `finalcount`에서 보게 될 것이며, `counterp[]` 배열을 들여다 볼 때, 이 종료되는 쓰레드의 것은 건너뛰어서 올바른 전체 값을 얻게 될 겁니다.

이 방법은 업데이트 쓰레드에게 어토믹이 아닌 더하기와 거의 똑같은, 또한 선형적으로 확장하는 성능을 제공합니다. 다른 한편, 동시의 읽기들은 하나의 전역 락을 위해 경쟁하며, 따라서 처참한 성능과 지옥 같은 확장성을 가질 겁니다. 하지만, 이는 값 증가는 자주 일어나지만 읽기는 거의 일어나지 않는 통계적 카운터에서는 문제가 아닐 겁니다. 물론, 이 방법은 배열 기반의 방법에 비해 상당히 복잡한데, 특정 쓰레드의 쓰레드별 변수는 해당 쓰레드가 종료될 때 사라진다는 사실 때문입니다.

Quick Quiz 5.22: 좋아요, 하지만 리눅스 커널은 CPU 별 카운터의 값의 합을 읽을 때 락을 잡을 필요가 없죠. 그런데 왜 유저 스페이스 코드는 이걸 해야만 하죠???

배열 기반의 방법도 `__thread` 기반의 방법도 훌륭한 업데이트 쪽 성능과 확장성을 제공합니다. 하지만, 이 이익은 큰 수의 쓰레드들 하에서는 읽기 쪽의 비용을 초래합니다. 다음 섹션은 여전히 업데이트 쪽의 확장성을 유지하면서 읽기 쪽의 비용을 줄이는 한가지 방법을 알아봅니다.

5.2.4 Eventually Consistent Implementation

업데이트 쪽 확장성을 유지하면서 읽기 쪽 성능을 크게 개선하는 한가지 방법은 일관성 요구사항을 약화시키는 것입니다. 이전 섹션에서의 카운팅 알고리즘은 `read_count()`의 수행 시작 시점과 종료 시점에서 이상적 카운터가 가질 값들의 중간 어딘가의 값을 낼 것이 보장되어 있습니다. *Eventual consistency* [Vog09] (결과적 일관성)은 더 약한 보장을 제공합니다: `inc_count()` 호출이 없다면, `read_count()`의 호출은 언젠가는 정확한 카운트를 반환합니다.

우리는 전역 카운터를 가짐으로써 결과적 일관성을 제공합니다. 하지만, 업데이트 쓰레드는 각자의 쓰레드 별 카운터만을 업데이트 합니다. 이 쓰레드별 카운터의 값을 전역 카운터로 옮기는 별도의 쓰레드 하나가 제공됩니다. 읽기 쓰레드는 단순히 이 전역 카운터의 값에만 액세스 합니다. 만약 업데이트 쓰레드들이 활동 중이라면, 이 읽기 쓰레드에 의해 사용되는 값은 낡아버린 벼랑일 겁니다만, 일단 업데이트가 모두 끝나면, 이 전역 카운터는 언젠가 결과적으로는 실제 값에 이를 것입니다—따라서 이 방법은 결과적 일관성이라 할 수 있습니다.

이 구현이 Listing 5.5 (`count_stat_eventual.c`)에 보여져 있습니다. 라인 1-2는 앞서 이야기한, 카운터의 값을 따라가는 쓰레드별 변수와 전역 변수를 보이며, 라인 3은 (정확한 카운터 값과 함께 프로그램을 종료하고자 하는 경우를 위해) 종료를 조정하는 `stopflag`를 보입니다. 라인 5-10에 보인 `inc_count()` 함수는 Listing 5.3에 있는 것과 비슷합니다. 라인 12-15에 보인 `read_count()` 함수는 단순히 `global_count` 변수의 값을 반환합니다.

하지만, 라인 36-46에 보인 `count_init()` 함수는 라인 17-34에 보인, 모든 쓰레드를 순회하며 쓰레드별 지역 `counter`의 값을 합해서 `global_count` 변수에 저장하는 `eventual()` 쓰레드를 만듭니다. 이 `eventual()` 쓰레드는 임의로 선택된 1 밀리세컨드를 매 패스마다 기다립니다.

라인 48-54에 보인 `count_cleanup()` 함수는 종료를 조정합니다. 여기서와 `eventual()`에서의 `smp_mb()` 호출은 `global_count`로의 모든 업데이트가 `count_cleanup()` 호출을 뒤따르는 코드에게 보일 것을 보장합니다.

이 방법은 극단적으로 빠른 카운터 읽기를 선형적 카운터 업데이트 확장성을 지원하면서도 가능하게 합니다. 하지만, 이 훌륭한 읽기 쪽 성능과 업데이트 쪽 확장성은 `eventual()`을 수행하는 추가된 쓰레드의 비용도 부과합니다.

Quick Quiz 5.23: Listing 5.5의 `inc_count()`는 왜 어토믹 인스트럭션을 사용해야 하나요? 어쨌건, 우린

Listing 5.5: Array-Based Per-Thread Eventually Consistent Counters

```

1  DEFINE_PER_THREAD(unsigned long, counter);
2  unsigned long global_count;
3  int stopflag;
4
5  static __inline__ void inc_count(void)
6  {
7      unsigned long *p_counter = &__get_thread_var(counter);
8
9      WRITE_ONCE(*p_counter, *p_counter + 1);
10 }
11
12 static __inline__ unsigned long read_count(void)
13 {
14     return READ_ONCE(global_count);
15 }
16
17 void *eventual(void *arg)
18 {
19     int t;
20     unsigned long sum;
21
22     while (READ_ONCE(stopflag) < 3) {
23         sum = 0;
24         for_each_thread(t)
25             sum += READ_ONCE(per_thread(counter, t));
26         WRITE_ONCE(global_count, sum);
27         poll(NULL, 0, 1);
28         if (READ_ONCE(stopflag))
29             smp_mb();
30         WRITE_ONCE(stopflag, stopflag + 1);
31     }
32 }
33 return NULL;
34 }
35
36 void count_init(void)
37 {
38     int en;
39     thread_id_t tid;
40
41     en = pthread_create(&tid, NULL, eventual, NULL);
42     if (en != 0) {
43         fprintf(stderr, "pthread_create: %s\n", strerror(en));
44         exit(EXIT_FAILURE);
45     }
46 }
47
48 void count_cleanup(void)
49 {
50     WRITE_ONCE(stopflag, 1);
51     while (READ_ONCE(stopflag) < 3)
52         poll(NULL, 0, 1);
53     smp_mb();
54 }
```

쓰레드별 카운터를 접근하는 여러 쓰레드가 있는 거잖아요!



Quick Quiz 5.24: Listing 5.5 의 `eventual()` 함수에서의 단일 글로벌 쓰레드는 전역 락 만큼이나 심각한 병목 아닐까요?



Quick Quiz 5.25: Listing 5.5 의 `read_count()` 가 반환하는 예상값은 쓰레드의 수가 늘어날수록 더더욱 부정확해지지 않나요?



Quick Quiz 5.26: Listing 5.5 에 보인 결과적으로 일관되는 알고리즘에서는 읽기도 업데이트도 극단적으로 낮은 오버헤드를 가지고 극단적인 확장성을 갖는데, 왜 어떤 사람들은 읽기 성능이 나쁜, Section 5.2.2 에 보인 구현을 신경쓸까요?



Quick Quiz 5.27: Listing 5.5 의 `read_count()` 에 의해 반환되는 추정값의 정확도는 어떻게 되나요?



5.2.5 Discussion

이 세개의 구현은 통계적 카운터를 위한 단일 프로세서에서에 가까운 성능을 뽑아낼 수 있음을 보입니다, 병렬 기계에서 돌아가고 있긴 하지만요.

Quick Quiz 5.28: 패킷의 크기는 다양하다는 점을 놓고 생각할 때, 패킷의 수를 세는 것과 패킷들의 전체 바이트 수를 세는 것 사이에는 어떤 근본적 차이가 있을까요?



Quick Quiz 5.29: 읽기 쓰레드는 모든 쓰레드의 카운터를 합해야 한다는 점을 놓고 보면, 이 카운터 읽기 오퍼레이션은 큰 수의 쓰레드를 가졌을 때 긴 시간을 요할 수 있습니다. 값증가 오퍼레이션이 빠르고 확장성 있게 유지하면서 읽기 쓰레드들 역시 합리적인 성능과 확장성만이 아니라 좋은 정확도를 취할 수 있는 방법은 없을까요?



이 섹션에서 무엇이 이야기 되었는지를 생각하면, 여러분은 이제 이 챕터의 시작 부분에서 이야기 된 네트워킹을 위한 통계적 카운터에 대한 Quick Quiz 에 대답할 수 있어야 합니다.

5.3 Approximate Limit Counters

An approximate answer to the right problem is worth a good deal more than an exact answer to an approximate problem.

John Tukey

카운팅의 또 다른 특수한 경우는 한계 검사에 연관되는 때입니다. 예를 들어, Quick Quiz 5.3 의 대략적 구조체 할당 한계 문제에서 이야기 되었듯, 일단 사용 중인 구조체의 갯수가 어떤 한계, 이 경우, 10,000 을 넘어선다면 모든 해당 구조체 할당이 실패해야 하도록 할당된 구조체의 갯수를 유지해야 한다고 해 봅시다. 더 나아가서 이 구조체는 수명이 짧아서 이 한계는 가끔씩만 초과되며, 이 한계는 약간의 정도는 초과되어도 괜찮다고 생각해 봅시다 (이 한계가 정확해야 한다면 Section 5.4 을 참고하시기 바랍니다).

5.3.1 Design

리미트 카운터를 위한 가능한 한가지 설계는 10,000이라는 한계를 쓰레드의 수로 나누고, 각 쓰레드에게 고정된 크기의 구조체 풀을 주는 것입니다. 예를 들어, 100 개의 쓰레드가 있다면, 각 쓰레드는 100개 구조체를 담는 각자의 풀을 관리하는 겁니다. 이 방법은 간단하고, 일부 경우에는 잘 동작합니다만, 해당 구조체가 한 쓰레드에 의해 할당되고 다른 쓰레드에 의해 해제되는 혼란 경우를 처리하지 못합니다 [MS93]. 한 측면에서 보자면, 특정 쓰레드가 자신이 해제하는 모든 구조체에 대한 크레딧을 가져간다면, 대부분의 해제를 하는 쓰레드는 사용할 수도 없는 여유분의 구조체를 가지는 반면 대부분의 할당을 하는 쓰레드는 구조체가 동나버릴 겁니다. 다른 측면에서 보자면, 해제된 구조체에 대한 크레딧을 그것을 할당한 CPU 가 가져간다면, CPU 들은 각자의 카운터를 조정할 필요가 생겨서, 이는 높은 비용의 어토믹 인스트럭션이나 다른 쓰레드간 통신 수단을 필요로 할 겁니다.²

요약하자면, 많은 중요한 워크로드에서, 이런 이 카운터를 완전히 분할할 수는 없습니다. 카운터를 분할하는 것은 Section 5.2 에서 이야기한 세개의 방법에서의 훌륭한 업데이트 쪽 성능을 가져온 것이었으나, 이는 어떤 회의론을 불러일으킬 수도 있습니다. 하지만, Section 5.2.4 에서 소개한 결과적으로 일관적인 알고리즘은 하나의 흥미로운 힌트를 제공합니다. 이 알고리즘이 책들의 두 집합을, 즉 업데이트 쓰레드를 위한 쓰레드별 counter

² 그러나, 각 구조체가 항상 그것을 할당한 것과 같은 CPU (또는 쓰레드)에 의해 해제된다면, 이 간단한 분할 방법은 무척 잘 동작할 겁니다.

변수와 읽기 쓰레드를 위한 `global_count` 를 가졌으며, 이 쓰레드별 `counter` 와 결과적으로 일관적이게 하기 위해 주기적으로 `global_count` 를 업데이트 하는 `eventual()` 쓰레드를 가짐을 기억하십시오. 이 쓰레드별 `counter` 는 `global_count` 가 완전한 값을 유지하는 동안 이 카운터 값을 완전히 분할합니다.

리미트 카운터를 위해, 우린 이 카운터를 부분적으로 분할하는 이 테마의 한 변종을 사용할 수 있습니다. 예를 들어, 네개의 쓰레드가 있고 이것들이 쓰레드별 `counter` 만이 아니라 쓰레드별 최대값 (`countermax` 라고 해봅시다) 을 갖는다고 해봅시다.

그런데 각 쓰레드가 자신의 `counter` 를 증가시켜야 하는데 `counter` 가 `countermax` 와 같다면 어떻게 될까요? 여기서의 트릭은 이 쓰레드의 `counter` 값의 절반을 `globalcount` 로 옮기고, 이어서 `counter` 를 증가시키는 겁니다. 예를 들어, 특정 쓰레드의 `counter` 와 `countermax` 변수가 똑같이 10이라면, 다음 일을 합니다:

1. 전역 락을 잡습니다.
2. `globalcount` 에 5 를 더합니다.
3. 이 합을 균형맞추기 위해, 이 쓰레드의 `counter` 에서 5 를 뺍니다.
4. 이 전역 락을 놓습니다.
5. 이 쓰레드의 `counter` 를 증가시켜서, 그 값이 6이 되게 합니다.

이 과정이 여전히 전역 락을 필요로 하긴 하지만, 이 락은 다섯번의 값 증가 오퍼레이션에 한번만 잡으면 되므로, 이 락의 경쟁 수준을 크게 떨어뜨립니다. 우린 이 경쟁을 `countermax` 값을 증가시킴으로써 우리가 원하는 대로 떨어뜨릴 수 있습니다. 하지만, `countermax` 의 값을 증가시킴으로써 발생하는 페널티는 `globalcount` 의 정확도의 하락입니다. 이를 자세히 보자면, 네개의 CPU 를 가진 시스템에서, `countermax` 가 10 이면, `globalcount` 는 최대 40 카운트의 에러를 가질 수 있습니다. 반면에, `countermax` 가 100 으로 증가된다면, `globalcount` 는 400 카운트의 에러까지도 가질 수 있습니다.

이는 이 카운터의 합계값으로부터 `globalcount` 의 차이를 얼마나 신경쓰는가 하는 질문을 일으킵니다, 참고로 이 합계값은 `globalcount` 과 각 쓰레드의 `counter` 변수의 합입니다. 이 질문에 대한 답은 이 합계값이 이 카운터의 한계 (`globalcountmax` 라고 부릅시다)로부터 얼마나 떨어져 있으느냐에 달려 있습니다. 이 두 값의 차이가 크면 클수록, 더 큰 `countermax` 가

Listing 5.6: Simple Limit Counter Variables

```

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);

```

`globalcountmax` 리미트를 초월하는 리스크를 갖지 않을 수 있습니다. 이는 특정 쓰레드의 `countermax` 변수의 값은 이 차이에 기반해 설정될 수 있음을 의미합니다. 이 한계로부터 멀 때에는 쓰레드별 `countermax` 변수는 성능과 확장성에 최적화하기 위해 더 큰 값을 갖고, 이 한계에 가까워지면 이 변수는 `globalcountmax` 한계와의 체크에서의 에러를 최소화하기 위해 더 작은 값을 갖는 겁니다.

이 설계는 병렬 *fastpath* 의 한 예로, 흔한 경우는 비싼 인스트럭션과 쓰레드간 상호 작용 없이 수행되지만, 때때로는 더 보수적으로 설계된 (그리고 높은 오버헤드의) 전역 알고리즘을 사용하게 되는 흔한 설계 패턴입니다. 이 설계 패턴은 Section 6.4 에서 더 자세히 다뤄집니다.

5.3.2 Simple Limit Counter Implementation

Listing 5.6 이 이 구현에 의해 사용되는 쓰레드별 변수와 전역 변수를 보이고 있습니다. 쓰레드별 `counter` 와 `countermax` 변수는 각각 연관된 쓰레드의 로컬 카운터와 해당 카운터의 상한선입니다. 라인 3 의 `globalcountmax` 는 합해진 카운터의 상한선을 담고 있으며, 라인 4 의 `globalcount` 변수는 전역 카운터입니다. `globalcount` 의 합과 각 쓰레드의 `counter` 는 전체 카운터의 합산값을 제공합니다. 라인 5 의 `globalreserve` 변수는 최소한 모든 쓰레드별 `countermax` 변수의 합산값입니다. 이 변수들 사이의 관계가 Figure 5.5 에 그려져 있습니다:

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

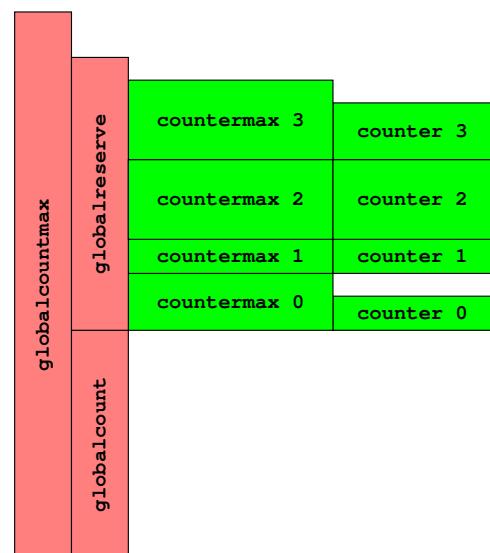
`counterp[]` 배열의 각 원소는 연관된 쓰레드의 `counter` 변수를 참조하며, 마지막으로, `gblcnt_mutex` 스판락은 모든 전역 변수를 보호하는데, 달리 말하자면, 어떤 쓰레드도 `gblcnt_mutex` 를 얻지 않고서는 어떤 전역 변수도 액세스하거나 수정할 수 없습니다.

Listing 5.7: Simple Limit Counter Add, Subtract, and Read

```

1 static __inline__ int add_count(unsigned long delta)
2 {
3     if (countermax - counter >= delta) {
4         WRITE_ONCE(counter, 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 static __inline__ int sub_count(unsigned long delta)
21 {
22     if (counter >= delta) {
23         WRITE_ONCE(counter, 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 static __inline__ 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 += READ_ONCE(*counterp[t]);
48     spin_unlock(&gblcnt_mutex);
49     return sum;
50 }

```

**Figure 5.5:** Simple Limit Counter Variable Relationships**Listing 5.8:** Intuitive Fastpath

```

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

```

Listing 5.7 는 add_count(), sub_count(), 그리고 read_count() 함수들을 (count_lim.c) 보입니다.

Quick Quiz 5.30: Listing 5.7 는 왜 Section 5.2 에서 보인 inc_count() 와 dec_count() 인터페이스 대신 add_count() 와 sub_count() 를 제공하는 거죠?

라인 1-18 는 특정 값 delta 를 이 카운터에 더하는 add_count() 를 보입니다. 라인 3 는 이 쓰레드의 counter 에 delta 를 더할 공간이 있는지 검사하고, 그렇다면 라인 4 에서 그걸 더하고 라인 5 에서 성공했음을 리턴합니다. 이게 add_count() 의 fastpath 로, 여기선 어토믹 오퍼레이션을 사용하지 않고, 쓰레드별 변수들만을 참조하며, 어떤 캐ши 미스도 일으키지 않을 겁니다.

Quick Quiz 5.31: Listing 5.7 의 라인 3에서의 이상한 형태의 조건은 뭐죠? Listing 5.8 에서 보인 더 직관적인 형태의 fastpath 는 어떤가요?

라인 3에서의 테스트가 실패하면, 우린 전역 변수들에 액세스 해야만 하며, 따라서 라인 7에서 gblcnt_mutex 를 잡아야 하며, 이 락은 실패의 경우 라인 11에서, 성공의 경우 라인 16에서 해제합니다. 라인 8은 Listing 5.9에 보인 globalize_count() 를 호출하는데, 여기선 쓰레드 지역 변수들을 정리하고, 전역 변수들을

필요한대로 조정 하여서 전체 과정을 단순화 시킵니다. (하지만 제 말만 듣고 그렇겠거니 하지 말고 스스로 코드해 보세요!) 라인 9 와 10 는 `delta` 의 합이 가능할지 체크해 보는데, 여기선 Figure 5.5 의 작은 부호를 앞서는 표현이 (가장 왼쪽의) 두개의 빨간 막대의 높이 차를 의미합니다. `delta` 를 합하는 게 가능하지 않다면, 라인 11 는 (앞서 언급되었듯) `gblcnt_mutex` 를 해제하고 라인 12 는 실패를 의미하는 값을 리턴합니다.

그렇지 않다면, 우린 `slowpath` 를 취합니다. 라인 14 에서는 `globalcount` 에 `delta` 를 더하고, 이어서 라인 15 에서는 이 전역 변수와 쓰레드별 변수를 모두 업데이트 하기 위해 (Listing 5.9 에 보인) `balance_count()` 를 호출합니다. 이 `balance_count()` 호출은 일반적으로 `fastpath` 를 다시 가능하게 하기 위해 이 쓰레드의 `countermax` 값을 설정할 겁니다. 라인 16 는 (다시, 앞서 이야기 되었듯) `gblcnt_mutex` 를 해제하고, 마지막으로 라인 17 는 성공을 의미하는 값을 리턴합니다.

Quick Quiz 5.32: Listing 5.7 에서 `globalize_count()` 는 왜 나중에 `balance_count()` 를 호출해 그것을 다시 채우기 위한 이유만으로 쓰레드별 변수를 0 으로 초기화 시키나요? 왜 그 쓰레드별 변수를 그냥 0 이 아닌 채로 두지 않는 거죠?

라인 20-36 는 이 카운터에서 `delta` 를 빼는 `sub_count()` 를 보입니다. 라인 22 은 이 쓰레드별 카운터가 이 빼기를 할 수 있는지 보고, 그렇다면 라인 23 는 이 빼기를 행하고 라인 24 에서 성공을 의미하는 값을 반환합니다. 이 라인들이 `sub_count()` 의 `fastpath` 를 형성하며, `add_count()` 에서와 마찬가지로, 이 `fastpath` 는 어떤 비용이 높은 오퍼레이션도 수행하지 않습니다.

이 `fastpath` 가 `delta` 빼기를 하지 못한다면, 수행은 라인 26-35 의 `slowpath` 로 이어집니다. 이 `slowpath` 는 전역 상태에 접근해야만 하므로, 라인 26 는 `gblcnt_mutex` 를 획득하는데, 이는 라인 29 에서 (실패의 경우) 또는 라인 34 에서 (성공의 경우) 해제됩니다. 라인 27 는 Listing 5.9 에 보인 `globalize_count()` 를 호출하는데, 이는 다시 쓰레드 지역 변수들을 정리하고 전역 변수들을 필요한대로 조정합니다. 라인 28 는 이 카운터가 `delta` 빼기를 할 수 있는지 체크하고, 그렇지 않다면 라인 29 에서 `gblcnt_mutex` 를 (앞서 이야기 했듯) 해제하고 라인 30 에서 실패를 의미하는 값을 리턴합니다.

Quick Quiz 5.33: `add_count()` 에서는 `globalreserve` 가 우리를 위해 세어졌는데, Listing 5.7 의 `sub_count()` 에서는 왜 그렇지 않나요?

Quick Quiz 5.34: Listing 5.7 에 보인 `add_count()` 를 한 쓰레드가 호출하고, 다른 쓰레드가 `sub_count()`

를 호출한다고 해봅시다. 카운터의 값은 0이 아님에도 불구하고 `sub_count()` 는 실패를 리턴하지 않을까요?

반면, 라인 28 에서 이 카운터가 `delta` 빼기를 할 수 있다고 확인되면, 우린 이 `slowpath` 를 마무리 합니다. 라인 32 는 이 빼기를 하고 이어서 라인 33 에서는 전역 변수와 쓰레드별 변수를 모두 업데이트 하기 위해 (`fastpath` 가 다시 가능해 지길 바라며) `balance_count()` (Listing 5.9 에 보여져 있습니다) 를 호출합니다. 라인 34 는 `gblcnt_mutex` 를 해제하고, 라인 35 는 성공을 의미하는 값을 리턴합니다.

Quick Quiz 5.35: Listing 5.7 에는 왜 `add_count()` 와 `sub_count()` 가 모두 있죠? 왜 단순히 `add_count()` 에 음수를 넘기지 않는 건가요?

라인 38-50 는 이 카운터의 합산값을 반환하는 `read_count()` 를 보입니다. 이 함수는 라인 43 에서 `gblcnt_mutex` 를 획득하고 라인 48 에서 이를 해제하여, `add_count()` 와 `sub_count()` 에서의 전역적 오퍼레이션들을 배제시키며, 우리가 곧 보겠지만, 쓰레드 생성과 종료 역시 배제시킵니다. 라인 44 은 지역 변수 `sum` 을 `globalcount` 의 값으로 초기화 시키고, 이어서 라인 45-47 의 루프는 쓰레드별 `counter` 변수들의 값을 합합니다. 이어서 라인 49 은 이 합을 반환합니다.

Listing 5.9 은 Listing 5.7 에서 보인 `add_count()`, `sub_count()`, 그리고 `read_count()` 기능들에서 사용된 유ти리티 함수들을 보입니다.

라인 1-7 는 `globalize_count()` 를 보아는데, 이 함수는 현재 쓰레드의 쓰레드별 카운터를 9 으로 만들고 전역 변수들을 적절히 조정합니다. 이 함수가 이 카운터의 합산값을 바꾸지 않고, 그 대신 이 카운터의 현재 값이 어떻게 표현되는지를 바꿈을 알아두는 게 중요합니다. 라인 3 는 이 쓰레드의 `counter` 변수의 값을 `globalcount` 에 더하고, 라인 4 는 `counter` 의 값을 0 으로 만듭니다. 비슷하게, 라인 5 는 쓰레드별 `countermax` 를 `globalreserve` 에서 빼고, 라인 6 는 `countermax` 를 0 으로 만듭니다. 이 함수와 `balance_count()` 를 읽을 때는 Figure 5.5 를 참고하는 게 도움이 될 텐데, 이어서 이걸 보겠습니다.

라인 9-19 는 대략적으로 말해서 `globalize_count()` 의 반대 역할인 `balance_count()` 를 보입니다. 이 함수의 일은 현재 쓰레드의 `countermax` 변수를 이 카운터가 `globalcountmax` 한계를 넘어버리는 위험을 방지하는 가장 큰 값으로 설정하는 것입니다. 현재 쓰레드의 `countermax` 변수를 바꾸는 것은 물론 Figure 5.5 를 다시 참고하면 볼 수 있듯이 연관된 `counter`, `globalcount` 그리고 `globalreserve` 를 조정할 것이 필요합니다. 이걸 함으로써, `balance_count()` 는 `add_count()` 와 `sub_count()` 의 오버헤드 낮은 fast-

Listing 5.9: Simple Limit Counter Utility Functions

```

1 static __inline__ void globalize_count(void)
2 {
3     globalcount += counter;
4     counter = 0;
5     globalreserve -= countermax;
6     countermax = 0;
7 }
8
9 static __inline__ 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 }

```

path 의 사용을 극대화 시킵니다. `globalize_count()`에서와 마찬가지로, `balance_count()`는 이 카운터의 합산값을 바꾸는 것이 허용되지 않습니다.

라인 11-13 는 `globalcount` 나 `globalreserve`에 의해서 커버되지 않는 `globalcountmax`의 부분에 대한 이 쓰레드의 지분을 계산하고, 이 값을 이 쓰레드의 `countermax`에 할당합니다. 라인 14 는 `globalreserve`로의 이 연관된 조정을 행합니다. 라인 15 은 이 쓰레드의 `counter`를 0과 `countermax` 사이 중간 값으로 설정합니다. 라인 16 는 `globalcount`가 실제로 `counter`의 이 값을 수용할 수 있을지 검사하고, 그렇지 않다면 라인 17 에서 `counter`를 적절히 감소시킵니다. 마지막으로, 양쪽 경우 모두, 라인 18 에서 연관된 조정을 `globalcount`에 행합니다.

Quick Quiz 5.36: Listing 5.9 의 라인 15에서 왜 `counter`를 `countermax / 2`로 설정하는 거죠? 그냥 `countermax` 카운트를 취하는 게 더 간단하지 않을까요?



Figure 5.6에 보인 것처럼 카운터들의 관계가 첫번째 `globalize_count()`와 뒤이은 `balance_count()`의

수행에 의해 어떻게 바뀌는지 보는 게 도움이 될 겁니다. 시간은 왼쪽에서 오른쪽으로 흐르며, 가장 왼쪽의 구성은 대략적으로 Figure 5.5 의 그 것입니다. 가운데의 구성은 쓰레드 0에 의해 `globalize_count()`가 수행된 후의 이 같은 카운터들의 관계를 보입니다. 이 그림에서 볼 수 있듯이, 쓰레드 0의 `counter`(그림 내의 “c 0”)a 는 `globalcount`에 더해졌으며, 그 사이 `globalreserve`의 값은 똑같은 양만큼 줄었습니다. 쓰레드 0의 `counter`도 그것의 `countermax`(그림 내의 “cm 0”)도 0이 되었습니다. 다른 세개의 쓰레드의 카운터들은 변하지 않았습니다. 이 변화는 이 카운터의 전체 값에는 영향을 주지 않았는데, 가장 바닥쪽의, 가장 왼쪽과 중간 구성을 잇는 점선을 통해 보여져 있음을 알아 두시기 바랍니다. 달리 말하면, `globalcount`와 네 쓰레드의 `counter` 변수들의 값의 합은 두 구성 모두에서 같습니다. 비슷하게, 이 변화는 `globalcount`와 `globalreserve`의 합에 영향을 끼치지 않았는데, 위쪽 점선을 통해 보입니다.

오른쪽 구성은 다시 쓰레드 0에 의해 `balance_count()`가 호출된 후의 이 카운터들의 관계를 보입니다. 각 세개의 구성으로부터 위쪽으로 확장되는 세로선에 의해 보이는 남아 있는 카운트의 4분의 1이 쓰레드 0의 `countermax`에 더해지고 그 절반이 쓰레드 0의 `counter`에 더해졌습니다. 쓰레드 0의 `counter`에 더해진 양만큼이 또한 이 카운터의 전체 값(다시 말하지만 이는 `globalcount`와 세 쓰레드의 `counter` 변수의 값의 합입니다)을 바꾸는 걸 막기 위해 `globalcount`에서 빼졌으며, 이 역시 다시 말하지만 가장 아래쪽의, 가운데와 오른쪽 구성을 잇는 두개의 점선으로 표시되어 있습니다. `globalreserve` 변수 역시 이 변수가 네 쓰레드의 `countermax` 변수의 값의 합으로 남아있게끔 조정되었습니다. 쓰레드 0의 `counter`는 그것의 `countermax` 보다 작으므로, 쓰레드 0은 한번 더 이 카운터를 지역적으로 증가시킬 수 있습니다.

Quick Quiz 5.37: Figure 5.6에서, 비록 남아 있는 카운트의 최대 한계까지의 4분의 1이 쓰레드 0에 할당되어 있다고는 해도, 가운데와 오른쪽 구성을 잇는 위쪽의 점선이 보이듯 남아있는 카운트의 8분의 1만이 소모됩니다. 왜 그런거죠?



라인 21-28 는 `count_register_thread()`를 보이는데, 이 함수는 새로 생성된 쓰레드의 상태를 설정합니다. 이 함수는 단순히 새로 생성된 쓰레드의 `counter` 변수로의 포인터를 연관된 `counterp[]` 배열 내 원소에 `gblcnt_mutex`의 보호 아래 넣습니다.

마지막으로, 라인 30-38 는 `count_unregister_thread()`를 보이는데, 이 함수는 곧 종료될 쓰레드의 상태를 정리합니다. 라인 34 는 `gblcnt_mutex`를 획득하고 라인 37에서 이를 해제합니다. 라

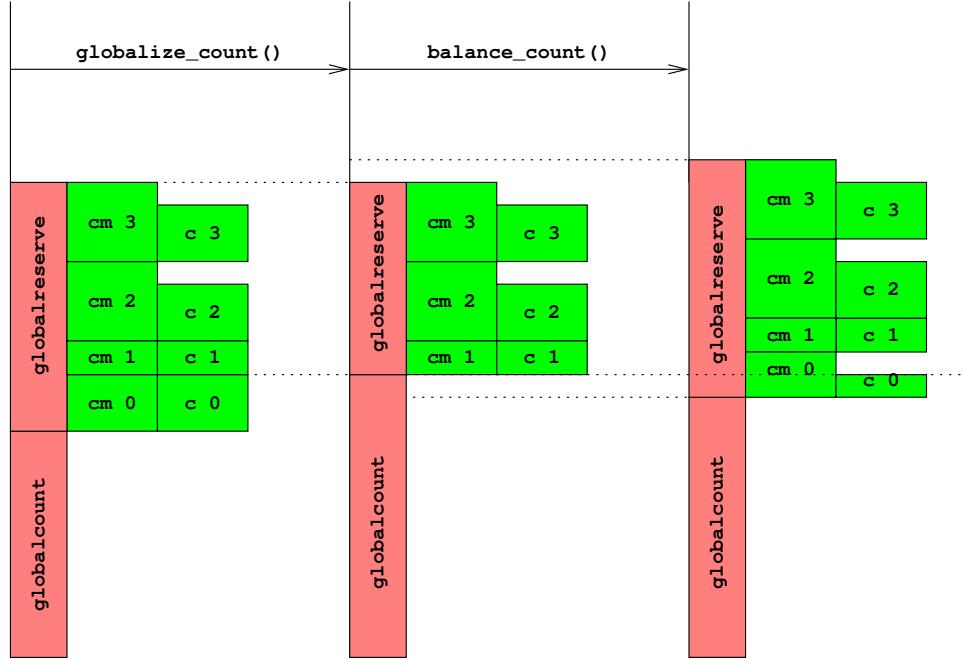


Figure 5.6: Schematic of Globalization and Balancing

인 35 에서는 이 쓰레드의 카운터 상태를 지우기 위해 `globalize_count()` 를 호출하고, 라인 36 는 이 쓰레드의 `counterp[]` 배열 내 원소를 지웁니다.

5.3.3 Simple Limit Counter Discussion

이런 종류의 카운터는 `add_count()` 와 `sub_count()` 의 fastpath 내의 비교와 브랜치들로 인한 오버헤드도 있지만 합산값이 0에 가까울 때 상당히 빠릅니다. 하지만, 쓰레드별 `countermax` 예약값의 사용은 이 카운터의 합산값이 `globalcountmax`에 전혀 가깝지 않을 때 조차도 `add_count()` 가 실패할 수 있음을 의미합니다. 비슷하게, `sub_count()` 는 이 카운터의 합산값이 0 근처도 아닐 때 조차도 실패할 수 있습니다.

많은 경우, 이는 받아들여질 수 없습니다. 비록 `globalcountmax` 가 대략적 한계를 의도했다 하더라도, 정확히 얼마큼 대략적인지가 조절되는 한계도 종종 있습니다. 대략적인 정도를 제한하는 한 가지 방법은 쓰레드별 `countermax` 인스턴스의 값의 상한값을 정하는 것입니다. 이 작업이 다음 섹션에서 다뤄집니다.

5.3.4 Approximate Limit Counter Implementation

이 구현(`count_lim_app.c`)은 앞의 섹션의 것들((Listings 5.6, 5.7, 그리고 5.9) 과 상당히 유사하므로, 여기엔

Listing 5.10: Approximate Limit Counter Variables

```

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

```

Listing 5.11: Approximate Limit Counter Balancing

```

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 }

```

변경 사항만 보입니다. Listing 5.10 는 Listing 5.6 와 동일 하지만, 쓰레드별 `countermax` 변수의 가능한 최대값을 설정하는 `MAX_COUNTERMAX` 를 추가합니다.

비슷하게, Listing 5.11 은 Listing 5.9 의 `balance_count()` 함수와 똑같지만, 쓰레드별 `countermax` 변수

의 MAX_COUNTERMAX 한계를 강제하는 라인 6 와 7 를 추가합니다.

5.3.5 Approximate Limit Counter Discussion

이 변경들은 앞의 버전에서 보여진 한계 부정확성을 크게 줄여줍니다만, 다른 문제를 나타냅니다: MAX_COUNTERMAX 가 어떤 값이든 워크로드에 의존적인 일부 분량의 액세스를 fastpath 로부터 떨어뜨릴 겁니다. 쓰레드의 수가 늘어나면, fastpath 외의 수행은 성능과 확장성 모두에 병목이 될 겁니다. 하지만, 이 문제는 뒤로 미루고, 일단 정확한 한계의 카운터로 넘어가 봅시다.

5.4 Exact Limit Counters

Exactitude can be expensive. Spend wisely.

Unknown

Quick Quiz 5.4 에서 이야기 된 정확한 구조체 할당 한계 문제를 해결하기 위해서, 우린 정확히 언제 그 한계가 초과되었는지를 말해줄 수 있는 한계 카운터가 필요 합니다. 그런 한계 카운터를 구현하는 한가지 방법은 그 수를 예약한 쓰레드가 그것을 포기하게 만드는 것입니다. 이걸 위한 한가지 방법은 어토믹 인스트럭션을 사용하는 것입니다. 물론, 어토믹 인스트럭션은 fastpath 를 느리게 만들 겁니다만, 다른 한편으로는 시도도 안해 보는 것도 웃긴 일일 겁니다.

5.4.1 Atomic Limit Counter Implementation

불행히도, 한 쓰레드가 다른 쓰레드로부터 안전히 카운트를 제거하고자 한다면, 두 쓰레드는 해당 쓰레드의 counter 와 countermax 변수들을 원자적으로 조정해야 합니다. 이걸 위한 일반적인 방법은 이 두 변수를 하나의 변수로 결합시키는 것으로, 예를 들어 32비트 변수가 있다면, 앞쪽 16비트는 counter 를 나타내게 하고 뒤쪽 16비트는 countermax 를 나타내게 하는 겁니다.

Quick Quiz 5.38: 왜 이 쓰레드의 counter 와 countermax 변수들을 하나의 단위로 원자적 조정해야 하죠? 그것들 각자를 원자적으로 조정하는 것으로 충분하지 않을까요?

간단한 어토믹 한계카운터를 위한 변수와 액세스 함수들이 Listing 5.12 (count_lim_atomic.c) 에 보여져 있습니다. 앞의 알고리즘에서의 counter 와 countermax 변수들은 라인 1 에 보인 단일 변수

Listing 5.12: Atomic Limit Counter Variables and Access Functions

```

1 atomic_t __thread counterandmax = ATOMIC_INIT(0);
2 unsigned long globalcountmax = 1 << 25;
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 __inline__ void
11 split_counterandmax_int(int cami, int *c, int *cm)
12 {
13     *c = (cami >> CM_BITS) & MAX_COUNTERMAX;
14     *cm = cami & MAX_COUNTERMAX;
15 }
16
17 static __inline__ void
18 split_counterandmax(atomic_t *cam, int *old, int *c, int *cm)
19 {
20     unsigned int cami = atomic_read(cam);
21
22     *old = cami;
23     split_counterandmax_int(cami, c, cm);
24 }
25
26 static __inline__ int merge_counterandmax(int c, int cm)
27 {
28     unsigned int cami;
29
30     cami = (c << CM_BITS) | cm;
31     return ((int)cami);
32 }
```

counterandmax 에 앞부분 절반은 counter 로, 뒷부분 절반은 countermax 로 쓰이게끔 합쳐졌습니다. 이 변수는 실제 표현은 int 인 atomic_t 타입입니다.

라인 2-6 는 Listing 5.10 의 것들과 비슷한 역할을 하는 globalcountmax, globalcount, globalreserve, counterp, 그리고 gblcnt_mutex 의 정의를 보입니다. 라인 7 는 counterandmax 의 각 절반의 비트 수를 제공하는 CM_BITS 를 정의하며, 라인 8 는 counterandmax 의 각 절반에 들어갈 수 있는 최대값인 MAX_COUNTERMAX 를 정의합니다.

Quick Quiz 5.39: Listing 5.12 의 라인 7 는 어떻게 C 표준을 위반하나요?



라인 10-15 는 atomic_t counterandmax 변수 아래의 int 를 받아서 counter(c) 와 countermax(cm) 부분으로 나눠주는 split_counterandmax_int() 함수를 보입니다. 라인 13 는 이 int 의 앞쪽 절반을 빼어내서 인자 c 로 명시된 대로 결과를 위치시키며, 라인 14 는 이 int 의 뒤쪽 절반을 빼어내서 인자 cm 으로 명시된 대로 결과를 위치시킵니다.

라인 17-24 는 라인 20 에서 명시된 변수로부터 아래의 int 를 가져와서 라인 22 의 old 인자로 명시된 대로 저장하고 라인 23 에서 그것을 쪼개기 위해

`split_counterandmax_int()` 를 호출하는 `split_counterandmax()` 함수를 보입니다.

Quick Quiz 5.40: 단 하나의 `counterandmax` 변수만 있는데, 왜 Listing 5.12 의 line 18 에서는 포인터를 넘기는 거죠?

라인 26-32 는 `split_counterandmax()` 의 반대라고 생각될 수 있는 `merge_counterandmax()` 함수를 보입니다. 라인 30 는 `c` 와 `cm` 으로 각각 전달된 `counter` 와 `countermax` 값을 병합하고 그 결과를 리턴합니다.

Quick Quiz 5.41: Listing 5.12 의 `merge_counterandmax()` 는 직접 `atomic_t` 에 저장을 하는 대신 `int` 를 리턴하나요?

Listing 5.13 은 `add_count()` 와 `sub_count()` 함수들을 보입니다.

라인 1-32 는 라인 8-15 에 fastpath 가 있고 나머지 부분은 이 함수의 slowpath 인 `add_count()` 를 보입니다. Fastpath 의 라인 8-14 은 라인 13-14 에서 실제 CAS 를 행하는 `atomic_cmpxchg()` 기능을 가지고 구현된 compare-and-swap (CAS) 루프를 형성합니다. 라인 9 은 현재 쓰레드의 `counterandmax` 변수를 자신의 `counter` (`c`) 와 `countermax` (`cm`) 컨포넌트로 쪼개고, 아래의 `int` 는 `old` 에 위치시킵니다. 라인 10 는 `delta` 의 양이 지역적으로 처리될 수 있는지 검사하고 (정수형 오버플로우를 막기 위해 신경쓰면서), 그 렇지 않다면 라인 11 가 slowpath 로 전환합니다. 그 렇지 않다면, 라인 12 는 업데이트 된 `counter` 값을 원래의 `countermax` 값과 함께 `new` 로 결합시킵니다. 라인 13-14 의 `atomic_cmpxchg()` 기능은 어토믹하게 이 쓰레드의 `counterandmax` 변수를 `old` 와 비교하고, 이 비교가 성공했다면 자신의 값을 `new` 에 업데이트 합니다. 이 비교가 성공했다면, 라인 15 는 성공을 의미하는 값을 리턴하고, 그렇지 않다면 수행은 라인 8 의 루프로 이어집니다.

Quick Quiz 5.42: 우웩! Listing 5.13 의 라인 11 의 추한 `goto` 는 웬말이죠? `break` 문 모르세요???

Quick Quiz 5.43: Listing 5.13 의 라인 13-14 의 `atomic_cmpxchg()` 기능은 왜 실패할 수 있죠? 어쨌건, 우린 기존 값을 라인 9 에서 가져온 후 바꾸지 않았잖아요!

Listing 5.13 의 라인 16-31 는 라인 17 에서 획득되고 라인 24 와 30 에서 해제되는 `gblcnt_mutex` 로 보호되는 `add_count()` 의 slowpath 를 보입니다. 라인 18 는 이 쓰레드의 상태를 전역 카운터로 옮기는 `globalize_count()` 를 호출합니다. 라인 19-20 는 `delta` 값이 현재 전역 상태에 의해 처리될 수 있는지 검사하고, 그

Listing 5.13: Atomic Limit Counter Add and Subtract

```

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_counterandmax(&counterandmax, &old, &c, &cm);
10        if (delta > MAX_COUNTERMAX || c + delta > cm)
11            goto slowpath;
12        new = merge_counterandmax(c + delta, cm);
13    } while (atomic_cmpxchg(&counterandmax,
14                           old, new) != old);
15
16    return 1;
17
18    slowpath:
19    spin_lock(&gblcnt_mutex);
20    globalize_count();
21    if (globalcountmax - globalcount -
22        globalreserve < delta) {
23        flush_local_count();
24        if (globalcountmax - globalcount -
25            globalreserve < delta) {
26            spin_unlock(&gblcnt_mutex);
27            return 0;
28        }
29    }
30    globalcount += delta;
31    balance_count();
32    spin_unlock(&gblcnt_mutex);
33    return 1;
34
35    int sub_count(unsigned long delta)
36    {
37        int c;
38        int cm;
39        int old;
40        int new;
41
42        do {
43            split_counterandmax(&counterandmax, &old, &c, &cm);
44            if (delta > c)
45                goto slowpath;
46            new = merge_counterandmax(c - delta, cm);
47        } while (atomic_cmpxchg(&counterandmax,
48                           old, new) != old);
49
50        return 1;
51
52        slowpath:
53        spin_lock(&gblcnt_mutex);
54        globalize_count();
55        if (globalcount < delta) {
56            flush_local_count();
57            if (globalcount < delta) {
58                spin_unlock(&gblcnt_mutex);
59                return 0;
60            }
61        }
62        globalcount -= delta;
63        balance_count();
64        spin_unlock(&gblcnt_mutex);
65        return 1;
66    }

```

Listing 5.14: Atomic Limit Counter Read

```

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_counterandmax(counterp[t], &old, &c, &cm);
14            sum += c;
15        }
16    spin_unlock(&gblcnt_mutex);
17    return sum;
18 }

```

렇지 않다면 라인 21에서 모든 쓰레드의 지역 상태를 전역 카운터로 쓸어넘기는 `flush_local_count()`를 호출하고, 이어서 라인 22-23에서 `delta`가 처리될 수 있는지 다시 검사합니다. 만약 이 모든 것 후에도 `delta` 더하기가 될 수 없다면, 라인 24는 `gblcnt_mutex`를 (앞에서 이야기 된 것처럼) 해제하고 라인 25에서 실패를 의미하는 값을 리턴합니다.

그렇지 않다면, 라인 28는 `delta`를 전역 카운터에 더하고, 라인 29는 카운트를 적절하다면 각 지역 상태로 흘러리며, 라인 30에서 (다시 말하지만, 앞에서도 이야기했듯) `gblcnt_mutex`를 해제하고, 마지막으로 라인 31에서 성공을 의미하는 값을 리턴합니다.

Listing 5.13의 라인 34-63는 `add_count()`와 비슷한 구조를 가져서 라인 41-48에 `fast path`를, 라인 49-62에 `slowpath`를 갖는 `sub_count()` 함수를 보입니다. 이 함수의 라인별 분석은 독자 여러분의 연습을 위한 것으로 남겨두겠습니다.

Listing 5.14은 `read_count()`를 보입니다. 라인 9는 `gblcnt_mutex`를 획득하고 라인 16는 이를 해제합니다. 라인 10은 지역 변수 `sum`을 `globalcount`의 값으로 초기화하고, 라인 11-15의 루프는 쓰레드별 카운터를 이 합에 더하는데, 라인 13에서는 `split_counterandmax`를 사용해 각 쓰레드별 카운터를 격리시킵니다. 마지막으로, 라인 17은 이 합을 반환합니다.

Listings 5.15 and 5.16는 유ти리티 함수들인 `globalize_count()`, `flush_local_count()`, `balance_count()`, `count_register_thread()`, 그리고 `count_unregister_thread()`를 보입니다. `globalize_count()`를 위한 코드는 Listing 5.15의 라인 1-12에 보이고 있으며, 앞의 알고리즘과 비슷합니다만, 이제 `counter`와 `countermax`를 `counterandmax`로부터 조셉 필요가 있는 라인 7이 추가되었습니다.

모든 쓰레드의 지역 카운터 상태를 전역 카운터로 옮기는 `flush_local_count()`의 코드가 라인 14-32에

Listing 5.15: Atomic Limit Counter Utility Functions 1

```

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

```

보여져 있습니다. 라인 22는 `globalreserve`의 값이 모든 쓰레드별 카운트를 허용하는지 검사하고, 그렇지 않다면 라인 23에서 리턴합니다. 그렇지 않다면, 라인 24는 지역 변수 `zero`를 0이 된 `counter`와 `countermax`의 조합으로 초기화 시킵니다. 라인 25-31의 루프는 각 쓰레드를 돌아갑니다. 라인 26는 현재 쓰레드가 캉너터 상태를 가지고 있는지 검사하고, 그렇다면 라인 27-30가 이 상태를 전역 카운터로 옮깁니다. 라인 27는 어토믹하게 현재 쓰레드의 상태를 가져오면서 그 값을 0으로 만듭니다. 라인 28은 이 상태를 `counter`(지역 변수 `c`)와 `countermax`(지역 변수 `cm`) 부분으로 조셉니다. 라인 29는 이 쓰레드의 `counter`와 `globalcount`를 더하며, 그동안 라인 30은 이 쓰레드의 `countermax`를 `globalreserve`로부터 뺍니다.

Quick Quiz 5.44: 쓰레드가 간단하게 `counterandmax` 변수를 Listing 5.15의 라인 14에서의 `flush_local_count()`가 비운 후에 곧바로 다시 채우는 건 왜 안되나요?



Quick Quiz 5.45: Listing 5.15의 line 27에서 `flush_local_count()`가 `counterandmax` 변수를 액세스하는 동안 `add_count()`와 `sub_count()`의 동시에 수행되는 fastpath들이 해당 변수를 간섭하는 것은 무엇이 막고 있습니까?



Listing 5.16: Atomic Limit Counter Utility Functions 2

```

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_counterandmax(c, cm);
21    atomic_set(&counterandmax, 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] = &counterandmax;
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 }

```

Listing 5.16의 라인 1-22는 호출하는 쓰레드의 지역 `counterandmax` 변수를 재충전하는 `balance_count()`의 코드를 보입니다. 이 함수는 앞의 알고리즘과 상당히 유사한데 합쳐진 `counterandmax` 변수를 처리하는 것이 요구된다는 경계를 갖습니다. 라인 24에서 시작하는 `count_register_thread()` 함수와 라인 33에서 시작하는 `count_unregister_thread()` 함수와 함께 이 코드의 자세한 분석은 독자 여러분의 몫으로 남겨둡니다.

Quick Quiz 5.46: `atomic_set()` 기능은 명시된 `atomic_t`에 간단한 값 할당을 할 뿐인데, Listing 5.16의 `balance()`의 라인 21은 어떻게 이 변수를 업데이트 할 수 있죠?



다음 섹션은 이 설계를 평가해 봅니다.

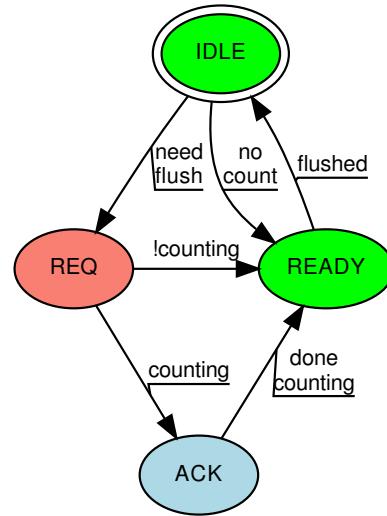


Figure 5.7: Signal-Theft State Machine

5.4.2 Atomic Limit Counter Discussion

이것은 이 카운터가 그 양쪽 한계까지 닿는 것을 실제로 허용하는 첫번째 구현입니다만, fastpath에 어토믹 오퍼레이션을 추가하는 비용 아래 그렇게 하며, 이는 일부 시스템에서는 이 fastpath를 상당히 느려지게 만들니다. 일부 워크로드는 이 속도 저하를 감당할 수 있겠으나, 더 나은 읽기 쪽 성능을 위한 알고리즘을 찾아볼 가치가 있습니다. 그런 알고리즘들 중 하나는 다른 쓰레드로부터 카운트를 가져오기 위해 시그널 핸들러를 사용합니다. 시그널 핸들러는 시그널을 받은 쓰레드의 컨텍스트에서 수행되므로, 어토믹 오퍼레이션은 필요하지 않게 되는데, 다음 섹션에서 이를 보겠습니다.

Quick Quiz 5.47: 하지만 시그널 핸들러는 수행되는 동안 다른 CPU로 옮겨질 수 있습니다. 이 가능성은 한 쓰레드와 이 쓰레드를 언터럽트 한 시그널 핸들러 사이에서의 안정적인 통신을 위해 어토믹 인스트럭션과 메모리 배리어를 필요로 하게 하지 않나요?



5.4.3 Signal-Theft Limit Counter Design

이제 쓰레드별 상태는 이 연관된 쓰레드에 의해서만 조정되지만, 이 시그널 핸들러와의 동기화를 할 필요가 있을 겁니다. 이 동기화는 Figure 5.7에 보인 상태 머신에 의해 제공됩니다.

이 상태 머신은 IDLE 상태에서 시작되며, `add_count()`나 `sub_count()`가 이 쓰레드의 지역 카운트와 전역 카운트의 조합이 현재 요청을 처리할 수 없다는 것을 발견하면, 이 연관된 slowpath는 각 쓰레드의 `theft` 상태를 REQ로 설정합니다(이 쓰레드가 카운트

가 없어서 곧바로 READY로 가는 경우가 아니라면요). gblcnt_mutex 락을 잡는 slowpath 만이 초록색으로 표시되었듯 IDLE 상태로부터 다른 상태가 되는게 허용되어 있습니다.³ 그러면 이 slowpath는 각 쓰레드로 시그널을 보내고, 연관된 시그널 핸들러가 연관된 쓰레드의 theft 와 counting 변수를 체크합니다. 이 theft 상태가 REQ라면, 이 시그널 핸들러는 이 상태를 바꾸는 것이 허용되지 않으며, 따라서 그냥 리턴합니다. 그렇지 않고 counting 변수가 셋 되어 있어서 현재 쓰레드의 fastpath 가 진행 중임을 알리고 있다면, 이 시그널 핸들러는 theft 상태를 ACK로, 그렇지 않다면 READY로 바꿉니다.

만약 이 theft 상태가 ACK라면, 파란 색으로 표시되었듯 fastpath 만이 이 theft 상태를 바꾸는게 허용됩니다. 이 fastpath 가 완료되었을 때, theft 상태를 READY로 바꿉니다.

일단 이 slowpath 가 이 쓰레드의 theft 상태가 READY 임을 보면, 이 slowpath 는 이 쓰레드의 카운트를 가져갈 수 있습니다. 그러면 이 slowpath 는 이 쓰레드의 theft 상태를 IDLE로 설정합니다.

Quick Quiz 5.48: Figure 5.7에서, REQ theft 상태는 왜 빨간색으로 칠해졌나요?

■

Quick Quiz 5.49: Figure 5.7에서, 분리된 REQ 와 ACK theft 상태를 갖는 것의 요지가 무엇인가요? 왜 이것들을 하나의 REQACK 상태로 만들어서 이 상태 머신을 더 간단하게 만들지 않죠? 그러면 시그널 핸들러든 fastpath 든 먼저 그 상태에 도달한 사람이 상태를 READY로 만들면 될텐데요.

■

5.4.4 Signal-Theft Limit Counter Implementation

Listing 5.17 (count_lim_sig.c) 이 시그널 기반 카운터 구현에 사용되는 데이터 구조를 보입니다. 라인 1-7는 앞의 섹션에서 설명된 쓰레드별 상태 머신을 위한 상태들과 값들을 정의합니다. 라인 8-17는 앞의 구현들과 비슷하지만, 쓰레드의 countermax 와 theft 변수들에 원격 접속을 허용하기 위해 라인 14 와 15를 각각 추가했습니다.

Listing 5.18은 쓰레드별 변수들과 전역 변수들 사이에서 카운트를 올리기 위해 필요한 함수들을 보입니다. 라인 1-7는 앞의 구현과 동일한 globalize_count() 를 보입니다. 라인 9-19는 카운트 가져오기 프로세스에서 사용되는 시그널 핸들러인 flush_local_count_sig() 를 보입니다. 라인 11 와 12은 theft 상태가

³ 이 책의 흑백 버전을 위해서 말하자면, IDLE과 READY는 초록색, REQ는 빨강, 그리고 ACK는 파랑색으로 칠해져 있습니다.

Listing 5.17: Signal-Theft Limit Counter Data

```

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

```

REQ 인지 체크하고, 그렇지 않다면 변경 없이 리턴합니다. 라인 13은 값 가져오기 위한 변수의 샘플링이 해당 변수로의 어떤 변경보다도 이전에 이루어졌음을 보장하기 위한 메모리 배리어를 수행합니다. 라인 14은 theft 상태를 ACK 으로 설정하고, 라인 15 가 이 쓰레드의 fastpath 가 수행 중이지 않음을 보게 된다면, 라인 16 가 theft 상태를 READY로 설정합니다.

Quick Quiz 5.50: Listing 5.18 의 flush_local_count_sig() 함수에서, 쓰레드별 변수인 theft 의 사용은 왜 READ_ONCE() 와 WRITE_ONCE() 로 짜여있나요?

■

라인 21-49는 모든 쓰레드의 지역 카운트를 날리기 위해 slowpath에서 호출되는 flush_local_count() 를 보입니다. 라인 26-34의 루프는 지역 카운트를 가진 각 쓰레드의 theft 상태를 진행시키며, 또한 해당 쓰레드에 시그널을 보냅니다. 라인 27은 존재하지 않는 쓰레드들을 건너뛰게 합니다. 그렇지 않다면, 라인 28는 현재 쓰레드가 어떤 지역 카운트를 가지고 있는지 보고, 그렇지 않다면 라인 29은 이 쓰레드의 theft 상태를 READY로 만들고 라인 30에서 다음 쓰레드로 넘어갑니다. 그렇지 않다면, 라인 32는 이 쓰레드의 theft 상태를 REQ로 만들고 라인 33은 이 쓰레드에 시그널을 보냅니다.

Quick Quiz 5.51: Listing 5.18에서, 왜 line 28이 다른 쓰레드의 countermax 변수에 직접 접근하는게 안전하죠?

■

Quick Quiz 5.52: Listing 5.18에서, line 33은 왜 현재 쓰레드가 자신에게 시그널을 보내는지 체크하지 않는 거죠?

■

Listing 5.18: Signal-Theft Limit Counter Value-Migration Functions

```

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 (READ_ONCE(theft) != THEFT_REQ)
12         return;
13     smp_mb();
14     WRITE_ONCE(theft, THEFT_ACK);
15     if (!counting) {
16         WRITE_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                 WRITE_ONCE(*theftp[t], THEFT_READY);
30                 continue;
31             }
32             WRITE_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 (READ_ONCE(*theftp[t]) != THEFT_READY) {
39             poll(NULL, 0, 1);
40             if (READ_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         WRITE_ONCE(*theftp[t], THEFT_IDLE);
48     }
49 }
50
51 static void balance_count(void)
52 {
53     countermax = globalcountmax - globalcount -
54         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 }

```

Listing 5.19: Signal-Theft Limit Counter Add Function

```

1 int add_count(unsigned long delta)
2 {
3     int fastpath = 0;
4
5     WRITE_ONCE(counting, 1);
6     barrier();
7     if (READ_ONCE(theft) <= THEFT_REQ &&
8         countermax - counter >= delta) {
9         WRITE_ONCE(counter, counter + delta);
10        fastpath = 1;
11    }
12    barrier();
13    WRITE_ONCE(counting, 0);
14    barrier();
15    if (READ_ONCE(theft) == THEFT_ACK) {
16        smp_mb();
17        WRITE_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 }

```

Quick Quiz 5.53: Listings 5.17 and 5.18에 보인 코드는 GCC 와 POSIX에서 동작합니다. 이게 ISO C 표준도 다르게 하기 위해선 뭐가 필요할까요?



라인 35-48의 루프는 각 쓰레드가 READY 상태에 도달하길 기다렸다가 해당 쓰레드의 카운트를 가져옵니다. 라인 36-37는 모든 존재하지 않는 쓰레드를 건너뛰고, 라인 38-42의 루프는 현재 쓰레드의 theft 상태가 READY가 될 때까지 기다립니다. 라인 39은 우선순위 역전 문제를 방지하기 위해 1밀리세컨드 동안 블록되고, 라인 40가 이 쓰레드의 시그널이 아직 도착하지 않았음을 판단하면, 라인 41는 이 시그널을 다시 보냅니다. 수행은 이 쓰레드의 theft 상태가 READY가 되었을 때 라인 43에 도달하며, 따라서 라인 43-46는 값 가져오기를 합니다. 라인 47은 이제 이 쓰레드의 theft 상태를 IDLE로 되돌립니다.

Quick Quiz 5.54: Listing 5.18에서, 라인 41은 왜 시그널을 다시 보낼까요?



라인 51-63는 앞의 예와 비슷한 balance_count()를 보입니다.

Listing 5.20: Signal-Theft Limit Counter Subtract Function

```

1 int sub_count(unsigned long delta)
2 {
3     int fastpath = 0;
4
5     WRITE_ONCE(counting, 1);
6     barrier();
7     if (READ_ONCE(theft) <= THEFT_REQ &&
8         counter >= delta) {
9         WRITE_ONCE(counter, counter - delta);
10    fastpath = 1;
11 }
12 barrier();
13 WRITE_ONCE(counting, 0);
14 barrier();
15 if (READ_ONCE(theft) == THEFT_ACK) {
16     smp_mb();
17     WRITE_ONCE(theft, THEFT_READY);
18 }
19 if (fastpath)
20     return 1;
21 spin_lock(&gblcnt_mutex);
22 globalize_count();
23 if (globalcount < delta) {
24     flush_local_count();
25     if (globalcount < delta) {
26         spin_unlock(&gblcnt_mutex);
27         return 0;
28     }
29 }
30 globalcount -= delta;
31 balance_count();
32 spin_unlock(&gblcnt_mutex);
33 return 1;
34 }
```

Listing 5.19 은 `add_count()` 함수를 보입니다. Fastpath 는 라인 5-20 에 위치하며, slowpath 는 라인 21-35 에 있습니다. 라인 5 는 쓰레드별 `counting` 변수를 1 로 설정해서 이 쓰레드를 인터럽트하는 모든 뒤따르는 시그널 핸들러들은 `theft` 상태를 READY 가 아닌 ACK 으로 설정해서 이 fastpath 가 올바르게 완료되도록 합니다. 라인 6 은 컴파일러가 fastpath 몸체 중 어떤 것도 `counting` 설정을 앞서도록 재배치 하는 것을 방지합니다. 라인 7 와 8 는 이 쓰레드별 데이터가 `add_count()` 를 처리할 수 있는지, 그리고 진행 중인 값 가져오기가 없는지 검사하고, 그렇다면 라인 9 에서 이 fastpath 더하기를 수행하고 라인 10 에서 이 fastpath 가 취해졌음을 알립니다.

어떤 경우든, 라인 12 는 이 컴파일러가 fastpath 몸체가 라인 13 를 뒤따르도록 재배치 하는 것, 즉 모든 뒤따르는 시그널 핸들러가 값 가져오기를 제대로 못하게 할 수 있는 행위를 못하게 막습니다. 라인 14 는 또한 컴파일러의 재배치를 불가하게 하고, 이어서 라인 15 은 이 시그널 핸들러가 `theft` 의 READY 로의 상태 변경을 뒤로 미뤘는지 검사하고, 그렇다면 라인 16 에서 라인 17 가 상태를 READY 로 설정한 것을 보는 모든 CPU 는 라인 9 의 효과도 볼 수 있을 것을 확실히 합니다. 라인 9 에서의 fastpath 더하기가 수행되었다면, 라인 20 는 성공을 리턴합니다.

Listing 5.21: Signal-Theft Limit Counter Read Function

```

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

Listing 5.22: Signal-Theft Limit Counter Initialization Functions

```

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(EXIT_FAILURE);
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 }
```

그렇지 않다면, 우린 라인 21 에서 시작하는 slowpath 로 넘어갑니다. 이 slowpath 의 구조는 앞의 예들과 비슷하므로, 그 분석은 독자 여러분의 연습문제로 남겨둡니다. 비슷하게, Listing 5.20 의 `sub_count()` 의 구조는 `add_count()` 의 것과 동일하므로, `sub_count()` 의 분석 역시 독자 여러분들의 연습문제로 남겨두며, Listing 5.21 의 `read_count()` 역시 마찬가지입니다.

Listing 5.22 의 라인 1-12 는 `flush_local_count_sig()` 를 SIGUSR1 의 핸들러로 설정하고, `flush_local_count()` 에서의 `pthread_kill()` 호출이 `flush_local_count_sig()` 를 실행시키게 하는 `count_init()` 를 보입니다. 쓰레드 등록과 등록 해제

를 위한 코드는 앞의 예와 비슷하므로, 그에 대한 분석은 독자 여러분의 연습 문제로 남겨둡니다.

5.4.5 Signal-Theft Limit Counter Discussion

제 코어 여섯개짜리 x86 랩톱에서 이 시그널 기반 구현은 어토믹 구현에 비해 8배나 빨리 돌아갑니다. 이게 항상 선호될 만 할까요?

이 시그널 기반 구현은 펜티엄 4 시스템에서는 어토믹 인스트럭션이 느리므로 상당히 선호될 수 있겠습니다만, 오래된 80386 기반의 *Sequent Symmetry* 시스템은 어토믹 구현의 더 짧은 코드 길이를 훨씬 잘 수행해낼 수 있을 겁니다. 그러나, 이 증가된 업데이트 쪽 성능은 높아진 읽기 쪽 오버헤드의 비용과 함께 옵니다: 이 POSIX 시그널은 공짜가 아닙니다. 만약 궁극의 성능이 핵심이라면, 여러분은 여러분의 어플리케이션이 실제 배포되는 시스템에서도 그것들을 모두 측정해 봄야 합니다.

Quick Quiz 5.55: POSIX 시그널은 느리기만 한 게 아니고, 시그널을 각 쓰레드에 보내는 것은 확장이 안됩니다. 만약 여러분이 (예를 들어) 10,000 쓰레드를 가지고 있고 이 읽기 쪽이 빠르게 만들어야 한다면 어떻게 하겠습니까?

이건 하지만 왜 고품질 API 가 그렇게 중요한가에 대한 한가지 이유에 불과합니다: 그것은 계속해서 바뀌는 하드웨어 성능 특성에 의해 필요시 되는대로 구현이 바뀔 수 있게 해줍니다.

Quick Quiz 5.56: 만약 여러분이 원하는 건 이 정확한 리미트 카운터가 아래쪽 한계에는 정확하지만 위쪽 한계에는 정확하지 않아도 되는 것이라면 어떻게 하시겠습니까?

5.4.6 Applying Exact Limit Counters

이 섹션에서 소개된 정확한 한계 카운터 구현들이 매우 유용할 수 있긴 하지만, 이 카운터의 값이 항상 0 근처라면 별로 도움이 되지 못할 텐데, I/O 기기로의 액세스 수를 세는 게 그럴 수도 있을 겁니다. 그런 0 근처에서의 카운팅의 높은 오버헤드는 우리가 거기 얼마나 많은 참조가 있는지에 대해선 보통 관심이 없다는 점에서 특히 고통스럽습니다. Quick Quiz 5.5 에 나온 제거 가능한 I/O 기기 액세스 카운트 문제에서 이야기 되었듯이, 액세스 횟수는 누군가가 정말로 이 기기를 제거하려고 할 때 같은 드문 경우를 제외하고는 상관없습니다.

이 문제에 대한 한가지 간단한 해결책은 이 카운터가 효율적으로 동작할 수 있을 만큼 그 값이 0으로부터 멀다는 것을 보장하기 위해 큰 “편향치” (예를 들어, 10

억 정도) 를 더하는 것입니다. 누군가가 이 기기를 제거하고자 한다면, 이 편향치는 이 카운터 값으로부터 빼집니다. 마지막 몇 액세스를 카운팅 하는건 상당히 비효율적이게 되겠지만, 그 전의 많은 액세스는 온전한 속도로 카운트 되었을 거라는 게 핵심입니다.

Quick Quiz 5.57: 편향된 카운터를 사용할 때 상황을 더 좋게 하기 위해 여러분이 해보았을 법한 더 나은 방법은 무엇이 있을까요?

편향된 카운터가 상당히 도움되고 유용할 수 있지만, 이는 page 49 에서 나온, 제거 가능한 I/O 기기 액세스 카운트 문제의 부분적 해결책일 뿐입니다. 기기를 제거하고자 할 때, 우린 현재 I/O 액세스의 정확한 횟수만 알아야 하는 것이 아니라, 시작부터의 미래 액세스도 예측해야 합니다. 이를 달성하는 한가지 방법은 이 카운터를 업데이트 할 때 reader-writer 락을 읽기 모드로 획득하고, 이 카운터를 체크할 때에는 쓰기 모드로 획득하는 것입니다. I/O 를 하기 위한 코드는 다음과 같을 수 있을 겁니다:

```

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 1 는 이 락을 읽기 모드로 획득하고, 라인 3 또는 7에서 이를 해제합니다. 라인 2 는 이 기기가 제거되었는지 검사하고, 그렇다면 라인 3에서 이 락을 해제하고 라인 4에서 이 I/O 를 취소하거나 이 기기가 제거될 것임을 생각할 때 적절할 무슨 행동이든 취합니다. 그렇지 않다면, 라인 6 는 이 액세스 카운트를 증가시키고, 라인 7 는 이 락을 해제하며, 라인 8 가 I/O 를 행하고, 라인 9 는 이 액세스 카운트를 감소시킵니다.

Quick Quiz 5.58: 웃기네요! 이 카운터를 업데이트하기 위해 reader-writer 락을 읽기 모드로 획득한다구요? 뭐하는 짓이예요???

이 기기를 제거하기 위한 코드는 다음과 같을 수 있습니다:

```

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 1 는 이 락을 쓰기 모드로 획득하고 라인 4 은 이를 해제합니다. 라인 2 는 이 기기가 제거되는 중임을 알리고, 라인 5-7 의 루프는 모든 I/O 오퍼레이션이 완료 되기를 기다립니다. 마지막으로, 라인 8 는 기기 제거를 준비하기 위해 필요한 모든 추가적 처리를 행합니다.

Quick Quiz 5.59: 실제 시스템에서는 어떤 다른 문제들이 해결되어야 할까요?



5.5 Parallel Counting Discussion

This idea that there is generality in the specific is of far-reaching importance.

Douglas R. Hofstadter

이 챕터에서는 전통적인 카운팅 기능을 둘러싼 안정성, 성능, 그리고 확장성 문제를 소개했습니다. C 언어의 ++ 오퍼레이터는 멀티쓰레드 코드에서 안정적으로 동작할 것으로 보장되지 않으며, 단일 변수로의 아토믹 오퍼레이션은 성능도 확장성도 좋지 않습니다. 그래서 이 챕터에서는 일부 특수 경우에 성능도 확장성도 무척 좋을 수 있는 카운팅 알고리즘을 여러개 보였습니다.

이 카운팅 알고리즘들에서의 교훈들을 다시 돌아볼 가치가 있을 겁니다. 그런 이유로, Section 5.5.1 에서는 성능과 확장성을 요약하고, Section 5.5.2 에서는 특수화의 필요를 논하며, 마지막으로 Section 5.5.3 에서는 배운 교훈들을 나열하고 이 교훈들을 바탕으로 확장될 뒤의 챕터들로의 주의를 요청합니다.

5.5.1 Parallel Counting Performance

Table 5.1 의 위쪽 절반은 앞서 본 네개의 병렬 통계적 카운팅 알고리즘들의 성능을 보입니다. 네개의 알고리즘 모두 업데이트에 대해 거의 완벽한 선형 확장성을 보입니다. 이 쓰레드별 변수 구현 (count_end.c) 은 배열 기반 구현 (count_stat.c) 보다 업데이트에 있어 무척 빠르지만, 많은 수의 코어 위에서는 읽기가 더 느리며, 많은 병렬 읽기 쓰레드가 존재할 때에는 상당한 락 경쟁으로 힘들어합니다. 이 경쟁은 Chapter 9 에서 소개하는 뒤로 미뤄 처리하기 기법으로 Table 5.1 의 count_end_rcu.c 열에서 보였듯 해결될 수 있습니다. 미뤄 처리하기는 또한 결과적 일관성 덕분에 count_stat_eventual.c 열에서도 빛을 발합니다.

Quick Quiz 5.60: Table 5.1 의 count_stat.c 열에서, 우린 읽기쪽의 쓰레드 수에 따른 선형적 확장성을 볼 수 있습니다. 더 많은 쓰레드가 존재할수록 더 많

은 쓰레드별 카운터가 합산되어야 하는데 이게 어떻게 가능하죠?



Quick Quiz 5.61: Table 5.1 의 네번째 열에서 조차도, 이 통계적 카운터 구현의 읽기쪽 성능은 상당히 무섭네요. 그런데도 왜 이걸 신경쓰죠?



Table 5.1 의 아래쪽 절반은 병렬 한계 카운팅 알고리즘의 성능을 보입니다. 한계의 정확한 제한은 업데이트 쪽 성능 페널티에 큰 영향을 끼칩니다. 이 x86 시스템에서 그 페널티는 아토믹 오퍼레이션을 시그널로 대체함으로써 크게 줄어들지만요. 이 모든 구현이 동시에 읽기 쓰레드들이 있을 때에는 읽기 쪽 락 경쟁으로 고통받습니다.

Quick Quiz 5.62: Table 5.1 의 아래쪽 절반에 보여진 성능 데이터를 놓고 보면, 우린 항상 아토믹 오퍼레이션 보다 시그널을 선호해야 하겠군요, 그렇죠?



Quick Quiz 5.63: Table 5.1 의 아래쪽 절반에 보여진 읽기 쓰레드의 락 경쟁을 해결하기 위해 진보된 기법들이 사용될 수 있을까요?



요약하자면, 이 챕터는 여러 특수한 경우에 성능도 확장성도 굉장히 좋은 카운팅 알고리즘 여럿을 보였습니다. 하지만 우리의 병렬 카운팅은 특수 경우에만 국한되어야만 할까요? 모든 경우에 효율적으로 동작하는 범용 알고리즘을 가지면 더 좋지 않을까요? 다음 섹션인이 질문들을 돌아봅니다.

5.5.2 Parallel Counting Specializations

이 알고리즘이 각각의 특수한 경우에만 잘 동작한다는 사실은 일반적인 병렬 프로그래밍의 주요 문제로 여겨질수도 있습니다. 어쨌건, C 언어의 ++ 오퍼레이터는 싱글 쓰레드 코드에서는 잘 동작하는데, 특수한 경우가 아니라 일반적인 경우에 그렇습니다. 그렇죠?

이 논리는 약간의 사실을 담고 있지만, 핵심은 방향을 잘못 잡고 있습니다. 문제는 병렬성이 아니라 확장성입니다. 이를 이해하기 위해, 먼저 C 언어의 ++ 오퍼레이터에 대해 생각해 봅시다. 사실은 그것이 일반적으로 잘 동작하는게 아니고, 제한된 범위의 숫자에 대해서만 그렇다는 것입니다. 여러분이 1,000 자리 십진수를 처리해야 한다면, C 언어 ++ 오퍼레이터는 여러분을 위해 잘 동작하지 않을 것입니다.

Quick Quiz 5.64: ++ 오퍼레이터는 1,000 자리 숫자에도 잘 동작해요! 오퍼레이터 오버로딩이라고 못들어 보셨나요???



Table 5.1: Statistical/Limit Counter Performance on x86

Algorithm (count_*.c)	Section	Exact?	Updates (ns)	Reads (ns)			
				1 CPU	8 CPUs	64 CPUs	420 CPUs
stat	5.2.2		6.3	294	303	315	612
stat_eventual	5.2.4		6.4	1	1	1	1
end	5.2.3		2.9	301	6,309	147,594	239,683
end_rcu	13.5.1		2.9	454	481	508	2,317
lim	5.3.2	N	3.2	435	6,678	156,175	239,422
lim_app	5.3.4	N	2.4	485	7,041	173,108	239,682
lim_atomic	5.4.1	Y	19.7	513	7,085	199,957	239,450
lim_sig	5.4.4	Y	4.7	519	6,805	120,000	238,811

이 문제는 숫자에만 국한된 게 아닙니다. 여러분이 데이터를 저장하고 검색해야 한다고 생각해 봅시다. ASCII 파일을 사용해야 할까요? XML? 관계형 데이터베이스? 링크드 리스트? 밀집된 배열? B-tree? Radix tree? 또는 데이터가 저장되고 검색되는 것을 허용하는 다른 수많은 자료 구조와 환경 중 하나? 이는 여러분이 무엇을 해야 하는지, 얼마나 빨리 해야 하는지, 그리고 여러분의 데이터 셋이 얼마나 큰지에 달려있습니다—순차적 시스템에서 조차도요.

비슷하게, 여러분이 카운팅을 해야 한다면, 여러분의 해결책은 얼마나 큰 수를 다뤄야 하는지, 얼마나 많은 CPU 가 해당 숫자를 동시에 처리해야 하는지, 그 숫자가 어떻게 사용되는지, 그리고 어떤 수준의 성능과 확장성을 여러분이 필요로 하는지에 의존될 것입니다.

이 문제는 또한 소프트웨어만의 것도 아닙니다. 사람들이 작은 개울을 건널 수 있게 하기 위한 다리를 설계하는 것은 하나의 나무 널빤지를 만드는 것만큼이나 간단할 수도 있습니다. 하지만 콜럼비아 강의 수 킬로미터에 달하는 거리를 위해서라면 널빤지 하나만 사용하지는 않을 것이고, 콘크리트 트럭을 옮겨야 하는 다리를 위해서는 앞의 설계는 조언이 될 수도 없을 겁니다. 짧게 말해서, 다리 설계가 증가하는 길이와 부하에 따라 달라져야 하는 만큼, 소프트웨어 설계 역시 증가하는 CPU 갯수에 따라 달라져야 합니다. 그렇다고는 하나, 이 과정을 자동화 해서, 소프트웨어가 하드웨어 구성과 워크로드의 변화에 적응할 수 있게 하는게 좋을 겁니다. 실제로 이런 종류의 자동화에 대한 연구가 있었으며 [AHS⁺03, SAH⁺03], 리눅스 커널은 제한된 정도의 바이너리 재작성을 포함한 부팅 시점의 재설정을 가지고 있습니다. 이런 종류의 적응은 주류 시스템의 CPU 수가 계속 증가하는 만큼 점점 더 중요해 질 겁니다.

요약하자면, Chapter 3에서 이야기 된 것처럼, 물리 법칙은 그것이 다리와 같은 기계 장치들에 제약을 가하는 것과 같이 병렬 소프트웨어에도 제약을 가합니다. 이런

제약들은 특수화를 강요합니다, 소프트웨어의 경우에는 실제 하드웨어와 워크로드에 맞춰 특수화 선택을 자동화 할 수 있을 수도 있지만 말입니다.

물론, 범용화 된 카운팅조차 상당히 특수화 되어 있습니다. 우린 컴퓨터를 가지고 다른 많은 일도 해야만 합니다. 다음 섹션은 우리가 카운터에서 얻은 교훈들을 이 책의 뒷부분에서 다룰 주제들과 연결지어 봅니다.

5.5.3 Parallel Counting Lessons

이 챕터를 시작하는 문단은 우리의 카운팅에 대한 연구가 병렬 프로그래밍으로의 훌륭한 소개를 제공할 것이라고 약속했습니다. 이 섹션은 이 챕터에서의 교훈과 뒤의 여러 챕터에서 소개될 것들 사이의 명확한 연결을 만듭니다.

이 챕터의 예제들은 중요한 확장성과 성능을 위한 도구가 분할하기 임을 보였습니다. 카운터는 Section 5.2에서 논한 통계적 카운터에서처럼 완전히 분할 될 수도 있고, Sections 5.3 and 5.4에서 이야기 된 한계 카운터에서처럼 부분적으로 분할될 수도 있습니다. 분할하기는 Chapter 6에서 훨씬 더 깊게 다뤄질 것이고, 부분적 분할하기는 Section 6.4에서 *parallel fastpath* 라는 이름으로 특별히 다뤄질 것입니다.

Quick Quiz 5.65: 하지만 우리가 모든 것을 분할해야 한다면, 왜 공유 메모리 멀티쓰레딩을 고려하죠? 문제를 완전하게 분할하고 각각 각자의 주소공간을 가지는 여러 프로세스로 돌리는 게 어떤가요?



이 부분적으로 분할된 카운팅 알고리즘은 전역 데이터를 보호하기 위해 락킹을 사용했고, 락킹은 Chapter 7의 주제입니다. 대조적으로, 분할된 데이터는 연관된 쓰레드의 완전한 제어 하에 있어서 어떤 동기화도 필요치 않게 되는 경향이 있었습니다. 이 데이터 소유권은 Section 6.3.4에서 소개되고 Chapter 8에서 더 자세히 다뤄질 겁니다.

정수 더하기와 빼기는 일반적인 동기화 오퍼레이션에 비하면 무척이나 비용이 저렴해서, 합리적인 확장성을 이루기 위해선 동기화 오퍼레이션을 검소하게 사용할 것이 요구됩니다. 이를 해내는 한가지 방법은 이 더하기와 빼기 오퍼레이션을 한번에 몰아서 해서, 이 비용이 저렴한 오퍼레이션이 여러개가 하나의 동기화 오퍼레이션으로 처리되게 하는 것입니다. 한번에 몰아서 처리하기 최적화의 또 다른 종류의 하나는 Table 5.1에 리스트 된 카운팅 알고리즘들 각각에 의해 사용되었습니다.

마지막으로, Section 5.2.4에서 이야기 된 결과적으로 일관적인 통계적 카운터는 일을 뒤로 미루는 것(이 경우, 전역 카운터를 업데이트 하는 것)이 상당한 성능과 확장성이득을 제공함을 보였습니다. 이 방법은 일반적인 경우를 위한 코드가 그렇지 않다면 사용할 수 있을 법한 것들보다 훨씬 비용이 저렴한 동기화 오퍼레이션을 사용하는 것이 가능하게 했습니다. Chapter 9는 미뤄서 처리하기가 성능, 확장성, 심지어 real-time 반응까지 향상시킬 수 있는 몇 가지 방법을 더 알아봅니다.

요약을 요약하자면:

1. 분할하기는 성능과 확장성을 향상시킵니다.
2. 부분적 분할하기, 즉 일반적 코드 패쓰에만 분할하기를 적용하는 것은 거의 항상 잘 동작한다.
3. 부분적 분할하기는 코드에 적용될 수 있지만 (Section 5.2의 통계적 카운터의 분할된 업데이트와 분할되지 않은 읽기처럼), 시간에 대해서도 적용될 수 있습니다 (Section 5.3과 Section 5.4의 한계로부터 멀때는 훨씬 빠르게 동작하고, 한계에 가까울 때에는 느리게 동작하는 한계 카운터들처럼).
4. 시간을 통한 분할하기는 비싼 전역 오퍼레이션의 횟수를 줄이고, 그럼으로써 동기화 오버헤드를 줄이고, 결국 성능과 확장성을 향상시키기게끔 종종 업데이트를 지역적으로 몰아서 합니다. Table 5.1은 몰아서 하기를 상당히 많이 사용합니다.
5. 읽기만 하는 코드 패쓰는 읽기만 하도록 남아있어야 합니다: 공유 메모리로의 가짜 동기화 쓰기는 Table 5.1의 `count_end.c` 열에서 보여진 것처럼 성능과 확장성을 죽일 수 있습니다.
6. 자연의 현명한 사용은 Section 5.2.4에서 보인 것처럼 성능과 확장성을 향상시킵니다.
7. 병렬 성능과 확장성은 보통 균형잡기입니다: 어떤 지점을 넘어서면, 일부 코드 패쓰를 최적화 하는 것은 다른 부분의 성능을 하락시킵니다. Table 5.1의 `count_stat.c` 와 `count_end_rcu.c` 열이 이 점을 잘 보입니다.

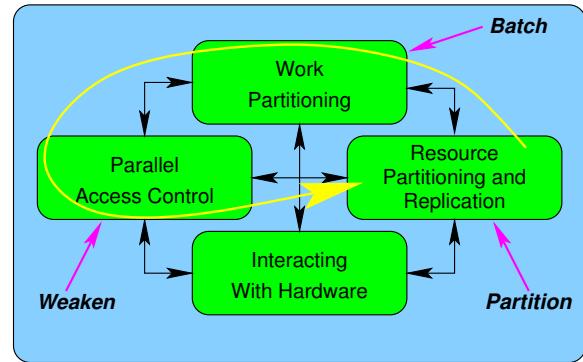


Figure 5.8: Optimization and the Four Parallel Programming Tasks

8. 다른 수준의 성능과 확장성은 다른 많은 요소들처럼 알고리즘과 자료구조 설계에 영향을 끼칩니다. Figure 5.1 가 이 점을 잘 보이고 있습니다: 어토믹 값 증가는 두개 CPU를 갖는 시스템에서는 완전히 허용 가능하지만 여덟개 CPU를 갖는 시스템에서는 말도 안됩니다.

계속해서 요약을 해보자면, 우린 “큰 세개의” 성능과 확장성 증가 방법이 있는데, 각각 (1) CPU 와 쓰레드들에 걸쳐 분할하기, (2) 더 많은 일이 각각의 비용 높은 동기화 오퍼레이션을 통해 처리되도록 몰아서 처리하기, 그리고 (3) 가능한 곳에서는 동기화 오퍼레이션을 약화시키기. 대략적인 법칙으로, 여러분은 이 방법들을 page 15의 Figure 2.6의 토론에서 앞서 이야기 되었듯이 순서대로 적용해야 합니다. Figure 5.8에 보여진 대로, 분할하기 최적화는 “Resource Partitioning and Replication”에, 몰아서 처리하기 최적화는 “Work Partitioning”으로, 그리고 약화시키기 최적화는 “Parallel Access Control”에 적용되어야 합니다. 물론, 디지털 신호 처리기 (DSP), field-programmable gate arrays (FPGAs), 또는 범용 그래픽 처리기 (GPGPU)를 가지고 있다면, 설계 과정 중에 “Interacting With Hardware” 부분에 대해서도 많은 관심을 기울여야 합니다. 예를 들어, GPGPU의 하드웨어 쓰레드와 메모리 연결성 구조는 매우 주의 깊은 분할하기와 몰아서 처리하기 설계 결정에 큰 영향을 끼칠겁니다.

짧게 말해서, 이 챕터의 시작점에서 이야기 되었듯, 카운팅의 단순성은 복잡한 동기화 기능들이나 복잡한 데이터 구조로부터 방해받지 않고서 많은 기본적 동시성 문제를 탐험할 수 있게 해주었습니다. 그런 동기화 도구들과 데이터 구조들은 뒤의 챕터에서 다루어집니다.

Chapter 6

Partitioning and Synchronization Design

이 챕터는 성능, 확장성, 그리고 응답 시간을 균형맞추기 위해 관용구 또는 “디자인 패턴” [Ale79, GHJV95, SSRB00]을 사용해서 현대의 상용 멀티코어 시스템의 장점을 갖추도록 소프트웨어를 어떻게 설계하는지 설명합니다. 제대로 조개진 문제는 간단하고, 확장 가능하며, 높은 성능을 갖춘 해결책을 이끄는 반면, 잘 조개지지 못한 문제는 느리고 복잡한 해결책을 이끌어냅니다. 이 챕터는 여러분의 코드에 조개기를 몰아서 하기 (batching) 와 규칙 약화시키기 (weakening) 과 함께 설계하는 것을 도울 겁니다. “설계”라는 단어가 무척 중요합니다: 여러분은 조개기를 첫번째로, 두번째로 몰아서 하기, 세번째로 규칙 약화시키기를, 그리고 네번째로 코드를 짜야 합니다. 이 순서를 바꾸는 것은 종종 대단한 좌절감과 함께 안 좋은 성능과 확장성을 이끌어냅니다.¹

그러므로, Section 6.1에서는 파티셔닝 (partitioning) 연습문제를 소개하고, Section 6.2에서는 파티셔닝 가능성 설계 영역을 리뷰하며, Section 6.3에서 동기화 단위 크기 선택을 이야기 하고, Section 6.4에서 일반적인 경우에 속도와 확장성을 제공하는 fastpath를 제공하면서 일반적이지 않은 경우에는 더 간단하지만 덜 확장성 있는 “slow path”를 사용하는, 중요한 병렬-fastpath 디자인 패턴을 개략적으로 살펴보며, 마지막으로 Section 6.5에서는 파티셔닝 다음 영역을 간략하게 바라봅니다.

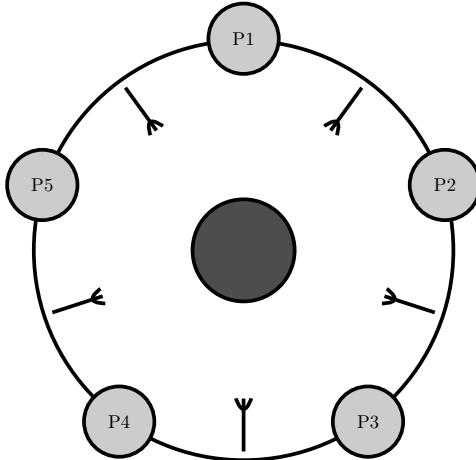


Figure 6.1: Dining Philosophers Problem

6.1 Partitioning Exercises

Whenever a theory appears to you as the only possible one, take this as a sign that you have neither understood the theory nor the problem which it was intended to solve.

Karl Popper

파티셔닝이 2000년대 초에 그랬던 것보다 더 널리 이해되고 있지만, 그 가치는 여전히 과소평가 되어 있습니다. 따라서 Section 6.1.1에서는 고전의 식사하는 철학자들 (Dining Philosophers) 문제를 더 병렬적인 관점으로 바라보고 Section 6.1.2에서는 양극단을 가지는 큐 (queue)를 다시 봅니다.

¹ 물리 법칙에 대한 다른 훌륭한 회피 기술인 읽기 전용 복사는 (read-only replication) Chapter 9에서 다뤄집니다.

6.1.1 Dining Philosophers Problem

Figure 6.1는 고전의 Dining Philosophers problem [Dij71]의 다이어그램을 보입니다. 이 문제는 생각하고 먹기 위해 두개의 포크가 필요한 “무척 어려운 종류의 스파게티”를 먹는 다섯명의 철학자들로 구성됩니다.² 한명의 철학자는 그 또는 그녀의 바로 오른쪽과 왼쪽에 있는 포크만 사용할 수 있는데, 충분히 스파게티를 먹기 전까진 포크를 내려놓지 않습니다.

목표는 말 그대로 기아를 방지할 수 있는 알고리즘을 만드는 것입니다. 가능한 한가지 기아 시나리오는 모든 철학자가 각자의 왼쪽 포크를 동시에 집어드는 경우입니다. 이들 중 누구도 그들이 식사를 끝내기 전까지는 자신의 포크를 내려놓지 않을 것이므로, 그리고 이들 중 누구도 이들 중 한명이라도 식사를 끝내기 전까지는 두번째 포크를 갖지 못할 것이므로, 이들은 모두 굶게 됩니다. 최소 한명의 철학자를 식사할 수 있게 하는 것만으로는 충분치 않음을 알아 두시기 바랍니다. Figure 6.2가 보이듯, 오직 소수의 철학자가 기아에 빠지는 것조차도 방지되어야 합니다.

Dijkstra의 해결책은 1980년대 말 또는 1990년대 초에는 적절치 못하게 된, 무시할만한 통신 딜레이이라는 가정을 적용하면 잘 동작하는 전역 세마포어를 사용했습니다.³ 보다 최신의 해결책은 Figure 6.3에 보인 것처럼 포크에 수를 매기는 것입니다. 각 철학자는 그 또는 그녀의 접시 옆에 있는 더 낮은 수를 가지는 포크를 집고, 그다음 다음 포크를 집습니다. 따라서, 이 그림에서 가장 위쪽에 앉은 철학자는 왼쪽 포크를 먼저 집어들고, 이어서 오른쪽 포크를 집어드는데, 나머지 철학자들은 각자의 오른쪽 포크를 먼저 집어듭니다. 두명의 철학자들이 포크 1을 먼저 집어들려고 노력할 것이므로, 그리고 이 두 철학자들 중 한명만이 성공할 것이므로, 네명의 철학자에게 다섯개의 포크가 사용 가능하게 될 겁니다. 이 네명의 철학자 중 최소 한명은 두개의 포크를 가지게 될거고 따라서 식사를 할 수 있습니다.

이 자원에 숫자를 매기고 그 숫자 순서대로 자원을 획득하는 일반적인 기법은 테드락 방지 기법으로 널리 사용되었습니다. 하지만, 모두가 배고픈데 한번에 단 한명의 철학자만이 식사를 하는 상황이 초래되는 사건의 연속을 쉽게 상상해 볼 수 있습니다:

1. P2 가 포크 1 을 집어들어, P1 이 포크를 집는 걸 방지합니다.
2. P3 가 포크 2 를 집어듭니다.

² 포크 대신 젓가락으로 생각해도 좋습니다.

³ 2021년의 시각에서 Dijkstra 를 모욕하기는 너무도 쉽습니다. 50년이나 지난 이야기니까요. 여전히 Dijkstra 를 모욕해야 한다고 느끼신다면, 저는 무언가를 출판하고, 50년을 기다린 후, 여러분의 아이디어가 그 시간동안의 시험을 얼마나 잘 버텨냈는지 보라는 조언을 하겠습니다.

3. P4 가 포크 3 를 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.

요약하자면, 이 알고리즘은 동시에 두 철학자가 식사를 하기 충분한 것 이상의 포크가 존재함에도 불구하고 모든 철학자가 배고플 때에도 한번에 하나의 철학자만 식사를 하는 상황을 초래할 수 있습니다. 이보다 더 잘할 수 있어야 합니다!

한가지 해결책이 Figure 6.4에 그려져 있는데, 이 파티셔닝 기법을 더 잘 보이기 위해 다섯명이 아닌 네명의 철학자만 포함하고 있습니다. 여기서 위쪽과 오른쪽의 철학자들은 한쌍의 포크를 공유하며, 아래쪽과 왼쪽의 철학자는 다른 한쌍의 포크를 공유합니다. 만약 모든 철학자들이 동시에 배고파지면, 최소 두명은 항상 동시에 식사를 할 수 있습니다. 또한, 그림에 보여져 있듯이, 이 포크들은 이제 한쌍으로 묶여있을 수 있어서 두개씩 동시에 집어지고 내려놓아질 수 있게 되어, 획득과 해제 알고리즘을 단순화 시킵니다.

Quick Quiz 6.1: 이 Dining Philosophers 문제에 더 나은 해결책이 있을까요?

이는 “수평 병렬성” [Inm85] 또는 “데이터 병렬성”의 한 예로, 이 철학자들 쌍 간에는 어떤 의존성이 없기 때문에 그렇게 이름지어졌습니다. 수평적으로 병렬인 데이터 처리 시스템에서, 데이터의 특정 항목은 복사된 소프트웨어 컴포넌트들 중 하나에 의해서만 처리될 겁니다.

Quick Quiz 6.2: 그리고 어떤 의미에서 이 “수평적 병렬성”은 “수평적”이라고 불릴 수 있는 건가요?

6.1.2 Double-Ended Queue

Double-ended queue 는 양 끝을 통해 추가되거나 제거될 수 있는 원소들의 리스트를 갖는 데이터 구조입니다 [Knu73]. 락 기반의 구현으로는 double-ended queue의 양 끝단에서의 동시적 운용이 어려움으로 알려져 있습니다 [Gro07]. 이 섹션은 파티셔닝 설계 전략이 어떻게 합리적으로 간단한 구현에 이르게 할 수 있는지 보이고, 뒤따르는 섹션들에서는 세개의 범용 접근법을 봅니다.

6.1.2.1 Left- and Right-Hand Locks

간단해 보이는 한가지 전략은 왼쪽 끝으로의 enqueue 와 dequeue 오퍼레이션을 위한 왼쪽 락과 오른쪽 끝으로의 오퍼레이션들을 위한 오른쪽 락을 갖는 doubly

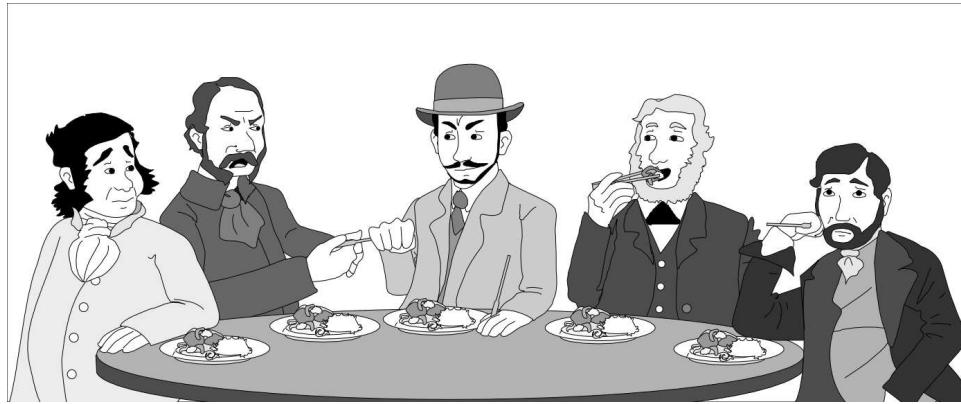


Figure 6.2: Partial Starvation Is Also Bad

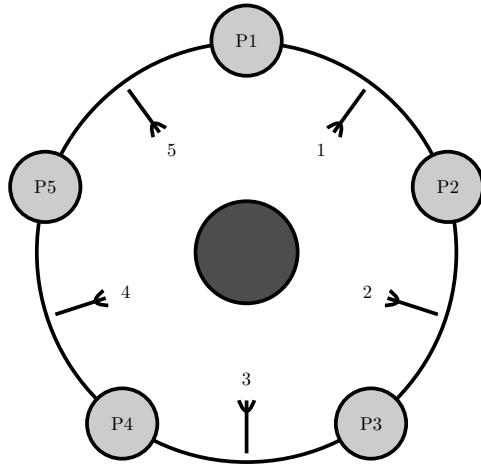


Figure 6.3: Dining Philosophers Problem, Textbook Solution

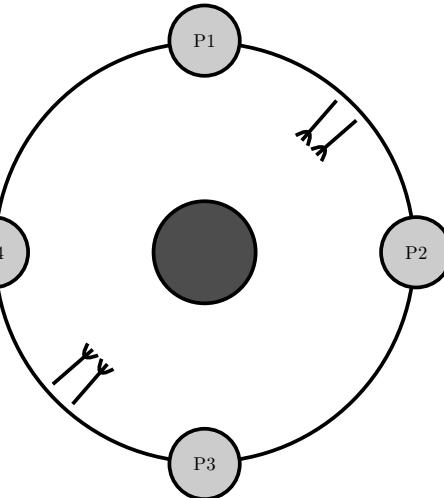


Figure 6.4: Dining Philosophers Problem, Partitioned

linked list를 사용하는 것으로, Figure 6.5에 보이는 것과 같습니다. 하지만, 이 방법의 문제는 이 리스트에 네개 미만의 원소만이 존재할 때에는 이 두 락의 도메인이 겹친다는 것입니다. 이 겹침은 어떤 원소를 제거하는 것이 그 원소만이 아니라 그것의 왼쪽과 오른쪽 이웃 원소에게도 영향을 끼친다는 사실 때문입니다. 이 도메인들은 이 그림에 색깔로 표시되어 있는데, 아래쪽으로의 줄무늬를 가진 파랑은 왼쪽 락의 도메인을, 위쪽으로의 줄무늬를 가진 빨강은 오른쪽 락의 도메인을, 그리고 (줄무늬가 없는) 보라색은 겹치는 도메인을 표시합니다. 이 방식으로 동작하는 알고리즘을 만드는 것도 가능하지만, 다섯개 미만이 아닌 특수 경우들이 존재한다는 사실은 커다랗고 빨간 경고등을 띠우는데, 이 리스트의 다른 끝쪽에서의 동시의 행동들이 이 queue를 언제든 하나의 특수 경우에서 다른 경우로 바꿀 수 있다는 점

에서 특히 그렇습니다. 다른 설계를 고려하는 게 훨씬 나을 겁니다.

6.1.2.2 Compound Double-Ended Queue

겹치지 않는 락 도메인을 강제하기 위한 한가지 방법이 Figure 6.6에 보여져 있습니다. 두개의 double-ended queue들이 동시에 동작하는데, 각각 자신의 락으로 보호됩니다. 이는 원소들이 결국은 한쪽 double-ended queue에서 다른 쪽으로 옮겨져야 하며, 이때는 양쪽 락이 모두 잡혀야만 함을 의미합니다. 테드락을 방지하기 위해 간단한 락 계층이 사용될 수 있는데, 예를 들면 오른쪽 락을 잡기 전에 항상 왼쪽 락을 잡는 겁니다. 그러면 우리는 조건 없이 왼쪽 queue에 원소를 왼쪽 집어넣기하고 오른쪽 queue에 원소를 오른쪽 집어넣기를 할 수

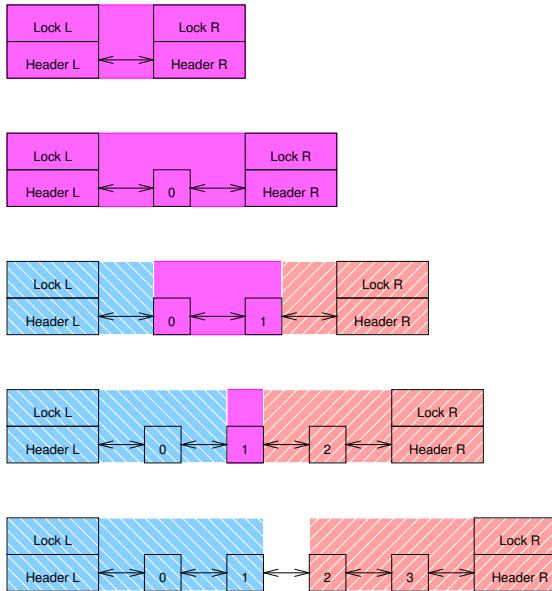


Figure 6.5: Double-Ended Queue With Left- and Right-Hand Locks

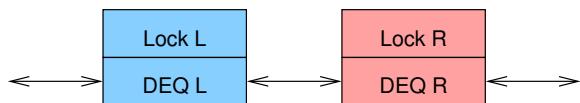


Figure 6.6: Compound Double-Ended Queue

있으므로, 같은 double-ended queue에 두개의 락을 적용하는 것보다는 훨씬 간단할 것입니다. 비어있는 queue에서 꺼내기를 하려 할 때 주요한 복잡도가 나타나는데, 이 경우에는 다음과 같은 처리가 필요합니다:

1. 오른쪽 락을 잡고 있다면, 이를 해제하고 왼쪽 락을 잡습니다.
2. 오른쪽 락을 잡습니다.
3. 두 queue에 원소를 고르게 분배시킵니다.
4. 제거하려는 원소가 있다면 제거합니다.
5. 두 락을 모두 해제합니다.

Quick Quiz 6.3: 이 compound double-ended queue 구현에서는, 이 락을 해제하고 재획득하는 동안에 이 queue가 비어있지 않게 되면 어떡해야 하죠?

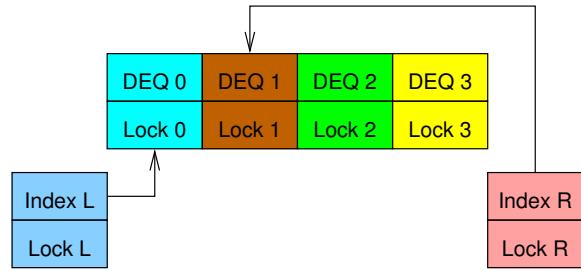


Figure 6.7: Hashed Double-Ended Queue

이 결과로 나오는 코드는 (`locktdeq.c`) 상당히 간단합니다. 원소 재분배 오퍼레이션은 주어진 원소를 두 queue 사이에서 어느 쪽으로든 옮길 수도 있어서, 시간을 낭비하고, 최적의 성능을 위해선 워크로드 종속적 휴리스틱을 필요로 할 수도 있습니다. 이게 어떤 경우에는 최선의 방법이 될 수도 있겠습니다만, 알고리즘을 더 큰 결정성을 가지고 시도해 보는 것도 흥미로울 겁니다.

6.1.2.3 Hashed Double-Ended Queue

데이터 구조를 결정론적으로 쪼개는 가장 효과적이고도 가장 간단한 방법은 해싱입니다. 각 원소에 이 리스트에서의 그것의 위치에 기반한 숫자를 부여해서 비어있는 queue에 왼쪽으로 들어온 첫번째 원소는 0으로 수가 부여되고 비어있는 queue에 오른쪽으로 들어온 첫번째 원소는 1이라는 수가 부여되는 식으로 double-ended queue를 간단히 해싱하는게 가능합니다. 왼쪽으로 들어오는 원소들은 계속해서 작아지는 수를 부여받고 (-1, -2, -3, ...), 오른쪽으로 들어오는 원소들은 계속해서 증가하는 수를 부여받습니다 (2, 3, 4, ...). 핵심은, 이 숫자가 이 queue에서의 위치를 암시하므로, 해당 원소의 수를 정말로 표현할 필요는 없다는 겁니다.

이 방법에서는, 우리는 왼쪽 인덱스를 보호하기 위해 하나의 락을 할당하고, 오른쪽 인덱스를 위해 또 다른 락을, 그리고 각 해쉬 체인을 위해 또 하나의 락을 사용합니다. Figure 6.7 가 네개의 해쉬 체인이 있다는 가정 하에 이로 인해 만들어지는 데이터 구조를 보입니다. 락 도메인은 겹치지 않으며, 데드락은 체인 락 전에 인덱스 락을 획득함으로써 회피되며, 특정 타입의(인덱스 또는 체인) 락은 한번에 두개 이상 획득하지 않습니다.

각 해쉬 체인은 그 자체로 double-ended queue이며, 이 예에서는 각각이 네개의 원소씩을 쥐고 있습니다. Figure 6.8 의 가장 위쪽은 하나의 원소가 (“R₁”) 오른쪽으로 넣어지고, 해쉬 체인 2를 참조하게끔 증가된 오른쪽 인덱스를 갖게 된 후의 상태를 보입니다. 이 그림의 가운데 부분은 세개의 원소가 추가적으로 오른쪽으로 넣어진 후의 상태를 보입니다. 보시다시피, 인덱스들은 각자의 초기 상태로 되돌아가져 있습니다 (Figure 6.7

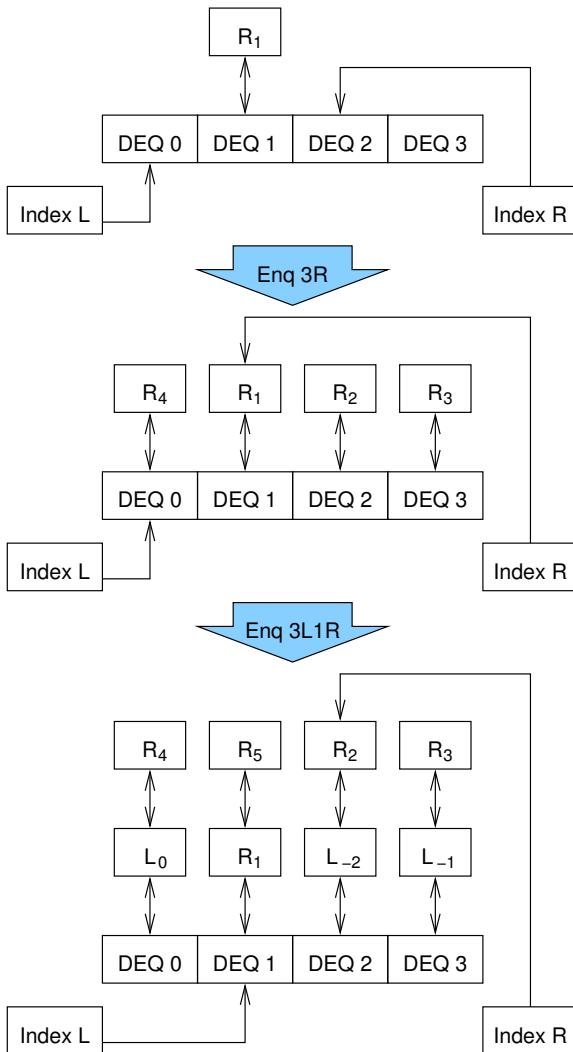


Figure 6.8: Hashed Double-Ended Queue After Insertions

를 보세요), 하지만 각 해시 체인은 이제 비어 있습니다. 이 그림의 아래쪽은 세개의 추가적인 원소가 왼쪽으로 들어오고 또 하나의 원소가 오른쪽으로 들어온 후의 상태를 보입니다.

Figure 6.8에 보인 마지막 상태로부터, 왼쪽 dequeue 오퍼레이션은 원소 “L₋₂”를 리턴하고, 왼쪽 인덱스가 해시 체인 2를 참조하는대로 둘 것이며, 이 체인은 단 하나의 원소만을 (“R₂”) 가지고 있게 됩니다. 이 상태에서, 왼쪽 enqueue가 오른쪽 enqueue와 동시에 수행되면 락 경쟁이 발생할 겁니다만, 그런 경쟁 상태의 확률은 더 큰 해시 테이블을 사용함으로써 무척 낮은 수준으로 줄일 수 있습니다.

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

Listing 6.1: Lock-Based Parallel Double-Ended Queue Data Structure

```

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

```

Figure 6.9는 16개의 원소가 네개 해시 버킷 기반의 병렬 double-ended queue에 어떻게 관리될지 보입니다. 아랫단의 단일 락 기반 double-ended queue 각각은 전체 병렬 double-ended queue의 1/4 조각을 갖습니다.

Listing 6.1은 평범하게 락으로 관리되는 double-ended-queue 구현을 제공하는 struct deq가 존재한다는 가정 하에 앞에서 이야기한 것에 연관되는 C-언어 데이터 구조를 보입니다. 이 데이터 구조는 라인 2에 왼쪽 락을, 라인 3에 왼쪽 인덱스를, 라인 4에 오른쪽 락을 (실제 구현에서는 캐쉬라인 사이즈로 정렬됨), 라인 5에 오른쪽 인덱스를, 그리고 마지막으로, 라인 6에 간단한 락 기반 double-ended queue의 배열을 가지고 있습니다. 고성능 구현체는 거짓 공유를 피하기 위해 패딩이나 특수한 정렬 지시어 등을 사용할 겁니다.

Listing 6.2(lockhdeq.c)은 enqueue와 dequeue 함수의 구현을 보입니다.⁴ 이야기는 왼쪽 오퍼레이션들에 집중할텐데, 오른쪽 오퍼레이션들은 쉽게 왼쪽의 것들로부터 나올 것이기 때문입니다.

라인 1-13는 왼쪽 dequeue를 하고 가능하면 원소를, 그렇지 않다면 NULL을 리턴하는 pdeq_pop_1()을 보입니다. 라인 6는 왼쪽 스핀락을 획득하고, 라인 7는 dequeue될 인덱스를 계산합니다. 라인 8는 이 원소를 꺼내고, 라인 lrefcheck에서 이 결과가 NULL이 아닐 것으로 판명되면, 라인 11에서 락을 해제하고, 마지막으로 라인 12에서 거기 원소가 있었다면 그것을, 아니면 NULL을 리턴합니다.

라인 29-38는 특정 원소를 왼쪽으로 enqueue하는 pdeq_push_1()을 보입니다. 라인 33는 왼쪽 락을 획득하고, 라인 34에서 왼쪽 인덱스를 잡습니다. 라인 35

⁴ 어떤 언어를 사용해서든 다양한 형태의 구현을 만들 수 있겠습니까만, 그건 독자 여러분의 연습 문제로 남겨 두겠습니다.

Listing 6.2: Lock-Based Parallel Double-Ended Queue Implementation

```

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 }

```

는 이 원소를 이 왼쪽 인덱스로 가리키지는 double-ended queue에 왼쪽으로 enqueue gkqslek. 라인 36는 이어서 이 왼쪽 인덱스를 업데이트하고 라인 37에서 이 락을 해제합니다.

앞에서도 이야기 했듯, 오른쪽 오퍼레이션들은 왼쪽의 것들과 완전히 비슷하므로, 그것들의 분석은 독자 여러분의 연습 문제로 남겨 둡니다.

Quick Quiz 6.4: 해쉬 기반의 double-ended queue는 좋은 해결책일까요? 왜 그렇고 왜 그렇지 않을까요?

6.1.2.4 Compound Double-Ended Queue Revisited

이 섹션은 조합된 double-ended queue를 빼어 있지 않은 queue에서 이제 빼어 있는 queue로 모든 원소를 옮기는 간단한 균형 맞추기를 사용하는 방법으로 다시 풀어봅니다.

Quick Quiz 6.5: 빈 queue로 모든 원소를 옮긴다구요? 이 미친 해결책이 대체 어떤 세상에서는 최적인거죠???

앞의 섹션에서 보인 해쉬 기반 구현과 대조적으로, 이 조합 구현은 락도 어토믹 오퍼레이션도 사용하지 않는 순차적 구현의 double-ended queue 위에서 구현될 겁니다.

Listing 6.3이 이 구현을 보이고 있습니다. 해쉬 기반의 구현과 달리, 이 조합 구현은 비대칭적이어서, pdeq_pop_l()과 pdeq_pop_r() 구현을 별개로 봐야만 합니다.

Quick Quiz 6.6: 조합된 병렬 double-ended queue 구현은 왜 대칭적일 수 없죠?

pdeq_pop_l() 구현이 이 코드의 라인 1~16에 보여져 있습니다. 라인 5은 왼쪽 락을 획득하는데, 이 락은 라인 14에서 해제됩니다. 라인 6은 아랫단의 double-ended queue의 왼쪽으로부터 원소를 빼내려 시도하고, 이게 성공하면 간단히 이 원소를 리턴하기 위해 라인 8~13을 건너뜁니다. 그렇지 않다면, 라인 8은 오른쪽 락을 획득하고, 라인 9에서 오른쪽 queue로부터 원소를 왼쪽 꺼내기하고 라인 10에서 오른쪽 queue의 모든 남아있는 원소를 왼쪽 queue로 옮기며, 라인 11에서 오른쪽 queue를 초기화하고, 라인 12에서 이 오른쪽 락을 해제합니다. 만약 존재한다면 라인 9에서 dequeue된 원소가 리턴됩니다.

pdeq_pop_r() 구현이 이 코드의 라인 18~38에 보여져 있습니다. 전과 같이, 라인 22은 이 오른쪽 락을 획득하고 (그리고 라인 36에서 이를 해제합니다), 라인 23에서 오른쪽 queue로부터 원소 하나를 오른쪽 꺼내기 하려 시도하고, 만약 성공한다면 이 원소를 단순히 리턴하기 위해 라인 25~35를 건너뜁니다. 하지만,

Listing 6.3: Compound Parallel Double-Ended Queue Implementation

```

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->rdeq);
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     spin_lock(&d->llock);
43     deq_push_l(e, &d->ldeq);
44     spin_unlock(&d->llock);
45 }
46
47 void pdeq_push_r(struct cds_list_head *e, struct pdeq *d)
48 {
49     spin_lock(&d->rlock);
50     deq_push_r(e, &d->rdeq);
51     spin_unlock(&d->rlock);
52 }

```

라인 24 이 dequeue 할 원소가 존재하지 않았음을 알아내면 라인 25 은 이 오른쪽 락을 해제하고 라인 26-27 가 올바른 순서로 두 락을 잡습니다. 라인 28 는 이제 오른쪽 queue 로부터 원소 하나를 다시 오른쪽 꺼내기 하려 시도하고, 라인 29 가 이 두번째 시도가 실패했음을 확인하면, 라인 30 은 왼쪽 queue 로부터 원소 하나를 오른쪽 dequeue 하고 (원소가 있었다면), 라인 31 는 모든 남아있는 원소들을 왼쪽 queue 로부터 오른쪽 queue 로 옮기고, 라인 32 은 왼쪽 queue 를 초기화 합니다. 어떻게 되었든, 라인 34 은 왼쪽 락을 해제합니다.

Quick Quiz 6.7: Listing 6.3 의 라인 28 에서의 오른쪽 dequeue 재시도가 왜 필요하죠?



Quick Quiz 6.8: 분명 왼쪽 락은 가끔은 획득 가능할 거예요!!! 그런데도 왜 Listing 6.3 의 라인 25 에서는 무조건적으로 오른쪽 락을 해제해야 하는 거죠?



pdeq_push_l() 구현이 Listing 6.3 의 라인 40-45 에 보여져 있습니다. 라인 42 은 왼쪽 스팬락을 획득하고, 라인 43 은 이 원소를 왼쪽 큐에 왼쪽으로 집어넣고, 마지막으로 라인 44 은 이 락을 해제합니다. pdeq_push_r() 구현은 (라인 47-52 에 보여져 있습니다) 상당히 비슷합니다.

Quick Quiz 6.9: 하지만 데이터가 한 방향으로만 흐르는 경우에, Listing 6.3 에 보인 알고리즘은 마지막 원소를 가져가서 아랫단의 double-ended queue 를 비울 때마다 양 끝단이 같은 락을 획득하려 할 겁니다. 이는 이 알고리즘이 양 끝단으로 동시에 액세스를 제공하는 것이 이 queue 가 상당히 큰 수의 원소를 가지고 있을 때조차 실패할 수 있음을 의미하지 않나요?



6.1.2.5 Double-Ended Queue Discussion

이 compound 구현은 Section 6.1.2.3 에서 보인 해쉬 기반 구현에 비해 약간 복잡합니다만, 여전히 간단합니다. 물론, 더 영리한 균형맞추기 방법은 더 복잡할 수 있겠습니다만, 여기 보인 이 간단한 방법은 소프트웨어 대체제에 비해서도 [DCW¹¹], 심지어 하드웨어의 도움을 받는 알고리즘에 비해서도 [DLM¹⁰] 성능이 좋은 것으로 드러났습니다. 그러나, 이런 방법에 있어 우리가 바랄 수 있는 최선은 두배의 확장성으로, 최대 두개의 쓰레드가 dequeue 의 락을 동시에 잡고 있을 수 있기 때문입니다. 이 한계는 또한 Michael 의 compare-and-swap 기반의 것과 같은 [Mic03] non-blocking 동기화 기반의 알고리즘에도 적용됩니다.⁵

⁵ 이 논문은 특수한 double-compare-and-swap (DCAS) 인스ตร리션의 double-ended queue 의 lock-free 구현을 만드는 데 필요하지 않음을 보였다는 점에서 흥미롭습니다. 대신, 흔한 compare-and-swap (예: x86 cmpxchg) 으로도 충분합니다.

Quick Quiz 6.10: Double-ended queue 문제에는 왜 한개가 아니라 두개의 해결책이 있는 거죠?

사실, Dice 등에 의해 이야기된 것처럼 [DLM⁺10], 동기화 되지 않은 싱글쓰레드 기반 double-ended queue는 그들이 연구한 어떤 병렬 구현들보다도 상당히 성능이 좋습니다. 따라서, 핵심은 공유된 queue에 enqueue, dequeue 하는 데에는 그 구현과 관계 없이 상당한 오버헤드가 있을 수 있다는 것입니다. 이는 Chapter 3에서 이야기한 것들을 놓고 생각해 보면, 이 queue의 엄격한 first-in-first-out (FIFO) 특성을 놓고 볼 때, 전혀 놀랍지 않을 겁니다.

더 나아가서, 이 엄격한 FIFO queue는 사실 사용자에게 보이는 *linearization point* [HW90]⁶에 대해서만 엄격한 FIFO이며 이 예제에서는 이 linearization point가 각 크리티컬 섹션 내에 여럿 있습니다. 이 queue들은 (예를 들어) 각 개별 오퍼레이션이 시작된 시점에 대해서는 엄격한 FIFO가 아닙니다 [HKLP12]. 이는 동시성 프로그램에서는 엄격한 FIFO 특성이 전혀 가치 있지 않음을 의미하며, 실제로 Kirsch 등은 향상된 성능과 확장성을 제공하는 덜 엄격한 queue를 선보였습니다 [KLP12].⁷ 그렇게 이야기 했지만, 여러분이 여러분의 동시성 프로그램이 사용하는 모든 데이터를 하나의 queue로 보낸다면, 여러분은 여러분의 전체 설계를 다시 생각해 봐야 합니다.

6.1.3 Partitioning Example Discussion

Section 6.1.1의 Quick Quiz의 답에서 주어진 dining philosophers 문제의 최적의 해결책은 “수평적 병렬성” 또는 “데이터 병렬성”의 훌륭한 예입니다. 이 경우의 동기화 오버헤드는 거의 (또는 심지어 정확히) 없습니다. 반대로, double-ended queue 구현은 한 쓰레드에서 다른 쓰레드로 데이터가 움직인다는 점에서 “수직적 병렬성” 또는 “파이프라이닝”的 예입니다. 파이프라이닝을 위해 더 타이트한 협력이 필요해질수록 주어진 수준의 효율성을 얻기 위해 더 많은 단위의 일을 필요로 합니다.

Quick Quiz 6.11: 연결 기반 double-ended queue는 해쉬 기반 double-ended queue 보다 두배나 빠르게 동작합니다, 제가 이 해쉬 테이블의 크기를 미친듯이 크게 늘려도 말이죠. 왜 그런거죠?

⁶ 짧게 요약해서, linearization point는 어떤 함수의 내에서 이 함수가 영향을 만들어 냈다고 말할 수 있는 하나의 지점입니다. 이 각 기반의 구현에서, linearization point는 이 크리티컬 섹션 내의 모든 곳이라고 할 수 있습니다.

⁷ Nir Shavit은 대략 같은 이유로 완화된 스택을 만들었습니다 [Sha11]. 이 상황은 어떤 사람들을 linearization point는 이론가들에겐 유용하지만 개발자들에게 그렇지 않다고 믿게 만들고, 그런 자료 구조와 알고리즘의 설계자들이 얼마나 그들의 사용자들의 실제 필요를 고려했을지 의심하게 만들니다.

Quick Quiz 6.12: Double-ended queue를 위해 동시성을 제어하는 훨씬 나은 방법이 있을까요?

이 두 예는 파티셔닝이 병렬 알고리즘을 고안하는데 있어 얼마나 강력한지 보입니다. Section 6.3.5에서는 세번째 예인 행렬 곱셈을 간단히 보겠습니다. 하지만, 이 세개의 예 모두 병렬 프로그램 분야에서의 더 나은 설계를 필요로 하는데, 이 주제는 뒤 섹션에서 이야기합니다.

6.2 Design Criteria

One pound of learning requires ten pounds of commonsense to apply it.

Persian proverb

최고의 성능과 확장성을 얻는 한가지 방법은 최고의 가능한 병렬 프로그램에 수렴할 때까지 그저 해킹을 계속하는 것입니다. 불행히도, 여러분의 프로그램이 마이크로 레벨로 작은게 아니라면, 가능한 병렬 프로그램의 범위는 너무 커서 우주의 수명 내로 그런 수렴이 이뤄질 거라 보장할 수 없습니다. 그전에, “최고의 가능한 병렬 프로그램”이란 정확히 무엇일까요? 어쨌건, Section 2.2은 성능, 생산성, 그리고 범용성 외에는 병렬 프로그래밍의 목표를 이야기 하지 않았고 최고의 가능한 성능은 생산성과 범용성에 있어서의 비용으로 나타날 것입니다. 우리는 그 프로그램이 쓸모없어지기 전에 받아들여질 수 있을만큼 좋은 병렬 프로그램을 얻을 수 있게끔 하기 위해 설계 시점에 고수준의 선택들을 분명하게 할 수 있어야 합니다.

하지만, 실제로 실세계 설계를 만들기 위해선 더 자세한 설계 기준이 필요한데 이 섹션에서 이를 다룹니다. 실제 세계이므로, 이 기준은 더 또는 덜 충돌하곤 해서, 설계자가 조심스럽게 결과로 나오는 트레이드오프를 조절해야 할 것을 필요로 합니다.

이와 같이, 이 기준은 설계에 동작하는 “법칙”으로 생각될 수 있는데, 이 법칙들 사이에서의 좋은 트레이드오프들이 “디자인 패턴” [Ale79, GHJV95]이라고 불립니다.

이 세개의 병렬 프로그래밍 목표를 이루기 위한 설계 기준은 속도 향상, 경쟁, 일 대비 동기화 비율, 읽기 대비 쓰기 비율, 그리고 복잡도입니다:

속도 향상 (speedup): Section 2.2에서 이야기 되었듯 향상된 성능이 병렬화를 위해 필요한 시간과 문제들을 감수하는 주요 이유입니다. 속도 향상은 어떤 프로그램의 순차적 버전을 수행하는데 걸리는 시

간 대비 병렬 버전을 수행하는데 걸리는 시간의 비율입니다.

경쟁 (contention): 병렬 프로그램에 의해 바쁘게 유지될 수 있는 것보다 그 프로그램에 더 많은 수의 CPU가 사용된다면 이 여분의 CPU들은 경쟁에 의해 유용한 일을 하지 못하게 됩니다. 이는 락 경쟁, 메모리 경쟁, 또는 다른 성능 문제를 일으키는 것이 될 수도 있습니다.

일-대-동기화 비율: 단일 프로세서를 사용하고 싱글쓰레드 기반이며 프리에임션 (preemption) 불가능한, 그리고 인터럽트도 불가능한⁸ 버전의 병렬 프로그램은 어떤 동기화 도구도 필요치 않을 겁니다. 따라서, 이 기능들에 의해 소모되는 모든 시간은 (통신 캐쉬 미스는 물론 메세지 응답 시간, 락킹 도구, 어토믹 인스트럭션, 그리고 메모리 배리어를 포함) 이 프로그램이 하고자 하는 용요한 일에 직접적으로 기여하지 않는 오버헤드입니다. 여기서의 중요한 측정할 것은 동기화 오버헤드와 크리티컬 섹션의 코드의 오버헤드의 관계로, 더 큰 크리티컬 섹션은 더 큰 동기화 오버헤드를 겪을 수 있습니다. 이 일 대비 동기화 비율은 동기화 효율성의 개념에 연관되어 있습니다.

읽기 대비 쓰기 비율: 가끔만 업데이트 되는 데이터 구조는 종종 분할되기보다는 복사되고, 더 나아가 읽기 쓰레드의 동기화 오버헤드를 쓰기 쓰레드의 비용을 늘려 줄이는 비대칭적인 동기화 기능을 사용해 보호함으로써 전체적인 동기화 오버헤드를 줄일 수도 있습니다. 비슷한 최적화들이 Chapter 5에서 논의 되었듯 자주 업데이트 되는 데이터 구조에서도 가능합니다.

복잡도 (complexity): 병렬 프로그램은 똑같은 순차적 프로그램보다 이 병렬 프로그램은 이 순차적 프로그램보다 훨씬 큰 상태 공간을 가지기 때문에 더 복잡합니다. 정규적 구조를 갖는 큰 상태 공간은 가끔은 쉽게 이해될 수 있긴 하지만요. 병렬 프로그래머는 이 더 커다란 상태 공간의 맥락에서 동기화 도구, 메세징, 락킹 설계, 크리티컬 섹션 인식, 그리고 데드락을 고려해야만 합니다.

이 거대한 복잡도는 종종 더 높은 개발과 유지 비용으로 번역됩니다. 따라서, 예산상의 한계는 존재하는 프로그램에 가해질 수 있는 변경의 수와 종류에 제약을 가하게 되는데, 어떤 정도의 속도 향상은 오로지 그만큼의 시간과 문제 만큼만 가치 있기 때문입니다. 더 나쁜게, 추가된 복잡도는 성능과 확장성을 실제로 줄일 수 있습니다.

⁸ 인터럽트 마스킹을 통해서든 그걸 알지 못해서든.

따라서, 어떤 지점을 넘어서면서 부터는, 병렬화 보다 더 비용이 저렴하고 효과적인 잠재적 순차적 최적화가 있을 수도 있습니다. Section 2.2.1에서 이야기 되었듯, 병렬화는 많은 성능 최적화 방버들 중 하나일 뿐이며, 더 나아가 CPU 기반의 병목현상에 대해서만 대부분 적용되는 최적화입니다.

이 표준들은 최대 성능향상을 강제하기 위해 함께 사용될 겁니다. 앞의 세개의 표준은 깊이 상호 관계되어 있으며, 따라서 이 섹션의 나머지 부분은 이 상호관계를 분석해 봅니다.⁹

이 표준들은 요구사항 명세서의 한 부분으로 나타나게 될 수도 있음을 알아두시기 바랍니다. 예를 들어, 속도 향상은 상대적으로 원하는 것으로 동작하거나 (“더 빠를수록 더 좋음”) 해당 워크로드의 절대적 요구사항이 될 수도 (“이 시스템은 최소 초당 1,000,000 웹 히트를 지원해야만 한다”) 있습니다. 전통적 디자인 패턴 언어들은 상대적으로 원하는 것을 힘 (force) 이라 이야기하고 절대적 요구사항은 맥락 (context) 이라고 이야기합니다.

이 설계 표준간의 관계에 대한 이해는 병렬 프로그램을 위한 설계상 적절한 트레이드오프를 찾는데 도움이 될 겁니다.

1. 프로그램이 배타적 락 크리티컬 섹션에서 더 적은 시간을 보낼수록 잠재적 속도 향상은 커집니다. 이는 한번에 하나의 CPU 만이 해당 배타적 락 크리티컬 섹션을 수행할 수 있으므로, 암달의 법칙 [Amd67] 의 결론입니다.

더 구체적으로 보면, 바운드 되지 않은 선형 확장성을 위해선, 프로그램이 배타적 크리티컬 섹션에서 보내는 시간은 CPU 의 수가 늘어날수록 감소해야만 합니다. 예를 들어, 프로그램은 가장 제한적인 배타적 락 크리티컬 섹션에서 그 수행 시간의 10분의 1보다 훨씬 적은 시간만을 보내지 않는다면 10개의 CPU 까지 확장되지 못할 겁니다.

2. 경쟁 효과는 실제 속도 향상이 사용 가능한 CPU 의 수보다 적을 때 상당한 CPU 와/또는 절대적 시간을 소모합니다. CPU 의 수와 실제 속도 향상 사이의 차이가 커질수록, 이 CPU 들은 덜 효율적으로 사용될 겁니다. 비슷하게, 필요한 효율성이 커질수록, 얻을 수 있는 속도 향상은 더 작아질 겁니다.

3. 사용 가능한 동기화 기능들이 그것들이 지키는 크리티컬 섹션에 비해 높은 오버헤드를 갖는다면 속도 향상을 개선하는 최선의 방법은 이 동기화 기

⁹ 실제 세계의 병렬 시스템은 데이터 구조 레이아웃, 메모리 크기, 메모리 계층 응답시간, 대역폭 한계, 그리고 I/O 문제 등의 많은 추가적인 설계 표준을 갖게 될겁니다.

능들이 사용되는 횟수를 줄이는 겁니다. 이는 크리티컬 섹션들을 몰아서 처리하는 것, 데이터 소유권 (Chapter 8 을 참고하세요) 의 사용, 비대칭 기능의 사용 (Section 9 을 참고하세요), 또는 코드 락킹과 같은 거친 규모의 설계를 사용하는 것으로 할 수 있습니다.

4. 이 크리티컬 섹션이 그걸 지키는 기능들에 비해 높은 오버헤드를 갖는다면, 속도 향상을 개선하는 최고의 방법은 reader/writer 락킹, 데이터 락킹, 비대칭, 또는 데이터 소유권을 사용해 병렬성을 증가시키는 것입니다.
5. 이 크리티컬 섹션이 그걸 지키는 기능들에 비해 높은 오버헤드를 갖고 이 지켜지는 데이터 구조가 수정되기보다 읽히는 경우가 훨씬 많다면, 병렬성을 증가시키는 최고의 방법은 reader/writer 락킹 또는 비대칭 기능을 사용하는 겁니다.
6. SMP 성능을 개선하는 많은 변화들, 예를 들어 락 경쟁을 줄이는 것은 또한 리얼타임 반응시간을 개선합니다 [McK05c].

Quick Quiz 6.13: 이 모든 크리티컬 섹션에 대한 문제들은 우리가 단순히 크리티컬 섹션을 갖지 않는 non-blocking 동기화 [Her90] 를 항상 사용해야 함을 의미하지 않나요?

경쟁 (contention) 은 많은 모습을 가지고 있음을 다시 이야기할 가치가 있는데, 락 경쟁, 메모리 경쟁, 캐시 오버플로우, 열 제어, 그 외에도 많은 것들이 포함됩니다. 이 챕터는 주로 락과 메모리 경쟁에 대해서만 봅니다.

6.3 Synchronization Granularity

Doing little things well is a step toward doing big things better.

— Harry F. Banks

Figure 6.10 는 동기화 granularity 의 다른 단계들에 대한 그림을 보이는데, 각각이 다음 섹션들에서 설명됩니다. 이 섹션들은 기본적으로 락킹에 초점을 맞추고 있습니다만, 비슷한 granularity 이슈가 모든 형태의 동기화에 존재합니다.

6.3.1 Sequential Program

프로그램이 단일 프로세서에서 충분히 빠르게 돌아간다면, 그리고 다른 프로세스, 쓰레드, 또는 인터럽트 핸

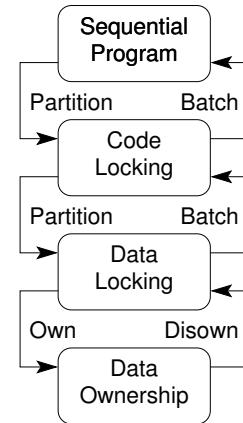


Figure 6.10: Design Patterns and Lock Granularity

들러와 상호작용을 하지 않는다면, 여러분은 동기화 도구들을 없애고 그것들의 오버헤드와 복잡도를 아껴야 합니다. 몇년 전, 무어의 법칙 이 최종적으로는 모든 프로그램을 이 카테고리로 강제로 위치시킬거라는 주장이 있었습니다. 하지만 Figure 6.11 에서 보이듯이, 싱글쓰레드 성능의 폭발적 증가는 2003년 즈음에 멈췄습니다. 따라서, 증가하는 성능은 지속적으로 병렬성을 필요로 할 겁니다.¹⁰ 2006년에 Paul 은 이 문장의 첫 번째 버전을 듀얼코어 랩탑에서 타이핑 했다는 것을 생각해 보면, 그리고 2020년에 추가된 많은 그래프들이 소켓당 56개의 하드웨어 쓰레드를 가진 시스템에서 만들어졌다는 것을 생각해보면, 병렬성은 훌륭하고 정말 존재합니다. 또한 이더넷 대역폭이 Figure 6.12 에 보인 바와 같이 지속적으로 성장하고 있음을 이야기 하는게 중요합니다. 이 성장은 통신 로드를 제어하기 위해 멀티쓰레드 서버 개발을 계속하는 모티베이션이 될 겁니다.

이는 여러분이 모든 프로그램을 멀티 쓰레드 방식으로 코딩해야 함을 의미하지 않음을 알아두시기 바랍니다. 다시 말하지만, 프로그램이 싱글 프로세서에서도 충분히 빨리 돌아간다면 SMP 동기화 기능들의 오버헤드와 복잡도를 아끼십시오. Listing 6.4 의 해시 테이블 탐색 코드의 단순도는 이 점을 잘 보입니다.¹¹ 핵심은 병렬성으로 인한 속도 향상은 일반적으로 CPU 수에 의해 제한된다는 것입니다. 반면, 예를 들자면 조심스러운

¹⁰ 이 그림은 이론상 클라우드 하나 이상의 인스트럭션들을 끝낼 수 있는 최신의 CPU 들에 대해서는 클락 주파수를 보이고, 가장 간단한 인스트럭션 하나를 수행하는데에도 여러 클락을 필요로 하는 오래된 CPU 들에 대해서는 MIPS 를 보이고 있습니다. 이런 방법을 취하는 이유는 최신의 CPU 들의 클라우드 여러 인스트럭션을 끝낼 수 있는 능력은 일반적으로 메모리 시스템 성능에 의해 제한되기 때문입니다.

¹¹ 이 섹션의 예제는 Hart 등의 것 [HMB06] 에서 가져왔고, 여러 파일에서 관련된 코드를 명쾌함을 위해 가져옴으로써 조금 수정되었습니다.

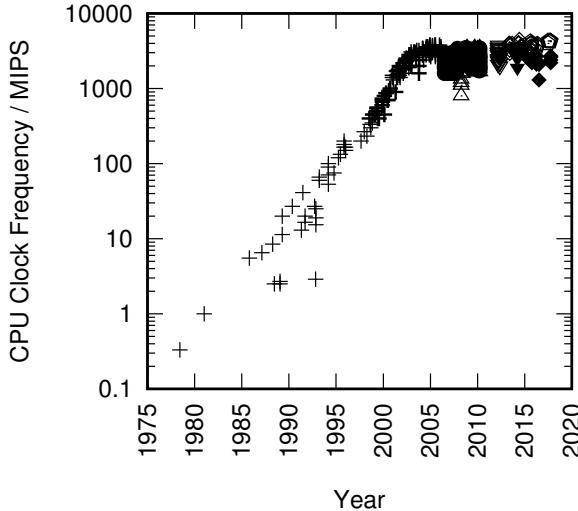


Figure 6.11: MIPS/Clock-Frequency Trend for Intel CPUs

데이터 구조의 선택과 같은 순차적 최적화로 인한 속도 향상은 얼마든지 할 수 있습니다.

달리 말하면, 여러분이 이런 행복한 상황에 처해 있지 않다면, 계속 읽으세요!

Listing 6.4: Sequential-Program Hash Table Search

```

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

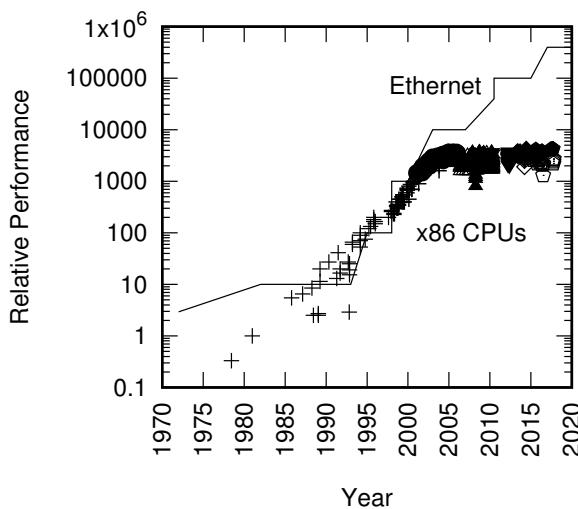


Figure 6.12: Ethernet Bandwidth vs. Intel x86 CPU Performance

6.3.2 Code Locking

코드 락킹은 그것이 글로벌 락을 사용한다는 사실 때문에 무척 간단합니다.¹² 이 기법은 특히 존재하는 프로그램을 멀티프로세서에서 돌아갈 수 있게끔 코드 락킹을 사용하도록 개량하기가 쉽습니다. 만약 이 프로그램이 하나의 공유 리소스만 가지고 있다면, 코드 락킹은 최적의 성능을 제공할 수도 있습니다. 하지만, 많은 더 크고 복잡한 프로그램들이 많은 수행들이 크리티컬 섹션에서 일어날 것을 요구하며, 이는 결국 코드 락킹이 그 확장성을 제한하게끔 만듭니다.

따라서, 여러분은 크리티컬 섹션 또는 약간의 확장만이 필요한 곳에서 수행 시간의 작은 부분만을 소모하는 프로그램에서 코드 락킹을 사용해야 합니다. 이런 경우에, 코드 락킹은 Listing 6.5에서 보이는 것과 같이 그 순차적 버전과 매우 비슷한 비교적 간단한 프로그램을 제공할 겁니다. 하지만, Listing 6.4의 `hash_search()`에서의 간단한 비교 결과 리턴은 이 리턴 전에 락을 해제해야 하므로 세개의 선언문이 되었음을 알아 두시기 바랍니다.

불행히도, 코드 락킹은 특히 여러 CPU들이 동시에 락을 획득하려 할 때 벌어지는 “락 경쟁”에 취약합니다.

¹² 여러분이 그대신 데이터 구조들을 위해 여러 락을 사용한다면, 또는 Java의 경우 동기화된 인스턴스들을 가지고 클래스들을 사용한다면, 여러분은 Section 6.3.3에서 이야기 되는 “데이터 락킹”을 사용하고 있는 겁니다.

Listing 6.5: Code-Locking Hash Table Search

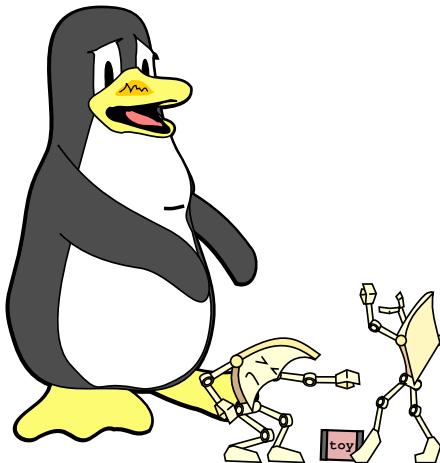
```

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 }

```

다. 어린 아이들의 그룹을 (또는 아이처럼 행동하는 노인들의 그룹을) 관리해야 하는 SMP 프로그래머들은 Figure 6.13.에 그려진 것과 같은, 그 중 하나만을 가질 때의 위험을 알고 있을 겁니다.

이 문제에 대한 한가지 해결책, 이름하여 “data locking” 이 다음 섹션에서 설명됩니다.

**Figure 6.13:** Lock Contention**Listing 6.6:** Data-Locking Hash Table Search

```

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 }

```

6.3.3 Data Locking

많은 데이터 구조가 쪼개지고 이 데이터 구조의 각 조각이 각자의 락을 가질 수 있습니다. 그러면 각 데이터 구조의 조각을 위한 크리티컬 섹션은 병렬로 수행될 수 있습니다, 각 조각을 위한 크리티컬 섹션은 한번에 하나씩만 수행될 수 있긴 하지만요. 경쟁이 줄어야 할 때, 그리고 동기화 오버헤드가 속도 향상을 제한하지 않을 때, 여러분은 데이터 락킹을 사용해야 합니다. 데이터 락킹은 지나치게 큰 크리티컬 섹션을 여러 데이터 구조로 분산시켜서 경쟁을 줄이는데, 예를 들어 Listing 6.6에 보인 것과 같이 해시테이블에 해시 버킷별 크리티컬 섹션을 유지하는 식입니다. 그렇게 증가된 확장성은 역시 추가된 데이터 구조, `struct bucket`의 형태로 복잡도를 약간 높입니다.

Figure 6.13에 보인 것과 같은 경쟁적인 상황과 반대로, 데이터 락킹은 Figure 6.14에 보인 것과 같이 하모니를 조장하며 병렬 프로그램에서 이는 거의 항상 향상된 성능과 확장성으로 귀결됩니다. 이런 이유로, 데이터 락킹은 Sequent에 의해 그 커널에서 무척 많이 사용되었습니다 [BK85, Inm85, Gar90, Dov90, MD92, MG92, MS93].

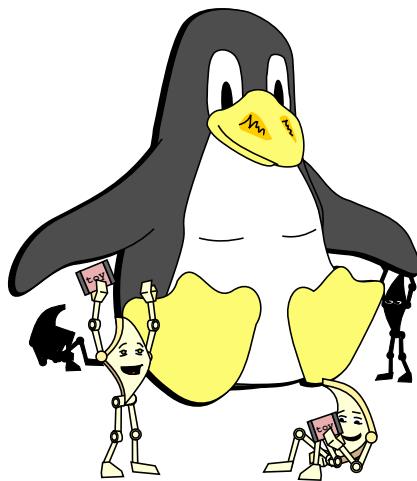


Figure 6.14: Data Locking

하지만, 어린 아이들을 맡아본 사람들은 또한번 증언할 수 있겠지만, 가지고 놀기 충분한 것을 주는 것만으로는 평안을 보장할 수 없습니다. 비슷한 상황이 SMP 프로그램에서도 일어날 수 있습니다. 예를 들어, 리눅스 커널은 파일과 디렉토리들의 캐시 (“dcache” 라 불립니다)를 유지합니다. 이 캐시의 각 항목은 각자의 락을 가지고 있습니다만, 루트 디렉토리와 그 직접적 후손에 연관된 항목들은 다른 것들에 비해 훨씬 더 자주 접근될 가능성이 높습니다. 이는 많은 CPU들이 이런 인기 있는 항목들의 락에 대해 경쟁하게 되는 상황을 초래하고, 이는 Figure 6.15에 보인 것과 다르지 않은 상황을 초래할 수 있습니다.

많은 경우, 알고리즘은 데이터 편중을 줄이도록 설계될 수 있고, 어떤 경우에는 이를 완전히 제거할 수도 있습니다 (예를 들면, 리눅스 커널의 dcache에서처럼요 [MSS04, Cor10a, Bro15a, Bro15b, Bro15c]). 데이터 락킹은 해쉬 테이블과 같은 쪼개질 수 있는 데이터 구조에서, 그리고 여러 항목이 각각 주어진 데이터 구조의 인스턴스를 나타내는 상황에서도 종종 사용됩니다. 리눅스 커널의 task 리스트는 뒤의 것의 한 예로, 각 task 구조체는 각자의 `alloc_lock`과 `pi_lock`을 갖습니다.

동적으로 할당되는 구조체들에 대한 데이터 락킹에서의 주요 과제는 이 구조체가 이 락이 잡혀 있는 동안 존재를 유지함을 보장하는 것입니다 [GKAS99]. Listing 6.6의 코드는 이 과제를 락들을 정적으로 할당된, 따라서 결코 해제되지 않는 해시 버킷에 위치시킴으로써 해결했습니다. 하지만, 이 트릭은 해시 테이블의 크기가 변할 수 있다면, 따라서 락이 이제 동적으로 할당되어야 한다면 동작하지 않을 겁니다. 이 경우, 해시 버킷은 그 락이 획득되어 있는 동안은 해제되는 것을 방지할 방법이 필요할 겁니다.

Quick Quiz 6.14: 어떤 구조체를 그것의 락이 획득되어 있는 동안 해제되지 않게 하는 방법들로는 어떤 게 있나요?

■

6.3.4 Data Ownership

데이터 소유권 (data ownership)은 주어진 데이터 구조를 쓰레드 또는 CPU들에게 분할하여, 각 Thread/CPU가 각자의 데이터 구조 부분을 아무런 동기화 오버헤드 없이 액세스 할 수 있게 합니다. 하지만, 한 쓰레드가 어떤 다른 쓰레드의 데이터에 액세스하고자 한다면, 이 첫번째 쓰레드는 직접 그럴 수는 없습니다. 그 대신, 그 첫번째 쓰레드는 이 두번째 쓰레드와 통신을 해서 이 두번째 쓰레드가 그 첫번째 쓰레드 대신 이 오퍼레이션을 수행하거나, 그 데이터를 첫번째 쓰레드에게 넘겨줘야만 합니다.

데이터 소유권은 애매해 보일 수도 있겠습니다만, 매우 자주 사용됩니다:

1. 한 CPU 또는 쓰레드에 의해서만 액세스 가능한 모든 변수는 (C 와 C++에서의 `auto` 변수 같은) 해당 CPU 또는 쓰레드에게 소유됩니다.
2. 유저 인터페이스의 한 인스턴스는 연관된 사용자의 컨텍스트를 소유합니다. 병렬 데이터베이스 엔진과 상호작용하는 어플리케이션들이 완전한 순차적 프로그램인 것마냥 작성되는 것은 매우 흔한 일입니다. 그런 어플리케이션은 유저 인터페이스와 사용자의 현재 동작을 소유합니다. 따라서 명시적 병렬성은 데이터베이스 엔진 스스로에게 국한됩니다.

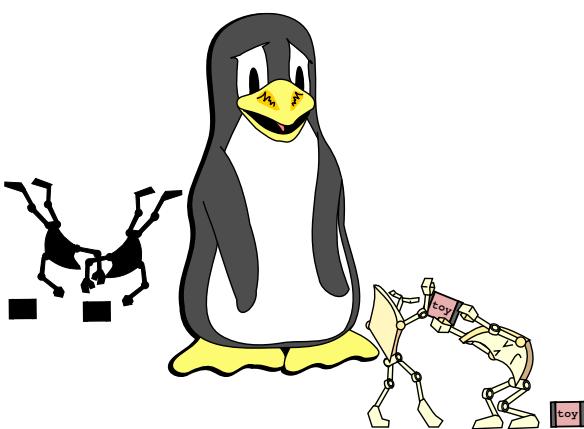


Figure 6.15: Data Locking and Skew

3. 패러미터를 가질 수 있는 시뮬레이션들은 각 쓰레드가 패러미터 공간의 특정 영역에 대한 소유권을 각 쓰레드가 갖게 함으로써 종종 간단히 병렬화 됩니다. 이런 종류의 문제를 위해 설계된 컴퓨팅 프레임워크도 있습니다 [Uni08a].

상당한 데이터 공유가 있다면, 이 쓰레드 또는 CPU 들 간의 통신은 상당한 복잡도와 오버헤드를 초래할 수 있습니다. 더 나아가, 가장 많이 사용되는 데이터가 단일 CPU에게 소유된다면, 해당 CPU는 “핫스팟”이 될 것이고, Figure 6.15에서 보인 것과 같은 결과에 이르게 할 수 있습니다. 하지만, 공유가 필요치 않은 상황에서는 데이터 소유권이 이상적인 성능을 이루며, Listing 6.4에 보인 것처럼 순차적 프로그램만큼이나 코드도 단순해집니다. 그런 상황은 종종 “당황스러울 만큼 병렬적”이라고 이야기 되며, 가장 좋은 상황에서는 앞서 Figure 6.14에서 보인 것과 같은 상황을 가능하게 합니다.

데이터 소유권의 또 다른 중요한 예는 데이터 읽기 전용일 때 일어나는데, 모든 쓰레드가 복사를 통해 그것을 “소유” 합니다.

데이터 소유권은 Chapter 8에서 더 자세히 다뤄집니다.

6.3.5 Locking Granularity and Performance

이 섹션은 수학적 동기화 효율성 관점에서 락킹 세밀도와 성능을 바라봅니다. 수학에 관심이 없는 독자 여러분은 이 섹션을 건너뛰실 수도 있겠습니다.

이는 M/M/1 큐(queue)에 기반해 하나의 공유 전역 변수에 대해 동작하는 동기화 메커니즘의 효율성을 위한 거친 큐잉(queueing) 모델을 사용합니다. M/M/1 큐잉 모델은 지수적으로 분산된 “inter-arrival rate” λ 와 지수적으로 분산된 “service rate” μ 에 기반합니다. 이 inter-arrival rate λ 는 이 시스템이 동기화로부터 자유로웠더라면 초당 수행했을 동기화 오퍼레이션의 평균 횟수로 생각될 수 있는데, 달리 말하면 λ 는 각 동기화가 아닌 일 단위의 오버헤드의 반대 오버헤드입니다. 예를 들어, 만약 각 일의 단위가 트랜잭션이라면, 그리고 각 트랜잭션이 수행되는데 1 밀리세컨드를 소요한다면, 동기화 오버헤드를 제외하고, λ 는 초당 1,000 트랜잭션이 됩니다.

Service rate μ 는 비슷하게 정의됩니다만, 각 트랜잭션 오버헤드가 0일 때, 그리고 CPU들이 서로의 동기화 오퍼레이션의 완료를 기다려야 한다는 사실을 무시할 때 이 시스템이 처리할 초당 동기화 오퍼레이션의 평균 수를 의미하는데, 달리 말하자면 μ 는 경쟁이 없을 때의 동기화 오버헤드라고 대략 생각될 수 있습니다. 예를 들어, 각 트랜잭션의 동기화 오퍼레이션이 하나의 어토믹 값 증가 인스트럭션을 사용하며, 이 컴퓨터 시스템은 각

CPU에서 5 나노세컨드마다 지역변수의 어토믹 값 증가가 가능하다고 해 봅시다 (Figure 5.1를 참고하세요).¹³ 따라서 μ 의 값은 초당 200,000,000 어토믹 값 증가가 됩니다.

물론, λ 의 값은 공유 변수의 값을 증가하는 CPU의 수가 늘어날수록 증가되는데, 각 CPU는 개별적으로 트랜잭션을 처리할 수 있기 때문입니다 (다시 말하지만, 동기화를 무시한 겁니다):

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

여기서 n 은 CPU의 수이고 λ_0 는 단일 CPU의 트랜잭션 처리 능력입니다. 경쟁이 없을 때 단일 CPU가 단일 트랜잭션을 처리하는데 걸릴 것으로 예상되는 시간은 $1/\lambda_0$ 임을 알아 두시기 바랍니다.

CPU들은 서로가 이 단일 공유 변수의 값을 증가할 기회를 위해 “줄을 서서 기다려야” 하기 때문에, 예상되는 전체 대기 시간을 위해 M/M/1 큐잉 모델 표현을 사용할 수 있습니다:

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

앞의 λ 값을 대입해 보면:

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

이제, 효율성은 동기화가 있을 때 트랜잭션 하나를 처리하는데 드는 시간 ($T + 1/\lambda_0$) 대비 동기화가 없을 때의 그것의 ($1/\lambda_0$) 비율입니다:

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

앞의 T 값을 대입하고 간략화 하면:

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

하지만 μ/λ_0 의 값은 동기화 오버헤드 자체 (컨텐션 부재시) 대비 트래잭션 처리에 걸리는 시간 (동기화 오버헤드 부재시)의 비율입니다. 우리가 이 비율을 f 라고 부르면:

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

¹³ 물론, 같은 공유 변수를 값 증가시키는 8개의 CPU가 존재한다면, 각 CPU는 자신의 값 증가를 위한 5 나노세컨드를 사용하기 전에 다른 CPU들이 모두 값 증가를 할 수 있도록 최소 35 나노세컨드를 기다려야 합니다. 실제로, 그 기다림은 이 변수를 한 CPU에서 다른 CPU로 옮겨야 하는 필요성 때문에 더 길어질 겁니다.

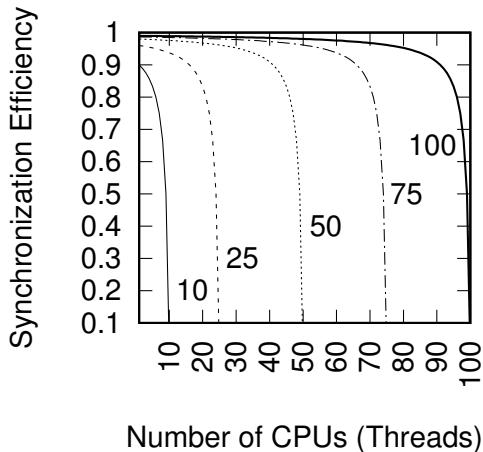


Figure 6.16: Synchronization Efficiency

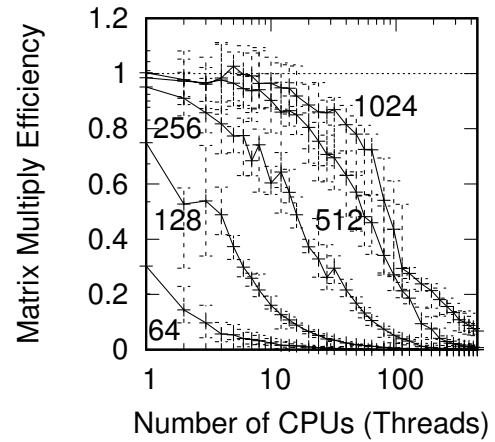


Figure 6.17: Matrix Multiply Efficiency

Figure 6.16는 동기화 효율성 e 를 오버헤드 비율 f 의 일부 값에 대한 CPU/쓰레드 갯수 n 의 함수로 그려냅니다. 예를 들어, 5 나노세컨드 어토미크 값 증가 연산을 다시 생각해 보면, $f = 10$ 선은 매 50 나노세컨드마다 각 CPU가 어토미크 값 증가를 하는 상황에 연관되며, $f = 100$ 선은 각 CPU가 매 500 나노세컨드마다 어토미크 값 증가를 하는 상황에 연관되어서, 결국은 수백개의 (어쩌면 수천개의) 인스트럭션들에 연관되게 됩니다. 각 선이 증가하는 CPU 또는 쓰레드 갯수에 의해 빠르게 떨어지는 것을 보면, 우린 단일 전역 공유 변수에 대한 어토미크 값 조정에 기반한 동기화 메커니즘은 현재의 상용 하드웨어 위에서 많이 사용될 경우 잘 확장되지 못할 것이라고 결론내릴 수 있습니다. 이는 Chapter 5에서 이야기 했던 병렬 카운팅 알고리즘을 이끌게 하는 원동력에 대한 대략적 수학적 묘사입니다. 여러분의 실제 세계는 다를 수 있습니다.

그러나, 효율성의 개념은 유용한데, 동기화가 적거나 정형적이지 않을 때 조차도 그렇습니다. 예를 들어 한 행렬의 행들이 다른 행렬의 열들과 곱해져 (“dot 연산을 통해”) 세번째 행렬의 각 항을 만들게 되는 행렬 곱셈을 생각해 봅시다. 이 오퍼레이션들은 서로 충돌하지 않으므로, 첫번째 행렬의 행들을 쓰레드들로 분리시키고, 각 쓰레드는 결과 행렬의 각 행을 계산하게 하는 것이 가능합니다. 따라서 이 쓰레드들은 동기화 오버헤드 걱정 없이 완전히 개별적으로 `matmul.c`처럼 동작할 수 있습니다. 따라서 어떤 사람들은 완전한 효율성인 1.0을 생각할 겁니다.

하지만, Figure 6.17는 다른 이야기를 하는데, 특히 64-by-64 행렬 곱셈에서는 0.3 이상의 효율성을 결코 얻지 못하는데, 단일 쓰레드로 동작할 때조차 그려하며, 더 많

은 쓰레드가 사용되면 효율성이 떨어집니다.¹⁴ 128-by-128 행렬은 좀 낫습니다만, 여전히 추가되는 쓰레드에 맞춰 훨씬 나은 성능을 보이지는 못합니다. 256-by-256 행렬은 제법 잘 확장됩니다만, 약간의 CPU 들까지만 그렇습니다. 512-by-512 행렬 곱셈의 효율성은 10개 쓰레드까지만 1.0 보다 좀 낮게 측정되며, 심지어 1024-by-1024 행렬 곱셈도 수십개의 쓰레드가 사용되었을 때에는 확연히 흔들립니다. 그러나, 이 그림은 batching의 성능과 확장성 이익을 확연히 보이고 있습니다: 여러분이 동기화 오버헤드를 일으켜야만 한다면, 여러분의 돈의 가치를 얻을 겁니다.

Quick Quiz 6.15: 어떻게 싱글쓰레드 기반 64-by-64 행렬 곱셈이 1.0 보다 낮은 효율성을 가질 수 있죠? Figure 6.17의 모든 선들은 하나의 쓰레드만 사용했을 때에는 1.0의 효율성을 보여야 하는 거 아닌가요?

■
이 비효율성을 놓고 보면, Section 6.3.3에서 이야기된 데이터 락킹이나 다음 섹션에서 이야기될 병렬-fastpath 접근법과 같은 더 확장 가능한 접근법을 알아보는게 가치있을 겁니다.

Quick Quiz 6.16: 데이터 병렬 기법이 어떻게 행렬 곱셈을 도울 수 있죠? 이건 이미 데이터 병렬이라구요!!!

¹⁴ Figure 6.16에서의 부드러운 선과 대조적인 Figure 6.17의 날은 에러바와 요동치는 선들은 그 실제 세계의 특성에 대한 증거를 보입니다.

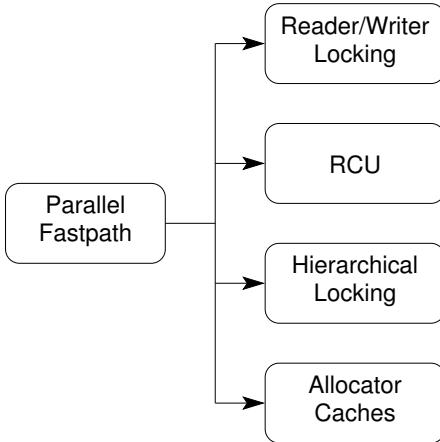


Figure 6.18: Parallel-Fastpath Design Patterns

6.4 Parallel Fastpath

There are two ways of meeting difficulties: You alter the difficulties, or you alter yourself to meet them.

Phyllis Bottome

미세하게 조정된 (그리고 따라서 일반적으로 성능이 높은) 설계들은 보통 거칠게 조정된 설계들에 비해 복잡합니다. 많은 경우, 대부분의 오버헤드는 코드의 작은 부분에서 기인합니다 [Knu73]. 그러나 그 작은 부분에 노력을 기울이지 않을 이유가 무엇일까요?

이것이 병렬-fastpath 설계패턴을 뒷받침하는 아이디어로, 적극적으로 전체 알고리즘을 병렬화 하려면 필요할 복잡성을 피하고 일반적인 경우의 코드 패쓰를 병렬화 하는 것입니다. 여러분은 여러분이 병렬화 하고자 하는 특정 알고리즘만이 아니라, 그 알고리즘이 사용되는 워크로드에 대해서도 이해해야 합니다. 병렬 fastpath를 만드는 데에는 상당한 창의성과 설계 노력이 필요합니다.

병렬 fastpath는 다른 패턴들을 조합하여 (하나는 fastpath 용, 다른 하나는 다른 곳을 위해) 따라서 템플릿 패턴입니다. 다음의 병렬 fastpath의 예들은 Figure 6.18에 그려진 것처럼 각자 하나씩의 전용 패턴을 갖습니다:

1. Reader/Writer Locking (Section 6.4.1에서 설명됩니다).
2. Reader/writer 락킹의 고성능 대체제로 사용될 수도 있는 Read-copy update (RCU)는 Section 9.5에서 설명됩니다. 다른 대안으로는 해저드 포인터 (Section 9.3)와 시퀀스 락킹 (Section 9.4) 등이 있

Listing 6.7: Reader-Writer-Locking Hash Table Search

```

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 }
  
```

습니다. 이런 대안들은 이 챕터에서 더 이야기 되지 않습니다.

3. Section 6.4.2에서 이야기 되는 계층적 락킹 ([McK96a]).
4. 리소스 할당 캐쉬 ([McK96a, MS93]). 더 자세한 내용을 위해선 Section 6.4.3을 보세요.

6.4.1 Reader/Writer Locking

동기화 오버헤드가 크지 않다면 (예를 들어, 프로그램이 큰 크리티컬 섹션과 함께 거친 단위의 병렬성을 사용하고 있다면), 그리고 그 크리티컬 섹션의 작은 부분만이 데이터를 수정한다면, 여러 읽기 쓰레드가 병렬로 수행될 수 있게 하는 것은 확장성을 크게 향상시킬 수 있습니다. 쓰기 쓰레드는 읽기 쓰레드와 서로들을 모두 배제시켜야 합니다. Reader-writer 락킹의 여러 구현이 존재하는데, Section 4.2.4에서 설명된 POSIX 구현이 포함됩니다. Listing 6.7은 해쉬 탐색이 reader-writer 락킹을 이용해 어떻게 구현될 수 있는지 보입니다.

Reader/writer 락킹은 비대칭 락킹의 간단한 한가지 예입니다. Snaman [ST87]은 여러 클러스터 시스템에서 사용된 화려한 여섯개의 비대칭 락킹 설계모드를 설명합니다. 일반적인 락킹과 reader-writer 락킹이 Chapter 7에서 자세하게 설명되어 있습니다.

Listing 6.8: Hierarchical-Locking Hash Table Search

```

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 }

```

6.4.2 Hierarchical Locking

계층적 (hierarchical) 락킹의 아이디어는 어떤 미세 단위 락을 획득할지 선택하는 동안만 거대 단위 락을 갖자는 것입니다. Listing 6.8은 우리의 해쉬 테이블 탐색이 어떻게 계층적 락킹을 적용하도록 될 수 있는지 보입니다만, 또한 이 방법의 큰 약점도 보입니다: 우린 두번째 락을 획득하기 위한 오버헤드를 지불했습니다만, 우린 이를 짧은 시간동안만 쥐고 있습니다. 이 경우, 더욱 간단한 데이터 락킹 접근법은 더 간단할 것이고 성능도 높을 가능성이 큽니다.

Quick Quiz 6.17: 어떤 경우에 계층적 락킹이 잘 동작하나요?

**6.4.3 Resource Allocator Caches**

이 섹션은 병렬 고정 클록 크기 메모리 할당자의 간략화된 설명을 제공합니다. 더 자세한 설명은 리눅스 커널 [Tor03]의 문서에서 [MG92, MS93, BA01, MSK01, Eva11, Ken20] 찾을 수 있습니다.

6.4.3.1 Parallel Resource Allocation Problem

병렬 메모리 할당자가 마주하는 기본적인 문제는 일반적인 경우 극단적으로 빠른 메모리 할당과 해제를 제공해야 한다는 필요성과 선호되지 않는 할당과 해제 패턴 하에서는 메모리를 효율적으로 분산시켜야 한다는 필요성 사이의 긴장입니다.

이 긴장을 자세히 이해하기 위해, 이 문제에 대한 데이터 소유권의 간단한 적용을 생각해 봅시다—단순히 각 CPU가 각자의 것을 갖게끔 메모리를 조각내는 겁니다. 예를 들어, 12개의 CPU 와 64 기가바이트의 메모리를 가진 시스템을 생각해 봅시다, 제가 지금 쓰는 랩탑이죠. 우린 각 CPU에 5 기가바이트 메모리 영역을 할당하고, 각 CPU가 각자의 영역에서 락킹과 복잡도, 오버헤드 없이 할당을 하게끔 할 수 있습니다. 불행히도, 이 방법은 간단한 생산자-소비자 워크로드에서 일어나듯 CPU 0 가 메모리를 할당하기만 하고 CPU 1은 해제를 하기만 할 때 실패합니다.

다른 극단인 코드 락킹은 거대한 락 컨텐션과 오버헤드로 고통받게 됩니다 [MS93].

6.4.3.2 Parallel Fastpath for Resource Allocation

흔히 사용되는 해결책은 각 CPU가 약간의 블록의 캐시를 소유하는 병렬 fastpath를 사용하며 추가적인 블록들을 위해서는 코드 락킹을 사용하는 거대한 공유 메모리 풀을 사용하는 것입니다. 어떤 CPU가 이 메모리 블록을 독점하는 것을 막기 위해, 우린 각 CPU의 캐시에 있을 수 있는 블록의 수에 제한을 겁니다. 두개의 CPU가 있는 시스템에서, 메모리 블록의 흐름은 Figure 6.19에 보인 것과 같을 겁니다: 특정 CPU가 자신의 풀이 꽉 찬 상태에서 한 블록을 해제하려 할 때, 그 블록은 전역 풀로 보내지게 되며, 비슷하게 해당 CPU가 자신의 풀이 비어있는 상태에서 하나의 블록을 할당하려 할 때, 그 블록은 전역 풀에서 얻어집니다.

6.4.3.3 Data Structures

할당자 캐시의 “장난감” 구현을 위한 실제 데이터 구조가 Listing 6.9에 보여져 있습니다. Figure 6.19의 “글로벌 풀”은 struct globalmempool 타입의 globalmem으로, 두개의 CPU 풀은 쓰레드별 변수인 struct perthreadmempool 타입의 perthreadmem 변수로 구현되었습니다. 이 두개의 데이터 구조는 모두 각자의 pool 필드 안에 블록들로의 포인터들의 집합을 가지는데, 인덱스 0부터 채워지는 구조를 갖습니다. 따라서, 만약 globalmem.pool[3] 가 NULL이라면, 이 할당의 인덱스 4부터 끝까지의 남은 것들은 또한 NULL이어야만 합니다. cur 필드는 이 pool 배열의 가장 높은 숫자를 갖는 꽉찬 원소의 인덱스를 갖거나, 모든 원소가

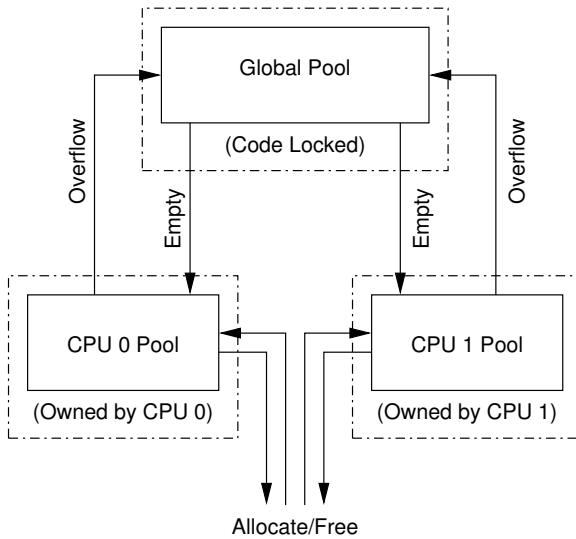


Figure 6.19: Allocator Cache Schematic

Listing 6.9: Allocator-Cache Data Structures

```

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 perthreadmempool {
11     int cur;
12     struct memblock *pool[2 * TARGET_POOL_SIZE];
13 };
14
15 DEFINE_PER_THREAD(struct perthreadmempool, perthreadmem);

```

비어있을 때에는 -1 을 갖습니다. `globalmem.pool[0]` 부터 `globalmem.pool[globalmem.cur]` 까지의 모든 원소는 차 있어야만 하며, 나머지는 모두 비어있어야만 합니다.¹⁵

이 풀 데이터 구조의 동작이 Figure 6.20 위에 그려져 있는데, 여섯개의 상자는 pool 필드를 구성하는 포인터 배열을 나타내며 그들을 앞의 숫자는 cur 필드를 나타냅니다. 그림자로 칠해진 상자는 NULL 이 아닌 포인터를 나타내며, 빈 박스는 NULL 포인터를 나타냅니다. 중요하지만 어쩌면 혼란스러울지도 모르는 이 데이터 구조의 불변의 법칙은 cur 필드가 항상 NULL 이 아닌 포인터의 갯수보다 작다는 것입니다.

¹⁵ 풀 크기 (TARGET_POOL_SIZE 와 GLOBAL_POOL_SIZE) 둘 모두 비현실적으로 작습니다만, 이 작은 크기는 그 동작에 대한 느낌을 갖기 위해 이 프로그램을 한단계 움직이게 하는 것을 쉽게 해줍니다.

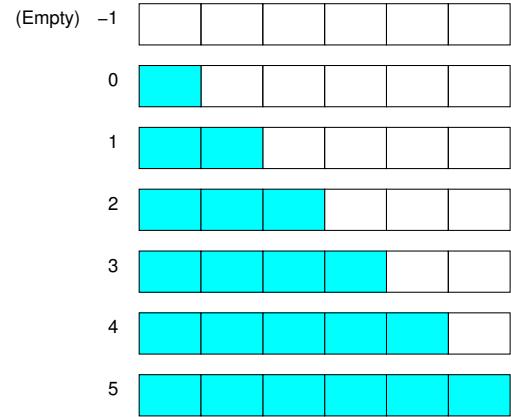


Figure 6.20: Allocator Pool Schematic

6.4.3.4 Allocation Function

Listing 6.10에 할당 함수 `memblock_alloc()`이 보입니다. 라인 7은 현재 쓰레드의 쓰레드별 풀을 집고, 라인 8은 그에 비어 있는지 검사합니다.

만약 그렇다면, 라인 9-16은 글로벌 풀로부터 라인 9에서 얻어지고 라인 16에서 놓아지는 스피너를 하여 글로벌 풀로부터 쓰레드별 풀을 다시 채웁니다. 라인 10-14는 블록들을 글로벌 풀로부터 쓰레드별 풀로 이로컬 풀이 그 목표 크기 (반) 까지 도달하거나 글로벌 풀이 비어버릴 때까지 옮기며 라인 15은 이 쓰레드별 풀의 카운트를 옳은 값으로 설정합니다.

어떤 경우든, 라인 18은 이 쓰레드별 풀이 여전히 비어있는지 검사하고, 그렇지 않다면 라인 19-21은 블록 하나를 제거하고 그것을 리턴합니다. 그렇지 않다면, 라인 23은 메모리가 남아있지 않는 슬픈 이야기를 전합니다.

6.4.3.5 Free Function

Listing 6.11은 메모리 블록 해제 함수를 보입니다. 라인 6은 이 쓰레드의 풀로의 포인터를 가져오고, 라인 7은 이 쓰레드별 풀이 가득 찬는지 검사합니다.

만약 그렇다면, 라인 8-15는 이 쓰레드별 풀의 절반을 글로벌 풀로 라인 8 와 14에서 스피너를 획득하고 해제하면서 넘겨서 비웁니다. 라인 9-12는 블록들을 로컬 풀에서 글로벌 풀로 옮기는 반복문을 구현하며, 라인 13은 이 쓰레드별 풀의 카운트를 올바른 값으로 설정합니다.

어떤 경우든, 라인 16은 새로 해제된 블록을 쓰레드별 풀에 위치시킵니다.

Listing 6.10: Allocator-Cache Allocator Function

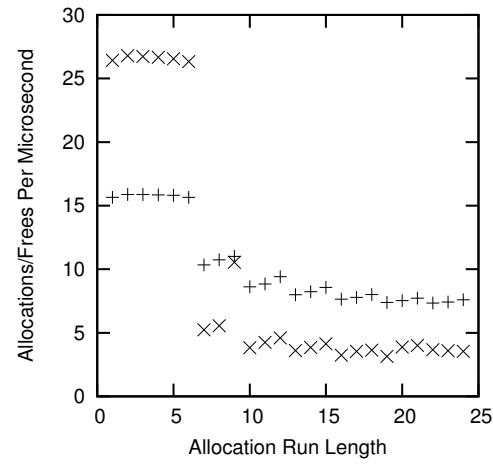
```

1 struct memblock *memblock_alloc(void)
2 {
3     int i;
4     struct memblock *p;
5     struct perthreadmempool *pcpp;
6
7     pcpp = &__get_thread_var(perthreadmem);
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 }
```

Listing 6.11: Allocator-Cache Free Function

```

1 void memblock_free(struct memblock *p)
2 {
3     int i;
4     struct perthreadmempool *pcpp;
5
6     pcpp = &__get_thread_var(perthreadmem);
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 6.21:** Allocator Cache Performance

Quick Quiz 6.18: 이 리소스 할당자 설계는 Section 5.3에서 이야기한 대략적 리미트 카운터의 그것과 닮지 않았나요?



6.4.3.6 Performance

대략적인 성능 결과¹⁶ 가 Figure 6.21에 보여져 있는데, 이 결과는 1 GHz (4300 bogomips per CPU)의 속도로 각 CPU의 캐시에 여섯 블락이 허용된 듀얼코어 인텔 x86에서 수행되었습니다. 이 마이크로 벤치마크에서, 각 쓰레드는 반복적으로 한 그룹의 블락을 할당하고 모든 블록을 해제하며, 이 그룹에 있는 블록의 갯수는 x 축에 표시된 “할당 수행 길이”가 됩니다. Y 축은 마이크로 세컨드당 성공한 할당/해제 쌍의 수를 보입니다—실패한 할당은 세어지지 않습니다. “X”는 두개 쓰레드 수행에서의 것이며, “+”는 단일 쓰레드 수행에서의 것입니다.

수행 길이 여섯까지는 선형적으로 확장되며 대단한 성능을 제공하지만, 여섯을 넘는 수행 길이부터는 낮은 성능을 보이며 거의 항상 음의 확장을 보임을 보시기 바랍니다. 따라서 이는 TARGET_POOL_SIZE를 충분히 크게 잡는게 중요함을 보이는 데, 실제 상황에서는 다행히도 그러기가 상당히 쉬우며 [MSK01], 특히 오늘날의 큰 메모리에선 그렇습니다. 예를 들어, 대부분의 시스템에서 TARGET_POOL_SIZE를 100으로 설정하는 건 상당히 합리적인데, 이는 할당과 해제가 쓰레드별 풀에서 최소한 99%의 시간동안 처리될 것을 보장합니다.

¹⁶ 이 데이터는 통계적으로 의미있는 방식으로 모아지지 않았으며, 따라서 많은 회의적으로 의심 섞인 시선으로 보여져야 합니다. 좋은 데이터 수집과 축소 방법은 Chapter 11에서 이야기 됩니다. 그러나, 반복된 수행이 비슷한 결과를 냈으며, 이 결과들은 비슷한 알고리즘의 보다 주의 깊은 평가 결과와 일치합니다.

그림에서 볼 수 있듯이, 일반적인 경우의 데이터 소유권이 허용되는 상황은 (여섯까지의 수행 길이) 락이 획득되어야만 하는 경우에 비해 상당히 향상된 성능을 제공합니다. 일반적인 경우에 동기화를 피하는 것은 이 책의 반복적인 테마가 될 겁니다.

Quick Quiz 6.19: Figure 6.21에서, 수행 길이를 증가시킴에 따라 성능이 올라감에는 세개의 샘플 그룹에 의하는 패턴이 있는데 예를 들면 수행 길이 10, 11, 12가 그렇습니다. 왜죠?

Quick Quiz 6.20: 할당 실패는 두개 쓰레드 테스트에서 수행 길이가 19 이상일 경우에 관측되었습니다. 글로벌 풀의 크기가 40이고 쓰레드별 타겟 풀 크기 s 가 셋이고, 쓰레드 수 n 이 2 이며, 이 쓰레드별 풀이 초기에는 어떤 메모리도 사용되지 않고 있으므로 비어있음을 놓고 생각해 보면, 실패가 일어날 수 있는 최소한의 할당 수행 길이 m 은 무엇일까요? (각 쓰레드는 반복적으로 m 블록의 메모리를 할당하고, 이어서 m 블록의 메모리를 해제함을 기억하시기 바랍니다.) 달리 생각하면, n 쓰레드가 풀 크기 s 를 가지고 있으며 각 쓰레드가 반복적으로 m 블록의 메모리를 할당받고 이어서 m 블록의 메모리를 해제함을 고려할 때, 글로벌 풀의 크기는 얼마나 커야 할까요? *Note:* 정확한 답을 얻기 위해선 여러분이 `smpalloc.c` 소스 코드를 들여다 보고 그걸 한 단계씩 수행해 보기도 할 것을 필요로 할 겁니다. 경고 했어요!

6.4.3.7 Real-World Design

이 장난감 병렬 리소스 할당자는 상당히 간단했습니다만, 실제 세계의 설계는 이 방법에서 여러가지 방향으로 확장될 수 있습니다.

첫째로, 실제 세계의 할당자는 이 장난감 예에서의 한가지 크기와 달리 넓은 범위의 할당 크기를 다뤄야 합니다. 이를 위한 한가지 인기있는 방법은 1980년대 후반 BSD 메모리 할당자 [MK88]에서처럼 내부 파편화와 외부 파편화 사이의 밸런스를 잡을 수 있게끔 고정된 크기들의 집합을 제공하는 것입니다. 이렇게 하는 것은 “globalmem” 변수가 크기별로 복사될 것을, 그리고 연관된 락 역시 비슷하게 복사될 것을 필요로 해서 이 장난감 프로그램의 코드 락킹이 아닌 데이터 락킹으로 귀결될 것을 의미합니다.

둘째로, 제품 품질 시스템은 메모리를 재활용 할 수 있어야 하는데, 블락들을 페이지와 같은 더 커대한 구조체로 합칠 수 있어야만 합니다 [MS93]. 이 합치기는 또한 락으로 보호되어야 하는데, 이것 역시 사이즈별로 복사될 수 있을 겁니다.

셋째로, 합쳐진 메모리는 아랫단의 메모리 시스템으로 반환되어야만 하며, 메모리의 페이지를 역시 아랫

Table 6.1: Schematic of Real-World Parallel Allocator

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

단의 메모리 시스템으로부터 할당되어야만 합니다. 이 단계에서 필요한 락킹은 아랫단 메모리 시스템에 의존되지만, 코드 락킹이 될수도 있습니다. 코드 락킹은 이 단계에서 종종 허용되곤하는데, 이 단계는 잘 설계된 시스템에서는 가끔씩만 사용되기 때문입니다 [MSK01].

이 실제 세계 설계의 거대한 복잡성에도 불구하고, 아랫단의 아이디어는 동일합니다— Table 6.1에 보인 것과 같은 반복된 어플리케이션의 병렬 fastpath.

6.5 Beyond Partitioning

It is all right to aim high if you have plenty of ammunition.

Hawley R. Everhart

이 챕터는 선형적으로 확장하는 간단한 병렬 프로그램을 설계하는데 데이터 파티셔닝이 어떻게 사용될 수 있는지 알아보았습니다. Section 6.3.4은 데이터 복사의 확률에 대한 힌트를 제공했는데, 이는 Section 9.5에서 큰 효과를 발휘할 겁니다.

파티셔닝과 복사를 활용하는 것의 주요 목표는 선형적인 속도 향상을 얻는 것으로, 달리 말하면 필요한 전체 일의 양이 CPU 또는 쓰레드의 수가 증가함에 따라 크게 증가하지 않음을 보장하는 것입니다. 파티셔닝과, 또는 복사를 통해 해결될 수 있으며 결과적으로 선형적 속도 향상을 이룰 수 있는 문제는 당황스럽게 병렬적입니다. 하지만 우린 이를 더 잘 할 수 있을까요?

이 질문에 답하기 위해, 미궁과 미로 문제를 살펴 볼 시다. 물론, 미궁과 미로는 수백만년동안 유혹의 대상이었으며 [Wik12], 따라서 이것들이 생체 컴퓨터 [Ada11], GPGPU [Eri08], 심지어 분산 하드웨어 [KFC11]를 포함한 컴퓨터들을 통해 생성되고 풀이되었음을 놀랄 일이 아닐 겁니다. 미로에 대한 병렬 풀이가 대학에서의 수업에서의 프로젝트로 가끔 사용되곤 하며 [ETH11, Uni10]

Listing 6.12: SEQ Pseudocode

```

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 }
```

병렬 프로그래밍 프레임워크의 장점을 시연하는 장치로 사용됩니다 [Fos10].

흔한 조언은 parallel work-queue 알고리즘 (PWQ) [ETH11, Fos10] 을 사용하라는 것입니다. 이 섹션은 PWQ를 순차적 알고리즘 (SEQ) 과, 그리고 대안적인 병렬 알고리즘과 무작위적으로 생성된 사각형 미로를 풀어보면서 비교함으로써 이 조언을 평가해보겠습니다. Section 6.5.1 은 PWQ에 대해 논하고, Section 6.5.2 은 대안적인 병렬 알고리즘을 논하며, Section 6.5.3 은 그것의 이해적인 성능을 분석하고, Section 6.5.4 은 이 대안적 병렬 알고리즘으로부터 향상된 순차적 알고리즘을 도출해 보며, Section 6.5.5 은 추가적 성능 비교를 해보고, 마지막으로 Section 6.5.6 은 미래의 방향을 제시하고 결론을 맺습니다.

6.5.1 Work-Queue Parallel Maze Solver

PWQ는 Listing 6.12 (maze_seq.c를 위한 수도코드)에 보인 SEQ에 기반합니다. 미로는 셀의 2D 배열과 선형 배열 기반의 `->visited` 라 이름지어진 work queue로 표현됩니다.

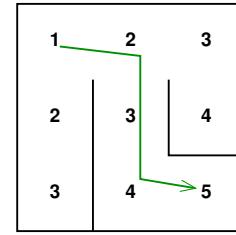
라인 7은 이 시작셀을 방문하고, 라인 8-21의 반복문의 각 반복은 하나의 셀로 향하는 통로를 따라갑니다. 라인 9-13의 루프는 방문되지 않은 이웃과 함께 방문된 셀들을 위해 `->visited[]` 배열을 스캔하고, 라인 14-19의 루프는 이 이웃에 의해 향해지는 부속 미로의 fork를 따라갑니다. 라인 20은 바깥 반복문을 따라서 다음 패스를 위한 초기화를 합니다.

`maze_try_visit_cell()`의 수도코드가 Listing 6.13 (maze.c)의 라인 1-12에 보여져 있습니다. 라인 4는 `c`와 `t` 셀이 근접하고 연결되었는지 체크하며, 라인 5는 `t` 셀이 아직 방문되지 않았는지 검사합니다.

Listing 6.13: SEQ Helper Pseudocode

```

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 6.22:** Cell-Number Solution Tracking

`celladdr()` 함수는 이 특정된 셀의 주소를 리턴합니다. 어떤 체크든 실패한다면, 라인 6은 실패를 리턴합니다. 라인 7은 다음 셀을 가리키며, 라인 8은 `->visited[]` 배열의 다음 슬랏에 이 셀을 기록하고, 라인 9는 이 슬랏이 이제 찾음을 알리며, 라인 10은 이 셀이 방문되었음을 기록하고 또한 미로의 시작점으로부터의 거리도 기록합니다. 라인 11은 이제 성공을 리턴합니다.

`maze_find_any_next_cell()`의 수도코드가 Listing 6.13 (maze.c)의 라인 14-28에 보여져 있습니다. 라인 17은 현재 셀의 거리 더하기 1을 취하고, 라인 19, 21, 23, 그리고 25는 이 셀을 각 방향에서 검사하고, 라인 20, 22, 24, 그리고 26는 이 연관된 셀이 다음 셀 후보라면 `true`를 리턴합니다. `prevcol()`, `nextcol()`, `prevrow()`, 그리고 `nextrow()`는 각각 배열 인덱스 변환 오퍼레이션을 행합니다. 이 셀들 중 어느 것도 후보가 아니라면, 라인 27는 `false`를 리턴합니다.

이 경로는 시작점이 좌상단이고 끝 셀이 우하단인 Figure 6.22에 보여진 것처럼 시작점으로부터의 셀들의

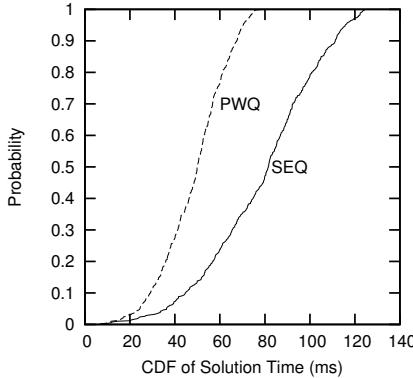


Figure 6.23: CDF of Solution Times For SEQ and PWQ

수를 셈으로써 이 미로에 기록됩니다. 끝지점으로부터 시작해서 연속적으로 값이 감소하는 셀을 따라가면 해결책이 나옵니다.

병렬 work-queue solver는 Listing 6.12 와 6.13에 보인 알고리즘의 간단한 병렬화 버전입니다. Listing 6.12의 라인 10은 fetch-and-add를 사용해야만 하며, 지역 변수 *vi*는 다양한 쓰레드 사이에 공유되어야만 합니다. Listing 6.13의 라인 5와 10은 CAS 반복문으로 구성되어야만 하며, 이 반복문에서의 CAS의 실패는 이 미로에서의 루프의 존재를 가리킵니다. 이 코드에서의 라인 8-9는 *->visited[]* 배열에 셀들을 기록하려는 동시적 시도들을 중재하기 위해 fetch-and-add를 사용해야만 합니다.

이 방법은 500개의 다른 500x500 크기의 무작위로 생성된 미로에 대한 이 두 알고리즘의 해결 시간의 누적 분포함수(CDF)를 보이는 Figure 6.23에 보인 것처럼 2.53 GHz에서 동작하는 듀얼 CPU Lenovo W500에서 상당한 성능향상을 보입니다. 이 CDF의 x축으로의 상당한 겹치는 부분은 Section 6.5.3에서 다뤄질 겁니다.

흥미롭게도, 순차적인 해결책-경로 탐색은 병렬 알고리즘에서도 바뀌지 않았습니다. 하지만, 이는 이 병렬 알고리즘의 상당한 약점을 덮지 못합니다: 항상 최대 하나의 쓰레드만이 이 해결책 경로에서 성과를 낼 수 있습니다. 이 약점은 다음 섹션에서 다룹니다.

6.5.2 Alternative Parallel Maze Solver

짧은 미로 해결기는 종종 양 끝에서 시작될 것을 요구 받고, 이 조언은 자동화된 미로 탐색의 맥락에서 최근 더욱 반복되었습니다 [Uni10]. 이 조언은 운영체제 커널 [BK85, Inm85]와 어플리케이션 [Pat10] 모두를 위한 병렬 프로그래밍의 맥락에서 강력한 병렬화 전략인 파티셔닝이 됩니다. 이 섹션은 해결 경로의 반대 끝단에서

Listing 6.14: Partitioned Parallel Solver Pseudocode

```

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 (READ_ONCE(mp->done))
14                return 1;
15            c = visited[vi];
16        }
17        do {
18            if (READ_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 (!READ_ONCE(mp->done));
24    return 1;
25 }
```

시작하는 두개의 자식 쓰레드를 사용해 이 전략을 적용해 보고, 성능과 확장성 결과에 대해 짧게 알아봅니다.

Listing 6.14 (*maze_part.c*)에 보인 파티셔닝 병렬 알고리즘 (PART)은 SEQ와 비슷하지만 몇 가지 중요한 차이점이 있습니다. 첫째로, 각 자식 쓰레드는 각자의 *visited* 배열을 가지며, 이는 부모로부터 라인 1에 보인 것처럼 전달받는데, 모두 [-1, -1]으로 초기화 되어야만 합니다. 라인 7은 이 배열로의 포인터를 쓰레드별 변수 *myvisited*에 도움 주는 함수들에 의한 접근을 허용하기 위해 저장하며, 비슷하게 지역별 방문 인덱스로의 포인터도 저장합니다. 둘째로, 부모는 각 자식들 대신 첫번째 셀을 방문하는데, 이 자식들은 이를 라인 8에서 얻습니다. 셋째로, 이 미로는 한 자식이 다른 자식이 방문한 셀을 찾는 순간 해결됩니다. 넷째, 각 자식은 따라서 라인 13, 18, 그리고 23에 보인 것처럼 주기적으로 *->done* 필드를 체크해야 합니다. *READ_ONCE()* 기능은 뒤따르는 로드들을 합치거나 이 값을 다시 로드할 수 있는 모든 컴파일러 최적화를 불능화 시켜야 합니다. C++11 *volatile relaxed load*로도 충분할 겁니다 [Smi19]. 마지막으로, *maze_find_any_next_cell()* 함수는 한 셀을 방문되었다고 표시하기 위해 *compare-and-swap*을 사용해야만 합니다만, 쓰레드 생성과 대기에 의해 제공된 것들 외에는 순서보장이 필요치 않습니다.

*maze_find_any_next_cell()*의 수도코드는 Listing 6.13에 보인 것과 동일합니다만, *maze_try_visit_cell()*의 수도코드는 다르며, Listing 6.15에 보이고 있습니다. 라인 8-9는 이 셀들이 연결되어 있는지 검사하고, 그렇지 않다면 실패를 리턴합니다. 라인 11-18의 루프는 이 새 셀을 방문되었다고 표시합니다. 라인 13는 이게 이미 방문되었는지 검사하고, 그 경우에는 라인 16

Listing 6.15: Partitioned Parallel Helper Pseudocode

```

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 = READ_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 }

```

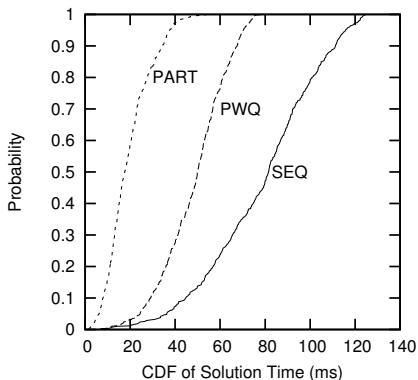


Figure 6.24: CDF of Solution Times For SEQ, PWQ, and PART

에서 실패를 리턴합니다만, 라인 14 낸 우리가 다른 쓰레드를 만났는지 검사한 후이며, 그 경우에는 라인 15에서 해결이 찾아졌다고 알립니다. 라인 19는 새로운 셀로 업데이트를 하며, 라인 20 와 21 는 이 쓰레드의 방문된 셀들 배열을 업데이트하고 라인 22에서 성공을 리턴합니다.

성능 테스트는 놀라운 결과를 드러냈는데, Figure 6.24에 보여 있습니다. PART의 중간 해결 시간(17밀리세컨드)은 SEQ의 것(79밀리세컨드)보다 두개의 쓰레드만을 사용하고 있음에도 네배가 넘게 빨랐습니다. 다음 섹션에서 이 결과를 분석해 볼니다.

6.5.3 Performance Comparison I

예상 외의 성능 결과에 대해 해야 할 첫번째 반응은 버그 검사입니다. 이 알고리즘은 실제로 유효한 미로 해법을

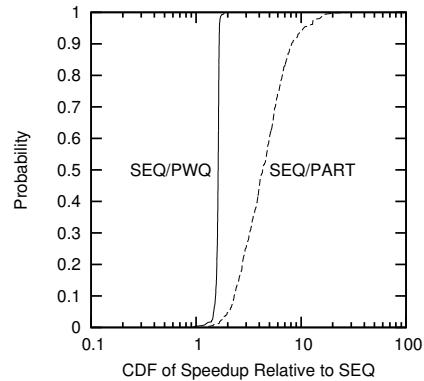


Figure 6.25: CDF of SEQ/PWQ and SEQ/PART Solution-Time Ratios

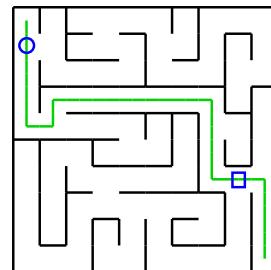


Figure 6.26: Reason for Small Visit Percentages

찾아냈지만, Figure 6.24 의 CDF 그림은 비종속적 데이터 포인트들을 가정합니다. 이건 그 경우가 아닙니다: 이 성능 테스트는 무작위로 미로를 만들어내고, 모든 미로 해법기를 해당 미로에 대해 동작시킵니다. 따라서 Figure 6.25 에 보인 것처럼 각각의 생성된 미로에 대한 해결책 시간의 비율을 CDF 로 그리는게 합리적이며, CDF 들의 겹침을 크게 줄여줍니다. 이 그림은 일부 미로에 대해선, PART 가 SEQ 보다 40 배가 넘게 빠릅니다. 대조적으로, PWQ 는 SEQ 보다 두배 이상 빠르지 않습니다. 두 쓰레드를 사용하고 40배의 속도향상을 이룬것에 대해선 설명이 필요합니다. 어쨌건, 이건 그냥 파티셔닝 가능성은 쓰레드를 추가하기만 한다는 것이 전체 계산 비용을 늘리지는 않음을 의미하는 당황스러울 정도의 병렬성이 아닙니다. 이건 그보다는 굴욕적인 병렬성입니다: 쓰레드를 추가하는 것이 전체 계산 비용을 줄여서 알고리즘의 초선형적 속도향상을 이룹습니다.

계속된 조사는 PART 가 가끔은 미로의 셀들 중 2% 미만의 것들만을 방문한 반면, SEQ 와 PWQ 는 9% 미만을 방문한 적이 없음을 드러냈습니다. 이 차이에 대한 이유는 Figure 6.26 에 보여져 있습니다. 좌상단부터 해결책을 찾는 쓰레드가 원을 만나면, 다른 쓰레드는 미로

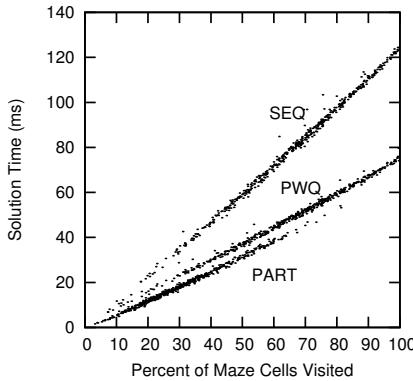


Figure 6.27: Correlation Between Visit Percentage and Solution Time

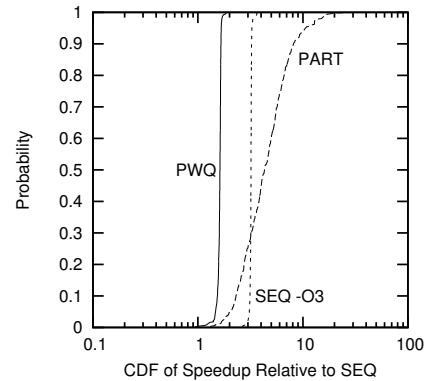


Figure 6.29: Effect of Compiler Optimization (-O3)

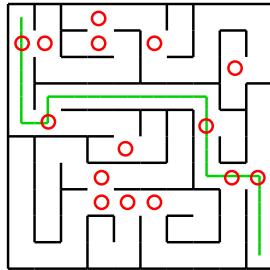


Figure 6.28: PWQ Potential Contention Points

의 우상단에 닿지 못합니다. 비슷하게, 이 다른 쓰레드가 네모를 만나면 첫번째 쓰레드는 미로의 좌하단에 닿지 못합니다. 따라서, PART는 해법이 아닌 셀들의 작은 부분만을 방문하게 될 겁니다. 요약하자면, 이 초선형적 속도향상은 서로의 방향을 향하는 쓰레드들 때문입니다. 이는 각 쓰레드가 서로의 방향 바깥으로 유지시키려 노력해왔던 병렬 프로그래밍에 대한 수십년의 경험에 상당히 대조적인 결과입니다.

Figure 6.27 는 이 세개의 방법들의 해결책 탐색 시간과 방문된 셀들 사이의 강한 상관관계를 확인시켜 줍니다. PART 의 그림의 비탈은 SEQ 의 것보다 작아서, PART 의 한쌍의 쓰레드는 미로의 주어진 부분을 SEQ 의 단일 쓰레드가 하는 것보다 빠르게 방문함을 보입니다. PART 의 그림은 또한 작은 방문 퍼센티지로 기울어 있어서, PART가 더 적은 전체 일을 하며, 따라서 군용적인 병렬성을 보였음을 확인시켜 줍니다.

PWQ 에 의해 방문된 셀들의 부분은 SEQ 의 것과 비슷합니다. 또한, PWQ 의 해법 탐색 시간은 PART 의 그것보다 큰데, 비슷한 방문 비율에 대해서 조차 그렇습니다. 이에 대한 이유가 Figure 6.28 에 그려져 있는데, 각 셀에 두개 이상의 이웃을 가진 빨간 원이 있습니다. 그런 각각의 셀은 PWQ 에서 컨텐션을 유발할 수 있는데, 한

쓰레드가 들어갈 수 있지만 두 쓰레드는 나올 수 없어서, 이 챕터의 앞부분에서 이야기 했듯 성능을 하락시키게 되기 때문입니다. 대조적으로, PART는 그런 컨텐션을 한번만 일으키는데, 해결책이 찾아졌을 때입니다. 물론, SEQ는 결코 컨텐션을 갖지 않습니다.

PART 의 속도 향상이 인상적이긴 하지만, 순차적 최적화를 무시해선 안됩니다. Figure 6.29 는 -O3 로 컴파일 되었을 때 SEQ 가 최적화 되지 않은 PWQ 보다 두배 가량 빨라서, 최적화 되지 않은 PART 에 근접함을 보입니다. 세 알고리즘을 모두 -O3 로 컴파일 했을 때의 결과는 Figure 6.25 에 보인 것과 같이 (더 빨라졌긴 하지만) 비슷한 결과를 보이는데, PWQ 가 SEQ 에 비해 속도 향상은 거의 제공하지 않아, 암달의 법칙 [Amd67] 이 유지됨도 보입니다. 하지만, 목표가 최적화 자체가 아니라 최적화 되지 않은 SEQ 에 비해 두배의 성능을 달성하는 것이라면, 컴파일러 최적화는 상당히 매력적인 것입니다.

캐시 정렬과 패딩은 거짓 공유를 줄임으로써 성능을 종종 향상시킵니다. 하지만, 이 미로 해법 탐색 알고리즘에서 미로 셀 배열을 정렬하고 패딩하는 것은 1000x1000 미로에 대해 최대 42 % 까지 성능을 떨어뜨립니다. 캐시 지역성은 거짓 공유를 막는 것보다 더 중요하며, 특히 큰 미로에 대해선 그렇습니다. 더 작은 20-by-20 이나 50-by-50 미로들에 대해선 정렬과 패딩이 PART 의 경우 최대 40 % 성능 향상을 만들어냈습니다만, 그런 작은 크기에 대해서는 SEQ 가 어쨌건 더 나은데 PART 가 쓰레드생성과 정리에 드는 오버헤드를 감당할 충분한 시간이 없기 때문입니다.

요약하자면, 파티셔닝 된 병렬 미로 해법 탐색기는 알고리즘상의 초선형적 속도향상의 흥미로운 예입니다. 만약 “알고리즘상의 초선형적 속도향상” 이 인지부조화를 일으킨다면, 다음 섹션으로 넘어가시기 바랍니다.

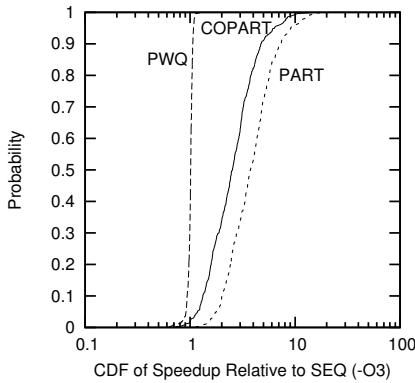


Figure 6.30: Partitioned Coroutines

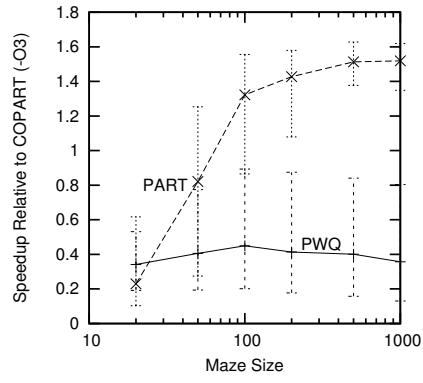


Figure 6.32: Varying Maze Size vs. COPART

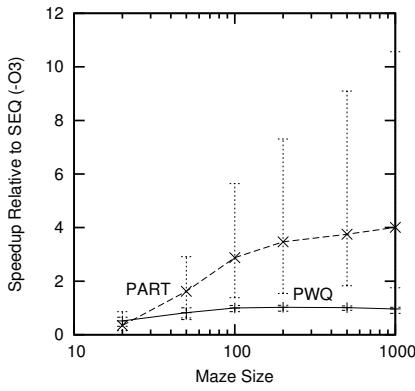


Figure 6.31: Varying Maze Size vs. SEQ

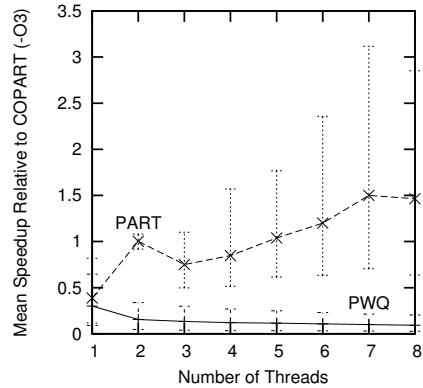


Figure 6.33: Mean Speedup vs. Number of Threads, 1000x1000 Maze

6.5.4 Alternative Sequential Maze Solver

알고리즘상의 초선형적 속도향상의 존재는 예를 들면 Listing 6.14 의 메인 do-while 루프를 통과하는 각 패스에서 쓰레드간에 일일이 컨텍스트 스위칭 (context switching)을 하는 식의, 코루틴을 통한 병렬성을 시뮬레이션 해볼 것을 제안합니다. 이 컨텍스트 스위칭은 이 컨텍스트가 변수 c 와 vi 만으로 구성되어 있기 때문에 간단합니다: 이 효과를 이를 수 있는 많은 방법들 가운데, 이는 컨텍스트 스위치 오버헤드와 방문 퍼센티지 사이의 좋은 트레이드오프입니다. Figure 6.30 에서 볼 수 있듯, 이 코루틴 알고리즘 (COPART)은 상당히 효과적으로, 단일 쓰레드로 두개 쓰레드를 사용하는 PART 의 30 % 정도 성능이 됩니다 (`maze_2seq.c`).

6.5.5 Performance Comparison II

Figure 6.31 와 6.32 는 당야한 미로 크기에 대해 이 효과를 보이는데, 두개의 쓰레드를 사용해 동작하는 PWQ 와 PART 를 SEQ 또는 COPART 에 대해 각각 비교하

는데, 90-percent-confidence 에러바도 보입니다. PART 는 100-by-100 과 그보다 큰 미로에 있어선 SEQ 에 비교해 초선형적인, 그리고 COPART 에 대해선 약간의 확장성을 보입니다. 대략 200-by-200 미로 크기부터는 전력 소모는 대략적으로 높은 주파수에서는 주파수의 제곱에 비례하여 [Mud01] 두 쓰레드에서의 1.4x 확장은 똑같은 해법 탐색 시간에서의 단일 쓰레드와 같은 전력을 소모하므로 200-by-200 미로 크기부터는 PART 는 COPART 의 이론적 에너지 효율성 손익분기를 넘어섭니다. 대조적으로, PWQ 는 SEQ 와 COPART 모두에 대해 최적화가 되지 않은 상태가 아니고는 낮은 확장성을 보입니다: Figure 6.31 와 6.32 는 -O3 를 사용해 만들어졌습니다.

Figure 6.33 는 PWQ 와 PART 의 성능을 COPART 에 상대적으로 보입니다. 두개가 넘는 쓰레드로 돌아가는 PART 를 위해선, 추가된 쓰레드는 시작과 끝 셀을 연결하는 대각선상에서 똑같이 나뉘어진 위치에서 시작되었습니다. 두개 이상의 쓰레드를 사용해 동작하는 PART

의 조기 종료를 알아내기 위해 단순화된 연결 상태 라우팅 [BG87]이 사용되었습니다(한 쓰레드가 시작과 끝과 모두 연결되었을 때 해결책이 찾아진 것으로 취급됩니다). PWQ는 무척 처참한 성능을 보입니다만, PART는 두개 쓰레드에서 손익분기를 치고 5개 쓰레드에서도 그런데, 다섯개 쓰레드 이후부터 약간의 속도향상만 달성합니다. 이론적 전력 효율성 손익분기는 7개와 8개 쓰레드에 대해선 90-percent-confidence 간격 내에 있습니다. 두개 쓰레드에서 피크를 치는 이유는 (1) 두개 쓰레드의 경우에서의 종료 인식의 낮은 복잡도와 (2) 세번째와 그 이후 쓰레드들이 유용한 진척을 낼 확률이 낮다는 사실입니다: 처음의 두 쓰레드만이 해결책의 선상에서 시작될 것이 보장됩니다. Figure 6.32 의, 결과에 의해 실망스러운 이 성능은 2.66 GHz에서 동작하는 더 크고 오래된 Xeon 시스템의 덜 가깝게 통합된 하드웨어 때문입니다.

6.5.6 Future Directions and Conclusions

많은 할일이 남아있습니다. 첫째로, 이 섹션은 미로 해결책 중 사람이 하는 기법 하나만을 적용해 봤습니다. 다른 방법으로는 미로의 부분들을 배제시키기 위해 벽을 따라가고 앞서 탐색된 경로의 위치에 기반해 내부 시작 지점을 고르는 방법이 있습니다. 둘째로, 시작과 끝 지점의 다른 선택은 다른 알고리즘을 선호할 수도 있습니다. 셋째로, PART 알고리즘의 첫 두개 쓰레드 위치 선택이 명료하긴 하지만, 남아있는 쓰레드의 시작 위치 지정에는 여러 선택이 있을 수 있습니다. 최적의 위치 선정은 시작과 끝 지점에 의존적일 수도 있습니다. 넷째로, 해결책 없는 미로와 순환 |로에 대한 연구는 흥미로운 결과를 낼 수도 있습니다. 다섯째로, 가벼운 C++11 어토믹 오퍼레이션은 성능을 개선할 수도 있습니다. 여섯째로, 이 성능향상을 3차원 미로(또는 심지어 그보다 높은 차원의 미로)에 대해 비교해 보는 것도 흥미로울 겁니다. 마지막으로, 미로들에 대해 부끄러운 병렬성은 코루틴을 사용한 더 효율적인 순차적 구현이 가능함을 알렸습니다. 부끄럽도록 병렬적인 알고리즘은 항상 더 효율적인 순차적 구현을 이끌어낼까요, 아니면 코루틴 컨텍스트 스위치 오버헤드가 성능향상을 짊어 삼기는 필연적으로 부끄러울 정도의 병렬 알고리즘이 있을까요?

이 섹션은 미로 해결 알고리즘의 병렬화를 선보이고 분석했습니다. 전통적인 work queue 기반 알고리즘은 컴파일러 최적화가 사용되지 않을 때만 잘 동작했으므로, 이는 고수준/높은 오버헤드의 언어를 사용해 얻어진 기존의 결과들은 최적화의 발달과 함께 더이상 유효하지 않을 수도 있음을 의미합니다.

이 섹션은 병렬화를 개선된 순차적 알고리즘의 길을 여는 순차적 알고리즘의 파생이라기보다는 첫번째 최적화 기법으로 사용하는 분명한 예를 보였습니다. 높은

수준의 설계 시점에서의 병렬성 적용은 흥미로운 연구 주제가 될 겁니다. 이 섹션은 미로 해결 문제를 담백한 확장성부터 부끄러울 정도로 병렬적일 정도까지 그리고 그 뒤로 다시 돌아가며 알아봤습니다. 이 경험이 병렬화가 이미 존재하는 프로그램에 대해 회고될 작은 수준의 최적은 아닌 사소한 최적화보다는 전체 어플리케이션 최적화 기법의 설계 시점에 처음으로 고려되는 존재가 되도록 하기를 바랍니다.

6.6 Partitioning, Parallelism, and Optimization

Knowledge is of no value unless you put it into practice.

Anton Chekhov

가장 중요한 건, 이 챕터가 병렬성을 설계 수준에서 적용하는 게 훌륭한 결과를 내놓음을 보이긴 했지만, 이 마지막 섹션은 이게 충분치 않음을 보였습니다. 미로 탐색과 같은 탐색 문제들에 대해, 이 섹션은 탐색 전략이 병렬 설계보다도 더 중요함을 보였습니다. 그렇습니다, 이 미로의 경우에는, 현명하게 병렬성을 적용하는 게 훌륭한 탐색 전략을 찾아냈지만, 이런 종류의 행운은 분명한 탐색 전략 자체의 대체는 아닙니다.

Section 2.2에서 앞서 이야기 되었듯, 병렬성은 많은 잠재적 최적화 중 하나일 뿐입니다. 성공적인 설계는 가장 중요한 최적화에 초점을 맞춰야 합니다. 저는 다른 쪽으로 이야기 하고 싶을 수도 있지만, 최적화는 병렬화일 수도, 그렇지 않을 수도 있습니다.

하지만, 병렬성이 올바른 최적화인 많은 경우를 위해, 다음 섹션은 동기화를 위해 가장 많이 사용되는 것인 락킹을 다뤄봅니다.

Chapter 7

Locking

최근의 동시성 연구에서, 락킹은 종종 악당의 역할을 맡게 됩니다. 락킹은 deadlock, convoying, starvation, unfairness, data race, 그리고 모든 다른 동시성에서의 죄악적인 것들을 일으킨다는 비난을 받고 있습니다. 흥미롭게도, 제품 수준 공유 메모리 병렬 소프트웨어에서의 아주 많은 일을 처리하는 사람의 역할 역시 락킹이 맡고 있습니다. 이 챕터는 Figure 7.1 와 7.2에서 그린 것과 같은 이 악당과 영웅 사이의 이분법을 들여다 봅니다.

이 지킬과 하이드 이분법에는 여러 이유들이 있습니다:

1. 락킹의 죄악 중 많은 것들은 대부분의 경우에 잘 동작하는 실용적 설계 디자인이 있는데, 예를 들면:
 - (a) 데드락 방지를 위해 락 계층을 사용하는 것.
 - (b) 리눅스 커널의 lockdep [Cor06a] 과 같은 데드락 탐지 도구들.
 - (c) 배열, 해시 테이블, radix tree 와 같은, 챕터 10에서 다루어질 락킹에 친화적인 데이터 구조들.
2. 락킹의 죄들 중 일부는 잘 설계되지 못한 프로그램에서만 이를 수 있는 높은 수준의 contention에서만 존재합니다.
3. 락킹의 죄들 중 일부는 락킹과 함께 다른 동기화 메커니즘들을 사용함으로써 방지될 수 있습니다. 이런 다른 메커니즘들에는 통계적 카운터 (Chapter 5를 참고하세요), 레퍼런스 카운터 (Section 9.2를 참고하세요), 해저드 포인터 (Section 9.3를 참고하세요), 시퀀스 락킹 읽기 쓰레드 (Section 9.4를 참고하세요), RCU (see Section 9.5), 그리고 간단한 non-blocking 데이터 구조들 (Section 14.2를 참고하세요) 이 있습니다.
4. 최근 전까지, 거의 모든 거대 공유 메모리 병렬 프로그램들이 비밀리에 개발되었으며, 따라서 이런 실용적 해결책을 배우기가 쉽지 않았습니다.

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*

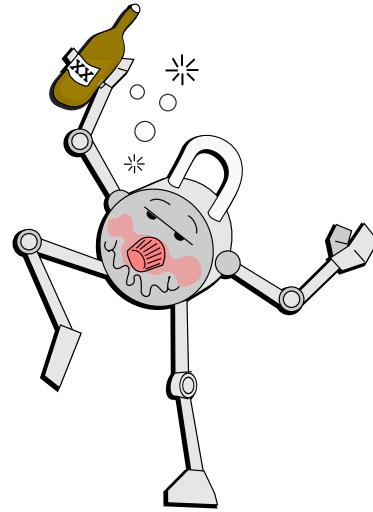


Figure 7.1: Locking: Villain or Slob?

5. 락킹은 일부 소프트웨어 작품에 대해서는 무척 잘 동작하지만 다른 것들에는 무척 잘 동작하지 못합니다. 락킹이 잘 동작하는 작품을 위해 일해온 개발자들은 락킹이 잘 동작하지 못하는 작품을 위해 일해온 사람들에 비해 락킹에 대해 훨씬 긍정적인 의견을 가지게 될 것이라 예상할 수 있는데, 이에 대해 Section 7.5에서 다릅니다.
6. 모든 좋은 이야기에는 악당이 필요하기 마련이며, 락킹은 연구 논문의 희생양으로 일하는 길고 영광스러운 역사를 가지게 되었습니다.

Quick Quiz 7.1: 희생양 역할을 하는게 어떻게 영광스러운 것일 수 있죠???

■
이 챕터는 락킹의 더 심각한 죄들을 막기 위한 여러 방법을 알아봅니다.

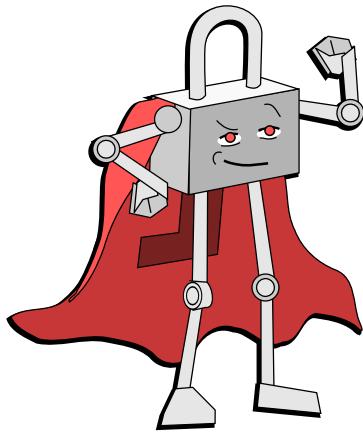


Figure 7.2: Locking: Workhorse or Hero?

7.1 Staying Alive

I work to stay alive.

Bette Davis

락킹이 deadlock 과 starvation 으로 인해 비난받는다는 점을 놓고 볼 때, 공유 메모리 병렬 개발자들의 중요한 걱정 중 하나는 단순히 살아있음을 유지하는 것입니다. 따라서 다음 섹션들에서는 deadlock, lovelock, starvation, unfairness, 그리고 비효율성에 대해 다룹니다.

7.1.1 Deadlock

Deadlock 은 어느 쓰레드 그룹의 각 멤버가 같은 그룹 내의 다른 멤버의 락을 기다리면서 각자 최소 하나씩 락을 쥐고 있을 때 발생합니다. 이는 하나의 쓰레드만 가지고 있는 그룹들에서도 이 쓰레드가 이미 쥐고 있는 비회귀적인 락을 획득하려 할 때 발생합니다. Deadlock 은 따라서 하나의 쓰레드와 하나의 락만 존재할 때조차도 일어날 수 있습니다!

어떤 외부 개입이 없이는, deadlock 은 영원히 갑니다. 어떤 쓰레드도 그것을 쥐고 있는 쓰레드가 그 락을 해제하기 전까지는 그 락을 획득할 수 없으나, 그 락을 쥐고 있는 쓰레드는 기다리고 있는 락을 획득하기 전까지는 쥐고 있는 락을 놓을 수 없습니다.

Figure 7.3 에 보인 것과 같이 쓰레드와 락을 노드로 표현하고 방향성 있는 그래프로 deadlock 시나리오를 표현할 수 있습니다. 락으로부터 쓰레드로의 화살표는 이 쓰레드가 이 락을 쥐고 있음을 의미하는데, 예를 들어 쓰레드 B 는 락 2 와 4 를 쥐고 있습니다. 쓰레드로부터 락으로의 화살표는 이 쓰레드가 이 락을 기다리고 있음

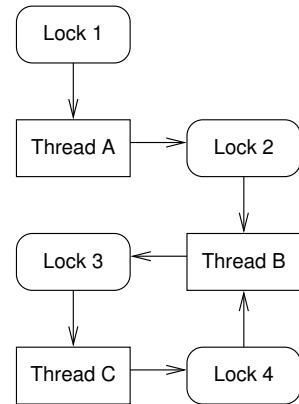


Figure 7.3: Deadlock Cycle

을 의미하는데, 예를 들어 쓰레드 B 는 락 3 을 기다리고 있습니다.

Deadlock 시나리오는 항상 최소 하나의 deadlock 사이클을 가질 겁니다. Figure 7.3 에서, 이 사이클은 쓰레드 B, 락 3, 쓰레드 C, 락 4, 그리고 다시 쓰레드 B 로 구성됩니다.

Quick Quiz 7.2: 하지만 락 기반의 deadlock 의 정의는 각 쓰레드가 최소 하나의 락을 쥔 채 어떤 다른 쓰레드가 쥐고 있는 락을 기다린다였습니다. 사이클이 존재하는지 어떻게 알 수 있죠?

■

존재하는 deadlock 으로부터의 회복을 할 수 있는 데 이터베이스 시스템과 같은 소프트웨어 환경이 존재하지만, 이 방법은 쓰레드 중 하나가 죽임당하거나 락이 어떤 쓰레드들로부터 강제로 빼앗겨져야 할 것을 필요로 합니다. 이런 죽임과 강제 빼앗기는 트랜잭션들에 대해서는 잘 동작합니다만, 커널과 어플리케이션 수준의 락킹 사용에서는 종종 문제가 됩니다: 결과적으로 부분적으로만 업데이트 된 구조들을 다루는 것은 무척 복잡하고, 재앙에 가깝고, 애러를 만들기 내기 쉽습니다.

따라서, 커널과 어플리케이션은 deadlock 의 존재를 막아야 합니다. Deadlock 방지 전략에는 락킹 계층 (Section 7.1.1.1), 지역적 락킹 계층 (Section 7.1.1.2), 레이어 기반 락킹 계층 (Section 7.1.1.3), 락들로의 포인터들을 포함하는 API 들을 다루는 전략 (Section 7.1.1.4), 조건적 락킹 (Section 7.1.1.5), 필요한 락들을 모두 획득하고 시작하기 (Section 7.1.1.6), 한번에 하나의 락만 잡기 설계 (Section 7.1.1.7), 그리고 시그널/인터럽트 핸들러를 위한 전략 (Section 7.1.1.8) 등이 있습니다. 모든 경우에 완벽하게 동작하는 deadlock 방지 전략은 존재하지 않지만, 사용 가능한 것들 중에서 적절한 도구를 선택하는 것은 가능합니다.

7.1.1.1 Locking Hierarchies

락킹 계층은 락들을 순서지어서 순서에 맞지 않게 락을 획득하는 것을 막습니다. Figure 7.3에서, 우린 락들을 숫자 기반으로 순서를 지어서, 어떤 쓰레드가 같거나 큰 수의 락을 쥐고 있다면 락을 획득하지 못하게 할 수 있습니다. 쓰레드 B는 락 4를 전 상태에서 락 3을 획득하려 하고 있으므로 이 계층 규칙을 위반하고 있습니다. 이 위반이 이 deadlock이 일어날 수 있게 했습니다.

다시 말하지만, 락킹 계층을 적용하기 위해선, 락들을 순서를 짓고 순서 외의 락 획득을 막아야 합니다. 거대한 프로그램에서는 락킹 계층을 강제하기 위해 리눅스 커널의 lockdep [Cor06a]과 같은 도구들을 사용하는 게 현명합니다.

7.1.1.2 Local Locking Hierarchies

하지만, 락킹 계층의 전역적 특성은 그것을 라이브러리 함수에 적용하기 어렵게 합니다. 무엇보다도, 특정 라이브러리 함수를 사용하는 어떤 프로그램이 아직 작성되지 않았을 때, 이 불쌍한 라이브러리 함수 구현자는 아직 정의되지 않은 락킹 계층을 따를 수 있겠습니까?

한 가지 특수한 (하지만 흔한) 경우는 이 라이브러리 함수가 호출자의 코드를 호출하지 않을 때입니다. 이 경우, 이 호출자의 락은 이 라이브러리의 락을 잡고 있는 사이에 얻어지려 하지 않을 것이며, 따라서 라이브러리와 호출자 모두의 락이 포함된 데드락 사이클은 존재하지 않을 것입니다.

Quick Quiz 7.3: 이 규칙에 대한 어떤 예외가 있어서, 라이브러리 코드가 호출자의 함수를 전혀 호출하지 않음에도 라이브러리와 호출자 양쪽의 락이 포함된 데드락 사이클이 존재할 수도 있나요?



하지만 이 라이브러리 함수는 호출자의 코드를 수행한다고 가정해 봅시다. 예를 들어, `qsort()`는 호출자가 제공한 비교 함수를 호출합니다. 이 비교 함수는 일반적으로는 변하지 않는 로컬 데이터를 가지고 동작해서, Figure 7.4에 보인 것처럼 락을 잡을 필요가 없을 겁니다. 하지만 어떤 사람은 키가 변화하는 데이터 모음을 정렬할 만큼 미쳐 있어서 이 비교 함수가 락을 잡을 것을 필요로 할 수도 있는데, 이것은 Figure 7.5에 보인 것처럼 데드락을 유발할 수도 있습니다. 이 라이브러리 함수는 어떻게 데드락을 막을 수 있을까요?

이 경우의 황금 규칙은 “알려지지 않은 코드를 수행하기 전에 모든 락을 해제하라”입니다. 이 규칙을 따르기 위해, `qsort()` 함수는 비교 함수를 호출하기 전에 자신의 모든 락을 내려 놓아야 합니다. 따라서 `qsort()`는 비교 함수가 호출자의 락을 잡는 동안 자신의 어떤 락도 쥐고 있지 않으며, 따라서 데드락이 방지됩니다.

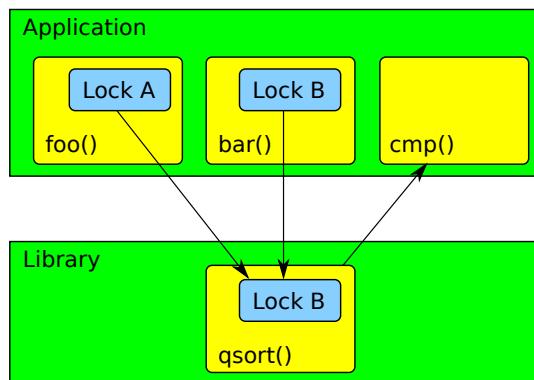


Figure 7.4: No `qsort()` Compare-Function Locking

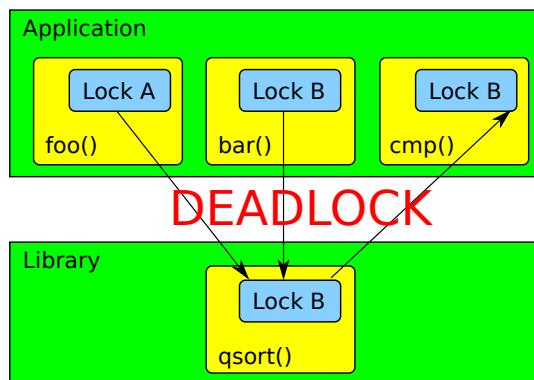


Figure 7.5: Without `qsort()` Local Locking Hierarchy

Quick Quiz 7.4: 하지만 `qsort()`가 비교 함수를 호출하기 전에 자신의 락을 모두 내려놓는다면, 다른 `qsort()` 쓰레드와의 레이스로부터 어떻게 자신을 보호하나요?



로컬 락킹 계층화의 장점을 보기 위해, Figure 7.5와 7.6를 비교해 보시기 바랍니다. 두 그림에서, 어플리케이션 함수 `foo()`와 `bar()`는 각각 락 A와 B를 잡고서 `qsort()`를 호출합니다. 이것은 `qsort()`의 병렬 구현이므로, 락 C를 잡습니다. 함수 `foo()`는 함수 `cmp()`를 `qsort()`에 넘기고, `cmp()`은 락 B를 잡습니다. 함수 `bar()`는 간단한 정수 비교 함수 (여기엔 보이지 않았습니다)를 `qsort()`에 넘기고, 이 간단한 함수는 어떤 락도 잡지 않습니다.

이제, `qsort()`가 Figure 7.5에 보인 것처럼 `cmp()`를 호출하는 동안 락 C를 잡고 있어서 앞의 황금 규칙을 어긴다면, 데드락이 발생할 수 있습니다. 이를 자세히 보

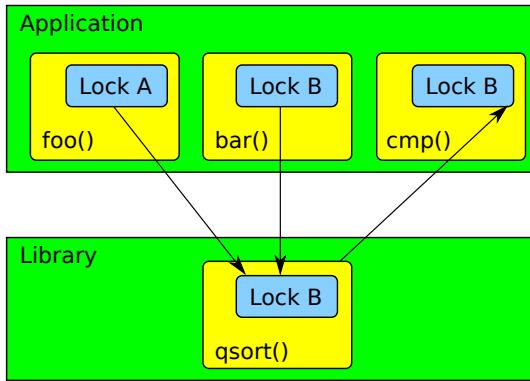


Figure 7.6: Local Locking Hierarchy for `qsort()`

기 위해, 한 쓰레드가 두 번째 쓰레드가 동시에 `bar()`를 수행하는 사이에 `foo()`를 호출한다고 생각해 봅시다. 첫 번째 쓰레드의 `qsort()` 호출은 락 C를 잡을 것이고, 이어서 `cmp()`를 호출할 때 락 B를 잡을 수 없을 겁니다. 하지만 첫 번째 쓰레드는 락 C를 잡고 있고, 따라서 두 번째 쓰레드의 `qsort()`는 그 락을 잡을 수 없고, 따라서 락 B를 해제할 수 없어서 데드락에 이르게 됩니다.

대조적으로, `qsort()` 각 락 C를 `qsort()`의 관점에서는 알려지지 않은 코드인 이 비교 함수를 호출하기 전에 내려놓는다면, Figure 7.6에 보인 것과 같이 데드락이 방지됩니다.

각 모듈이 알려지지 않은 코드를 호출하기 전에 모든 락을 내려놓는다면, 각 모듈이 개별적으로 데드락을 방지하고 있는 한 전체 데드락이 방지됩니다. 따라서 이 규칙은 데드락 분석을 상당히 단순화하고 모듈성을 크게 개선합니다.

7.1.1.3 Layered Locking Hierarchies

불행히도, `qsort()`가 비교 함수를 호출하기 전에 자신의 락들을 모두 내려놓는 것은 불가능할 수도 있습니다. 이런 경우, 우린 알려지지 않은 코드를 호출하기 전에 모든 락을 내려놓아 로컬 락킹 계층을 구성하지 못할 겁니다. 하지만, 우리는 그 대신에 Figure 7.7에 보인 것처럼 레이어 기반 락킹 계층을 만들 수 있습니다. 여기서, `cmp()` 함수는 락 A, B, 그리고 C를 모두 획득한 후에 새로운 락 D를 사용해서 데드락을 막습니다. 따라서 우리는 글로벌 데드락 계층에 세 개의 레이어를 갖는 셈인데, 첫 번째 레이어는 락 A와 B를, 두 번째 레이어는 락 C를, 그리고 세 번째 레이어는 락 D를 갖습니다.

기계적으로 `cmp()`를 새 락 D를 사용하도록 바꾸는 것은 일반적으로 불가능함을 알아 두시기 바랍니다. 상당히 그 반대입니다: 설계 수준에서의 근본적 변경이

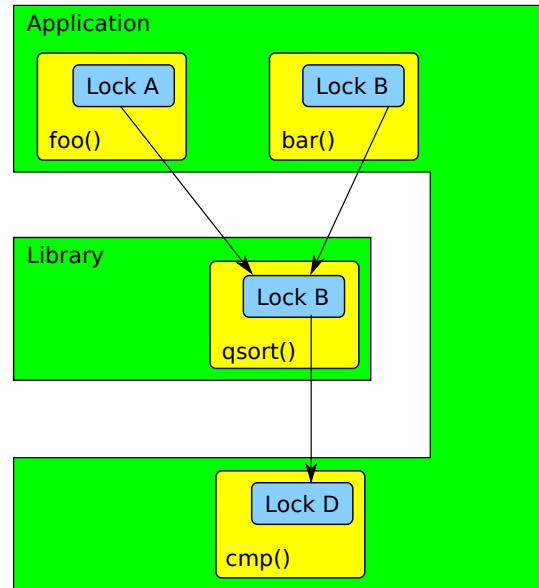


Figure 7.7: Layered Locking Hierarchy for `qsort()`

종종 필요합니다. 그러나, 그런 변경에 필요한 노력을 일반적으로 데드락을 방지하기 위한 비용으로는 작은 것입니다. 이 부분에 대해 더 말해보자면, 이 잠재적 데드락은 어떤 코드가 만들어지기 전, 설계 시간에 발견되는 것이 선호되어야 합니다!

알려지지 않은 코드를 호출하기 전에 모든 락을 내려놓는 것이 불가능한 또 다른 예를 위해서는 Listing 7.1 (`locked_list.c`)에 보인 것과 같은 링크드 리스트를 순회하는 코드를 생각해 보시기 바랍니다. `list_start()` 함수는 이 리스트에서 락을 획득하고 첫 번째 원소를 (존재한다면) 리턴하고, `list_next()`는 이 리스트의 다음 원소로의 포인터를 리턴하거나 이 리스트의 끝에 도달했다면 이 락을 해제하고 `NULL`을 리턴합니다.

Listing 7.2이 이 리스트 순회자가 어떻게 사용될 수 있는지 보입니다. 라인 1-4는 하나의 정수를 담는 `list_ints` 원소를 정의하며, 라인 6-17는 이 리스트를 어떻게 순회하는지 보입니다. 라인 11은 리스트를 잡고 첫 번째 원소로의 포인터를 가져오고, 라인 13은 우리의 `list_ints` 구조체로의 포인터를 제공하며, 라인 14는 연관된 정수를 출력하고, 라인 15는 다음 원소로 넘어갑니다. 이는 상당히 간단하며, 모든 락킹을 숨기고 있습니다.

즉, 각 리스트 원소를 처리하는 코드가 데드락을 초래하게끔 그 스스로 `list_start()`나 `list_next()` 호출을 건너 잡하는 락을 획득하지 않는다면 락킹은

숨겨져 있습니다. 우린 이 락킹 계층이 리스트 순회자 락킹을 처리하게끔 레이어를 추가함으로써 데드락을 막을 수 있습니다.

이 레이어 기반 접근법은 임의의 많은 수의 레이어들로 확장될 수 있습니다만, 추가되는 각 레이어는 락킹 설계의 복잡도를 증가시킵니다. 그런 복잡도 증가는 일반적으로 몇몇 종류의 객체 지향 설계에 불편을 가중시키는데, 제멋대로 많은 객체 그룹 사이를 오가는 컨트롤의 경우 그렇습니다.¹ 이 객체 지향 설계 습관과 데드락 방지의 필요성 사이의 미스매치는 병렬 프로그래밍이 누군가에게는 어렵다고 인식되는 중요한 이유입니다.

고도의 레이어 기반 계층에 대한 일부 대안이 Chapter 9에서 다뤄집니다.

7.1.4 Locking Hierarchies and Pointers to Locks

일부 예외가 있기는 하지만, 락으로의 포인터를 포함하는 외부 API는 종종 잘못 설계된 API입니다. 내부의 락을 다른 소프트웨어 컴포넌트에 넘기는 것은 어쨌건 핵심 설계 교리인 정보 은닉의 안티테제입니다.

Quick Quiz 7.5: 락으로의 포인터가 함수에 넘겨지는 흔한 경우 하나를 이야기해 보시죠.

한가지 예외는 어떤 것들을 넘겨주는 함수인데 이 호출자의 락이 이 넘겨주기가 완료될 때까지는 잡혀져 있어야 하지만 이 락이 이 함수의 리턴 전에 해제되어야 하는 경우입니다. 그런 함수의 예 중 하나로는 POSIX `pthread_cond_wait()` 함수가 있겠는데, 여기선 잃어버린 깨우기 때문에 멈춰있는 경우를 막기 위해 `pthread_mutex_t`로의 포인터가 넘겨집니다.

Quick Quiz 7.6: `pthread_cond_wait()` 이 먼저 이 뮤텍스를 해제하고 다시 그 뮤텍스를 획득한다는 사실은 데드락의 가능성을 제거하지 않나요?

요약하자면, 여러분이 외부로의 API에 락으로의 포인터를 인자로 또는 리턴 값으로 노출하고 있다면, 여러분의 API 설계를 다시 고민해 보세요. 그게 맞는 행동일 수도 있습니다만, 경험은 그렇지 않을 가능성이 높다고 이야기 합니다.

7.1.1.5 Conditional Locking

하지만 합리적인 락킹 계층이 존재하지 않는다고 생각해 봅시다. 이는 실제 삶에서 일어날 수 있는데, 예를 들어, 패킷이 양방향으로 떠다니는 어떤 종류의 레이어화 된 네트워크 프로토콜 스택, 또는 예를 들어 분산된 락 관리자 구현이 있겠습니다. 네트워킹의 경우, 패킷을 한 레이어에서 다른 레이어로 넘길 때 양 레이어로부터

Listing 7.1: Concurrent List Iterator

```

1 struct locked_list {
2     spinlock_t s;
3     struct cds_list_head h;
4 };
5
6 struct cds_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 cds_list_head *list_next(struct locked_list *lp,
13                                 struct cds_list_head *np)
14 {
15     struct cds_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 }
```

Listing 7.2: Concurrent List Iterator Usage

```

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

¹ 이에 대한 이름들 중 하나는 “객체 지향 스파게티 코드”입니다.

Listing 7.3: Protocol Layering and Deadlock

```

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);

```

Listing 7.4: Avoiding Deadlock Via Conditional Locking

```

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);

```

락을 잡아야 할 수도 있습니다. 패킷은 프로토콜 스택의 위로도 아래로도 갈 수 있다는 점을 생각하면, 이는 Listing 7.3에 그런 것처럼 데드락을 일으키기 위한 훌륭한 방법입니다. 여기서, 통신선을 향해 스택의 아래 방향으로 움직이는 패킷은 다른 레이어의 락을 순서 반대로 잡아야만 합니다. 통신선으로부터 멀어지는, 스택 위쪽으로 움직이는 패킷은 락을 순서대로 잡는다는 점을 놓고 보면, 라인 4에서의 락 획득은 데드락을 일으킬 수 있습니다.

이 경우에 데드락을 방지하는 한 가지 방법은 락킹 계층을 암시하지만 락을 반대 순서로 잡아야 할 경우에는 Listing 7.4에 보인 것처럼 그것을 조건적으로 잡는 것입니다. 무조건적으로 layer-1 락을 잡는 대신, 라인 5은 `spin_trylock()` 기능을 사용해 조건적으로 락을 잡습니다. 이 기능은 이 락이 잡을 수 있다면 바로 잡지만 (0이 아닌 값을 리턴합니다), 그렇지 않다면 락을 잡지 않은 채 0을 리턴합니다.

이 `spin_trylock()`이 성공했다면, 라인 15은 필요한 layer-1 처리를 진행합니다. 그렇지 않다면, 라인 6은 락을 내려놓고, 라인 7과 8는 그것들을 옮은 순서로 획득합니다. 불행히도, 시스템에는 여러 네트워킹 디바이스가 있을 수 있으며 (예를 들면, 이더넷과 와이파이), 따라서 `layer_1()` 함수는 라우팅 결정을 내려야만 합니다. 이 결정은 언제든 바뀔 수 있는데, 특히나 이 시스템이 모바일이라면 그렇습니다.² 따라서, 라인 9은

이 결정을 다시 검사하고, 그게 변경되었다면, 이 락을 내려놓고 재시작해야 합니다.

Quick Quiz 7.7: Listing 7.3에서 Listing 7.4로의 변환은 어디서든 적용될 수 있나요?

Quick Quiz 7.8: 하지만 Listing 7.4의 복잡도는 그게 데드락을 방지한다는 점을 생각하면 가치있죠, 그렇죠?

7.1.1.6 Acquire Needed Locks First

조건적 락킹의 중요한 특수 경우에는 모든 필요한 락들이 어떤 처리가 이어지기 전에 획득됩니다. 이 경우, 처리는 면등적일 필요가 없습니다: 이미 획득되어 있는 락을 먼저 내려놓지 않고는 해당 락을 획득할 수 없다면, 모든 락을 내려놓고 다시 시도합니다. 모든 락들이 획득된 다음에야 어떤 처리든 이어질 수 있습니다.

하지만, 이 방법은 Section 7.1.2에서도 이야기 될 *livelock*을 초래할 수 있습니다.

Quick Quiz 7.9: Section 7.1.1.6에서 설명된 “필요한 락들을 먼저 획득하기” 접근법을 사용할 때 *livelock*은 어떻게 회피할 수 있나요?

연관된 접근법인 two-phase locking [BHG87]은 트랜잭션 데이터베이스 시스템에서 오랫동안 제품군에서 사용되었습니다. Two-phase locking의 첫 번째 단계에서는 락들이 획득되지만 해제되지 않습니다. 일단 모든 필요한 락들이 획득되면, 트랜잭션은 두 번째 단계로 들어가서, 락들은 해제되지만 하지 획득되지 않습니다. 이 락킹 접근법은 데이터베이스가 트랜잭션에 serializability 보장을 제공하게 하는데, 달리 말하자면 트랜잭션에 의해 보이고 만들어지는 모든 값들이 모든 트랜잭션 사이의 어떤 전역적 순서와 함께 일관적일 것을 보장하기 위함입니다. 그런 많은 시스템이 트랜잭션을 중단할 수 있는 기능에 의존하는데, 이게 모든 락들이 획득되기 전까지 공유된 데이터에 어떤 변경도 만드는 걸 회피함으로써 단순화 될 수 있긴 합니다. *Livelock*과 *deadlock*은 그런 시스템에서 문제가 됩니다만, 실용적인 해결책은 여러 데이터베이스 교재에서 찾을 수 있을 겁니다.

7.1.1.7 Single-Lock-at-a-Time Designs

어떤 경우에는 락을 중첩하는 것을 막을 수 있어서 *deadlock*을 회피할 수도 있습니다. 예를 들어, 문제가 완전히 분할 가능하다면, 각 분할된 조각별로 하나의 락을 할당할 수 있습니다. 그러면 해당 조각에서 일하는 쓰레드는 연관된 락 하나만 잡으면 됩니다. 어떤 쓰레드도 한번에 두 개 이상의 락을 잡지 않으므로, *deadlock*은 불가능합니다.

² 그리고, 1900년대와 달리, 모바일 환경은 혼합니다.

하지만, 어떤 락도 잡히지 않은 상태에서도 필요한 데 이터 구조가 존재함을 보장하기 위한 어떤 메커니즘이 있어야만 합니다. 그런 메커니즘 중 하나가 Section 7.4에서 설명되며 다른 것들 몇 가지가 Chapter 9에서 설명됩니다.

7.1.1.8 Signal/Interrupt Handlers

시그널 핸들러가 연관되는 deadlock은 시그널 핸들러 내에서 `pthread_mutex_lock()`을 호출하는 것이 합법적이지 않음을 알리는 것으로 종종 금방 사라지기도 합니다. 하지만, 시그널 핸들러에서 호출될 수 있는 손으로 만든 락킹 기능을 사용하는 것도(많은 경우 현명치 못한 일이지만) 가능합니다. 거의 모든 운영체제 커널이 시그널 핸들러와 비슷한 인터럽트 핸들러 내에서 락을 잡을 수 있게 합니다.

여기서의 트릭은 시그널(또는 인터럽트) 핸들러 내에서 잡힐 수 있는 어떤 락을 잡을 때마다 시그널을 블록(또는 경우에 따라 인터럽트를 불능화)시키는 것입니다. 더 나아가서, 그런 락을 잡을 때에는 시그널을 블록시키지 않고서 시그널 핸들러의 바깥에서 잡힌 어떤 락도 잡으려 시도해선 안됩니다.

Quick Quiz 7.10: 락 A가 어떤 시그널 핸들러의 내부에선 결코 잡히지 않았지만, 락 B는 쓰레드 컨텍스트에서도 시그널 핸들러에서도 잡혔다고 해봅시다. 나아가서 락 A는 가끔 시그널이 블록되지 않은 상태에서도 잡힌다고 해봅시다. 락 B를 잡고 있는 사이에 락 A를 획득하는 건 왜 불법인가요?

■ 어떤 락이 여러 시그널을 위한 핸들러에 의해 획득된다면, 해당 락이 하나의 시그널 핸들러에 의해서 획득될 때 조차도 해당 락이 획득될 때마다 이 시그널들은 모두 블록되어야만 할 겁니다.

Quick Quiz 7.11: 시그널 핸들러 내에서 어떻게 시그널들을 합법적으로 블록할 수 있죠?

■ 불행히도, 시그널을 블록하고 블록 해제하는 것은 리눅스를 포함한 일부 운영체제에선 값비싼 행위이므로, 성능에 대한 염려는 시그널 핸들러 내에서 획득되는 락들은 시그널 핸들러에서만 획득되며, 어플리케이션 코드와 시그널 핸들러 사이에서 통신하는 데에는 lockless 동기화 메커니즘이 사용되어야 함을 의미합니다.

또는 시그널 핸들러는 치명적 에러를 처리하는 경우를 제외하고는 완전히 사용되지 않을 수도 있겠습니다.

Quick Quiz 7.12: 시그널 핸들러 내에서 락을 잡는 게 그렇게 나쁜 생각이라면, 그걸 안전하게 하는 방법을 이야기 하는 이유는 뭐죠?

Listing 7.5: Abusing Conditional Locking

```

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 }
```

7.1.1.9 Discussion

공유메모리 병렬 프로그래머가 사용할 수 있는 굉장히 많은 deadlock 회피 기법들이 존재합니다만, 그것들 중 어느 것도 잘 들어맞지 않는 순차적 프로그램들이 있습니다. 이는 전문가 프로그래머들이 그들의 도구상자에 여러개의 도구를 가지고 다니는 이유입니다: 락킹은 강력한 동시성 도구이지만, 다른 도구들로 처리되는 게 나은 일들도 있습니다.

Quick Quiz 7.13: 간단한 락킹 계층, 레이어나 다른 것들이 존재하지 않게끔 제어를 객체 그룹 간에 자유로이 넘겨대는 객체 지향 어플리케이션에서³는 이 어플리케이션을 어떻게 병렬화 시킬 수 있을까요?

■ 그러나, 이 섹션에서 이야기된 전법들은 많은 상황에서 유용함으로 증명되었습니다.

7.1.2 Livelock and Starvation

조건적 락킹이 효과적인 deadlock 회피 메커니즘이 될 수 있지만, 오용될 수 있습니다. 예를 들어 Listing 7.5에 보인 아름답도록 대칭적인 예를 예로 들어 봅시다. 이 예의 아름다움은 추악한 livelock을 감춥니다. 이를 보기 위해, 아래와 같은 이벤트들의 순서를 생각해 봅시다:

1. 쓰레드 1이 라인 4에서 `lock1`을 잡고, `do_one_thing()`을 호출합니다.

³ “객체 지향 스파게티 코드”라고도 알려져 있습니다.

2. 쓰레드 2 가 라인 18 에서 lock2 를 잡고 do_a_third_thing() 을 호출합니다.
3. 쓰레드 1 이 라인 6 에서 lock2 를 잡으려 하지만, 쓰레드 2 가 이를 잡고 있으므로 실패합니다.
4. 쓰레드 2 가 라인 20 에서 lock1 을 잡으려 하지만, 쓰레드 1 이 이를 잡고 있으므로 실패합니다.
5. 쓰레드 1 이 라인 7 에서 lock1 을 놓고, 라인 3 의 retry 로 점프합니다.
6. 쓰레드 2 가 라인 21 에서 lock2 를 내려놓고, 라인 17 의 retry 로 점프합니다.
7. 이 livelock 이 처음부터 반복됩니다.

Quick Quiz 7.14: Listing 7.5 에 보인 livelock 은 어떻게 회피될 수 있나요?

Livelock 은 한 그룹의 쓰레드 중 하나가 아니라 전체가 기아에 빠지는 극단적 형태의 starvation 이라 할 수 있습니다.⁴

Livelock 과 starvation 은 소프트웨어 트랜잭션을 메모리 구현의 심각한 문제이며, 이 문제를 캡슐화하기 위해 컨텐션 매니저의 개념이 도입되었습니다. 락킹의 경우, 간단한 exponential backoff 는 livelock 과 starvation 을 종종 해결할 수 있습니다. 아이디어는 Listing 7.6 에 보인 것과 같이 각 채시도 사이의 딜레이를 지수적으로 증가시키는 겁니다.

Quick Quiz 7.15: Listing 7.6 의 코드에는 어떤 문제들이 있나요?

더 나은 결과를 위해, backoff 는 제한되어야 하며, 이보다도 나은 높은 컨텐션에서의 결과는 Section 7.3.2 에서 더 이야기 되는 queued locking [And90] 을 통해 얻어질 수 있습니다. 물론, 최고의 방법은 낮은 락 컨텐션을 유지함으로써 이런 문제들을 방지하는 좋은 병렬 설계를 사용하는 것입니다.

7.1.3 Unfairness

Unfairness 는 쓰레드들 중 특정 락을 가지고 경쟁하는 일부가 획득의 특권을 갖는, 덜 심각한 형태의 starvation 으로 생각될 수 있습니다. 이는 공유된 캐시나 NUMA 특성을 갖는 기계에서 일어날 수 있는데, 예를 들면 Figure 7.8 와 같습니다. CPU 0 이 모든 다른 CPU 들이 획득하고자 하는 락을 놓으면, CPU 0 과 1 사이에 공유된 연결부는

⁴ Livelock, starvation, 그리고 unfairness 같은 용어들의 정확한 정의에 너무 매달려 있지는 마세요. 어떤 그룹의 쓰레드들이 적절한 진행을 내는 것을 막는 모든 것은 고쳐져야 하는 버그이며, 논쟁적인 이름들은 버그를 고치지 않습니다.

Listing 7.6: Conditional Locking and Exponential Backoff

```

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 }
```

CPU 1 이 CPU 2-7 보다 이득을 얻을 것을 의미합니다. 따라서 CPU 1 은 락을 획득할 가능성이 높습니다. CPU 0 이 이 락을 다시 잡으려 할만큼 충분히 CPU 1 이 이 락을 길게 소유한다면, 그리고 그 반대의 경우에도, 이 락은 CPU 2-7 을 제끼고 CPU 0 과 1 사이에서 돌아다닐 겁니다.

Quick Quiz 7.16: Unfairness 를 회피할 정도로 락 경쟁을 낮게 유지하는 좋은 병렬 설계를 사용하는게 낫지 않을까요?

7.1.4 Inefficiency

락은 어토믹 인스트럭션과 메모리 배리어를 사용해 구현되며 종종 캐시 미스를 일으킵니다. Chapter 3 에서 보았듯, 이 인스트럭션들은 상당히 미용이 높은데, 간단한 인스트럭션들보다 대략 수백배 높은 오버헤드를 갖습니다. 이는 락킹에 있어 중요한 문제가 될 수 있습니다: 여러분이 하나의 인스트럭션을 락으로 보호한다면, 여러분은 이 오버헤드를 백배 가량 증가시키게 됩니다. 완벽한 확장성을 가정한다 해도, 같은 코드를 락킹 없이 수행하는 하나의 CPU 의 성능을 내려면 백개의 CPU 가 필요할 겁니다.

이 상황은 Section 6.3, 특히 Figure 6.16 에서 설명한 동기화-granularity 트레이드오프의 중요성을 강조합니다:

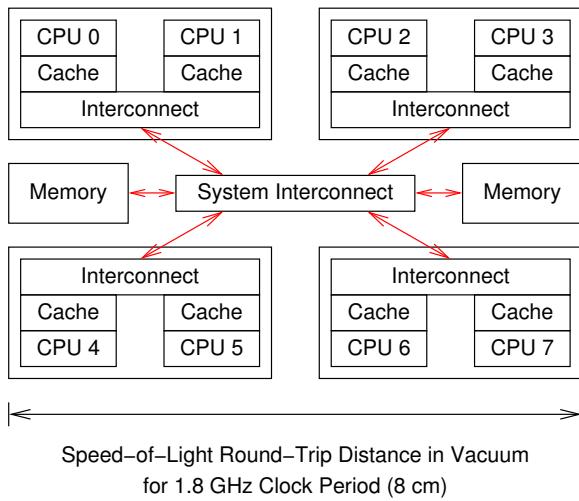


Figure 7.8: System Architecture and Lock Unfairness

너무 coarse 한 granularity 는 확장성을 제한하는 반면, 너무 fine 한 granularity 는 지나친 동기화 오버헤드를 초래합니다.

락을 잡는 건 높은 비용을 초래하지만, 일단 잡고 나면, 이 CPU의 캐쉬는 적어도 큰 크리티컬 섹션에 대해서는 효과적인 성능 부스터가 됩니다. 또한, 일단 락을 잡고 나면, 이 락에 의해 보호되는 데이터는 다른 쓰레드로부터의 간섭 없이 이 락을 잡은 쓰레드에 의해 접근될 수 있습니다.

Quick Quiz 7.17: 락을 잡은 쓰레드는 어떻게 간섭 받을 수 있나요?

■

7.2 Types of Locks

Only locks in life are what you think you know, but don't. Accept your ignorance and try something new.

Dennis Vickers

놀랄만큼 많은 수의 락 타입이 존재하는데, 이 짧은 챕터가 담을 수 있는 정도보다 큽니다. 다음 섹션들은 배타적 락 (Section 7.2.1), reader-writer 락 (Section 7.2.2), multi-role 락 (Section 7.2.3), 그리고 scoped 락킹 (Section 7.2.4)을 이야기 합니다.

7.2.1 Exclusive Locks

배타적 락은 말 그대로입니다: 한번에 하나의 쓰레드만이 이 락을 잡을 수 있습니다. 따라서, 이런 락을 잡은

쓰레드는 해당 락에 의해 보호되는 모든 데이터로의 배타적 접근 권한을 가지며, 그런 이유로 그런 이름이 붙었습니다.

물론, 이는 이 락이 이 락에 의해 보호되는 데이터에 대한 모든 접근 시에 잡힌다는 것을 가정하고 있습니다. 도움을 줄 수 있는 도구가 일부 있기는 하지만(예를 들어 Section 12.3.1을 참고하세요), 이 락이 항상 획득된다는 것을 보장하는 궁극적 책임은 개발자에게 있습니다.

Quick Quiz 7.18: 배타적 락을 잡자마자 곧바로 놔버리는, 즉 텅 빈 크리티컬 섹션 같은 것을 갖는 것은 말이 될까요?

■

무조건적으로 배타적 락을 획득하는 것이 두 가지 영향을 끼침을 알아두는 것이 중요합니다: (1) 해당 락을 앞서 잡은 모든 쓰레드가 그것을 놓기를 기다림, 그리고 (2) 이 락이 해제되기 전까지는 모든 이 락을 잡으려는 시도가 블록됨. 그 결과, 락 획득 시점에서, 모든 동시의 해당 락에 대한 획득은 앞서 락을 잡고 있던 쓰레드와 뒤따라 락을 잡는 쓰레드로 분리됩니다. 다른 종류의 배타적 락은 다른 분리 전략을 사용하는데 [Bra11, GGL⁺19], 예를 들면:

1. 락을 먼저 획득하고자 한 쓰레드가 먼저 락을 획득하는 엄격한 FIFO.
2. 충분히 먼저 락을 획득하고자 한 쓰레드가 먼저 락을 획득하는 대략적 FIFO.
3. 비슷한 시점에 락을 획득하려 시도하는 어떤 낮은 우선순위 쓰레드보다 더 높은 우선순위를 갖고 일찍 락을 획득하고자 한 쓰레드가 먼저 락을 획득하게 되는, 하지만 같은 우선순위의 쓰레드 사이에는 약간의 FIFO 순서 규칙이 적용되는 우선순위 단계 기반 FIFO.
4. 새로운 락 획득 쓰레드는 타이밍에 관계 없이 해당 락을 획득하려는 쓰레드 중 하나가 무작위로 뽑혀지는 무작위 방식.
5. 특정 락 획득 시도가 결코 락을 획득하지 못하게 될 수도 있는 unfair 방식 (see Section 7.1.3).

불행히도, 더 강력한 보장을 제공하는 락킹 구현은 일반적으로 더 높은 오버헤드를 일으켜서, 제품들에서 사용되는 다양한 락킹 구현의 모티베이션이 되었습니다. 예를 들어, 리얼타임 시스템은 종종 다른 것들보다도 우선순위 단계를 갖는 어느정도의 FIFO 순서 규칙을 필요로 하는 반면 (Section 14.3.5.1를 참고하세요), 높은 컨텐션을 갖는 비 리얼타임 시스템은 starvation을 피하기 충분한 순서규칙만을 필요로 할 수 있으며, 마지막으로, 컨텐션을 방지하게끔 설계된 비 리얼타임 시스템은 fairness 자체가 필요하지 않을 수도 있습니다.

7.2.2 Reader-Writer Locks

Reader-writer 락 [CHP71]은 여러 읽기 쓰레드가 동시에 락을 쥐고 있거나 하나의 쓰기 쓰레드만이 이 락을 쥐고 있을 수 있는 것을 허용합니다. 그러면, 이론상으로 reader-writer 락은 종종 읽히고 가끔만 쓰여지는 데이터에 대해 훌륭한 확장성을 허용해야만 합니다. 실전에서는, 이 확장성은 이 reader-writer 락 구현에 의존적입니다.

고전적인 reader-writer 락 구현은 원자적으로 조정되는 카운터와 플래그 집합을 사용합니다. 이런 종류의 구현은 짧은 크리티컬 섹션을 갖는 배타적 락킹과 같은 문제로 고통받게 됩니다: 이 락을 획득하고 해제하는 오버헤드가 간단한 인스트럭션 하나의 오버헤드의 수 배배는 큽니다. 물론, 이 크리티컬 섹션이 충분히 길다면, 이 락을 획득하고 해제하는데 드는 오버헤드는 무시할 수 있게 됩니다. 하지만, 한번에 하나의 쓰레드만이 이 락을 조정할 수 있으므로, 필요한 크리티컬 섹션 크기는 CPU 수에 비례해 늘어납니다.

쓰레드별 배타적 락을 사용해 훨씬 읽기 쓰레드에게 우호적인 reader-writer 락을 설계하는 것도 가능합니다 [HW92]. 읽기를 하려면, 자신의 락만을 획득합니다. 쓰기를 하려면, 모든 락을 획득합니다. 쓰기 쓰레드가 없을 때에는, 각 읽기 쓰레드는 어토믹 인스트럭션과 메모리 배리어 오버헤드만을 일으키며, 캐쉬 미스도 일으키지 않아서, 락킹 기능에게는 상당히 좋습니다. 불행히도, 쓰기 쓰레드는 어토믹 인스트럭션과 메모리 배리어 오버헤드에 대해 캐쉬 미스를 일으켜야만 합니다—여기에는 쓰레드의 수를 곱한 만큼의 오버헤드.

짧게 요약해서, reader-writer 락은 여러 상황에서 상당히 유용할 수 있습니다만, 각각의 구현의 종류들은 각자의 단점을 갖습니다. Reader-writer 락킹의 혼란 사용 예는 매우 긴 읽기 쪽 크리티컬 섹션을 사용하는데, 수백 마이크로세컨드에서 심지어 밀리세컨드까지 되는 길이가 선호됩니다.

배타적 락에서와 같이, reader-writer 락 획득은 모든 앞의 충돌하는 해당 락을 쥐고 있던 쓰레드가 이를 해제하기 전까지는 완료될 수 없습니다. 만약 어떤 락이 읽기 모드로 잡히면, 읽기 모드 획득은 곧바로 성공할 수 있습니다만, 쓰기 모드 획득은 더이상 이 락을 읽기 모드로 잡고 있는 쓰레드가 없을 때까지 기다려야만 합니다. 락이 쓰기 모드로 잡혀 있다면, 모든 락 획득 시도는 이 쓰기 쓰레드가 이 락을 해제하기 전까지 기다려야만 합니다. 역시 배타적 락에서와 마찬가지로, 다른 reader-writer 락 구현은 읽기 쓰레드와 쓰기 쓰레드에게 다른 수준의 FIFO 순서 규칙을 제공합니다.

하지만 많은 수의 읽기 쓰레드가 이 락을 잡고 있고 한 쓰기 쓰레드가 이 락을 잡기 위해 기다리고 있다고 생각해 봅시다. 쓰기 쓰레드가 starvation에 빠질 가능성이 있는데도 읽기 쓰레드들은 이 락을 잡고 있는 걸

계속하도록 허용해야 할까요? 비슷하게, 한 쓰기 쓰레드가 이 락을 잡고 있고 많은 수의 읽기 쓰레드와 쓰기 쓰레드가 이 락을 잡으려 기다리고 있다고 해봅시다. 현재의 쓰기 쓰레드가 이 락을 해제할 때, 이 락은 읽기 쓰레드와 다른 쓰기 쓰레드 중 누구에게 주어져야 할까요? 이게 읽기 쓰레드에게 주어진다면, 얼마나 많은 읽기 쓰레드가 다음 쓰기 쓰레드가 이 락을 잡기 전에 이 락을 잡도록 허용해야 할까요?

이 질문들에는 다른 수준의 복잡도, 오버헤드, 그리고 fairness를 갖는 많은 답변이 가능합니다. 다른 구현은 다른 비용을 가질 수도 있는데, 예를 들어 어떤 종류의 reader-writer 락은 읽기 모드로 잡힌 상태에서 쓰기 모드로 잡히는 변경 과정에서 엄청나게 큰 지연시간을 일으킵니다. 여기 몇 가지 가능한 방법들이 있습니다:

1. 읽기 쓰레드를 선호하는 구현은 무조건적으로 쓰기 쓰레드보다 읽기 쓰레드를 선호해서, 쓰기 모드 락 획득을 무한정 블록되어 있게 할 수도 있습니다.
2. Batch-fair 구현은 읽기 쓰레드와 쓰기 쓰레드가 모두 이 락을 획득하려 할 때, 둘 다 batching을 통해 합리적인 액세스를 갖습니다. 예를 들어, 이 락은 CPU 당 다섯개의 읽기 쓰레드, 이어서 두개의 쓰기 쓰레드, 다음엔 CPU 당 다섯개의 추가적 읽기 쓰레드, 이런 식으로 주어질 수 있습니다.
3. 쓰기 쓰레드를 선호하는 구현은 무제한적으로 읽기 쓰레드보다 쓰기 쓰레드를 선호해서, 읽기 모드 락 획득을 무한정 블록되어 있게 할 수도 있습니다.

물론, 이 차이는 높은 락 컨텐션 조건 아래에서만 문제됩니다.

락의 대기/블록이라는 본성을 항상 마음에 새겨두시기 바랍니다. 이는 Chapter 9의 확장성 있는 고성능의 특수 목적 락킹 대체제들에 대한 토론에서 다시 이야기될 겁니다.

7.2.3 Beyond Reader-Writer Locks

Reader-writer 락과 배타적 락은 허용 정책에 있어 다른 것을 갖습니다: 배타적 락은 최대 하나의 쓰레드만 락을 잡는 것을 허용하는 반면, reader-writer 락은 읽기 모드로 얼마든지 되는 수의 쓰레드가 락을 잡는 것을 (하지만 쓰기 모드로는 하나의 쓰레드만) 허용합니다. 매우 많은 허용 정책이 존재할 수 있는데, 그 중 하나는 Table 7.1에 보인 VAX/VMS 분산 락 매니저 (distributed lock manager: DLM)입니다. 비어있는 셀은 호환 가능한 모드를 나타내며, “X”로 채워진 셀은 비호환 모드를 의미합니다.

VAX/VMS DLM은 여섯개의 모드를 사용합니다. 비교의 목적을 위해 보자면, 배타적 락은 두개의 모드를

Table 7.1: VAX/VMS Distributed Lock Manager Policy

	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

(획득되지 않았음과 획득되었음), reader-writer 락은 세 개의 모드를(획득되지 않았음, 읽기 모드로 획득되었음, 그리고 쓰기 모드로 획득되었음) 사용합니다.

첫번째 모드는 null, 또는 쥐어지지 않았음입니다. 이 모드는 모든 다른 모드와 호환 가능합니다, 이는 다음과 같은 예상을 갖습니다: 만약 한 쓰레드가 어떤 락을 쥐고 있지 않다면, 이는 다른 쓰레드가 해당 락을 획득하는 것을 막지 않아야 합니다.

두번째 모드는 동시 읽기로, 배타 모드를 제외한 모든 다른 모드와 호환 가능합니다. 이 동시 읽기 모드는 어떤 데이터 구조에 동시의 업데이트를 허용하면서 대략적 통계를 내기 위해 사용될 수 있습니다.

세번째 모드는 동시 쓰기로, null, 동시 읽기, 그리고 동시 쓰기 모드와 호환 가능합니다. 이 동시 쓰기 모드는 동시의 읽기와 업데이트가 계속 이뤄지게 허용하면서도 대략적 통계를 업데이트 하기 위해 사용될 수 있습니다.

네번째 모드는 보호된 읽기로, null, 동시 읽기, 그리고 보호 읽기 모드와 호환 가능합니다. 이 보호된 읽기 모드는 어떤 데이터 구조에 동시의 읽기는 허용하면서도 동시의 업데이트는 금지하면서 일관적인 스냅샷을 얻기 위해 사용될 수 있습니다.

다섯번째 모드는 보호된 쓰기로, null 과 동시 읽기 모드와 호환됩니다. 이 보호된 쓰기 모드는 보호된 읽기를 간섭할 수 있지만 동시 읽기에는 문제를 일으키지 않는 데이터 구조에 업데이트를 하기 위해 사용될 수 있습니다.

여섯번째 마지막 모드는 배타 모드로, null 과만 호환됩니다. 이 배타 모드는 모든 다른 액세스를 배타시켜야 할 때 사용될 수 있습니다.

배타적 락과 reader-writer 락은 VAX/VMS DLM에 의해 애뮬레이션 될 수 있다는 점은 흥미롭습니다. 배타적 락은 null 과 배타 모드만 사용할 것이며, reader-writer

락은 null, 보호 읽기, 그리고 보호 쓰기 모드만을 사용할 겁니다.

Quick Quiz 7.19: VAX/VMS DLM이 reader-writer 락을 애뮬레이션 하는 다른 방법도 있을까요?

VAX/VMS DLM 정책이 분산 데이터베이스를 위한 광범위한 제품군에서의 사용을 보이긴 했지만, 공유 메모리 어플리케이션에서는 그만큼 널리 쓰이지 않았습니다. 이에 대한 그럴싸한 한가지 이유는 분산 데이터베이스의 거대한 통신 오버헤드가 VAX/VMS DLM의 더 복잡한 허용 정책의 큰 오버헤드를 가릴 수 있다는 것입니다.

그러나, VAX/VMS DLM은 락킹 뒤의 컨셉이 얼마나 유연할 수 있는지를 보여주는 흥미로운 경우입니다. 이는 또한 VAX/VMS의 여섯 모드에 비하면 훨씬 많은 30개가 넘는 락킹 모드를 가지기도 하는, 현대의 DBMS들에 의해 사용되는, 락킹에 대한 매우 간단한 소개이기도 합니다.

7.2.4 Scoped Locking

지금까지 설명해온 락킹 기능들은 획득과 해제 기능, 예를 들면 `spin_lock()`과 `spin_unlock()` 같은 것을 필요로 합니다. 다른 방법은 객체 지향 “resource allocation is initialization” (RAII) 패턴 [ES90]입니다.⁵ 이 패턴은 C++ 같은 언어의 auto 변수들에 종종 적용되는데, 연관된 *constructor* 가 이 객체의 *scope* 진입 시에 호출되고, 연관된 *destructor* 가 이 *scope* 을 빠져나올 때 호출됩니다. 이는 이 *constructor* 가 락을 획득하고 *destructor* 가 이를 해제하는 식으로 락킹에도 적용될 수 있습니다.

이 방법은 상당히 유용할 수 있으며, 실제로 저는 1990년에 이것이야말로 필요한 유일한 락킹 타입이라고 생각했습니다.⁶ RAII 락킹의 한가지 매우 좋은 특성은 여러분이 이 *scope* 을 빠져나가는 모든 코드 경로에서 조심스레 락을 해제할 필요가 없다는 것으로, 이는 문제있는 여러 버그들을 제거할 수 있습니다.

하지만, RAII는 어두운 부분도 존재합니다. RAII는 락 획득과 해제를 캡슐화 하는 것을 상당히 어렵게 하는데, 예를 들면 *iterator*에서 그렇습니다. 많은 *iterator* 구현에서, 여러분은 이 락을 이 *iterator*의 “시작” 함수에서 잡고 “종료” 함수에서 해제하고 싶을 겁니다. RAII 락킹은 그러는 대신 이 락의 획득과 해제가 어떤 *scope*의 단계에서 이뤄질 것을 필요로 해서, 그런 캡슐화를 어렵거나 심지어 불가능하게 합니다.

엄격한 RAII 락킹은 또한 크리티컬 섹션들을 겹치는 것을 불가능하게 하는데, *scope* 들이 중첩되어야 하기

⁵ https://www.stroustrup.com/bs_faq2.html#finally에 더 분명히 설명되어 있긴 합니다.

⁶ Sequent Computer Systems에서의 나중의 병렬성에 대한 제일은 매우 빠르게 이게 잘못된 방향이었음을 알려주었습니다.

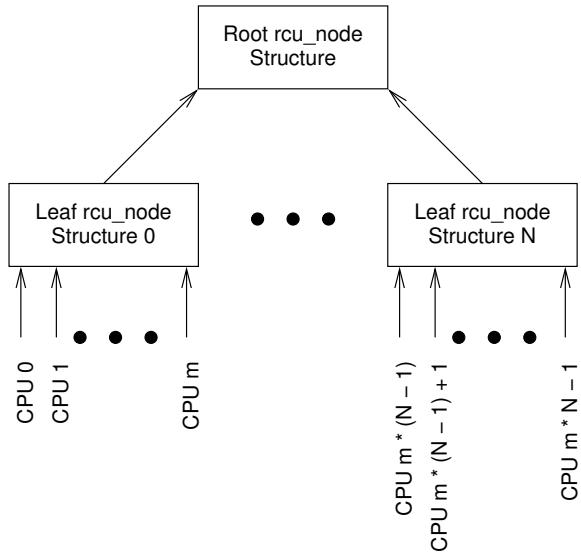


Figure 7.9: Locking Hierarchy

때문입니다. 이는 여러 유용한 구조를 어렵거나 불가능하게 만드는데, 예를 들면 어떤 이벤트를 동시에 단정하기 위한 여러 시도들 사이의 조정을 위한 락킹 트리 같은 것입니다. 많은 그룹의 동시적 시도들 가운데 하나만이 성공해야 하는데, 나머지 시도들에 대한 최고의 전략은 최대한 빠르고 고통 없이 실패하는 것입니다. 그러지 않으면, 락 컨텐션은 거대한 시스템에서는 (“거대함”이란 수백개의 CPU를 의미합니다) 병적이 될 것입니다. 따라서, C++17 [Smi19]은 `unique_lock` 클래스에서는 엄격한 RAII로부터 벗어나는데, 크리티컬 섹션의 `scope`이 명시적 락 획득과 해제 기능에 의해 얻어지는 정도와 거의 똑같은 제어를 가능하게 합니다.

리눅스 커널 RCU의 엄격한 RAII에 친화적이지 않은 데이터 구조의 예가 Figure 7.9에 보여져 있습니다. 여기서, 각 CPU는 `leaf rcu_node` 구조체를 할당받으며, 각 `rcu_node` 구조체는 각자의 부모 (`->parent`라 이름지어진)로의 포인터를 가지며, 이 구조는 `NULL ->parent` 포인터를 갖는 루트 `rcu_node` 구조체까지 이어집니다. 부모당 자식 `rcu_node` 구조체의 갯수는 다양할 수 있는데, 보통은 32 또는 64입니다. 각 `rcu_node` 구조체는 또한 `->fqsllock`이라 이름지어진 락을 갖습니다.

일반적인 접근법은 각 CPU가 반복적으로 자신의 `leaf rcu_node` 구조체의 `->fqsllock`을 획득하고, 거기 성공하면 그 부모를 획득하려 시도하며, 그 후에는 자식의 락을 내려놓는 겁니다. 또한, 각 레벨마다, CPU는 전역 `gp_flags` 변수를 검사하고, 이 변수가 어떤 다른 CPU가 이 이벤트를 단정했음을 알리며, 이 첫 번째 CPU는 이 경쟁에서 빠집니다. 이 획득하고 해제하기 흐름은 `gp_flags` 변수가 누군가가 이 토너먼트에서 승리했음을

Listing 7.7: Conditional Locking to Reduce Contention

```

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 = (READ_ONCE(gp_flags)) ||
9               !raw_spin_trylock(&rnp->fqsllock);
10        if (rnp_old != NULL)
11            raw_spin_unlock(&rnp_old->fqsllock);
12        if (ret)
13            return;
14        rnp_old = rnp;
15    }
16    if (!READ_ONCE(gp_flags)) {
17        WRITE_ONCE(gp_flags, 1);
18        do_force_quiescent_state();
19        WRITE_ONCE(gp_flags, 0);
20    }
21    raw_spin_unlock(&rnp_old->fqsllock);
22 }

```

알리거나, `->fqsllock` 획득 시도가 실패하거나, 또는 루트 `rcu_node` 구조체의 `->fqsllock` 획득이 성공했을 때 까지 계속됩니다. 루트 `rcu_node` 구조체의 `->fqsllock`이 획득되면, `do_force_quiescent_state()`라는 이름의 함수가 호출됩니다.

이를 구현하는 간략화 된 코드가 Listing 7.7에 보여져 있습니다. 이 함수의 목적은 `do_force_quiescent_state()` 함수를 호출해야 할 필요를 동시에 느끼는 CPU들 사이의 중재를 하는 것입니다. 언제든, `do_force_quiescent_state()`의 인스턴스 중 하나만이 액티브한 것만이 말이 되며, 따라서 여러 동시의 호출자가 있을 때, 우린 그것들 중 최대 하나만이 실제로 `do_force_quiescent_state()`를 호출하게 해야 하며, 나머지는 (최대한 고통 없고 빠르게) 포기하고 떠나야 합니다.

이 때문에, 라인 7-15의 루프의 각 패스는 `rcu_node` 구조의 한 단계 위로 넘어가려 시도합니다. 만약 `gp_flags` 변수가 이미 설정되었거나 (라인 8 현재 `rcu_node` 구조체의 `->fqsllock`을 획득하려는 시도가 실패했다면 (라인 9), 지역 변수 `ret` 가 1로 설정됩니다. 라인 10이 지역 변수 `rnp_old`가 `NULL`이 아님을, 즉 우리가 `rnp_old`의 `->fqsllock`을 쥐고 있음을 알게 된다면, 라인 11은 이 락을 놓습니다(하지만 부모 `rcu_node` 구조체의 `->fqsllock`을 획득하려는 시도가 만들어진 다음입니다). 라인 12이 라인 8 또는 9이 포기할 어떤 이유를 봤음을 알게 되면, 라인 13은 호출자에게 리턴합니다. 그렇지 않다면, 우린 현재 `rcu_node` 구조체의 `->fqsllock`을 획득했어야 하며, 따라서 라인 14는 이 루프의 다음 패스로의 이행 준비를 위해 이 구조체로의 포인터를 지역 변수 `rnp_old`에 저장합니다.

만약 컨트롤이 라인 16에 이르면, 우린 이 토너먼트에 승리한 것이며, 이제 루트 `rcu_node` 구조체의 `->`

fqslock 을 잡습니다. 라인 16 이 여전히 전역 변수 gp_flags 이 0임을 보게 된다면 라인 17 는 gp_flags 를 1로 설정하고, 라인 18 는 do_force_quiescent_state() 를 호출하며, 라인 19 은 gp_flags 를 0 으로 다시 리셋합니다. 어떤 경우이든, 라인 21 는 루트 rCU_node 구조체의 ->fqslock 을 해제합니다.

Quick Quiz 7.20: Listing 7.7 의 코드는 우습고 복잡합니다! 왜 하나의 전역 락을 조건적으로 획득하지 않죠?

Quick Quiz 7.21: 잠시만요! 우리가 Listing 7.7 의 라인 16 에서 이 토너먼트에 승리했다면 우린 do_force_quiescent_state() 의 일을 해야만 하게 됩니다. 이게 정확히 어떻게 승리라고 할 수 있나요?

이 함수는 계층적 락킹의 드물지 않은 패턴을 보입니다. 이 패턴은 엄격한 RAII 락킹을 이용해 구현하기 어려우며,⁷ 앞서 이야기 된 iterator 캡슐화도 마찬가지이고, 따라서 명시적 lock/unlock 도구들은 (또는 C++17 스타일의 unique_lock escape) 앞으로도 한동안 필요할 겁니다.

7.3 Locking Implementation Issues

When you translate a dream into reality, it's never a full implementation. It is easier to dream than to do.

Shai Agassi

개발자들은 예를 들면 POSIX pthread 뮤텍스 락 [Ope97] 시스템에서 제공되는 락킹 기능을 무엇이든 사용함으로써 거의 항상 최선의 봉사를 받습니다. 그러나, 극한의 워크로드와 환경에서 나오는 도전사항들을 고려하면서 예제 구현을 연구해 보는 것은 도움이 될 수 있습니다.

7.3.1 Sample Exclusive-Locking Implementation Based on Atomic Exchange

이 섹션은 listing 7.8 에 보인 구현을 리뷰해 봅니다. 이 락을 위한 데이터 구조는 라인 1 에 보인 것과 같이 그저 int 입니다만, 어떤 완전한 타입도 될 수 있습니다. 이 락의 초기 값은 라인 2에 보인 것과 같이 0 으로, “락되지 않음” 을 의미합니다.

⁷ 많은 RAII 락킹이 락을 획득된 scope 바깥, 그게 해제되었을 scope 으로 누출시키는 방법을 제공하는 이유입니다. 하지만, 일부 객체는 이 scope 누출을 중재해야만 하며, 이는 RAII 를 사용하지 않는 명시적 락킹 도구에 비해 복잡도를 더할 수 있습니다.

Listing 7.8: Sample Lock Based on Atomic Exchange

```

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

Quick Quiz 7.22: Listing 7.8 의 라인 2 에서 보인 명시적인 초기화를 사용하는 대신 C 언어의 기본 0 으로 초기화 기능을 사용하지 않나요?

락 획득은 라인 4-10 에 보인 xchg_lock() 함수에 의해 수행됩니다. 이 함수는 중첩된 반복문을 사용하는데, 바깥 반복문은 반복적으로 락의 값을 1 값으로 (“락되었음” 을 의미) 원자적 교환을 합니다. 만약 기존 값이 이미 1 이었다면 (달리 말하자면 다른 누군가가 이미 이 락을 잡았다면), 안쪽 반복문 (라인 7-8) 는 이 락이 획득 가능해질 때까지 수행되며, 이는 바깥 루프가 이 락을 획득하기 위한 또 다른 시도를 하는 시점이 됩니다.

Quick Quiz 7.23: Listing 7.8 의 라인 7-8 에 있는 안쪽 반복문을 왜 신경쓰나요? 그냥 라인 6 에서 원자적 교환 오퍼레이션을 반복하는게 어떤가요?

락 해제는 라인 12-15 에서 보인 xchg_unlock() 함수에 의해 수행됩니다. 라인 14 는 원자적으로 락의 값을 0 (“락이 잡히지 않음”) 으로 교환하며, 따라서 이를 해제되었다고 표시합니다.

Quick Quiz 7.24: Listing 7.8 의 라인 14 에서는 왜 간단히 이 락에 0 을 저장하지 않나요?

이 락은 test-and-set 락 [SR84] 의 간단한 예입니다만, 제품 단계에서 순수한 스피드으로써 많이 사용된 메커니즘과도 무척 비슷합니다.

7.3.2 Other Exclusive-Locking Implementations

어토믹 인스트럭션에 기반한 락킹 구현들은 무척 많이 존재하는데, 그 중 많은 것들은 Mellor-Crummey 와 Scott [MCS91] 의 고전 페이퍼에 의해 리뷰되었습니다. 이 구현들은 다차원 설계 트레이드오프 [GGL⁺19, Gui18, McK96b] 에서의 다른 지점들을 보입니다. 예를 들어, 앞의 섹션에서 본 어토믹 교환 기반의 test-and-set

락은 컨텐션이 낮을 때 잘 동작하며 작은 메모리 사용량의 장점을 갖습니다. 이 락은 그걸 사용할 수 없는 쓰레드에게 락을 주는 것을 방지합니다만, 그로 인해 높은 컨텐션 수준에서는 unfairness나 심지어 starvation을 겪을 수도 있습니다.

반면, 한때 리눅스 커널에서 사용되었던 티켓락 [MCS91]은 높은 컨텐션 수준에서도 unfairness를 방지합니다. 하지만, 그 엄격한 FIFO 규칙의 결과로, 이 락을 어쩌면 preemption 당했거나 interrupt 당해서 현재는 그걸 사용할 수도 없는 쓰레드에게 줄 수도 있습니다. 다른 한편, 이 preemption과 interrupt에 대해 너무 걱정하지는 않는게 중요합니다. 어쨌건, 많은 경우에 이 preemption과 interrupt는 이 락이 획득된 직후에도 일어날 수 있습니다.⁸

Test-and-set 락과 티켓락을 포함해 대기 쓰레드가 하나의 메모리 위치에서 기다리고 있어야 하는 모든 락킹 구현들은 높은 컨텐션 수준에서는 성능 문제로 고통받습니다. 문제는 락을 해제하는 쓰레드가 이 연관된 메모리 위치의 값을 업데이트 해야만 한다는 것입니다. 낮은 컨텐션에서는 이게 문제가 되지 않습니다: 연관된 캐시 라인은 여전히 로컬이고 이 락을 쥐고 있는 쓰레드에 의해 쓰여질 수 있을 가능성이 높습니다. 반면에, 높은 수준의 컨텐션에서는 이 락을 획득하려 하는 각 쓰레드가 이 캐시 라인의 read-only 복사본을 가지고 있을 것이고, 이 락을 쥐고 있는 쓰레드는 그런 모든 복사본들을 이 락을 해제하기 위한 업데이트를 진행하기 전에 무효화 시켜야 합니다. 일반적으로, 더 많은 CPU와 쓰레드가 존재할수록, 높은 컨텐션 조건 아래에서 락을 해제하려 할 때 발생하는 오버헤드가 거대해집니다.

이 음의 확장성은 여러 다른 queue 기반 락 구현의 모티베이션이 되었으며 [And90, GT90, MCS91, WKS94, Cra93, MLH94, TS93], 그 중 일부는 최신 버전의 리눅스 커널에서 사용되었습니다 [Cor14b]. Queue 기반 락은 높은 캐시 무효화 오버헤드를 각 쓰레드에게 queue 원소를 할당함으로써 방지합니다. 이 queue 원소들은 이 락이 대기 쓰레드들에게 돌아갈 순서를 조정하는 queue로 연결되어 있습니다. 핵심은 각 쓰레드가 각자의 queue 원소에서 스판을 하므로, 락을 쥐고 있는 쓰레드는 다음 쓰레드의 CPU의 캐시에 있는 queue 상 첫번째 원소만 무효화 하면 된다는 것입니다. 이 조정은 높은 수준의 컨텐션 아래에서 락 이동 오버헤드를 크게 줄여줍니다.

더 최근의 queue 기반 락 구현은 또한 시스템의 구조를 신경쓰는데, starvation을 회피하면서도 락을 지역적으로 얻는 것을 선호하는 것입니다 [SSVM02, RH03, RH02, JMRR02, MCM02]. 이 중 많은 것들이 전통적으

⁸ 별개로, 높은 락 컨텐션을 처리하는 최고의 방법은 그걸 방지해 버리는 겁니다! 하지만 높은 락 컨텐션은 피할 수 없는 상황도 존재하며, 높은 수준의 컨텐션에 대응하는 방법들을 연구하는 것은 좋은 정신적 훈련입니다.

로 disk I/O 스케줄링에 사용되었던 엘레베이터 알고리즘과 비슷한 것으로 생각될 수 있습니다.

불행히도, 높은 컨텐션 아래에서 queue 기반 락의 효율성을 개선하는 같은 스케줄링 로직이 낮은 컨텐션에서의 오버헤드도 증가시킵니다. 그래서 Beng-Hong Lim과 Anat Agarwal은 test-and-set 락을 낮은 컨텐션에서 사용하고 높은 컨텐션에서는 queue 기반 락으로 교체하는 방식으로 간단한 test-and-set 락을 queue 기반 락과 결합시켰는데, 따라서 낮은 단계의 컨텐션에서의 낮은 오버헤드와 높은 컨텐션에서의 높은 처리량과 fairness를 얻었습니다. Browning 등은 비슷한 시도를 했습니다만, 별도의 flag의 사용을 회피했으며, 따라서 test-and-set fast path는 간단한 test-and-set 락에서 사용되는 것과 같은 인스트럭션만을 사용하게 했습니다 [BMMM05]. 이 방법은 제품 단계에서 사용되었습니다.

높은 컨텐션에서 발생하는 또 다른 문제는 락을 쥐고 있는 쓰레드가 지연될 때로, 특히 이 지연이 preemption으로 인한 것일 때인데, 이는 우선순위 역전에 이를 수 있는데, 즉 낮은 우선순위 쓰레드가 락을 잡고, 중간 우선순위 CPU-bound 쓰레드에게 preemption 당하고, 결국 높은 우선순위 프로세스가 이 락을 잡으려 하는 동안 블록되어 있게 하는 것입니다. 이 결과는 CPU-bound 중간 우선순위 프로세스가 높은 우선순위 프로세스의 수행을 막는 것입니다. 한가지 해결책은 우선순위 계승 [LR80]으로, 이에 대한 일부 논란 [Yod04a, Loc02]에도 불구하고 리얼타임 컴퓨팅에 널리 사용되어 왔습니다 [SRL90, Cor06b].

우선순위 역전을 회피하는 또 다른 방법은 락이 잡혀 있는 동안의 preemption을 방지하는 것입니다. 락이 잡힌 동안 Preemption을 방지하는 것은 처리량 또한 개선시키므로, 대부분의 독점 UNIX 커널들은 어떤 종류의 스케줄러 친화적 동기화 메커니즘 [KWS97]을 제공하는데, 크게는 특정 규모 데이터베이스 벤더의 노력 덕분입니다. 이런 메커니즘은 보통 특정 코드 영역 내에서는 preemption이 금지되어야 함을 알리는 힌트의 형태를 취하는데, 이 힌트는 일반적으로 기계 레지스터에 위치하게 됩니다. 이런 힌트는 종종 특정 기계 레지스터의 한 bit을 설정하는 형태를 취하는데, 이 메커니즘을 위한 극단적으로 낮은 락 획득별 오버헤드를 가능하게 합니다. 대조적으로, 리눅스는 이 힌트를 금지하고, 그 대신 futexes [FRK02, Mol06, Ros06, Dre11]라고 불리는 메커니즘으로부터 비슷한 결과를 얻습니다.

흥미롭게도, 어토믹 인스트럭션은 락을 구현하는데 엄격히 말해 필요하지 않습니다 [Dij65, Lam74]. 간단한 로드와 스토어에 기반한 락킹 구현을 둘러싼 문제들에 대한 훌륭한 설명은 Herlihy와 Shavit의 교재 [HS08, HSLS20]에서 찾을 수 있을 겁니다. 여기서 나온 핵심은 그런 구현은 현재로써 적은 실용적 어플리케이션을 가지고 있으나, 그것들에 대한 주의 깊은 연구는 재미있

고 깨달음을 주는 경험일 수 있다는 것입니다. 그러나, 아래에 설명된 한 가지 예외와 함께, 그런 연구는 독자 여러분의 둑으로 남겨져 있습니다.

Gamsa 등 [GKAS99, Section 5.3] 은 CPU 들 사이를 토큰이 순환하는 토큰 기반 메커니즘을 이야기 합니다. 이 토큰이 특정 CPU 에 도달했을 때, 이 CPU 는 이 토큰에 의해 보호되는 모든 것에 대한 배타적 액세스를 갖습니다. 토큰 기반 메커니즘을 구현하기 위한 여러 방법이 있을 수 있는데, 예를 들면:

1. CPU 별 플래그를 유지하는데 이 플래그는 처음에는 하나의 CPU 를 제외하곤 모두 0 으로 초기값이 지정되어 있습니다. 어떤 CPU 의 플래그가 0 이 아닐 때, 그 CPU 는 이 토큰을 갖습니다. 토큰을 가지고 하는 일이 끝나면, 이 CPU 는 자신의 플래그를 0 으로 설정하고 다음 CPU 의 플래그를 1 으로 (또는 뭐가 됐든 0 이 아닌 값으로) 설정합니다.
2. CPU 별 카운터를 유지하는데, N 이 이 시스템의 CPU 의 갯수라 할 때 0 부터 $N - 1$ 의 범위를 가질 거라 가정할 수 있는 연관된 CPU 의 숫자로 초기 값을 갖습니다. 한 CPU 의 카운터가 다음 CPU 의 것보다 큰 값을 가질 때 (카운터의 최대 값을 지정해 두고), 이 첫번째 CPU 는 이 토큰을 줍니다. 이 CPU 가 토큰을 가지고 할 일이 끝나면, 다음 CPU 의 카운터를 자신의 카운터보다 1 큰 값으로 설정합니다.

Quick Quiz 7.25: 카운터의 최대값이 정해져 있는데 어떻게 한 카운터가 다른 것보다 크다고 말할 수 있나요?

Quick Quiz 7.26: 뭐가 낫나요, 카운터 방법인가요 플래그 방법인가요?

이 락은 다른 CPU 들이 락을 사용하고 있지 않을 때 조차도 특정 CPU 가 락을 즉각 획득할 수 없다는 점에서 일반적이지 않습니다. 그 대신, 이 CPU 는 이 토큰이 스스로에게 도달하기를 기다려야만 합니다. CPU 들이 크리티컬 섹션에 주기적으로 액세스 해야 하지만 토큰 순환 비율의 변동성을 제어해야 할 때 유용합니다. Gamsa 등 [GKAS99] 은 read-copy update (Section 9.5 을 참고하세요) 의 변종을 만드는 데에 이를 이용했습니다만, 메모리 할당자에 의해 사용되는 CPU 별 캐쉬 비우기 [MS93], CPU 별 데이터 구조의 garbage collection, 또는 공유 저장소 (또는 대용량 저장소, 뭐가 됐던) 로의 CPU 별 데이터 비우기 같은 주기적 CPU 별 오퍼레이션의 보호에도 사용될 수 있습니다.

리눅스 커널은 현재 queue 기반 스피너 [Cor14b] 을 사용하고 있습니다만 다양한 컨텐션 수준에서도 좋은 성능을 제공하는 구현의 복잡도 때문에 그 경로는 항상 부드럽지는 않았습니다 [Mar18, Dea18]. 점점 많은

Listing 7.9: Per-Element Locking Without Existence Guarantees

```

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 }
```

사람들이 병렬 하드웨어와 증가하는 양의 병렬 코드에 익숙해져 가고 있으므로, 우린 더 많은 특수 목적 락킹 기능이 나타날 것을 계속 기대할 수 있는데, Guerraoi 등의 연구 [GGL⁺19, Gui18] 를 예로 들 수 있겠습니다. 그러나, 이 중요한 안전에 대한 팁을 주의 깊게 고려하셔야 합니다: 언제든 일반적으로 가능할 때에는 표준 동기화 기능을 사용하세요. 표준 동기화 기능의 스스로 만드는 것들 대비 큰 이점은 표준 기능들은 일반적으로 훨씬 버그가 적다는 것입니다.⁹

7.4 Lock-Based Existence Guarantees

Existence precedes and rules essence.

Jean-Paul Sartre

병렬 프로그래밍에서의 핵심 도전사항은 존재 보장 [GKAS99] 을 제공하여, 특정 객체로의 액세스 시도가 해당 액세스 시도 동안은 객체의 존재에 의존할 수 있게 하는 것입니다. 어떤 경우, 존재 보장은 묵시적입니다:

1. 기본 모듈의 전역 변수와 정적 로컬 변수들은 해당 어플리케이션이 수행되는 동안은 존재합니다.
2. 로드된 모듈의 전역 변수와 정적 로컬 변수는 이 모듈이 로드된 상태를 유지하는 동안은 존재합니다.
3. 모듈은 그것의 함수들이 활동 상태의 인스턴스를 하나라도 가지고 있는 동안은 로드된 상태를 유지합니다.

⁹ 그리고 그래요, 전 제 스스로 만든 동기화 기능을 공유한 적 있진 해요. 하지만, 여러분은 제 머리카락이 그런 일을 시작하기 전에 비해 훨씬 하얗게 센 걸 알 수 있을 겁니다. 우연일까요? 어쩌면요. 하지만 여러분은 정말로 여러분의 머리가 일찍 하얗게 세는 리스크를 지고 싶으십니까?

4. 특정 함수 인스턴스의 스택 상 변수들은 이 인스턴스가 리턴할 때까지는 존재합니다.
5. 여러분이 어떤 함수 내에서 수행 중이거나(직접적으로 또는 간접적으로) 해당 함수에서 호출되었다면, 이 함수는 활동 상태 인스턴스를 갖습니다.

이 묵시적 존재 보장은 간단합니다만, 묵시적 존재 보장에 관련된 버그는 실제로 일어날 수 있습니다.

Quick Quiz 7.27: 묵시적 존재 보장에 기대는게 어떻게 버그에 이를 수 있죠?

■ 하지만 더 흥미로운—그리고 문제가 있을 수 있는—보장은 힙 메모리와 연관됩니다: 동적으로 할당된 데이터 구조는 그것이 해제될 때까지 존재할 겁니다. 해결해야 하는 문제는 이 구조체의 해제를 같은 구조체로의 동시에 액세스와 동기화 시켜야 한다는 것입니다. 이를 위한 한가지 방법은 락킹과 같은 명시적 보장입니다. 주어진 구조체가 이 락을 잡은 상태에서만 해제된다면, 이 락을 잡는 것은 이 구조체의 존재를 보장합니다.

하지만 이 보장은 이 락 자체의 존재에 의존합니다. 이 락의 존재를 보장하는 한가지 간단한 방법은 이 락을 전역 변수에 두는 것입니다만, 전역 락킹은 확장성 제한의 단점을 갖습니다. 데이터 구조체의 크기의 증가에 따라 개선되는 확장성을 제공하는 한가지 방법은 이 구조체의 각 원소마다 락을 두는 것입니다. 불행히도, 데이터 원소를 보호해야 하는 락을 데이터 원소 자체에 두는 것은 Listing 7.9 에 보인 것과 같이 일부 레이스 컨디션을 일으킵니다.

Quick Quiz 7.28: 우리가 제거해야 하는 원소가 Listing 7.9 의 라인 8 에 있는 리스트의 첫번째 원소가 아니라면 어떻게 되는 걸까요?

■ 이런 레이스 컨디션들 중 하나를 자세히 보기 위해, 다음 이벤트들을 생각해 봅시다:

1. 쓰래드 0 이 `delete(0)` 를 호출하고, 이 listing 의 라인 10 에 도달, 락을 잡습니다.
2. 쓰래드 1 이 `delete(0)` 을 호출하고, 라인 10 에 도달하지만 쓰래드 0 이 이미 락을 잡고 있으므로 여기서 맴돕니다.
3. 쓰래드 0 이 라인 11-14 을 수행하고, 해쉬 테이블에서 원소를 제거하고, 락을 해제하고, 원소를 메모리 할당 해제합니다.
4. 쓰래드 0 이 수행을 계속해서 메모리를 할당받아 방금 해제한 바로 그 메모리 블록을 받습니다.
5. 쓰래드 0 은 이제 이 메모리 블록을 어떤 다른 타입의 구조체로 초기화 시킵니다.

Listing 7.10: Per-Element Locking With Lock-Based Existence Guarantees

```

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 }

```

6. 쓰래드 1 의 `spin_lock()` 오퍼레이션은 자신이 `p->lock` 이라 믿었던 게 더이상 스팬락이 아니게 되었으므로 실패합니다.

어떤 존재 보장이 없으므로, 이 데이터 원소의 정체성은 이 쓰래드가 이 원소의 락을 라인 10 에서 획득하려 하는 사이에 바뀔 수 있습니다!

이 예를 고치는 한가지 방법은 해시된 전역 락 집합을 사용해서 각 해시 버킷이 각자의 락을 가지게 하는 것으로, Listing 7.10 에 보여져 있습니다. 이 방법은 데이터 원소로의 포인터를 얻기 전에(라인 10) 올바른 락을 획득할 수 있게(라인 9) 해줍니다. 이 방법이 이 listing 에 보인 해시 테이블과 같이 단일 분할 가능 데이터 구조체에 포함되어 있는 원소들에는 상당히 잘 동작하지만, 주어진 데이터 원소가 여러 해시 테이블 또는 트리나 graph 같은 더 복잡한 데이터 구조체에서는 문제가 있을 수 있습니다. 해결책은 이 문제만 해결할 뿐만 아니라, 락 기반 소프트웨어 트랜잭션 메모리 구현의 기반을 형성합니다 [ST95, DSS06]. 하지만, Chapter 9 는 존재 보장을 제공하는 더 간단한—그리고 더 빠른—방법을 설명합니다.

7.5 Locking: Hero or Villain?

You either die a hero or live long enough to become the villain.

Aaron Eckhart

실제 삶에서도 종종 그렇듯, 락킹은 그게 실제 문제에 어떻게 사용되는가에 따라 영웅이 될수도 악당이 될수도 있습니다. 제 경험상, 전체 어플리케이션을 작성하는 사람들은 락킹으로도 행복하고, 병렬 라이브러리를 작

성하는 사람들은 덜 행복하며, 기존에 존재하던 순차적 라이브러리를 병렬화 하는 사람들은 무척 불행해 합니다. 다음 섹션은 이런 관점상의 차이의 어떤 이유들을 논합니다.

7.5.1 Locking For Applications: Hero!

어플리케이션 전체 (또는 커널 전체)를 작성할 때, 개발자는 동기화 설계를 포함해 설계에 대한 완전한 제어를 갖습니다. Chapter 6에서 이야기 되었듯 이 설계가 파티셔닝을 잘 하고 있다면, 락킹은 제품 수준 병렬 소프트웨어에서의 많은 사용이 보이듯 굉장히 효과적인 동기화 메커니즘이 될 수 있습니다.

그러나, 그런 소프트웨어가 일반적으로 도의화 설계의 기본을 락킹으로 하긴 하지만, 그런 소프트웨어는 또한 거의 항상 특수한 카운팅 알고리즘 (Chapter 5), 데이터 소유권 (Chapter 8), 레퍼런스 카운팅 (Section 9.2), 해저드 포인터 (Section 9.3), 시퀀스 락킹 (Section 9.4), and read-copy update (Section 9.5)를 포함한 다른 동기화 메커니즘도 사용합니다. 또한, 실제 사람들은 데드락 탐지 [Cor06a], 락 획득/해제 밸런스 [Cor04b], 캐쉬 미스 분석 [The11], 하드웨어 카운터 기반 프로파일링 [EGMDB11, The12b], 그 외에도 여러가지를 위한 도구를 사용합니다.

주의 깊은 설계라면, 동기화 메커니즘들, 좋은 도구, 락킹의 좋은 조합의 사용은 어플리케이션과 커널에 잘 동작합니다.

7.5.2 Locking For Parallel Libraries: Just Another Tool

어플리케이션과 커널과는 달리, 라이브러리의 설계자는 이 라이브러리가 상호작용할 코드의 락킹 디자인을 알 수 없습니다. 사실, 그 코드는 수년간 작성되었을 수도 있습니다. 따라서 라이브러리 설계자는 더 적은 제어를 가지고 있고 그들의 동기화 설계를 할 때 더 많은 주의를 해야 합니다.

데드락은 물론 일반적인 걱정사항이며, Section 7.1.1에서 이야기한 테크닉들이 적용되어야 합니다. 따라서 대중적인 데드락 회피 전략 한가지는 이 라이브러리의 락이 둘러싼 프로그램의 락킹 계층의 독립적 하위 트리임을 분명히 하는 것입니다. 하지만, 이는 보이는 것보다 어려울 수 있습니다.

한가지 복잡한 부분은 Section 7.1.1.2에서 이야기 되었는데, 라이브러리 함수의 어플리케이션 코드로의 호출로, `qsort()`의 비교 함수 인자가 그 케이스가 되었습니다. 또 다른 복잡한 부분은 시그널 핸들러와의 상호작용입니다. 어떤 어플리케이션 시그널 핸들러가 이 라이브러리 함수 내에서 받은 시그널로부터 호출되었다면 이 라이브러리 함수가 시그널 핸들러를 직접 호

출했다면 분명 데드락이 발생할 수 있습니다. 마지막 복잡한 부분은 `fork()`/`exec()` 사이에서 사용될 수 있는 라이브러리 함수에서 있을 수 있는데, 예를 들면 `system()` 함수의 사용 때문일 수 있습니다. 이 경우, 여러분의 라이브러리 함수가 `fork()` 시점에 락을 잡고 있다면, 자식 프로세스는 이 락이 잡힌 채 삶을 시작하게 됩니다. 이 락을 해제할 이 쓰레드는 자식이 아니라 부모에서 수행되고 있으므로, 이 자식이 여러분의 라이브러리 함수를 호출한다면, 데드락이 일어날 겁니다.

이런 경우 데드락 문제를 회피하기 위해 다음 전략들이 사용될 수 있습니다:

1. 콜백도 시그널도 사용하지 않기.
2. 콜백이나 시그널 핸들러에서 락을 잡지 않기.
3. 호출자가 동기화를 제어할 수 있게 하기.
4. 라이브러리 API 가 락킹을 호출자에게 넘기도록 패러미터를 제공하기.
5. 명시적으로 콜백 데드락을 방지하기.
6. 명시적으로 시그널 핸들러 데드락을 방지하기.
7. `fork()` 호출을 방지하기.

이 전략들 각각이 다음 섹션들에서 이야기 됩니다.

7.5.2.1 Use Neither Callbacks Nor Signals

어떤 라이브러리 함수가 콜백을 방지하고 어플리케이션은 시그널을 전체 방지한다면, 이 라이브러리 함수에 의한 모든 락 획득은 이 락킹 계층 트리의 잎이 됩니다. 이런 정리는 Section 7.1.1.1에서 이야기한 것처럼 데드락을 방지합니다. 이 전략은 적용될 수 있는 곳에서는 굉장히 잘 동작하지만, 시그널 핸들러를 사용해야만 하는 일부 어플리케이션이 존재하며, 콜백을 필요로 하는 라이브러리 함수도 (Section 7.1.1.2에서 이야기한 `qsort()` 같은) 존재합니다.

다음 섹션에서 이야기 될 전략은 이런 경우에 종종 사용될 수 있습니다.

7.5.2.2 Avoid Locking in Callbacks and Signal Handlers

콜백도 시그널 핸들러도 락을 획득하지 않는다면, 그 것들은 데드락 사이클에 관여될 수 없는데, 이는 다시 라이브러리 함수들이 락킹 계층 트리의 이파리가 되게 하는 간단한 락킹 계층을 가능하게 합니다. 이 전략은 콜백이 단순히 그들에게 넘겨진 두개의 값을 비교하기만 하는 대부분의 `qsort` 사용처에 잘 적용됩니다. 이

전략은 또한 많은 시그널 핸들러에도 놀랍도록 잘 동작하는데, 특히 시그널 핸들러 내에서 락을 획득하는게 일반적으로 눈살을 찌푸리게 하지만 [Gro01],¹⁰ 이 어플리케이션이 시그널 핸들러에서 복잡한 데이터 구조를 조정해야 한다면 실패할 수 있습니다.

복잡한 데이터 구조가 수정되어야 함에도 시그널 핸들러에서 락을 잡는 걸 막는 몇 가지 방법이 여기 있습니다:

1. Section 14.2.1에서 이야기 하겠지만 non-blocking 동기화에 기반한 간단한 데이터 구조를 사용합니다.
2. 이 데이터 구조가 합리적인 non-blocking 동기화의 사용에는 너무 복잡하다면, non-blocking enqueue 오퍼레이션을 허용하는 queue를 만듭니다. 시그널 핸들러에서는 이 복잡한 데이터 구조를 조정하는 대신 요구되는 변경을 설명하는 원소를 이 queue에 넣습니다. 그럼 별도의 쓰레드가 이 queue에서 원소를 제거하고 평범한 락킹을 사용해 필요한 변경을 진행할 수 있습니다. 당장 사용할 수 있는 동시적 queue의 구현들이 여럿 있습니다 [KLP12, Des09b, MS96].

이 전략은 때때로의 수동적인 또는 (선호되건대) 자동화된 콜백과 시그널 핸들러의 검사가 강제 되어야 합니다. 이 검사를 할 때에는, 어토믹 오퍼레이션을 가지고 집에서 스스로 만든 락을 (현명치 못하게) 사용하는 똑똑한 코더들을 주의해야 합니다.

7.5.2.3 Caller Controls Synchronization

호출자가 동기화를 제어할 수 있게 하는 것은 이 라이브러리 함수가 독립적이어서 각자 개별적으로 동기화될 수 있는, 호출자에게 보여지는 데이터 구조 인스턴스에 동작할 때 굉장히 잘 동작합니다. 예를 들어, 이 라이브러리 함수가 검색 트리에 동작할 때, 그리고 만약 이 어플리케이션이 커다란 수의 개별적 검색 트리를 필요로 한다면, 이 어플리케이션은 각 트리에 락을 연관지을 수 있습니다. 그러면 이 어플리케이션은 필요한 대로 락을 획득하고 해제할 수 있어서, 이 라이브러리는 병렬화에 전혀 신경을 쓰지 않아도 됩니다. 대신, 이 어플리케이션은 이 병렬성을 제어하며, 따라서 락킹은 매우 잘 동작할 수 있어서, Section 7.5.1에서 이야기한 것과 같이 됩니다.

하지만, 이 전략은 이 라이브러리가 내부의 동시성을 필요로 하는 데이터 구조에 적용될 때에는 실패할 수 있는데, 예를 들면 해시 테이블이나 병렬 정렬입니다. 이런

¹⁰ 하지만 이 표준의 말들은 영리한 코더들이 각자의 집에서 어토믹 오퍼레이션을 가지고 스스로 만든 락킹 기능을 사용하는 걸 막지는 않습니다.

경우, 이 라이브러리는 자신의 동기화를 직접 제어해야 합니다.

7.5.2.4 Parameterize Library Synchronization

여기서의 아이디어는 이 라이브러리의 API에 어떤 락을 획득해야 하며, 어떻게 획득하고 해제해야 하는지, 또는 둘다 명시할 수 있게끔 인자를 더하는 것입니다. 이 전략은 어플리케이션이 어떤 락을 획득하고 (해당 락으로의 포인터를 넘김으로써) 그것들을 어떻게 획득하는지 (락 획득과 해제 함수로의 포인터들을 넘김으로써) 명시함으로써 데드락을 회피하는 전체적 일을 맡게 하지만, 또한 해당 라이브러리 함수가 어디서 락이 획득되고 해제되어야 하는지 결정함으로써 자신의 동시성을 제어할 수 있게 합니다.

특히, 이 전략은 라이브러리 코드는 어떤 시그널이 어떤 락에 의해 블록될 수 있는지 걱정할 필요 없게 하면서 락 획득과 해제 함수가 시그널을 블록시킬 수 있게 해줍니다. 이 전략에 의해 사용되는 걱정의 분리는 상당히 효과적일 수 있습니다만, 어떤 경우에는 다음 섹션에 설명되는 전략들이 더 나을 수도 있습니다.

그러나, 락으로의 명시적인 포인터를 외부 API에 넘기는 것은 Section 7.1.1.4에서 설명되었듯 매우 조심스럽게 고려되어야 합니다. 이 방법이 때로는 해야 할 올바른 일이지만, 여러분은 대안적 설계를 먼저 알아봄으로써 스스로를 위해줘야 합니다.

7.5.2.5 Explicitly Avoid Callback Deadlocks

이 전략의 기본 규칙은 Section 7.1.1.2에서 이야기 되었습니다: “알려지지 않은 코드를 호출하기 전에 모든 락을 해제하라.” 이것은 어플리케이션이 라이브러리의 락킹 계층을 무시할 수 있게 하기 때문에 보통 최고의 방법입니다: 라이브러리는 어플리케이션의 전체 락킹 계층으로부터 고립된 서브트리의 이파리 레벨로 남게 됩니다.

알려지지 않은 코드를 호출하기 전에 모든 락을 해제하는게 불가능한 경우에는 Section 7.1.1.3에서 이야기한 레이어 기반 락킹 계층이 잘 동작합니다. 예를 들어, 알려지지 않은 코드가 시그널 핸들러라면, 이는 이 라이브러리 함수가 모든 락 획득에 걸쳐 시그널을 블록함을 암시하는데, 이는 복잡하고 느릴 수 있습니다. 따라서, 시그널 핸들러가 (아무래도 현명치 못하게) 락을 획득하는 경우, 다음 섹션의 전략이 도움이 될 수 있을 겁니다.

7.5.2.6 Explicitly Avoid Signal-Handler Deadlocks

어떤 라이브러리 함수가 락을 잡지만 시그널은 블록하지 않는다고 알려져 있다고 생각해 봅시다. 나아가서

이 함수를 시그널 핸들러의 안과 밖 모두에서 호출해야 하며, 이 라이브러리 함수를 고칠 수 없다고 생각해 봅시다. 물론, 어떤 특별한 행동이 취해지지 않는다면, 이 라이브러리 함수가 자신의 락을 잡고 있는 동안 어떤 시그널이 들어온다면 이 시그널 핸들러가 같은 락을 또 획득하려 할 같은 라이브러리 함수를 호출한다면 데드락이 발생할 수 있습니다.

그런 데드락은 다음과 같이 회피될 수 있습니다:

1. 이 어플리케이션이 이 라이브러리 함수를 시그널 핸들러 내에서 호출한다면, 해당 시그널은 이 함수가 시그널 핸들러 밖에서 호출될 때는 항상 시그널이 블록되어 있어야 합니다.
2. 이 어플리케이션이 이 라이브러리 함수를 해당 시그널 핸들러 내에서 획득된 락을 잡고 있는 동안 호출한다면, 이 시그널은 이 라이브러리 함수가 이 시그널 핸들러의 바깥에서 호출될 때마다 항상 블록되어 있어야 합니다.

이 규칙은 리눅스 커널의 lockdep 락 의존성 검사기 [Cor06a] 과 비슷한 도구를 사용해 강제될 수 있습니다. Lockdep의 위대한 강점 중 하나는 사람의 직감에 의한 실수를 저지르지 않는다는 것입니다 [Ros11].

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

앞에서도 언급되었듯, 라이브러리 함수를 수행시키는 어떤 쓰레드가 다른 같은 쓰레드가 `fork()` 를 호출하는 시점에 어떤 락을 잡고 있다면, 이 부모의 메모리가 자식을 생성하기 위해 복사된다는 사실은 이 락이 이 자식의 컨텍스트에는 처음부터 잡혀있을 것을 의미합니다. 이 락을 해제할 쓰레드는 부모에서는 돌아가고 있지만 이 자식에서는 그렇지 않은데, 이는 이 자식의 이 락의 복사본은 결코 해제되지 않을 것을 의미합니다. 따라서, 이 자식에서의 같은 라이브러리 하무를 호출하는 모든 시도는 데드락을 초래할 겁니다.

이 문제를 해결하는 실용적이고 간단한 방법은 이 프로세스가 여전히 싱글 쓰레드인 동안 자식 프로세스를 `fork()` 하고 이 자식 프로세스를 싱글 쓰레드인채로 남겨두는 것입니다. 그러면 더 많은 자식 프로세스를 생성하려는 요청은 이 최초 자식 프로세스에게 전달될 수 있는데, 이는 모든 필요한 `fork()` 와 `exec()` 시스템 콜을 멀티쓰레드로 돌아가는 부모 프로세스 대신 안전하게 이뤄질 수 있습니다.

이 문제를 위한 덜 실용적이고 덜 간단한 또 다른 해결책은 이 라이브러리 함수가 이 락의 소유자가 여전히 동작 중인지 확인하고, 그렇지 않다면 이를 재 초기화하고 획득함으로써 이 락을 “깨트리는” 겁니다. 하지만, 이 방법은 두가지 취약점이 있습니다:

1. 이 락에 의해 보호되는 데이터 구조는 어떤 중간적 상태에 있어서 낙관적으로 이 락을 깨트리는 것은 임의의 메모리 오염을 초래할 수도 있습니다.
2. 이 자식이 추가적인 쓰레드를 만든다면, 두개의 쓰레드는 이 락을 동시적으로 깨트릴 수 있는데, 그 결과는 두 쓰레드 모두 자신들이 락을 소유하고 있다고 믿게 되는 것입니다. 이는 역시 임의의 메모리 오염을 초래할 수 있습니다.

`pthread_atfork()` 함수는 이 상황을 해결하기 위해 제공됩니다. 아이디어는 세개의 함수를 등록시켜서, 하나는 `fork()` 전에 부모에 의해 호출되고, 하나는 부모에 의해 `fork()` 전에 호출되고, 마지막 하나는 자식에 의해 `fork()` 후에 호출되게 하는 것입니다. 그러면 적절한 정리가 이 세개의 지점에서 이뤄질 수 있습니다.

하지만, `pthread_atfork()` 핸들러를 코딩하는 것은 일반적으로 상당히 섬세함을 경고 드려 둡니다. `pthread_atfork()` 가 가장 잘 동작하는 경우는 해당 데이터 구조가 자식에 의해 간단히 재 초기화 될 수 있는 경우입니다.

7.5.2.8 Parallel Libraries: Discussion

사용된 전략에 관계 없이, 라이브러리의 API에 대한 설명은 그 전략에 대한, 그리고 호출자가 이 전략에 어떻게 상호 작용해야 하는지에 대한 분명한 설명을 포함해야만 합니다. 짧게 말해서, 락킹을 사용해서 병렬 라이브러리를 구성하는 것은 가능하지만, 병렬 어플리케이션을 구성하는 것만큼 쉽지는 않습니다.

7.5.3 Locking For Parallelizing Sequential Libraries: Villain!

사용 가능한 저가형 멀티코어 시스템의 발전으로 인해, 싱글 쓰레드 기반 사용을 염두에 두고 설계된 존재하는 라이브러리를 병렬화 시키는 것은 흔한 작업이 되었습니다. 이 병렬성과 관계 없이 너무도 흔한 일들은 병렬 프로그래밍 관점에서 보기에는 무척 결점이 많은 라이브러리 API를 초래할 수 있습니다. 그런 결점 후보에는 다음과 같은 것들이 포함됩니다:

1. 파티셔닝의 무시적 금지.
2. 락킹을 필요로 하는 콜백 함수들.
3. 객체 지향 스파게티 코드.

이 락킹에 대한 결점들과 그로 인한 결론들을 다음 섹션들에서 이야기 합니다.

7.5.3.1 Partitioning Prohibited

여러분이 싱글 쓰레드 기반 해쉬 테이블 구현을 작성하고 있었다고 해봅시다. 이 해시 테이블 내의 전체 아이템의 갯수를 유지하는 것은 쉽고 빠르며, 각 아이템 더하기와 제거하기 오퍼레이션 시에 이 정확한 갯수를 리턴하는 것 역시 쉽고 빠릅니다. 그러나 그러지 않을 이유가 뭐겠어요?

그 한가지 이유는 정확한 수치의 카운터는 Chapter 5에서 알아 봤듯이 멀티코어 시스템에서는 성능이 좋지도 않고 확장도 잘 되지 않기 때문입니다. 그 결과, 해쉬 테이블의 이 병렬화된 구현은 성능도 좋지 않고 확장도 잘 되지 않을 겁니다.

그럼 이에 대해 뭘 할 수 있을까요? 한가지 방법은 Chapter 5에 나온 알고리즘 중 하나를 사용해 근사치 카운터를 리턴하는 것입니다. 또 다른 방법은 항목 수 자체를 제공하지 않는 겁니다.

어떤 방법이든, 왜 원소 추가와 제거 오퍼레이션이 정확한 카운트를 필요로 하는지 해쉬 테이블의 사용처를 조사하는 것입니다. 여기 몇가지 가능성이 있습니다:

1. 언제 해쉬 테이블을 크기 조정할지 판단하기. 이 경우, 근사치 카운터는 무척 잘 동작할 겁니다. 잘 파티셔닝된 체인별 방법으로 계산되고 유지되는 가장 긴 체인의 길이에 기반해 크기 재조정을 하는 것도 유용할 수 있습니다.
2. 전체 해쉬 테이블을 순회하는데 걸리는 시간의 예측치를 만들기. 이 경우에도 근사치 카운트는 잘 동작합니다.
3. 분석 목적으로, 예를 들면 아이템을 해쉬테이블 사이에 이동시킬 때 없어진 아이템을 검사하기 위해. 이는 분명 정확한 카운트를 필요로 합니다. 하지만, 이 사용처가 분석이라는 기본에 비춰볼 때, 해쉬 체인의 길이만을 유지하고 추가와 제거 오퍼레이션을 락으로 막아둔 상태에서 가끔 그 값들의 합산을 구하는 것으로도 충분할 수도 있습니다.

병렬 라이브러리 API의 성능과 확장성에 존재하는 일부 제약들에 대한 강한 이론적 기본들이 이제 존재하는 것으로 드러났습니다 [AGH^{11a}, AGH^{11b}, McK11b]. 병렬 라이브러리를 설계하는 사람들은 모두 이 제약들에 주의를 기울여야 합니다.

사실은 동시성에 친화적이지 못한 API 때문인 문제들에 대해 락킹을 비난하는 것은 너무 쉽지만, 그러는게 도움이 되진 못합니다. 다른 한편, (말하자면) 1985년에 이 선택을 했던 불운한 개발자에 공감하는 것밖에는 별 다른 선택지가 없기도 합니다. 그 당시에 병렬성을 예상한 개발자는 드물고 용기 있는 경우였을 것이고, 실제로 좋은 병렬성 친화적 API를 만들어내기 위해선 종명성과 행운이라는 더욱 드문 조합이 필요했을 겁니다.

시간은 변합니다, 그리고 코드는 그와 함께 변화해야 합니다. 그러나, 대중적 라이브러리에는 거대한 수의 사용자들이 있을 수 있는데, 이 API로의 호환성을 해치는 변화는 상당히 명확한 일일 겁니다. 이미 존재하는 많이 사용되는 순차적으로만 동작하는 API를 보강하기 위해 병렬성에 친화적인 API를 추가하는 것은 일반적으로 최고의 행동입니다.

그러나, 사람의 본성이 그러한 만큼, 우리의 불운한 개발자가 그 또는 그녀의 스스로의 불쌍한 (이해할 수는 있지만) API 설계 선택에 대해 불만을 표시할 가능성이 높을 것으로 예상할 수 있습니다.

7.5.3.2 Deadlock-Prone Callbacks

Section 7.1.1.2, 7.1.1.3, 그리고 7.5.2는 규율 없는 콜백의 사용이 어떻게 락킹의 비애로 이어질 수 있는지 이야기했습니다. 이 섹션들은 또한 이런 문제들을 막기 위해 여러분의 라이브러리 함수를 어떻게 설계해야 하는지 이야기했습니다만, 병렬 프로그래밍 경험이 없는 1990년대의 프로그래머가 그런 설계를 따랐을 거라고 기대하는 것은 비현실적입니다. 따라서, 존재하는 콜백을 많이 사용하는 싱글쓰레드 기반 라이브러리를 병렬화 하려 하는 누군가는 락킹의 악마성을 저주할 가능성이 높을 겁니다.

매우 많은 콜백 기반 라이브러리의 사용이 있다면, 기준에 있던 사용자들이 그들의 코드를 점진적으로 변경할 수 있게끔 병렬성 친화적 API를 라이브러리에 추가하는게 현명할 수 있습니다. 대안적으로, 어떤 사람들은 이런 경우에 트랜잭션 메모리의 사용을 추천합니다. 배심원은 여전히 트랜잭션 메모리에 찬성하지 않지만, Section 17.2은 그것의 강점과 약점을 이야기합니다. 하드웨어 트랜잭션 메모리는 (Section 17.3에서 이야기 됩니다) 하드웨어 트랜잭션 메모리의 구현이 진행 보장을 제공하지 않는다면 (많지 않은 수의 구현만이 이를 제공합니다) 여기서 도움이 될 수 없음을 이야기 해두는 게 중요합니다. 상당히 실용적으로 보일 수 있는 (덜 흥분했다면) 다른 대안은 Section 7.1.1.5, 그리고 7.1.1.6에서 이야기 된 것들, 그리고 Chapter 8와 9에서 이야기 될 것들이 있겠습니다.

7.5.3.3 Object-Oriented Spaghetti Code

객체지향 프로그래밍은 1980년대 또는 1990년대에 주류가 되었으며, 그 결과 무척 많은 싱글쓰레드 기반 객체지향 코드가 제품 단계에 존재하게 되었습니다. 비록 객체지향성이 가치 있는 소프트웨어 기법이 될 수 있지만, 규율 없는 객체의 사용은 객체 지향 스파게티 코드를 쉽게 초래할 수 있습니다. 객체 지향 스파게티 코드에서, 제어는 객체에서 객체로 근본적으로 무작위적으로 날

아다녀서, 코드를 이해하기 어렵게, 어쩌면 락킹 계층을 조정하기가 불가능하게까지 합니다.

많은 사람들이 그런 코드는 어떤 경우든 정리되어야 한다고 주장하겠지만, 그런 일은 실제 하기보다 말로만 하기가 훨씬 쉽습니다. 여러분이 그런 야수를 병렬화 하는 일을 맡게 되었다면, 여러분은 Section 7.1.1.5, 와 7.1.1.6에서 이야기된, 그리고 Chapter 8와 9에서 이야기될 기법들을 사용해 락킹을 저주할 가능성을 줄여야 합니다. 이 상황은 트랜잭션 메모리에 영감을 주는 사용 예로 나타나며, 그러므로 그것을 시도해 보는 것도 가치가 있을 수 있습니다. 그러나, 동기화 메커니즘의 선택은 Chapter 3에서 이야기 된 하드웨어의 습관을 고려해 이루어져야 합니다. 어쨌건, 동기화 메커니즘의 오버헤드가 보호되는 오퍼레이션의 것보다 수십 수백 배 높다면, 그 결과는 그렇게 아름답지 못할 겁니다.

그리고 이는 이런 상황에서 물어볼 가치가 있는 질문에 이르게 합니다: 이 코드는 순차적으로 동작하도록 남겨져 있어야 하는가? 예를 들어, 어쩌면 병렬성은 쓰레드 단계가 아니라 프로세스 단계에서 사용되어야 할 수도 있습니다. 일반적으로, 어떤 작업이 매우 어렵다고 증명된다면, 그 특정 작업을 해내기 위한 대안적 방법만을 찾기보다는 그 문제 자체를 해결할 수도 있는 더 나은 대안적 작업을 찾는 게 가치 있을 수 있습니다.

7.6 Summary

Achievement unlocked.

Unknown

락킹은 아마도 가장 널리 사용된, 그리고 일반적으로 가장 유용한 동기화 도구입니다. 하지만, 그것은 어플리케이션과 라이브러리에 시작 단계에서부터 설계되었을 때 가장 잘 동작합니다. 언젠가는 병렬로 수행되어야 할, 이미 존재하는 싱글 쓰레드 기반 코드의 많은 양을 놓고 볼 때 락킹은 여러분의 병렬 프로그래밍 도구상자의 유일한 도구가 되지는 않아야 합니다. 다음의 몇 챕터들은 다른 도구들을 이야기 하고, 그것들이 락킹과 다른 것들과 함께 어떻게 최선의 하모니를 이룰 수 있는지 설명합니다.

Chapter 8

Data Ownership

락킹과 함께 나타나는 동기화 오버헤드를 막는 가장 간단한 한가지 방법은 데이터를 쓰레드들 사이에 (또는, 커널의 경우라면, CPU 사이에) 배분해서 데이터의 특정 부분은 하나의 쓰레드에 의해서만 액세스 되고 수정되게끔 하는 것입니다. 흥미롭게도, 데이터 소유권은 병렬 설계 기술의 “커다란 세가지” 각각을 커버합니다: 쓰레드 사이로 (또는, 경우에 따라선 CPU 사이에) 파티셔닝을 하고, 모든 로컬 오퍼레이션을 배치로 처리하며, 동기화 오퍼레이션의 제거가 극한의 로직으로 이르는 것을 완화시킵니다. 따라서 데이터 소유권이 널리 사용된다는 것은 놀라운 일이 아닙니다: 심지어 초심자들도 본능적으로 이를 사용합니다. 사실, 데이터 소유권은 워낙 널리 사용되어서 이 챕터에서는 새로운 예제를 소개하지는 않고, 앞의 챕터들에서 다루었던 것들을 다시 한번 다뤄 보겠습니다.

Quick Quiz 8.1: C 또는 C++로 공유 메모리 병렬 프로그램들을 (예를 들면, pthread를 사용해서) 만들 때 어떤 형태의 데이터 소유권이 제거하기 엄청 어렵나요?



데이터 소유권을 위한 여러 접근법들이 있습니다. Section 8.1은 데이터 소유권에서의 논리적 극한을 선보이는데, 각 쓰레드가 각자의 사적 주소 공간을 갖는 경우입니다. Section 8.2은 그 반대의 극한을 보는데, 데이터가 공유되지만 다른 쓰레드들은 이 데이터로의 다른 접근 권한을 갖습니다. Section 8.3은 함수 배달을 설명하는데, 이는 다른 쓰레드들이 특정 쓰레드에 의해 소유된 데이터로의 간접 액세스를 하게 허용하는 방법입니다. Section 8.4은 어떻게 지정된 쓰레드들이 특정 함수와 연관 데이터의 소유권을 할당받을 수 있는지 설명합니다. Section 8.5은 공유 데이터를 사용하는 알고리즘을 데이터 소유권을 사용하도록 변화시킴으로써 성능을 개선시키는 과정을 논합니다. 마지막으로, Section 8.6은 데이터 소유권을 일등시민으로 사용하는 소프트웨어 환경 몇가지를 보입니다.

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

A man's home is his castle

Ancient Laws of England

Section 4.1은 다음 예를 소개했습니다:

```
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 메모리를 공유하지 않는 별개의 프로세스로서 두개의 인스턴스를 병렬로 수행합니다. 따라서, 특정 프로세스의 모든 데이터는 그 프로세스에 의해 소유되며, 따라서 앞의 예에서의 데이터의 거의 모든 것이 소유되어 있습니다. 이 방법은 동기화 오버헤드를 거의 없앱니다. 그로 인한 극단적 단순성과 최적의 성능의 조합은 분명 무척 매력적입니다.

Quick Quiz 8.2: Section 8.1에서 보인 이 예에 남아 있는 동기화는 무엇인가요?



Quick Quiz 8.3: Section 8.1에 보인 예에 어떤 공유되는 데이터가 있나요?



이와 똑같은 패턴은 Listing 4.1 와 4.2에 보인 것처럼 sh 과 같이 C로도 작성될 수 있습니다.

이 병렬성의 간단한 형태를 반복하는 것은 속임수나 책임을 회피하는 행위가 아니고, 오히려 여러분의 코드를 더 빨리 돌아가게 만드는 간단하고 우아한 방법입니다. 이는 빠르고, 잘 확장되며, 프로그래밍하기 쉽고, 유지하기 쉬우며, 일이 되게 만듭니다. 또한, (가용한 곳에) 이 방법을 취하는 것은 개발자가 정교한 싱글쓰레드 최적화를 compute_it에 적용하거나, 이 방법이 적용될 수 없는 부분의 코드에 정교한 병렬 프로그래밍

패턴을 적용하는 것과 같은 다른 곳에 집중할 시간을 벌어줍니다. 좋아하지 않을 이유가 있나요?

다음 섹션은 공유 메모리 병렬 프로그램에서의 데이터 소유권의 사용에 대해 이야기 합니다.

8.2 Partial Data Ownership and `pthreads`

Give thy mind more to what thou hast than to what thou hast not.

Marcus Aurelius Antoninus

동시적 카운팅 (Chapter 5 을 참고하세요) 은 데이터 소유권을 무척 많이 사용합니다만, 살짝 꼬여 있습니다. 쓰레드들은 다른 쓰레드에 의해 소유된 데이터를 수정하지 못합니다만, 그걸 읽을 수 있습니다. 짧게 말해, 공유 메모리의 사용은 소유권과 접근 권한의 미묘한 차이가 있는 개념을 허용합니다.

예를 들어, page 53 의 Listing 5.4 에서 보인 쓰레드 별 통계적 카운터 구현을 생각해 봅시다. 여기서, `inc_count()` 는 연관된 쓰레드의 `counter` 의 인스턴스만을 업데이트하는 반면, `read_count()` 는 모든 쓰레드의 `counter` 의 인스턴스를 접근하지만 수정하지는 않습니다.

Quick Quiz 8.4: 각 쓰레드가 각자의 쓰레드별 변수 인스턴스를 읽지만 다른 쓰레드의 인스턴스에 쓰기를 하는 부분적 데이터 소유권이 말이 되긴 할까요?

부분적 데이터 소유권은 리눅스 커널 내에서도 흔합니다. 예를 들어, 특정 CPU 는 자신의 CPU 별 변수를 인터럽트가 불능화 된 상태에서만 읽는게 허용될 수도 있고, 다른 CPU 는 이 첫번째 CPU 의 CPU 별 변수를 연관된 CPU 별 락을 잡은 상태에서만 읽기가 허용될 수도 있습니다. 그러면 이 CPU 는 자신이 소유한 CPU 별 변수의 집합을 인터럽트를 불능화 했고 자신의 CPU 별 락을 잡고 있는 상태에서라면 업데이트 하는게 허용될 겁니다. 이 조정은 각 CPU 의 매우 오버헤드가 낮은 스스로가 소유한 CPU 별 변수 집합으로의 액세스를 허용하는 reader-writer 락으로 생각될 수 있습니다. 이 테마에는 무척 많은 변종들이 있습니다.

그 자신의 부분에서, 순수한 데이터 소유권은 또한 일반적이고 유용한데, 예를 들어 Section 6.4.3 의 page 89 에서부터 설명된 쓰레드별 메모리 할당자 캐시가 한 예가 되겠습니다. 이 알고리즘에서, 각 쓰레드의 캐시는 해당 쓰레드에 완전히 사유되어 있습니다.

8.3 Function Shipping

If the mountain will not come to Muhammad, then Muhammad must go to the mountain.

Essays, Francis Bacon

앞의 섹션은 쓰레드들이 다른 쓰레드의 데이터에 접근하는 약화된 형태의 데이터 소유권을 이야기 했습니다. 이는 데이터를 그것을 필요로 하는 함수에게 가져다 주는 것으로 생각될 수 있습니다. 하나의 대안적 방법은 함수를 데이터에게 보내는 것입니다.

그런 방법이 page 64 에서 시작하는 Section 5.4.3 에 그려져 있는데, 특히 page 66 의 Listing 5.18 에 있는 `flush_local_count_sig()` 와 `flush_local_count()` 함수가 그것입니다.

`flush_local_count_sig()` 함수는 보내진 함수처럼 동작하는 시그널 핸들러입니다. `flush_local_count()` 의 `pthread_kill()` 함수는 시그널을 보내고—함수를 배송하고—이 보내진 함수가 수행될 때 까지 기다립니다. 이 보내진 함수는 동시에 수행되는 `add_count()` 나 `sub_count()` 들과의 (page 66 의 Listing 5.19 와 page 67 의 Listing 5.20 를 참고하세요) 상호작용이라는 드물지만은 않은 추가적 복잡도를 갖습니다.

Quick Quiz 8.5: POSIX 시그널 외에 어떤 다른 메커니즘이 함수 보내기를 위해 사용될 수 있을까요?



8.4 Designated Thread

Let a man practice the profession which he best knows.

Cicero

앞의 섹션들은 각 쓰레드가 각자의 복사본을 또는 각자의 데이터 일부를 가질 수 있게 하는 방법들을 설명했습니다. 대조적으로, 이 섹션은 함수적 해제 방법을 설명하는데, 특별한 특정 쓰레드가 그 일을 하는데 필요한 데이터로의 권리를 갖는 형태입니다. Section 5.2.4 의 결과적으로 일관적인 카운터 구현이 한 예를 제공합니다. 이 구현은 Listing 5.5 의 라인 17-34 에 보인 `eventual()` 함수를 수행하는 특별한 쓰레드를 갖습니다. 이 `eventual()` 쓰레드는 주기적으로 쓰레드별 카운트를 글로벌 카운터로 끌어와서, 글로벌 카운터로의 액세스는 그 이름이 말하듯 결과적으로 실제 값으로 수렴하게 만듭니다.

Quick Quiz 8.6: 하지만 Listing 5.5 의 라인 17–34에 있는 `eventual()` 함수의 어느 부분도 실제로 `eventual()` 쓰레드에 소유되어 있지 않습니다! 이게 어떻게 데이터 소유권인가요???

■

8.5 Privatization

There is, of course, a difference between what a man seizes and what he really possesses.

Pearl S. Buck

공유 메모리 병렬 프로그램의 성능과 확장성을 개선하는 한가지 방법은 그것을 공유 데이터를 특정 쓰레드가 소유하는 사적 데이터로 변환시키는 것입니다.

이에 대한 완벽한 예 하나가 Section 6.1.1 의 Quick Quizz 중 하나의 답에 있는데, Dining Philosophers 문제를 표준적 교재의 해법보다 훨씬 나은 성능과 확장성을 제공하는 해법을 만들기 위해 사유화를 사용합니다. 원래의 문제는 철학자들 한 쌍 사이에 하나의 포크만 존재하는 테이블에 앉아있는 다섯명의 철학자들을 두어서, 최대 두명의 철학자들만이 동시에 식사를 할 수 있게 합니다.

우리는 추가적인 다섯개의 포크를 제공해서 각 철학자가 그 또는 그녀의 소유된 사적 포크 한쌍을 가질 수 있게 해서 이 문제를 간단히 사유화 할 수 있습니다. 이는 모든 다섯명의 철학자들이 동시에 식사를 할 수 있게 하며, 또한 특정 종류의 질병의 확산을 크게 줄여줍니다.

다른 경우에는 사유화는 비용을 내포합니다. 예를 들어, page 57 의 Listing 5.7 에 보여진 간단한 한계 카운터를 생각해 봅시다. 이는 쓰레드들이 다른 쓰레드의 데이터를 읽을 수 있는 알고리즘의 한 예입니다만, 각자의 소유하고 있는 데이터만을 업데이트 할 수 있습니다. 이 알고리즘의 빠른 리뷰는 쓰래드 사이를 가로지르는 액세스들만이 `read_count()` 의 합산 반복문에 존재함을 보입니다. 만약 이 반복문이 제거된다면, 우린 더 효율적인 순수한 데이터 소유권으로 넘어갈 수 있습니다만, `read_count()` 의 덜 정확한 결과라는 비용을 치르게 됩니다.

Quick Quiz 8.7: 쓰래드별 데이터의 완전한 사유화를 유지하면서 높은 정확도를 유지하는게 가능할까요?

■

부분적 사유화 또한 가능한데, 약간의 동기화 필요성을 갖긴 하지만 완전히 공유된 경우보다는 덜합니다. Section 4.3.4.4 에 어떤 부분적 사유화 가능성이 보여졌습니다. Chapter 9 는 안전하게 공적 데이터 구조를 사적인 것으로 만드는 방법을 제공함으로써 임시적 데이터 소유권의 부분을 소개할 겁니다.

요약해서, 사유화는 병렬 프로그래머의 도구상자의 강력한 도구입니다만, 주의와 함께 사용되어야만 합니다. 모든 다른 동기화 도구들과 마찬가지로, 성능과 확장성을 떨어뜨리는 동시에 복잡도를 높일 가능성을 갖고 있습니다.

8.6 Other Uses of Data Ownership

Everything comes to us that belongs to us if we create the capacity to receive it.

Rabindranath Tagore

데이터 소유권은 쓰래드 액세스나 업데이트를 가로지르 필요 없게끔 데이터가 조각내어질 수 있을 때 가장 잘 동작합니다. 다행히, 이 상황은 병렬 프로그래밍 환경의 넓은 다양성 내에서도 합리적 수준으로 흔합니다.

데이터 소유권의 예는 다음을 포함합니다:

1. MPI [MPI08] 와 BOINC [Uni08a] 과 같은 모든 메세지 패싱 환경.
2. 맵리듀스 [Jac08].
3. RPC, 웹 서비스, 그리고 백엔드 데이터베이스 서버를 두는 모든 시스템과 같은 클라이언트-서버 시스템.
4. 무공유 데이터베이스 시스템.
5. 별개의 프로세스별 어드레스 스페이스를 갖는 fork-join 시스템.
6. Erlang 언어와 같은 프로세스 기반 병렬성.
7. C 언어의 쓰래드 환경에서의 스택 상의 auto 변수와 같은 사적 변수들.
8. 특히 GPGPU 를 위해 잘 갖추어져 있는 많은 병렬 선형대수 알고리즘들.¹
9. 각 연결 (flwo) (흐름-flow- 이라고도 불립니다 [DKS89, Zha89, Mck90]) 이 특정 쓰래드에 할당되는 네트워킹을 위해 조정된 운영체계 커널. 이런 시도의 최근의 한 예는 IX 운영체제입니다 [BPP⁺16]. IX 는 Section 9.5 에서 설명될 동기화 메커니즘을 사용하는 공유 데이터 구조를 갖습니다.

¹ 하지만 그 외에도 아주 많은 어플리케이션들이 GPGPU 로 포팅되어 있음을 알아두시기 바랍니다 [Mat17, AMD20, NVi17a, NVi17b].

데이터 소유권은 아마도 존재하는 동기화 메커니즘들 중 가장 감사를 못받은 것입니다. 제대로 사용되었을 때, 이 방법은 상대가 없는 단순성, 성능, 그리고 확장성을 제공합니다. 아마도 이것의 단순성이 그것이 가져 마땅한 존중을 방해했을 겁니다. 더 큰 성능과 확장성이 감소된 복잡도와 연결된 것에 대해 더이상 할말이 없게 끔 데이터 소유권의 교묘함과 힘에 대한 더 많은 감사가 더 많은 수준의 존중으로 이어지길 바랍니다.

Chapter 9

Deferred Processing

일을 미루기라는 전략은 기록된 역사의 시작 전부터 있어왔습니다. 이것은 때로 꾸물대기 또는 심지어 순전한 게으름이라고 조소받았습니다. 하지만, 지난 수십년의 작업은 이 전략의 병렬 알고리즘을 단순화하고 능률화하는 그 가치를 입증했습니다 [KL80, Mas92]. 믿든 말든, 병렬 프로그래밍에서의 “게으름”은 종종 부지런함보다 성능도 좋고 확장성도 좋습니다! 이 성능과 확장성의 이득은 일을 미루는 것은 동기화 기능의 약화를 가능케 한다는, 따라서 동기화 오버헤드를 줄인다는 사실에 기인합니다. 일 미루기의 일반적인 방법은 레퍼런스 카운팅 (Section 9.2), 해저드 포인터 (Section 9.3), 시퀀스 락킹 (Section 9.4), 그리고 RCU (Section 9.5)가 있습니다. 마지막으로, Section 9.6는 이 챕터에서 다루어지는 일 미루기 방법들 가운데 어떤 것 어떻게 고르는지 설명하며 Section 9.7에서는 업데이트를 이야기 합니다. 하지만 먼저, Section 9.1는 이 방법들을 비교하고 대비하는데 사용할 예제 알고리즘 하나를 소개합니다.

9.1 Running Example

An ounce of application is worth a ton of abstraction.

Booker T. Washington

이 챕터는 이 방법들의 가치를 보이고 그것들을 비교할 수 있게끔 단순화된 패킷 라우팅 알고리즘 하나를 사용하겠습니다. 라우팅 알고리즘은 각각의 송신 TCP/IP 패킷들을 올바른 네트워크 인터페이스로 전달하기 위해 운영체제 커널에서 사용됩니다. 이 특정 알고리즘은 BSD UNIX에서 사용된 1980년대의 고전 packet-train-optimized 알고리즘의 [Jac88] 간략화된 버전으로, 간단한 링크드 리스트로 구성됩니다.¹ 현대의 라우팅 알고리즘은 더 복잡한 데이터 구조를 사용합니다만, 간단한 알고리즘은 간단한 세팅에서 병렬성만의 문제를 잘 볼 수 있게 도와줄 겁니다.

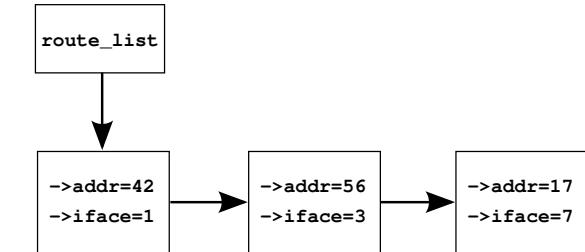


Figure 9.1: Pre-BSD Packet Routing List

우리는 탐색 키를 출발지와 도착지 IP 주소와 포트번호로 구성되는 네가지의 값에서 간단한 정수 하나로 줄여서 이 알고리즘을 더욱 단순화 시킵니다. 탐색되고 리턴되는 값 역시 단순한 정수 하나가 되며, 따라서 이 데이터 구조는 주소 42의 패킷을 인터페이스 1로, 주소 56을 인터페이스 3으로, 그리고 주소 17을 인터페이스 7로 전달시키는 Figure 9.1에 보인 것과 같은 것입니다. 이 리스트는 빈번하게 탐색되고 드물게 업데이트 됩니다. Chapter 3에서 우리는 유한한 빛의 속도와 물질의 원자성과 같은 불편한 물리 법칙을 피하기 위한 최선의 방법은 데이터를 쪼개거나 읽기가 대부분인 공유에 의존하는 것임을 배웠습니다. 이 챕터는 읽기가 대부분인 공유 기법을 Pre-BSD 패킷 라우팅에 적용해봅니다.

Listing 9.1 (`route_seq.c`)은 Figure 9.1에 연관되는 간단한 싱글쓰레드 구현을 보입니다. 라인 1-5는 `route_entry` 구조를 정의하고 라인 6는 `route_list` 헤더를 정의합니다. 라인 8-20는 `route_lookup()`을 정의하는데, 이 함수는 순차적으로 `route_list`를 탐색하고 연관된 `>iface`를 리턴하거나 해당 라우팅 항목이 없으면 `ULONG_MAX`를 리턴합니다. 라인 22-33는

¹ 달리 말하자면, 이는 OpenBSD 도, NetBSD 도, 심지어 FreeBSD 도 아니고 그저 Pre-BSD 라 불릴 수 있습니다.

Listing 9.1: Sequential Pre-BSD Routing Table

```

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, &route_list, re_next) {
14         if (rep->addr == addr) {
15             ret = rep->iface;
16             return ret;
17         }
18     }
19     return ULONG_MAX;
20 }
21
22 int route_add(unsigned long addr, unsigned long interface)
23 {
24     struct route_entry *rep;
25
26     rep = malloc(sizeof(*rep));
27     if (!rep)
28         return -ENOMEM;
29     rep->addr = addr;
30     rep->iface = interface;
31     cds_list_add(&rep->re_next, &route_list);
32     return 0;
33 }
34
35 int route_del(unsigned long addr)
36 {
37     struct route_entry *rep;
38
39     cds_list_for_each_entry(rep, &route_list, re_next) {
40         if (rep->addr == addr) {
41             cds_list_del(&rep->re_next);
42             free(rep);
43             return 0;
44         }
45     }
46     return -ENOENT;
47 }

```

`route_add()`를 정의하는데, 이 함수는 `route_entry` 구조체를 할당받고, 초기화 한 후, 이 리스트에 그것을 추가하는데, 메모리 할당 실패의 경우엔 `-ENOMEM`을 리턴합니다. 마지막으로, 라인 35-47는 `route_del()`을 정의하는데, 이 함수는 특정 `route_entry` 구조체를 만약 존재한다면 제거하고 메모리 해제하고, 그렇지 않으면 `-ENOENT`를 리턴합니다.

이 싱글쓰레드 구현은 이 챕터에서 다양한 동시적 구현들의 프로토타입 역할을 하게 되며, 이상적인 확장성과 성능의 예측에도 사용됩니다.

Listing 9.2: Reference-Counted Pre-BSD Routing Table Lookup (BUGGY!!!)

```

1 struct route_entry {
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     WRITE_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 && atomic_dec_and_test(&rep->re_refcnt))
30             re_free(rep);
31         rep = READ_ONCE(*repp);
32         if (rep == NULL)
33             return ULONG_MAX;
34         do {
35             if (READ_ONCE(rep->re_freed))
36                 abort();
37             old = atomic_read(&rep->re_refcnt);
38             if (old <= 0)
39                 goto retry;
40             new = old + 1;
41         } while (atomic_cmpxchg(&rep->re_refcnt,
42                                old, new) != old);
43         repp = &rep->re_next;
44     } while (rep->addr != addr);
45     ret = rep->iface;
46     if (atomic_dec_and_test(&rep->re_refcnt))
47         re_free(rep);
48     return ret;
49 }

```

9.2 Reference Counting

I am never letting you go!

Unknown

레퍼런스 카운팅은 객체가 너무 이르게 메모리 해제되는 걸 방지하기 위해 참조 수를 기록합니다. 그것으로서, 이것은 최소한 1960년대 Weizenbaum의 논문 [Wei63] 까지 거슬러 올라가는 길고 영광스런 사용 역사를 가졌습니다. Weizenbaum은 레퍼런스 카운팅을 그게 이미 잘 알려진 것처럼 이야기 했으나, 1950년대나 심지어 1940년대까지 거슬러 올라갈 수도 있습니다. 그리고 어

Listing 9.3: Reference-Counted Pre-BSD Routing Table Add/Delete (BUGGY!!!)

```

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

```

쩌면 그보다 심지어 더 나아가서, 크고 위험한 기계를 수리하는 사람들은 패드락을 통해 구현된 기계적 레퍼런스 카운팅 기법을 오랫동안 사용해 왔습니다. 기계에 들어가기 전에, 각 작업자는 패드락을 기계의 on/off 스위치 위에서 잠금으로 이 기계가 작업자가 안에 들어가 있는 사이에 켜지는 걸 방지했습니다. 따라서 레퍼런스 카운팅은 Pre-BSD 라우팅의 동시적 구현을 위한 시간으로 증명된 좋은 후보가 될 수 있습니다.

그런 이유로, Listing 9.2 은 데이터 구조와 route_lookup() 함수를 보이고 Listing 9.3 은 route_add() 와 route_del() 함수를 보입니다(모두 route_refcnt.c 에 있습니다). 이 알고리즘들은 Listing 9.1 에 보인 순차적 알고리즘과 상당히 비슷하므로 차이점만 이야기 하겠습니다.

Listing 9.2 부터 시작해 보자면, 라인 2 는 실제 레퍼런스 카운터를 추가하고, 라인 6 는 메모리 해제 후 사용 체크 필드인 ->re_freed 를 추가하고, 라인 9 은 동시적 업데이트를 동기화 하는데 사용할 routelock 을 추가

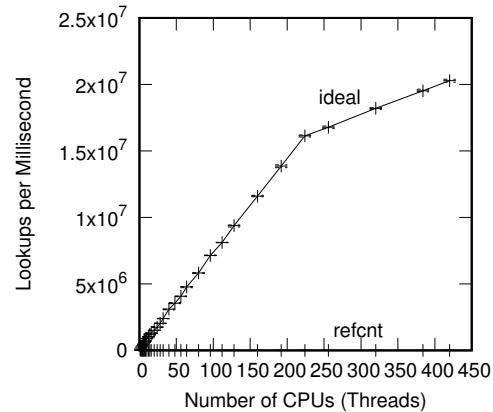


Figure 9.2: Pre-BSD Routing Table Protected by Reference Counting

하며, 라인 11-15 는 ->re_freed 를 세팅하고 메모리 해제 후 사용 베그를 체크하기 위한 route_lookup() 을 활성화 시키는 re_free() 를 추가합니다. route_lookup() 내에서, 라인 29-30 는 앞 원소의 레퍼런스 카운트를 해제하고 그 카운트가 0이 되면 메모리 해제 시키며 라인 34-42 는 새 원소의 레퍼런스를 획득하며, 라인 35 와 36 는 메모리 해제 후 사용 체크를 수행합니다.

Quick Quiz 9.1: 메모리 해제 후 사용 체크를 왜 신경쓰죠?



Listing 9.3 에서, 라인 11, 15, 24, 32, 그리고 39 는 동시적 업데이트들을 동기화 시키기 위한 락킹을 넣습니다. 라인 13 은 메모리 해제 후 사용 체크 필드인 ->re_free 를 초기화하고 마지막으로 라인 33-34 는 레퍼런스 카운트의 값이 0일 때 re_free() 를 호출합니다.

Quick Quiz 9.2: Listing 9.3 의 route_del() 은 메모리 해제될 원소로의 링단을 보호하기 위해서는 레퍼런스 카운트를 사용하지 않나요?



Figure 9.2 는 총 448개 하드웨어 쓰레드를 갖는 여덟개 소켓의 소켓당 28 코어를 갖는 하이퍼쓰레드 기반 2.1 GHz x86 시스템에서의 열개 원소를 갖는 리스트의 읽기 전용 워크로드에서의 레퍼런스 카운팅의 성능과 확장성을 보입니다 (hps.2019.12.02a/1scpu. hps). “ideal” 트레이스는 Listing 9.1 에 보인 순차적 코드를 통해 만들어졌는데, 이건 읽기 전용 워크로드이기 때문에 동작합니다. 레퍼런스 카운팅의 성능은 참담하고 그 확장성은 그보다 더한데, x 축에 구분할 수 없는 “refcnt” 트레이스로 보여져 있습니다. Chapter 3 에서의 시각에 따르면 이건 놀랍지 않습니다: 레퍼런스 카운트 획득과 해제는 그게 사용되지 않았다면 읽기만 이루어

졌을 워크로드에 빈번한 공유 메모리 쓰기를 추가하여, 물리 법칙으로부터 상당한 보복을 일으킵니다. 원래 그 래야 하듯, 세상의 모든 희망적 생각들이 현대의 디지털 전자장치에 사용되는 빛의 속도를 높이거나 원자의 크기를 줄이지는 않습니다.

Quick Quiz 9.3: Figure 9.2 의 224 CPU 에서의 “ideal” 선의 끊어짐은 무엇 때문인가요? 곧은 선이어야 하지 않아요?

Quick Quiz 9.4: Figure 9.2 의 refcnt 트레이스는 최소한 x-axis 에서 좀 떨어져야 하는 거 아닌가요???

하지만 이건 심지어 더 나빠집니다.

반복적으로 `route_add()` 와 `route_del()` 을 호출하는 여러 업데이터 쓰레드를 수행시키는 것은 금방 Listing 9.2 의 line 36 에 있는 메모리 해제 후 사용 버그를 알리는 `abort()` 문을 수행되게 합니다. 이는 곧 레퍼런스 카운트는 확장성과 성능을 저하시킬 뿐만 아니라 필요한 보호를 제공하는데도 실패함을 의미합니다.

Figure 9.1 에 보인 리스트로 보이는 이 메모리 해제 후 사용 버그를 일으키는 순차적 이벤트 중 하나는 다음과 같습니다:

1. 쓰레드 A 가 주소 42 를 탐색하여, Listing 9.2 의 `route_lookup()` 의 라인 32 에 도달합니다. 달리 말하자면, 쓰레드 A 는 첫번째 원소로의 포인터를 갖지만, 아직 그것으로의 참조는 획득하지 않았습니다.
2. 쓰레드 B 가 주소 42 의 route 원소를 제거하기 위해 Listing 9.3 의 `route_del()` 을 호출합니다. 이는 성공적으로 마무리 되며, 이 원소의 `->re_refcnt` 필드는 값 1 을 가졌으므로 `->re_freed` 필드를 설정하고 이 원소를 메모리 해제하기 위해 `re_free()` 를 호출합니다.
3. 쓰레드 A 는 `route_lookup()` 의 수행을 계속합니다. `rep` 포인터는 NULL 이 아니지만, 라인 35 는 그 `->re_freed` 필드가 0 이 아님을 보게 되므로, 라인 36 는 `abort()` 를 호출합니다.

문제는 이 레퍼런스 카운트가 보호되어야 할 객체 내에 위치해 있지만, 이는 이 레퍼런스 카운트 자체가 획득되는 그 순간에는 보호가 없음을 의미합니다! 이는 Gamsa 등 [GKAS99] 에 의해 이야기된 레퍼런스 카운팅에 대비되는 락킹 문제입니다. 어떤 사람은 `route` 원소별 레퍼런스 카운트 획득을 보호하기 위한 글로벌 락이나 레퍼런스 카운트를 상상해 볼 수 있겠지만, 이는 상당한 컨텐션 문제를 일으킬 겁니다. 동시적 환경에서 안전한 레퍼런스 카운트 획득을 허용하는 알고리즘이

존재하지만 [Val95], 그것들은 극단적으로 복잡하고 예외에 취약할 뿐 아니라 [MS95], 치참한 성능과 확장성을 제공합니다 [HMBW07].

짧게 요약하자면, 동시성은 레퍼런스 카운팅의 유용성을 분명 줄였습니다!

Quick Quiz 9.5: 동시성이 “레퍼런스 카운팅의 유용성을 분명 줄였다” 면, 리눅스 커널에는 왜 그렇게 많은 레퍼런스 카운터가 존재하나요?

그렇다고는 하나, 때로는 어떤 문제를 해결하기 위해 선 그걸 완전 다른 방식으로 볼 필요가 있습니다. 다음 섹션은 훌륭한 성능과 확장성을 제공하는 안에서 바깥으로 향하는 레퍼런스 카운트로 생각 될 수 있는 방법을 알아보겠습니다.

9.3 Hazard Pointers

If in doubt, turn it inside out.

Zara Carpenter

동시적 레퍼런스 카운팅의 문제를 막는 한가지 방법은 레퍼런스 카운터를 뒤집어 구현하는 것으로, 즉, 데이터 원소에 저장된 정수의 값을 증가시키는 대신 CPU 별 (또는 쓰레드별) 리스트에 데이터 원소로의 포인터를 저장하는 것입니다. 이 리스트의 각 원소는 해저드 포인터 [Mic04] 라고 불립니다.² 그러면 주어진 데이터 원소의 “가상 레퍼런스 카운터”의 값은 이 원소를 참조하는 해저드 포인터의 갯수를 세는 것으로 구해질 수 있습니다. 따라서, 이 원소가 읽기 쓰레드에 의해 액세스 될 수 없게 되다면, 해당 원소를 참조하는 해저드 포인터가 더이상 존재하지 않게 되어서, 이 원소는 안전히 메모리 해제될 수 있습니다.

물론, 이는 해저드 포인터 획득이 동시의 삭제와의 위험한 레이스를 막기 위해 아주 조심스럽게 다뤄져야 함을 의미합니다. Listing 9.4 에 하나의 구현이 보여져 있는데, 라인 1-16 에 `hp_try_record()` 가, 라인 18-27 에 `hp_record()` 가, 그리고 라인 29-33 에 `hp_clear()` 가 있습니다 (`hazptr.h`).

라인 16 의 `hp_try_record()` 매크로는 `_h_t_r_impl()` 함수를 위한 캐스팅 래퍼로, `p` 에 의해 참조되는 포인터를 `hp` 에 의해 참조되는 해저드 포인터에 저장합니다. 이게 성공하면, 저장된 포인터의 값을 리턴합니다. 해당 포인터가 NULL 이 되는 관계로 여기 실패하면, NULL 을 리턴합니다. 마지막으로, 업데이트와의 레이스 때문에 실패하면 특수한 `HAZPTR_POISON` 토큰을 리턴합니다.

² 독립적으로 다른 사람들에 의해서도 발명되었습니다 [HLM02].

Listing 9.4: Hazard-Pointer Recording and Clearing

```

1 static inline void *_h_t_r_impl(void **p,
2                                 hazard_pointer *hp)
3 {
4     void *tmp;
5
6     tmp = READ_ONCE(*p);
7     if (!tmp || tmp == (void *)HAZPTR_POISON)
8         return tmp;
9     WRITE_ONCE(hp->p, tmp);
10    smp_mb();
11    if (tmp == READ_ONCE(*p))
12        return tmp;
13    return (void *)HAZPTR_POISON;
14 }
15
16 #define hp_try_record(p, hp) _h_t_r_impl((void **)(p), hp)
17
18 static inline void *hp_record(void **p,
19                             hazard_pointer *hp)
20 {
21     void *tmp;
22
23     do {
24         tmp = hp_try_record(*p, hp);
25     } while (tmp == (void *)HAZPTR_POISON);
26     return tmp;
27 }
28
29 static inline void hp_clear(hazard_pointer *hp)
30 {
31     smp_mb();
32     WRITE_ONCE(hp->p, NULL);
33 }

```

Quick Quiz 9.6: 해저드 포인터 논문들은 삭제된 원소들을 표시하기 위해 각 포인터의 바닥 비트들을 사용하는데 HAZPTR_POISON 은 왜 사용되나요?



라인 6은 보호되어야 할 객체로의 포인터를 읽습니다. 라인 8이 이 포인터가 NULL 이거나 특수한 삭제된 객체 토큰인 HAZPTR_POISON 이었음을 발견하게 되면, 호출자에게 실패를 알리기 위해 이 포인터의 값을 리턴합니다. 그렇지 않다면, 라인 9는 명시된 해저드 포인터에 이 포인터를 저장하고, 라인 10은 라인 11에서의 원래 포인터의 다시 읽기와 이 저장 사이의 순서를 완전히 강제합니다. (메모리 순서 규칙에 대한 더 많은 내용을 위해선 Chapter 15를 참고하시기 바랍니다.) 원래 포인터의 값이 변경되지 않았다면, 이 해저드 포인터는 가리켜진 객체를 보호하고, 이 경우 라인 12는 이 객체로의 포인터를 리턴하는데, 이는 또한 호출자에게 성공을 알립니다. 그렇지 않고, 이 포인터가 두개의 READ_ONCE() 사이에서 변경되었다면, 라인 13은 실패를 알립니다.

Quick Quiz 9.7: Listing 9.4의 hp_try_record()는 왜 데이터 원소로의 이중 간접 경로를 갖나요? void ** 대신 void *를 쓰는게 어떻습니까?



hp_record() 함수는 상당히 단순합니다: 리턴 값이 HAZPTR_POISON 이 아닌 무엇인가일 때까지 반복적으로 hp_try_record()를 호출합니다.

Quick Quiz 9.8: 왜 hp_try_record()를 신경쓰죠? 그냥 실패에 내성 있는 hp_record() 함수를 그냥 사용하는게 더 쉽지 않나요?



hp_clear() 함수는 심지어 더 간단한데, 이 해저드 포인터에 의해 보호되는 객체의 호출자의 사용과 이 해저드 포인터를 NULL로 만드는 것 사이의 완전한 순서를 강제하기 위해 smp_mb()가 사용됩니다.

해저드 포인터로 보호되는 객체가 연결된 데이터 구조에서 일단 제거되어서 미래의 해저드 포인터 사용 읽기 쓰래드들에게 접근될 수 없게 되면, 그 객체는 Listing 9.5 (hazptr.c)의 라인 48-56에 보인 hazptr_free_later()로 넘겨집니다. 라인 50 와 51는 이 객체를 쓰래드별 리스트 rlist에 넣고 라인 52에서 rcount에 이 객체의 수를 셹니다. 라인 53가 충분히 많은 수의 객체들이 이제 들어와 있다는 것을 보게 되면, 라인 54은 그 중 일부를 메모리 해제하기 위해 hazptr_scan()을 호출합니다.

이 리스트의 라인 6-46에 hazptr_scan() 함수가 보여져 있습니다. 이 함수는 고정된 쓰래드의 최대 갯수 (NR_THREADS) 와 고정된 크기의 해저드 포인터 배열이 사용될 수 있게끔 하는 쓰래드당 해저드 포인터의 고정된 최대 갯수 (K)에 의존하고 있습니다. 어떤 쓰래드든 이 해저드 포인터들을 스캔해야 할 수 있으므로, 각 쓰래드는 각자의 배열을 가지고 있는데, 이는 쓰래드별 변수 gplist를 통해 참조됩니다. 라인 14가 이 쓰래드는 아직 자신의 gplist를 할당받지 않았음을 알게 되면, 라인 15-18가 이 할당을 진행합니다. 라인 20에서의 메모리 배열은 이 쓰래드에 의한 모든 객체의 제거가 라인 22-28가 모든 해저드 포인터를 스캔하고 NULL이 아닌 포인터들을 plist 배열에 넣고 psize에 그 수를 세기 전에 볼 수 있음을 보장하게 합니다. 라인 29에서의 메모리 배열은 이 해저드 포인터들의 읽기가 어떤 객체의 메모리 해제보다도 먼저 일어날 것을 보장되게 합니다. 라인 30은 이어서 이 배열을 정렬해서 아래의 바이너리 탐색이 가능하게 합니다.

라인 31와 32는 이 쓰래드의 곧 메모리 해제할 객체들 리스트의 모든 원소들을 제거하고, 그것들을 지역적인 tmplist에 위치시킨 후 라인 33에서 그 수를 0으로 만듭니다. 라인 34-45에 있는 반복문의 각 수행에서는 곧 메모리 해제될 객체들 각각을 처리합니다. 라인 35와 36는 tmplist에서 첫번째 객체를 제거하고, 라인 37와 38는 이 객체를 보호하는 해저드 포인터가 있음을 확인하고, 라인 39-41는 그것을 rlist에 되돌려 놓습니다. 그렇지 않으면, 라인 43은 이 객체를 메모리 해제합니다.

Listing 9.5: Hazard-Pointer Scanning and Freeing

```

1 int compare(const void *a, const void *b)
2 {
3     return ( *(hazptr_head_t **)a - *(hazptr_head_t **)b );
4 }
5
6 void hazptr_scan()
7 {
8     hazptr_head_t *cur;
9     int i;
10    hazptr_head_t *tmplist;
11    hazptr_head_t **plist = gplist;
12    unsigned long psize;
13
14    if (plist == NULL) {
15        psize = sizeof(hazptr_head_t *) * K * NR_THREADS;
16        plist = (hazptr_head_t **)malloc(psize);
17        BUG_ON(!plist);
18        gplist = plist;
19    }
20    smp_mb();
21    psize = 0;
22    for (i = 0; i < H; i++) {
23        uintptr_t hp = (uintptr_t)READ_ONCE(HP[i].p);
24
25        if (!hp)
26            continue;
27        plist[psize++] = (hazptr_head_t *)hp & ~0x1UL;
28    }
29    smp_mb();
30    qsort(plist, psize, sizeof(hazptr_head_t *), compare);
31    tmplist = rlist;
32    rlist = NULL;
33    rcount = 0;
34    while (tmplist != NULL) {
35        cur = tmplist;
36        tmplist = tmplist->next;
37        if (bsearch(&cur, plist, psize,
38                    sizeof(hazptr_head_t *), compare)) {
39            cur->next = rlist;
40            rlist = cur;
41            rcount++;
42        } else {
43            hazptr_free(cur);
44        }
45    }
46 }
47
48 void hazptr_free_later(hazptr_head_t *n)
49 {
50     n->next = rlist;
51     rlist = n;
52     rcount++;
53     if (rcount >= R) {
54         hazptr_scan();
55     }
56 }

```

Listing 9.6: Hazard-Pointer Pre-BSD Routing Table Lookup

```

1 struct route_entry {
2     struct hazptr_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_hazptr;
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 = hp_try_record(repp, &my_hazptr[offset]);
22         if (!rep)
23             return ULONG_MAX;
24         if ((uintptr_t)rep == HAZPTR_POISON)
25             goto retry;
26         rep = &rep->re_next;
27     } while (rep->addr != addr);
28     if (READ_ONCE(rep->re_freed))
29         abort();
30     return rep->iface;
31 }

```

Pre-BSD 라우팅 예제는 데이터 구조와 route_lookup()을 위해 Listing 9.6에 보인 것처럼, route_add()와 route_del()을 위해선 Listing 9.7에 보인 것처럼 해저드 포인터를 사용할 수 있습니다 (route_hazptr.c). 레퍼런스 카운팅에서와 같이, 이 해저드 포인터 구현은 page 126의 Listing 9.1에 보인 순차적 알고리즘과 상당히 비슷해서 여기선 차이점만 이야기하겠습니다.

Listing 9.6부터 시작하면, 라인 2는 해저드 포인터 해제를 기다리는 객체들을 리스트에 넣는데 사용되는 ->hh 필드를 보이며, 라인 6는 미모리 해제 후 사용 버그를 감지하기 위한 ->re_freed 필드를 보이고, 라인 21는 해저드 포인터 획득을 시도하기 위해 hp_try_record()를 호출합니다. 리턴값이 NULL이면, 라인 23은 호출자에게 찾지 못했음을 리턴합니다. hp_try_record() 호출이 삭제와 레이스 상태에 빠진다면, 라인 25는 라인 18의 retry로 돌아와 이 리스트를 시작부터 다시 순회합니다. do-while 루프는 희망한 원소가 찾아질 때까지 돌아가는데, 만약 이 원소가 이미 메모리 해제되어 있다면 라인 29가 이 프로그램을 종료시킵니다. 그렇지 않다면, 이 원소의 ->iface 필드가 호출자에게 리턴됩니다.

라인 21는 사용하기 더 쉬운 hp_record() 대신 hp_try_record()를 호출해서 hp_try_record()의 실패 시에는 전체 탐색을 재시작합니다. 그리고 그런 재시작은 정확성을 위해 분명 필요합니다. 이를 보기 위해,

원소 A, B, 그리고 C 를 담고 있으며 다음 이벤트를 겪는 해저드 포인터로 보호되는 링크드 리스트를 생각해 봅시다:

1. 쓰레드 0 이 원소 B 로의 해저드 포인터를 저장합니다 (아마 원소 A 로부터 원소 B 로 순회했을 겁니다).
2. 쓰레드 1 이 이 리스트에서 원소 B 를 제거하여, 원소 B 에서 원소 C 로의 포인터를 이 제거를 표시하기 위해 특별한 HAZPTR_POISON 값으로 설정합니다. 쓰레드 0 가 원소 B 로의 해저드 포인터를 가지고 있으므로, 아직 메모리 해제는 되지 않습니다.
3. 쓰레드 1 이 이 리스트에서 원소 C 를 제거합니다. 원소 C 를 참조하는 해저드 포인터가 존재하지 않으므로 곧바로 메모리 해제됩니다.
4. 쓰레드 0 이 이제 제거된 원소 B 의 다음 원소로의 해저드 포인터를 획득하려 합니다만 hp_try_record() 는 HAZPTR_POISON 값을 리턴하여, 호출자가 이 리스트의 시작으로부터 순회를 재시작하게 합니다.

이건 꽉이나 좋은 일인데, B 의 다음 원소는 이제는 메모리 해제된 원소 C 로, 이 말은 쓰레드 0 의 다음 액세스는 임의의 끔찍한 메모리 오염을 초래할 것을 의미하는데, 특히나 원소 C 의 메모리가 이후 어떤 다른 목적으로 재할당 되었다면 그렇습니다. 따라서, 해저드 포인터를 사용하는 읽기 쓰레드는 일반적으로 동시의 삭제를 발견했을 때 전체 순회를 재시작 해야 합니다. 이런 재시작은 종종 어떤 전역의 (그리고 따라서 불변하는) 포인터로 되돌아가야 합니다만, 가끔은 중간의 어떤 지점이 예를 들면 현재 쓰레드가 락을 잡고 있다던지 레퍼런스 카운트를 잡고 있다던지 하는 이유로 여전히 살아 있을 것이 보장된다는 전제 하에 그 중간 지점부터 시작될 수도 있습니다.

Quick Quiz 9.9: 읽기 쓰레드는 “일반적으로” 재시작해야만 한다구요? 그 예외는 어떤 것들인가요?

■
해저드 포인터를 사용하는 알고리즘은 연결된 데이터 구조를 통한 순회를 어떤 단계에서든 재시작할 수도 있으며, 그런 알고리즘은 일반적으로 업데이트를 위해 필요한 모든 해저드 포인터를 얻기 전에 데이터 구조에 어떠한 변경도 가하지 않아야만 합니다.

Quick Quiz 9.10: 하지만 이런 해저드 포인터의 제약들은 다른 형태의 레퍼런스 카운팅에도 적용되지 않나요?

■

Listing 9.7: Hazard-Pointer Pre-BSD Routing Table Add/Delete

```

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

```

이 해저드 포인터 제약은 해저드 포인터가 각 CPU 나 쓰레드에 지역적인 위치에 저장되어 순회가 순회되는 데이터 구조로의 어떠한 쓰기도 없이 가능하게 하기 때문에 읽기 쓰레드에게 큰 이득을 초래합니다. page 71 의 Figure 5.8 로 돌아가 보자면, 해저드 포인터는 CPU 캐시가 자원 복사를 하게 해서, 결국 병렬 액세스 제어 메커니즘을 약화시키며 따라서 성능과 확장성을 증가시킵니다.

해저드 포인터 순회를 재시작하는 것의 또 다른 장점은 최소 메모리 사용량의 감소입니다: 해저드 포인터에 의해 현재 참조되지 않는 모든 객체는 곧바로 메모리 해제됩니다. 대조적으로, Section 9.5 은 읽기 쪽의 재시도를 막는 (그리고 읽기 쪽 오버헤드를 최소화시키는), 그러나 훨씬 큰 메모리 사용량을 초래할 수 있는 메커니즘을 다룹니다.

Listing 9.7 에 route_add() 와 route_del() 함수가 보여져 있습니다. 라인 10 는 ->re_freed 를 초기화하고, 라인 31 은 새로 제거된 객체의 ->re_next 필드를 마킹하며, 라인 33 는 이 객체를 일단 그래도 안전하게 되면 그 객체를 메모리 해제할 hazptr_free_later()

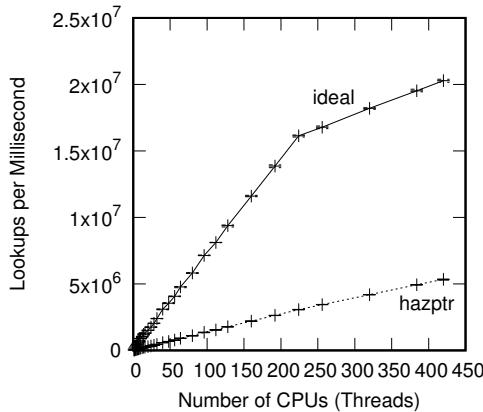


Figure 9.3: Pre-BSD Routing Table Protected by Hazard Pointers

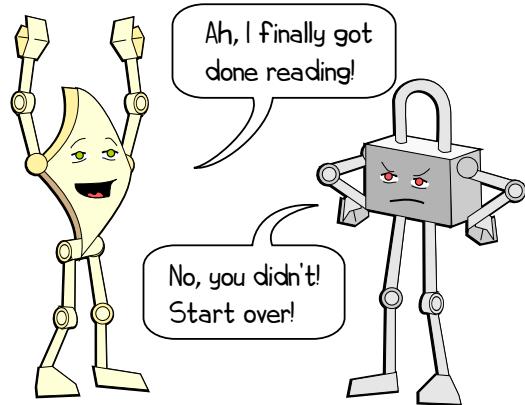


Figure 9.4: Reader And Uncooperative Sequence Lock

함수에 전달합니다. Spinlock 들은 Listing 9.3 에서와 동일하게 동작합니다.

Figure 9.3 는 해저드 포인터로 보호되는 Pre-BSD 라우팅 알고리즘의 성능을 Figure 9.2 와 같은 읽기 전용 워크로드에 대해 보입니다. 해저드 포인터가 레퍼런스 카운팅보다는 훨씬 잘 확장되지만, 해저드 포인터는 여전히 읽기 쓰레드가 공유 메모리에 쓰기를 할 것을 필요로 하며 (레퍼런스의 지역성에 있어 훨씬 개선되긴 했지만), 전체 메모리 배리어와 각 순회된 객체의 재시도 검사를 필요로 합니다. 따라서, 해저드 포인터 성능은 여전히 이상적인 것에 비해선 멍니다. 다른 한편, 동시의 레퍼런스 카운팅의 안일한 방법과 달리, 해저드 풍니터는 동시의 업데이트를 포함하는 워크로드에 대해 정확하게 동작할 뿐 아니라, 훌륭한 확장성을 제공합니다. 다른 메커니즘과의 추가적인 성능 비교는 Chapter 10 와 다른 출판물들 [HMBW07, McK13, Mic04] 에서 찾을 수 있을 겁니다.

Quick Quiz 9.11: Figure 9.3 는 하이퍼쓰레드가 일으키는 224 쓰레드에서의 성능 하락의 기미를 보이지 않습니다. 왜 그런거죠?



Quick Quiz 9.12: “Structured Deferral: Synchronization via Procrastination” [McK13] 이라는 논문은 해저드 포인터가 이상에 가까운 성능을 보임을 이야기 했습니다. Figure 9.3 에는 무슨 일이 벌어진 건가요???



다음 섹션에서는 읽기 쪽 쓰기와 객체별 메모리 배리어 사용을 제거하는 시퀀스 락을 사용해 해저드 포인터를 개선해 봅니다.

9.4 Sequence Locks

It'll be just like starting over.

John Lennon

출간된 시퀀스 락 기록은 [Eas71, Lam77] reader-writer 락킹의 그것만큼이나 오래 되었습니다만, 시퀀스 락은 상대적으로 덜 알려져 있습니다. 시퀀스 락은 리눅스 커널에서 읽기 쓰레드에게 일관적인 상태로 보여야만 하는 읽기가 대부분인 데이터를 위해 사용됩니다. 하지만, reader-writer 락킹과 달리, 읽기 쓰레드는 쓰기 쓰레드를 배제시키지 않습니다. 그 대신, 해저드 포인터처럼, 시퀀스 락은 읽기 쓰레드가 동시의 쓰기 쓰레드로부터의 활동을 탐지한 경우 오퍼레이션을 재시도하게 합니다. Figure 9.4 에서 볼 수 있듯, 읽기 쓰레드들이 아주 가끔 재시도를 해야 하게끔 코드를 설계하는게 중요합니다.

Quick Quiz 9.13: 이 시퀀스 락 이야기는 왜 Chapter 7 에 있지 않죠, 이건 락킹이잖아요?



Listing 9.8: Sequence-Locking Reader

```

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

```

Listing 9.9: Sequence-Locking Writer

```

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

```

시퀀스 락킹의 핵심 컴포넌트는 시퀀스 숫자로, 업데이트 쓰레드가 없을 때에는 짹수가 되며 진행중인 업데이트가 있을 때에는 훌수가 됩니다. 그럼 읽기 쓰레드는 각 액세스 전과 후에 값을 스냅샷 찍어둘 수 있습니다. 두개의 스냅샷 중 어느 하나라도 훌수를 가지고 있다면, 또는 이 두 스냅샷이 다르다면, 동시의 업데이트가 있었다는 말이 되며, 읽기 쓰레드는 이 액세스의 결과를 폐기하고 재시도 해야 합니다. 따라서 읽기 쓰레드는 시퀀스 락으로 보호되는 데이터에 접근할 때 Listing 9.8에 보인 `read_seqbegin()` 과 `read_seqretry()` 함수를 사용해야 합니다. 스기 쓰레드는 각 업데이트 전과 후에 이 값을 증가시켜야만 하고 한번에 하나의 쓰기 쓰레드만이 허용됩니다. 따라서 쓰기 스레드는 시퀀스 락으로 보호되는 데이터를 업데이트 할 때 Listing 9.9에 보인 `write_seqlock()` 과 `write_sequnlock()` 을 사용해야 합니다.

그 결과, 시퀀스 락으로 보호되는 데이터는 임의의 큰 수의 동시 읽기 쓰레드를 가질 수 있지만, 한번에 하나의 쓰기 쓰레드만 가질 수 있습니다. 시퀀스 락킹은 리눅스 커널에서 시간 계측에서 측정 정도를 보호하기 위해 사용됩니다. 시퀀스 락킹은 또한 경로명 순회 중 동시의 이름 변경 오퍼레이션을 감지하기 위해 사용됩니다.

시퀀스 락의 간단한 구현이 Listing 9.10 (`seqlock.h`)에 보여 있습니다. `seqlock_t` 데이터 구조는 라인 1~4에 보여져 있는데 시퀀스 숫자와 함께 쓰기 쓰레드들을 순서짓기 위한 락을 포함하고 있습니다. 라인 6~10는 이름이 의미하듯 `seqlock_t`를 초기화 하는 `seqlock_init()` 를 보입니다.

라인 12~19는 시퀀스 락 읽기 쪽 크리티컬 섹션을 시작하는 `read_seqbegin()` 을 보입니다. 라인 16는 시퀀스 카운터의 스냅샷을 취하고, 라인 17는 이 호출자의 크리티컬 섹션이 시작되기 전으로 이 스냅샷 오퍼레이션이 순서지어지도록 합니다. 마지막으로, 라인 18는 이 스냅샷의 값을 (least-significant bit 이 지워진채로) 리턴하는데, 호출자는 이를 나중의 `read_seqretry()` 호출 시에 넘길 겁니다.

Quick Quiz 9.14: Listing 9.10 의 `read_seqbegin()` 은 왜 이미 망한 읽기가 시작되게 하는 대신 아래쪽 bit 이 셋 되었는지 검사하고 내부적으로 재시도 하지 않나요?



라인 21~29는 연관된 `read_seqbegin()` 호출 이후 최소 하나의 쓰기 쓰레드가 존재했다면 `true` 를 리턴하는 `read_seqretry()` 를 보입니다. 라인 26는 이 호출자의 라인 27의 시퀀스 카운터의 새 스냅샷을 읽어오기 전의 크리티컬 섹션을 순서짓습니다. 라인 28는 이 시퀀스 카운터가 변화되었는지 검사하는데, 달리 말하면 최소 하나의 쓰기 쓰레드가 있었는지 검사하고, 그렇다면 `true` 를 리턴합니다.

Listing 9.10: Sequence-Locking Implementation

```

1 typedef struct {
2     unsigned long seq;
3     spinlock_t lock;
4 } seqlock_t;
5
6 static inline void seqlock_init(seqlock_t *slp)
7 {
8     slp->seq = 0;
9     spin_lock_init(&slp->lock);
10 }
11
12 static inline unsigned long read_seqbegin(seqlock_t *slp)
13 {
14     unsigned long s;
15
16     s = READ_ONCE(slp->seq);
17     smp_mb();
18     return s & ~0x1UL;
19 }
20
21 static inline int read_seqretry(seqlock_t *slp,
22                                 unsigned long oldseq)
23 {
24     unsigned long s;
25
26     smp_mb();
27     s = READ_ONCE(slp->seq);
28     return s != oldseq;
29 }
30
31 static inline void write_seqlock(seqlock_t *slp)
32 {
33     spin_lock(&slp->lock);
34     ++slp->seq;
35     smp_mb();
36 }
37
38 static inline void write_sequnlock(seqlock_t *slp)
39 {
40     smp_mb();
41     ++slp->seq;
42     spin_unlock(&slp->lock);
43 }

```

Quick Quiz 9.15: Listing 9.10 의 line 26 의 `smp_mb()` 는 왜 필요한가요?



Quick Quiz 9.16: Listing 9.10 의 코드에서는 더 완화된 형태의 메모리 배리어를 사용할 순 없나요?



Quick Quiz 9.17: 시퀀스 락킹 업데이트 쓰레드가 읽기 쓰레드를 starve 시키는 걸 무엇이 방지하나요?



라인 31~36는 간단히 락을 획득하고 이 시퀀스 숫자를 증가시키며, 이 값 즈가가 이 호출자의 크리티컬 섹션 전으로 순서 지어지게 보장하는 메모리 배리어를 수행하는 `write_seqlock()` 을 보입니다. 라인 38~43는 이 호출자의 크리티컬 섹션이 라인 41에서의 시퀀스 숫자 증가 전으로 순서 지어지게 하는 메모리 배리어를 수행하고 이 락을 해제하는 `write_sequnlock()` 을 보입니다.

Listing 9.11: Sequence-Locked Pre-BSD Routing Table Lookup (BUGGY!!!)

```

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 = READ_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 (READ_ONCE(rep->re_freed))
30         abort();
31     ret = rep->iface;
32     if (read_seqretry(&sl, s))
33         goto retry;
34     return ret;
35 }

```

Quick Quiz 9.18: 만약 뭔가 다른게 쓰기 쓰레드들을
직렬화 시켜서 이 락이 필요 없게 만들면 어떤가요?

Quick Quiz 9.19: Listing 9.10의 라인 2의 seq는 왜
unsigned long 이 아니라 unsigned 인가요? 어쨌건,
unsigned 가 리눅스 커널에 충분히 좋다면, 모두에게도
충분히 좋아야 하지 않나요?

그래서 시퀀스 락킹이 Pre-BSD 라우팅 테이블에 적용되면 무슨 일이 벌어질까요? Listing 9.11은 데이터 구조와 route_lookup() 을, Listing 9.12은 route_add() 와 route_del() (route_seqlock.c) 을 보입니다. 이 구현은 또다시 앞의 섹션에서의 비슷했던 것과 비슷하므로, 차이점만 들여다보겠습니다.

Listing 9.11에서, 라인 5는 라인 29와 30에서 검사되는 ->re_freed 를 추가합니다. 라인 8은 라인 24과 33에서의 라인 17의 retry 라벨로 브랜칭 되고 route_lookup() 의 라인 18, 23, 그리고 32에 의해서 사용되는 시퀀스 락을 추가합니다. 그 영향은 업데이트와 동시에 수행되는 모든 탐색을 재시도 하는 것입니다.

Listing 9.12: Sequence-Locked Pre-BSD Routing Table Add/ Delete (BUGGY!!!)

```

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

```

Listing 9.12에서, 라인 11, 14, 23, 31, 그리고 39는 이 시퀀스 락을 잡고 놓으며, 라인 10과 33는 ->re_freed 을 다룹니다. 따라서 이 구현은 상당히 단순합니다.

이것은 또한 읽기 전용 워크로드에서 여전히 이상적인 것에 비해선 떨어지지만 더 잘 성능을 내는데, Figure 9.5에서 이를 볼 수 있습니다. 더 나쁜게, 메모리 해제 후 사용 문제에 취약합니다. 문제는 읽기 쓰레드가 이미 메모리 해제된 구조체를 read_seqretry() 가 동시에 업데이트를 경고하기 전에 액세스 함으로써 segmentation violation 을 일으킬 수도 있다는 겁니다.

Quick Quiz 9.20: 이 버그는 고쳐질 수 있나요? 달리 말하자면, 시퀀스 락을 동시에 추가, 삭제, 그리고 탐색을 지원하는 링크드 리스트를 보호하는 동기화 메커니즘으로만 시퀀스 락을 사용할 수 있나요?

시퀀스 락의 읽기쪽과 쓰기쪽 크리티컬 섹션 모두 트랜잭션들로 생각될 수 있으며, 시퀀스 락킹은 따라서 제한된 형태의, Section 17.2에서 다룬 트랜잭션

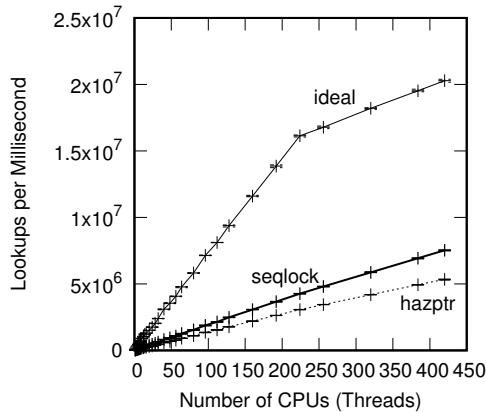


Figure 9.5: Pre-BSD Routing Table Protected by Sequence Locking

메모리 (transactional memory)로 생각할 수 있습니다. 시퀀스 락킹의 제한점들은: (1) 시퀀스 락킹은 업데이트를 제한하며 (2) 시퀀스 락킹은 업데이트 쓰레드에 의해 메모리 해제될 수도 있는 객체들로의 포인터들을 순회할 수 없습니다. 이 제한들은 물론 트랜잭션 메모리에 의해 극복됩니다만 시퀀스 락킹과 다른 동기화 기능들을 조합해서도 극복할 수 있습니다.

시퀀스 락은 쓰기 쓰레드가 읽기 쓰레드를 뒤로 미뤄지게 할 수 있고, 그 반대로 마찬가지입니다. 이는 unfairness와 심지어는 쓰기가 많은 워크로드에서의 starvation을 초래할 수 있습니다.³ 다른 한편, 쓰기 쓰레드가 없다면, 시퀀스 락 읽기 쓰레드는 합리적 수준으로 빠르고 선형적으로 확장합니다. 두 세계에서 최고를 바라는 건 사람 뿐입니다: 읽기 쪽 실패도, starvation도 가능하지 않은 빠른 읽기 쓰레드. 또한, 시퀀스 락킹의 포인터에 대한 제약도 극복할 수 있으면 좋을 겁니다. 다음 섹션은 정확히 이런 속성을 가진 동기화 메커니즘을 선보입니다.

9.5 Read-Copy Update (RCU)

“Free” is a *very* good price!

Tom Peterson

앞의 섹션들에서 다루어진 모든 메커니즘은 특정한 행동을 그것이 안전히 이루어질 수 있을 때까지 미루는 여러 전략 중 하나를 사용했습니다. Section 9.2에서

³ Dmitry Vyukov는 읽기 쓰레드 starvation을 줄일 수 있는 (하지만 슬프게도 제거할 수는 없는) 한가지 방법을 설명했습니다: <http://www.1024cores.net/home/lock-free-algorithms/reader-writer-problem/improved-lock-free-seqlock>.

다른 레퍼런스 카운터는 읽기 쓰레드를 방해할 수 있는 행동을 뒤로 미루기 위해 명시적 카운터를 사용했고 이는 읽기 쪽 컨텐션과 그로 인한 낮은 확장성의 결과를 가져왔습니다. Section 9.3은 쓰래드별 포인터 리스트의 모습을 한 암시적 카운터를 사용했습니다. 이는 읽기 쪽 컨텐션은 제거했으나, 읽기 쓰레드가 쓰기와 조건적 브랜치는 물론, 읽기 쪽 완전한 메모리 배리어 또는 리얼타임에 친화적이지 않은 프로세서간 인터럽트 (inter-processor interrupt) 기능을 사용해야 했습니다.⁴ Section 9.4에서 선보인 시퀀스 락 또한 읽기 쪽 컨텐션은 제거하나, 포인터 순회를 보호하지 않으며, 해저드 포인터와 같이, 읽기 쪽에서의 완전한 메모리 배리어 또는 업데이트 쪽에서의 프로세서간 인터럽트를 필요로 합니다. 이 방법들의 단점들은 더 나은 것은 불가능한지 질문을 던지게 합니다.

이 섹션은 *read-copy update (RCU)*를 소개하는데, 자주 업데이트 되는 공유 데이터로의 비싼 쓰기 없이 읽기 쓰레드가 소스 코드의 특정 영역과 연관지어지게 하는 API를 제공합니다. 이 섹션의 나머지 부분은 여러 다른 관점에서 RCU를 알아봅니다. Section 9.5.1은 고전적 RCU 소개를 제공하고, Section 9.5.2는 기본적인 RCU 컨셉을 다루며, Section 9.5.3은 RCU의 리눅스 커널 API를 선보이고, Section 9.5.4는 흔한 RCU 사용 예를 소개하고, Section 9.5.5는 RCU와 연관된 최근의 작업들을 다루고, Section 9.5.6는 일부 RCU 연습문제를 제공하고, 마지막으로 Section 9.7은 업데이트를 다룹니다.

9.5.1 Introduction to RCU

앞의 섹션들에서 이야기된 접근법들은 좋은 확장성을 제공하지만 분명 Pre-BSD 라우팅 테이블에 이상적이진 않은 성능을 제공했습니다. 따라서, “오직 너무 멀리 가본 사람만이 자기가 얼마나 멀리 갈 수 있는지 안다” 정신에 입각해,⁵ 동시의 업데이트의 존재에도 불구하고 동시의 읽기 쓰레드들이 싱글 쓰레드 텁색에서와 동일한 어셈블리 언어 인스트럭션들을 수행하는 알고리즘을 통해 더 멀리 가보겠습니다. 물론, 이 칭찬할 만한 목표는 심각한 구현 가능성 질문을 불러일으킬 수 있겠습니다만, 시도도 하지 않으면 성공할 수도 없습니다!

9.5.1.1 Minimal Insertion and Deletion

구현이 가능하긴 한지에 대한 걱정을 최소화하기 위해, NULL 이거나 하나의 구조체로의 참조일 하나의 글로

⁴ 어떤 중요한 특수 경우에는, 이 추가적 일이 Folly 오픈소스 라이브러리 (<https://github.com/facebook/folly>)에 구현되어 있는 UnboundedQueue와 ConcurrentHashMap 데이터 구조에서 보여지듯 연결 카운팅을 사용하는 것으로 제거될 수 있습니다.

⁵ T. S. Eliot에게 사과드립니다.

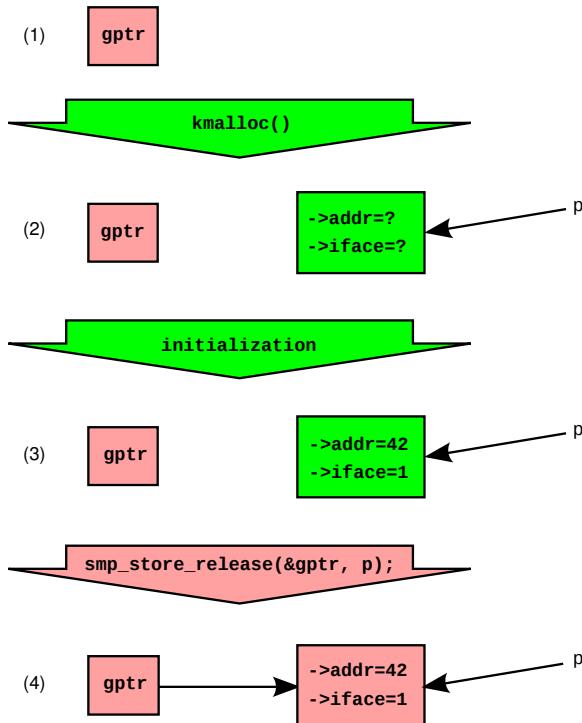


Figure 9.6: Insertion With Concurrent Readers

별 포인터로 존재하는 최소한의 데이터 구조에 집중해 봅니다. 이 데이터 구조는 최소한의 것이긴 하지만, 제품 단계에서 상당히 많이 사용되는 것이기도 합니다 [RH18]. 삽입을 위한 고전적 방법이 Figure 9.6에 보여져 있는데, 위에서 아래로 시간이 흐름에 따라 달라지는 네개의 상태를 보입니다. 첫번째 열은 최초의 상태를 보이는데, gptr이 NULL입니다. 두번째 열에서, 우리는 물음표로 보여지듯 초기화되지 않은 이 구조체를 할당합니다. 세번째 열에서, 우린 이 구조체를 초기화합니다. 마지막으로, 네번째 열에서 우린 gptr을 이 새로 할당되고 초기화된 원소를 참조하도록 업데이트합니다.

우린 이 gptr로의 값 할당이 간단한 C-언어의 할당문을 사용할 수 있길 바랄 겁니다. 불행히도, Section 4.3.4.1이 이 바람을 가로막습니다. 따라서, 업데이트 쓰레드는 간단한 C-언어 할당문 대신 이 그림에 보여진 것처럼 `smp_store_release()` 또는 뒤에서 보이겠지만 `rcu_assign_pointer()`를 사용해야 합니다.

비슷하게, 어떤 사람들은 읽기 쓰레드가 gptr의 값을 얻어오기 위해 하나의 C-언어 할당문을 사용할 수 있기를, 그리고 과거의 값인 NULL이나 새로이 설치된 포인터를 읽어와 어떤 경우든 유효한 값을 읽을 것이 보장되기를 바랄 겁니다. 불행히도, Section 4.3.4.1은 이

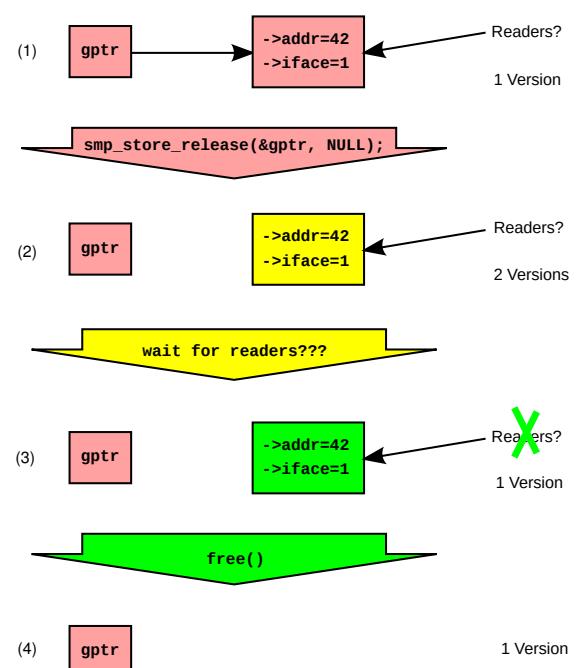


Figure 9.7: Deletion With Concurrent Readers

바람 역시 가로막습니다. 이 보장을 얻기 위해선, 읽기 쓰레드는 그 대신 `READ_ONCE()`, 또는, 뒤에서 보이겠지만 `rcu_dereference()`를 사용해야 합니다. 하지만, 대부분의 현대 컴퓨터 시스템에서, 이 읽기 쪽 기능들은 싱글 쓰레드 기반 코드에서 일반적으로 사용될 것과 동일한 하나의 로드 인스트럭션으로 구현될 수 있습니다.

Figure 9.6를 읽기 쓰레드의 시점에서 다시 보면, 앞의 세개 상태에서 모든 읽기 쓰레드는 gptr을 NULL 값을 갖는 것으로 봅니다. 네번째 상태에 진입하면서, 일부 읽기 쓰레드는 gptr이 여전히 NULL 값을 갖는다고 볼 수 있는 반면 다른 쓰레드들은 새로 설치된 원소를 참조하고 있는 것으로 볼 수 있지만, 어떤 시점 이후로는 모든 읽기 쓰레드가 이 새 원소를 보게 될 겁니다. 항상, 모든 읽기 쓰레드는 gptr을 유효한 포인터를 담고 있다고 볼 겁니다. 따라서, 동시의 읽기 쓰레드들이 싱글 쓰레드 기반 코드에서 일반적으로 사용할 것과 동일한 기계 인스트럭션들을 수행하면서 새로운 데이터를 연결된 데이터 구조에 추가하는 것이 정말 가능합니다. 이 동시 읽기기에의 비용이 없는 방법은 훌륭한 성능과 확장성을 제공하며, 리얼타임 사용처에서 뛰어나게 잘 사용될 수 있습니다.

삽입은 물론 상당히 유용합니다만, 금방이든 더 나중이든, 데이터를 지워야 하기도 할 겁니다. Figure 9.7에 보이듯, 첫번째 단계는 쉽습니다. Section 4.3.4.1에서의 교훈을 다시 상기해 보면, `smp_store_release()` 가

이 포인터를 NULL로 만들기 위해 사용되어, 이 그림의 첫번째 열에서 두번째 열로 넘어갑니다. 이 시점에서, 기존부터 존재하던 읽기 쓰레드는 `->addr`는 42 값을 그리고 `->iface`는 1 값을 갖는 것으로 볼 수 있지만, 새로 시작된 읽기 쓰레드는 NULL 포인터를 볼 것으로, 즉 동시에 읽기 쓰레드들이 이 그림의 “2 Versions”로 표시된 것처럼 현 상태에 대해 다른 의견을 가질 수 있습니다.

Quick Quiz 9.21: Figure 9.7는 NULL 포인터를 저장하는데 왜 `smp_store_release()`를 사용하나요? NULL 포인터 저장에 대해 순서지울 구조체 초기화가 없다는 점을 감안하면 `WRITE_ONCE()` 만으로도 이 경우에는 괜찮지 않을까요?



Quick Quiz 9.22: 동시에 수행되는 읽기 쓰레드는 Figure 9.7에 그려진 수행 순서에 따르면 `gptr`의 값에 대해 동의하지 않을 수 있습니다. 이건 뭔가 문제있지 않나요???



우린 열 3에서 볼 수 있듯 모든 기존부터 존재한 읽기 쓰레드들이 완료하길 기다리는 것만으로 단일 버전으로 돌아올 수 있습니다. 이 지점에서, 모든 기존부터 존재하던 읽기 쓰레드는 종료되었고, 뒤따르는 읽기 쓰레드는 어느 것도 기존 데이터 아이템으로의 경로를 갖지 못하므로, 그걸 참조하는 어떤 읽기 쓰레드도 존재할 수 없습니다. 따라서 그 아이템은 열 4에서 보여지듯 안전히 메모리 해제될 수 있습니다.

따라서, 기존부터 존재한 읽기 쓰레드들이 완료되길 기다릴 방법이 존재한다면 읽기 쓰레드들이 싱글 쓰레드 수행 시에나 적합할 것과 동일한 기계 인스트럭션들만을 수행하면서도 연결된 데이터 구조에 데이터를 추가할 수도 삭제할 수도 있습니다. 그러나 앞서 했던 것들이 너무 멀리 간 것만은 아닐 수도 있습니다!

하지만 어떻게 기존부터 존재한 읽기 쓰레드들이 실제로 완료되었다고 이야기 할 수 있을까요? 이 질문이 다음 섹션의 주제입니다.

9.5.1.2 Waiting for Readers

읽기 쓰레드의 기다리기를 레퍼런스 카운팅 기반으로 하고 싶을 수 있겠습니다만, Chapter 5의 Figure 5.1은 현재의 레퍼런스 카운팅은 Section 9.2에서 이미 보았듯 굉장히 오버헤드를 초래함을 보았습니다. 해저드 포인터는 이 오버헤드를 상당히 줄입니다만, 우리가 Section 9.3에서 보았듯이, 완전히 없애진 못합니다. 그렇다고는 하나, 많은 RCU 구현이 매우 주의 깊은 캐시 지역적 카운터의 사용을 합니다.

두번째 접근법은 메모리 동기화가 비쌈을 파악하고, 따라서 레지스터를 대신 사용하는데, 이는 각 CPU 또는

쓰레드의 프로그램 카운터 (PC)로, 따라서 읽기 쓰레드에게는 적어도 동시의 업데이트의 부재 시에는 오버헤드를 일으키지 않습니다. 업데이트 쓰레드는 각 적절한 PC를 반복적으로 살펴보고, 그 PC가 읽기 쪽 코드 내에 위치해 있지 않다면, 연관된 CPU 또는 쓰레드는 조용한 상태 (quiescent state)에 있는 것이며, 따라서 이는 새로이 제거된 데이터 원소로의 액세스를 할 수도 있는 읽기 쓰레드는 모두 완료되었다는 신호가 됩니다. 모든 CPU 또는 쓰레드의 PC가 모두 모든 읽기 쓰레드의 바깥에 있음이 확인된다면, 이 유예 기간 (grace period)가 완료됩니다. 이 방법은 몇 가지 심각한 기술적 어려움도 가지고 있는데, 메모리 순서 맞추기, 읽기 쓰레드에 의해 가끔 호출되는 함수들, 그리고 항상 신나는 코드 모습 최적화가 포함됩니다. 그러나, 이 방법은 제품 단계에서 사용되었다고 이야기 됩니다 [Ash15].

세번째 접근법은 모든 합리적 읽기 쓰레드의 수명을 충분히 넘어설 정도로 긴 고정된 시간동안 그냥 기다리는 것입니다 [Jac93, Joh95]. 이는 하드 리얼타임 시스템에서는 잘 동작합니다만 [RLPB18], 그보다 덜 진귀한 환경에서라면, Murphy는 비합리적일 정도로 오래 살아있는 읽기 쓰레드에도 준비해야 하는게 무척 중요하다고 말합니다. 이를 자세히 보기 위해, 그러는데 실패했을 때의 결과를 생각해 봅시다: 어떤 데이터 항목은 이 비합리적인 읽기 쓰레드가 여전히 그것을 참조하고 있는 동안 메모리 해제될 수 있고, 그 항목은 곧바로 재할당되어 심지어 다른 타입의 데이터 항목으로 사용되고 있을 수 있습니다. 그러면 이 비합리적 읽기 쓰레드와 부주의한 재할당자는 같은 메모리를 두개의 매우 다른 목적으로 사용하려 할 것입니다. 그 결과는 디버깅 하기에 무척 어려울 것입니다.

네번째 접근법은 영원히 기다리는 것으로, 그렇게 하게 가장 비합리적인 읽기 쓰레드조차도 처리해 줄 것이라는 점에서 안전합니다. 이 접근법은 또한 “메모리 누출”이라고도 불리며, 메모리 누출은 시기 상조의 그리고 불편한 리부팅을 요구한다는 사실 때문에 나쁜 평판을 가지고 있습니다. 그러나, 이는 업데이트 비율과 시스템 가동시간이 모두 꽤 정확히 그 한계가 정해져 있을 때라면 사용될 수 있는 전략입니다. 예를 들어, 이 방법은 시스템이 높은 가용성을 가진 클러스터여서 이 클러스터가 정말로 높은 가용성을 유지한다는 것을 분명히 하기 위해 주기적으로 고장나는 것이라면 잘 동작할 수 있습니다.⁶ 메모리를 누출하는 것은 또한 garbage collector를 가진 경우에도 사용될 수 있을텐데, 이 경우는 garbage collector가 이 누출을 막는 것으로 생각될 수 있습니다 [KL80]. 하지만, 여러분의 환경이 garbage collector를 지원하지 않는다면, 마저 읽으십시오!

⁶ 이 주기적 고장을 강제하는 프로그램은 가끔 “chaos monkey”라고 불립니다: <https://netflix.github.io/chaosmonkey/>. 하지만, 너무 오래 수행되는 시스템에 의해 야기되는 혼란을 무시하는 것도 실수가 될수도 있습니다.

다섯번째 접근법은 전통적 세상을-멈추기 (stop-the-world) garbage collector에 의해 예시되는 주기적으로 “세상을 멈추기”를 사용해 주기적 고장내기를 막습니다. 이 방법은 또한 각 일하는 날의 끝마다 시스템의 전원을 끄는게 일반적 행동이었던, 어디나 있는 연결성 전의 수십년간 많이 사용되었습니다. 하지만, 오늘날의 항상 연결되어 있고 항상 켜져있는 세상에서는, 세상을 멈추기는 응답 시간을 크게 악화시킬 수 있는데, 이는 동시적 garbage collector의 개발의 모티베이션 중 하나가 되었습니다 [BCR03]. 더 나아가, 우린 모든 앞서서부터 존재하고 있는 읽기 쓰레드가 모두 완료되기를 기다려야 하기는 하지만, 그것들이 동시에 완료되어야 할 필요는 없습니다.

이 발견은 여섯번째 접근법을 이끌어내는데, 한번에 한 CPU 또는 쓰레드를 멈추는 것입니다. 이 방법은 읽기 쓰레드의 응답 시간을 전혀 악화시키지 않는다는 장점을 갖습니다. 더 나아가서, 많은 어플리케이션들이 이미 앞의 앞서서부터 존재하고 있는 읽기 스레드들이 모두 완료된 후에만 도달할 수 있는 상태 (*quiescent state* 라 불립니다)를 갖습니다. 트랜잭션 처리 시스템에서는, 한 쌍의 연속된 트랜잭션 사이의 시간이 *quiescent state* 일 수 있습니다. 반응형 시스템에서는, 연속된 한 쌍의 이벤트 사이의 상태가 *quiescent state* 일 겁니다. Preemption 기반이지 않은 운영체제 커널에서는, 컨텍스트 스위치가 *quiescent state* 가 될 수 있습니다 [MS98a]. 어떤 형태든, 모든 CPU 그리고/또는 쓰레드가 하나의 *quiescent state* 를 지난다면, 이 시스템은 하나의 *grace period* 를 완료했다고 이야기 되며, 이 지점에서는 이 *grace period* 의 시작 시점에서 존재했던 모든 읽기 쓰레드는 완료되었다는게 보장됩니다. 그 결과, 이 *grace period* 의 시작 전에 제거된 데이터 항목들은 메모리 해제되기에 안전합니다.⁷

Preemption이 없는 운영체제 커널에서 컨텍스트 스위치가 유효한 *quiescent state* 가 되려면 읽기 쓰레드는 Figure 9.6 와 9.7의 *gptr* 포인터에 의해 얻어지는 데이터 구조 인스턴스를 참조하는 동안은 블록킹 되는 걸 방지해야만 합니다. 이 블록킹 방지 제약은 순수 스핀락에서도 비슷한 제약으로 존재하는데, 여기선 CPU는 스핀락을 잡고 있는 동안 블록킹 되는게 금지됩니다. 이 제약이 없다면, 모든 CPU는 블록된 쓰레드에 의해 잡혀 있는 스핀락을 획득하려 스핀하는 쓰레드들에 의해 소모될 수 있습니다. 이 스핀하는 쓰레드들은 그것들이 이 락을 획득하기 전까지는 CPU를 놓지 않을 것이지만, 이 락을 쥐고 있는 쓰레드는 이 스핀하고 있는 쓰레드들 중 하나가 CPU를 하나라도 놓기 전까지는 이 락을 놓아주지 못할 겁니다. 이는 고전적 데드락 환경이며, 이

⁷ RCU는 단순히 메모리 회수 미루기보다 훨씬 많은 걸 할 수 있으나, 회수 미루기는 RCU의 가장 흔한 사용 예이며, 따라서 시작하기에 훌륭한 지점입니다.

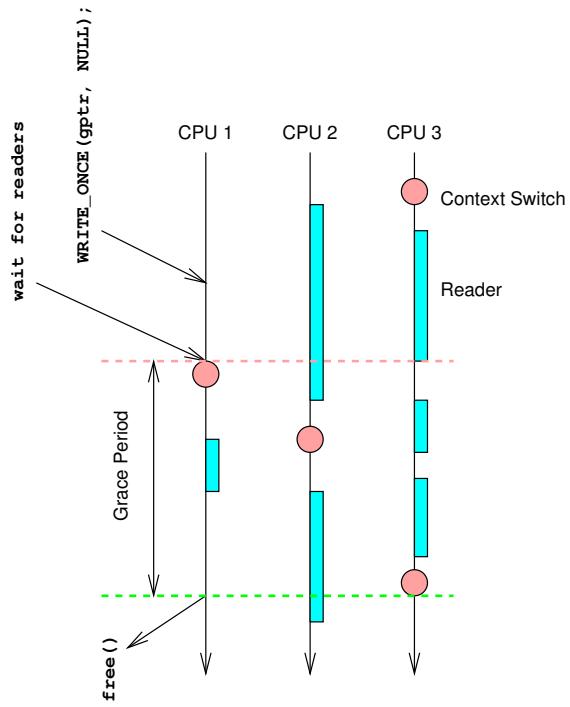


Figure 9.8: QSBR: Waiting for Pre-Existing Readers

데드락은 스핀락을 잡고 있는 동안은 블록킹을 방지하는 것으로 막아집니다.

다시, 이 동일한 제약이 *gptr* 을 역참조하는 읽기 쓰레드에게도 적용됩니다: 그런 쓰레드는 그것들이 이 가리켜진 데이터 항목을 사용하는 것을 마무리 하기 전까지는 블록되지 않아야 합니다. 업데이트 쓰레드가 막 *smp_store_release()* 를 수행하는 것을 완료한 Figure 9.7의 두번째 열로 돌아와서, CPU 0이 컨텍스트 스위치를 했다고 생각해 봅시다. 읽기 쓰레드는 이 링크드 리스트를 순회하는 동안은 블록되는 것이 허용되지 않았으므로, 우린 CPU 0에서 수행되었을 수도 있는 모든 앞의 읽기 쓰레드들이 완료되었다고 보장됩니다. 이 논리를 다른 CPU 들로 확장하면, 각 CPU 가 모두 컨텍스트 스위치 수행을 목격했다면, 모든 앞의 읽기 쓰레드가 완료되었음이, 새로이 제거된 데이터 원소를 참조하고 있는 어떤 읽기 쓰레드도 더이상 존재하지 않음이 보장됩니다. 그럼 이 업데이트 쓰레드는 안전하게 이 데이터 원소를 메모리 해제할 수 있어, Figure 9.7의 가장 아래에 보인 상태에 도달합니다.

이 방법은 *quiescent state based reclamation* (QSBR) [HMB06] 라고 명명되었습니다. 하나의 QSBR 방법이 Figure 9.8에 시간이 이 그림의 꼭대기에서 아래로 흐르는 모습으로 그려져 있습니다. CPU 1은 현재 데이터 항목을 제거하는 *WRITE_ONCE()* 를 수행하고

(아마도 이 포인터 값을 앞서 읽었고 자신을 적절한 동기화를 하고 있게 했을 겁니다), 읽기 쓰레드를 기다립니다. 이 대기 오퍼레이션은 즉각적인 컨텍스트 스위치를 초래하는데, 이는 quiescent state 로 (핑크색 원으로 표시되어 있습니다), CPU 1 에서의 모든 앞선 읽기는 완료되었음을 의미합니다. 이어서, CPU 2 가 컨텍스트 스위치를 하여, CPU 1 과 2 의 모든 읽기 쓰레드는 이제 완료되었음이 알려집니다. 마지막으로, CPU 3 이 컨텍스트 스위치를 합니다. 이 지점에서, 전체 시스템의 모든 읽기 쓰레드는 완료되었음이 알려지며, 따라서 grace period 가 종료되어 CPU 1 이 오래된 데이터 항목을 메모리 해제하는 것을 허용합니다.

Quick Quiz 9.23: Figure 9.8 에서, CPU 3 의 이 오래된 데이터 항목으로의 액세스를 했을 수 있는 읽기 쓰레드는 이 grace period 가 시작하기도 전에 이미 완료되었어요! 그런데 왜 CPU 3 의 마지막 컨텍스트 스위치를 신경쓰나요???



9.5.1.3 Toy Implementation

제품 수준 QSBP 구현은 무척 복잡할 수 있지만, preemption 기반이 아닌 리눅스 커널에서의 장난감 수준 구현은 굉장히 간단합니다:

```

1 void synchronize_rcu(void)
2 {
3     int cpu;
4
5     for_each_online_cpu(cpu)
6         sched_setaffinity(current->pid, cpumask_of(cpu));
7 }
```

`for_each_online_cpu()` 기능은 모든 CPU 를 순회하고, `sched_setaffinity()` 함수는 현재 쓰레드가 명시된 CPU 에서 수행되게 하는데, 이는 이 대상 CPU 가 컨텍스트 스위치를 수행하게끔 강제합니다. 따라서, `for_each_online_cpu()` 가 완료되고 나면, 각 CPU 는 한번의 컨텍스트 스위치는 수행한 것인데, 이는 곧 모든 앞서서부터 존재해온 읽기 쓰레드가 모두 완료되었음을 보장합니다.

이 방법은 제품 수준이 아님을 알아 두시기 바랍니다. 여러개의 특수 상황에 대한 처리와 여러개의 강력한 최적화의 필요는 제품 수준 구현이 무척 복잡함을 의미합니다. 이에 더해, preemption 기반 환경을 위한 RCU 구현은 읽기 쓰레드가 실제로 무언가를 더할 것을 필요로 하는데, 리얼타임이 아닌 리눅스 커널 환경에서는 `rcu_read_lock()` 과 `rcu_read_unlock()` 을 각각 `preempt_disable()` 과 `preempt_enable()` 로 간단히 정의하는 것으로 될 수 있습니다.⁸ 그러나, 이

⁸ Preemption 되는 읽기쪽 크리티컬 섹션을 다루는 어떤 장난감 수준 RCU 구현들이 Appendix B 에 보여져 있습니다.

Listing 9.13: Insertion and Deletion With Concurrent Readers

```

1 struct route *gptr;
2
3 int access_route(int (*f)(struct route *rp))
4 {
5     int ret = -1;
6     struct route *rp;
7
8     rCU_read_lock();
9     rp = rCU_dereference(gptr);
10    if (rp)
11        ret = f(rp);
12    rCU_read_unlock();
13    return ret;
14 }
15
16 struct route *ins_route(struct route *rp)
17 {
18     struct route *old_rp;
19
20     spin_lock(&route_lock);
21     old_rp = gptr;
22     rCU_assign_pointer(gptr, rp);
23     spin_unlock(&route_lock);
24     return old_rp;
25 }
26
27 int del_route(void)
28 {
29     struct route *old_rp;
30
31     spin_lock(&route_lock);
32     old_rp = gptr;
33     RCU_INIT_POINTER(gptr, NULL);
34     spin_unlock(&route_lock);
35     synchronize_rcu();
36     free(old_rp);
37     return !old_rp;
38 }
```

간단한 preemption 불가 방법은 이론적으로 완벽하며, 읽기 쪽 동기화를 동시의 업데이트가 있음에도 불구하고 비용 없이 제공하는게 가능함을 보입니다. 실제로, Listing 9.13 은 어떻게 읽기 (`access_route()`), Figure 9.6 의 삽입, (`ins_route()`) Figure 9.7 의 삭제가 (`del_route()`) 구현될 수 있는지 보입니다. (약간 더 그럴싸한 라우팅 테이블은 Section 9.5.4.1 에 보여져 있습니다.)

Quick Quiz 9.24: Listing 9.13 의 `rcu_read_lock()` 과 `rcu_read_unlock()` 의 요점이 뭔가요? 이 quiescent state 들이 스스로 자신을 이야기하게 하는건 어떤가요?



Quick Quiz 9.25: Listing 9.13 의 `rcu_dereference()`, `rcu_assign_pointer()`, 그리고 `RCU_INIT_POINTER()` 의 요점이 뭔가요? 그냥 `READ_ONCE()`, `smp_store_release()`, 그리고 `WRITE_ONCE()` 를 각각 대신 사용하는 건 어떤가요?



Listing 9.13 으로 돌아가서, `route_lock` 이 `ins_route()` 와 `del_route()` 를 호출하는 동시의 업데이트들 사이의 동기화를 위해 사용되었음을 알아두시기

바랍니다. 하지만, 이 락은 `access_route()` 를 호출하는 읽기 쓰레드에 의해서는 획득되지 않습니다: 읽기 쓰레드들은 그대신 Section 9.5.1.2 에서 이야기 된 QSBR 기법을 통해 보호됩니다.

`ins_route()` 는 Figure 9.6 가 항상 NULL 일 거라고 가정하는 `gptr` 의 기존 값을 리턴할 뿐임을 알아 두시기 바랍니다. 이는 NULL 이 아닌 값을 가지고 뭘 할지 알아내는 건 호출자의 책임이며, 읽기 쓰레드들이 확정되지 않은 기간동안 그에 대한 참조를 하고 있을 수도 있다는 사실에 의해 복잡해지는 일입니다. 호출자는 다음 접근법 중 하나를 사용할 수도 있을 겁니다:

1. 가리켜진 구조체의 안전한 메모리 해제를 위해 `synchronize_rcu()` 를 사용합니다. 이 방법은 RCU 관점에서는 올바르지만, 소프트웨어 엔지니어링 관점에서는 새기 쉬운 API 문제를 갖는다고 논쟁될 수도 있습니다.
2. 리턴된 포인터가 NULL 이 아닌지 단정을 짓습니다.
3. 이전의 값을 복원하기 위해 `ins_route()` 의 다음 호출로 리턴된 포인터를 넘깁니다.

대조적으로, `del_route()` 는 새로 삭제된 데이터 항목을 안전하게 메모리 해제하기 위해 `synchronize_rcu()` 와 `free()` 를 사용합니다.

Quick Quiz 9.26: 하지만 기존 구조체가 메모리 해제되어야 하지만 `ins_route()` 의 호출자가 성능에 대한 고려 때문 또는 이 호출자가 RCU 읽기 크리티컬 섹션 내에서 수행되고 있다던지 해서 블록될 수 없다면 어떻게 하죠?

■ 이 예는 RCU 로 보호되는 데이터 구조를 읽고 업데이트 하는 하나의 일반적 접근법을 보이고 있습니다만, 매우 다양한 사용 예가 존재하며, 많은 것들이 Section 9.5.4 에서 다뤄집니다.

요약하자면, 싱글쓰레드 기반 읽기 함수에 의해 수행될 것과 동일한 구성의 기계 인스트럭션을 수행하는 읽기 쓰레드에 의해 순회될 수 있는 연결된 동시적 데이터 구조들을 만드는게 정말 가능합니다. 다음 섹션은 RCU 의 고수준 특성들을 요약합니다.

9.5.1.4 RCU Properties

RCU 의 핵심 속성은 읽기가 업데이트를 기다릴 필요 없다는 것입니다. 이 속성은 RCU 구현이 낮은, 또는 심지어 전혀 없는 비용을 읽기 쓰레드에게 제공할 수 있게 하여, 낮은 오버헤드와 훌륭한 확장성을 가능하게 합니다. 이 속성은 RCU 읽기 쓰레드들과 업데이트 쓰레드들이 유용한 동시의 진행을 할 수 있게 합니다. 대비적으로, 전통적인 동기화 기능들은 비용이 높은 명령들

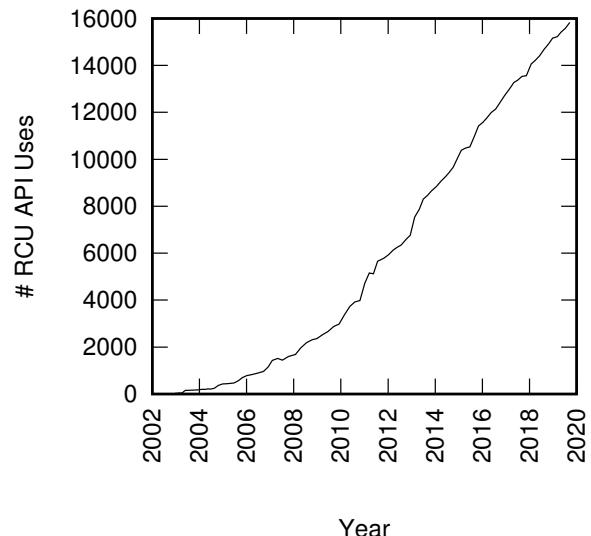


Figure 9.9: RCU Usage in the Linux Kernel

을 사용해 엄격한 상호배제를 강요해야만 하여 오버헤드를 높이고 확장성을 떨어뜨리지만 또한 일반적으로 읽기 쓰레드들과 업데이트 쓰레드들이 유용한 동시의 진행을 할 수 없게 만듭니다.

Quick Quiz 9.27: Section 9.4 의 seqlock 또한 읽기 쓰레드들과 업데이트 쓰레드들이 유용한 동시의 진행을 할 수 있게 하지 않나요?



RCU 는 `rcu_read_lock()` 과 `rcu_read_unlock()` 기능을 이용해 읽기 쓰레드들의 범위를 지정하고 객체들의 여러 버전을 관리해 각 읽기 쓰레드가 각 객체의 일관적인 모습을 볼 것을 보장해 주고 `synchronize_rcu()` 와 같은 업데이트 쪽 기능들을 이용해 객체들이 그것을 사용하고 있을 수 있는 모든 읽기 쓰레드들이 완료되기 전까지는 메모리 해제되지 않게 보장합니다. RCU 는 객체의 새로운 버전을 계산하고 읽기 위한 효율적이고 확장성 있는 메커니즘을 제공하기 위해 `rcu_assign_pointer()` 와 `rcu_dereference()` 를 사용합니다. 이 메커니즘들은 해저드 포인터와 유사한 복사와 규칙 완화 최적화를 사용하지만 읽기 쪽 재시도는 없게 하여 읽기 경로는 극단적으로 빠르게 하면서 일거리를 읽기 쪽과 쓰기 쪽으로 나눕니다. `CONFIG_PREEMPT=n` 리눅스 커널을 포함한 어떤 경우에는 RCU 의 읽기 쪽 기능들은 오버헤드를 아예 갖지 않습니다.

하지만 이 속성이 실제 상황에서도 유용할까요? 이 질문이 다음 섹션에서 답해집니다.

9.5.1.5 Practical Applicability

RCU 는 2002sus 10월부터 리눅스 커널에서 사용되었습니다 [Tor02]. RCU API 의 사용은 Figure 9.9 에서 볼 수 있듯, 그 후로 상당히 증가했습니다. 실제로, Listing 9.13 의 것과 매우 비슷한 코드가 리눅스 커널에서 사용되고 있습니다. RCU 는 Section 9.5.5 에서 이야기 되듯 리눅스 커널에 의해 받아들여지기 전에도 후에도 널리 사용되어왔습니다.

따라서 RCU 가 폭넓은 실용적 사용가능성이 있다고 말해도 안전할 겁니다.

이 섹션에서 이야기 된 최소의 예는 RCU에 대한 좋은 소개가 됩니다. 하지만, RCU의 효과적인 사용을 위해선 여러분이 문제를 다른 시각으로 볼 것을 필요로 하곤 합니다. 따라서 RCU의 기본을 알아보는게 유용한데, 이를 다음 섹션에서 해봅니다.

9.5.2 RCU Fundamentals

이 섹션은 앞의 섹션에서 다루어졌지만 특정 예나 사용 경우에 독립적인 바닥의 것들을 다시 살펴봅니다. 실제 코드와 무척 가까운 삶을 사는 것을 선호하는 사람들은 이 섹션에서 다루는 기본적인 것들은 생략하고 싶을 겁니다.

RCU 는 세개의 기본적 메커니즘을 통해 만들어지는 데, 첫번째 것은 삽입을 위해, 두번째 것은 삭제를 위해, 그리고 세번째 것은 읽기 쓰레드가 동시에 삽입과 삭제를 견딜 수 있게 하는데에 사용됩니다. Section 9.5.2.1 은 삽입을 위해 사용되는 게시-구독 메커니즘을, Section 9.5.2.2 은 앞서서부터 존재하고 있는 RCU 읽기 쓰레드들을 기다리는 것이 삭제를 가능하게 하는지, 그리고 Section 9.5.2.3 은 최근에 업데이트된 객체들의 여러 버전을 유지하는 것이 어떻게 동시에 삽입과 삭제를 허용하는지 이야기 합니다. 마지막으로, Section 9.5.2.4 은 RCU 의 기본을 요약합니다.

9.5.2.1 Publish-Subscribe Mechanism

RCU 읽기 쓰레드들은 RCU 업데이트 쓰레드들로부터 배제되지 않기 때문에, RCU로 보호되는 데이터 구조는 읽기 쓰레드가 그것을 액세스하고 있는 동안에도 변경될 수도 있습니다. 이 액세스된 데이터 항목은 옮겨졌거나, 삭제되었거나, 교체되었을 수도 있습니다. 데이터 구조는 읽기 쓰레드가 존재하는 동안 “기다려주지” 않기 때문에, 각 읽기 쓰레드의 액세스는 이 RCU로 보호되는 데이터 항목의 현재 버전을 구독하고 있는 것으로 생각될 수 있습니다. 이에 대해 업데이트 쓰레드의 동작은 새로운 버전을 게재하는 것으로 생각될 수 있습니다.

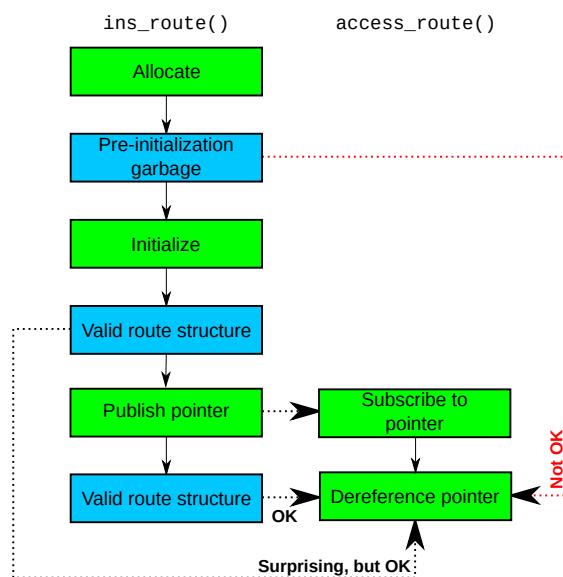


Figure 9.10: Publication/Subscription Constraints

불행히도, Section 4.3.4.1에서 이야기 되었고 Section 9.5.1.1에서 반복해 이야기 되었듯, 이런 게재와 구독 오퍼레이션에 평범한 액세스를 사용하는 것은 혼명치 못합니다. 그 대신 컴파일러와 CPU 모두에게 주의를 요함을 알릴 필요가 있는데, Listing 9.13의 `ins_route()` (와 그 호출자)와 `read_gptr()`이 동시에 수행될 때의 상호작용을 그린 Figure 9.10이 이를 잘 보이고 있습니다.

Figure 9.10 의 `ins_route()` 행은 초기화 전의 쓰레기 값을 갖는 새 `route` 구조체를 메모리 할당하는, `ins_route()` 의 호출자를 보입니다. 이 호출자는 이어서 새로 할당된 구조체를 초기화 하고, 이 새로운 `route` 구조체로의 포인터를 재하는 `ins_route()` 를 호출합니다. 재하는 이 구조체의 내용에 영향을 끼치지 않으며, 따라서 재하는 후에도 유효한 것이 됩니다.

이 그림의 `access_route()` 행은 구독되고 역참조되는 포인터를 보입니다. 이 역참조 오퍼레이션은 당연히 초기화 전의 쓰레기값이 아닌 유효한 `route` 구조체를 봐야 하는데 쓰레기 값을 역참조하는 것은 메모리 오염, 시스템 깨짐, 그리고 시스템 정지를 유발할 수 있기 때문입니다. 앞에서 이야기 되었듯, 그런 쓰레기 값을 방지하는 것은 게재와 구독 오퍼레이션이 컴파일러와 CPU 모두에게 필요한 순서를 지켜줄 것을 알려야 함을 의미합니다.

게재는 `ins_route()` 의 호출자의 초기화가 실제 게재 오퍼레이션의 포인터 저장 전으로 순서지어질 것을 보장하는 `rcu_assign_pointer()` 에 의해 수행됩니다. 또한, `rcu_assign_pointer()` 는 동시의 읽기 쓰

레드가 이 포인터의 기존 값이나 새 값만 보지, 이 두 값의 어떤 결합된 형태를 보지는 않는다는 의미로 어토믹해야만 합니다. 이 요구사항들은 C11의 store-release 오퍼레이션에 의해 지켜지며, 실제로 리눅스 커널에서 `rcu_assign_pointer()`는 C11의 store-release 와 비슷한 `smp_store_release()`를 사용해 정의됩니다.

동시의 업데이트들이 필요하다면 같은 포인터로의 동시적인 `rcu_assign_pointer()` 호출을 중재하기 위해 어떤 종류의 동기화 메커니즘이 필요함을 알아두시기 바랍니다. 리눅스 커널에서, 락킹은 그러기 위해 선택됩니다만, 어떤 동기화 메커니즘이든 사용될 수 있습니다. 특히 가벼운 동기화 메커니즘의 예는 Chapter 8의 데이터 소유권입니다: 각 포인터가 특정 쓰레드에 의해 소유된다면, 해당 쓰레드는 해당 포인터로의 `rcu_assign_pointer()`를 추가적인 동기화 오버헤드 없이 수행할 수 있습니다.

Quick Quiz 9.28: RCU 업데이트 쓰레드를 위한 데이터 소유권의 사용은 이 업데이트들이 연관된 단일쓰레드 기반 코드의 것과 정확히 같은 명령들을 사용해 수행될 수 있음을 의미하지 않나요?

■ 구독은 해당 포인터의 읽기가 역참조보다 먼저 수행될 것을 순서잡는 `rcu_dereference()`에 의해 수행됩니다. `rcu_assign_pointer()` 와 비슷하게, `rcu_dereference()`는 읽혀진 값이 하나의 스토어에 의해서 만들어진 것이어야 한다는 점에서 원자적이어야 하는데, 예를 들어, 컴파일러는 이 읽기를 조개 수행하지 않아야 합니다.⁹ 불행히도, 컴파일러의 `rcu_dereference()` 지원은 여전히 작업중입니다 [MWB¹⁷, MRP¹⁷, BM18]. 그 중에는 리눅스 커널은 volatile 로드, 다양한 CPU 아키텍쳐의 세부 사항, 코딩 제약 [McK14c] 에, 그리고 DEC Alpha [Cor02]의 경우에는 메모리 배리어 인스트럭션에 의존해야 합니다. 하지만, 다른 아키텍쳐에서는 `rcu_dereference()`가 하나의 로드 인스트럭션만을 만들어 내어서, 동일한 단일쓰레드 기반 코드와 같아집니다. 코딩 제약은 Section 15.3.2에서 자세히 다루어집니다만, 필드 선택 (“->”)의 흔한 경우가 잘 동작합니다. 읽기 쪽 궁극적 성능을 필요로 하지 않는 소프트웨어라면 그 대신 비용이 있긴 하지만 필요한 순서 규칙을 제공하는 C11 `acquire load`를 대신 사용할 수도 있습니다. 조만간 `rcu_dereference()`의 더 가벼운 컴파일러 지원이 나오길 많은 사람들이 바라고 있습니다.

요약하자면, 포인터의 재생성을 위한 `rcu_assign_pointer()`의 사용과 그것을 구독하기 위한 `rcu_dereference()`의 사용은 Figure 9.10에 그려진 “Not OK” 쓰레기 읽기를 방지합니다. 따라서 이 두개의 기능

은 동기의 읽기를 막치지 않으면서 연결된 구조체들에 새 데이터를 추가할 수 있습니다.

Quick Quiz 9.29: 하지만 어떤 읽기 쓰레드가 어떤 링크드 리스트를 순회하는 동안 업데이트 쓰레드들이 여러 데이터 아이템들을 이 리스트에 넣고 빼고 있다고 생각해 봅시다. 특히, 이 리스트가 초기에는 원소 A, B, 그리고 C를 가지고 있고 어느 업데이트 쓰레드가 원소 A를 삭제하고 이어서 새 원소 D를 이 리스트의 끝에 추가했다고 해 봅시다. 읽기 쓰레드는 {A, B, C, D}를 볼 수 있는데, 그런 원소의 연속은 실제로 존재한 적이 없는 것입니다! 어떤 대안적 우주에서는 그게 “동시의 읽기 쓰레드를 방해하지 않는다”라고 여겨집니까???

■

읽기 쓰레드들을 방해하지 않으면서 연결된 구조에 데이터를 추가하는 것은 좋은 일입니다. 이는 싱글쓰레드 기반 읽기 쓰레드에 비해 추가적인 읽기 쪽 비용을 높이지 않으면서도 행해질 수 있으니까요. 하지만, 많은 경우에 데이터의 삭제도 필요한데, 이게 다음 섹션의 주제입니다.

9.5.2.2 Wait For Pre-Existing RCU Readers

그 가장 기본적인 형태에서, RCU 는 일이 끝나길 기다리는 방법입니다. 물론, 일이 끝나길 기다리기 위한 수많은 다른 방법들이 존재하는데, 레퍼런스 카운팅, reader-writer 락, 디벤트, 등등이 포함됩니다. RCU 의 큰 장점은 (대충 말하자면) 20,000 개의 다른 것들을 명시적으로 그들을 각자를 추적할 필요 없이, 그리고 명시적 추적 방식에서는 피할 수 없는 성능 하락, 확장성 한계, 복잡한 데드락 시나리오, 그리고 메모리 누수 문제 없이 할 수 있다는 것입니다.

RCU의 경우, 기다려야 하는 것들 각각은 RCU 읽기 크리티컬 섹션이라고 불립니다. Section 9.5.1.3에서 힌트가 주어진 것처럼, RCU 읽기 크리티컬 섹션은 `rcu_read_lock()` 기능으로 시작하고 연관된 `rcu_read_unlock()` 기능으로 완료됩니다. RCU 읽기 크리티컬 섹션은 중첩될 수 있고, quiescent state를 포함하지 않는다면 열만큼의 코드든 포함할 수 있는데, 예를 들어 리눅스 커널에서는 컨텍스트스위치가 하나의 quiescent state이기 때문에 RCU 읽기 크리티컬 섹션 내에서 잠드는 게 불법입니다.¹⁰ 여러분이 이 규칙을 따른다면, 여러분은 앞서서부터 존재해온 RCU 읽기 크리티컬 섹션을 얼마나 드지 완료되길 기다릴 수 있으며, `synchronize_rcu()`의 사용이 그런 실제 기다림을 합니다.

RCU 읽기 크리티컬 섹션과 뒤따르는 RCU grace period 사이의 관계는 Figure 9.11에 그려진 것처럼 만약-그렇다면 관계입니다. 어떤 크리티컬 섹션의 어떤

⁹ 즉, 컴파일러는 이 로드를 Section 4.3.4.1의 “load tearing (로드 찢기)”로 설명된 것처럼 여러개의 작은 로드로 조개선 안됩니다.

¹⁰ 하지만, SRCU [McK06] 라 불리는 특수한 형태의 RCU 는 RCU 읽기 크리티컬 섹션 내에서의 일반적 잠들기를 허용합니다.

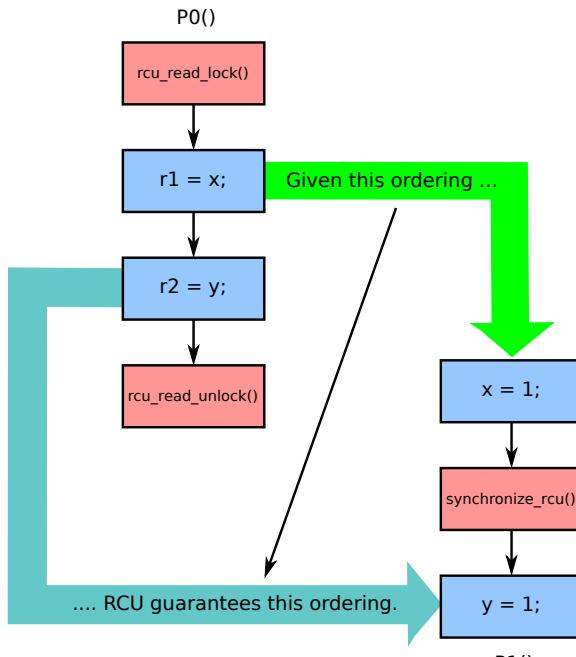


Figure 9.11: RCU Reader and Later Grace Period

부분이든 주어진 grace period의 시작을 앞섰다면, RCU는 그 크리티컬 섹션 전체가 이 grace period의 끝을 앞설 것을 보장합니다. 이 그림에서, P0()의 x로의 액세스는 P1()의 이 변수로의 액세스를 앞서며, 따라서 이 P1()의 synchronize_rcu() 호출로 만들어진 grace period 역시 앞섭니다. 따라서 P0()의 y로의 액세스 역시 P1()의 액세스를 앞설 것이 보장됩니다. 이 경우, r1의 마지막 값이 0이라면, r2의 마지막 값은 0일 것이 보장됩니다.

Quick Quiz 9.30: r1과 r2의 어떤 다른 마지막 값이 Figure 9.11에서 가능한가요?

What other final values of r1 and r2 are possible in Figure 9.11? ■

RCU 읽기 크리티컬 섹션과 그 앞의 RCU grace period 사이의 관계 또한 Figure 9.12에 그려진 것과 같은 만약-그렇다면 관계입니다. 특정 크리티컬 섹션의 어떤 부분이든 특정 grace period의 종료를 뒤따른다면, RCU는 이 크리티컬 섹션 전체가 이 grace period의 시작을 뒤따를 것을 보장합니다. 이 그림에서, P0의 y로의 액세스는 P1()의 해당 변수로의 액세스를 뒤따르며, 따라서 P1()의 synchronize_rcu() 호출로 생성된 grace period를 뒤따릅니다. 따라서 P0()의 x로의 액세스는 P1()의 액세스를 뒤따를 것이 보장됩니다. 이 경우, r2의 마지막 값이 1이라면, r1의 마지막 값은 1이 될 것이 보장됩니다.

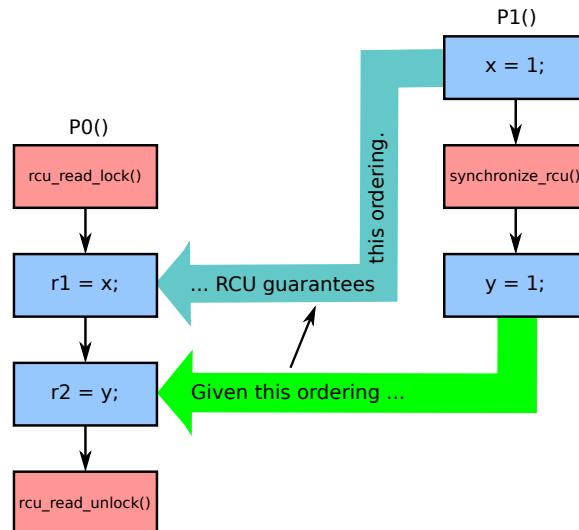


Figure 9.12: RCU Reader and Earlier Grace Period

Quick Quiz 9.31: Figure 9.12에서 P0()의 두 액세스가 반대 순서로 이루어지면 어떻게 될까요?

마지막으로, Figure 9.13에 보인 것과 같이, RCU 읽기 크리티컬 섹션은 RCU grace period에 완전히 겹쳐질 수 있습니다. 이 경우, r1의 마지막 값은 1이고 r2의 마지막 값은 0일 수 있습니다.

하지만, r1의 마지막 값은 0인데 r2의 마지막 값은 1일 수는 없습니다. 이는 RCU 읽기 크리티컬 섹션이 하나의 grace period를 완전히 겹쳤다는 것인데, 이는 불가능합니다 (또는 최소한 RCU의 버그를 의미합니다). RCU의 읽기 쓰래드 기다리기 보장은 따라서 두개의 부분으로 이루어집니다: (1) 만약 어떤 RCU 읽기 크리티컬 섹션의 어떤 부분이든 특정 grace period의 시작을 앞선다면, 이 크리티컬 섹션의 모든 부분이 이 grace period의 종료를 앞서게 된다. (2) 만약 어떤 RCU 읽기 크리티컬 섹션의 어떤 부분이든 어떤 grace period의 끝을 뒤따른다면, 이 크리티컬 섹션의 모든 부분이 이 grace period의 시작을 뒤따른다. 이 정의는 거의 모든 RCU 기반 알고리즘에 충분합니다만 이보다 더 많은 걸 원하는 분을 위해 RCU의 수행 가능한 정형적 모델이 리눅스 커널 v4.17과 그 다음 버전들에서 가능한데, Section 12.3.2에서 다뤄집니다. 또한, RCU의 순서 강제 기능들이 훨씬 자세하게 Section 15.4.2에서 다뤄집니다.

Quick Quiz 9.32: Figures 9.11–9.13의 P0()의 액세스들이 스토어였다면 무슨 일이 일어날까요?

RCU의 읽기 쓰래드 기다리기 능력이 Figures 9.11–9.13에 보인 것처럼 변수에 값 할당을 하기 위해 정말로 사용된다고 해도, Section 9.5.1에서 보인 것처럼 연결된

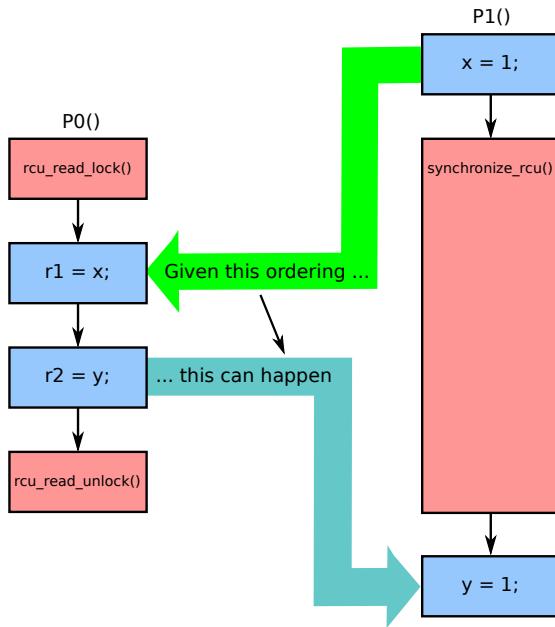


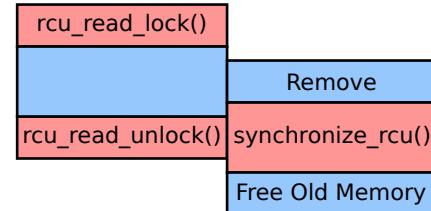
Figure 9.13: RCU Reader Within Grace Period

구조체들에서 제거된 데이터 항목을 안전하게 메모리 해제하기 위해 더 빈번히 사용됩니다. 일반적인 프로세스는 다음의 수도코드로 그려집니다:

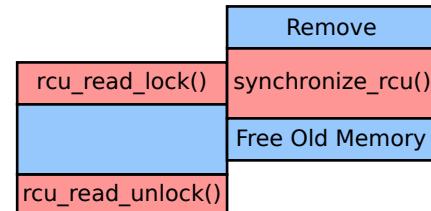
- 변경을 가하는데, 예를 들면 링크드 리스트에서 원소를 제거합니다.
- 모든 앞서서부터 존재해온 RCU 읽기 크리티컬 섹션의 완료되기를 기다립니다 (예를 들면, `synchronize_rcu()`를 통해서).
- 정리를 하는데, 예를 들면 앞서서 교체된 원소를 메모리 해제합니다.

이 더욱 추상화된 절차는 Figures 9.11–9.13 보다 더욱 추상화된 디어그램을 필요로 하는데, 특정 리트머스 테스트가 됩니다. 어쨌건, RCU 구현은 RCU 업데이트와 RCU 읽기 크리티컬 섹션의 형태와 무관하게 정확히 동작해야만 합니다. Figure 9.14 가 네개의 가능한 시나리오를 시간이 위에서 아래로 흐르는 형태로 보이며 이 필요를 충족합니다. 각 시나리오에서, RCU 읽기 쓰레드는 원쪽의 박스 더미로 표현되며 RCU 업데이트 쓰레드는 오른쪽 더비로 표현됩니다.

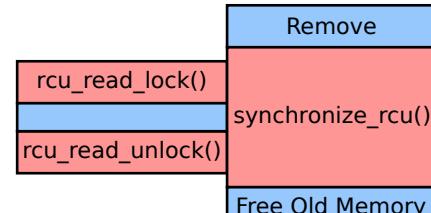
첫번째 시나리오에서 읽기 쓰레드는 업데이트 쓰레드가 삭제를 시작하기 전에 수행을 시작하며, 따라서 이 읽기 쓰레드가 제거된 데이터 원소로의 레퍼런스를 가질 수 있습니다. 따라서, 이 업데이트 쓰레드는 이 원소를 이 읽기 쓰레드가 완료되기 전까지 메모리 해제하



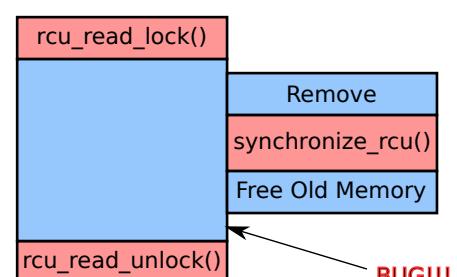
(1) Reader precedes removal



(2) Removal precedes reader



(3) Reader within grace period



(4) Grace period within reader (BUG!!!)

Figure 9.14: Summary of RCU Grace-Period Ordering Guarantees

지 않아야 합니다. 두번째 시나리오에서 읽기 쓰레드는 이 제거가 완료되기 전까지 시작하지 않습니다. 이 읽기 쓰레드는 이미 제거된 데이터 원소로의 레퍼런스를 얻을 수 없으며, 따라서 이 원소는 이 읽기 쓰레드가 완료되기 전에 메모리 해제되어도 됩니다. 세번째 시나리오는 두번째와 같지만 언제 이 읽기 쓰레드가 이 원소로의 레퍼런스를 얻을 수 없는지, 여전히 이 읽기 쓰레드가 완료될 때까지 이 원소의 메모리 해제를 미뤄도 괜찮은지 보입니다. 네번째 마지막 시나리오에서는 이 읽기 쓰레드가 업데이트 쓰레드가 이 데이터 원소를 제거하기 전에 시작하지만 이 원소는 이 읽기 쓰레드가 완료되기 전에 (올바르지 못하게) 메모리 해제됩니다. 이 디어그램은 따라서 RCU의 읽기 쓰레드 기다리기 기능을 그립니다: grace period 가 주어졌을 때, 각 읽기 쓰레드는 이 grace period 의 종료 전에 종료되며, 이 grace period 의 시작 후에 시작되거나 둘 다인데 이 grace period 내에 전체 크리티컬 섹션이 포함되는 경우입니다.

RCU 읽기 쓰레드는 업데이트가 진행중인 동안에도 진행을 할 수 있으므로, 다른 읽기 쓰레드는 이 데이터 구조의 상태에 대해 다른 해석을 할 수 있는데, 다음 섹션에서 이 주제를 다룹니다.

9.5.2.3 Maintain Multiple Versions of Recently Updated Objects

이 섹션은 RCU가 여러 버전의 데이터를 관리함으로써 어떻게 동기화로부터 자유로운 읽기 쓰레드를 수용하는지 이야기 합니다. 이 이야기는 `del_route()`와 동시에 수행되는 읽기 쓰레드들이 (Listing 9.13을 참고하세요) 오래된 `route` 구조체 또는 빈 리스트를 볼 수도 있지만 어느 쪽이든 유효한 결과를 얻게 되는, Section 9.5.1.1의 Figure 9.7에 의한 복수 버전의 소개로부터 시작합니다. 물론, Figure 9.6를 자세히 보면 `ins_route()` 호출은 역시 동시에 읽기 쓰레드들이 다른 버전들을 보게 할 수 있음을 알 수 있습니다: 초기의 빈 리스트 또는 새로이 삽입된 `route` 구조체. 레퍼런스 카운팅 (Section 9.2)과 해저드 포인터 (Section 9.3) 둘 다 동시에 읽기 쓰레드들이 다른 버전을 보게 할 수 있지만, RCU의 가벼운 읽기 쓰레드는 이를 더욱 그럴 수 있게 합니다.

하지만, 복수의 버전을 유지하는 것은 더욱 놀라울 수 있습니다. 예를 들어, 읽기 쓰레드가 동시에 업데이트 되는 링크드 리스트를 순회하는 읽기 쓰레드가 있는 Figure 9.15를 생각해 봅시다.¹¹ 이 그림의 첫번째 줄에서, 읽기 쓰레드는 데이터 아이템 A를 참조하고, 두번째 줄에서는 B로 넘어가서, 지금까지 B가 뒤따르는 A를 봤습니다. 세번째 줄에서는, 업데이트 쓰레드가 원소 A를 제거하고 네번째 줄에서 업데이트 쓰레드가 이 리스

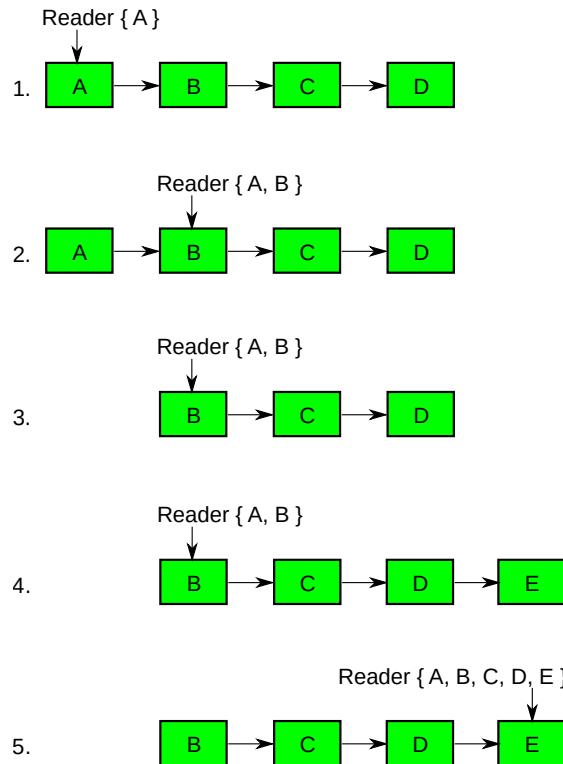


Figure 9.15: Multiple RCU Data-Structure Versions

트의 끝에 원소 E를 추가합니다. 마지막 다섯번째 줄에서는 읽기 쓰레드가 순회를 완료해서 원소 A부터 E까지를 모두 봤습니다.

그런 리스트가 존재한 적은 없었다는 제외하면 말입니다. 이 상황은 다른 동시의 읽기 쓰레드가 다른 버전들을 보게 되는 Figure 9.7에 보인 상황보다도 더 놀라울 수도 있습니다. 대조적으로, Figure 9.15에서 읽기 쓰레드는 실제로는 존재한 적이 없는 버전을 보는 것입니다!

이 이상한 상황을 해결하는 한가지 방법은 완화된 의미론을 사용하는 것입니다. 읽기 쓰레드의 순회는 전체 순회 동안 존재한 모든 데이터 항목을 만날 수 있어야 하며 (B, C, 그리고 D), 그 순회의 한 부분 동안에만 존재한 데이터 항목은 볼 수도 못 볼 수도 있습니다. 따라서, 이 특정 경우에는, 읽기 쓰레드의 순회가 이 모든 다섯 개 원소를 만나는 게 완전히 합법적입니다. 이 결과가 문제시 되다면, 이 상황을 해결하는 또 다른 방법은 더 강력한 동기화 메커니즘, 예를 들면 reader-writer 락킹이나 타임스탬프 또는 버저닝의 영리한 사용을 사용하는 것입니다. 물론, 더 강력한 메커니즘은 더 비용이 높을 것이지만, 그러면 또 다시 이야기 하지만 엔지니어링 삶은 모두 선택과 트레이드오프에 대한 것입니다.

¹¹ RCU 링크드 리스트 API는 Section 9.5.3에서 찾을 수 있습니다.

이 상황은 이상하게 보일 수도 있겠으나, 이는 실제 세계와 완전하게 일관적입니다. Section 3.2에서 보았듯이, 빛의 유한한 속도는 컴퓨터 시스템에서도 무시될 수 없으며, 이 시스템 바깥에서도 무시될 수 없는 것이 거의 분명합니다. 이는 결국 시스템 바깥의 실제 세계의 상태를 나타내는 시스템의 모든 데이터는 항상 그리고 영원히 시간이 지나있을 것이며, 따라서 실제 세계와는 비일관적일 것임을 의미합니다. 그 결과, 실제 세계 데이터를 다루는 알고리즘은 비일관된 데이터를 처리해야만 합니다. 많은 경우, 이런 알고리즘은 시스템 내부의 비일관성을 처리하는 데에도 완전히 기능할 것입니다.

Section 9.1에서 이야기 된 pre-BSD 패킷 라우팅 예가 그런 경우입니다. 라우팅 리스트의 내용들은 라우팅 프로토콜에 의해 설정되며, 이 프로토콜은 라우팅 비 안정성을 막기 위해 상당한 지연을 (몇초 또는 심지어 몇분) 일으킵니다. 따라서, 일단 라우팅 업데이트가 특정 시스템에 일단 가해지면, 패킷들을 한동안은 잘못된 길로 보낼 수 있습니다. 이 업데이트가 진행되는 중의 몇 마이크로세컨드 동안 패킷 몇개를 더 잘못된 길로 보내는 것은 자연된 라우팅 업데이트를 처리하는 더 높은 단계의 똑같은 프로토콜 동작이 내부의 비일관성을 처리할 것이기 때문에 분명 문제가 안됩니다.

비일관성을 감내해야 하는 것은 인터넷 라우팅만의 상황이 아닙니다. 반복하지만, 시스템 바깥의 상태를 추적하는 시스템 내의 데이터를 처리하는 모든 알고리즘은 비일관성을 감내해야 하는데, 보안 정책 (종종 사람으로 구성된 위원회에 의해 설정됩니다), 저장장치 구성, WiFi 액세스 포인트, 마이크로폰, 헤드셋, 카메라, 마우스, 프린터를 포함한 제거될 수 있는 하드웨어, 그 외에도 수많은 것들이 포함됩니다. 더욱이, Figure 9.9에 보인 많은 수의 리눅스 커널 RCU API 사용은 리눅스 커널의 레퍼런스 카운팅의 많은 사용과 다른 프로젝트에서의 증가되는 해저드 포인터 사용과 엮여서 그런 비일관성에 대한 감내는 누군가가 상상할 수 있는 것보다 훨씬 더 일반적임을 보입니다. 이는 특히 단일 항목 탐색이 순회보다 훨씬 더 흔하다는 것을 놓고 보면 그렇습니다: 어쨌건, (1) 동시의 업데이트는 전체 순회보다는 단일 항목 탐색에 영향을 덜 끼칠 것입니다, 그리고 (2) 고립된 단일 항목 탐색은 그런 비일관성을 알아낼 수 없습니다.

더 이론적인 시점에서 보면, RCU 읽기 쓰레드가 싱글 쓰레드 기반 프로그램에서 수행될 것과 정확히 동일한 기계 명령들을 수행함에도 불구하고 업데이트 쓰레드들과 완전하게 순서가 지어진 것으로 여겨지는 일부 특수한 경우도 존재합니다. 예를 들어, 페이지 139의 Listing 9.13으로 돌아가, 각 읽기 쓰레드가 평생 정확히 한번 `access_route()`를 수행하며, 그 외에는 읽기 쓰레드와 업데이트 쓰레드 사이에 어떤 소통도 없다고 해봅시다. 그러면 `access_route()`의 호출 각각은 이

코드 리스트의 `access_route()`의 라인 11에서 액세스 되는 `route` 구조체를 만들어낸 `ins_route()` 호출 뒤로 순서지어지고 모든 뒤따르는 `ins_route()`나 `del_route()` 호출의 앞으로 순서지어질 수 있습니다.

요약하자면, 복수의 버전을 유지하는 것이 바로 RCU 읽기 쓰레드의 극단적으로 낮은 오버헤드를 가능하게 하는 것이며, 앞에서 이야기 되었듯 많은 알고리즘은 복수의 버전들로 인해 당황하지 않습니다. 하지만, 복수의 버전을 처리할 수 없는 알고리즘도 분명 존재합니다. 그런 알고리즘이 RCU를 사용할 수 있게 조정하는 기술들도 존재합니다만 [McK04], 이는 이 섹션의 범위 밖입니다.

Exercises 이 예들은 전체 업데이트 오퍼레이션 동안 `mutex` 가 잡혀 있음을 가정했는데, 이는 한번에 최대 두개의 버전의 리스트만 존재함을 의미합니다.

Quick Quiz 9.33: 리스트의 두개 이상의 버전이 존재할 수 있게 하기 위해 삭제 예제를 어떻게 수정하시겠습니까?

Quick Quiz 9.34: 리스트의 RCU 버전은 한번에 몇 개까지 존재할 수 있나요?

9.5.2.4 Summary of RCU Fundamentals

이 섹션은 RCU 기반 알고리즘의 기본 토대적 컴포넌트들을 설명했습니다:

1. 새로운 데이터 추가를 위한 발행-구독 메커니즘,
2. 앞서서부터 존재해온 RCU 읽기 쓰레드의 종료를 기다리기 위한 방법 (자세한 내용은 Section 15.4.2을 참조하세요), 그리고
3. 동시의 RCU 읽기 쓰레드를 무심히 기다리게 하거나 피해 끼치지 않고 변경을 허용하기 위해 여러 버전을 관리하는 방법.

Quick Quiz 9.35: `rcu_read_lock()` 도 `rcu_read_unlock()` 도 스핀이나 블록을 하지 않는데 어떻게 RCU 읽기 쓰레드들을 RCU 업데이트 쓰레드가 지연시킬 수 있죠?

이 세개의 RCU 컴포넌트는 데이터가 싱글 쓰레드 기반 구현의 읽기 쓰레드가 사용할 것과 동일한 기계 인스트럭션을 수행하는 동시의 읽기 쓰레드에도 불구하고 데이터를 업데이트 할 수 있게 합니다. 이 RCU 컴포넌트는 놀랍도록 다양한 RCU 기반 알고리즘을 구현하는데 다른 방법들로 조합될 수 있습니다. 하지만, 더 높은 수준의 추상화에서 작업하는게 보통은 낫습니다.

그러므로, 다음 섹션은 리스트와 같이 간단한 데이터 구조를 포함하는 리눅스 커널 API를 설명합니다.

9.5.3 RCU Linux-Kernel API

이 섹션은 RCU를 리눅스 커널 API의 관점에서 봅니다.¹² Section 9.5.3.1은 RCU의 종료까지 기다리기 API를 보이고, Section 9.5.3.2은 RCU의 발행-구독과 버전 관리 API들을, Section 9.5.3.3은 RCU의 리스트 처리 API를, Section 9.5.3.4은 RCU의 분석 API를, 그리고 Section 9.5.3.5은 RCU의 다양한 API들이 어떤 맥락에서 사용될 수 있는지 보입니다. 마지막으로, Section 9.5.3.6은 결론적 요약을 제공합니다.

커널 내부에 큰 관심이 없는 독자 여러분은 이 섹션을 건너뛰어 page 156의 Section 9.5.4으로 넘어가실 수도 있겠습니다.

9.5.3.1 RCU has a Family of Wait-to-Finish APIs

“RCU란 무엇인가”에 대한 가장 간단한 답은 RCU는 API라는 것입니다. 예를 들어, 리눅스 커널에서 사용되는 RCU 구현이 RCU, “잠들 수 있는” RCU (SRCU), 태스크 기반 RCU (Tasks RCU)의 읽기 쓰래드 기다리기 부분과 일반적 API를 각각 보이고 있는 Table 9.1로, 그리고 이 API의 발행-구독 부분을 보이는 Table 9.2로 요약되어 있습니다.¹³

RCU를 처음 접하신다면, 각각 리눅스 커널의 RCU API 집합 중 하나의 멤버를 요약하는 Table 9.1의 열 중 하나에만 집중하는 걸 고려해 보시기 바랍니다. 예를 들어, 리눅스 커널에서 RCU가 어떻게 사용되는지 이해하는게 주요 관심사라면, “RCU”가 가장 빈번하게 사용되므로 시작하기 좋을 겁니다. 다른 한편, RCU를 그 자체만으로 이해하고자 한다면, “Tasks RCU”가 가장 간단한 API를 가졌습니다. 나중에 언제든 다른 열로 돌아오셔도 됩니다.

만약 여러분이 RCU와 친숙하다면, 이 표들은 유용한 레퍼런스로 사용될 수 있을 겁니다.

Quick Quiz 9.36: Table 9.1의 일부 셀들은 왜 느낌표를 (“!”) 가지고 있나요?

“RCU” 열은 세가지 리눅스 커널 RCU 구현의 [McK19c, McK19a] 집합에 연관되는데, RCU read-side 크리티컬 섹션의 `rcu_read_lock()`, `rcu_read_lock_bh()`, 또는 `rcu_read_lock_sched()`로 시작하여 `rcu_read_unlock()`, `rcu_read_unlock_bh()`,

¹² Userspace RCU의 API는 다른 곳에 문서화되어 있습니다 [MDI13c].

¹³ 이 인용은 v4.20과 이후 버전을 다룹니다. 그 전 버전의 리눅스 커널 RCU API에 대한 문서는 다른 곳에서 찾을 수 있습니다 [McK08d, McK14d].

또는 `rcu_read_unlock_sched()`로 각각 종료됩니다. Bottom halves, 인터럽트, 또는 preemption을 불능화 시키는 모든 코드 영역 또한 RCU read-side 크리티컬 섹션으로 동작합니다. 연관된 동기 update-side 기능들은 `synchronize_rcu()`와 `synchronize_rcu_expedited()`, 그리고 그와 유사한 `synchronize_net()`으로, 현재 수행 중인 모든 종류의 RCU read-side 크리티컬 섹션들이 완료되기를 기다립니다. 이 기다림의 길이가 “grace period”라고 알려져 있으며, `synchronize_rcu_expedited()`는 증가된 CPU 오버헤드와 IPI를 댓가로 grace period 응답시간을 줄이게끔 설계되었습니다. 비동기적 update-side 기능인 `call_rcu()`는 다음 grace period 후에 특정 함수를 특정 인자와 함께 수행합니다. 예를 들어, `call_rcu(p, f)`는 다음 grace period 후에 “RCU callback” `f(p)`이 호출되게 할 겁니다. `call_rcu()`를 사용하는 리눅스 커널 모듈이 언로딩 될 때라던지와 같이 모든 RCU callback들이 완료되기를 기다려야 하는 상황도 있습니다 [McK07e]. `rcu_barrier()` 기능이 그 일을 합니다.

Quick Quiz 9.37: 엄청나게 많은 RCU read-side 크리티컬 섹션이 무한정 `synchronize_rcu()` 호출을 기다리게 하는 것은 어떻게 예방하나요?

■

Quick Quiz 9.38: `synchronize_rcu()` API는 모든 앞서서부터 존재해온 인터럽트 핸들러가 완료되길 기다립니다, 맞죠?

■

마지막으로, RCU는 Section 9.5.4.8에서 이야기 되었듯 type-safe 메모리 [GC96]를 제공하기 위해 사용될 수 있습니다. RCU의 맥락에서, type-safe 메모리는 특정 데이터 원소의 타입이 그 원소에 접근하는 RCU read-side 크리티컬 섹션 내에서는 바뀌지 않음을 보장합니다. RCU 기반 type-safe 메모리의 사용을 위해선, `kmem_cache_create()`에 `SLAB_TYPESAFE_BY_RCU`를 넘기면 됩니다.

Table 9.1의 “SRCU” 열은 `srcu_read_lock()`과 `srcu_read_unlock()`으로 구분지어지는 SRCU read-side 크리티컬 섹션 내에서 일반적인 잠들기를 허용하는 특수화된 RCU API를 보입니다 [McK06] 하지만, RCU와 달리, SRCU의 `srcu_read_lock()`은 연관된 `srcu_read_unlock()`에 넘겨져야만 하는 값을 리턴합니다. 이 차이점은 SRCU 사용자는 각 SRCU 사용을 위해 `srcu_struct`를 하나 할당해야 하는 사실 때문입니다. 이 개별의 `srcu_struct` 구조체는 SRCU read-side 크리티컬 섹션이 연관되지 않은 `synchronize_srcu()`와 `synchronize_srcu_expedited()` 수행을 블록되게 하는 것을 방지합니다. 물론, SRCU read-side 크리티컬 섹션 내에서의 `synchronize_srcu()`나 `synchronize_srcu_expedited()` 호출은 스스로

Table 9.1: RCU Wait-to-Finish APIs

	RCU: Original	SRCU: Sleeping readers	Tasks RCU: Free tracing trampolines	Tasks RCU Rude: Free idle task tracing trampolines	Tasks RCU Trace: Protect sleepable BPF programs
Read-side critical-section markers	<code>rcu_read_lock()</code> ! <code>rcu_read_unlock()</code> ! <code>rcu_read_lock_bh()</code> <code>rcu_read_unlock_bh()</code> <code>rcu_read_lock_sched()</code> <code>rcu_read_unlock_sched()</code> (Plus anything disabling bottom halves, preemption, or interrupts.)	<code>srcu_read_lock()</code> <code>srcu_read_unlock()</code>	Voluntary context switch	Voluntary context switch and preempt-enable regions of code	<code>rcu_read_lock_trace()</code> <code>rcu_read_unlock_trace()</code>
Update-side primitives (synchronous)	<code>synchronize_rcu()</code> <code>synchronize_net()</code> <code>synchronize_rcu_expedited()</code>	<code>synchronize_srcu()</code> <code>synchronize_srcu_expedited()</code>	<code>synchronize_rcu_tasks()</code> <code>synchronize_rcu_tasks_rude()</code>	<code>synchronize_rcu_tasks_rude()</code> <code>synchronize_rcu_tasks_trace()</code>	
Update-side primitives (asynchronous / callback)	<code>call_rcu()!</code> <code>call_rcu()</code>	<code>call_srcu()</code> <code>call_rcu_tasks()</code>	<code>call_rcu_tasks_rude()</code> <code>call_rcu_tasks_trace()</code>		
Update-side primitives (wait for callbacks)	<code>rcu_barrier()</code>	<code>srcu_barrier()</code>	<code>rcu_barrier_tasks()</code> <code>rcu_barrier_tasks_rude()</code>	<code>rcu_barrier_tasks_rude()</code> <code>rcu_barrier_tasks_trace()</code>	
Update-side primitives (initiate / wait)	<code>get_state_synchronize_rcu()</code> <code>cond_synchronize_rcu()</code>				
Update-side primitives (free memory)	<code>kfree_rcu()</code>				
Type-safe memory	SLAB, TYPESAFE, BY RCU				
Read side constraints	No blocking (only preemption)	No synchronize_srcu() with same srcu_struct	No voluntary context switch	Neither blocking nor preemption	No RCU tasks trace grace period
Read side overhead	CPU-local accesses (barrier() on PREEMPT=n)	Simple instructions, memory barriers	Free	CPU-local accesses (free on PREEMPT=n)	CPU-local accesses
Asynchronous update-side overhead	sub-microsecond	sub-microsecond	sub-microsecond	sub-microsecond	sub-microsecond
Grace-period latency	10s of milliseconds	Seconds	Milliseconds	Milliseconds	10s of milliseconds
Expedited grace-period latency	10s of microseconds	N/A	N/A	N/A	N/A

의 데드락을 초래할 수 있으므로, 방지되어야 합니다. RCU 에서와 같이, SRCU 의 `synchronize_srcu_expedited()` 는 `synchronize_srcu()` 에 비해 grace period 응답시간을 줄여 줍니다만, 증가된 CPU 오버헤드를 맷가로 치룹니다.

Quick Quiz 9.39: 어떤 조건에서 `synchronize_srcu()` 는 SRCU read-side 크리티컬 섹션 내에서 안전하게 사용될 수 있나요?



일반적인 RCU 와 비슷하게, 비동기적 `call_srcu()` 함수를 통해 스스로 데드락 걸리는 것을 막을 수 있습니다. 하지만, 단일 태스크가 SRCU 콜백들을 매우 빠르게 등록할 수 있으므로 `call_srcu()` 를 사용할 때에는 특별한 주의가 필요합니다. SRCU 가 읽기 쓰레드들이 임의의 기간동안 블록할 수 있다는 점을 두고 볼 때, 이는 임의의 커다란 양의 메모리를 소모할 수 있습니다. 대조적으로, 동기적인 `synchronous_srcu()` 인터페이스에서 태스크는 다음 grace period 를 기다리기 시작하기 전에 현재 grace period 대기를 완료해야만 합니다.

또한 RCU 와 비슷하게, 모든 앞서 호출된 `call_srcu()` 콜백이 완료되기를 기다리는 `srcu_barrier()` 함수도 있습니다.

달리 말하면, SRCU 는 개발자들이 그 범위를 제한할 수 있게 허용함으로써 자신의 극단적으로 완화된 진행보장을 보상합니다.

Table 9.1 의 “Tasks RCU” 열은 리눅스 커널 트레이싱 (tracing) 에서 사용되는 trampoline 들의 메모리 해제를 중재하는데 특수화된 RCU API 를 보입니다. 이 trampoline 들은 트레이싱 되는 코드의 한 지점에서 실제 트레이싱을 하는 코드로 제어를 전환하는 데 사용됩니다. 특정 trampoline 에서 수행되는 모든 코드가 이 trampoline 을 메모리 해제하기 전에 종료되었음을 보장할 것이 당연히 필요합니다.

Tracing 되는 코드에의 변화는 일반적으로 하나의 `jump` 또는 `call` 인스트럭션으로 제한되며, 따라서 `rcu_read_lock()` 과 `rcu_read_unlock()` 을 구현하는데 필요한 코드를 수용할 수 없습니다. 이 trampoline 은 `rcu_read_lock()` 과 `rcu_read_unlock()` 호출을 담을 수도 없습니다. 이를 이해하기 위해, 특정 trampoline 을 이제 수행하기 시작하려는 CPU 를 생각해 봅시다. 이 CPU 는 아직 `rcu_read_lock()` 을 수행하지 않았으므로, 언제든 메모리 해제될 수 있어서 이 CPU 에게 치명적 놀라움을 안겨줄 수 있습니다. 따라서, trampoline 은 tracing 된 코드 또는 trampoline 자신에 의해 수행되는 동기화 기능으로 보호될 수 없습니다. 이는 어떻게 trampoline 이 보호되어야 하는지 질문을 떠오르게 합니다.

이 질문에 대답하기 위한 열쇠는 trampoline 코드가 직접적으로 간접적으로 자발적 컨텍스트 스위치

를 하는 코드를 결코 포함하지 않는다는 것을 파악하는 것입니다. 이 코드는 preemption 당할 수 있지만, 직접적으로 간접적으로는 `schedule()` 을 결코 호출하지 않습니다. 이는 자발적 컨텍스트 스위치와 idle 수행을 유일한 quiescent state 로 간주하는 RCU 변종을 제안합니다. 이 변종이 Tasks RCU 입니다.

Tasks RCU 는 read-side 마킹 기능이 없다는 점에서 일반적이지 않은데, 그 주요 사용 케이스가 어디에도 그런 마킹을 할 수 없는 경우라는 걸 생각하면 좋은 일입니다. 대신, `schedule()` 호출이 곧 quiescent state 로 동작합니다. 업데이트는 모든 앞서서부터 존재해온 trampoline 수행이 완료되기를 기다리기 위해 `synchronize_rcu_tasks()` 를, 또는 그 비동기 버전인 `call_rcu_tasks()` 를 사용할 수 있습니다. 또한 모든 앞서 호출된 `call_rcu_tasks()` 에 연관된 콜백들의 완료를 기다리기 위한 `rcu_barrier_tasks()` 도 존재합니다. 아직 요구가 없었기 때문에 `synchronize_rcu_tasks_expedited()` 함수는 존재하지 않습니다만, 그것의 유용한 변종을 구현하는 건 도전해볼만 할 것입니다.

Quick Quiz 9.40: CONFIG_PREEMPT_NONE=y 로 빌드된 커널에서라면 preemption 이 불능화 되었고 trampoline 은 직접적으로 간접적으로는 `schedule()` 을 호출하지 않으니 `synchronize_rcu()` 는 모든 trampoline 을 기다리지 않나요?



“Tasks RCU Rude” 열은 Section 9.5.1.3 에서 선보인 장난감 구현의 보다 효과적인 변종을 제공합니다. 이 변종은 각 CPU 가 컨텍스트 스위치를 수행해서 모든 자발적 컨텍스트 스위치나 모든 preemption 가능한 코드 영역이 quiescent state 로 동작하게 합니다. Tasks RCU Rude 변종은 리눅스 커널의 workqueue 기능을 동시적 컨텍스트 스위치를 강제하기 위한 장치로 사용하는데, 그 장난감 구현에서는 CPU 하나씩 순차적으로 사용하는 접근법과 대조됩니다. 이 API 는 Tasks RCU 의 것과 대칭되는데, 명시적 read-side 마커의 부재 또한 그렇습니다.

마지막으로, “Tasks RCU Trace” 열은 SRCU 의 그 것과 비슷한 기능을 갖지만 훨씬 빠른 read-side 마커는 예외입니다.¹⁴ 하지만, 이 속도는 이 마커들이 메모리 배리어 인스트럭션을 수행하지 않는다는 사실에서 기인하는 것으로, Tasks RCU Trace grace period 는 종종 모든 CPU 에게 IPI 를 보내야 하고 항상 전체 태스크 리스트를 스캔해야 함을 의미합니다. 그러나, 그로 인한 grace period 응답시간은 무척 짧아서 RCU 의 그것과 라이벌이 될만 합니다.

¹⁴ 그리고 따라서 Tasks RCU 계열에서 명시적 read-side 마커를 갖는게 비일상적입니다!

9.5.3.2 RCU has Publish-Subscribe and Version-Maintenance APIs

다행히, Table 9.2에 보인 RCU 발행-구독과 버전 관리 기능은 앞서 이야기한 모든 RCU 변종에 적용됩니다. 이 동일성이 더 많은 코드가 공유되고 API 증식을 줄일 수 있게 합니다. RCU 발행-구독 API의 원래 목적은 메모리 배리어를 이 API 내로 물어버려서 리눅스 커널 프로그래머가 리눅스가 지원하는 20+ CPU 군 [Spr01] 각각의 메모리 순서 규칙 모델에 대한 전문가가 될 필요 없이 RCU를 사용할 수 있게 하는 것이었습니다.

이 기능들은 포인터에 직접 동작하며, RCU로 보호되는 배열과 트리 같은 연결된 데이터 구조들을 생성하는데 유용합니다. 링크드 리스트의 특수한 경우는 Section 9.5.3.3에 설명된 별도의 API 집합으로 처리됩니다.

첫번째 카테고리는 새 데이터 항목으로의 포인터를 발행합니다. `rcu_assign_pointer()` 기능은 모든 앞의 초기화가 완화된 순서 규칙 기계에서도 포인터 할당 전에 행해졌을 것을 보장합니다. `rcu_replace_pointer()` 기능은 `rcu_assign_pointer()` 가 하는 것과 똑같이 이 포인터를 업데이트 합니다만, 기존의 값을 리턴하는데, `rcu_dereference_protected()` 가 하는 것과 동일한데 (아래를 보시기 바랍니다), lockdep 표현 역시 포함합니다. 이 교체는 업데이트 쓰레드가 새 포인터를 발행하면서 기존 포인터에 의해 참조되는 구조체를 해제해야 할 때 유용합니다.

Quick Quiz 9.41: 일반적으로, `rcu_dereference()`에 넘겨지는 포인터는 항상 Table 9.2의 포인터 발행 함수 중 하나, 예를 들면 `rcu_assign_pointer()`를 사용해 업데이트 되어야만 합니다.

이 규칙에서 예외는 무엇이겠습니까?

Quick Quiz 9.42: 이 순회와 업데이트 기능들이 모든 RCU API 집합 멤버들과 사용될 수 있다는 데에 어떤 단점은 없습니까?

`rcu_pointer_handoff()` 기능은 간단히 그것의 전체 인자를 리턴합니다만, RCU read-side 크리티컬 섹션 밖으로 새어나가는 포인터들을 검사하는 도구를 만드는데 유용합니다. `rcu_pointer_handoff()`의 사용은 그런 도구에게 이 구조체의 보호가 RCU에서 예를 들면 락킹이나 레퍼런스 카운팅 같은 어떤 다른 메커니즘으로 넘겨졌음을 알립니다.

`RCU_INIT_POINTER()` 매크로는 아직 읽기 쓰레드에게 노출되지 않은 RCU로 보호되는 포인터를 초기화하기 위해, 또는 RCU로 보호되는 포인터를 NULL로 만들기 위해 사용될 수 있습니다. 이 제한된 경우들에서, `rcu_assign_pointer()`에 의해 제공되는 메모리 배리어 인스트럭션은 필요치 않습니다. 비슷하게, `RCU_`

`POINTER_INITIALIZER()`는 데이터 구조들 내의 RCU로 보호되는 포인터들의 쉬운 초기화를 허용하기 위한 GCC스타일 구조체 초기화를 제공합니다.

두번째 카테고리는 데이터 항목으로의 포인터를 구독하거나, 또는 대안적으로, 안전하게 RCU로 보호되는 포인터들을 순회합니다. 다시 말하지만, 단순히 C 언어 액세스를 사용한 포인터로딩은 가리켜지는 데이터의 초기화 전 쓰레기 값을 볼 수 있게 할 수 있습니다. 비슷하게, 이 포인터를 어떤 수단을 사용해서든 RCU read-side 크리티컬 섹션 바깥에서 읽는 행위는 가리켜진 객체가 언제든 메모리 해제되게 할 수 있습니다. 하지만, 이 포인터가 그저 테스트 되고 역참조 되지 않을 것이라면, 가리켜진 객체의 메모리 해제는 문제가 아닐 겁니다. 이 경우, `rcu_access_pointer()`가 사용될 수 있습니다. 하지만, 보통 RCU read-side 보호가 필요하며, 따라서 `rcu_dereference()` 기능은 이 `rcu_dereference()` 호출이 `rcu_read_lock()`, `srcu_read_lock()`, 또는 어떤 다른 RCU read-side 마커의 보호 아래 이루어졌음을 검증하기 위해 리눅스 커널의 lockdep 기능 [Cor06a]을 사용합니다. 대조적으로, `rcu_access_pointer()` 기능은 lockdep과 연관되지 않으며, 따라서 RCU read-side 크리티컬 섹션 외부에서 사용되어도 lockdep의 불평을 일으키지 않을 겁니다.

보호가 필요치 않은 또 다른 상황은 업데이트 쪽 코드가 RCU로 보호되는 포인터를 업데이트 쪽 락을 잡은 채 액세스 하는 경우입니다. `rcu_dereference_protected()` API 멤버가 이 상황을 위해 제공됩니다. 이것의 첫번째 패러미터는 RCU로 보호되는 포인터이며, 두번째 패러미터는 이 액세스가 안전하기 위해 어떤 락이 잡혀져 있어야만 하는지를 설명하는 lockdep 표현입니다. 읽기 쓰레드와 업데이트 쓰레드 양쪽에서 수행되는 코드는 역시 lockdep 표현을 받지만 이 락을 잡지 않는 읽기 쪽 코드에서도 호출될 수 있는 `rcu_dereference_check()`를 사용할 수 있습니다. 어떤 경우에, 이 lockdep 표현은 매우 복잡할 수 있는데, 예를 들어 세밀한 락킹을 사용할 때에는, 큰 수의 락들 중 어떤 것도 잡혀 있을 수 있고, 그 중 어떤 것이 적용되는지 파악하기 무척 어려울 수도 있습니다. 이러한(바라건대 드문) 경우, `rcu_dereference_raw()`는 보호를 제공하지만 읽기 쓰레드 또는 특수한 락이 잡힌 상태에서 호출되는지를 검사하지는 않습니다. `rcu_dereference_raw_notrace()` API 멤버는 비슷하게 동작하지만 추적될 수 없으며, 따라서 트레이싱 코드에서 안전히 사용될 수 있습니다.

어떤 연결된 구조든 포인터를 사용해 액세스 될 수 있지만, 더 높은 수준의 구조가 도움될 수 있습니다. 따라서 다음 섹션은 다양한 종류의 RCU로 보호되며 리눅스 커널에서 사용되는 링크드 리스트들을 알아봅니다.

Table 9.2: RCU Publish-Subscribe and Version Maintenance APIs

Category	Primitives	Overhead
Pointer publish	<code>rcu_assign_pointer()</code>	Memory barrier
	<code>rcu_replace_pointer()</code>	Memory barrier (two of them on Alpha)
	<code>rcu_pointer_handoff()</code>	Simple instructions
	<code>RCU_INIT_POINTER()</code>	Simple instructions
	<code>RCU_POINTER_INITIALIZER()</code>	Compile-time constant
Pointer subscribe (traversal)	<code>rcu_access_pointer()</code>	Simple instructions
	<code>rcu_dereference()</code>	Simple instructions (memory barrier on Alpha)
	<code>rcu_dereference_check()</code>	Simple instructions (memory barrier on Alpha)
	<code>rcu_dereference_protected()</code>	Simple instructions
	<code>rcu_dereference_raw()</code>	Simple instructions (memory barrier on Alpha)
	<code>rcu_dereference_raw_notrace()</code>	Simple instructions (memory barrier on Alpha)

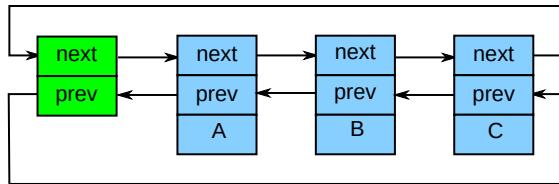


Figure 9.16: Linux Circular Linked List (list)



Figure 9.17: Linux Linked List Abbreviated

9.5.3.3 RCU has List-Processing APIs

`rcu_assign_pointer()` 와 `rcu_dereference()` 가 이론상으로는 모든 상상할 수 있는 RCU 로 보호되는 데이터 구조를 만들 수 있지만, 실제로는 더 높은 단계의 것을 쓰는게 종종 낫습니다. 따라서, `rcu_assign_pointer()` 와 `rcu_dereference()` 는 리눅스의 리스트 조작 API 의 특수한 RCU 변종에 내포되어 있습니다. 리눅스는 네개의 양방향 링크드 리스트 변종을 갖는데, 순환형 `struct list_head` 와 선형 `struct hlist_head/struct hlist_node`, `struct hlist_nulls_head/struct hlist_nulls_node`, 그리고 `struct hlist_bk_head/struct hlist_bk_node` 쌍들입니다. 앞의 것은 Figure 9.16 에 보여져 있는 형태를 띠는데, (가장 왼쪽의) 녹색 상자가 리스트 헤더를 표현하며 (오른쪽 세개의) 파란 상자들은 이 리스트의 원소들을 표현합니다. 이 표기법은 다루기 성가시며 따라서 헤더가 아닌 (파랑) 원소들만 보이는 Figure 9.17 에 보인 것처럼 간략화 할 겁니다.

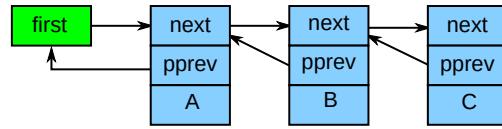


Figure 9.18: Linux Linear Linked List (hlist)

리눅스의 `hlist`¹⁵ 는 선형 리스트로, 이는 Figure 9.18 에 보이듯 헤더에 순환형 리스트에 필요한 두개의 포인터가 아니라 한개의 포인터만 필요합니다. 따라서, `hlist` 의 사용은 커다란 해쉬 테이블의 해쉬 버킷 배열을 위한 메모리 사용량을 절반으로 줄일 수 있습니다. 앞에서와 같이, 이 표기법은 다루기 귀찮으므로, `hlist` 구조는 Figure 9.17 에 보인 것과 같은 `list_head` 스타일 리스트로 간략화 해 표기하겠습니다.

`hlist_nulls` 라 이름지어진 리눅스의 `hlist` 의 한 변종은 여러 구별된 NULL 포인터들을 제공합니다만, 그 것 외에는 Figure 9.18 에 보인것과 같은 레이아웃을 갖습니다. 이 변종에서, 0 값의 낮은 위치 비트를 갖는 `->next` 포인터는 NULL 포인터 타입을 의미합니다. 이 타입의 리스트는 lockless 읽기 쓰레드들이 어떤 노드가 한 리스트에서 다른 리스트로 옮겨진 것을 파악할 수 있게 하기 위해 사용됩니다. 예를 들어, 해쉬 테이블의 각 버킷은 그것의 NULL 포인터를 마킹하기 위해 그 인덱스를 사용할 수도 있습니다. 읽기 쓰레드가 자신이 시작한 버킷의 인덱스와 맞지 않는 NULL 포인터를 마주치면, 이 읽기 쓰레드는 자신이 순회하고 있는 원소가 순회 사이에 다른 버킷으로 옮겨졌음을 알게 됩니다. 이 읽기 쓰레드는 리스트의 끝에 도달했는지 알기 위해 `is_a_nulls()` 함수 (`hlist_nulls` NULL 포인터를 받으면 `true` 를 리턴합니다) 를, NULL 포인터의 타입을 가져오

¹⁵ “h” 는 해쉬테이블을 의미하는데, 리눅스의 양방향 포인터 순환형 링크드 리스트에 비해 메모리 사용량을 절반으로 줄입니다.

기 위해 `get_nulls_value()` (이 함수는 인자의 NULL 포인터 식별자를 리턴합니다)를 사용할 수 있습니다. `get_nulls_value()` 가 예상치 않은 값을 리턴하면, 읽기 쓰레드는 올바른 행동을 취할 수 있는데, 예를 들면 시작부터 순회를 재시작 하는 것입니다.

Quick Quiz 9.43: 하지만 `hlist_nulls` 읽기 쓰레드가 다른 버킷으로 갔다가 다시 돌아오면 어떻게 하죠?

`hlist_nulls`에 대한 더 많은 정보는 `rculist_nulls.rst` 파일 (오래된 커널에선 `rculist_nulls.txt`)에서 제공되는 도움 될만한 예제 코드들과 함께 리눅스 커널의 소스 트리에서 얻을 수 있습니다.

리눅스의 `hlist`의 또 다른 변종은 비트 락킹을 결합하며, `hlist_bl`이라 이름지어졌습니다. 이 변종은 Figure 9.18에 보인 것과 동일한 레이아웃을 사용하지만, 헤드 포인터의 (그림의 “첫번째”) 낮은 위치 비트를 이 리스트를 잠그기 위해 사용됩니다. 이 방법 또한 메모리 사용량을 줄이는데, 이게 아니면 포인터 자체와 함께 별도의 스펀락이 저장되어야 하기 때문입니다.

이 링크드 리스트 변종을 위한 API 멤버들이 Table 9.3에 요약되어 있습니다. 더 많은 정보는 리눅스 커널 소스 트리의 Documentation/RCU 디렉토리와 Linux Weekly News [McK19b]에서 얻을 수 있습니다.

하지만, 이 섹션의 나머지 부분은 `list_replace_rcu()`의 사용에 대해 확장해 보는데, 이 API 멤버가 RCU에게 그 이름을 주었기 때문입니다. 이 API 멤버는 여러 필드를 가지는 이 리스트의 중간에 있는 원소가 원자적으로 업데이트 되어서 어떤 읽기 쓰레드든 기존의 값들의 집합 또는 새로운 값들의 집합을 봄아 하지, 그 두 집합의 섞인 값을 보지는 않아야 하는 복잡한 업데이트를 행하기 위해 사용됩니다. 예를 들어, 링크드 리스트의 각 노드는 정수 필드 `->a`, `->b`, 그리고 `->c`를 가질 수도 있으며, 그 값을 5, 6, 그리고 7에서 5, 2, 그리고 3으로 각각 업데이트 해야 할 수도 있습니다.

이 원자적 업데이트를 행하는 코드 구현은 단순합니다:

```

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);

```

다음 이야기는 이 코드를 따라가는데, 그 상태 변화를 그리기 위해 Figure 9.19를 사용합니다. 각 원소의 세 값은 필드 `->a`, `->b`, 그리고 `->c`를 각각 나타냅니다. 빨간색으로 칠해진 원소들은 읽기 쓰레드에 의해 참조될 수도 있으며, 읽기 쓰레드는 업데이트 쓰레드와 직접적으로 동기화 하지 않기 때문에, 읽기 쓰레드는 이 전체

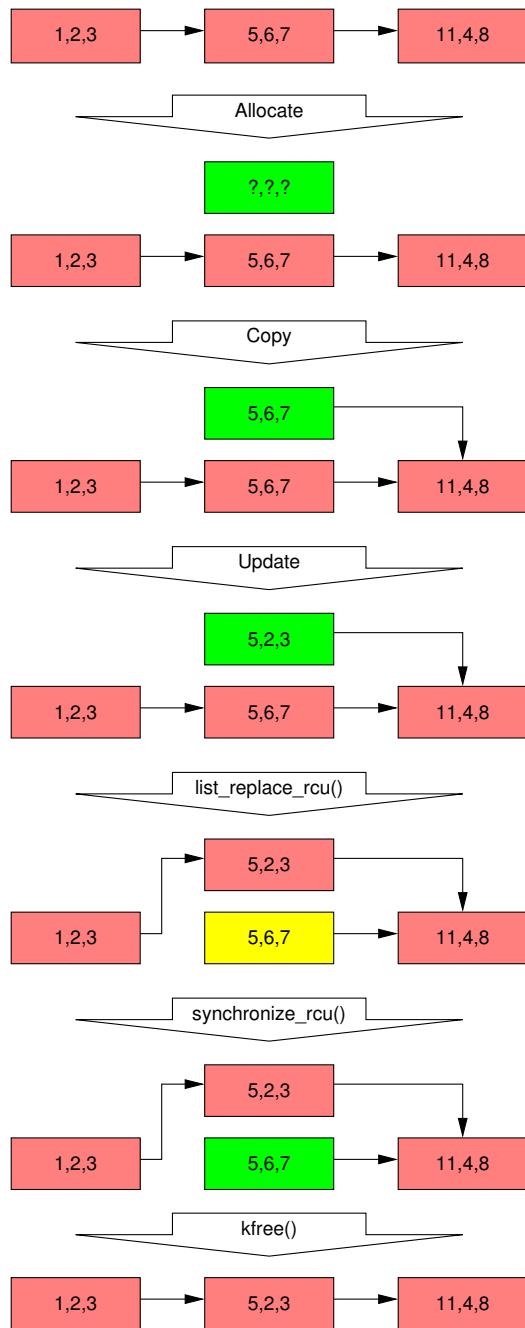


Figure 9.19: RCU Replacement in Linked List

Table 9.3: RCU-Protected List APIs

list: Circular doubly linked list	hlist: Linear doubly linked list	hlist_nulls: Linear doubly linked list with marked NUL pointer, with up to 31 bits of marking	hlist_b1: Linear doubly linked list with bit locking
Structures			
struct list_head	struct hlist_head	struct hlist_nulls_head	struct hlist_b1_head
Initialization			
	struct hlist_node	struct hlist_nulls_node	struct hlist_b1_node
Full traversal			
	INIT_LIST_HEAD_RCU()		
		hlist_for_each_entry_rcu()	hlist_b1_for_each_entry_rcu()
		hlist_for_each_entry_rcu_bh()	hlist_b1_for_each_entry_safe()
		hlist_for_each_entry_rcu_notrace()	
Resume traversal			
		hlist_for_each_entry_continue_rcu()	hlist_for_each_entry_continue_rcu()
		hlist_for_each_entry_from_rcu()	hlist_for_each_entry_continue_rcu_bh()
		hlist_for_each_entry_from_rcu()	
Stepwise traversal			
		hlist_entry_rcu()	hlist_nulls_first_rcu()
		hlist_entry_lockless()	hlist_nulls_next_rcu()
		list_first_or_null_rcu()	
		list_next_rcu()	
		list_next_or_null_rcu()	
Add			
		hlist_first_rcu()	hlist_nulls_first_rcu()
		hlist_next_rcu()	hlist_nulls_next_rcu()
		hlist_pprev_rcu()	
		hlist_add_tail_rcu()	
Delete			
		hlist_add_before_rcu()	hlist_b1_add_head_rcu()
		hlist_add_behind_rcu()	hlist_b1_set_first_rcu()
		hlist_add_head_rcu()	
		hlist_add_tail_rcu()	
Replace			
		hlist_del_rcu()	hlist_b1_del_rcu()
		hlist_del_init_rcu()	hlist_b1_del_init_rcu()
Splice			
		hlist_replace_rcu()	
		list_splice_init_rcu()	list_splice_tail_init_rcu()

교체 과정과 동시에 수행될 수도 있습니다. 뒤쪽으로의 포인터와 tail로부터 head로의 링크는 간략화를 위해 생략되었음을 알아두시기 바랍니다.

포인터 p를 포함해 이 리스트의 처음 상태는 앞의 삭제 예와 동일한데, 이 그림의 첫번째 행에 보여져 있습니다.

다음의 텍스트는 어떤 읽기 쓰레드든 이 두 값들 중 하나만을 보게 하는 방식으로 5,6,7 원소를 5,2,3으로 교체하는지 설명합니다.

라인 15은 교체 원소를 할당하여, Figure 9.19의 두 번째 행에 이릅니다. 이 지점에서, 어떤 읽기 쓰레드도 새로 할당된 원소로의 참조를 잡지 못하며(초록색 색깔로 표시되어 있습니다), 초기화되어 있지 않습니다(물음표로 표시되어 있습니다).

라인 16은 기존 원소를 새 원소로 복사하여, Figure 9.19의 세번째 행의 상태에 이릅니다. 새로 할당된 원소는 여전히 읽기 쓰레드에 의해 참조될 수 없지만, 이제 초기화되었습니다.

Figure 9.19의 네번째 행에 보여져 있듯 라인 17은 q->b의 값을 “2”로, 그리고 라인 18은 q->c의 값을 “3”으로 업데이트 합니다. 새로 할당된 구조체는 여전히 읽기 쓰레드에게 액세스될 수 없음을 생각해주시기 바랍니다.

이제, 라인 19는 새 원소가 마침내 읽기 쓰레드에게 보여질 수 있게끔 교체를 행하며, 따라서 Figure 9.19의 다섯번째 행에 보여져 있듯 빨갛게 칠해집니다. 이 시점에서는 아래에서 보여지듯 우리는 이 리스트의 두 버전을 갖게 됩니다. 기존부터 존재해온 읽기 쓰레드는 5,6,7 원소를 볼 수도 있지만(따라서 노란색으로 칠해져 있습니다), 새로운 읽기 쓰레드는 5,2,3 원소를 볼 것입니다. 하지만 모든 읽기 쓰레드는 이 값들 중 하나만 보지, 두 값들의 혼합된 것을 보지는 못할 것이 보장됩니다.

라인 20의 synchronize_rcu()가 리턴한 후에는 하나의 grace period가 지나갔을 것이며, 따라서 list_replace_rcu() 전에 시작된 모든 읽기는 완료되었을 것입니다. 특히, 5,6,7 원소로의 참조를 가지고 있던 모든 읽기 쓰레드는 그들의 RCU read-side 크리티컬 섹션이 종료되었을 것이 보장되며, 따라서 참조를 계속 쥐고 있는 것이 금지됩니다. 따라서, 더이상 기존 값으로의 참조를 쥐고 있는 읽기 쓰레드는 더이상 존재하지 않습니다. Figure 9.19의 여섯번째 행에 초록색으로 칠함으로써 표시되어 있습니다. 읽기 쓰레드들의 관점에서, 우린 이제 새 원소가 기존 원소를 교체한, 리스트의 단일한 버전으로 돌아왔습니다.

라인 21의 kfree()가 완료된 후, 이 리스트는 Figure 9.19의 마지막 행에 보여진 것과 같아질 것입니다.

RCU가 이 교체의 경우 후에 이름지어지긴 했으나, 리눅스 커널에서의 주요하고 많은 RCU 사용 예는 Sec-

Table 9.4: RCU Diagnostic APIs

Category	Primitives
Mark RCU pointer	<code>__rcu</code>
Debug-object support	<code>init_rcu_head()</code> <code>destroy_rcu_head()</code> <code>init_rcu_head_on_stack()</code> <code>destroy_rcu_head_on_stack()</code>
Stall-warning control	<code>rcu_cpu_stall_reset()</code>
Callback checking	<code>rcu_head_init()</code> <code>rcu_head_after_call_rcu()</code>
lockdep support	<code>rcu_read_lock_held()</code> <code>rcu_read_lock_bh_held()</code> <code>rcu_read_lock_sched_held()</code> <code>srcu_read_lock_held()</code> <code>rcu_is_watching()</code> <code>RCU_LOCKDEP_WARN()</code> <code>RCU_NONIDLE()</code> <code>rcu_sleep_check()</code>

tion 9.5.2.3의 Figure 9.15에 보여진 것처럼 간단하고 비의존적인 삽입과 삭제에 기반해 있습니다.

다음 섹션은 RCU를 사용하는 코드의 디버깅을 하는 개발자들을 돋는 API들을 봅니다.

9.5.3.4 RCU Has Diagnostic APIs

Table 9.4은 RCU의 진단 API들을 보입니다.

`__rcu`는 RCU로 보호된 포인터를 표시하는데, 예를 들면 “`struct foo __rcu *p;`”와 같은 형태입니다. `rcu_dereference()`로 넘겨질 수도 있는 포인터들이 그렇게 표시될 수 있습니다만, `rcu_dereference()`로부터 리턴된 값을 잡고 있는 포인터들은 그렇지 않아야 합니다. 변수들에 이 표시를 제공함으로써, 구조체 필드, 함수 패러미터, 그리고 리턴 값들은 리눅스 커널의 `sparse` 툴이 RCU로 보호되는 포인터들이 평범한 C 언어 로드와 스토어를 통해 부적절하게 액세스되는 상황을 파악할 수 있게 합니다.

객체 디버깅 지원은 리눅스 커널의 메모리 할당자로부터 얻어진 구조체의 부분인 모든 `rcu_head` 구조체에 대해 자동적입니다만, 각자의 특수 목적 메모리 할당자를 사용하는 경우에는 할당과 해제 시에 각각 `init_rcu_head()`와 `destroy_rcu_head()`를 사용할 수 있습니다. 함수 호출 스택에서 할당된 `rcu_head` 구조체를 사용하는 경우엔(그런 경우가 있습니다!) 첫번째 사용 전에 `init_rcu_head_on_stack()`을, 마지막 사용 후, 그러나 해당 함수에서 리턴하기 전에 `destroy_rcu_head_on_stack()`을 사용할 수 있습니다. 객체 디버깅 지원은 동일한 `rcu_head` 구조체를 `call_rcu()`와

그 친구들에게 빠르게 잇달아 보냄으로써 이중 메모리 해제 종류의 메모리 할당 버그의 `call_rcu()` 버전의 버그를 감지할 수 있게 합니다.

Stall 경고 제어는 `rcu_cpu_stall_reset()`을 통해 제공되는데, 호출자가 현재 grace period의 남은 시간 동안 RCU CPU stall 경고를 멈출 수 있게 합니다. RCU CPU stall 경고는 어떤 RCU read-side 크리티컬 섹션이 지나치게 긴 시간 동안 수행되는 상황을 집어낼 수 있게 하며, 커널 디버거와 같은 것들이 예를 들면 breakpoint를 만났거나 하는 경우엔 이 경고를 끌 수 있게 하는게 유용합니다.

Callback 검사는 `rcu_head_init()` 와 `rcu_head_after_call_rcu()`를 통해 제공됩니다. 앞의 것은 `rcu_head` 구조체가 `call_rcu()`에 넘겨지기 전에 호출되며, 그러면 `rcu_head_after_call_rcu()`는 해당 callback이 명시된 함수와 함께 수행되었는지 검사할 겁니다.

Lockdep [Cor06a] 지원은 `rcu_read_lock_held()`, `rcu_read_lock_bh_held()`, `rcu_read_lock_sched_held()`, 그리고 `srcu_read_lock_held()`를 포함하는데, 이것들 각각은 연관된 종류의 RCU read-side 크리티컬 섹션 내에서 호출되었다면 `true`를 리턴합니다.

Quick Quiz 9.44: Tasks RCU 를 위한 `rcu_read_lock_tasks_held()` 는 왜 존재하지 않나요?



`rcu_read_lock()` 은 idle 루프 내에서 사용될 수 없으며 전력 효율에 대한 고민이 idle 루프를 상당히 화려하게 만든 이유로, `rcu_is_watching()`은 `rcu_read_lock()`의 사용이 허용되는 컨텍스트에서 호출되었을 때 `true`를 리턴합니다. `srcu_read_lock()`은 idle에서도, 심지어 오프라인된 CPU에서도 사용될 수 있으며, 이는 `rcu_is_watching()`이 SRCU에는 적용되지 않음을 의미함을 다시 주의하시기 바랍니다.

`RCU_LOCKDEP_WARN()`은 lockdep 이 활성화 되어 있으며 그 인자가 `true`로 평가될 때 경고를 냅니다. 예를 들어, `RCU_LOCKDEP_WARN(!rcu_read_lock_held())`는 RCU read-side 크리티컬 섹션 바깥에서 호출되었을 때 경고를 낼 겁니다.

`RCU_NONIDLE()`은 RCU를 완전한 인자로 전달된 명령문이 수행될 때를 보게 강제할 수 있습니다. 예를 들어, `RCU_NONIDLE(WARN_ON(!rcu_is_watching()))`은 결코 경고를 내지 않을 겁니다. 하지만, 2020-2021 사이의 변화는 RCU의 범위를 idle 루프 내까지 확장시켰으며, 이는 `RCU_NONIDLE()`의 필요를 크게 줄이거나 심지어 제거할 수 있을 겁니다.

마지막으로, `rcu_sleep_check()`는 RCU, RCU-bh, 또는 RCU-sched read-side 크리티컬 섹션 내에서 호출되었을 때 경고를 냅니다.

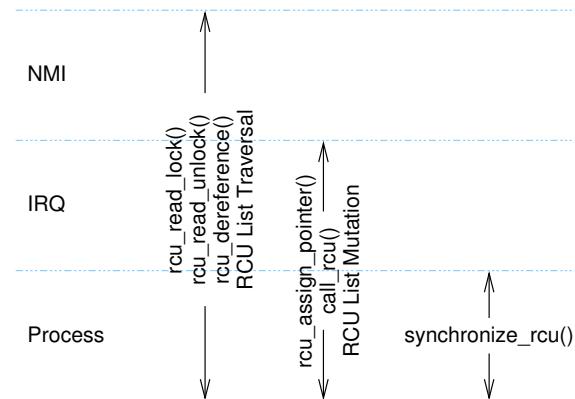


Figure 9.20: RCU API Usage Constraints

9.5.3.5 Where Can RCU's APIs Be Used?

Figure 9.20는 어떤 API가 어떤 커널 내 환경에서 사용될 수 있는지 보입니다. RCU read-side 기능들은 NMI를 포함한 어떤 상황에서도 사용될 수 있으며 RCU 업데이트와 비동기 grace period 기능들은 NMI를 제외한 모든 환경에서 사용될 수 있고, 마지막으로 RCU 동기적 grace period 기능들은 프로세스 컨텍스트에서만 사용될 수 있습니다. RCU 리스트 순회 기능들은 `list_for_each_entry_rcu()`, `hlist_for_each_entry_rcu()` 등을 포함합니다. 비슷하게, RCU 리스트 업데이트 기능들은 `list_add_rcu()`, `hlist_del_rcu()` 등을 포함합니다.

RCU의 다른 제품군에서의 기능들이 서로를 대체할 수 있는데, 예를 들어 `srcu_read_lock()`은 `rcu_read_lock()`이 사용될 수 있는 모든 context에서 사용될 수 있습니다.

9.5.3.6 So, What is RCU Really?

그 핵심에 있어, RCU는 삽입을 위한 발행과 구독, 모든 RCU 읽기 쓰레드의 종료를 기다리기, 그리고 여러 버전의 관리를 지원하는 API 이상도 이하도 아닙니다. 그러나, RCU 위에서 더 높은 수준의 것들을 만드는게 가능한데, reader-writer 락킹, 레퍼런스 카운팅, 그리고 존재 보장 구조 등이 포함되며 Section 9.5.4에 나열되어 있습니다. 더 나아가서, 저는 리눅스 커뮤니티가 흥미로운 RCU의 새로운 사용처를 찾는 것을 그들이 모든 동기화 기능들에 대해 커널에서 그려는 것처럼 계속할 것을 의심치 않습니다.

물론, RCU의 더 완벽한 모습은 여러분이 이 API를 가지고 할 수 있는 모든 것을 포함해야 할 겁니다.

하지만, 많은 사람들을 위해, RCU의 완벽한 모습은 예제 RCU 구현을 포함해야만 합니다. 따라서 Appendix B는 점점 복잡도와 기능을 증가시켜가는 “장난감”

Table 9.5: RCU Usage

Mechanism RCU Replaces	Section
Reader-writer locking	Section 9.5.4.2
Restricted reference-counting	Section 9.5.4.3
Bulk reference-counting	Section 9.5.4.4
Garbage collector	Section 9.5.4.5
Multi-version concurrency control	Section 9.5.4.6
Existence Guarantees	Section 9.5.4.7
Type-Safe Memory	Section 9.5.4.8
Wait for things to finish	Section 9.5.4.9

RCU 구현 시리즈를 제공합니다만, 어떤 사람들은 고전적인 “User-level Implementation of Read-Copy Update” [DMS¹²] 를 선호할 수도 있겠습니다. 그 외의 모든 사람을 위해, 다음 섹션은 일부 RCU 사용처를 살펴봅니다.

9.5.4 RCU Usage

이 섹션은 “RCU 는 무엇인가?”라는 질문을 RCU 가 무엇을 제공할 수 있는지 사용의 관점에서 답해봅니다. RCU 는 존재하는 메커니즘을 교체하는데 가장 자주 사용되므로, Table 9.5 에 보인 것처럼 우린 그런 메커니즘과의 관계에 주로 집중해서 RCU 를 알아 봅니다. 이 표의 섹션들에 이어서 Section 9.5.4.10 은 요약을 제공합니다.

9.5.4.1 RCU for Pre-BSD Routing

Listing 9.14 과 9.15 는 RCU 로 보호되는 Pre-BSD 라우팅 테이블의 코드를 보입니다 (route_rcu.c). 앞의 것은 데이터 구조와 route_lookup() 을, 뒤의 것은 route_add() 와 route_del() 을 보입니다.

Listing 9.14 에서, 라인 2 는 RCU 교체에 사용되는 ->rh 필드를 더하고 라인 6 는 use-after-free 검사 필드인 ->re_freed 를 더하며, 라인 16, 22, 그리고 26 는 RCU read-side 보호를, 라인 20 와 21 는 use-after-free 검사를 더합니다. Listing 9.15 에서, 라인 11, 13, 30, 라인 34, 그리고 39 는 update-side 락킹을 더하고 라인 12 와 33 는 RCU update-side 보호를 더하고, 라인 35 는 route_cb() 가 하나의 grace period 가 지난 후 호출되게 하며 라인 17-24 는 route_cb() 를 정의합니다. 이는 동작하는 동시적 구현을 위한 최소한의 추가된 코드입니다.

Figure 9.21 는 읽기 전용 워크로드에서의 성능을 보입니다. RCU 는 상당히 잘 확장되며 거의 이상적인 성능을 제공합니다. 하지만, 이 데이터는 rcu_read_lock() 과 rcu_read_unlock() 에 작은 양의 코드를 생성하는

Listing 9.14: RCU Pre-BSD Routing Table Lookup

```

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, re_next) {
18         if (rep->addr == addr) {
19             ret = rep->iface;
20             if (READ_ONCE(rep->re_freed))
21                 abort();
22             rCU_read_unlock();
23             return ret;
24         }
25     }
26     rCU_read_unlock();
27     return ULONG_MAX;
28 }
```

Listing 9.15: RCU Pre-BSD Routing Table Add/Delete

```

1 int route_add(unsigned long addr, unsigned long interface)
2 {
3     struct route_entry *rep;
4
5     rep = malloc(sizeof(*rep));
6     if (!rep)
7         return -ENOMEM;
8     rep->addr = addr;
9     rep->iface = interface;
10    rep->re_freed = 0;
11    spin_lock(&routelock);
12    cds_list_add_rcu(&rep->re_next, &route_list);
13    spin_unlock(&routelock);
14    return 0;
15 }
16
17 static void route_cb(struct rCU_head *rhp)
18 {
19     struct route_entry *rep;
20
21     rep = container_of(rhp, struct route_entry, rh);
22     WRITE_ONCE(rep->re_freed, 1);
23     free(rep);
24 }
25
26 int route_del(unsigned long addr)
27 {
28     struct route_entry *rep;
29
30     spin_lock(&routelock);
31     cds_list_for_each_entry(rep, &route_list, re_next) {
32         if (rep->addr == addr) {
33             cds_list_del_rcu(&rep->re_next);
34             spin_unlock(&routelock);
35             call_rcu(&rep->rh, route_cb);
36             return 0;
37         }
38     }
39     spin_unlock(&routelock);
40     return -ENOENT;
41 }
```

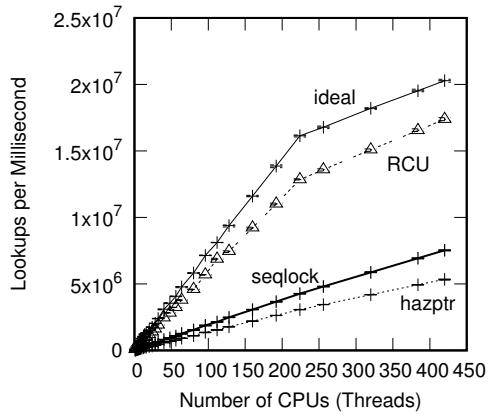


Figure 9.21: Pre-BSD Routing Table Protected by RCU

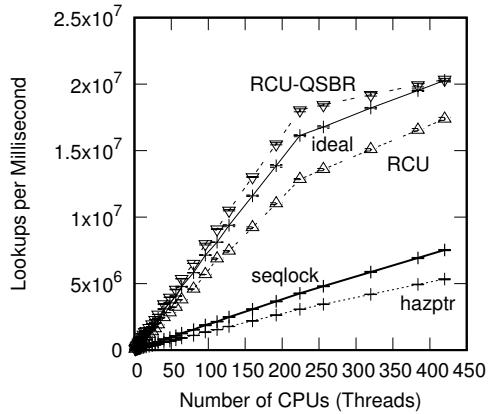


Figure 9.22: Pre-BSD Routing Table Protected by RCU QSBR

userspace RCU [Des09b, MDJ13c] 의 RCU_SIGNAL 버전을 사용해 만들어졌습니다. `rcu_read_lock()` 과 `rcu_read_unlock()` 에 어떤 코드도 생성하지 않는 QSBR 버전의 RCU를 사용하면 어떻게 될까요? (RCU QSBR에 대한 논의를 위해선 Section 9.5.1, 그리고 특히 Figure 9.8를 보시기 바랍니다.)

이에 대한 답이 RCU QSBR의 성능과 확장성은 실제로 이상적인 동기화 없는 워크로드의 것을 넘어선을 보이는 Figure 9.22에 보여져 있습니다.

Quick Quiz 9.45: 잠깐요, 뭐라고요??? 어떻게 RCU QSBR이 이상적인 것보다 나을 수 있죠? 어떤 쓰레기 같은 이상에 대한 정의가 그것이 모든 가능한 결과 중 최선이 아니게 할 수 있죠???



Quick Quiz 9.46: RCU QSBR의 읽기 성능이 그렇게 좋은데, 왜 다른 userspace 버전을 선정쓰죠?

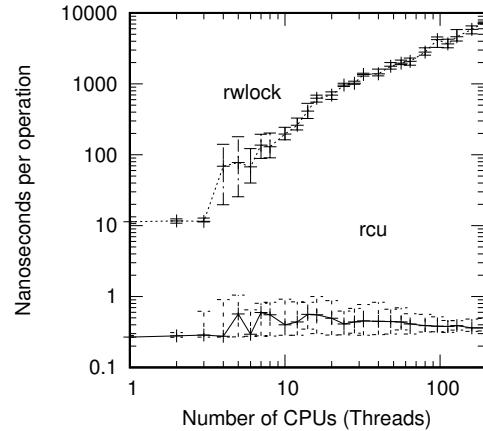


Figure 9.23: Performance Advantage of RCU Over Reader-Writer Locking

9.5.4.2 RCU is a Reader-Writer Lock Replacement

리눅스 커널에서의 RCU의 가장 흔한 사용처는 아마도 읽기만 집중적인 상황에서의 reader-writer 락킹 대체일 겁니다. 그러나, 이런 RCU 사용은 처음에 제게 즉각적으로 적절하게 느껴지지 않았고, 실제로 저는 1990년대 초에 범용 RCU 구현을 만들기 전에 가벼운 reader-writer 락을 구현하기로 했습니다 [HW92].¹⁶ 이 가벼운 reader-writer 락으로 제가 하고자 했던 모든 것은 결국 RCU를 이용해 구현되었습니다. 사실, 그건 이 가벼운 reader-writer 락이 처음 사용되기 3년이나 전의 일이었습니다. 정말 바보가 된 것 같았죠!

RCU와 reader-writer 락 사이의 핵심 유사성은 둘다 병렬로 구현될 수 있는 read-side 크리티컬 섹션을 갖는다는 것입니다. 실제로, 어떤 경우에는 RCU API 멤버들을 연관된 reader-writer 락 API 멤버들로 대체하는 게 가능합니다. 하지만 무엇보다, 왜 그런걸 신경쓰죠?

RCU의 장점은 성능, 데드락 내성, 그리고 리얼타임 응답시간을 포함합니다. 물론, RCU의 제한점들도 존재하는데 읽기 쓰레드와 업데이트 쓰레드가 동시에 수행된다는 사실, 낮은 우선순위 RCU 읽기 쓰레드가 높은 우선순위 쓰레드를 grace period 하나가 지나갈 때까지 블록시킨다는 것, 그리고 grace period 응답시간이 수 밀리세컨드까지 길어질 수 있다는 점이 포함됩니다. 이런 장점과 한계점들이 다음 문단들에서 논의됩니다.

Performance 리눅스 커널 RCU의 reader-writer 락킹 대비 읽기 성능 장점이 448개 2.10GHz Intel x86 CPU 시스템에서 측정되어 만들어진 Figure 9.23에 보여져 있습니다.

¹⁶ 2.4 리눅스 커널의 `brlock`과, 더 최신의 리눅스 커널의 `lglock`과 비슷합니다.

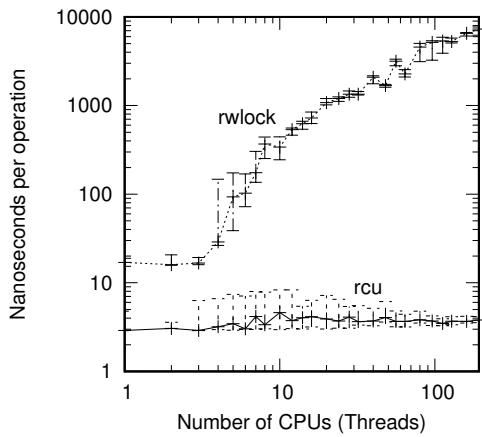


Figure 9.24: Performance Advantage of Preemptible RCU Over Reader-Writer Locking

Quick Quiz 9.47: 뭐요? 2.10GHz에서의 클락 시간은 약 500 picosecond 인데 RCU 가 300-picosecond 도 안되는 오버헤드를 갖는다는 말을 나더러 믿으라구요?

Quick Quiz 9.48: 이 책의 초기 버전에서는 RCU 읽기 오버헤드가 1 picosecond 미만이라 하지 않았나요? 무슨 일이 있었던 거죠???

Quick Quiz 9.49: Figure 9.23 의 rcu 에는 왜 그리 큰 편차가 존재하는 거죠?

Reader-writer 락킹은 단일 CPU에서 RCU 보다 수십 배 느리며, 192 CPU에서는 수만 배보다 더 느리다는 것을 보시기 바랍니다. 대조적으로, RCU는 상당히 잘 확장합니다. 두 경우 모두, 여러 바들은 30회의 측정 결과 전체 범위를 보이며, 선은 그 중간값을 보입니다.

더 정확한 그림은 CONFIG_PREEMPT 커널에서 얻어질 수 있겠지만, 448rodml 2.10GHz x86 CPU 시스템에서 측정된 Figure 9.24에 보이듯 RCU는 여전히 단일 CPU에서 약 일곱배, 192 CPU에서 수천배 reader-writer 락 보다 빠릅니다. 큰 수의 CPU에서 reader-writer 락킹의 높은 편차값을 보세요. 여러바는 데이터의 전체 범위를 그립니다.

Quick Quiz 9.50: 시스템은 448 개의 하드웨어 쓰레드를 갖는데, 왜 192 CPU 까지만 측정하나요?

물론, Figures 9.23 and 9.24의 reader-writer 락킹의 낮은 성능은 비현실적인 0의 길이를 갖는 크리티컬 섹션에 의해 과장된 것입니다. RCU의 성능 이득은 크리티컬 섹션의 길이가 길어질수록 줄어듭니다. 이런 감소가 앞의 그림과 같은 시스템에서 수행된 Figure 9.25에 보여져 있습니다. 여기서, y 축은 read-side 기능의

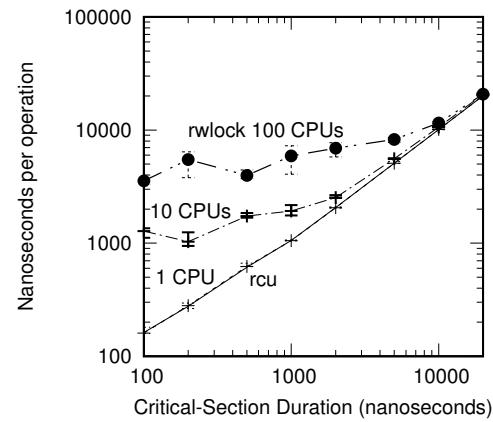


Figure 9.25: Comparison of RCU to Reader-Writer Locking as Function of Critical-Section Duration, 192 CPUs

오버헤드와 크리티컬 섹션의 오버헤드의 합을, x 축은 크리티컬 섹션의 오버헤드를 나노세컨드 단위로 표현합니다. 하지만 y 축이 로그스케일이어서 이 그림에서의 작은 차이가 여전히 상당한 차이를 표현함을 기억해 두시기 바랍니다. 이 그림은 non-preemptible RCU를 보입니다만, preemptible RCU의 read-side 오버헤드가 3 나노세컨드란 점을 놓고 보면, 그 그림은 Figure 9.25와 거의 같을 겁니다.

Quick Quiz 9.51: Figure 9.25의 마이크로세컨드 미만 기간에서의 더 큰 에러 범위는 왜 그런거죠?

Reader-writer 락킹을 위한 세개의 선이 있는데, 위의 것은 100 CPU, 그 다음은 10 CPU, 그리고 아래의 것은 1 CPU에서의 것입니다. 따라서 CPU의 수가 커지고 크리티컬 섹션이 짧아질수록 RCU의 성능 이득은 커집니다. 이런 성능 이득은 100-CPU 시스템이 더이상 드물지 않고 여러 시스템 콜이 (그리고 그것들이 내포하는 RCU read-side 크리티컬 섹션) 마이크로세컨드 내에 완료된다는 사실에 의해 과소평가되었습니다.

첨언하자면, 다음 문단에서 이야기 되듯, RCU read-side 기능은 거의 완전히 데드락에 내성을 가지고 있습니다.

Deadlock Immunity RCU가 읽기가 대부분인 워크로드에 상당한 성능 이득을 제공하지만, RCU를 만든 주요 이유는 사실 그것의 read-side 데드락 내성을 위함이었습니다. 이 내성은 RCU read-side 기능이 블록하거나 스판하거나 뒤쪽으로 브랜칭하지 않아서 그 수행시간이 정해져 있다는 사실에서 기인합니다. 따라서 그것들은 데드락 사이클에 참여될 수가 없습니다.

Quick Quiz 9.52: 이 데드락 내성에 예외가 있나요? 만약 그렇다면 어떤 이벤트들의 연속이 데드락을 이르게 할 수 있나요?

■ RCU 의 read-side 데드락 내성의 한가지 흥미로운 결론은 RCU 읽기 쓰레드를 무조건적으로 RCU 업데이트 쓰레드로 업그레이드 할 수 있다는 것입니다. Reader-writer 락킹에서 그런 업그레이드를 시도하는 것은 데드락을 초래합니다. RCU read-to-update 업그레이드를 하는 코드 조각은 다음과 같을겁니다:

```

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, 그리고 RCU read-side 보호 아래 수행됨에 주의하십시오.

RCU 의 데드락 내성의 또다른 흥미로운 결론은 커다란 범주의 우선순위 역전 문제에 대한 내성입니다. 예를 들어, 낮은 우선순위 RCU 읽기 쓰레드가 높은 우선순위 RCU 업데이트 쓰레드가 update-side 락을 잡는 걸 방지하지 못합니다. 비슷하게, 낮은 우선순위 RCU 업데이트 쓰레드는 높은 우선순위 RCU 읽기 쓰레드가 RCU read-side 크리티컬 섹션에 들어가는 것을 막지 못합니다.

Quick Quiz 9.53: 데드락과 우선순위 역전에 모두 내성이 있다??? 사실이라기엔 너무 좋아보이는데요. 이게 가능하다는 걸 제가 왜 믿어야 하죠?

Realtime Latency RCU read-side 기능들은 스펜하지도 블록하지도 않기 때문에, 훌륭한 realtime 반응시간을 제공합니다. 추가적으로, 앞에서도 이야기 했듯 이는 RCU read-side 기능들과 락이 연관된 우선순위 역전 문제에 내성이 있음을 의미합니다.

하지만, RCU 는 더 복잡한 우선순위 역전 시나리오에는 문제가 있을 수 있는데, 예를 들면 높은 우선순위의 프로세스는 낮은 우선순위의 RCU 읽기 쓰레드에 의해 -rt 커널에서 한 RCU grace period 가 지나갈 때까지 블록될 수 있습니다. 이는 RCU 우선순위 부스팅을 통해 해결될 수 있습니다 [McK07d, GMTW08].

RCU Readers and Updaters Run Concurrently RCU 읽기 쓰레드는 스펜도 블록도 하지 않으므로, 그리고 업데이트 쓰레드는 어떤 종류의 롤백이나 abort 도 하지 않으므로, RCU 읽기 쓰레드와 업데이트 쓰레드는

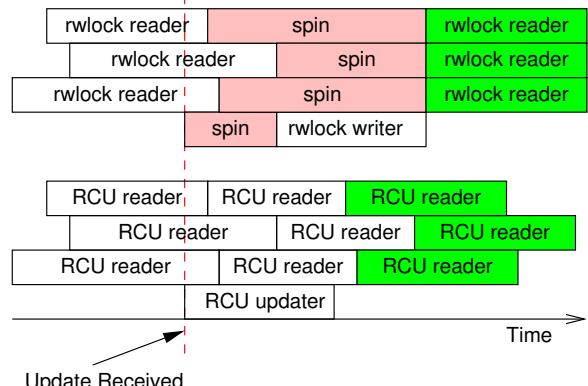


Figure 9.26: Response Time of RCU vs. Reader-Writer Locking

동시에 수행되어야만 합니다. 이는 RCU 읽기 쓰레드가 오래되어 더이상 유효하지 않은 데이터를 읽을 수도 있고, 비일관적인 모습을 보게 될 수도 있음을, 따라서 reader-writer 락킹에서 RCU 로의 전환에 혼란을 야기할 수 있음을 의미합니다.

하지만, 놀라우리만치 많은 상황에서 비일관성과 오래되어 더이상 유효하지 않은 데이터는 문제가 되지 않습니다. 고전적인 예는 네트워킹 라우팅 테이블입니다. 라우팅 업데이트는 특정 시스템까지 닿는데 상당한 시간을 요하므로 (수초에서 수분까지도), 시스템은 이 업데이트가 도착하기 전까지 상당한 시간동안 패킷을 잘못된 방향으로 보낼 수 있습니다. 업데이트를 몇 추가적인 밀리세컨드 동안 잘못된 방향으로 보내는 건 보통 문제가 아닙니다. 더 나아가, RCU 업데이트 쓰레드는 RCU 읽기 쓰레드가 끝나기를 기다리지 않고 변화를 만들 수 있으므로, RCU 읽기 쓰레드는 Figure 9.26 에 보인 것처럼 batch-fair reader-writer 락킹의 읽기 쓰레드보다 빠르게 변화를 볼 수도 있습니다.

일단 업데이트가 도달하면, rwlock 쓰기 쓰레드는 마지막 읽기 쓰레드가 완료되기 전까지 진행될 수 없으며, 뒤따르는 읽기 쓰레드는 이 쓰기 쓰레드가 완료되기 전까지 진행될 수 없습니다. 하지만, 이 뒤따르는 읽기 쓰레드는 오른쪽 상자의 초록색으로 표시되었듯 이 새 값을 읽을 수 있는게 보장됩니다. 대조적으로, RCU 읽기 쓰레드와 업데이트 쓰레드는 서로를 블록할 수 없어서, RCU 읽기 쓰레드가 이 업데이트된 값을 더 빨리 볼 수 있게 합니다. 물론, 이것들의 수행이 RCU 업데이트 쓰레드의 그것과 겹치므로, 모든 RCU 읽기 쓰레드는 업데이트된 값을 읽을 수도 있는데, 이 업데이트 전부터 시작된 세개의 읽기 쓰레드 역시 포함됩니다. 하지만 초록색으로 칠해진 가장 오른쪽 RCU 읽기 쓰레드만이 업데이트된 값을 볼 것으로 보장됩니다.

Reader-writer 락킹과 RCU 는 그저 다른 보장을 제공할 뿐입니다. Reader-writer 락킹에서 어떤 쓰기 쓰레드의 시작보다 나중에 시작된 읽기 쓰레드는 새 값을 볼 것이 보장되고, 이 쓰레드가 스픈하고 있는 사이 시작되려 하는 읽기 쓰레드는 새 값을 볼 수도 보지 못할 수도 있는데, 사용되는 rwlock 구현의 읽기/쓰기 쓰레드 선호도에 의존적입니다. 대조적으로, RCU 에서 업데이트 쓰레드가 완료되기 전에 시작된 모든 읽기 쓰레드는 새 값을 볼 것이 보장되며, 업데이트 쓰레드가 시작한 후에 완료된 모든 읽기 쓰레드는 타이밍에 따라 새 값을 볼 수도 보지 못할 수도 있습니다.

여기서의 핵심은 reader-writer 락킹이 컴퓨터 시스템 내에서는 일관성을 실제로 보장하지만, 이 일관성이 바깥 세상의 비일관성의 증가를 대가로 오는 경우가 존재한다는 것입니다. 달리 말하자면 reader-writer 락킹은 바깥 세상 관점에서 조용하게 오래되어 유효하지 않아진 데이터를 대가로 내부적 일관성을 획득합니다.

그러나, 시스템 내에서의 비일관성과 오래되어 유효하지 않아진 데이터가 제어되지 못하는 상황도 있습니다. 다행히, 비일관성과 오래되어 유효하지 않아진 데이터를 막는 여러 방법이 존재하며 [McK04, ACMS03], 레퍼런스 카운팅에 기반한 일부 방법이 Section 9.2에서 다루어집니다.

Low-Priority RCU Readers Can Block High-Priority Reclaimers Realtime RCU [GMTW08] 또는 SRCU [McK06] 에서, preemption 당한 읽기 쓰레드는 grace period 가 완료되는 것을 막을 것인데, 심지어 높은 우선순위 작업이 그 grace period 완료를 기다리며 블록되어 있더라도 그렇습니다. Realtime RCU 는 call_rcu() 로 synchronize_rcu() 를 대체하거나 RCU 우선순위 부스팅을 이용해 [McK07d, GMTW08] 이를 막을 수 있는데, 2008년 초 기준으로 아직 실험적 상태입니다. SRCU 와 QRCU 를 우선순위 부스팅과 결합시키는게 필요해질 수도 있지만, 분명한 실제 세계에서의 필요가 확인되기 전까지는 아닙니다.

RCU Grace Periods Extend for Many Milliseconds Userspace RCU [Des09b, MDJ13c], 가속된 grace period, 그리고 Appendix B 에 이야기 된 여러 “장난감” RCU 구현들의 예외가 있지만, RCU grace period 는 수 밀리세컨드까지 늘어날 수 있습니다. 가능한 곳에 비동기 인터페이스를 (call_rcu() 와 call_rcu_bh()) 사용하는 것을 포함해 그런 오랜 지연을 문제가 되지 않게 하는 여러 기법이 존재하지만 이는 RCU 가 읽기가 대부분인 상황에서 사용되는 관습적 규칙의 주요 이유 중 하나입니다.

Code: Reader-Writer Locking vs. RCU Code 최선의 경우, reader-writer 락킹에서 RCU 로의 전환은 Listing 9.16, 9.17, 그리고 9.18 에 보인 것처럼 상당히 단순할 수 있는데, 이것들은 모두 Wikipedia [MPA⁺06] 에서 가져온 것들입니다.

하지만, 이런 전환이 항상 단순하지는 않습니다. 이는 Listing 9.18 의 spin_lock() 도 synchronize_rcu() 도 Listing 9.17 의 읽기 쓰레드를 배제시키지 않기 때문입니다. 첫째로, spin_lock() 은 rCU_read_lock() 과 rCU_read_unlock() 과 상호작용하지 않으며 따라서 그것들을 배제하지 않습니다. 둘째로, write_lock() 과 synchronize_rcu() 가 앞서서부터 존재한 읽기 쓰레드를 기다리지만 write_lock() 만이 뒤따르는 읽기 쓰레드의 시작을 방지합니다.¹⁷ 따라서, synchronize_rcu() 는 읽기 쓰레드를 배제하지 못합니다. 따라서 reader-writer 락킹을 사용하는 많은 상황이 RCU 로 쉽게 바뀔 수 있다는 사실은 놀랍습니다.

Reader-writer 락킹을 RCU 로 교체하는 더 자세한 경우들이 다른 곳에서도 발견될 수 있을 겁니다 [Bro15a, Bro15b].

Semantics: Reader-Writer Locking vs. RCU Semantics Reader-writer 락킹 semantic 은 거칠고 비정규적으로 다음의 세가지 임시적 제약으로 요약될 수 있습니다:

- 쓰기 락 획득은 모든 읽기 락을 잡은 쓰레드들이 그 락을 놓기를 기다립니다.
- 쓰기 락 획득은 모든 쓰기 락을 잡은 쓰레드들이 그 락을 놓기를 기다립니다.
- 읽기 락 획득은 모든 쓰기 락을 잡은 쓰레드들이 그 락을 놓기를 기다립니다.

RCU 는 제약 #3 을 완전히 사라지게 하고 나머지 두개도 다음과 같이 완화시킵니다:

- 쓰기 쓰레드는 모든 앞서서부터 존재해온 읽기 락을 잡은 쓰레드들을 업데이트의 파괴적인 단계를 (일반적으로 메모리 해제) 처리하기 전에 기다립니다.
- 쓰기 쓰레드는 서로간에 필요한 바에 맞춰 동기화 합니다.

물론 이 완화가 RCU 구현이 훌륭한 성능과 확장성을 얻을 수 있게 합니다. RCU 사용자들은 이 완화를 놀랄 만큼 많은 방법으로 보상합니다만 일반적으로 공간적 제약을 사용합니다:

¹⁷ 누구였든 이걸 Paul 에게 지적해준 분에게 감사를.

Listing 9.16: Converting Reader-Writer Locking to RCU: Data

```

1 struct el {           1 struct el {
2   struct list_head lp; 2   struct list_head lp;
3   long key;           3   long key;
4   spinlock_t mutex;  4   spinlock_t mutex;
5   int data;           5   int data;
6   /* Other data fields */ 6   /* Other data fields */
7 };                   7 };
8 DEFINE_RWLOCK(listmutex); 8 DEFINE_SPINLOCK(listmutex);
9 LIST_HEAD(head); 9 LIST_HEAD(head);

```

Listing 9.17: Converting Reader-Writer Locking to RCU: Search

```

1 int search(long key, int *result) 1 int search(long key, int *result)
2 { 2 {
3   struct el *p; 3   struct el *p;
4   4
5   read_lock(&listmutex); 5   rCU_read_lock();
6   list_for_each_entry(p, &head, lp) { 6   list_for_each_entry_rcu(p, &head, lp) {
7     if (p->key == key) { 7     if (p->key == key) {
8       *result = p->data; 8       *result = p->data;
9       read_unlock(&listmutex); 9       rCU_read_unlock();
10      return 1; 10      return 1;
11    } 11    }
12  } 12  }
13  read_unlock(&listmutex); 13  rCU_read_unlock();
14  return 0; 14  return 0;
15 } 15 }

```

Listing 9.18: Converting Reader-Writer Locking to RCU: Deletion

```

1 int delete(long key) 1 int delete(long key)
2 { 2 {
3   struct el *p; 3   struct el *p;
4   4
5   write_lock(&listmutex); 5   spin_lock(&listmutex);
6   list_for_each_entry(p, &head, lp) { 6   list_for_each_entry(p, &head, lp) {
7     if (p->key == key) { 7     if (p->key == key) {
8       list_del(&p->lp); 8       list_del_rcu(&p->lp);
9       write_unlock(&listmutex); 9       spin_unlock(&listmutex);
10      synchronize_rcu(); 10      synchronize_rcu();
11      kfree(p); 11      kfree(p);
12      return 1; 12      return 1;
13    } 13    }
14    write_unlock(&listmutex); 14    spin_unlock(&listmutex);
15    return 0; 15    return 0;
16 } 16 }

```

- 새 데이터는 새로 할당된 메모리에 위치합니다.
- 오래된 데이터는 메모리 해제되지만, 다음의 조건이 충족된 후에만 합니다:
 - 데이터가 뒤의 읽기 쓰레드에 의해 선 접근 불 가능하게끔 링크 삭제되었을 것, 그리고
 - 뒤따르는 RCU grace period 가 지났을 것.

요약해서, RCU 는 자신의 쓰기 성능과 확장성을 시공간적 제약에 기반한 semantic 을 통해 얻습니다.

9.5.4.3 RCU is a Restricted Reference-Counting Mechanism

Grace period 는 진행중인 RCU read-side 크리티컬 섹션이 있는 동안은 완료될 수 없으므로, RCU read-side 기능들은 제한된 레퍼런스 카운팅 메커니즘으로 사용될 수 있습니다. 예를 들어, 다음 코드 조각을 생각해봅시다:

```

1 rCU_read_lock(); /* acquire reference. */
2 p = rCU_dereference(head);
3 /* do something with p. */
4 rCU_read_unlock(); /* release reference. */

```

`rcu_read_lock()` 기능은 `p` 로의 레퍼런스를 얻는 것으로 생각될 수도 있는데, `rcu_dereference()` 가 `p`에 값을 할당한 뒤에 시작되는 grace period 는 우리가 매칭되는 `rcu_read_unlock()` 전까지는 종료될 수 없기 때문입니다. 이 레퍼런스 카운팅 방법은 우리가 RCU read-side 크리티컬 섹션 내에서는 블록될 수 없으며 RCU read-side 크리티컬 섹션을 한 태스크에서 다른 태스크로 넘길 수도 없다는 사실로 제약이 됩니다.

이런 제약과 무관하게, 다음 코드는 안전하게 `p` 를 삭제합니다:

```

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);

```

`head` 로의 할당은 미래의 `p` 로의 레퍼런스가 얻어지는 것을 방지하고, `synchronize_rcu()` 는 모든 앞서 획득된 레퍼런스가 해제되기를 기다립니다.

Quick Quiz 9.54: 하지만 잠시만요! 이건 RCU 를 reader-writer 락킹의 대체제로 생각할 때 사용될 수 있는 것과 완전히 똑같은 코드예요! 뭐죠?

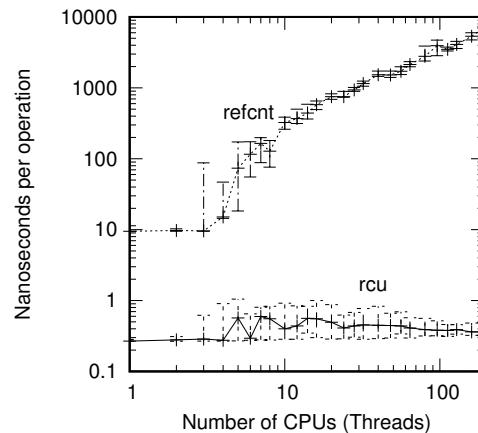


Figure 9.27: Performance of RCU vs. Reference Counting

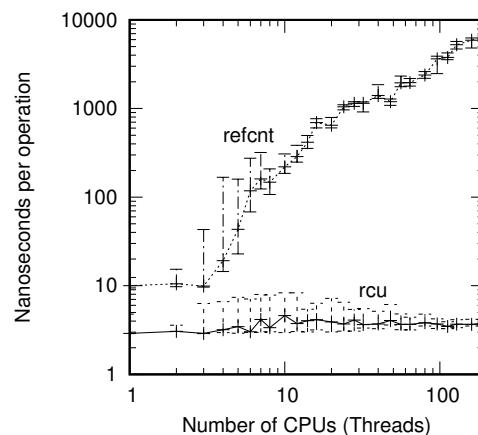


Figure 9.28: Performance of Preemptible RCU vs. Reference Counting

물론, RCU 는 Section 13.2 에서 이야기 된 것처럼 전통적인 레퍼런스 카운팅과 결합될 수도 있습니다.

그런데 이걸 왜 신경씁니까? 다시 말하지만, 답은 성능으로, 448-CPU 2.1 GHz Intel x86 시스템에서 non-preemptible 과 preemptible 리눅스 커널 RCU 를 사용해 각각 얻어진 Figure 9.27 와 9.28 에서 보인 것과 같습니다. Non-preemptible RCU 의 레퍼런스 카운팅 대비 장점은 단일 CPU 에서 수십배를 넘으며 192 CPU 에서는 수만배까지 달합니다. Preemptible RCU 의 장점은 단일 CPU 에서 약 세배에서 192 CPU 에서 수천배까지 이릅니다.

하지만, reader-writer 락킹에서와 같이, RCU 의 이 성능 이득은 같은 시스템에서 Figure 9.29 에 보인 것처럼 대부분 큰 수의 CPU 와 짧은 기간의 크리티컬 섹션으로부터 나왔습니다. 더해서, reader-writer 락킹에서와 같이, 많은 시스템 콜은 (그리고 따라서 그것들이

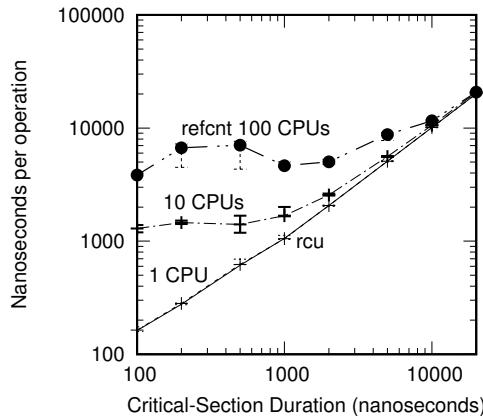


Figure 9.29: Response Time of RCU vs. Reference Counting, 192 CPUs

포함하는 모든 RCU read-side 크리티컬 섹션은) 수 마이크로세컨드 내에 완료됩니다.

그러나, RCU에 의해 발생하는 제약들은 상당히 귀찮을 수 있습니다. 예를 들어, 많은 경우, RCU read-side 크리티컬 섹션 내에서의 잠자는 것의 금지는 그 목표를 의미 없게 할 수도 있습니다. 다음 섹션은 이 문제를 해결하는 동시에 적어도 일부 경우에서는 전통적 레퍼런스 카운팅의 복잡도를 줄이는 방법들을 살펴봅니다.¹⁸

9.5.4.4 RCU is a Bulk Reference-Counting Mechanism

앞의 섹션에서 이야기 되었듯, 전통적 레퍼런스 카운터는 일반적으로 특정 데이터 구조, 또는 특정 데이터 구조체들의 그룹과 연관지어집니다. 하지만, 단일한 글로벌 레퍼런스 카운터를 다양한 데이터 구조를 위해 유지하는 것은 보통 이 레퍼런스 카운트를 포함하는 캐쉬 라인이 이리저리 이동되게 합니다. 그런 캐쉬 라인 움직임은 성능을 상당히 떨어뜨릴 수 있습니다.

대조적으로, RCU의 가벼운 read-side 기능은 무시 가능한 수준의 성능 하락과 함께 극단적으로 빈번한 read-side 사용을 가능하게 하여 RCU가 약간 또는 없는 성능 하락을 갖는 “bulk 레퍼런스 카운팅”으로 사용될 수 있게 합니다. 블록되어야 하는 코드 섹션을 가로질러 단일 태스크를 통해 레퍼런스가 얻어져야만 하는 상황에서는 Sleepable RCU (SRCU) [McK06]가 사용될 수 있을 겁니다. 이는 레퍼런스가 한 태스크에서 다른 태스크로 “전달되는”, 예를 들면 어떤 레퍼런스가 I/O 시작 시에 획득되고 연관된 완료 인터럽트 핸들러에서 해제되는 것과 같은 드물지는 않은 상황은 처리할 수 없을 겁니다.

¹⁸ 다른 경우들은 Section 9.3에서 설명된 해저드 포인터 메커니즘을 통해 더 잘 처리될 수도 있습니다.

(원칙적으로, 이는 SRCU 구현에 의해 처리될 수 있습니다만, 실질적으로는 이게 좋은 트레이드오프인지 아직 명확치 않습니다.)

물론, RCU는 그 자체의 제약을 가지는데, `srcu_read_lock()`에서 리턴된 값이 연관된 `srcu_unlock()`에 전달되어야 하며, 어떤 RCU 기능도 하드웨어 인터럽트 핸들러나 non-maskable 인터럽트 (NMI) 핸들러에서 수행되어선 안된다는 것입니다. 이 제약에 의해 얼마나 많은 문제가 있는지, 그것들이 어떻게 처리되는 것이 최선인지에 대해선 아직 판단 중입니다.

9.5.4.5 RCU is a Poor Man's Garbage Collector

RCU를 처음 배우는 사람들의 드물지 않은 느낌은 “RCU는 가비지 컬렉터 같은 거다!”입니다. 이 느낌은 상당한 사실을 담고 있지만, 잘못된 생각의 방향을 갖게 될 수도 있습니다.

RCU와 자동화된 가비지 컬렉터 (GC) 사이의 관계에 대해 생각하는 최선의 방법은 RCU는 GC를 쓰레기 를 모으는 타이밍을 자동으로 결정해 준다는 점에서 달았다는 것이겠습니다만, RCU는 GC와 다음과 같은 점에서 다릅니다: (1) 프로그래머는 일일이 언제 특정 데이터 구조가 정리될 수 있는지 알려야 하며, (2) 프로그래머는 일일이 데이터가 참조되고 있을 수도 있는 RCU read-side 크리티컬 섹션을 표시해야 합니다.

이 차이점에도 불구하고, 달은 부분도 꽤 깊습니다. 실제로, 제가 알고 있는 첫번째 RCU와 비슷한 메커니즘의 사용은 grace period를 제어하기 위한 레퍼런스 카운트 기반 가비지 컬렉터 [KL80] 였으며, RCU와 가비지 컬렉터 사이의 연결은 더 최근에도 이야기 되었습니다 [SWS16]. 그러나, RCU를 생각하는 더 나은 방법이 다음 섹션에 설명되어 있습니다.

9.5.4.6 RCU is an MVCC

RCU는 또한 단순화된, 완화된 일관성 기준을 갖는 multi-version concurrency control (MVCC) 메커니즘으로 생각될 수 있습니다. 이 멀티 버전 이야기는 Section 9.5.2.3에서 다루어진 바 있습니다. 하지만, 그 기본적 형태에서, RCU는 버전 일관성을 특정 RCU로 보호되는 데이터 원소에 대해서만 제공합니다.

그러나, 더 높은 단계에서의 버전 일관성을 복구하기 위한 여러 기법이 사용될 수 있는데, 예를 들면 sequence locking을 사용하거나 (Section 13.4.1를 참고하세요) 추가적인 우회 방법을 사용하는 겁니다 (Section 13.5.4를 참고하세요).

Listing 9.19: Existence Guarantees Enable Per-Element Locking

```

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 }

```

9.5.4.7 RCU Provides Existence Guarantees

Gamsa 등은 [GKAS99] 존재 보장에 대해 논하고 RCU 를 닮은 어떤 메커니즘이 어떻게 이런 존재 보장을 제공하기 위해 사용될 수 있는지 설명하며 (해당 PDF의 페이지 7의 Section 5를 보세요), Section 7.4은 락킹을 사용해 어떻게 존재 보장을 할 수 있는지를 그렇게 하는 것의 단점과 함께 설명합니다. 그 효과는 어떤 RCU로 보호되는 데이터 원소든 하나의 RCU read-side 크리티컬 섹션 내에서 액세스 된다면, 그 데이터 원소는 해당 RCU read-side 크리티컬 섹션의 기간 동안은 존재하는 것이 보장된다는 것입니다.

Listing 9.19은 어떻게 RCU 기반의 존재 보장이 원소별 락킹을 가능하게 하는지 해쉬 테이블에서 원소 하나를 제거하는 함수를 통해 보입니다. 라인 6은 해쉬 함수를 계산하고, 라인 7은 RCU read-side 크리티컬 섹션에 진입합니다. 라인 9가 이 해쉬 테이블의 연관된 버킷이 비었거나 거기 있는 원소가 우리가 삭제하고자 하는 것이 아님을 발견하면, 라인 10은 RCU read-side 크리티컬 섹션을 종료하고 라인 11에서 실패를 알립니다.

Quick Quiz 9.55: 우리가 제거하려는 원소가 Listing 9.19의 라인 9의 리스트의 첫번째 원소가 아니면 어떻게 되죠?

그렇지 않다면, 라인 13은 update-side 스피너를 획득하고, 라인 14는 이어서 이 원소가 여전히 우리가 원하는 것인지 검사합니다. 만약 그렇다면, 라인 15는 이 RCU read-side 크리티컬 섹션을 나오고, 라인 16는 이를 테이블에서 삭제하며, 라인 17에서 락을 해제하고,

라인 18에서 앞서서부터 존재해온 모든 RCU read-side 크리티컬 섹션이 완료되길 기다리며, 라인 19에서 이번에 제거된 원소를 메모리 해제하고, 라인 20에서 성공을 알립니다. 만약 이 원소가 더이상 우리가 원하던 것이 아니었다면, 라인 22에서 이 락을 내려놓고, 라인 23에서 RCU read-side 크리티컬 섹션을 나온 후, 라인 24에서 이 특정 키를 제거하는데 실패했음을 알립니다.

Quick Quiz 9.56: Listing 9.19의 라인 15에서는 라인 17에서 락을 내려놓기 전에 RCU read-side 크리티컬 섹션을 빠져나가는게 왜 괜찮나요?

Quick Quiz 9.57: Listing 9.19의 라인 23에서는 왜 라인 22에서 락을 내려놓기 전에 RCU read-side 크리티컬 섹션을 빠져나가지 않나요?

독자 여러분은 이게 Section 9.5.4.9에서 설명된, 원래의 “RCU는 무언가가 끝나길 기다리는 방법 중 하나다” 주제의 약간의 변종일 뿐임을 알게 될 것임을 경고드립니다. 그것들은 또한 Section 7.4에서 이야기된 락킹 기반 존재 보장 대비 데드락 내성의 장점을 알려줄지도 모릅니다.

9.5.4.8 RCU Provides Type-Safe Memory

많은 수의 lockless 알고리즘들이 특정 데이터 원소가 그것을 참조하는 특정 RCU read-side 크리티컬 섹션 내에서 동일한 정체를 유지할 것을 요구하지 않습니다—하지만 그 데이터 원소가 동일한 타입을 유지한다는 조건 하에서만입니다. 달리 말하자면, 이 lockless 알고리즘들은 주어진 데이터 원소가 그것이 참조되고 있는 사이에 메모리 해제되고 동일한 타입의 구조로 재할당되는 것을 감내할 수 있지만, 그 타입의 변경은 금지됩니다. 학계에서는 “type-safe memory” 라 불리는 [GC96]이 보장은 앞의 섹션에서의 존재 보장보다 약하며, 따라서 이걸 가지고 작업하기는 많이 힘듭니다. 리눅스 커널 내에서의 type-safe memory 알고리즘은 slab 캐쉬의 사용을 가능하게 하는데, 이 캐쉬들을 SLAB_TYPESAFE_BY_RCU로 표시하여 메모리 해제된 slab을 시스템 메모리로 반환할 때 RCU가 사용되게 합니다. 이런 RCU의 사용은 그런 slab의 어떤 사용중인 원소든 그 slab 내에 존재할 것을, 따라서 모든 앞서서부터 존재한 RCU read-side 크리티컬 섹션의 기간 동안 그 타입이 유지될 것을 보장합니다.

Quick Quiz 9.58: 하지만 여러 쓰레드에 임의의 긴 RCU read-side 크리티컬 섹션들이 존재하여 어느 시점에서든 시스템 내에 최소 하나의 RCU read-side 크리티컬 섹션을 수행하는 쓰레드가 존재하면 어떻게 되죠? 그건 SLAB_TYPESAFE_BY_RCU slab의 어떤 데이터든

시스템에 반환되는 것을 막아서 OOM 상황에 이르지 않을까요?

SLAB_TYPESAFE_BY_RCU 는 kmem_cache_alloc() 이 kmem_cache_free() 로 메모리 해제된 메모리가 즉시 재할당 되는 것을 막지 않음을 알아두는 것이 중요합니다! 실제로, rcu_dereference() 로 막 반환된, SLAB_TYPESAFE_BY_RCU 로 보호되는 데이터 구조는 임의의 많은 횟수동안 메모리 해제되고 재할당 되었을 수 있는데, rcu_read_lock() 의 보호 아래에서도 그렇습니다. 그 대신, SLAB_TYPESAFE_BY_RCU 는 kmem_cache_free() 가 완전히 메모리 해제된 데이터 구조들의 slab 을 RCU grace period 가 지나기 전에 시스템에 반환하는 것을 막습니다. 요약하자면, 주어진 RCU read-side 크리티컬 섹션이 특정 SLAB_TYPESAFE_BY_RCU 데이터 원소가 무한정 빈번하게 메모리 해제되고 재할당 되는 것을 볼 수 있더라도, 그 원소의 탑입은 그 크리티컬 섹션이 종료되기 전까지 바뀌지 않을 것을 보장합니다.

따라서 이 알고리즘은 일반적으로 새로 참조된 데이터 구조가 정말로 원했던 것인지 검사하는 검증 단계를 갖습니다 [LS86, Section 2.5]. 이 검증 단계는 이 데이터 구조의 특정 부분이 메모리 해제-재할당 프로세스에 의해 수정되지 않았을 것을 요구합니다. 그런 검증은 일반적으로 똑바로 하기가 무척 어려우며, 복잡하고 어려운 버그를 숨길 수 있습니다.

따라서, type-safe 기반의 lockless 알고리즘이 매우 적은 수의 어려운 상황에선 무척 도움이 될 수 있지만, 여러분은 가능한 곳에는 존재 보장을 대신 사용해야 합니다. 어쨌든 더 간단한 게 거의 항상 더 낫습니다!

9.5.4.9 RCU is a Way of Waiting for Things to Finish

Section 9.5.2 에서 이야기 되었듯 RCU 의 중요한 컴포넌트는 RCU 읽기 쓰레드가 끝나기를 기다리는 방법입니다. RCU 의 큰 장점 중 하나는 여러분이 수천개의 다른 것들이 모두 끝나기를 명시적으로 그것을 각각을 추적할 필요없이, 그리고 명시적 추적 방법에서는 피할 수 없는 성능 하락, 확장성 제한, 복잡한 데드락 시나리오, 그리고 메모리 누출 사고를 걱정할 필요 없이 기다릴 수 있다는 것입니다.

이 섹션에서는 어떻게 synchronize_sched() 의 읽기 쪽 비슷한 (하드웨어 오퍼레이션과 인터럽트를 불능화 시키는 기능들 포함 preemption 을 불능화 시키는 모든 것을 포함하는) 것이 락킹을 가지고는 구현하기 어려운 non-maskable interrupt (NMI) 핸들러와의 상호작용을 구현할 수 있게 하는지 보입니다. 이 방법은 “Pure RCU” [McK04] 이라 명명되었으며, 리눅스 커널의 많은 장소에서 사용됩니다.

Listing 9.20: Using RCU to Wait for NMIs to Finish

```

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 }

```

그런 “Pure RCU” 디자인의 기본적 형태는 다음과 같습니다:

1. 변경을, 예를 들어, OS 가 NMI 에 대응하는 방법에 대해서, 만듭니다
2. 모든 앞서서부터 존재해온 read-side 크리티컬 섹션이 완전히 종료하기를 기다립니다 (예를 들면, synchronize_sched() 기능을 이용해서). 여기서의 핵심은 뒤따르는 RCU read-side 크리티컬 섹션은 뭐가 됐든 만들어진 변경을 볼 것이 보장된다는 겁니다.
3. 정리를 하는데, 예를 들면 변경이 성공적으로 만들 어졌음을 알리는 상태를 리턴합니다.

이 섹션의 나머지 부분은 리눅스 커널에서 가져온 예제 코드를 보입니다. 이 예에서, nmi_stop() 함수는 모든 진행중인 NMI 알림이 연관된 리소스를 메모리 해제하기 전에 완료되었음을 보장하기 위해 synchronize_sched() 를 사용합니다. 이 코드의 간략화 된 코드가 Listing 9.20 에 보여져 있습니다.

라인 1-4 는 크기와 길이가 정해지지 않은 엔트리의 배열을 담는 profile_buffer 구조체를 정의합니다. 라인 5는 profile 버퍼 하나를 가리키는 포인터를 정의하는데, 이는 어딘가에서 동적으로 할당된 메모리 영역을 가리키도록 초기화 될 겁니다.

라인 7-16 는 nmi_profile() 함수를 정의하는데, 이 함수는 NMI 핸들러 내에서 호출됩니다. 따라서, 이

함수는 preemption 될 수도 없고 평범한 인터럽트 핸들러에 의해 인터럽트될 수도 없습니다만, 캐쉬 미스, ECC 에러, 그리고 같은 코어의 다른 하드웨어 쓰레드에 의해 빼앗기는 사이클로 지연될 수는 있습니다. 라인 9 는 profile 버퍼로의 지역 포인터를 DEC Alpha에서의 메모리 순서를 보장하기 위해 `rcu_deref()` 기능을 이용해 가져오고, 라인 11과 12는 현재 할당된 profile 버퍼가 없으면 이 함수에서 빠져나가며, 라인 13과 14는 `pcvalue` 인자가 범위 밖이면 이 함수에서 나갑니다. 그렇지 않다면, 라인 15는 `pcvalue` 인자에 의해 인덱스 되는 profile 버퍼 엔트리를 증가시킵니다. 버퍼에서 크기를 저장하는 것은 범위 검사가 버퍼와 매치됨을 보장하는데, 커다란 버퍼가 갑자기 작은 것으로 교체될 때에도 그려함을 알아 두시기 바랍니다.

라인 18-27는 `nmi_stop()` 함수를 정의하는데, 이 함수의 호출자는 상호 배타적일 (예를 들어, 올바른 락을 잡고 있을) 필요가 있습니다. 라인 20는 이 profile 버퍼로의 포인터를 가져오고, 라인 22과 23는 버퍼가 없으면 이 함수를 나갑니다. 그렇지 않다면, 라인 24은 이 profile 버퍼 포인터를 NULL로 만들고 (완화된 메모리 순서 규칙 기계에서의 메모리 순서를 유지하기 위해 `rcu_assign_pointer()` 기능을 사용합니다.) 라인 25는 하나의 RCU Sched grace period 가 지나가길 기다리는데, 특히, NMI 핸들러를 포함한 모든 preemption 불가한 코드 영역들이 완료되길 기다립니다. 수행이 라인 26 까지 이어진다면, 기존 버퍼로의 포인터를 얻은 `nmi_profile()` 인스턴스는 모두 리턴했을 것이 보장됩니다. 따라서 이 버퍼를 메모리 해제해도 안전하며, 여기선 `kfree()`를 사용합니다.

Quick Quiz 9.59: `nmi_profile()` 함수가 preemption 가능했다고 쳐 봅시다. 이 예제가 올바르게 동작하려면 무엇이 바뀌어야 할까요?

요약하자면, RCU는 동적인 profile 버퍼들 간의 교체를 쉽게 해줍니다 (이걸 어토믹 오퍼레이션 또는 락킹을 사용해서 이걸 효율적으로 하려고 시도 해 보세요!). 하지만, 앞의 섹션에서 보았듯 RCU는 보통 더 높은 수준의 추상화 단계에서 사용됩니다.

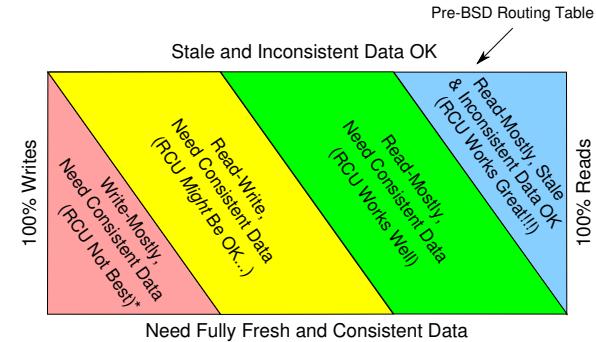
9.5.4.10 RCU Usage Summary

그 핵심에 있어서, RCU는 다음의 것들을 제공하는 API 이상도 이하도 아닙니다:

- 새로운 데이터를 추가하기 위한 발행-구독 메커니즘,
- 앞서서부터 존재해온 RCU 읽기 쓰레드들이 끝나기를 기다리는 하나의 방법, 그리고

- 동시의 RCU 읽기 쓰레드에 해를 끼치거나 지연되게 하지 않으면서 여러 버전을 유지할 수 있게 하는 규칙.

그러나, RCU 위에서 더 높은 수준의 것들을 만드는게 가능한데, 앞의 섹션에서 이야기한 reader-writer 락킹, 레퍼런스 카운팅, 그리고 존재 보장 구조등이 포함되겠습니다. 더 나아가, 리눅스 커뮤니티가 흥미로운 RCU는 물론이고 다른 동기화 기능들의 흥미로운 새로운 사용처를 발견하길 계속할 것이라 믿어 의심치 않습니다.



- 1. RCU provides ABA protection for update-friendly synchronization mechanisms
- 2. RCU provides bounded wait-free read-side primitives for real-time use

Figure 9.30: RCU Areas of Applicability

그 사이에는, Figure 9.30 가 RCU 가 어디에 유용할 수 있는지에 대한 대략적 경험적 규칙을 보입니다.

그림의 위에 있는 파란 상자를 통해 보여지듯, RCU는 오래되어 더이상 유효하지 않고 비일관적인 데이터가 허용 가능하며 (이에 대한 더 많은 내용을 위해 아래를 보세요) 읽기가 대부분인 상황에서 가장 잘 동작합니다. 이 경우에 대한 리눅스 커널에서의 규범적인 예는 라우팅 테이블입니다. 라우팅 업데이트가 인터넷을 통해 전파되기까지는 수초에서 수분까지도 걸릴 수 있기 때문에, 시스템은 패킷을 잘못된 방향으로 상당한 시간동안 보내고 있을 겁니다. 몇 밀리세컨드 정도 더 그것들을 잘못된 방향으로 보내길 계속하는 작은 가능성을 갖는 것은 거의 항상 문제가 아닙니다.

일관적인 데이터가 필요하지만 읽기가 대부분인 워크로드를 가지고 있다면, RCU는 초록색의 “read-mostly, need consistent data” 상자로 보여지듯 잘 동작합니다. 이 경우의 한가지 예는 리눅스 커널의 사용자 수준 System-V 세마포어 ID의 연관된 커널 내부 데이터 구조로의 매핑입니다. 세마포어는 그것들이 생성되고 파괴되는 것보다 훨씬 더 빈번하게 사용되며, 따라서 이 매핑은 읽기가 대부분입니다. 하지만, 세마포어 오퍼레이션을 이미 삭제된 세마포어에 대해 행하는 것은 에러가 됩니다. 이 일관성에 대한 필요는 커널 내부 세마포어 데이터 구조 안에 있는 락과 세마포어가 삭제될 때 설치되는

“deleted” 플래그를 사용해 처리됩니다. “deleted” 플래그가 설치된 커널 내부 데이터 구조로 어떤 사용자 ID 가 매핑되면, 이 데이터 구조는 무시되어 이 사용자 ID 는 무효한 것으로 표시됩니다.

이는 읽기 쓰레드가 이 세마포어 자체를 표현하는 데 데이터 구조를 위한 락을 잡아야 할것을 필요로 하지만, 데이터 구조의 매핑을 위해 락킹을 적용할 수 있게 합니다. 따라서 읽기 쓰레드는 ID에서 데이터 구조로의 매핑을 위해 사용되는 트리를 락 없이 순회할 수 있으며, 이는 결국 성능, 확장성, 그리고 리얼타임 반응성을 상당히 개선합니다.

노란색의 “read-write” 상자로 표시되었듯, RCU 는 일관적인 데이터가 필요한 read-write 워크로드에서도 유용할 수 있습니다만, 다른 여러 동기화 기능들과 함께 사용되는게 보통입니다. 예를 들어, 최근의 리눅스 커널에서의 directory-entry 캐쉬는 시퀀스 락, per-CPU 락, 그리고 데이터 구조별 락과 함께 RCU 를 사용해서 일반적인 경우에는 pathname 들을 락 없이 순회할 수 있게 합니다. 비록 RCU 가 이 read-write 케이스에 무척 유용할 수 있지만, 그런 사용은 읽기가 대부분인 경우에 비해 더 복잡한 경우가 많습니다.

마지막으로, 이 그림의 바닥에 빨간 상자로 표시되었듯, 업데이트가 대부분이며 일관적인 데이터를 필요로 하는 워크로드는 RCU 를 사용하기 좋은 경우가 드뭅니다만, 몇가지 예외가 있긴 합니다 [DMS¹²]. 또한, Section 9.5.4.8에서 이야기 되었듯, 리눅스 커널 내에서는 SLAB_TYPESAFE_BY_RCU 슬랩 할당자 플래그가 type-safe 메모리를 RCU 읽기 쓰레드에게 제공하는데, 이는 non-blocking 동기화와 다른 lockless 알고리즘들을 상당히 단순화 시킬 수 있습니다.

요약하자면, RCU는 새로운 데이터를 추가하기 위한 발행-구독 메커니즘, 앞서서부터 조제해온 RCU 읽기 쓰레드들이 끝나길 기다리는 한가지 방법, 그리고 동시에의 RCU 읽기 쓰레드에게 해를 입히거나 지연시키지 않으면서 업데이트를 하기 위한 여러 버전을 유지하는 규칙을 제공하는 API입니다. 이 RCU API는 읽기가 대부분인 상황에 가장 잘 동작하며, 특히 오래되어 더이상 유효하지 않고 비일관적인 데이터가 어플리케이션에 의해 통제될 수 있을 때 더욱 그렇습니다.

9.5.5 RCU Related Work

RCU 와 닮은 무언가에 대한 알려진 첫번째 언급은 Donald Knuth [Knu73, page 413 of Fundamental Algorithms] 의 Joseph Weizenbaum 의 FORTRAN 을 위한 SLIP 리스트 처리기 [Wei63]에 대한 버그 레포트의 형태를 가겠습니다. Knuth는 버그 레포팅에서 SLIP이 grace-period 보장 같은 것을 갖고 있지 않다고 했습니다.

RCU 와 닮은 무언가에 대한 버그 레포트가 아닌 알려진 첫번째 언급은 Kung 와 Lehman 의 유명한 논

문 [KL80]에서였습니다. 학계에서는 이 기법에 대한 일부 추가적 사용이 있었습니다만 [ML82, ML84, Lis88, Pug90, And91, PAB⁺95, CAK⁺96, RSB⁺97, GKAS99], 이 분야에서의 대부분의 일은 협업에 종사하는 사람들에 의해 이루어졌습니다 [RTY⁺87, HOS89, Jac93, Joh95, SM95, SM97, SM98, MS98a]. 2000년에 이르러, 그 주도권은 오픈소스 프로젝트들로 넘겨졌는데, 그 중 가장 두각을 드러낸 건 리눅스 커널 커뮤니티였습니다 [Rus00a, Rus00b, MS01, MAK⁺01, MSA⁺02, ACMS03].¹⁹

하지만, 2010년 중반에 이르러, 여러 커뮤니티와 연구기관에서의 RCU 연구과 개발에 대한 반가운 급증이 있었습니다 [Kaa15]. Section 9.5.5.1은 RCU의 사용을 설명하고, Section 9.5.5.2은 RCU 구현을 설명하며 (구현을 만들고 사용한 일들도 함께), Section 9.5.5.3은 RCU와 그것의 사용에 대한 검증과 테스트를 설명합니다.

9.5.5.1 RCU Uses

Portland State University (PSU) 의 Phil Howard 와 Jon Walpole 는 RCU 를 소프트웨어 transactional memory 를 사용해 업데이트를 동기화하며 red-black tree 에 적용했습니다 [How12, HW11]. Josh Triplett 와 Jon Walpole (PSU 소속) 는 RCU 를 크기 재조정 가능한 해쉬 테이블에 적용했습니다 [Tri12, TMW11, Cor14c, Cor14d]. 다른 RCU로 보호되는 resizable 해쉬 테이블이 Herbert Xu [Xu10] 와 Mathieu Desnoyers [MDJ13a]에 의해 만들어진 바 있습니다.

MIT 의 Austin Clements, Frans Kaashoek, 그리고 Nickolai Zeldovich 는 RCU로 최적화된 balanced binary tree (Bonsai) [CKZ12]을 만들고 이 tree 를 리눅스 커널의 VM 서브시스템에 리눅스 커널의 mmap_sem read-side contention 을 줄이려 적용했습니다. 이것은 많은 minor page fault 를 일으키는 마이크로벤치마크를 가지고 수십배의 속도 상승과 최소 80개 CPU 까지의 확장성을 보였습니다. 이는 Peter Zijlstra [Zij14]에 의해 더 일찍 개발된 패치와 유사하며, 둘 다 파일시스템 데이터 구조는 RCU 읽기 쓰레드에 안전하지 않다는 사실에 의해 제한되었습니다. Clements 등은 이 제한을 page fault 코드 경로를 anonymous page 에 한해 최적화 하는 것으로 회피했습니다. 더 최근에는, 파일시스템 데이터 구조가 RCU 읽기 쓰레드에 안전하게 되어서 [Cor10a, Cor11], 어쩌면 이 작업이 anonymous page 만이 아니라 모든 종류의 page 를 위해 구현될 수도 있을 겁니다—Peter Zijlstra 는 실제로 최근에 이것의 프로토 타입을 만들었으며 Laurent Dufour 는 이 일을 계속했습니다.

MIT 의 Yandong Mao 와 Robert Morris, 그리고 Harvard 대학의 Eddie Kohler 는 B+ tree 와 trie 의 아이디

¹⁹ 200개가 넘는 항목의 인용 리스트를 이 책을 위한 LATEX 소스의 bib/RCU.bib에서 찾을 수 있을 겁니다.

어를 결합한, Masstree [MKM12]라는 이름의 또 다른 RCU로 보호되는 tree를 만들었습니다. 이 tree는 RCU로 보호되는 해쉬 테이블보다 2.5배 느리긴 하지만, 해쉬 테이블과 달리 key range 오퍼레이션을 제공합니다. 또한, Masstree는 긴 공유 키 접두어와 함께 효율적인 객체 저장을 지원하며, 더 나아가 대용량 저장장치로의 로깅을 통해 영구성을 제공합니다.

그 논문은 Masstree는 업데이트를 영구히 저장하고 memcached는 그렇지 않음에도 Masstree의 성능이 memcached의 그것과 견줄만하다고 이야기 합니다. 이 논문은 또한 Masstree의 성능을 영구 데이터 저장기인 MongoDB, VoltDB, 그리고 Redis와 비교하고 Masstree의 상당한 성능 이득을 보고하는데, 어떤 경우에는 수백배에 달합니다. MIT의 Stephen Tu, Wenting Zheng, Barbara Liskov, 그리고 Samuel Madden와 Kohler의 또 다른 논문은 [TZK⁺13] Masstree를 Silo라는 이름의 in-memory 테이터베이스에 적용하여 널리 알려진 트랜잭션 처리 벤치마크에서 초당 700K 트랜잭션(분당 42M 트랜잭션)을 달성했습니다. 흥미롭게도, Silo는 락을 잡고 있는 동안 grace period의 오버헤드 없이도 linearizability를 보장합니다.

Technion의 Maya Arbel과 Hagit Attiya는 RCU로 보호되는 탐색 트리에 Masstree와 같이 동시의 업데이트를 가능하게 하는 좀 더 정밀한 접근을 취했습니다 [AA14]. 이 논문은 이 tree에서의 모든 오퍼레이션이 linearizable하다는 증명을 포함한 정확도의 증명을 포함합니다. 불행히도, 이 구현은 락을 잡고 있는 동안 grace period의 전체 응답시간을 일으키는 것으로 linearizability를 달성하는데, 이는 업데이트만 있는 워크로드에서는 확장성을 떨어뜨립니다. 이 문제를 해결하는 한가지 방법은 linearizability를 포기하는 것입니다만 [HKLP12, McK14b], Arbel과 Attiya는 그 대신 grace period 응답시간을 줄이는 RCU 변종을 만들었습니다. 물론, 공짜는 없으며 이 RCU 변종은 32 CPU 정도에서 확장성 한계를 보입니다. Linearizability를 포기해서 성능과 확장성을 모두 얻는 것에 대해서도 이야기 해야 할게 많지만, 학계가 대안적 RCU 구현을 탐구하는 것은 무척 보기 좋습니다.

9.5.5.2 RCU Implementations

Mathieu Desnoyers는 트레이싱에서의 사용을 위해 user-space RCU를 [Des09b, Des09a, DMS⁺12] 만들었으며, 이는 여러 프로젝트에서 사용되었습니다 [BD13].

프라하에 있는 Charles 대학의 연구자들 또한 RCU 구현을 위해 일했는데, Andrej Podzimek [Pod10]과 Adam Hraska [Hra13]의 박사 학위 논문들이 포함됩니다.

Yujie Liu (Lehigh University), Victor Luchangco (Oracle Labs), 그리고 Michael Spear (also Lehigh) [LLS13]는 확장 가능한 non-zero indecator (SNZI) [ELLM07]

를 grace-period 메커니즘으로써의 서비스로 만들었습니다. 의도는 확장성을 제한하게 될 것으로 여겨지는 linearizability 요구사항을 갖는 소프트웨어 transactional memory (Section 17.2을 참고하세요)를 구현하려는 것이었습니다.

RCU 비슷한 메커니즘들은 Java에서도 찾아볼 수 있습니다. Sivaramakrishnan 등은 [SZJ12] Java의 garbage collector와 상호작용할 때 필요한 read barrier를 제거하기 위해 RCU와 비슷한 메커니즘을 사용했고, 그 결과 상당한 성능 향상을 이뤘습니다.

Shanghai Jiao Tong University의 Ran Liu, Heng Zhang, 그리고 Haibo Chen는 최적화된 “수동적 reader-writer 락” [LZC14]를 위해 사용한 특수한 RCU 변종을 만들었는데, Gautham Shenoy [She06]와 Srivatsa Bhat [Bha14]에 의해 만들어진 것과 비슷합니다. Liu 등의 논문은 여러 측면에서 흥미롭습니다 [McK14e].

Mike Ash는 Apple의 Objective-C 런타임에 있는 RCU 비슷한 기능에 대한 설명을 작성했습니다 [Ash15]. 이 접근법은 read-side 크리티컬 섹션을 지정된 코드 영역으로 인식하여 zero read-side 오버헤드를 얻는 또 다른 방법의 자격을 갖지만, 여러 함수들을 사용하는 커다란 read-side 크리티컬 섹션에서의 흥미로운 실용적 문제들을 내포하고 있기는 합니다.

Pedro Ramalhete와 Andreia Correia [RC15]는 읽기 쓰레드에게 lock-free 진행 보장을 제공하기 위해 한쌍의 reader-writer 락을 사용하기는 하는 “Poor Man’s RCU”를 만들었습니다 [MP15a].

Maya Arbel과 Adam Morrison [AM15]는 update-side 락을 grace period 사이에 잡는 알고리즘들을 효율적으로 지원하기 위해 grace-period를 줄이려 하는 “Predicate RCU”라는 걸 만들었습니다. 이는 grace period 내로 batching되는 update들을 줄였으며 확장성을 줄였습니다만, 짧은 grace period를 제공하는데에는 성공했습니다.

Quick Quiz 9.60: 그냥 grace period를 기다리기 전에 락을 놓거나 grace period를 기다리는 대신 call_rcu() 같은 걸 사용하지 않죠?

Alexander Matveev (MIT), Nir Shavit (MIT and Tel-Aviv University), Pascal Felber (University of Neuchâtel), 그리고 Patrick Marlier (역시 University of Neuchâtel) [MSFM15]는 명시적으로 read-only 트랜잭션을 표시하는 소프트웨어 transactional memory라 생각될 수 있는 RCU 비슷한 메커니즘을 만들었습니다. 그들의 사용처는 grace period를 거쳐 락을 잡을 것을 필요로 했는데, 이는 확장성을 제한했습니다 [MP15a, MP15b]. 이는 rcutorture 테스트 수트를 잘 상요한 첫번째 학계의 RCU 관련 작품이었으며, 또한 v4.4에 받아들여진,

리눅스 커널 RCU 로의 성능 향상을 제출한 첫번째 사례였습니다.

Alexander Matveev 의 RLU 는 Jaeho Kim 등이 [KMK¹⁹] MV-RLU 로 이어졌습니다. 이 작업은 grace period 전반의 락을 잡지 않아도 되게 하고 비동기적 period 를 허용함으로써, 예를 들면 `synchronize_rcu()` 대신 `call_rcu()` 를 사용하여 동시적 업데이트를 허용함으로써 RLU 의 확장성을 개선했습니다. 이 논문은 또한 page 17.2.3.3 의 Section 17.2.3.3 에서 더 나루게 될 흐으미로운 성능 측정 선택을 했습니다.

Adam Belay 등은 IX 운영체제에서 TCP/IP 의 address-resolution protocol (ARP) 에서 사용되는 데이터 구조들을 보호하는 RCU 구현을 만들었습니다 [BPP¹⁶].

Geoff Romer 와 Andrew Hunter (둘 다 Google 소속) 는 singleton 데이터 구조의 RCU 보호를 위한 cell 기반 API 를 C++ 표준에 추가하여 하였습니다 [RH18].

Dimitrios Siakavaras 등은 HTM 과 RCU 를 탐색 tree 에 적용하였고 [SNGK17, SBN²⁰], Christina Giannoula 등은 HTM 과 RCU 를 color graph 에 사용했으며 [GGK18], SeongJae Park 등은 HTM 과 RCU 를 NUMA 시스템에서의 높은 락킹 contention 을 최적화하는데 사용했습니다.

Alex Kogan 등은 RCU 를 확장성 있는 주소 공간을 위한 range 락킹의 토대에 적용했습니다 [KDI20].

9.5.5.3 RCU Validation

2017년 초에는 거의 모든 버그가 잠재적인 보안 문제로 일반적으로 여겨졌으며, 따라서 검증과 테스트가 첫번째 걱정거리였습니다.

Stony Brook University 의 연구자들은 RCU 를 인지하는 data-race 탐지기를 만들었습니다 [Dug10, Sey12, SRK¹¹]. IMDEA 의 Alexey Gotsman, Tel Aviv University 의 Noam Rinetzky, 그리고 University of Oxford 의 Hongseok Yang 은 RCU 의 공식적인 semantic 을 분리된 논리구조로 표현하는 논문을 [GRY12] 출간하고 동시성의 다른 분야에도 그 일을 이어갔습니다.

Joseph Tassarotti (Carnegie-Mellon University), Derek Dreyer (Max Planck Institute for Software Systems), 그리고 Viktor Vafeiadis (also MPI-SWS) [TDV15] 는 userspace RCU [Des09b, DMS¹²] 의 quiescent-state 기반 reclamation (QSBR) 변종의 공식적 정확성 검증을 해냈습니다. Lihao Liang (University of Oxford), Paul E. McKenney (IBM), Daniel Kroening, 그리고 Tom Melham (둘 다 Oxford) [LMKM16] 는 C bounded model checker (CBMC) [CKL04] 를 사용해 리눅스 커널 Tree RCU 의 상당한 부분에 대한 기계적 정확성 검사를 해냈습니다. Lance Roy [Roy17] 는 CMBC 를 사용해 리눅스 커널 sleepable RCU (SRCU) [McK06] 의 상당한 부분에 대한 비슷한 정확성 검사를 해냈습니다. 마지막으로,

Michalis Kokologiannakis 과 Konstantinos Sagonas (National Technical University of Athens) [KS17a, KS19] 는 Nighugg 라는 도구를 [LSLK14] 사용해 리눅스 커널 Tree RCU 의 더 큰 부분에 대한 기계적 정확성 증명을 해냈습니다.

이러한 시도들 중 어떤 것도 그 검증 도구를 테스트하기 위해 의도적으로 RCU 에 추가된 버그들 외에는 어떤 버그도 찾지 못했습니다. 대조적으로, Alex Groce (Oregon State University), Iftekhar Ahmed, Carlos Jensen (both also OSU), 그리고 Paul E. McKenney (IBM) [GAJM15] 는 `rcutorture` 테스트 수트의 범위를 테스트하기 위해 리눅스 커널 RCU 의 소스코드를 자동으로 변경시켰습니다. 그 노력은 이 테스트 수트의 범위에 있는 여러 구멍을 찾아냈는데, 그 중 하나는 Tiny RCU 의 숨겨져 있던 실제 버그를 하나 찾아냈습니다 (그리고 고쳐졌습니다).

행운이 함께 한다면, 이 모든 검증 작업은 결국 더 많고 더 나은 동시성 코드 검증 도구들을 만들어지게 할 것입니다.

9.5.6 RCU Exercises

이 섹션은 여러분을 이 책의 앞부분에 나온 여러 예제들에 RCU 를 적용하도록 하는 여러 Quick Quizz 들로 이루어져 있습니다. 각 Quick Quiz 에의 답은 어떤 힌트를 제공하며, 그 해결책이 길게 설명되는 뒤의 섹션들로의 연결도 제공합니다. `rcu_read_lock()`, `rcu_read_unlock()`, `rcu_dereference()`, `rcu_assign_pointer()`, 그리고 `synchronize_rcu()` 기능들은 이 문제들의 대부분에 충분할 겁니다.

Quick Quiz 9.61: Listing 5.4 (`count_end.c`) 에 보인 통계적 카운터 구현은 `read_count()` 의 합산을 보호하기 위해 전역적 락을 사용했는데, 이는 나쁜 성능과 음의 확장성으로 이어졌습니다. 여러분은 `read_count()` 에 훌륭한 성능과 좋은 확장성을 제공하기 위해 RCU 를 어떻게 사용할 수 있겠습니다. (`read_count()` 의 확장성은 모든 쓰레드의 카운터를 스캔해야 한다는 필요로 인해 제한되어야 할 수 있음을 명심하십시오.)

■

Quick Quiz 9.62: Section 5.4.6 은 제거될 수 있는 기기로의 I/O 액세스를 세기 위한 두 쌍의 코드 조각을 보였습니다. 이 코드 조각들은 reader-writer 락을 잡아야 하는 이유로 (I/O 를 시작하는) fastpath 에서의 높은 오버헤드로 고생이 심했습니다. 여러분은 훌륭한 성능과 확장성을 제공하기 위해 RCU 를 어떻게 사용하겠습니까? (I/O 액세스를 하는 일반적인 경우를 위한 코드 조각의 성능은 기기 제거 코드 조각의 것보다 훨씬 중요함을 명심하십시오.)

■

9.6 Which to Choose?

Choose always the way that seems the best, however rough it may be; custom will soon render it easy and agreeable.

Pythagoras

Section 9.6.1 은 높은 수준에서의 개요를 제공하고 Section 9.6.2 에서는 이 챕터에서 보인 미뤄 처리하기 기법들 간의 차이를 자세히 설명합니다. 여기서의 이야기는 충분히 길어서 읽기 쓰레드가 순회 사이에 참조를 잡고 있지 않는, 그리고 원소들이 언제든 어디에든 더해지고 빼질 수 있는 연결된 데이터 구조를 가정합니다. Section 9.6.3 은 이어서 해저드 포인터, 시퀀스 락킹, 그리고 RCU 의 제품 단계에서 공개된 일부 사용 예를 짚어 보입니다. 이는 여러분이 이 기법들 중 어느것을 골라야 할지에 대해 도움을 줄 것입니다.

9.6.1 Which to Choose? (Overview)

Table 9.6 은 deferred-reclamation 기법을 다른 것과 차별화 시키는 높은 수준에서의 속성들 몇가지를 보입니다.

“Readers” 열은 Figure 9.22 에서 보인 결과들을 요약해 보이는데, 레퍼런스 카운팅을 제외한 모든 것들이 충분히 빠르고 확장되는 읽기 쓰레드를 보입니다.

“Number of Protected Objects” 열은 각 기법의 읽기 보호를 기록하기 위해 필요한 외부 저장장치를 평가합니다. RCU 는 quiescent state 에 의존하므로, 읽기 쓰레드들을 표현하기 위해 객체 내에도 외부에도 저장소를 필요로 하지 않습니다. 레퍼런스 카운팅은 각 객체의 구조 내에 하나의 정수를 사용할 수 있으며, 추가적인 저장소를 필요로 하지 않습니다. 해저드 포인터는 외부로부터 객체로의 포인터를 준비해야 하며, 주어진 CPU 나 쓰레드가 동시에 참조할 수도 있는 최대 갯수의 객체를 모두 다룰 수 있기에 충분한 포인터들을 가져야 합니다. 물론, 시퀀스 락은 포인터 순회 보호를 제공하지 않는데, 그것이 정적 데이터에만 보통 쓰이는 이유입니다.

Quick Quiz 9.63: 사용자들은 왜 필요에 따라 동적으로 해저드 포인터를 할당할 수 없나요?

“Duration of Production” 은 사용자가 특정 객체를 얼마나 긴 시간동안 보호할 수 있는지에 대한(존재한다면) 제약을 이야기 합니다. 레퍼런스 카운팅과 해저드 포인터는 둘 다 부작용 없이 연장된 시간 동안 보호할 수 있습니다만, 심지어 단 하나의 객체로의 RCU 레퍼런스를 유지하는 것도 모든 다른 RCU 가 메모리 해제되는 것을 방지할 수 있습니다. 따라서 RCU 읽기 쓰레드는 시스템 메모리 부족을 방지하기 위해 상대적으로 짧아야

하며, SRCU, Tasks RCU, 그리고 Tasks Trace RCU 같은 특수 목적 구현은 이 규칙에 있어 예외가 됩니다. 다시 말하지만, 시퀀스 락은 포인터 순회 보호를 제공하지 않는데, 그것이 정적 데이터에만 보통 쓰이는 이유입니다.

“Need for Traversal Retries” 열은 특정 객체로의 새로운 참조가 RCU 에서 그런 것처럼 무조건적으로 획득될 수 있는지, 또는 레퍼런스 카운팅, 해저드 포인터, 그리고 시퀀스 락에서와 같이 그 참조 획득이 실패할 수 있고 따라서 재시도를 해야 하게 하는지 이야기 합니다. 레퍼런스 카운팅과 해저드 포인터의 경우, 어떤 객체가 삭제되려 하는 중에 참조를 얻으려 할 때에만 재시도가 필요한데, 이 주제는 다음 섹션에서 더 자세히 다뤄집니다. 시퀀스 락킹은 물론 어떤 업데이트와 동시에 수행될 때에나 그 크리티컬 섹션을 재시도 해야만 합니다.

Quick Quiz 9.64: 하지만 리눅스 커널의 `kref` 레퍼런스 카운터는 보장된 무조건적 참조 획득을 허용하지 않나요?

물론, 각 열은 각 상황마다 다른 정도의 중요성을 가질 겁니다. 예를 들어, 여러분의 현재 코드가 해저드 포인터에서 읽기 쪽에 확장성 문제가 있다면, 해저드 포인터가 참조 획득 재시도를 필요로 한다는 것은 여러분의 현재 코드가 이를 이미 처리하고 있으므로 문제가 되지 않습니다. 비슷하게, 커널과 낮은 단계 어플리케이션들이 그런 것처럼 반응 시간 고려가 이미 읽기 쓰레드 순회 시간을 제한하고 있다면, RCU 가 기간 제한 요구를 갖는다는 것은 여러분의 코드가 이미 그것을 처리하고 있으므로 문제 되지 않습니다. 같은 맥락에서, 읽기 쓰레드가 순회하고 있는 객체들에 이미 쓰기를 했어야만 한다면, 레퍼런스 카운터의 읽기 쪽 오버헤드는 그렇게 중요하지 않을 겁니다. 물론, 보호되어야 할 데이터가 정적으로 할당된 변수에 있다면, 시퀀스 락킹의 풍니터 보호 불가함은 상관없습니다.

마지막으로, 해저드 포인터와 RCU 를 동적 자연 샘플링에 기반해 동적으로 바꿔 사용하려는 시도가 [BGHZ16] 있습니다. 이는 해저드 포인터와 RCU 사이의 선택을 수행시점으로 미루며, 선택의 책임을 소프트웨어에게 넘깁니다.

어쨌든, 이 표는 이 기법들 사이에서 선택을 할 때 큰 도움이 될 겁니다. 하지만 더 자세한 결 원하는 분들은 다음 섹션을 이어서 읽기 바랍니다.

9.6.2 Which to Choose? (Details)

Table 9.7 은 이 챕터에서 소개된 네개의 deferred-processing 기법들 중 하나를 선택해야 할 때 도움을 줄 수 있는 더 자세한 경험상의 법칙을 제공합니다.

“Existence Guarantee” 열에 보이듯, 여러분이 연결된 데이터 원소들에 대한 존재 보장을 해야 한다면, 여러분은 레퍼런스 카운팅, 해저드 포인터, 또는 RCU 를

Table 9.6: Which Deferred Technique to Choose? (Overview)

Property	Reference Counting	Hazard Pointers	Sequence Locks	RCU
Readers	Slow and unscalable	Fast and scalable	Fast and scalable	Fast and scalable
Number of Protected Objects	Scalable	Unscalable	No protection	Scalable
Duration of Protection	Can be long	Can be long	No protection	User must bound duration
Need for Traversal Retries	If race with object deletion	If race with object deletion	If race with any update	Never

Table 9.7: Which Deferred Technique to Choose? (Details)

Property	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	Use Case	N/A	Use Case
Lines of Code	94	79	79	73

사용해야만 합니다. 시퀀스 락은 존재 보장을 제공하지 않는 대신 업데이트 탐자와 업데이트를 마주친 read-side 크리티컬 섹션의 재시도를 제공합니다.

물론, “Updates and Readers Progress Concurrently” 열에서 보였듯 이 업데이트 탐지는 시퀀스 락킹이 업데이트 쓰레드와 읽기 쓰레드가 동시에 진행을 내는 것을 허용하지 않음을 암시합니다. 어쨌건, 그런 진행을 방지하는 것이 시퀀스 락킹을 상요하는 첫번째 이유입니다! 이 상황은 시퀀스 락킹을 레퍼런스 카운팅, 해저드 포인터, 또는 RCU 와 결합해 존재 보장과 업데이트 탐지를 제공하는 방법들을 가리킵니다. 실제로, 리눅스 커널은 pathname 탐색 시에 RCU 와 시퀀스 락킹을 이 방법으로 결합해 사용합니다.

“Contention Among Readers”, “Reader Per-Critical-Section Overhead”, 그리고 “Reader Per-Object Traversal Overhead” 열은 이 기법들의 read-side 오버헤드에 대한 대략적 느낌을 제공합니다. 레퍼런스 카운팅의 오버헤드는 상당히 클 수 있는데, 각 객체 순회마다 필요한 완전히 순서잡힌 read-modify-write 어토믹 오퍼레이션을 포함합니다. 해저드 포인터는 각 데이터 원소 순회마다 메모리 배리어 오버헤드를 일으키며, 시퀀스 락은 크리티컬 섹션을 수행하려 할 때마다 한 상의 메모리 배리어 오버헤드를 일으킵니다. RCU 구현체들의 오버헤드는 제로부터 각 read-side 크리티컬 섹션마다의 한 쌍의 메모리 배리어까지 다양하며, 따라서 RCU 를 특히 많은 데이터 원소를 순회하는 read-side 크리티컬 섹션을 가진 경우에 대해 최고의 성능을 제공할 수 있게 합니다. 물론, 모든 deferred-processing 변종들의 read-side 오버헤드는 각 read-side 오퍼레이션이 더 많은 데이터를 처리하게끔 batching 을 사용해 줄일 수 있습니다.

Quick Quiz 9.65: 하지만 Section 9.3 의 quick quiz 들의 답 중 하나는 짹을 이룬 비대칭적 배리어로 해저드 포인터의 읽기 쪽 `smp_mb()` 를 제거할 수 있다고 하지 않았던가요?

“Reader Forward Progress Guarantee” 열은 RCU 만이 제한된 wait-free 진행 보장을 가짐을 보이는데, 이는 RCU 가 제한된 수의 인스트럭션을 수행함으로써 끝이 정해져 있는 순회를 해낼 수 있음을 의미합니다.

“Reader Reference Acquisition” 열은 RCU 만이 무조건으로 참조 획득을 할 수 있음을 보입니다. 시퀀스 락을 위한 항목은 “Unsafe” 인데, 다시 말하지만 시퀀스 락은 참조를 획득하는게 아니라 업데이트를 탐지하기 때문입니다. 레퍼런스 카운팅과 해저드 포인터는 둘 다 해당 획득이 실패하면 순회를 재시작 해야 합니다. 이를 자세히 보기 위해, 객체 A, B, C, 그리고 D 를 그 순서대로 가지고 있는 링크드 리스트가 다음 사건들을 겪는다고 생각해 봅시다:

1. 한 읽기 쓰레드가 객체 B 로의 참조를 획득합니다.

2. 어느 업데이트 쓰레드가 객체 B 를 제거하지만 앞의 읽기 쓰레드가 참조를 가지고 있으므로 메모리 해제는 하지 않습니다. 이 리스트는 이제 객체 A, C, 그리고 D 를 가지고 있으며, 객체 B 의 `->next` 포인터는 `HAZPTR_POISON` 으로 설정되어 있습니다.
3. 이 업데이트 쓰레드는 객체 C 를 제거해서 이 리스트는 객체 A 와 D 만 가지게 됩니다. 객체 C 로의 참조는 존재하지 않으므로 즉각 메모리 해제됩니다.
4. 앞의 읽기 쓰레드는 지금은 제거된 객체 B 의 다음 객체로 넘어가려 하지만 오염된 `->next` 포인터가 이를 막습니다. 이는 좋은 일인데, 그러지 않는다면 객체 B 의 `->next` 포인터는 freelist 를 가리킬 것이기 때문입니다.
5. 따라서 이 읽기 쓰레드는 리스트의 시작부터 순회를 재시작해야만 합니다.

따라서, 참조를 획득하는데 실패했을 때, 해저드 포인터나 레퍼런스 카운터 순회는 그 순회를 처음부터 재시작 해야만 합니다. 예를 들자면 링크드 리스트를 갖는 트리와 같은 중첩된 연결 기반 데이터 구조체의 경우, 이 순회는 가장 바깥의 데이터 구조로부터 재시작되어야만 합니다. 이 상황은 RCU 에게 상당한 사용하기 쉬운 장점을 부여합니다.

하지만, RCU 의 사용하기 쉬운 장점은 공짜로 오지 않습니다. “Memory Footprint” 열에서 볼 수 있습니다. RCU 의 무조건적 참조 획득 지원은 어떤 RCU 읽기 쓰레드가 완료되기 전까지는 그 쓰레드에 의해 접근될 수 있는 모든 객체를 메모리 해제하지 않아야만 함을 의미합니다. 그래서 RCU 는 제한되지 않은 메모리 사용량을 가지는데, 최소한 업데이트가 그걸 제한하지 않는다면 그렇습니다. 반대로, 레퍼런스 카운팅과 해저드 포인터는 그 데이터 원소들을 동시에 읽기 쓰레드들이 정말로 참조하는 만큼만 유지하면 됩니다.

이 메모리 사용량과 참조 획득 실패 사이의 갈등이 리눅스 커널에서는 RCU 와 레퍼런스 카운터의 조합으로 해결되었습니다. RCU 는 짧은 수명을 갖는 것들의 참조를 위해 사용되는데, 이는 RCU read-side 크리티컬 섹션 이 짧아질 수 있음을 의미합니다. 이 짧은 RCU read-side 크리티컬 섹션은 결국 연관된 RCU grace period 역시 짧아질 수 있어서, 메모리 사용량을 제한하게 됩니다. 긴 수명을 갖는 소수의 데이터 원소를 위해선 레퍼런스 카운팅이 사용됩니다. 이는 레퍼런스 획득 실패의 복잡도는 그런 적은 데이터 원소를 위해서만 처리되면 됨을 의미합니다: 대용량 참조 획득은 무조건적인데, RCU 데입니다. 레퍼런스 카운팅을 다른 동기화 메커니즘과

조합하는 것에 대한 더 많은 정보를 위해선 Section 13.2 을 보시기 바랍니다.

“Reclamation Forward Progress” 열은 해저드 포인터가 non-blocking 업데이트를 제공할 수 있음을 [Mic04, HLM02] 보입니다. 레퍼런스 카운팅은 구현에 따라 그럴 수도 안그럴 수도 있습니다. 하지만, 시퀀스 락킹은 non-blocking 업데이트를 제공할 수 없는데, 그것의 update-side 락 때문입니다. RCU 업데이트 쓰래드는 읽기 쓰래드를 기다려야만 하는데, 이 역시 완벽한 non-blocking update에 속하지 않습니다. 하지만, 그 유일한 블록킹 오퍼레이션은 메모리 해제를 위한 기다림인데, 이는 많은 목적에 있어 non-blocking 만큼이나 좋은 [DMS⁺12] 상황에 이르게 합니다.

“Automatic Reclamation” 열에서 보였듯, 레퍼런스 카운팅만이 메모리 해제를 자동화 할 수 있으며, 그렇다 하더라도 순회하지 않는 데이터 구조에 대해서만 그렇습니다. 해저드 포인터와 RCU를 위한 몇몇 사용 예는 *link count*를 사용한 자동화된 메모리 회수를 제공할 수 있는데, 레퍼런스 카운트와 비슷하지만 데이터 구조의 다른 부분으로부터 들어오는 연결에 대해서만 적용되는 것으로 생각될 수 있습니다 [Mic18].

마지막으로, “Lines of Code” 열은 Pre-BSD Routing Table 구현의 크기를 보이는데, 상대적 사용의 쉬움에 대한 대략적 느낌을 제공합니다. 그렇다고는 하나, 레퍼런스 카운팅과 세퀀스 락킹 구현은 버그가 많으며, 올바른 레퍼런스 카운팅 구현은 복잡한 것으로 여겨집니다 [Val95, MS95] 알아 두시기 바랍니다. 올바른 시퀀스 락킹 구현은 시퀀스 락킹이 동시에 업데이트를 탐지하기 위한, 또한 안전한 참조 획득을 위한, 예를 들면 해저드 포인터나 RCU와 같은 어떤 다른 동기화 메커니즘을 추가로 필요로 합니다.

이 기법들을 별도로 또는 조합해서 이용하는 더 많은 경험을 얻게 될수록 이 섹션에서 소개한 이 경험적 법칙은 더 다듬어져야 할 겁니다. 그러나, 이 섹션은 현재의 최선의 결과를 반영하고 있습니다.

9.6.3 Which to Choose? (Production Use)

이 섹션은 해저드 포인터, 시퀀스 락킹, 그리고 RCU의 드러난 제품상 사용들을 일부 짚어 봅니다. 레퍼런스 카운팅은 생략되는데, 그게 중요치 않아서가 아니라, 오히려 널리 사용될 뿐만 아니라 거의 반세기 전부터 교재에 많이 문서화 되었기 때문입니다. 이 다른 기법들의 제품상 사용을 나열함으로써 바라는 이득은 연구를 위한—또는 버그를 찾기 위한, 예를 제공하는 것입니다.²⁰

²⁰ Mathias Stearn, Matt Wilson, David Goldblatt, LiveJournal 사용자 fanf, Nadav Har'El, Avi Kivity, Dmitry Vyukov, Raul Guitierrez S., Twitter 사용자 @peo3, Paolo Bonzini, 그리고 Thomas Monjalon에게 이 수많은 사용 예를 알려준 데 대해 감사드립니다.

9.6.3.1 Production Uses of Hazard Pointers

2010년, Keith Bostic은 WiredTiger [Bos10]에 해저드 포인터를 추가했습니다. 2015년에 발표된 MongoDB 3.0은 WiredTiger를 포함했으며 따라서 해저드 포인터 역시 포함하게 되었습니다.

2011년, Samy Al Bahra는 Concurrency Kit 라이브러리 [Bah11b]에 해저드 포인터를 추가했습니다.

2014년, MAXIM Khizhinsky는 libcds [Khi14]에 해저드 포인터를 추가했습니다.

2015년, David Gwynne는 OpenBSD에 해저드 포인터 형태로 공유 레퍼런스 포인터를 추가했습니다 [Gwy15].

2017~2018년, Rust language의 `arc-swap` [Van18]와 `conc` [cut17] crate는 그들 스스로의 해저드 포인터 구현을 포함시켰습니다.

2018년, Maged Michael은 널리 사용되는 Facebook의 Folly 라이브러리 [Mic18]에 해저드 포인터를 추가했습니다.

9.6.3.2 Production Uses of Sequence Locking

2003년, 리눅스 커널은 x86의 `gettimeofday()` 시스템콜 구현에서 사용된 추가적 기법의 범용화된 형태로 시퀀스 락킹을 v2.5.60에 추가했습니다 [Cor03].

2011년, Samy Al Bahra는 Concurrency Kit 라이브러리에 시퀀스 락킹을 추가했습니다 [Bah11c].

2013년, Paolo Bonzini는 QEMU 애뮬레이터에 간단한 시퀀스 락을 추가했습니다 [Bon13].

2016년, Alexis Menard는 Chromium의 시퀀스 락 구현을 추상화 시켰습니다 [Men16].

2018년, `jemalloc()`에 간단한 시퀀스 락 구현이 추가되었습니다. 이 라이브러리는 또한 시퀀스 락킹을 담은 메커니즘으로 관리되는 특수 목적 `queue`를 갖습니다.

9.6.3.3 Production Uses of RCU

1980년대 언젠가, IBM의 VM/XA는 RCU와 비슷한 메커니즘은 passive serialization을 도입했습니다 [HOS89].

19983년, DYNIX/ptx는 RCU를 도입했습니다 [MS98a, SM95].

2022년, 리눅스 커널은 Dipankar Sarma의 RCU 구현을 도입했습니다 [Tor02].

2009년, userspace RCU 프로젝트가 시작되었습니다 [Des09b].

2010년, Knot DNS 프로젝트가 userspace RCU를 사용하기 시작했습니다 [Slo10]. 같은 해, OSv 커널이

RCU 구현을 더했으며 [Kiv13], 나중에는 RCU 로 보호되는 링크드 리스트와 [Kiv14b] 해쉬 테이블을 [Kiv14a] 추가했습니다.

2011년, Samy Al Bahra 는 Concurrency Kit 라이브러리에 epochs 를 (RCU 의 한 형태 [Fra04, FH07]) 추가했습니다 [Bah11a].

2012년, NetBSD 는 v6.0 에 이르러 앞서 언급한 passive serialization 을 사용하기 시작했습니다 [The12a]. Passive serialization 은 그 중에서도 NetBSD packet filter (NPF) 에 사용되었습니다 [Ras14].

2015년, Paolo Bonzini 는 userspace RCU 라이브러리의 fork 를 통해 QEMU emulator 에 RCU 지원을 추가했습니다 [BD13, Bon15].

2015년, Maxim Khizhinsky 는 libcds 에 RCU 를 추가했습니다 [Khi15].

2016년, Mindaugas Rasiukevicius 는 libqsbr 을 구현했는데, 이는 QSBR 과 epoch 기반 reclamation (EBR) [Ras16] 을 포함했는데, 둘 다 RCU 구현의 종류들입니다.

Sheth 등 [SWS16] 은 Go 의 가비지 컬렉터가 RCU 같은 기능을 제공하는 것의 가치를 선보였으며, Go 프로그래밍 언어는 이 기능을 제공할 수 있는 Value 타입을 제공합니다.²¹

Matt Kein 은 Envoy Proxy 에 사용된 RCU 같은 메커니즘을 설명했습니다 [Kle17].

2018년, Honnappa Nagarahalli 는 Data Plane Development Kit (DPDK) 에 RCU 라이브러리를 추가했습니다 [Nag18].

Stjepan Glavina 는 epoch 기반 RCU 구현을 Rust 언어의 동시성 지원 “crate” 집합의 crossbeam 에 병합했습니다.

마지막으로, 모든 가비지 컬렉션 기반 동시성 언어가 (Go 만 제외하고요!) RCU 구현의 업데이트 쪽을 추가적인 비용 없이 얻었습니다.

9.6.3.4 Summary of Production Uses

언젠가는 시퀀스 락킹, 해저드 포인터, 그리고 RCU 가 레퍼런스 카운터 만큼이나 널리 사용되는 날이 아마 올 겁니다. 그 날이 오기 전까지는, 이 메커니즘들의 현재 제품상 사용들은 이 메커니즘들 사이에서의 선택은 물론이고 그것들 각각의 가장 좋은 적용법은 어떤지 보일 겁니다.

다음 섹션은 이 챕터에서 이야기 된 많은 읽기 대 부분일 때를 위한 메커니즘에 문제를 제기하는 업데이트에 대해 이야기 합니다.

9.7 What About Updates?

The only thing constant in life is change.

François de la Rochefoucauld

이 챕터에서 이야기된 미뤄서 처리하기 기법들은 읽기가 대부분인 상황에 거의 직접적으로 적용이 가능합니다만, “하지만 업데이트는요?”라는 질문을 떠올리게 합니다. 어쨌건, 읽기 쓰레드의 성능과 확장성을 증가시키는 건 모두 좋지만, 쓰기 쓰레드에게도 훌륭한 성능과 확장성을 원하는 것은 자연스러운 일입니다.

우린 이미 쓰기 쓰레드를 위한 높은 성능과 확장성을 제공하는 상황을 하나 봤는데, Chapter 5 에서 알아본 카운팅 알고리즘들입니다. 이 알고리즘들은 부분적으로 분리된 데이터 구조들을 사용하여 업데이트가 지역적으로 일어날 수 있는 동안 더 비싼 읽기들은 전체 데이터 구조를 가로지르며 덧셈을 해야만 했습니다. Silas Boyd-Wickizer 는 이 방향을 OpLog 를 생성하게끔 범용화 시켰는데, 이를 그는 리눅스 커널 pathname 탐색, VM 역 매핑, 그리고 stat() 시스템콜에 적용했습니다 [BW14].

“Disruptor” 라 불리는 또 다른 방법이 많은 양의 입력 데이터 스트림을 처리하는 어플리케이션들을 위해 설계되었습니다. 이 방법은 single-producer-single-consumer FIFO queue 에 기반하여 동기화의 필요성을 최소화 시키기 위함입니다 [Sut13]. Java 어플리케이션에서 Disruptor 는 또한 가비지 컬렉터 사용을 최소화 시키는 이점이 있습니다.

그리고 물론, 가능한 곳에서는 완전히 조각난 또는 “shard” 된 시스템은 Chapter 6 에서 이야기 되었듯 훌륭한 성능과 확장성을 제공합니다.

다음 챕터는 여러 타입의 데이터 구조의 맥락에서 업데이트를 살펴봅니다.

²¹ <https://golang.org/pkg/sync/atomic/#Value>, 특히 “Example (ReadMostly)” 를 참고하세요.

Chapter 10

Data Structures

알고리즘에 대한 심각한 토론은 그것들의 데이터 구조의 시간 복잡도를 포함합니다 [CLRS01]. 하지만, 병렬 프로그램에서, 시간 복잡도는 동시성 효과를 포함하는데 이 효과는 Chapter 3에 보인 것처럼 압도적으로 클 수 있기 때문입니다. 달리 말하자면, 좋은 프로그래머의 데이터 구조 관계는 동시성에 연관된 그러한 측면을 포함해야 합니다.

Section 10.1은 이 챕터의 데이터 구조들의 모티베이션을 제공하는 어플리케이션을 소개합니다. Chapter 6은 파티셔닝이 확장성을 어떻게 개선하는지 보였으나, Section 10.2는 파티셔닝 할 수 있는 데이터 구조들을 이야기합니다. Chapter 9은 일부 행동을 미뤄서 처리하기가 어떻게 성능과 확장성을 모두 크게 개선할 수 있는지 보였는데, Section 10.3 가 이 주제를 이야기 합니다. Section 10.4는 파티셔닝 불가능한 데이터 구조를 알아보고, 그것을 성능과 확장성을 모두 개선할 수 있게끔 읽기가 대부분이고 파티셔닝 가능한 부분으로 쪼개는 것을 봅니다. 이 챕터는 모든 동시성 데이터 구조의 자세한 내용을 다루기는 어려우므로 Section 10.5가 중요한 것들 일부를 더 다릅니다. 최선의 성능과 확장성은 사후 마이크로 최적화보다는 설계에서 나오지만, 마이크로 최적화는 Section 10.6에서 이야기하는 것처럼 더도 아니고 덜도 아니고 절대적인 최고의 가능한 성능과 확장성을 위해서 필요합니다. 마지막으로, Section 10.7은 이 챕터의 요약을 제공합니다.

10.1 Motivating Application

The art of doing mathematics consists in finding that special case which contains all the germs of generality.

David Hilbert

우리는 성능을 평가하기 위해 Schrödinger's Zoo 어플리케이션을 사용하겠습니다 [McK13]. Schrödinger는

Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

Linus Torvalds

많은 수의 동물을 가지고 있는 동물원을 가지고 있으며, 그는 이 동물원의 각 동물이 하나의 데이터 항목으로 표현되는 인메모리 데이터베이스를 사용해 동물들을 추적하고자 할 겁니다. 각 동물은 키로 사용되는 고유의 이름을 가지고 있으며, 각 동물을 위해 관리되는 다양한 데이터를 가질 겁니다.

탄생, 포획, 그리고 구매가 데이터 추가로 이어지며 사망, 해방, 그리고 판매는 데이터 삭제로 이어집니다. Schrödinger's zoo는 쥐와 곤충을 포함해 큰 수의 수명이 짧은 동물을 가지고 있으므로 이 데이터베이스는 높은 업데이트율을 다룰 수 있어야만 합니다. Schrödinger의 동물들에 관심이 있는 사람들은 그것들을 질의할 수 있으며, Schrödinger는 그의 쥐들이 그들의 운명을 확인하고 있는 것이 아닌가 의심하게 될 정도로 고양이를 향한 수상하게 많은 질의율을 발견했습니다. 그게 어디서 시작되었든, Schrödinger의 어플리케이션은 단일 데이터 원소로의 높은 질의율을 처리해야만 합니다.

곧 보게 되겠지만, 이 간단한 어플리케이션은 동시성 데이터 구조에 대한 도전이 될 수 있습니다.

10.2 Partitionable Data Structures

Finding a way to live the simple life today is the most complicated task.

Henry A. Courtney, updated

오늘날 사용되는 데이터 구조의 수는 거대하여서, 그들을 다루는 여러 교재가 존재합니다. 이 섹션은 해쉬 테이블이라는 단일한 데이터 구조에 집중합니다. 이 집중된 전략은 어떻게 동시성이 데이터 구조와 상호작용하는지에 대한 무척 깊은 조사를 가능하게 하며, 실제로 많이 사용되는 데이터 구조에 집중할 수 있게 합니다. Section 10.2.1은 설계를 개략적으로 보고, Section 10.2.2은 그 구현을 선보입니다. 마지막으로, Section 10.2.3은 그로 인한 성능과 확장성을 이야기 합니다.

10.2.1 Hash-Table Design

Chapter 6 는 괜찮은 성능과 확장성을 얻기 위해 파티셔닝을 적용해야 하는 필요를 강조했으므로, 파티셔닝 가능성은 데이터 구조를 고를 때 첫번째 고려사항이 되어야만 합니다. 이 고려사항은 병렬성에서 널리 사용되는 해쉬 테이블에 의해 잘 만족됩니다. 해쉬 테이블은 컨셉상으로 간단하여, 해쉬 버킷의 배열로 구성됩니다. 하나의 해쉬 함수가 주어진 원소의 키를 이 원소가 저장되어 있는 해쉬 버킷으로 매핑합니다. 따라서 각 해쉬 버킷은 해쉬 체인이라 불리는 원소들의 링크드 리스트를 갖습니다. 올바르게 구성되었을 경우, 이 해쉬 체인은 상당히 짧아서, 해쉬 테이블이 그 원소들에 굉장히 효율적으로 접근할 수 있게 합니다.

Quick Quiz 10.1: 하지만 체인으로 연결된 해쉬 테이블은 많은 종류 중 하나입니다. 왜 체인으로 연결된 해쉬 테이블에 집중하죠?

■
추가적으로, 각 버킷은 각자의 락을 가지고 있어서 이 해쉬 테이블에서의 다른 버킷의 원소들은 완전히 독립적으로 더해지고 삭제되고 탐색될 수 있습니다. 따라서 큰 수의 버킷을 (그리고 따라서 락들을) 가지며 각 버킷은 작은 수의 원소를 갖는 커다란 해쉬 테이블은 훌륭한 확장성을 제공합니다.

10.2.2 Hash-Table Implementation

Listing 10.1 (`hash_bkt.c`)은 간단한 체이닝과 해쉬 버킷별 락킹을 사용하는 고정 크기 해쉬 테이블에 사용되는 데이터 구조 집합을 보이며, Figure 10.1는 그것들이 어떻게 같이 동작하는지 보입니다. `hashtab` 구조체는 (Listing 10.1의 라인 11–15) 네개의 `ht_bucket` 구조체를 (Listing 10.1의 라인 6–9) 가지며, `->ht_nbuckets` 필드는 버킷의 갯수를 제어하고 `->ht_cmp` 필드는 키 비교 함수로의 포인터를 갖습니다. 그런 각 버킷은 리스트 헤더 `->htb_head` 와 하나의 락 `->htb_lock`을 갖습니다. 그것들의 `->hte_next` 필드를 통한 이 리스트 헤더는 `ht_elem` 구조체를 (Listing 10.1의 라인 1–4) 연결하며, 각 `ht_elem` 구조체는 또한 연관된 원소의 해쉬 값을 `->hte_hash` 필드에 캐쉬해둡니다. `ht_elem` 구조체는 복잡한 키를 가지고 있을 수도 있는 더 커다란 구조체에 포함됩니다.

Figure 10.1는 두개의 원소를 갖는 버킷 0과 하나의 원소를 갖는 버킷 2를 보입니다.

Listing 10.2은 매핑과 락킹 함수들을 보입니다. 라인 1과 2는 `HASH2BKT()` 매크로를 보이는데, 이 매크로는 해쉬 값을 연관된 `ht_bucket` 구조체로 매핑합니다. 이 매크로는 간단한 모듈로 연산을 사용합니다: 더 강력한 해싱이 필요하다면, 호출자는 키에서 해쉬 값으로의 매핑 시에 그것을 구현해야 합니다. 남아있는 두 함수

Listing 10.1: Hash-Table Data Structures

```

1 struct ht_elem {
2     struct cds_list_head hte_next;
3     unsigned long hte_hash;
4 };
5
6 struct ht_bucket {
7     struct cds_list_head htb_head;
8     spinlock_t htb_lock;
9 };
10
11 struct hashtab {
12     unsigned long ht_nbuckets;
13     int (*ht_cmp)(struct ht_elem *htep, void *key);
14     struct ht_bucket ht_bkt[0];
15 };

```

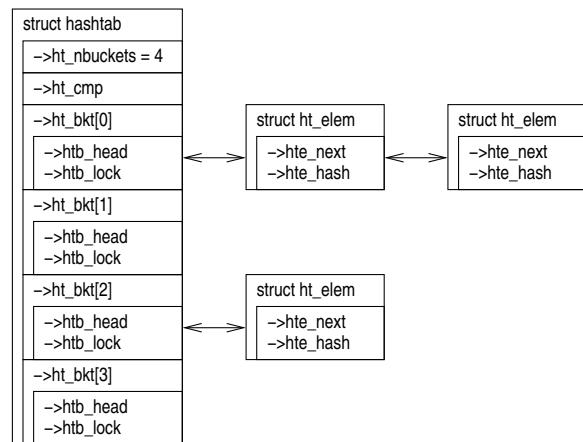


Figure 10.1: Hash-Table Data-Structure Diagram

는 명시된 해쉬 값에 연관된 `->htb_lock`을 획득하고 해제합니다.

Listing 10.3은 `hashtab_lookup()`을 보이는데, 이 함수는 명시된 해쉬와 키를 가지고 그에 연관된 원소가 존재한다면 그 원소로의 포인터를, 아니라면 NULL을 리턴합니다. 이 함수는 해쉬 값과 키로의 포인터를 둘다 받는데 이는 이 함수의 사용자들이 임의의 키와 임의의 해쉬 함수를 사용할 수 있게 하기 위함입니다. 라인 8은 이 해쉬 값에서 연관된 해쉬 버킷으로의 포인터로의 매핑을 합니다. 라인 9–14의 루프의 각 패스는 이 버킷의 해쉬 체인의 원소를 하나씩 검사합니다. 라인 10은 이 해쉬 값이 맞는지 검사하고, 그렇지 않다면 라인 11에서 다음 원소로 넘어갑니다. 라인 12는 실제 키가 들어맞는지 확인하고, 그렇다면 라인 13에서 이 맞는 원소로의 포인터를 리턴합니다. 어떤 원소도 들어맞지 않으면, 라인 15 이 NULL을 리턴합니다.

Quick Quiz 10.2: 하지만 Listing 10.3의 라인 10–13에서의 두번의 비교는 키가 `unsigned long`에 들어갈 때에는 비효율적이지 않나요?

■

Listing 10.2: Hash-Table Mapping and Locking

```

1 #define HASH2BKT(htp, h) \
2   (&(htp)->ht_bkt[h % (htp)->ht_nbuckets])
3
4 static void hashtab_lock(struct hashtab *htp,
5                           unsigned long hash)
6 {
7   spin_lock(&HASH2BKT(htp, hash)->htb_lock);
8 }
9
10 static void hashtab_unlock(struct hashtab *htp,
11                           unsigned long hash)
12 {
13   spin_unlock(&HASH2BKT(htp, hash)->htb_lock);
14 }

```

Listing 10.3: Hash-Table Lookup

```

1 struct ht_elem *
2 hashtab_lookup(struct hashtab *htp, unsigned long hash,
3                 void *key)
4 {
5   struct ht_bucket *htb;
6   struct ht_elem *htep;
7
8   htb = HASH2BKT(htp, hash);
9   cds_list_for_each_entry(htep, &htb->htb_head, hte_next) {
10     if (htep->hte_hash != hash)
11       continue;
12     if (htp->ht_cmp(htep, key))
13       return htep;
14   }
15   return NULL;
16 }

```

Listing 10.4: Hash-Table Modification

```

1 void hashtab_add(struct hashtab *htp, unsigned long hash,
2                   struct ht_elem *htep)
3 {
4   htep->hte_hash = hash;
5   cds_list_add(&htep->hte_next,
6               &HASH2BKT(htp, hash)->htb_head);
7 }
8
9 void hashtab_del(struct ht_elem *htep)
10 {
11   cds_list_del_init(&htep->hte_next);
12 }

```

Listing 10.4 은 해쉬 테이블로부터 원소를 각각 더하고 빼는 `hashtab_add()` 와 `hashtab_del()` 함수들을 보입니다.

`hashtab_add()` 함수는 단순히 원소의 해쉬 값을 라인 4에서 설정하고, 라인 5와 6에서 연관된 버킷에 이를 더합니다. `hashtab_del()` 함수는 단순히 명시된 원소를 그것이 어디였든 연결되어 있던 곳에서 제거하는데, 해쉬 체인 리스트의 양방향 링크드 리스트 덕입니다. 이 두 함수를 호출하기 전에, 호출자는 어떤 다른 쓰레드도 이 같은 버킷을 접근하거나 수정하고 있지 않다는 것을 분명히 해야 하는데, 예를 들면 `hashtab_lock()` 을 호출할 수 있습니다.

Listing 10.5 은 `hashtab_alloc()` 과 `hashtab_free()` 를 보이는데, 이것들은 각각 해쉬 테이블 메모리

Listing 10.5: Hash-Table Allocation and Free

```

1 struct hashtab *
2 hashtab_alloc(unsigned long nbuckets,
3               int (*cmp)(struct ht_elem *htep, void *key))
4 {
5   struct hashtab *htp;
6   int i;
7
8   htp = malloc(sizeof(*htp) +
9               nbuckets * sizeof(struct ht_bucket));
10  if (htp == NULL)
11    return NULL;
12  htp->ht_nbuckets = nbuckets;
13  htp->ht_cmp = cmp;
14  for (i = 0; i < nbuckets; i++) {
15    CDS_INIT_LIST_HEAD(&htp->ht_bkt[i].htb_head);
16    spin_lock_init(&htp->ht_bkt[i].htb_lock);
17  }
18  return htp;
19 }
20
21 void hashtab_free(struct hashtab *htp)
22 {
23   free(htp);
24 }

```

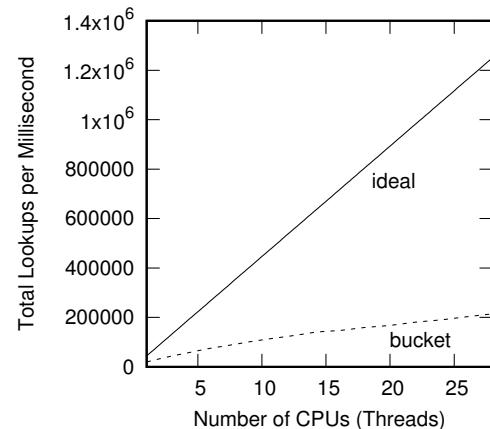


Figure 10.2: Read-Only Hash-Table Performance For Schrödinger's Zoo

할당과 해제를 합니다. 할당은 라인 8-9에서 메모리 할당과 함께 시작됩니다. 라인 10 이 메모리가 부족함을 확인하면, 라인 11 이 호출자에게 NULL 을 리턴합니다. 그렇지 않다면, 라인 12 와 13 가 버킷의 갯수와 키 비교 함수로의 포인터를 초기화 하며, 라인 14-17 의 루프는 버킷 자체를 초기화 하는데, 라인 15 의 체인 리스트 헤더와 라인 16 의 락 초기화를 포함합니다. 마지막으로, 라인 18 은 새로 할당된 해쉬 테이블로의 포인터를 리턴합니다. 라인 21-24 에서의 `hashtab_free()` 함수는 직관적입니다.

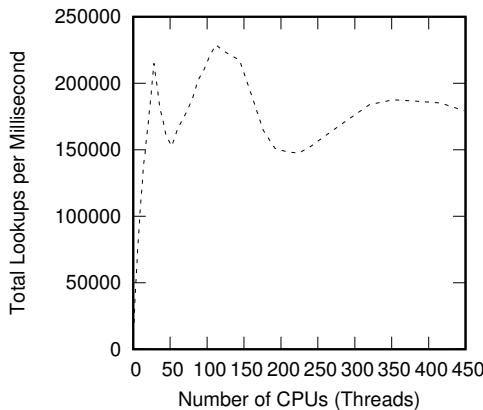


Figure 10.3: Read-Only Hash-Table Performance For Schrödinger's Zoo, 448 CPUs

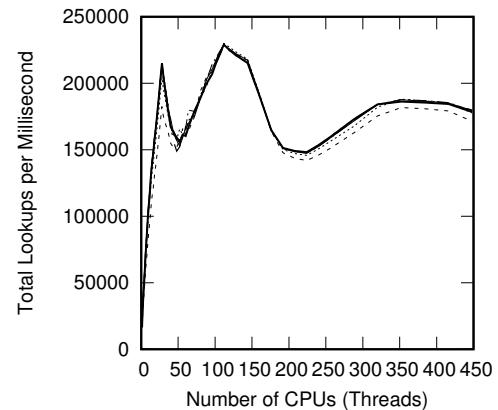


Figure 10.4: Read-Only Hash-Table Performance For Schrödinger's Zoo, Varying Buckets

10.2.3 Hash-Table Performance

버킷 락킹 기반 해쉬 테이블을 262,144 버킷을 가지고 사용한 2.1 GHz 28 코어 싱글 소켓 Intel Xeon 시스템에서의 성능 결과는 Figure 10.2에 보인 것과 같습니다. 이 성능은 거의 선형으로 확장됩니다만, 이상적인 성능 수준에는 다다르지 못하는데, 겨우 28 CPU에서 조차 그렇습니다. 이는 부분적으로는 락 획득과 해제가 단일 CPU에서는 캐쉬 미스를 내지 않지만 두 개 이상의 CPU에서는 그려기 때문입니다.

그리고 더 많은 CPU에서는 상황이 더 나빠질 뿐입니다. Figure 10.3에 보인 것과 같습니다. 우린 이상적 성능을 보일 필요는 없습니다: 29 CPU와 그 뒤에서의 성능은 지옥보다도 나쁠 것이 너무 분명합니다. 이는 적은 수의 CPU에서의 성능만 사용한 외삽의 위험성을 분명히 강조합니다.

물론, 이 성능 추락의 한가지 가능한 이유는 더 많은 해쉬 버킷이 필요해서일 수도 있습니다. 우린 해쉬 버킷의 수를 늘려서 이를 테스트 할 수 있습니다.

Quick Quiz 10.3: 단순히 해쉬 버킷의 수를 늘리는 대신, 존재하는 해쉬 버킷을 캐쉬 열라인시키는게 낫지 않을까요?

■

하지만, Figure 10.4에서 볼 수 있듯이, 여러 버킷을 연결하는 것은 거의 효과가 없습니다: 확장성은 여전히 끔찍합니다. 특히, 29 CPU를 넘어서는 순간 상당한 하락을 보입니다. 분명 무언가가 있습니다.

문제는 이 멀티 소켓 시스템에서, CPU 0-27과 225-251은 Figure 10.5에 보인 것처럼 같은 소켓에 매핑되어 있다는 것입니다. 테스트 수행은 처음 28 CPU에 국한되어 있으므로 성능이 상당히 잘 나오지만, 소켓 0의 CPU 0-27과 소켓 1의 CPU 28이 사용되는 테스트에서는 데이터를 소켓 경계 안팎으로 보내는 오버헤드를

일으킵니다. 이는 Section 3.2.1에서 이야기 했듯 상당히 성능을 하락시킬 수 있습니다. 짧게 말해서, 커다란 멀티 소켓 시스템은 완전한 파티셔닝에 더해 참조의 좋은 지역성을 필요로 합니다. 이 챕터의 나머지 부분은 해쉬 테이블 자체 내에서 좋은 참조 지역성을 제공하는 방법들을 이야기 할 것입니다만, 그 사이 좋은 참조 지역성을 제공하는 한가지 다른 방법은 해쉬 테이블에 많은 데이터 원소를 넣는 것임을 알아두시기 바랍니다. 예를 들어, Schrödinger는 그의 동물들의 사진이나 심지어 비디오를 이 해쉬 테이블의 각 원소에 넣음으로써 훌륭한 캐쉬 지역성을 얻을 수도 있을 겁니다. 하지만 해쉬 테이블이 작은 데이터 원소를 가져야 하는 경우를 위해선, 계속 읽으세요!

Quick Quiz 10.4: 소켓을 가로지르는 경우의 Schrödinger의 동물원의 나쁜 확장성을 생각하면, 이 어플리케이션의 여러 복사본을 각 복사본은 동물들의 부분집합만을 가지고 각 복사본을 단일 소켓으로 가둬 수행시키는 건 어떤가요?

■

지금까지 이야기된 Schrödinger의 동물원의 한가지 핵심 요소는 읽기 전용이라는 것입니다. 이는 락 획득으로 발생한 캐쉬 미스로 인한 성능 하락을 더욱 고통스럽게 만듭니다. 우린 이 해쉬 테이블 자체에 대한 업데이트를 하지 않음에도 불구하고 여전히 메모리에 쓰기를 위한 값을 지불하고 있습니다. 물론, 이 해쉬 테이블이 절대 업데이트 되지 않는다면, 상호 배제를 완전히 제거할 수 있습니다. 이 방법은 상당히 간단하며 독자 여러분의 연습문제로 남겨둡니다. 하지만 가끔씩의 업데이트만 있다 하더라도, 쓰기를 회피하는 것은 캐쉬 미스를 막으며, 읽기가 대부분인 데이터가 모든 캐쉬에 복사되어 되어 결국 참조 지역성을 높이게 됩니다.

Socket	Hyperthread	
	0	1
0	0-27	224-251
1	28-55	252-279
2	56-83	280-307
3	84-111	308-335
4	112-139	336-363
5	140-167	364-391
6	168-195	392-419
7	196-223	420-447

Figure 10.5: NUMA Topology of System Under Test

따라서 다음 섹션은 업데이트가 드물지만 언제든 일어날 수 있고, 읽기가 대부분인 상황에서 할 수 있는 최적화 기법들을 알아봅니다.

10.3 Read-Mostly Data Structures

Adapt the remedy to the disease.

Chinese proverb

파티셔닝 된 데이터 구조가 훌륭한 확장성을 제공할 수 있지만, NUMA 효과는 성능과 확장성 모두에 상당한 저하를 초래할 수 있습니다. 또한, read-side 동기화는 읽기가 대부분인 상황에서 성능을 떨어뜨릴 수 있습니다. 그러나, Section 9.5 에서 소개된 RCU 를 사용해 성능과 확장성을 모두 얻을 수 있습니다. 비슷한 결과가 해저드 포인터를 (hazptr.c) [Mic04] 이용해서도 얻어질 수 있는데, 이 섹션에 보여진 성능 결과가 포함될 겁니다 [McK13].

10.3.1 RCU-Protected Hash Table Implementation

버킷별 락킹과 함께 RCU 로 보호되는 해쉬 테이블에서, 업데이트 쓰레드는 Section 10.2 에서 보인 것과 같이 락킹을 사용하지만 읽기 쓰레드는 RCU 를 사용합니다. 데이터 구조는 Listing 10.1 에 보인 것 그대로이며, HASH2BKT(), hashtab_lock(), 그리고 hashtab_unlock() 함수는 Listing 10.2 에 보인 그대로입니다. 그러나, 읽기 쓰레드는 Listing 10.6 에 보인, hashtab_lock_lookup() 과 hashtab_unlock_lookup() 에 포함된 더 가벼운 동시성 제어를 사용합니다.

Listing 10.6: RCU-Protected Hash-Table Read-Side Concurrency Control

```

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 }

```

Listing 10.7: RCU-Protected Hash-Table Lookup

```

1 struct ht_elem *hashtab_lookup(struct hashtab *htp,
2                                 unsigned long hash,
3                                 void *key)
4 {
5     struct ht_bucket *htb;
6     struct ht_elem *htep;
7
8     htb = HASH2BKT(htp, hash);
9     cds_list_for_each_entry_rcu(htep,
10                                &htb->htb_head,
11                                hte_next) {
12         if (htep->hte_hash != hash)
13             continue;
14         if (htp->ht_cmp(htep, key))
15             return htep;
16     }
17     return NULL;
18 }

```

Listing 10.7 은 RCU 로 보호되는 버킷별 락을 사용하는 해쉬 테이블을 위한 hashtab_lookup() 을 보입니다. 이는 cds_list_for_each_entry() 가 cds_list_for_each_entry_rcu() 로 대체되었다는 것을 제외하면 Listing 10.3 의 것과 동일합니다. 이 두 함수들 모두 htb->htb_head 로 참조되는 해쉬 체인을 순회합니다만 cds_list_for_each_entry_rcu() 는 동시의 삽입 시에 메모리 순서를 올바르게 강제합니다. 이게 이 두 해쉬 테이블 구현 사이의 중요한 차이점입니다: 순수한 버킷별 락킹 구현과 달리, RCU 로 보호되는 구현은 탐색이 삽입과 삭제와 동시에 이루어질 수 있게 하며, cds_list_for_each_entry_rcu() 와 같은 RCU 를 인지하는 기능들은 이 추가된 동시성을 올바르게 처리할 필요를 갖습니다. 또한 hashtab_lookup() 의 호출자는 RCU read-side 크리티컬 섹션 내에 있어야만 하는데, 예를 들면 호출자는 hashtab_lookup() 호출 전에 hashtab_lock_lookup() 을 (그리고 물론 얼마 후에는 hashtab_unlock_lookup() 을) 호출해야만 함을 알아두시기 바랍니다.

Quick Quiz 10.5: 하지만 해쉬 테이블 내의 원소들이 탐색과 동시에 제거될 수 있다는 것은, 탐색이 찾아진 직후 제거된 데이터 원소로의 참조를 리턴할 수 있다는 의미 아닌가요?

■

Listing 10.8: RCU-Protected Hash-Table Modification

```

1 void hashtab_add(struct hashtab *htp,
2                   unsigned long hash,
3                   struct ht_elem *htep)
4 {
5     htep->hte_hash = hash;
6     cds_list_add_rcu(&htep->hte_next,
7                       &HASH2BKT(htp, hash)->htb_head);
8 }
9
10 void hashtab_del(struct ht_elem *htep)
11 {
12     cds_list_del_rcu(&htep->hte_next);
13 }

```

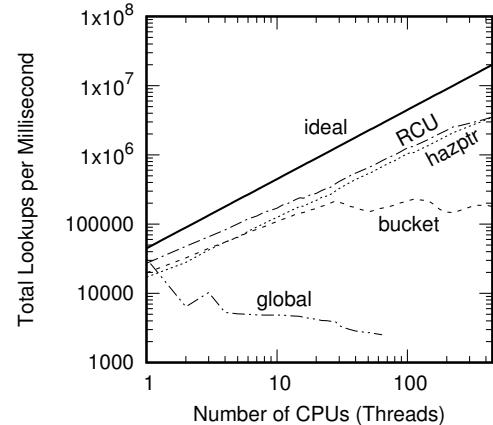
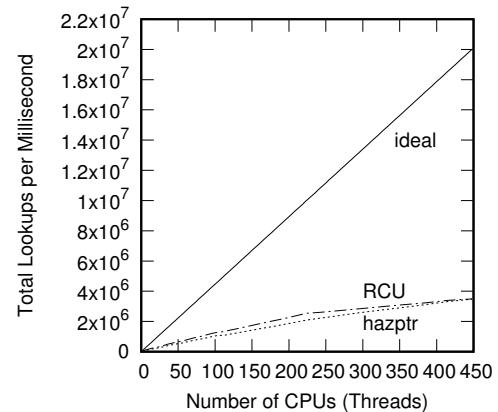
Listing 10.8 은 `hashtab_add()` 와 `hashtab_del()` 을 보이는데, 둘 다 Listing 10.4 에 보인 RCU 기반의 아닌 해쉬 테이블의 것들과 상당히 비슷합니다. `hashtab_add()` 함수는 어떤 원소를 탐색되고 있는 사이 이 해쉬 테이블에 추가하는 경우 올바른 순서를 보장하기 위해 `cds_list_add()` 대신 `cds_list_add_rcu()` 를 사용합니다. `hashtab_del()` 함수는 원소가 삭제되기 전에 탐색되는 경우를 허용하기 위해 `cds_list_del_init()` 대신 `cds_list_del_rcu()` 를 사용합니다. `cds_list_del_init()` 와 달리, `cds_list_del_rcu()` 는 앞으로의 포인터를 원래대로 두어서 `hashtab_lookup()` 이 새로이 제거된 원소의 다음 원소를 따라갈 수 있게 합니다.

물론, `hashtab_del()` 을 호출한 다음에는, 해당 원소의 메모리를 해제하기 전에 하나의 RCU grace period 를 기다려야 하며 (예: `synchronize_rcu()` 호출을 통해), 그렇지 않으면 이 새로이 삭제된 원소의 메모리를 재사용하게 되어버립니다.

10.3.2 RCU-Protected Hash Table Performance

Figure 10.6 는 RCU 로 보호되는 해쉬 테이블과 해저드 포인터로 보호되는 해쉬 테이블의 읽기만 할 때의 성능을 앞의 섹션에서의 벅킷별 락킹 구현과 비교해 보입니다. 볼 수 있듯, RCU 와 해저드 포인터 둘 다 읽기만 할 때의 복사가 NUMA 효과를 방지하므로 벅킷별 락킹에 비해 성능도 확장성도 좋습니다. 쓰레드의 수가 늘어날수록 차이가 커집니다. 전역적 락킹을 하는 구현에서의 결과 역시 보여져 있으며, 예상한대로 그 결과는 벅킷별 락킹 구현에 비해서 조차도 나쁩니다. RCU 는 해저드 포인터보다 약간 더 낫습니다.

Figure 10.7 는 같은 데이터를 선형 스케일로 보입니다. 여기선 전역 락킹에서의 선을 x 축에 거의 붙여버리지만, RCU 와 해저드 포인터의 이상적이지는 못한 성능을 더 잘 분간할 수 있게 합니다. 둘 다 224 CPU 에서 약간 하락을 보이며, 이는 하드웨어 멀티쓰레딩 때문입니다. 32 개 이하 CPU 에서, 각 쓰레드는 자신을 위한 코어가

**Figure 10.6:** Read-Only RCU-Protected Hash-Table Performance For Schrödinger's Zoo**Figure 10.7:** Read-Only RCU-Protected Hash-Table Performance For Schrödinger's Zoo, Linear Scale

있습니다. 이 시스템에서, RCU 는 해저드 포인터보다 나은 결과를 보이는데 해저드 포인터의 읽기 쪽 메모리 배리어가 이 코어 내에서의 능비되는 시간을 초래하기 때문입니다. 요약하자면, RCU 는 싱글 하드웨어 쓰레드를 해저드 보이터보다 더 잘 사용할 수 있습니다.

이 상황이 224 CPU 에선 바뀝니다. RCU 는 각 코어의 절반 이상의 자원을 싱글 하드웨어 쓰레드에서 사용하므로, RCU 는 각 코어의 두 번째 하드웨어 쓰레드로부터는 상대적으로 적은 이득을 얻습니다. 해저드 포인터의 선 역시 224 CPU 에서 하락하지만 덜 극적인데, 두 번째 하드웨어 쓰레드가 첫 번째 하드웨어 쓰레드는 메모리 배리어 응답시간으로 인해 멈춘 시간을 사용할 수 있기 때문입니다. 뒤의 섹션들에서 보게 되겠지만, 이 두 번째 하드웨어 쓰레드의 이득은 워크로드에 종속적입니다.

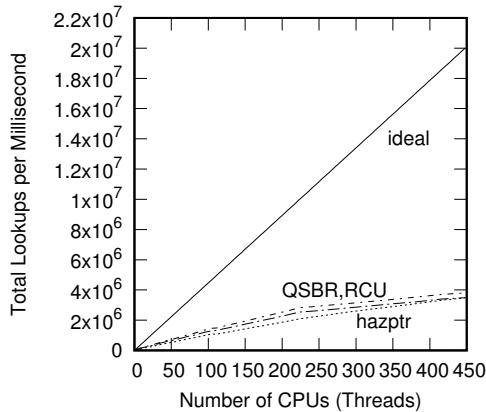


Figure 10.8: Read-Only RCU-Protected Hash-Table Performance For Schrödinger's Zoo including QSBR, Linear Scale

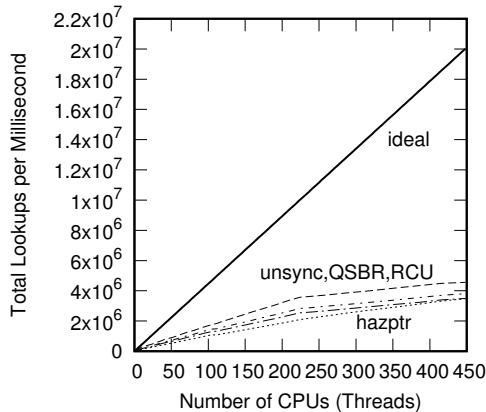


Figure 10.9: Read-Only RCU-Protected Hash-Table Performance For Schrödinger's Zoo including QSBR and Unsynchronized, Linear Scale

하지만 RCU의 성능은 왜 이상적인 경우보다 다섯 배나 적을까요? 한가지 가능성은 `rcu_read_lock()`과 `rcu_read_unlock()`으로 조절되는 쓰레드별 카운터가 성능 하락을 일으킨다는 것입니다. 그래서 Figure 10.8는 RCU의 QSBR 변종 결과를 더하는데, 여기선 read-side 기능들이 아무일도 하지 않습니다. 그리고 QSBR은 RCU 보다 조금 낫긴 하지만, 여전히 이상적인 것에 비해선 다섯배나 적은 성능입니다.

Figure 10.9는 이게 동기화 필요 없이 읽기만 하는 벤치마크이므로 가능한 전혀 동기화 하지 않는 경우의 결과를 더합니다. 동기화가 전혀 없는 경우에 조차 성능은 이상적인 경우보다 훨씬 낫습니다.

문제는 이 시스템이 28개 코어를 갖는 소켓을 가지고 있는데, 각 소켓은 page 24의 Figure 3.2에 보인

것처럼 작은 캐시 크기를 갖는다는 것입니다. 각 해쉬 버킷은 (`struct ht_bucket`) 56 바이트를 사용하며 각 원소는 (`struct zoo_he`) RCU와 QSBR의 경우 72 바이트를 사용합니다. Figure 10.9를 만드는데 사용된 벤치마크는 262,1444 버킷과 최대 262,144 원소를 사용하여, 총 33,554,448 바이트를 사용하는데, 1,048,576 바이트인 L2 캐시의 서른배 이상일 뿐 아니라 불편하게도 40,370,176 바이트의 L3 캐시 크기에도 근접하는데, 특히 이 캐시가 11 way 만을 가진다는 점을 생각하면 더 불편합니다. 이는 L2 캐시가 넘치는 것은 항상 있을 것이고 L3 캐시가 넘치는 것 또한 드문 일이 아닐 것이어서, 그 결과 발생하는 캐시 미스가 성능을 하락시킬 겁니다. 이 경우, 병목은 CPU가 아니라 하드웨어 메모리 시스템에 있습니다.

이 메모리 시스템 병목의 추가적인 증거가 동기화되지 않은 코드를 검사함으로써 찾아질 수도 있습니다. 이 코드는 락이 필요 없으므로 각 해쉬 버킷은 RCU와 QSBR의 경우의 56 바이트에 비해 적은 16 바이트만을 사용합니다. 비슷하게, 각 해쉬 테이블 원소는 RCU와 QSBR의 72 바이트에 비해 적은 56 바이트만을 사용합니다. 따라서 단일 CPU의 동기화되지 않은 수행이 QSBR이나 RCU 양쪽에 비해 반쯤 빠른 것은 놀라운 일이 아닙니다.

Quick Quiz 10.6: Page 178의 Figure 10.4가 해쉬 테이블 크기를 변화시키는게 거의 아무 효과도 내지 못하는 걸 보이는데 해쉬 테이블 크기가 문제라고 여기서 확신할 수 있나요? 문제는 그게 아니라 false sharing 일수도 있지 않을까요?

■
메모리 사용량이 더 줄어들면 어떻게 될까요? Page 482의 Figure 9.22는 pre-BSD 라우팅 테이블로 표현되는 훨씬 작은 데이터 구조에서는 RCU가 정말 이상에 근접한 성능을 얻음을 보입니다.

Quick Quiz 10.7: 이 커다란 시스템에서는 메모리 시스템이 상당한 병목입니다. 유용한 뭔가를 하기 충분한 메모리 대역폭을 주지 않고서 시스템에 448 CPU를 장착해서는 왜 고생할까요?

■
앞에서 언급했듯, Schrödinger는 그의 고양이의 인기에 놀랐습니다만 [Sch35], 이 인기를 그의 설계에 적용할 필요를 인식했습니다. Figure 10.10는 64-CPU 수행에서의 결과를 변화하는 CPU 수에 따라 그 고양이 탐색만을 하는 워크로드에 대해 보입니다. RCU와 해저드 포인터 모두가 도전에 잘 응합니다만, 버킷 락킹은 음의 확장을 보이며, 결국은 글로벌 락킹만큼이나 나쁜 성능을 보입니다. 모든 CPU가 그 고양이 탐색만을 한다면 그 고양이의 버킷에 연관된 락이 글로벌 락의 의도와 목적에 부합하게 되므로 놀라운 일이 아닙니다.

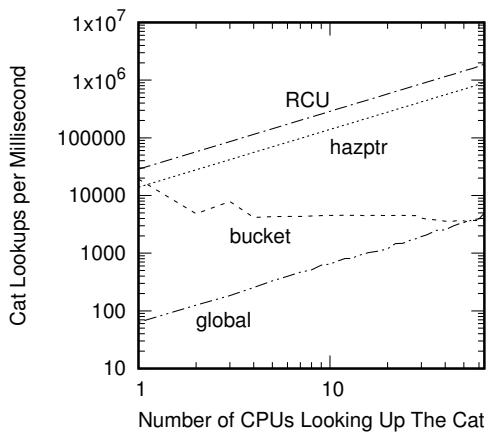


Figure 10.10: Read-Side Cat-Only RCU-Protected Hash-Table Performance For Schrödinger's Zoo at 64 CPUs

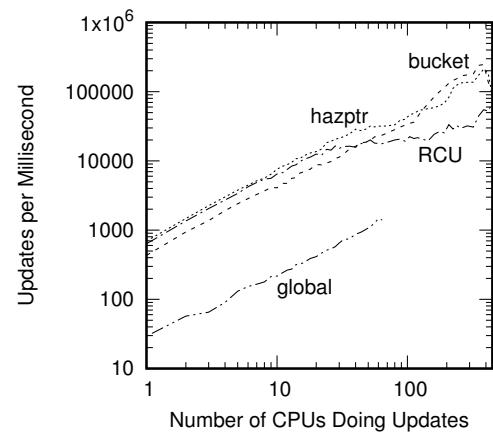


Figure 10.12: Update-Side RCU-Protected Hash-Table Performance For Schrödinger's Zoo

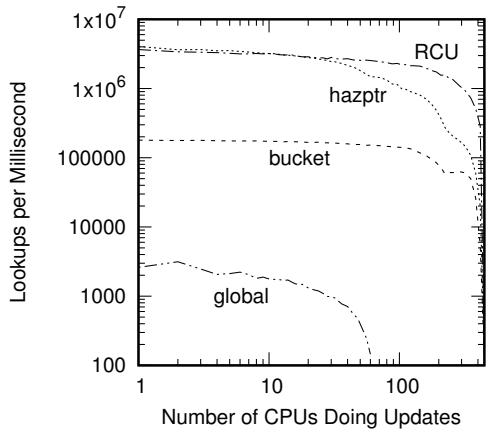


Figure 10.11: Read-Side RCU-Protected Hash-Table Performance For Schrödinger's Zoo in the Presence of Updates

이 고양이만 찾기 벤치마크는 완전히 파티셔닝 된 샤팅 접근법의 잠재적 문제 한가지를 보입니다. 이 고양이의 파티션에 연관된 CPU 만이 그 고양이에 접근할 수 있어서 고양이만 찾기 처리량을 제한합니다. 물론, 무척 많은 어플리케이션이 좋은 부하 분산 속성을 가지며 그런 어플리케이션에서 샤팅은 무척 잘 동작합니다. 하지만, 샤팅은 “핫스팟”을 아주 잘 처리하지는 않으며, Schrödinger 의 고양이는 그런 핫스팟을 보이는 한가지 예일 뿐입니다.

우리가 데이터를 읽기만 할거라면, 애초에 동시성 제어가 필요 없을 겁니다. 따라서 Figure 10.11 는 읽기 쓰레들에서의 업데이트의 효과를 보입니다. 이 그림의 가장 왼쪽에서는 하나의 CPU 를 제외한 모든 CPU 가 탐색을 하며, 오른쪽에서는 448 CPU 가 모두 업데이

트를 합니다. 이 모든 구현에서, 밀리세컨드당 루프업 횟수는 업데이트를 하는 CPU 의 수가 늘어날수록 감소하는데, 448 CPU 가 모두 업데이트를 할때는 물론 밀리세컨드당 0 탐색에 달습니다. 해저드 포인터와 RCU 모두 버킷별 락킹에 비해 잘 동작하는데 그것들의 읽기 쓰레드는 업데이트 쪽 락 경쟁을 증가시키지 않기 때문입니다. RCU 는 업데이트 쓰레드의 수가 증가함에 따라 해저드 포인터에 비해 더 잘 동작하는데 해저드 포인터의 읽기 쪽 메모리 배리어 때문으로, 특히 업데이트가 존재할 때, 그리고 수행이 하나 이상의 소켓을 관여시킬 때 메모리 오버헤드는 훨씬 큰 오버헤드를 일으킵니다. 따라서 현대의 하드웨어는 메모리 배리어 수행을 상당히 최적화 해서 읽기만 있는 경우에는 메모리 배리어 오버헤드를 줄였을 것으로 보입니다.

Figure 10.11 는 증가하는 업데이트 비율이 탐색에 끼치는 영향을 보이며, Figure 10.12 는 증가하는 업데이트 비율이 업데이트 스스로에게 끼치는 영향을 보입니다. 다시 말하지만, 이 그림의 왼쪽은 하나의 CPU 를 제외한 모두가 탐색을 하며 오른쪽은 모든 448개 CPU 가 업데이트를 하고 있습니다. 해저드 포인터와 RCU 는 상당한 이점을 가지고 시작하는데, 버킷 락킹과 달리 읽기 쓰레드들은 업데이트 쓰레드를 배제시키지 않기 때문입니다. 그러나, 업데이트를 하는 CPU 의 수가 늘어날수록 업데이트 쪽 오버헤드는 그 존재를 드러내기 시작하는데, RCU 에서 먼저 그려하고 이어서 해저드 포인터에서 그려합니다. 물론, 이 세개의 구현은 모두 글로벌 락킹보다는 낫습니다.

탐색 성능의 차이점이 업데이트 비율에 의해 영향을 받았을 가능성이 상당히 있습니다. 이를 확인하는 한가지 방법은 인위적으로 버킷별 락킹과 해저드 포인터의 업데이트 비율을 RCU 의 그것과 맞게 조정하는 것입니다. 그렇게 하는 것이 버킷별 락킹의 탐색 성능을 눈에

띄게 개선하지는 않으며, 해저드 포인터와 RCU 사이의 차이를 메우지도 않습니다. 하지만, 해저드 포인터의 읽기 쪽 메모리 배리어를 제거하는 것은 (따라서 안전치 않은 구현이 되는 것은) 해저드 포인터와 RCU 사이의 차이를 거의 메웁니다. 이 안전하지 않은 해저드 포인터 구현은 벤치마킹 목적에는 충분히 안전하겠지만 제품에서의 사용에는 결코 추천되지 않습니다.

Quick Quiz 10.8: 28 CPU에서의 결과를 448 CPU로 외삽하는 것의 위험성은 Section 10.2.3에서 분명히 보여졌습니다. 하지만 왜 448 CPU에서의 결과로 그 이상을 외삽하는 것은 안전한가요?

■

10.3.3 RCU-Protected Hash Table Discussion

RCU와 해저드 포인터 구현의 한가지 결론은 한쌍의 동시적인 읽기 쓰레드는 이 고양이의 상태에 대해 다른 의견을 가질 수도 있다는 겁니다. 예를 들어, 한 읽기 쓰레드는 이 고양이의 데이터 구조로의 포인터를 그것이 제거되기 전에 가져왔고 다른 읽기 쓰레드는 그 직후에 같은 포인터를 가져왔을 수 있습니다. 첫번째 읽기 쓰레드는 그 고양이가 살아있다고, 두번째 쓰레드는 그 고양이가 죽었다고 믿을 겁니다.

이 상황은 Schrödinger의 고양이에 완전히 들어맞습니다만, 평범한 양자가 아닌 고양이에게도 상당히 합리적인 일입니다. 어쨌건, 어떤 동물이 정확히 언제 태어나고 죽었는지를 규정하기란 불가능합니다.

이를 위해 우리가 어떤 고양이의 죽음을 맥박으로 규정한다고 해봅시다. 이는 우리가 죽음을 판단하기 전까지 마지막 맥박으로부터 정확히 얼마나 기다려야 하는지 하는 질문을 던집니다. 1 밀리세컨드만 기다리는건 분명 우스운 일일텐데 그럼 건강하게 살아있는 고양이도 1초에 한번 이상 죽었다고—그리고 나서는 부활했다고—규정되어야 할 것이기 때문입니다 마찬가지로 한다을 다 기다리는 것도 우스운 일일텐데, 그때엔 이 불쌍한 고양이의 죽음을 냄새로도 분명해졌을 것이기 때문입니다.

어떤 동물의 심장은 몇초간 멈췄다가 다시 움직일 수 있으므로 죽음의 빠른 탐지와 거짓된 알람의 가능성 사이의 트레이드오프가 있습니다. 두명의 수의사들이 마지막 맥박과 죽음 판정 사이의 시간에 대해 다른 의견을 가질 수도 있습니다. 예를 들어, 어떤 수의사는 마지막 맥박으로부터 30초 후 죽음을 판정하지만 다른 수의사는 1분을 기다리려 할 수도 있습니다. 이 경우, 이 두명의 수의사는 마지막 맥박으로부터 두번째 30초 동안 이 고양이의 상태에 대해 다른 의견을 보일 것인데, Figure 10.13에 이 상태가 그려져 있습니다.



Figure 10.13: Even Veterinarians Disagree!

Heisenberg는 우리에게 이런 종류의 불확실성을 가지고 살라고 가르쳤는데 [Hei27], 컴퓨팅 하드웨어와 소프트웨어도 비슷하게 동작하므로 이는 좋은 일입니다. 예를 들어, 여러분은 컴퓨팅 하드웨어의 한 조각이 고장났는지 어떻게 아나요? 종종 그게 빨리 응답하지 않기 때문이라고 생각합니다. 이 고양이의 맥박과 마찬가지로 이는 이 하드웨어가 정말 고장났는지 아니면 그저 조금 느려졌는지 하는 불확실성의 기간을 초래합니다.

더 나아가, 대부분의 컴퓨팅 시스템은 바깥의 세계와 상호작용하고자 의도됩니다. 따라서 바깥과의 일관성은 상당히 중요합니다. 그러나, page 159의 Figure 9.26에서 보았듯 내부 일관성의 증가는 감소된 외부 일관성의 비용으로 치루어질 수 있습니다. RCU와 해저드 포인터 같은 기법들은 더 나은 외부 일관성을 얻기 위해 약간의 내부 일관성을 포기합니다.

요약하자면 내부 일관성은 모든 문제 영역에서의 자연스러운 부분이 아니며 성능, 확장성, 외부와의 일관성, 때로는 그 모든 것의 측면에서 큰 비용을 초래하는 경우가 많습니다 [HKLP12, HHK⁺13, Rin13].

10.4 Non-Partitionable Data Structures

Don't be afraid to take a big step if one is indicated.
You can't cross a chasm in two small steps.

David Lloyd George

고정크기 해쉬테이블은 완벽하게 파티셔닝 될 수 있지만, 크기 재조정 가능한 해쉬 테이블은 크기를 늘리거나 줄일 때 파티셔닝에 있어서의 문제를 드러내는데, Figure 10.14에 이 점이 그려져 있습니다. 그러나, 높은

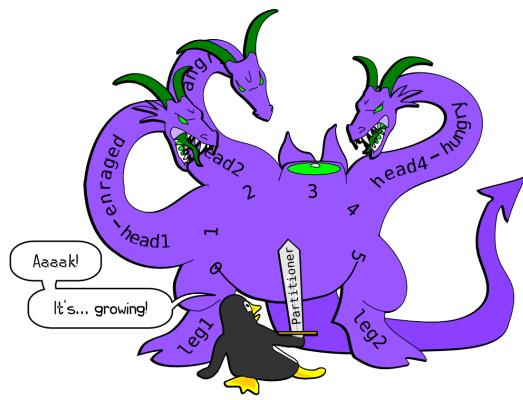


Figure 10.14: Partitioning Problems

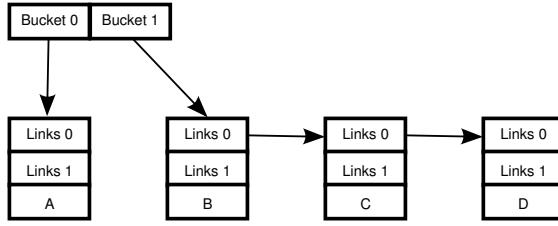


Figure 10.15: Growing a Two-List Hash Table, State (a)

성능의 확장 가능한 RCU로 보호되는 해쉬 테이블을 만드는게 가능함이 드러났는데, 이를 다음 섹션들에서 설명합니다.

10.4.1 Resizable Hash Table Design

2000년대 초의 상황과는 좋은 의미로 다르게, 확장 가능하며 RCU로 보호되는 해쉬 테이블에는 최소 세 개의 종류가 있습니다. 첫번째 (그리고 가장 간단한) 것은 리눅스 커널을 위한 것으로 Herbert Xu [Xu10]에 의해 개발되었으며, 다음 섹션에서 설명됩니다. 나머지 두개는 Section 10.4.4에서 간단히 다릅니다.

첫번째 해쉬 테이블 구현의 핵심 아이디어는 각 데이터 원소가 두개의 리스트 포인터를 가질 수 있어서, 하나는 현재 RCU 읽기 쓰레드에 의해 (그리고 RCU를 사용하지 않는 업데이터에 의해) 사용중인 것으로 설정되며 나머지는 새로운 크기 재조정 가능한 해쉬 테이블을 구성하는데 사용됩니다. 이 접근법은 탐색, 삽입, 그리고 삭제를 크기 재조정 오퍼레이션과 동시에 (그것들 각자 역시) 행해질 수 있게 합니다.

크기 재조정 오퍼레이션은 Figures 10.15–10.18, 예보인 것과 같이 진행되는데, Figure 10.15에 초기의 두개 버킷 상태가 보여져 있고 그림에서 그림으로 시간이

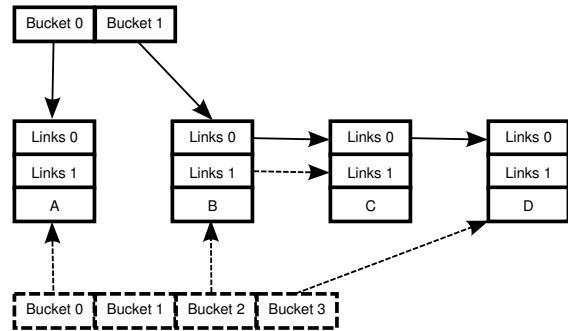


Figure 10.16: Growing a Two-List Hash Table, State (b)

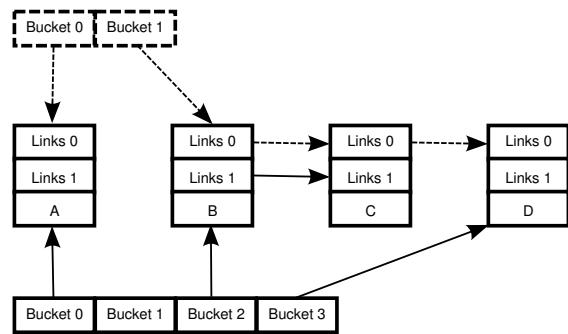


Figure 10.17: Growing a Two-List Hash Table, State (c)

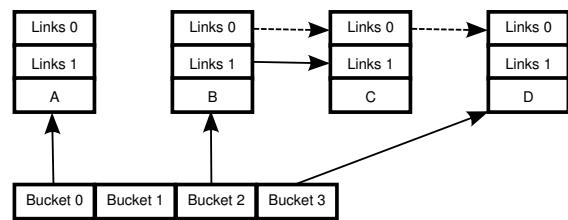


Figure 10.18: Growing a Two-List Hash Table, State (d)

진행됩니다. 최초 상태는 각 원소를 해쉬 버킷에 연결하기 위해 0-인덱스 링크를 사용합니다. 네개 버킷 배열이 할당되고, 1-인덱스 링크가 이 원소들을 이 네개의 새로운 해쉬 버킷으로 연결하기 위해 사용됩니다. 이는 Figure 10.16에 보인 상태 (b)가 되어, 각 읽기 쓰레드는 여전히 원래의 두개 버킷 배열을 사용합니다.

이 새로운 네개 버킷 배열은 읽기 쓰레드들에게 노출되고 이어서 모든 읽기 쓰레드를 위해 하나의 grace period를 기다려서 Figure 10.17에 보인 상태 (c)에 이릅니다. 이 상태에서, 모든 읽기 쓰레드는 이 새로운 네개 버킷 배열을 사용하고 있는데, 이는 오래된 두개

Listing 10.9: Resizable Hash-Table Data Structures

```

1 struct ht_elem {
2     struct rcu_head rh;
3     struct cds_list_head hte_next[2];
4 };
5
6 struct ht_bucket {
7     struct cds_list_head htb_head;
8     spinlock_t htb_lock;
9 };
10
11 struct ht {
12     long ht_nbuckets;
13     long ht_resize_cur;
14     struct ht *ht_new;
15     int ht_idx;
16     int (*ht_cmp)(struct ht_elem *hेप, void *key);
17     unsigned long (*ht_gethash)(void *key);
18     void *(*ht_getkey)(struct ht_elem *hेप);
19     struct ht_bucket ht_bkt[0];
20 };
21
22 struct ht_lock_state {
23     struct ht_bucket *hbp[2];
24     int hls_idx[2];
25 };
26
27 struct hashtab {
28     struct ht *ht_cur;
29     spinlock_t ht_lock;
30 };

```

버킷 배열은 이제 메모리 해제되어서 Figure 10.18에 보인 상태 (d)에 이를 수 있음을 의미합니다.

이 설계는 상대적으로 간단한 구현을 가능하게 하는데, 이게 다음 섹션의 주제입니다.

10.4.2 Resizable Hash Table Implementation

크기 재조정은 고전적인 간접 조정 단계를 추가하는 접근법으로 행해지는데, 이 경우 Listing 10.9 (`hash_resize.c`)의 라인 11–20에 보여진 ht 구조체입니다. 라인 27–30에 보여진 hashtab 구조체는 현재의 ht 구조체로의 구조체와 함께 동시에 해쉬 테이블을 크기 조정들을 직렬화하기 위한 스피너를 갖습니다. 우리가 고전적인 락 또는 어토믹 오퍼레이션 기반 구현을 사용하기로 했다면, 이 hashtab 구조체는 성능과 확장성 측면에서 상당한 병목이 될 수 있었을 겁니다. 하지만, 크기 재조정 오퍼레이션은 상대적으로 흔히 행해지지 않기 때문에, RCU를 잘 사용할 수 있을 겁니다.

ht 구조체는 라인 12의 `->ht_nbuckets`에 보여진 것처럼 해쉬 테이블의 구체적 크기를 표현합니다. 이 크기는 버킷 배열을 담는 같은 구조체 (line 19의 `->ht_bkt[]`)에도 크기와 배열 사이의 어긋남을 막기 위해 저장됩니다. 라인 13의 `->ht_resize_cur` 필드는 크기 재조정 오퍼레이션이 진행 중이 아니라면 -1과 동일한데, 원소들이 새로운 해쉬 테이블에 삽입되고 있는

버킷의 인덱스를 가리키는데, 라인 14의 `->ht_new`를 통해 참조됩니다. 크기 재조정 오퍼레이션이 진행중이지 않다면, `->ht_new`는 NULL입니다. 따라서, 크기 재조정은 새로운 ht 구조체를 할당하고 그것을 `->ht_new` 포인터를 통해 참조하며, 이어서 `->ht_resize_cur`를 기존 테이블의 버킷을 통해 진행시킴으로써 이루어집니다. 모든 원소가 이 새 테이블에 더해진 후에는 새 테이블이 hashtab 구조체의 `->ht_cur` 필드로 연결됩니다. 모든 기존 읽기 쓰레드들이 완료된 후에는 기존 해쉬 테이블의 ht 구조체는 메모리 해제될 수 있습니다.

라인 15의 `->ht_idx` 필드는 이 두개의 리스트 포인터 중 어느 것이 이 해쉬 테이블 현현물에 의해 사용되고 있는지를 나타내며, 라인 3의 ht_elem 구조체의 `->ht_next[]` 배열을 인덱스 합니다.

라인 16–18의 `->ht_cmp()`, `->ht_gethash()`, 그리고 `->ht_getkey()` 필드는 원소별 키와 해쉬 함수를 정의합니다. `->ht_cmp()` 함수는 명시된 키를 명시된 원소와 비교하고, `->ht_gethash()`는 명시된 키의 해쉬를 계산하며, `->ht_getkey()`는 감싸는 데이터 원소로부터 키를 얻어옵니다.

라인 22–25에 보인 ht_lock_state는 새로운 hashtab_lock_mod()로부터의 락 상태를 hashtab_add(), hashtab_del() 그리고 hashtab_unlock_mod()에 전달하기 위해 사용됩니다. 이 상태는 이 알고리즘이 동시에 크기 재조정 오퍼레이션 사이에 잘못된 버킷을 향하는 것을 막습니다.

ht_bucket 구조체는 이전과 같으며, ht_elem 구조체는 앞의 구현에서는 하나의 리스트 포인터 집합만 제공했던데 반해 리스트 포인터 집합의 두개 원소 배열을 제공한다는 점만이 다릅니다.

고정크기 해쉬 테이블에서 버킷 현택은 상당히 간단했습니다: 단순히 해쉬 값을 연관된 버킷 인덱스로 변환했습니다. 반대로, 크기를 재조정 할 때에는 버킷의 기존과 새 것들 중 어떤 것에서 선택을 할지 결정해야 합니다. 기존 테이블에서 선택될 수 있는 버킷이 이미 새 테이블로 분산되었다면, 이 버킷은 새 테이블에서는 물론이고 기존 테이블에서도 선택되어야 합니다. 반대로 이 기존 테이블에서 선택되었을 수 있는 버킷이 아직 분산되지 않았다면, 이 버킷은 기존 테이블에서 선택되어야 합니다.

라인 1–11에 `ht_get_bucket()`를, 라인 13–28에 `ht_search_bucket()`를 보이는 Listing 10.10에 버킷 선택이 보여져 있습니다. `ht_get_bucket()` 함수는 크기 재조정을 허용하지 않으면서 명시된 해쉬 테이블 내에서 명시된 키에 연관된 버킷으로의 참조를 반환합니다. 이 함수는 또한 라인 7에서 이 키에 연관된 버킷 인덱스를 패러미터 `b`로 참조되는 위치에 저장하며, 라인 9에서 패러미터 `h`로 참조되는 위치에 (`h`가 NULL이 아니라면) 이 키에 연관되는 해쉬 값을 저장합니다.

Listing 10.10: Resizable Hash-Table Bucket Selection

```

1 static struct ht_bucket *
2 ht_get_bucket(struct ht *htp, void *key,
3               long *b, unsigned long *h)
4 {
5     unsigned long hash = htp->ht_gethash(key);
6
7     *b = hash % htp->ht_nbuckets;
8     if (*h)
9         *h = hash;
10    return &htp->ht_bkt[*b];
11 }
12
13 static struct ht_elem *
14 ht_search_bucket(struct ht *htp, void *key)
15 {
16     long b;
17     struct ht_elem *htep;
18     struct ht_bucket *htbp;
19
20     htp = ht_get_bucket(htp, key, &b, NULL);
21     cds_list_for_each_entry_rcu(htep,
22                                 &htbp->htb_head,
23                                 hte_next[htp->ht_idx]) {
24         if (htp->ht_cmp(htep, key))
25             return htep;
26     }
27     return NULL;
28 }

```

`ht_search_bucket()` 함수는 명시된 해쉬 테이블 버전 내에서 명시된 키를 위한 탐색을 합니다. 라인 20은 명시된 키에 연관된 버킷으로의 참조를 얻습니다. 라인 21-26의 반복문은 그 버킷을 찾아서 라인 24가 들어맞음을 발견하면 라인 25는 이를 감싸는 데이터 원소로의 포인터를 반환하게 합니다. 그렇지 않고 들어맞는 경우가 없다면 라인 27은 실패를 알리기 위해 `NULL`을 반환합니다.

Quick Quiz 10.9: Listing 10.10의 코드는 크기 재조정 프로세스가 선택된 버킷을 지나가는 걸 어떻게 보호합니까?



이 `ht_get_bucket()`과 `ht_search_bucket()`의 구현은 탐색과 수정이 크기 재조정 오퍼레이션과 동시에 이루어질 수 있게 합니다.

읽기 쪽 동시성 제어는 Listing 10.6에 보여졌듯 RCU에 의해 제공됩니다만, 업데이트 쪽 동시성 제어 기능인 `hashtab_lock_mod()`과 `hashtab_unlock_mod()`는 Listing 10.11에 보이듯 동시에의 크기 재조정 오퍼레이션의 가능성을 처리해야만 합니다.

`hashtab_lock_mod()`는 이 리스트의 라인 1-25에 있습니다. 라인 10은 데이터 구조가 순회 도중 메모리 해제되는 걸 막기 위해 RCU read-side 크리티컬 섹션에 들어가고, 라인 11은 현재 해쉬 테이블로의 참조를 획득하며, 라인 12은 이 키에 연관되는 해쉬 테이블의 버킷으로의 참조를 얻습니다. 라인 13은 이 버킷의 락을 획득하는데, 이는 이 버킷이 동시에의 크기 재조정 오퍼레이션에 의해 분산되는 것을 막을 것인데, 물론

Listing 10.11: Resizable Hash-Table Update-Side Concurrency Control

```

1 static void
2 hashtab_lock_mod(struct hashtab *htp_master, void *key,
3                   struct ht_lock_state *lsp)
4 {
5     long b;
6     unsigned long h;
7     struct ht *htp;
8     struct ht_bucket *htbp;
9
10    rcu_read_lock();
11    htp = rcu_dereference(htp_master->ht_cur);
12    htp = ht_get_bucket(htp, key, &b, &h);
13    spin_lock(&htbp->htb_lock);
14    hbp[0] = htp;
15    hls_idx[0] = htp->ht_idx;
16    if (b > READ_ONCE(htp->ht_resize_cur)) {
17        lsp->hbp[1] = NULL;
18        return;
19    }
20    htp = rcu_dereference(htp->ht_new);
21    htp = ht_get_bucket(htp, key, &b, &h);
22    spin_lock(&htbp->htb_lock);
23    hbp[1] = htp;
24    hls_idx[1] = htp->ht_idx;
25 }
26
27 static void
28 hashtab_unlock_mod(struct ht_lock_state *lsp)
29 {
30     spin_unlock(&lsp->hbp[0]->htb_lock);
31     if (lsp->hbp[1])
32         spin_unlock(&lsp->hbp[1]->htb_lock);
33     rcu_read_unlock();
34 }

```

이 버킷이 이미 분산되어 있다면 아무 영향을 끼치지 못할 것이긴 합니다. 라인 14-15는 버킷 포인터와 포인터 집합 인덱스를 `ht_lock_state` 구조체의 연관된 필드에 저장하는데, 이는 `hashtab_add()`, `hashtab_del()`, 그리고 `hashtab_unlock_mod()`에게 이 정보를 전달하는 것입니다. 라인 16는 동시에의 크기 재조정 오퍼레이션이 이미 이 버킷을 새 해쉬 테이블로 분산시켰는지 검사하고, 그렇지 않다면 라인 17에서 이미 크기 재조정된 해쉬 버킷이 없음을 알리고 라인 18에서 선택된 해쉬 버킷의 락이 잡힌 채 (따라서 동시에의 크기 재조정 오퍼레이션이 이 버킷을 분산시키는 것을 막은 채), 그리고 RCU read-side 크리티컬 섹션 안에 있는 채리턴합니다. 기존 테이블의 락은 새 테이블의 것들 전에 항상 잡혀 있으므로, 그리고 RCU의 사용이 한번에 두개를 넘는 버전이 조재하는 것을 막기 때문에 데드락 사이클이 방지되어 데드락이 회피됩니다.

그렇지 않고 동시에의 크기 재조정 오퍼레이션이 이미 이 버킷을 분산시켰다면, 라인 20은 새 해쉬 테이블로 넘어가서 라인 21이 이 키에 연관된 버킷을 선택하고 라인 22에서 이 버킷의 락을 획득합니다. 라인 23-24는 이 버킷 포인터와 포인터 집합 인덱스를 `ht_lock_state` 구조체의 연관된 필드에 저장하는데, 다시 말하지만 이 정보를 `hashtab_add()`, `hashtab_del()`, 그리

Listing 10.12: Resizable Hash-Table Access Functions

```

1 struct ht_elem *
2 hashtab_lookup(struct hashtab *htp_master, void *key)
3 {
4     struct ht *htp;
5     struct ht_elem *htep;
6
7     htp = rCU_dereference(htp_master->ht_cur);
8     htep = ht_search_bucket(htp, key);
9     return htep;
10 }
11
12 void hashtab_add(struct ht_elem *htep,
13                     struct ht_lock_state *lsp)
14 {
15     struct ht_bucket *htbp = lsp->hbp[0];
16     int i = lsp->hls_idx[0];
17
18     cds_list_add_rcu(&htep->hte_next[i], &htbp->htb_head);
19     if ((htbp = lsp->hbp[1])) {
20         cds_list_add_rcu(&htep->hte_next[!i], &htbp->htb_head);
21     }
22 }
23
24 void hashtab_del(struct ht_elem *htep,
25                     struct ht_lock_state *lsp)
26 {
27     int i = lsp->hls_idx[0];
28
29     cds_list_del_rcu(&htep->hte_next[i]);
30     if (lsp->hbp[1])
31         cds_list_del_rcu(&htep->hte_next[!i]);
32 }

```

고 `hashtab_unlock_mod()`에게 전달하기 위함입니다. 이 버킷은 이미 크기 재조정 되었고 `hashtab_add()`와 `hashtab_del()`이 기존과 새로운 `ht_bucket` 구조체 모두에 영향을 끼쳤으므로, 두개의 버킷 각각 하나씩 두개의 락이 잡혀 있습니다. 더해서, `ht_lock_state` 구조체의 각 배열의 원소들 다 사용되었는데, [0] 원소는 기존 `ht_bucket` 구조체를 따르고 있고 [1] 원소는 새 구조체를 따르고 있습니다. 다시 말하지만, `hashtab_lock_mod()`는 RCU read-side 크리티컬 섹션에 있습니다.

`hashtab_unlock_mod()` 함수는 `hashtab_lock_mod()`에 의해 획득된 락들을 해제합니다. 라인 30은 기존 `ht_bucket` 구조체의 락을 해제합니다. 라인 31이 크기 재조정 오퍼레이션이 진행 중이라는 드문 이벤트를 감지하면 라인 32은 새 `ht_bucket` 구조체의 락을 해제합니다. 어떤 경우든 라인 33은 RCU read-side 크리티컬 섹션을 빠져나갑니다.

Quick Quiz 10.10: 한 쓰레드가 크기 재조정 오퍼레이션 중에 이 해쉬 테이블에 원소를 하나 넣는다고 생각해 봅시다. 뒤따르는 크기 재조정 오퍼레이션이 이 삽입 전에 완료됨으로 인해 이 삽입이 사라지는 것은 어떻게 방지됩니까?

이제 우린 버킷 선택과 동시성 제어를 가지고 있으므로, 이 크기 재조정 가능한 해쉬 테이블을 탐색하고

업데이트할 준비가 되었습니다. `hashtab_lookup()`, `hashtab_add()`, 그리고 `hashtab_del()` 함수가 Listing 10.12에 있습니다.

이 리스트의 라인 1-10에 있는 `hashtab_lookup()` 함수가 해쉬 탐색을 합니다. 라인 7은 현재 해쉬 테이블을 가져오고 라인 8이 이 키에 연관된 버킷을 찾습니다. 라인 9는 탐색된 원소로의 포인터를, 탐색이 실패한 경우 `NULL`을 반환합니다. 호출자는 RCU read-side 크리티컬 섹션 내에 있어야만 합니다.

Quick Quiz 10.11: Listing 10.12의 `hashtab_lookup()` 함수는 동시의 크기 재조정 오퍼레이션을 무시합니다. 이는 읽기 쓰레드가 크기 재조정 오퍼레이션 중에 추가된 원소를 놓칠 수도 있음을 의미하지 않습니까?

이 리스트의 라인 12-22에 있는 `hashtab_add()` 함수는 이 해쉬 테이블에 새 데이터 원소를 추가합니다. 라인 15는 이 새 원소가 추가될 현재의 `ht_bucket` 구조체를 고르고, 라인 16는 이 포인터 상의 인덱스를 가져옵니다. 라인 18은 이 새 원소를 현재의 해쉬 버킷에 더합니다. 라인 19가 이 버킷이 새 버전의 해쉬 테이블로 분산되었음을 파악하면 라인 20가 이 새 원소를 연관된 새 버킷에도 더합니다. 호출자는 동시성을 제어할 책임을 갖는데, 예를 들어 `hashtab_add()`의 호출 전에는 `hashtab_lock_mod()`를, 후에는 `hashtab_unlock_mod()`를 호출하는 식입니다.

이 리스트의 라인 24-32에 있는 `hashtab_del()` 함수는 이 해쉬 테이블에서 존재하는 원소 하나를 제거합니다. 라인 27는 포인터 쌍의 인덱스를 가져오고 라인 29에서 명시된 원소를 현재 테이블에서 제거합니다. 라인 30가 이 버킷이 새 버전의 해쉬 테이블로 분산되었음을 파악하면 라인 31가 명시된 원소를 연관된 새 버킷에서도 제거합니다. `hashtab_add()`에서와 마찬가지로 호출자는 동시성 제어의 책임을 가지며 이 동시성 제어는 동시의 크기 재조정 오퍼레이션과의 동기화만으로도 충분합니다.

Quick Quiz 10.12: Listing 10.12의 `hashtab_add()`와 `hashtab_del()` 함수는 크기 재조정 오퍼레이션이 진행 중인 사이에 두개의 해쉬 버킷을 업데이트 할 수 있습니다. 이는 크기 재조정 오퍼레이션의 빈도가 무시할 만한 수준이 아니라면 성능 저하를 일으킬 수도 있을 겁니다. 그런 경우에 업데이트의 비용을 줄일 수 있지 않습니까?

실제 크기 재조정은 page 188의 Listing 10.13에 보여져 있습니다. 라인 16은 조건적으로 꼭대기 단계의 `->ht_lock`을 획득하고, 이 획득이 실패하면 라인 17에서 크기 재조정이 이미 진행 중임을 알리는 `-EBUSY`를 리턴합니다. 그렇지 않다면, 라인 18이 현재 해쉬

Listing 10.13: Resizable Hash-Table Resizing

```

1 int hashtab_resize(struct hashtab *htp_master,
2                     unsigned long nbuckets,
3                     int (*cmp)(struct ht_elem *htep, void *key),
4                     unsigned long (*gethash)(void *key),
5                     void *(*getkey)(struct ht_elem *htep))
6 {
7     struct ht *htp;
8     struct ht *htp_new;
9     int i;
10    int idx;
11    struct ht_elem *htep;
12    struct ht_bucket *htbp;
13    struct ht_bucket *htbp_new;
14    long b;
15
16    if (!spin_trylock(&htp_master->ht_lock))
17        return -EBUSY;
18    htp = htp_master->ht_cur;
19    htp_new = ht_alloc(nbuckets,
20                      cmp ? cmp : htp->ht_cmp,
21                      gethash ? gethash : htp->ht_gethash,
22                      getkey ? getkey : htp->ht_getkey);
23    if (htp_new == NULL) {
24        spin_unlock(&htp_master->ht_lock);
25        return -ENOMEM;
26    }
27    idx = htp->ht_idx;
28    htp_new->ht_idx = !idx;
29    rcu_assign_pointer(htp->ht_new, htp_new);
30    synchronize_rcu();
31    for (i = 0; i < htp->ht_nbuckets; i++) {
32        htpb = &htp->ht_bkt[i];
33        spin_lock(&htpb->htb_lock);
34        cds_list_for_each_entry(htep, &htpb->htb_head, hte_next[idx]) {
35            htpb_new = ht_get_bucket(htp_new, htp_new->ht_getkey(htep), &b, NULL);
36            spin_lock(&htpb_new->htb_lock);
37            cds_list_add_rcu(&htep->hte_next[!idx], &htpb_new->htb_head);
38            spin_unlock(&htpb_new->htb_lock);
39        }
40        WRITE_ONCE(htp->ht_resize_cur, i);
41        spin_unlock(&htpb->htb_lock);
42    }
43    rcu_assign_pointer(htp_master->ht_cur, htp_new);
44    synchronize_rcu();
45    spin_unlock(&htp_master->ht_lock);
46    free(htp);
47    return 0;
48 }

```

테이블로의 레퍼런스를 가져오고, 라인 19–22에서 원하는 크기의 새 해쉬 테이블을 할당합니다. 새로운 해쉬 키 함수가 명시되었다면, 그것들이 새 해쉬 테이블에서 사용되며, 그렇지 않다면 기존 테이블의 것들이 유지됩니다. 라인 23이 메모리 할당 실패를 감지하면 라인 24이 `->ht_lock`을 해제하고 라인 25에서 실패 알림을 리턴합니다.

라인 27는 현재 테이블의 인덱스를 가져오고 라인 28는 그것의 해쉬 테이블로의 역을 저장하여서 이 두개의 해쉬 테이블이 서로의 링크드 리스트를 덮어쓰는 것을 막습니다. 이어서 라인 29은 새 테이블로의 레퍼런스를 기존 테이블의 `->ht_new` 필드에 저장함으로써 버킷 분산 프로세스를 시작합니다. 라인 30는 새 테이블을

모르는 모든 읽기 쓰레드가 크기 재조정 오퍼레이션이 이어지기 전에 완료될 것을 보장합니다.

라인 31–42의 반복문의 각 패스는 기존 해쉬 테이블의 버킷들 중 하나의 내용물들을 새로운 해쉬 테이블로 분산시킵니다. 라인 32는 기존 테이블의 현재 버킷으로의 레퍼런스를 가져오고 라인 33는 그 버킷의 스판락을 획득합니다.

Quick Quiz 10.13: Listing 10.13의 `hashtab_resize()` 함수에서, 라인 29에서의 `->ht_new`로의 업데이트가 라인 40에서의 `->ht_resize_cur` 업데이트 보다 전에 일어난 것으로 `hashtab_add()`와 `hashtab_del()`의 관점에 보일 것을 무엇이 보장합니까? 달리 말하자면, `hashtab_add()`와 `hashtab_del()`이 `->`

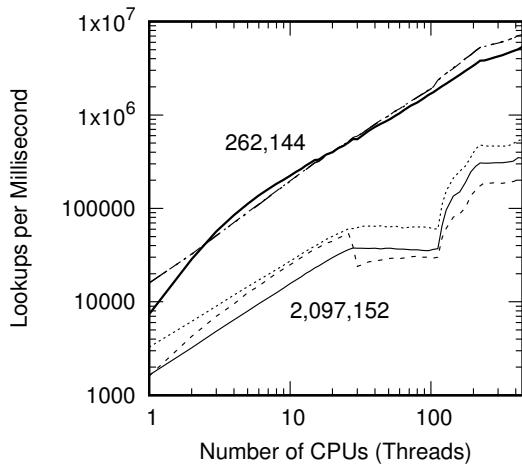


Figure 10.19: Overhead of Resizing Hash Tables Between 262,144 and 524,288 Buckets vs. Total Number of Elements

ht_new에서 얻어진 NULL 포인터를 역참조 하는 것을 무엇이 방지합니까?

라인 34-39의 반복문의 각 패스는 현재의 기존 테이블을 버킷으로부터 데이터 원소 하나를 연관된 새 테이블의 버킷으로 더하며, 이 더하기 오퍼레이션 동안 새 테이블의 버킷의 락을 잡고 있습니다. 라인 40는 이 버킷이 분산되었음을 알리기 위해 `->ht_resize_cur`를 업데이트 합니다. 마지막으로, 라인 41는 기존 테이블 버킷 락을 해제합니다.

모든 기존 테이블 버킷이 새 테이블로 분산되고 나면 수행은 라인 43에 도달합니다. 라인 44는 (기존 테이블을 아직 참조하고 있을 수도 있는) 모든 읽기 쓰레드들이 완료되기를 기다립니다. 라인 45는 크기 재조정 직렬화 락을 해제하고, 라인 46는 기존 해쉬 테이블을 메모리 해제하며, 마지막으로 라인 47를 성공을 리턴합니다.

Quick Quiz 10.14: Listing 10.13의 라인 40에는 왜 `WRITE_ONCE()`가 있나요?

1 밀리세컨드를 멈추며 왔다갔다 하는 해쉬 테이블을 보입니다.

가장 위의 세개 선은 262,144 개 원소 해쉬 테이블의 것입니다. 점선은 두개의 고정 크기 해쉬 테이블에 연관되며, 실선은 크기 재조정 가능한 해쉬 테이블의 것입니다. 이 경우, 짧은 해쉬 체인이 평범한 탐색 오버헤드를 낮게 만들어 크기 재조정 오버헤드가 대부분의 경우 지배적이게 만듭니다. 또한, 전체 해쉬 테이블은 L3 캐시에 딱 맞습니다.

아래의 세개 선은 2,097,152-원소 해쉬 테이블의 것입니다. 위쪽 선은 262,144-버킷 고정 크기 해쉬 테이블의 것이고, 중간의 것은 낮은 CPU 갯수의, 그리고 바닥의 것은 높은 CPU 갯수의 크기 재조정 가능한 해쉬 테이블의 것이며, 남은 선은 524,288-버킷 고정크기 해쉬 테이블의 것입니다. 이제 버킷당 평균 여덟개의 원소가 있다는 사실만이 성능 하락을 일으킬 것이라 예상할 수 있으며 실제로 그래프가 이를 보입니다. 하지만 더 나쁜게, 이 해쉬 테이블의 원소들은 128 MB를 차지하는데, 이는 각 소켓의 39 MB L3 캐시를 넘치게 하여, Section 3.2.2에서 보인 것과 비슷한 성능 결과를 일으킵니다. 결과적으로 일어나는 캐시 넘침은 메모리 시스템이 읽기만 하는 벤치마크에서 조차도 관여되며, 아래 세개 선의 선형에 가까운 정도에서 볼 수 있듯이, 이 메모리 시스템은 심각한 병목이 될 수 있습니다.

Quick Quiz 10.15: Figure 10.19에 보인 크고 작은 해쉬 테이블들 사이의 성능 차이 중 얼마큼이 긴 해쉬 체인 때문이었고 얼마큼은 메모리 시스템 병목 때문이었나요?

Table 3.1의 마지막 행을 보면, 첫번째 28 CPU는 첫번째 소켓에 있어서, 코어당 CPU 하나 기준임을 기억하면 크기 재조정 가능한 해쉬 테이블의 28 CPU 넘어서의 성능이 떨어지는 모습을 설명할 수 있습니다. 이 성능 감소가 크긴 하지만, 그것은 지속적으로 크기를 재조정하기 때문임을 기억하시기 바랍니다. 524,288 버킷으로 일단 크기 재조정하고 나면 분명 나을 것이며, 2,097,152 원소의 1/8로의 단 한번의 크기 재조정하여 버킷당 원소의 평균 갯수를 위의 세개 선에서처럼 줄이면 더 나을 겁니다.

이 데이터에서의 핵심은 RCU로 보호되는 크기 재조정 가능한 해쉬 테이블이 그 고정크기 버전만큼이나 성능과 확장성이 좋다는 것입니다. 실제 크기 재조정 오퍼레이션 동안의 성능은 물론 각 원소의 포인터들로 행해지는 업데이트에 의해 생겨난 캐시 미스들 같은 걸로 인해 떨어질 것이고 이 효과는 메모리 시스템이 병목이 될 때 가장 잘 드러날 겁니다. 이는 해쉬 테이블이 상당한 정도로 크기 재조정되어야 함을 알리며, 그 이력 현상은 너무 잦은 크기 재조정 오퍼레이션으로 인한 성능 하락을 막기 위해 적용되어야 합니다. 메모

10.4.3 Resizable Hash Table Discussion

Figure 10.19는 262,144 개와 2,097,152 개 원소의 해쉬 테이블을 크기 재조정 하는 것을 고정 크기의 것과 비교해 봅니다. 이 그림은 각 원소 갯수에 대해 세개의 선을 보이는데, 하나는 고정 크기 262,144 버킷 해쉬 테이블을, 다른 하나는 고정 크기 524,288 버킷 해쉬 테이블을, 그리고 마지막은 크기 재조정 가능하며 262,144 개와 524,288 개 버킷 사이를 크기 재조정 오퍼레이션 사이

리가 충분한 환경에서는 해쉬 테이블 크기가 줄어드는 것보다 훨씬 적극적으로 더 늘어나야 할 겁니다.

또 다른 핵심은 `hashtab` 구조체가 파티셔닝 불가하기는 하나 읽기가 대부분이라는 것으로 RCU 사용을 추천하게 합니다. 이 크기 재조정 가능한 해쉬 테이블의 성능과 확장성은 RCU로 보호되는 고정크기 해쉬 테이블의 그것에 무척 근접함을 놓고 보면, 우린 이 접근법이 상당히 성공적이라 결론내려야 하겠습니다.

마지막으로, 삽입, 삭제, 그리고 탐색이 크기 재조정 오퍼레이션과 동시에 수행될 수 있음을 알아두는게 중요합니다. 이 동시성은 커다란 해쉬 테이블을 크기 재조정할 때, 특히 상당한 응답 시간 제한을 지켜야 하는 어플리케이션에서는 치명적으로 중요합니다.

물론, `ht_elem` 구조체의 포인터 쌍은 약간의 메모리 오버헤드를 일으키는데, 다음 섹션에서 이야기 됩니다.

10.4.4 Other Resizable Hash Tables

이 섹션의 앞부분에서 설명된 크기 재조정 가능한 해쉬 테이블의 한가지 단점은 메모리 소모입니다. 각 데이터 원소는 하나가 아닌 두 쌍의 링크드 리스트 포인터를 갖습니다. 한 쌍의 것만 가지며 RCU로 보호되고 크기 재조정도 가능한 해쉬 테이블을 만들 수 있을까요?

결과는 “그렇다”로 드러났습니다. Josh Triplett 등은 [TMW11] 점진적으로 연관된 해쉬 체인들을 쪼개고 합쳐서 읽기 쓰레드가 항상 유효한 해쉬 체인을 크기 재조정 오퍼레이션 중간의 모든 지점에서 볼 수 있게 하는 *relativistic hash table*을 만들었습니다. 이 점진적 쪼개기와 합치기는 읽기 쓰레드들이 다른 해쉬 체인에 있는 원소를 보는 것이 문제되지 않는다는 사실에 기반합니다: 이게 일어나면, 읽기 쓰레드는 키 미스매치로 인해 이 관계없는 데이터를 그냥 무시할 겁니다.

이 *relativistic* 해쉬 테이블의 크기를 두배 줄이는 과정이 Figure 10.20에 보여져 있는데, 이 경우 두개 베킷 해쉬 테이블을 한개 베킷 해쉬 테이블, 달리 말하면 하나의 리니어 리스트로 줄입니다. 이 프로세스는 기존의 커다란 해쉬 테이블의 한 쌍의 베킷을 새 작은 해쉬 테이블의 단일 베킷으로 합침으로써 행해집니다. 이 프로세스가 올바르게 동작하기 위해 우린 두 테이블을 위한 해쉬 함수를 분명히 제한해야 합니다. 그런 제한 하나는 두 테이블에서 모두 똑같은 해쉬 함수를 사용하되 큰 테이블에서 작은 테이블로 줄일 때에는 아래 비트를 버릴 것입니다. 예를 들어, 기존의 두개 베킷 해쉬 테이블은 이 값의 위쪽 두 비트를 사용하는 반면, 새 한개 베킷 해쉬 테이블은 이 값의 위쪽 한개 비트만 사용할 수 있습니다. 이 방법으로, 기존 큰 해쉬 테이블의 인접한 수와 홀수 베킷은 새 작은 해쉬 테이블의 단일 베킷으로 모아질 수 있으며, 여전히 이 단일 베킷의 원소를 모두 다룰 수 있는 하나의 해쉬 값을 가질 수 있습니다.

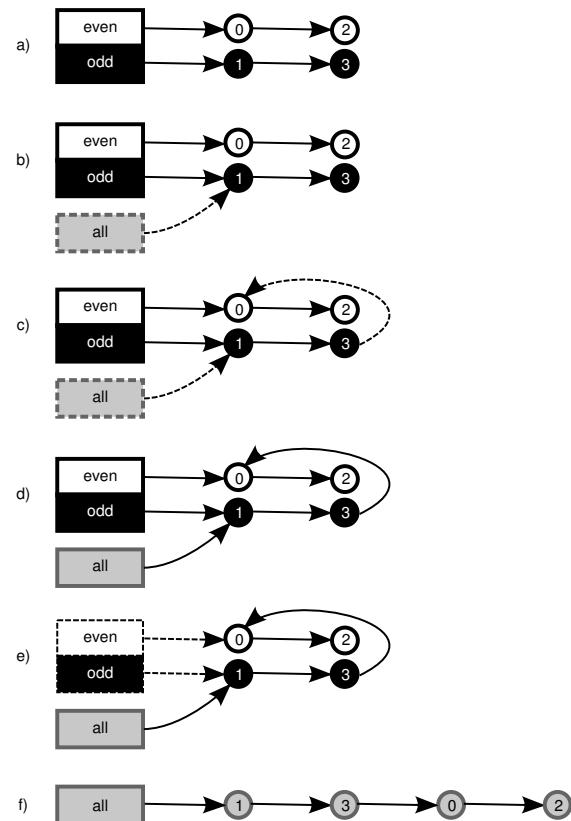


Figure 10.20: Shrinking a Relativistic Hash Table

최초의 상태가 이 그림의 꼭대기에 보여져 있고, 초기 상태 (a)로부터 시작해서 시간은 위에서 아래로 흐릅니다. 크기 줄이기 프로세스는 새 작은 베킷 배열을 할당하는 것으로 시작하고, 이 작은 배열의 각 베킷이 기존 해쉬 테이블의 연관된 베킷 쌍 중 하나의 첫번째 원소를 참조하게 하여 상태 (b)에 이르게 합니다.

이어서 두 해쉬 체인이 서로 이어져서 상태 (c)에 이릅니다. 이 상태에서, 짝수 원소를 탐색하는 읽기 쓰레드는 변화를 느끼지 못하며, 원소 1과 3을 찾는 읽기 쓰레드 역시 변화를 보지 못합니다. 그러나, 홀수 원소를 탐색하는 읽기 쓰레드들은 원소 0과 2도 보게 될 겁니다. 모든 홀수 원소는 이 두 원소들과 동일하지 않다고 비교될 것이므로 이는 문제가 되지 않습니다. 약간의 성능 하락이 있겠지만, 이는 또한 일단 이 작은 해쉬 테이블이 완전히 만들어지면 어차피 겪게 될 성능 하락입니다.

다음으로, 이 작은 해쉬 테이블이 읽기 쓰레드들에게 접근 가능해져서 상태 (d)가 됩니다. 기존 읽기 쓰레드들은 여전히 기존 큰 해쉬 테이블을 순회하고 있을 수 있어서 이 상태에서는 두 해쉬 테이블이 모두 사용중이 됩니다.

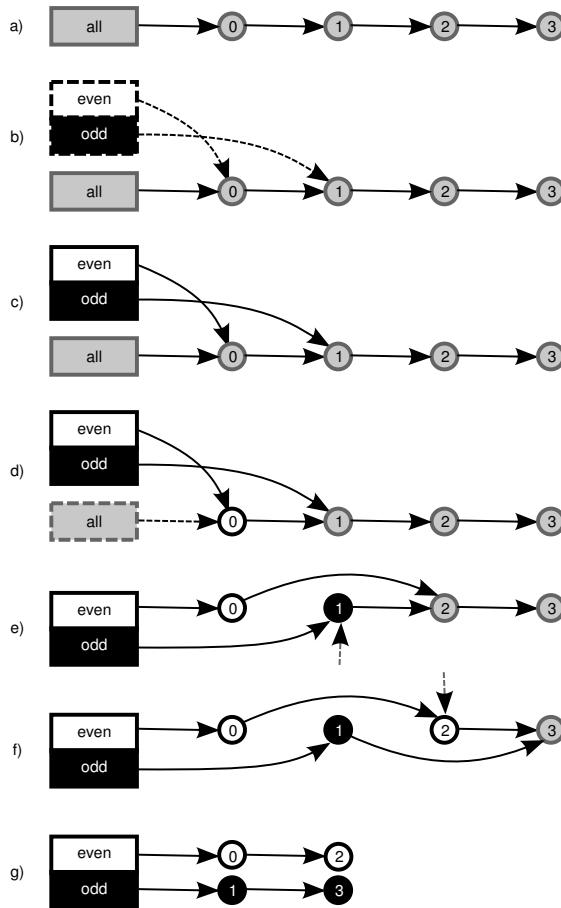


Figure 10.21: Growing a Relativistic Hash Table

다음 단계는 모든 앞서서부터 존재해온 읽기 쓰레드들이 완료되길 기다리는 것으로 상태 (e) 가 됩니다. 이 시점에서, 모든 읽기 쓰레드는 새 작은 해쉬 테이블을 사용하고 있으므로 기존의 큰 해쉬 테이블의 버킷은 메모리 해제될 수 있어서 마지막 상태 (f) 에 이릅니다.

Relativistic 해쉬 테이블의 크기를 키우는 것은 줄이는 과정의 반대입니다만, Figure 10.21 에 보인 것처럼 더 많은 grace-period 단계가 필요합니다. 처음 상태 (a) 가 이 그림의 꼭대기에 있고, 시간이 위에서 아래로 흐릅니다.

새로운 커다란 두개 버킷 해쉬 테이블을 할당하는 것으로 시작해 상태 (b) 에 이릅니다. 이 새 버킷들 각각은 그 버킷을 향하게 되는 첫번째 원소를 참조하고 있음을 주의하세요. 이 새 버킷들은 읽기 쓰레드들에게 공개되어 상태 (c) 에 이릅니다. 이 상태에서, 짝수 번호 해쉬 버킷을 횡단하는 읽기 쓰레드들만이 원소 0에 다다르게 되는데, 그래서 흰색으로 칠해져 있습니다.

이 시점에서, 기존의 작은 해쉬 버킷들은 메모리 해제될 수 있습니다만 많은 구현들은 이 오래된 버킷들을 항목들의 리스트를 연관된 새 버킷으로 “풀어내는” 과정을 추적하기 위해 사용합니다. 그런 원소들의 첫번째 연속된 순회의 마지막 짝수 번호 원소는 이제 뒤따르는 짝수 번호 원소를 참조하도록 업데이트 된 다음 포인터를 가집니다. 다음 grace period 오퍼레이션 후에는 상태 (e) 가 됩니다. 새로 화살표는 풀어져야 할 다음 원소를 가리키며, 원소 1은 이제 홀수 번호 해쉬 버킷을 횡단하는 읽기 쓰레드만이 그것에 다다를 수 있음을 알리기 위해 검은 색으로 칠해져 있습니다.

다음으로, 마지막 그린 원소들의 첫번째 연속 수행에서의 마지막 홀수 번호 원소는 이제 뒤따르는 홀수 번호 원소를 참조하도록 업데이트 된 다음 포인터를 가지게 되었습니다. 다음 grace-period 오퍼레이션 후에는 상태 (f) 에 다다릅니다. 마지막 풀어내기 오퍼레이션 (grace-period 오퍼레이션 포함)은 마지막 상태 (g) 에 이르게 합니다.

요약하면, relativistic 해쉬 테이블은 원소별 리스트 포인터의 갯수를 크기 재조절 중에 발생하는 추가적인 grace period 의 비용으로 줄입니다. 이 추가적인 grace period 는 삽입, 삭제, 그리고 탐색이 크기 재조정 오퍼레이션과 동시에 이루어지므로 일반적으로는 문제가 아닙니다.

$O(1)$ 삭제를 유지하면서도 원소별 메모리 오버헤드를 한쌍의 포인터에서 하나의 포인터로 줄일 수 있음이 드러났습니다. 이는 split-order 리스트를 [SS06] RCU 보호와 결합하여 [Des09b, MDJ13a] 이루어졌습니다. 해쉬 테이블의 데이터 원소들은 하나의 정렬된 링크드 리스트로 구성되고, 각 해쉬 버킷은 해당 버킷의 첫번째 원소를 참조합니다. 원소들은 각자의 다음으로의 포인터 필드의 아래쪽 비트를 설정하는 것으로 삭제되며, 이 원소들은 그것들을 마주하는 다음 순회에서 리스트로부터 제거됩니다.

이 RCU로 보호되는 split-order 리스트는 복잡하지만 모든 삽입, 삭제, 그리고 탐색 오퍼레이션에 lock-free 진행 보장을 제공합니다. 그런 보장은 real-time 오플리케이션에서는 중요할 수 있습니다. 최신 버전의 userspace RCU 라이브러리 [Des09b]에서 그 구현을 구할 수 있습니다.

10.5 Other Data Structures

All life is an experiment. The more experiments you make the better.

Ralph Waldo Emerson

앞의 섹션들은 파티셔닝 가능성을 통해 (Section 10.2), 효율적인 읽기가 대부분인 액세스 패턴 처리를 통해 (Section 10.3), 또는 파티셔닝 불가능성을 막기 위한 읽기가 대부분인 기법 적용을 통해 (Section 10.4) 동시성을 개선하는 데이터 구조들에 집중했습니다.

병렬 사용을 위한 해쉬 테이블의 큰 장점 가운데 하나는 적어도 크기 재조정 되고 있지 않는 동안은 완벽히 파티셔닝 가능하다는 것입니다. 이 파티셔닝 가능성과 크기를 독립적으로 유지하는 한 가지 방법은 trie라고도 불리는 radix 트리를 사용하는 것입니다. Trie는 중첩된 해쉬 테이블이 집합이라고 생각될 수 있으며, 따라서 요구되는 파티셔닝 가능성을 제공합니다. Trie의 단점 중 하나는 높도가 옅은 키 공간은 비효율적인 메모리 사용을 초래할 수 있다는 겁니다. 이 단점을 막기 위해 사용 가능한 여러 압축 기법이 있는데, 순회 전에 키 값을 더 작은 키 공간으로 해쉬 하는 기법이 [ON07] 포함됩니다. Radix 트리는 리눅스 커널을 포함해 [Pig06] 실전에서 많이 사용되고 있습니다.

해쉬 테이블과 trie 모두에 중요한 특수한 경우 하나는 아마도 가장 오래된 데이터 구조인 배열과 그것의 다차원 버전, 즉 행렬입니다. 행렬들의 완벽히 파티셔닝 가능한 본성은 동시적 수학 알고리즘에서 널리 사용됩니다.

스스로 밸런스를 맞추는 트리는 동시성 코드에서 널리 사용되는데, AVL 트리와 red-black 트리가 가장 널리 알려진 예일 겁니다 [CLRS01]. AVL 트리를 병렬화 하려는 앞선 시도들은 복잡하고 모두가 효율적이지는 않았습니다만 [Eli80], 보다 최근의 red-black 트리 작업은 읽기 쓰레드들에게 RCU를 사용하게 하고 읽기와 업데이트 각각을 보호하는데 락들의 해쉬된 배열을 제공함으로써 [HW11, HW13] 더 나은 성능과 확장성을 제공합니다¹. Red-black 트리는 적극적으로 밸런스를 맞추어서 순차적 프로그램에서는 잘 동작하지만 병렬의 사용에서는 꼭 그렇지만은 않음이 드러났습니다. 따라서 최근의 작업은 RCU로 보호되는 “bonsai tree”를 사용했는데, 이것은 덜 적극적으로 밸런스를 맞추어서 [CKZ12], 더 효율적인 동시 업데이트를 위해 트리의 최적 깊이를 희생합니다.

¹ SwissTM의 [DFGG11]의 형태로 제공하는데, 이는 개발자가 공유되지 않는 액세스를 표시해 두는 형태의 소프트웨어 트랜잭션 메모리의 한 변종입니다.

동시성 skip list 역시 RCU 읽기 쓰레드에 적용될 수 있으며, RCU를 닮은 초기의 학계에서의 사용에서 [Pug90] 실제로 소개된 바 있습니다.

동시의 양 끝단 queue는 Section 6.1.2에서 이야기 되었고, 동시의 스택과 queue는 긴 역사를 [Tre86] 가지고 있습니다만, 보통 가장 인상적인 성능과 확장성을 갖지는 않습니다. 그것들은 다만 동시성 라이브러리의 흔한 기능입니다 [MDJ13b]. 연구자들은 최근 스택과 queue의 순서 규칙을 완화하는 것을 제안했는데 [Sha11], 관련된 연구에서는 완화된 순서규칙의 queue가 실제로 엄격한 FIFO queue 보다 더 나은 순서 속성을 가짐을 드러냈습니다 [HKLP12, KLP12, HHK⁺13].

동시적 데이터 구조들에 대한 계속된 연구는 놀라운 특성을 갖는 최신 알고리즘들을 만들어낼 가능성이 높아 보입니다.

10.6 Micro-Optimization

The devil is in the details.

Unknown

이 섹션에 보인 데이터 구조들은 그 아래 있는 시스템의 캐쉬 계층에 맞춘 조정 없이 단순하게 짜였습니다. 또한, 많은 구현이 키에서 해쉬로의 변환과 그 외에도 갖은 오퍼레이션들을 위해 함수 포인터를 사용했습니다. 이 방법이 단순성과 이식성을 제공하지만, 많은 경우 이는 성능을 약간 포기합니다.

다음 섹션들은 특수화, 메모리 보호, 그리고 하드웨어 고려를 다룹니다. 이 주제에 대한 분명한 논문을 위한 이 짧은 섹션들을 놓치지 마세요. 이 책 전체는 특정 CPU에서의 최적화를 다루는데, 오늘날 흔히 사용되는 CPU 제품군을 알아봅시다.

10.6.1 Specialization

Section 10.4에 보인 크기 재조정 가능한 해쉬 테이블은 임의의 타입의 키를 사용했습니다. 이는 상당한 유연성을 가능하게 해서 어떤 종류의 키든 사용될 수 있게 하지만, 포인터를 통한 함수 호출을 통해 상당한 오버헤드를 일으킵니다. 오늘날의 하드웨어는 이 오버헤드를 줄이기 위해 훌륭한 브랜치 예측 기법을 사용합니다만, 다른 한편 실제 세계의 소프트웨어는 오늘날의 커다란 하드웨어 브랜치 예측 테이블로도 담기지 못할 정도로 큰 경우가 많습니다. 이는 특히 포인터를 통한 호출의 경우 그런데, 브랜치 예측 하드웨어가 취해진/취해지지 않은 브랜치 정보에 더해 포인터를 기록해야만 하기 때문입니다.

이 오버헤드는 해쉬 테이블 구현을 특정 키 타입과 해쉬 함수로 특수화 시킴으로써 제거할 수 있는데, 예를 들면 C++ 템플릿을 사용하는 겁니다. 그렇게 하는 것은 page 185 의 Listing 10.9 에 보인 ht 구조체로부터 \rightarrow `ht_cmp()`, \rightarrow `ht_gethash()`, 그리고 \rightarrow `ht_getkey()` 함수 포인터들을 제거합니다. 이는 또한 이 포인터들을 통한 함수 호출들을 제거하여, 컴파일러가 그 결과 고정된 함수들을 인라인 시켜서 호출 명령의 오버헤드 뿐 아니라 인자 처리 오버헤드까지 제거할 수 있게 합니다.

또한, 이 크기 재조정 가능한 해쉬 테이블은 동시성 제어로부터 베킷 선택을 분리하는 API에 맞게 설계되었습니다. 이게 단일한 고문 테스트가 이 챕터의 모든 해쉬 테이블 구현을 테스트 할 수 있게 하지만, 이는 또한 많은 오퍼레이션들이 해쉬를 계산하고 가능한 크기 재조정 오퍼레이션들과 한번이 아니라 두번 상호작용해야만 하게 함을 의미합니다. 성능이 중요한 환경에서 `hashtab_lock_mod()` 함수는 또한 선택된 베킷으로의 참조를 반환해서 뒤따르는 `ht_get_bucket()` 호출을 제거할 겁니다.

Quick Quiz 10.16: `hashtorture.h` 코드는 `ht_get_bucket()` 기능을 포함하는 버전의 `hashtab_lock_mod()`를 갖도록 수정될 수 없나요?

Quick Quiz 10.17: 이 특수화가 얼마나 이득을 제공하나요? 그게 정말 가치있나요?

그렇다고는 하나, 제가 1970년대 초에 처음 프로그램을 배우기 시작했을 때 쓸 수 있었던 것들에 비교해 현대 하드웨어의 큰 이득 중 하나는 훨씬 적은 특수화가 필요하다는 겁니다. 이는 4 킬로바이트 주소 공간의 시대에 가능했던 것에 비해 훨씬 큰 생산성을 가능하게 합니다.

10.6.2 Bits and Bytes

이 챕터에서 이야기 된 해쉬 테이블들은 메모리를 아끼기 위한 시도를 거의 하지 않았습니다. 예를 들어, page 185 의 Listing 10.9 에 보인 ht 구조체의 \rightarrow `ht_idx` 필드는 항상 1 또는 0의 값만 갖는데도 메모리의 32비트를 사용합니다. 이는 예를 들어 \rightarrow `ht_resize_key` 필드의 비트를 하나 훔치는 것으로 제거될 수 있습니다. \rightarrow `ht_resize_key` 필드는 메모리의 모든 바이트 주소를 담을 수 있을 만큼 크며 ht_bucket 구조체는 1바이트보다 커서 \rightarrow `ht_resize_key` 필드는 여분의 비트 여려개를 가질 수 있으므로 동작 가능합니다.

이런 류의 비트 담기 속임수는 많이 복사되는 데이터 구조에서 자주 사용되는데 리눅스 커널의 page 구조체가 한 예입니다. 그러나, 이 크기 재조정 가능한 해쉬 테이블의 ht 구조체는 그렇게 많이 복사되지는 않습니다. 그보다는 ht_bucket 구조체에 집중해야 합니다.

ht_bucket 구조체의 크기를 줄이기 위한 두개의 주요 기회가 있습니다: (1) \rightarrow `htb_lock` 필드를 \rightarrow `htb_head` 포인터들 중 하나의 아래쪽 비트에 위치시키기와 (2) 필요한 포인터의 갯수 줄이기.

첫번째 기회를 위해선 리눅스 커널의 `include/linux/bit_spinlock.h` 헤더 파일에 있는 비트 스픈락을 사용할 수도 있습니다. 이것들은 리눅스 커널의 공간이 중요한 데이터 구조에서 사용됩니다만, 그것들의 단점도 있습니다:

1. 이것들은 전통적인 스픈락 기능들에 비해 상당히 느립니다.
2. 이것들은 리눅스 커널의 lockdep 데드락 탐지 도구에 [Cor06a] 협력하지 않습니다.
3. 이것들은 락 소유권을 기록하지 않아, 디버깅을 더 어렵하게 합니다.
4. 이것들은 -rt 커널의 우선순위 높이기와 협력하지 않아서, bit 스픈락을 잡을 땐 preemption 을 불능화 시켜야 하며 이는 real-time 응답시간을 나쁘게 만들 수 있습니다.

이런 단점에도 불구하고 bit 스픈락은 메모리가 중요한 때 매우 유용합니다.

두번째 기회의 한 측면은 Section 10.4에서 보인 크기 재조정 가능한 해쉬 테이블에서는 한 쌍의 집합이 필요했던 베킷 리스트 포인터들을 하나의 집합으로 줄인 크기 재조정 가능한 해쉬 테이블을 보인 Section 10.4.4에서 다뤘습니다. 또 다른 방법은 이 챕터에서 사용한 양방향 링크드 리스트 대신 단방향 링크드 베킷 리스트를 사용하는 것일 겁니다. 이 방법의 한가지 단점은 삭제가 추가적 오버헤드를 갖게 된다는 것인데, 제거하는 포인터를 나중의 제거를 위해 표시해 두거나 살제되는 원소를 위한 베킷 리스트를 탐색해야 할 겁니다.

요약하자면, 최소한의 메모리 오버헤드 대비 성능과 단순성의 트레이드오프가 있습니다. 다행히, 현대 시스템에서 가능한 상대적으로 큰 메모리는 메모리 오버헤드 대비 성능과 단순성을 우선시 할 수 있게 했습니다. 그러나, 오늘날의 큰 메모리 시스템에서 조차도² 메모리 오버헤드를 줄이기 위한 상당한 노력이 가끔은 필요합니다.

10.6.3 Hardware Considerations

오늘날의 컴퓨터들은 대부분 데이터를 CPU 와 메인 메모리 사이에서 32 바이트에서 256 바이트 사이의 고정 크기 블록으로 옮깁니다. 이 블록들은 캐시 라인이라 불리며, Section 3.2에서 다뤘듯 높은 성능과 확장성을

² 수기가바이트 메모리를 갖는 스마트폰 있죠, 여러분?

Listing 10.14: Alignment for 64-Byte Cache Lines

```

1 struct hash_elem {
2     struct ht_elem e;
3     long __attribute__((aligned(64))) counter;
4 };

```

위해 무척 중요합니다. 성능과 확장성을 모두 죽이는 한가지 오래된 방법은 호환되지 않는 변수들을 같은 캐쉬라인에 두는 겁니다. 예를 들어, 크기 재조정 가능한 해쉬 테이블의 데이터 원소 하나가 자주 증가되는 카운터와 같은 캐쉬라인에 `ht_elem` 구조체를 두었다고 생각해 봅시다. 이 같은 카운터 값 증가는 이 캐쉬라인이 이 값 증가를 하는 CPU에게 존재하지만 다른 곳에는 존재하지 않게 만들 겁니다. 만약 다른 CPU들이 이 원소를 담는 해쉬 버킷 리스트를 순회하려 하면, 비싼 캐쉬 미스를 일으켜서 성능과 확장성을 모두 떨어뜨릴 겁니다.

64 바이트 캐쉬 라인의 시스템에서 이 문제를 해결하는 한가지 방법이 Listing 10.14에 보여 있습니다. 여기서 GCC의 `aligned` 지시어가 `->counter`와 `ht_elem` 구조체를 다른 캐쉬 라인에 위치하게 강제합니다. 이는 CPU들이 같은 카운터 값 증가에도 불구하고 해쉬 버킷 리스트를 완전한 속도로 순회할 수 있게 합니다.

물론, 이는 “우린 캐쉬 라인이 64 바이트 크기란 걸 어떻게 하나?”라는 질문을 떠올리게 합니다. 리눅스 시스템에서, 이 정보는 `/sys/devices/systems/cpu/cpu*/cache/` 디렉토리에서 얻을 수 있으며, 설치 프로세스가 어플리케이션을 이 시스템의 하드웨어 구조에 맞춰 다시 빌드되게 할 수도 있습니다. 그러나, 여러분의 어플리케이션이 리눅스가 아닌 시스템에서도 돌아가게 하고 싶다면 좀 더 어려울 수 있습니다. 더 나아가서, 여러분이 리눅스에서만 수행하려 해도, 그런 스스로를 수정하는 설치는 겸증의 문제를 갖습니다. 예를 들어, 32 바이트 캐쉬라인에서는 잘 동작하지만, 거짓 공유로 인해 64 바이트 캐쉬라인에서는 성능이 떨어질 수도 있습니다.

다행히, 1995년 어느 논문에 [GKPS95] 모아진, 실전에서 합리적 수준으로 잘 동작하는 경험적 법칙이 있습니다.³ 첫번째 그룹의 규칙들은 구조체들을 캐쉬 구조에 맞춰 재배열 하는 것에 관여됩니다:

1. 읽기가 대부분인 데이터를 자주 업데이트 되는 데이터와 떨어뜨려 놓으세요. 예를 들어, 읽기가 대부분인 데이터를 구조체의 앞부분에 두고 자주 업데이트 되는 데이터는 끝부분에 두세요. 드물게 접근되는 데이터를 그 사이에 두세요.
2. 그 구조체가 필드들의 그룹들을 가지고 있고 각 그룹이 독립적 코드 경로에 의해 업데이트 된다면, 이

³ 이 규칙들 중 여럿을 Orran Krieger의 허락 하에 여기 바꿔 쓰고 확장했습니다.

그룹들을 서로로부터 분리시키세요. 다시 말하지만, 자주 접근되지 않는 데이터를 이 그룹들 사이에 두는게 도움이 됩니다. 어떤 경우에는 그런 각각의 그룹을 원래 구조체에 의해 참조되는 분리된 구조체에 두는 것 역시 말이 될 수도 있습니다.

3. 가능하다면 업데이트가 주로 되는 데이터는 CPU, 쓰레드, 또는 태스크와 연관시키십시오. Chapter 5의 카운터 구현에서 우린 이 경험적 법칙의 매우 효과적인 예를 여럿 보았습니다.
4. 한 단계 더 나아가서, 여러분의 데이터를 Chapter 8에서 이야기한 것처럼 CPU 별, 쓰레드별, 또는 태스크별로 분리시키세요.

자동화된 기록 기반 구조체 필드 재정렬을 위한 연구가 몇몇 있었습니다 [GDZE10]. 이는 멀티쓰레드 소프트웨어에서 훌륭한 성능과 확장성을 위해 필요한 더 고통스러운 작업들을 더 쉽게 만들어줄 수도 있습니다. 락을 다루는 추가적 경험적 법칙들이 있습니다:

1. 자주 수정되는 데이터를 보호하며 경쟁도 심한 락이 있다면 다음 방법들 중 하나를 사용하세요:
 - (a) 그 락을 그것이 보호하는 데이터와 다른 캐쉬 라인에 두세요.
 - (b) 경쟁이 심한 경우를 위해 조정된, queued lock 같은 락을 사용하세요.
 - (c) 락 경쟁을 줄이기 위해 설계를 다시 하세요. (이 방법이 최고입니다만, 항상 간단하지는 않습니다.)
2. 경쟁이 심하지 않은 락은 그것이 보호하는 데이터와 같은 캐쉬 라인에 두세요.
3. 읽기가 대부분인 데이터를 해저드 포인터, RCU, 또는, 긴 크리티컬 섹션을 갖는 경우, reader-writer 락으로 보호하세요.

물론, 이것들은 절대적 법칙이 아니라 경험적 규칙입니다. 주어진 상황에 무엇이 가장 적합한지 알기 위해선 실험이 필요합니다.

10.7 Summary

There's only one thing more painful than learning from experience, and that is not learning from experience.

Archibald MacLeish

이 챕터는 완전한 파티셔닝이 불가능한 크기 재조정한 해쉬 테이블을 포함해서 해쉬 테이블을 주로 보았습니다.

다. Section 10.5 은 해쉬 테이블 외의 데이터 구조에 대해 짧게 들여다 보았습니다. 그러나, 이 해쉬 테이블 분석은 고성능 확장형 데이터 액세스를 둘러싼 많은 문제들에 대한 훌륭한 소개가 될텐데, 그런 문제들로는 다음과 같은 것들이 포함됩니다:

1. 완전히 파티셔닝된 데이터 구조는 예를 들면 단일 소켓 시스템과 같은 작은 시스템에서 잘 동작합니다.
2. 더 큰 시스템은 완전한 파티셔닝에 더해 참조의 지역성을 필요로 합니다.
3. 해저드 포인터와 RCU 같은 읽기가 대부분인 경우를 위한 기법들은 읽기가 대부분인 워크로드에 대해 좋은 참조 지역성을 제공하며 따라서 큰 시스템에서도 훌륭한 성능과 확장성을 제공합니다.
4. 읽기가 대부분인 경우를 위한 기법들은 또한 크기 재조정 가능한 해쉬 테이블과 같은 완전한 파티셔닝이 불가능한 일부 데이터 구조에서도 잘 동작합니다.
5. 커다란 데이터 구조들은 CPU 캐시를 넘치게 해서 성능과 확장성을 떨어뜨릴 수 있습니다.
6. 데이터 구조를 특정 워크로드에 특수화 시킴으로써 추가적인 성능과 확장성을 얻을 수 있는데, 예를 들어 범용 키를 32-bit 정수로 만드는 겁니다.
7. 이식성과 극단의 성능을 위한 요구사항들은 종종 서로 부딪히지만 이 두 요구사항 집합 사이에서 좋은 균형을 맞출 수 있는 일부 데이터 구조 배치 기법들이 존재합니다.

그렇다고는 하나, 성능과 확장성은 안정성 없이는 쓸모가 없으므로, 다음 챕터는 검증을 다룹니다.

If it is not tested, it doesn't work.

Unknown

Chapter 11

Validation

전 바로 동작하는 병렬 프로그램을 일부 만들어 봤지만, 그건 지난 30년간 많은 수의 병렬 프로그램을 만들었기 때문일 뿐입니다. 그리고 정말로 처음부터 동작한 게 아니라 그렇다고 생각할 뿐이어서 절 바보로 만든 병렬 프로그램을 훨씬 더 많이 만들었습니다.

따라서 저는 제 병렬 프로그램을 검증할 필요가 있습니다. 검증 아래의 기본 트릭은 컴퓨터는 뭐가 잘못되었는지 안다는 것을 깨닫는 겁니다. 그러므로 그게 여러분에게 말을 해주게 하는게 여러분의 일입니다. 그런 이유로 이 챕터는 기계를 심문하는 법에 대한 짧은 강의라 생각할 수 있겠습니다. 하지만 여러분은 좋은 경찰/나쁜 경찰 루틴은 놔둘 수 있습니다. 이 챕터는 그보다 훨씬 더 정교하고 효과적인 방법들을 다루는데, 대부분의 컴퓨터가 적어도 우리가 아는 한 나쁜 경찰 대신 좋은 경찰에게 말을 할 수 없다는 걸 두고 생각하면 특히 그렇습니다.

더 긴 강의는 오래되었지만 가치있는 것 [Mye79] 은 물론이고 검증에 대한 많은 최신의 교재에서 찾을 수 있을 겁니다. 검증은 모든 형태의 소프트웨어에 걸쳐 굉장히 중요한 주제이며, 그것 자체만으로도 상당한 공부의 가치가 있습니다. 그러나, 이 책은 기본적으로 동시성에 대한 것이므로 이 챕터는 이 굉장히 중요한 주제에 대해 걸 halk기보다 조금만 더 들어가 봅니다.

Section 11.1 은 디버깅의 철학을 소개합니다. Section 11.2 는 기록하기를 이야기 하고, Section 11.3 는 보장하기를 이야기 하며, Section 11.4 는 정적 분석을 다룹니다. Section 11.5 는 10,000 개나 되는 눈알이 여러분의 코드를 들여다 보는 것이 아닌 경우에 도움이 될 수 있는 코드 리뷰의 일반적이지 않은 방법을 이야기 합니다. Section 11.6 는 병렬 소프트웨어의 확률의 사용을 알아봅니다. 성능과 확장성은 병렬 프로그래밍의 첫번째 필요이므로, Section 11.7 가 그 주제를 다룹니다. 마지막으로, Section 11.8 는 간단한 요약과 피해야 할 통계적 함정들을 나열합니다.

하지만 세개의 최고의 디버깅 도구는 요구사항에 대한 이해, 탄탄한 설계, 그리고 밤에 잘 자는 것임을 잊지 마세요!

11.1 Introduction

If debugging is the process of removing software bugs, then programming must be the process of putting them in.

Edsger W. Dijkstra

Section 11.1.1 는 버그의 근원지를 이야기하고, Section 11.1.2 는 소프트웨어를 검증할 때 필요한 마음가짐을 이야기 합니다. Section 11.1.3 는 언제 여러분이 검증을 시작해야 하는지 이야기하며, Section 11.1.4 는 놀랍도록 효과적인 오픈소스 코드 리뷰 방법론과 공동체 기반 테스팅을 다룹니다.

11.1.1 Where Do Bugs Come From?

버그는 개발자로부터 옵니다. 기본적인 문제는 사람의 뇌는 컴퓨터 소프트웨어를 염두에 두고 진화하지 않았다는 겁니다. 그 대신, 사람의 뇌는 다른 사람의 뇌와 동물의 뇌와의 콘서트를 통해 진화되었습니다. 이 역사 탓에, 다음의 세가지 컴퓨터의 특성은 종종 사람의 직관에는 충격으로 다가오곤 합니다.

1. 인공지능의 제단에서의 거대한 희생에도 불구하고, 컴퓨터는 직관이 없습니다.
2. 컴퓨터는 사용자의 의도를 이해하지 못하는데, 좀 더 정식적으로 말하자면, 컴퓨터는 일반적으로 마음의 이론이 없습니다.
3. 컴퓨터는 파편화된 계획을 가지고는 유용한 무언가 할 수 없고, 모든 가능한 경우의 수에 대한 모든 자세한 사항을 필요로 합니다.

앞의 두가지는 논란이 되지 않을 텐데, 아마도 가장 유명한 Clippy 와 Microsoft Bob 과 같은 많은 수의 실패한 제품들이 있기 때문입니다. 사용자들에게 사람처럼 보이려 시도함으로써, 이 두개의 제품들은 그것들이 가질 수 없는 상식과 마음의 이론에 대한 기대를 일으켰습니다. 이제는 스마트폰에서 사용 가능한 소프트웨어 비서들은 더 나을 것으로 보이지만, 2021년 기준으로 그 감상은 여렷이 훈재되어 있습니다. 그렇다고는 하나, 모든 개발자들은 여전히 오래된 방법으로 개발을 하고 있습니다: 그 비서들이 최종 사용자에게 도움이 될 수 있으나, 그 개발자 본인에게는 충분히 그렇지 못합니다.

사람의 파편화된 계획에 대한 사랑은 더 많은 설명을 필요로 하는데, 그것이 고전적인 양날의 검임을 생각하면 특히 그렇습니다. 이 파편화된 계획에 대한 사랑은 계획을 이끄는 사람이 (1) 상식과 (2) 이 계획을 이끄는데 필요한 것들과 의도에 대한 훌륭한 이해를 가지고 있을 거라는 가정 때문인 것 같습니다. 두번재 가정은 특히나 계획을 짜는 사람과 그 계획을 수행하는 사람이 같은 사람이라는 상식에서 만들어집니다: 이 경우, 그 계획은 장애물이 나타날 때마다 무의식적으로 고쳐질 것인데, 특히 그 사람이 해당 문제에 대한 훌륭한 이해를 가지고 있을 때 그렇습니다. 실제로, 파편화된 계획에 대한 사랑은 인간에게 잘 동작했는데, 부분적으로는 계획될 수 없는 계획을 시도하다가 굽어주는 대신 음식을 찾을 수 있는 무작위적 행동을 취하는게 나았기 때문입니다. 그러나, 우리 모두가 전문가인 일상의 파편화된 계획의 유용성은 컴퓨터에 저장된 프로그램의 미래의 유용성을 보장하지 않습니다.

더 나아가, 파편화된 계획을 따라가야 할 필요성은 인간의 정신에도 중요한 영향을 끼쳤는데, 인간의 역사상에서는 삶이 종종 어렵고 위험했다는 사실 때문입니다. 날카로운 이빨과 발톱의 폭력을 마주할 가능성이 높은 파편화된 계획을 수행하는 것이 미친것과 같은 수준의 긍정성을—실제로 대부분의 인간에게 존재하는 수준의 긍정성—필요로 함은 놀랍지 않을 겁니다. 이런 미친 수준의 긍정성은 프로그래밍 능력에 대한 자기 평가에도 확장되는데, 코드 인터뷰 능력의 효과(그리고 논란)이 그 증거가 됩니다 [Bra07]. 사실, 미친 정도보다 덜한 수준의 긍정성을 갖는 사람들에 대한 의학적 용어는 “의학적으로 우울함”입니다. 그런 사람들은 보통 그들의 일상에 상당한 어려움을 가져서 아마도 반직관적인 미친 수준의 긍정성의 평범하고 건강한 삶에의 중요성을 강조합니다. 더 나아가, 여러분이 미친듯 긍정적이지 않다면, 여러분은 어렵지만 가치있는 프로젝트를 시작할 가능성도 낮습니다.¹

¹ 이 경험적 규칙에는 유명한 예외가 일부 있습니다. 어떤 사람들은 어렵거나 위험한 프로젝트를 그들의 우울증에 대한 최소한 임시적 탈출구로써 취하기도 합니다. 다른 사람들은 잃을 게 없습니다: 그 프로젝트는 글자 그대로 삶과 죽음의 문제입니다.

Quick Quiz 11.1: 컴퓨팅에서 파편화된 계획을 진행해야 할 때는 언제죠?

중요한 특수 경우 중 하나는 가치있지만 그것을 구현하는데 드는 시간을 정당화 할 정도로 가치있지는 않은 프로젝트입니다. 이 특수한 경우는 상당히 흔하며 그 초기 증상 중 하나는 결정권자가 그 프로젝트를 구현하는데 충분히 투자하고자 하지 않음입니다. 이에 대한 자연적인 반응은 개발자가 그 프로젝트를 시작하는데 혀락을 받을 수 있도록 비현실적으로 긍정적인 예상치를 만들어내는 것입니다. 만약 그 조직이 충분히 강하고 그 조직의 결정권자가 영향력이 충분하지 않다면, 스케줄이 지켜지지 못하고 부담이 커지는 결과가 나긴 하지만 그 프로젝트는 성공될 겁니다. 그러나, 그 조직이 충분히 강하지 못하고 그 추정치는 쓰레기였음이 분명해지자마자 결정권자가 그 프로젝트를 취소하는데 실패한다면, 그 프로젝트는 조직을 죽일 수도 있습니다. 이는 그 프로젝트를 택하는 다른 조직이 생겨서 그걸 완성하거나 취소하거나 그것에 의해 죽는 결과를 낳을 수도 있습니다. 그런 프로젝트는 여러 조직을 죽인 후에야 성공될 수도 있습니다. 사람들은 그런 연쇄 조직 살인마 프로젝트를 결국 성공시킨 그 조직이 충분한 정도의 겸손성을 가져서 다음 프로젝트에 의해 죽지 않길 바라야만 할 겁니다.

Quick Quiz 11.2: 누가 조직을 신경씁니까? 어쨌건, 중요한 건 그 프로젝트라구요!

미친 수준의 긍정성의 중요한 점은 아마도 그게 버그의 (그리고 어쩌면 조직의 실패의) 핵심 근원지일 수도 있다는 겁니다. 따라서 질문은 “커다란 프로젝트를 시작하는데 필요한 수준의 긍정성을 유지하면서도 버그를 낮은 수준으로 유지할 수 있을까?”입니다. 다음 섹션이 이 수수께끼를 풀어봅니다.

11.1.2 Required Mindset

어떤 검증을 하려 할 때에는 다음의 정의들을 명심하십시오:

1. 버그가 없는 프로그램은 사소한 프로그램들 뿐이다.
2. 안정적인 프로그램은 알려진 버그가 없을 뿐이다.

이 정의들로부터 모든 안전하고 사소하지 않은 프로그램은 여러분이 알지 못하는 버그를 최소 하나는 가지고 있다는 결론이 논리적으로 도출됩니다. 따라서, 사소하지 않은 프로그램에 취해졌으나 어떤 버그도 찾지 못한 모든 검증은 그 자체로 실패입니다. 따라서 좋은 검증은 파멸의 수행입니다. 이 말은 여러분이 무언가를

부수는 걸 즐기는 성격이라면 검증은 여러분을 위한 일이 될 것을 의미합니다.

Quick Quiz 11.3: 여러분이 다음처럼 보이는 time 커맨드의 출력 결과를 처리하는 스크립트를 짠다고 생각해 봅시다:

real	0m0.132s
user	0m0.040s
sys	0m0.008s

그 스크립트는 입력에 애러가 있는지, 그리고 애러가 있는 time 결과를 받았다면 적절한 진단 결과를 낼 수 있도록 검사를 해야 합니다. 싱글쓰레드 프로그램에 의해 생성되는 time 결과를 사용하기 위해 여러분은 이 프로그램의 테스트를 위해 어떤 테스트 입력을 제공해야 할까요?

■ 하지만 아마도 여러분은 매번 처음부터 완벽하게 코드를 짜는 슈퍼 프로그래머일지도 모릅니다. 그렇다면, 축하합니다! 이 챕터를 건너뛰어도 좋습니다. 하지만 여러분이 제 회의론을 용서해 주길 정말 바랍니다. 보세요, 저는 처음부터 완벽한 코드를 썼다고 주장하는 사람을 너무 많이 만났는데, 앞서 이야기한 공정성과 지나친 자신감을 생각하면 너무 놀랍지는 않은 일입니다. 그리고 여러분이 정말 슈퍼 프로그래머라 하더라도, 여러분은 덜 치명적인 작업을 디버깅 하고 있는 것일 뿐일지도 모릅니다.

그 외의 우리들을 위한 한가지 방법은 우리의 미친 공정성과(물론, 난 그걸 프로그램으로 짤 수 있어!) 상당한 부정성(동작하는 것 같아, 하지만 난 저기 어딘가 버그가 숨어 있어야 한다는 걸 알았어!) 사이에서 번갈아 가는 것입니다. 여러분이 무언가를 부수길 좋아한다면 도움이 됩니다. 그렇지 않다면, 또는 무언가를 부수기에 대한 여러분의 즐거움이 다른 사람의 것을 부수기로 제한된다면, 여러분의 코드를 부수길 좋아하는 사람을 찾아서 그들이 그려길 도우십시오.

또 다른 도움이 되는 마음가짐은 다른 사람들이 여러분의 코드의 버그를 찾아내는 걸 즐오하는 겁니다. 이 즐오는 여러분이 그 버그를 찾는 사람이 될 가능성을 높이기 위해 여러분의 코드를 모든 방법으로 고문할 동기가 될 겁니다. 그저 이 즐오를 여러분의 코드에서 버그를 찾는 누군가에게 진심으로 감사를 전할 수 있을 정도로 늦추기만 하세요! 어쨌건, 그렇게 함으로써, 그들은 여러분이 그걸 쫓아가는데 생기는 문제를 줄여준 것이며, 아마도 그들은 여러분의 코드를 훑어보느라 상당한 개인적 지출을 했을 겁니다.

또 다른 도움이 되는 마음가세는 학습된 회의론입니다. 보세요, 여러분이 그 코드를 이해한다고 빙는 것은 여러분이 그것으로부터 무엇도 배울 수 없음을 의미합니다. 아, 하지만 여러분은 여러분이 그걸 짜거나 리뷰했으므로 완전히 이해하고 있다는 걸 안다구요? 미안하지



Figure 11.1: Validation and the Geneva Convention

만 버그의 존재는 여러분의 이해가 최소한 부분적으로는 잘못되었음을 의미합니다. 한가지 치료법은 여러분이 진실이라 생각하는 걸 써내려보고 Sections 11.2–11.5에서 이야기된 것처럼 이 지식을 재차 확인하는 겁니다. 객관적 진실은 항상 여러분이 알고 있다고 생각하는 것을 뛰어넘을 수 있습니다.

마지막 마음가세는 누군가의 삶이 여러분의 코드가 올바른지에 의존적일 수 있음을 고려하는 겁니다. 이를 보는 한가지 방법은 일관적으로 좋은 것을 만드는 것은 일어날 수 있는 많은 나쁜 일에 그걸 방지하거나 그 나쁜 일들을 처리하는 데 노력하는 눈을 가지고 집중하는 것을 필요로 합니다.² 이 나쁜 일에 대한 가능성은 여러분이 버그를 찾기 위해 여러분의 코드를 고문하는 동기가 되어줄 수도 있습니다.

이 다양한 마음가짐은 여러 사람이 다양한 수준의 공정성과 다른 마음가짐을 가지고 프로젝트에 기여할 가능성을 높여줍니다. 제대로 조직화 된다면 이는 잘 동작할 것입니다.

어떤 사람들은 격렬한 검증을 Figure 11.1에 보인 것처럼 고문의 형태로 생각할 수도 있겠습니다.³ 그런 사람들은 Tux 만화와는 별개로, Figure 11.2에 보인 것처럼 그들이 정말로 애니메이션화 될 수 없는 객체를 고문하고 있음을 상기하는 게 좋을 수도 있습니다. 그렇지 않은 사람들은 자신의 코드를 고문하는데 실패하면 그 코드가 자신을 고문할 수 있음을 알아 두십시오!

² 이 철학을 더 알아보려면, Chris Hadfield의 “An Astronaut’s Guide to Life on Earth”라는 제목의 훌륭한 책의 “The Power of Negative Thinking”이라는 챕터를 보세요.

³ 우리 사이의 냉소는 이 사람들이 검증이 그들이 고쳐야 하게 될 버그를 찾을지 두려워할지도 모르겠다는 질문을 낼지도 모르겠습니다.

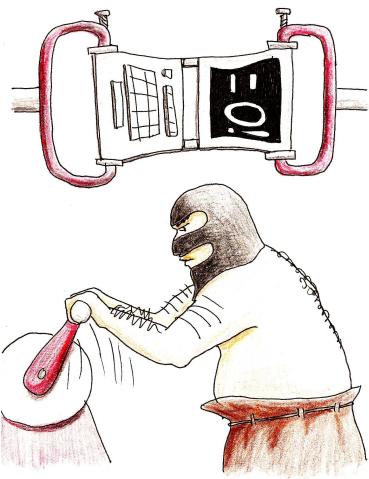


Figure 11.2: Rationalizing Validation

그러나, 이는 프로젝트의 생애 중 구체적으로 언제 검증을 시작해야 하는지 질문을 던지는데, 다음 섹션에서 이 주제를 다룹니다.

11.1.3 When Should Validation Start?

검증은 정확히 그 프로젝트가 시작될 때 시작되어야 합니다.

이를 자세히 보기 위해, 버그를 추적하는 것은 작은 것보다 큰 프로그램에서 훨씬 더 어려움을 생각하세요. 따라서, 버그를 추적하는데 필요한 시간과 노력을 최소화하기 위해, 여러분은 코드의 작은 단위를 테스트해야 합니다. 여러분이 이 방법으로 모든 버그를 찾지는 못하겠지만, 상당한 부분을 찾을 것이며, 그게 버그를 찾고 그걸 고치기에 훨씬 쉬울 겁니다. 이 단계에서의 테스트는 또한 여러분에게 여러분의 전반적 설계에 있는 큰 결함을 알려줘서 설계로 인해 망가진 코드를 짜는데 낭비하는 시간을 줄이게 도와줄 겁니다.

하지만 여러분의 설계를 검증하기 전에 왜 코드를 기다립니까?⁴ 바라건대 Chapter 3 와 4 를 읽는 것이 안타깝게 흔한 설계 결함을 막는데 필요한 정보를 여러분에게 주었을 겁니다만, 여러분의 설계를 어떤 동료와 이야기하거나 단순히 그걸 써보는 것만으로도 추가적인 결함을 없애는 걸 도와줄 겁니다.

하지만, 여러분이 설계가 있을 때까지 검증을 대기하는 것은 너무 오래 기다리는 것인 경우가 너무 많습니다. 여러분의 자연적인 수준의 궁정성이 여러분으로 하여금 여러분이 요구사항을 완전히 이해하기도 전에 설계를 시작하게 하지는 않았을까요? 이 질문에 대한

⁴ 고전적인 “일단 코드를 짜야만 한다. 그러면 생각할 동기를 갖는다”라는 말에도 불구하고요.

대답은 거의 항상 “그렇다”입니다. 결함 있는 요구사항을 막는 한가지 좋은 방법은 여러분의 사용자를 아는 겁니다. 그들을 정말 잘 대접하려면, 여러분은 그들과 함께 살아봐야 합니다.

Quick Quiz 11.4: 제가 코딩을 시작하기도 전부터 이 검증 일을 하라고 말하는 거예요??? 그건 시작도 안 하기에 아주 좋은 방법처럼 들리는데요!!!

첫번째 종류 프로젝트는 종종 급속 프로토타이핑이나 애자일과 같은 다른 방법론을 사용합니다. 여기서 이를 프로토타이핑의 주요 목표는 올바른 구현을 만드는게 아니라 그 프로젝트의 요구사항을 배우는 겁니다. 하지만 이는 여러분이 검증을 하지 않음을 의미하지 않습니다; 그건 여러분이 그걸 다르게 접근함을 의미합니다.

그런 방법 중 하나는 다원주의 관점을 취하는데, 검증 도구가 그 문제를 해결하는데 맞지 않는 코드를 제거하게 합니다. 이 관점에서, 적극적 검증 도구는 여러분의 소프트웨어의 상요 가능성을 위해 필수적입니다. 그러나, 이 방법을 그 논리적 결론을 위해 취하는 것은 상당히 결손한 것인데, 이는 우리 개발자들이 우리의 주의깊게 만든 변경이 다원주의 관점에서는 무작위적 기형이라는 것을 인정해야 하기 때문입니다. 다른 한편으로, 이 결론은 수정의 7퍼센트는 최소 하나의 버그를 만든다는 오랜 경험으로 [BJ12] 지지됩니다.

여러분의 검증 도구는 얼마나 적극적이어야 할까요? 그것이 찾는 버그가 여러분의 소프트웨어 설계의 토대를 위협하는게 아니라면, 그건 아직 충분히 적극적이지 않습니다. 어쨌건, 여러분의 설계는 여러분의 코드가 그런만큼 버그에 취약하며, 여러분이 여러분의 설계 내의 버그를 빨리 찾아서 고칠수록 그 설계 버그를 코딩하는데 낭비하는 시간을 줄일 수 있습니다.

Quick Quiz 11.5: 소프트웨어의 정확성을 정말로 테스트 할 수 있다고 말하는 거예요??? 그게 불가능하다는 건 모두가 안다구요!!!

이 조언은 첫번째 종류의 프로젝트에 적용됨을 다시 반복할 가치가 있겠습니다. 만약 여러분이 그대신 잘 알려진 영역의 프로젝트를 한다면 이전의 경험으로부터 배우기를 거부하는 건 상당히 바보스러운 일이 될 겁니다. 하지만 여러분은 여전히 그 프로젝트의 시작 단계부터 검증을 시작해야 합니다만, 다른 사람들의 힘들게 얻어진 요구사항과 함정들에 안내를 받기 바랍니다.

똑같이 중요한 질문은 “언제 검증이 멈춰야 하는가?”입니다. 최고의 답은 “마지막 변경으로부터 얼마간 시간이 지난후”입니다. 모든 변경은 버그를 만들 잠재적 가능성이 있으며, 따라서 모든 변경이 검증되어야 합니다. 더 나아가, 검증 개발은 프로젝트의 생애 전체에 걸쳐 지속되어야 합니다. 어쨌건, 앞의 다원주의 특성은

버그가 여러분의 검증 도구에 적응할 것을 암시합니다. 따라서, 여러분이 자신의 검증 도구를 계속해서 개선하지 않으면, 여러분의 프로젝트는 자연적으로 검증 도구에 면역이 있는 버그 무리를 쌓을 겁니다.

하지만 삶은 트레이드오프이고, 검증 도구에 투자되는 모든 시간은 프로젝트 자체를 개선하는데에는 직접적으로 투자될 수 없습니다. 이런 종류의 선택은 결코 쉽지 않으며, 검증에 지나치게 투자하는 것은 부족하게 투자하는 것만큼이나 문제가 됩니다. 하지만 이는 삶이 쉽지 않음을 알리는 또 하나의 것일 뿐입니다.

이제 우리는 프로젝트의 시작 시점에 (더 빨리 할 수 없다면!) 검증을 시작해야 하며 검증과 검증 개발 모두 프로젝트의 생애 동안 지속되어야 함에 동의했으니 다음 섹션들은 그 가치가 증명된 여러 검증 기법들과 방법들을 다룹니다.

11.1.4 The Open Source Way

오픈소스 프로그래밍 방법론은 상당히 효과적이고 상당한 코드 리뷰와 테스트를 포함하는 것으로 증명되었습니다.

저는 개인적으로 오픈소스 커뮤니티의 상당한 코드 리뷰의 효과에 대해 증언할 수 있습니다. 리눅스 커널을 위한 제 첫번째 패치들은 한 노드의 메모리에 매핑해 둔 파일에 다른 노드가 쓰기를 할 수 있는 분산 파일시스템에 관련되어 있었습니다. 이 경우, 쓰기 오퍼레이션 동안 파일시스템의 일관성을 유지하기 위해 영향 받는 페이지를 그것의 메모리 매핑으로부터 무효화 시킬 필요가 있습니다. 저는 한 패치에서 첫번째 시도를 했으며, 오픈소스 격언인 “빨리 보내고 자주 보내라”에 따라, 그 패치를 보냈습니다. 그리고 나서 저는 제가 그걸 어떻게 테스트 할지 생각했습니다.

하지만 저는 심지어 제가 전반적인 테스트 전략을 결정하기도 전에 제 패치에 대한 일부 버그를 지적하는 답장을 받았습니다. 저는 그 버그를 고치고 패치를 다시 보낸 후 저의 테스트 전략을 생각하기로 돌아왔습니다. 하지만, 제가 어떤 테스트 코드를 작성할 기회를 갖기 전에 저는 대시 보낸 패치에 대해 더 많은 버그를 지적하는 답장을 받았습니다. 이 과정은 그 자체로 여러번 반복되었으며, 저는 제가 이 패치를 정말로 테스트할 기회가 있긴 할지 확신할 수 없었습니다.

이 경험은 유명한 오픈소스 격언을 상기시켰습니다: 충분한 눈알이 있다면 모든 버그는 간단해진다 [Ray99].

그러나, 여러분이 어떤 코드나 패치를 보내기 전에 약간의 질문을 던져볼 가치가 있습니다:

1. 그 수많은 눈알 중 얼마나 많은 눈알이 정말로 여러분의 코드를 볼까요?
2. 그 중 얼만큼이 여러분의 버그를 정말 찾기 충분할 정도로 경험 많고 현명 할까요?

3. 그것들은 구체적으로 언제 코드를 보기 시작할까요?

저는 운이 좋았습니다: 제 패치로 제공된 기능을 원한 사람이 누군가 있었는데, 그 사람은 분산 파일시스템에 오랜 경험이 있었고, 거의 곧바로 제 패치를 들여다 보았습니다. 아무도 제 패치를 들여다보지 않았다면, 리뷰도 없었을 것이고 따라서 그 버그들 중 어떤 것도 발견되지 않았을 겁니다. 제 패치를 들여다 본 살마들이 분산 파일시스템에 대한 경험이 없었다면, 그들이 그 모드 버그를 찾았을 가능성은 낮습니다. 그들이 몇달 또는 심지어 몇년간 코드를 보기를 미뤘다면, 전 그 패치가 어떻게 동작할 것으로 여겨졌는지 잊어서 그걸 고치기를 훨씬 어렵게 했을 가능성이 높습니다.

그러나, 우린 오픈소스 개발의 두번째 주의인 상당한 테스트를 잊지 말아야 합니다. 예를 들어, 엄청나게 많은 사람들이 리눅스 커널을 테스트 합니다. 어떤 사람들은 패치들을 제출되자마자, 아마 여러분의 것을 포함해, 테스트합니다. 또 다른 사람들은 -next 트리를 테스트하는데, 이는 도움이 되지만 여러분이 그 패치를 작성한 시점으로부터 그것들이 -next 트리에 들어가기까지는 몇주에서 몇달까지도 지연이 있을 수 있기 때문에 그 패치는 여러분의 마음 속에서 그렇게 싱싱하지는 않을 겁니다. 또 다른 어떤 사람들은 메인테이너의 트리들을 테스트하는데, 종종 비슷한 시간 지연이 있곤 합니다.

약간의 사람들은 코드가 메인라인 또는 마스터 소스 트리에 (리눅스 커널의 경우 Linus 의 트리) 들어가기 전까지 테스트 하지 않습니다. 여러분의 메인테이너가 여러분의 패치를 테스트 되기 전까지 받아들이지 않는다면, 이는 여러분에게 데드락 상황을 일으킵니다: 여러분의 패치는 테스트 되기 전까지 받아들여지지 않는데 그것은 받아들여지기 전까지 테스트 되지 않을 겁니다. 그러나, 많은 사람들과 단체가 코드가 리눅스 배포판에 들어가기 전까지는 테스트 하지 않음을 놓고 보면 메인라인 코드를 테스트 하는 사람들은 여전히 상대적으로 적극적입니다.

그리고 어떤 사람이 여러분의 패치를 테스트 한다고 해도, 그들이 여러분의 버그를 찾는데 필요한 하드웨어와 소프트웨어 구성과 워크로드를 수행할지는 보장되지 않습니다.

따라서, 오픈소스 프로젝트를 위한 코드를 작성할 때 라 해도 여러분은 여러분 자신의 테스트 도구를 개발하고 수행할 준비를 해야 합니다. 테스트 개발은 제대로 가치를 평가받지 못했지만 가치있는 기술이므로 사용 가능한 존재하는 테스트 도구들의 모든 이점을 취하십시오. 테스트 개발의 중요성 탓에 우린 그에 대한 더 많은 이야기를 그 주제 전용의 책에 남겨야 하겠습니다. 따라서 다음 섹션들은 여러분이 이미 좋은 테스트 도구를 가지고 있다는 가정 하에 여러분의 코드의 버그를 찾는 방법에 대해 이야기 합니다.

11.2 Tracing

The machine knows what is wrong. Make it tell you.

Unknown

모든게 실패하면, `printk()` 를 추가하세요! 또는, 여러분이 사용자 모드 C 언어 어플리케이션을 작업하고 있다면, `printf()` 를요.

이유는 간단합니다: 어떻게 수행이 코드의 특정 포인트에 이르렀는지 모르겠다면, 뭐가 일어났는지 알기 위해 그 코드의 앞부분에 print 문을 뿌려주세요. (사용자 어플리케이션을 위해) `gdb` 또는 (리눅스 커널 디버깅을 위해) `kgdb` 를 사용해서 비슷한 효과를 더 편리하고 유연하게 얻을 수 있습니다. 훨씬 더 정교한 도구들이 존재하는데, 최근의 것들은 실패 지점으로부터 시간을 되돌리는 기능도 제공합니다.

이런 간단한 테스트 도구들은 모두 가치 있는데, 대부분의 시스템이 64K 이상의 메모리와 4 MHz 보다 빠른 CPU 를 갖는 지금은 더욱 그렇습니다. 이런 도구들을 위한 많은 서적이 있으므로, 이 챕터는 약간만 더 이야기하겠습니다.

그러나, 이런 도구들은 모두 여러분의 fastpath 에서 뭐가 잘못되고 있는지 보려고 할 때 심각한 단점을 보이는데, 이 도구들은 종종 지나친 오버헤드를 갖는다는 것입니다. 이런 목적을 위한 특수한 추적 도구가 있는데, 실행 시간 데이터 수집의 오버헤드를 최소화하기 위해 데이터 소유권 기법을 (Chapter 8를 보세요) 사용합니다. 리눅스 커널에서의 한 가지 예는 “trace events” [Ros10b, Ros10c, Ros10d, Ros10a] 로, 극단적으로 낮은 오버헤드를 가지고 데이터를 수집할 수 있게 하는 CPU 별 베피를 사용합니다. 심지어 그렇다고 해도 추적 기능을 켜는 것은 때로는 버그를 숨기기 충분할 정도로 타이밍을 바꿀 수 있어서 Section 11.6 과 특히 Section 11.6.4에서 이야기한 *heisenbug* 를 초래할 수 있습니다. 커널에서, BPF 는 커널 내의 데이터를 줄일 수 있어서, 커널에서 userspace 로의 정보 교환 오버헤드를 줄일 수 있습니다 [Gre19]. Userspace 코드에서는 여러분을 도울 수 있는 도구가 굉장히 많습니다. Brendan Gregg 의 블로그가 시작하기에 좋을 겁니다.⁵

여러분이 *heisenbug* 를 막는다 하더라도, 다른 문제들이 여러분을 기다립니다. 예를 들어, 기계는 정말 모든 걸 알고 있다 하더라도, 그것이 아는 것은 거의 항상 여러분의 머리가 다룰 수 있는것보다 많습니다. 이런 이유로, 높은 품질의 테스트 도구들은 일반적으로 큰 결과를 분석하는 고도화된 스크립트를 갖춥니다. 하지만 알아두세요—스크립트는 여러분이 그것에게 말해달라는 것만 말합니다. 제 `rcutorture` 스크립트는 그런 점에서 한 예입니다.

나: 그 스크립트의 초기 버전은 RCU grace period 가 무한정 멈춰있는 테스트에 대해 만족을 했습니다. 이는 물론 RCU grace period 멈춤을 탐지하기 위한 수정을 초래했지만, 그 스크립트가 제가 그것들에게 찾으라고 한 문제들만을 찾을 것이라는 사실은 변하지 않습니다. 하지만 여러분이 탄탄한 설계를 갖지 않는다면 여러분은 여러분의 스크립트가 무엇을 검사해야 하는지 모를 수 있음을 명심하세요!

특히 `printk()` 호출과 관련한 추적의 또 다른 문제는 그 오버헤드가 제품단계에서의 사용을 금지시킬 수 있다는 것입니다. 그런 경우엔 단정문이 도움이 됩니다.

11.3 Assertions

No man really becomes a fool until he stops asking questions.

Charles P. Steinmetz

Assertions are usually implemented in the following manner:

```
1 if (something_bad_is_happening())
2 complain();
```

이 패턴은 종종 C 전처리기 매크로나 언어 내재 기능 안에 들어가게 되는데 예를 들면 리눅스 커널에서는 `WARN_ON(something_bad_is_happening())` 으로 표현됩니다. 물론, `something_bad_is_happening()` 이 종종 사실이라면, 결과 출력물은 다른 문제들을 숨길 수도 있는 보고서가 될 것이므로, `WARN_ON_ONCE(something_bad_is_happening())` 가 더 적절할 겁니다.

Quick Quiz 11.6: `WARN_ON_ONCE()` 를 어떻게 구현할 수 있나요?

병렬 코드에서 일어날 수도 있는 한가지 나쁜 무언가는 특정 락을 잡은 상태에서 호출될 것으로 예상되는 함수가 그 락을 잡지 않은채 호출되는 것입니다. 그런 함수는 가끔 “호출자는 `foo_lock` 을 잡고서만 이 함수를 호출해야 함” 같은 무언가를 그 함수의 머리부분 주석에 가지고 있습니다만, 그런 주석은 누군가가 그걸 정말로 읽지 않는다면 좋은 일을 하지 않습니다. 수행 가능한 문장이 훨씬 많은 가치를 제공합니다. 그래서 리눅스 커널의 `lockdep` 기능은 [Cor06a, Ros11] 특정 락이 잡혀져 있는지 검사하는 `lockdep_assert_held()` 함수를 제공합니다. 물론, `lockdep` 은 상당한 오버헤드를 일으키며, 따라서 제품에서는 도움이 되지 않을 수도 있습니다.

⁵ <http://www.brendangregg.com/blog/>

병렬 코드에서의 특히나 나쁜 무언가는 데이터로의 예상되지 못한 동시의 접근입니다. Kernel Concurrency Sanitizer (KCSAN) [Cor16a] 은 READ_ONCE() 와 WRITE_ONCE() 같은 존재하는 표시를 어떤 액세스가 경고 메세지를 낼 가치가 있는지 아는데 사용합니다. KCSAN 은 상당한 false-positive 비율을 갖는데, C 를 어셈블리에 추가적 문법을 가진 것으로 생각하는 개발자의 관점에서는 특히나 그렇습니다. 따라서 KCSAN 은 알려진 데이터 경쟁을 포기하기 위해 data_race() 기능을 제공하며, 데이터 경쟁을 명시적으로 검사하기 위해 ASSERT_EXCLUSIVE_ACCESS() 와 ASSERT_EXCLUSIVE_WRITER() 또한 제공합니다 [EMV^{20a}, EMV^{20b}].

그러니 검사가 필요한 곳에서 뭘 할 수 있느냐는 알겠는데, 실행시간 검사 오버헤드가 허용될 수 없는 경우에는 어떡하죠? 한가지 방법은 정적 분석으로, 다음 섹션에서 다룹니다.

11.4 Static Analysis

A lot of automation isn't a replacement of humans but of mind-numbing behavior.

Summarized from Stewart Butterfield

정적 분석은 한 프로그램이 다른 프로그램을 입력으로 받고 해당 프로그램에 있는 에러와 취약점을 보고하는 하나의 검증 기법입니다. 흥미롭게도, 거의 모든 프로그램이 컴파일러나 인터프리터를 통해 정적 분석 됩니다. 이 도구들은 완벽과는 거리가 멀지만 에러를 찾아내는 기술은 지난 수십년간 상당히 개선되었는데, 부분적으로는 분석을 진행하는데 64K 바이트보다 훨씬 큰 메모리를 사용할 수 있게 되었기 때문입니다.

원래의 UNIX lint 도구는 [Joh77] 상당히 유용했는데, 그것의 기능들 중 상당히 많은 것들이 C 컴파일러에 들어오긴 했습니다. 그러나 오늘날 사용되고 있는 link 같은 도구들이 있습니다. Sparse 정적 분석기는 [Cor04b] 다음의 것들을 풍마한 리눅스 커널의 고수준 문제들을 찾아냅니다.

1. User-space 구조체로의 포인터의 잘못된 사용.
2. 너무 긴 상수로부터의 값 할당.
3. 빈 switch 문.
4. 맞지 않는 락 획득과 해제 기능들.
5. 잘못된 per-CPU 기능의 사용.

6. RCU 포인터가 아닌 것에의 RCU 기능 사용과 그 반대.

컴파일러가 자신의 정적 분석 기능을 계속해서 늘려갈 것으로 보이지만, 이 sparse 정적 분석기는 컴파일러 바깥에서의 정적 분석의 장점, 특히 어플리케이션 특수 버그의 탐색을 선보입니다. Sections 12.4–12.5 는 더 정교한 형태의 정적 분석을 설명합니다.

11.5 Code Review

If a man speaks of my virtues, he steals from me; if he speaks of my vices, then he is my teacher.

Chinese proverb

코드 리뷰는 인간이 분석을 하는 정적 분석의 특수한 형태입니다. 이 섹션은 검사 (inspection), 수행 따라가기 (walkthroughs), 그리고 자가 검진 (self-inspection) 을 다룹니다.

11.5.1 Inspection

전통적으로 정식 코드 검사는 정식으로 정의된 역할을 가지고 대면 회의에서 수행됩니다: 운영자, 개발자, 그리고 한명 혹은 두명의 참여자들. 개발자는 코드를 읽어나가며 그게 무엇을 하고 왜 동작하는지 설명합니다. 한명 혹은 두명의 참여자들은 질문을 던지고 문제를 제기하여, 저자의 잘못된 가정을 드러내길 바라며, 운영자의 역할은 발생하는 충돌을 처리하고 기록을 하는 것입니다. 이 과정은 버그를 찾는데 굉장히 효과적인데, 특히 모든 참여자가 해당 코드에 친숙하면 그렇습니다.

그러나, 이 대면 정식 진행은 세계의 리눅스 커널 공동체에서는 잘 동작하지 않을 수 있습니다. 그대신, 각 개인들은 분리된 채로 코드를 리뷰하고 이메일이나 IRC 를 통해 의견을 제공합니다. 기록 남기기는 이메일 기록이나 IRC 로그를 통해 제공되며, 운영자는 간혹 발생하는 의견 대립 시에 자발적으로 서비스를 제공합니다. 이 프로세스는 합리적 수준으로 잘 동작하는데, 모든 참여자가 해당 코드에 친숙할 때 특히 그렇습니다. 실제로, 전통적인 정식 분석 대비 리눅스 커널 공동체 방식의 장점 중 하나는 코드에 친숙하지 않은, 따라서 저자의 잘못된 가정에 의해 눈이 가려지지 않고 코드를 테스트 할수도 있는 사람들의 기여 확률이 높다는 것입니다.

Quick Quiz 11.7: 숨겨져 있는 리눅스 커널 해커들의 어떤 잘못된 가정을 당신은 이야기 하는 건가요???



리눅스 커널 공동체의 리뷰 프로세스는 개선의 여지가 분명 있을 것입니다:

1. 가끔은 효과적인 리뷰를 하기에 필요한 시간과 전문성을 가진 사람이 부족합니다.
2. 모든 리뷰 토론이 기록된다고는 하지만, 통찰이 잊혀지고 사람들이 토론을 찾아내는데 실패함으로써 “소실” 되고는 합니다.
3. 의견 대립은 때로는 해결하기가 어려운데 대립하는 사람들이 다른 목표, 경험, 단어집을 사용할 때 특히 그렇습니다.

아마도 필요한 개선들 중 일부는 continuous-integration 형태의 테스팅을 통해 제공될 것입니다만, 테스팅보다 리뷰를 통해 더 쉽게 발견되는 버그들이 많습니다. 따라서, 리뷰를 할 때에는 커밋 로그, 버그 레포트, 그리고 LWN 기사 등의 관련된 문서들을 볼 가치가 있습니다. 이 문서들은 여러분이 필요한 전문성을 빠르게 얻을 수 있게 도울 겁니다.

11.5.2 Walkthroughs

전통적인 코드 흐름 따라가기는 정식 검사와 비슷하지만 특정 테스트 케이스로 수행되는 코드에 대해 “컴퓨터 역할하기” 그룹이 참여되는 점이 다릅니다. 일반적인 코드 흐름 따라가기 팀은 운영자, 비서 (발견된 버그를 기록), 테스트 전문가 (테스트 케이스를 생성) 그리고 한명 혹은 두명의 참여자로 구성됩니다. 이는 굉장히 효과적일 수 있지만, 역시 굉장히 시간을 소모합니다.

제가 정식 코드 흐름 따라가기에 참여해본 지도 수십 년이 되었으며, 오늘날의 코드 흐름 따라가기는 한단계씩 넘어가기 기능을 사용하는 debugger를 사용할 거라 생각합니다. 다음과 같이 특히 가학적인 진행을 생각해 볼 수 있겠습니다:

1. 테스터가 테스트 케이스를 보입니다.
2. 운영자가 이 테스트 케이스를 입력으로 해서 이 코드를 debugger에서 시작합니다.
3. 각 코드 구문이 문장이 수행되기 전에 개발자는 해당 구문의 결과와 그 결과가 왜 올바른지 설명해야 합니다.
4. 그 결과물이 개발자의 예상과 다르다면 잠재적 버그로 구분됩니다.
5. 병렬 코드에서는 “동시성의 상어”가 어떤 코드가 이 코드와 동시에 수행될 수도 있는지, 그 동시성이 왜 문제가 되지 않는지 묻습니다.

분명 가학적입니다. 효과적일까요? 아마도요. 참여자들이 요구사항, 소프트웨어 도구, 데이터 구조, 그리고 알고리즘에 대해 잘 이해하고 있다면 코드 흐름 따라가기는 굉장히 효과적일 수 있습니다. 그렇지 않다면, 코드 흐름 따라가기는 종종 시간낭비가 됩니다.

11.5.3 Self-Inspection

개발자들이 보통 스스로의 코드를 검사하는데 완전 효과적이지는 않지만, 합리적인 대안이 존재하지 않는 경우가 많습니다. 예를 들어, 개발자는 그 코드를 볼 권한을 가진 유일한 사람일 수 있거나, 다른 권한을 가진 개발자들은 너무 바쁘거나, 문제의 코드가 너무 이상해서 프로토타입을 보이기 전까진 어떤 다른 개발자들도 진지하게 보려 하지 않을 수 있습니다. 이런 경우, 다음 수순이 도움이 될 수 있는데, 복잡한 병렬 코드의 경우 특히 그렇습니다:

1. 요구사항, 데이터 구조 다이어그램, 그리고 설계상 선택들의 이유를 가진 설계 문서를 작성합니다.
2. 전문가에게 자문을 구하고 필요에 따라 설계 문서를 업데이트 합니다.
3. 종이에 펜을 가지고 코드를 작성하고, 나타나는 에러를 고칩니다. 이미 존재하는 거의 똑같은 코드를 참고하려는 유혹에 저항하고, 그 대신 그걸 복사합니다.
4. 각 단계마다, 여러분의 가정을 소리내 읽어보고 질문해 보며, 그걸 확인하기 위해 단정문과 제한 테스트를 삽입합니다.
5. 에러가 있었다면, 깨끗한 종이에 코드를 복사하고 나오는 에러를 고칩니다. 마지막 두개의 복사본이 동일할 때까지 반복합니다.
6. 분명치 않은 모든 코드에 대해 정확성 증명을 작성합니다.
7. 소스코드 제어 시스템을 사용합니다. 일찍 커밋합니다; 자주 커밋합니다.
8. 코드 조각을 바닥부터 테스트 합니다.
9. 모든 코드가 합쳐지면(하지만 가능하면 그 전부터) 전체 기능과 스트레스 테스트를 합니다.
10. 코드가 모든 테스트를 통과하면, 코드 단계 문서화를 하는데, 앞서 이야기한 설계 문서의 확장판이 될 겁니다. 필요에 따라 코드와 테스트 코드를 모두 고칩니다.

제가 이 절차를 새 RCU 코드에 적용할 때, 보통은 오직 약간의 버그만이 마지막에 남습니다. 약간의 두드러진 (그리고 당혹스러운) 예외와 함께 [McK11a], 저는 보통이 버그들을 다른 사람들보다 먼저 찾아내곤 합니다. 그러나, 리눅스 커널 사용자의 수와 다양성이 증가함에 따라 이는 더욱 어려워지고 있습니다.

Quick Quiz 11.8: 왜 존재하는 코드를 종이에 펜으로 복사하죠??? 그건 필사 과정의 여러 가능성을 높일 뿐 아닙니까?

Quick Quiz 11.9: 이 과정은 옛기도로 지나친 엔지니어링이예요! 이 방법으로 작성되는 합리적 양의 소프트웨어를 기대할 수 있습니까???

Quick Quiz 11.10: 종이 위에 펜으로 복사하기 후에 결과 코드를 타이핑하는 와중에 버그를 발견하면 어떡하나요?

앞의 절차는 새 코드에 잘 동작합니다만, 여러분이 이미 작성한 코드를 검사해야 한다면 어떻게 할까요? 물론 여러분은 앞의 절차를 여러분이 작성한 특수한 경우의 기존 코드에 적용할 수 있습니다만 [FPB79], 다음 방법이 덜 절망적인 환경에서는 도움이 될 수 있습니다:

1. 여러분이 좋아하는 문서화 도구를 (LaTeX, HTML, OOpenOffice, 또는 단순한 ASCII) 사용하여 문제의 코드의 고수준 설계를 설명하세요. 데이터 구조와 그것들이 어떻게 업데이트 되는지 설명하기 위해 많은 다이어그램을 사용하세요.
2. 코드의 복사본을 만들고, 모든 코멘트를 제거하세요.
3. 코드가 무엇을 하는지 문장마다 문서화하세요.
4. 버그를 발견하면 고치세요.

코드의 자세한 내용을 설명하는 것은 버그를 찾는 훌륭한 방법이므로 [Mye79] 이는 잘 동작합니다. 이 두 번째 절차는 다른 사람의 코드를 이해하는데도 좋은 방법인데, 첫번째 단계만으로도 종종 충분하기 합니다.

다른 사람에 의한 리뷰와 검사가 아마도 더 효율적이고 효과적이긴 하겠으나, 앞의 절차는 어떤 이유가 되었든 다른 사람들을 관여시킬 수 없는 경우에 상당히 도움이 됩니다.

이 지점에서, 여러분은 이 따분한 서류작업 없이 어떻게 병렬 코드를 작성할 수 있을지 궁금할 수도 있겠습니다. 여기 이를 위해 시간이 증명한 방법들이 있습니다:

1. 병렬 라이브러리 함수를 사용해 확장될 수 있는 순차적 프로그램을 작성하세요.

2. Map-reduce, BOINC, 또는 웹 어플리케이션 서버와 같은 병렬 프레임워크를 위한 순차적 플러그인을 만드세요.
3. 여러분의 문제를 완전히 조각내고 통신 없이 병렬로 동작하는 복수의 순차적 프로그램을 구현하세요.
4. 도구가 문제를 자동으로 분해하고 병렬화 할 수 있는 어플리케이션 영역들 (예를 들어 선형대수) 중 하나만 붙잡으세요.
5. 병렬 프로그래밍 도구의 극단적인 규칙적 사용을 해서, 만들어지는 코드가 정확함이 쉽게 보이게하세요. 하지만 주의하세요: 더 나은 성능과 확장성을 위해 규칙을 “그냥 약간만” 부수고 싶은 유혹을 항상 느낄 겁니다. 규칙을 부수는 건 종종 일반적인 망가짐을 초래합니다. 즉, 여러분이 이 섹션에서 설명된 서류작업을 주의깊게 하지 않다면요.

하지만 슬픈 사실은 여러분이 이 서류작업을 했거나 앞의 서류작업을 회피하기 위한 방법들을 더 또는 덜 했든, 버그는 존재한다는 것입니다. 그외의 것이 없다면 더 많은 사용자와 더 큰 사용자의 다양성은 더 빨리 더 많은 버그를 노출시킬 것인데, 그 사용자들이 원래의 개발자는 고려하지 않은 일을 할 때 특히 그렇습니다. 다음 섹션은 병렬 소프트웨어를 검증할 때 너무나도 흔하게 발생하는 확률적 버그를 어떻게 다룰지 이야기합니다.

Quick Quiz 11.11: 잠깐만요! 대체 왜 소프트웨어의 추상화된 부분이 가끔만 실패하죠???

11.6 Probability and Heisenbugs

With both heisenbugs and impressionist art, the closer you get, the less you see.

Unknown

여러분의 병렬 프로그램은 가끔 실패했습니다. 하지만 여러분은 앞의 섹션에서 배운 기법을 사용해 문제를 찾아내고 고쳤습니다! 축하합니다!!!

이제 질문은 버그가 발생하는 확률을 줄였거나, 관련된 버그들 중 하나만 고친 거이거나, 관계없는 무능한 변경을 만든게 아니고 여러분이 그 버그를 정말로 고쳤음을 확신하기 위해 얼마나 많은 테스트를 해야 하는지입니다. 짧게 말해서, Figure 11.3의 영원한 질문에 대한 답은 무엇일까요?

불행히도, 솔직한 답은 완벽한 확신을 위해선 무한한 양의 테스트가 필요하다는 것입니다.

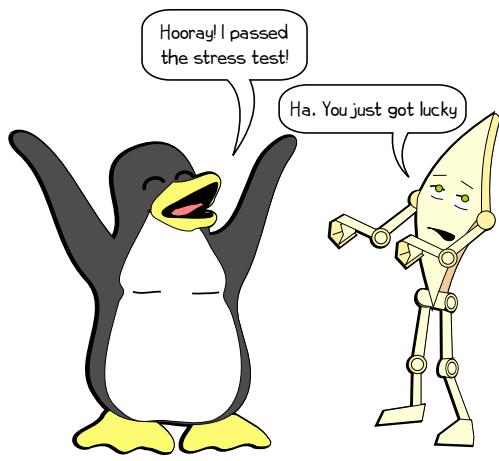


Figure 11.3: Passed on Merits? Or Dumb Luck?

Quick Quiz 11.12: 여러분이 폐기할 수 있는 무척 많은 시스템을 가지고 있다고 해봅시다. 예를 들어, 현재의 클라우드 가격대에서, 여러분은 낮은 가격으로 많은 CPU를 구매할 수 있습니다. 모든 실용적 목적의 충분한 확실성을 위해 이 방법을 사용하는 건 어때요?

하지만 높은 확률을 선호해서 절대적 확실성을 포기 하려 한다고 생각해 봅시다. 그럼 우린 강력한 통계 도구를 이 문제에 적용할 수 있습니다. 그러나, 이 섹션은 간단한 통계 도구들에 집중합니다. 이 도구들은 굉장히 도움이 되지만 이 섹션을 읽는게 통계 수업을 대체할 수는 없음을 알아두세요.⁶

우리의 간단한 통계 도구를 시작하기 위해, 우린 개별 테스트를 할지 연속적 테스트를 할지 정해야 합니다. 개별 테스트는 잘 정의된 개별 테스트 수행들을 갖습니다. 예를 들어, 리눅스 커널 패치의 부팅 테스트는 개별 테스트의 한 예입니다: 커널은 부팅되거나 안되거나입니다. 여러분이 한시간동안 커널 부팅 테스트를 할 수 있더라도, 테스트를 수행하는데 사용한 시간보다 커널을 부팅하려 시도한 횟수와 부팅이 성공한 횟수에 더 관심 있을 겁니다. 기능 테스트는 개별 테스트인 경향이 있습니다.

반면에, 만약 제 패치가 RCU에 관련되어 있다면, 저는 RCU를 테스트하는 기묘한 커널 모듈인 `rcutorture`를 아마 수행할겁니다. 개별 테스트의 성공적인 종료 시에 로그인 프롬프트 시그널을 보이는 커널 부팅과 달리, `rcutorture`는 커널이 크래쉬하거나 여러분이 멈추라고 하기 전까지 RCU를 고문하길 계속할 겁니다.

⁶ 전 통계 수업을 강력히 추천합니다. 제가 들었던 일부 통계 수업은 제가 수업 듣느라 쓴 시간보다 훨씬 많은 가치를 제공했습니다.

`rcutorture` 테스트 시간은 여러분이 그걸 몇번이나 시작하고 종료했는지보다 관심있는 대상이 됩니다. 따라서, `rcutorture`는 많은 스트레스 테스트를 포함하는 카테고리인 연속 테스트의 한 예입니다.

개별 테스트를 위한 통계는 더 간단하고 연속 테스트들보다 친숙하며, 개별 테스트를 위한 통계는 연속 테스트들을 위한 서비스에 사용될 수도 있는데, 정확도가 약간 떨어지긴 합니다. 따라서 개별 테스트부터 이야기를 시작해 봅시다.

11.6.1 Statistics for Discrete Testing

어떤 버그가 한번 수행시 10 % 발생확률을 가지며 우리가 다섯번 수행한다고 해봅시다. 그 중 최소 한번의 수행은 실패할 확률을 어떻게 계산할까요? 여기 한가지 방법이 있습니다:

1. 특정 수행이 성공할 확률, 90 % 를 계산합니다.
2. 다섯번의 수행이 모두 성공할 확률, 0.9 를 다섯번 곱한 값, 약 59 % 를 계산합니다.
3. 모든 다섯번의 수행이 성공하거나 최소 한번은 실패하거나 이므로 예상된 성공 확률은 59 % 를 100 % 에서 빼서 41 % 의 실패 확률을 도출합니다.

수식을 선호하는 사람들을 위해, 한번 실패할 확률을 f 라 합시다. 한번 성공할 확률은 그러면 $1-f$ 이며 모든 n 테스트가 성공할 확률은 S_n 입니다:

$$S_n = (1 - f)^n \quad (11.1)$$

실패할 확률은 $1 - S_n$, 또는:

$$F_n = 1 - (1 - f)^n \quad (11.2)$$

Quick Quiz 11.13: 뭐라구요??? 앞의 10 % 실패율의 다섯번의 테스트 예를 이 식에 대입해보면 59,050 % 를 얻게 되는데 이건 말이 안되잖아요!!!

어떤 테스트가 10 % 회 실패했다고 가정해 봅시다. 여러분의 수정이 정말 도움이 되었음을 99 % 확신하려면 얼마나 많이 테스트를 반복해야 할까요?

이 질문을 묻는 다른 방법은 “실패 확률이 99 % 까지 오르게 하려면 몇번이나 테스트를 수행해야 할까요?”입니다. 어쨌건, 한번이라도 실패를 보게 될 확률이 99 % 이기 충분할 정도로 테스트를 반복한다면, 그리고 거기서 실패를 보지 못했다면, 이 “성공” 이 행운 덕일 뿐일 확률은 1 % 뿐입니다. 그리고 Equation 11.2 에 $f = 0.1$ 을 넣고 n 을 변화시켜보면, 테스트 수행당 10 % 실패 확률을 갖는 상황에서 최소 한번은 실패할 확률은 43

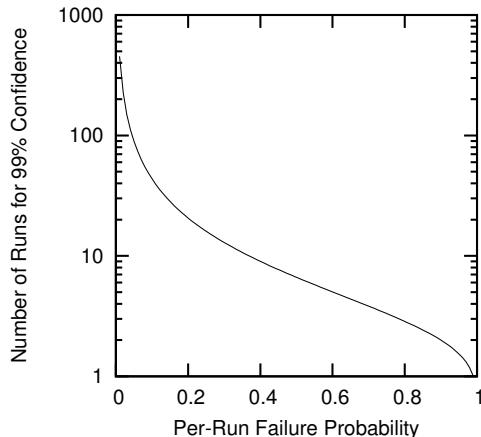


Figure 11.4: Number of Tests Required for 99 Percent Confidence Given Failure Rate

회 수행에서 98.92 %, 44회 수행에서 99.03 % 가 됩니다. 그러니 우리의 수정사항을 포함한 테스트 수행이 44회 반복해서 실패하지 않는다면 우리의 수정사항이 정말로 도움이 되었을 확률이 99 % 가 됩니다.

하지만 반복적으로 Equation 11.2에 수를 넣는 건 지루할 수 있으니, 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 필요한 테스트 횟수는 다음과 같이 됩니다:

$$n = \frac{\log(1 - F_n)}{\log(1 - f)} \quad (11.6)$$

Equation 11.6에 $f = 0.1$ 과 $F_n = 0.99$ 를 대입하면 43.7 이 나오는데, 우리의 수정사항이 정말 개선을 만들었음에 99 % 자신하려면 44번의 연속된 테스트 성공이 있어야 함을 의미합니다. 이는 앞의 방법으로 얻어진 수와 일치하므로, 안심해도 되겠습니다.

Quick Quiz 11.14: Equation 11.6에서, 로그의 밑은 10인가요, 2인가요, 아니면 e인가요?

Figure 11.4 가 이 함수의 그림을 보입니다. 놀랍지 않게도, 덜 빈번하게 각 테스트가 실패할수록, 버그가 고쳐졌다고 99 % 확신하는데 필요한 테스트 수행 횟수가 늘어납니다. 이 버그가 1 % 의 테스트에서만 실패한다면, 458 회의 테스트가 필요합니다. 실패 확률이 줄어들수록, 필요한 테스트 횟수는 늘어나서, 실패 확률이 0 이 되면 무한대가 됩니다.

이 이야기의 교훈은 여러분이 드물게 발생하는 버그를 찾았을 때, 여러분이 조심스레 만들어져 훨씬 높은 실패 확률을 갖는 테스트를 사용할 때 여러분의 테스트 업무가 훨씬 쉬워질 것이라는 겁니다. 예를 들어, 여러분의 테스트가 실패 확률을 1 % 에서 30 % 로 늘린다면 99 % 확신을 위한 테스트 횟수는 458 에서 13 으로 떨어집니다.

하지만 이 13회 테스트 수행은 여러분의 수정이 “어떤 개선”을 만들었다는 99 % 자신감만을 줍니다. 여러분의 수정 사항이 실패 확률을 열배 줄였다는 자신감을 99 % 얻고자 한다고 해봅시다. 얼마나 많은 테스트 성공 횟수가 필요할까요?

30 % 실패 확률에서 열배 개선이면 3 % 실패 확률일 겁니다. 이를 Equation 11.6에 대입하면:

$$n = \frac{\log(1 - 0.99)}{\log(1 - 0.03)} = 151.2 \quad (11.7)$$

따라서 열배 개선은 대략 열배 더 많은 테스트를 필요로 합니다. 확정성은 불가능하며, 높은 확률은 매우 비쌉니다. 이것이 테스트가 더 빠르게 수행되고 실패가 잘 발생하게 하는게 고도로 안정화된 소프트웨어를 개발하는데 필수의 기술인지에 대한 이유입니다. 이 기술들은 Section 11.6.4에서 다루어집니다.

11.6.2 Statistics Abuse for Discrete Testing

하지만 열시간마다 약 세번 실패하는 반복적 테스트가 있고, 그 실패를 유발했다고 여겨지는 버그를 고쳤다고 해봅시다. 여러분이 실패 확률을 줄였음에 99 % 자신하기 위해선 이 테스트를 실패 없이 얼마나 오래 수행해야 할까요?

통계에 상당한 폭력을 가하지 않고, 우린 한시간 수행을 30 % 실패 확률을 갖는 개별 테스트로 재정의 할 수 있습니다. 그러면 앞의 섹션에서의 결과가 그 테스트가 13시간동안 실패 없이 수행된다면 우리의 수정사항이 이 프로그램의 안정성을 정말 개선했다는 99 % 자신을 가질 수 있다고 합니다.

독실한 확률가는 이 방법을 허락하지 않을 것이지만, 슬픈 사실은 이런 종류의 확률 남용에 의한 오류들은 여러분의 실패 확률 예측 상의 오류에 비하면 무척 작다는 겁니다. 그러나, 다음 섹션은 더 엄격한 방법을 알아봅니다.

11.6.3 Statistics for Continuous Testing

실패 확률의 기본 공식은 Poisson 분포입니다:

$$F_m = \frac{\lambda^m}{m!} e^{-\lambda} \quad (11.8)$$

여기서 F_m 은 테스트에서 m 회 실패할 확률이며 λ 는 단위 시간당 예상되는 실패율입니다. 모든 고급 확률 교재에서는 상당한 도출이 있겠는데, 예를 들어 Feller의 고전인 “An Introduction to Probability Theory and Its Applications” [Fel50] 이 있겠으며, 더 직관적인 도출은 이 책의 첫번째 판본에서 [McK14a, Equations 11.8–11.26] 찾아볼 수 있을 겁니다.

Section 11.6.2 의 예를 Poisson 분포를 사용해 다시 작업해 봅시다. 이 예는 시간당 30 % 실패율을 갖는 테스트를 사용하며 질문은 수정이 실패 확률을 줄였음을 99 % 확신하기 위해 얼마나 오래 에러 없이 테스트를 돌려야 하는지임을 다시 말씀드립니다. 이 경우, m 은 0이며, 따라서 Equation 11.8 은 다음과 같이 요약됩니다:

$$F_0 = e^{-\lambda} \quad (11.9)$$

이를 푸는데는 F_0 를 0.01 로 하고 λ 를 구할 것이 필요해지는데, 다음과 같습니다:⁷

$$\lambda = -\ln 0.01 = 4.6 \quad (11.10)$$

시간당 0.3 실패를 가지므로, 필요한 시간은 $4.6/0.3 = 14.3$ 로, Section 11.6.2 에서의 방법으로 계산된 13 시간의 10 % 입니다. 보통 여러분의 실패 확률이 10 % 근처임을 모를 것을 놓고 생각하면, Section 11.6.2 에서 이야기된 더 간단한 방법이 거의 항상 충분히 좋을 겁니다.

더 일반적으로는, 만약 우리가 단위 시간당 n 회 실패를 한다면, 그리고 어떤 수정사항이 이 실패 확률을 줄였음을 P % 확신하려면 다음 공식을 사용할 수 있습니다:

$$T = -\frac{1}{n} \ln \frac{100 - P}{100} \quad (11.11)$$

Quick Quiz 11.15: 어떤 버그가 어떤 테스트를 시간당 평균 세번 실패를 하게 한다고 해봅시다. 어떤 수정사항이 그 실패 확률을 충분히 줄였음을 99.9 % 확신하기 위해선 얼마나 오래 에러 없는 테스트 수행을 해야 할까요?

이전과 같이, 버그가 덜 빈번히 발생하고 필요한 확신의 수준이 높을수록 에러 없는 테스트 수행은 더 길게 필요해집니다.

어떤 테스트가 시간당 한번 정도 실패하지만 어떤 버그 수정 후, 24시간의 테스트에서 두번만 실패한다고 해봅시다. 이 버그로 이끄는 실패가 무작위적으로 발생한다고 가정하면, 두번째 수행에서의 적은 수의 실패가 무작위적 선택에 의한 것일 확률이 얼마나 될까요? 달리 말하자면, 그 수정이 버그에 영향을 줬음에 우린 얼마나

자신할 수 있을까요? 이 확률은 Equation 11.8 을 다음과 같이 더해서 계산할 수 있을 겁니다:

$$F_0 + F_1 + \dots + F_{m-1} + F_m = \sum_{i=0}^m \frac{\lambda^i}{i!} e^{-\lambda} \quad (11.12)$$

이는 다음과 같이 요약될 수 있는 Poisson 누적 분포 함수입니다:

$$F_{i \leq m} = \sum_{i=0}^m \frac{\lambda^i}{i!} e^{-\lambda} \quad (11.13)$$

여기서 m 은 긴 시간의 테스트 수행(이 경우, 두시간)에서의 오류 횟수이며 λ 는 긴 테스트 수행에서(이 경우, 24시간) 예상되는 오류의 수입니다. $m = 2$ 와 $\lambda = 24$ 를 이 식에 대입하면 두번 이하 실패 확률은 1.2×10^{-8} 이 되는데, 달리 말하면 이 수정이 버그에 어떤 영향을 주었다는 높은 수준의 자신을 가질 수 있습니다.⁷

Quick Quiz 11.16: 이 모든 factorial 과 exponential 을 더하는 건 고통입니다. 더 쉬운 방법은 없나요?

Quick Quiz 11.17: 하지만 기다려요!!! 약간의 실패가 있음을 생각하면 (0번 실패할 확률을 포함해), Equation 11.13 은 m 이 무한으로 감에 따라 1 의 값이 되지 않나요?

Poisson 분포는 테스트 결과를 분석하기 위한 강력한 도구입니다만 이 마지막 예에서는 여전히 24시간 테스트 수행에서의 두번의 실패가 남아있었음이 사실입니다. 그런 낮은 실패 확률은 매우 긴 테스트 수행을 초래합니다. 다음 섹션은 이 상황을 개선하는 반직관적인 방법들을 이야기 합니다.

11.6.4 Hunting Heisenbugs

이 생각은 또한 heisenbug 의 설명을 돋습니다: 추적과 단정을 더하는 것은 버그가 나타나는 확률을 쉽게 줄이는데, 극단적으로 가벼운 추적과 단정 메커니즘이 치명적으로 중요한 이유이다.

“Heisenbug” 라는 용어는 특정 입자의 특정 시간에 서의 위치와 속도를 정확하게 알아내는 것은 불가능하다고 이야기하는 Heisenberg 의 Uncertainty Principle (불확정성 원리) [Hei27] 에 영감을 받아 만들어졌습니다. 어떤 입자의 위치를 정확하게 측정하려는 모든 시도는 그것의 속도의 불확정성을 증가시키며 그 반대

⁷ 물론, 이는 여러분이 남은 두번의 실패를 초래한 버그(들)을 찾아서 고쳐야 한다는데에 대한 변명이 되지 않습니다!

역시 마찬가지입니다. 비슷하게, heisenbug 를 추적하려는 시도는 그 증상이 크게 바뀌거나 심지어 완전히 사라지게도 만듭니다.⁸

만약 해당 물리학계가 이 문제의 이름에 영감을 주었다면, 해당 물리학계가 그 해법에도 영감을 주는게 공평합니다. 다행히도, 입자물리학은 그 일을 하고 있습니다: Heisenbug 를 무효화 시키기 위해 anti-heisenbug 를 만드는 게 어떻습니까? 또는, 아마 더 정확히는, heisenbug 의 heisen 특성을 무효화 시키기 위해? 특정 heisenbug 를 위한 anti-heisenbug 를 만드는 것은 과학보다는 예술의 영역인데, 다음 섹션들이 그걸 하기 위한 방법들을 설명합니다:

1. 경주가 일어나기 쉬운 영역에 지연을 추가합니다 (Section 11.6.4.1).
2. 워크로드의 강도를 증가시킵니다 (Section 11.6.4.2).
3. 의심가는 서브시스템들을 고립시킵니다 (Section 11.6.4.3).
4. 일반적이지 않은 이벤트를 시뮬레이션 합니다 (Section 11.6.4.4).
5. 실패에 가까운 것들을 세어봅니다 (Section 11.6.4.5).

이것들에 이어 고찰이 이어집니다 Section 11.6.4.6.

11.6.4.1 Add Delay

Section 5.1 의 셈을 놓치기 쉬운 코드를 생각해 봅시다. `printf()` 문을 더하는 것은 놓쳐진 셈을 크게 줄이거나 심지어 없앨 수도 있습니다. 그러나, 읽고-더하고-저장하기 순서를 읽고-더하고-지연하고-저장하기 순서로 바꾸는 것은 셈을 잃는 사고를 크게 증가시킬 겁니다 (해보세요!). 어떤 경주 조건에 (race condition) 관련된 버그를 찾았다면, 이런식으로 지연을 더하는 것으로 anti-heisenbug 를 만드는게 종종 가능합니다.

물론, 이는 일단 그 경주 조건을 어떻게 찾을 것인가 하는 질문을 남깁니다. 이는 약간 암흑예술에 가깝지만, 그걸 찾기 위해 여러분이 해볼 수 있는 것들이 좀 있습니다.

한가지 방법은 경주 조건은 종종 그 경주에 관여된 데이터 일부를 오염시키곤 한다는 겁니다. 따라서 모든 오염된 데이터의 동기화를 이중 검사하는 건 좋습니다. 당장 경주 조건을 찾아내지는 못하더라도 이 오염된

⁸ “Heisenbug”라는 용어는 잘못된 호칭인데, 대부분의 heisenbug 는 고전 물리학의 *observer effect*로 설명되기 때문입니다. 그러나, 그 이름은 유행하지 못했습니다.

데이터로의 액세스 전후에 지연을 더하는 것은 실패 확률을 줄일 수도 있습니다. 이 지연들을 어떤 정리된 방법으로 (예: 바이너리 탐색) 더하고 뺀으로써 여러분은 이 경주 조건에 대해 뭔가를 배울지도 모릅니다.

Quick Quiz 11.18: 이 오염이 어떤 연관되지 않은 포인터에 영향을 끼쳐서 그게 이후 오염을 일으킨다면 이 방법이 어떻게 도움이 되겠습니까?

또 다른 중요한 방법은 소프트웨어와 하드웨어 구성 을 다양하게 해보고 통계적으로 상당한 실패 확률의 차이를 보는 겁니다. 그러면 여러분은 실패 확률에 가장 큰 차이를 만드는 소프트웨어나 하드웨어 구성 변경에 영향 받는 코드를 집중해서 볼 수 있습니다. 그 코드를 예를 들면 고립시켜서 테스트 하는게 도움이 될 수도 있습니다.

소프트웨어 구성에 있어 중요한 지점은 변경의 역사로, `git bisect` 가 그리 유용한 이유입니다. 변경 역사를 `bisect` 하는 것은 heisenbug 의 정체에 대한 매우 가치있는 단서를 줄 수 있습니다.

Quick Quiz 11.19: 전 `bisect` 를 했지만 거대한 커밋을 찾는데 그쳤어요. 이제 어떡하죠?

하지만 여러분이 의심스러운 코드 영역을 발견하면 그 실패 확률을 높이기 위해 지연을 추가할 수 있습니다. 앞서 봤듯이, 실패의 확률을 높이는 것은 연관된 수정사항에 대한 높은 확신을 얻기 쉽게 해줍니다.

그러나, 평범한 디버깅 기법을 가지고는 문제를 추적하기가 상당히 어려운 경우가 있습니다. 다음 섹션은 다른 대안들을 보입니다.

11.6.4.2 Increase Workload Intensity

특정 테스트 집합이 특정 서브시스템에 상대적으로 낮은 자극을 가해서 타이밍에의 작은 변화가 heisenbug 를 사라지게 하는 경우가 있습니다. 이 경우를 위한 anti-heisenbug 를 만드는 한가지 방법은 워크로드의 강도를 높이는 것으로 버그의 확률을 높이는 좋은 기회를 갖습니다. 그 확률이 충분히 증가했다면, 그 버그가 사라지지 않게 하면서 추적과 같은 가벼운 검사기법을 활용할 수 있습니다.

워크로드의 강도를 어떻게 증가시킬 수 있을까요? 그건 프로그램에 의존적이지만 여기 시도해볼 만한 몇 가지가 있습니다:

1. CPU 를 추가합니다.
2. 프로그램이 네트워킹을 사용한다면 더 많은 네트워크 어댑터와 더 빠른 원격 시스템을 더합니다.
3. 그 프로그램이 문제가 일어날 때 상당한 I/O 를 한다면 (1) 더 많은 저장장치를 더하거나, (2) 예를

들어 SSD 나 disk 를 대체하는 더 빠른 저장장치를 사용하거나, (3) 대용량 저장소를 메인 메모리로 대체하기 위해 RAM 기반 파일시스템을 사용합니다.

- 문제의 크기를 바꿔보는데, 예를 들어 병렬 행렬 곱셈을 하고 있다면, 행렬의 크기를 바꿔봅니다. 더 큰 문제는 더 많은 복잡도를 더하지만 더 작은 문제는 종종 경쟁 수준을 증가시킵니다. 크기를 늘려야 할지 줄여야 할지 모르겠다면 그냥 다 해보세요.

그러나, 버그가 특정 서브시스템 내에 있고 프로그램의 구조가 그 서브시스템에 가해지는 자극의 양을 제한하는 경우가 종종 있습니다. 다음 섹션은 이 상황을 다뤄봅니다.

11.6.4.3 Isolate Suspicious Subsystems

의심가는 서브시스템에 많은 자극을 가하기가 어렵거나 불가능하게 프로그램이 짜여 있다면, 한가지 유용한 anti-heisenbug 는 그 서브시스템을 고립시켜 테스트하는 스트레스 테스트입니다. 리눅스 커널의 `rcutorture` 모듈은 정확히 이 방법을 RCU 에 적용합니다: 그러면 제품 환경에서 RCU 에 더 많은 자극을 가하는 것은 RCU 버그가 제품이 아닌 테스트 중에 발견될 확률을 높입니다.⁹

실제로, 병렬 프로그램을 만들때에는 컴포넌트들을 개별적으로 스트레스 테스트 하는게 현명합니다. 그런 컴포넌트 수준 스트레스 테스트는 시간낭비처럼 보일 수 있지만, 약간의 컴포넌트 수준 테스트가 시스템 레벨 디버깅 시간을 크게 아낄 수 있습니다.

11.6.4.4 Simulate Unusual Events

Heisenbug 는 때로는 메모리 할당 실패, 조건적 락 획득 실패, CPU 핫플러그 오퍼레이션, 타임아웃, 패킷 손실, 기타 등등의 흔하지 않은 사건 때문에 일어납니다. 이런 종류의 heisenbug 를 위한 anti-heisenbug 를 만드는 한가지 방법은 가짜 실패를 만드는 것입니다.

예를 들어, `malloc()` 을 직접 호출하는 대신 무조건적으로 NULL 을 리턴할지, 아니면 정말로 `malloc()` 을 수행하고 그 결과 포인터를 리턴할지 결정하는데 무작위수를 사용하는 같은 wrapper 함수를 호출하는 겁니다. 가짜 실패를 만드는 것은 병렬 프로그램 뿐 아니라 순차적 프로그램에도 안정성을 높이는데 훌륭한 방법입니다.

Quick Quiz 11.20: 조건적 락킹 기능은 왜 이 가짜 실패 기능을 제공하지 않죠?



⁹ 슬프게도 확률을 1까지 높이지는 못하지만요.

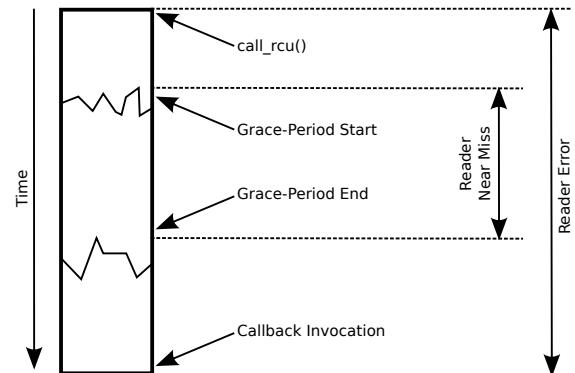


Figure 11.5: RCU Errors and Near Misses

11.6.4.5 Count Near Misses

버그는 많은 경우 그렇거나 아니거나여서 버그는 일어났거나 아니거나일 뿐 그 중간이 없습니다. 그러나, 버그가 실패를 초래하진 않았지만 나타났을 것 같은 거의 실수함 을 정의하는 것이 가끔은 가능합니다. 예를 들어, 여러분의 코드가 어떤 로봇을 걷게 한다고 해봅시다. 이 로봇의 쓰러짐은 여러분의 프로그램에 버그가 있음으로 간주됩니다만, 비틀거리고 회복되는 것은 거의 실수함으로 간주될 수도 있을 겁니다. 로봇이 한시간에 한번만 쓰러지지만 몇분에 한번씩 비틀거린다면 여러분은 비틀거림 횟수를 쓰러짐 횟수에 더함으로써 디버깅을 빠르게 진행할 수 있을 겁니다.

동시성 프로그램에서, 시간 기록은 거의 놓침을 파악하는데 때때로 사용될 수 있습니다. 예를 들어 락킹 기능은 상당한 지연을 유발하므로 같은 락의 다른 획득으로 보호되는 한쌍의 오퍼레이션 사이에 너무 짧은 지연이 있다면 이 너무 짧은 지연은 거의 실수했음으로 세어질 수 있을 겁니다.¹⁰

예를 들어, 어떤 RCU 우선순위 부스팅의 낮은 확률 버그는 집중된 `rcutorture` 테스트에서 백시간마다 한번 가량 빌생했습니다. 이 버그의 확률이 상당히 줄었음을 99 % 확신하기 위해선 500 시간의 실패 없는 테스팅이 필요했으므로, 이 실패를 찾기 위한 `git bisect` 과정은 고통스럽게 느렸을 겁니다—또는 굉장히 커다란 테스트 환경이 필요했을 겁니다. 다행히도, 이 테스트 되는 RCU 오퍼레이션은 RCU grace period 대기만이 아니라 시작하기 위한 grace period 의 이전 대기, 그리고 이 RCU grace period 의 완료 후 RCU 콜백을 호출하기 위한 다음 대기를 포함했습니다. `rcutorture` 에러와 거의 실수했음 사이의 이 차이가 Figure 11.5 에 보여져

¹⁰ 물론, 이 경우 여러분은 여러분의 환경에서 사용 가능한 `lock-held()` 같은 기능을 사용하는게 나을수도 있을 겁니다. `lock-held()` 기능이 없다면, 하나 만드세요!

있습니다. 완벽한 오류의 요건을 갖추기 위해, 하나의 RCU read-side 크리티컬 섹션은 하나의 grace period를 시작한 `call_rcu()`로부터 앞의 grace period의 남은 부분을 지나 `call_rcu()`에 의해 시작된 grace period 전체를 지나 (구불거리는 선 사이의 영역으로 표시되었습니다), 이 grace period의 종료로부터 콜백 호출 사이의 지연까지 지나도록 연장되어야 하는데, “Error” 화살표로 표시되어 있습니다. 그러나, RCU의 정형적 정의는 RCU read-side 크리티컬 섹션이 단일 grace period를 넘어 연장될 수 없게 하는데, “Near Miss” 화살표로 표시되어 있습니다. 이는 거의 실수함을 애러 조건으로 사용할 것을 추천하게 되지만, 언제 특정 grace period가 시작되고 끝났는지에 대해 구불거리는 선으로 표시된 것처럼 다른 CPU들은 다른 의견을 가질 수 있으므로 문제가 됩니다.¹¹ 따라서 이 거의 실수함을 오류로 사용하는 것은 false positive를 초래할 수 있는데, 이는 자동화된 `rcutorture` 테스트에서는 지양되어야 합니다.

운좋게도 `rcutorture`는 grace period의 거의 실수 버전에 민감한 일부 통계를 포함합니다. 앞서 이야기 했듯, 이 통계들은 RCU의 상태 변수들로의 비동기적 액세스 덕에 false positive가 될 수 있지만, 이 false positive는 IBM 메인프레임과 x86 같은 강한 순서 규칙 시스템에서는 굉장히 드물어서 천시간 테스팅 중 한번 정도 일어납니다.

이런 거의 실수함은 대략 한시간에 한번 일어나는데, 실제 실패보다 백배 이상 자주 일어나는 겁니다. 이런 거의 실수함을 사용하는 것은 이 버그의 근본 원인이 일주일도 안되어 파악될 수 있게 하였고 효과적이었음을 높은 수준으로 확인할 수 있는 수정사항을 하루도 안되어 만들 수 있게 했습니다. 대조적으로, 실제 오류만을 선호해 거의 실수함을 배제하는 것은 몇달의 디버깅과 검증 시간을 필요로 했을 겁니다.

거의 실수를 세는 것을 정리하자면, 일반적 접근법은 흔하지 않은 실패 세기를 그 실패와 연관되어 있을 것으로 여겨지며 더 흔한 거의 실수함으로 대체하는 것입니다. 이런 거의 실수함은 실제 실패의 heisenbug에 대한 anti-heisenbug로 여겨지는데, 거의 실수함은 더 흔하게 발생하고, 예를 들면 여러분이 디버깅을 위해 코드에 무언가를 추가하는 것 같은 코드 변경에 더 잘 버티기 때문입니다.

11.6.4.6 Heisenbug Discussion

주의 깊은 독자는 이 섹션이 Section 11.6.1, 11.6.2, 그리고 11.6.3의 정교한 수학과 강력하게 대비될 만큼 흐릿하고 정성적이었음을 알아차렸을 수도 있을 겁니다. 여러분이 정교함과 수학을 사랑한다면, 이 섹션이

¹¹ 실제 세계에서, idle CPU는 많은 최근 grace period를 완전히 인지하지 못할 수 있으므로 이 선들은 훨씬 더 구불거릴 수 있습니다.

적용되는 상황이 앞의 섹션들이 적용되는 것들에 비해 훨씬 더 혼함에 실망할지도 모르겠습니다.

사실, 더 흔한 경우는 여러분이 여러분의 코드에 버그가 있을 거라 믿는 이유가 있을지 몰라도 여러분은 어떤 그런 버그가 있는지, 무엇이 그것을 유발하는지, 그게 어떻게 발현되는지, 또는 어떤 조건이 그 발현의 확률에 영향을 주는지 모른다는 겁니다. 이 너무나도 흔한 경우에, 통계는 여러분을 돋지 못합니다.¹² 즉, 통계는 여러분을 직접적으로 돋지 못합니다. 그러나 통계는 만약 여러분이 실수할 수 있음을 인정하는데 필요한 겸손을 가지고 있다면, 그 실수의 확률을 줄일 수 있다면 (예를 들어, 충분히 잠을 자는 식으로), 여러분이 과거에 만든 실수의 종류와 수가 미래에 만들 실수의 종류와 수를 말해준다면 큰 도움이 될 수 있습니다. 예를 들어, 저는 초기화 코드의 작지만 중요한, 그리고 대부분의 또는 심지어 전체 병렬 프로그램을 정확하게 하는 작지만 중요한 부분을 잊는 안타까운 경향을 갖습니다—초기화에서의 명청한 누락을 제외하고요. 일단 제가 저는 이런 종류의 실수에 취약함을 인정하면, 저 스스로를 제 초기화 코드를 재차 확인하게끔 하는게 조금 더 쉬워졌습니다 (하지만 쉽지 않았습니다!). 이렇게 하는건 제가 시간 내에 많은 버그를 찾을 수 있게 해줬습니다.

Taleb의 명명법 [Tal07]을 사용하자면, 백조는 우리가 재현할 수 있는 버그입니다. 우린 많은 수의 테스트를 수행하고 버그의 확률을 추정하는데 일반적인 통계를 사용하고, 제안된 수정사항에 대한 우리의 자신감을 추정하는데에 일반적 통계를 또 사용할 수 있습니다. 기대되지 않은 버그는 흑조입니다. 우린 그것에 대해 무엇도 모르고, 그게 일어나게 한 어떤 테스트도 없으며, 통계는 도움이 되지 않습니다. 우리의 행동, 특히 우리가 만드는 실수들의 수와 종류를 연구하는 것은, 흑조를 회색조로 만들 수 있습니다. 우린 그 버그가 정확히 무엇인지는 모를 수도 있지만 그것의 수와 어찌면 그 종류에 대한 어떤 아이디어를 얻을 수도 있습니다. 일반적인 통계는 여전히 도움이 되지 않지만 (우리가 그 버그들 중 하나를 재현할 수 있기 전까지는요) 튼튼한 테스트 방법은¹³ 큰 도움이 될 수 있습니다. 따라서 목표는 흑조를 회색으로 바꾸기 위해 경험과 좋은 검증 수법들을 사용하고, 집중된 테스트와 분석을 통해 회색조를 흰색으로 바꾸고, 이 백조를 고치기 위해 일반적 방법을 사용하는 것입니다.

그러나, 지금까지 우리는 병렬 프로그램의 기능에서의 버그에만 집중했습니다. 하지만, 성능이 병렬 프로그램에서의 첫번째 요구사항이므로 (아니라면, 왜 순차적 프로그램을 작성하지 않습니까?), 다음 섹션은 성능 버그에 대해 이야기 해봅니다.

¹² 여러분의 프로그램이 무엇을 할거라 기대되는지 여러분이 알고 있고 여러분의 프로그램이 충분히 작다면 (여러분이 생각하는 경우는 아닐 확률이 큽니다만), Chapter 12에서 설명되는 정형적 검증 도구가 도움될 겁니다.

¹³ brutal이라고도 이야기 되지요.

11.7 Performance Estimation

There are lies, damn lies, statistics, and benchmarks.

Unknown

병렬 프로그램은 보통 성능과 확장성 요구사항을 갖는데, 어쨌건 성능이 문제가 아니라면 왜 순차적 프로그램을 쓰지 않겠습니까? 궁극의 성능과 선형적 확장성은 필요하지 않을 수도 있지만 그것의 최적화된 순차적 버전보다 느리게 동작하는 병렬 프로그램은 잘 쓰지 않을 겁니다. 그리고 매 마이크로세컨드와 매 나노세컨드가 필요한 실제 사례들이 있습니다. 따라서, 병렬 프로그램을 위해선 부족한 성능이 부정확성만큼이나 큰 버그입니다.

Quick Quiz 11.21: 웃기네요!!! 어쨌건, 남들보다 늦게 올바른 답을 얻는게 올바르지 않은 답을 얻는것보다는 낫지 않겠습니까???

Quick Quiz 11.22: 하지만 그 모든 어플리케이션 병렬화의 어려운 일을 하고자 한다면, 왜 그걸 옳게 하지 않죠? 왜 최적의 성능과 선형 확장성 미만의 것에 만족합니까?

따라서 병렬 프로그램을 검증하는 것은 그 성능을 검증하는 것도 포함해야합니다. 하지만 성능을 검증함은 그 프로그램을 평가하기 위해 수행할 워크로드와 성능 지표를 필요로 함을 의미합니다. 이 필요성은 종종 성능 벤치마크에 의해 충족되는데, 다음 섹션에서 이야기 됩니다.

11.7.1 Benchmarking

종종 오남용되지만, 벤치마크는 유용하고 널리 사용되므로, 그것을 너무 나쁘게만 보는건 도움이 되지 않습니다. 벤치마크는 테스트 추가적 일거리부터 국제적 표준 까지 다양한 범위에 존재하지만 그것의 정형성 수준과 관계없이 벤치마크는 네개의 주요 목적을 충족합니다:

1. 경쟁 구현들의 비교를 위한 정당한 프레임워크의 제공.
2. 사용자들이 신경쓰는 방향으로 구현을 개선하는데 있어서의 경쟁적인 에너지에 집중할 것.
3. 벤치마크되는 구현의 사용 예로써 기능할 것.
4. 여러분의 경쟁자의 제안사항 대비 여러분의 소프트웨어의 장점을 나타내기 위한 마케팅 도구로써 기능할 것.

물론, 완전하게 공평한 유일한 프레임워크는 의도된 어플리케이션 그 자체입니다. 그러나 벤치마킹의 공정성에 관심 있는 사람들은 왜 어플리케이션 그 자체를 벤치마크로 사용하는 대신 완벽하지 않은 벤치마크를 만드는 고생을 할까요?

실제 어플리케이션을 수행하는 것은 사실 그게 실용적이라면 최선입니다. 불행히도, 그것은 다음 이유들로 실용적이지 못합니다:

1. 그 어플리케이션은 독점되어 있을 수도 있고, 여러분은 그 어플리케이션을 수행할 권리를 얻지 못할 수도 있습니다.
2. 그 어플리케이션은 여러분이 접근할 수 있는 것보다 많은 하드웨어를 필요로 할 수도 있습니다.
3. 그 어플리케이션은 예를 들어 개인정보 규정 같은 여러분이 접근할 수 없는 데이터를 사용할 수도 있습니다.
4. 그 어플리케이션은 성능과 확장성 문제를 재현하는데 간편한 수준보다 오랜 시간을 필요로 할 수도 있습니다.¹⁴

그 어플리케이션을 추상화한 벤치마크를 만드는 게이 장애물을 극복하는데 도움됩니다. 조심스럽게 구축된 벤치마크는 성능, 확장성, 에너지 효율, 그 외에도 여러 가지를 개선하는데 도움이 됩니다. 그러나, 벤치마킹 노력에 너무 많은 투자를 하는 건 피하십시오. 적어도 얼마간은 어플리케이션 그 자체에 투자하는 게 중요합니다 [Gra91].

11.7.2 Profiling

많은 경우, 여러분의 소프트웨어의 꽤 작은 부분이 성능과 확장성 단점의 대부분에 책임을 집니다. 그러나, 개발자들은 조사를 통해 실제 병목지점을 찾아내지는 못하기로 악명이 높습니다. 예를 들어, 커널 버퍼 할당자의 경우, 모든 관심은 이 할당자의 실행 시간의 몇 퍼센트만 차지하는 것으로 드러난 밀집 배열 탐색에 집중되었습니다. 한 논리 분석기에 의해 수집된 수행 프로파일은 이 문제의 대부분을 차지하는 캐쉬 미스에 관심을 집중시켰습니다 [MS93].

성능과 확장성 버그를 추적하는 고전적이지만 매우 효과적인 방법은 여러분의 프로그램을 디버거를 통해 수행하고, 주기적으로 인터럽트하고, 각 인터럽트마다 모든 쓰레드의 스택을 기록하는 겁니다. 여기서의 가설은 만약 무언가가 여러분의 프로그램을 느리게 하고

¹⁴ 마이크로벤치마크가 도움이 될 수 있지만, Section 11.7.4 를 참고하시기 바랍니다.

있다면, 그건 여러분의 쓰레드의 수행중에 보일거라는 겁니다.

그렇다고는 하나, 여러분의 관심을 최선의 곳에 집중하도록 더 잘 도울 도구들이 여럿 있습니다. 두개의 대중적인 선택은 `gprof` 와 `perf` 입니다. 단일 프로세스 프로그램에서 `perf` 를 사용하기 위해선 여러분의 커맨드 앞에 `perf record` 를 붙이고 그 커맨드가 완료된 후 `perf report` 라고 입력하세요. 멀티 쓰레드 프로그램의 성능 디버깅 도구를 위한 상당한 작업이 진행중이며, 이는 이 어려운 일을 쉽게 만들어 줄 겁니다. 다시 말하지만, 시작을 위한 좋은 지점은 Brendan Gregg의 블로그입니다.¹⁵

11.7.3 Differential Profiling

여러분이 매우 큰 시스템을 사용하고 있지 않다면 확장성 문제는 항상 분명하진 않을 겁니다. 그러나, 훨씬 작은 시스템을 사용할 때에도 임박한 확장성 문제를 파악하는게 가능할 때도 있습니다. 그러기 위한 한가지 기법은 차이점 프로파일링이라 불립니다.

아이디어는 여러분의 워크로드를 두개의 다른 조건 집합에서 수행하는 겁니다. 예를 들어, 그걸 두개의 CPU에서 돌려보고, 네개 CPU에서도 돌려볼 수도 있습니다. 여러분은 그대신 시스템에 주어지는 부하, 네트워크 어댑터 갯수, 대용량 저장 기기 갯수, 등을 변화시켜 줄 수도 있을 겁니다. 그리고나서 두 수행의 프로파일링 결과를 수집하여 연관된 프로파일링 측정 결과를 수학적으로 결합합니다. 예를 들어, 여러분의 주요 걱정이 확장성이라면 연관된 측정의 비율을 취하고, 그걸 내림차순으로 정렬할 수 있겠습니다. 그럼 확장성 문제의 주 용의자가 그 리스트의 꼭대기에 나올겁니다 [McK95, McK99].

`perf` 같은 어떤 도구들은 내장된 차이점 프로파일링 지원이 있습니다.

11.7.4 Microbenchmarking

マイクロベンチマーク는 어떤 알고리즘 또는 데이터 구조가 소프트웨어의 큰 봄통에 내포될 가치가 있는지를 판단하기 위한 깊은 평가를 위해 유용합니다.

マイクロベン치マー크에 대한 한가지 흔한 접근법은 시간을 측정하고 테스트 되는 코드를 몇번 반복해 수행한 후 시간을 다시 측정하는 겁니다. 이 두 시간 사이의 차이를 반복 횟수로 나누면 그 테스트 되는 코드를 수행하는데 필요한 시간이 됩니다.

불행히도, 이 방법은 다음을 포함한 여러 오류가 포함될 수 있습니다:

1. 시간 측정의 오버헤드가 포함될 수 있습니다. 이 오류의 원천은 반복 횟수를 늘림으로써 임의의 작은 값으로 줄어들 수 있습니다.
2. 테스트의 처음 몇번의 반복은 캐시 미스나 (더 나쁘게는) page fault를 일으켜서 측정값을 크게 만들 수 있습니다. 이 오류의 원천 역시 반복 횟수를 늘려서 줄일 수 있으며, 측정을 시작하기 전에 몇번의 워밍업 반복을 수행함으로써 완전히 제거할 수 있는 경우도 많습니다. 대부분의 시스템은 특정 프로세스가 page fault를 일으켰는지 파악하기 위한 방법을 가지고 있으며, 여러분은 성능이 영향받은 수행을 제거하기 위해 이 방법을 사용해야 합니다.
3. 예를 들면 무작위 메모리 오류 같은 어떤 종류의 간섭은 너무 가끔 일어나서 테스트의 여러 반복 집합을 수행함으로써 처리할 수 있습니다. 간섭의 수준이 통계적으로 심각하다면, 모든 성능이 특이하게 나온 결과는 통계적으로 제거될 수 있습니다.
4. 이 테스트의 모든 반복은 시스템의 다른 활동에 간섭받았을 수도 있습니다. 간섭의 원천은 다른 어플리케이션, 시스템 유ти리티와 daemon, 기기 인터럽트, 펌웨어 인터럽트 (시스템 관리 인터럽트 또는 SMI를 포함), 가상화, 메모리 오류, 그 외에도 여러가지가 포함됩니다. 이 간섭의 원천이 무작위적으로 발생한다고 가정하면, 그것들의 영향은 반복 횟수를 줄임으로써 제거될 수 있습니다.
5. 열처리가 확장성을 낮출 수 있는데 CPU 활동을 늘리는 것은 열 발생을 늘리며, 적절한 냉방장치가 없는 시스템에서는 (대부분이 그렇습니다!) 이게 CPU 갯수가 늘어날수록 CPU 속도를 낮추는 결과를 초래할 수 있습니다.¹⁶ 물론, 여러분이 어떤 어플리케이션을 그것이 제품 환경에서 동작할 때 예상되는 행동을 평가하기 위해 테스트 한다면 그런 열처리는 그저 마주해야 할 사실일 뿐입니다. 그렇지 않고 여러분이 이론적 확장성에 관심있다면 적절한 냉방장치를 갖춘 시스템을 사용하거나 그 냉방장치가 처리할 수 있는 수준까지 CPU 속도를 낮추세요.

첫번째와 네번째 간섭은 서로 충돌하는 조언을 제공하는데, 우리가 실제 세계에 살고 있다는 하나의 징표입니다. 다음 섹션들은 이 충돌을 처리하는 방법들을 알아봅니다.

Quick Quiz 11.23: 예를 들면 캐시와 메모리 배치 사이의 상호작용 같은 다른 오류의 원천들은 어떤가요?

¹⁵ <http://www.brendangregg.com/blog/>

¹⁶ 적절한 냉방장치가 있는 시스템은 게임 시스템처럼 보이는 경향이 있습니다.

다음 섹션은 이 측정 오류를 다루는 방법들을 이야기하는데, Section 11.7.5은 어떤 형태의 간섭을 막는 격리 기법을 다루고, Section 11.7.6은 어떤 간섭으로 오염된 측정 결과를 제거하기 위한 간섭 파악 방법을 다룹니다.

11.7.5 Isolation

리눅스 커널은 한 그룹의 CPU를 바깥의 간섭에서 격리화 하는 여러 방법을 제공합니다.

첫째로, 다른 프로세스, 쓰레드, 그리고 태스크로부터의 간섭을 알아봅시다. POSIX `sched_setaffinity()` 시스템콜은 대부분의 태스크를 특정 CPU 집합으로부터 떨어뜨리고 여러분의 테스트를 같은 그룹에 묶이게 하기 위해 사용될 수 있습니다. 리눅스에 있는 사용자 단계의 `taskset` 커맨드가 같은 목적으로 사용될 수 있습니다만, `sched_setaffinity()` 와 `taskset` 둘 다 높은 권한을 필요로 합니다. 리눅스에 있는 control groups (`cgroups`) 가 이 목적으로 사용될 수 있습니다. 이 접근법은 간섭을 줄이는데 상당히 효과적이며 많은 경우 충분합니다. 그러나, 이 방법은 한계가 있는데, 예를 들어 이 방법은 시스템 유지를 위해 사용되는 per-CPU 커널 쓰레드에는 아무일도 못합니다.

Per-CPU 커널 쓰레드로부터의 간섭을 막는 한 가지 방법은 여러분의 테스트를 높은 리얼타임 우선 순위로 돌리는 것으로, 예를 들면 POSIX `sched_setscheduler()` 시스템콜을 사용할 수 있습니다. 하지만 여러분이 이걸 하면 여러분은 암묵적으로 무한 반복을 막는 책임을 진다는 것을 의미하는데, 그러지 않으면 여러분의 테스트가 커널의 일부분을 동작하지 못하게 하기 때문입니다. 이는 스파이더맨 원칙의 한 예입니다: “큰 힘에는 큰 책임이 따른다.” 그리고 기본 리얼타임 처리 구성이 그런 문제를 종종 처리하지만, 그게 여러분의 리얼타임 쓰레드가 데드라인을 놓치게 할 수도 있습니다.

이 방법들은 프로세스, 쓰레드, 그리고 태스크로부터의 간섭을 크게 줄이거나 어쩌면 제거조차 할 수 있습니다. 하지만, 최소한 쓰레드 기반 인터럽트의 부재 시에는 기기 인터럽트로부터의 간섭은 막지 못합니다. 리눅스는 인터럽트 벡터당 하나씩 숫자로 이름지어진 디렉토리를 갖는 `/proc/irq` 디렉토리를 통한 쓰레드 기반 인터럽트의 일부 조정을 허락합니다. 각 숫자 이름 디렉토리는 `smp_affinity` 와 `smp_affinity_list` 를 갖습니다. 충분한 권한을 가지고 있다면 여러분은 특정 CPU 집합으로의 인터럽트를 제한하기 위해 이 파일들에 값을 쓸 수 있습니다. 예를 들어, “`echo 3 > /proc/irq/23/smp_affinity`” 또는 “`echo 0-1 > /proc/irq/23/smp_affinity_list`”는 주어진 권한이 충분하면 벡터 23의 인터럽트를 CPU 0과 1에 국한시킬 겁니다. 여러분 시스템의 인터럽트 벡터 리스트, 각 CPU에 의해 각 인터럽트가 얼마나 많이 처리되었는

지, 그리고 어떤 기기가 각 인터럽트 벡터를 사용하는지 알기 위해 “`cat /proc/interrupts`”를 사용할 수 있습니다.

여러분의 시스템의 모든 인터럽트 벡터를 위한 비슷한 커맨드를 돌리는 것은 인터럽트를 CPU 0과 1에 국한시켜서 나머지 CPU들은 간섭으로부터 자유롭게 할 겁니다. 또는 간섭으로부터 거의 자유롭게요, 어쨌건. 각 CPU에서 일어나는 `scheduling-clock` 인터럽트가 사용자 모드로 발생함이 드러났습니다.¹⁷ 여기 더해서 여러분은 여러분이 인터럽트를 몰아넣은 CPU 집합은 그 부하를 감당하기 충분함을 보장해야 합니다.

하지만 이는 테스트 되는 같은 운영체제 인스턴스에서 수행되는 프로세스와 인터럽트만 처리합니다. 예를 들면 KVM을 수행하는 리눅스와 같은, 그 스스로 하이퍼바이저 위에서 수행되는 guest OS에서 테스트를 하고 있다고 생각해 봅시다. 이론상 여러분은 guest OS 단계에서 할 수 있는 기법을 하이퍼바이저 단계에 적용할 수 있으나, 하이퍼바이저 단계 오퍼레이션들은 인가된 사람에게만 허용되는게 흔합니다. 또한, 이 기법들 중 어느것도 펌웨어 단계 간섭에는 적용되지 않습니다.

Quick Quiz 11.24: 테스트 되는 코드를 격리하기 위해 추천된 기법들 또한 그 코드의 성능에 영향을 끼치지 않을까요? 특히나 그게 더 큰 어플리케이션 내에서 돌아간다면요.



물론, 흥미로운 행동을 만드는게 정말 그 간섭이라면, 여러분은 오히려 간섭이 잘 일어나게 해야 할텐데, 이 경우엔 그걸 막을 수 없는게 문제가 되지 않습니다. 하지만 정말 간섭이 없는 측정을 필요로 한다면, 간섭을 막는 대신 다음 섹션에 설명되는 것처럼 간섭을 파악해야 할수도 있습니다.

11.7.6 Detecting Interference

간섭을 막을 수 없다면, 어쩌면 여러분은 그걸 탐지하고 거기 영향받은 테스트 수행으로부터의 결과들을 제거할 수 있을 겁니다. Section 11.7.6.1은 그런 제거를 위한 추가적 측정을 설명하며, Section 11.7.6.2은 통계 기반 제거를 설명합니다.

11.7.6.1 Detecting Interference Via Measurement

리눅스를 포함한 많은 시스템이 어떤 형태의 간섭이 일어났는지 탐지하기 위한 방법들을 제공합니다. 예를 들어, 프로세스 기반의 간섭은 컨텍스트 스위치를 초래하는데, 리눅스 기반 시스템에서는 `/proc/<PID>/sched`

¹⁷ 하나의 수행가능 태스크만 있는 CPU에서는 `scheduling-clock` 인터럽트가 불능화 되게 하는 `NO_HZ_FULL` adaptive-ticks 프로젝트를 Frederic Weisbecker가 이끌고 있습니다. 2021년 기준으로 이는 거의 완료되었습니다.

Listing 11.1: Using `getrusage()` to Detect Context Switches

```

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 ru1;
8     struct rusage ru2;
9
10    if (getrusage(RUSAGE_SELF, &ru1) != 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 (ru1.ru_nvcsw == ru2.ru_nvcsw &&
20            ru1.ru_nivcsw == ru2.ru_nivcsw);
21 }

```

의 `nr_switches` 를 통해 알 수 있습니다. 비슷하게, 인터럽트 기반의 간섭은 `/proc/interrupt` 파일을 통해 알 수 있습니다.

파일들을 열고 읽는 것은 낮은 오버헤드가 아니며, Listing 11.1에 보인 것처럼 `getrusage()` 시스템콜을 통해 특정 쓰레드의 컨텍스트 스위치 횟수를 알 수 있습니다. 이 시스템콜은 또한 minor page fault 와 (`ru_minflts`) major page fault (`ru_majflts`) 횟수를 파악하는데에도 사용될 수 있습니다.

불행히도, 메모리 에러와 펌웨어 간섭을 탐지하는건 상당히 시스템에 따라 다른데, 가상화에 따른 간섭 탐지에서도 마찬가지입니다. 제거가 탐지보다 낮고 탐지가 통계보다 낮지만, 통계를 사용해야만 할 때도 존재하는데, 이 주제를 다음 섹션에서 다룹니다.

11.7.6.2 Detecting Interference Via Statistics

모든 통계 분석은 데이터에 대한 가정에 기반하며, 성능 마이크로벤치마크는 종종 다음 가정을 갖습니다:

1. 더 작은 측정이 큰 측정보다 정확할 확률이 높다.
2. 좋은 데이터의 측정 불확정도는 알려져 있다.
3. 테스트 수행의 어느정도 부분은 좋은 데이터를 만든다.

더 작은 측정이 큰 측정보다 정확할 확률이 높다는 사실은 측정을 올림순으로 정렬하면 더 생산적이 될 것을 암시합니다.¹⁸ 측정의 불확정도가 알려져 있다는 사실은 이 정도의 불확정도 아래의 측정을 받아들일 수 있게 합니다. 간섭의 효과가 이 불확정도보다 크다면, 이는

¹⁸ 오래된 말을 빌리자면, “정렬부터 하고 질문은 나중에 하라.”

나쁜 데이터의 제거를 쉽게 할 겁니다. 마지막으로, 어느 정도 (예를 들어, 3분의 1)는 좋은 결과일 것이라 가정될 수 있다는 사실은 정렬된 리스트의 첫번째 부분을 생각 없이 받아들일 수 있게 하며, 그럼 이 데이터는 가정된 측정 오류를 넘어서는, 측정된 데이터의 자연적 편차를 추정하는데 사용될 수 있습니다.

방법은 이 정렬된 리스트의 앞부분의 특정 갯수의 원소들을 취하고 일반적인 원소간 차이를 사용하는 것으로, 그 갯수는 사용 가능한 값들의 최대 한계를 얻기 위해 리스트의 원소의 갯수로 곱해질 수도 있을 겁니다. 그럼 이 알고리즘은 반복적으로 이 리스트의 다음 원소를 고려합니다. 이게 앞의 최대 한계 아래라면, 그리고 앞의 원소와 다음 원소 사이의 거리가 평균 원소간 거리보다 멀지 않다면 리스트의 이 부분은 받아들여지며, 그만 다음 원소가 받아들여지고 이 과정이 반복됩니다. 그렇지 않다면, 이 리스트의 나머지 것들은 제거됩니다.

Listing 11.2은 이 방법을 구현하는 간단한 `sh/awk` 스크립트를 보입니다. 입력은 `x` 값을 띠는 임의의 긴 리스트의 `y` 값들로 구성되며, 출력은 각 입력 줄을 위한 한 줄로 구성되는데 다음 필드를 갖습니다:

1. `x` 값.
2. 선택된 데이터의 평균 값.
3. 선택된 데이터의 최소값.
4. 선택된 데이터의 최대값.
5. 선택된 데이터 항목의 갯수.
6. 입력 데이터 항목의 갯수.

이 스크립트는 다음의 선택적 인자 세개를 받습니다:

--divisor: 리스트를 나눌 조각의 갯수로, 예를 들면 이 값이 4라면 데이터 원소의 앞쪽 4분의 1은 좋을 것으로 가정됩니다. 기본값은 3입니다.

--relerr: 상대적 측정 오류값. 이 스크립트는 이 오류 값보다 작은 값들간의 차이는 모든 목적과 의도에 동일하다고 여깁니다. 기본값은 0.01로, 1%와 같습니다.

--trendbreak: 데이터의 추세를 깨뜨리는 원소간 거리의 비율. 예를 들어, 지금까지 받아들여진 데이터 간의 평균거리가 1.5라면, 그리고 이 추세 깨뜨림 비율이 2.0이라면, 그리고 다음 데이터 값이 마지막 값에서 최소 3.0 넘게 다르다면, 이는 추세의 깨짐을 형성합니다. (물론 이 상대적 오류가 3.0보다 크지 않다면, “깨짐”은 무시됩니다.)

Listing 11.2: Statistical Elimination of Interference

```

1 div=3
2 rel=0.01
3 tre=10
4 while test $# -gt 0
5 do
6   case "$1" in
7     --divisor)
8     shift
9     div=$1
10    ;;
11   --relerr)
12   shift
13   rel=$1
14   ;;
15   --trendbreak)
16   shift
17   tre=$1
18   ;;
19   esac
20   shift
21 done
22
23 awk -v divisor=$div -v relerr=$rel -v trendbreak=$tre '{
24   for (i = 2; i <= NF; i++)
25     d[i - 1] = $i;
26   asort(d);
27   i = int((NF + divisor - 1) / divisor);
28   delta = d[i] - d[1];
29   maxdelta = delta * divisor;
30   maxdelta1 = delta + d[i] * relerr;
31   if (maxdelta1 > maxdelta)
32     maxdelta = maxdelta1;
33   for (j = i + 1; j < NF; j++)
34     if (j <= 2)
35       maxdiff = d[NF - 1] - d[1];
36     else
37       maxdiff = trendbreak * (d[j - 1] - d[1]) / (j - 2);
38     if (d[j] - d[1] > maxdelta && d[j] - d[j - 1] > maxdiff)
39       break;
40   }
41   n = sum = 0;
42   for (k = 1; k < j; k++) {
43     sum += d[k];
44     n++;
45   }
46   min = d[1];
47   max = d[j - 1];
48   avg = sum / n;
49   print $1, avg, min, max, n, NF - 1;
50 }'

```

Listing 11.2의 라인 1-3는 이 패러미터들의 기본값을 설정하고, 라인 4-21는 이 패러미터에 대한 커맨드 라인의 덮어쓰기를 처리합니다. 라인 23에서의 awk 수행은 divisor, relerr, 그리고 trendbreak 변수를 sh의 것들과 맞춥니다. 일반적인 awk 방식에서, 라인 24-50는 각 입력 줄별로 수행됩니다. 라인 24와 25 사이의 반복문은 입력 y 값을 d 배열에 복사하는데, 라인 26가 증가 순서로 정렬을 하는 배열입니다. 라인 27는 divisor를 적용하고 반올림 함으로써 믿을 수 있는 y 값을 계산합니다.

라인 28-32는 y 값의 최대값의 아래쪽 한계인 maxdelta를 계산합니다. 여기서, 라인 29는 데이터의 믿어지는 영역을 넘어서는 값의 차이를 divisor로

곱하는데, 이는 믿어지는 영역의 값들의 차이를 전체 y 값 집합으로 외삽합니다. 그러나, 이 값은 상대적 오류보다는 훨씬 작을 수도 있으므로, 라인 30은 절대적 오류값을 ($d[i] * relerr$) 구하고 이를 데이터의 믿어지는 영역의 차이인 delta에 더합니다. 라인 31와 32는 이 두 값 중 큰 값을 구합니다.

라인 33-40의 반복문의 각 수행은 좋은 데이터 집합에 다른 데이터 값을 더하려 시도합니다. 라인 34-39는 추세 깨뜨림 차이를 계산하는데, 라인 34는 우리가 아직 추세를 계산하기 충분한 값을 갖고 있지 않다면 이 한계를 불능화 시키며, 라인 37는 trendbreak를 좋은 데이터 집합의 데이터 값들의 쌍들 사이 차이값의 평균으로 곱합니다. 라인 38가 후보 데이터 값이 최대 값의 하한선을 (maxdelta) 넘어서고 이 후보 데이터 값과 그 앞의 것 사이의 차이가 추세 깨뜨림 차이 (maxdiff)를 넘어서면 파악하면, 라인 39가 이 반복문을 나갑니다: 우린 모든 좋은 데이터 집합을 얻었습니다.

라인 41-49는 이어서 통계를 계산하고 출력합니다.

Quick Quiz 11.25: 이 방법은 그저 이상해요! 왜 통계 수업에서 배운대로 중간값과 표준편차를 이용하지 않죠?



Quick Quiz 11.26: 하지만 믿어지는 데이터 그룹의 모든 y 값이 0이면 어떡하죠? 그건 이 스크립트가 모든 0이 아닌 값을 제거하게 하지 않을까요?



통계적 간섭 탐지가 유용할 수 있지만, 마지막 수단으로만 사용되어야 합니다. 간섭을 차단하는게 최선이고 (Section 11.7.5), 그게 불가능하다면 측정을 통해 간섭을 탐지하십시오 (Section 11.7.6.1).

11.8 Summary

To err is human! Stop being human!!!

Ed Nofziger

검증은 결코 정확한 과학이 되지 못하겠지만, 조직화된 접근은 여러분이 여러분의 일을 위해 올바른 검증 도구를 선택하는 걸 도와서 Figure 11.6에 그려진 것과 같은 상황을 막아줄 것이기에 큰 도움이 되어줄 겁니다.

핵심 선택은 통계입니다. 이 챕터에 설명된 방법이 대부분의 경우 매우 잘 동작하겠지만, Halting 문제 [Tur37, Pul00] 덕분에 그것들도 나름의 한계가 있습니다. 우리에겐 다행스럽게도, 우린 어떤 프로그램이 정지할 것인지에 대해서만이 아니라 Section 11.7에서 이야기된 것처럼 정지하기 전 얼마나 오래 동작할 것인가에 대해서도 예측해볼 수 있는 특수한 경우가 무척 많습니다.

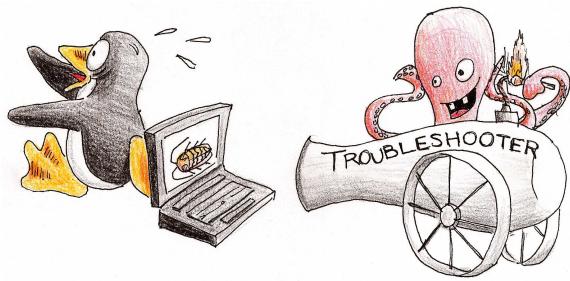


Figure 11.6: Choose Validation Methods Wisely

더 나아가서, 어떤 프로그램이 올바르게도 그렇지 않게도 동작할 수 있는 경우, 우린 시간 중 얼마큼의 분량이 올바르게 동작하는지 Section 11.6에서 설명된 것처럼 예측해 볼 수 있습니다.

그러나, 이런 예측의 안정성을 생각하지 않는 것은 무모한 용기입니다. 어쨌건, 우린 코드와 데이터 구조의 거대한 복잡도를 하나의 숫자로 요약하고 있습니다. 그런 무모한 용기로부터 굉장히 많은 경우 벗어날 수 있다해도, 모든 코드와 데이터를 추상화 하는 것은 때때로 심각한 문제를 일으킬 겁니다.

한가지 가능한 문제는 가변성으로, 반복된 수행이 굉장히 다른 결과를 내놓는 경우입니다. 이 문제는 종종 표준편차를 사용해 처리됩니다만, 거대하고 복잡한 프로그램의 동작을 두개의 숫자로 요약하는 것은 하나의 숫자를 사용하는 것만큼이나 용감한 행위입니다. 컴퓨터 프로그래밍에서, 놀라운 일은 중간값 또는 중간값과 표준편차를 사용하는게 종종 충분하다는 겁니다. 그렇다고 하나, 보장은 없습니다.

가변성의 원인 중 하나는 오염변인입니다. 예를 들어, 링크드 리스트 탐색에 소요된 CPU 시간은 리스트의 길이에 의존적일 겁니다. 매우 다른 길이의 리스트에서의 수행 결과를 가지고 평균을 내는 것은 유용하지 않을 것이며, 중간값에 표준편차를 더하는 것도 크게 낫지는 않을 겁니다. 해야할 올바른 일은 리스트 길이를 제어하는 것으로, 그 길이를 하나로 정하거나 CPU 시간을 리스트 길이에 따른 함수로 측정하는 겁니다.

물론, 이 조언은 여러분이 오염변인을 인지하고 있다는 가정을 하고 있는데, Murphy는 그렇지 않을 거라 말합니다. 저는 냉방장치 (구동 시에 상당한 전력을 사용하며 따라서 컴퓨터에 공급되는 전압을 순간적으로 너무 낮아지게 하며 그 결과 가끔은 실패를 유발함), 캐쉬 상태 (성능에 이상한 변화를 초래함), I/O 오류 (디스크 오류, 패킷 손실, 중첩된 이더넷 MAC 주소), 그리고 심지어 급속 잡수 (원래는 높은 정밀도의 음파 위치 탐지와 길안내를 위해 사용되어야 하는 응답기들에 영향을 끼침) 만큼이나 다변하는 오염변수들을 가진 프로젝트에 참여된 적이 있습니다. 그리고 이는 충분한 수면이

그렇게나 효과적인 디버깅 도구인 이유 중 하나일 뿐입니다.

요약하자면, 검증은 항상 시스템의 행동에 대한 어떤 측정을 필요로 합니다. 유용해지기 위해, 이 측정은 시스템의 상당한 요약이어야만 하는데 잘못된 것이었을 수 있음을 의미합니다. 그러니 흔히 말하듯, “조심하세요. 거기 있는건 실제 세계입니다.”

하지만 2017년 기준으로 전세계에 200억개 이상의 인스턴스가 수행중이라 여겨지는 리눅스 커널을 검증하고 있다면 어떨까요? 이 경우, 하나의 시스템에서 백만년에 한번 발생하는 버그는 설치된 전체 시스템에서 하루에 40번도 넘게 발생할 겁니다. 이 버그를 한시간에 한번 발생할 50% 확률을 갖는 테스트를 위해선 그 버그의 발생 확률을 10의 10제곱배 이상 증가시켜야 할 것이며, 이는 오늘날의 테스트 방법론에 커다란 도전이 됩니다. 가끔은 그런 상황에 좋은 영향을 끼칠 수 있는 중요한 도구가 정형 검증으로, 다음 챕터, 더 정확하게는 Section 17.4의 주제입니다.

검증 계획 선택, 그걸 테스트 하는 것, 정형 검증, 또는 모두가 Section 12.7에서 다루어집니다.

Beware of bugs in the above code; I have only proved it correct, not tried it.

Donald Knuth

Chapter 12

Formal Verification

병렬 알고리즘은 작성하기도 어렵지만 디버깅 하기는 더 어렵습니다. 테스트는 필수적이지만 치명적인 경주 조건이 극단적으로 낮은 발생 확률을 가질 수도 있기 때문에 충분치 못합니다. 정확성 증명은 가치있을 수 있지만 종국에는 원래의 알고리즘 만큼이나 사람의 오류에 취약합니다. 또한, 정확성 증명은 여러분의 가정, 요구사항 상의 부족함, 아래의 소프트웨어와 하드웨어 기능에 대한 잘못된 이해, 또는 여러분이 증명을 구성하는데 있어 생각치 않은 오류 등에 있는 오류는 찾을 거라 예상될 수 없습니다. 이는 정형적 방법들이 테스트를 대체할 수는 없음을 의미합니다. 그러나, 정형적 방법들은 여러분의 검증 도구상자에 가치있는 추가 물품이 될 수 있습니다.

모든 경주 조건을 어떻게든 찾아내는 도구를 갖는 건 무척 도움이 될 겁니다. 그런 도구가 여럿 존재하는데, 예를 들어 Section 12.1 는 범용 상태 공간 탐색 도구인 Promela 와 Spin 을 소개하고, Section 12.2 은 비슷하게 특수 목적 ppcmem 도구를 소개하며, Section 12.3 는 공리적 접근법의 예를 알아보며, Section 12.4 는 간단히 SAT solver 들을 살펴보며, Section 12.5 는 짧게 stateless 모델 검사기를 알아보고, Section 12.6 는 병렬 알고리즘을 검증하기 위한 정형적 검증 도구들의 사용을 요약하며, 마지막으로 Section 12.7 에서는 얼마나 많은 그리고 어떤 종류의 검증을 주어진 소프트웨어 프로젝트에 적용해야 할지 이야기 합니다.

12.1 State-Space Search

Follow every byway / Every path you know.

“Climb Every Mountain”, Rodgers & Hammerstein

이 섹션은 많은 종류의 멀티 쓰레드 코드의 전체 상태 공간 탐색을 해낼 수도 있는 범용 Promela 와 Spin 도구들을 소개합니다. 이것들은 데이터 통신 프로토콜을 검증하기 위해 사용됩니다. Section 12.1.1 은 Promela 와 Spin 을

소개하는데 어토믹과 어토믹하지 않은 값 증가를 검증하는 두개의 워밍업 연습을 포함합니다. Section 12.1.2 은 Promela 의 사용법을 소개하는데 Promela 의 문법의 C 의 그것과 비교와 예제 커멘드 라인들을 포함합니다. Section 12.1.3 은 락킹을 검증하기 위해 Promela 가 어떻게 사용되는지 알아보고, 12.1.4 는 “QRCU” 라 이름지어진 일반적이지 않은 RCU 구현을 검증하는데 Promela 를 사용해 보며, 마지막으로 Section 12.1.5 은 RCU 의 dyntick-idle 구현 초기 버전에 Promela 를 적용합니다.

12.1.1 Promela and Spin

Promela 는 프로토콜을 검증하기 위해 설계된 언어이지만 작은 병렬 알고리즘을 검증하는데에도 사용될 수 있습니다. 여러분은 여러분의 알고리즘과 정확성 제한들을 C 같은 Promela 언어로 재작성할 수 있으며 Spin 을 사용해 그것을 컴파일하고 수행할 수 있는 C 프로그램으로 변환할 수 있습니다. 그 결과 프로그램은 여러분의 알고리즘의 전체 상태 공간 탐색을 해내는데, 여러분이 Promela 프로그램에 짜넣은 단정들을 검증하거나 반례를 찾아냅니다.

이 전체 공간 탐색은 굉장히 강력할 수 있지만 또한 양날의 검이 될 수 있습니다. 여러분의 알고리즘이 너무 복잡하거나 여러분의 Promela 구현이 부주의 하다면, 메모리에 들어가는 것보다 많은 상태가 있을 수 있습니다. 더 나아가서, 충분한 메모리가 있더라도 이 상태 공간 탐색은 예상되는 우주의 수명보다 긴 시간동안 수행될 수도 있습니다. 따라서, 이 도구는 작지만 복잡한 병렬 알고리즘을 위해 사용하십시오. 이를 적당한 크기의 알고리즘에 (전체 리눅스 커널은 놔두세요) 생각없이 적용하는 것은 나쁜 결과를 초래할 겁니다.

Promela 와 Spin 은 <https://spinroot.com/spin/whatispin.html> 에서 다운로드 받을 수 있습니다.

위의 사이트는 또한 Gerard Holzmann 의 Promela 와 Spin 에 대한 훌륭한 교재 [Hol03] 로의 링크와 검색 가능한 온라인 레퍼런스들을 <https://www.spinroot.com/spin/Man/index.html> 에서 제공합니다.

Listing 12.1: Promela Code for Non-Atomic Increment

```

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 }

```

이 섹션의 나머지는 병렬 알고리즘을 디버깅하기 위해 어떻게 Promela를 사용하는지, 간단한 예에서 시작해 복잡한 사용법으로 넘어가면서 알아봅니다.

12.1.1.1 Warm-Up: Non-Atomic Increment

Listing 12.1은 어토믹하지 않은 값 증가에서 초래되는 교재에 나오는 경주 조건을 선보입니다. 라인 1은 수행할 프로세스의 수를 정의하고 (상태 공간에의 영향을 보기 위해 우린 이 값을 변화시켜 볼 겁니다), 라인 3은 카운터를 정의하며, 라인 4는 라인 29-39에 나오는 단정문을 구현하는데 사용됩니다.

라인 6-13은 어토믹하지 않게 카운터를 증가시키는 프로세스 하나를 정의합니다. 인자 *me*는 프로세스 수로, 코드의 뒤쪽에 있는 초기화 블록에서 설정됩니다. 간단한 Promela 선언문들은 각각 어토믹한 걸로 가정되므로, 우린 이 값을 증가를 라인 10-11의 두개 선언문으로 조개야 합니다. 라인 12에서의 값을 할당은 이 프로세스의 완료를 표시합니다. Spin 시스템은 상태 공간을 모든 가능한 상태 순서들을 포함해 완전히 탐색하므로,

전통적인 스트레스 테스트에 사용되는 반복문은 필요 없습니다.

라인 15-40는 초기화 블록으로, 가장 먼저 수행됩니다. 라인 19-28는 실제로 초기화를 하며, 라인 29-39는 단정을 수행합니다. 둘 다 불필요한 상태 공간 증가를 막기 위해 원자적 블록으로 정의됩니다: 이것들은 알고리즘의 부분이 아니므로, 이것들을 원자적으로 만들도록 써 검증 범위를 넓지는 않습니다.

라인 21-27의 do-od 구성은 C에서의 case label 표현을 허용하는 switch 문을 담는 for (;;) 반복문으로 생각될 수 있는 Promela 반복문을 구현합니다. 조건 블록 (:: 접두어로 표시됨)은 비결정적으로 스캔됩니다만, 이 경우에는 한번에 단 하나의 조건만 잡힐 수 있습니다. 이 do-od의 라인 22-25에 있는 첫번째 블록은 *i* 번째 값 증가 쓰레드의 진행 셀을 초기화하고, *i* 번째 값 증가 쓰레드의 프로세스를 수행하며, 이어서 변수 *i*의 값을 증가시킵니다. 라인 26에 있는 do-od의 두번째 블록은 이 프로세스들이 시작되고 나면 이 반복문을 빠져나갑니다.

라인 29-39의 어토믹 블록도 진행 카운터의 합을 구하는 비슷한 do-od 반복문을 갖습니다. 라인 38의 assert() 문은 모든 프로세스가 완료되었다면 모든 카운트가 올바르게 기록되었을 것을 검증합니다.

여러분은 이 프로그램을 다음과 같이 빌드하고 수행할 수 있습니다:

```

spin -a increment.spin      # Translate the model to C
cc -DSAFETY -o pan pan.c  # Compile the model
./pan                      # Run the model

```

이는 Listing 12.2에 보인 것과 같은 출력을 낼 겁니다. 첫번째 줄은 우리의 단정이 위배되었음을 (어토믹하지 않은 값 증가를 했으니 예상된 바입니다!) 말합니다. 두번째 줄은 이 단정이 어떻게 위배되었는지에 대한 설명이 *trail* 파일에 쓰여 있음을 말합니다. “Warning” 줄은 우리의 모델이 완벽하지 않았음을 다시 말합니다. 두번째 문단은 진행된 상태 탐색의 종류를 설명하는데 이 경우 단정 위배와 올바르지 않은 종료 상태입니다. 세 번째 문단은 상태 크기 통계를 보입니다: 이 작은 모델은 45개의 상태만을 가졌습니다. 마지막 라인은 메모리 사용량을 보입니다.

trail 파일은 다음과 같이 사람이 읽을 수 있는 형태로 변환될 수 있습니다:

```

spin -t -p increment.spin

```

이는 Listing 12.3에 보인 것과 같은 출력을 냅니다. 보이듯, init 블록의 첫번째 부분은 두개의 값 증가 프로세스를 만들었는데, 둘 다 카운터를 먼저 읽어들이고, 값을 증가한 후 저장하여, 수를 잊습니다. 단정문이 이어서 실행되고 전역 상태가 표시됩니다.

Listing 12.2: Non-Atomic Increment Spin Output

```

pan:1: assertion violated
  ((sum<2)|| (counter==2)) (at depth 22)
pan: wrote increment.spin.trail

(Spin Version 6.4.8 -- 2 March 2018)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
  never claim           - (none specified)
  assertion violations + - (disabled by -DSAFETY)
  cycle checks          + - (disabled by -DSAFETY)
  invalid end states   + - (disabled by -DSAFETY)

State-vector 48 byte, depth reached 24, errors: 1
  45 states, stored
  13 states, matched
  58 transitions (= stored+matched)
  53 atomic steps
hash conflicts:          0 (resolved)

Stats on memory usage (in Megabytes):
  0.003  equivalent memory usage for states
         (stored*(State-vector + overhead))
  0.290  actual memory usage for states
128.000  memory used for hash table (-w24)
  0.534  memory used for DFS stack (-m10000)
128.730  total actual memory usage

```

Table 12.1: Memory Usage of Increment Model

# incrementers	# states	total memory usage (MB)
1	11	128.7
2	52	128.7
3	372	128.7
4	3,496	128.9
5	40,221	131.7
6	545,720	174.0
7	8,521,446	881.9

12.1.1.2 Warm-Up: Atomic Increment

이 예를 고치는 건 Listing 12.4 에 보인 것처럼 값 증가 프로세스의 몸체를 어토믹 블록에 두는 것처럼 쉽습니다. 어떤 사람은 간단히 두 쌍의 명령문을 `counter = counter + 1`로 바꿀 수도 있을 텐데, Promela 명령문은 어토믹 하기 때문입니다. 어떤 방법이든, 이 수정된 모델을 수행하는 것은 Listing 12.5 에 보인 것처럼 오류 없는 상태 공간 순회를 내놓습니다.

Table 12.1 은 모델된 값 증가 쓰레드의 수에 따른 (NUMPROCS 재정의에 의한) 상태 갯수와 사용된 메모리를 보입니다.

따라서 불필요하게 큰 모델을 수행하는 것은 장려되지 않는데, 882 MB 는 현대 데스크탑과 랙톱 기계에서의 한계 내이긴 합니다.

이 예를 머리 속에 새겨두고, Promela 모델을 분석하는데 사용되는 커맨드를 더 자세히 알아보고 더 정교한 예를 들여다 봅시다.

12.1.2 How to Use Promela

소스 파일 `qrcu.spin` 을 가지고, 다음과 같은 커맨드를 사용할 수 있습니다:

Given a source file `qrcu.spin`, one can use the following commands:

`spin -a qrcu.spin`

상태 기계를 전체 탐색하는 `pan.c` 파일을 만듭니다.

`cc -DSAFETY [-DCOLLAPSE] [-DMA=N] -o pan`

`pan.c`

생성된 상태 기계 탐색을 컴파일 합니다. `-DSAFETY` 는 여러분이 단정문들만을 가지고 있다면 (그리고 어쩌면 `never` 명령문을) 적합한 최적화를 생성합니다. 여러분이 liveness, fairness, 그리고 forward-progress 체크를 갖는다면, 여러분은 `-DSAFETY` 없이 컴파일을 해야할 수도 있습니다. 여러분이 사용할 수 있음에도 `-DSAFETY` 를 사용하지 않는다면 프로그램은 이를 알려줄 겁니다.

`-DSAFETY` 에 의해 생성되는 최적화는 상당히 속도를 향상시키므로 사용할 수 있다면 여러분은 그걸 사용해야 합니다. 여러분이 `-DSAFETY` 를 사용할 수 없는 상황 중 한 예는 `-DNP` 를 통해 livelock ("non-progress cycles" 라고도 알려져 있습니다) 을 검사할 때입니다.

옵션 `-DCOLLAPSE` 는 state vector compression 모드를 위한 코드를 생성합니다.

또 다른 옵션인 `-DMA=N` 은 느리지만 공격적인 상태 공간 메모리 압축 모드를 위한 코드를 생성합니다.

`./pan [-mN] [-wN]`

이는 실제로 상태 공간을 탐색합니다. 상태의 수는 매우 작은 상태 기계를 가지고도 수천만에 이를 수 있으므로 여러분은 큰 메모리의 기계를 필요로 할 겁니다. 예를 들어, 3개의 업데이트 쓰레드와 2개의 읽기 쓰레드를 갖는 `qrcu.spin` 은 `-DCOLLAPSE` 플래그를 가지고도 10.5 GB 메모리를 필요로 했습니다.

`./pan` 이 다음과 같이 말한다면, 여러분은 완벽한 탐색을 위해 `-mN` 옵션을 사용해 최대 깊이를 늘려야 합니다: "error: max search depth too small". 기본값은 `-m10000`입니다.

Listing 12.3: Non-Atomic Increment Error Trail

```

using statement merging
 1:  proc 0 (:init::1) increment.spin:21 (state 1) [i = 0]
 2:  proc 0 (:init::1) increment.spin:23 (state 2) [((i<2))]
 2:  proc 0 (:init::1) increment.spin:24 (state 3) [progress[i] = 0]
Starting incrementer with pid 1
 3:  proc 0 (:init::1) increment.spin:25 (state 4) [(run incrementer(i))]
 4:  proc 0 (:init::1) increment.spin:26 (state 5) [i = (i+1)]
 5:  proc 0 (:init::1) increment.spin:23 (state 2) [((i<2))]
 5:  proc 0 (:init::1) increment.spin:24 (state 3) [progress[i] = 0]
Starting incrementer with pid 2
 6:  proc 0 (:init::1) increment.spin:25 (state 4) [(run incrementer(i))]
 7:  proc 0 (:init::1) increment.spin:26 (state 5) [i = (i+1)]
 8:  proc 0 (:init::1) increment.spin:27 (state 6) [((i>=2))]
 9:  proc 0 (:init::1) increment.spin:22 (state 10) [break]
10:  proc 2 (incrementer:1) increment.spin:11 (state 1) [temp = counter]
11:  proc 1 (incrementer:1) increment.spin:11 (state 1) [temp = counter]
12:  proc 2 (incrementer:1) increment.spin:12 (state 2) [counter = (temp+1)]
13:  proc 2 (incrementer:1) increment.spin:13 (state 3) [progress[me] = 1]
14: proc 2 terminates
15:  proc 1 (incrementer:1) increment.spin:12 (state 2) [counter = (temp+1)]
16:  proc 1 (incrementer:1) increment.spin:13 (state 3) [progress[me] = 1]
17: proc 1 terminates
18:  proc 0 (:init::1) increment.spin:31 (state 12) [i = 0]
18:  proc 0 (:init::1) increment.spin:32 (state 13) [sum = 0]
19:  proc 0 (:init::1) increment.spin:34 (state 14) [((i<2))]
19:  proc 0 (:init::1) increment.spin:35 (state 15) [sum = (sum+progress[i])]
19:  proc 0 (:init::1) increment.spin:36 (state 16) [i = (i+1)]
20:  proc 0 (:init::1) increment.spin:34 (state 14) [((i<2))]
20:  proc 0 (:init::1) increment.spin:35 (state 15) [sum = (sum+progress[i])]
20:  proc 0 (:init::1) increment.spin:36 (state 16) [i = (i+1)]
21:  proc 0 (:init::1) increment.spin:37 (state 17) [((i>=2))]
22:  proc 0 (:init::1) increment.spin:33 (state 21) [break]
spin: increment.spin:39, Error: assertion violated
spin: text of failed assertion: assert(((sum<2)|| (counter==2)))
23:  proc 0 (:init::1) increment.spin:39 (state 22) [assert(((sum<2)|| (counter==2)))]
spin: trail ends after 23 steps
#processes: 1
          counter = 1
          progress[0] = 1
          progress[1] = 1
23:  proc 0 (:init::1) increment.spin:41 (state 24) <valid end state>
3 processes created

```

Listing 12.4: Promela Code for 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 }

```

Listing 12.5: Atomic Increment Spin Output

```

(Spin Version 6.4.8 -- 2 March 2018)
+ Partial Order Reduction

Full statespace search for:
    never claim           - (none specified)
    assertion violations + 
    cycle checks          - (disabled by -DSAFETY)
    invalid end states   + 

State-vector 48 byte, depth reached 22, errors: 0
    52 states, stored
    21 states, matched
    73 transitions (= stored+matched)
    68 atomic steps
hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):
    0.004   equivalent memory usage for states
             (stored*(State-vector + overhead))
    0.290   actual memory usage for states
    128.000  memory used for hash table (-w24)
    0.534   memory used for DFS stack (-m10000)
    128.730  total actual memory usage

unreached in proctype incrementer
    (0 of 5 states)
unreached in init
    (0 of 24 states)

```

-wN 옵션은 해쉬테이블 크기를 명시합니다. 전체 상태 공간 탐색을 위한 기본 값은 -w24 입니다.¹

여러분의 기계가 충분한 메모리를 가지고 있는지 확신치 못한다면, 한 윈도우에서 top 을 돌리고 다른 윈도우에서 ./pan 을 돌리세요. ./pan 윈도우에 포커스를 유지해서 필요하면 그 수행을 중지 시킬 수 있게 하세요. CPU 시간이 100 % 아래로 떨어지는 순간 ./pan 을 강제 종료시키세요. ./pan 을 수행하는 윈도우로부터 포커스를 제거했다면 윈도우 시스템이 다른 무언가를 하는데 충분한 메모리를 줘는데 많은 시간을 소요해 여러분을 기다려야 하게 만들 겁니다.

메모리 소모를 막는 다른 방법은 -DMEMLIM=N 커널 파일러 플래그입니다. -DMEMLIM=2000 은 그 최대 값을 2GB 로 만듭니다.

¹ Spin 버전 6.4.6 과 6.4.8 에서. 2011년 7월 10일자의 Spin 온라인 매뉴얼에서는 전체 탐색 모드를 위한 기본 값은 -w19 라고 이야기하는데, 실제 동작과 맞지 않습니다.

출력을 캡쳐해 두는 걸, 특히 여러분이 원격의 기계에서 작업 중이라면 잊지 마세요.

여러분의 모델이 forward-progress 체크를 포함한다면 여러분은 ./pan 에의 -f 커맨드라인 인자를 통해 “weak fairness” 를 활성화 시켜야 할 겁니다. 여러분의 forward-progress 검사가 accept 라벨을 포함한다면, -a 인자도 필요할 겁니다.

spin -t -p qrcu.spin

오류를 발견한 수행에 의한 trail 파일 결과물을 가지고 그 오류를 일으킨 단계들을 출력합니다. -g 플래그는 변경된 전역 변수들을 포함시키고 -l 플래그는 변경된 지역 변수들도 포함시킵니다.

12.1.2.1 Promela Peculiarities

모든 프로그래밍 언어가 비슷한 기반을 갖지만, Promela 는 C, C++, 또는 Java 로 코드를 짜는데 익숙한 사람들에게 조금 놀라울 겁니다.

1. C 에서, “;” 는 문장을 끝냅니다. Promela 에서는 그것들을 구분합니다. 다행히, 더 최신 버전의 Spin 은 “여분의” 세미콜론들을 더 포기했습니다.
2. Promela 의 루프 구조물인 do 문은 조건을 취합니다. 이 do 문은 if-then-else 반복문을 상당히 닮았습니다.
3. C 의 switch 문에서는 매칭되는 경우가 없을 때 전체 구분이 생략됩니다. Promela 의 것에서는 알아볼 수 있는 연관된 에러 메세지 없이 에러를 냅니다. 따라서, 오류 결과물이 잘못 없는 코드를 가리킬 때에는 여러분이 if 나 do 문에 조건을 남겨두지 않았는지 검사해 보세요.
4. C 에서 스트레스 테스트를 할 때, 어떤 사람들은 의심되는 오퍼레이션을 다른 것들 각각에 대해 바복적으로 경주시켜볼 겁니다. Promela 에서는 그 대신 하나의 경주를 만드는데, Promela 는 그 하나의 경주로부터 나올 수 있는 모든 가능한 경우를 탐색하기 때문입니다. 간혹 여러분은 Promela 에서 반복문을 짜야하기도 한데, 예를 들어 여러 오퍼레이션들이 겹치지만 그렇게 하는게 여러분의 상태 공간의 크기를 크게 증가시키는 경우입니다.

5. C 에서, 할 수 있는 가장 쉬운 일은 반복문의 진행과 종료를 추적하기 위해 반복문 카운터를 갖는 것입니다. Promela 에서, 반복문 카운터는 역병과 같이 막아져야 하는데 그게 상태 공간을 폭발시키기 때문입니다. 다른 한편, 변수들 중 어떤 것도 단조적으로 증가하거나 감소하지 않는 한 무한 루프는

Promela에서 단점을 갖지 않습니다—Promela는 그 반복문을 지나는 얼마나 많은 경우가 정말로 문제시 되는지 알아내고 자동으로 그 지점 이후의 수행을 제거할 겁니다.

6. C 고문 테스트 코드에서, 태스크별 제어 변수를 갖는게 종종 현명합니다. 그것들은 읽기 쉽고, 테스트 코드의 디버깅을 상당히 쉽게 합니다. Promela에서, 태스크별 제어 변수는 다른 대안이 없을 때에만 상요되어야 합니다. 이를 자세히 보기 위해, 다섯 개의 검증을 위한 태스크가 있고 각 태스크의 완료를 알리는 비트를 하나씩 사용한다고 해봅시다. 이는 32개의 상태를 만듭니다. 대조적으로, 간단한 카운터는 여섯개의 상태만을 가져서 다섯배 감소를 시킵니다. 이 다섯배의 규모는 여러분이 10GB 메모리를 소모하는 1500만개의 상태로 고생하고 있지 않을 때까지는 문제로 보이지 않을 수도 있을 겁니다!
7. C 고문 테스트 코드와 Promela 모두에서의 가장 어려운 일은 좋은 단정문을 만드는 겁니다. Promela는 또한 각 코드 줄마다 복사되는 단정문 같이 동작하는 `never` 문을 지원합니다.
8. 분할해 정복하기는 Promela에서 상태 공간을 제어하에 두는데에 굉장히 도움됩니다. 거대한 모델을 두개의 대략 동일한 절반으로 쪼개는 것은 각 절반이 전체의 대략 제곱근 크기가 되게 합니다. 예를 들어, 백만개의 상태가 결합된 모델은 천개 상태의 모델 한쌍으로 크기가 줄어들 수도 있습니다. 이 두개의 더 작은 모델들은 Promela에 의해 더 적은 메모리로 더 빨리 처리될 뿐 아니라 사람들이 이 두개의 더 작은 알고리즘들을 더 쉽게 이해할 수 있게 합니다.

12.1.2.2 Promela Coding Tricks

Promela는 프로토콜을 분석하기 위해 설계되었으므로 이를 병렬 프로그램에 사용하는 건 약간 낭용하는 것인 줄 합니다. 다음 트릭들은 여러분이 Promela를 안전히 낭용하는 걸 도울 겁니다:

1. 메모리 순서 재배치. 전역변수 `x` 와 `y` 를 지역변수 `r1` 과 `r2`로 복사하는 한쌍의 명령문이 있고, 순서가 중요하지만(예: 락으로 보호되지 않음), 메모리 배리어가 없는 경우를 생각해 봅시다. 이는 Promela에서 다음과 같이 모델링 될 수 있습니다:

```

1 if
2 :: 1 -> r1 = x;
3         r2 = y
4 :: 1 -> r2 = y;
5         r1 = x
6 fi

```

Listing 12.6: Complex Promela Assertion

```

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

```

Listing 12.7: Atomic Block for 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 }

```

이 `if` 문에서의 두개의 갈래는 비결정적으로 선택될 것인데, 둘 다 가능하기 때문입니다. 전체 상태 공간이 탐색되므로, 두개의 선택들이 모든 경우에 결국은 만들어질 겁니다.

물론, 이 트릭은 너무 자주 사용되면 여러분의 상태 공간을 폭증하게 할 겁니다. 또한, 이는 여러분이 가능한 재배치를 예상할 것을 요구합니다.

2. 상태 축소. 복잡한 단정문이 있다면 그것들을 `atomic`에서 평가하세요. 어쨌건, 그것들은 알고리즘의 한 부분이 아닙니다. 복잡한 단정문의 한 예는(뒤에서 더 자세히 이야기 하겠습니다) Listing 12.6에 보인 것과 같습니다.

이 단정문을 원자적이 아닌 방식으로 평가할 이유가 없는데, 그건 알고리즘의 실제 부분이 아니기 때문입니다. 각 명령문이 상태를 증가시키므로 이런 이를 Listing 12.7에 보인 것처럼 `atomic` 블록에 넣음으로써 쓸모없는 상태들의 수를 줄일 수 있습니다.

3. Promela는 함수를 제공하지 않습니다. 그대신 C 전처리기 매크로를 사용하셔야 합니다. 그러나, 조합을 통한 폭발을 막기 위해 조심스럽게 사용하셔야 합니다.

이제 더 많은 예를 알아볼 준비가 되었습니다.

Listing 12.8: Promela Code for Spinlock

```

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

```

12.1.3 Promela Example: Locking

락은 일반적으로 유용하므로 Listing 12.8에 보인 것처럼 여러 Promela 모델에 포함될 수 있는 `lock.h`에 의해 `spin_lock()`과 `spin_unlock()`이 제공됩니다. `spin_lock()` 매크로는 라인 2-11의 `do-od` 무한반복문을 포함하는데, 라인 3의 “1”의 보호 덕입니다. 이 반복문의 몸체는 `if-fi` 문을 포함하는 하나의 어토믹 블록입니다. 이 `if-fi` 문은 반복이 아니라 한번의 수행만 된다는 점을 제외하곤 `do-od` 문과 비슷하게 구성됩니다. 락이 라인 5에서 잡히면 라인 6는 이를 획득하고 라인 7는 감싸고 있는 `do-od` 반복문을 깹니다(그리고 이 어토믹 블록에서 나갑니다). 다른 한편 이 락이 이미 라인 8에서 잡혀 있다면 우린 아무것도 하지 않고(`skip`), `if-fi`와 이 어토믹 블록을 실패로 끝내서 그 바깥 반복문의 다음 반복을 취해서 이 락이 획득 가능할 때까지 이를 반복합니다.

`spin_unlock()` 매크로는 단순히 락을 잡혀있지 않다고 표시합니다.

Promela는 완전한 순서규칙을 가정하므로 메모리 배리어는 필요치 않음에 주의하십시오. 모든 Promela 상태에서, 모든 프로세스는 현재 상태와 그 위에서 우리에게 가해진 상태 변화의 순서에 동의합니다. 이는 일부 컴퓨터 시스템에서(1990년대의 MPIS와 PA-RISC 같은) 사용된 “순차적으로 일관적인(sequentially consistent)” 메모리 모델과 비슷합니다. 앞서 언급되었듯, 그리고 뒤의 예에서 살펴보게 되겠지만 완화된 메모리 순서 규칙은 명시적으로 코딩되어야 합니다.

이 매크로들은 Listing 12.9에 보인 Promela 코드로 테스트 됩니다. 이 코드는 값 증가를 테스트하는데 사용된 것과 유사한데, 락을 사용하는 프로세스가 라인 3의 `N_LOCKERS` 매크로로 정의되어 있습니다. 뮤텍스 자체는 라인 5에, 락 소유자를 추적하기 위한 배열은 라인 6에 정의되어 있으며, 라인 7은 하나의 프로세스만이 락을 잡음을 검증하기 위한 단정문 코드에 의해 사용됩니다.

락을 사용하는 프로세스는 라인 9-18에 있는데, 단순히 라인 13에서 락을 잡고 라인 14에서 락을 잡았음을

Listing 12.9: Promela Code to Test Spinlocks

```

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 }

```

표시하고 라인 15에서 이를 철회한 후 라인 16에서 락을 내려놓기를 영원히 반복합니다.

라인 20-44의 초기화 블록은 현재 락 사용 프로세스의 `havelock` 배열을 라인 26에서 초기화하고 현재 락 사용 프로세스를 라인 27에서 시작하며, 라인 28에서 다음 락 사용 프로세스로 진행합니다. 일단 모든 락 사용 프로세스가 시작되면, `do-od` 반복문은 라인 29로 이동하는데, 단정문을 검사합니다. 라인 30과 31은 통제 변수를 초기화하고, 라인 32-40는 원자적으로 `havelock` 배열의 원소들의 값을 합하며, 라인 41는 단정을 하고, 라인 42에서 반복문을 나갑니다.

Listing 12.8과 12.9의 두 코드 조각을 `lock.h`와 `lock.spin` 파일들에 각각 넣고 다음 명령들을 사용해 모델을 수행시킬 수 있습니다.

```

spin -a lock.spin
cc -DSAFETY -o pan pan.c
./pan

```

Listing 12.10: Output for Spinlock Test

```
(Spin Version 6.4.8 -- 2 March 2018)
+ Partial Order Reduction

Full statespace search for:
  never claim          - (none specified)
  assertion violations + 
  cycle checks          - (disabled by -DSAFETY)
  invalid end states   + 

State-vector 52 byte, depth reached 360, errors: 0
  576 states, stored
  929 states, matched
  1505 transitions (= stored+matched)
  368 atomic steps
hash conflicts:          0 (resolved)

Stats on memory usage (in Megabytes):
  0.044   equivalent memory usage for states
           (stored*(State-vector + overhead))
  0.288   actual memory usage for states
  128.000  memory used for hash table (-w24)
  0.534   memory used for DFS stack (-m10000)
  128.730  total actual memory usage

unreached in proctype locker
  lock.spin:19, state 20, "-end-"
  (1 of 20 states)
unreached in init
  (0 of 22 states)
```

그 결과물은 Listing 12.10에 보인 것과 비슷한 무엇일 겁니다. 예상대로, 이 수행은 실패를 갖지 않습니다 (“errors: 0”).

Quick Quiz 12.1: 락 사용 프로세스에 도달되지 못한 명령문이 있는 이유는 뭐죠? 어쨌건, 이건 전체 상태 공간 탐색 아니었나요?

Quick Quiz 12.2: 이 예에서 Promela 코드 스타일 문제는 무엇이 있을까요?

12.1.4 Promela Example: QRCU

이 마지막 예는 Oleg Nesterov의 QRCU [Nes06a, Nes06b]의 실제 사용된 것이지만 `synchronize_qrcu()`의 빠른경로를 더 빠르게 하기 위해 수정된 버전을 선보입니다.

그런데 먼저, QRCU란 무엇일까요?

QRCU는 SRCU [McK06]의 변종으로 더 높은 읽기 오버헤드를 (전체 변수에 대한 원자적 값 증가와 감소) 극단적으로 낮은 grace-period 응답시간을 위해 포기합니다. 읽기 쓰레드가 없다면 grace period는 1マイク로 세컨드 미만의 시간에 파악되는데, 대부분의 다른 RCU 구현의 수 밀리세컨드 grace period 응답시간에 비해 무척 빠릅니다.

1. QRCU 도메인을 정의하는 `qrcu_struct`가 있습니다. SPCU처럼 (그리고 다른 RCU 변종들과는

달리) QRCU의 행동은 전역적이지 않으며 특정 `qrcu_struct`에 명시적입니다.

2. QRCU read-side 크리티컬 섹션을 정의하는 `qrcu_read_lock()`과 `qrcu_read_unlock()`이 있습니다. 연관된 `qrcu_struct`가 이 함수들에 넘겨져야만 하며, `qrcu_read_lock()`의 리턴값은 `qrcu_read_unlock()`에 넘겨져야 합니다.

예를 들면:

```
idx = qrcu_read_lock(&my_qrcu_struct);
/* read-side critical section. */
qrcu_read_unlock(&my_qrcu_struct, idx);
```

3. 모든 앞서서부터 존재한 QRCU read-side 크리티컬 섹션이 완료될 때까지 기다리는 `synchronize_qrcu()` 함수가 있지만, SPCU의 `synchronize_srcu()`와 같이, QRCU의 `synchronize_qrcu()`는 같은 `qrcu_struct`를 사용하는 read-side 크리티컬 섹션만 기다립니다.

예를 들어, `synchronize_qrcu(&your_qrcu_struct)`는 앞의 QRCU read-side 크리티컬 섹션을 기다리지 않을 겁니다. 대조적으로 `synchronize_qrcu(&my_qrcu_struct)`는 같은 `qrcu_struct`를 공유하므로 기다려야 할 겁니다.

리눅스 커널을 위한 QRCU 패치가 만들어졌습니다만 [McK07c], 리눅스 커널에 받아들여지지는 못할 것 같습니다.

Listing 12.11: QRCU Global Variables

```
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;
```

QRCU를 위한 Promela 코드로 돌아와서, 전역 변수들은 Listing 12.11에 보인 것과 같습니다. 이 예는 락킹을 사용하고 `lock.h`를 포함합니다. 읽기 쓰레드와 쓰기 쓰레드의 수 모두 두개의 `#define` 문을 통해 변화될 수 있어서 우리에게 조합적 상태 수 폭발을 일으킬 수 있는 두개의 방법을 제공합니다. `idx` 변수는 `ctr` 변수의 두 원소 중 무엇이 읽기 쓰레드에 의해 사용될지 통제하며, `readerprogress` 변수는 언제 모든 읽기 쓰레드가 끝날지 정하는 단정문을 가능하게 합니다 (QRCU 업데이트는 모든 앞서서부터 존재한 읽기 쓰레드가 그들의 QRCU read-side 크리티컬 섹션을 완료하기 전까지 허용되지 않으므로). `readerprogress`

배열 원소들은 다음과 같은 값을 가져서 연관된 읽기 쓰레드의 상태를 표시합니다:

- 0: not yet started.
- 1: within QRCU read-side critical section.
- 2: finished with QRCU read-side critical section.

마지막으로, `mutex` 변수는 업데이트 쓰레드의 느린 경로를 순차화 시키기 위해 사용됩니다.

Listing 12.12: QRCU Reader Process

```

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    }

```

QRCU 읽기 쓰레드는 Listing 12.12에 보인 `qrcu_reader()` 프로세스로 모델링 되었습니다. 라인 5-16의 `do-od` 반복문은 라인 6의 “1” 보호를 가져서 무한 반복문이 됩니다. 라인 7은 전역 인덱스의 현재 값을 가져오며 라인 8-15는 그 값이 0이 아니었다면 원자적으로 이를 증가시키고 (`atomic_inc_not_zero()`) 반복문을 종료합니다. 라인 17은 RCU read-side 크리티컬 섹션 내로의 진입을 표시하고 라인 18은 이 크리티컬 섹션으로부터 빠져나옴을 표시하는데, 둘 다 우리가 뒤에서 마주할 `assert()` 문의 이익을 위함입니다. 라인 19는 우리가 값 증가시킨 카운터의 값을 감소시키고 RCU read-side 크리티컬 섹션을 빠져나옵니다.

Listing 12.13: QRCU Unordered Summation

```

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]

```

Listing 12.13에 보인 C 전처리기 매크로는 한쌍의 카운터의 값을 더해서 완화된 메모리 순서 규칙을 애뮬레이션 합니다. 라인 2-13는 카운터들 중 하나를 읽어오고 라인 14는 이 쌍의 다른 하나를 읽어와 그것들을 더합니다. 어토믹 블록은 하나의 `do-od` 문으로 구성됩니다. 이 `do-od` 문은 (라인 3-12) Promela 가 둘 중 하나를 비결정적으로 선택하게 하는 라인 4과 8에서의 보호를 갖는 무조건적인 두개의 분기를 갖는다는 점에서 일반적이지 않습니다 (그러나 다시 말하지만, 전체 상태 공간 탐색은 Promela 가 결국은 모든 가능한 상황의 선택을 하게 합니다). 첫번째 분기는 0번째 카운터를 읽어오고 `i`를 1로 설정하며 (라인 14가 첫번째 카운터를 읽어오도록), 두번째 분기는 그 반대를 행해서 첫번째 카운터를 읽어오고 `i`를 0으로 설정합니다 (라인 14가 두번째 카운터를 읽어오도록).

Quick Quiz 12.3: 이 `do-od` 문을 더 간단하게 짜는 방법이 있을까요?

`sum_unordered` 매크로를 봤으니 이제 Listing 12.14에 보인 업데이트 쪽 프로세스로 넘어가 봅시다. 이 업데이트 쪽 프로세스는 라인 7-57의 연관된 `do-od` 문을 무한정 반복합니다. 반복문의 각 패스는 먼저 전역 `readerprogress` 배열을 지역 `readerstart` 배열로 라인 12-21에서 스냅샷 찍어둡니다. 이 스냅샷은 라인 53에서의 단정을 위해 사용될 겁니다. 라인 23은 `sum_unordered`를 수행하며, 이어서 라인 24-27는 이 빠른 수행경로가 잠재적으로 사용 가능할 경우 `sum_unordered`를 재수행 합니다.

라인 28-40는 필요하면 느린 수행경로를 수행하는데, 라인 30과 38은 업데이트 쪽 락을 잡고 내려놓으며, 라인 31-33는 인덱스를 뒤집고, 라인 34-37는 모든 앞서서부터 존재한 읽기 쓰레드들이 완료되기를 기다립니다.

라인 44-56는 이어서 `readerprogress` 배열의 현재 값을 `readerstart` 배열에 모아진 그것들과 비교하여, 이 업데이트 전에 시작된 모든 읽기 쓰레드는 여전히 진행중이지 않게 강제합니다.

Quick Quiz 12.4: 라인 12-21와 라인 44-56 내의 오퍼레이션들은 현재의 모든 제품 마이크로프로세서에 원자적 구현이 없음에도 왜 어토믹 블록이 사용되었나요?

Quick Quiz 12.5: 라인 24-27에서의 카운터들의 재합산이 정말로 필요한가요?

이제 남은건 Listing 12.15의 초기화 블록 뿐입니다. 이 블록은 단순히 라인 5-6의 카운터 쌍을 초기화하고, 라인 7-14에서 읽기 프로세스들을 시작시키며, 라인 15-21에서 업데이트 프로세스들을 시작시킵니다.

Listing 12.14: QRCU Updater Process

```

1 proctype qrcu_updater(byte me)
2 {
3     int i;
4     byte readerstart[N_QRCU_READERS];
5     int sum;
6
7     do
8     :: 1 ->
9
10    /* Snapshot reader state. */
11
12    atomic {
13        i = 0;
14        do
15        :: i < N_QRCU_READERS ->
16            readerstart[i] = readerprogress[i];
17            i++
18        :: i >= N_QRCU_READERS ->
19            break
20        od
21    }
22
23    sum_unordered;
24    if
25    :: sum <= 1 -> sum_unordered
26    :: else -> skip
27    fi;
28    if
29    :: sum > 1 ->
30        spin_lock(mutex);
31        atomic { ctr[!idx]++ }
32        idx = !idx;
33        atomic { ctr[!idx]-- }
34        do
35        :: ctr[!idx] > 0 -> skip
36        :: ctr[!idx] == 0 -> break
37        od;
38        spin_unlock(mutex);
39    :: else -> skip
40    fi;
41
42    /* Verify reader progress. */
43
44    atomic {
45        i = 0;
46        sum = 0;
47        do
48        :: i < N_QRCU_READERS ->
49            sum = sum + (readerstart[i] == 1 &&
50                          readerprogress[i] == 1);
51            i++
52        :: i >= N_QRCU_READERS ->
53            assert(sum == 0);
54            break
55        od
56    }
57    od
58 }

```

Listing 12.15: QRCU Initialization Process

```

1 init {
2     int i;
3
4     atomic {
5         ctr[idx] = 1;
6         ctr[!idx] = 0;
7         i = 0;
8         do
9             :: i < N_QRCU_READERS ->
10                readerprogress[i] = 0;
11                run qrcu_reader(i);
12                i++
13            :: i >= N_QRCU_READERS -> break
14        od;
15        i = 0;
16        do
17            :: i < N_QRCU_UPDATER ->
18                run qrcu_updater(i);
19                i++
20            :: i >= N_QRCU_UPDATER -> break
21        od
22    }
23 }

```

Table 12.2: Memory Usage of QRCU Model

updaters	readers	# states	depth	memory (MB) ^a
1	1	376	95	128.7
1	2	6,177	218	128.9
1	3	99,728	385	132.6
2	1	29,399	859	129.8
2	2	1,071,181	2,352	169.6
2	3	33,866,736	12,857	1,540.8
3	1	2,749,453	53,809	236.6
3	2	186,202,860	328,014	10,483.7

^a Obtained with the compiler flag `-DCOLLAPSE` specified.

이 모든 일은 상태 공간을 줄이기 위해 어토믹 블록에서 행해집니다.

12.1.4.1 Running the QRCU Example

이 QRCU 예제를 수행하기 위해, 앞 섹션의 코드 조각들을 `qrcu.spin`이라는 이름의 파일에 결합시키고 `spin_lock()`과 `spin_unlock()`의 정의들을 `lock.h`라는 이름의 파일에 넣으십시오. 이어서 이 QRCU 모델을 빌드하고 수행하기 위해 다음 커맨드를 사용하세요:

```

spin -a qrcu.spin
cc -DSAFETY [-DCOLLAPSE] -o pan pan.c
./pan [-mN]

```

이 출력물은 이 모델이 Table 12.2에 보인 모든 경우를 통과함을 보입니다. 세 개의 읽기 쓰레드와 세 개의 업데이트 쓰레드를 수행하는게 나을 수도 있겠으나, 간단한 외삽은 이게 대략 0.5 테라바이트 메모리를 요할 것을 보입니다. 어떠하시겠어요?

./pan 은 메모리 부족 시에 조언을 주는데, 예를 들어 세개의 읽기 쓰레드와 세개의 업데이트 쓰레드를 시도하면:

```
hint: to reduce memory, recompile with
-DCOLLAPSE # good, fast compression, or
-DMA=96 # better/slower compression, or
-DHC # hash-compaction, approximation
-DBITSTATE # supertrace, approximation
```

크게 증가된 탐색 오버헤드의 비용으로 상태 공간을 강하게 압축하는 코드를 생성하는 -DMA=N 컴파일러 플래그를 제안받은대로 시도해봅시다:

```
spin -a qrcu.spin
cc -DSAFETY -DMA=96 -O2 -o pan pan.c
./pan -m20000000
```

여기서, 깊이 한계 20,000,000 은 간단한 외삽을 통해 추론된 예상 깊이보다 열배 이상 큽니다. 이게 메모리 사용량을 늘리지만, 너무 빽빽한 깊이 제한으로부터 기인하는 불완전한 탐색으로 인한 긴 수행 낭비를 막습니다. 이 수행은 POWER9 서버에서 3일보다 조금 더 걸렸습니다. 그 결과는 Listing 12.16 에 보여져 있습니다. 이 Spin 수행은 오직 6.5 GB 전체 메모리 사용만 가지고 성공적으로 완료되었는데, 이는 대략 0.5 테라바이트 사용량을 보이는 -DCOLLAPSE 사용보다 거의 백배 가량 낮은 수치입니다.

Quick Quiz 12.6: 상태들로 인해 점유되는 메모리에서 200대1 감소에 연관되는 0.48 % 압축률! 이 상태 공간 탐색은 정말로 완벽한가요???

참고를 위해, Table 12.3 은 -DCOLLAPSE 와 -DMA=N 컴파일러 플래그가 사용되었을 때의 Spin 결과를 요약 합니다. 메모리 사용량은 최소의 충분한 탐색 깊이를 통해 얻어졌고 -DMA=N 패러미터는 표에 보여져 있습니다. -DCOLLAPSE 수행을 위한 해쉬테이블의 크기는 작은 상태 공간을 해싱하는데 너무 많은 메모리가 사용되는 것을 막기 위해 ./pan 의 -wN 옵션을 통해 조정되었습니다. 따라서 메모리 사용량은 Table 12.2 에 보인 것보다 작은데, 해쉬테이블 크기는 -w24 의 기본값부터 시작합니다. 수행시간은 POWER9 서버에서의 것으로 -DCOLLAAPSE 보다 -DMA=N 이 대략 열배 가까이 높은 CPU 오버헤드로 고생함을 보이지만, 다른 한편 메모리 오버헤드는 열배 이상 줄입니다.

여기까진 좋습니다. 하지만 업데이트 쓰레드와 읽기 스레드를 조금 더 늘리는 것은 -DMA=N 을 사용하더라도 메모리를 소모시킬 겁니다.² 그러나 어떡하죠? 여기 몇 가지 가능한 접근법이 있습니다:

² 대안적으로, CPU 소모량은 지나쳐질 거니다.

Listing 12.16: 3 Readers 3 Updaters QRCU Spin Output with -DMA=96

```
(Spin Version 6.4.6 -- 2 December 2016)
+ Partial Order Reduction
+ Graph Encoding (-DMA=96)

Full statespace search for:
  never claim           - (none specified)
  assertion violations  +
  cycle checks          -
  invalid end states   +
  (disabled by -DSAFETY)

State-vector 96 byte, depth reached 2055621, errors: 0
MA stats: -DMA=84 is sufficient
Minimized Automaton: 56420520 nodes and 1.75128e+08 edges
9.6647071e+09 states, stored
9.7503813e+09 states, matched
1.9415088e+10 transitions (= stored+matched)
7.2047951e+09 atomic steps

Stats on memory usage (in Megabytes):
1142905.887  equivalent memory usage for states
              (stored*(State-vector + overhead))
5448.879    actual memory usage for states
              (compression: 0.48%)
1068.115    memory used for DFS stack (-m20000000)
1.619       memory lost to fragmentation
6515.375    total actual memory usage

unreached in proctype qrcu_reader
  (0 of 18 states)
unreached in proctype qrcu_updater
  qrcu.spin:102, state 82, "-end-"
  (1 of 82 states)
unreached in init
  (0 of 23 states)

pan: elapsed time 2.72e+05 seconds
pan: rate 35500.523 states/second
```

1. 더 작은 수의 읽기 쓰레드와 업데이트 쓰레드가 일반적인 경우를 검증하기 충분한지 봅니다.
2. 수작업으로 정확성을 증명합니다.
3. 더 적합한 도구를 사용합니다.
4. 분할해 정복합니다.

다음 섹션은 이 방법들 각각을 이야기 합니다.

12.1.4.2 How Many Readers and Updaters Are Really Needed?

한가지 방법은 qrcu_updater() 의 Promela 코드를 주의깊게 들여다보고 유일한 전역적 상태 변화는 락 아래에서만 일어남을 깨닫는 겁니다. 따라서, 한번에 단 하나의 업데이트 쓰레드만 읽기 쓰레드나 다른 업데이트 쓰레드에게 보이는 상태를 고칠 수 있습니다. 이는 Promela 는 전체 상태 공간 탐색을 하므로 단일 업데이트 쓰레드에 의해 순차적으로 변화들이 일어남을 의미합니다. 따라서, 최대 두개의 업데이트 쓰레드가 필요합니

Table 12.3: QRCU Spin Result Summary

updaters	readers	# states	depth reached	-DCOLLAPSE			-DMA=N		
				-wN	memory (MB)	runtime (s)	N	memory (MB)	runtime (s)
1	1	376	95	12	0.10	0.00	40	0.29	0.00
1	2	6,177	218	12	0.39	0.01	47	0.59	0.02
1	3	99,728	385	16	4.60	0.14	54	3.04	0.45
2	1	29,399	859	16	2.30	0.03	55	0.70	0.13
2	2	1,071,181	2,352	20	49.24	1.45	62	7.77	5.76
2	3	33,866,736	12,857	24	1,540.70	62.5	69	111.66	326
3	1	2,749,453	53,809	21	125.25	4.01	70	11.41	19.5
3	2	186,202,860	328,014	28	10,482.51	390	77	222.26	2,560
3	3	9,664,707,100	2,055,621				84	5,557.02	266,000

다: 하나는 상태를 변화시키고 다른 하나는 혼란스러워지기 위해.

읽기 쓰레드와의 상황은 덜 분명한데, 각 읽기 쓰레드는 하나의 read-side 크리티컬 섹션을 수행하고는 종료 되기 때문입니다. 빠른 수행 경로는 카운터에서 0 또는 1까지만 볼 수 있으므로 일반적인 읽기 쓰레드의 수는 제한되어 있다고 말할 수도 있겠습니다. 이는 조사해 볼 가치 있는 부분일텐데, 실제로 다음 섹션에서 이야기 될 완전한 정확성 증명을 이릅니다.

12.1.4.3 Alternative Approach: Proof of Correctness

비정형적인 증명은 [McK07c] 다음과 같습니다:

1. `synchronize_qrcu()` 가 너무 빨리 끝나려면 정의에 의해 `synchronize_qrcu()` 의 전체 수행 중간에 최소 하나의 읽기 쓰레드가 존재했어야만 합니다.
2. 이 읽기 쓰레드에 연관된 카운터는 이 시간 동안 최소 1이었을 겁니다.
3. `synchronize_qrcu()` 코드는 전체 시간 동안 이 카운터 중 하나는 최소 1이었을 것을 강제합니다.
4. 따라서, 어느 시점에서든, 카운터들 중 하나는 최소 2이거나 두 카운터 모두 최소 1이 됩니다.
5. 그러나, `synchronize_qrcu()` 빠른 수행 경로 코드는 한번에 카운터들 중 하나만 읽을 수 있습니다. 따라서 빠른 수행 경로 코드가 첫번째 카운터를 0인 동안 읽어오지만 카운터 뒤집기와 경주를 해서 두번째 카운터는 1로 보일 수 있습니다.
6. 그런 경주 상황 동안엔 최대 하나의 읽기 쓰레드가 존재할 수 있는데, 그렇지 않으면 합산은 2 이상이 되어 업데이트 쓰레드가 느린 수행 경로를 취하게 할 것이기 때문입니다.

7. 그러나 그 경주가 빠른 수행 경로의 카운터들에 대한 첫번째 읽기에서, 그리고 또다시 두번째 읽기에서 일어났다면, 두 카운터 뒤집기가 있었어야 합니다.

8. 이 업데이트 쓰레드는 카운터를 한번만 뒤집으므로, 그리고 업데이트 쪽 락이 한쌍의 업데이트 쓰레드가 동시에 카운터를 뒤집는 것을 방지하므로, 빠른 수행 경로 코드가 뒤집기와 두 번 경주할 수 있는 유일한 방법은 첫번째 업데이트 쓰레드가 완료되었을 경우 뿐입니다.

9. 하지만 첫번째 업데이트 쓰레드는 앞서서부터 존재해온 읽기 쓰레드들이 완료되기 전까지는 완료되지 못합니다.

10. 따라서, 빠른 수행 경로가 카운터 뒤집기와 두 번 경주하려면, 모든 앞서서부터 존재해온 읽기 쓰레드들이 완료되었어야 해서 빠른 수행 경로를 취하는게 안전합니다.

물론, 모든 병렬 알고리즘이 이렇게 간단한 증명을 갖지는 않습니다. 그런 경우에는 더 많은 적합한 도구들을 모아야 합니다.

12.1.4.4 Alternative Approach: More Capable Tools

Promela 와 Spin 이 상당히 유용하지만 더 적합한 많은 도구들이 사용 가능한데, 특히 하드웨어 검증에 있어 그렇습니다. 이는 저수준 병렬 알고리즘들에서는 종종 그렇듯 여러분의 알고리즘을 하드웨어 설계용 VHDL 언어로 변환할 수 있다면 이 도구들을 여러분의 코드에 적용할 수 있음을 의미합니다 (예를 들어, 이는 첫번째 realtime RCU 알고리즘을 위해 행해졌습니다). 그러나, 그런 도구들은 상당히 비용이 높을 수 있습니다.

상용 멀티프로세싱의 발전이 결국은 멎진 상태 공간 축소 기능을 갖춘 강력한 자유 소프트웨어 모델 검사기의 출현을 가능하게 할 수도 있겠으나, 지금 당장은 도움이 되지 않습니다.

이와는 별개로, 고정된 양의 메모리를 필요로 하는 대략적 탐색을 지원하는 Spin 기능이 있습니다만, 저는 병렬 알고리즘을 검증하는데 있어 근사치를 믿지는 못했습니다.

또 다른 방법은 분할해 정복하기일 수 있습니다.

12.1.4.5 Alternative Approach: Divide and Conquer

커다란 병렬 알고리즘을 개별적으로 증명 가능한 작은 조각들로 쪼개는 게 종종 가능합니다. 예를 들어, 100억 개의 상태를 갖는 모델은 두 개의 100,000개 상태 모델로 쪼개질 수도 있습니다. 이 방법을 취하는 것은 Promela 같은 도구가 여러분의 알고리즘을 검증하기 쉽게 할뿐만 아니라 여러분의 알고리즘을 이해하기 쉽게 해주기도 합니다.

12.1.4.6 Is QRCU Really Correct?

QRCU는 정말로 정확할까요? 우리는 그렇다고 말하는 Promela 기반의 기계적 증명과 손을 통한 증명을 보았습니다. 하지만, Alglave et al. [AKT13]의 논문은 다르게 말합니다 (page 12 아래쪽의 Section 5.1을 보세요). 뭐가 맞을까요?

둘 다 맞는 것으로 드러났습니다! QRCU가 정형 검증 벤치마크에 추가되었을 때, 그것의 메모리 배리어는 생략되었고, 따라서 버그 있는 버전의 QRCU가 되었습니다. 따라서 여기서의 진짜 뉴스는 여러 정형 검증 도구들이 이 버그있는 QRCU를 잘못되게도 올바르다고 검증했습니다. 그리고 이게 정형 검증 도구들 그 자체들도 버그가 투입된 버전의 코드를 통해 검증되어야 하는 이유입니다. 특정 도구가 투입된 버그를 찾지 못한다면, 그 도구는 분명 믿지 못할 것입니다.

Quick Quiz 12.7: 하지만 다른 정형 검증 도구들은 종종 특정 종류의 버그를 찾기 위해 설계됩니다. 예를 들어, 매우 적은 정형 검증 도구들은 명세서 내의 오류를 찾을 겁니다. 그러니 이 “분명 믿지 못할만 하다”는 좀 센 말 아닌가요?

■
따라서, 여러분이 QRCU를 사용하려 한다면, 주의하세요. 그것의 정확성 증명 그 자체는 정확할 수도 아닐 수도 있습니다. 이는 Donald Knuth가 무척 오래전 이야기 했듯 정형 검증이 테스트를 완전히 대체할 수 없는 이유 중 하나입니다.

Quick Quiz 12.8: 여기 설명된 QRCU 알고리즘을 위한 두 개의 개별적 정확성 증명이 있고 부정확성 증명

은 다른 알고리즘에 대한 것이었는데 왜 의심의 여지가 있죠?

■

12.1.5 Promela Parable: dynticks and Preemptible RCU

2005년 8월 이후 개발된 -rt 패치셋 [Mol05]의 RCU 구현 비슷한 preemption 가능한 RCU 변종이 메인라인 리눅스에 리얼타임 워크로드의 지원 하에 2008년 초에 받아들여졌습니다. Preemption 가능한 RCU는 오래된 RCU 구현은 RCU read-side 크리티컬 섹션들 사이에 preemption을 불가능하게 해 상당히 큰 리얼타임 응답 시간을 초래했으므로 리얼타임 워크로드를 위해 필요했습니다.

그러나, 기존 -rt 구현의 단점 중 하나는 각 grace period 가 CPU가 저전력 “dynticks-idle” 상태에 있어 RCU read-side 크리티컬 섹션을 수행할 수 없을 때 조차도 각 CPU에서 수행되어야 하게 한다는 것입니다. Dynticks-idle 상태의 아이디어는 idle CPU는 전력을 아끼기 위해 물리적으로 전원이 꺼져야 한다는 것입니다. 요약하자면, preemption 가능한 RCU는 최신 리눅스 커널의 중요한 에너지 절약 기능을 불능화 시킬 수 있습니다. Josh Triplett과 Paul McKenney가 CPU들이 RCU grace period 동안 저전력 상태를 유지할 수 있게 하는 (그래서 리눅스 커널의 에너지 절약 기능을 유지하는) 방법들을 이야기 했지만, Steve Rostedt이 새로운 dyntick 구현을 -rt 패치셋의 preemption 가능한 RCU에 결합시키기 전까지는 문제가 되지 않았습니다.

이 조합은 Steve의 시스템 중 하나를 부팅 중 멈추게 만들었으므로, 10월에 Paul은 preemption 가능한 RCU의 grace period 처리에 대한 dynticks 친화적 수정사항을 짰습니다. Steve는 `irq_enter()`와 `irq_exit()` 인터럽트 진입/종료 함수들에서 호출되는 `rcu_irq_enter()`와 `rcu_irq_exit()` 인터페이스를 짰습니다. 이 `rcu_irq_enter()`와 `rcu_irq_exit()` 함수들은 dynticks-idle CPU가 순간적으로 RCU read-side 크리티컬 섹션을 포함하는 인터럽트 핸들러를 위해 전원을 켜지게 되는 상황을 RCU가 안정적으로 처리할 수 있게 했습니다. 이 변경사항과 함께 Steve의 시스템은 안정적으로 부팅되었지만 Paul은 우리가 코드를 처음 시도부터 올바르게 짰을리 없다는 가정과 함께 코드를 주기적으로 검사했습니다.

Paul은 2007년 10월부터 2008년 2월까지 반복적으로 코드를 리뷰했고 거의 항상 최소 한 개의 버그를 발견했습니다. 한 경우에는 Paul은 심지어 그 버그가 실제로 없다는 것을 알아차리기 전에 수정사항을 짜고 테스트하기까지 했으며 모든 경우에 그 “버그”는 존재하지 않는 것으로 드러났습니다.

2월의 끝무렵, Paul 은 이 게임에 지쳤습니다. 그래서 그는 Promela 와 Spin 의 도움을 요청하기로 결정했습니다. 다음은 일곱개의 점차 현실적이 되어가는 Promela 모델들을 보이는데, 마지막 것은 상태 공간을 위해 약 40GB 를 소모합니다.

더 중요한건, Promela 와 Spin 이 매우 미묘한 버그를 하나 찾았다는 겁니다!

Quick Quiz 12.9: 그래요, 그거 훌륭하네요! 이제 제가 40GB 메인 메모리가 없다면 뭘 하면 될까요???

여전히 더 작은 상태 공간을 갖는 더 간단하고 더 빠른 알고리즘이 나을 겁니다. 간단한 관찰자에게도 정확성이 분명한 간단한 알고리즘은 그보다도 나을 거구요!

Sections 12.1.5.1–12.1.5.4 는 preemption 가능한 RCU 의 dynticks 인터페이스에 대한 개론을 제공하고, Section 12.1.6 의 이 인터페이스에 대한 검증 이야기가 이어집니다.

12.1.5.1 Introduction to Preemptible RCU and dynticks

Per-CPU dynticks_progress_counter 변수는 dynticks 와 preemption 가능한 RCU 사이의 중심 인터페이스입니다. 이 변수는 연관된 CPU 가 dynticks-idle 모드에 있을 때에는 짹수를, 그렇지 않을 때에는 훌수를 갖습니다. CPU 는 다음 세가지 이유로 dynticks-idle 모드를 빠져나갑니다:

1. 태스크 수행을 시작하기 위해,
2. 중첩되어있을 수 있는 인터럽트 핸들러들의 가장 마지막 것을 진입할 때, 그리고
3. NMI 핸들러에 진입할 때.

Preemption 가능한 RCU 의 grace-period 메커니즘은 언제 dynticks-idle CPU 가 안전하게 무시될 수 있을지 파악하기 위해 dynticks_progress_counter 변수의 값을 샘플링합니다.

다음 세개의 섹션은 리눅스 커널 v2.6.25-rc4 의 grace-period 메커니즘에서의 task 인터페이스, interrupt/NMI 인터페이스, 그리고 dynticks_progress_counter 변수의 사용에 대한 짧은 설명을 제공합니다.

12.1.5.2 Task Interface

특정 CPU 가 더이상 수행할 태스크가 없어 dynticks-idle 모드에 진입할 때, 그 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 를 증가시키고 그 결과가 짹수인지 검사하는데, dynticks_progress_counter 의 새 값을 보는 모든 다른 CPU 가 앞의 RCU read-side 크리티컬 섹션의 완료 또한 볼 것을 보장하기 위해 메모리 배리어를 먼저 수행합니다.

비슷하게, dynticks-idle 모드에 있는 어떤 CPU 가 새 수행 가능한 task 를 시작하려 준비할 때, 그 CPU 는 rCU_exit_noHZ() 를 호출합니다:

```
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 함수 역시 dynticks_progress_counter 를 증가시키는데, 어떤 다른 CPU 가 뒤따르는 RCU read-side 크리티컬 섹션의 결과를 본다면 그 CPU 는 또한 dynticks_progress_counter 의 증가된 값을 볼것을 보장하기 위해 메모리 배리어를 이어서 수행합니다. 마지막으로, rCU_exit_noHZ() 는 그 증가된 값이 훌수임을 검사합니다.

rCU_enter_noHZ() 와 rCU_exit_noHZ() 함수는 CPU 가 task 수행을 위해 dynticks-idle 모드에 진입하고 빠져나오는 것을 처리하지만 인터럽트는 처리하지 않습니다, 이는 다음 섹션에서 다룹니다.

12.1.5.3 Interrupt Interface

rCU_irq_enter() 와 rCU_irq_exit() 함수는 각각 인터럽트/NMI 진입과 종료를 처리합니다. 물론, 중첩된 인터럽트들은 올바르게 처리되어야만 합니다. 인터럽트들이 중첩되었을 가능성은 인터럽트 또는 NMI handler 에 진입할 때 (rCU_irq_enter() 에서) 증가되고 빠져나올 때 (rCU_irq_exit() 에서) 감소되는 두번째 per-CPU 변수, rCU_update_flag 에 의해 처리됩니다. 또한, 앞서서부터 존재한 in_interrupt() 기능은 중첩된 인터럽트/NMI 와 가장 바깥 사이를 구분하기 위해 사용됩니다.

인터럽트 진입은 rCU_irq_enter() 에 의해 다음과 같이 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 }

```

라인 3은 현재 CPU의 수를 가져오고, 라인 5와 6는 `rcu_update_flag` 중첩 카운터를 그게 0이 아니라면 증가시킵니다. 라인 7-9는 우리가 인터럽트의 가장 바깥 단계에 있는지 검사하고, 그렇다면 `dynticks_progress_counter` 가 증가되어야 하는지 봅니다. 그렇다면 라인 10은 `dynticks_progress_counter` 를 증가시키고 라인 11에서 메모리 배리어를 수행하며, 라인 12에서 `rcu_update_flag` 를 증가시킵니다. `rcu_exit_nohz()` 에서와 같이, 이 메모리 배리어는 이 인터럽트 핸들러 내에서 (이 `rcu_irq_enter()` 호출을 뒤따르는) 어떤 RCU read-side 크리티컬 섹션의 효과를 보는 모든 다른 CPU가 `dynticks_progress_counter` 의 증가된 값도 볼 수 있게 합니다.

Quick Quiz 12.10: 그냥 `rcu_update_flag` 의 값을 증가시키고 `rcu_update_flag` 의 기존 값이 0이었을 때 `dynticks_progress_counter` 의 값을 증가시키지 않는 이유는 뭐죠???



Quick Quiz 12.11: 하지만 라인 7 가 우리가 가장 바깥 인터럽트에 있음을 확인하면 우린 `dynticks_progress_counter` 를 항상 값 증가시켜야 하지 않나요?



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 }

```

라인 3은 현재 CPU의 수를 앞에서와 같이 가져옵니다. 라인 5는 `rcu_update_flag` 가 0이 아닌지 확인하고, 그렇지 않다면 즉시 리턴합니다 (이 함수의 끝으로 이동함으로써). 그렇지 않다면, 라인 6부터 12가 수행됩니다. 라인 6는 `rcu_update_flag` 의 값을 감소시키고 그 결과가 0이 아니면 리턴합니다. 라인 8는 우리가 실제로 중첩된 인터럽트의 가장 바깥을 나가고 있음을

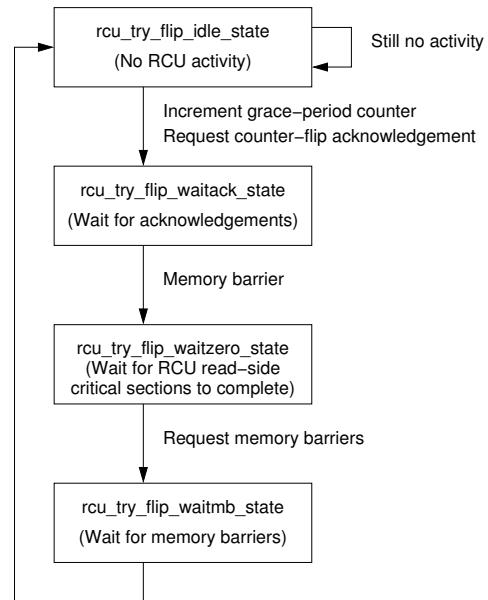


Figure 12.1: Preemptible RCU State Machine

검사하고, 라인 9는 메모리 배리어를 수행하며, 라인 10 가 `dynticks_progress_counter` 의 값을 증가시키며, 라인 11 와 12 는 이 변수가 이제 짹수값을 검사합니다. `rcu_enter_nohz()` 에서와 마찬가지로, 메모리 배리어는 `dynticks_progress_counter` 의 값 증가를 목격한 모든 다른 CPU 가 이 인터럽트 핸들러 내에서의 (`rcu_irq_exit()` 호출을 통한) RCU read-side 크리티컬 섹션의 효과 역시 볼 것을 보장합니다.

이 두개의 섹션은 `dynticks_progress_counter` 변수가 dynticks-idle 모드로의 진입과 종료 동안 테스크와 인터럽트, 그리고 NMI 에서 어떻게 유지되는지 설명했습니다. 다음 섹션은 이 변수가 preemption 가능한 RCU 의 grace-period 메커니즘에서 사용되는지 설명합니다.

12.1.5.4 Grace-Period Interface

Figure 12.1 에서 보인 preemption 가능한 RCU grace-period 상태 네개 가운데 `rcu_try_flip_waitack_state` 와 `rcu_try_flip_waitmb_state` 상태들만이 다른 CPU 들이 응답하기를 기다려야 합니다.

물론, 어떤 CPU 가 dynticks-idle 상태에 있다면 우린 그걸 기다려야 합니다. 따라서, 이 두개의 상태들 가운데 하나를 진입하기 전에 진행중인 상태는 각 CPU 의 `dynticks_progress_counter` 변수의 스냅샷을 찍고, 이를 또다른 CPU별 변수, `rcu_dyntick_snapshot` 에 저장합니다. 이는 아래에 보인 `dyntick_save_progress_counter()` 호출을 통해 이루어집니다:

```

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 }

```

rcu_try_flip_waitack_state 상태는 아래에 보인 `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 }

```

라인 7 와 8 은 현재와 스텝샷 버전의 `dynticks_progress_counter` 를 각각 집어옵니다. 라인 9 에서의 메모리 배리어는 뒤의 `rcu_try_flip_waitzero_state` 에서의 카운터 검사가 이 카운터 읽기 뒤에 이루어질 것을 보장합니다. 라인 10 와 11 은 그 스냅샷이 취해진 이후 그 CPU 가 `dynticks-idle` 상태에 머물러 있었다면 0 을 리턴합니다(어떤 CPU 와의 통신도 필요치 않음을 의미). 비슷하게 라인 12 와 13 는 그 CPU 가 처음엔 `dynticks-idle` 상태에 있었거나 `dynticks-idle` 상태를 완전히 지나왔다면 0 을 리턴합니다. 이 두 경우 모두, CPU 가 이 grace-period 카운터의 기존 값을 얻어올 방법은 없습니다. 이 조건들 중 어느것도 유지되지 않는다면, 라인 14 은 1 을 리턴해서 CPU 가 명시적으로 응답할 필요가 있음을 의미합니다.

그 부분을 위해, `rcu_try_flip_waitmb_state` 상태는 아래 보인 `rcu_try_flip_waitmb_needed()` 를 호출합니다:

```

1 static inline int
2 rCU_try_flip_waitmb_needed(int cpu)
3 {
4     long curr;
5     long snap;
6
7     curr = per_cpu(dynticks_progress_counter, cpu);
8     snap = per_cpu(rcu_dyntick_snapshot, cpu);
9     smp_mb();
10    if ((curr == snap) && ((curr & 0x1) == 0))
11        return 0;
12    if (curr != snap)
13        return 0;
14    return 1;
15 }

```

이는 `rcu_try_flip_waitack_needed()` 와 상당히 비슷한데, 차이점은 라인 12 와 13 로, `dynticks-idle` 상

태로, 또는 그로부터의 모든 전환은 `rcu_try_flip_waitmb_state` 상태에 의해 필요한 메모리 배리어를 수행하기 때문입니다.

아제 우리는 RCU 와 `dynticks-idle` 상태 사이의 인터페이스에 연관된 모든 코드를 보았습니다. 다음 섹션은 이 코드를 검증하기 위한 Promela 모델을 만들어봅니다.

Quick Quiz 12.12: 이 섹션의 코드에 있는 어떤 버그를 발견했습니까?



12.1.6 Validating Preemptible RCU and dynticks

이 섹션은 `dynticks` 와 RCU 사이의 인터페이스를 단계적으로 검증하기 위한 Promela model 을 개발하는데, Sections 12.1.6.1-12.1.6.7 각각은 프로세스 단계 코드에서 시작해 단정문들을 추가하고 인터럽트, 마지막으로 NMI 까지 한단계씩 설명합니다.

Section 12.1.6.8 는 이를 통해(다시) 배운 교훈들을 나열하고 Sections 12.1.6.9-12.1.6.15 는 RCU 의 `dynticks` 문제를 위한 간단한 해결책을 보입니다.

12.1.6.1 Basic Model

이 섹션은 프로세스 단계 `dynticks` 진입/종료와 grace-period 처리 코드를 Promela [Hol03] 으로 번역합니다. 2.6.25-rc4 커널의 `rcu_exit_nohz()` 와 `rcu_enter_nohz()` 로부터 시작해서 이를 다음과 같이 반복문 내에서 `dynticks-idle` 모드를 빠져나오고 들어가는 과정을 모델링하는 Promela 프로세스 하나에 넣을 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 }

```

라인 6 와 20 는 반복문을 정의합니다. 라인 7 는 반복문 카운터 `i` 가 `MAX_DYNTICK_LOOP_NOHZ` 한계를 넘어서면 이 반복문을 빠져나갑니다. 라인 8 은 이 반복문의

각 패스에서 라인 9-19를 수행하라고 이 반복문에게 이야기합니다. 라인 7 와 8 에서의 조건들은 서로에게 배타적이므로, 일반적인 Promela 의 사실 조건 무작위 선택은 불능화 됩니다. 라인 9 와 11 는 `rcu_exit_nohz()` 의 어토믹하지 않은 `dynticks_progress_counter` 값 증가를 모델링하며, 라인 12 는 `WARN_ON()` 을 모델링 합니다. `atomic` 은 간단히 Promela 상태 공간을 줄이는데, `WARN_ON()` 은 엄밀히 말해 이 알고리즘의 부분이 아니기 때문입니다. 라인 14-18 는 비슷하게 `rcu_enter_nohz()` 를 위한 값 증가와 `WARN_ON()` 을 모델링 합니다. 마지막으로, 라인 19 는 반복문 카운터를 증가시킵니다.

따라서 이 반복문의 각 패스는 `dynticks-idle` 모드를 빠져나오고 (예를 들면, 태스크를 수행 시작하는), 이어서 `dynticks-idle` 모드에 재진입하는 (예를 들면, 같은 태스크가 블록 당하는) CPU 를 모델링합니다.

Quick Quiz 12.13: `rcu_exit_nohz()` 와 `rcu_enter_nohz()` 에서의 메모리 배리어는 왜 Promela 에서 모델링 되지 않죠?

Quick Quiz 12.14: `rcu_exit_nohz()` 에 이어 `rcu_enter_nohz()` 가 수행되는 걸 모델링 하는건 좀 이상하지 않나요? 종료 전에 진입하는 걸 모델링하는게 더 자연스럽지 않을까요?

다음 단계는 RCU 의 grace-period 처리 인터페이스의 모델링입니다. 이를 위해, 우린 2.6.25-rc4 커널의 `dyntick_save_progress_counter()`, `rcu_try_flip_waitack_needed()`, `rcu_try_flip_waitmb_needed()` 에 더해 `rcu_try_flip_waitack()` 와 `rcu_try_flip_waitmb()` 를 모델링해야 합니다. 다음의 `grace_period()` Promela 프로세스는 preemption 가능한 RCU 의 grace-period 처리 동안 호출될 이 함수들을 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 }

```

라인 6-9 는 반복문의 한계를 출력하고 (하지만 오류의 경우 “.trail” 파일에만 합니다) `rcu_try_flip_idle()` 의 코드와 현재 CPU 의 `dynticks_progress_counter` 변수의 스냅샷을 뜨는 `dyntick_save_progress_counter()` 호출을 모델링합니다. 이 두 라인은 상태 공간을 줄이기 위해 어토믹하게 수행됩니다.

라인 10-22 은 `rcu_try_flip_waitack()` 의 연관된 코드와 그것의 `rcu_try_flip_waitack_needed()` 호출을 모델링합니다. 이 반복문은 각 CPU로부터의 카운터 뒤집기 응답을 기다리는 grace-period 상태머신을 모델링하는데, `dynticks-idle` CPU 와 상호작용하는 부분만 모델링합니다.

라인 23 은 `rcu_try_flip_waitzero()` 와 그것의 `dyntick_save_progress_counter()` 호출을 모델링하는데, 다시 한번 이 CPU 의 `dyntick_save_progress_counter` 변수의 스냅샷을 찍습니다.

마지막으로, 라인 24-36 는 `rcu_try_flip_waitack()` 의 연관된 코드와 그것의 `rcu_try_flip_waitack_needed()` 호출을 모델링 합니다. 이 반복문은 각 CPU 가 메모리 배리어 호출을 기다리는 grace-period 상태머신을 모델링하는데, 이번에도 `dynticks-idle` CPU 들과 상호작용하는 부분들만을 모델링 합니다.

Quick Quiz 12.15: 잠시만요! 리눅스 커널에서, `dynticks_progress_counter` 와 `rcu_dyntick_snapshot` 모두 CPU 별 변수들입니다. 그런데 왜 그것들이 전역 변수들로 모델링 되는거죠?

그 결과 나오는 모델은 (`dyntickRCU-base.spin`) `runspin.sh` 스크립트로 수행되었을 때, 691개의 상태를 생성하고 오류 없이 통과하는데, 실패를 찾을 수 있는 단정문이 전혀 존재하지 않는다는 사실을 생각하면 전혀 놀랍지 않습니다. 따라서 다음 섹션은 안전을 위한 단정문들을 추가합니다.

12.1.6.2 Validating Safety

안전한 RCU 구현은 grace period 가 grace period 의 시작 전에 시작된 어떤 RCU 읽기 쓰레드보다도 먼저 완료되

면 안됩니다. 이게 다음과 같이 세개의 상태를 취하는 `grace_period_state` 변수에 의해 모델링 됩니다.

```
1 #define GP_IDLE      0
2 #define GP_WAITING  1
3 #define GP_DONE      2
4 byte grace_period_state = GP_DONE;
```

`grace_period()` 프로세스는 이 변수를 아래에 보인 것처럼 grace-period 단계가 진행중인 것으로 설정합니다:

```
1 proctype grace_period()
2 {
3     byte curr;
4     byte snap;
5
6     grace_period_state = GP_IDLE;
7     atomic {
8         printf("MDLN = %d\n", MAX_DYNTICK_LOOP_NOHZ);
9         snap = dynticks_progress_counter;
10        grace_period_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    grace_period_state = GP_DONE;
26    grace_period_state = GP_IDLE;
27    atomic {
28        snap = dynticks_progress_counter;
29        grace_period_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    grace_period_state = GP_DONE;
45 }
```

라인 6, 10, 25, 26, 29, 그리고 44는 `dyntick_nohz()` 프로세스가 기본 RCU 안전성 속성을 검증할 수 있게끔 이 변수를 (가능한 곳에는 알고리즘 기반 오퍼레이션들을 어토믹하게 결합해서) 업데이트합니다. 이 검증의 형태는 `grace_period_state` 변수의 값이 RCU 읽기 쓰래드들이 그럴듯하게 존재할 수 있는 시간 동안 `GP_IDLE`에서 `GP_DONE`으로 변하지 못함을 단정하는 것입니다.

Quick Quiz 12.16: 라인 25 와 26의 한쌍의 뒤에서 뒤로의 `grace_period_state` 변경이 있는데, 우린 라인 25의 변경을 놓치지 않을 꺼라고 어떻게 확신하죠?

`dyntick_nohz()` Promela 프로세스는 이 검증을 다음과 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 = (grace_period_state == GP_IDLE);
14                assert((dynticks_progress_counter & 1) == 0);
15            }
16            atomic {
17                tmp = dynticks_progress_counter;
18                assert(!old_gp_idle ||
19                    grace_period_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 }
```

라인 13는 `grace_period_state` 변수의 값이 테스크 수행 시작 시점에 `GP_IDLE`이라면 새로운 `old_gp_idle` 플래그를 설정하고, 라인 18와 19의 단정문은 `grace_period_state` 변수가 테스크 수행 중간에 `GP_DONE`으로 변경되었다면 발화하는데, 하나의 RCU read-side 크리티컬 섹션은 전체 시간 동안 이어질 수 있음을 생각하면 불법적인 일입니다.

결과되는 모델은 (`dyntickRCU-base-s.spin`) `runspin.sh` 스크립트로 수행되었을 때, 964개의 상태를 생성하고 오류 없이 통과되는데, 안심되는 결과입니다. 그러나, 안전성이 치명적으로 중요하다고는 해도 무한정 grace period에 멈춰있는 것을 막는 것도 무척 중요합니다. 따라서 다음 섹션은 liveness 검증을 다룹니다.

12.1.6.3 Validating Liveness

Liveness는 증명하기 어렵긴 하지만, 여기 적용할 수 있는 간단한 속임수가 있습니다. 첫번째 단계는 `dyntick_nohz()`를 `dyntick_nohz_done` 변수를 통해 그것의 완료를 알리게 하는 것으로, 다음의 라인 27에 보인 것과 같습니다:

```
1 proctype dyntick_nohz()
2 {
```

```

3  byte tmp;
4  byte i = 0;
5  bit old_gp_idle;
6
7  do
8  :: i >= MAX_DYNTICK_LOOP_NOHZ -> break;
9  :: i < MAX_DYNTICK_LOOP_NOHZ ->
10 tmp = dynticks_progress_counter;
11 atomic {
12     dynticks_progress_counter = tmp + 1;
13     old_gp_idle = (grace_period_state == GP_IDLE);
14     assert((dynticks_progress_counter & 1) == 1);
15 }
16 atomic {
17     tmp = dynticks_progress_counter;
18     assert(!old_gp_idle || grace_period_state != GP_DONE);
19 }
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 변수가 있으면, 우린 다음과 같이 불필요한 블록킹을 검사하기 위해 `grace_period()`에 단정문들을 넣을 수 있습니다:

```

1 proctype grace_period()
2 {
3     byte curr;
4     byte snap;
5     bit shouldexit;
6
7     grace_period_state = GP_IDLE;
8     atomic {
9         printf("MDLN = %d\n", MAX_DYNTICK_LOOP_NOHZ);
10        shouldexit = 0;
11        snap = dynticks_progress_counter;
12        grace_period_state = GP_WAITING;
13    }
14    do
15    :: 1 ->
16        atomic {
17            assert(!shouldexit);
18            shouldexit = dyntick_nohz_done;
19            curr = dynticks_progress_counter;
20            if
21                :: (curr == snap) && ((curr & 1) == 0) ->
22                    break;
23                :: (curr - snap) > 2 || (snap & 1) == 0 ->
24                    break;
25                :: else -> skip;
26            fi;
27        }
28    od;
29    grace_period_state = GP_DONE;
30    grace_period_state = GP_IDLE;
31    atomic {
32        shouldexit = 0;
33        snap = dynticks_progress_counter;
34        grace_period_state = GP_WAITING;
35    }
36    do
37    :: 1 ->
38        atomic {
39            assert(!shouldexit);
40            shouldexit = dyntick_nohz_done;
41            curr = dynticks_progress_counter;

```

```

42        if
43            :: (curr == snap) && ((curr & 1) == 0) ->
44                break;
45            :: (curr != snap) ->
46                break;
47            :: else -> skip;
48        fi;
49    }
50    od;
51    grace_period_state = GP_DONE;
52 }

```

라인 10에서 0으로 초기화되는 `shouldexit` 변수를 라인 5에 추가했습니다. 라인 17는 `shouldexit`이 아직 설정되지 않았을 것을 단정하며, 라인 18는 `shouldexit`을 `dyntick_nohz()`에 의해 유지되는 `dyntick_nohz_done`으로 설정합니다. 따라서 이 단정문은 우리가 `dyntick_nohz()`가 수행을 완료한 후 카운터 뒤집기 확인 대기 반복문의 패스를 하나 이상 취하려 할 때 발생될 겁니다. 어쨌건, `dyntick_nohz()`가 완료되었다면, 우리를 반복문 바깥으로 나가게 강제할 더 많은 상태 변화가 있을 수 없으므로, 이 상태를 두번 진입함은 무한 반복을 의미하여, 결국 `grace period`가 끝나지 않음을 의미합니다.

라인 32, 39, 그리고 40는 두번째 (메모리 배리어) 반복문에 대해 비슷한 형태로 동작합니다.

그러나, 이 모델 (`dyntickRCU-base-sl-busted.spin`)을 수행하는 것은 실패를 초래하는데, 라인 23가 잘못된 변수가 짹수임을 검사하기 때문입니다. 실패시, `spin`은 그 실패를 오게 한 상태들의 순서를 기록하는 “trail” 파일 (`dyntickRCU-base-sl-busted.spin.trail`)을 씁니다. `spin`이 이 상태 순서를 재기록하여 수행된 상태들과 변수들의 값을 프린트하게 하기 위해 “`spin -t -p -g -l dyntickRCU-base-sl-busted.spin`” 커맨드를 사용하세요 (`dyntickRCU-base-sl-busted.spin.trail.txt`).

우린 `dyntick_nohz()` 프로세스가 step 34에서 완료되었음을 볼 수 있지만 (“34:”를 검색하세요), `grace_period()` 프로세스는 이 반복문을 빠져나가는데 실패했음을 볼 수 있습니다. `curr`의 값은 6이고 (step 35를 보세요) `snap`의 값은 5입니다 (step 17을 보세요). 따라서 라인 21에서의 첫번째 조건은 “`curr != snap`”이므로 만족되지 않으며, 라인 23에서의 두번째 조건 역시 `snap`이 훌수이고 `curr`이 `snap`보다 1만 크므로 성립되지 않습니다.

따라서 이 두개의 조건들 중 하나는 옳지 않아야 합니다. 첫번째 조건을 위한 `rcu_try_flip_waitack_needed()`의 주석에 따르면:

CPU가 전체 시간동안 dynticks 모드에 있었고 인터럽트, NMI, SMI, 또는 뭐든 취하지 않았다면 그것은 `rcu_read_lock()`의 가운데 있을 수 없고, 따라서 그것이 수행하는 다음 `rcu_read_lock()`은 이 카운터의 새 값을 가져야

합니다. 따라서 우린 이 CPU 가 이미 카운터를 확인했다고 여겨도 안전합니다.

첫번째 조건은 실제로 이에 부합하는데, “curr == snap” 이고 curr 이 짹수라면 연관된 CPU 는 전체 시간동안 dynticks-idle 모드에 있었기 때문입니다. 그러니 두번째 조건의 주석을 봅시다:

CPU 가 활성화된 irq 핸들러 없이 dynticks idle phase 를 진입했거나 통과했다면, 앞에서와 같이 우린 이 CPU 가 이미 카운터를 확인했다고 여겨도 안전합니다.

이 조건의 첫번째 부분은 옳은데, curr 와 snap 의 차이가 2라면, 그 사이에 최소 하나의 짹수가 있을 것 이어서 dynticks-idle phase 를 완전히 지나왔을 겁니다. 그러나, 이 조건의 두번째 부분은 *started* 를 dynticks-idle 모드에서 가지고 있고 이 모드에서 종료 되었음을 의미 합니다. 따라서 우린 snap 이 아니라 curr 가 짹수인지 검사합니다.

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 }
```

라인 10-13 는 이제 합쳐지고 간략화 되어서 다음과 같이 될 수 있습니다. 비슷한 간략화가 rcu_try_flip_waitmb_needed() 에도 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 }
```

이 모델에 (dyntickRCU-base-sl.spin) 연관된 수정을 가하는 것은 오류 없이 통과하는 661개의 상태를 갖는 올바른 검증을 만듭니다. 그러나, 이 liveness 검증의 첫번째 버전은 그 검증 자체 내의 버그 때문에 이

버그를 잡는데 실패했음을 이야기 해 둘 가치가 있습니다. 이 livness 검증 버그는 grace_period() 프로세스 내에 무한 반복문을 넣고 이 liveness 검증 코드가 이 문제를 발견하지 못함을 확인함으로써 발견되었습니다!

우린 이제 안전과 liveness 조건을 검증했습니다만, 수행되고 블록되는 프로세스에 한정되었습니다. 우린 인터럽트도 다뤄야 하는데, 이는 다음 섹션에서 다룹니다.

12.1.6.4 Interrupts

Promela에서 인터럽트를 모델링 하는 두가지 방법이 있습니다:

1. dynticks_nohz() 프로세스의 모든 명령문 사이에 인터럽트 핸들러를 넣는 C 전처리기 속임수를 사용하거나
2. 별도의 프로세스로 인터럽트 핸들러를 모델링하는 겁니다.

잠깐 생각해보면 두번째 방법이 더 작은 상태 공간을 가질 것 같지만 그 인터럽트 핸들러는 어떻게든 dynticks_nohz() 에 대해서는 원자적으로, 그러나 grace_period() 에는 원자적이지 않게 수행되어야 할 겁니다.

다행히도, Promela는 여러분이 어토믹 명령문에서 분기해 빠져나오는 것을 허용합니다. 이 기법은 우리가 인터럽트 핸들러가 어떤 플래그를 설정하게 하고, dynticks_nohz() 를 원자적으로 이 플래그를 검사한 후 이 플래그가 설정되어 있을 때만 수행하게 다시 짤 수 있게 합니다. 이는 다음과 같이 Promela 명령문과 라벨을 취하는 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 매크로는 다음과 같이 사용될 수 있겠습니다:

```
EXECUTE_MAINLINE(stmt1,
                  tmp = dynticks_progress_counter)
```

이 매크로의 라인 2 은 명시된 명령문 라벨을 생성합니다. 라인 3-8 는 in_dyntick_irq 변수를 검사하는 어토믹 블록이며, 이 변수가 설정되어 있다면(인터럽트 핸들러가 동작 중임을 알림), 어토믹 블록에서 그 라벨로 분기해 나갑니다. 그렇지 않다면, 라인 6 는 명시된 명령문을 수행합니다. 전체적인 효과는 요구되는 대로 메인라인 수행이 인터럽트가 동작 중인 동안은 항상 수행이 멈춰져 있다는 것입니다.

12.1.6.5 Validating Interrupt Handlers

첫번째 단계는 `dyntick_nohz()` 를 다음과 같이 `EXECUTE_MAINLINE()` 의 형태로 변환하는 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        EXECUTE_MAINLINE(stmt1,
11        tmp = dynticks_progress_counter);
12        EXECUTE_MAINLINE(stmt2,
13        dynticks_progress_counter = tmp + 1;
14        old_gp_idle = (grace_period_state == GP_IDLE);
15        assert((dynticks_progress_counter & 1) == 1));
16        EXECUTE_MAINLINE(stmt3,
17        tmp = dynticks_progress_counter;
18        assert(!old_gp_idle || grace_period_state != GP_DONE));
19        EXECUTE_MAINLINE(stmt4,
20        dynticks_progress_counter = tmp + 1;
21        assert((dynticks_progress_counter & 1) == 0));
22        i++;
23    od;
24    dyntick_nohz_done = 1;
25 }

```

`EXECUTE_MAINLINE()` 에 여러개의 명령문이 넘겨질 때에는 라인 12-15 에서와 같이 그 모든 명령문이 원자적으로 수행되어야 함을 알아두는게 중요합니다.

Quick Quiz 12.17: 하지만 하나의 `EXECUTE_MAINLINE()` 그룹 내의 명령문들이 비원자적으로 수행되어야 한다면 어떡하죠?



Quick Quiz 12.18: 하지만 `dynticks_nohz()` 프로세스가 조건적인 “if” 나 “do” 명령문을 가져서 이 구조의 몸체 명령문들이 비원자적으로 수행되어야 하는 경우는 어떡하죠?



다음 단계는 `dyntick_irq()` 프로세스를 인터럽트 핸들러를 모델링하도록 작성하는 겁니다:

```

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 = (grace_period_state == GP_IDLE);
29        assert(!old_gp_idle || grace_period_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 }

```

라인 7-49 의 반복문은 최대 `MAX_DYNTICK_LOOP_IRQ` 의 인터럽트들을 모델링 하는데 라인 8 와 9 는 반복 조건을 형성하고 라인 47 는 이 제어 변수의 값을 증가시킵니다. 라인 10 는 `dyntick_nohz()`에게 인터럽트 핸들러가 수행중임을 이야기하고, 라인 46 는 `dyntick_nohz()`에게 이 핸들러가 완료되었음을 이야기 합니다. 라인 50 은 liveness 검증을 위해 사용되는데, `dyntick_nohz()`의 연관된 부분과 동일합니다.

Quick Quiz 12.19: 라인 46 와 47 (“`in_dyntick_irq = 0;`” 와 “`i++;`”) 는 왜 어도 막하게 수행되나요?



라인 11-25 는 `rcu_irq_enter()` 를 모델링하고 라인 26 와 27 는 `_irq_enter()` 의 연관된 부분을 모델링합니다. 라인 28-30 는 `dynticks_nohz()` 의 연관된 부분과 매우 같은 방식으로 안전성을 검사합니다. 라인 31 와 32 는 `_irq_exit()` 의 연관된 부분을 모델링 하며, 마지막으로 라인 33-44 는 `rcu_irq_exit()` 을 모델링 합니다.

Quick Quiz 12.20: 인터럽트의 어떤 특성을 이 `dynticks_irq()` 프로세스는 모델링할 수 없나요?



`grace_period()` 프로세스는 다음과 같이 될 수 있습니다:

```

1 proctype grace_period()
2 {
3     byte curr;
4     byte snap;

```

```

5   bit shouldexit;
6
7   grace_period_state = GP_IDLE;
8   atomic {
9     printf("MDLN = %d\n", MAX_DYNTICK_LOOP_NOHZ);
10    printf("MDLI = %d\n", MAX_DYNTICK_LOOP_IRQ);
11    shouldexit = 0;
12    snap = dynticks_progress_counter;
13    grace_period_state = GP_WAITING;
14  }
15  do
16    :: 1 ->
17    atomic {
18      assert(!shouldexit);
19      shouldexit = dyntick_nohz_done && dyntick_irq_done;
20      curr = dynticks_progress_counter;
21      if
22        :: (curr - snap) >= 2 || (curr & 1) == 0 ->
23          break;
24        :: else -> skip;
25      fi;
26    }
27  od;
28  grace_period_state = GP_DONE;
29  grace_period_state = GP_IDLE;
30  atomic {
31    shouldexit = 0;
32    snap = dynticks_progress_counter;
33    grace_period_state = GP_WAITING;
34  }
35  do
36    :: 1 ->
37    atomic {
38      assert(!shouldexit);
39      shouldexit = dyntick_nohz_done && dyntick_irq_done;
40      curr = dynticks_progress_counter;
41      if
42        :: (curr != snap) || ((curr & 1) == 0) ->
43          break;
44        :: else -> skip;
45      fi;
46    }
47  od;
48  grace_period_state = GP_DONE;
49 }

```

grace_period() 의 구현은 앞의 것과 매우 유사합니다. 유일한 차이는 새 인터럽트 카운트 패러미터를 더하기 위해 라인 10를 더하는 것과 liveness 검사를 위한 새로운 dyntick_irq_done 변수를 라인 19와 39에 더하는 것, 그리고 물론 라인 22와 42의 최적화입니다.

이 모델은 (dyntickRCU-irqnn-ssl.spin) 약 50만 개의 상태를 갖고 오류 없이 통과하는 옳은 검증이 됩니다. 그러나, 이 버전의 모델은 중첩된 인터럽트를 다루지 않습니다. 이 주제는 다음 섹션에서 다룹니다.

12.1.6.6 Validating Nested Interrupt Handlers

중첩된 인터럽트 핸들러는 dyntick_irq() 내의 반복 문의 몸체를 다음과 같이 쪼개서 모델링 할 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 = (grace_period_state == GP_IDLE);
38        :: else -> skip;
39      fi;
40    }
41    i++;
42    :: j < i ->
43    atomic {
44      if
45        :: j + 1 == i ->
46          assert(!old_gp_idle ||
47                 grace_period_state != GP_DONE);
48        :: else -> skip;
49      fi;
50    }
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 앞의 dynticks_irq() 프로세스와 비슷합니다. 여기선 두번째 카운터 변수인 j 를 라인 5에 더해서 i 는 인터럽트 핸들러로의 진입 횟수를 세고 j 는 종료를 셉니다. 라인 7의 outermost 변수는 언제 grace_period_state 변수가 안전 검사를 위해 샘플

링 되어야 하는지 판단하는 것을 돋습니다. 라인 10과 11의 반복문 종료 검사는 특정 횟수의 인터럽트 핸들러가 진입은 물론 종료되었음을 요구하게끔 업데이트 되었으며, i의 값 증가는 이 인터럽트 진입 모델의 마지막인 라인 41로 옮겨졌습니다. 라인 13–16는 이게 중첩된 인터럽트들 가운데 가장 바깥의 것인지 알리기 위해, 그리고 dyntick_nohz() 프로세스에 의해 사용되는 in_dyntick_irq 변수의 값을 설정하기 위해 outermost 변수의 값을 설정합니다. 라인 34–40는 grace_period_state 변수의 상태를 읽지만, 가장 바깥의 인터럽트 핸들러에 있을 때만입니다.

라인 42는 인터럽트 종료 모델링을 위한 do-loop 조건을 갖습니다: 우리가 진입한 것보다 적은 수의 인터럽트를 종료했다면, 다른 인터럽트를 종료하는 것이 합법적입니다. 라인 43–50은 안전성을 검사하지만 우리가 가장 바깥의 인터럽트 단계를 나갈 때만입니다. 마지막으로, 라인 65–68는 인터럽트 종료 카운트인 j의 값을 증가시키며, 이게 가장 바깥의 인터럽트 단계라면 in_dyntick_irq의 값을 지웁니다.

이 모델은 (dyntickRCU-irq-ssl.spin) 오류 없이 통과하며 50만개를 조금 넘는 상태를 갖는 올바른 검증이 됩니다. 그러나, 이 버전의 모델은 NMI를 처리하지 않는데, 이를 다음 섹션에서 다룹니다.

12.1.6.7 Validating NMI Handlers

NMI에도 인터럽트에서와 같은 범용적 접근법을 취하는데, NMI는 중첩되지 않음을 염두에 둡니다. 이는 다음과 같은 dyntick_nmi() 프로세스가 됩니다:

```

1 proctype dyntick_nmi()
2 {
3     byte tmp;
4     byte i = 0;
5     bit old_gp_idle;
6
7     do
8     :: i >= MAX_DYNTICK_LOOP_NMI -> break;
9     :: i < MAX_DYNTICK_LOOP_NMI ->
10    in_dyntick_nmi = 1;
11    if
12    :: rcu_update_flag > 0 ->
13        tmp = rcu_update_flag;
14        rcu_update_flag = tmp + 1;
15    :: else -> skip;
16    fi;
17    if
18    :: !in_interrupt &&
19        (dynticks_progress_counter & 1) == 0 ->
20        tmp = dynticks_progress_counter;
21        dynticks_progress_counter = tmp + 1;
22        tmp = rcu_update_flag;
23        rcu_update_flag = tmp + 1;
24    :: else -> skip;
25    fi;
26    tmp = in_interrupt;
27    in_interrupt = tmp + 1;
28    old_gp_idle = (grace_period_state == GP_IDLE);
29    assert(!old_gp_idle ||
30          grace_period_state != GP_DONE);

```

```

31    tmp = in_interrupt;
32    in_interrupt = tmp - 1;
33    if
34    :: rcu_update_flag != 0 ->
35        tmp = rcu_update_flag;
36        rcu_update_flag = tmp - 1;
37    if
38    :: rcu_update_flag == 0 ->
39        tmp = dynticks_progress_counter;
40        dynticks_progress_counter = tmp + 1;
41    :: else -> skip;
42    fi;
43    :: else -> skip;
44    fi;
45    atomic {
46        i++;
47        in_dyntick_nmi = 0;
48    }
49    od;
50    dyntick_nmi_done = 1;
51 }

```

물론, 우리가 NMI를 가짐은 다른 컴포넌트들에 조정을 필요로 합니다. 예를 들어, EXECUTE_MAINLINE() 매크로는 이제 다음과 같이 dyntick_nmi_done을 검사함으로써 인터럽트 핸들러 (in_dyntick_irq) 외에도 NMI 핸들러에 주의를 기울여야 합니다 (in_dyntick_nmi).

```

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     }

```

우린 또한 dyntick_irq()가 dyntick_nmi()를 배제하게 하기 위해 in_dyntick_nmi를 검사하는 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 나아가 dyntick_irq()를 EXECUTE_IRQ()를 수행하게끔 다음과 같이 변환시켜야 합니다:

It is further necessary to convert dyntick_irq() to EXECUTE_IRQ() as follows:

```

1 proctype dyntick_irq()
2 {
3     byte tmp;
4     byte i = 0;
5     byte j = 0;
6     bit old_gp_idle;
7     bit outermost;
8
9     do

```

```

10  :: i >= MAX_DYNTICK_LOOP_IRQ &&
11  :: j >= MAX_DYNTICK_LOOP_IRQ -> break;
12  :: i < MAX_DYNTICK_LOOP_IRQ ->
13  atomic {
14      outermost = (in_dyntick_irq == 0);
15      in_dyntick_irq = 1;
16  }
17  stmt1: skip;
18  atomic {
19      if
20          :: in_dyntick_nmi -> goto stmt1;
21          :: !in_dyntick_nmi && rcu_update_flag ->
22              goto stmt1_then;
23          :: else -> goto stmt1_else;
24          fi;
25      }
26  stmt1_then: skip;
27      EXECUTE_IRQ(stmt1_1, tmp = rcu_update_flag)
28      EXECUTE_IRQ(stmt1_2, rcu_update_flag = tmp + 1)
29  stmt1_else: skip;
30  stmt2: skip; atomic {
31      if
32          :: in_dyntick_nmi -> goto stmt2;
33          :: !in_dyntick_nmi &&
34              !in_interrupt &&
35                  (dynticks_progress_counter & 1) == 0 ->
36                      goto stmt2_then;
37          :: else -> goto stmt2_else;
38          fi;
39      }
40  stmt2_then: skip;
41      EXECUTE_IRQ(stmt2_1,
42          tmp = dynticks_progress_counter)
43      EXECUTE_IRQ(stmt2_2,
44          dynticks_progress_counter = tmp + 1)
45      EXECUTE_IRQ(stmt2_3, tmp = rcu_update_flag)
46      EXECUTE_IRQ(stmt2_4, rcu_update_flag = tmp + 1)
47  stmt2_else: skip;
48      EXECUTE_IRQ(stmt3, tmp = in_interrupt)
49      EXECUTE_IRQ(stmt4, in_interrupt = tmp + 1)
50  stmt5: skip;
51  atomic {
52      if
53          :: in_dyntick_nmi -> goto stmt4;
54          :: !in_dyntick_nmi && outermost ->
55              old_gp_idle = (grace_period_state == GP_IDLE);
56          :: else -> skip;
57          fi;
58      i++;
59  }
60  :: j < i ->
61  stmt6: skip;
62  atomic {
63      if
64          :: in_dyntick_nmi -> goto stmt6;
65          :: !in_dyntick_nmi && j + 1 == i ->
66              assert(!old_gp_idle ||
67                  grace_period_state != GP_DONE);
68          :: else -> skip;
69          fi;
70  }
71  EXECUTE_IRQ(stmt7, tmp = in_interrupt);
72  EXECUTE_IRQ(stmt8, in_interrupt = tmp - 1);
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_3_else: skip;
101  atomic {
102      j++;
103      in_dyntick_irq = (i != j);
104  }
105  od;
106  dyntick_irq_done = 1;
107 }

```

우린 “if” 문을 직접 코딩했음을 (예를 들어, 라인 17-29) 기억해 두시기 바랍니다. 또한, 엄격히 지역적인 상태를 처리하는 명령들은 (예를 들면 라인 59) dyntick_nmi()를 배제할 필요 없습니다.

마지막으로 grace_period()는 약간의 변경만을 필요로 합니다:

```

1 proctype grace_period()
2 {
3     byte curr;
4     byte snap;
5     bit shouldexit;
6
7     grace_period_state = GP_IDLE;
8     atomic {
9         printf("MDL_NOHZ = %d\n", MAX_DYNTICK_LOOP_NOHZ);
10        printf("MDL_IRQ = %d\n", MAX_DYNTICK_LOOP_IRQ);
11        printf("MDL_NMI = %d\n", MAX_DYNTICK_LOOP_NMI);
12        shouldexit = 0;
13        snap = dynticks_progress_counter;
14        grace_period_state = GP_WAITING;
15    }
16    do
17        :: 1 ->
18        atomic {
19            assert(!shouldexit);
20            shouldexit = dyntick_nohz_done &&
21                dyntick_irq_done &&
22                dyntick_nmi_done;
23            curr = dynticks_progress_counter;
24            if
25                :: (curr - snap) >= 2 || (curr & 1) == 0 ->
26                    break;
27                :: else -> skip;
28                fi;
29            }
30        od;
31        grace_period_state = GP_DONE;
32        grace_period_state = GP_IDLE;
33        atomic {
34            shouldexit = 0;
35            snap = dynticks_progress_counter;
36            grace_period_state = GP_WAITING;
37        }
38    do
39        :: 1 ->

```

```

40     atomic {
41         assert(!shouldexit);
42         shouldexit = dyntick_nohz_done &&
43             dyntick_irq_done &&
44             dyntick_nmi_done;
45         curr = dynticks_progress_counter;
46         if
47             :: (curr != snap) || ((curr & 1) == 0) ->
48                 break;
49             :: else -> skip;
50         fi;
51     }
52 od;
53 grace_period_state = GP_DONE;
54 }

```

라인 11에 새로운 MAX_DYNTICK_LOOP_NMI 패러미터를 위한 `printf()`를 추가했으며 라인 22와 44에서 `shouldexit` 할당에 `dyntick_nmi_done`을 추가했습니다.

이 모델은 (`dyntickRCU-irq-nmi-ssl.spin`) 오류 없이 통과하는 수억개 상태를 가지는 올바른 검증이 됩니다.

Quick Quiz 12.21: Paul은 정말로 항상 그의 코드를 이 고통스럽게 점진적인 방식으로 작성하나요?



12.1.6.8 Lessons (Re)Learned

이 노력은 어떤 교훈들을 (다시) 배우게 해 줍니다:

1. **Promela 와 Spin은 인터럽트/NMI-핸들러 상호작용을 검증할 수 있습니다.**
2. **코드를 문서화하는 것은 버그를 찾는 것을 돋습니다.** 이 경우, 문서화 노력은 이어진 패치에서 [McK08b] 볼 수 있듯 `rcu_enter_nohz()` 와 `rcu_exit_nohz()`의 잘못 배치된 메모리 배리어를 찾게 해 줍니다.

```

static inline void rcu_enter_nohz(void)
{
+    mb();
-    __get_cpu_var(dynticks_progress_counter)++;
}
static inline void rcu_exit_nohz(void)
{
-    mb();
+    __get_cpu_var(dynticks_progress_counter)++;
}

```

3. **여러분의 코드를 일찍, 자주, 파멸적 수준까지 검증하십시오.** 이 노력은 다음 패치에서 [McK08c] 볼 수 있는 테스트나 디버깅하기 무척 어려웠을 `rcu_try_flip_waitack_needed()`에 있는 작은 버그를 찾아냈습니다.

-	if ((curr - snap) > 2 (snap & 0x1) == 0)
+	if ((curr - snap) > 2 (curr & 0x1) == 0)

4. **여러분의 검증 코드를 항상 검증하세요.** 이를 위한 일반적인 방법은 고의적인 버그를 넣고 그 검증 코드가 이를 찾는지 검사하는 겁니다. 물론, 그 검증 코드가 이 버그를 찾아내지 못한다면 여러분은 그 버그 자체를 검증해야 하며, 그 과정은 무한히 반복되어야 합니다. 그러나, 여러분 스스로가 이 위치에 있다면, 충분한 숙면을 취하는 것이 극단적으로 효과적인 디버깅 기술이 될 수 있습니다. 그러면 여러분은 검증을 검증하는 분명한 기법은 의도적으로 검증되는 코드에 넣는 것임을 알게 될 겁니다. 이 검증이 그것들을 찾지 못한다면, 그 검증은 분명 버그가 있습니다.

5. **어토믹 명령을 사용하는 것은 검증을 단순화 시킬 수 있습니다.** 불행히도, `cmpxchg` 어토믹 명령을 사용하는 것은 치명적인 IRQ 빠른 수행 경로를 느려지게 만들 수 있으므로, 그것들은 이 경우에 적합하지 않습니다.

6. **복잡한 정형 검증의 필요성은 종종 여러분의 설계를 다시 고민해볼 필요를 알립니다.**

이 마지막 지점에 이르러서 dynticks 문제를 위한 훨씬 더 간단한 해법이 있음이 드러났는데, 이를 다음 섹션에서 보입니다.

12.1.6.9 Simplicity Avoids Formal Verification

Preemption 가능한 RCU를 위한 dynticks 인터페이스의 복잡도의 주된 요인은 IRQ와 NMI 둘 다 같은 코드 경로와 같은 상태 변수들을 사용한다는 사실입니다. 이는 간접적으로 Manfred Spraul에 의해 제안된 [Spr08] 계층적 RCU [McK08a]에 의해 구현된 것과 같이 IRQ와 NMI에 별도의 코드 경로와 변수들을 제공하는 방향으로 이끌었습니다. 이 작업물은 v2.6.29 개발 주기 중간에 메인라인 커널에 들어갔습니다 [McK08e].

12.1.6.10 State Variables for Simplified Dynticks Interface

Listing 12.17는 새로운 CPU 별 상태 변수들을 보입니다. 이 변수들은 여러 개별적 RCU 구현들이 (예: `rcu` 와 `rcu_bh`) 편리하고 효율적으로 dynticks 상태를 공유할 수 있게끔 구조체로 그룹지어져 있습니다. 다음과 같이, 이것들은 개별적인 CPU 별 변수로 생각될 수 있습니다.

`dynticks_nesting`, `dynticks`, 그리고 `dynticks_snap` 변수들은 IRQ 코드 경로를 위한 것들이고

Listing 12.17: Variables for Simple Dynticks Interface

```

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 };

```

dynticks_nmi 와 dynticks_nmi_snap 변수들은 NMI 코드 경로를 위한 것들입니다만, NMI 코드 경로는 dynticks_nesting 변수를 참조할수도 (그러나 수정하지는 않습니다) 있습니다. 이 변수들은 다음과 같이 사용됩니다:

dynticks_nesting

이는 연관된 CPU 가 RCU read-side 크리티컬 섹션들을 모니터링 해야하는 이유들의 수를 셹니다. 그 CPU 가 dynticks-idle 모드에 있다면, 이는 IRQ 중첩 수준을 세며, 그렇지 않다면 IRQ 중첩 수준보다 1 큰 값을 갖습니다.

dynticks

이 카운터의 값은 연관된 CPU 가 dynticks-idle 모드에 있고 해당 CPU에서 현재 수행중인 IRQ 핸들러가 없으면 짹수이며, 그렇지 않다면 훌수입니다. 달리 말하자면, 이 카운터의 값이 훌수라면 연관된 CPU 는 RCU read-side 크리티컬 섹션 내에 있을 수 있습니다.

dynticks_nmi

이 카운터의 값은 연관된 CPU 가 NMI 핸들러에 있으면 훌수이지만 그 NMI 가 이 CPU 가 IRQ 핸들러를 수행중이지 않은 상태에서 dynticks-idle 모드에 있을 때 도착했을 때에만 그렇습니다. 그렇지 않다면, 이 카운터의 값은 짹수가 됩니다.

dynticks_snap

이는 dynticks 카운터의 스냅샷이 되지만 현재 RCU grace period 가 너무 긴 시간동안 연장되었을 때에만 그렇습니다.

dynticks_nmi_snap

이는 dynticks_nmi 의 스냅샷이 되지만, 이번에도 현재 RCU grace period 가 너무 긴 시간으로 연장되었을 때에만 그렇습니다.

dynticks 와 dynticks_nmi 가 둘다 특정 시간 기간 동안 짹수 값을 가렸다면, 연관된 CPU 는 그 시간동안 하나의 quiescent state 를 통과했습니다.

Listing 12.18: Entering and Exiting Dynticks-Idle Mode

```

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 }

```

Quick Quiz 12.22: 하지만 어떤 IRQ 핸들러가 완료되기 전에 NMI 핸들러가 수행을 시작했고 그 NMI 핸들러는 두번째 IRQ 핸들러가 시작될 때까지 수행을 계속하면 어떻게 되죠?



12.1.6.11 Entering and Leaving Dynticks-Idle Mode

Listing 12.18 는 “nohz” 모드라고도 알려져 있는 dynticks-idle 모드를 지입하고 빠져나오는 rCU_enter_nohz() 와 rCU_exit_nohz() 를 보입니다.

라인 6 는 모든 앞선 메모리 액세스가 (RCU read-side 크리티컬 섹션으로부터의 액세스를 포함할 수 있습니다) dynticks-idle 모드로의 진입을 표시하는 것들보다 앞서서 다른 CPU 들에게 보일 것을 보장합니다. 라인 7 와 12 는 IRQ를 불능화하고 다시 활성화 합니다. 라인 8 은 현재 CPU 의 rCU_dynticks 구조체로의 포인터를 얻어오고, 라인 9 는 현재 CPUdml dynticks 카운터를 증가시키는데 우리는 프로세스 컨텍스트에서 dynticks-idle 모드에 진입하고 있으므로 이 카운터는 짹수가 될 겁니다. 마지막으로 라인 10 는 dynticks_nesting 의 값을 감소시키는데, 이제 0이 될 겁니다.

rcu_exit_nohz() 함수는 상당히 비슷하지만 dynticks_nesting 을 감소시키는게 아니라 증가시키고 반대의 dynticks 극성을 검사합니다.

Listing 12.19: NMIs From Dynticks-Idle Mode

```

1 void rcu_nmi_enter(void)
2 {
3     struct rcu_dynticks *rdtp;
4
5     rdtp = &__get_cpu_var(rcu_dynticks);
6     if (rdtp->dynticks & 0x1)
7         return;
8     rdtp->dynticks_nmi++;
9     WARN_ON(!(rdtp->dynticks_nmi & 0x1));
10    smp_mb();
11 }
12
13 void rcu_nmi_exit(void)
14 {
15     struct rcu_dynticks *rdtp;
16
17     rdtp = &__get_cpu_var(rcu_dynticks);
18     if (rdtp->dynticks & 0x1)
19         return;
20     smp_mb();
21     rdtp->dynticks_nmi++;
22     WARN_ON(rcntp->dynticks_nmi & 0x1);
23 }

```

12.1.6.12 NMIs From Dynticks-Idle Mode

Listing 12.19 는 dynticks-idle 모드에서 RCU에게 NMI 진입과 종료를 알리는 `rcu_nmi_enter()` 와 `rcu_nmi_exit()` 함수들을 보입니다. 그러나, NMI가 IRQ 핸들러의 중간에 도착한다면 RCU는 이미 이 CPU에서의 RCU read-side 크리티컬 섹션들을 찾는 중이었을 것이어서 `rcu_nmi_enter()` 의 라인 6 와 7 와 `rcu_nmi_exit()` 의 라인 18 와 19는 dynticks 가 훌수였다면 조용히 리턴합니다. 그렇지 않다면, 이 두 함수는 `dynticks_nmi` 를 증가시키는데 `rcu_nmi_enter()` 는 이를 훌수로 만들고 `rcu_nmi_exit()` 는 이를 짹수로 만듭니다. 두 함수 모두 이 값 증가와 가능한 RCU read-side 크리티컬 섹션들 사이에 라인 10 와 20에서 메모리 배리어를 수행합니다.

12.1.6.13 Interrupts From Dynticks-Idle Mode

Listing 12.20 는 IRQ 컨텍스트로의 진입과 종료를 RCU에게 알리는 `rcu_irq_enter()` 와 `rcu_irq_exit()` 함수들을 보입니다. `rcu_irq_enter()` 의 라인 6 는 `dynticks_nesting` 의 값을 증가시키고, 이 변수가 이미 0이 아니었다면 라인 7에서 조용히 리턴합니다. 그렇지 않았다면, 라인 8은 `dynticks` 의 값을 증가시키는데 그러면 그 값은 훌수가 되어서 이 CPU는 이제 RCU read-side 크리티컬 섹션을 수행할 수 있는 사실과 일관성을 갖게 됩니다. 라인 10은 따라서 `dynticks` 의 값 증가가 뒤따르는 IRQ 핸들러가 수행할 수도 있는 모든 RCU read-side 크리티컬 섹션들에게 보일 것을 보장하기 위해 메모리 배리어를 수행합니다.

`rcu_irq_exit()` 의 라인 18은 `dynticks_nesting` 의 값을 감소시키고, 그 결과가 0이 아니라면 라인 19

Listing 12.20: Interrupts From Dynticks-Idle Mode

```

1 void rcu_irq_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_irq_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(rcntp->dynticks & 0x1);
23     if (_>get_cpu_var(rcu_data).nxtlist ||
24         _>get_cpu_var(rcu_bh_data).nxtlist)
25         set_need_resched();
26 }

```

에서 조용히 리턴합니다. 그렇지 않다면, 라인 20는 라인 21에서의 `dynticks` 값 증가가 앞의 IRQ 핸들러가 수행했을 수도 있는 모든 RCU read-side 크리티컬 섹션 뒤에 보일 것을 보장하기 위해 메모리 배리어를 수행합니다. 라인 22은 `dynticks` 가 이제 짹수여서 어떤 RCU read-side 크리티컬 섹션들도 dynticks-idle 모드에서는 나타날 수 없다는 사실과 일관될 것을 검사합니다. 라인 23-25는 앞의 IRQ 핸들러가 어떤 RCU 콜백을 등록해 두었는지 검사하고, 그렇다면 reschedule API를 사용해 이 CPU가 dynticks-idle 모드에서 나가도록 강제합니다.

12.1.6.14 Checking For Dynticks Quiescent States

Listing 12.21은 특정 CPU의 `dynticks` 와 `dynticks_nmi` 카운터의 스냅샷을 가져오는 `dyntick_save_progress_counter()` 를 보입니다. 라인 8 와 9은 이 두 변수의 값을 지역변수에 스냅샷 뜨고, 라인 10은 Listings 12.18, 12.19, 그리고 12.20의 함수들 내의 메모리 배리어들과 짹을 맞추기 위해 메모리 배리어를 수행합니다. 라인 11 와 12은 `rcu_implicit_dynticks_qs()` 의 다음 호출을 위한 스냅샷을 기록해 두고, 라인 13은 그 CPU가 진행중인 IRQ나 NMI 없이 dynticks-idle 모드에 있어서 (달리 말하자면 두 스냅샷이 모두 짹수여서) 연장된 quiescent state에 있는지 검사합니다. 그렇다면, 라인 14 와 15는 이 이벤트를 세고, 라인 16은 이 CPU가 quiescent state에 있었다면 true를 리턴합니다.

Listing 12.22는 어떤 CPU가 `dynticks_save_progress_counter()` 호출에 뒤이어 dyntick-idle 모

Listing 12.21: Saving Dyntick Progress Counters

```

1 static int
2 dyntick_save_progress_counter(struct rcu_data *rdp)
3 {
4     int ret;
5     int snap;
6     int snap_nmi;
7
8     snap = rdp->dynticks->dynticks;
9     snap_nmi = rdp->dynticks->dynticks_nmi;
10    smp_mb();
11    rdp->dynticks_snap = snap;
12    rdp->dynticks_nmi_snap = snap_nmi;
13    ret = ((snap & 0x1) == 0) && ((snap_nmi & 0x1) == 0);
14    if (ret)
15        rdp->dynticks_fqs++;
16    return ret;
17 }

```

Listing 12.22: Checking 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    }
18    return rcu_implicit_offline_qs(rdp);
20 }

```

드에 진입했는지 검사하기 위해 호출되는 `rcu_implicit_dynticks_qs()`를 보입니다. 라인 9 와 11 은 연관된 CPU 의 `dynticks` 와 `dynticks_nmi` 변수들의 새 스냅샷을 취하며 라인 10 와 12 은 `dynticks_save_progress_counter()` 에 의해 앞서 저장된 스냅샷들을 얻어옵니다. 라인 13 는 이어서 Listings 12.18, 12.19, 그리고 12.20 의 함수들의 메모리 배리어들과 짹을 맞추기 위해 메모리 배리어를 수행합니다. 라인 14-15 는 이어서 이 CPU 가 현재 quiescent state 에 있는지 (`curr` 와 `curr_nmi` 가 짹수) 아니면 마지막 `dynticks_save_progress_counter()` 호출 후 quiescent state 를 하나 지났는지 (`dynticks` 와 `dynticks_nmi` 가 변경되었는지) 검사합니다. 이 검사들이 이 CPU 는 dyntick-idle quiescent state 를 지났음을 확인한다면 라인 16 는 그 사실을 세고 라인 17 은 이 사실의 알림을 리턴합니다. 어느 경우든, 라인 19 는 RCU 이 오프라인된 CPU 를 기다리는 결과를 만들 수 있는 레이스 컨디션을 검사합니다.

Quick Quiz 12.23: 이는 여전히 무척 복잡합니다. 간단하게 CPU 별 비트를 갖는 `cpumask_t` 를 가지고 그 비트를 IRQ 나 NMI 핸들러에 진입할 때 지우고 종료할 때 값 설정하는건 어떨까요?

■ 리눅스 커널 RCU 의 dyntick-idle 코드는 이 이후로도 Andy Lutomirski 의 제안에 [McK15b] 의해 한번 더 재작성 되었습니다만 이젠 정리하고 다음 주제로 넘어갈 시간입니다.

12.1.6.15 Discussion

관점을 약간 비트는 것이 RCU 를 위한 dynticks 인터페이스의 상당한 단순화를 이루었습니다. 이 단순화를 이끈 핵심 변경사항은 IRQ 와 NMI 컨텍스트 사이에 공유되는 것을 최소화 하는 것이었습니다. 이 단순화된 인터페이스에서의 유일한 공유 정보는 NMI 컨텍스트에서 IRQ 변수들로의 (`dynticks` 변수) 참조들입니다. 이런 종류의 공유는 괜찮은데, NMI 함수는 이 변수를 결코 업데이트 하지 않아서 그 값은 NMI 핸들러의 생명주기동안 변하지 않기 때문입니다. 이 공유의 제한은 개별 함수들이 한번에 하나씩 이해될 수 있게 해서 NMI 가 IRQ 함수의 수행 도중 언제든 공유된 상태를 변경할 수도 있었던 Section 12.1.5 에 설명된 상황과는 행복한 대비를 이룹니다.

검증은 좋은 일이 될 수 있지만, 단순화는 더 낫습니다.

12.2 Special-Purpose State-Space Search

Jack of all trades, master of none.

Unknown

Promela 와 Spin 이 여러분이 모든 (작은) 알고리즘을 얼마든지 검증할 수 있게 해주지만, 그것들의 큰 범용성은 가끔 문제가 될 수 있습니다. 예를 들어, Promela 는 메모리 모델이나 특정 종류의 순서 재배치 의미를 이해하지 못합니다. 따라서 이 섹션은 제품 단계 시스템에서 사용되는 메모리 모델을 이해해서 완화된 순서 코드의 검증을 크게 단순화 시키는 상태 공간 탐색 도구들을 소개합니다.

예를 들어, Section 12.1.4 는 완화된 메모리 순서 규칙을 위한 처리를 위해 어떻게 Promela 를 다뤄야 하는지를 보였습니다. 이 방법이 잘 동작하긴 하나, 이는 개발자가 그 시스템의 메모리 모델을 완전히 이해할 것을 필요로 합니다. 불행히도, 일부의 (존재한다면) 개발자들만이 현대 CPU 의 복잡한 메모리 모델을 완전히 이해합니다.

Listing 12.23: PPCMEM Litmus Test

```

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           | lzw r3,0(r4) ;
12 lwarx r11,r10,r12 | ;
13 stwxz r11,r10,r12 | ;
14 bne Fail1  | ;
15 isync      | ;
16 lzw r3,0(r4) | ;
17 Fail1:    | ;
18
19 exists
20 (0:r3=0 /\ 1:r3=0)

```

따라서, 또 다른 접근법은 Cambridge 대학의 Peter Sewell 와 Susmit Sarkar, INRIA 의 Luc Maranget, Francesco Zappa Nardelli, 그리고 Pankaj Pawan, 그리고 Oxford 대학의 Jade Alglave 가 IBM 의 Derek Williams 와 협업해 만들어낸 PPCMEM 도구와 같은, 이 메모리 순서 규칙을 이미 이해하고 있는 도구를 사용하는 것입니다. 이 연구 그룹은 Power, Arm, x86 은 물론이고 C/C++11 표준의 메모리 모델을 정형화 시키고 [Smi19], Power 와 Arm 정형화에 기초에 PPCMEM 도구를 만들었습니다.

Quick Quiz 12.24: 하지만 x86은 강한 메모리 규칙을 가지고 있는데 왜 그 메모리 모델을 정형화 시키죠?

■

PPCMEM 도구는 리트머스 테스트를 입력으로 받습니다. 샘플 리트머스 테스트가 Section 12.2.1 에서 선보입니다. Section 12.2.2 는 이 리트머스 테스트를 동일한 C-언어 프로그램으로 연관지어보고, Section 12.2.3 은 이 리트머스 테스트에 PPCMEM 을 적용하는지 설명하며, Section 12.2.4 은 그 의미를 이야기 합니다.

12.2.1 Anatomy of a Litmus Test

PPCMEM 을 위한 PowerPC 리트머스 테스트가 Listing 12.23 에 보여져 있습니다. ARM 인터페이스도 같은 방식으로 동작하지만 Arm 명령어들이 Power 명령어들로 대체되었고 시작 부분의 “PPC” 도 “ARM” 으로 교체되었습니다.

이 예에서, 라인 1 은 시스템의 타입을 (“ARM” 또는 “PPC”) 알리며 이 모델의 제목을 포함합니다. 라인 2 은 이 테스트를 위한 대안적 이름을 위한 공간을 제공하는데, 여러분은 앞의 예에서처럼 빈 줄로 보통 놔둘 겁니다. 주석은 라인 2 와 3 사이에 Ocaml (또는 Pascal) 문법의 (* *) 를 사용해 삽입될 수 있습니다.

라인 3-6 는 모든 레지스터를 위한 초기 값을 제공합니다; 각각은 P:R=V 의 형태로, P 는 프로세스 지시어이고, R 은 레지스터 지시어이며, V 는 그 값입니다. 예를 들어, 프로세스 0 의 레지스터 r3 는 초기에 값 2를 가지고 있습니다. 만약 그 값이 변수라면 (이 예에서는 x, y, 또는 z) 그 레지스터는 그 변수의 주소로 초기화 되어 있습니다. 또한, 변수들의 내용들도 초기화가 가능한데, 예를 들어 x=1 은 x 의 값을 1로 초기화 시킵니다. 초기화 되지 않은 변수들은 기본적으로 값이 0이 되어서, 이 경우 x, y, 그리고 z 는 모두 초기값 0을 갖습니다.

라인 7 는 두 프로세스를 위한 식별자를 제공해서 라인 4 의 0:r3=2 가 P0:r3=2 로 대신 쓰여질 수 있게 합니다. 라인 7 는 필요하며, 이 지시어는 Pn 의 형태여야 하는데, n 은 열 수로, 가장 왼쪽의 열이 0으로 시작합니다. 이는 불필요하게 엄격해 보일 수 있겠으나, 실제 사용 시에 상당한 혼란을 방지해 줍니다.

Quick Quiz 12.25: Listing 12.23 의 라인 8 는 왜 레지스터를 초기화 시키나요? 왜 그대신 라인 4 와 5 에서 초기화 시키지 않죠?

■

라인 8-17 은 각 프로세스를 위한 코드입니다. 특정 프로세스는 P0의 라인 11 와 P1 의 라인 12-17 에서의 경우처럼 라인을 갖지 않을 수 있습니다. 라벨과 분기가 허용되는데, 라인 14 에서 분기가, 라인 17 에 라벨이 선보여 있습니다. 그러나, 너무 자유로운 분기의 사용은 상태 공간을 폭증시킬 수 있습니다. 반복문의 사용은 여러분의 상태 공간을 폭증시키기 위한 특히 좋은 방법입니다.

라인 19-20 는 단정을 보이는데, 여기서는 우리가 P0 와 P1 의 r3 레지스터가 두 쓰레드가 모두 수행을 끝낸 후 모두 0이 될 수 있는지에 우리가 관심있음을 보입니다. P0 와 P1 이 각자의 r3 레지스터에서 둘 다 0을 보게 된다면 비참한 실패를 유발할 수 있는 많은 사용 경우가 있기 때문에 중요합니다.

이는 여러분이 간단한 리트머스 테스트를 만드는데 충분한 정보가 될겁니다. 추가적인 문서들을 구할 수 있습니다만, 그런 추가적 문서의 많은 부분은 실제 하드웨어에서 테스트를 수행하기 위한 다른 연구 도구를 위한 것입니다. 아마도 더 중요한 건, 온라인 도구를 통해 (<https://www.cl.cam.ac.uk/~pes20/ppcmem/> 의 “Select ARM Test” 와 “Select POWER Test” 버튼을 통해 사용 가능합니다) 이미 존재하는 많은 수의 리트머스 테스트를 사용 가능하다는 것일 겁니다. 이런 이미 존재하는 리트머스 테스트들 중 하나는 여러분의 Power 또는 Arm 메모리 순서규칙 질문에 대한 답을 줄 가능성이 상당할 겁니다.

Listing 12.24: Meaning of PPCMEM Litmus Test

```

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 }

```

12.2.2 What Does This Litmus Test Mean?

P0의 라인 8과 9는 C 명령문 `x=1`과 동일한데 라인 4는 P0의 레지스터 `r2`가 `x`의 주소가 되게 정의하기 때문입니다. P0의 라인 12와 13은 load-linked (Arm 용어에서의 “load register exclusive” 이자 Power 용어에서의 “load reserve”) 와 store-conditional (Arm 용어에서의 “store register exclusive”)의 기억을 각각 돋는 장치들입니다. 함께 사용되었을 때, 이것들은 하나의 어토믹 명령 시퀀스를 만드는데 `x86 lock; cmpxchg` 명령으로 예시될 수 있는 compare-and-swap 시퀀스와 대략적으로 비슷합니다. 더 높은 단계의 추상화 단계로 넘어가서 라인 10-15의 시퀀스는 리눅스 커널의 `atomic_add_return(&z, 0)`과 동일합니다. 마지막으로, 라인 16은 C 명령문 `r3=y`와 대략적으로 동일합니다.

P1의 라인 8과 9는 C 명령문 `y=1`과 동일하며, 라인 10은 메모리 배리어로, 리눅스 커널 명령문 `smp_mb()`과 동일하며 라인 11은 C 명령문 `r3=x`와 동일합니다.

Quick Quiz 12.26: 하지만 Listing 12.23의 라인 17, 즉 Fail1: 라벨에 무언가 벌어지길 할까요?

■ 이를 모두 종합해서, 이 전체 리트머스 테스트의 C-언어 동일 버전이 Listing 12.24에 보여져 있습니다. 핵심은 `atomic_add_return()`이 (리눅스 커널이 요구하듯) 완전한 메모리 배리어로 동작한다면 P0와 P1()의 `r3` 변수는 수행이 완료된 후 둘 다 0일 수 없다는 것입니다.

다음 섹션은 이 리트머스 테스트를 어떻게 수행하는지 설명합니다.

12.2.3 Running a Litmus Test

앞서 언급되었듯, 리트머스 테스트는 메모리 모델에 대한 이해를 도울 수 있는 <https://www.cl.cam.ac.uk/~pes20/ppcmem/>를 통해 대화형태로 수행될 수 있습니다. 그러나, 이 방법은 사용자가 전체 상태공간 탐색

Listing 12.25: PPCMEM Detects an Error

```

./ppcmem -model lwsync_read_block \
          -model coherence_points filename.litmus
...
States 6
0:r3=0; 1:r3=0;
0:r3=0; 1:r3=1;
0:r3=1; 1:r3=0;
0:r3=1; 1:r3=1;
0:r3=2; 1:r3=0;
0:r3=2; 1:r3=1;
Ok
Condition exists (0:r3=0 /\ 1:r3=0)
Hash=e2240ce2072a2610c034cc4fc964e77
Observation SB+lwsync-RMW-lwsync+isync Sometimes 1

```

Listing 12.26: PPCMEM on Repaired Litmus Test

```

./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

```

을 일일이 진행할 것을 필요로 합니다. 여러분이 모든 가능한 이벤트 순서를 검사하는 것은 매우 어려운 계분명하므로, 이를 위한 목적의 별도의 도구가 제공됩니다 [McK11d].

Listing 12.23에 보인 리트머스 테스트는 read-modify-write 명령을 포함하므로, 우린 이 커맨드 라인에 `-model` 인자를 더해야 합니다. 이 리트머스 테스트가 `filename.litmus`에 저장되어 있다면, 이는 Listing 12.25에 보인 출력을 냅텐데, 여기서 ...는 큰 양의 진행을 알리는 출력물을 의미합니다. 상태들은 `0:r3=0; 1:r3=0;`를 포함하는데, `atomic_add_return()`의 구형 PowerPC 구현이 전체 배리어로 동작하지 않음을 다시 알립니다. 마지막 줄의 “Sometimes”는 이를 알립니다: 이 단정문이 항상은 아니지만 일부 수행에서는 발동되었습니다.

이 리눅스 커널 버그의 수정은 P0의 `isync`를 `sync`로 바꾸는 것으로, Listing 12.26에 보인 형태가 됩니다. 여기서 볼 수 있듯, `0:r3=0; 1:r3=0;`는 상태 리스트에 나타나지 않으며, 마지막 행은 “Never”라고 말합니다. 따라서, 이 모델은 공격 수행 시퀀스는 일어날 수 없다고 예측합니다.

Quick Quiz 12.27: Arm 리눅스 커널도 비슷한 버그를 가지고 있나요?

■

Quick Quiz 12.28: Listing 12.23 provide sufficient ordering 의 라인 10 에 있는 `lwsync` 는 충분한 순서 규칙을 제공하나요?

■

12.2.4 PPCMEM Discussion

이 도구들은 Arm 와 Power 에서 수행되는 저수준 병렬 기능을 작업하는 사람들에게 큰 도움이 될 것을 약속합니다. 이 도구들은 본질적인 한계도 가지고 있습니다:

1. 이 도구들은 연구용 프로토타입이며, 따라서 지원되지 않는 경우들이 있습니다.
2. 이 도구들은 IBM 이나 Arm 에 의해 각각의 CPU 아키텍쳐에 대한 공식적 성명을 갖추지 못했습니다. 예를 들어, 두 회사 모두 이 도구들의 어떤 버전에 대해서든 언제든 버그를 보고할 권리 가지고 있습니다. 따라서 이 도구들은 실제 하드웨어에서의 철저한 스트레스 테스트의 대체물이 될 수 없습니다. 더 나아가서, 이것들이 기반하고 있는 도구들과 모델 모두 여전히 개발 중이며 언제든 바뀔 수 있습니다. 다른 한편, 이 모델은 연관된 하드웨어 전문가의 자문 아래 개발되었으므로, 이게 해당 아키텍쳐에 대한 충분한 표현이라고 자신을 가질 좋은 이유가 있기도 합니다.
3. 이 도구들은 명령 집합의 부분집합만을 다룹니다. 이 부분집합은 많은 목적에 있어 충분했으나, 여러분의 목표는 다양할 겁니다. 특히, 이 도구는 워드 (word) 크기 (32 비트) 액세스만을 처리하며, 그 워드는 올바르게 정렬되어 있어야만 합니다.³ 또한, 이 도구는 더 완화된 Arm 메모리 배리어 명령 변종들 일부도, 산술도 다루지 않습니다.
4. 이 도구들은 작은 수의 쓰레드에서 돌아가는 작은 수의 반복문 없는 코드 조각들에 제한되어 있습니다. 더 큰 예제는 Promela 와 Spin 같은 비슷한 도구에서와 같이 상태 공간 폭증을 야기합니다.
5. 전체 상태 공간 탐색은 어떻게 각 공격이 되는 상태에 이르렀는지를 알리지 않습니다. 그러나, 일단 그 상태에 닿는게 가능하다는 걸 알게 된다면 대화형 도구를 사용해 그 상태를 찾는건 너무 어렵지 않은게 일반적입니다.
6. 이 도구들은 복잡한 데이터 구조에는 썩 좋지 못합니다만, “`x=y; y=z; z=43;`” 형태의 초기화 명령들을 이용해 극단적으로 간단한 링크드 리스트 순회를 만드는 건 가능합니다.

³ 하지만 최근의 연구는 여러 크기가 혼재된 액세스도 주목하고 있습니다 [FSP⁺17].

7. 이 도구들은 memory mapped I/O 나 기기 레지스터들을 다루지 않습니다. 물론, 그런 것들을 처리하는 것은 그것들이 공식화 될 것을 필요로 하는데, 아직은 공식화 되지 않은 것으로 보입니다.

8. 이 도구들은 여러분이 단정문을 짜넣는 문제들만 텁지합니다. 이 약점은 모든 정형적 방법들에 공통된 것이며, 테스트가 여전히 중요한 또 다른 이유입니다. 이 챕터의 시작에서 인용된 Donald Knuth 의 불멸의 명언을 인용하자면, “Beware of bugs in the above code; I have only proved it correct, not tried it.”

그러나, 이 도구들의 강점들 중 하나는 이 아키텍쳐들에 허용된 동작들 전체를 모델링하도록 설계되었다는 것으로, 그 동작들은 합법적이지만 현재의 하드웨어 구현이 아직 부주의한 소프트웨어 개발자들에게 영향을 주지 않은 것들을 포함합니다. 따라서, 이 도구들에 의해 신뢰되는 알고리즘은 실제 하드웨어에서 수행될 때 약간의 추가적 안전성 허용범위를 갖습니다. 더 나아가, 실제 하드웨어에서의 테스트는 버그를 발견할 수만 있습니다; 그런 테스트는 근본적으로 어떤 사용이 올바른지를 증명할 수는 없습니다. 이를 이해하려면, 연구자들이 그들의 모델을 검증하기 위해 실제 하드웨어에서 1000억개의 테스트를 주기적으로 수행했음을 생각해 보세요. 그 중 한 경우, 아키텍쳐상으로는 허용된 동작인 1760억 회의 수행에도 불구하고 발생하지 않았습니다 [AMP⁺11]. 대조적으로, 전체 상태 공간 탐색은 이 도구가 코드 조각의 올바름을 증명할 수 있게 합니다.

정형적 방법론들과 도구들은 테스트의 대체물이 될 수 없음을 한번 더 반복할 가치가 있습니다. 중요한 사실은, 예를 들어 리눅스 커널과 같은 거대하며 신뢰성 있는 동시성 소프트웨어 작품을 만드는 것은 매우 어렵다는 것입니다. 따라서 개발자들은 이 목표를 위해 모든 도구를 적용할 준비를 해야 합니다. 이 챕터에서 선보인 도구들은 테스트를 통해 발견하기 (추적하기도) 매우 어려운 버그들을 찾을 수 있게 합니다. 다른 한편, 테스트는 이 챕터에 선보인 도구들이 영원히 처리하지 못할 만큼 큰 몸체의 소프트웨어에 적용될 수 있습니다. 항상 그렇듯, 그 일에 맞는 도구를 사용하세요!

물론, 여러분의 일이 더 간단해지게끔 병렬 코드를 쉽게 분할되게끔 설계하고 더 높은 수준의 기능들을 (락, sequence counter, 어토믹 오퍼레이션, 그리고 RCU 같은) 사용해서 이 단계의 일을 회피하는게 최고입니다. 그리고 여러분이 정말로 저수준 메모리 배리어와 read-modify-write 명령들을 사용해야만 한다고 하더라도, 여러분의 더 보수적인 이런 날카로운 도구들의 사용이 여러분의 삶은 더 쉽게 만들어 줄 겁니다.

Listing 12.27: IRIW Litmus Test

```

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) | lzw r3,0(r2) | lzw r3,0(r4) ;
12           | sync      | sync      ;
13           | lzw r5,0(r4) | lzw r5,0(r2) ;
14
15 exists
16 (2:r3=1 /\ 2:r5=0 /\ 3:r3=1 /\ 3:r5=0)

```

12.3 Axiomatic Approaches

Theory helps us to bear our ignorance of facts.

George Santayana

PPCMEM 도구가 Listing 12.27에 보인 유명한 “independent reads of independent writes” (IRIW) 리트머스 테스트를 해결할 수 있지만, 그러기 위해선 최소 CPU 시간 14 시간을 필요로 하며 최소 10 gigabyte 상태 공간을 생성합니다. 그러나, 이상황은 이 문제를 풀기 위해선 방대한 레퍼런스 매뉴얼을 숙독하고 증명을 시도하고 전문가들과 논의하고 그 마지막 결과에 대해 의심해야 했던, PPCMEM 이 발전하기 전 상황에 비하면 커다란 진보입니다. 14시간이 긴 시간처럼 보일 수 있지만, 몇주 또는 심지어 몇달에 비하면 훨씬 짧은 시간입니다.

그러나, 이 필요한 시간은 두개의 별개 변수에 값을 저장하는 두개의 쓰레드와 이 두개의 변수들에서 반대 순서로 읽기를 하는 두개의 또다른 쓰레드로 구성된 이 리트머스 테스트의 간단성을 놓고 보면 약간 놀랍기도 합니다. 단정문은 이 두개의 값 읽기 쓰레드가 두개의 값 저장의 순서에 대해 다른 의견을 표출하면 터집니다. 표준 메모리 순서 규칙 리트머스 테스트들 중에서도 이건 꽤 간단한 편입니다.

이 소모되는 시간과 공간의 양에 대한 한가지 이유는 PPCMEM이 추적 기반의 전체 상태 공간 탐색을 한다는 것으로, 이는 이 도구가 아키텍처 수준에서 발생 가능한 모든 이벤트의 순서 조합을 생성하고 평가해야 함을 의미합니다. 이 수준에서는 읽기와 스기가 모두 화려하게 장식된 이벤트와 동작들의 조합으로 연관되어서, 완전히 탐색되어야만 하는 매우 거대한 상태 공간을 초래하며, 이는 결국 큰 메모리와 CPU 소모로 이어집니다.

물론, 이 추적들 중 많은 것들은 다른 것들과 비슷한데, 비슷한 추적들을 하나의 이벤트로 취급하는게 성능을 개선할 수도 있음을 시사합니다. 그런 한가지 접근법은 Alglave 등 [AMT14] 공리적 접근으로, 메모리 모델을

Listing 12.28: Expanded IRIW Litmus Test

```

1 PPC IRIW5.litmus
2 ""
3 (* Traditional IRIW, but with five stores instead of *)
4 (* just one. *)
5 {
6 0:r1=1; 0:r2=x;
7 1:r1=1;           1:r4=y;
8           2:r2=x; 2:r4=y;
9           3:r2=x; 3:r4=y;
10 }
11 P0      | P1      | P2      | P3      ;
12 stw r1,0(r2) | stw r1,0(r4) | lzw r3,0(r2) | lzw r3,0(r4) ;
13           | sync      | sync      ;
14           | lzw r5,0(r4) | lzw r5,0(r2) ;
15
16 addi r1,r1,1 | addi r1,r1,1 | sync      ;
17           | stw r1,0(r2) | stw r1,0(r4) ;
18           | addi r1,r1,1 | sync      ;
19           | stw r1,0(r2) | stw r1,0(r4) ;
20           | addi r1,r1,1 | sync      ;
21           | stw r1,0(r2) | stw r1,0(r4) ;
22
23 exists
24 (2:r3=1 /\ 2:r5=0 /\ 3:r3=1 /\ 3:r5=0)

```

표현하기 위한 공리 집합을 만들고 리트머스 테스트들을 이 공리들에 의해 증명될 수 있는 이론들로 변환합니다. 그에 의해 만들어지는, “herd” 라 불리는 도구는 Listing 12.27에 보인 IRIW 리트머스 테스트를 포함해 PPCMEM 같은 도구에 입력되는 것과 똑같은 리트머스 테스트를 받습니다.

그러나, PPCMEM은 IRIW를 푸는데 14 CPU 시간을 소요하는데 반해 herd는 17 밀리세컨드만에 이를 완료하여, 백만배 이상의 속도향상을 보입니다. 그렇다고는 하나, 이 문제는 근본적으로 지수적으로 우린 herd가 큰 문제에 있어서는 지수적으로 속도하락을 보일 것이라 예상할 수 있습니다. 그리고 이게 실제로 일어나는 일인데, 예를 들어, Listing 12.28에 보인 것과 같이 쓰기 CPU마다 네개의 쓰기를 더하면, herd는 50,000 배 이상 느려져서 15 분이 넘는 CPU 시간을 필요로 합니다. 쓰레드를 추가하는 것 역시 폭발적인 속도 하락을 초래합니다 [MS14].

그 폭발적 본성에도 불구하고, PPCMEM과 herd는 모두 x86 시스템에서의 queued-lock handoff를 포함한 핵심 병렬 알고리즘을 검사하는데 상당히 유용함이 증명되었습니다. herd의 약점은 Section 12.2.4에서 보인 PPCMEM의 그것과 비슷합니다. PPCMEM과 herd가 다른 결과를 내는 눈에 잘 안띄는 (그러나 분명 존재하는) 경우들이 있으며, 2021년 기준으로 이런 서로 다른 결과의 문제들 중 전부는 아니지만 많은 것들이 해결되었습니다.

이 리트머스 테스트들이 (Listing 12.23처럼) 어셈블리어가 아닌 (Listing 12.24처럼) C로 쓰여질 수 있다면 도움이 될 겁니다. 이는 이제 가능한데, 다음 섹션에서 이를 다룹니다.

Listing 12.29: Locking Example

```

1 C Lock1
2
3 {}
4
5 P0(int *x, spinlock_t *sp)
6 {
7     spin_lock(sp);
8     WRITE_ONCE(*x, 1);
9     WRITE_ONCE(*x, 0);
10    spin_unlock(sp);
11 }
12
13 P1(int *x, spinlock_t *sp)
14 {
15     int r1;
16
17     spin_lock(sp);
18     r1 = READ_ONCE(*x);
19     spin_unlock(sp);
20 }
21
22 exists (1:r1=1)

```

Listing 12.30: Broken Locking Example

```

1 C Lock2
2
3 {}
4
5 P0(int *x, spinlock_t *sp1)
6 {
7     spin_lock(sp1);
8     WRITE_ONCE(*x, 1);
9     WRITE_ONCE(*x, 0);
10    spin_unlock(sp1);
11 }
12
13 P1(int *x, spinlock_t *sp2) // Buggy!
14 {
15     int r1;
16
17     spin_lock(sp2);
18     r1 = READ_ONCE(*x);
19     spin_unlock(sp2);
20 }
21
22 exists (1:r1=1)

```

12.3.1 Axiomatic Approaches and Locking

공리적 접근법은 Listing 12.29 (C-Lock1.litmus)에 보인 락 기반 리트머스 테스트로 보여지듯 더 높은 수준의 언어와 더 높은 수준의 동기화 기능들에도 적용될 수 있습니다. 이 리트머스 테스트는 리눅스 커널 메모리 모델 (Linux Kernel Memory Model: LKMM) [AMM⁺18, MS18]로 모델링 될 수 있습니다. 예상되었듯, *herd*의 결과는 *Never*를 포함해서, *P1()* 이 *x* 가 값 1을 가질 수 없음을 올바르게 파악해냅니다.⁴

Quick Quiz 12.29: Listing 12.29에 보인 것과 같은 리트머스 테스트를 *herd*로 돌리기 위해선 뭘 해야 하나요?



물론, Listing 12.30 (C-Lock2.litmus)에 보인 것처럼 *P0()* 와 *P1()* 이 다른 락을 사용한다면 모두 끝입니다. 그리고 이 경우, *herd*의 출력물은 *Sometimes*를 포함해서 다른 락의 사용이 *P1()* 으로 하여금 *x* 가 값 1을 갖는 것을 볼 수 있게 함을 올바르게 알립니다.

Quick Quiz 12.30: 왜 락킹을 직접 모델링해서 일을 복잡하게 만들죠? 어토믹 오퍼레이션을 이용해 간단히 락킹을 에뮬레이션 하는게 어떤가요?



하지만 락킹은 직접 모델링 될 수 있는 유일한 동기화 기능이 아닙니다: 다음 섹션은 RCU를 봅니다.

12.3.2 Axiomatic Approaches and RCU

공리적 접근법 역시 RCU에 관련된 리트머스 테스트를 분석할 수 있습니다 [AMM⁺18]. 그 결과, Listing 12.31 (C-RCU-remove.litmus)이 정통적인 RCU로 조정되는 링크드 리스트 원소 삭제에 관련된 리트머스 테스트를 보입니다. 락킹 리트머스 테스트에서와 같이, 이 RCU 리트머스 테스트는 LKMM에 의해 모델링 가능한데, RCU의 에뮬레이션 모델링 대비 비슷한 성능이득을 갖습니다. 라인 6는 *x*를 리스트 헤드로 보이며, 초기에는 *y*를 참조하는데, 라인 5에서 값 2로 초기화됩니다.

라인 9-14의 *P0()*는 이 리스트에서 원소 *y*를 원소 *z*와 교체하고 (라인 11), grace period 하나를 기다린 후 (라인 12), 마지막으로 *free()*를 에뮬레이션 하기 위해 *y*의 값을 0으로 만들어 (라인 13) 원소 *y*를 제거합니다. 라인 16-25의 *P1()*은 RCU read-side 크리티컬 섹션 내에서 (라인 21-24) 구행되며, 리스트 헤드를 집어와 (라인 22) 다음 원소를 읽습니다 (라인 23). 이 다음 원소는 0이 아니어야 하는데, 즉, 아직 메모리 해제되지 않았어야 합니다 (라인 28). 여러 다른 변수들은 디버깅 목적이 출력들입니다 (라인 27).

이 리트머스 테스트를 돌려울 때 *herd* 툴의 출력물은 *Never*를 포함해서, *P0()* 가 예상대로 메모리 해제된 원소에는 결코 액세스 하지 않음을 알립니다. 역시 예상된 대로, 라인 12를 제거하는 것은 *P0()* 가 메모리 해제된 원소를 액세스 하게 하는데, *herd* 출력물에 *Sometimes*로 알려집니다.

Roman Penyaev [Pen18]에 의해 제안된 더 복잡한 예제의 리트머스 테스트가 Listing 12.32 (C-RomanPenyaev-list-rcu-rr.litmus)에 보여져 있습니다. 이 예제에서, 읽기 쓰레드는 (라인 12-35의

⁴ *herd* 도구의 출력물은 PPCMEM의 그것과 호환되므로, 출력물의 포맷을 보이는 예를 위해선 Listings 12.25과 12.26를 보시기 바랍니다.

Listing 12.31: Canonical RCU Removal Litmus Test

```

1 C C-RCU-remove
2
3 {
4     int z=1;
5     int y=2;
6     int *x=y;
7 }
8
9 P0(int ***x, int *y, int *z)
10 {
11     rCU_assign_pointer(*x, z);
12     synchronize_rcu();
13     WRITE_ONCE(*y, 0);
14 }
15
16 P1(int ***x, int *y, int *z)
17 {
18     int *r1;
19     int r2;
20
21     rCU_read_lock();
22     r1 = rCU_dereference(*x);
23     r2 = READ_ONCE(*r1);
24     rCU_read_unlock();
25 }
26
27 locations [1:r1; x; y; z]
28 exists (1:r2=0)

```

P0()로 모델링 됩니다) 마지막으로 액세스 된 리스트 원소로의 포인터를 변수 c에 “흘리며” round-robin 형태로 이 링크드 리스트를 액세스 합니다. 업데이트 쓰레드들은 (라인 37-49의 P1()으로 모델링 됩니다) 원소를 하나 제거하고, 현재 또는 미래 읽기 쓰레드를 방해하는 것을 방지하여 주의합니다.

Quick Quiz 12.31: 잠깐요!!! RCU read-side 크리티컬 섹션 바깥으로 포인터를 흘리는 건 치명적인 버그 아닌가요???

라인 4-8는 초기의 링크드 리스트와 tail을 정의합니다. 리눅스 커널에서 이는 쌍방향 링크드 순환 리스트이지만, herd는 현재로써는 그런 복잡한 걸 모델링 할 수 없습니다. 여기서의 전략은 그대신 그 끝이 결코 닿을 수 없을 만큼 충분히 긴 단방향 링크드 리스트를 사용하는 겁니다. 라인 9는 변수 c를 정의하는데, 뒤따르는 RCU read-side 크리티컬 섹션들 사이의 리스트 포인터를 캐쉬하는데 사용됩니다.

여기서도 라인 12-35의 P0()는 읽기 쓰레드를 모델링 합니다. 이 프로세스는 이 리스트를 round-robin 형태로 순회하는 한쌍의 읽기 쓰레드를 모델링하는데, 첫번째 읽기 쓰레드는 라인 19-26에, 두번째 읽기 쓰레드는 라인 27-34에 있습니다. 라인 20는 c에 캐쉬된 포인터를 읽어오고, 라인 21가 이 포인터가 NULL인 것을 보게 되면 라인 22는 이 리스트의 시작부터 재시작 합니다. 어느 경우든, 라인 24는 다음 리스트 원소로 넘어가고, 라인 25는 이 원소로의 포인터를 변수 c에 다시 저장합니다. 라인 19-34는 이 프로세스를 반복

Listing 12.32: Complex RCU Litmus Test

```

1 C C-RomanPenyaev-list-rcu-rr
2
3 {
4     int *z=1;
5     int *y=z;
6     int *x=y;
7     int *w=x;
8     int *v=w;
9     int *c=w;
10 }
11
12 P0(int ***c, int ***v)
13 {
14     int *r1;
15     int *r2;
16     int *r3;
17     int *r4;
18
19     rCU_read_lock();
20     r1 = READ_ONCE(*c);
21     if (r1 == 0) {
22         r1 = READ_ONCE(*v);
23     }
24     r2 = rCU_dereference((int **)r1);
25     smp_store_release(c, r2);
26     rCU_read_unlock();
27     rCU_read_lock();
28     r3 = READ_ONCE(*c);
29     if (r3 == 0) {
30         r3 = READ_ONCE(*v);
31     }
32     r4 = rCU_dereference((int **)r3);
33     smp_store_release(c, r4);
34     rCU_read_unlock();
35 }
36
37 P1(int ***c, int ***v, int ***w, int ***x, int ***y)
38 {
39     int *r1;
40
41     rCU_assign_pointer(*w, y);
42     synchronize_rcu();
43     r1 = READ_ONCE(*c);
44     if ((int **)r1 == x) {
45         WRITE_ONCE(*c, 0);
46         synchronize_rcu();
47     }
48     smp_store_release(x, 0);
49 }
50
51 locations [1:r1; c; v; w; x; y]
52 exists (0:r1=0 \vee 0:r2=0 \vee 0:r3=0 \vee 0:r4=0)

```

하지만, r1과 r2 대신 레지스터 r3과 r4를 사용합니다. Listing 12.31에서와 같이, 이 리트머스 테스트는 free()를 에뮬레이션하기 위해 0을 저장하여, 라인 52는 이 네개의 레지스터 중 어느것이든 NULL인지, 즉 0인지 검사합니다.

P0()는 RCU로 보호되는 포인터를 첫번째 RCU read-side 크리티컬 섹션에서 다음 것으로 누출하므로, P1()은 그 업데이트를 (x의 삭제) 매우 조심스럽게 진행해야 합니다. 라인 41는 w를 y에 연결함으로써 x를 제거합니다. 라인 42는 뒤따르는 읽기 쓰레드들 중 어느 것도 링크드 리스트를 통해 x로의 경로를 갖지 못할 때까지 기다립니다. 라인 43는 c를 읽어오고, 라인 44가 c가

이번에 제거된 `x`를 참조함을 확인하면 라인 45에서 `c`를 `NULL`로 만들고 라인 46에서 다시 읽기 쓰레드를 기다려서, 뒤따르는 읽기 쓰레드들 중 어느 것도 `c`에서 `x`를 읽지 못하게 합니다. 어느 경우든, 라인 48는 `x`에 0을 저장하여 `free()`를 애뮬레이션 합니다.

Quick Quiz 12.32: Listing 12.32에서, 읽기 쓰레드는 왜 `P1()`이 라인 45에서 `c`를 0으로 만든 후 `c`를 곧바로 읽어오고 나중에 이 같은 값을 `c`에 그게 0이 된 직후에 저장하여 이 값 0으로 만들기 오퍼레이션을 무효화하지 못하나요?

■ 이 리트머스 테스트를 돌렸을 때 `herd`의 결과는 `Never`를 포함해서 `P0()`가 예상대로 결코 메모리 해제된 원소를 액세스 하지 않음을 알립니다. 역시 예상대로, `synchronize_rcu()`를 제거하는 것은 `P1()`이 메모리 해제된 원소를 액세스 하게 하며, `herd` 결과에 `Sometimes`로 알려집니다.

Quick Quiz 12.33: Listing 12.32에서, 라인 48 바로 전에 `synchronize_rcu()`를 한번만 호출하지 않는 이유가 뭐죠?

■ **Quick Quiz 12.34:** 역시 Listing 12.32에서, 라인 48는 `smp_store_release()` 대신 `WRITE_ONCE()`를 사용할 수 없나요?

■ 이 섹션들은 공리적 접근법이 어떻게 락킹과 RCU 같은 동기화 기능들을 C-언어 리트머스 테스트를 사용해 성공적으로 모델링 할 수 있는지 보였습니다. 더 장기적인 바람은 공리적 접근법이 심지어 더 높은 수준의 소프트웨어 제품들을 모델링 해서 폭발적인 검증 속도 향상을 가능하게 하는 것입니다. 이는 잠재적으로 훨씬 더 큰 소프트웨어 시스템에 대한 공리적 검증을 가능하게 할 겁니다. 또 다른 대안은 이진 논리의 공리들을 다음 섹션에서 설명하는 것처럼 서비스로 만드는 겁니다.

12.4 SAT Solvers

Live by the heuristic, die by the heuristic.

Unknown

경계가 정해진 반복문과 재귀를 갖는 유한한 프로그램은 논리 표현으로 변환될 수 있는데, 이는 이 프로그램의 단정들을 그 입력으로 표현할 수도 있습니다. 그런 논리적 표현을 가지면 어떤 가능한 입력의 조합이 그런 단정 중 하나가 터지게 할 수 있는지 알아보는게 꽤 흥미로울 겁니다. 그 입력이 이진 변수들의 조합으로 표현된다면 이는 satisfiability 문제라고도 알려져 있는, 단순한

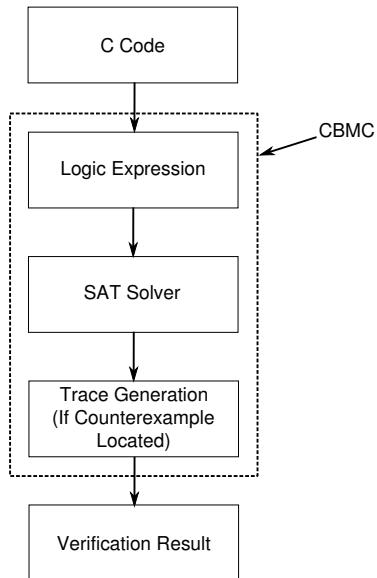


Figure 12.2: CBMC Processing Flow

SAT입니다. SAT 풀이기는 하드웨어 검증에 널리 사용되며, 상당한 진보의 모티베이션이 되었습니다. 1990년대 초의 세계급 SAT 풀이기는 100개의 이진 변수들의 논리 표현을 처리할 수 있을 수도 있지만, 2010년대 초에는 100만개 변수의 SAT 풀이기도 사용 가능해졌습니다 [KS08].

또한, SAT 풀이기를 위한 프론트엔드 프로그램은 C 코드를 자동으로 논리 표현으로 변환할 수 있어서, 단정들을 받아서 array-bounds 오류 같은 오류 조건들을 위한 단정들을 만들 수 있습니다. 한 예는 C bounded model check, 또는 cbmc 라 불리는 것으로, 많은 리눅스 배포판에서 사용 가능합니다. 이 도구는 사용하기가 무척 쉬운데, cbmc test.c 가 test.c를 검사하는데 충분하며, Figure 12.2에 보인 처리 흐름을 만듭니다. 이 쉬운 사용성은 정형적 검증이 재귀 테스트 프레임워에 포함되는 것을 가능하게 하므로 무척이나 중요합니다. 대조적으로, 특수 목적 언어로의 간단한 1차원 변환을 필요로 하는 전통적 도구들은 설계 시점 검증에만 제한됩니다.

더 최근 들어, SAT 풀이기는 병렬 코드를 처리하기 시작했습니다. 이 풀이기들은 입력 코드를 single static assignment (SSA) 형태로 변환하고, 모든 허용된 액세스 순서들을 생성합니다. 이 접근법은 잘 동작할 것 같습니다만, 실전에서 얼마나 잘 사용될지는 두고봐야 하겠습니다. 한 가지 좋은 신호는 cbmc를 리눅스 커널 RCU에 적용한 2016년의 작업물입니다 [LMKM16, LMKM18, Roy17]. 이 작업에서는 RCU의 최소 설정을 사용하고

작은 수의 쓰레드를 사용하는 시나리오를 검증했지만, 성공적으로 리눅스 커널 C 코드를 사용했고 유용한 결과를 만들어냈습니다. 이 C 코드로부터 만들어진 논리 표현은 9000만개의 변수와 4500만개의 절을 가져서 수십 기가바이트의 메모리를 사용했으며 이 SAT 풀이기가 올바른 결과를 만들기까지 CPU 시간으로 80 시간을 요했습니다.

그러나, 리눅스 커널 해커는 그들의 코드가 자동으로 검증되었다는 주장에 대해 회의적인 느낌을 받을 수도 있으며, 그런 해커는 수십년 전부터 있어온 많은 다른 회의적 시각을 찾을 수 있을 겁니다 [DMLP79]. 그런 회의적 시각을 건설적으로 표현하는 한가지 방법은 검증되었다고 주장되는 코드의 버그가 내포된 버전을 제공하는 겁니다. 이 정형적 검증 도구가 그렇게 추가된 버그를 모두 찾는다면, 우리의 해커는 이 도구의 기능에 좀 더 자신을 얻을 수도 있습니다. 물론, 그 해커가 아직 인지하지 못하고 있는 버그를 찾는 도구는 그보다도 더한 만족을 발생시킬 겁니다. 그리고 이게 왜 git이 20개의 다른 브랜치를 가지며 각 브랜치는 리눅스 커널 RCU에 추가된 버그를 잠재적으로 갖는 이유입니다 [McK17]. 누구든 정형 검증 도구를 가지고 이 검증 도전 문제들에 도전해 보시기를 진심으로 환영하는 바랍니다.

현재, cbmc는 여러 추가된 버그를 찾을 수 있지만 RCU 메인테이너가 알고 있지 못하는 버그를 아직은 찾지 못했습니다. 그러나, SAT 풀이기가 언젠가 병렬 코드의 동시성 버그를 찾는데 유용해질 날을 희망할 이유가 여럿 있습니다.

12.5 Stateless Model Checkers

He's making a list, he's permuting it twice...

with apologies to Haven Gillespie and J. Fred Coots

앞 섹션에서 설명한 SAT 풀이기 접근법은 상당히 편리하고 강력하지만, 상태를 포함한 모든 가능한 수행을 완전히 따라가는 것은 상당한 오버헤드를 일으킵니다. 실제로, 이 메모리와 CPU 시간 오버헤드는 적당하게 검증될 수 있는 프로그램의 크기를 급격하게 제한할 수 있는데, 이는 더 큰 프로그램에서는 덜 정확한 접근법이 버그를 찾을 수도 있을까 하는 질문을 일으킵니다.

여전히 배심원들은 이 질문에 대해 고민 중이지만, Nidhugg [LSLK14] 와 같은 stateless 모델 검사기들은 Figure 12.3에 보인 것처럼 몇몇 경우에 비슷한 수준의 쉬운 사용성을 가지고 더 큰 프로그램들을 처리했습니다 [KS17b]. 또한, Nidhugg는 리눅스 커널 RCU 검증 시나리오에서 cbmc 보다 수십배 빨랐습니다. 물론, Nidhugg의 속도와 확장성 이득은 그게 데이터 비결정

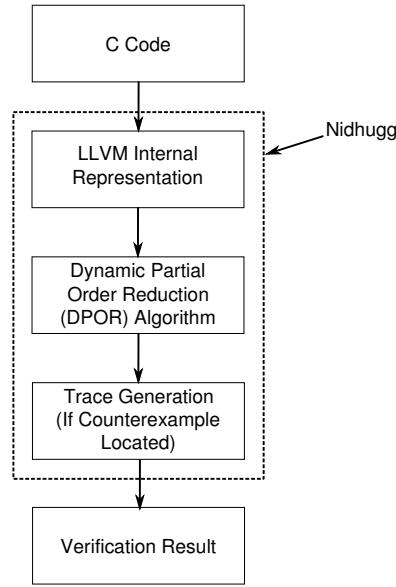


Figure 12.3: Nidhugg Processing Flow

성을 처리하지 않는다는 사실에서 나옵니다만, 이 특정 검증 시나리오에서는 중요한 게 아니었습니다.

그렇다고 하나, cbmc에서와 같이 Nidhugg는 리눅스 커널 RCU 메인테이너가 아직 몰랐던 버그를 찾지는 못했습니다. 그러나, 리눅스 커널 RCU의 한 역사적 버그는 그 메인테이너의 생각과 다른 커밋에 의해 수정되었음을 보이는 게 가능했는데, 이는 Nidhugg와 같은 stateless 모델 검사기가 언젠가는 병렬 코드의 동시성 버그를 찾는데 유용해질 것이라는 추가적 희망을 갖게 해줍니다.

12.6 Summary

Western thought has focused on True-False; it is high time to shift to Robust-Fragile.

Nassim Nicholas Taleb, summarized

이 챕터에서 이야기한 정형 검증 기법들은 작은 병렬 알고리즘을 검증하는데 매우 강력한 도구입니다만 그게 여러분의 도구상자의 유일한 도구여선 안됩니다. 정형 검증에 전념한 수십년의 세월에도 불구하고 테스트는 거대 병렬 소프트웨어 시스템의 검증 주력 도구로 남아 있습니다 [Cor06a, Jon11, McK15c].

그렇지만 이게 영원할 거라는 보장은 없습니다. 이를 위해, 2017년 기준으로 200억개가 넘는 리눅스 커널 인스턴스가 있을 것으로 추정됨을 생각해 보십시오. 리

뉴스 커널이 평균 백만년에 한번 일어나는 버그를 가지고 있다고 해봅시다. 앞 챕터의 끝에서 이야기 되었듯, 이 버그는 이 전체 설치 기반에서 매일 50번씩 나타납니다. 하지만 대부분의 정형 검증 기법은 매우 작은 코드기반에서만 사용될 수 있다는 사실도 여전합니다. 그래서 동시성 코드를 짜는 사람들은 뭘 해야 할까요?

첫번째 버그, 첫번째 연관된 버그, 마지막 연관된 버그, 그리고 마지막 버그를 찾는다는 관점에서 생각해보세요.

첫번째 버그는 보통 조사나 컴파일러 진단에 의해 찾아집니다. 갈수록 정교해지는 컴파일러 진단 기능은 경량화된 일종의 정형 검증을 제공하기 하지만, 그것들을 그런 관점에서 생각하는 건 흔치 않습니다. 이는 한편으로는 부분적으로 “내가 그걸 사용한다면, 그건 정형 검증일 수 없다”고 말하는 사용자의 이상한 편견 때문이고, 다른 한편으로는 컴파일러 진단과 검증 연구 사이의 큰 차이 때문입니다.

첫번째 연관된 버그는 조사나 컴파일러 진단을 통해 찾아질 수도 있지만, 이 두개의 단계가 오탈자나 거짓 양성만을 찾는건 일반적이지 않습니다. 어느 쪽이던, 많은 연관된 버그들, 즉, 실제로 제품 환경에서 발생할 수 있는 버그들은 테스트를 통해 발견됩니다.

테스트가 예상된 또는 실제 사용 시나리오에 의해 만들어지는 경우, 마지막 연관 버그가 테스트를 통해 찾아지는 건 드물지 않습니다. 이 상황은 정형 검증에 대한 완전한 각하의 동기를 유발할 수도 있겠으나, 연관 없는 버그들은 black-hat 공격 덕분에 가장 피곤한 순간에 연관되게 나타나는 짜증나는 습관을 갖고 있습니다. 따라서 계속해서 그 비중이 늘어나고 있는 보안에 치명적인 소프트웨어에서는 마지막 버그를 찾고 고치기 위한 강한 동기가 있습니다. 테스트는 마지막 버그를 찾을 수 없으며, 따라서 정형 검증의 역할이 존재할 것이며, 정형 검증은 그 역할에 있어서 성장을 지속할 것을 가정할 수 있습니다. 이 챕터가 보였듯, 현재의 정형 검증 시스템은 상당한 한계를 가지고 있습니다.

Quick Quiz 12.35: 하지만 충분히 낮은 단계의 소프트웨어는 모든 의도와 목적에 있어 black hat에 착취당하는데 면역이 있어야 하지 않나요?

정형적 검증은 종종 테스트보다 이용되기가 훨씬 어려움을 유의하시기 바랍니다. 이는 부분적으로 통념적 발언이며, 정형 검증이 더 익숙해져서 사용하기 쉬워지는 날이 오길 바랄 이유가 됩니다. 그러나, 매우 간단한 테스트 도구는 임의의 거대 소프트웨어 시스템에 있는 심각한 버그를 찾을 수 있습니다. 대조적으로, 정형 검증을 적용하는데 필요한 노력은 시스템 크기가 커질수록 극적으로 증가하는 것으로 보입니다.

저는 더도 덜도 아니고 지난 30년간 때때로 정형 검증을 사용하며 그 강력함을 느꼈는데, 전체 소프트웨어를

구성하는 작은 부분에 대한 설계 시점 검증이 그런 때였습니다. 더 큰 전체 소프트웨어는 물론 테스트를 통해 검증되었습니다.

Quick Quiz 12.36: L4 마이크로 커널의 전체 검증을 놓고 보면, 정형 검증에 대한 이 제한적 시선은 약간 시대에 뒤쳐진 것 아닌가요?



마지막 한가지 방법은 Section 11.1.2에서 소개된 다음의 두 정의와 그에 의미하는 결론을 고려하는 겁니다:

Definition: 버그가 없는 프로그램은 사소한 프로그램이다.

Definition: 신뢰성 있는 프로그램은 알려진 버그가 없다.

Consequences: 모든 사소하지 않으며 신뢰성 있는 프로그램은 최소 하나의 아직 알려지지 않은 버그를 가지고 있다.

이런 관점에서, 검증 분야에서의 발전은 두가지 효과를 가질 수 있습니다: (1) 사소한 프로그램의 수의 증가 또는 (2) 신뢰성 있는 프로그램의 수의 감소. 물론, 인류의 멀티코어 시스템과 소프트웨어에 대한 의존의 증가는 사소한 프로그램의 수의 빠른 증가에 대한 강력한 동기 부여가 될 겁니다.

하지만, 여러분의 코드가 너무 복잡해서 스스로가 정형 검증 도구에 크게 의존하고 있음을 발견하게 된다면 여러분의 설계를 주의 깊게 다시 생각해 봐야 하는데, 특히 여러분의 정형 검증 도구가 여러분의 코드를 특수 목적 언어로 손으로 번역될 것을 필요로 한다면 그렇습니다. 예를 들어, Section 12.1.5에서 보인 preemption 가능한 RCU의 dynticks 인터페이스의 복잡한 구현은 Section 12.1.6.9에서 이야기 된 것처럼 훨씬 더 간단한 대안 구현을 갖게 되었습니다. 다른 모든게 동일하다면 더 간단한 구현이 올바름의 증명에 훨씬 낫습니다.

그리고 정형 검증 기법과 시스템에서 일하는 분들에게 열려 있는 도전 사항은 이 요약이 틀렸음을 증명하는 겁니다! 이 일을 돋기 위해, Verification Challenge 6이 가능합니다 [McK17]. 해보세요!!!

12.7 Choosing a Validation Plan

Science is a first-rate piece of furniture for one's upper chamber, but only given common sense on the ground floor.

Oliver Wendell Holmes, updated

여러분의 프로젝트에 쓰기 위해 어떤 검증을 사용해야 할까요?

소프트웨어는 특히, 그리고 공학에서는 일반적으로 종종 그렇듯, 답은 “경우에 따라 다르다”입니다.

테스트를 수행하는 것도 정형 검증을 돌리는 것도 여러분의 프로젝트를 바꾸지는 않음을 주의하세요. 최선의 경우, 그런 노력은 나중에 고쳐지는 버그를 찾는 간접 효과를 만들어냅니다. 그렇다고 하나, 버그를 고치는 것은 불편성, 금전 손실, 재물 손상, 심지어 사망을 방지할 수도 있습니다. 분명, 이런 종류의 간접 효과는 굉장히 가치있을 수 있습니다.

앞서 보았듯, 불행히도 특정 검증 노력이 중요한 버그들을 찾을지 예측하기는 어렵습니다. 따라서 너무 적은 투자를 하기가 너무 쉽습니다—또는 투자를 하는데 아예 실패할 수도 있는데, 특히 실제 세계 소프트웨어 프로젝트에서는 거의 항상 그렇듯 개발 예측이 너무 낙관적이었다고 증명되거나, 예산이 예상과 달리 부족하거나 한 경우 그렇습니다.

그러나 검증에 투자하는 결정은 종종 강한 개성을 가진 경험 많은 사람들에 의해 이루어집니다. 그러나 다른 의결권자들 역시 강한 개성을 가지고 있을 수도 있기 때문에 보장은 없습니다. 더 나쁜게, 다른 의결권자들은 당황스러운 버그들이 최종 사용자에게 탈출하는 것을 허용한 비싼 검증 노력의 이야기를 가져올 수도 있습니다. 그러니 상처입고 머리가 하얗게 세고 기분 상한 노병이 그 날을 잘 이끌 수도 있지만, 더 잘 조직화 된 접근법이 더 유용할 수 있습니다.

다행히도, 검증에의 투자에 대한 강한 재정적 비유가 있는데, 보험 정책입니다.

보험 정책과 검증 노력은 둘 다 지속적 선행 투자를 필요로 하며, 일어날 수도 일어나지 않을 수도 있는 재앙에 대한 방어를 합니다. 더 나아가서, 둘 다 다양한 종류에 대한 배제를 갖습니다. 예를 들어, 해안가를 위한 보험 정책은 조석간만에 의한 피해를 배제하며, 우린 여지껏 모든 버그를 찾아낼 수 있는 검증 방법을 하나도 찾지 못했습니다.

더해서, 보험과 검증에 지나친 투자를 할 수도 있습니다. 한가지만 예를 들어보면, 모든 개발 예산을 쏟아부은 검증 계획은 태양이 폭발하는 것을 위한 보험 정책만큼이나 의미없습니다.

한가지 방법은 소프트웨어 예산의 특정 부분을 검증에 맡기며, 그 부분은 그 소프트웨어의 중요도에 의존하게 하여서 안전성에 치명적인 항공용 소프트웨어는 숙제용의 것에 비해 더 큰 비중의 예산을 검증에 사용하게 하는 겁니다. 가능하다면 기존의 비슷한 프로젝트에서의 경험이 사용되어야 합니다. 그러나, 검증에의 투자는 프로젝트와 동시에 시작되게 할 필요가 있는데, 그러지 않는다면 코딩에 투자하는 초과비용이 검증 노력을 침해할 겁니다.

경험 많은 사람들로 스타트업 프로젝트를 구성하는 것은 검증 노력에 대한 과다 투자를 초래할 수 있습니다.

너무 많은 보험을 구매해서 파산하는게 가능한 것처럼, 너무 많은 비용을 테스트에 투자해서 프로젝트를 망하게 할 수도 있습니다. 어떤 사용 예가 중요할지 아직 분명치 않은 처음으로 이루어지는 종류의 프로젝트에서는 모든 가능한 사용 시나리오를 위한 테스트를 하는건 상당한 시간, 에너지, 그리고 재정의 낭비일 것이기 때문에 특히 그렇습니다.

그러나, 스타트업 프로젝트에서 지원되는 일이 더욱 일상화 될수록, 사용자들은 종종 실패에 덜 너그려워지므로 검증의 필요가 증가합니다. 이런 투자 전환을 관리하는 것은 상당히 어려울 수 있는데, 특히나 사용자들이 그들의 정확한 사용처를 공개할 수 없거나 그리고 싶어하지 않는 너무나도 흔한 경우에 특히 그렇습니다. 그런 경우에는 버그 레포트와 사용자들과의 토론으로부터 그 사용처를 역공학 해내는게 치명적으로 중요해집니다. 이런 사용처가 더 잘 이해될수록 버그를 찾고 수정하는데 드는 비용을 줄이는데 지속적 통합의 (continuous integration) 사용이 더 도움 됩니다.

어떤 소프트웨어 프로젝트의 검증 사용의 진화 예가 Figure 12.4에 보여져 있습니다. figure에 보여져 있듯이, 리눅스 커널 RCU는 RCU가 이 커널에 받아들여진지 2년 후에 배포된 리눅스 커널 v2.6.15 까지도 어떤 검증 코드도 가지고 있지 않았습니다. 테스트 도구는 리눅스 커널 v2.6.19-v2.6.21에서 전체 코드 중 최대 비중을 가졌습니다. 이 비중은 v2.6.25에서 리얼타임 어플리케이션을 위한 preemption 가능한 RCU가 받아들여짐에 따라 빠르게 감소했습니다. 이 감소는 RCU API가 preemption 가능한 종류와 불가능한 종류의 RCU에서 동일했기 때문이었습니다. 이는 결국 이미 존재하는 테스트 도구가 두 변종에 모두 적용 가능해서 리눅스 커널 RCU 코드가 상당히 늘어났다 해도, 테스트를 늘릴 필요는 없음을 의미했습니다.

Figure 12.4의 뒤따르는 막대들은 RCU 코드가 상당히 증가했음을, 그러나 그에 연관된 검증 코드는 그보다도 더 극적으로 증가했음을 보입니다. 리눅스 커널 v3.5는 `rcu_barrier()` API를 위한 테스트 코드를 추가해서 테스트 범위에 오랫동안 있던 구멍을 메웠습니다. 리눅스 커널 v3.14는 자동화된 테스트와 테스트 결과 분석을 추가해, RCU를 지속적 통합 (continuous integration) 가능하게 했습니다. 리눅스 커널 v4.7은 RCU의 `update-side` 기능들을 위한 성능 검증 도구를 더했습니다. 리눅스 커널 v4.12는 개선된 `update-side` 확장성을 갖는 Tree SRCU를 추가했고, v4.13은 구식의 덜 확장 가능한 Ssrcu 구현을 제거했습니다. 리눅스 커널 v5.0은 `nolibc` 라이브러리를 `tools/include/nolibc`로 옮기기 전에 `rcutorture` 스크립트 디렉토리에 잠시 가지고 있었습니다. 리눅스 커널 v5.8은 RCU의 Tasks Trace와 Rude 변종을 추가했습니다. 리눅스 커널 v5.9은 `read-side` 성능 테스트 도구인 `refscale.c`를 추가했습니다.

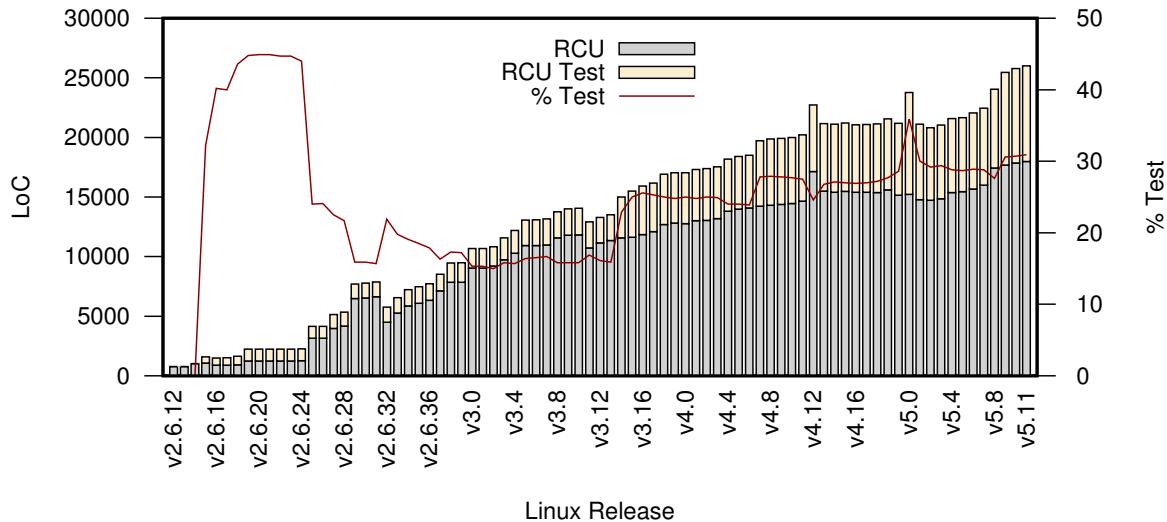


Figure 12.4: Linux-Kernel RCU Test Code

그 외에도 많은 변경 사항들을 리눅스 커널의 git 저장소에서 찾아볼 수 있습니다.

우린 검증 예산이 프로젝트마다, 그리고 특정 프로젝트에서도 시간에 따라 다름을 알아보았습니다. 하지만 검증에의 투자는 테스트와 정형 검증 사이에서 어떻게 나뉘어야 할까요?

이 질문은 컴파일러가 더 적극적인 정형 검증 기법들을 스스로의 진단 기능에 더욱 늘려가고 정형 검증 도구가 성숙해감에 따라 자연스럽게 답변될 겁니다. 또한, 리눅스 커널 lockdep과 KCSAN 도구들은 Section 11.3에서 이야기 된 것처럼 정형 검증을 수행 시간 분석과 결합하는 것의 장점을 보이고 있습니다. 또 다른 결합 기법은 수행에서 수집된 기록을 분석합니다 [dOCdO19]. 현재로써 최고의 전략은 일단 테스트에 집중하고 테스트로 처리되지 않은 프로젝트의 부분을 위해 정형 검증을 위한 명시적 작업을 예약하는 것이며, 이는 예외적 견고성을 필요로 합니다. 예를 들어, 리눅스 커널 RCU는 테스트에 기본적으로 의존합니다만 이 챕터에서 이야기 된 정형 검증을 때때로 사용했습니다.

요약하자면, 동시성 소프트웨어를 위한 검증 계획을 고르는 것은 과학보다는 예술로 남아있는 공학의 한 분야로 남아있습니다. 그러나, 점점 증가하는 엄격한 방법들이 더 널리 사용될 거라 예상할 많은 이유가 있습니다.

Chapter 13

Putting It All Together

이 chapter 는 동시성 프로그래밍 퍼즐들에 대한 힌트를 약간 제공합니다. Section 13.1 는 카운터 수수께끼를 생각해 보고, Section 13.2 는 레퍼런스 카운팅을 다시 들여다보며, Section 13.3 는 해저드 포인터를 돋고, Section 13.4 는 시퀀스 락킹의 특수한 경우들을 요약하며, 마지막으로 Section 13.5 는 RCU 의 구조를 알아봅니다.

13.1 Counter Conundrums

Ford carried on counting quietly. This is about the most aggressive thing you can do to a computer, the equivalent of going up to a human being and saying “Blood . . . blood . . . blood . . . blood . . .”

Douglas Adams

이 section 는 카운터 수수께끼들에 대한 해결책들을 정리해 봅니다.

13.1.1 Counting Updates

Schrödinger (Section 10.1 참고) 가 각 동물의 업데이트 횟수를 세고 싶어하며, 이 업데이트는 데이터 원소별 락을 이용해 동기화 된다고 해봅시다. 이 카운팅은 어떻게 해야 가장 잘 할 수 있을까요?

물론, Chapter 5 에서의 카운팅 알고리즘들이 얼마든지 충분할 수도 있지만, 최적의 방법은 상당히 간단합니다. 각 데이터 원소에 카운터를 놓고, 그 원소의 락의 보호 아래 그 카운터를 증가시키는 겁니다!

읽기 쓰레드가 락 없이 이 숫자에 접근하면, 업데이트 쓰레드는 이 카운터를 업데이트 하는데에 `WRITE_ONCE()` 를 사용하고 락 없는 읽기 쓰레드는 이를 읽기 위해 `READ_ONCE()` 를 사용해야 합니다.

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

13.1.2 Counting Lookups

Schrödinger 는 각 동물을 위한 탐색 횟수도 세고 싶어하며, 탐색은 RCU 로 보호된다고 해봅시다. 이 경우에 무엇이 최선의 카운팅 방법일까요?

한 가지 방법은 Section 13.1.1 에서 이야기 되었듯 이 탐색 카운터를 원소별 락을 이용해 보호하는 것입니다. 불행히도, 이는 각 탐색이 락을 획득할 것을 필요로 해서 큰 시스템에서는 상당한 병목지점이 될 겁니다.

또 다른 방법은 카운팅이 “안된다고 말하기”로, `noatime` mount 옵션의 예를 따릅니다. 이 방법이 적용 가능하다면, 이게 분명 최선입니다: 어쨌건, 아무것도 안하는 것보다 빠른건 없습니다. 이 탐색 카운트가 없어질 수 없다면, 계속 읽으세요!

Chapter 5 에서의 카운터들이 무엇이든 사용될 수 있겠는데, Section 5.2 의 통계적 카운터가 아마 가장 흔한 선택일 겁니다. 하지만, 이는 큰 메모리 사용량을 초래 합니다: 필요한 카운터의 수는 데이터 원소의 수 곱하기 쓰레드의 수입니다.

이 메모리 오버헤드가 지나치다면, Figure 10.3 에 보인 해쉬 테이블 성능을 참고해 CPU 별 카운터 대신 코어별 또는 소켓별 카운터를 두는 것이 한 방편이 되겠습니다. 이는 이 카운터 값 증가가 어토믹 오퍼레이션이 될 것을 필요로하는데, 특정 쓰레드가 언제든 다른 CPU 로 옮겨질 수 있는 사용자 모드 수행에서는 특히 그렇습니다.

어떤 원소들이 매우 빈번하게 탐색된다면, 쓰레드별로 특정 원소를 위한 로그 항목들이 병합될 수 있는 로그를 두어서 업데이트를 몰아서 하는 방법이 여럿 있습니다. 특정 로그 항목이 충분히 큰 값 증가를 가지거나 충분한 시간이 흐른 후에는 이 로그 항목들이 연관된 데이터 원소에 적용될 수 있습니다. Silas Boyd-Wickizer 는 이 방법들을 정형화 시켰습니다 [BW14].

13.2 Refurbish Reference Counting

Counting is the religion of this generation. It is its hope and its salvation.

Gertrude Stein

레퍼런스 카운팅이 컨셉상으로는 간단한 기법이지만, 실제로 동시성 소프트웨어에 적용하기에는 많은 어려움이 숨어있습니다. 어쨌건, 어떤 객체가 이론 제거를 당하지 않는다면, 그걸 위한 레퍼런스 카운터를 가질 이유 자체가 없습니다. 하지만 그 객체가 제거될 수 있다면, 레퍼런스 획득 프로세스 자체를 제거하는 건 뭘로 막나요?

동시성 소프트웨어에서의 레퍼런스 카운터 사용을 재고하기 위한 여러 방법들이 있는데, 다음의 것들이 포함됩니다:

1. 레퍼런스 카운트 조정 중에는 객체의 밖에 있는 락을 잡아야 합니다.
2. 이 객체는 0이 아닌 레퍼런스 카운트를 가지고 생성되며, 새 레퍼런스는 이 레퍼런스 카운터의 값이 0이 아닐 때에만 획득될 수 있습니다. 어떤 쓰레드가 특정 객체로의 참조를 가지고 있지 않다면, 참조를 이미 가지고 있는 다른 쓰레드의 도움이 필요할 수 있습니다.
3. 어떤 경우, 해저드 포인터가 레퍼런스 카운터의 자체 교체를 위해 사용될 수 있습니다.
4. 객체를 위한 존재 보장(existance guarantee)이 제공되어서 어떤 다른 것이 참조를 얻으려 하는 사이에 이 객체가 메모리 해제되는 것을 막습니다. 존재 보장은 자동화된 garbage collector에 의해 종종 제공되며, Sections 9.3 and 9.5에서 보았듯 해저드 포인터와 RCU에 의해 각각 제공되기도 합니다.
5. 타입 안전성 보장(type-safety guarantee)이 객체에 제공됩니다. 참조가 획득된 후에는 정체성 검사가 반드시 추가로 수행되어야 합니다. 타입 안전성은 특수 목적 메모리 할당자에 의해 제공될 수 있는데, Section 9.5에서 보인, 리눅스 커널에 있는 SLAB-TYPESAFE_BY_RCU가 한 예가 되겠습니다.

물론, 존재 보장을 제공하는 모든 메커니즘은 그 정의에 따라 타입 안전성 보장도 제공합니다. 이는 레퍼런스 획득 보호를 위한 네개의 일반적 카테고리를 형성합니다: 레퍼런스 카운팅, 해저드 포인터, 시퀀스 락킹, 그리고 RCU.

Table 13.1: Synchronizing Reference Counting

Acquisition	Release			
	Locks	Reference Counts	Hazard Pointers	RCU
Locks	—	CAM	M	CA
Reference Counts	A	AM	M	A
Hazard Pointers	M	M	M	M
RCU	CA	MCA	M	CA

Quick Quiz 13.1: 레퍼런스 카운터가 0이 아닐 때에만 참조를 획득하는 간단한 compare-and-swap 오퍼레이션을 사용해 구현하는 건 어떤가요?

핵심 레퍼런스 카운팅 문제는 참조의 획득과 객체의 메모리 해제 사이의 동기화이므로, Table 13.1에 보인 것처럼 아홉개의 가능한 메커니즘 조합이 존재합니다. 이 표는 레퍼런스 카운팅 메커니즘들을 다음의 넓은 카테고리들로 나눕니다:

1. 어토믹 오퍼레이션, 메모리 배리어, 정렬 제한 무엇도 없는 간단한 카운팅 (“—”).
2. 메모리 배리어 없이 하는 원자적 카운팅 (“A”).
3. 해제 시에만 메모리 배리어를 필요로 하는 원자적 카운팅 (“AM”).
4. 원자적 획득과 결합된 검사를 가지며 해제 시에만 메모리 배리어를 필요로 하는 원자적 카운팅 (“CAM”).
5. 원자적 획득 오퍼레이션과 검사가 결합된 원자적 카운팅 (“CA”).
6. 전체 메모리 배리어와 검사가 결합된 간단한 카운팅 (“M”).
7. 원자적 획득과 검사가 결합되고 획득 시에 메모리 배리어도 요구되는 원자적 카운팅 (“MCA”).

그러나, 값을 반환하는 모든 리눅스 커널의 원자적 오퍼레이션들은 메모리 배리어를 포함하게 정의되어 있으므로,¹ 모든 해제 오퍼레이션은 메모리 배리어를 포함하며, 모든 검사되는 획득 오퍼레이션 역시 메모리 배리어를 포함합니다. 따라서, “CA”와 “MCA”는 “CAM”과 동일해서, 첫번째 네개와 여섯번째의 경우들에 대한 것들만 남습니다: “—”, “A”, “AM”, “CAM”,

¹ `atomic()`와 `ATOMIC_INIT()`는 예외입니다.

and “M”. 다음 섹션들은 참조 획득과 해제가 매우 빈번하고 레퍼런스 카운트가 0인지는 매우 드물게 검사될 필요가 있을 때 성능을 개선할 수 있는 최적화 기법들을 소개합니다.

13.2.1 Implementation of Reference-Counting Categories

락킹으로 보호되는 간단한 카운팅이 (“-”) Section 13.2.1.1에 소개되고, 메모리 배리어 없는 원자적 카운팅이 (“A”) Section 13.2.1.2에 설명되며, 획득시 메모리 배리어를 갖는 원자적 카운팅이 (“AM”) Section 13.2.1.3에, 검사와 해제시 메모리 배리어를 갖는 원자적 카운팅이 (“CAM”) Section 13.2.1.4에 설명됩니다. 해저드 포인터의 사용은 page 9.3의 Section 9.3과 Section 13.3에 설명됩니다.

13.2.1.1 Simple Counting

어토믹 오퍼레이션도 메모리 배리어도 사용하지 않는 간단한 카운팅은 레퍼런스 카운터 획득과 해제가 둘 다 같은 락으로 보호될 때 사용됩니다. 이 경우, 레퍼런스 카운트 자체는 어토믹하지 않게 조정될 수 있다는게 분명한데, 락이 모든 필요한 배제, 메모리 배리어, 어토믹 오퍼레이션, 그리고 컴파일러 최적화 해제를 제공하기 때문입니다. 이는 레퍼런스 카운터 외에도 다른 오퍼레이션들을 보호하기 위해 락이 필요할 때 선택 가능한 방법입니다만, 락이 해제된 후에 이 객체로의 참조가 있어야 하는 경우는 아닙니다. Listing 13.1은 간단한 어토믹하지 않은 레퍼런스 카운팅을 구현하는데 사용될 수도 있는 간단한 API를 보입니다—간단한 레퍼런스 카운팅은 거의 항상 open-code 되지만요.

13.2.1.2 Atomic Counting

간단한 원자적 카운팅은 참조를 얻는 모든 CPU가 이미 참조를 쥐고 있을 때 사용될 수 있습니다. 이 스타일은 하나의 CPU가 스스로의 사적 사용을 위해 객체를 생성하지만 다른 CPU, 태스크, 타이머 핸들러 등에게 액세스를 허용해야만 할 때 사용됩니다. 이 객체를 넘기는 모든 CPU는 먼저 그걸 건네받을 쪽을 대신해 새로운 참조를 획득하거나 건네기를 한 후에 액세스를 하기를 삼가야 합니다. 리눅스 커널의 경우, `kref` 기능이 이 형태의 레퍼런스 카운팅을 구현하는데 사용되는데, Listing 13.2에 보여진 바와 같습니다.²

이 경우에는 락킹이 모든 레퍼런스 카운트 오퍼레이션을 보호하지 않으며, 이말은 두개의 다른 CPU가

² 리눅스 v4.10 기준입니다. 리눅스 v4.11은 완화된 순서 규칙 플랫폼에서의 효율성을 개선한 `refcount_t` API를 도입했는데, 가능적으로는 대체된 `atomic_t`와 동일합니다.

Listing 13.1: Simple Reference-Count API

```

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 }
```

동시에 레퍼런스 카운트를 조정할 수도 있음을 의미하기 때문에 어토믹 카운팅이 필요합니다. 평범한 값 증가/감소가 사용되었다면, 한쌍의 CPU는 둘 다 동시에 레퍼런스 카운트를 읽어와서 둘 다 값 “3”을 얻을 수 있습니다. 둘 모두 각자의 값을 증가시키면 둘 다 “4”를 얻게 되고, 둘 다 이 값을 카운터에 다시 저장하게 됩니다. 이 카운터의 새로운 값은 “5”여야 하므로, 값 증가 하나가 사라졌습니다. 따라서, 어토믹 오퍼레이션은 카운터 증가와 감소 둘 다에 사용되어야만 합니다.

해제가 락킹, 해저드 포인터, 또는 RCU로 보호된다면, 메모리 배리어는 필요하지 않지만, 다른 이유 때문입니다. 락킹의 경우, 락은 모든 필요한 메모리 배리어를 제공하고 (그리고 컴파일러 최적화를 해제합니다), 또한 한쌍의 해제가 동시에 수행되는 것을 방지합니다. 해저드 포인터와 RCU의 경우, 정리는 뒤로 미뤄지며, 모든 필요한 메모리 배리어나 컴파일러 최적화 해제는 해저드 풍니터나 RCU에 의해 제고오닙니다. 따라서, 두개의 CPU가 마지막 두개의 참조를 동시에 해제하려면, 실제 정리는 두 CPU 모두 각자의 해저드 포인터를 해제하거나 RCU read-side 크리티컬 섹션을 끝낼 때까지 미뤄질 겁니다.

Quick Quiz 13.2: 한 CPU가 마지막 참조를 해제한 직후에 다른 CPU가 참조를 얻는 경우를 위한 보호는 왜 필요하지 않죠?



하나의 어토믹 데이터 항목으로 구성되는 `kref` 구조체가 Listing 13.2의 라인 1~3에 보여져 있습니다. 라인 5~8의 `kref_init()` 함수는 카운터를 값 “1”로 초기화 합니다. `atomic_set()` 기능은 단순한 값 할당으로,

Listing 13.2: Linux Kernel kref API

```

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 }

```

그 이름은 `atomic_t` 데이터 타입에서 기인했지 그 동작에서 기인한게 아님을 알아두시기 바랍니다. `kref_init()` 함수는 객체 생성 중에 그 객체가 다른 CPU에게 사용 가능해지기 전에 호출되어야만 합니다.

라인 10-14의 `kref_get()` 함수는 무조건적으로 이 카운터를 원자적으로 값 증가시킵니다. `atomic_inc()` 기능은 명시적으로 컴파일러 최적화를 모든 플랫폼에서 해제할 필요는 없지만 `kref` 기능이 다른 모듈에 있고 리눅스 커널 빌드 프로세스는 모듈간 최적화를 하지 않는다는 사실 때문에 사실상 같은 효과가 있습니다.

라인 16-28의 `kref_sub()` 함수는 원자적으로 카운터의 값을 감소시키며, 그 결과가 0이면 라인 24은 명시된 `release()` 함수를 호출하고 라인 25에서 리턴하여, 호출자에게 `release()` 가 호출되었음을 알립니다. 그렇지 않으면 `kref_sub()` 은 0을 리턴하여 호출자에게 `release()` 가 호출되지 않았음을 알립니다.

Quick Quiz 13.3: Listing 13.2 의 라인 22에서 `atomic_sub_and_test()` 가 수행된 직후에, 어떤 다른 CPU가 `kref_get()` 을 호출했다고 해봅시다. 다른 CPU는 이제 해제된 객체로의 불법적인 참조를 갖게 되지 않나요?



Quick Quiz 13.4: `kref_sub()` 이 0을 리턴해서 `release()` 함수가 호출되지 않았음을 알렸다고 해봅시다. 호출자는 어떤 조건에서 객체의 지속되는 존재를 믿을 수 있나요?

**Listing 13.3:** Linux Kernel dst_clone API

```

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 }

```

Quick Quiz 13.5: 간단하게 해제 함수로 `kfree()` 를 넘기지 않는 이유가 뭐죠?



13.2.1.3 Atomic Counting With Release Memory Barrier

해제 시의 메모리 배리어와 함께 어토믹 레퍼런스 카운팅 하기는 리눅스 커널의 네트워킹 계층에서 패킷 라우팅에 사용되는 목적지 캐쉬를 추적하기 위해 사용됩니다. 실제 구현은 상당히 더 많은 것이 관여되어 있습니다; 이 섹션은 이 사용처에 들어맞는 `struct dst_entry` 레퍼런스 카운트 처리 관점에 집중하는데, Listing 13.3에 보여 있습니다.³

호출자가 이미 해당 `dst_entry` 로의 참조를 가지고 있다면 `dst_clone()` 기능이 사용될 수 있는데, 커널 내의 어떤 다른 부분에 넘겨질 수 있는 다른 참조를 얻는 경우입니다. 호출자에 의해 참조가 이미 획득되어 있으므로, `dst_clone()` 은 어떤 메모리 배리어도 수행 할 필요가 없습니다. `dst_entry` 를 다른 부분에 넘기느 행위는 메모리 배리어를 필요로 할 수도 아닐 수도 있습니다만, 그런 메모리 배리어가 필요하다면 `dst_entry` 를 넘기는데 사용되는 메커니즘 내에 내재될 겁니다.

`dst_release()` 기능은 어떤 환경에서든 수행될 수 있으며 호출자는 `dst_release()` 호출 직전에 `dst_entry` 구조체의 원소들을 참조할 수도 있습니다. 따라서 `dst_release()` 기능은 라인 14에 컴파일러와 CPU가 액세스를 잘못 순서짓는 것을 막기 위해 메모리 배리어를 갖습니다.

`dst_clone()` 과 `dst_release()` 를 사용하는 프로그래머는 메모리 배리어는 신경쓸 필요 없고, 이 두 기

³ 리눅스 v4.13 기준입니다. 리눅스 v4.14는 더 많은 디버깅 검사를 위해 우회 단계들이 추가되었지만, 전체적인 효과는 버그 없이는 동일합니다.

능을 사용하는데 필요한 규칙만을 신경써야 함을 유의하시기 바랍니다.

13.2.1.4 Atomic Counting With Check and Release Memory Barrier

호출자가 어떤 객체로의, 지금은 가지고 있지 않은 새 참조를 획득할 수 있어야만 하는데, 그 객체의 존재는 보장되어 있는 상황을 생각해봅시다. 초기의 레퍼런스 카운트 획득은 이제 레퍼런스 카운트 해제와 동시에 수행될 수 있다는 사실이 복잡도를 증가시킵니다. 한 레퍼런스 카운트 해제 오퍼레이션이 이 레퍼런스 카운트의 새 값이 0임을 확인해서 이 객체를 정리해도 안전하다는 신호를 날렸다고 해봅시다. 그런 정리가 시작된 후에는 레퍼런스 카운트 획득을 허용하지 않아야 하는 게 분명하므로, 그 획득은 레퍼런스 카운트가 0인지에 대한 검사를 포함해야 합니다. 이 검사는 어토믹 값 증가 오퍼레이션의 한 부분이어야 하는데, 아래에 보이는 것과 같습니다.

Quick Quiz 13.6: 왜 0 레퍼런스 카운트 검사는 then 절에서 어토믹 값 증가를 하는 간단한 if 문을 통해서 이뤄질 수 없죠?

■ 리눅스 커널의 `fget()` 과 `fput()` 기능은 이런 형태의 레퍼런스 카운팅을 사용합니다. 이 함수들의 간략화된 버전이 Listing 13.4에 보여져 있습니다.⁴

`fget()`의 라인 4는 현재 프로세스의 파일 디스크립터 테이블로의 포인터를 가져오는데 이는 다른 프로세스들과 공유될 수도 있습니다. 라인 6은 RCU read-side 크리티컬 섹션을 진입하는 `rcu_read_lock()`을 수행합니다. 모든 뒤따르는 `call_rcu()` 기능에서의 콜백 함수는 여기에 맞는 `rcu_read_unlock()`이 호출될 때 까지 (이 예에서는 라인 10 또는 14) 미루어질 겁니다. 라인 7은 `fd` 인자에 의해 명시된 파일 디스크립터에 연관된 파일 구조체를 탐색하는데, 뒤에서 설명됩니다. 이 명시된 파일 디스크립터에 연관된 열린 파일이 있다면 라인 9는 원자적으로 레퍼런스 카운트를 획득하려 시도합니다. 여기에 실패하면, 라인 10-11은 이 RCU read-side 크리티컬 섹션을 빠져나와 실패를 알립니다. 그렇지 않고 이 시도가 성공적이었다면 라인 14-15은 이 read-side 크리티컬 섹션을 빠져나오고 이 파일 구조체로의 포인터를 리턴합니다.

`fcheck_files()` 기능은 `fget()`을 위한 도움을 주는 함수입니다. 라인 22는 이 태스크의 현재 파일 디스크립터 테이블로의 RCU로 보호되는 포인터를 안전하게 읽어오기 위해 `rcu_dereference()`를 사용하며, 라인 24는 이 명시된 파일 디스크립터가 범위 내에 있는지

⁴ 리눅스 v2.6.38 기준입니다. 추가적인 `O_PATH` 기능이 v2.6.39에 추가되었고, v3.14에서 refactoring이 이루어졌으며 v4.1에서는 `mmap_sem` 경쟁을 완화시켰습니다.

Listing 13.4: Linux Kernel `fget/fput` API

```

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 }

```

확인합니다. 그렇다면 라인 25는 이 파일 구조체로의 포인터를 읽어오는데, 여기서도 `rcu_dereference()` 기능을 사용합니다. 라인 26는 이어서 성공 시에는 파일 구조체로의 포인터를, 실패 시에는 NULL을 리턴합니다.

`fput()` 기능은 파일 구조체로의 참조를 해제합니다. 라인 31는 이 레퍼런스 카운트를 원자적으로 감소시키며, 그 결과가 0이 된다면 라인 32은 이 파일 구조체를 모든 현재 수행중인 RCU read-side 크리티컬 섹션이 완료된 후에, 즉 한 RCU grace period가 지난 후에 메모리 해제하기 위해 `call_rcu()`를 호출합니다 (`call_rcu()`의 두번째 인자로 명시되는 `file_free_rcu()` 함수를 통해).

이 grace period가 완료되면, `file_free_rcu()` 함수는 라인 39에서 이 파일 구조체로의 포인터를 얻어와서 라인 40에서 이를 메모리 해제합니다.

따라서 이 코드 조작은 객체 내의 레퍼런스 카운트가 증가되는 동안 그 존재를 보장하기 위해 어떻게 RCU가 사용될 수 있는지 보입니다.

13.2.2 Counter Optimizations

값 증가와 감소가 혼합, 그러나 0인지 검사가 드문 어떤 경우들에선 Chapter 5에서 이야기 된 것처럼 CPU 또는 태스크별 카운터를 갖는게 유용합니다. 예를 들어, 이 기법을 RCU에 적용한 sleepable read-copy update (SRCU) 논문을 [McK06] 보시기 바랍니다. 이 방법은 값 증가와 감소 기능들에서의 어토믹 명령들이나 메모리 배리어의 필요를 제거합니다만, 여전히 코드를 변경하는 컴파일러 최적화는 해제되어야 합니다. 또한, 합쳐진 레퍼런스 카운트가 0이 되는지 검사하는 `synchronize_srcu()` 같은 기능은 상당히 느릴 수 있습니다. 이는 이 기법들이 참조가 혼하에 얻어지고 해제되지만 0 레퍼런스 카운트 검사는 가끔만 필요해지는 상황을 위해 설계되었습니다.

하지만, 레퍼런스 카운트의 사용이 그렇지 않으면 읽기만 해도 될 데이터 구조에 (많은 경우 원자적으로) 쓰기를 요구하게 되는게 혼합니다. 이 경우, 레퍼런스 카운트는 읽기 쓰레드에게 비싼 캐쉬 미스를 유발시킵니다.

따라서 읽기 쓰레드들이 횡단하는 데이터 구조에 쓰기를 하지 않아도 되는 동기화 메커니즘을 찾아볼 가치가 있습니다. 그런 한가지 가능성은 Section 9.3에서 다룬 해저드 포인터이고, 또다른 하나는 Section 9.5에서 다룬 RCU입니다.

13.3 Hazard-Pointer Helpers

It's the little things that count, hundreds of them.

Cliff Shaw

이 섹션은 해저드 테이블을 다룰 때 생길 수 있는 문제들을 알아봅니다. 이 문제들은 많은 다른 탐색 구조체들에서도 발생할 수 있음을 유의하시기 바랍니다.

13.3.1 Scalable Reference Count

레퍼런스 카운트가 성능이나 확장성의 병목이 된다고 해봅시다. 뭘 할 수 있을까요?

한가지 방법은 해저드 포인터를 대신 사용하는 겁니다.

해저드 포인터에는 몇가지 차이가 있는데, 그중 가장 눈여겨 볼만한 건 언제 연관된 레퍼런스 카운트가 0이 되었는지 확인하는 비용이 굉장히 높다는 겁니다.

이 문제를 해결하는 한가지 방법은 레퍼런스 카운트와 해저드 포인터 사이에 부하를 쪼개는 겁니다. 각 데이터 원소는 이 원소를 참조하는 다른 데이터 원소

의 수를 추적하는 동안 읽기 쓰레드는 해저드 포인터를 사용하는 겁니다.

이 방법을 효율적이면서 올바르게 사용하기는 상당한 노력이 필요하며, 따라서 관심 있는 독자 여러분은 Folly 오픈소스 라이브러리에 구현되어 있는 `UnboundQueue`와 `ConcurrentHashMap` 데이터 구조를 살펴보시기 바랍니다.⁵

13.4 Sequence-Locking Specials

The girl who can't dance says the band can't play.

Yiddish proverb

이 섹션은 시퀀스 락의 특별한 사용처들을 알아봅니다.

13.4.1 Correlated Data Elements

두개 이상의 원소들을 연관지어 봐야 하는 해쉬 테이블을 가지고 있다고 해봅시다. 이 원소들은 함께 업데이트되며, 첫번째 원소의 기존 버전을 다른 원소의 새 버전과 함께 보고 싶지 않습니다. 예를 들어, Schrödinger는 그의 in-memory 데이터베이스에 그의 동물들에 대해 확장된 가족을 넣고 싶습니다. Schrödinger는 결혼과 이혼이 급작스럽게 일어나지는 않음을 알지만, 그는 또한 전통주의자이기도 합니다. 따라서, 그는 그의 데이터베이스가 신부는 결혼했는데 신랑은 그렇지 않은, 또는 그 반대의 경우를 보이기를 원치 않습니다. 또한, Schrödinger가 전통주의자라 생각한다면, 여러분은 그의 가족 구성원들 중 일부와 대화해 보세요! 달리 말하자면, Schrödinger는 그의 데이터베이스가 결혼에 있어 일관적이길 원합니다.

한가지 방법은 시퀀스 락을 사용해서 (Section 9.4를 참고하세요), 결혼에 관련된 업데이트가 `write_seqlock()`의 보호 아래 진행되고 결혼 일관성이 피료한 읽기는 `read_seqbegin()` / `read_seqretry()` 반복문 아래에서 진행되게 하는 겁니다. 시퀀스 락은 RCU 보호의 대체제가 아님에 유의하세요: 시퀀스 락은 동시의 수정으로부터 보호를 해줍니다만, 동시의 삭제로부터의 보호를 위해선 여전히 RCU가 필요합니다.

이 방법은 연관된 원소의 수가 작고 원소들에의 읽기 시간이 짧으며, 업데이트 비율이 낮을 때 상당히 잘 작동합니다. 그렇지 않다면, 읽기 쓰레드가 영원히 완료되지 못할 수도 있게끔 업데이트가 빈번하게 이어날 수도 있습니다. Schrödinger는 이런 문제가 일어날 만큼 그의 가장 덜 정상적인 지인들이 결혼하고 곧바로 이혼 할 거라 생각하지 않지만, 그는 이 문제가 다른 환경에

⁵ <https://github.com/facebook/folly>

서는 일어날 수 있음을 알고 있습니다. 이 읽기 쓰레드 starvation 문제를 해결하는 한가지 방법은 읽기 쓰레드가 너무 많은 재시도를 했다면 업데이트 쪽 기능을 사용하게 하는 것입니다만, 이는 성능과 확장성을 모두 악화시킬 수 있습니다. Starvation 을 막는 다른 방법은 여러 시퀀스 락을 상호하는 것으로, 이 Schrödinger 의 경우엔 종당 하나가 될 수 있겠습니다.

또한, 만약 업데이트 쪽 기능이 너무 자주 사용된다면, 락 컨텐션으로 인해 낮은 성능과 확장성이 초래될 겁니다. 이를 막는 한가지 방법은 원소별 시퀀스 락을 두고 부부의 결혼 상태를 업데이트 할 때 부부 두명의 락을 모두 잡는 겁니다. 읽기 쓰레드는 이 한쌍의 멤버들의 결혼 상태에 대한 모든 변화에 대해 안정적인 읽기를 위해 이 부부의 락들 중 하나만 가지고 재시도 반복을 할 수 있습니다. 이는 높은 결혼과 이혼율에 의한 컨텐션을 막을 수 있으나, 데이터베이스의 한번의 스캔 동안의 모든 결혼 상태에 대한 일관적 시각을 얻기를 복잡하게 만들니다.

결혼 상태가 그러길 바라듯이 원소 그룹 짓기가 잘 정의되었고 영구적이라면, 가능한 한가지 방법은 데이터 원소로의 포인터들을 더해서 특정 그룹의 멤버들을 함께 연결짓는 겁니다. 그러면 읽기 쓰레드는 첫번째 원소가 발견되면 같은 그룹의 데이터 원소들을 접근하기 위해 이 포인터들을 따라갈 수 있습니다.

이 기법은 리눅스 커널에서 널리 사용되는데, dcache subsystem에서 [Bro15b] 특히 그렇습니다. 비슷한 방법이 해저드 포인터를 통해서도 동작할 수 있음을 알아 두시기 바랍니다.

또 다른 방법은 데이터 원소들을 파편화하고 각 업데이트가 그 업데이트로 영향 받는 모든 데이터 원소를 위한 모든 시퀀스 락에 대해 쓰기 권한 획득을 하게 합니다. 물론, 이 쓰기 권한 획득은 데드락을 막기 위해 신중히 가해져야 합니다. 읽기 쓰레드는 여러 시퀀스 락에 대한 읽기 권한 획득을 필요로 하겠습니다만 읽기 쓰레드가 하나의 데이터 원소만 읽어야 하는 놀랍도록 흔한 상황에서는 하나의 시퀀스 락에 대해서만 읽기 권한 획득이 필요합니다.

이 방법은 성공한 읽기 쓰레드에게 sequential consistency 를 제공하여, 각자는 특정 업데이트의 효과를 보거나 보지 못하며, 모든 중간의 업데이트는 읽기 쪽 재시도를 하게 합니다. Sequential consistency 는 극단적으로 강력한 보장으로, 동일하게 강한 제한과 높은 오버헤드를 수반합니다. 이 경우, 우린 읽기 쓰레드가 starvation 에 빠질 수 있거나 업데이트 쪽 락을 획득해야 할 수도 있음을 보았습니다. 업데이트가 흔하지 않은 경우에 이는 잘 동작하지만, 이는 업데이트가 재시도되는 읽기 쓰레드가 액세스하는 데이터에는 여향을 주지도 않은 업데이트에 조차 재시도를 불필요하게 강제합니다. 따위라서 Section 13.5.4 는 읽기 쓰레드의 starvation

을 막을 뿐 아니라 모든 형태의 읽기 쪽 재시도를 막는 완화된 형태의 일관성을 다룹니다.

13.4.2 Upgrade to Writer

Section 9.5.4.2 에서 이야기 된 것과 같이, RCU 는 읽기 쓰레드가 쓰기 쓰레드로 업그레이드 되는 것을 허용합니다. 이 능력은 RCU 로 보호되는 데이터 구조를 읽는 읽기 쓰레드가 현재 원소에 업데이트가 필요함을 알았을 때 상당히 유용합니다. 이걸 시퀀스 락을 가지고 하려면 어떻게 될까요?

이런 시퀀스 락 기법은 리눅스 커널에서 사용되고 있는데, 예를 들어 `drivers/infiniband/hw/hfi1/sdma.c` 의 `sdma_flush()` 함수에서 그렇습니다. 그 효과는 읽기 쓰레드의 재시도를 막는 겁니다. 따라서 이 기법은 읽기 쓰레드가 재시도를 필요로 하는 어떤 조건을 파악했을 때 사용됩니다.

13.5 RCU Rescues

With great doubts comes great understanding, with little doubts comes little understanding.

Chinese proverb

이 섹션은 이 책의 앞부분에서 이야기된 일부 예제들에 RCU 를 어떻게 적용할 수 있는지 보입니다. 어떤 경우, RCU 는 더 간단한 코드를 제공하고 어떤 경우에는 더 나은 성능과 확장성을 제공하며, 또 다른 경우에는 이를 모두 제공합니다.

13.5.1 RCU and Per-Thread-Variable-Based Statistical Counters

Section 5.2.3 는 단순한 값 증가의 (C 의 `++ 연산자`) 것만 큼이나 훌륭한 성능과 선형적 확장성을 제공하는—그러나 `inc_count()` 를 통한 값 증가 시에만—통계적 카운터의 구현을 보였습니다. 불행히도, `read_count()` 를 통해 값을 읽어야 하는 쓰레드는 전역 락을 획득해야 했으며, 따라서 높은 오버헤드를 일으키고 낮은 확장성으로 고통받아야 했습니다. 이 락 기반의 구현 코드는 Page 53 의 Listing 5.4 에 있습니다.

Quick Quiz 13.7: 대체 왜 우린 전역 락을 필요로 했죠?



13.5.1.1 Design

여기서 우리가 원하는건 `inc_count()` 만이 아니라 `read_count()`에서도 훌륭한 성능과 확장성을 얻기 위해 `read_count()`에서 순회를 하는 쓰레드를 `final_mutex` 가 아닌 RCU를 사용하게 하는 겁니다. 그러나, 계산된 값에 대한 정확도를 놓치고 싶지도 않습니다. 특히, 특정 쓰레드가 종료될 때, 우린 분명 이 종료되는 쓰레드의 값을 읽고 싶지 않고, 그걸 중복해 셀수도 없습니다. 그런 오류는 결과의 완전한 예측과 동일한 비정 확성을 초래한느데 달리 말하면 그런 오류는 그 결과를 완전히 쓸모없게 만들 겁니다. 그리고 실제로, `final_mutex`의 목적 중 하나는 쓰레드가 `read_count()` 수행 중간에 왔다가 갔다가 하지 않음을 보장하는 겁니다.

따라서, `final_mutex`를 제거하고자 한다면, 일관성을 보장하는 어떤 다른 방법을 사용해야 합니다. 한가지 방법은 앞서 종료된 모든 쓰레드를 위한 전체 카운트와 쓰레드별 카운터로의 포인터들의 배열을 하나의 구조체에 두는 겁니다. 그런 구조체는 일단 `read_count()`에서 사용 가능하게 하고 일관성을 유지한다면 `read_count()` 가 일관된 데이터를 볼 수 있게 합니다.

13.5.1.2 Implementation

Listing 13.5 의 라인 1-4 는 `countarray` 구조체를 보이는데, 이 구조체는 앞서 종료된 쓰레드들로부터의 카운트를 위한 `->total` 필드와 현재 수행 중인 쓰레드 각각으로의 쓰레드별 `counter`로의 포인터 배열인 `counterp[]` 를 갖습니다. 이 구조체는 특정 `read_count()` 수행이 파악된 수행중인 쓰레드 집합에 일관된 전체를 볼 수 있게 합니다.

라인 6-8 는 쓰레드별 `counter` 변수, 현재 `countarray` 구조체를 참조하는 전역 포인터 `countarrayp`, 그리고 `final_mutex` 스피너의 정의를 담고 있습니다.

라인 10-13 는 Listing 5.4 에서 달라지지 않은 `inc_count()` 를 보입니다.

라인 15-31 는 `read_count()` 를 보이는데, 상당히 달라졌습니다. 라인 22 와 29 는 `rcu_read_lock()` 과 `rcu_read_unlock()` 을 사용해 `final_mutex` 의 획득과 해제를 대체합니다. 라인 23 는 현재 `countarray` 구조체를 지역 변수 `cap` 으로 스냅샷 하기 위해 `rcu_dereference()` 를 사용합니다. RCU 의 올바른 사용은 이 `countarray` 구조체가 라인 29 에서의 현재 RCU read-side 크리티컬 섹션의 종료까지는 남아 있을 것을 보장합니다. 라인 24 는 `sum` 을 `cap->total` 로 초기화하는데, 이는 앞서 종료된 쓰레드의 카운트의 합입니다. 라인 25-27 는 현재 수행중인 쓰레드들에 연관된 쓰레드별 카운터를 더하며, 마지막으로 라인 30 에서 이 합을 리턴합니다.

Listing 13.5: RCU and Per-Thread Statistical Counters

```

1 struct countarray {
2     unsigned long total;
3     unsigned long *counterp[NR_THREADS];
4 };
5
6 unsigned long __thread counter = 0;
7 struct countarray *countarrayp = NULL;
8 DEFINE_SPINLOCK(final_mutex);
9
10 __inline__ void inc_count(void)
11 {
12     WRITE_ONCE(counter, counter + 1);
13 }
14
15 unsigned long read_count(void)
16 {
17     struct countarray *cap;
18     unsigned long *ctrp;
19     unsigned long sum;
20     int t;
21
22     rcu_read_lock();
23     cap = rcu_dereference(countarrayp);
24     sum = READ_ONCE(cap->total);
25     for_each_thread(t) {
26         ctrp = READ_ONCE(cap->counterp[t]);
27         if (ctrp != NULL) sum += *ctrp;
28     }
29     rcu_read_unlock();
30     return sum;
31 }
32
33 void count_init(void)
34 {
35     countarrayp = malloc(sizeof(*countarrayp));
36     if (countarrayp == NULL) {
37         fprintf(stderr, "Out of memory\n");
38         exit(EXIT_FAILURE);
39     }
40     memset(countarrayp, '\0', sizeof(*countarrayp));
41 }
42
43 void count_register_thread(unsigned long *p)
44 {
45     int idx = smp_thread_id();
46
47     spin_lock(&final_mutex);
48     countarrayp->counterp[idx] = &counter;
49     spin_unlock(&final_mutex);
50 }
51
52 void count_unregister_thread(int nthreadexpected)
53 {
54     struct countarray *cap;
55     struct countarray *capold;
56     int idx = smp_thread_id();
57
58     cap = malloc(sizeof(*countarrayp));
59     if (cap == NULL) {
60         fprintf(stderr, "Out of memory\n");
61         exit(EXIT_FAILURE);
62     }
63     spin_lock(&final_mutex);
64     *cap = *countarrayp;
65     WRITE_ONCE(cap->total, cap->total + counter);
66     cap->counterp[idx] = NULL;
67     capold = countarrayp;
68     rcu_assign_pointer(countarrayp, cap);
69     spin_unlock(&final_mutex);
70     synchronize_rcu();
71     free(capold);
72 }

```

countarrayp 의 초기 값은 라인 33-41에서의 count_init()을 통해 제공됩니다. 이 함수는 첫번째 쓰레드가 생성되기 전에 수행되어, 초기 구조체를 할당하고 0으로 초기화 하며 countarrayp로 그 포인터를 할당합니다.

라인 43-50는 count_register_thread() 함수를 보이는데, 이 함수는 새로 생성된 쓰레드 각각에 의해 호출됩니다. 라인 45는 현재 쓰레드의 인덱스를 얻어오고, 라인 47는 final_mutex를 획득하며, 라인 48은 이 쓰레드의 counter로의 포인터를 설치하고 라인 49은 final_mutex를 해제합니다.

Quick Quiz 13.8: 여보세요!!! Listing 13.5의 라인 48은 앞서서부터 존재한 countarray 구조의 값을 수정하잖아요! 이 구조체는 일단 read_count()에게 접근 가능해지면 일관적인 상태로 남는다고 하지 않았나요???

라인 52-72는 count_unregister_thread()를 보이는데, 이 함수는 각 쓰레드가 종료되기 직전에 호출됩니다. 라인 58-62는 새 countarray 구조체를 할당하고, 라인 63는 final_mutex를 획득하고 라인 69에서 해제합니다. 라인 64는 현재 countarray의 내용을 새로 할당된 버전에 복사하며, 라인 65는 종료되고 있는 쓰레드의 counter를 새 구조체의 ->total에 더하며, 라인 66은 이 종료되는 중인 쓰레드의 counterp[] 배열 원소를 NULL로 만듭니다. 라인 67은 이어서 현재(곧 과거 버전이 될) countarray 구조체로의 포인터를 보존하고, 라인 68은 countarray 구조체의 새 버전을 설치하기 위해 rCU_assign_pointer()를 사용합니다. 라인 70은 하나의 grace period가 지나가길, 그래서 동시에 read_count()를 수행 중일 수 있는, 따라서 기존 countarray 구조체로의 참조를 가지고 있을 수도 있는 쓰레드들이 각자의 RCU read-side 크리티컬 섹션을 빠져나와 모든 그런 참조를 내려놓을 때까지 기다립니다. 라인 71은 이제 기존의 countarray 구조체를 안전하게 메모리 해제할 수 있습니다.

Quick Quiz 13.9: 고정된 크기의 counterp 배열을 가지고, 이 코드는 어떻게 쓰레드의 수의 고정된 최대 한계 문제를 어떻게 회피하나요?

13.5.1.3 Discussion

Quick Quiz 13.10: 와! Listing 5.4의 42 라인 코드에 의해 Listing 13.5는 70 라인 코드를 갖는군요. 이 여분의 복잡도는 그럴만한 가치가 있나요?

RCU의 사용은 종료되는 쓰레드가 다른 쓰레드가 종료되는 쓰레드의 __thread 변수 사용을 마쳤음이 보장될 때까지 기다리는 것을 가능하게 해줍니다. 이는

read_count() 함수가 락킹을 제거할 수 있게 해주며, 따라서 inc_count()와 read_count() 함수 모두에 훌륭한 성능과 확장성을 제공하게 해줍니다. 그러나, 이 성능과 확장성은 코드 복잡도의 약간의 증가의 비용을 갖습니다. 컴파일러와 라이브러리 개발자들이 안전한 쓰레드간 __thread 변수 접근을 위해 user-level RCU [Des09b]를 채택해서 __thread 변수들의 사용자들에게 보여지는 복잡도를 낮출 수 있기를 희망합니다.

13.5.2 RCU and Counters for Removable I/O Devices

Section 5.4.6는 제거 가능한 기기로의 I/O 액세스를 세기 위한 한쌍의 코드 조각을 보였습니다. 이 코드 조각들은 reader-writer 락을 획득함으로써 생기는 fastpath(I/O 시작)에서의 높은 오버헤드로 고통받았습니다.

이 섹션은 이 오버헤드를 막기 위해 RCU가 어떻게 사용될 수 있는지 보입니다.

I/O를 수행하는 코드는 원래의 것과 상당히 비슷한데 원본에서의 reader-writer 락 read-side 크리티컬 섹션이 RCU read-side 크리티컬 섹션으로 대체되었습니다:

```

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 }
```

이 RCU read-side 기능들은 최소한의 오버헤드를 가지며, 따라서 희망한대로 fastpath 속도를 증가시킵니다.

기기를 제거하는 코드 조각의 업데이트된 버전은 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 우리는 reader-writer 락을 배타적 스팬락으로 교체하고 모든 RCU read-side 크리티컬 섹션이 완료되기를 기다리기 위해 synchronize_rcu()를 추가합니다. synchronize_rcu() 때문에, 우리가 일단 라인 6에 다다르면 모든 남아있는 I/O는 세어졌음을 압니다.

물론, synchronize_rcu()의 오버헤드는 클 수 있으나, 기기의 제거가 상당히 드물게 이어질 것이라 생각하면 이는 좋은 트레이드오프입니다.

Listing 13.6: RCU-Protected Variable-Length Array

```

1 struct foo {
2     int length;
3     char *a;
4 };

```

Listing 13.7: Improved 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 };

```

13.5.3 Array and Length

Listing 13.6에 보인 것과 같은 RCU로 보호되는 유동 길이 배열이 있다고 해봅시다. 배열 `->a[]`의 길이는 언제든 동적으로 변화할 수 있고, 그 길이는 `->length` 필드에 의해 주어집니다. 물론, 이는 다음과 같은 race condition을 만듭니다:

1. 이 배열은 초기엔 16 캐릭터 길이이며, 따라서 `->length`는 16입니다.
2. CPU 0이 `->length`의 값을 읽어와 16을 얻습니다.
3. CPU 1이 이 배열의 길이를 8로 줄이고, 이 새 8-캐릭터 블록 메모리로의 포인터를 `->a[]`에 할당합니다.
4. CPU 0이 `->a[]`로의 새 포인터를 가져오고 새 값을 원소 12에 저장합니다. 이 원소는 8개 캐릭터만 가지므로 이는 SEGV 또는 (더 나쁜) 메모리 오염을 일으킵니다.

이걸 어떻게 막을 수 있을까요?

한가지 방법은 Chapter 15에서 다룬 메모리 배리어를 주의 깊게 사용하는 겁니다. 이는 동작하지만, 읽기 쪽의 오버헤드를 일으키며, 어쩌면 더 나쁘게도 명시적 메모리 배리어의 사용을 평균으로 합니다.

더 나은 방법은 Listing 13.7 [ACMS03]에 보인 것처럼 값과 배열을 같은 구조체에 넣는 것입니다. 그러면 새 배열을 (`foo_a` 구조체) 할당하는 것은 자동적으로 그 배열의 길이를 위한 공간을 제공합니다. 이는 어떤 CPU가 `->fa`로의 참조를 얻으면 `->length`는 `->a[]`에 맞을 것이 보장됨을 의미합니다.

1. 이 배열은 초기에 16 캐릭터 길이이므로, `->length`는 16입니다.
2. CPU 0이 `->fa`의 값을 읽어 값 16과 16 바이트 배열을 갖는 구조체로의 포인터를 얻어옵니다.

Listing 13.8: Uncorrelated Measurement Fields

```

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 };

```

3. CPU 0이 `->fa->length`를 읽어 값 16을 얻습니다.
4. CPU 1이 이 배열을 길이 8로 줄이고, 8 캐릭터 메모리 블록을 새롭게 담고 있는 새로운 `foo_a` 구조체로의 포인터를 `->fa`에 할당합니다.
5. CPU 0이 `->a[]`로부터 새로운 포인터를 가져오고, 새 값을 원소 12에 저장합니다. 하지만 CPU 0은 여전히 16 바이트 배열을 갖는 기존의 `foo_a` 구조체를 참조하고 있으므로 아무 문제 없습니다.

물론, 두 경우 모두 CPU 1은 기존 배열을 메모리 해제하기 전에 하나의 grace period를 기다려야만 합니다. 이 방법의 더 범용적인 버전을 다음 섹션에서 보입니다.

13.5.4 Correlated Fields

Schödinger의 동물들 각각이 Listing 13.8에 보인 데이터 원소로 표현된다고 해봅시다. `meas_1`, `meas_2`, 그리고 `meas_3` 필드는 주기적으로 업데이트되는 연관된 측정의 집합입니다. 읽기 쓰레드들은 이 세개의 값을 단일한 측정 업데이트로부터 본다는 게 굉장히 중요합니다: 만약 어떤 읽기 쓰레드가 `meas_1`의 기존 값을 읽지만 `meas_2`와 `meas_3`의 새 값을 읽는다면, 그 읽기 쓰레드는 완전한 혼란에 빠질 겁니다. 읽기 쓰레드가 이 세개의 값의 연관 집합을 보는 걸 어떻게 보장할 수 있을까요?

한가지 방법은 새로운 `animal` 구조체를 할당하고, 기존 구조체를 새 구조체에 복사한 후, 새 구조체의 `meas_1`, `meas_2`, 그리고 `meas_3` 필드를 업데이트 한 후 포인터를 업데이트 함으로써 기존 구조체를 새 구조체로 교체하는 겁니다. 이는 모든 읽기 쓰레드가 측정값들의 연관 집합을 볼 것을 보장합니다만, `->photo[]` 필드 때문에 큰 구조체의 복사를 필요로 합니다. 이 복사는 받아들여질 수 없을 정도로 큰 오버헤드를 일으킬 수도 있습니다.

또 다른 방법은 Listing 13.9 [McK04, Section 5.3.4]에서 보인 것처럼 우회의 단계를 사용하는 겁니다. 새로운 측정값이 취해질 때, 새로운 `measurement` 구조체가

⁶ 이 상황은 앞의 section에서는 데이터 원소들의 그룹에 대한 일관된 모습을 봐야 했던 것과 달리 여기선 읽기 쓰레드가 특정 데이터 원소의 일관된 모습을 보기만 하면 된다는 걸 제외하면 Section 13.4.1에서 이야기한 것과 비슷합니다.

Listing 13.9: Correlated Measurement Fields

```

1 struct measurement {
2     double meas_1;
3     double meas_2;
4     double meas_3;
5 };
6
7 struct animal {
8     char name[40];
9     double age;
10    struct measurement *mp;
11    char photo[0]; /* large bitmap. */
12 };

```

할당되고, 새 측정값으로 채워지며, `animal` 구조체의 `->mp` 필드는 `rcu_assign_pointer()`를 사용해 이 새로운 `measurement` 구조체를 가리키게 업데이트 됩니다. 하나의 `grace period`가 지난 후, 기존의 `measurement` 구조체는 메모리 해제될 수 있습니다.

Quick Quiz 13.11: 하지만 Listing 13.9에 보인 방법은 추가적인 캐쉬 미스를 일으켜서 추가적인 읽기 쪽 오버헤드를 일으키지 않나요?

■ 이 방법은 읽기 쓰레드가 선택된 필드들의 연관 값을 볼 수 있게 하지만, 최소한의 읽기 쪽 오버헤드를 일으킵니다. 이 데이터 원소별 일관성은 읽기 쓰레드가 각 데이터 원소만 바라보는 혼한 경우에는 충분합니다.

13.5.5 Update-Friendly Traversal

해쉬 테이블 내의 모든 원소에 대한 통계적 스캐닝이 필요하다고 해봅시다. 예를 들어, Schrödinger는 그의 모든 동물들의 평균 신장 대비 체중을 계산하고 싶을 수도 있습니다.⁷ 더 나아가 Schrödinger는 이 통계 스캐닝이 진행중인 중간에 해쉬 테이블에 더해지거나 빼진 동물로 인한 약간의 오류는 무시하고자 한다고 해봅시다. 이때의 동시성 제어를 위해 Schrödinger는 뭘 해야 할까요?

한가지 방법은 이 통계적 스캐닝 전체를 하나의 RCU read-side 크리티컬 섹션으로 감싸는 겁니다. 이는 업데이트가 스캐닝에 지나친 영향을 끼치지 않으면서 동시에 일어날 수 있게 합니다. 특히, 이 스캐닝은 업데이트를 막지 않으며 반대도 마찬가지여서, 매우 큰 수의 원소들을 가지고 있는 해쉬 테이블의 스캐닝을 심지어 높은 업데이트율을 가지고 있을 때에도 우아하게 지원해줍니다.

Quick Quiz 13.12: 하지만 이 스캐닝은 크기 조절 가능 해쉬 테이블이 크기 재조정인 동안에는 어떻게 동작하나요? 그런 경우, 기존의 해쉬 테이블도 새 해쉬

테이블도 그 해쉬 테이블 내에 모든 원소를 가지고 있다 고 보장되지 않습니다!

13.5.6 Scalable Reference Count Two

레퍼런스 카운트가 성능 또는 확장성의 병목이 되고 있다고 해봅시다. 뭘 할 수 있을까요?

또 다른 방법은 각 레퍼런스 카운트를 위한 CPU 별 카운터를 사용하는 것으로, Chapter 5의 알고리즘, 특히 Section 5.4의 정확한 한계 카운터와 어떤 점에서 비슷합니다. 이 카운터들을 위한 CPU 별 모드와 전역 모드 사이의 전환의 필요는 한편에서는 비싼 값 증가와 감소를 초래하거나 (Section 5.4.1) 다른 편에선 POSIX 시그널의 사용을 초래합니다 (Section 5.4.3).

또 다른 대안은 CPU 별과 글로벌 모드 사이의 전환을 RCU로 조정하는 겁니다. 각 업데이트는 RCU read-side 크리티컬 섹션 내에서 진행되며, 각 업데이트는 CPU 별 카운터를 업데이트 할지 전역 값을 업데이트 할지 알리는 플래그를 검사합니다. 모드를 전환하기 위해선 플래그를 업데이트하고, 하나의 `grace period`를 기다린 후, 남아있는 모든 카운트를 CPU 별 카운터에서 글로벌 카운터로, 그리고 그 반대의 경우에도 마찬가지로 옮깁니다.

리눅스 커널은 `percpu_ref` 형태의 레퍼런스 카운터에 이 방법을 사용하는데, 관심있는 독자 여러분은 참고하시기 바랍니다.

⁷ 왜 그런 정보가 유용하나요? 날 때리세요! 하지만 그룹별 통계는 종종 유용합니다.

If a little knowledge is a dangerous thing, just think what you could do with a lot of knowledge!

Unknown

Chapter 14

Advanced Synchronization

이 챕터는 lockless 알고리즘과 병렬 리얼타임 시스템에서 사용되는 동기화 기법들을 다룹니다.

Lockless 알고리즘이 극단적 요구사항을 마주할 때에는 상당히 도움이 되지만, 만병통치약은 아닙니다. 예를 들어, Chapter 5 의 끝에서 이야기 되었듯, lockless 알고리즘을 생각하기 전에 여러분은 충분히 파티셔닝, 배칭, 그리고 잘 테스트되고 패키지 된 완화된 API 들을 (Chapters 8 and 9 를 참고하세요) 사용해야 합니다.

하지만 그걸 다 해본 후에도 여러분은 이 챕터에 설명되는 고급 기법들을 필요로 하게 될 수도 있습니다. 그런 경우, Section 14.1 는 락을 회피하기 위해 그동안 사용되어온 기법들을 요약하고 Section 14.2 는 non-blocking 동기화에 대한 간단한 개론을 제공합니다. 메모리 순서 규칙은 또한 매우 중요합니다만 그건 자신만의 chapter, 즉 Chapter 15 를 갖습니다.

고급 동기화의 두번째 형태는 병렬 리얼타임 컴퓨팅에서 필요한 더 강한 진행 보장을 제공하는데, Section 14.3 의 주제입니다.

14.1 Avoiding Locks

We are confronted with insurmountable opportunities.

Walt Kelly

락킹이 제품단계의 병렬성을 위한 일등공신이지만, 많은 상황에서 성능, 확장성, 그리고 리얼타임 응답이 모두 lockless 기법을 사용해 크게 개선될 수 있습니다. 그런 lockless 기법의 특히 인상적인 예는 Section 5.2 에서 설명한 통계적 카운터로 락만이 아니라 read-modify-write 어토믹 오퍼레이션, 메모리 배리어, 심지어 카운터 값 증가를 위한 캐쉬 미스까지 제거했습니다. 우리가 다른 다른 예제들은 다음을 포함합니다:

1. Chapter 5 의 많은 다른 카운팅 알고리즘의 빠른 수행경로들.
2. Section 6.4.3 의 자원 할당 캐쉬의 빠른 수행경로.
3. Section 6.5 의 미로 풀이기.
4. Chapter 8 의 데이터 소유권 기법.
5. Chapter 9 의 레퍼런스 카운팅, 해저드 포인터, 그리고 RCU 기법들.
6. Chapter 10 의 탐색 코드경로들.
7. Chapter 13 의 많은 기법들.

요약하자면, lockless 기법들은 상당히 유용하며 널리 사용됩니다. 그러나, lockless 기법은 예를 들면 `inc_count()`, `memblock_alloc()`, `rcu_read_lock()` 등 의 잘 정의된 API 뒤에 숨겨지는게 가장 좋습니다. 그 이유는 lockless 기법의 훈련되지 않은 사용은 처리하기 어려운 버그를 만드는 좋은 방법이기 때문입니다. 그런 버그를 찾고 고치는게 애초에 나오지 않게 하는 것보다 쉽다고 생각한다면 부디 Chapters 11 and 12 을 다시 읽어보시기 바랍니다.

14.2 Non-Blocking Synchronization

Never worry about theory as long as the machinery does what it's supposed to do.

Robert A. Heinlein

Non-blocking synchronization (NBS) [Her90] 라는 용어는 다음과 같이 다른 진행 보장 [ACHS13] 을 갖는 일곱개의 클래스의 linearizable 알고리즘을 의미합니다:

1. *Bounded wait-free synchronization*: 모든 쓰레드가 특정 유한 시간 내에 진행을 냅니다 [Her91]. 이 단계는 이뤄질 수 없는 것으로 널리 여겨지는데, Alitarh 등이 이를 생략한 [ACHS13] 이유일 수도 있습니다.
2. *Wait-free synchronization*: 모든 쓰레드가 유한 시간 내에 진행을 냅니다 [Her93].
3. *Lock-free synchronization*: 최소 하나의 쓰레드는 유한 시간 내에 진행을 냅니다 [Her93].
4. *Obstruction-free synchronization*: 경쟁이 없을 때 모든 쓰레드가 유한 시간 내에 진행을 냅니다 [HLM03].
5. *Clash-free synchronization*: 경쟁이 없을 때 최소 하나의 쓰레드는 유한 시간 내에 진행을 냅니다 [ACHS13].
6. *Starvation-free synchronization*: 실패가 없을 때 모든 쓰레드가 유한 시간 내에 진행을 냅니다 [ACHS13].
7. *Deadlock-free synchronization*: 실패가 없을 때 최소 하나의 쓰레드는 진행을 냅니다 [ACHS13].

NBS 클래스 1, 2, 그리고 3 이 1990년대 초에, 클래스 4는 2000년대 초, 그리고 클래스 5는 2013년에 처음 입안되었습니다. 마지막 두개의 클래스는 수십년 동안 비공식적으로 널리 쓰였으나 2013년에 다시 정해졌습니다.

이론상, 모든 병렬 알고리즘은 wait-free 형태로 만들 수 있으나, 널리 쓰이는 NBS 알고리즘은 상대적으로 적습니다. 이것들 중 일부를 다음 섹션에서 나열합니다.

14.2.1 Simple NBS

아마도 가장 간단한 NBS 알고리즘은 `fetch-and-add` (`atomic_add_return()`) 기능을 사용한 원자적 정수 카운터 업데이트입니다.

14.2.1.1 NBS Sets

또다른 간단한 NBS 알고리즘은 배열에 정수들의 집합을 구현합니다. 여기서 배열 인덱스는 이 집합의 멤버일 수도 있는 값을 알리며 배열의 원소는 그 값이 정말로 집합 멤버인지 알립니다. NBS 알고리즘의 *linearizability* 기준은 이 배열에 대한 읽기와 쓰기가 어토믹 인스트럭션을 사용하거나 메모리 배리어를 동반해야 할 것을 필요로 합니다만, 그리 드물지는 않은 경우에 *linearizability*는 중요치 않으며, 간단한 `volatile` 로드와 스트어

만으로도 충분한데, 예를 들어 `READ_ONCE()` 와 `WRITE_ONCE()` 를 사용하는 겁니다.

NBS 집합은 비트맵을 사용해 구현될 수도 있는데, 이 집합의 멤버일 수도 있는 각 값이 하나의 bit 에 연관되는 겁니다. 읽기와 쓰기는 일반적으로 원자적 bit 조정 인스트럭션을 통해 이루어져야 하는데, `compare-and-swap` (`cmpxchg()` 또는 `CAS`) 인스트럭션들도 사용될 수 있는 합니다.

14.2.1.2 NBS Counters

Section 5.2 에서 설명한 통계적 카운터 알고리즘들은 *bounded-wait-free* 로 여겨질 수 있으나, 그 합이 정확한 아니라 대략적인 것으로 여겨진다는 귀여운 정의상의 트릭이 사용될 때만 그렇습니다.¹ `read_count()` 함수가 이 카운터들의 합을 구하는데 걸리는 시간의 길이에 의해 정해지는 충분히 큰 오류 범위가 주어진다면 어떤 *linearizable* 하지 않은 행동이든 일어날 수 있다는 증명은 불가능합니다. 이 알고리즘은 리눅스 커널에서 가장 널리 쓰이는 NBS 알고리즘일 겁니다.

14.2.1.3 Half-NBS Queue

또다른 흔한 NBS 알고리즘은 userspace-RCU 라이브러리 구현 [Des09b] 을 보이는 Listing 14.1 에 보인 것과 같이 원소들이 원자적 교환 인스트럭션을 이용해 넣어지고 [MS98b], 이어서 이 새 원소의 앞 원소의 `->next` 포인터가 저장되는 원자적 `queue` 일 겁니다. 라인 9 은 `tail` 포인터를 새 원소를 참조하게 업데이트하면서 지역 변수 `old_tail` 에 그 앞 원소로의 참조를 저장합니다. 라인 10 는 이어서 앞 원소의 `->next` 포인터가 이 새로 추가된 원소를 참조하게 업데이트하고, 마지막으로 라인 11 는 이 `queue` 가 초기에 비어 있었는지 여부를 알리는 값을 리턴합니다.

하나의 원소를 `dequeue` 하기 위해선 상호적 배타가 필요하지만 (따라서 `dequeue` 는 `block` 됩니다), 이 `queued` 의 전체 내용에 대한 `non-blocking` 제거가 가능합니다. 불가능한 건 특정 원소를 `non-blocking` 형식으로 `dequeue` 하는 겁니다: `enqueue` 를 한 쓰레드는 라인 9 와 10 사이에서 실패했을 수도, 따라서 요청된 원소는 부분적으로 `enqueue` 되었을 수도 있습니다. 이는 `enqueue` 는 NBS 이지만 `dequeue` 는 `blocking` 인 half-NBS 알고리즘이 됩니다. 그러나 이 알고리즘은 현실에서 널리 쓰이고 있는데, 부분적으로는 대부분의 제품 소프트웨어가 임의의 `fail-stop` 오류들에 내성을 가질 필요를 갖지 않기 때문입니다.

¹ 인용이 필요합니다. 전 Mark Moir로부터 이 트릭을 들었습니다.

Listing 14.1: NBS Enqueue Algorithm

```

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 }

```

14.2.1.4 NBS Stack

Listing 14.2 는 NBS 스택을 이루는 lock-free push 와 bounded wait-free pop 을 뽑내는 LIFO push 알고리즘 (lifo-push.c) 을 보입니다. 이 알고리즘의 원조는 알려지지 않았으나, 1975년에 얻어진 특허 [BS75] 에서 참조되었습니다. 이 특허는 1973년에 청구되었는데 여러분의 편집자가 하나의 CPU 만을 가지고 있던 그의 컴퓨터를 구매하기 몇달 전이었습니다.

라인 1-4 는 임의의 값과 스택의 다음 구조체로의 포인터를 포함하는 `node_t` 구조체를 보이며 라인 7 은 스택 꼭대기 포인터를 보입니다.

`list_push()` 함수가 라인 9-20 에 펼쳐져 있습니다. 라인 11 은 새 노드를 할당하고 라인 14 는 이를 초기화 합니다. 라인 17 는 새로 할당된 노드의 `->next` 포인터를 초기화 하고, 라인 18 는 이를 스택에 밀어 넣으려 합니다. 라인 19 가 `cmpxchg()` 가 실패했음을 파악하면, 반복문을 통한 또 다른 시도가 이루어집니다. 그렇지 않다면, 이 새 노드는 성공적으로 넣어졌으며, 이 함수는 호출자에게 리턴됩니다. 라인 19 는 두개의 동시의 `list_push()` 인스턴스가 스택에 push 를 하려는 시도로 인한 경주 상황을 해결함을 유의하십시오. `cmpxchg()` 는 하나에는 성공하고 다른 하나에는 실패할 것이어서 그 다른 쪽은 재시도를 하게 하며, 따라서 스택의 두 노드에 대한 임의의 순서를 선택합니다.

`list_pop_all()` 함수가 라인 23-34 에 있습니다. 라인 25 의 `xchg()` 문은 원자적으로 스택의 모든 노드를 제거하고, 그 결과 만들어지는 리스트의 `head` 를 지역 변수 `p` 에 저장하고 `top` 을 `NULL` 로 만듭니다. 이 어토믹 오퍼레이션은 `list_pop_all()` 로의 동시 호출들을 줄 세웁니다: 그 중 하나는 리스트를 얻을 것이고, 다른 것들은 `NULL` 포인터를 얻을 것인데 동시에 `list_push()` 호출이 없었다는 가정 하에서입니다.

Listing 14.2: NBS Stack Algorithm

```

1 struct node_t {
2     value_t val;
3     struct node_t *next;
4 };
5
6 // LIFO list structure
7 struct node_t* top;
8
9 void list_push(value_t v)
10 {
11     struct node_t *newnode = malloc(sizeof(*newnode));
12     struct node_t *oldtop;
13
14     newnode->val = v;
15     oldtop = READ_ONCE(top);
16     do {
17         newnode->next = oldtop;
18         oldtop = cmpxchg(&top, newnode->next, newnode);
19     } while (newnode->next != oldtop);
20 }
21
22
23 void list_pop_all(void (foo)(struct node_t *p))
24 {
25     struct node_t *p = xchg(&top, NULL);
26
27     while (p) {
28         struct node_t *next = p->next;
29
30         foo(p);
31         free(p);
32         p = next;
33     }
34 }

```

`p`에 비어있지 않은 리스트를 얻는 `list_pop_all()` 인스턴스는 이 리스트를 라인 27-33 에 놓여진 반복문을 통해 처리합니다. 라인 28 는 `->next` 포인터를 읽어오고, 라인 30 는 `foo()` 에 의해 참조되는 함수를 현재 노드에 대해 호출하고, 라인 31 는 현재 노드를 메모리 해제하며, 라인 32 는 `p`를 이 반복문의 다음 회차를 위해 설정합니다.

하지만 한쌍의 `list_push()` 인스턴스가 단일 Node A 를 초기에 담고 있는 리스트에 동시에 `list_pop_all()` 과 함께 호출되었다고 해봅시다. 이 시나리오가 만들 수 있는 한 방법은 아래와 같습니다:

1. 첫번째 `list_push()` 인스턴스는 새 Node B 를 밀어넣고, 라인 17 를 지나 수행되어, Node A 로의 포인터를 Node B 의 `->next` 포인터에 저장합니다.
2. `list_pop_all()` 인스턴스가 완료되어, `top` 을 `NULL` 로 설정하고 Node A 를 메모리 해제합니다.
3. 두번째 `list_push()` 인스턴스가 완료되어, 새 Node C 를 밀어넣지만 Node A 에 속했던 메모리를 할당받게 됩니다.
4. 첫번째 `list_push()` 인스턴스가 라인 18 에서 `cmpxchg()` 를 수행합니다. 새 Node C 는 새로 메모리 해제된 Node A 와 동일한 주소를 가지므로 이

`cmpxchg()` 는 성공하고 이 `list_push()` 인스턴스는 완료됩니다.

두개의 밀어넣기와 전체 꺼내기가 Node A 의 메모리의 재사용에도 불구하고 모두 성공적으로 수행되었음을 유의하세요. 이는 일반적이지 않은 특성입니다: 대부분의 데이터 구조는 ABA 문제라고 종종 불리는 문제에 대한 보호를 필요로 합니다.

하지만 이는 어셈블리어로 쓰인 알고리즘에만 적용됩니다. 슬픈 사실은 대부분의 언어들이(C 와 C++ 포함) Node B 의 `->next` 포인터에 저장된 기존 Node A 로의 포인터 같은, 수명이 끝난 객체로의 포인터를 지원하지 않는다는 것입니다. 실제로, 컴파일러는 두 포인터가 (p 와 q 라고 해봅시다) `malloc()` 에 대한 두개의 다른 호출에서 리턴되었다면 그 포인터들은 동일하지 않아야 한다고 가정할 권리를 갖습니다. 실제 컴파일러들은 정말로 `p==q` 비교에 응답으로 항상 `false` 를 생성할 겁니다. 메모리 해제되었지만 호환 가능한 타입의 객체로 재할당된 객체로의 포인터는 좀비 보인터라고 명명됩니다.

많은 동시성 어플리케이션이 주의 깊게 메모리 할당자를 컴파일러로부터 숨겨서 컴파일러가 옮기지 않은 가정을 하는 것을 방지함으로써 이 문제를 막습니다. 이런 난독화 시도는 현재 실전에서 잘 동작합니다만, 언젠가는 갈수록 강화되어가는 최적화 기법들의 희생자가 될 수도 있습니다. 이 문제를 해결하기 위한 C 와 C++ 표준위원회의 노력이 진행중입니다[MMS19, MMM⁺20]. 그동안은, ABA에 내성이 있는 알고리즘을 아주 주의하며 짜도록 연습해 주시기 바랍니다.

Quick Quiz 14.1: 왜 C 와 C++ 같은 고전 언어를 더 현대적인 걸로 대체하지 않죠?



14.2.2 Applicability of NBS Benefits

가장 널리 인용된 NBS 의 장점은 그것의 진행 보장, fail-stop 버그에 대한 내성, 그리고 linearizability 에서 나옵니다. 다음 섹션들 중 하나에서 이것들 각각을 논합니다.

14.2.2.1 NBS Forward Progress Guarantees

NBS 의 진행 보장은 이를 리얼타임 시스템에서 사용하도록 많은 사람들에게 제안하게 만들었으며 NBS 알고리즘은 실제로 매우 많은 그런 시스템에서 사용되었습니다. 그러나, 진행 보장은 리얼타임 프로그래밍의 기초를 이루는 것들과 많은 면에서 직교합니다:

1. 리얼타임 진행 보장은 일반적으로 그것들과 연관된 어떤 특정 시간이 있는데, 예를 들면 “스케줄

응답시간은 100 마이크로세컨드 미만이어야 한다” 같은 겁니다. 반대로, NBS 의 가장 흔한 형태는 명확한 경계 없이 어떤 유한 시간 내에 만들어질 것만을 보장합니다.

2. 리얼타임 존재 보장은 종종 확률적인데, 소프트 리얼타임에서의 “99.9 % 확률로 스케줄 응답시간은 100 마이크로세컨드 미만이어야 한다” 같은 겁니다. 대조적으로, 많은 NBS 의 진행 보장은 무조건적입니다.
3. 리얼타임 존재 보장은 종종 환경적 제한에 따라 조건적인데, 예를 들면: (1) 가장 높은 우선순위 태스크에 한하여, (2) 각 CPU 가 각자의 시간 중 최소 특정 부분은 idle 로 보내며, (3) I/O 비율이 어떤 명시된 최대값 미만일 때. 대조적으로, NBS 의 진행 보장은 종종 무조건적인데, 최근의 NBS 는 조건적 보장을 수용하기 합니다 [ACHS13].
4. 리얼타임 프로그램의 환경에서 중요한 요소 중 하나는 스케줄러입니다. NBS 알고리즘은 최악의 경우의 악마같은 스케줄러를 가정합니다. 반대로, 리얼타임 시스템은 스케줄러가 알려진 모든 제한을 만족시키기 위해 최선을 다하며 그런 제한이 없을 때에는 프로세스 우선순위를 따르고 같은 우선순위의 프로세스에게는 공정한 스케줄링을 하기 위해 최선을 다한다고 가정합니다. 이런 악마적이지 않은 스케줄러에 대한 가정은 리얼타임 프로그램이 NBS 에 요구되는 것 같은 것들보다 [ACHS13, Bra11] 간단한 알고리즘을 사용할 수 있게 합니다.
5. 리얼타임 진행 보장은 보통 소프트웨어 버그가 없을 때에 적용됩니다. 반대로, 많은 종류의 NBS 보장은 fail-stop 버그가 있을 때 조차도 적용됩니다.
6. NBS 진행 보장은 linearizability 를 내포합니다. 반대로, 리얼타임 존재 보장은 종종 linearizability 같은 순서 규칙 제한에 종속되지 않습니다.

다시 반복하자면, 이런 차이에도 불구하고 NBS 알고리즘 여럿이 리얼타임 프로그램에서 굉장히 유용합니다.

14.2.2.2 NBS Fail-Stop Tolerance

NBS 알고리즘들 가운데 wait-free 동기화 (경계선이 있든 없든), lock-free 동기화, obstruction-free 동기화, 그리고 clash-free 동기화는 fail-stop 버그의 존재에도 불구하고 진행을 보장합니다. Fail-stop 버그의 한 예는 어떤 쓰레드가 무한정 preemption 되게 할 수도 있습니다. 곧 보게 되겠지만, 이 fail-stop 내성은 유용할 수 있지만, 사실 fail-stop 에 내성이 있는 메커니즘 집합을 염는 것은

꼭 fail-stop 내성이 있는 시스템을 만들지는 않습니다. 이를 더 자세히 보기 위해, wait-free queue 여럿을 이용해 만들어져서 어떤 원소가 그 중 하나의 queue에서 제거되고 처리되고 다음 queue에 더해지는 시스템을 생각해 봅시다.

어떤 쓰레드가 enqueue 오퍼레이션 중간에 preemption 된다면 이론상 이 queue의 wait-free 본성이 진행을 보장하므로 아무 문제 없습니다. 하지만 실전에서는 이 wait-free queue의 fail-stop 내성이 이 queue를 이용하는 코드에까지 확장되지 않으므로 이 처리중인 원소가 사라져버립니다.

그러나 NBS의 제한된 fail-stop 내성이 유용한 일부 어플리케이션이 있습니다. 예를 들어, 어떤 네트워크 기반 웹 어플리케이션에서 fail-stop 이벤트는 결국 재전송을 초래하는데, fail-stop 이벤트로 사라진 모든 일을 재시작하게 할 겁니다. 따라서 그런 어플리케이션을 수행하는 시스템은 상당한 부하를 받게 되는데, 스케줄러가 더이상 어떤 합리적인 공정성 보장을 하지 못할 수준까지도 이를 수 있습니다. 반대로, 어떤 쓰레드가 락을 잡은 채로 fail-stop 된다면, 이 어플리케이션은 재시작되어야 할 수도 있습니다. 그러나, NBS는 이 제한된 영역에서 조차 만병통치약이 아닌데, 순수한 스케줄링 지연으로 인한 가짜 재전송의 가능성 때문입니다. 어떤 경우에는 queueing 지연을 막기 위해 부하를 줄이는게 더 효율적일 수도 있으며, 이는 또한 스케줄러의 공정성을 제공하는 능력을 개선해서 fail-stop 이벤트를 줄이거나 제거하고, 따라서 재시도 오퍼레이션의 수를 줄여서 결국 더욱 부하를 줄이게 될 겁니다.

14.2.2.3 NBS Linearizability

Linearizability는 상당히 유용할 수 있으며, 특히 엄격한 락킹과 완전한 순서 규칙의 어토믹 오퍼레이션을² 통해 만들어진 동시성 코드를 분석할 때 특히 그렇습니다. 더 나아가, 이 완전한 순서 규칙의 어토믹 오퍼레이션의 처리는 자동적으로 간단한 NBS 알고리즘을 처리합니다.

그러나, 복잡한 NBS 알고리즘의 linearization 포인트는 종종 그 알고리즘 내에 깊이 묻혀 있으며, 따라서 그런 알고리즘의 부분을 구현하는 라이브러리 함수의 사용자에게는 보이지 않습니다. 따라서, 사용자가 복잡한 NBS 알고리즘의 linearizability 속성으로부터 혜택을 받는다는 말에는 깊은 의심을 가져야 합니다 [HKLP12].

가끔은 개발자들이 동시성 코드의 올바름 증명을 하기 위해 linearizability가 필요하다는게 단정되곤 합니다. 그러나, 그런 증명은 규칙이라기보다 예외이며, 증명을 만들어내는 현대의 개발자들은 종종 linearizability에 의존하지 않는 현대의 증명 기법들을 사용합니다. 더 나아가, 개발자들은 종종 전체 명세를 필요로 하지 않는

² 예를 들어, 리눅스 커널의 값을 반환하는 어토믹 오퍼레이션.

현대의 증명 기법들을 사용하는데, 개발자들은 종종 그 명세를 시작 이후에 배우게 되며, 한번에 한 버그씩 찾는다는 점에서 그렇습니다. 그런 증명 기법들 중 일부가 Chapter 12에서 이야기 되었습니다.³

Linearizability가 동시성 명세보다 더 자연적이라고 이야기 되는 순차적 명세와 잘 연결된다는 이야기가 종종 있습니다 [RR20]. 하지만 이 주장은 우리의 상당히 동시적인 객체 우주에서는 통하지 않습니다. 이 우주의 존재들은 동시성을 처리할 능력을 가질 것으로 기대되며, 팀 스포츠나 작은 아이들을 봐야 하는 사람들에게 특히 그렇습니다. 또한, 순차적 컴퓨팅을 가르치는 것은 여전히 어떤 흑마술로 여겨진다는 점을 생각하면 [PBCE20], 동시성 컴퓨팅을 가르치는 것은 비슷한 혼란의 상태로 예상하는게 합리적입니다. 따라서, 하나의 증명 기법에만 주목하는 것은 좋은 방향이 아닐 가능성이 높습니다.

다시 말하지만, linearizability는 많은 상황에서 매우 유용함을 이해하세요. 그리고 또 다시 말하지만, 망치라는 권위있는 도구 역시 그렇습니다. 하지만 언젠가는 망치는 내려놓고 키보드를 골라야만 하는 컴퓨팅의 분야 중 한 부분이 나옵니다. 비슷하게, linearizability가 그 일을 하는데 최고의 도구는 아닌 순간도 있을 겁니다.

Linearizability의 단점 일부를 인지하고 있는 linearizability 대변인들도 있습니다 [RR20]. Linearizability를 확장하자는 제안도 있는데, 예를 들면 interval-linearizability로, 완료되는데 0이 아닌 시간을 필요로 하는 오퍼레이션들의 혼란 경우를 처리하기 위한 것입니다 [CnRR18]. 이 제안들이 현대의 동시성 소프트웨어 작품들을 다룰 수 있는 이론이 될 것인지는 여전히 지켜봐야 하는 상태인데, Chapter 12에서 이야기한 증명 기법 여럿이 이미 많은 현대 동시성 소프트웨어 제품을 처리하고 있다는 점에서 특히 그렇습니다.

14.2.3 NBS Discussion

완전한 non-blocking queue [MS96]를 만드는 것은 가능하지만 그런 queue는 앞에서 이야기한 half-NBS 알고리즘에 비해 훨씬 더 복잡합니다. 여기서의 교훈은 여러분의 실제 요구사항을 주의깊게 고려하라는 것입니다. 불필요한 요구사항을 완화시키는 것은 종종 단순성, 성능, 그리고 확장성에 커다란 개선을 가져옵니다.

최근의 연구는 요구사항 완화를 위한 또 다른 중요한 방법을 보입니다. 공평한 스케줄링을 제공하는 시스템은 non-blocking 동기화만 제공하는 알고리즘을 수행 중일 때에도 wait-free 동기화의 이득 대부분을 가질 수

³ 어떤 linearizability의 대변인과의 기억에 남을만한 토론은 질문을 초래했습니다: “linearizability가 중요한 이유가 1980년대의 증명 기법을 구원하기 위해서란 말인가?” 그 대변인은 즉각적으로 궁정적인 답을 했고, 특정 현대 증명 기법을 험담하는데 시간을 좀 보냈습니다. 충분히 이상하게도, 그 기법은 리눅스 커널 RCU에 성공적으로 적용된 것 중 하나였습니다.

있음이 이론 [ACHS13] 과 실제 [AB13] 에서 밝혀졌습니다. 제품단계에서 사용되는 대부분의 스케줄러는 실제로 공평성을 제공하므로, wait-free 동기화를 제공하는 더 복잡한 알고리즘은 더 간단하고 더 빠른 non-wait-free 알고리즘 대비 실질적 이득을 제공하지 않는 경우가 많습니다.

흥미롭게도, fair scheduling 은 실제로는 종종 존중받는 이득적 제한이라는 것입니다. 다른 제한들은 blocking 알고리즘들이 결정적인 리얼타임 반응을 얻을 수 있게 합니다. 예를 들어, (1) 특정 우선순위 레벨에서 FIFO 순서로 얻어진 fair lock, (2) 우선순위 역전 방지 (예를 들어, 우선순위 계승 [TS95, WTS96] 또는 우선순위 상한 제한), (3) 제한된 수의 쓰레드, (4) 제한된 크리티컬 섹션 길이, (5) 제한된 읽기, 그리고 (6) fail-stop 버그의 부재를 두고 본다면 락 기반 어플리케이션은 결정적 반응 시간을 제공할 수 있습니다 [Bra11, SM04]. 이 접근법은 물론 blocking 과 wait-free 동기화 사이의 경계선을 흐리게 만드는데, 모두에게 좋은 일입니다. 희망컨대 이론적 프레임워크는 실전에서 사용되는 소프트웨어를 묘사하는 능력을 계속해서 개선할 겁니다.

이론이 우선해야 한다고 생각하는 분은 운영체제에 대해 “Theory follows practice” [Den15] 라고 한, 비길데 없이 훌륭한 Peter Denning 또는 공학 전체에 대해 “In all branches of engineering science, the engineering starts before the science; indeed, without the early products of engineering, there would be nothing for the scientist to study!” [Mor07] 라고 한 탁월한 Tony Hoare 를 참고하시기 바랍니다. 그러나, 적절한 이론이 만들어지면,⁴ 그걸 사용하는게 현명할 겁니다.

14.3 Parallel Real-Time Computing

One always has time enough if one applies it well.

Johann Wolfgang von Goethe

컴퓨팅에서 중요한 발전중인 분야는 병렬 리얼타임 컴퓨팅입니다. Section 14.3.1 는 “리얼타임 컴퓨팅” 의 여러 정의를 알아봐서 일반적인 느낌이 아닌 의미 있는 기준으로 넘어갑니다. Section 14.3.2 는 리얼타임 응답이 필요한 어플리케이션의 종류를 알아봅니다. Section 14.3.3 는 병렬 리얼타임 컴퓨팅이 우리 곁에 있음을 알리고 언제 그리고 왜 병렬 리얼타임 컴퓨팅이 유용한지 이야기 합니다. Section 14.3.4 는 병렬 리얼타임 시스템이 어떻게 구현될 수 있는지 간단한 개론을 제공하며, Sections 14.3.5 and 14.3.6 는 운영체제와 어플리케이-

션에 각각 집중합니다. 마지막으로, Section 14.3.7 는 여러분의 어플리케이션이 리얼타임 기능을 필요로 하는지를 어떻게 결정하는지 설명합니다.

14.3.1 What is Real-Time Computing?

리얼타임 컴퓨팅을 분류하는 한가지 전통적 방법은 하드 리얼타임과 소프트 리얼타임으로 카테고리를 나누는 것으로, 하드 리얼타임 어플리케이션은 기한을 결코 어기지 않지만 소프트 리얼타임 어플리케이션은 꽤 자주 기한을 어길 수 있습니다.

14.3.1.1 Soft Real Time

이 소프트 리얼타임의 정의에서의 문제를 찾기는 쉽습니다. 일단, 이 정의에 의하면 모든 소프트웨어 조각은 소프트 리얼타임 어플리케이션이라고 볼릴 수 있습니다: “나의 어플리케이션은 백만개 지점의 Fourier transform 을 0.5 피코세컨드만에 계산할 수 있습니다.” “말도 안돼!!! 이 시스템의 클락 사이클은 3000 피코세컨드가 넘어요!” “아, 하지만 그건 소프트 리얼타임 어플리케이션이에요!” “소프트 리얼타임” 이라는 용어가 어떤 식으로든 쓰인다면, 어떤 한계가 분명 필요합니다.

따라서 우리는 특정 소프트 리얼타임 어플리케이션은 최소 특정 확률로는 그 응답 시간 요구사항을 지켜야 한다고 말할 수도 있겠는데, 예를 들어 99.9 % 시간으로는 20 마이크로세컨드 내에 실행되어야만 한다는 식입니다.

이는 어플리케이션이 응답 시간 요구사항을 지키지 못했을 때 뭘 해야 하는지에 대한 질문 역시 던지게 합니다. 그에 대한 답은 어플리케이션마다 다르겠지만, 한가지 가능성은 제어되는 시스템이 가끔의 늦은 제어 행동을 문제되지 않게 할 충분한 안정성과 관성을 갖고 있는 겁니다. 또 다른 가능성은 어플리케이션이 결과를 계산하는 두 가지 방법을 가지고 있는데 빠르고 결정적이지만 부정확한 결과를 내는 것과 예상 불가한 시간을 필요로 하지만 아주 정확한 방법의 두 가지입니다. 한 가지 합리적인 방법은 두 가지 방법을 병렬로 시작하고 정확한 방법이 시간 내에 계산을 마무리 하지 못하면 그걸 종료시키고 빠르지만 부정확한 방법의 결과를 사용하는 겁니다. 빠르지만 부정확한 방법의 한 후보는 현재 시간 기간 동안에는 제어 행동을 취하지 않는 것이고 또 다른 후보는 앞 시간 기간 동안 취해진 제어 행동을 똑같이 취하는 겁니다.

요약하자면, 정확히 얼마나 그게 소프트 한지 없이 소프트 리얼타임을 말하는 건 말이 안됩니다.

⁴ 이론의 첫번째 적절한 몸체는 종종 첫번째 제안된 이론의 몸체와 별개임도 유의하시기 바랍니다.



Figure 14.1: Real-Time Response, Meet Hammer

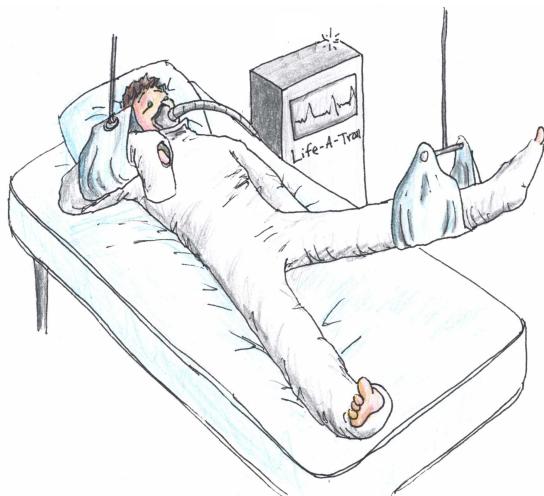


Figure 14.2: Real-Time Response: Hardware Matters

14.3.1.2 Hard Real Time

대조적으로, 하드 리얼타임의 정의는 상당히 결정적입니다. 어쨌건, 특정 시스템은 제한시간을 항상 지키거나 그러지 못하거나 합니다.

불행히도, 이 정의의 엄격한 적용은 세상에 어떤 하드 리얼타임 시스템도 존재하지 못할 것을 의미합니다. 이에 대한 이유가 Figure 14.1에 재미있게 묘사되어 있습니다. 그래요, 여러분은 튼튼한 시스템을 만들 수 있는데, 중복되는 부분도 있을 수 있습니다. 하지만 여러분의 적은 언제나 더 큰 망치를 가질 수 있습니다.

그리고 또다시, 명백히 단순한 하드웨어 문제일 뿐 아니라 아주 큰 하드웨어 문제인 상황에서 소프트웨어

를 탓하는 건 공평치 않을 수 있습니다.⁵ 이는 우리가 하드 리얼타임 소프트웨어를 항상 그 기한을 지킬 소프트웨어가 아니라 하드웨어 문제가 없을 때에만 지키는 것으로 정의할 것을 제안합니다. 불행히도, 실패는 항상 선택지인 것은 아닌데, Figure 14.2에 상황이 묘사되어 있습니다. 우린 이 그림에 그려진 불쌍한 신사가 우리의 “당신의 비극적 죽음을 이르게 하는 기한 어김이 일어난다면 그건 소프트웨어 문제 때문은 분명 아닐테니 안심하고 쉬세요!”라고 말하는 걸로 안심할 거라고 단순히 예상할 수 없습니다. 하드 리얼타임 응답은 전체 시스템의 속성이지 소프트웨어만의 것이 아닙니다.

하지만 우리가 완벽을 목표로 할 수 없다면, 알림을 줄 수 있는데, 앞서 언급된 소프트 리얼타임에서의 허아법과 비슷한 겁니다. 그러면 Figure 14.2의 Life-a-Tron이 기한을 어길 거 같다면 병원 직원에게 경고를 보낼 수 있습니다.

불행히도, 이 방법은 Figure 14.3에 그려진 사소한 문제가 있습니다. 항상 즉각적으로 기한을 지키지 못할 것을 알리는 시스템은 글로 쓰여진 법을 준수하지만 전혀 쓸모없습니다. 시스템이 특정 분량의 시간 동안은 기한을 지킬 것에 대한, 또는 특정 횟수 이상 반복적으로 기한을 어기지 않을 것에 대한 요구사항이 분명히 존재합니다.

우린 그냥 그럴싸하게 들리는 방법을 하드 리얼타임에도 소프트 리얼타임에도 취할 수 없음이 분명합니다. 따라서 다음 섹션은 좀 더 실제 세계의 방법을 취합니다.

14.3.1.3 Real-World Real Time

“하드 리얼타임 시스템은 항상 기한을 지킵니다!” 같은 문장은 분명 눈에 띠고 기억하기 쉽지만, 실제 세계 리얼타임 시스템을 위해선 다른 무언가가 필요합니다. 그 결과 나오는 명세는 기억하기 더 어렵지만, 환경, 워크로드, 그리고 그 리얼타임 어플리케이션 자체에 대한 한계를 내포함으로써 리얼타임 시스템의 구성을 간단화 시킬 수 있습니다.

Environmental Constraints 환경상의 제약은 “하드 리얼타임”에 의해 암시되는 응답 시간에 대한 열린 결말의 약속에 대한 반대를 처리합니다. 이 제약은 허용되는 운영 온도, 공기 질, 전자기파의 수준과 종료, 그리고 Figure 14.1에서 짚었듯, 충격과 진동의 수준을 명시할 수도 있습니다.

물론, 어떤 제약들은 다른 것들보다 지키기 쉽습니다. 얼마든지 되는 사람들이 상용 컴퓨터 부품들이 종종 얼듯한 기온에서 동작하지 않음을 배웠는데, 이는 기후 제어 요구사항들을 제시합니다.

⁵ 또는, 현대의 망치를 놓고 보면, 큰 강철 문제입니다.



Figure 14.3: Real-Time Response: Notification Insufficient

예전 어떤 동료는 한번 어떤 리얼타임 시스템을 강한 염소 화합물을 갖는 대기에서 운영하는 문제를 겪었습니다. 그는 지혜롭게 이를 그 하드웨어를 설계하는 동료에게 넘겼습니다. 그 결과, 제 동료는 컴퓨터를 둘러싸는 대기상의 화합물에 대한 환경 제약을 넣었으며 하드웨어 설계자들은 물리적 봉인을 통해 이를 지켰습니다.

또 다른 오랜 대학 친구는 컴퓨터로 티타늄 잉곳을 전공 상태에서 산업용 강도의 아크를 사용하여 뺏어내는, 컴퓨터 제어 시스템을 개발했습니다. 때때로, 이 아크는 티타늄 잉곳을 뺏어내는 경로에 따분해지고 훨씬 짧고 재미있는 땅으로 향하는 경로를 선택했습니다. 우리가 물리 수업에서 모두 배웠듯, 전자의 흐름에서의 급작스런 변화는 전자기파를 만들어내며, 더 큰 흐름에서의 더 큰 변화에는 더 강한 전자기파를 만들어냅니다. 그리고 이 경우, 그로 인한 전자기파 파동은 400미터 떨어진 위치의 wkrdms “고무 오리” 안테나의 리드에 4분의 1볼트 차이를 만들어내기 충분했습니다. 이는 inverse-square 법칙에 의해 근처의 컨덕터가 더 높은 전압을 받게 되었다는 것을 의미합니다. 이는 이 뺏어내기 과정을 제어하는 컴퓨터를 만드는 컨덕터도 포함되었습니다. 특히, 그 컴퓨터의 리셋 선에 가해진 전압은 정말로 그 컴퓨터를 리셋시키기에 충분했어서 모두를 어리둥절하게 했습니다. 이 상황은 정교한 격리장치와 제가 지금껏 들어본 가장 낮은 bitrate 인 9600 baud 의 광섬유 네트워크를 포함한 하드웨어를 사용해 해결되었습니다. 덜 극적인 전자기파 환경은 오류 탐지와 수정 코드의 사용을 통해 소프트웨어 단에서 해결될 수 있는 경우가 많습니다. 그러나, 오류 탐지와 수정 코드가 실패율을 줄일 수는 있으나 그를 없앨 수는 없음을, 따라서 하드 리얼타임

응답을 얻는데에 대한 또 다른 장애가 됨을 기억하는게 중요합니다.

최소한의 수준의 에너지가 필요한 상황들도 있는데, 예를 들어 시스템의 전원 리드에서 기기를 거쳐 모니터되거나 제어되어야 하는 시스템 바깥 부분과 통신을 하는 부분들입니다.

Quick Quiz 14.2: 하지만 배터리 기반 시스템은 어떤가요? 그것들은 시스템 내로 흘러들어오는 전원을 필요로 하지 않잖아요.

여러 시스템이 상당한 수준의 충격과 진동을 갖는 환경에서 동작하게끔 만들어지는데, 예를 들어 엔진 제어 시스템이 있습니다. 우리가 지속적인 진동에서 간헐적인 충격으로 넘어가면 보다 격렬한 요구사항이 생깁니다. 예를 들어, 제가 학부생이던 시절, 저는 오래된 Athena ballistics 컴퓨터를 마주했는데, 이는 수류탄이 근처에서 폭발하더라도 평범하게 동작할 수 있도록 설계되었습니다.⁶ 그리고 마지막으로, 비행기에서 사용되는 “블랙 박스”는 사고 이전에도, 이후에도 동작을 계속해야만 합니다.

물론, 환경상의 충격과 공격에 대해 더 튼튼하게 하드웨어를 만들 수 있습니다. 수많은 독창적인 기계적 충격 흡수 장비가 충격과 진동의 영향을 줄일 수 있으며, 여러 겹의 보호 장비가 저전력 전자기파 방사선 영향을 줄일 수 있으며, 오류 수정 코딩이 고전력 방사선 영향을 줄일 수 있고, 다양한 도기류 사용과 봉인 기술이 공기 질의

⁶ 수십년이 지나, 어떤 종류의 컴퓨터 시스템을 위한 수락 테스트는 큰 폭발을 포함하게 되었으며, 어떤 종류의 통신 네트워크는 “ballistic jamming”이라 불리는 것을 처리할 수 있어야만 하게 되었습니다.

영향을 줄일 수 있고, 수많은 가열과 냉방 시스템이 온도 영향을 처리할 수 있습니다. 극단적인 경우에는 세겹의 모듈 기반 중복이 시스템의 한 부분에서의 문제가 전체 시스템의 부정확한 동작을 초래하는 가능성을 줄일 수 있습니다. 그러나, 이 모든 방버들이 한가지 공통적인 것을 갖습니다: 그것들이 실패 확률을 줄일 수는 있으나 없앨 수는 없습니다.

이런 환경적 문제는 튼튼한 하드웨어를 통해 해결됩니다만 다음 두 섹션에서의 워크로드와 어플리케이션 제약은 종종 소프트웨어로 처리됩니다.

Workload Constraints 사람들과 마찬가지로 부하를 지나치게 줌으로써 리얼타임 시스템이 기한을 지키지 못하게 하는게 종종 가능합니다. 예를 들어, 시스템이 너무 자주 인터럽트 된다면, 리얼타임 어플리케이션을 처리하기에 CPU 대역폭이 부족할 수 있습니다. 이 문제에 대한 하드웨어 해결책은 인터럽트가 시스템에 전달되는 비율에 제한을 가하는 것입니다. 가능한 소프트웨어 해결책은 인터럽트가 너무 자주 끄르어오면 잠깐 인터럽트를 꺼버리거나, 너무 빈번한 인터럽트를 발생시키는 기기를 리셋하거나, 인터럽트를 아예 막고 polling 을 하는 방법 등이 있습니다.

과부하는 또한 queueing 효과로 인해 응답 시간을 악화시켜서 리얼타임 시스템이 CPU 대역폭을 지나치게 사용하게 하지 않는게 일반적이므로 동작 중인 시스템은 (말하자면) 80 % idle 시간을 갖게 합니다. 이 방법은 저장장치와 네트워크 장치에도 적용됩니다. 어떤 경우, 별도의 저장장치와 네트워크 장치는 리얼타임 어플리케이션의 높은 우선순위 부분에의 전용을 위해 예약될 수도 있습니다. 요약하자면, 리얼타임 시스템에서는 처리량보다 응답시간이 더 중요하므로 하드웨어가 거의 idle 상태가 되는건 드문 일이 아닙니다.

Quick Quiz 14.3: 하지만 queueing 이론에서의 결론을 놓고 보면, 낮은 사용율은 최악의 경우 응답시간을 개선하기보다는 평균 응답시간을 개선할 뿐이지 않나요? 그리고 최악의 경우 응답시간이 리얼타임 시스템에서 정말 신경쓰는 것이지 않나요?

■

물론, 충분히 낮은 사용율을 유지하는 것은 설계와 구현 전반에 상당한 주의를 필요로 합니다. 기한을 어기게 만드는 작은 기능상의 문제 같은 건 없습니다.

Application Constraints 어떤 오퍼레이션에는 다른 것들보다 제한된 응답시간을 제공하기가 더 쉽습니다. 예를 들어, 인터럽트와 wake-up 오퍼레이션에 대해서는 응답 시간 명세를 보기엔 상당히 흔하지만 (예를 들어) 파일시스템 unmount 오퍼레이션에 대해서는 드문 일입니다. 그 이유 중 하나는 파일시스템 unmount 오퍼레이션은 해당 파일시스템의 메모리 내에 있는 모든

데이터를 대용량 저장장치로 비워내야 한다는 점에서 오퍼레이션이 해야 할 수도 있는 일들의 양을 제한하기가 무척 어렵다는 겁니다.

이는 리얼타임 어플리케이션이 제한된 응답시간이 합리적으로 제공될 수 있는 오퍼레이션들로 국한되어야 함을 의미합니다. 다른 오퍼레이션은 어플리케이션의 비 리얼타임 부분이 되거나 아예 없어져야 합니다.

어플리케이션의 비 리얼타임 부분에 대한 제약도 있을 수 있습니다. 예를 들어, 비 리얼타임 어플리케이션은 리얼타임 부분을 위한 CPU를 사용해도 될까요? 어플리케이션의 리얼타임 부분이 일반적이지 않게 바쁘게 일할 것으로 예상되는 시간이 있고, 그런 경우에 어플리케이션의 비 리얼타임 부분은 그 시간대에 수행되는 게 허용될까요? 마지막으로, 어플리케이션의 리얼타임 부분은 비 리얼타임 부분의 처리량을 얼마나 악화시켜도 괜찮을까요?

Real-World Real-Time Specifications 앞의 섹션들에서 볼 수 있었듯, 실제 세계의 리얼타임 명세서는 환경, 워크로드, 그리고 어플리케이션 자체에 대한 제약을 포함해야 합니다. 또한, 어플리케이션의 리얼타임 부분이 사용할 수 있는 오퍼레이션들에 대해서는 그 오퍼레이션을 구현하는 하드웨어와 소프트웨어에 대한 제약이 있어야만 합니다.

그런 오퍼레이션 각각에 대해, 이 제약들은 최대 응답 시간 (그리고 최소 응답 시간도)과 그 응답 시간을 지키는 확률을 포함할 수도 있습니다. 100 % 확률은 연관된 오퍼레이션이 하드 리얼타임 서비스를 제공해야만 함을 의미합니다.

어떤 경우, 응답시간과 필요한 그걸 지킬 확률은 문제의 오퍼레이션에의 인자들에 따라 다양해질 수 있습니다. 예를 들어, 지역 LAN을 통한 네트워크 오퍼레이션은 (말하자면) 100 마이크로세컨드 내에 완료될 확률이 대륙별 WAN을 통한 같은 네트워크 오퍼레이션보다 훨씬 높을 겁니다. 더 나아가, 구리나 광섬유 LAN을 통한 네트워크 오퍼레이션은 시간이 걸리는 재전송 없이 완료될 확률이 극단적으로 높을 것인 반면, 신호가 쉽게 놓쳐지는 WiFi 네트워크를 통한 동일한 네트워크 오퍼레이션은 짧은 기한을 놓칠 확률이 훨씬 높을 겁니다.⁷ 비슷하게, 춤출히 결합된 SSD로부터의 읽기는 구식의 USB로 연결된 회전 기반 녹슨 디스크 드라이브에서의 같은 읽기보다 훨씬 빠르게 완료될 거라 예상될 수 있을 겁니다.

어떤 리얼타임 어플리케이션은 오퍼레이션의 다른 단계들을 통과합니다. 예를 들어, 회전하는 통나무에서 나무의 얇은 껍데기 (“베니어 (veneer)”라고 불립니다)

⁷ 중요한 안전 요령: USB 기기로부터의 최악의 경우 응답 시간은 극단적으로 길 수 있습니다. 따라서 리얼타임 시스템은 중요 경로로부터 USB 기기를 떼어놓게끔 주의를 기울여야 합니다.

를 벗겨내는 합판 선반을 제어하는 리얼타임 시스템은: (1) 통나무를 선반에 올리고, (2) 이 통나무를 이 선반의 고정판에 위치시켜서 이 통나무의 가장 큰 원통을 포함한 부분이 칼날에 노출되게 하고, (3) 이 통나무를 돌리기 시작하고, (4) 이 통나무로부터 베니어를 벗겨내게끔 칼의 위치를 지속적으로 변화시키고, (5) 껍질을 벗겨내기엔 너무 작은 통나무의 나머지 중심 부위를 제거하고, (6) 다음 통나무를 기다립니다. 이 오퍼레이션의 여섯 단계는 각자의 기한과 환경 제약이 있을 수도 있는데, 예를 들어 단계 4의 기한은 기한 6의 것보다 훨씬 짧아서 초단위보다는 밀리세컨드 단위일 거라 예상할 수 있습니다. 따라서 어떤 사람들은 낮은 우선순위 작업은 단계 4 보다는 단계 6에서 수행될 거라 예상할 수 있겠습니다. 어느 경우든, 단계 4의 보다 가혹한 요구 사항을 지원하기 위해 하드웨어, 드라이버, 소프트웨어 구성의 주의 깊은 선택이 필요할 겁니다.

이 단계별 접근 방법의 핵심 장점은 응답시간 예산이 조개질 수 있어서 어플리케이션의 다양한 부분들이 각자의 응답시간 예산을 가지고 개별적으로 개발될 수 있다는 겁니다. 물론, 모든 다른 예산들과 마찬가지로, 어떤 부분이 전체 예산 중 얼만큼을 갖게 되는가와 같은 때때로의 충돌이 있을 수 있고, 공유 목표에 대한 감지와 강한 리더쉽이 이런 충돌을 빠르게 해결하는데 도움이 될 수 있습니다. 그리고, 역시 다른 기술적 예산들과 마찬가지로, 응답시간에 대한 올바른 집중과 응답 시간 문제에 대한 이른 경고를 보장하기 위해 강한 검증 노력이 필요합니다. 성공적인 검증 노력은 거의 항상 좋은 테스트 집합을 포함하는데, 이론가들에게는 불만족스러울 수도 있지만 일이 되게끔 돋는 미덕을 갖습니다. 2021년 초기준으로, 대부분의 실제 세계의 리얼타임 시스템은 정형적 증명보다는 수용 테스트를 사용합니다.

그러나, 리얼타임 시스템을 검증하기 위한 테스트 집합의 많은 사용은 매우 실질적인 단점을 갖는데, 리얼타임 소프트웨어가 하드웨어와 소프트웨어의 특정 구성에 대해서만 검증된다는 겁니다. 추가적인 구성을 도하는 것은 추가적인 비싸고 시간을 소모하는 테스트를 요합니다. 아마도 정형 검증 분야는 이 상황을 바꾸기 충분하게끔 발전할 겁니다만, 2021년 초기준으로는 큰 발전이 필요합니다.

Quick Quiz 14.4: 수십년간의 치열한 연구 덕에 정형 검증은 이미 상당히 가능합니다. 추가적인 발전이 정말로 필요한가요, 아니면 이는 그저 정형 검증의 위대한 힘을 게으르게 무시하기를 계속하려는 어떤 실제 일을 하는 사람의 변명인가요?

■
어플리케이션의 리얼타임 부분에 대한 응답시간 요구 사항에 더해, 어플리케이션의 비 리얼타임 부분에 대한 성능과 확장성 요구 사항도 있을 수 있습니다. 이 추가적인 요구 사항들은 궁극의 리얼타임 응답시간이 종종

확장성과 평균 성능을 악화시킴으로써 얻어진다는 사실을 반영합니다.

소프트웨어 엔지니어링 요구 사항 역시 중요할 수 있는데, 특히 큰 팀에 의해 개발되고 유지되어야만 하는 큰 어플리케이션에서 특히 그렇습니다. 이 요구 사항들은 종종 더 높은 모듈화와 실패 격리를 선호합니다.

이는 제품 단계의 리얼타임 시스템을 위한 환경 제약과 응답 기한을 명세하는데 필요할 것들에 대한 단순한 개론입니다. 이 단순한 개론이 리얼타임 컴퓨팅에의 그럴싸하게 들리기만 하는 접근법의 부적당함을 잘 보였길 바랍니다.

14.3.2 Who Needs Real-Time?

모든 컴퓨팅은 사실 리얼타임 컴퓨팅이라고 주장할 수도 있습니다. 한 가지 예를 들어보자면, 여러분이 온라인으로 생일 선물을 구매한다면, 여러분은 그 선물을 그걸 받는 사람의 생일 전에도 착할 것을 예상할 겁니다. 그리고 실제로 21세기에 접어드는 시점의 웹 서비스는 1초 미만의 응답시간 제약을 확인했으며 [Boh01], 요구 사항은 시간이 흐름에 따라 완화되지 않았습니다 [DHJ⁺07]. 그러나 이는 응답시간 요구 사항이 비 리얼타임 시스템과 어플리케이션에 의해 간단히 얻어질 수 없는 리얼타임 어플리케이션에 집중하는데 유용합니다. 물론, 하드웨어 비용이 낮아지고 대역폭과 메모리 크기가 늘어남에 따라, 리얼타임과 비 리얼타임 사이의 경계는 계속 움직일 겁니다만, 그런 움직임은 결코 나쁜 일이 아닙니다.

Quick Quiz 14.5: 무엇이 “비 리얼타임 시스템과 어플리케이션에 의해 간단히 얻어질 수 있는가”에 따라 리얼타임을 비 리얼타임과 차별하는 건 우스운 일입니다! 그런 차이에 대한 이론적 기반은 분명 존재하지 않습니다!!! 더 나은 방법은 없을까요???

■
리얼타임 컴퓨팅은 제조부터 항공전자공학, 과학 어플리케이션, 아마도 가장 극적으로는 별빛의 깜빡임을 제거하기 위한 거대한 지구 범위 망원경에 사용되는 빛 제어 시스템, 앞서 이야기한 항공전자공학을 포함한 군대 어플리케이션, 그리고 처음으로 기회를 포착하는 컴퓨터가 대부분의 수익을 얻을 투자 서비스 어플리케이션까지 다양한 산업 제어 어플리케이션에서 사용됩니다. 이는 “제품 생산”, “삶”, “죽음”, 그리고 “돈”의 네 개 영역으로 카테고리화 될 수 있습니다.

투자 서비스 어플리케이션은 다른 세개의 카테고리로부터 돈은 물질이 아니라는 점에서 약간 차별화 되는데, 이는 비 컴퓨팅 응답이 상당히 작음을 의미합니다. 대조적으로, 다른 세개의 카테고리에서 피할 수 없는 기계적 지연은 어플리케이션의 리얼타임 응답에서 지연을 줄이는 것이 제공하는 이익은 적거나 아예 없음을 의미합니다. 이는 리얼타임 정보 처리 어플리케이션들 가

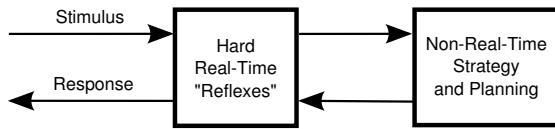


Figure 14.4: Real-Time Reflexes

운데 투자 서비스가 더 낮은 응답시간의 어플리케이션을 보통 이기는 경쟁을 직면함을 의미합니다. 초래되는 응답시간 요구사항은 여전히 279 페이지의 “Real-World Real-Time Specifications” 문단에 명시한 것과 같을 수 있지만, 이 요구사항에 대한 일반적이지 않은 본성은 투자와 정보 처리 어플리케이션을 “리얼타임”이라기보다는 “낮은 응답시간”으로 표현하게 했습니다.

우리가 그걸 뭐라 부르든, 리얼타임 컴퓨팅에 대한 상당한 필요가 존재합니다 [Pet06, Inm07].

14.3.3 Who Needs Parallel Real-Time?

누가 병렬 리얼타임 컴퓨팅을 필요로 하는가는 덜 명확합니다만 저비용 멀티코어 시스템의 발전은 상관없이 이를 전면에 드러냈습니다. 불행히도, 전통적인 리얼타임 컴퓨팅의 수학 기반은 약간의 규칙 증명을 위한 예외와 함께 단일 CPU 시스템을 가정합니다 [Bra11]. 다행히도, 현대의 컴퓨팅 하드웨어를 리얼타임 수학에 맞추는 몇 가지 방법이 존재하며, 일부 리눅스 커널 해커들은 학계가 이 전환을 만들게끔 장려했습니다 [dOCdO19, Gle10].

한 가지 방법은 많은 링리얼타임 시스템이 생체 신경 시스템을 닮아서 응답시간은 Figure 14.4에 그려진 것처럼 리얼타임 반응부터 비 리얼타임 전략화와 계획까지 다양하다는 사실을 인식하는 겁니다. 센서로부터 값을 읽거나 구동기를 제어하는 것 같은 하드 리얼타임 반응은 단일 CPU나 FPGA 같은 특수 목적 하드웨어에서 실시간으로 동작합니다. 어플리케이션의 비 리얼타임 전략 짜기나 계획 짜기 부분은 나머지 CPU에서 수행됩니다. 전략짜기와 계획 짜기는 통계적 분석, 주기적 측정, 사용자 인터페이스, 공급망 활동, 그리고 준비를 포함할 수도 있습니다. 높은 컴퓨팅 부하의 준비 활동을 예로 들어보면, 279 페이지의 “Real-World Real-Time Specifications” 문단에서 이야기한 껍질 벗기기 어플리케이션을 다시 생각해 보십시오. 하나의 CPU가 하나의 통나무를 벗기는데 필요한 높은 속력의 리얼타임 연산을 하는 동안, 나머지 CPU들은 고품질 나무의 가장 큰 원통을 얻기 위해 다음 통나무를 어떻게 위치시킬지 결정하기 위해 다음 통나무의 크기와 모양을 분석하고 있을 수도 있습니다. 많은 어플리케이션은 비 리얼타임과 리얼타임 컴포넌트를 가지고 있는 것으로 드러났으므로 [BMP08], 이 방법은 전통적 리얼타임 분석이 현대의

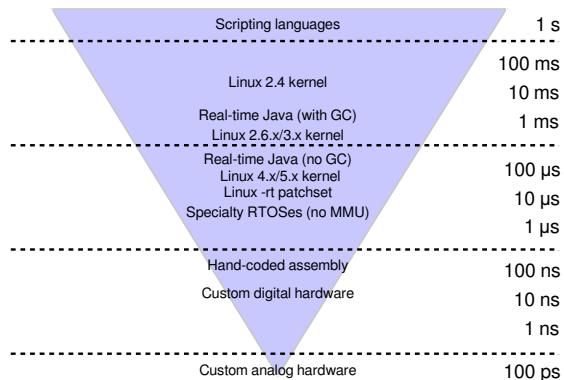


Figure 14.5: Real-Time Response Regimes

멀티코어 하드웨어와 결합되게 하는데 종종 사용될 수 있습니다.

간단한 방법은 하나의 하드웨어 쓰레드만 남겨둬서 이미 정착된 단일 프로세서 리얼타임 컴퓨팅의 수학으로 되돌아가는 겁니다. 그러나, 이 방법은 잠재적 비용과 전력 효율성 이득을 포기합니다. 그러나, 이 장점을 얻기 위해서는 Chapter 3에서 다른 병렬 성능 장애들을 극복하는 것이 필요하며 평균만이 아니라 최악의 경우를 대비해야 합니다.

따라서 병렬 리얼타임 시스템을 구현하는 것은 상당히 어려울 수 있습니다. 이를 넘어서기 위한 여러 방법들이 다음 섹션에 소개됩니다.

14.3.4 Implementing Parallel Real-Time Systems

우리는 두개의 주요 리얼타임 시스템, 즉 event-driven과 polling을 살펴볼 겁니다. Event-driven 리얼타임 시스템은 대부분의 시간을 idle 상태로 지내며 운영체제에 의해 어플리케이션에 전달된 이벤트에 리얼타임으로 반응합니다. 대안적으로, 이 시스템은 비 리얼타임 워크로드를 뒤에서 수행할 수 있습니다. Polling 리얼타임 시스템은 CPU에 제한되어 있고 입력을 읽어오고 출력을 업데이트하는 짧은 반복문을 갖는 리얼타임 쓰레드를 가집니다. 이 짧은 polling 반복은 종종 완전히 사용자 모드에서 동작해서 어플리케이션의 user-mode 주소 공간에 매핑된 하드웨어 레지스터를 읽고 쓸 수 있습니다. 대안적으로, 어떤 어플리케이션들은 이 polling 반복을 커널에 위치시키는데 예를 들면 load 할 수 있는 커널 모듈을 사용하는 겁니다.

선택된 스타일과 관계 없이, 리얼타임 시스템을 구현하는데 사용되는 방법은 기한에 종속적일텐데, 예를 들면 Figure 14.5에 보인 것과 같습니다. 이 그림의 꼭대기로부터 보면, 여러분이 1초가 넘는 응답시간에도 문제

없다면 여러분의 리얼타임 어플리케이션을 구현하는데 스크립트 언어를 사용할 수도 있을 것을 겁니다—그리고 스크립트 언어는 놀랍도록 자주 사용됩니다만, 제가 이를 꼭 추천해야 할 이유는 아닙니다. 요구되는 응답시간이 수십 밀리세컨드를 넘는다면 오래된 2.4 버전의 리눅스 커널이 사용될 수 있는데, 역시 제가 이를 꼭 추천해야 할 이유는 아닙니다. 특수한 리얼타임 Java 구현은 *garbage collector*를 사용하면서도 수 밀리세컨드의 리얼타임 응답시간을 제공할 수 있습니다. 리눅스 2.6.x 와 3.x 커널은 주의 깊게 구성되고 튜닝되고 리얼타임 친화적 하드웨어에서 수행된다면 수백 마이크로세컨드의 리얼타임 응답시간을 제공할 수 있습니다. 특수 리얼타임 Java 구현은 *garbage collector*의 사용이 주의 깊게 막아진다면 100 마이크로세컨드 미만의 리얼타임 응답시간을 제공할 수 있습니다. (하지만 *garbage collector*의 사용을 제한함은 Java의 거대한 표준 라이브러리 사용을 막아서 Java의 생산성이 이득을 막음을 알아두십시오.) 리눅스 4.x 와 5.x 커널은 100 밀리세컨드 미만의 응답시간을 제공합니다만 2.6.x 와 3.x 커널과 같은 주의가 필요합니다. -rt 패치셋을 포함한 리눅스 커널은 20 마이크로세컨드 미만의 응답시간을 제공할 수 있으며, MMU 없이 동작하는 특수 리얼타임 운영체제들은 (RTOS들) 10 마이크로세컨드 미만의 응답시간을 제공할 수 있습니다. 마이크로세컨드 미만 응답시간을 얻는데에는 손으로 짜여진 어셈블리나 심지어 특수 목적 하드웨어가 필요합니다.

물론, 스택 전반에 대한 주의 깊은 구성과 튜닝이 필요합니다. 특히, 하드웨어나 펌웨어가 리얼타임 응답시간을 제공하지 못한다면 소프트웨어가 잃어버린 시간을 위해 할 수 있는 건 없습니다. 더 나쁜 것이, 고성능 하드웨어는 가끔 더 좋은 처리량을 위해 최악의 경우 행동을 희생합니다. 실제로, 인터럽트가 불능화된 상태에서 도는 짧은 반복문에서의 타이밍은 고품질 난수 생성기의 기본을 제공할 수 있습니다 [MOZ09]. 더 나아가서, 일부 펌웨어는 다양한 관리 업무를 위해 사이클을 훔치기도 하는데 어떤 경우에는 희생자 CPU의 하드웨어 클락을 재 프로그래밍 시켜서 그 기록을 지우기까지 합니다. 물론, 사이클 훔치기는 가상화 환경에서는 예상된 행동입니다만 사람들은 가상화 환경에서의 리얼타임 응답을 위해 작업하고 있습니다 [Gle12, Kis14]. 따라서 여러분의 하드웨어와 펌웨어의 리얼타임 기능을 평가하는게 무척 중요합니다.

하지만 좋은 리얼타임 하드웨어와 펌웨어가 있다면 스택의 다음 단계는 운영체제로, 다음 섹션에서 이를 다룹니다.

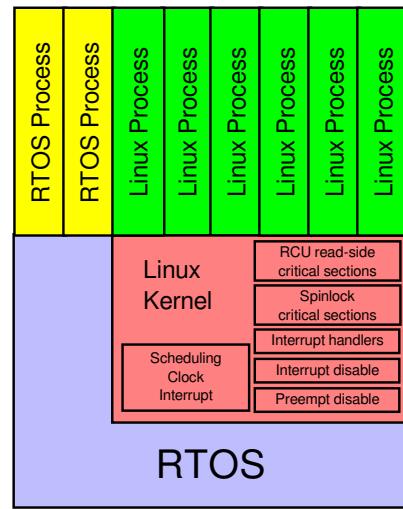


Figure 14.6: Linux Ported to RTOS

14.3.5 Implementing Parallel Real-Time Operating Systems

리얼타임 시스템을 구현하기 위한 여러 전략이 있습니다. 한 가지 방법은 범용 비 리얼타임 OS를 특수 목적 리얼타임 운영체제 (RTOS) 위에 포팅하는 것으로, Figure 14.6에 보여진 것과 같습니다. 초록색 “Linux Process” 박스들은 리눅스 커널에서 돌아가는 비 리얼타임 프로세스를 나타내며, 노란 “RTOS Process” 박스들은 RTOS에서 돌아가는 리얼타임 프로세스들을 나타냅니다.

이는 리눅스 커널이 리얼타임 기능을 얻기 전까지 매우 대중적인 전략이었으며, 여전히 사용됩니다 [xen14, Yod04b]. 그러나, 이 방법은 어플리케이션이 RTOS에서 돌아가는 부분과 리눅스에서 돌아가는 부분으로 분리될 것을 필요로 합니다. 예를 들어 RTOS로부터의 POSIX 시스템콜을 리눅스에서 도는 관리 쓰레드로 넘기는 것과 같이 두 환경을 비슷해 보이게 하는 것은 가능하나, 언제나 깔끔하게 떨어지지 않는 부분이 있습니다.

또한, RTOS는 하드웨어와 리눅스 커널 둘 다와 상호 작용해야 해서, 하드웨어와 커널 양쪽에의 변경에 대한 상당한 관리를 요합니다. 더 나아가, 그런 RTOS 각각은 종종 자신만의 시스템콜 인터페이스와 시스템 라이브러리 집합을 가져서, 에코시스템과 개발자들을 분열시킬 수 있습니다. 실제로, 이 문제들은 RTOS와 리눅스의 결합을 이끈 것으로 보여지는데, 이 방법은 RTOS의 완전한 리얼타임 능력에의 접근을 허용하는 한편 어플리케이션의 비 리얼타임 코드는 리눅스의 오픈소스 에코시스템에 완전한 접근을 허용했기 때문입니다.

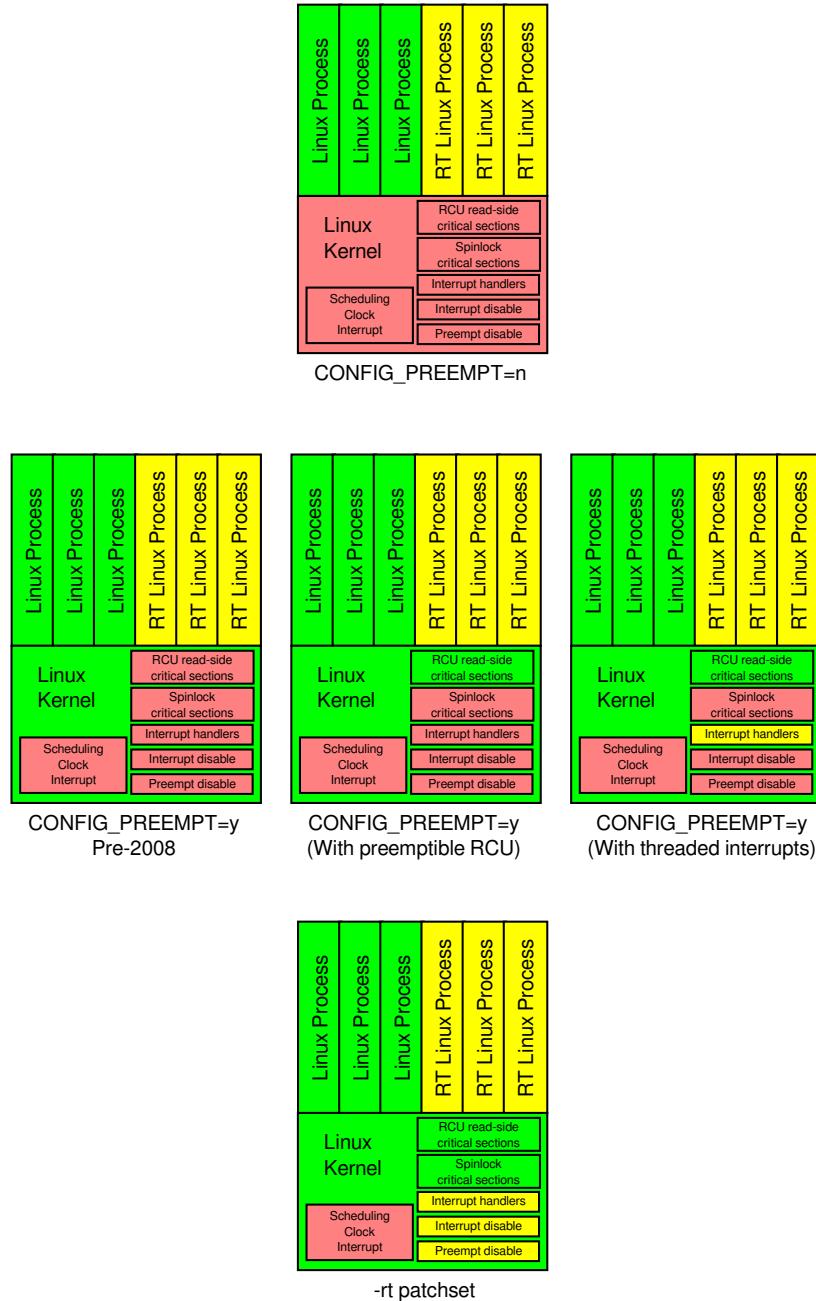


Figure 14.7: Linux-Kernel Real-Time Implementations

RTOS 와 리눅스 커널을 결합시키는 건 리눅스 커널이 최소한의 리얼타임 기능만을 가지고 있던 때에는 영리하고 유용한 단기적 방법이었습니다만, 이는 또한 리눅스 커널에 리얼타임 기능을 추가하는 동기가 되었습니다. 이 목표를 향한 진행이 Figure 14.7에 보여 있습니다. 왼쪽은 preemption이 불능화된, 따라서 리얼타임 기능을 갖지 못한 리눅스 커널을 보입니다. 가운데 열은 preemption이 허용된 상태에서의 메인라인 리눅스 커널의 리얼타임 기능이 늘어나는 과정을 보입니다. 마지막으로, 오른쪽은 -rt 패치셋이 적용되어 리얼타임 기능이 최대화된 상태의 리눅스 커널을 보입니다. -rt 패치셋으로부터의 기능이 메인라인에 더해졌으므로, 메인라인 리눅스 커널의 기능은 시간에 따라 점차 늘어가고 있습니다. 그러나, 가장 필요한 리얼타임 어플리케이션은 -rt 패치셋을 계속 사용하고 있습니다.

Figure 14.7의 꼭대기에 보인 preemption 불가 커널은 CONFIG_PREEMPT=n과 함께 빌드되어서 리눅스 커널 내에서의 수행은 preemption 될 수 없습니다. 이는 커널의 리얼타임 응답시간은 리눅스 커널의 가장 긴, 실제로도 긴, 코드 경로에 제한됨을 의미합니다. 그러나, 사용자 모드 수행은 preemption 가능하므로 우상단에 보인 리얼타임 리눅스 프로세스들은 좌상단에 보인 비 리얼타임 리눅스 프로세스를 비 리얼타임 리눅스 프로세스가 사용자 모드에서 수행중인 동안에는 언제든 preempt 시킬 수 있을 겁니다.

Figure 14.7의 가운데 열은 리눅스의 preemption 가능한 커널 개발 과정의 세 단계(왼쪽에서 오른쪽으로)를 보입니다. 세 단계 모두에서, 리눅스 커널의 대부분의 프로세스 단계 코드는 preemption 될 수 있습니다. 이는 물론 리얼타임 응답시간을 크게 개선합니다만 여전히 RCU read-side 크리티컬 섹션, spinlock 크리티컬 섹션, 인터럽트 핸들러, 인터럽트가 불능화된 코드 구역, 그리고 preemption이 불능화된 코드 구역에서는 이 그림의 가운데 열의 가장 왼쪽의 빨간 박스들로 보인 것처럼 여전히 preemption이 불가능합니다. Preemption 가능한 RCU의 발전은 가장 오른쪽 다이어그램이 보이듯 RCU read-side 크리티컬 섹션이 preemption 가능하게 만들었습니다. 물론, 다른 리얼타임 기능이 시간의 흐름에 따라 더해졌습니다만, 이 모두를 이 다이어그램에 쉽게 보일 수가 없습니다. 그에 대해서는 Section 14.3.5.1에서 이야기 하겠습니다.

Figure 14.7의 아랫열은 쓰레드화된(그리고 따라서 preemption 가능한) 많은 기기를 위한 인터럽트 핸들러를 갖는, 따라서 이 드라이버들의 “인터럽트 불능화된” 영역들을 preemption 가능하게 하는 -rt 패치셋을 보입니다. 이 드라이버들은 각 드라이버의 프로세스 단계 부분들을 쓰레드화된 인터럽트 핸들러와 동기화 시키기 위해 락킹을 대신 사용합니다. 마지막으로, 어떤 경우에는 preemption의 불능화가 migration 불능화로 대체됩니다.

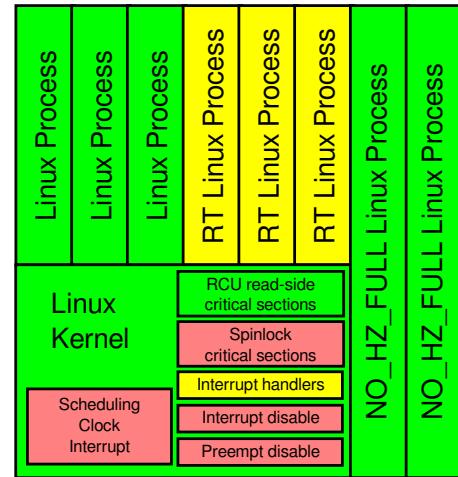


Figure 14.8: CPU Isolation

이는 -rt 패치셋을 사용해 수행되는 많은 시스템에서의 상당한 응답시간 개선을 보입니다 [RMF19, dOCdO19].

마지막 방법은 리얼타임 프로세스를 방해할 수 있는 모든 것을 치우는 것으로, Figure 14.8에 보인 것처럼 이 프로세스가 필요한 모든 CPU로부터 다른 일들을 모두 제거해 버리는 겁니다. 이는 3.10 리눅스 커널에서 CONFIG_NO_HZ_FULL Kconfig 패러미터를 통해 구현되었습니다 [Cor13, Wei12]. 이 방법은 최소 하나의 관리 CPU, 예를 들면 커널 daemon 같은 것이 뒤에서 수행되게 할 것을 필요로 함을 알아두는게 중요합니다. 그러나, 비 관리 CPU에서 수행될 수 있는 태스크가 단 하나일 때에는 스케줄링 클락 인터럽트가 해당 CPU에서는 꺼져버려서 상관과 OS jitter의 중요한 원천이 제거됩니다. 일부 예외와 함께, 이 커널은 다른 일들이 비 관리 CPU로부터 사라지게 하지는 않으나 특정 CPU에 하나의 태스크만 수행 가능할 때의 성능을 낫게 만들어 줍니다. 여러 userspace 도구들이 특정 CPU가 두개 이상의 수행 가능 태스크를 갖지 못하게 만드는데 사용될 수 있습니다. 올바르게 구성된다면, CONFIG_NO_HZ_FULL은 베어메탈 시스템의 그것에 근접하는 리얼타임 쓰레드 수준의 성능을 제공합니다 [ACA⁺18].

물론, 이 방법들 가운데 무엇이 리얼타임 시스템을 위한 최고의 방법인지에 대해선 상당한 논쟁이 있었으며 그 논쟁은 상당한 시간동안 진행되었습니다 [Cor04a, Cor04c]. 보통 그렇듯, 답은 “경우에 따라 다르다” 인 듯 보이는데, 다음 섹션들에서 이를 다릅니다. Section 14.3.5.1은 이벤트 주도 리얼타임 시스템을 다루고, Section 14.3.5.2는 CPU-제한 polling 반복문을 사용하는 리얼타임 시스템을 알아봅니다.

14.3.5.1 Event-Driven Real-Time Support

이벤트 주도 리얼타임 어플리케이션을 위한 운영체제 지원은 상당히 많은 게 필요할 수 있습니다만 이 셙션은 그 중 몇 가지, 즉 타이머, 쓰레드 인터럽트, 우선순위 계승, preemptible RCU, 그리고 preemptible spinlock에만 집중합니다.

Timer는 리얼타임 오퍼레이션들을 위해 분명 무척 중요합니다. 어쨌건, 무언가가 특정 시간에 완료되었다고 기술할 수가 없다면, 그 시간에 어떻게 응답을 하겠습니까? 비 리얼타임 시스템에서조차도, 많은 수의 타이머가 만들어졌으며, 따라서 그것들은 아주 효율적으로 다루어져야 합니다. 예로는 TCP 연결을 위한 재전송 타이머(거의 항상 만료 전에 취소될),⁸ 시간이 명시된 지연(드물게 취소될, `sleep(1)`과 같은), 그리고 `poll()` 시스템콜을 위한 시간 기반 종료(거의 항상 만료 전에 취소될) 등이 있겠습니다. 따라서 그런 타이머를 위한 좋은 데이터 구조는 추가와 제거 기능이 빠르고 추가된 타이머의 갯수에 대해 $O(1)$ 인 우선순위 queue가 될 겁니다.

이 목적을 위한 고전적인 데이터 구조는 *calendar queue*로, 리눅스 커널에서는 *timer wheel*이라 불립니다. 이 오래된 데이터 구조는 이산적 이벤트 시뮬레이션에도 널리 사용됩니다. 아이디어는 시간은 수치화된다는 것으로, 예를 들어 리눅스 커널에서 시간 단위는 스케줄링 클락 인터럽트 기간입니다. 시간은 정수로 표현될 수 있으며, 정수가 아닌 시간을 위한 타이머를 추가하려는 시도는 가까운 정수 시간 단위로 변환될 겁니다.

한 가지 단순한 구현은 시간의 아래쪽 비트들로 인덱스 되는 배열을 할당하는 것입니다. 이는 이론상으로는 동작합니다만, 실제 시스템에서는 거의 항상 취소되는 긴 시간의 타임아웃을 (예를 들어, TCP session을 위한 두시간짜리 `keepalive` 타임아웃) 무척 많이 생성합니다. 이 긴 시간의 타임아웃은 작은 배열에서 문제를 일으킬 수 있는데, 대부분의 시간은 아직 만료되지 않은 타임아웃을 넘기느라 낭비될 것이기 때문입니다. 다른 한편, 우아하게 많은 수의 장기간 타임아웃을 수용할 수 있는 충분히 큰 배열은 너무 많은 메모리를 소모할텐데, 특히 성능과 확장성에 대한 염려는 각 CPU마다 그런 배열을 하나씩 요구할 것이라는 점에서 그렇습니다.

이 문제를 해결하는 일반적인 방법은 계층적인 여러 배열을 제공하는 것입니다. 이 계층의 가장 아랫단에서 각 배열 원소는 하나의 시간 단위를 의미합니다. 두 번째 계층에서, 각 배열 원소는 N 개의 시간 유닛을 의미하는데 N 은 각 배열의 원소 수입니다. 세 번째 계층에서, 각 배열 원소는 N^2 시간 유닛을 의미하며, 그렇게 계층의

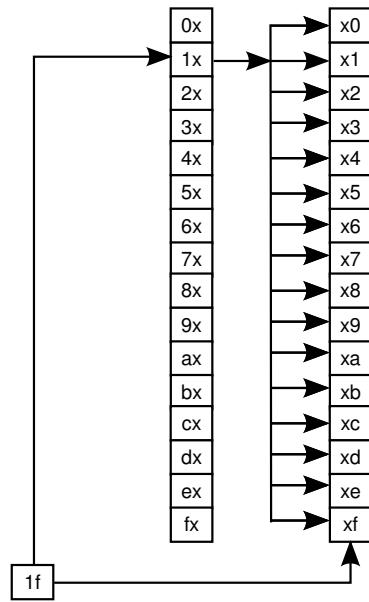


Figure 14.9: Timer Wheel

위쪽으로 반복됩니다. 이 방법은 개별적 배열들이 다른 비트를 통해 인덱스 될 수 있게 하는데, Figure 14.9에서 비현실적으로 작은 8 비트 클락을 보인 것과 같습니다. 여기서, 각 원소는 16 개의 원소를 가지므로, 시간의 아랫쪽 네개의 비트는 (현재 0xf) 아랫단(오른쪽) 배열을 인덱스 하며, 다음 네개의 비트(현재 0x1)은 그 윗 단계를 인덱스 합니다. 따라서, 우린 16 개의 원소를 갖는 두개의 배열을 가져서 총 32개 원소를 갖는데, 이는 하나님의 배열을 사용했다면 필요했을 256개 원소 배열보다 훨씬 작습니다.

이 방법은 처리량 기반 시스템에서 무척 잘 동작합니다. 각 타이머 오퍼레이션은 작은 상수의 $O(1)$ 이며, 각 타이머 원소는 최대 $m + 1$ 회 접근되는데, m 은 계층의 전체 높이입니다.

불행히도, timer wheel은 리얼타임 시스템에서 두 가지 이유로 잘 동작하지 못합니다. 첫 번째 이유는 Figures 14.10 and 14.11에 보여진 것처럼 타이머 정확도와 오버헤드 사이의 거친 트레이드오프가 있다는 겁니다. Figure 14.10에서, 타이머 처리는 밀리세컨드당 한번만 발생하여, 많은 (그러나 전부는 아닌!) 워크로드에 있어서 받아들여지기 충분히 낮은 오버헤드를 유지합니다만, 이는 또한 밀리세컨드 단위보다 작은 단위의 타임아웃은 불가능함을 의미합니다. 다른 한편, Figure 14.11은 10マイ크로세컨드마다 타이머 처리가 발생하는 경우를 보이는데, 대부분의 (그러나 전부는 아닌!) 워크로드에게 받아들여지기 충분한 세밀한 시간 단위를 제공하지

⁸ 최소한 합리적으로 낮은 패킷 손실율을 가정하면요!



Figure 14.10: Timer Wheel at 1 kHz



Figure 14.11: Timer Wheel at 100 kHz

만, 타이머를 매우 빈번하게 처리하기에 시스템은 다른 일을 할 시간을 충분히 갖지 못할 수도 있습니다.

두번째 이유는 타이머를 높은 단계에서 낮은 단계로 연결되어야 한다는 필요입니다. Figure 14.9 로 돌아가서, 우린 윗단 (왼쪽) 배열의 원소에 enqueue 된 모든 타이머는 아랫단 (오른쪽) 배열로 연결되어서 그것들의 시간이 도착했을 때 호출될 수 있게 해야 합니다. 불행히도, 연결을 기다리는 많은 수의 타임아웃이 있을 수 있는데, 큰 수의 단계를 갖는 timer wheel에서는 특히 그렇습니다. 통계의 힘이 이 연결을 처리량 기반 시스템에서는 문제가 아니게 만듭니다만, 리얼타임 시스템에서는 응답 시간 악화로 문제를 초래할 수 있습니다.

물론, 리얼타임 시스템은 단순히 다른 데이터 구조를 사용할 수도 있는데, 예를 들면 heap이나 tree 같은

것으로, 추가와 삭제 오퍼레이션에서 $O(1)$ 제한을 포기하고 데이터 구조 유지 오퍼레이션들에서 $O(\log n)$ 제한을 얻는 겁니다. 이는 특수 목적 RTOS 들에서는 좋은 선택일 수 있으나, 리눅스와 같이 극단적으로 많은 수의 타이머를 지원하는 범용 시스템에서는 비효율적입니다.

리눅스 커널의 -rt 패치셋에서 선택한 해결책은 나중의 행동을 스케줄 하는 타이머와 TCP 패킷 손실과 같은 낮은 가능성의 에러의 처리를 스케줄 하는 타임아웃을 다르게 처리하는 겁니다. 한가지 핵심 발견은 에러 처리는 일반적으로 특별히 시간에 치명적이지 않으며, 따라서 timer wheel 의 밀리세컨드 단위가 충분히 좋다는 겁니다. 또 다른 핵심 발견은 에러 처리 타임아웃은 일반적으로 매우 일찍, 종종 윗단에서 아랫단으로 연결되기 전에 취소된다는 겁니다. 또한, 타이머 이벤트보다 많은 에러 처리 타임아웃을 갖는 시스템이 흔해서 $O(\log n)$ 데이터 구조는 타이머 이벤트를 위한 받아들여질 수 있는 수준의 성능을 제공합니다.

그러나, 더 잘할 수도 있는데, 타이머를 연결하는 걸 그냥 거부하는 겁니다. 연결 대신, calendar queue 의 아랫단으로 연결되어야 할 타이머는 그대로 있습니다. 이는 이 시간 기간 내에서의 몇퍼센트의 에러를 초래합니다만 이게 문제일 수 있는 소수의 상황에서는 tree 기반의 고해상도 타이머 (hrtimers) 를 사용할 수 있습니다.

요약해서, 리눅스 커널의 -rt 패치셋은 에러 처리 타임아웃을 위해서는 timer wheel 을 쓰고 타이머 이벤트를 위해선 tree 를 사용해서 각 카테고리에 필요한 품질의 서비스를 제공합니다.

Threaded interrupts 는 리얼타임 응답 악화의 심각한 원천, 즉 Figure 14.12 에 보인 것과 같은 오래 수행되는 인터럽트 핸들러를 처리하기 위해 사용됩니다. 이 응답 시간은 하나의 인터럽트로 큰 수의 이벤트를 전달할 수 있는, 즉 인터럽트 핸들러가 이 이벤트 전체를 처리하기 위해 연장된 시간동안 수행될 것을 의미하는 기기에서 특히 문제가 될 수 있습니다. 더 나빠질 수 있는건 여전히 수행중인 인터럽트 핸들러에게 새 이벤트를 전달할 수 있는 기기들로, 그런 인터럽트 핸들러는 무한히 수행될 수도 있어서 리얼타임 응답을 무한히 악화시킬 수 있습니다.

이 문제를 해결하는 한가지 방법은 Figure 14.13 에 보인 쓰레드화된 인터럽트 (threaded interrupts) 를 사용하는 겁니다. 설정 가능한 우선순위로 수행되며 `reemption` 가능한 IRQ 쓰레드의 컨텍스트에서 수행되는 인터럽트 핸들러입니다. 그럼 이 기기 인터럽트 핸들러는 짧은 시간동안만 수행되는데, IRQ 쓰레드가 새 이벤트를 알아차릴 수 있을 만큼만 긴 정도입니다. 그럼에서 보였듯, 쓰레드화된 인터럽트는 링러타임 응

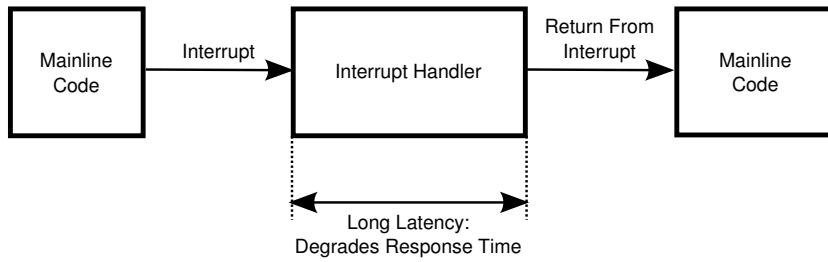


Figure 14.12: Non-Threaded Interrupt Handler

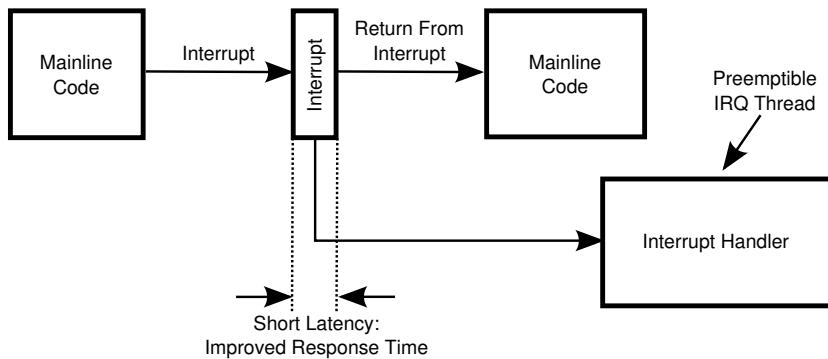


Figure 14.13: Threaded Interrupt Handler

답을 크게 개선할 수 있는데, 부분적으로는 IRQ 쓰레드의 컨텍스트에서 수행되는 인터럽트 핸들러가 높은 우선순위 리얼타임 쓰레드에게 preemption 당할 수 있기 때문입니다.

하지만 공짜 점심은 없으며 쓰레드화된 인터럽트에게 단점이 있습니다. 그런 단점 중 하나는 증가되는 인터럽트 응답시간입니다. 인터럽트는 즉각 수행되는 대신, 이 핸들러의 수행 부분은 IRQ 쓰레드가 수행될 수 있을 때까지 미뤄집니다. 물론, 인터럽트를 생성하는 기기가 리얼타임 어플리케이션의 중요 경로에 있지 않다면 이는 문제가 되지 않습니다.

또 다른 단점은 잘못 짜여진 높은 우선순위 리얼타임 코드가 인터럽트 핸들러를 기아 상태에 빠지게 할 수 있다는 것으로, 예를 들어 네트워크 코드가 수행되지 못하게 해서 결국 문제를 디버깅하기 어렵게 만드는 겁니다. 따라서 개발자들은 높은 우선순위 리얼타임 코드를 작성할 때 큰 주의를 기울여야 합니다. 이를 **스파이더맨 원칙**이라고 부릅니다: 큰 힘에는 큰 책임이 따릅니다.

Priority inheritance 는 다른것들 가운데에서도 preemption 가능한 인터럽트 핸들러에서 획득되는 락들로 인해 일어날 수 있는 우선순위 역전을 [SRL90] 처리하기 위해 사용됩니다. 낮은 우선순위 쓰레드가 락을

잡았는데 중간 우선순위의 쓰레드당 최소 하나는 되는 쓰레드들에 의해 preemption 당한다고 생각해 봅시다. 인터럽트가 발생하면 높은 우선순위 IRQ 쓰레드가 이 중간 우선순위 쓰레드들 가운데 하나를 preempt 하지만 낮은 우선순위 쓰레드가 쥐고 있는 락을 획득하려 하기 전까지만입니다. 불행히도, 이 낮은 우선순위 쓰레드는 수행을 시작하기 전까지 이 락을 높을 수 없는데, 중간 우선순위 쓰레드가 그걸 못하게 하고 있습니다. 따라서 높은 우선순위 IRQ 쓰레드는 중간 우선순위 쓰레드들 중 하나가 CPU를 놓기 전까지 이 락을 획득하지 못합니다. 요약하자면, 중간 우선순위 쓰레드가 간접적으로 높은 우선순위 IRQ 쓰레드를 막고 있게 되어, 고전적인 우선순위 역전이 형성됩니다.

이 우선순위 역전은 낮은 우선순위 쓰레드는 락을 잡고 있는 동안 인터럽트를 불능화 시켜야 할 것이어서 중간 우선순위 쓰레드가 preempt 시키는 걸 막을 것이기 때문에 쓰레드화 되지 않은 인터럽트에서도 발생될 수 있음을 알아두십시오.

우선순위 계승 (priority-inheritance) 해결책에서, 락을 획득하려는 높은 우선순위 쓰레드는 자신의 우선순위를 그 락을 잡고 있는 낮은 우선순위 쓰레드에게 그 락이 해제될 때까지 기부해서 장기간의 우선순위 역전을 막습니다.



Figure 14.14: Priority Inversion and User Input

물론, 우선순위 계승은 자신의 한계를 가지고 있습니다. 예를 들어, 여러분이 여러분의 어플리케이션을 우선순위 역전을 완전히 막게 설계할 수 있다면, 여러분은 뭔가 더 높은 응답시간을 얻을 겁니다 [Yod04b]. 우선순위 계승은 최악의 경우 응답시간에 한쌍의 컨텍스트 전환을 더하므로 이는 놀랍지 않습니다. 그러나, 우선순위 계승은 응답시간의 무한한 대기를 제한된 증가로 변환시키며 우선순위 계승의 소프트웨어 엔지니어링 이득은 많은 어플리케이션이 있어 그 응답시간 비용보다 큰 이득을 줄 겁니다.

또 다른 한계는 특정 운영체제의 문맥 내에서의 락 기반 우선순위 역전만을 해결한다는 겁니다. 이게 해결할 수 없는 우선순위 역전 시나리오 중 하나는 CPU를 사용하는 중간 우선순위 프로세스 집합에 의해 *preemption* 당한 낮은 우선순위 프로세스에 의해 쓰,여질 메세지를 네트워크 소켓에서 기다리는 높은 우선순위 쓰레드입니다. 또한, 우선순위 계승을 사용자 입력에 적용하는 것의 한가지 잠재적 단점이 Figure 14.14에 그려져 있습니다.

마지막 한계는 reader-writer 락킹에 연관됩니다. 우리가 매우 큰 수의, 예를 들어 수천개의, 낮은 우선순위 쓰레드를 가지고 있으며 이들 각각은 특정 reader-writer 락을 읽기 모드로 잡는다고 해봅시다. 이 쓰레드가 모두 CPU 당 최소 하나는 되는 중간 우선순위 쓰레드 집합에 의해 *preemption* 당한다고 해봅시다. 마지막으로, 높은 우선순위 쓰레드가 깨어나서 이 reader-writer 락을 쓰기 모드로 획득하려 한다고 해봅시다. 이 락을 읽기 모드로 쥐고 있는 쓰레드의 우선순위를 얼마나 열심히 높이는가에 관계없이, 이는 높은 우선순위 쓰레드가 쓰기모드 락 획득을 완료하기까지는 긴 시간이 걸릴 수 있을 겁니다.

이 reader-writer 락 우선순위 역전 난제에 대한 여러 해결책이 있습니다:

1. 한번에 하나의 reader-writer 락의 읽기 모드 획득만을 허용합니다. (이는 리눅스 커널의 -rt 패치셋에서 전통적으로 취해진 방법입니다.)
2. 한번에 특정 reader-writer 락의 N 개 읽기 모드 획득만을 허용하며, N 은 CPU 갯수가 되게 합니다.
3. 한번에 특정 reader-writer 락의 N 개 읽기 모드 획득만을 허용하며, N 은 개발자에 의해 어떻게든 정해질 수 있게 합니다. 리눅스 커널의 -rt 패치셋이 이 방법을 취할 기회가 있습니다.
4. 높은 우선순위 쓰레드가 낮은 우선순위 쓰레드에 의해 읽기 모드로 획득될 수 있는 reader-writer 락을 쓰기 모드로 획득하는 걸 금합니다. (이는 우선순위 제한 프로토콜 [SRL90]의 한 변종입니다.)

Quick Quiz 14.6: 하지만 한번에 하나의 reader-writer 락 읽기 모드 획득만을 허용한다면, 그건 배타적 락과 동일한 거 아닌가요???

동시의 읽기 쓰레드를 허용하지 않는 제약은 결국 허용 불가능하게 되었으므로, -rt 개발자들은 리눅스 커널이 reader-writer spinlock을 어떻게 사용하는지 주의 깊게 들여다 보았습니다. 그들은 시간에 치명적인 코드는 reader-writer 락을 쓰기 모드로 획득하는 커널 부분을 드물게만 사용하므로 쓰기 쓰레드의 starvation의 가능성은 모든 걸 망칠 정도는 아님을 배웠습니다. 따라서 그들은 쓰기 모드 획득이 다른 것들 가운데 우선순위 계승을 사용하지만 읽기 모드 획득은 쓰기 모드 획득들 보다 높은 우선순위를 갖는 리얼타임 reader-writer 락을 만들었습니다. 이 방법은 실전에서 잘 동작하는 것으로 보였으며, 여러분의 사용자가 정말로 필요한 게 무엇인지 분명히 이해하는게 중요하다는 또 다른 교훈입니다.

이 구현의 한가지 흥미로운 세부사항은 *rt_read_lock()*과 *rt_write_lock()* 함수들이 RCU read-side 크리티컬 섹션을 진입하며 *rt_read_unlock()*과 *rt_write_unlock()* 함수들이 이 크리티컬 섹션을 빠져 아온다는 점입니다. 비 리얼타임 커널의 reader-writer 락킹 함수는 크리티컬 섹션 내에서 *preemption*을 금하기 때문에 필요하며, *synchronize_rcu()*는 모든 앞서 존재한 reader-writer 락 크리티컬 섹션들이 완료될 때까지 기다릴 것이라는 사실에 의존하는 reader-writer locking 사용처가 정말 존재합니다. 이를 여러분을 위한 교훈으로 삼으세요: 여러분의 사용자들이 정말 필요한 게 무엇인지 이해하는 것은 성능만이 아니라 운영을 고치는데 매우 중요합니다. 그뿐만이 아니라, 여러분의 사용자가 정말로 필요한 건 시간에 따라 변합니다.

이는 -rt 커널의 reader-writer 락킹 크리티컬 섹션이 RCU 우선순위 높이기에 영향 받는다는 부가 작용을

Listing 14.3: Preemptible Linux-Kernel RCU

```

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     barrier();
10    if (!--current->rcu_read_lock_nesting)
11        barrier();
12    if (READ_ONCE(current->rcu_read_unlock_special.s)) {
13        rCU_read_unlock_special(t);
14    }
15 }

```

일으킵니다. 이는 reader-writer 락 읽기 스레드가 연장된 시간 동안 preemption 당하는 문제에 대한 부분적 해결책을 제공합니다.

또한 reader-writer 락을 RCU 로 변환함으로써 reader-writer 락 우선순위 역전을 막는 것도 가능한데, 다음 섹션에서 간단히 이야기 합니다.

Preemptible RCU 는 가끔 Section 9.5에서 이야기되었듯 reader-writer 락킹의 대체제로 사용될 수 있습니다 [MW07, MBWW12, McK14d]. 사용될 수 있는 곳에서라면, 그것은 읽기 쓰레드와 쓰기 쓰레드가 동시에 수행되는 것을 허용하는데, 이는 낮은 우선순위 읽기 쓰레드가 높은 우선순위 업데이트 쓰레드에게 어떤 종류의 우선순위 역전 행위를 일으키는 것도 막아줍니다. 그러나, 이게 유용하려면 오래 수행되는 RCU read-side 크리티컬 섹션을 preempt 시킬 수 있어야 합니다 [GMTW08]. 그러지 않는다면 긴 RCU read-side 크리티컬 섹션은 상당한 리얼타임 응답시간을 초래할 겁니다.

그래서 preemption 가능한 RCU 구현이 리눅스 커널에 추가되었습니다. 이 구현은 커널 내의 모든 태스크 각각의 상태를 개별적으로 추적해야 할 필요를 preempt된 태스크들의 리스트를 그것들의 현재 RCU read-side 크리티컬 섹션 내에 유지함으로써 막았습니다. Grace period 는 다음의 경우에 종료될 수 있습니다: (1) 모든 CPU 가 현재 grace period 시작 이전부터 효과를 발휘한 RCU read-side 크리티컬 전부를 완료했다면, 그리고 (2) 그런 앞서서부터 존재한 크리티컬 섹션 내에서 preempt 되었던 태스크들이 그들을 그들의 리스트에서 제거했다면. 이 구현의 간략화된 버전이 Listing 14.3에 보여져 있습니다. __rcu_read_lock() 함수는 라인 1-5에 있고 __rcu_read_unlock() 함수는 라인 7-15에 있습니다.

__rcu_read_lock() 의 라인 3 는 중첩된 rCU_read_lock() 호출의 횟수를 담는 태스크별 카운터의 값을 증가시키고, 라인 4 는 컴파일러가 RCU read-side

크리티컬 섹션의 뒤따르는 코드를 rCU_read_lock() 앞으로 재배치 하는 걸 막습니다.

__rcu_read_unlock() 의 라인 9 은 컴파일러가 크리티컬 섹션 내의 코드를 이 함수의 나머지와 재배치시키는 것을 막습니다. 라인 10 는 중첩 카운트의 값을 감소시키고 그게 0이 되었는지 검사하는데, 달리 말하자면 이게 중첩 집합의 가장 바깥 rCU_read_unlock()에 연관되었는지 확인합니다. 그렇다면, 라인 11 는 컴파일러가 이 중첩 상태 업데이트를 라인 12 의 특별 처리 검사와 재배치 하는 걸 막습니다. 특별 처리가 필요하다면, 13 의 rCU_read_unlock_special() 이 이를 처리합니다.

필요할 수 있는 특별한 처리가 여러 종류 있지만, 우린 RCU read-side 크리티컬 섹션이 preemption 되었을 때 필요한 것에만 집중하겠습니다. 이 경우, 그 태스크는 스스로를 자신의 RCU read-side 크리티컬 섹션 내에서 처음 preemption 되었을 때 자신이 추가된 리스트에서 스스로를 제거해야 합니다. 그러나, 이 리스트는 락으로 보호됨에 유의할 필요가 있는데, 이는 rCU_read_unlock() 이 더이상 lockless 가 아님을 의미합니다. 그러나, 가장 높은 우선순위 쓰레드는 preemption 되지 않을 것이며, 따라서 그런 최고 우선순위 쓰레드에게 rCU_read_unlock() 은 어떤 락도 획득하려 하지 않을 겁니다. 또한, 주의 깊게 구현된다면 락킹은 리얼타임 소프트웨어를 동기화 하는데 사용될 수 있습니다 [Bra11, SM04].

Quick Quiz 14.7: Listing 14.3 의 라인 12 의 t->rcu_read_unlock_special.s 의 로드 직후에 preemption 이 일어났다고 해봅시다. 그럼 그 태스크는 rCU_read_unlock_special() 을 호출하지 못하게 될 수도 있고, 따라서 현재 grace period 를 막고 있는 태스크들의 리스트에서 스스로를 제거할 수 없게 되어서 그 grace period 가 무한히 연장되게 하지 않을까요?

■

RCU 의 또다른 중요한 리얼타임 기능은 RCU 콜백 수행을 커널 쓰레드로 넘기는 겁니다. 이를 사용하기 위해선 여러분의 커널은 CONFIG_RCU_NOCB_CPU=y 와 함께 빌드되고 어떤 CPU 에게 넘겨질 것인지 명시하는 rCU_nocbs= 커널 부팅 패러미터와 함께 부팅되어야 합니다. 대안적으로, Section 14.3.5.2에서 이야기된 nohz_full= 커널 부팅 패러미터에 의해 명시되는 모든 CPU 는 또한 그들의 RCU 콜백들을 떠넘기게 됩니다.

요약하자면, 이 preemption 가능한 RCU 구현은 읽기가 대부분인 데이터 구조에 대해 큰 수의 읽기 쓰레드에게 우선순위 높이기를 할 때 피할 수 없는 지연 없이, 그리고 콜백 수행으로 인한 지연 없이 리얼타임 응답을 가능하게 합니다.

Preemptible spinlocks 는 리눅스 커널의 긴 스픈락 기반 크리티컬 섹션 기간 때문에 -rt 패치셋의 중요한 부분입니다. 이 기능은 아직 메인라인에 들어오지 못했습니다: 그건 컨셉상으로는 스픈락을 위한 sleeplock의 간단한 대체제이지만, 상대적으로 논란의 여지가 있는 것으로 드러났습니다. 또한 메인라인 리눅스 커널에 있는 리얼타임 기능들은 굉장히 많은 사용처에 이미 충분해서 2010년대 초반 -rt 패치셋 개발을 느려지게 만들었습니다 [Edg13, Edg14]. 그러나, preemption 가능한 스픈락은 수십 마이크로세컨드 아래의 리얼타임 응답시간을 이뤄야 하는 태스크에게는 절대적으로 필요합니다. 다행히도, 리눅스 재단은 -rt 패치셋의 남아있는 코드를 메인라인으로 옮기기 위한 기금을 지원하기 위한 노력을 만들었습니다.

Per-CPU variables 는 성능상의 이유로 리눅스 커널에서 많이 사용됩니다. 리얼타임 어플리케이션에서는 불행하게도 많은 per-CPU 변수의 사용처가 그런 변수들 여럿의 업데이트와 연관될 필요가 있으며 그들은 일반적으로 preemption 을 끄는 것으로 제공되어서 결국 리얼타임 응답시간을 악화시킵니다. 리얼타임 어플리케이션은 per-CPU 변수 업데이트를 하는 어떤 다른 방법이 분명 필요합니다.

한가지 대안은 per-CPU 스픈락을 제공하는 것으로 앞서 언급했듯 실제로는 sleeplock 이어서 그것의 크리티컬 섹션은 preemption 될 수 있으며 따라서 우선순위 상속이 제공됩니다. 이 방법에서, per-CPU 변수 그룹을 업데이트 하는 코드는 현재 CPU의 스픈락을 획득하고, 업데이트를 한 후, 획득했던 락을 내려놓아야 하며 이 과정에서 preemption 이 수행 흐름을 djEjs 다른 CPU로 옮기는 이전을 만들었을 수도 있음을 숙지해야 합니다. 그러나, 이 방법은 오버헤드와 데드락을 모두 일으킵니다.

다른 대안은 2021년 초에 -rt 패치셋에서 사용된 것으로, preemption 불능화를 migration 불능화로 바꾸는 겁니다. 이는 해당 커널 쓰레드가 per-CPU 변수 업데이트 동안 자신의 CPU에 머무르게 됨을 보장합니다만, preemption 때문에 다른 커널 쓰레드가 같은 변수에 대한 업데이트를 중간에 훌뿌릴 수 있게 합니다. 이게 문제가 되지 않는 통계 모으기 같은 경우도 있습니다. 그런 업데이트 중간의 preemption 이 문제가 되는 놀랍도록 드문 경우, 그 사용처는 아마도 그 사용처에 한정된 per-CPU 락 집합을 통해 그 업데이트를 올바르게 동기화 시켜야만 합니다. 락을 사용하는 건 또다시 데드락 가능성은 불러옵니다만, 이 락들의 사용처별 이라는 특성이 그런 데드락을 관리하고 막기 쉽게 해줍니다.

Closing event-driven remarks. 예를 들면 데드라인 스케줄링 [dO18b, dO18a] 같은, 세계급 리얼타임 응답

시간을 얻는데 치명적으로 중요한 다른 리눅스 커널 커뮤니티도 물론 많습니다만, 이 섹션에 열거된 것들은 -rt 패치셋에 의해 증강된 리눅스 커널의 작업에 대한 좋은 느낌을 줄 겁니다.

14.3.5.2 Polling-Loop Real-Time Support

첫 인상으로는, polling 반복문의 사용은 모든 가능한 운영체제 간섭 문제를 없앨 것으로 보일 수도 있습니다. 어쨌건, 어떤 CPU가 절대 커널에 들어가지 않는다면, 커널은 완전히 그림 밖에 있습니다. 그리고 커널을 밖에 두는 전통적인 방법은 단순히 커널을 갖지 않는 것으로, 많은 리얼타임 어플리케이션이 실제로 기계 바로 위에서 돌아가는데, 특히 8비트 마이크로컨트롤러에서 수행되는 것들이 그렇습니다.

어떤 사람들은 현대의 운영체제 커널에서 단순히 하나의 CPU에 성능이 종속되는 사용자 모드 쓰레드를 특정 CPU 위에서 돌림으로써 모든 간섭 원인을 제거하고 순수 기계 성능을 얻을 수 있기를 바랄 수도 있습니다. 실제는 물론 더 복잡하지만, Frederic Weisbecker에 의해 구현이 이끌어졌고 리눅스 커널에 3.10 버전에 받아들여진 NO_HZ_FULL [Cor13, Wei12] 덕분에 정말 그렇게 하는게 가능해지고 있습니다. 그렇다고 하나, 있을 수 있는 여러 OS jitter의 원인을 제어해야 하기 때문에 그런 환경을 올바르게 구성하기 위해선 상당한 주의가 필요합니다. 아래의 이야기는 여러 OS jitter 원인을 다루는데, 기기 인터럽트, 커널 쓰레드와 daemon, 스케줄러 리얼타임 쓰로틀링 (이건 블록이 아니라 기능입니다!), 타이머, 비 리얼타임 디바이스 드라이버, 커널 내부 전역 동기화, 스케줄링 클락 인터럽트, 페이지 폴트, 그리고 마지막으로 비 리얼타임 하드웨어와 펌웨어를 포함합니다.

인터럽트는 OS jitter의 많은 부분의 훌륭한 원천입니다. 불행히도, 인터럽트는 시스템이 바깥 세상과 소통하기 위해 절대적으로 필요합니다. 이 OS jitter와 바깥 세상과의 연락을 유지하는 것 사이의 충돌을 해결하는 한가지 방법은 작은 수의 최소 운영 CPU를 예약하고 모든 인터럽트를 이 CPU 들로 향하도록 강제하는 겁니다. 리눅스 소스 트리의 Documentation/IRQ-affinity.txt 파일은 어떻게 기기 인터럽트를 특정 CPU로 향하게 할 수 있는지 설명하는데, 2021년 초 기준으로는 다음과 같습니다:

```
$ echo 0f > /proc/irq/44/smp_affinity
```

이 커맨드는 인터럽트 #44를 CPU 0~3에 가둡니다. 스케줄링 클락 인터럽트는 특수한 처리를 필요로 하며, 이 섹션의 뒷부분에서 다룸을 알아두십시오.

OS jitter의 두번째 원천은 커널 쓰레드와 daemon입니다. RCU의 grace-period kthread (rcu_bh, rcu_

preempt, 그리고 rcu_sched) 와 같은 개별 커널 쓰레드는 taskset 커맨드, sched_setaffinity() 시스템 콜, 또는 cgroups 를 통해 어떤 CPU 에게든 할당될 수 있습니다.

Per-CPU kthread 는 종종 더 어려우며, 가끔은 하드웨어 구성과 워크로드 배치를 강제합니다. 이런 kthread 로부터의 OS jitter 를 막기 위해선 특정 종류의 하드웨어가 리얼타임 시스템에 부착되지 않았거나 모든 인터럽트와 I/O 초기화가 최소 운영용 CPU 에서만 처리되거나, 일들이 실제 일을 하는 CPU 에서 처리되지 않게끔 특수한 커널 Kconfig 또는 부팅 패러미터를 선택하거나, 실제 일을 하는 CPU 가 결코 커널에 들어가지 않게 해야 합니다. 특정 per-kthread 를 위한 조언을 리눅스 커널 소스 Documentation 디렉토리의 admin-guide/kernel-per-CPU-kthreads.rst 에서 찾을 수 있을 겁니다.

리얼타임 우선순위를 가지고 수행되는 CPU 를 주로 사용하는 쓰레드에 대한 OS jitter 의 세번째 원천은 스케줄러 자신입니다. 이는 의도적인 디버깅 기능으로, 리얼타임 어플리케이션이 무한 반복문 버그가 있다 해도 비리얼타임 작업이 초당 최소 50 밀리세컨드는 시간을 얻음을 보장하기 위해서입니다. 그러나, 여러분이 polling 반복문 스타일의 리얼타임 어플리케이션을 수행 중일 때에는 이 디버깅 기능을 꺼야 할 겁니다. 이는 다음과 같이 될 수 있습니다:

```
$ echo -1 > /proc/sys/kernel/sched_rt_runtime_us
```

이 커맨드를 수행하기 위해선 여러분은 물론 root 권한을 가지고 있어야 하며, 또한 앞서 이야기된 스파이더 맨 원칙을 주의 깊게 고려해야 할 겁니다. 위험을 최소화 하는 한가지 방법은 앞의 문단에서 이야기된 것처럼 인터럽트와 커널 쓰레드/daemon 을 CPU 를 주로 사용하는 리얼타임 쓰레드가 수행되는 모든 CPU 로부터 격리시키는 것입니다. 또한, Documentation/scheduler 디렉토리의 것들을 주의 깊게 읽어야 합니다. sched_rt-group.rst 파일의 내용은 특히 중요한데, 여러분이 CONFIG_RT_GROUP_SCHED Kconfig 패러미터를 통해 켜지는 cgroups 리얼타임 기능을 사용한다면 더욱 그렇습니다.

OS jitter 의 네번째 원천은 타이머에서 옵니다. 많은 경우, 특정 CPU 를 커널 바깥에 두는 것은 타이머가 해당 CPU 에 스케줄 되는 것을 막습니다. 한가지 중요한 예외는 반복되는 타이머로, 특정 타이머 핸들러가 같은 타이머의 다음 발생을 예약하는 경우입니다. 그런 타이머가 어떤 이유로든 특정 CPU 에서 시작된다면, 그 타이머는 해당 CPU 에서 주기적으로 수행을 계속해서 무한히 OS jitter 를 가할 겁니다. 반복되는 타이머를 다른 곳으로 넘기는 한가지 잔인하지만 효과적인 방법은

CPU 를 주로 사용하는 리얼타임 어플리케이션 쓰레드를 수행하는 모든 작업용 CPU 를 CPU hotplug 기능을 이용해 offline 시키고, 이 같은 CPU 를 다시 online 시킨 후, 여러분의 리얼타임 어플리케이션을 시작하는 겁니다.

OS jitter 의 다섯번째 원천은 리얼타임으로 개발되지 않은 디바이스 드라이버에 의해 제공됩니다. 오래된 정식 예를 들어보면, 2005년 VGA 드라이버는 인터럽트를 끈 채 프레임 버퍼를 0 으로 채움으로써 화면을 깜빡였는데, 이는 수십 밀리세컨드의 OS jitter 를 발생시켰습니다. 디바이스 드라이버가 일으키는 OS jitter 를 막는 한가지 방법은 리얼타임 시스템에서 많이 사용된, 따라서 리얼타임 버그가 수정된 기기를 주의 깊게 선택하는 겁니다. 또 다른 방법은 해당 기기의 인터럽트와 해당 기기를 사용하는 모든 코드를 지정된 최소 운영용 CPU 에 가두는 겁니다. 세번째 방법은 이 기기의 리얼타임 워크로드 지원을 위한 기능을 테스트하고 리얼타임 버그를 고치는 겁니다.⁹

OS jitter 의 여섯번째 원천은 일부 커널 내부 전체 시스템 동기화 알고리즘에 의해 제공되는데, 가장 눈에 띄는 건 아마도 전역 TLB-flush 알고리즘일 겁니다. 이는 메모리 언매핑 오퍼레이션, 특히 커널 내에서의 언매핑 오퍼레이션을 막는 것으로 회피될 수 있습니다. 2021년 초 기준으로 커널 내부 언매핑 오퍼레이션을 막는 방법은 커널 모듈 언로딩을 막는 겁니다.

OS jitter 의 일곱번째 원천은 스케줄러 클락 인터럽트와 RCU 콜백 수행에 의해 제공됩니다. 이것들은 여러분의 커널을 NO_HZ_FULL Kconfig 패러미터를 켜 채로 빌드하고 nohz_full= 패러미터가 리얼타임 쓰레드를 수행하게 될 작업용 CPU 들의 리스트를 명시하게 함으로써 막아질 수 있습니다. 예를 들어, nohz_full=2-7 은 CPU 2, 3, 4, 5, 6, 그리고 7 을 작업용 CPU 로 지정하여서 CPU 0 과 1 을 최소 운영용 CPU 로 둡니다. 작업용 CPU 는 각 작업용 CPU 에 두개 이상의 수행 가능 태스크가 있지 않은 한 스케줄링 클락 인터럽트를 일으키지 않으며, 각 작업용 CPU 의 RCU 콜백은 최소 운영용 CPU 중 하나에서 수행될 겁니다. 스케줄링 클락 인터럽트를 끈 CPU 는 거기 하나의 수행 가능 태스크만 있을 것이기 때문에 그 CPU 는 *adaptive ticks mode* 또는 nohz_full 모드에 있다고 이야기 됩니다. 시스템의 나머지 부분들에 의해 발생될 최소 운영 로드를 처리하기 충분한 최소 운영용 CPU 를 보장하는게 중요한데, 이를 위해선 주의 깊은 벤치마크와 튜닝이 필요할 겁니다.

OS jitter 의 여덟번째 원천은 페이지풀트입니다. 대부분의 리눅스 구현은 메모리 보호를 위해 MMU 를 사용하므로, 이런 시스템에서 돌아가는 리얼타임 어플

⁹ 여러분이 이 방법을 선택한다면, 다른 사람들의 이익을 위해 여러분의 수정을 엔스터프에 보내주시기 바랍니다. 어쨌건, 여러분의 어플리케이션을 나중 버전의 리눅스 커널에 포팅하려면, 여러분은 그런 “다른 사람들” 이 될 겁니다.

Listing 14.4: Locating Sources of OS Jitter

```

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

```

리케이션은 페이지풀트를 당할 수 있습니다. `mlock()`과 `mlockall()` 시스템콜을 사용해서 여러분의 어플리케이션의 페이지를 메모리에 고정시켜서 페이지풀트를 막으십시오. 물론, 스파이더맨 원칙이 적용되는데, 너무 많은 메모리를 고정시키는 것은 시스템이 다른 일을 하는 걸 막기 때문입니다.

OS jitter의 아홉번째 원천은 불행히도 하드웨어와 펌웨어입니다. 따라서 리얼타임 사용을 위해 설계된 시스템을 사용하는게 중요합니다.

불행히도, OS jitter 원천의 리스트는 결코 완벽할 수 없는데, 커널의 새로운 버전이 나옴에 따라 바뀔 것이기 때문입니다. 이는 OS jitter의 새로운 원천을 추적할 수 있을 필요를 만듭니다. CPU를 주로 사용하는 사용자모드 쓰레드가 돌아가는 CPU N 이 있다면, Listing 14.4에 보인 커맨드는 이 CPU가 커널에 진입한 모든 시간의 리스트를 만들 겁니다. 물론, 라인 5의 N 은 문제의 CPU의 숫자로 바뀌어야 하며, 라인 2의 1은 커널 내에서의 함수 호출 단계를 추가적으로 보기 위해 높아질 수 있습니다. 결과로 나오는 트레이스는 OS jitter의 원천을 추적해 들어가는 걸 도울 수 있습니다.

항상 그렇듯, 공짜 점심은 없고 `NO_HZ_FULL`도 예외가 아닙니다. 앞서 언급되었듯, `NO_HZ_FULL`은 커널/유저 전환 비용을 더 높게 하는데 멜타 프로세스 처리와 RCU 같은 커널 서브시스템에게 그 전환을 알려야 하는 필요 때문입니다. 거친 경험적 법칙으로, `NO_HZ_FULL`은 많은 종류의 리얼타임과 컴퓨팅이 주인 워크로드에 도움이 됩니다만, 높은 비율로 시스템콜과 I/O를 사용하는 워크로드를 해칩니다 [ACA⁺18]. 추가적인 제한, 트레이드오프, 그리고 구성상의 조언이 리눅스 소스 트리의 Documentation/timers/no_hz.rst에서 얻어질 수 있습니다.

봐왔듯이, CPU 위주의 리얼타임 쓰레드를 리눅스와 같은 범용 OS에서 돌리면서 기계 수준의 성능을 얻는 것은 세세한 부분에 대한 상당한 주의를 필요로 합니다. 자동화는 도움이 되겠고 어떤 자동화는 적용되었습니까만, 상대적으로 적은 사용자들을 놓고 볼 때, 자동화는 상대적으로 느리게 적용될 거라 예상될 수 있습니다. 그러나, 범용 운영체제를 돌리면서 기계에 근접한 성능을 얻는 것은 어떤 종류의 리얼타임 시스템에 대한 제한을 완화시킬 것이라 기대됩니다.

14.3.6 Implementing Parallel Real-Time Applications

리얼타임 어플리케이션을 개발하는 것은 넓은 영역의 주제들을 포함하며, 이 섹션은 그 중 일부 영역에 대해서만 다룹니다. 그러므로, Section 14.3.6.1은 리얼타임 어플리케이션에서 공통적으로 사용되는 소프트웨어 컴포넌트를 알아보고, Section 14.3.6.2에서는 어떻게 polling-loop 기반 어플리케이션들이 구현될 수 있는지 간략히 알아보며, Section 14.3.6.3은 스트리밍 어플리케이션의 비슷한 개론을 제공하고, Section 14.3.6.4는 이벤트 기반 어플리케이션을 다룹니다.

14.3.6.1 Real-Time Components

엔지니어링의 모든 영역에서 그렇듯, 생산성과 안정성을 위해 수많은 컴포넌트가 필수적입니다. 이 섹션은 리얼타임 소프트웨어 컴포넌트의 완전한 카탈로그가 아니라—그런 카탈로그는 여러 권의 책을 꽉 채울 겁니다—사용할 수 있는 컴포넌트들의 종류에 대한 짧은 개론입니다.

리얼타임 소프트웨어 컴포넌트를 위해 봐야 할 자연스런 장소는 wait-free 동기화 [Her91]를 제공하는 알고리즘일 것이며, 실제로 lockless 알고리즘들은 리얼타임 컴퓨팅에서 매우 중요합니다. 그러나, wait-free 동기화는 유한 시간 내에서의 진행만을 보장합니다. 100년은 유한하나, 이는 여러분의 마감기한이 마이크로세컨드나 밀리세컨드로 측정된다면 도움이 되지 않을 겁니다.

그렇다고 하나, 제한된 범위의 응답 시간을 제공하는 일부 중요한 wait-free 알고리즘들이 정말로 존재하는데, atomic test and set, atomic exchange, atomic fetch-and-add, single-producer/single-consumer FIFO queue 기반 circular array, 그리고 여러 per-thread 파티셔닝 기반 알고리즘들이 포함됩니다. 더해서, 최근의 연구는 lock-free 보장을 갖는 알고리즘들이¹⁰ 확률적으로 공평한 스케줄러와 fail-stop 버그의 부재를 가정한 상태에서 [ACHS13] 실제 사용처에서는 (wait-free의 개념으로) 동일한 응답시간을 제공한다는 발견을 확인했습니다. 이는 많은 비 wait-free stack과 queue가 리얼타임 용도에 적합할 수 있음을 의미합니다.

Quick Quiz 14.8: 하지만 fail-stop 버그에도 불구하고 올바른 운영을 하는게 가치 있는 fault-tolerance 속성 아닌가요?

실제 세계에서, 락킹은 이론적 염려에도 불구하고 리얼타임 프로그램에서 종종 사용됩니다. 그러나, 보다

¹⁰ Wait-free 알고리즘들은 모든 쓰레드가 유한한 시간 내에 진행을 만들어 냄을 보장하는 반면 lock-free 알고리즘은 최소 하나의 쓰레드는 유한시간 내에 진행을 만들어 냄만을 보장합니다. 더 자세한 내용을 위해선 Section 14.2을 참고하세요.

엄격한 제한 하에서, 락 기반의 알고리즘 또한 제한된 응답시간을 제공할 수 있습니다 [Bra11]. 이 제약은 다음을 포함합니다:

1. 공평한 스케줄러. 고정 우선순위 스케줄러의 공통적인 경우에서, 제한된 응답시간은 가장 높은 우선순위 쓰레드에게만 제공됩니다.
2. 해당 워크로드를 지원하기 충분한 대역폭. 이 제약은 지원하는 구현 규칙은 “일반적인 운영 중에는 모든 CPU 의 최소 50 % idle time 이 있어야 한다.” 또는, 보다 정형적으로, “제공되는 부하는 워크로드가 항상 스케줄 될 수 있을 것을 허용할 만큼 충분히 낮아야 한다.” 가 될 수 있겠습니다.
3. Fail-stop 버그의 부재.
4. 제한된 획득, 양도, 그리고 해제 응답시간을 제공하는 FIFO 락킹 기능들. 다시 말하지만, 우선순위 내에서 FIFO 인 락킹 기능들의 일반적인 경우에서 제한된 응답시간은 최고 우선순위 쓰레드에게만 제공됩니다.
5. 제한되지 않은 우선순위 역전의 방지. 이 챕터의 앞 부분에서 설명한 우선순위 한계 두기와 우선순위 상속이 충분할 겁니다.
6. 락 획득의 중첩 수준의 제한. 무한한 수의 락을 가질 수 있지만, 특정 쓰레드가 한번에 그 중 일부 이상은 (이상적으로는 하나만) 결코 획득하지 않을 때만 그렇습니다.
7. 제한된 수의 쓰레드. 앞의 제약들과 조합되어, 이 제약은 특정 락을 기다리는 쓰레드의 수는 제한되어 있을 것을 의미합니다.
8. 크리티컬 섹션에서 소요되는 시간의 제한. 특정 락을 기다리는 쓰레드의 수의 제한과 제한된 크리티컬 섹션 길이 때문에, 대기 시간 역시 제한됩니다.

Quick Quiz 14.9: 이 리스트 앞에 “포함”이라는 단어를 봤습니다. 다른 제약들도 있나요?

■
이 결과는 리얼타임 소프트웨어에서 사용하기 위한 풍부한 알고리즘과 데이터 구조를, 그리고 오랫동안 이어진 리얼타임 규칙을 검증할 기회를 가능하게 합니다.

물론, 주의 깊고 간단한 어플리케이션 설계도 굉장히 중요합니다. 세계 최고의 리얼타임 컴포넌트는 제대로 생각되지 않은 설계에서 만들어질 수 없습니다. 병렬 리얼타임 어플리케이션을 위해서, 동기화 오버헤드는 분명히 설계의 핵심 컴포넌트여야 합니다.

14.3.6.2 Polling-Loop Applications

많은 리얼타임 어플리케이션이 센서 데이터를 읽고 제어 범칙을 계산하고 제어 출력을 쓰는 하나의 CPU 위주 사용 반복문으로 구성됩니다. 센서 데이터를 제공하고 제어 출력을 취하는 하드웨어 레지스터가 어플리케이션의 주소공간에 매핑되어 있다면, 이 반복문은 시스템 콜로부터 완전히 자유로울 수도 있습니다. 하지만 스파이더맨 원칙을 기억하세요: 거대한 힘에는 거대한 책임이 따르며, 이 경우 그 책임은 하드웨어 레지스터에 잘못된 참조를 함으로써 하드웨어를 벽돌로 만드는 걸 막는 겁니다.

이 구성은 종종 기계 위에서 운영체제의 혜택 (또는 간섭) 없이 돌아갑니다. 그러나, 하드웨어 기능을 증가시키고 자동화의 수준을 높이는 것은 소프트웨어 기능을 높이는 동기가 되는데, 예를 들어 유저 인터페이스, 로깅, 레포팅 등, 모두 운영체제로부터 얻을 수 있는 것들입니다.

기계 위에서 돌아가는 혜택을 최대한 얻으면서도 범용 운영체제의 전체 기능에 접근할 수 있는 방법은 리눅스 커널의 NO_HZ_FULL 기능을 사용하는 것으로, Section 14.3.5.2 에 설명되어 있습니다.

14.3.6.3 Streaming Applications

비데이터 리얼타임 어플리케이션의 한가지 유형은 다양한 소스로부터 입력을 취하고 그걸 내부적으로 처리해서, 경고와 요약을 출력합니다. 이런 스트리밍 어플리케이션은 종종 고도로 병렬적이며, 동시에 다른 정보 소스를 처리합니다.

스트리밍 어플리케이션을 구현하는 한가지 접근법은 밀도 높은 순환형 FIFO 배열을 다른 처리 단계에 연결하는 겁니다 [Sut13]. 그런 FIFO 각각은 내부로 생산을 하는 하나의 쓰레드와 그걸 소비하는 (아마도 다른) 하나의 쓰레드를 갖습니다. Fan-in 과 fan-out 지점들은 데이터 구조보다 쓰레드를 사용하여, 여러 FIFO 의 출력이 병합되어야 한다면 별도의 쓰레드가 그것들로부터 입력을 받아 이 별도 쓰레드가 유일한 생산자가 되게끔 또 다른 FIFO 에 출력을 합니다. 비슷하게, 특정 FIFO 의 출력이 조개져야 한다면, 별도의 쓰레드가 이 FIFO로부터 입력을 받아와 여러 FIFO 로 필요한대로 출력을 냅니다.

이 방법은 제한적으로 보일 수도 있겠습니다만, 이는 쓰레드간의 통신을 취소한의 동기화 오버헤드를 가지고 가능하게 하며, 최소한의 동기화 오버헤드는 맞추기 쉽지 않은 응답시간 제약을 맞추기 위해 중요합니다. 이는 각 단계를 위한 처리양이 작아서 동기화 오버헤드가 처리 오버헤드에 비해 상당할 때 특히 사실입니다.

개별 쓰레드는 CPU 를 주로 사용할 수도 있는데, 이 경우라면 Section 14.3.6.2 의 조언이 적용됩니다. 다른

Listing 14.5: Timed-Wait Test Program

```

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 }

```

한편, 개별 쓰레드가 각자의 입력 FIFO로부터의 데이터를 기다리고 있다면, 다음 섹션의 조언이 적용됩니다.

14.3.6.4 Event-Driven Applications

중간 크기의 산업계 엔진으로 연료를 주입하는 것을 이벤트 주도 어플리케이션의 한 예로 들어 보겠습니다. 일반적인 운영 환경에서, 이 엔진은 연료가 상자점을 둘러싸 1도 간격으로 주입되게 해야 합니다. 1,500 RPM 회전율을 가정한다면 초당 25 회전 또는 초당 9,000 도 회전을 하는데, 이는 도당 111 마이크로세컨드를 의미합니다. 따라서 우린 이 연료 주입을 약 100 마이크로세컨드 시간 간격으로 스케줄 해야 합니다.

연료 주입을 주도하기 위해 시간 기반 대기가 사용되었고 여러분이 엔진을 만드는 중이라 하더라도, 전 회전 센서가 제공되길 바란다고 가정해 봅시다. 우린 이 시간 기반 대기 기능을 테스트 해야 하는데, Listing 14.5에 보인 것과 같은 테스트 프로그램을 사용해볼 수 있을 겁니다. 불행히도, 우리가 이 프로그램을 수행하면 우린 -rt 커널에서 조차 용납 불가능한 타이머 jitter를 얻게 됩니다.

한가지 문제는 CLOCK_REALTIME 으로, 이상하게 들리겠지만 이는 리얼타임에 사용되라고 만들어지지 않았습니다. 대신, 이것은 프로세스나 쓰레드에 의해 소요된 CPU 시간량에 반대되는 의미의 “리얼타임”을 의미합니다. 리얼타임 컴퓨팅에 사용되기 위해선 CLOCK_MONOTONIC 이 사용되어야 합니다. 그러나, 이 변경을 가지고서도 결과는 여전히 충분치 않습니다.

또 다른 문제는 이 쓰레드가 sched_setscheduler() 시스템콜을 사용해 리얼타임 우선순위가 되어야만 한다는 겁니다. 그러나 우린 여전히 페이지풀트를 보게 되므로 이 변화만으로는 불충분합니다. 우린 또한 페이지풀트를 막기 위해 mlockall() 시스템 콜을 사용해 이 어플리케이션의 메모리를 고정시켜야 합니다. 이 모든 변화를 가지고서야 결과는 마침내 받아들여질 수도 있습니다.

다른 상황에서는 추가적인 변경이 필요할 수도 있습니다. 시간이 치명적인 쓰레드를 각자의 CPU에 고정시켜야 할 수도 있고, 인터럽트는 그 CPU를 외의 것에

고정시켜야 할 수도 있습니다. 하드웨어와 드라이버를 주의 깊게 선택해야 하고 커널 설정을 주의 깊게 선택해야 할 수도 있습니다.

이 예에서 볼 수 있듯, 리얼타임 컴퓨팅은 상당히 무자비할 수 있습니다.

14.3.6.5 The Role of RCU

기온, 습도, 그리고 기압 등의 변화 등에 의해 점진적으로 변화할 수 있는 데이터로의 접근이 필요한 병렬 리얼타임 어플리케이션을 작성한다고 생각해 봅시다. 이 프로그램에 대한 리얼타임 응답 제약은 스피닝이나 블로킹이 허용되지 않을 정도로 가혹해서, 락킹도 쓸 수 없고, 재시도 반복문도 허용되지 않으므로 시퀀스 락킹이나 해저드 포인터도 쓸 수 없습니다. 다행히도, 기온과 기압은 일반적으로 제어될 수 있으므로, 하드코딩된 기본값 데이터가 보통 충분합니다.

그러나, 기온, 습도, 그리고 기압은 가끔 기본값으로부터 너무 멀리 변화될 수 있으며, 그런 상황에서는 기본값을 대체하는 데이터가 제공되어야 합니다. 기온, 습도, 그리고 기압은 점진적으로 변화하므로, 업데이트된 값을 제공하는 건 수분 내에는 일어나야만 하지만 급한 건 아닙니다. 이 프로그램은 상상컨대 cur_cal 이라 이름 지어진, 정적으로 할당되고 초기화 되었으며 a, b, 그리고 c 라 이름지어진 필드에 기본 조정값을 담고 있는 default_cal 을 참조하는 전역 포인터를 사용할 수 있을 겁니다. 그렇지 않다면, cur_cal 은 현재 조정값을 제공하는 동적으로 할당된 구조체를 가리킵니다.

Listing 14.6 는 RCU 가 이 문제를 해결하기 위해 어떻게 사용될 수 있는지 보입니다. 탐색은 라인 9-15 의 calc_control() 에 보인 것과 같이 소요시간이 결정되어 있으며 리얼타임 요구사항과 일관되어 있습니다. 업데이트는 더 복잡한데, 라인 17-35 의 update_cal() 에 보인 것과 같습니다.

Quick Quiz 14.10: 리얼타임 시스템은 종종 안전에 치명적인 어플리케이션을 위해 사용된다는 점, 그리고 수행시간 메모리 할당은 많은 안전에 치명적인 상황에서 금지되었다는 점을 놓고 볼 때, malloc() 은 뭘 위한거죠???



Quick Quiz 14.11: update_cal() 을 보호하기 위한 어떤 동기화가 필요하지 않나요?



이 예제는 어떻게 RCU 가 리얼타임 프로그램에 소요 시간이 결정된 읽기쪽 데이터 구조 액세스를 제공할 수 있는지 보입니다.

Listing 14.6: Real-Time Calibration Using RCU

```

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 14.15:** The Dark Side of Real-Time Computing

다른 한편, 리얼타임 시스템이 필요한데 사용하지 못하는 것 역시 문제를 일으킬 수 있는데, Figure 14.16에 보인 것과 같습니다. 이는 여러분이 여러분의 상사에게 미안함을 느끼게 하기 거의 충분합니다!

**Figure 14.16:** The Dark Side of Real-Fast Computing

경험적 법칙 하나는 다음 네가지 질문이 여러분의 선택을 도울 거라는 겁니다:

1. 장기간 평균 처리량이 유일한 목표인가?
2. 큰 부하가 응답시간을 악화시키는게 허용되는가?
3. 메모리가 부족해서 `mlockall()` 시스템콜의 사용이 불허되는가?
4. 어플리케이션의 기본 처리 항목이 완료되는데 100 밀리세컨드 이상을 소요하는가?

리얼타임과 정말 빠른 컴퓨팅 사이의 선택은 어려울 수 있습니다. 리얼타임 시스템은 비 리얼타임 컴퓨팅에 비해 처리량에 페널티를 종종 추가하므로, 필요하지 않은 곳에 리얼타임을 사용하는 것은 Figure 14.15에 보인 것처럼 현명치 못한 선택일 수 있습니다.

이 질문들 중 하나라도 그에 대한 답이 “그렇다”라면, 여러분은 리얼타임이 아니라 정말 빠른 시스템을 선택해야 하며, 그렇지 않다면 리얼타임이 여러분에게 맞을 수도 있습니다.

현명하게 선택하시고, 여러분이 리얼타임을 선택했다면, 여러분의 하드웨어, 펌웨어, 그리고 운영체제가 여러분의 일을 위해 잘 선택되었음을 분명히 하세요!

The art of progress is to preserve order amid change
and to preserve change amid order.

Chapter 15

Alfred North Whitehead

Advanced Synchronization: Memory Ordering

인과관계와 순서관계는 매우 직관적이며, 해커들은 종종 이 컨셉에 사로잡혀 있습니다. 이 직관은 순차적 코드만이 아니라 락킹과 같이 표준적인 상호 배제 메커니즘을 사용하는 병렬 코드를 작성하고 분석하고 디버깅할 때에 상당히 도움이 됩니다. 불행히도, 이 직관은 그런 메커니즘을 사용할 수 없는 코드에서는 완전히 망가집니다. 그런 코드 구현의 한가지 예는 표준 상호 배제 메커니즘 그 자체이며, 다른 예는 완화된 동기화를 사용하는 빠른 수행경로 구현들입니다. 직관에 대한 모욕에도 불구하고, 어떤 사람들은 그런 완화됨이 덕목이라고 주장합니다 [Alg13]. 덕목이든 악덕이든, 이 챕터는 여러분이 메모리 순서 규칙에 대한 이해를 도울 텐데 이는 연습과 함께 동기화 기능과 성능에 치명적인 빠른 수행경로를 구현하는데 충분할 겁니다.

Section 15.1은 실제 컴퓨터 시스템이 메모리 참조를 재배치 할 수 있음을 보이고, 왜 그러는지 이유를 보이며, 바라지 않는 재배치를 어떻게 막는지에 대한 정보를 제공합니다. Sections 15.2 and 15.3은 그걸 모르는 병렬 프로그래머를 고통스럽게 할 수 있는 하드웨어와 컴파일러에서 발생하는 문제들을 각각 다룹니다. Section 15.4는 메모리 순서 규칙을 높은 수준의 추상화로 모델링 하는 것의 장점을 봅니다. Section 15.5는 일부 하드웨어 플랫폼에서의 본 1자세한 부분을 다룹니다. 마지막으로, Section 15.6는 일부 유용한 경험적 법칙을 제공합니다.

Quick Quiz 15.1: 이 챕터는 첫번째 판본 이후로 다시 작성되었습니다. 메모리 순서 규칙은 2014년 이후로 그럴거나 바뀌었나요?

15.1 Ordering: Why and How?

Nothing is orderly till people take hold of it.
Everything in creation lies around loose.

Henry Ward Beecher, updated

메모리 순서 규칙을 위한 한가지 동기는 Listing 15.1 (C-SB+o-o+o-o.litmus)의 사소해 보이는 리트머스 테스트에서 볼 수 있는데, 첫눈에 보면 `exists` 절이 절대 발동되지 않을 것으로 보일 수도 있을 겁니다.¹ 어쨌건, `exists` 절에 보인 것처럼 `0:r2=0`라면,² 우린 쓰레드 `P0()`의 `x1`으로부터 `r2`로의 로드가 쓰레드 `P1()`의 `x1`으로의 스토어 전에 일어났을 것을 바랄 수도 있을 텐데, 이는 더 나아가 쓰레드 `P1()`의 `x0`로부터 `r2`로의 로드가 쓰레드 `P0()`의 `x0`로의 스토어 전에 이루어져서 `1:r2=2`이길, 따라서 `exists` 절이 발동되지 않길 바랄 수도 있습니다. 이 예는 대칭적이며, 따라서 `1:r2=0` 가 `0:r2=2`를 보장하길 바랄 수도 있을 겁니다. 불행히도, 메모리 배리어의 부재가 이 희망을 깨부릅니다. CPU는 쓰레드 `P0()`와 쓰레드 `P1()`의 문장들을 재배치할 권리를 갖는데, x86처럼 상대적으로 강한 순서 규칙의 시스템에서도 그렇습니다.

Quick Quiz 15.2: 컴파일러 또한 Listing 15.1의 쓰레드 `P0()`와 쓰레드 `P1()`의 메모리 액세스를 재배치 할 수 있습니다, 그렇죠?

■ 이 재배치에 대한 의지는 litmus7 [AMT14]와 같은 도구를 이용해 확인할 수 있는데, 이 도구는 이 반직

¹ 순수주의자들은 `exists` 절이 절대 만족 되지 ◊ 넣는다고 주장 하겠지만, 여기서의 “발동”은 단정을 은유합니다.

² 즉, 쓰레드 `P0()`의 지역 변수 `r2`의 인스턴스가 0이란 이야기입니다. 리트머스 테스트 명명법을 위해선 Section 12.2.1를 참고하시기 바랍니다.

Listing 15.1: Memory Misordering: Store-Buffering Litmus

```

Test
1 C C-SB+o-o+o-o
2
3 {}
4
5 P0(int *x0, int *x1)
6 {
7     int r2;
8
9     WRITE_ONCE(*x0, 2);
10    r2 = READ_ONCE(*x1);
11 }
12
13 P1(int *x0, int *x1)
14 {
15     int r2;
16
17     WRITE_ONCE(*x1, 2);
18     r2 = READ_ONCE(*x0);
19 }
20
21 exists (1:r2=0 /\ 0:r2=0)

```

관적인 순서가 제 x86 랩톱에서 100,000,000 시도 중 314회 발생함을 보였습니다. 충분히 이상하게도, 두개의 로드가 값 2를 반환하는 완전히 합법적인 결과는 덜 빈번하게 발생했는데, 이 경우 오직 167회였습니다.³ 여기서의 교훈은 분명합니다: 반직관성의 증가는 확률의 감소를 암시하지는 않습니다!

이어지는 섹션들은 이 직관이 정확히 어디서 부서지는지 보이고, 여러분이 이 함정을 피하는 걸 도울 메모리 순서 규칙에 대한 마음자세를 알려줍니다.

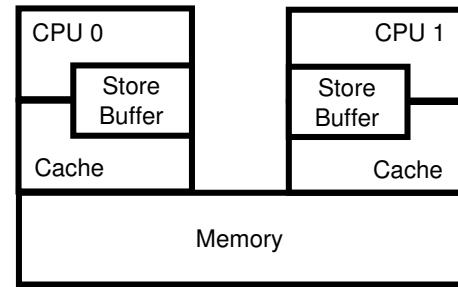
Section 15.1.1 는 왜 하드웨어가 메모리 액세스를 잘못 순서잡는지 간략히 설명하고, Section 15.1.2 는 여러분이 그런 잘못된 순서잡기를 어떻게 방해할 수 있는지 짧게 설명합니다. 마지막으로, Section 15.1.3 는 기본적인 경험적 규칙 몇 가지를 나열하는데, 이는 뒤의 섹션들에서 더 정리될 겁니다. 이 섹션은 하드웨어 순서 재배치에만 집중합니다만, 컴파일러가 하드웨어가 여지껏 꿈꿔왔던 것보다 훨씬 더 적극적으로 재배치를 함이 분명시 됩니다. 하지만 그 주제는 나중에 Section 15.3에서 다룰 겁니다.

15.1.1 Why Hardware Misordering?

하지만 일단 왜 잘못된 메모리 순서잡기가 일어나는 걸까요? CPU는 스스로 순서를 추적할 수 없나요? 무언가를 추적하는 것, 그게 어쨌건 우리가 컴퓨터를 갖는 이유 아닙니까?

실제로 많은 사람들이 그들의 컴퓨터가 사물을 추적하고 있을 것을 기대합니다만, 또한 많은 사람들은 사물을 빠르게 추적할 것을 강요합니다. 그러나, Chapter 3

³ 결과는 정확한 하드웨어 구성, 시스템이 얼마나 부하를 받았는지, 그외에도 여러가지에 따라 민감하게 달라질 수 있음에 주의하십시오.

**Figure 15.1:** System Architecture With Store Buffers

에서 보았듯이, 메인메모리는 메모리로부터 하나의 변수를 읽어오는데 필요한 시간동안 수백개의 명령을 수행할 수 있는 현대의 CPU 속도를 따라가지 못합니다. 따라서 CPU들은 계속해서 커져가는 캐쉬를 사용하는데, Figure 3.10에서 본 것과 같으며, 이는 특정 CPU에 의한 특정 변수의 첫번째 읽기는 Section 3.1.5에서 본 것처럼 비싼 캐쉬 미스를 일으키지만 해당 CPU에서의 뒤따르는 그 변수에 대한 반복된 읽기는 최초의 캐쉬 미스가 그 변수를 그 CPU의 캐쉬에 적재시켰으므로 매우 빠르게 처리될 겁니다.

하지만, 여러 CPU로부터의 공유된 변수들로의 빈번한 동시의 스토어를 처리할 필요 또한 생깁니다. 일관된 캐쉬 시스템에서는 캐쉬들이 특정 변수에 대한 여러 복사본을 쥐고 있다면 그 변수에 대한 모든 복사본이 같은 값을 가져야만 합니다. 이는 동시의 로드를 위해선 굉장히 잘 동작하지만 동시의 스토어에 대해서는 그렇지 않습니다: 각 스토어는 기존 값을 갖는 모든 복사본에 대해 뭔가를 해야만 하는데 (또다른 캐쉬 미스!), 유한한 빛의 속도와 원자적 물질의 본성을 생각하면, 성격 급한 소프트웨어 해커가 원하는 것보다는 더 느릴 겁니다.

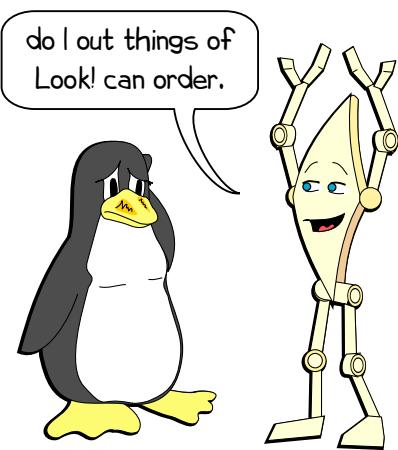
따라서 CPU들은 Figure 15.1에 보인 것처럼 스토어 버퍼를 사용합니다. 특정 CPU가 해당 CPU의 캐쉬에 존재하지 않는 변수에 스토어를 할 때, 그 새 값을 해당 CPU의 스토어 버퍼에 위치해집니다. 그러면 CPU는 이 스토어가 다른 CPU들의 캐쉬에 있는 변수들의 기존 값을 모두와 뭔가를 할동안 기다리지 않고 곧바로 다음 명령을 수행할 수 있습니다.

스토어 버퍼가 성능을 크게 향상시킬 수 있지만, 이는 명령들과 메모리 참조가 알맞지 않은 순서로 수행될 수 있게 하는데, 이는 결국 Figure 15.2에 보인 것처럼 상당한 혼란을 야기할 수 있습니다.

특히, 스토어 버퍼는 Listing 15.1에 보인 것과 같은 메모리 순서 잘못잡기를 일으킵니다. Table 15.1 가 이 잘못된 순서잡기를 일으키는 단계를 보입니다. 첫번째 열은 최초 상태를 보이는데, CPU 0은 자신의 캐쉬에 x1을 갖고 CPU 1은 자신의 캐쉬에 x0를 갖는데 두

Table 15.1: Memory Misordering: Store-Buffering Sequence of Events

CPU 0			CPU 1		
Instruction	Store Buffer	Cache	Instruction	Store Buffer	Cache
1 (Initial state)		x1==0	(Initial state)		x0==0
2 x0 = 2;	x0==2	x1==0	x1 = 2;	x1==2	x0==0
3 r2 = x1; (0)	x0==2	x1==0	r2 = x0; (0)	x1==2	x0==0
4 (Read-invalidate)	x0==2	x0==0	(Read-invalidate)	x1==2	x1==0
5 (Finish store)		x0==2	(Finish store)		x1==2

**Figure 15.2:** CPUs Can Do Things Out of Order

변수 모두 값 0을 갖습니다. 두번째 열은 각 CPU의 스토어 (Listing 15.1의 라인 9와 17)로 인한 상태 변화를 보입니다. 어느 CPU도 캐쉬 내에 스토어의 대상 변수를 갖지 않으므로, 두 CPU는 각자의 스토어를 각자의 스토어 버퍼에 저장해 둡니다.

Quick Quiz 15.3: 기다려요!!! Table 15.1의 두번째 열에서 x0 와 x1 둘 다 동시에 두 값, 즉 0 과 2를 갖습니다. 이게 어떻게 동작할 수 있죠???

세번째 열은 두개의 로드 (Listing 15.1의 라인 10와 18)을 보입니다. 각 CPU에 의해 읽혀지는 변수는 각 CPU의 캐쉬에 있어서, 각 로드는 즉시 캐쉬 내의 값을 리턴하는데, 그 값은 두 경우 모두 0입니다.

하지만 CPU의 일은 끝나지 않았습니다: 금방이든 나중이든, 그것들은 각자의 스토어 버퍼를 비워야만 합니다. 캐쉬는 캐쉬라인이라 불리는 상대적으로 큰 블록 단위로 데이터를 옮기므로, 그리고 각 캐쉬라인은 여러 변수를 담을 수 있으므로, 각 CPU는 자신의 스토어 버퍼 내에 있는 변수에 연관된 캐쉬라인의 부분을 업데이트

할 수 있게끔, 그러나 해당 캐쉬라인의 다른 부분을 망치지 않게끔 자신의 캐쉬에 해당 캐쉬라인을 가져야만 합니다. 각 CPU는 또한 해당 캐쉬라인이 다른 CPU의 캐쉬에 존재하지 않음을 보장해야하는데, 이를 위해 읽기 무효화 (read invalidation) 오퍼레이션이 사용됩니다. 네번째 열에서 보인 것처럼, 두 읽기 무효화 오퍼레이션이 완료된 후, 두 CPU는 캐쉬라인을 주고받아서, CPU 0의 캐쉬는 이제 x0를 가지고 있고 CPU 1의 캐쉬는 x1을 가지고 있게 됩니다. 이 두 변수가 새 집에 일단 도착하면, 각 CPU는 각자의 스토어 버퍼를 연관된 캐쉬라인으로 비워낼 수 있어서, 각 변수를 다섯번째 열에 보인 최종 값으로 만듭니다.

Quick Quiz 15.4: 하지만 그 값들은 캐쉬에서 메인 메모리로도 비워져야 하지 않나요?

■ 요약하자면, 스토어 버퍼는 CPU들이 스토어 명령을 효율적으로 처리할 수 있게 하기 위해 필요하지만, 그게 반직관적인 메모리 순서 잘못잡기를 초래할 수 있습니다.

하지만 여러분의 알고리즘이 정말로 메모리 참조가 순서잡하기를 필요로 한다면 뭘 할겁니까? 예를 들어, 여러분이 하나는 드라이버가 수행 중인지 말하고 다른 하나는 그 드라이버를 위해 대기중인 요청이 있는지 말하는 두개의 플래그를 사용해 드라이버와 통신하려 한다고 생각해 봅시다. 요청자는 요청 대기 플래그를 먼저 올리고, 이어서 드라이버 수행중 플래그를 검사해야 하며, 필요하면 그 드라이버를 깨워야 합니다. 드라이버는 모든 대기중 요청을 처리하고 나면, 드라이버 수행중 플래그를 내리고 재시작 되어야 하는지 보기 위해 요청 대기중 플래그를 검사해야 합니다. 이 매우 합리적인 방법은 하드웨어가 스토어와 로드를 순서대로 처리함을 어떻게든 보장하지 않고서는 동작하지 않습니다. 이게 다음 섹션의 주제입니다.

15.1.2 How to Force Ordering?

메모리 배리어(예를 들어, 리눅스 커널의 `smp_mb()`)를 사용해 순서의 환상을 지키는데 필요한 컴파일러 지시

Listing 15.2: Memory Ordering: Store-Buffering Litmus Test

```

1 C C-SB+o-mb-o+o-mb-o
2
3 {}
4
5 P0(int *x0, int *x1)
6 {
7     int r2;
8
9     WRITE_ONCE(*x0, 2);
10    smp_mb();
11    r2 = READ_ONCE(*x1);
12 }
13
14 P1(int *x0, int *x1)
15 {
16     int r2;
17
18     WRITE_ONCE(*x1, 2);
19     smp_mb();
20     r2 = READ_ONCE(*x0);
21 }
22
23 exists (1:r2=0 /\ 0:r2=0)

```

어와 동기화 기능 (예를 들어 락킹과 RCU) 이 존재함이 드러났습니다. 이 메모리 배리어들은 Arm, POWER, Itanium, 그리고 Alpha에서와 같이 명시적 명령일 수 있고, x86에서 종종 그런 것처럼 다른 명령에 내포될 수도 있습니다. 이런 표준 동기화 기능은 순서의 환상을 지키므로, 여러분이 가장 쉽게 일을 끝내는 방법은 그저 이 기능들을 사용하고 이 섹션은 그만 읽는 겁니다.

그러나, 여러분이 그 동기화 기능들 자체를 구현해야 한다면, 또는 여러분이 메모리 순서잡기가 어떻게 동작하는지 이해하는데 관심 있다면, 계속 읽으세요! 이 여정의 첫번째 정류장은 Listing 15.2 (C-SB+o-mb-o+o-mb-o.litmus) 인데, 여기선 P0() 와 P1() 의 스토어와 로드 사이에 리눅스 커널 전체 메모리 배리어인 smp_mb() 를 놓는다는 점 외에는 Listing 15.1 와 동일합니다. 이 배리어들은 제 x86 랩톱에서 100,000,000 회 시도 끝에 일어나는 반직관적 결과를 막습니다. 흥미롭게도, 이 배리어들로 인해 더해진 오버헤드는 두 로드가 값 2를 리턴하는 올바른 결과를 800,000 회 넘게 일어나게 하는데, 배리어가 없는 Listing 15.1에서의 167 회와 상당히 다른 결과입니다.

이 배리어들은 순서 규칙에 중요한 효과를 일으키는데, Table 15.2에 보인 것과 같습니다. 처음 두 열은 Table 15.1에 보인 것과 같지만, 그리고 세번째 열의 smp_mb() 명령은 상태를 바꾸지 않지만, 그것들은 스토어들이 (네번째와 다섯번째 열) 로드 (여섯번째 열) 이전에 완료되게 해서 Table 15.1에 보인 반직관적 결과를 막습니다. 변수 x0 와 x1은 여전히 두번째 열에서 두개 이상의 값을 갖지만 앞서 약속된대로, smp_mb() 호출은 결과적으로 일을 간단하게 만들어줌을 유의하시기 바랍니다.

smp_mb() 와 같은 전체 배리어가 굉장히 강한 순서 보장을 제공하지만, 그 위력은 하드웨어와 컴파일러 최적화 측면에서는 높은 비용으로 제공됩니다. 매우 많은 상황이 훨씬 저렴한 메모리 순서규칙 명령을 사용하는, 또는 아예 메모리 순서 강제 명령을 사용하지 않는 훨씬 완화된 순서 규칙 보장으로 처리될 수 있습니다.

Table 15.3는 리눅스 커널의 순서규칙 기능과 그것들의 보장사항에 대한 요약입니다. 각 열은 순서 규칙을 제공할 수도 안할 수도 있는 기능 또는 기능들의 카테고리를 보이며, “Prior Ordered Operation”과 “Subsequent Ordered Operation”로 라벨링 된 행은 그에 대해 순서가 잡힐 수도 또는 안잡힐 수도 있는 오퍼레이션들을 보입니다. “Y”를 포함하는 칸은 무조건적으로 순서 규칙이 제공됨을 나타내며 다른 글자들은 순서규칙이 부분적으로 또는 조건적으로 제공됨을 나타냅니다. 빈 칸은 순서규칙이 제공되지 않음을 나타냅니다.

“Store” 열은 atomic RMW 오퍼레이션의 스토어 부분도 포함합니다. 그에 더해, “Load” 열은 성공한 값을 반환하는 _relaxed() RMW 어토믹 오퍼레이션의 로드 부분을 포함하는데 결합된 “_relaxed() RMW operation” 열은 값을 반환하는 경우에 대한 간편한 결합된 레퍼런스를 제공합니다. 성공하지 못한 값을 반환하는 atomic RMW 오퍼레이션을 수행하는 CPU는 연관된 변수를 모든 다른 CPU의 캐쉬에서 무효화 시켜야만 합니다. 따라서, 성공하지 못한 값을 반환하는 atomic RMW 오퍼레이션은 스토어의 속성을 많이 갖게 되는데, 이는 “_relaxed() RMW operation” 열은 성공하지 못한 값을 반환하는 atomic RMW 오퍼레이션에도 적용됨을 의미합니다.

*_acquire 열은 smp_load_acquire(), cmpxchg_acquire(), xchg_acquire(), 등의 것을 포함합니다; *_release 열은 smp_store_release(), rCU_assign_pointer(), cmpxchg_release(), xchg_release() 등을 포함합니다; “Successful full-strength non-void RMW” 열은 atomic_add_return(), atomic_add_unless(), atomic_dec_and_test(), cmpxchg(), xchg() 등을 포함합니다. “Successful” 이란 단어는 atomic_add_unless(), cmpxchg_acquire(), 그리고 cmpxchg_release() 같은 실패를 피악했을 때에는 메모리나 순서에 어떤 영향도 끼치지 않는 기능들에 적용되는데, 앞의 “_relaxed() RMW operation” 열에서 나타내진 것과 같습니다.

“C” 행은 누적성과 전파성을 나타내는데, Sections 15.2.7.1 and 15.2.7.2에서 설명된 바와 같습니다. 최대 두개의 쓰레드만 관여될 때에는 이 행은 무시될 수 있습니다.

Table 15.2: Memory Ordering: Store-Buffering Sequence of Events

CPU 0			CPU 1		
Instruction	Store Buffer	Cache	Instruction	Store Buffer	Cache
1 (Initial state)		x1==0	(Initial state)		x0==0
2 x0 = 2; x0==2	x0==2	x1==0	x1 = 2; x1==2	x1==2	x0==0
3 smp_mb(); x0==2	x0==2	x1==0	smp_mb(); x1==2	x1==2	x0==0
4 (Read-invalidate) x0==2	x0==2	x0==0	(Read-invalidate) x1==2	x1==2	x1==0
5 (Finish store)		x0==2	(Finish store)		x1==2
6 r2 = x1; (2) x1==2	x1==2	r2 = x0; (2)			x0==2

Table 15.3: Linux-Kernel Memory-Ordering Cheat Sheet

Operation Providing Ordering	C	Prior Ordered Operation				Subsequent Ordered Operation					
		Self	R	W	RMW	Self	R	W	DR	DW	RMW
Store, for example, WRITE_ONCE()		Y									Y
Load, for example, READ_ONCE()		Y							Y	Y	Y
_relaxed() RMW operation		Y							Y	Y	Y
*_dereference()		Y							Y	Y	Y
Successful *_acquire()		R				Y	Y	Y	Y	Y	Y
Successful *_release()	C	Y	Y	Y	W						Y
smp_rmb()		Y		R		Y		Y			R
smp_wmb()			Y	W			Y		Y	W	
smp_mb() and synchronize_rcu()	CP	Y	Y	Y		Y	Y	Y	Y	Y	Y
Successful full-strength non-void RMW	CP	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
smp_mb__before_atomic()	CP	Y	Y	Y		a	a	a	a		Y
smp_mb__after_atomic()	CP	a	a	Y		Y	Y	Y	Y	Y	

Key:	C: Ordering is cumulative
	P: Ordering propagates
	R: Read, for example, READ_ONCE(), or read portion of RMW
	W: Write, for example, WRITE_ONCE(), or write portion of RMW
	Y: Provides the specified ordering
	a: Provides specified ordering given intervening RMW atomic operation
	DR: Dependent read (address dependency, Section 15.2.3)
	DW: Dependent write (address, data, or control dependency, Sections 15.2.3–15.2.5)
	RMW: Atomic read-modify-write operation
	Self: Orders self, as opposed to accesses both before and after
	SV: Orders later accesses to the same variable

Applies to Linux kernel v4.15 and later.

Quick Quiz 15.5: Table 15.3 의 열들은 꽤 무작위적이고 혼란스럽게 느껴지네요. 이 표의 개념적 기반이 있다면 무엇이죠???

Quick Quiz 15.6: 왜 Table 15.3 는 `smp_mb__after_unlock_lock()` 과 `smp_mb__after_spinlock()` 을 포함하고 있지 않죠?

이 표는 요약일 뿐이며, 따라서 메모리 순서 규칙에 대한 충분한 이해를 대체할 수 없습니다. 그런 이해를 얻기 위해서 다음 섹션이 기본적인 경험적 법칙들을 일부 제공합니다.

15.1.3 Basic Rules of Thumb

이 섹션은 매우 많은 상황에 “좋고 충분한” 일부 기본적 경험적 법칙을 제공합니다. 실제로 여러분은 이 경험적 법칙 외에는 별다른 것 없이도 훌륭한 성능과 확장성을 갖는 동시성 코드를 작성할 수 있습니다. 더 잘 정리된 경험적 법칙들은 Section 15.6 에서 제공됩니다.

Quick Quiz 15.7: 하지만 특정 프로젝트가 이 경험적 법칙에 근거해 설계되고 작성될 수 있는지는 어떻게 하나요?

쓰레드는 자신의 액세스는 순서대로 보게 된다. 이 규칙은 공유 변수로의 로드와 스토어가 각각 `READ_ONCE()` 과 `WRITE_ONCE()` 를 사용한다고 가정합니다. 그렇지 않다면, 컴파일러는 기본적으로 여러분의 코드를 휘저어 버릴 수 있으며⁴ 가끔은 CPU 역시 Section 15.5.4 에서 이야기된 것처럼 약간 일을 허트러트릴 수 있습니다.

순서 규칙은 조건적 if-then 규칙을 갖습니다. Figure 15.3 가 이 메모리 배리어를 보입니다. 두 메모리 배리어가 충분히 강력하다고 가정하고, CPU 1 의 Y1 액세스가 CPU 0 의 액세스 Y0 후에 이루어졌다면, CPU 1 의 액세스 X1 은 CPU 0 의 X0 액세스 후에 일어날 것이 보장됩니다. 어떤 메모리 배리어가 충분히 강력한지 모르겠다면, `smp_mb()` 가 비용이 높긴 하지만 항상 필요한 일을 해줍니다.

Quick Quiz 15.8: 특정 사용처에 있어 어떤 메모리 배리어가 충분히 강력한지 어떻게 알 수 있죠?

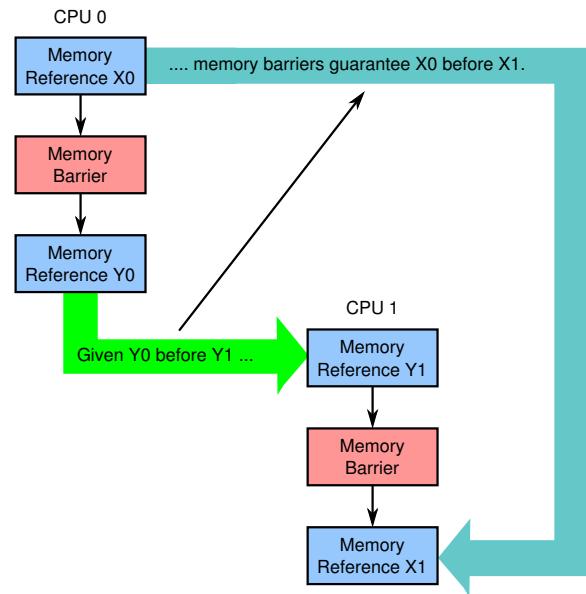


Figure 15.3: Memory Barriers Provide Conditional If-Then Ordering

Listing 15.2 이 그런 케이스입니다. 라인 10 와 19 에서의 `smp_mb()` 가 그 배리어로 동작하여, 라인 9에서의 `x0` 로의 스토어는 `X0` 이고, 라인 18에서의 `x1` 으로의 스토어는 `Y1` 이며, 라인 20에서의 `x0` 로드는 `X1` 입니다. 이 if-then 규칙을 단계별로 적용하면, 우린 라인 18에서의 `x1` 스토어가 `P0()` 의 지역변수 `r2` 가 값 0으로 되어 있다면 라인 11에서의 `x1` 로드보다 나중에 벌어졌음을 알게 됩니다. 그러면 이 if-then 규칙은 라인 20에서의 `x0` 로드는 라인 9에서의 `x0` 스토어보다 나중에 벌어질 거라 말합니다. 달리 말하자면 `P1()` 의 지역변수 `r2` 는 `P0()` 의 지역변수 `r2` 가 값 0으로 끝날 때에만 값 2로 끝납니다. 이는 메모리 순서 규칙 보장이 절대적이 아니라 조건적이라는 지점을 강조합니다.

비록 Figure 15.3 이 특별히 메모리 배리어를 언급하지만, 이 동일한 if-then 규칙이 리눅스 커널의 순서규칙 오퍼레이션 나머지에도 적용됩니다.

순서규칙 오퍼레이션은 반드시 짹을 이루어야 합니다. 여러분이 한 쓰레드에서의 오퍼레이션을 주의깊게 순서 맞췄지만 다른 쓰레드에서는 그렇게 하지 못했다면, 거기엔 순서가 없습니다. 이 if-then 규칙이 적용되게 하려면 두 쓰레드 모두 순서를 제공해야 합니다.⁵

순서규칙 오퍼레이션은 거의 항상 무언가를 빠르게 만들지 않습니다. 앞의 스토어가 메모리에 더 빠르게 비

⁴ 많은 컴파일러 작성자들은 이를 “최적화” 라 부르길 선호합니다.

⁵ Section 15.2.7.2 에서 짹 맞추기는 사이클로 일반화 될 겁니다.

위지게 하려고 메모리 배리어를 더하고 싶어진다면, 저항하세요! 순서규칙을 더하는건 일반적으로 속도를 느리게 만듭니다. 물론, page 157 의 Figure 9.22 에서 보았듯 명령을 더하는게 속도를 높이는 경우도 있습니다만, 그런 경우에는 주의 깊은 벤치마킹이 필요합니다. 그리고 그럴때 조차도, 여러분의 시스템에서는 속도를 약간 향상시켰더라도, 여러분의 사용자들의 시스템에서는 상당한 속도 저하를 일으켰을 수도 있습니다. 또는 여러분의 미래의 시스템에서도요.

순서규칙 오퍼레이션은 마법이 아닙니다. 여러분의 프로그램이 어떤 경주 조건 때문에 실패한다면, 여러분은 종종 버그를 제거하기 위해 메모리 순서규칙 오퍼레이션을 몇개 집어넣어지고 싶을 겁니다. 그보다 훨씬 나은 반응은 더 높은 단계의 기능들을 주의깊게 설계된 형식으로 사용하는 겁니다. 동시성 프로그래밍에 있어서, 버그가 존재하지 않게끔 설계를 하는게 버그 확률을 낮추기 위해 해킹을 하는 것보다 거의 항상 낮습니다.

오직 대략적 경험적 법칙만이 존재합니다. 이 경험적 법칙들이 다른 경험적 법칙들과 마찬가지로 상당한 실제 상황에서 사용될 수 있지만 여기에도 한계가 있습니다. 다음 섹션은 여러분의 이해를 높이는 반면 여러분의 직관을 괴롭히려는 의도로 만들어진 리트머스 테스트를 소개함으로써 이런 한계를 보일 겁니다. 이 리트머스 테스트는 Table 15.3 에서 보인 리눅스 커널 메모리 순서규칙 요약본에 의해 설명된 많은 개념을 명확히 할 것이며, 올바른 도구들을 사용하면 [AMM⁺18] 자동으로 분석될 수 있습니다. Section 15.6 는 이 요약본으로 되돌아와서 이 모든 사이에 있는 트릭과 함정들로부터의 배움에서 나오는 보다 잘 정리된 경험적 법칙들을 선보입니다.

Quick Quiz 15.9: 기다려요!!! 리트머스 테스트를 자동으로 분석하는 도구를 어디서 구할 수 있나요???



15.2 Tricks and Traps

Knowing where the trap is—that's the first step in evading it.

Duke Leto Atreides, “Dune”, Frank Herbert

이제 여러분은 하드웨어가 메모리 액세스 순서를 바꿀 수 있으며 여러분은 그걸 막을 수 있음을 알게 되었으니, 다음 단계는 여러분의 직관에 문제가 있음을 인정하는 겁니다. 이 고통스런 작업은 동시에 여러 값을 가질 수 있는 scalar 변수들을 보이는 코드를 제공할 Section 15.2.1,

Listing 15.3: Software Logic Analyzer

```

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 }

```

그리고 직관적으로는 올바르지만 실제 하드웨어에서는 차참하게 실패하는 코드를 보일 Sections 15.2.2 부터 15.2.7

에서 행해집니다. 이 과정을 통해 일단 여러분의 직관이 깨우쳐지면, 다음 섹션들은 메모리 순서 규칙이 따르는 기본 규칙들을 요약합니다.

하지만 먼저, 하나의 변수가 한 시점에 얼마나 많은 값을 가질 수 있는지 봅시다.

15.2.1 Variables With Multiple Values

한 변수가 잘 정의된 값의 연속을 잘 정의된 전역적 순서로 가질거라 생각하는건 자연스러운 일입니다. 불행히도, 이 여성의 다음 단계는 이 안락한 환상에 “안녕”이라고 말합니다. 바라건대, 이미 여러분은 Tables 15.1 and 15.2 의 두번째 열을 통해 이미 “안녕”이라고 말하기 시작했을 수도 있고, 그렇다면 이 섹션의 목적은 이 요점을 집으로 가져가는 것입니다.

이를 위해, Listing 15.3 에 보인 코드 조각을 봅시다. 이 코드 조각은 여러 CPU 들에 의해 병렬적으로 수행됩니다. 라인 1 는 현재 CPU 의 ID로 공유 변수를 설정하며, 라인 2 는 모든 CPU 들 사이에서 동기화 되는(불행히도 모든 CPU 군에서 가능한 일은 아닙니다!) 세밀하게 조정되는 하드웨어 “timebase” 카운터의 값을 전달하는 gettb() 함수로부터 여러 변수들을 초기화 하며, 라인 3-8 의 반복문은 이 변수에 이 CPU 가 할당한 값이 얼마나 오래 유지되는지 기록합니다. 물론, CPU 들 가운데 하나는 “승리” 할 것이고, 따라서 라인 6-7에서의 검사를 통과하지 못해 이 반복문을 빠져나오지 못할 겁니다.

Quick Quiz 15.10: Listing 15.3 의 코드 조각이 가지고 있는 실제 하드웨어에서는 옳지 못할 수도 있는 잘못된 가정은 무엇입니까?



반복문의 종료에 이어, firsttb 는 할당 잠시 후 취한 시간값을 가지고 lasttb 는 할당된 값을 아직 유지하고 있는 공유 변수의 마지막 샘플링 전에 취한 시간값, 또는 이 반복문에 진입하기 전에 이 공유 변수의 값이 바뀌었다면 firsttb 와 같은 값을 취합니다. 이는 각 CPU 의 state.variable 의 값에 대한 각 CPU 의 532 나노세컨드 동안의 시작을 그릴 수 있게 하는데, Figure 15.4 이

그것입니다. 이 데이터는 각자 한쌍의 하드웨어 쓰레드를 갖는 여덟개 코어를 가진 1.5GHz POWER5 시스템을 이용해 2006년에 수집되었습니다. CPU 1, 2, 3, 그리고 4는 이 값을 기록했으며, CPU 0은 테스트를 제어했습니다. 시간값 카운터 주기는 약 5.32 ns 이었는데, 중간 캐쉬 상태의 관측을 허용하기 충분히 세밀했습니다.

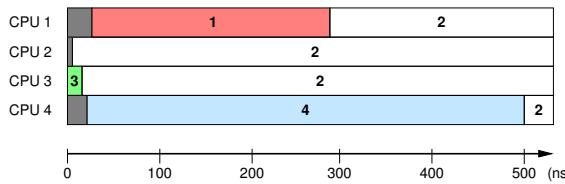


Figure 15.4: A Variable With Multiple Simultaneous Values

각 수평 막대는 특정 CPU의 시간에 따른 관측 결과를 나타내는데, 왼쪽 회색 영역은 연관 CPU의 첫번째 측정 전의 시간을 표시합니다. 첫 5ns 동안은 CPU 3 만이 이 변수의 값에 대한 의견이 있었습니다. 다음 10 ns 동안, CPU 2 와 3 은 이 변수의 값에 대해 동의하지 않았으나, 결국은 그 값이 “2” 라는데 동의했는데, 이는 실제로 최종적으로 모두가 동의하게 된 값이었습니다. 그러나, CPU 1 은 이 값이 “1” 이라고 거의 300 ns 동안 믿었으며 CPU 4는 거의 500 ns 동안 그 값이 “4” 라고 믿었습니다.

Quick Quiz 15.11: 어떻게 CPU 들이 같은 변수에서 같은 시각에 다른 값을 볼 수 있습니까?



Quick Quiz 15.12: CPU 2 와 3은 왜 그리 빨리 동의에 도달했으며, CPU 1 과 4는 그리 오래 걸렸나요?



그리고 네개 CPU의 상황이 흥미로웠다면, 같은 상황이지만 15개 CPU가 각자 시간 $t = 0$ 에 하나의 공유 변수에 각자의 수를 할당하는 경우를 보이는 Figure 15.5를 보시기 바랍니다. 이 그림의 두 다이어그램은 모두 Figure 15.4에서와 같은 방법으로 그려졌습니다. 유일한 차이는 수평 축의 단위가 시간단위 틱이란 것으로, 각 틱은 약 5.3 나노세컨드입니다. 따라서 전체 수행은 CPU 수의 증가에 맞춰 Figure 15.4에 기록된 이벤트에 비해 조금 더 오래 지속됩니다. 위 다이어그램은 전체 그림을 보이며, 아래의 것은 첫 50 틱을 확대해 보입니다. 다시 말하지만, CPU 0은 이 테스트를 조율하며, 따라서 어떤 값도 기록하지 않습니다.

모든 CPU가 결국은 마지막 값은 9 라는데 동의했습니다만 그 전에는 값 15 와 12 가 초기를 이끌었습니다. 아래 다이어그램에 수직선으로 표시된 시간 21 에는 이 변수의 값에 대한 14개의 다른 의견이 있었음을 보십시오. 또한 모든 CPU가 Figure 15.6에 보인 방향 있는 그래프로 보인 것처럼 순서는 일관적임을 보십시오.

Listing 15.4: Message-Passing Litmus Test (No Ordering)

```

1 C C-MP+o-wmb-o+o-o
2
3 {
4
5 P0(int* x0, int* x1) {
6     WRITE_ONCE(*x0, 2);
7     smp_wmb();
8     WRITE_ONCE(*x1, 2);
9 }
10
11 P1(int* x0, int* x1) {
12     int r2;
13     int r3;
14
15     r2 = READ_ONCE(*x1);
16     r3 = READ_ONCE(*x0);
17 }
18
19 exists (1:r2=2 /& 1:r3=0)

```

오. 더도 아니고 덜도 아니고, 이 그림들은 메모리 순서 오퍼레이션의 올바른 사용의 중요성을 강조합니다.

한 시점에 하나의 변수는 얼마나 많은 값을 가질 수 있을까요? 시스템의 스토어 베퍼당 하나입니다! 따라서 우리는 변수의 값과 시간의 흐름에 대한 안락한 직관에 작별을 고해야 하는 체계에 진입했습니다. 이는 메모리 순서 오퍼레이션이 필요한 체계입니다.

그러나 Chapters 3 and 6에서의 교훈을 기억하십시오. 모든 CPU가 같은 변수에 동시에 값을 저장하는 것은 병렬 프로그램을 설계하는 방법이 전혀 아닌데, 최소한 성능과 확장성이 여러분에게 중요한게 아니라면 그렇습니다.

불행히도, 메모리 순서 규칙은 여러분의 직관을 망치는 다른 많은 방법들을 갖고 이 방법들이 모두 성능과 확장성을 망치지는 않습니다. 다음 섹션은 관계없는 메모리 참조의 순서 재배치를 알아봅니다.

15.2.2 Memory-Reference Reordering

Section 15.1.1는 x86과 같이 상대적으로 강한 순서 규칙의 시스템도 앞의 스토어를 뒤의 로드와 최소한 이 스토어와 로드가 다른 변수를 향한 것일 때에는 순서를 바꿀 수 있음을 보였습니다. 이 섹션은 그 결과를 토대로 다른 로드와 스토어 조합을 알아봅니다.

15.2.2.1 Load Followed By Load

Listing 15.4 (C-MP+o-wmb-o+o-o.litmus)는 고전적인 *message-passing* 리트머스 테스트를 보이는데, x0는 메세지이고 x1은 메세지가 있는지 없는지 알리는 플래그입니다. 이 테스트에서, *smp_wmb()*는 *P0()*의 스토어가 순서잡히게 강제하지만, 로드를 위해선 어떤 순서규칙도 명시되지 않았습니다. x86 같은 상대적으로 강한 순서규칙의 아키텍처에서는 순서가 강제됩니다.

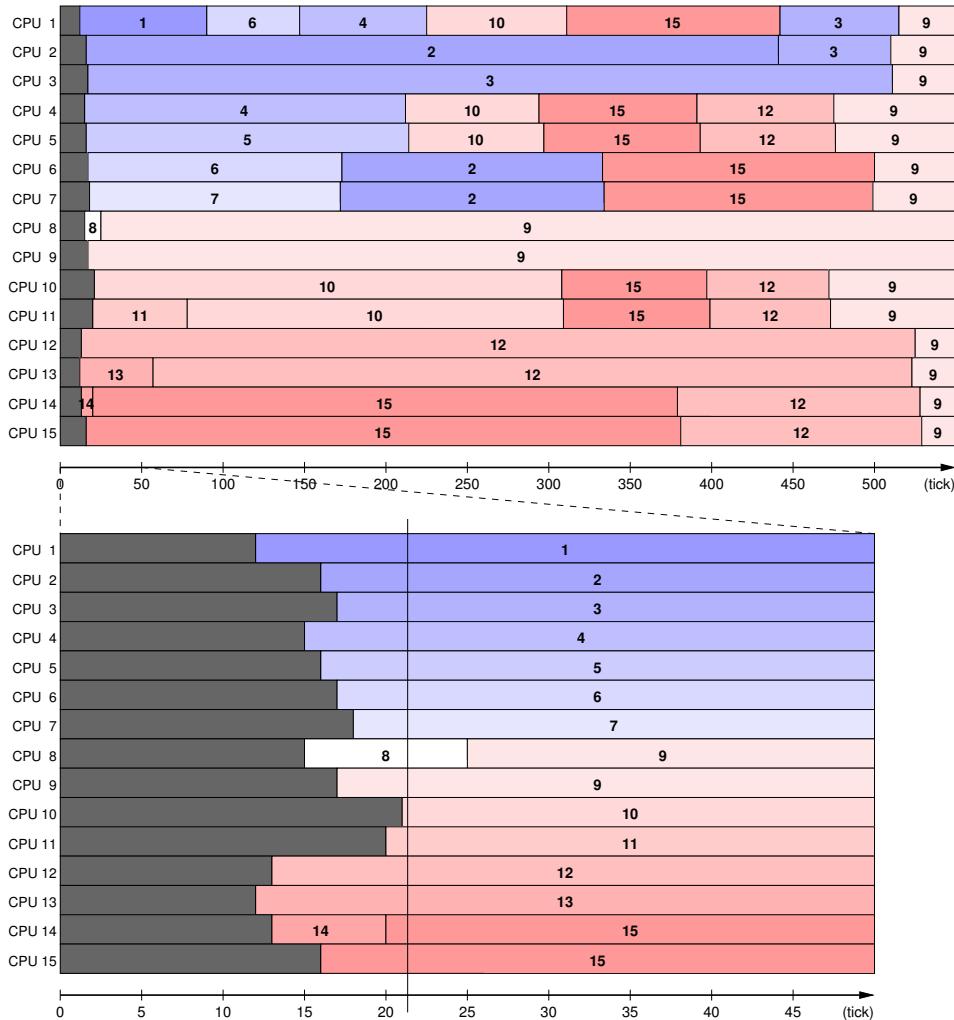


Figure 15.5: A Variable With More Simultaneous Values

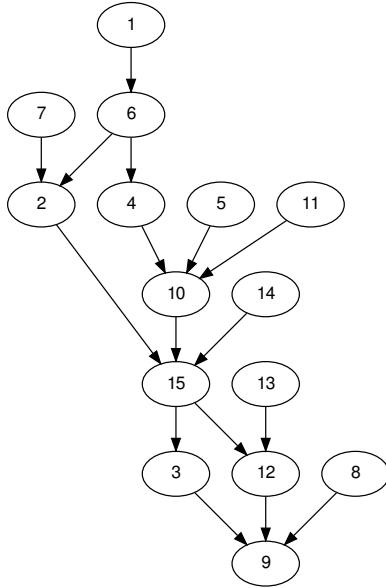


Figure 15.6: Possible Global Orders With More Simultaneous Values

그러나, 완화된 순서 규칙의 아키텍쳐에서는 종종 그러지 않습니다 [AMP⁺11]. 따라서, 이 리스트의 라인 19에서의 `exists` 절은 발동될 수 있습니다.

다른 위치로부터의 로드들을 재배치하는 이유 중 하나는 그렇게 하는게 앞의 로드가 캐쉬 미스를 일으키지만 뒤의 로드는 이미 존재하는 경우 수행을 빠르게 해준다는 겁니다.

Quick Quiz 15.13: 하지만 왜 load-load 재배치를 사용자에게 보이게 하죠? 중간에 스토어가 없는 흔한 경우에는 수행이 진행되게끔 투기적 수행 (speculative execution)을 하게 해서 순서 재배치가 안보이게끔 하는게 어떤가요?

따라서, 순서 잡힌 로드에 의존하는 이식성 있는 코드는 명시적 순서 규칙을 더해야 하는데, 예를 들면 Listing 15.5 (C-MP+o-wmb-o+o-rmb-o.litmus)의 라인 16에 보인 `smp_rmb()`를 더하는 것으로, 이는 `exists` 절이 발동되는 것을 막습니다.

15.2.2.2 Load Followed By Store

Listing 15.6 (C-LB+o-o+o-o.litmus)는 고전적인 *load-buffering* 리트머스 테스트를 보입니다. x86이나 IBM Mainframe 같은 상대적으로 강한 순서 규칙의 시스템은 앞의 로드를 뒤의 스토어와 재배치 하지 않습니다만, 많은 완화된 순서 규칙의 아키텍쳐는 그런 재배치

Listing 15.5: Enforcing Order of Message-Passing Litmus Test

```

1 C C-MP+o-wmb-o+o-rmb-o
2
3 {}
4
5 P0(int* x0, int* x1) {
6     WRITE_ONCE(*x0, 2);
7     smp_wmb();
8     WRITE_ONCE(*x1, 2);
9 }
10
11 P1(int* x0, int* x1) {
12     int r2;
13     int r3;
14
15     r2 = READ_ONCE(*x1);
16     smp_rmb();
17     r3 = READ_ONCE(*x0);
18 }
19
20 exists (1:r2=2 /\ 1:r3=0)

```

Listing 15.6: Load-Buffering Litmus Test (No Ordering)

```

1 C C-LB+o-o+o-o
2
3 {}
4
5 P0(int *x0, int *x1)
6 {
7     int r2;
8
9     r2 = READ_ONCE(*x1);
10    WRITE_ONCE(*x0, 2);
11 }
12
13 P1(int *x0, int *x1)
14 {
15     int r2;
16
17     r2 = READ_ONCE(*x0);
18     WRITE_ONCE(*x1, 2);
19 }
20
21 exists (1:r2=2 /\ 0:r2=2)

```

Listing 15.7: Enforcing Ordering of Load-Buffering Litmus Test

```

1 C C-LB+o-r+a-o
2
3 {}
4
5 P0(int *x0, int *x1)
6 {
7     int r2;
8
9     r2 = READ_ONCE(*x1);
10    smp_store_release(x0, 2);
11 }
12
13 P1(int *x0, int *x1)
14 {
15     int r2;
16
17     r2 = smp_load_acquire(x0);
18     WRITE_ONCE(*x1, 2);
19 }
20
21 exists (1:r2=2 /\ 0:r2=2)

```

Listing 15.9: Message-Passing Address-Dependency Litmus Test (No Ordering Before v4.15)

```

1 C C-MP+o-wmb-o+o-ad-o
2
3 {
4     y=1;
5     x1=y;
6 }
7
8 P0(int* x0, int** x1) {
9     WRITE_ONCE(*x0, 2);
10    smp_wmb();
11    WRITE_ONCE(*x1, x0);
12 }
13
14 P1(int** x1) {
15     int *r2;
16     int r3;
17
18     r2 = READ_ONCE(*x1);
19     r3 = READ_ONCE(*r2);
20 }
21
22 exists (1:r2=x0 /\ 1:r3=1)

```

Listing 15.8: Message-Passing Litmus Test, No Writer Ordering (No Ordering)

```

1 C C-MP+o-o+o-rmb-o
2
3 {}
4
5 P0(int* x0, int* x1) {
6     WRITE_ONCE(*x0, 2);
7     WRITE_ONCE(*x1, 2);
8 }
9
10 P1(int* x0, int* x1) {
11     int r2;
12     int r3;
13
14     r2 = READ_ONCE(*x1);
15     smp_rmb();
16     r3 = READ_ONCE(*x0);
17 }
18
19 exists (1:r2=2 /\ 1:r3=0)

```

를 정말로 허용합니다 [AMP⁺¹¹]. 따라서, 라인 21의 `exists` 절은 정말로 발동될 수 있습니다.

실제 하드웨어가 이런 재배치를 하는 것은 흔하지 않지만 [Mar17], 이렇게 하길 바라게 되는 한가지 상황은 로드가 캐시 미스를 일으키지만 스토어 버퍼는 거의 완전히 차있어서, 그리고 다음 스토어를 위한 캐시라인은 준비되어 있을 때입니다. 따라서, 이식성 있는 코드는 모든 필요한 순서규칙을 강제해야 하는데, Listing 15.7 (C-LB+o-r+a-o.litmus)에 보인 것과 같습니다. `smp_store_release()` 와 `smp_load_acquire()`는 라인 21의 `exists` 절이 결코 발동되지 않게 보장합니다.

15.2.2.3 Store Followed By Store

Listing 15.8 (C-MP+o-o+o-rmb-o.litmus)는 다시 고전적인 message-passing 리트머스 테스트를 보이는데, P1()의 로드를 위해선 `smp_rmb()`를 제공하지만 P0()의 스토어에는 어떤 순서규칙도 주지 않습니다. 여기서도 상대적으로 강한 순서규칙의 아키텍쳐는 순서를 강제합니다만 완화된 순서규칙의 아키텍쳐는 꼭 그러지만은 않는데 [AMP⁺¹¹], 이는 `exists` 절이 발동될 수 있음을 의미합니다. 그런 재배치가 도움될 수 있는 한가지 상황은 스토어 버퍼가 꽉차있고, 또다른 스토어가 수행될 준비가 되어있지만 가장 예전 스토어를 위한 캐시라인은 아직 사용 불가할 때입니다. 이런 경우, 스토어가 순서와 다르게 완료될 수 있게 하는건 수행이 진행될 수 있게 합니다. 따라서, 이식성 있는 코드는 명시적으로 스토어의 순서를 맞춰줘야 하는데, 예를 들면 Listing 15.5에 보인 것과 같아서, `exists` 절이 발동되는 걸 막습니다.

Quick Quiz 15.14: 강한 순서규칙의 시스템은 왜 불필요한 `smp_rmb()` 와 `smp_wmb()` 수행이라는 성능 비용을 지불해야 하죠? 완화된 순서규칙의 시스템이 그들의 잘못된 순서잡기 선택의 완전한 비용을 책임져야 하지 않나요???



15.2.3 Address Dependencies

주소 종속성 (*address dependency*)는 로드 명령에 의해 반환된 값이 뒤의 메모리 참조 명령에 의해 사용될 주소를 계산하는데 사용될 때 발생합니다.

Listing 15.10: Enforced Ordering of Message-Passing Address-Dependency Litmus Test (Before v4.15)

```

1 C C-MP+o-wmb-o+o-addr-o
2
3 {
4     y=1;
5     x1=y;
6 }
7
8 P0(int* x0, int** x1) {
9     WRITE_ONCE(*x0, 2);
10    smp_wmb();
11    WRITE_ONCE(*x1, x0);
12 }
13
14 P1(int** x1) {
15     int *r2;
16     int r3;
17
18     r2 = lockless_dereference(*x1); // Obsolete
19     r3 = READ_ONCE(*r2);
20 }
21
22 exists (1:r2=x0 /\ 1:r3=1)

```

Listing 15.9 (C-MP+o-wmb-o+o-addr-o.litmus)는 메세지 패싱 패턴의 연결 기반 변종을 보입니다. 헤드 포인터는 $x1$ 으로, int 변수 y 를 참조하도록 초기화되며 (라인 5), y 는 1으로 초기화되어 있습니다 (라인 4). $P0()$ 는 헤드 포인터 $x1$ 이 $x0$ 를 참조하도록 업데이트하지만 (라인 11), 그것을 2로 초기화한 후이며 (라인 9) 그 순서를 강제합니다 (라인 10). $P1()$ 은 헤드 포인터 $x1$ 을 가져오고 (라인 18), 이어서 참조된 값을 로드합니다 (라인 19). 따라서 라인 18에서의 로드로부터 라인 19 사이에 주소 종속성이 존재합니다. 이 경우, 라인 18에 의해 반환되는 값은 라인 19에 의해 사용되는 주소값이지만, 많은 변종이 가능한데, C-언어 \rightarrow 오퍼레이터를 통한 필드 액세스, 더하기, 빼기, 그리고 배열 인덱싱이 포함됩니다.⁶

어떤 사람들은 라인 18의 헤드 포인터로부터의 로드가 라인 19의 역참조 전으로 순서잡히길 바랄 수도 있겠는데, 리눅스 v4.15 이후에서는 실제로 그렇습니다. 그러나, v4.15 전에는 DEC Alpha에서는 그렇지 않은데, Section 15.5.1에서 자세한 사항이 이야기되어 있듯이 의존적 로드에 예측된 값을 사용하게 할 수 있습니다. 따라서, 오래된 버전의 리눅스에서는 Listing 15.9의 `exists` 절이 발동될 수 있습니다.

Listing 15.10은 라인 19의 `READ_ONCE()`를 DEC Alpha를 제외한 모든 플랫폼에서는 `READ_ONCE()`로 동작하지만 DEC Alpha에서는 `READ_ONCE()`에 이어 `smp_mb()`를 수행하며 따라서 모든 플랫폼에서 필요한 순서를 강제하게 되는 `lockless_dereference()`로 교체함으로써 v4.15 이전 리눅스 커널에서도 이게 DEC

⁶ 하지만 리눅스 커널에서 주소 종속성은 배열로의 포인터를 통해 이루어져야지 배열 인덱스를 통해 이루어져선 안됩니다.

Listing 15.11: S Address-Dependency Litmus Test

```

1 C C-S+o-wmb-o+o-addr-o
2
3 {
4     y=1;
5     x1=y;
6 }
7
8 P0(int* x0, int** x1) {
9     WRITE_ONCE(*x0, 2);
10    smp_wmb();
11    WRITE_ONCE(*x1, x0);
12 }
13
14 P1(int** x1) {
15     int *r2;
16
17     r2 = READ_ONCE(*x1);
18     WRITE_ONCE(*r2, 3);
19 }
20
21 exists (1:r2=x0 /\ x0=2)

```

Alpha에서 안정적으로 동작하게끔 하는 방법을,⁷ 즉 `exists` 절이 발동되는 걸 막는 방법을 보입니다.

하지만 그 의존적 오퍼레이션이 로드가 아닌 스토어라면, 예를 들어 Listing 15.11 (C-S+o-wmb-o+o-addr-o.litmus)에 보인 것과 같은 S 리트머스 테스트 [AMP¹¹] 같은 경우라면 어떨까요? 어떤 제품 수준 플랫폼도 스토어를 예측하진 않기 때문에, 라인 9의 `WRITE_ONCE()`는 라인 18의 `WRITE_ONCE()`를 덮어쓸 수 없으며, 이는 라인 21의 `exists` 절이 발동될 수 없음을, 심지어 DEC Alpha에서도, v4.15 전의 리눅스 커널에서도 그려함을 의미합니다.

Quick Quiz 15.15: 하지만 모든 플랫폼이 Listings 15.10 and 15.11의 `exists` 절을 정말로 발동시키지 않음을 어떻게 알죠?

Quick Quiz 15.16: SP, MP, LB, 이제는 S 까지. 이 리트머스 테스트 약자들은 어디서 왔고 누가 이걸 따라갈 수 있겠습니까?

그러나, Section 15.3.2에서 이야기 했듯 주소 종속성은 잘못 짜이기 쉽고 컴파일러 최적화에 의해 쉽게 깨짐에 유의하는게 중요합니다.

15.2.4 Data Dependencies

Data dependency (데이터 종속성)은 어떤 로드 명령에 의해 반환된 값이 뒤의 스토어 명령에 의해 저장되는 데이터를 계산하는데 사용될 때 발생합니다. 앞의 “데이터”에 유의하세요: 어떤 로드에 의해 반환된 값이

⁷ `lockless_dereference()`는 v4.15 이후 버전에서는 필요치 않으며, 따라서 그 리눅스 커널들에서는 존재하지 않음을 알아두시기 바랍니다. 또한 이 문장을 담고 있는 이 책의 버전에서도 필요치 않습니다.

Listing 15.12: Load-Buffering Data-Dependency Litmus Test

```

1 C C-LB+o-r+o-data-o
2
3 {}
4
5 P0(int *x0, int *x1)
6 {
7     int r2;
8
9     r2 = READ_ONCE(*x1);
10    smp_store_release(x0, 2);
11 }
12
13 P1(int *x0, int *x1)
14 {
15     int r2;
16
17     r2 = READ_ONCE(*x0);
18     WRITE_ONCE(*x1, r2);
19 }
20
21 exists (1:r2=2 /\ 0:r2=2)

```

뒤의 스토어 명령에 의해 사용되는 주소를 계산하는데 사용된다면 그건 주소 종속성이 됩니다.

Listing 15.12 (C-LB+o-r+o-data-o.litmus)는 Listing 15.7 와 유사하지만 P1()의 라인 17 와 18 사이 순서가 acquire load 대신 데이터 종속성으로 강제된다는 점이 다릅니다: 라인 17에 의해 로드되는 값이 라인 18에서 스토어 하는 값입니다. 이 데이터 종속성에 의해 제공되는 순서 규칙은 exists 절이 발동되는 것을 막는 데에 충분합니다.

Section 15.3.2에서 이야기했듯, 주소 종속성에서와 마찬가지로 데이터 종속성은 잘못 짜이기 쉽고 컴파일러 최적화에 의해 쉽게 부서집니다. 실제로, 데이터 종속성은 주소 종속성보다도 더 잘못 짜이기가 쉽습니다. 그 이유는 주소 종속성은 일반적으로 포인터 값을 사용하기 때문입니다. 대조적으로, Listing 15.12에서 보인 것과 같이 데이터 종속성을 단하나의 값을 통해 옮기고 싶어지게 마련이며, 이는 컴파일러가 이를 최적화해 제거해 버리기 충분합니다. 한가지만 예를 들자면, 로드된 정수가 상수 0으로 곱해진다면, 컴파일러는 그 결과가 0임을 알아서 로드된 값을 0으로 치환하여 이 종속성을 깨버릴 수 있습니다.

Quick Quiz 15.17: 하지만 기다려요!!! Listing 15.12의 라인 17는 이 로드를 휘발성으로 기록하는 READ_ONCE()를 사용하는데, 이는 컴파일러가 그 값이 나중에 0으로 곱해짐을 안다 하더라도 로드 명령을 만들어야 함을 의미합니다. 그런데 어떻게 컴파일러가 이 데이터 종속성을 깰 수 있죠?



요약하자면, 여러분은 컴파일러가 데이터 종속성을 깨버리는 걸 막을 때에만 거기 의존할 수 있습니다.

Listing 15.13: Load-Buffering Control-Dependency Litmus Test

```

1 C C-LB+o-r+o-ctrl-o
2
3 {}
4
5 P0(int *x0, int *x1)
6 {
7     int r2;
8
9     r2 = READ_ONCE(*x1);
10    smp_store_release(x0, 2);
11 }
12
13 P1(int *x0, int *x1)
14 {
15     int r2;
16
17     r2 = READ_ONCE(*x0);
18     if (r2 >= 0)
19         WRITE_ONCE(*x1, 2);
20 }
21
22 exists (1:r2=2 /\ 0:r2=2)

```

Listing 15.14: Message-Passing Control-Dependency Litmus Test (No Ordering)

```

1 C C-MP+o-r+o-ctrl-o
2
3 {}
4
5 P0(int* x0, int* x1) {
6     WRITE_ONCE(*x0, 2);
7     smp_store_release(x1, 2);
8 }
9
10 P1(int* x0, int* x1) {
11     int r2;
12     int r3 = 0;
13
14     r2 = READ_ONCE(*x1);
15     if (r2 >= 0)
16         r3 = READ_ONCE(*x0);
17 }
18
19 exists (1:r2=2 /\ 1:r3=0)

```

15.2.5 Control Dependencies

Control dependency (제어 종속성)은 로드 명령에 의해 반환되는 값이 뒤의 스토어 명령이 수행될지 말지를 결정하기 위해 검사될 때 발생합니다. “뒤의 스토어 명령”에 주목하세요: 많은 플랫폼이 로드 대 로드 제어 종속성을 지키지 않습니다.

Listing 15.13 (C-LB+o-r+o-ctrl-o.litmus)는 또 다른 로드 버퍼링 예를 보이는데, 이번에는 라인 17에서의 로드와 라인 19에서의 스토어의 순서를 맞추기 위해 제어 종속성을 (라인 18) 사용합니다. 이 순서 강제는 exists 절이 발동되는 걸 막기에 충분합니다.

그러나, 제어 종속성은 데이터 종속성보다도 최적화로 사라지기 쉬우며, Section 15.3.3가 여러분의 컴파일러가 제어 종속성을 깨버리는 걸 막기 위해 따라야 할 규칙들을 설명합니다.

제어 종속성은 로드에서 스토어로의 순서만을 제공함을 반복할 가치가 있습니다. 따라서, Listing 15.14 (C-MP+o-r+o-ctrl-o.litmus)의 라인 14-16에 보인 제어 종속성은 순서를 제공하지 않으며, 따라서 `exists` 절이 발동되는 것을 막지 않습니다.

요약하자면, 제어 종속성은 유용할 수 있으나 관리하기 어려운 항목입니다. 따라서 여러분은 성능 고려사항이 다른 해법을 허용치 않을 때에만 그걸 사용해야 합니다.

Quick Quiz 15.18: 제어 종속성은 언어 표준에 의해 강제시 된다면 더 강해지지 않을까요?

15.2.6 Cache Coherence

Cache-coherent (캐시 일관성이 지켜지는) 플랫폼에서, 모든 CPU 는 특정 변수의 로드와 스토어 순서에 대해 동의하게 됩니다. 다행히도, `READ_ONCE()` 와 `WRITE_ONCE()` 가 사용된다면 거의 모든 플랫폼이 cache-coherent 하게 되는데, Table 15.3 의 “SV” 행에 표시된 바와 같습니다. 불행히도, 이 속성은 여러번 이름지어 질 정도로 유명한데, “single-variable SC”,⁸ “single-copy atomic” [SF95], 그리고 간단히 “coherence” [AMP¹¹]라는 이름이 사용되어왔습니다. 이 개념을 위한 또 다른 용어를 발명해서 혼란을 가중시키기보다는 이 책에서는 “cache coherence (캐시 일관성)” 또는 “coherence (일관성)”라는 용어를 사용하겠습니다.

Listing 15.15 (C-CCIRIW+o+o+o-o+o-o.litmus) 는 캐시 일관성을 테스트하는 리트머스 테스트를 보이는 데, “IRIW” 는 “independent reads of independent writes” 를 의미합니다. 이 리트머스 테스트는 하나의 변수만을 사용하므로 `P2()` 와 `P3()` 는 `P0()` 와 `P1()` 의 스토어 순서에 동의해야만 합니다. 달리 말하면 `P2()` 가 `P0()` 의 스토어가 먼저 왔다고 믿는다면 `P3()` 는 `P1()` 의 스토어가 먼저 왔다고 믿지 말아야 합니다. 그리고 실제로 라인 33 의 `exists` 절은 이 상황이 나타나면 발동될 겁니다.

Quick Quiz 15.19: 하지만 Listing 15.15 에서, `P2()` 의 `r1` 과 `r2` 가 값 2 와 1 을 각각 관측했고 `P3()` 의 `r2` 와 `r4` 가 값 1 과 2 를 각각 관측한다면 그저 나쁜 결과가 되지 않나요?

하나의 메모리 영역으로의 겹치는 다른 크기의 로드와 스토어가 (C-언어의 `union` 키워드를 이용해 셋업될 수도 있겠습니다) 비슷한 순서 보장을 제공할 거라 예상하고 싶을 수 있습니다. 그러나, Flur et al. [FSP¹⁷] 은 그런 보장이 실제 하드웨어에서는 위배될 수 있음을

Listing 15.15: Cache-Coherent IRIW Litmus Test

```

1 C C-CCIRIW+o+o+o-o+o-o
2
3 {
4
5 P0(int *x)
6 {
7     WRITE_ONCE(*x, 1);
8 }
9
10 P1(int *x)
11 {
12     WRITE_ONCE(*x, 2);
13 }
14
15 P2(int *x)
16 {
17     int r1;
18     int r2;
19
20     r1 = READ_ONCE(*x);
21     r2 = READ_ONCE(*x);
22 }
23
24 P3(int *x)
25 {
26     int r3;
27     int r4;
28
29     r3 = READ_ONCE(*x);
30     r4 = READ_ONCE(*x);
31 }
32
33 exists(2:r1=1 / \ 2:r2=2 / \ 3:r3=2 / \ 3:r4=1)

```

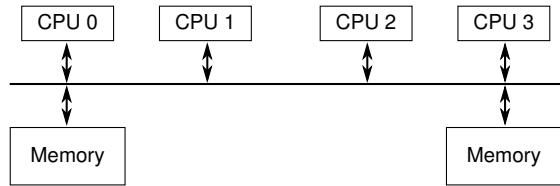


Figure 15.7: Global System Bus And Multi-Copy Atomicity

보이는 놀랍도록 간단한 리트머스 테스트를 발견했습니다. 따라서 코드가 특정 변수에 겹치지 않는 같은 크기의 정렬된 액세스만을 하도록 강제할 필요가 있는데, 적어도 이식성이 고려사항이라면 그렇습니다.⁹

더 많은 변수와 쓰레드를 더하는건 재배치와 다른 반직관적 행동을 할 영역을 늘리게 되는데, 다음 섹션에서 이를 이야기 합니다.

15.2.7 Multicopy Atomicity

완벽히 *multicopy atomic* [SF95] 플랫폼에서 수행되는 쓰레드는 스토어의 순서들에 대해 다른 변수들에 대해

⁹ 모든 스토어에 atomic RMW 오퍼레이션을 (예를 들어 `xchg()`) 사용하는게 sequentially consistent 순서 규칙을 제공할 거라 믿을 이유도 있지만 이 역시 아직 증명된 바 없습니다.

⁸ SC 는 sequentially consistent 를 의미함을 기억하세요.

서조차 동의할 것이 보장됩니다. 그런 시스템에 대한 유용한 상상적 모델은 Figure 15.7에 보인 단일 버스 아키텍쳐입니다. 만약 각 스토어가 버스 상의 메세지로 귀결된다면, 그리고 그 버스가 한번에 하나의 스토어만 수용할 수 있다면, 모든 CPU 쌍은 그들이 관측하는 모든 스토어의 순서에 대해 동의할 겁니다. 불행히도, 컴퓨터 시스템을 스토어 버퍼나 캐쉬조차 없이 이 그림에 보인대로 구축하는 것은 아주 느린 계산을 초래할 겁니다. 따라서 multicopy atomicity를 제공하는데 관심 있는 대부분의 CPU 제조사는 그대신 모든 CPU가 모든 스토어에 대한 순서에 동의해야 한다는 요구사항에서 해당 스토어를 가하는 CPU는 배제하여 약간 더 완화된 *other-multicopy atomicity* [ARM17, Section B2.3]를 제공합니다.¹⁰ 이는 CPU의 부분집합만이 스토어를 한다면, 다른 CPU들은 스토어의 순서에 동의하며, 따라서 “other-multicopy atomicity”에 “other”가 붙었습니다. Multicopy-atomic 플랫폼과 달리 other-multicopy-atomic 플랫폼에서는 스토어를 하는 CPU는 자신의 스토어를 일찍 관측하는게 허용되는데, 이는 뒤따르는 로드가 스토어 버퍼에서 새로 저장된 값을 직접 보는게 가능하게 하여 성능을 개선합니다.

Quick Quiz 15.20: Multicopy atomic과 other-multicopy atomic에서 다른 행동을 보이는 구체적 예를 들어줄 수 있을까요?



모든 플랫폼이 multi-copy atomicity의 어떤 변종을 제공하는 날이 언젠가는 오겠지만, 그 전까지는 non-multicopy-atomic 플랫폼이 존재하므로 소프트웨어는 이를 처리해야 합니다.

Listing 15.16 (C-WRC+o+o-data-o+o-rmb-o.litmus)이 multicopy atomicity를 보이는데, 즉 multicopy-atomic 플랫폼에서 라인 28의 exists 절은 발동될 수 없습니다. 대조적으로, non-multicopy-atomic 플랫폼에서 P1()의 액세스가 데이터 종속성으로 순서 잡히고 P2()의 액세스가 smp_rmb()에 의해 순서 잡힘에도 불구하고 exists 절은 발동될 수 있습니다. Multicopy atomicity의 정의는 모든 쓰레드가 스토어의 순서에 동의할 것을 요구하며 이는 모든 스토어가 모든 쓰레드에 동시에 도달하는 것으로 생각될 수 있습니다. 따라서, non-multicopy-atomic 플랫폼은 스토어가 다른 쓰레드에 다른 시간에 도달할 수 있게 합니다. 특히, P0()의 스토어는 그게 P2()에 도달하기 한참 전에 P1()에 도달할 수도 있는데, 이는 P1()의 스토어가 P0()의 스토어가 도달하기 전에 P2()에 도달할 수도 있을 가능성을 제기합니다.

¹⁰ 2021년 초 기준으로, Armv8과 x86는 other-multicopy atomicity를 제공하고, IBM mainframe은 완벽한 multicopy atomicity를 제공하며, PPC는 어떤 multicopy atomicity도 제공하지 않습니다. 보다 자세한 내용은 Table 15.5에 있습니다.

Listing 15.16: WRC Litmus Test With Dependencies (No Ordering)

```

1 C C-WRC+o+o-data-o+o-rmb-o
2 {
3 }
4
5 P0(int *x)
6 {
7     WRITE_ONCE(*x, 1);
8 }
9
10 P1(int *x, int* y)
11 {
12     int r1;
13
14     r1 = READ_ONCE(*x);
15     WRITE_ONCE(*y, r1);
16 }
17
18 P2(int *x, int* y)
19 {
20     int r2;
21     int r3;
22
23     r2 = READ_ONCE(*y);
24     smp_rmb();
25     r3 = READ_ONCE(*x);
26 }
27
28 exists (1:r1=1 /\ 2:r2=1 /\ 2:r3=0)

```

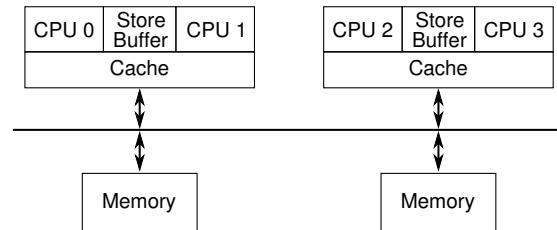


Figure 15.8: Shared Store Buffers And Multi-Copy Atomicity

이는 일반적인 물리법칙으로 제한되는 실제 시스템이 왜 Listing 15.16의 exists 절을 발동하지 않는지에 대한 질문을 자아냅니다. 그런 실제 시스템에 대한 만화적 그림이 Figure 15.8에 보입니다. CPU 0과 CPU 1은 스토어 버퍼를 공유하며, CPU 2와 3은 마찬가지입니다. 이는 CPU 1이 스토어 버퍼에서 값을 로드할 수 있으며, 따라서 CPU 0이 스토어 한 값을 즉각 볼 가능성이 있습니다. 대조적으로, CPU 2와 3은 연관된 캐쉬 라인이 그들에게 넘어오기까지 기다려야 합니다.

Quick Quiz 15.21: 그럼 누가 공유된 스토어 버퍼를 갖는 시스템을 설계하려 생각이나 하겠나요???



Table 15.4가 Listing 15.16의 exists 절을 발동시킬 수 있는 사건들의 순서를 보입니다. 이 사건들의 순서는

Table 15.4: Memory Ordering: WRC Sequence of Events

P0()		P0() & P1()		P1()		P2()	
Instruction	Store Buffer	Cache	Instruction	Instruction	Store Buffer	Cache	
1 (Initial state)		y==0	(Initial state)	(Initial state)		x==0	
2 x = 1;	x==1	y==0				x==0	
3 (Read-Invalidate x)	x==1	y==0	r1 = x (1)			x==0	
4	x==1 y==1	y==0	y = r1	r2 = y		x==0	
5	x==1	y==1	(Finish store)	(Read y)		x==0	
6 (Respond y)	x==1	y==1		(r2==1)		x==0 y==1	
7	x==1	y==1	smp_rmb()			x==0 y==1	
8	x==1	y==1	r3 = x (0)			x==0 y==1	
9	x==1	x==0 y==1		(Respond x)		y==1	
10 (Finish store)	x==1 y==1					y==1	

P0() 와 P1() 이 캐시와 스토어 버퍼를 Figure 15.8에 보인 것과 같은 방식으로 공유한다는 데 의존적입니다.

Quick Quiz 15.22: 그러나 P0() 와 P1() 은 스토어 버퍼와 캐시를 공유하지만 P2() 는 자신만의 그것들을 하나씩 갖는다는 건 공평하지 않지 않나요???

첫번째 열은 초기 상태를 보이는데, y의 초기값은 P0() 와 P1() 의 공유 캐시에 있으며 x의 초기값은 P2() 의 캐시에 있습니다.

두번째 열은 라인 7에서의 P0() 의 스토어의 즉각적 효과를 보입니다. x를 포함하는 캐시라인은 P0() 와 P1() 의 공유 캐시에 존재하지 않으므로 새 값(1)은 공유 스토어 버퍼에 저장됩니다.

세번째 열은 두 전환을 보입니다. 먼저, P0() 는 x를 포함하는 캐시라인을 가져오기 위해 read-invalidate 오퍼레이션을 일으켜서 x의 새 값을 공유 스토어 버퍼 밖으로 비워지게 합니다. 두번째로, P1() 은 x를 로드하는데 (라인 14), x의 새 값을 공유 스토어 버퍼에서 곧바로 얻을 수 있기 때문에 즉시 완료됩니다.

네번째 열 역시 두개의 전환을 보입니다. 먼저, P1() 의 y로의 스토어가 (라인 15) 즉각 효과를 발휘해 새 값을 공유 스토어 버퍼에 넣습니다. 두번째로, P2() 의 y로드 시작을 보입니다 (라인 23).

다섯번째 열은 두 전환을 이어 보입니다. 먼저, P1() 이 y 스토어를 완료해서 공유 스토어 버퍼를 캐시로 비워냅니다. 두번째로, P2() 가 y를 담는 캐시라인을 요청합니다.

여섯번째 열은 P2() 가 y를 담는 캐시라인을 받는 걸 보이는데, 이는 r2로의 로드를 마칠 수 있게 하는데, 여기선 값 1을 얻게 됩니다.

일곱번째 열은 P2() 가 smp_rmb() 를 수행하는 것을 (라인 24) 보이는데, 이로 인해 두 로드는 순서를 지킵니다.

여덟번째 열은 P2() 가 x로드를 수행하는 걸 보이는 데, P2() 의 캐시에서 값 0을 즉각 가져옵니다.

아홉번째 열은 P2() 가 마침내 세번째 열에서의 P0() 의 x를 담는 캐시라인에 대한 요구에 응답함을 보입니다.

마지막으로, 열번째 열은 P0() 가 스토어를 마쳐서 x에 대한 그것의 값을 공유 스토어 버퍼에서 공유 캐시로 비우는 것을 보입니다.

라인 28에서의 exists 절이 발동되었음을 유의하세요. r1과 r2의 값은 모두 1이며, r3의 마지막 값은 0입니다. 이 기묘한 결과는 P0() 의 x의 새 값이 P2() 와의 통신 한참 전에 P1() 에게 소통되었기 때문에 발생했습니다.

Quick Quiz 15.23: Table 15.4에서 P1()의 스토어에 비해 P0()의 스토어는 대체 왜 그렇게 느렸던 거죠? 달리 말하자면, Listing 15.16의 라인 28에서의 exists 절은 실제 시스템에서 정말로 발동되나요?

종속성이 순서를 제공하지만 그건 각 쓰레드에 국한되기 때문에 이 반직관적인 결과가 발생합니다. 이 세개 쓰레드의 예는 더 강한 순서를 필요로 하는데, 그것이 Sections 15.2.7.1부터 15.2.7.4의 주제입니다.

15.2.7.1 Cumulativity

Listing 15.16에 보인 세개 쓰레드의 예는 누적되는(cumulative) 순서, 또는 *cumulativity*를 필요로 합니다. 누적되는 메모리 순서 오퍼레이션은 그것을 앞서는 액세스만이 아니라 같은 변수에 대한 모든 쓰레드의 이전 액세스의 순서를 잡습니다.

종속성은 *cumulativity*를 제공하지 않는데, page 301의 Table 15.3에 있는 READ_ONCE() 열의 “C” 칸이 비

Listing 15.17: WRC Litmus Test With Release

```

1 C C-WRC+o+o-r+a-o
2 {}
3 {}
4
5 P0(int *x)
6 {
7     WRITE_ONCE(*x, 1);
8 }
9
10 P1(int *x, int* y)
11 {
12     int r1;
13
14     r1 = READ_ONCE(*x);
15     smp_store_release(y, r1);
16 }
17
18 P2(int *x, int* y)
19 {
20     int r2;
21     int r3;
22
23     r2 = smp_load_acquire(y);
24     r3 = READ_ONCE(*x);
25 }
26
27 exists (1:r1=1 /\ 2:r2=1 /\ 2:r3=0)

```

Listing 15.18: W+RWC Litmus Test With Release (No Ordering)

```

1 C C-W+RWC+o-r+a-o+o-mb-o
2 {}
3 {}
4
5 P0(int *x, int *y)
6 {
7     WRITE_ONCE(*x, 1);
8     smp_store_release(y, 1);
9 }
10
11 P1(int *y, int *z)
12 {
13     int r1;
14     int r2;
15
16     r1 = smp_load_acquire(y);
17     r2 = READ_ONCE(*z);
18 }
19
20 P2(int *z, int *x)
21 {
22     int r3;
23
24     WRITE_ONCE(*z, 1);
25     smp_mb();
26     r3 = READ_ONCE(*x);
27 }
28
29 exists(1:r1=1 /\ 1:r2=0 /\ 2:r3=0)

```

어있는 이유입니다. 그러나, “C” 칸의 “C”로 나타내진 것처럼, release 오퍼레이션은 cumulative를 제공합니다. 따라서, Listing 15.17 (C-WRC+o+o-r+a-o.litmus)는 Listing 15.16의 데이터 종속성을 release 오퍼레이션으로 교체합니다. Release 오퍼레이션은 cumulative 하므로 그 순서는 Listing 15.17의 라인 14에서의 P1()에 의한 x 로드만이 아니라 라인 7에서의 P0()의 x 스토어에도 순서가 적용됩니다—그러나 그 로드가 스토어 된 값을 반환했을 때만 그런데, 이는 라인 27의 exists 절의 1:r1=1과 맞아떨어집니다. 이는 P2()의 load-acquire 가 라인 24에서의 x 로드가 라인 7에서의 스토어 뒤에 일어나게 강제하여, 반환된 값은 1이며, 2:r3=0에 맞지 않게 되므로, 결국 exists 절이 발생되는 걸 막습니다.

이 순서 규칙은 Figure 15.9에 그림으로 그려져 있습니다. 또한 cumulative는 시간상의 한 단계에만 제한되지 않음을 알아두세요. 라인 7의 스토어 전에 어떤 쓰레드에 의해서든 x로의 스토어나 로드가 있었다면 그 앞의 로드나 스토어는 라인 24에서의 로드 전으로 순서잡힐 겁니다, r1과 r2가 둘 다 값 1을 가지고 있을 경우에만 그렇긴 하지만요.

요약하자면, cumulative 순서 오퍼레이션을 사용하는 것은 일부 상황에서 non-multicopy-atomic 행동을 제거할 수 있습니다. 그러나 cumulative 역시 제한이 있는데, 다음 섹션에서 이를 알아봅니다.

15.2.7.2 Propagation

Listing 15.18 (C-W+RWC+o-r+a-o+o-mb-o.litmus)는 심지어 완전한 메모리 배리어가 있다 하더라도 발생하는 cumulative와 store-release의 한계를 보입니다. 문제는 라인 8의 smp_store_release()가 cumulative를 갖는다 해도, 그리고 그 cumulative가 라인 26의 P2()의 로드를 순서잡는다 해도, smp_store_release()의 순서는 P1()의 로드(라인 17)와 P2()의 스토어(라인 24)를 거쳐 전파될 수 없다는 겁니다. 이는 라인 29의 exists 절이 정말로 발생될 수 있음을 의미합니다.

Quick Quiz 15.24: 하지만 그 리트머스 테스트에 최소 세개의 쓰레드가 존재하지 않는다면 전파에 대해선 걱정할 필요가 없을 거예요, 그렇죠?

■ 이 상황이 완전히 반직관적으로 보일 수도 있겠습니까만, 빛의 속도는 유한하고 컴퓨터의 크기는 0이 아님을 기억하세요. 따라서 P2()의 z로의 스토어의 효과가 P1()에 전파되기 위해선 시간이 걸리며, 이는 P1()의 z 읽기가 훨씬 나중에 이루어질 수 있으나 여전히 과거 값인 0을 읽을 수 있음을 의미합니다. 이 상황은 Figure 15.10에 보여져 있습니다: 로드가 이전 값을 봄은 이 로드가 새 값의 스토어보다 시간상으로 먼저 이루어졌음을 의미하지 않습니다.

Listing 15.18 또한 두개가 아닌 세개의 프로세스가 존재할 때의 메모리 배리어 짹 맞추기 한계를 보입니다.

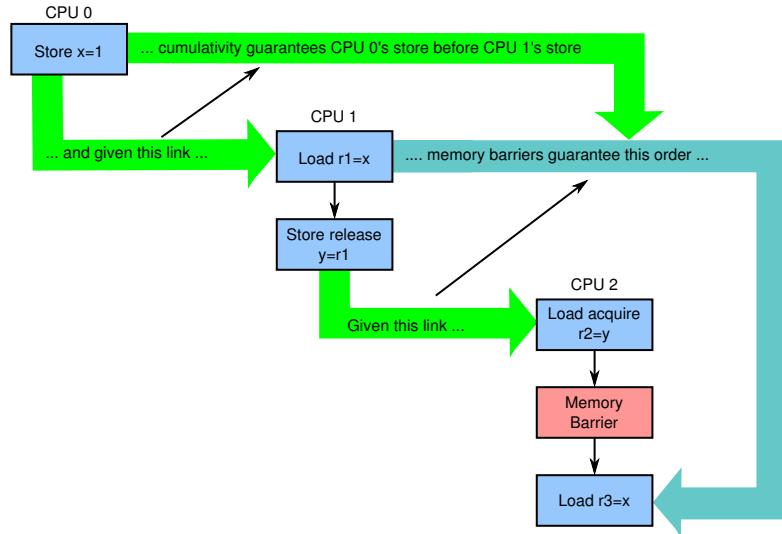


Figure 15.9: Cumulativity

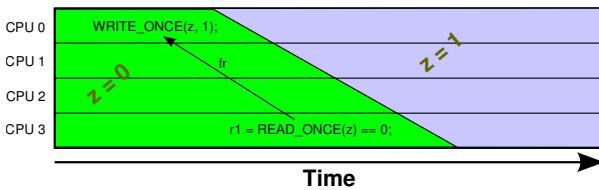


Figure 15.10: Load-to-Store is Counter-Temporal

이 더 복잡한 리트머스 테스트는 사이클이 존재한다고도 말해질 수 있는데, 메모리 배리어 짹맞추기는 두개 쓰래드 사이클의 특수한 경우입니다. Listing 15.18의 사이클은 P0() (라인 7와 8), P1() (라인 16와 17), P2() (라인 24, 25, 그리고 26), 그리고 다시 P0() (라인 7)를 거쳐 이루어집니다. exists 절이 이 사이클을 파악합니다: 1:r1=1은 라인 16에서의 `smp_load_acquire()` 이라인 8에서의 `smp_store_release()`에 의해 저장된 값을 반환했음을 의미하고, 1:r2=0는 라인 24에서의 `WRITE_ONCE()` 가 라인 17에서의 `READ_ONCE()`에 의해 반환된 값에 영향을 미치기엔 너무 늦게 이루어졌으며 마지막으로 2:r3=0가 라인 7에서의 `WRITE_ONCE()`는 라인 26에서의 `READ_ONCE()`에 의해 반환되는 값에 영향을 끼치기엔 너무 늦었음을 나타냅니다. 이 경우, exists 절이 발동될 수 있다는 사실은 이 사이클이 허가되었음을 의미합니다. 대조적으로, exists 절이 발동될 수 없는 경우에는 사이클이 금지 되었다고 말해집니다.

하지만 Listing 15.18의 라인 29에 있는 exists 절을 유지해야 한다면 어떻게 될까요? 한가지 방법은 P0()의 `smp_store_release()`를 Table 15.3의

Listing 15.19: W+WRC Litmus Test With More Barriers

```

1 C C-W+WRC+o-mb-o+a-o+o-+mb-
2
3 {}
4
5 P0(int *x, int *y)
6 {
7     WRITE_ONCE(*x, 1);
8     smp_mb();
9     WRITE_ONCE(*y, 1);
10 }
11
12 P1(int *y, int *z)
13 {
14     int r1;
15     int r2;
16
17     r1 = smp_load_acquire(y);
18     r2 = READ_ONCE(*z);
19 }
20
21 P2(int *z, int *x)
22 {
23     int r3;
24
25     WRITE_ONCE(*z, 1);
26     smp_mb();
27     r3 = READ_ONCE(*x);
28 }
29
30 exists(1:r1=1 / 1:r2=0 / 2:r3=0)

```

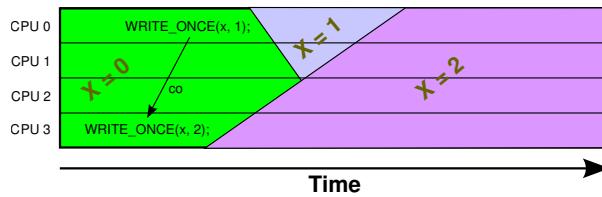


Figure 15.11: Store-to-Store is Counter-Temporal

Listing 15.20: 2+2W Litmus Test With Write Barriers

```

1 C C-2+2W+o-wmb-o+o-wmb-o
2
3 {}
4
5 P0(int *x0, int *x1)
6 {
7     WRITE_ONCE(*x0, 1);
8     smp_wmb();
9     WRITE_ONCE(*x1, 2);
10 }
11
12 P1(int *x0, int *x1)
13 {
14     WRITE_ONCE(*x1, 1);
15     smp_wmb();
16     WRITE_ONCE(*x0, 2);
17 }
18 exists (x0=1 /\ x1=1)

```

cumulativity 만이 아니라 전파 (propagation) 도 갖는다고 말하는 `smp_mb()`로 교체하는 것입니다. 그 결과는 Listing 15.19 (C-W+RWC+o-mb-o+a-o+o-mb-o.litmus)에 보여져 있습니다.

Quick Quiz 15.25: `smp_mb()` 가 전파 속성을 갖는데 왜 Listing 15.18 의 라인 25 에 있는 `smp_mb()` 는 왜 `exists` 절의 발동을 막지 않죠?



완벽성을 위해, Figure 15.11 는 동일한 변수로의 스토어 그룹들 가운데 “이기는” 스托어가 가장 나중에 시작된 스托어일 필요는 없음을 보입니다. 이는 page 305 의 Figure 15.5 를 주의 깊게 본 사람에겐 놀랍지 않을 겁니다.

Quick Quiz 15.26: 하지만 Listing 15.20 (C-2+2W+o-wmb-o+o-wmb-o.litmus) 에 보인 것처럼 순서잡힌 스托어만 갖는 리트머스 테스트에서, 연구자들은 Arm 과 Power 같은 완화된 순서 규칙의 시스템에서도 사이클이 금지됨을 보였습니다 [SSA+11]. 그런데 store-to-store 가 정말 항상 임시적이지 않나요???



하지만 가끔 시간은 우리 편입니다. 계속 읽으세요!

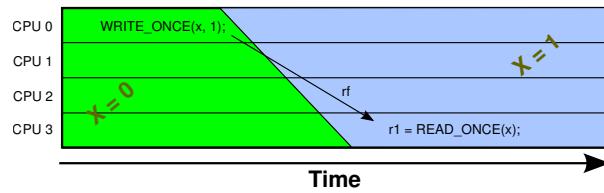


Figure 15.12: Store-to-Load is Temporal

Listing 15.21: LB Litmus Test With One Acquire

```

1 C C-LB+a-o+o-data-o+o-data-o
2
3 {}
4
5 P0(int *x0, int *x1)
6 {
7     int r2;
8
9     r2 = smp_load_acquire(x0);
10    WRITE_ONCE(*x1, 2);
11 }
12
13 P1(int *x1, int *x2)
14 {
15     int r2;
16
17     r2 = READ_ONCE(*x1);
18     WRITE_ONCE(*x2, r2);
19 }
20
21 P2(int *x2, int *x0)
22 {
23     int r2;
24
25     r2 = READ_ONCE(*x2);
26     WRITE_ONCE(*x0, r2);
27 }
28
29 exists (0:r2=2 /\ 1:r2=2 /\ 2:r2=2)

```

15.2.7.3 Happens-Before

Figure 15.12 에 보인 것처럼, 사용자에게 보이는 예측이 없는 플랫폼에서 로드가 특정 스托어로부터의 값을 리턴하면 빛의 유한한 속도와 현대 컴퓨팅 시스템의 크기 존재 덕분에 그 스托어는 로드가 행해진 것보다 이른 시간에 수행되었어야만 합니다. 이는 주의 깊게 짜여진 프로그램은 시간의 흐름 자체를 메모리 순서 오퍼레이션으로 사용하여 의존할 수 있음을 의미합니다.

물론, 스托어에서 로드로의 연결 외에는 아무것도 없는 page 306 의 Listing 15.6 에서 보이는 것처럼 시간의 흐름 그 자체만으로는 충분치 않은데, 이는 어떤 순서도 제공하지 않아서 여전히 `exists` 절을 발동시킬 수 있기 때문입니다. 그러나, 각 쓰레드가 가장 약한 순서만 제공할 때라도 `exists` 절은 발동될 수 없습니다. 예를 들어, Listing 15.21 (C-LB+a-o+o-data-o+o-data-o.litmus) 는 `P0()` 가 `smp_load_acquire()` 로 순서잡히고 `P1()` 과 `P2()` 가 모두 데이터 종속성으로 순서잡힌 걸 보입니다. Table 15.3 의 가장 위에 보인

Listing 15.22: Long LB Release-Acquire Chain

```

1 C C-LB+a-r+a-r+a-r+a-r
2
3 {}
4
5 P0(int *x0, int *x1)
6 {
7     int r2;
8
9     r2 = smp_load_acquire(x0);
10    smp_store_release(x1, 2);
11 }
12
13 P1(int *x1, int *x2)
14 {
15     int r2;
16
17     r2 = smp_load_acquire(x1);
18     smp_store_release(x2, 2);
19 }
20
21 P2(int *x2, int *x3)
22 {
23     int r2;
24
25     r2 = smp_load_acquire(x2);
26     smp_store_release(x3, 2);
27 }
28
29 P3(int *x3, int *x0)
30 {
31     int r2;
32
33     r2 = smp_load_acquire(x3);
34     smp_store_release(x0, 2);
35 }
36
37 exists (0:r2=2 /\ 1:r2=2 /\ 2:r2=2 /\ 3:r2=2)

```

Listing 15.23: Long ISA2 Release-Acquire Chain

```

1 C C-ISA2+o-r+a-r+a-r+a-o
2
3 {}
4
5 P0(int *x0, int *x1)
6 {
7     WRITE_ONCE(*x0, 2);
8     smp_store_release(x1, 2);
9 }
10
11 P1(int *x1, int *x2)
12 {
13     int r2;
14
15     r2 = smp_load_acquire(x1);
16     smp_store_release(x2, 2);
17 }
18
19 P2(int *x2, int *x3)
20 {
21     int r2;
22
23     r2 = smp_load_acquire(x2);
24     smp_store_release(x3, 2);
25 }
26
27 P3(int *x3, int *x0)
28 {
29     int r1;
30     int r2;
31
32     r1 = smp_load_acquire(x3);
33     r2 = READ_ONCE(*x0);
34 }
35
36 exists (1:r2=2 /\ 2:r2=2 /\ 3:r1=2 /\ 3:r2=0)

```

것과 비슷한 이 순서 규칙들은 `exists` 절이 발동되는 걸 막기 충분합니다.

Quick Quiz 15.27: Listing 15.21에 보인 것과 같은 종속성 만을 사용하는 리트머스 테스트를 만들 수 있습니까?

■
메모리 액세스를 순서잡기 위한 더 중요한 시간의 사용은 다음 섹션에서 다루어집니다.

15.2.7.4 Release-Acquire Chains

최소한의 release-acquire 연결이 page 307의 Listing 15.7에 보여 있습니다만, 이 연결은 Listing 15.22 (C-LB+a-r+a-r+a-r+a-r.a.litmus)에 보인 것처럼 훨씬 더 길어질 수 있습니다. 이 release-acquire 연결이 더 길어질수록, 시간의 흐름으로부터 더 많은 순서가 얻어져서 얼마나 많은 쓰레드가 연관되는지에 관계 없이 연관된 `exists` 절은 발동될 수 없습니다.

Release-acquire 연결은 본성적으로 스토어에서 로드로의 생성체이지만, 하나의 로드로부터 스토어로의 단계를 허용할 수 있음이, 그런 단계는 반-임시적이어야 하기 하지만, page 314의 Figure 15.10에 보인 것처럼 드

러났습니다. 예를 들어, Listing 15.23 (C-ISA2+o-r+a-r+a-r+a-o.litmus)는 세 단계 release-acquire 연결을 보입니다만, P3()의 마지막 액세스는 x0로부터의 `READ_ONCE()` 인데, 이는 P0()의 `WRITE_ONCE()`에 의해 액세스되는 것이어서 반-임시적인 로드에서 스토어로의 연결을 이 두 프로세스 사이에 만듭니다. 그러나, P0()의 `smp_store_release()` (라인 8)은 cumulative 하므로, P3()의 `READ_ONCE()`가 0을 리턴했다면, 이 cumulativeity는 `READ_ONCE()`가 P0()의 `smp_store_release()` 전으로 순서 잡힐 것을 강제합니다. 더해서, 이 release-acquire 연결은 (라인 8, 15, 16, 23, 24, 그리고 32) P3()의 `READ_ONCE()`가 P0()의 `smp_store_release()` 뒤로 순서잡기 강제합니다. P3()의 `READ_ONCE()`는 P0()의 `smp_store_release()` 전후에 다 있을 수는 없으므로, 둘 중 하나는 사실이어야 합니다:

1. P3()의 `READ_ONCE()`는 P0()의 `WRITE_ONCE()` 다음에 왔어서, `READ_ONCE()`는 값 2를 반환했어서, `exists` 절의 `3:r2=0`는 거짓입니다.

Listing 15.24: Long Z6.2 Release-Acquire Chain

```

1 C C-Z6.2+o-r+a-r+a-r+a-o
2
3 {}
4
5 P0(int *x0, int *x1)
6 {
7   WRITE_ONCE(*x0, 2);
8   smp_store_release(x1, 2);
9 }
10
11 P1(int *x1, int *x2)
12 {
13   int r2;
14
15   r2 = smp_load_acquire(x1);
16   smp_store_release(x2, 2);
17 }
18
19 P2(int *x2, int *x3)
20 {
21   int r2;
22
23   r2 = smp_load_acquire(x2);
24   smp_store_release(x3, 2);
25 }
26
27 P3(int *x3, int *x0)
28 {
29   int r2;
30
31   r2 = smp_load_acquire(x3);
32   WRITE_ONCE(*x0, 3);
33 }
34
35 exists (1:r2=2 /\ 2:r2=2 /\ 3:r2=2 /\ x0=2)

```

2. 이 release-acquire 연결은 이루어지지 않아서, 즉 exists 절의 $1:r2=2$, $2:r2=2$, 또는 $3:r1=2$ 는 거짓입니다.

어느 쪽이든, exists 절은 발동될 수 없는데, 이 리트머스 테스트가 악명높은 로드에서 스토어로의 연결을 P3() 와 P0() 사이에 가지고 있음에도 그렇습니다. 그러나 release-acquire 연결은 Listing 15.18에 보인 것처럼 하나의 로드에서 스토어로의 연결만을 가질 수 있음을 결코 잊지 마십시오.

Release-acquire 연결은 또한 Listing 15.24 (C-Z6.2+o-r+a-r+a-r+a-o.litmus)에 보인 것처럼 하나의 스토어에서 스토어로의 단계 하나를 허용할 수 있습니다. 앞의 예에서와 같이, smp_store_release()의 cumulativity가 release-acquire 연결이 임시적 본성과 결합되어 라인 35의 exists 절이 발동되는 것을 막습니다. 하지만 주의하세요: 두번째 스토어에서 스토어로의 단계를 더하는 것은 연관된 업데이트된 exists 절이 발동되는 걸 허가할 수 있습니다.

Quick Quiz 15.28: Listing 15.25에 보인 것처럼 우리가 하나의 로드에서 스토어로의 연결과 하나의 스토어에서 스토어로의 링크와 함께 짧은 release-acquire 연결을 갖는다고 해봅시다. 스토어에서 로드로의 연결이

Listing 15.25: Z6.0 Release-Acquire Chain (Ordering?)

```

1 C C-Z6.2+o-r+a-o+o-mb-o
2
3 {}
4
5 P0(int *x, int *y)
6 {
7   WRITE_ONCE(*x, 1);
8   smp_store_release(y, 1);
9 }
10
11 P1(int *y, int *z)
12 {
13   int r1;
14
15   r1 = smp_load_acquire(y);
16   WRITE_ONCE(*z, 1);
17 }
18
19 P2(int *z, int *x)
20 {
21   int r2;
22
23   WRITE_ONCE(*z, 2);
24   smp_mb();
25   r2 = READ_ONCE(*x);
26 }
27
28 exists(1:r1=1 /\ 2:r2=0 /\ z=2)

```

아닌 종류의 것은 하나씩만 있으므로 exists는 발동될 수 없습니다, 맞죠?

Quick Quiz 15.29: 스토어에서 로드로의 연결, 로드에서 스토어로의 연결, 그리고 스토어에서 스토어로의 연결이 있습니다. 하지만 로드에서 로드로의 연결은 어려울까요?

요약하자면, 올바르게 구성된 release-acquire 연결은 더 복잡한 메모리 순서 규약의 상당히 반직관적인 바다에 의해 둘러싸인 직관적 행복의 평화로운 섬을 형성합니다.

15.3 Compile-Time Consternation

Science increases our power in proportion as it lowers our pride.

Claude Bernard

C를 포함해 대부분의 언어는 복잡한 프로그래밍 경험에 없거나 조금만 있는 사람들에 의해 단일 프로세서 시스템에서 개발되었습니다. 그 결과, 명시적으로 달리 말해지지 않았다면 이 언어들은 현재 CPU가 메모리를 읽고 쓰는 유일한 것이라고 가정합니다. 이는 결국 이 언어들의 컴파일러의 최적화기가 여러분의 프로그램이 수행하는 메모리 참조의 순서, 수, 그리고 크기에 극적인

변화를 만들 준비가 되어있고 하려 한다는 것을 의미합니다. 실제로, 하드웨어에 의해 수행되는 순서 재배치는 상대적으로 온순한 것입니다.

이 섹션은 여러분이 컴파일러를 길들여서 컴파일 시점의 놀림을 막는 것을 돋습니다. Section 15.3.1은 컴파일러가 여러분의 코드의 메모리 참조를 파괴적으로 최적화하는 걸 막는지 설명하고, Section 15.3.2는 주소와 데이터 종속성을 보호하는 법을 설명하며, 마지막으로 Section 15.3.3은 그 미묘한 제어 종속성을 보호하는 법을 설명합니다.

15.3.1 Memory-Reference Restrictions

Section 4.3.4에서 이야기 되었듯, 별달리 이야기 되지 않는다면 컴파일러는 그 코드가 접근하는 변수에 다른 것들은 영향을 끼치지 않는다고 가정합니다. 더 나아가서, 이 가정은 단순한 설계 오류가 아니라 여러 표준에서 소중히 간직되는 것입니다.¹¹ 다음 섹션에 대한 준비로 이를 요약해 둘 가치가 있습니다.

C-언어에서의 “`r1 = a`” 또는 “`b = 1`”과 같은 할당문으로 만들어지는 평범한 액세스는 Section 4.3.4.1에서 설명한 공유 변수에 대한 실수입니다. 이런 실수를 막는 방법들은 page 43에서 시작하는 Sections 4.3.4.2–4.3.4.4에 설명되어 있습니다:

1. 평범한 액세스는 조개질 수 있는데, 예를 들어 컴파일러는 8 바이트 포인터를 한번에 한 바이트씩 액세스할 수 있습니다. 정렬된 기계 크기 액세스를 조개는 것은 `READ_ONCE()`와 `WRITE_ONCE()`를 사용해 막아질 수 있습니다.
2. 평범한 로드는 융합될 수 있는데, 예를 들어 같은 객체로부터의 앞선 로드의 결과가 여전히 기계 레지스터에 있다면, 컴파일러는 이를 메모리에서 다시 로드하는 대신 이 레지스터 내의 값을 재사용하는 식으로 최적화를 할 수도 있습니다. 로드 융합은 `READ_ONCE()`를 사용하거나 `barrier()`, `smp_rmb()`, 또는 Table 15.3에서 보인 다른 방법들을 이용해 순서를 강제함으로써 방지될 수 있습니다.
3. 평범한 스토어는 융합될 수 있어서 같은 변수로의 뒤따른 스토어가 있다면 앞의 스토어는 완전히 제거될 수 있습니다. 스토어 융합은 `WRITE_ONCE()`를 사용하거나 `barrier()`, `smp_wmb()`, 또는 Table 15.3에서 보인 다른 방법들을 이용해 순서를 강제함으로써 방지될 수 있습니다.
4. 평범한 액세스는 현대의 최적화 기능을 갖춘 컴파일러에 의해 놀라운 방법으로 재배치될 수 있습니다.

¹¹ 또는 어쩌면 이는 표준 설계 오류입니다.

다. 이 재배치는 앞에서 설명한 방법으로 순서를 강제해 막을 수 있습니다.

5. 평범한 로드는 만들어질 수 있는데, 예를 들어 레지스터 부족은 컴파일러가 앞서 로드된 값을 레지스터에서 지우고 나중에 다시 로드하게 만들 수 있습니다. 만들어진 로드는 `READ_ONCE()`를 사용하거나 해당 로드와 뒤따르는 그 변수의 값 사이에 `barrier()`를 사용해서 앞에 설명한 방법들로 순서를 강제해 막을 수 있습니다.
6. 스토어는 평범한 스토어 앞에 만들어질 수 있는데, 예를 들어 써질 위치를 임시 저장소로 사용하는 겁니다. 이는 `WRITE_ONCE()`의 사용으로 방지할 수 있습니다.

Quick Quiz 15.30: 컴파일러가 스토어를 만들어내는 걸 막기 위해 `barrier()` 호출을 평범한 스토어 직전에 위치시키는 건 어떤가요?



이 모든 공유 메모리 실수는 Section 4.3.4.4에서 설명한 것처럼 평범한 액세스 사이의 데이터 경주를 막음으로써 막아질 수 있습니다. 어쨌건, 데이터 경주가 없다면 앞서 이야기된 모든 컴파일러 최적화는 완전 안전합니다. 그러나 데이터 경주를 포함하는 코드에서 이 리스트는 컴파일러 최적화는 갈수록 적극적이 될 것이기 때문에 별다른 알림 없이 바뀔 수 있습니다.

요약해서, `READ_ONCE()`, `WRITE_ONCE()`, `barrier()`, `volatile`, 그 외에 page 301의 Table 15.3에서 이야기된 다른 기능들은 컴파일러가 여러분의 별별 알고리즘을 존재치 않게 최적화 해버리는 걸 막는 가치있는 도구들입니다. 컴파일러는 로드와 스토어 조개기를 막는 다른 메커니즘도 지원하기 시작했는데, 예를 들어 `memory_order_relaxed` 어토믹 로드와 스토어인데, 그러나 여전히 작업이 필요합니다 [Cor16b]. 또한, 컴파일러 이슈 외에도 `volatile`은 C11 어토믹 액세스를 포함한 액세스의 합침과 만들어냄을 막기 위해 여전히 필요합니다.

`READ_ONCE()`와 `WRITE_ONCE()`를 과하게 사용하게 될 수 있음을 알아두십시오. 예를 들어, 여러분이 특정 변수가 변하는 걸 막았다면 (그 변수로의 모든 업데이트를 락으로 보호하는 식으로), `READ_ONCE()`를 사용할 이유가 없습니다. 비슷하게, 다른 CPU나 쓰레드가 특정 변수를 읽는 것을 막았다면 (그 변수를 다른 CPU나 쓰레드가 그걸 액세스 하기 전에 최적화 한다거나 하는 식으로), `WRITE_ONCE()`를 사용할 이유가 없습니다. 그러나, 제 경험상 개발자들은 그들이 생각하는 것보다 더 자주 `READ_ONCE()`와 `WRITE_ONCE()`를 사용해야 하며, 불필요한 사용으로 인한 오버헤드는 매우 낮습니다. 반대로, 필요한 곳에 그것을 사용하지 않음에 따른 불이익은 매우 큼 수 있습니다.

15.3.2 Address- and Data-Dependency Difficulties

컴파일러는 주소나 데이터 종속성을 이해하지 못합니다만, 그걸 가르치려는, 또는 최소한 그걸 가르치는 방법을 표준화하려는 노력은 있긴 합니다 [MWB⁺17, MRP⁺17]. 그 사이에는 여러분의 컴파일러가 여러분의 종속성을 깨버리는 걸 막기 위해 주의를 기울일 필요가 있습니다.

15.3.2.1 Give your dependency chain a good start

여러분의 종속성 연결을 시작하는 로드는 적절한 순서를 지켜야 하는데, 예를 들면 `rcu_dereference()`나 `READ_ONCE()`를 사용하는 겁니다. 이 규칙을 지키는데 실패하면 심각한 부작용이 있을 수 있습니다:

1. Section 15.5.1에서 이야기 되었듯, DEC Alpha에서, 종속적 로드는 종속성 연결을 시작하는 로드에 대해 순서잡히지 않을 수도 있습니다.
2. 종속성 연결을 시작하는 로드가 C11 `non-volatile memory_order_relaxed` 로드라면, 컴파일러는 그 로드를 제거할 수 있는데, 예를 들면 과거에 로드된 값을 사용하는 겁니다.
3. 종속성 연결을 시작하는 로드가 평범한 로드라면, 컴파일러는 이 로드를 제거할 수 있는데, 앞에서와 마찬가지로 과거에 로드된 값을 사용하는 식입니다. 더 나쁜 것이, 로드를 한번이 아니라 두번 행할 수 있어서, 여러분의 코드의 다른 부분이 다른 값을 보게 할 수 있습니다—그리고 컴파일러는 정말로 이 일을 하는데, 특히 레지스터가 부족할 때 그렇습니다.
4. 종속성 연결의 시작에서 로드된 값은 포인터여야만 합니다. 이론상으로는, 그렇습니다, 여러분은 정수형을 로드할 수 있는데, 아마도 배열 인덱스로 이를 사용하기 위해서일 겁니다. 현실에서는, 컴파일러는 정수형에 대해 너무 많은 걸 알아서, 여러분의 종속성 연결을 깨부실 너무 많은 기회를 갖고 있습니다 [MWB⁺17].

15.3.2.2 Avoid arithmetic dependency breakage

여러분의 종속성 연결 내에서 포인터에 대해 어떤 산술 오퍼레이션을 행하는게 괜찮긴 하지만, 컴파일러에게 너무 많은 정보를 주는 건 조심해서 막아야 합니다. 어쨌건, 컴파일러가 이 포인터의 정확한 값을 판단하기 충분하게 학습된다면, 컴파일러는 그 포인터 자체가

Listing 15.26: Breakable Dependencies With Comparisons

```

1 int reserve_int;
2 int *gp;
3 int *p;
4
5 p = rcu_dereference(gp);
6 if (p == &reserve_int)
7     handle_reserve(p);
8 do_something_with(*p); /* buggy! */

```

Listing 15.27: Broken Dependencies With Comparisons

```

1 int reserve_int;
2 int *gp;
3 int *p;
4
5 p = rcu_dereference(gp);
6 if (p == &reserve_int) {
7     handle_reserve(&reserve_int);
8     do_something_with(reserve_int); /* buggy! */
9 } else {
10     do_something_with(*p); /* OK! */
11 }

```

아니라 그 정확한 값을 대신 사용할 수 있습니다. 컴파일러가 그렇게 되자마자 종속성은 깨지고 모든 순서는 사라집니다.

1. 포인터로부터 오프셋을 계산하는건 허용되지만 이 오프셋은 완전히 취소되지 않아야만 합니다. 예를 들어, 어떤 `char` 포인터 `cp` 가 있다면, `cp-(uintptr_t)cp` 는 오프셋을 취소시키고 컴파일러가 이 종속성 연결을 깨게 할 수 있습니다. 다른 한편, 오프셋 값을 서로와 함께 취소시키는건 완전히 안전하고 합법적입니다. 예를 들어, `a` 와 `b` 가 동일하다면 `cp+a-b` 는 동일 함수로, 종속성을 유지합니다.
2. 비교는 종속성을 깰 수 있습니다. Listing 15.26이 이게 어떻게 일어날 수 있는지 보입니다. 여기서 전역 포인터 `gp` 는 동적으로 할당된 정수를 가리키지만 메모리가 부족하면 `reserve_int` 변수를 가리킬 수도 있습니다. 이 `reserve_int`의 경우는 이 리스트의 라인 6와 7에서 보이듯 특수한 처리가 필요할 수도 있습니다. 하지만 컴파일러는 이 코드를 Listing 15.27에 보인 합리적인 형태로 변환할 수도 있는데, 절대 주소를 갖는 명령이 레지스터를 통해 제공되는 주소를 사용하는 명령보다 빠른 시스템에서 그렇습니다. 그러나, 라인 5와 라인 8의 역참조 사이에는 명확한 순서 규칙이 없습니다. 이는 단순한 하나의 예일 뿐임을 알아 두십시오: 비교를 통해 종속성 연결을 깨는 수많은 방법이 존재합니다.

Quick Quiz 15.31: Listing 15.26 의 라인 6에서 `&reserve_int` 와 비교하기 전에 포인터를 그냥 역참조하지 않는 이유가 뭐죠?

Quick Quiz 15.32: 하지만 두 포인터 변수를 비교하는건 안전합니다, 그렇죠? 어쨌건, 컴파일러는 두 변수의 값을 모르는데 그 비교에서 뭘 알 수 있겠습니까?

연속된 동일성 비교는 함께 취해졌을 경우 컴파일러에게 포인터의 정확한 값을 판단하기 충분한, 따라서 종속성을 깔 수 있는 정보를 줄 수도 있음을 명심하십시오. 더 나아가, 컴파일러는 하나의 동일성 비교로부터의 정보를 다른 정보와 결합해 정확한 값을 알아낼 수 있어서 또다시 종속성을 깔 수 있습니다. 배열의 원소로의 포인터는 특히 이 나중 부분의 종속성 깨짐에 취약합니다.

15.3.2.3 Safe comparison of dependent pointers

종속된 포인터들을 안전하게 비교하는 방법들이 있습니다:

1. NULL 포인터에 대한 비교. 이 경우, 컴파일러가 알 수 있는 모든 것은 포인터가 NULL이라는 것 뿐으로, 어차피 여러분은 이를 역참조할 수 없습니다.
2. 종속된 포인터가 비교 전이든 후든 결코 역참조되지 않는 경우.
3. 종속된 포인터가 매우 오래전에 변경된 객체를 참조하는 포인터와 비교되는 경우로, 여기서 “매우 오래 전”이라 하기 무조건적으로 안전한 값은 “컴파일 시점” 뿐. 핵심은 주소나 데이터 종속성이 이외의 무언가가 순서를 보장할 수 있다는 것.
4. 각각 적절한 종속성을 갖는 두개의 포인터 사이의 비교. 예를 들어, 각자 락을 포함하는 데이터 구조로의 종속성을 갖는 한쌍의 포인터가 있고 주소 순서대로 락을 잡아서 데드락을 회피하고자 하는 경우.
5. 비교가 같지 않고, 컴파일러는 종속성을 이끄는 포인터의 값을 추측할 정보를 갖고 있지 않은 경우.

포인터 비교는 상당히 복잡할 수 있으며, 따라서 Listing 15.28 의 예를 따라가보는 게 좋겠습니다. 이 예는 라인 1-5 의 간단한 `struct foo` 와 라인 6 와 7 에 각각 보인 두개의 전역 포인터 `gp1` 과 `gp2` 를 사용합니다. 이 예는 두 쓰레드를 사용하는데, 라인 9-22 의 `updater()` 와 라인 24-39 의 `reader()` 입니다.

`updater()` 쓰레드는 라인 13에서 메모리를 할당하고 할당에 실패하면 라인 14에서 불만을 표합니다.

Listing 15.28: Broken Dependencies With Pointer Comparisons

```

1  struct foo {
2      int a;
3      int b;
4      int c;
5  };
6  struct foo *gp1;
7  struct foo *gp2;
8
9  void updater(void)
10 {
11     struct foo *p;
12
13     p = malloc(sizeof(*p));
14     BUG_ON(!p);
15     p->a = 42;
16     p->b = 43;
17     p->c = 44;
18     rcu_assign_pointer(gp1, p);
19     WRITE_ONCE(p->b, 143);
20     WRITE_ONCE(p->c, 144);
21     rcu_assign_pointer(gp2, p);
22 }
23
24 void reader(void)
25 {
26     struct foo *p;
27     struct foo *q;
28     int r1, r2 = 0;
29
30     p = rcu_dereference(gp2);
31     if (p == NULL)
32         return;
33     r1 = READ_ONCE(p->b);
34     q = rcu_dereference(gp1);
35     if (p == q) {
36         r2 = READ_ONCE(p->c);
37     }
38     do_something_with(r1, r2);
39 }
```

라인 15-17는 새로 할당된 구조체를 초기화 하며, 이어서 라인 18에서 이 포인터를 `gp1`에 할당합니다. 라인 19와 20는 이어서 이 구조체의 필드 중 두개를 업데이트하는데, 라인 18이 그 필드들을 읽기 쓰레드에게 보이게 한 후입니다. 읽기 쓰레드에게 보이는 필드를 비동기적으로 업데이트 하는 것은 종종 버그임을 알아두세요. 그냥 이를 하는 합리적 사용처가 있지만, 그런 경우는 이 예에서보다 깊은 주의가 필요합니다.

마지막으로 라인 21는 이 포인터를 `gp2`에 할당합니다.

`reader()` 쓰레드는 먼저 `gp2`를 라인 30에서 가져오고, 라인 31와 32에서 이를 NULL과 비교한 후 동일하면 리턴합니다. 라인 33는 `->b` 필드를 가져오고 라인 30은 `gp2`를 가져옵니다. 라인 35가 이 라인 30과 30에서 가져온 포인터가 동일하다고 보게 되면, 라인 36는 `p->c`를 가져옵니다. 라인 36는 라인 30에서 가져온 포인터 `p`를 사용하지 라인 34에서 가져온 포인터 `q`를 사용하지 않음에 유의하세요.

그러나 이 차이는 문제가 되지 않을 수도 있습니다. 라인 35에서의 동일성 검사는 컴파일러가 (옳지 않게

도) 두 포인터가 동일하다고 결론내리게 할 수 있습니다, 실제로는 이것들이 다른 종속성을 가짐에도 불구하고 말입니다. 이는 컴파일러가 라인 36는 $r2 - q \rightarrow c$ 로 변환하게 할 수도 있는데, 이는 예상된 값 144가 아닌 값 44를 로드되게 할 수 있습니다.

Quick Quiz 15.33: 하지만 라인 35에서의 조건은 라인 36를 라인 34 다음으로 순서짓는 제어 종속성을 제공하지 않나요?

■

요약하자면, 여러분의 소스코드의 종속성 연결이 전혀 컴파일러에 의해 생성된 어셈블리 코드에서도 종속성 연결이 되기 위해선 큰 주의가 필요합니다.

15.3.3 Control-Dependency Calamities

제어 종속성은 특히 복잡한데, 현재의 컴파일러는 그걸 이해하지 못하고 쉽게 그걸 깨버릴 수 있기 때문입니다. 이 섹션의 규칙과 예제들은 여러분이 여러분의 컴파일러가 이를 무시하고 여러분의 코드를 깨부수는 것을 막는 것을 돋고자 작성되었습니다.

로드-로드 제어 종속성은 단순한 데이터 종속성 배리어가 아니라 완전한 읽기 메모리 배리어를 필요로 합니다. 다음 코드를 보세요:

```

1 q = READ_ONCE(x);
2 if (q) {
3     <data dependency barrier>
4     q = READ_ONCE(y);
5 }
```

이는 뜻한 효과를 내지 못하는데, 실제 데이터 종속성이 존재하지 않으며 CPU는 그 결과를 먼저 예측해서 수행을 단축시킬 수 있는 제어 종속성을 가지고 있어서, 다른 CPU는 y로부터의 로드를 x 로드 전에 일어난 것으로 볼 수 있습니다. 그런 경우 정말로 필요한건 다음과 같습니다:

```

1 q = READ_ONCE(x);
2 if (q) {
3     <read barrier>
4     q = READ_ONCE(y);
5 }
```

그러나, 스토어는 예측되지 않습니다. 이는 순서가 정말로 로드-스토어 제어 종속성에는 다음 예와 같이 제공됨을 뜻합니다:

```

1 q = READ_ONCE(x);
2 if (q)
3     WRITE_ONCE(y, 1);
```

제어 종속성은 일반적으로 다른 종류의 순서규칙 오퍼레이션과 짹을 이룹니다. 그러나, READ_ONCE()

도 WRITE_ONCE() 도 선택사항이 아님을 알아두세요! READ_ONCE() 가 없다면 컴파일러는 x 로드를 다른 x 로드와 융합시킬 수도 있습니다. WRITE_ONCE() 가 없다면 컴파일러는 y 로의 스토어를 다른 y 로의 스토어와 융합시킬 수도 있습니다. 둘 다 상당히 반직관적인 순서 효과를 낼 수 있습니다.

더 나쁜 것이, 컴파일러가 (예를 들어) 변수 x의 값이 항상 0이 아님을 증명할 수 있다면, 원래의 예를 “if” 문을 다음과 같이 제거할 권리를 갖습니다:

```

1 q = READ_ONCE(x);
2 WRITE_ONCE(y, 1); /* BUG: CPU can reorder!!! */
```

다음과 같이 “if” 문의 양 분기문의 동일한 스토어에 순서를 강제하고 싶을 겁니다:

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 }
```

불행히도, 현재의 컴파일러는 높은 최적화 단계에서는 이를 다음과 같이 변환할 수 있습니다:

```

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();
```

이제 x로부터의 로드와 y로의 스토어 사이에는 조건이 없어지며, 이는 CPU가 이들을 재배치할 권리를 가짐을 뜻합니다: 그 조건은 분명 필요하며, 모든 컴파일러 최적화가 적용된 뒤에도 어셈블리 코드 상에 존재해야만 합니다. 따라서, 이 예에서 순서가 필요하다면, 여러분은 명시적으로 메모리 순서 오퍼레이션을 추가해야 하는데, 예를 들면 release store입니다:

```

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 }
```

초기의 READ_ONCE()는 컴파일러가 x의 값을 추측하는 걸 막기 위해 여전히 필요합니다. 또한, 지역 변수 q를 가지고 하는 일에도 주의를 기울여야 하는데, 그렇지 않으면 컴파일러는 그 값을 추측하고 다시 필요한 조건을 없애버릴 수 있습니다. 예를 들면:

```

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 }
```

만약 MAX가 1로 정의되어 있다면, 컴파일러는 ($q \% \text{MAX}$)가 0과 같다는 걸 알아서, 컴파일러는 앞의 코드를 다음과 같이 변환할 권리가 있습니다:

```

1 q = READ_ONCE(x);
2 WRITE_ONCE(y, 2);
3 do_something_else();
```

이 변환된 코드에서 CPU는 x로부터의 로드와 y로의 스토어 사이에 순서를 지켜야 할 필요가 없습니다. 컴파일러를 제약하기 위해 barrier()를 추가하고 싶겠지만 이는 도움이 되지 않습니다. 조건은 사라졌고 barrier()는 이를 되돌리지 않습니다. 따라서, 여러분이 이 순서에 기대고 있다면, 아마도 다음과 같이 MAX가 1보다 크다는 걸 보장해야 합니다:

```

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 }
```

y로의 스토어는 다음을 다시 주의하세요. 이것들이 동일하다면, 앞에서 언급한 바와 같이 컴파일러는 이 스토어들을 “if” 문 바깥으로 가져올 수 있습니다.

여러분은 또한 이전 회로 평가에 지나치게 기대는 걸 막아야 합니다. 이 예를 생각해 보세요:

```

1 q = READ_ONCE(x);
2 if (q || 1 > 0)
3     WRITE_ONCE(y, 1);
```

첫번째 조건은 두번째 조건이 항상 참인 관계로 거짓일 수 없으므로, 컴파일러는 이를 다음과 같이 변환시켜 제어 종속성을 제거할 수 있습니다:

```

1 q = READ_ONCE(x);
2 WRITE_ONCE(y, 1);
```

이 예는 컴파일러가 여러분의 코드를 추측할 수 없게 보장할 필요를 강조합니다. 더 일반적으로, READ_ONCE()가 컴파일러에게 특정 로드를 위한 코드를 정밀 생성하게 강제하지만, 컴파일러가 그 값을 로드하게끔 강제하지는 않습니다.

또한, 제어 종속성은 if 문의 then 절과 else 절에만 적용됩니다. 특히, if 문을 뒤따르는 코드에는 적용되지 않을 수 있습니다.

```

1 q = READ_ONCE(x);
2 if (q)
3     WRITE_ONCE(y, 1);
4 else
5     WRITE_ONCE(y, 2);
6 WRITE_ONCE(z, 1); /* BUG: No ordering. */
```

컴파일러는 volatile 액세스를 재배치 할 수 없으며 조건 내의 y로의 쓰기를 재배치할 수도 없다고 주장하고 싶을 겁니다. 불행히도 이 경우, 컴파일러는 y로의 두개의 쓰기를 조건적 이동 명령으로 변환해서 다음과 같은 수도-어셈블리어를 만들어낼 수 있습니다:

```

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
```

완화된 순서규칙의 CPU는 x로부터의 로드와 z로의 스토어 사이에 어떤 종속성도 갖지 않습니다. 이 제어 종속성은 한쌍의 cmov 명령과 거기 종속된 스토어까지만 확장됩니다. 요약하면, 제어 종속성은 해당 “if”的 “then”과 “else”에만 (이 두 절에서 호출되는 함수 포함) 적용되지, “if”를 뒤따르는 코드까지는 아닙니다.

마지막으로, 제어 종속성은 cumulativity를 제공하지 않습니다.¹² 이는 두개의 연관된 리트머스 테스트, 즉 x와 y가 둘 다 0으로 초기화 되는 Listings 15.29 and 15.30를 통해 선보였습니다.

Listing 15.29 (C-LB+o-cgt-o+o-cgt-o.litmus)의 두개 쓰레드 예에서의 exists 절은 결코 발동되지 않을 겁니다. 제어 종속성이 cumulativity를 보장한다면 (사실은 보장하지 않음), Listing 15.30 (C-WWC+o-cgt-o+o-cgt-o+o.litmus)처럼 쓰레드를 하나 더 추가하는 것이 연관된 exists 절이 발동되지 않게끔 보장할 겁니다.

그러나 제어 종속성은 cumulativity를 제공하지 않으므로, 세개 쓰레드 리트머스 테스트의 exists 절은 발동될 수 있습니다. 세개 쓰레드 예가 순서를 제공해야 한다면, 여러분은 P0()의 로드와 스토어 사이, 즉 “if”문 직전 또는 직후에 smp_mb()를 추가해야 합니다. 더 나아가서, 원래의 두개 쓰레드 예는 잘못되기 쉬우니 사용되지 않아야 합니다.

¹² Cumulativity의 의미를 위해선 Section 15.2.7.1를 참고하세요.

Listing 15.29: LB Litmus Test With Control Dependency

```

1 C C-LB+o-cgt-o+o-cgt-o
2
3 {}
4
5 P0(int *x, int *y)
6 {
7     int r1;
8
9     r1 = READ_ONCE(*x);
10    if (r1 > 0)
11        WRITE_ONCE(*y, 1);
12 }
13
14 P1(int *x, int *y)
15 {
16     int r2;
17
18     r2 = READ_ONCE(*y);
19     if (r2 > 0)
20         WRITE_ONCE(*x, 1);
21 }
22
23 exists (0:r1=1 /\ 1:r2=1)

```

Listing 15.30: WWC Litmus Test With Control Dependency
(Cumulativity?)

```

1 C C-WWC+o-cgt-o+o-cgt-o+o
2
3 {}
4
5 P0(int *x, int *y)
6 {
7     int r1;
8
9     r1 = READ_ONCE(*x);
10    if (r1 > 0)
11        WRITE_ONCE(*y, 1);
12 }
13
14 P1(int *x, int *y)
15 {
16     int r2;
17
18     r2 = READ_ONCE(*y);
19     if (r2 > 0)
20         WRITE_ONCE(*x, 1);
21 }
22
23 P2(int *x)
24 {
25     WRITE_ONCE(*x, 2);
26 }
27
28 exists (0:r1=2 /\ 1:r2=1 /\ x=2)

```

Quick Quiz 15.34: Listing 15.30 의 P1()에 `smp_mb()`를 대신 추가할 순 없나요?

■ 다음 규칙들의 리스트가 이 섹션의 교훈들을 요약합니다:

1. 컴파일러는 제어 종속성을 이해하지 못하며, 따라서 컴파일러가 여러분의 코드를 망가뜨리지 못하게 하는건 여러분의 일입니다.
2. 제어 종속성은 앞의 로드를 뒤의 스토어에 대해 순서잡습니다. 그러나, 그것은 어떤 다른 종류의 순서도 보장하지 않습니다. 앞의 로드를 뒤의 로드에 대해서도, 앞의 스토어를 뒤의 어떤 것에 대해서도 순서잡지 않습니다. 이런 다른 형태의 순서가 필요하다면, `smp_rmb()`, `smp_wmb()`, 또는 앞의 스토어와 뒤의 로드 사이에의 순서라면 `smp_mb()`를 사용하십시오.
3. “if” 문의 양쪽 분기가 동일한 변수로의 같은 스토어로 시작한다면 제어 종속성은 그 스토어들을 순서잡지 않을 겁니다. 순서가 필요하다면, 그것들 앞에 `smp_mb()`를 두거나 `smp_store_release()`를 사용하세요. “if” 문의 양 분기의 시작에 `barrier()`를 두는 것은 충분치 않은데, 앞의 예에서 보인바와 같이 컴파일러의 최적화는 `barrier()`의 규칙을 지키면서도 제어 종속성을 파괴할 수 있기 때문임을 유의하십시오.
4. 제어 종속성은 앞의 로드와 뒤의 스토어 사이에 최소 하나의 수행 시간 조건을 필요로 하며, 이 조건은 앞의 로드와 연관되어야만 합니다. 컴파일러가 조건을 최적화 해서 없앨 수 있다면, 이는 순서 역시 최적화 해서 없앨 겁니다. 주의 깊은 `READ_ONCE()`와 `WRITE_ONCE()`의 사용은 필요한 조건을 지키는 걸 도울 수 있습니다.
5. 제어 종속성은 컴파일러가 종속성을 존재치 않게 재배치 하는 걸 막을 필요를 갖습니다. 주의 깊은 `READ_ONCE()`, `atomic_read()`, 또는 `atomic64_read()`의 사용은 여러분의 제어 종속성을 지키는 걸 도울 수 있습니다.
6. 제어 종속성은 이 종속성을 포함한 “if” 문의 “then”과 “else” 결과 이 두 절에서 호출하는 함수들에까지만 적용됩니다. 제어 종속성은 “if” 문의 끝을 뒤따르는 코드에는 적용되지 않습니다.
7. 제어 종속성은 일반적으로 다른 종류의 메모리 순서 오퍼레이션과 짹을 이룹니다.

8. 제어 종속성은 cumulativity를 제공하지 않습니다. Cumulativity가 필요하다면, 이를 제공하는 무언가, `smp_store_release()` 또는 `smp_mb()` 같은 걸 사용하십시오.

다시 말하지만, 많은 대중적 언어들이 싱글쓰레드 기반으로 설계되었습니다. 이 언어들을 멀티쓰레드 기반으로 성공적으로 사용하기 위해선 여러분이 메모리 참조와 종속성에 특별한 주의를 기울일 것을 요합니다.

15.4 Higher-Level Primitives

Method will teach you to win time.

Johann Wolfgang von Goethe

Section 12.3.1의 quick quiz 중 하나의 답은 높은 단계의 추상화로 모델링된 프로그램의 검증으로 인한 폭발적 속도 향상을 보였습니다. 이 섹션은 얼마나 더 높은 단계의 추상화가 동기화 기능 자체에 대한 더 깊은 이해를 제공할 수 있는지 알아봅니다. Section 15.4.1는 메모리 할당을 알아보고 Section 15.4.2는 RCU 까지 더 깊이 파고듭니다.

15.4.1 Memory Allocation

Section 6.4.3.2는 메모리 할당을 다루었고, 이 섹션은 연관된 메모리 순서 문제까지 이를 확장합니다.

핵심 요구사항은 어떤 메모리 블록이 할당 해제되기 전까지 해당 블록에 수행된 모든 액세스는 그 블록이 재할당 된 후에 수행된 모든 액세스보다 앞으로 순서를 잡혀야 한다는 겁니다. 할당 해제 전에 수행된 스토어가 재할당을 뒤따르는 스토어 다음으로 재배치 된다면 그건 어쨌건 잔인하고 비상식적인 메모리 할당기 버그일 겁니다. 그러나, 개발자에게 동적으로 할당된 메모리를 접근하는데 `READ_ONCE()`와 `WRITE_ONCE()`를 사용하도록 요구하는 것 역시 잔인하고 비상식적일 겁니다. 따라서 Section 4.3.4.1에서 이야기된 공유 변수 비극에도 불구하고 완전한 순서는 평범한 액세스에도 제공되어야 합니다.

물론, 각 CPU는 자신의 액세스는 순서대로 보고 컴파일러는 항상 CPU 간 비극을 일으킵니다. 이 사실들은 Listings 6.10 and 6.11에 각각 보인 `memblock_alloc()`과 `memblock_free()`의 락 없는 빠른 수행경로를 가능하게 합니다. 그러나, 이는 또한 개발자가 새로 할당된 메모리 블록으로의 포인터를 공개할 때 적절한 순서를 제공해야 할 (예를 들면, `smp_store_release()`를 이용해서) 책임이 있는 이유입니다.

그러나, 할당기는 쓰레드별 풀을 밸런스 맞출 때 순서를 제공해야만 합니다. 이 순서 잡기는 `memblock_alloc()`과 `memblock_free()`에서의 `spin_lock()`과 `spin_unlock()`을 통해 제공됩니다. 한 쓰레드에서 다른 쓰레드로 옮겨진 모든 블록에 대해, 기존 쓰레드는 해당 블록을 `globalmem` 풀에 위치시킨 후 `spin_unlock(&globalmem.mutex)`를 수행할 것이고, 새 쓰레드는 해당 블록을 자신의 쓰레드별 풀에 옮기기 전에 `spin_lock(&globalmem.mutex)`를 수행했을 겁니다. 이 `spin_unlock()`과 `spin_lock()`은 기존 쓰레드와 새 쓰레드 둘 다 기존 쓰레드의 액세스가 새 쓰레드의 것들 전에 일어난 것으로 볼 것을 보장할 겁니다.

Quick Quiz 15.35: 하지만 PowerPC는 리눅스 커널에서 완화된 unlock-lock 순서 속성을 가져서 unlock 전의 쓰기가 lock 뒤의 읽기와 재배치될 수 있게 하지 않나요?

■ 따라서, 일반적인 메모리 할당의 사용에 필요한 순서는 빠르지 않은 수행경로의 locking을 통해 제공되어, 빠른 수행경로는 여전히 동기화로부터 자유롭게 합니다.

15.4.2 RCU

Section 9.5.2에서 이야기된 바와 같이, RCU grace period의 기본적 속성은 이 간단한 두 부분의 보장입니다: (1) 특정 RCU read-side 크리티컬 섹션의 어떤 부분이 어떤 grace period의 시작을 앞선다면 이 크리티컬 섹션의 모든 부분이 그 grace period의 끝을 앞섭니다. (2) 특정 RCU read-side 크리티컬 섹션의 어떤 부분이 어떤 grace period의 끝을 뒤따른다면, 이 크리티컬 섹션의 모든 부분이 이 grace period의 시작을 뒤따릅니다. 이 보장들은 grace period가 우상단의 `call_rcu()` 호출과 좌하단의 연관된 RCU 콜백 수행 사이의 점선 화살표로 표시된 Figure 15.13가 이를 요약합니다.¹³

요약하자면, RCU read-side 크리티컬 섹션은 Listing 15.31 (C-SB+o-rcusync-o+r1-o-o-rul.litmus)에 보인 바와 같이 RCU grace period와 겹치지 않을 것이 보장되어 있습니다. r2 레지스터들은 둘 다 마지막 값으로 0을 가지거나 가지지 않을 수 있으나 그 중 최소 하나는 0이 아니어야만 하는데 (즉, `exists` 절에 의해 파악되는 사이클이 금지됩니다), RCU의 기본적인 grace-period 보장 덕분으로, 이 리트머스 테스트를 `herd`로 수행해서 볼 수 있습니다. 이 보장은 P1()의 크리티컬 섹션 내에서의 액세스 순서에 민감하지 않으며, 따라서 Listing 15.32¹⁴ 또한 같은 사이클을 막습니다.

¹³ 보다 자세한 사항을 위해선 page 143에서 시작하는 Figures 9.11–9.13를 참고해 주시기 바랍니다.

¹⁴ 물론 종속성은 RCU read-side 크리티컬 섹션 내에서의 액세스 재배치를 제한할 수 있습니다.

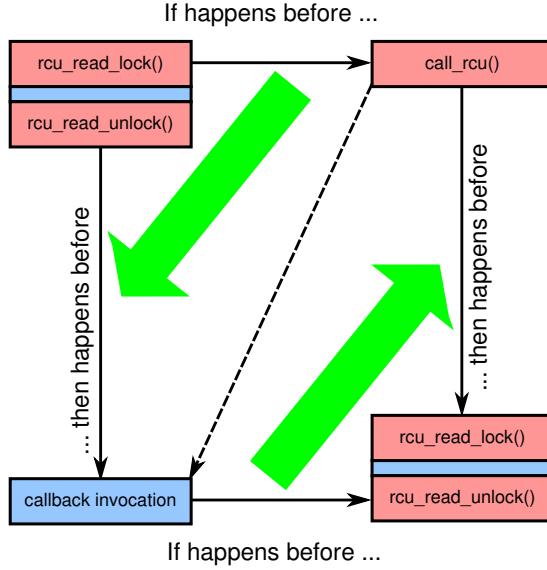


Figure 15.13: RCU Grace-Period Ordering Guarantees

Listing 15.32: RCU Fundamental Property and Reordering

```

1 C C-SB+o-rcusync-o+i-rl-o-o-rul
2
3 {}
4
5 P0(uintptr_t *x0, uintptr_t *x1)
6 {
7     WRITE_ONCE(*x0, 2);
8     synchronize_rcu();
9     uintptr_t r2 = READ_ONCE(*x1);
10 }
11
12 P1(uintptr_t *x0, uintptr_t *x1)
13 {
14     rCU_read_lock();
15     uintptr_t r2 = READ_ONCE(*x0);
16     WRITE_ONCE(*x1, 2);
17     rCU_read_unlock();
18 }
19
20 exists (1:r2=0 /\ 0:r2=0)

```

그러나, 이 정의는 불완전한데, 다음 질문들의 리스트에서 볼 수 있습니다:¹⁵

1. RCU grace period 무관하게 `rcu_read_lock()` 과 `rcu_read_unlock()`에 의해 어떤 순서가 제공되는가?
2. RCU read-side 크리티컬 섹션과 무관하게 `synchronize_rcu()` 와 `synchronize_rcu_expedited()`에 의해 어떤 순서가 제공되는가?
3. 만약 특정 RCU read-side 크리티컬 섹션의 모든 부분이 어떤 RCU grace period의 끝을 앞선다면, 이 크리티컬 섹션을 앞서는 액세스는 어떤가?
4. 만약 어떤 RCU read-side 크리티컬 섹션의 모든 부분이 어떤 RCU grace period의 시작을 뒤따른다면, 이 크리티컬 섹션을 뒤따르는 액세스는 어떤가?
5. 두개 이상의 RCU read-side 크리티컬 섹션 그리고/또는 두개 이상의 RCU grace period가 연관된 상황에서는 무슨 일이 벌어지느냐?
6. RCU 가 다른 메모리 순서 메커니즘과 결합되면 무슨 일이 벌어지는가?

이 질문들은 다음 섹션에서 다루어집니다.

15.4.2.1 RCU Read-Side Ordering

그것 자체로, RCU 의 read-side 기능인 `rcu_read_lock()` 과 `rcu_read_unlock()` 은 어떤 순서 규칙도

¹⁵ 이 중 일부는 LKMM 초기 작업 중 Jade Alglave 에 의해, 다른 여러가지는 다른 LKMM 참여자로부터 [AMM⁺18] Paul 에게 소개되었습니다.

Listing 15.33: RCU Readers Provide No Lock-Like Ordering

```

1 C C-LB+rl-o-o-rul+rl-o-o-rul
2
3 {}
4
5 P0(uintptr_t *x0, uintptr_t *x1)
6 {
7     rCU_read_lock();
8     uintptr_t r1 = READ_ONCE(*x0);
9     WRITE_ONCE(*x1, 1);
10    rCU_read_unlock();
11 }
12
13 P1(uintptr_t *x0, uintptr_t *x1)
14 {
15     rCU_read_lock();
16     uintptr_t r1 = READ_ONCE(*x1);
17     WRITE_ONCE(*x0, 1);
18     rCU_read_unlock();
19 }
20
21 exists (0:r1=1 /\ 1:r1=1)

```

Listing 15.34: RCU Readers Provide No Barrier-Like Ordering

```

1 C C-LB+o-rl-rul-o+o-rl-rul-o
2
3 {}
4
5 P0(uintptr_t *x0, uintptr_t *x1)
6 {
7     uintptr_t r1 = READ_ONCE(*x0);
8     rCU_read_lock();
9     rCU_read_unlock();
10    WRITE_ONCE(*x1, 1);
11 }
12
13 P1(uintptr_t *x0, uintptr_t *x1)
14 {
15     uintptr_t r1 = READ_ONCE(*x1);
16     rCU_read_lock();
17     rCU_read_unlock();
18     WRITE_ONCE(*x0, 1);
19 }
20
21 exists (0:r1=1 /\ 1:r1=1)

```

제공하지 않습니다. 특히, 그 이름에도 불구하고 Listing 15.33 (C-LB+rl-o-o-rul+rl-o-o-rul.litmus)에 보인 것처럼 이것들은 락처럼 동작하지 않습니다. 이 리트머스 테스트의 사이클은 허용되어 있습니다: r1 레지스터의 두 인스턴스가 모두 마지막 값 1을 가질 수 있습니다.

이 기능들 중 어떤 것도 배리어 같은 순서 속성을 갖지 않는데, at least not unless there is a grace period in the mix, as can be seen in Listing 15.34 (C-LB+o-rl-rul-o+o-rl-rul-o.litmus)에 보인 것처럼 어떤 grace period 가 섞여 있지 않은 한은 그렇습니다. 이 리트머스 테스트의 사이클은 허용되어 있습니다. (시도해 보세요!)

물론, 이 두 리트머스 테스트에서의 순서의 부재는 rCU_read_lock() 와 rCU_read_unlock() 이 모두 RCU의 QSBR 구현에서는 no-op이라는 점을 생각하면 결코 놀랍지 않을 겁니다.

Listing 15.35: RCU Updaters Provide Full Ordering

```

1 C C-SB+o-rcusync-o+o-rcusync-o
2
3 {}
4
5 P0(uintptr_t *x0, uintptr_t *x1)
6 {
7     WRITE_ONCE(*x0, 2);
8     synchronize_rcu();
9     uintptr_t r2 = READ_ONCE(*x1);
10 }
11
12 P1(uintptr_t *x0, uintptr_t *x1)
13 {
14     WRITE_ONCE(*x1, 2);
15     synchronize_rcu();
16     uintptr_t r2 = READ_ONCE(*x0);
17 }
18
19 exists (1:r2=0 /\ 0:r2=0)

```

Listing 15.36: What Happens Before RCU Readers?

```

1 C C-SB+o-rcusync-o+o-rl-o-rul
2
3 {}
4
5 P0(uintptr_t *x0, uintptr_t *x1)
6 {
7     WRITE_ONCE(*x0, 2);
8     synchronize_rcu();
9     uintptr_t r2 = READ_ONCE(*x1);
10 }
11
12 P1(uintptr_t *x0, uintptr_t *x1)
13 {
14     WRITE_ONCE(*x1, 2);
15     rCU_read_lock();
16     uintptr_t r2 = READ_ONCE(*x0);
17     rCU_read_unlock();
18 }
19
20 exists (1:r2=0 /\ 0:r2=0)

```

15.4.2.2 RCU Update-Side Ordering

RCU 읽기 쪽 것들과 달리, RCU update-side 기능들인 synchronize_rcu() 와 synchronize_rcu_expedited() 는 최소 smp_mb() 만큼은 강력한 메모리 순서를 제공하며 Listing 15.35에 보인 리트머스 테스트를 herd로 돌려봄으로써 볼 수 있습니다. 이 테스트의 사이클은 smp_mb() 를 놓는 것처럼 금지되어 있습니다. 이는 Table 15.3에 보인 정보를 놓고 보면 놀라운 일은 아닐 겁니다.

15.4.2.3 RCU Readers: Before and After

이 섹션을 읽기 전에, 사용 가능한 보장과 지속 관리되어야 하는 소프트웨어가 기댈 수 있는 보장의 달성을 생각하는게 좋을 겁니다. 이를 분명히 명심해 두고, 이 섹션은 보다 신기한 RCU 보장들을 보입니다.

Listing 15.36 (C-SB+o-rcusync-o+o-rl-o-rul.litmus)는 Listing 15.31에 보인 것과 유사하지만 RCU

Listing 15.37: What Happens After RCU Readers?

```

1 C C-SB+o-rcusync-o+r1-o-rul-o
2
3 {}
4
5 P0(uintptr_t *x0, uintptr_t *x1)
6 {
7   WRITE_ONCE(*x0, 2);
8   synchronize_rcu();
9   uintptr_t r2 = READ_ONCE(*x1);
10 }
11
12 P1(uintptr_t *x0, uintptr_t *x1)
13 {
14   rCU_read_lock();
15   WRITE_ONCE(*x1, 2);
16   rCU_read_unlock();
17   uintptr_t r2 = READ_ONCE(*x0);
18 }
19
20 exists (1:r2=0 /\ 0:r2=0)

```

읽기 쓰레드의 첫번째 액세스가 RCU read-side 크리티컬 섹션을 앞서는데, 보다 전통적인 (그리고 유지보수 가능한!) 방법 역시 안에 내포되어 있습니다. 어쩌면 놀랍게도, 이 리트머스 테스트를 *herd*로 돌리는 것은 Listing 15.31에서와 같은 결과를 냈습니다: 사이클이 금지되었습니다.

왜 그런 걸까요?

P1()의 액세스들은 모두 volatile 이어서, Section 4.3.4.2에서 이야기한 바와 같이 이 컴파일러는 이것들을 재배치 하지 못합니다. 이 말은 P1()의 WRITE_ONCE()에 의해 생성된 코드는 P1()의 READ_ONCE()를 앞서게 됩니다. 따라서, rCU_read_lock()과 rCU_read_unlock() 안에 메모리 배리어 명령을 넣는 RCU 구현은 P1()의 두 액세스의 순서를 하드웨어 단계까지 유지시킬 겁니다. 다른 한편, 인터럽트 기반의 상태 기계에 기반하는 RCU 구현 역시 이 순서를 *grace period*에 상대적으로 지킬 것인데 인터럽트가 인터럽트된 코드의 정확한 수행 위치에서 이뤄진다는 사실 때문입니다.

이는 결국 WRITE_ONCE()가 특정 RCU grace period의 끝을 뒤따른다면, 그 RCU read-side 크리티컬 섹션 내의 그리고 뒤따르는 액세스들은 같은 grace period의 시작을 뒤따르게 된다는 것입니다. 비슷하게, READ_ONCE()가 grace period의 시작을 앞선다면 그 크리티컬 섹션 내의 그리고 앞서는 모든 것은 같은 grace period의 끝을 앞서야만 합니다.

Listing 15.37 (C-SB+o-rcusync-o+r1-o-rul-o.litmus)도 비슷하지만 RCU read-side 크리티컬 섹션 뒤의 액세스를 알아봅니다. 이 테스트의 사이클 역시 금지되어 있으며, *herd*를 통해 검사할 수 있습니다. 그 이유는 Listing 15.36의 것과 비슷하며 독자 여러분의 연습을 위해 놔두겠습니다.

Listing 15.38 (C-SB+o-rcusync-o+o-rl-rul-o.litmus)는 이를 한단계 더 나아가서 P1()의 WRITE_

Listing 15.38: What Happens With Empty RCU Readers?

```

1 C C-SB+o-rcusync-o+o-rl-rul-o
2
3 {}
4
5 P0(uintptr_t *x0, uintptr_t *x1)
6 {
7   WRITE_ONCE(*x0, 2);
8   synchronize_rcu();
9   uintptr_t r2 = READ_ONCE(*x1);
10 }
11
12 P1(uintptr_t *x0, uintptr_t *x1)
13 {
14   WRITE_ONCE(*x1, 2);
15   rCU_read_lock();
16   rCU_read_unlock();
17   uintptr_t r2 = READ_ONCE(*x0);
18 }
19
20 exists (1:r2=0 /\ 0:r2=0)

```

ONCE()를 RCU read-side 크리티컬 섹션 앞으로 두고 P1()의 READ_ONCE()는 그를 뒤따르게 해서 텅 빈 RCU read-side 크리티컬 섹션을 만듭니다.

어쩌면 놀랍게도, 텅 빈 크리티컬 섹션에도 불구하고 RCU는 여전히 사이클의 형성을 막습니다. 이는 다시 한번 *herd*를 통해 검사될 수 있습니다. 더 나아가, 그 이유는 다시 한번 Listing 15.36의 것과 비슷합니다. 요약해 보자면, P1()의 WRITE_ONCE()가 특정 grace period의 끝을 뒤따른다면, P1()의 RCU read-side 크리티컬 섹션—그리고 그를 뒤따르는 모든 것—은 그 grace period의 시작을 뒤따라야만 합니다. 비슷하게, P1()의 READ_ONCE()가 특정 grace period의 시작을 앞선다면, P1()의 RCU read-side 크리티컬 섹션—그리고 이를 앞서는 모든 것—은 이 grace period의 뒤를 앞서야만 합니다. 두 경우 모두, 크리티컬 섹션의 비어있음을 관계 없습니다.

Quick Quiz 15.36: 잠깐만요! RCU의 QSBR 구현에서는 rCU_read_lock()과 rCU_read_unlock()에 어떤 코드도 만들어지지 않습니다. 이 말은 Listing 15.38의 RCU read-side 크리티컬 섹션은 그저 비어있음을, 완전히 존재하지 않음을 의미합니다!!! 그런데 어떻게 존재하지도 않는 무언가가 순서에 대해 뭔가 효과를 낼 수 있죠???

■

이 상황은 Listing 15.39 (C-SB+o-rcusync-o+o-o.litmus)에 보인 것처럼 rCU_read_lock()과 rCU_read_unlock()이 발생하면 무슨 일이 벌어지는지에 대한 질문을 이깁니다. *herd*로 볼 수 있듯 이 리트머스 테스트의 사이클은 허용되는데, 즉 r2의 두 인스턴스가 마지막 값으로 0을 가질 수 있습니다.

이는 텅 빈 RCU read-side 크리티컬 섹션이 순서를 제공한다는 사실을 놓고 볼 때 이상해 보입니다. 그리고 RCU의 QSBR 구현은 실제로 이 결과를 허용치 않는다

Listing 15.39: What Happens With No RCU Readers?

```

1 C C-SB+o-rcusync-o+o-o
2
3 {}
4
5 P0(uintptr_t *x0, uintptr_t *x1)
6 {
7     WRITE_ONCE(*x0, 2);
8     synchronize_rcu();
9     uintptr_t r2 = READ_ONCE(*x1);
10 }
11
12 P1(uintptr_t *x0, uintptr_t *x1)
13 {
14     WRITE_ONCE(*x1, 2);
15     uintptr_t r2 = READ_ONCE(*x0);
16 }
17
18 exists (1:r2=0 /\ 0:r2=0)

```

는 건 사실인데 P1()의 함수 몸체 전체가 preemption을 금지하고 있어서 P1()은 암묵적 RCU read-side 크리티컬 섹션 내에서 수행된다는 사실 때문입니다. 그러나, RCU는 비 QSBR 구현도 가지며, 이 구현을 사용하는 커널은 preemption 가능합니다. 이는 암묵적 RCU read-side 크리티컬 섹션이 존재치 않으며, 그 결과 RCU가 순서를 강제할 방법이 없다는 겁니다. 따라서, 이 리트머스 테스트의 사이클이 허용됩니다.

Quick Quiz 15.37: Listings 15.36, 15.37, 그리고 15.38의 리트머스 테스트들에 보인 P1()의 액세스들은 그것들이 Listing 15.31부터 Listing 15.32까지에서와 같은 방식으로 재배치 될 수 있나요?

**15.4.2.4 Multiple RCU Readers and Updaters**

`synchronize_rcu()`는 `smp_mb()` 보다 강력한 순서 규칙을 가지므로, SB 리트머스 테스트에 얼마나 많은 프로세스들이 있는가에 (Listing 15.35에서처럼), 관계 없이 `synchronize_rcu()`를 각 프로세스의 액세스 사이에 `synchronize_rcu()`를 두는 것은 사이클을 금지시킵니다. 또한, Listing 15.31에 보인 것처럼 한 프로세스는 `synchronize_rcu()`를 사용하고 다른 프로세스는 `rcu_read_lock()`을 사용할 때에도 사이클이 금지됩니다. 그러나, 두 프로세스가 Listing 15.33처럼 `rcu_read_lock()`과 `rcu_read_unlock()`을 사용한다면 사이클은 허용됩니다.

어떤 RCU로 보호되는 리트머스 테스트가 금지되고 어떤건 허용될지에 대한 일반적 규칙 같은게 있을까요? 이 섹션은 그 질문에 답해봅니다.

더 구체적으로, Listing 15.40에 보인 것처럼 리트머스 테스트가 하나의 RCU grace period와 두개의 RCU 임기 쓰레드가 있다면 어떻게 될까요? `herd`는 이 사

Listing 15.40: One RCU Grace Period and Two Readers

```

1 C C-SB+o-rcusync-o+rl-o-o-rul+rl-o-o-rul
2
3 {}
4
5 P0(uintptr_t *x0, uintptr_t *x1)
6 {
7     WRITE_ONCE(*x0, 2);
8     synchronize_rcu();
9     uintptr_t r2 = READ_ONCE(*x1);
10 }
11
12 P1(uintptr_t *x1, uintptr_t *x2)
13 {
14     rcu_read_lock();
15     WRITE_ONCE(*x1, 2);
16     uintptr_t r2 = READ_ONCE(*x2);
17     rcu_read_unlock();
18 }
19
20 P2(uintptr_t *x2, uintptr_t *x0)
21 {
22     rcu_read_lock();
23     WRITE_ONCE(*x2, 2);
24     uintptr_t r2 = READ_ONCE(*x0);
25     rcu_read_unlock();
26 }
27
28 exists (2:r2=0 /\ 0:r2=0 /\ 1:r2=0)

```

Listing 15.41: Two RCU Grace Periods and Two Readers

```

1 C C-SB+o-rcusync-o+o-rcusync-o+rl-o-o-rul+rl-o-o-rul
2
3 {}
4
5 P0(uintptr_t *x0, uintptr_t *x1)
6 {
7     WRITE_ONCE(*x0, 2);
8     synchronize_rcu();
9     uintptr_t r2 = READ_ONCE(*x1);
10 }
11
12 P1(uintptr_t *x1, uintptr_t *x2)
13 {
14     WRITE_ONCE(*x1, 2);
15     synchronize_rcu();
16     uintptr_t r2 = READ_ONCE(*x2);
17 }
18
19 P2(uintptr_t *x2, uintptr_t *x3)
20 {
21     rcu_read_lock();
22     WRITE_ONCE(*x2, 2);
23     uintptr_t r2 = READ_ONCE(*x3);
24     rcu_read_unlock();
25 }
26
27 P3(uintptr_t *x0, uintptr_t *x3)
28 {
29     rcu_read_lock();
30     WRITE_ONCE(*x3, 2);
31     uintptr_t r2 = READ_ONCE(*x0);
32     rcu_read_unlock();
33 }
34
35 exists (3:r2=0 /\ 0:r2=0 /\ 1:r2=0 /\ 2:r2=0)

```

이클이 허용되었다 이야기 합니다만, 왜 그런지 아는게 좋을 겁니다.¹⁶

요점은 CPU 가 P1() 과 P2() 의 WRITE_ONCE() 와 READ_ONCE() 를 재배치 할 수 있다는 겁니다. 그 재배치가 있다면, Figure 15.14 가 어떻게 이 사이클이 생성되는지 보입니다:

1. 이 다이어그램의 바닥 근처 점선 화살표로 그려진 것처럼 P0() 의 x1 읽기가 P1() 의 쓰기를 앞지릅니다.
2. P1() 의 쓰기는 P0() 의 grace period 의 종료를 뒤따르므로, P1() 의 x2 읽기는 P0() 의 grace period 의 시작을 앞설 수 없습니다.
3. P1() 의 x2 읽기가 P2() 의 쓰기를 앞지릅니다.
4. P2() 의 x2 쓰기가 P0() 의 grace period 의 종료를 앞지르므로, P2() 의 x0 읽기가 P0() 의 grace period 의 시작을 앞지르는 건 완전히 합법적입니다.
5. 따라서, P2() 의 x0 읽기는 P0() 의 쓰기를 앞지를 수 있어서 사이클이 형성됩니다.

하지만 grace period 가 하나 더 추가되면 어떤 일이 벌어질까요? 이 상황이 Listing 15.41 에 그려져 있는데, 하나의 SB 리트머스 테스트로 P0() 와 P1() 의 RCU grace period 를 가지고 P2() 와 P3() 는 RCU 읽기 쓰레드입니다. 다시 말하지만, CPU 는 RCU read-side 크리티컬 섹션 내에서의 액세스들을 재배치 할 수 있습니다. Figure 15.15 에 보인 것과 같아요. 이 사이클이 생성되려면 P2() 의 크리티컬 섹션은 P1() 의 grace period 후에 끝나야 하고 P3() 의 크리티컬 섹션은 같은 grace period 의 시작 뒤에 끝나야 하는데, 이는 또한 P0() 의 grace period 의 종료 후에 일어날 겁니다. 따라서, P3() 의 크리티컬 섹션은 P0() 의 grace period 시작 후에 시작되어야만 하는데, 이는 결국 P3() 의 x0 읽기가 P0() 의 쓰기를 앞설 수 없음을 의미합니다. 따라서, RCU read-side 크리티컬 섹션이 전체 RCU grace period 로 확장될 수 없으므로 사이클이 금지됩니다.

그러나, Figure 15.15 를 깊이 들여다보면 세번째 읽기 쓰레드를 추가하면 사이클이 허용됨을 알 수 있습니다. 이는 이 세번째 읽기 쓰레드가 P0() 의 grace period 전에 끝나서 같은 grace period 의 시작 전에 시작될 수 있기 때문입니다. 이는 결국 일반적인 규칙을 제안하는데, 그 것은: 이런 종류의 RCU 만 있는 리트머스 테스트에서,

¹⁶ 특히 Paul 이 Jade Alglave 와 RCU 순서 규칙을 범용화 하기 위해 일할 때 이 특별한 리트머스 테스트에 대한 생각을 여러번 바꿨다는 것을 놓고 보면요.

RCU read-side 크리티컬 섹션의 갯수만큼의 RCU grace period 가 존재한다면 사이클은 금지된다.¹⁷

15.4.2.5 RCU and Other Ordering Mechanisms

하지만 RCU 를 다른 순서 규칙 메커니즘과 결합한 리트머스 테스트는 어떨까요?

일반적 규칙은 사이클을 막는데에는 하나의 메커니즘만 사용된다는 겁니다.

예를 들어, Listing 15.33 로 되돌아 가 봅시다. 앞의 섹션에서의 일반적 규칙을 적용하면, 이 리트머스 테스트는 두개의 RCU read-side 크리티컬 섹션이 있고 어느 RCU grace period 도 없으므로, 사이클이 형성됩니다. 그러나 P0() 의 WRITE_ONCE() 가 smp_store_release() 로 대체되고 P1() 의 READ_ONCE() 가 smp_load_acquire() 로 대체된다면 어떻게 될까요?

RCU 는 여전히 사이클을 허용할 겁니다만, release-acquire 쌍은 금지합니다. 사이클을 금지하는데에 하나의 메커니즘만 사용되므로, 이 release-acquire 쌍이 이겨서 사이클이 금지되게 합니다.

또 다른 예로, Listing 15.40 로 돌아가 봅시다. 이 리트머스 테스트는 두개의 RCU 읽기 쓰래드가 있지만 하나의 grace period 만 있기에, 이 사이클은 허용되지 않습니다. 하지만 smp_mb() 가 P1() 의 액세스 쌍 사이에 있다고 해봅시다. 이 새 리트머스 테스트에서는 smp_mb() 의 추가 때문에 P2() 와 P1() 의 크리티컬 섹션 모두 P0() 의 grace period 의 끝 너머로 확장되는데, 이는 결국 P2() 의 x0 읽기가 P0() 의 쓰기를 앞서지 못하게 하며, 이는 Figure 15.16 의 빨간 점선 화살표로 그려져 있습니다. 이 경우, RCU 와 완전한 메모리 배리어는 사이클을 금지하기 위해 함께 동작하는데, RCU 는 P0(), P1() 그리고 P2() 사이의 순서를 유지하고 smp_mb() 는 P1() 과 P2() 사이의 순서를 유지합니다.

Quick Quiz 15.38: Listing 15.40 의 P2() 의 액세스 사이에 smp_mb() 가 위치되면 어떻게 될까요?

■
요약하자면, RCU 의 규칙이 실용적인 곳이라면 그것들은 이제 완전히 정형화 되었습니다 [MW05, DMS⁺12, GRY13, AMM⁺18].

더 높은 수준의 기능에 대한 자세한 의미가 보다 많은 정적 분석과 모델 검사를 가능하게 할 것을 희망합니다.

¹⁷ 흥미롭게도, Alan Stern 은 LKMM 의 맥락에서 Section 9.5.2 에 보인 RCU 의 기본적 속성 두 부분이 실제로 RCU 공리 [AMM⁺18] 라 불리는 이 보다 일반적 결과로 보이는 것을 암시함을 증명했습니다.

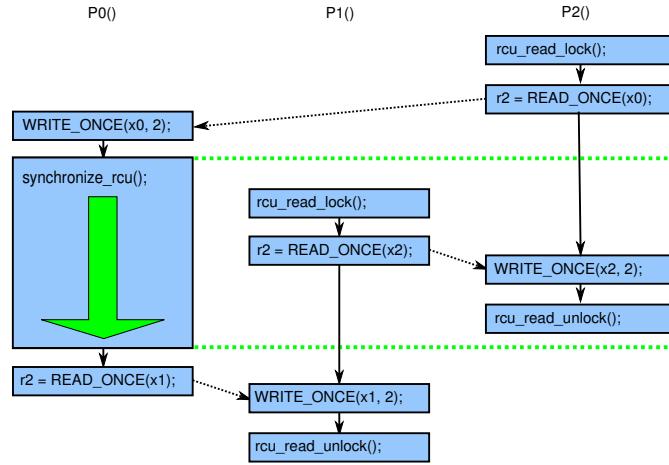


Figure 15.14: Cycle for One RCU Grace Period and Two RCU Readers

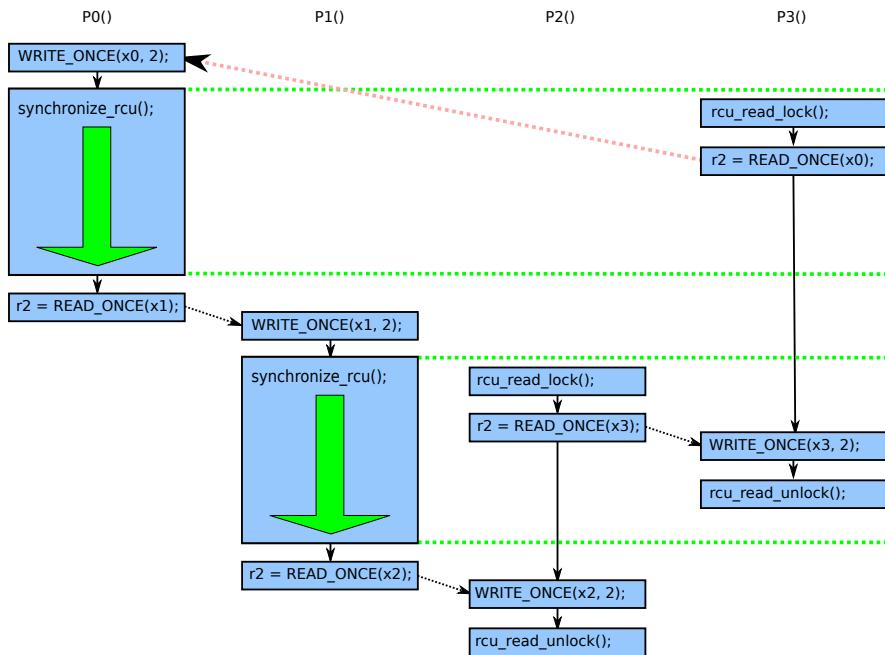


Figure 15.15: No Cycle for Two RCU Grace Periods and Two RCU Readers

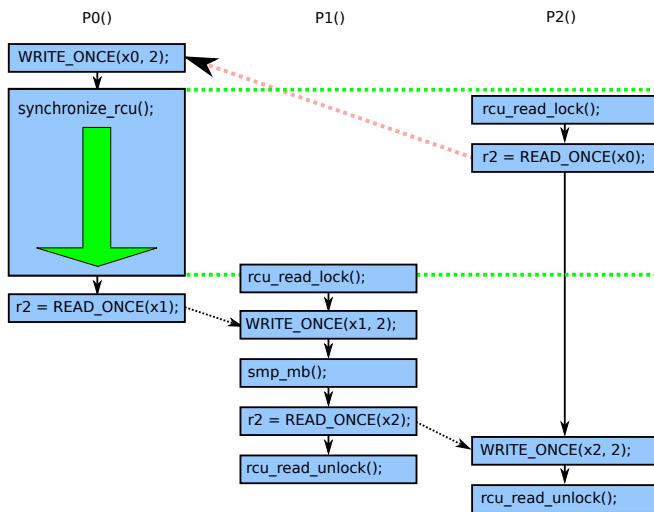


Figure 15.16: Cycle for One RCU Grace Period, Two RCU Readers, and Memory Barrier

15.5 Hardware Specifics

Rock beats paper!

Derek Williams

각 CPU 제품군은 각자만의 메모리 순서규칙에 대한 기묘한 접근법을 취하는데, 이는 Table 15.5에서 이야기된 바와 같이 이식성을 어렵게 할 수 있습니다. 실제로, 일부 소프트웨어 환경은 메모리 순서 오퍼레이션의 사용을 단순히 금지시켜서 프로그래머가 필요한 정도 까지는 사용할 수 있는 상호배제 기능까지만 사용할 수 있도록 제약시킵니다. 이 섹션은 각 CPU 제품군의 모든 (또는 심지어 대부분의) 부분을 다루는 레퍼런스 매뉴얼이 되려기보다는 대략적 비교를 제공하는 높은 단계의 개론으로 의도되었음을 명심해 주십시오. 전체 세부사항을 위해선 관심 있는 CPU의 레퍼런스 매뉴얼을 들여다 보시기 바랍니다.

Table 15.5로 돌아가서, 열들의 첫번째 그룹은 메모리 순서 속성을 보이고 두번째 그룹은 명령 속성을 보입니다.

첫번째 세 열은 특정 CPU가 Section 15.1과 Sections 15.2.2.1–15.2.2.3에서 이야기된 것처럼 네개의 가능한 로드와 스토어 조합을 재배치 할 수 있는지 보입니다. 다음 열 (“Atomic Instructions Reordered With Loads or Stores?”)은 특정 CPU가 어토믹 인스트럭션을 사용하는 로드와 스토어가 재배치 될 수 있는지 보입니다.

다섯번째와 여섯번째 열은 재배치와 의존성을 다룬는데, 이는 Sections 15.2.3–15.2.5에서 다루어졌고 Section 15.5.1에서 보다 자세한 내용이 설명되었습니다.

간략한 버전으로는 Alpha가 연결된 데이터 구조에 대한 읽기 쓰래드는 물론 업데이트 쓰래드에도 메모리 배리어를 필요로 하지만, 이 메모리 배리어는 v4.15 이후 리눅스 커널에서의 Alpha 아키텍쳐 전용 코드에 의해 제공된다는 겁니다.

다음 열, “Non-Sequentially Consistent”은 CPU의 평범한 로드와 스토어 명령이 sequential consistency에 의해 제약되는지 보입니다. 거의 모두가 성능상 이유로 이 방법으로는 제약되지 않습니다.

다음 두 열은 multicopy atomicity를 다루는데, Section 15.2.7에서 정의되었습니다. 첫번째는 완전한(그리고 드문) multicopy atomicity이고, 두번째 것은 완화된 다른 것들에 대한 multicopy atomicity입니다.

다음 열, “Non-Cache Coherent”은 Section 15.2.6에서 다루었던 여러 쓰래드의 단일 변수로의 액세스를 다룹니다.

마지막 세개의 열은 명령 수준의 선택과 문제를 다룹니다. 첫번째 열은 각 CPU가 load-acquire와 store-release를 어떻게 구현하는지 알리고, 두번째 열은 CPU들을 atomic-instruction 종류로 구분하며, 마지막 세번째 열은 특정 CPU가 일관적이지 않은 명령 캐시와 파이프라인을 갖는지 보입니다. 그런 CPU는 스스로를 수정하는 코드를 위해 특별한 명령을 수행해야 합니다.

메모리 순서 오퍼레이션에 “그냥 안된다고 하기”라는 혼란 방법은 적용되는 곳에서는 탁월하게 합리적입니다만, 리눅스 커널과 같이 직접적인 메모리 순서 오퍼레이션이 필요한 환경도 있습니다. 따라서, 리눅스는 주의깊게 선택된 최소 공통 분모 메모리 순서 기능 집합을 제공하는데, 다음과 같습니다:

Table 15.5: Summary of Memory Ordering

Property		CPU Family								
		Alpha	Armv7-A/R	Armv8	Itanium	MIPS	POWER	SPARC TSO	x86	z Systems
Memory Ordering	Loads Reordered After Loads or Stores?	Y	Y	Y	Y	Y	Y			
	Stores Reordered After Stores?	Y	Y	Y	Y	Y	Y			
	Stores Reordered After Loads?	Y	Y	Y	Y	Y	Y	Y	Y	Y
	Atomic Instructions Reordered With Loads or Stores?	Y	Y	Y		Y	Y			
	Dependent Loads Reordered?		Y							
	Dependent Stores Reordered?									
	Non-Sequentially Consistent?	Y	Y	Y	Y	Y	Y	Y	Y	Y
	Non-Multicopy Atomic?	Y	Y	Y	Y	Y	Y	Y	Y	Y
	Non-Other-Multicopy Atomic?	Y	Y		Y	Y	Y			
	Non-Cache Coherent?				Y					
Instructions	Load-Acquire/Store-Release?	F	F	i	I	F	b			
	Atomic RMW Instruction Type?	L	L	L	C	L	L	C	C	C
	Incoherent Instruction Cache/Pipeline?	Y	Y	Y	Y	Y	Y	Y	Y	Y

Key:

- b: Load-Acquire/Store-Release?
- b: Lightweight memory barrier
- F: Full memory barrier
- i: Instruction with lightweight ordering
- I: Instruction with heavyweight ordering
- Atomic RMW Instruction Type?
- C: Compare-and-exchange instruction
- L: Load-linked/store-conditional instruction

`smp_mb()` (완전한 메모리 배리어)는 로드와 스토어를 모두 순서잡습니다. 이는 메모리 배리어를 앞서는 로드와 스토어가 메모리 배리어를 뒤따르는 로드와 스토어보다 먼저 메모리에 가해질 것을 의미합니다.

`smp_rmb()` (읽기 메모리 배리어)는 로드만을 순서잡습니다.

`smp_wmb()` (쓰기 메모리 배리어)는 스토어만을 순서잡습니다.

`smp_mb__before_atomic()`은 `smp_mb__before_atomic()`을 앞서는 액세스들을 뒤의 RMW 어토믹 오퍼레이션을 뒤따르는 액세스에 대해 순서잡습니다. 완전히 순서잡힌 어토믹 RMW 오퍼레이션을 제공하는 시스템에서 이는 noop입니다.

`smp_mb__after_atomic()`은 앞의 RMW 어토믹 오퍼레이션을 앞서는 액세스들을 `smp_mb__after_atomic()`을 뒤따르는 액세스에 대해 순서잡습니다. 완전히 순서잡힌 어토믹 RMW 오퍼레이션을 제공하는 시스템에서는 이것 역시 noop입니다.

`smp_mb__after_spinlock()`은 락 획득을 앞서는 액세스들을 `smp_mb__after_atomic()`을 뒤따르는 것들에 대해 순서잡습니다. 완전히 순서 잡힌 락 획득을 제공하는 시스템에서는 이것 역시 noop입니다.

`mmiowb()`는 전역 스피너으로 보호되는 MMIO 쓰기를 순서잡으며, 2016년의 MMIO에 대한 LWN 기사에서 더 자세히 설명되었습니다 [MDR16].

`smp_mb()`, `smp_rmb()`, 그리고 `smp_wmb()` 기능들은 또한 컴파일러가 배리어 앞뒤로 메모리 재배치를 할 수 있는 최적화를 금지하게 합니다.

Quick Quiz 15.39: 어토믹 오퍼레이션과 `smp_mb_after_atomic()` 사이의 코드엔 무슨 일이 벌어지나요?

■ 이 기능들은 SMP 커널에서만 코드를 생성합니다만, UP (Uni Processor) 커널에서도 메모리 배리어를 생성하는 몇몇 UP 버전이 (`mb()`, `rmb()`, 그리고 `wmb()`) 존재합니다. 대부분의 상황에서는 `smp_` 버전들이 사용되어야 합니다. 그러나, 이 뒤의 기능들은 드라이버를 작성할 때 유용한데, MMIO 액세스는 UP 커널에서도 순서를 지켜야 하기 때문입니다. 메모리 순서 오퍼레이션의 부재 시에, CPU 와 컴파일러는 모두 액세스를 재배치할 수 있는데, 이는 최선의 경우에도 기기가 이상하게 동작하게 하며, 커널을 깨지게 하거나 여러분의 하드웨어를 손상시킬 수조차 있습니다.

따라서 대부분의 커널 프로그래머는 이 인터페이스를 사용하는 이상 각 CPU 의 이상한 메모리 순서 규칙에 걱정하지 않아도 됩니다. 여러분이 특정 CPU 의 아키텍쳐 특수 코드에 깊이 관여하고 있다면, 물론 마음대로 해도 됩니다.

더 나아가서, 리눅스의 락킹 기능 (spinlock, reader-writer 락, semaphore, RCU, ...) 은 모든 필요한 순서 기능을 내포합니다. 따라서 이 기능들을 올바르게 사용한 코드를 가지고 작업한다면 리눅스의 메모리 순서 기능들에 대해 염려하지 않아도 됩니다.

그렇다고는 하나, 각 CPU 의 메모리 일관성 모델에 대한 깊은 지식은 아키텍쳐 특수 코드나 동기화 기능을 작성할 때 디버깅에 매우 도움될 수 있습니다.

결단리로, 어떤 사람들은 얇은 지식은 매우 위험하다고 합니다. 많은 지식을 가지고 여러분이 입힐 수 있는 피해들을 생각해 보세요! 각 CPU 의 메모리 일관성 모델에 대해 이해하기 원하는 분들을 위해, 다음 섹션은 일부 유명하고 대중적인 CPU 들을 알아봅니다. 실제로 해당 CPU 의 문서를 읽는 것을 대체할 방법은 없지만, 이 섹션들은 좋은 개요를 제공합니다.

15.5.1 Alpha

수명이 한참 전에 끝난 CPU 에 대해 무언가 이야기하는 건 이상하게 보일 수 있지만, Alpha 는 종속적 로드를 재배치하는 유일한 주류 CPU 라서 흥미로운 경우이며, 따라서 리눅스 커널을 포함해 동시성 API 에 두드러지는 영향을 끼쳤습니다. 핵심 리눅스 커널 코드에서 Alpha 를 위해 필요했던 코드는 리눅스 커널 v4.15 을 끝으로 사라졌으며, 이 모든 지원을 위한 기록은 `smp_read_barrier_depends()` 와 `read_barrier_depends()` API 의 제거와 함께 사라졌습니다.

Listing 15.42: Insert and Lock-Free Search (No Ordering)

```

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_store_release(&head.next, p);
10    spin_unlock(&mutex);
11 }
12
13 struct el *search(long searchkey)
14 {
15     struct el *p;
16     p = READ_ONCE_OLD(head.next);
17     while (p != &head) {
18         /* Prior to v4.15, BUG ON ALPHA!!! */
19         if (p->key == searchkey) {
20             return (p);
21         }
22         p = READ_ONCE_OLD(p->next);
23     };
24     return (NULL);
25 }
```

다. 이 섹션은 2021년 초 기준으로 여전히 v4.15 전 버전의 리눅스 커널을 가지고 일하는 리눅스 커널 해커들이 꽤 존재하기 때문에 Second Edition 까지는 유지됩니다. 또한, 이 API 들이 제거될 수 있게 한 `READ_ONCE()` 의 변경은 여전히 Alpha 를 지원하는 유저스페이스 프로젝트들에 전파되지 않았을 수 있습니다.

Alpha 와 다른 CPU 사이의 종속적 로드에 대한 차이가 Listing 15.42 의 코드에 보여져 있습니다. 이 `smp_store_release()` 는 라인 6-8 의 원소 초기화가 그 원소가 라인 9 에서 리스트에 추가되기 전에 행해지게끔 보장해서 lock-free 탐색이 올바르게 동작하게끔 합니다. 즉, 이 보장이 Alpha 를 제외한 모든 CPU 에서 보장되게 합니다.

이 코드에서 `READ_ONCE_OLD()` 라고 표시된 v4.15 이전 버전에서의 `READ_ONCE()` 구현이 사용되면, Alpha 는 실제로 Listing 15.42 의 라인 19 의 코드가 라인 6-8에서의 초기화 전의 오래된 쓰레기 값은 볼 수 있게 합니다.

Figure 15.17 는 분할된 캐쉬를 가져서 캐쉬 라인을 교체하는 것이 캐쉬의 다른 파티션에서 수행될 수 있으며 공격적인 병렬 기계에서 이게 어떻게 일어날 수 있는지 보입니다. 예를 들어, Listing 15.42 의 라인 16 에서의 `head.next` 로드는 캐쉬 뱅크 0 을 액세스 했을 수도 있고, 라인 19 에서의 `p->key` 로드와 라인 22 에서의 `p->next` 로드는 캐쉬 뱅크 1 을 액세스 했을 수도 있습니다. Alpha 에서, `smp_store_release()` 는 Listing 15.42 의 라인 6-8 에서 수행된 (`p->next`, `p->key`, 그리고 `p->data`에 대한) 캐쉬 무효화가 라인 9 의 (`head.next`를 위한) 것보다 먼저 중간 연결부에 도달하게 하지만, 읽는 CPU 의 캐쉬 뱅크로의 전파 순서에 대해선 어떤

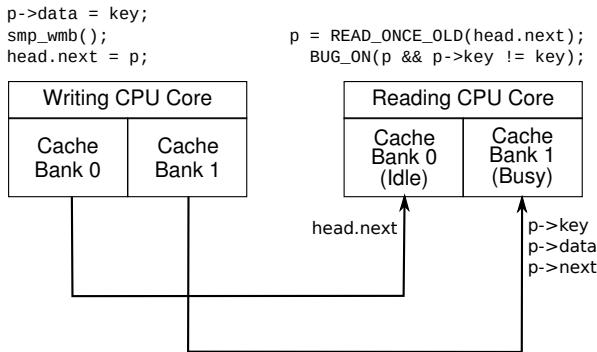


Figure 15.17: Why `smp_read_barrier_depends()` is Required in Pre-v4.15 Linux Kernels

보장도 하지 않습니다. 예를 들면, 읽는 CPU의 캐시 뱅크 1이 매우 바쁜 사이 뱅크 0은 한가할 수 있습니다. 이는 새 원소를 위한 (`p->next`, `p->key`, 그리고 `p->data`의) 캐시 무효화가 지연되게 하여서 읽는 CPU는 `head.next`의 새 값을 로드하지만 `p->key`와 `p->next`의 과거에 캐시된 값을 로드하게 할 수 있습니다. 그렇습니다, 이는 Alpha가 실제로 가리켜진 데이터를 그 포인터 자체보다 먼저 읽어올 수 있는데, 이는 이상하지만 사실입니다. 더 많은 정보를 위해선, 또는 제가 뭔가 잘못 이야기하고 있다고 느끼신다면 앞에서 이야기된 문서를 [Com01, Pug00] 읽어 보세요.¹⁸ 이 일반적이지 않은 순서 규칙의 이익은 Alpha가 더 간단한 캐시 하드웨어를 가질 수 있어서 결국 Alpha의 전성기에는 더 높은 클락 주파수를 가능하게 했다는 겁니다.

어떤 사람은 Alpha가 이 포인터 읽기와 뒤따르는 종속적 로드 사이에 순서를 지키게 하기 위해 `smp_rmb()`를 포인터 읽기와 그 역참조 사이에 위치시킬 수 있을 겁니다. 그러나, 이는 읽기 쪽의 데이터 종속성을 지켜주는 시스템에서는 (Arm, Itanium, PPC, 그리고 SPARC 등) 불필요한 오버헤드를 일으킬 겁니다. 그래서 이런 시스템에서의 오버헤드를 제거하기 위해 `smp_read_barrier_depends()` 기능이 리눅스 커널에 추가되었습니다만, Alpha의 `READ_ONCE()` 정의의 강화 덕에 v5.9 리눅스 커널에서는 사라졌습니다. 따라서, v5.9 기준으로, 코어 커널 코드는 더이상 DEC Alpha 때문에 걱정할 필요가 없습니다. 하지만, Listing 15.43의 라인 16와 21에서 보이듯 모든 최근의 커널 버전에서 안전하고 효율적으로 동작하는 `rcu_dereference()`를 사용하는게 낫습니다.

또한 모든 읽기를 하는 CPU가 쓰기를 하는 CPU의 쓰기를 순서대로 보게 하게끔 `smp_store_release()`

¹⁸ 물론, 영리한 독자 여러분은 Appendix C.6.1의 상상속 아키텍쳐의 경우처럼 (감사하게도) 이미 Alpha가 주변에 없음을 알아차렸을 겁니다.

Listing 15.43: Safe Insert and Lock-Free Search

```

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_store_release(&head.next, p);
10    spin_unlock(&mutex);
11 }
12
13 struct el *search(long searchkey)
14 {
15     struct el *p;
16     p = rcu_dereference(head.next);
17     while (p != &head) {
18         if (p->key == searchkey) {
19             return (p);
20         }
21         p = rcu_dereference(p->next);
22     }
23     return (NULL);
24 }
```

자리에 소프트웨어 메커니즘을 구현할 수도 있습니다. 이 소프트웨어 배리어는 inter-processor interrupt (IPI)들을 모든 다른 CPU에게 보냄으로써 구현될 수 있습니다. 그런 IPI를 받으면, CPU는 메모리 배리어 명령을 수행하므로, 리눅스 커널의 `sys_membARRIER()` 시스템콜이 제공하는 것과 비슷한 시스템 범위 메모리 배리어를 구현합니다. 데드락을 막기 위해 추가적 로직이 필요합니다. 물론, 데이터 종속성을 지키는 CPU는 그런 배리어를 간단히 `smp_store_release()`로 정의할 겁니다. 하지만, 리눅스 커뮤니티는 이 방법은 지나친 오버헤드를 발생시키며 [McK01], 그들의 시점에서는 이는 공격적인 리얼타임 응답 요구가 있는 상황에서는 완전히 부적절하다고 여깁니다.

리눅스 메모리 배리어 기능들은 그들의 이름을 Alpha 명령들에서 따와서 `smp_mb()`는 mb이고 `smp_rmb()`는 rmb이며 `smp_wmb()`는 wmb입니다. Alpha는 `READ_ONCE()`가 `smp_mb()`를 내포하는 유일한 CPU입니다.

Quick Quiz 15.40: Alpha의 `READ_ONCE()`는 왜 `rmb`가 아니라 `mb` 명령을 내포하나요?



Quick Quiz 15.41: DEC Alpha는 가능한 메모리 순서 규칙 중 가장 약한 것을 가지니 중요하지 않나요?



Alpha에 대한 더 많은 내용을 위해선 레퍼런스 매뉴얼을 [Cor02] 보시기 바랍니다.

15.5.2 Armv7-A/R

Arm 제품군 CPU는 임베디드 어플리케이션에서, 특히 휴대폰과 같이 전력이 제한된 어플리케이션에서 굉장히 인기있습니다. 이것의 메모리 모델은 POWER와

유사합니다만 (Section 15.5.6 를 참고하세요), Arm 은 다른 메모리 배리어 명령 집합을 사용합니다 [ARM10]:

DMB (data memory barrier) 는 명시된 종류의 오퍼레이션인 같은 타입의 뒤따르는 모든 오퍼레이션보다 앞서 완료된 것으로 보이게 합니다. 오퍼레이션의 “타입”은 모든 오퍼레이션이거나 쓰기만으로 제한될 수 있습니다 (Alpha 의 wmb 와 POWER eieio 인스트럭션과 유사합니다). 또한, Arm 은 캐쉬 일관성이 세개의 범위를 가질 수 있게 합니다: 단일 프로세서, 프로세서의 부분집합 (“inner”) 그리고 전역 (“outer”).

DSB (data synchronization barrier) 는 명시된 타입의 오퍼레이션이 뒤따르는 (모든 타입의) 오퍼레이션이 수행되기 전에 정말로 완료되게 합니다. 오퍼레이션의 “타입”은 DMB 의 것과 같습니다. DSB 명령은 Arm 아키텍쳐의 초기에는 DWB (drain write buffer 또는 data write barrier) 라고 불렸습니다.

ISB (instruction synchronization barrier) 는 CPU 파일라인을 비워서, ISB 를 뒤따르는 모든 명령이 ISB 의 완료 후에만 읽어지게 만듭니다. 예를 들어, 스스로를 수정하는 프로그램 (JIT 같은) 을 작성한다면, 여러분은 코드의 생성과 그것의 수행 사이에 ISB 를 수행해야 합니다.

이 명령들 중 어느 것도 리눅스의 rmb() 기능의 명세에 정확히 들어맞지 않으며, 따라서 완전한 DMB 를 구현합니다. DMB 와 DSB 명령은 이 배리어 전후로 순서잡힌 액세스들에 대한 회귀적 정의를 갖는데, POWER 의 cumulativity 와 유사한 효과를 내며, 이는 둘 다 Section 15.2.7.1 에서 설명된 LKMM 의 cumulativity 보다 더 강력한 것입니다.

Arm 은 또한 제어 종속성을 구현하여 어떤 조건적 분기가 어떤 로드에 종속적이라면, 그 조건적 분기 뒤에 수행되는 모든 스토어는 그 로드 뒤로 순서잡힙니다. 그러나, 조건적 분기를 뒤따르는 로드는 그 분기와 로드 사이에 ISB 명령이 있지 않은 한 순서잡힐 것으로 보장되지 않습니다. 다음 예를 생각해 봅시다:

```

1 r1 = x;
2 if (r1 == 0)
3     nop();
4 y = 1;
5 r2 = z;
6 ISB();
7 r3 = z;

```

이 예에서, 로드-스토어 제어종속성 순서규칙은 라인 1에서의 x 로드를 라인 4에서의 y 로의 스토어 전으로 순서잡습니다. 하지만, Arm 은 로드-로드 제어종속성을 지키지 않으므로, 라인 1에서의 로드는 라인 5

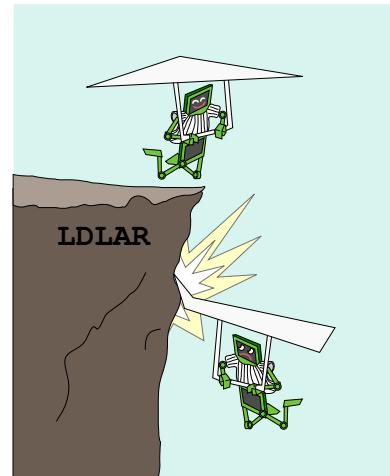


Figure 15.18: Half Memory Barrier

의 로드 뒤에 벌어질 수도 있습니다. 다른 한편, 라인 2에서의 조건적 분기와 라인 6의 ISB 명령은 라인 7의 로드가 라인 1의 로드 뒤에 일어날 것을 보장합니다. 추가적인 ISB 명령을 라인 2 와 5 사이 어딘가에 배치하는 것은 라인 1 와 5 사이의 순서를 강제함을 알아 두시기 바랍니다.

15.5.3 Armv8

Arm 의 Armv8 CPU 제품군은 [ARM17] Section 15.5.2에 설명된 32비트만 지원하는 CPU 와 달리 64비트 기능을 갖습니다. Armv8 의 메모리 모델은 그것의 Armv7 쪽 것들을 상당히 닮아있습니다만 load-acquire (LDLARB, LDLARH, 그리고 LDLAR) 와 store-release (STLLRB, STLLRH, 그리고 STLLR) 명령을 추가했습니다. 이 명령들은 “반쪽 메모리 배리어”로 동작해서 Armv8 CPU 는 앞의 액세스를 뒤의 LDLAR 명령과 재배치할 수 있지만 앞의 LDLAR 명령을 뒤의 액세스와는 재배치 할 수 없는데, Figure 15.18 에 이게 그려져 있습니다. 비슷하게, Armv8 CPU 는 앞의 STLLR 명령을 뒤따르는 액세스와 재배치 할 수 있지만, 뒤의 STLLR 과 앞의 액세스는 재배치 할 수 없습니다. 예상할 수 있듯, 이는 이 명령들은 C11 의 load-acquire 와 store-release 노선을 직접적으로 지원합니다.

그러나, Armv8 은 store-release 와 load-acquire 의 조합이 특정 상황에서는 완전한 배리어로 동작할 것을 강제함으로써 C11 메모리 모델보다 더한 것을 제공합니다. 예를 들어, Armv8 에서 어떤 스토어 뒤에 store-release 가 있고 그 뒤에 load-acquire, 그리고 로드가 있다면, 그리고 그것들이 모두 다른 변수를 향한 것이라면 모두 하나의 CPU 에서 행해진다면, 모든 CPU 는 처음의

스토어가 마지막의 로드를 앞선다고 보게 됩니다. 흥미롭게도, 대부분의 TSO 구조들은 (x86 과 mainframe 포함) 이 보장을 내지 않는데, 여기서의 두 로드는 두 스토어 앞으로 재배치 될 수 있기 때문입니다.

Armv8 은 `smp_mb__after_spinlock()` 이 완전체 배리어가 되어야 하는 두 구조 중 하나인데, 리눅스 커널 내에서의 상대적으로 약한 lock-acquisition 구현 때문입니다.

Armv8 은 또한 제조사가 공개적으로 수행 가능한 정식 모델과 함께 메모리 규칙을 정의한 [ARM17] 최초의 CPU 로 차별화 됩니다.

15.5.4 Itanium

Itanium 은 완화된 일관성 모델을 제공해서 명시적 메모리 배리어 명령이나 종속성이 없으면 Itanium 은 임의적으로 메모리 참조를 재배치 할 수 있습니다 [Int02b]. Itanium 은 `mf` 라는 이름의 메모리 장벽 명령을 제공합니다만 로드, 스토어, 그리고 일부 어토믹 인스트럭션을 위한 “반쪽 메모리 장벽” 변화기 또한 제공합니다 [Int02a]. `acq` 변화기는 뒤따르는 메모리 참조 명령이 `acq` 전으로 재배치 되는 것을 막습니다만 앞의 메모리 참조 명령이 `acq` 뒤로 재배치 되는 것은 허용하는데, Armv8 `load-acquire` 명령과 비슷합니다. 비슷하게 `rel` 변화기는 앞의 메모리 참조 명령이 `rel` 뒤로 재배치 되는 것은 막지만 뒤따르는 메모리 참조 명령이 `rel` 앞으로 재배치 되는 것은 허용합니다.

이 반쪽 메모리 장벽들은 크리티컬 섹션에 유용한데, 크리티컬 섹션은 오퍼레이션을 크리티컬 섹션에 밀어 넣는 건 안전하지만 그것들이 밖으로 새어나가면 치명적이기 때문입니다. 그러나, 이 속성을 갖는 몇몇 CPU 중 하나로써 Itanium 은 한때 리눅스의 락 획득과 해제에 연관된 메모리 순서 규칙을 정의했습니다.¹⁹ 이상하겠지만, 실제 Itanium 하드웨어는 `load-acquire` 와 `store-release` 명령을 완전한 배리어로 구현할 것이라는 소문을 가졌습니다. 그러나, Itanium 은 명령어 집합에 `load-acquire` 와 `store-release` 의 (실제가 아니라면) 컨셉을 소개한 최초의 주류 CPU 였습니다.

Quick Quiz 15.42: 하드웨어가 반쪽 메모리 배리어를 가질 수 있다면, 락킹 기능은 왜 컴파일러가 메모리 참조 명령을 락 기반 크리티컬 섹션 안으로 옮길 수 있게 하지 않죠?

Itanium `mf` 명령은 리눅스 커널의 `smp_rmb()`, `smp_mb()`, 그리고 `smp_wmb()` 기능에 사용되었습니다. 반대의 지속적인 루머에도 불구하고 “`mf`”라는 이름은 “memory fence”의 약자라고 합니다.

¹⁹ 지금은 PowerPC 가 이 모호한 혜택을 갖는 아키텍쳐입니다.

Itanium 은 또한 `mf` 명령을 포함한 `release` 오퍼레이션을 위한 전역적 완전 순서를 제공합니다. 이는 `transitivity` 의 노선을 제공하는데, 이는 특정 코드 조각이 어떤 액세스가 일어났음을 보게 된다면 모든 뒤의 코드 조각은 또한 이 앞의 액세스가 일어난 것으로 본다는 겁니다. 모든 관여된 코드 조각이 올바르게 메모리 배리어를 사용한다는 가정 하입니다.

마지막으로, Itanium 은 리눅스 커널을 지원하며 같은 변수로의 평범한 로드를 재배치 할 수 있는 유일한 아키텍쳐입니다. 리눅스 커널은 `READ_ONCE()` 가 특정 CPU 에 의한 같은 변수로의 것을 포함한 모든 `READ_ONCE()` 순서를 강제하는, `ld,acq` 명령으로 번역되는 `volatile` 로드를 만들어내기 때문에 이 문제를 막습니다.

15.5.5 MIPS

MIPS 메모리 모델 [Wav16, page 479] 은 Arm, Itanium, 그리고 POWER 의 것을 닮았는데, 기본적으로는 완화된 순서규칙을 가지지만 종속성은 지킵니다. MIPS 는 다양한 종류의 메모리 배리어 명령을 갖지만 그것들을 하드웨어 주의사항과 연결하지 않고, 그대신 리눅스 커널과 C++11 표준 [Smi19] 에 의해 제공되는 사용처에 연결하는데 Armv8 추가사항과 비슷한 방법입니다:

SYNC

OCTEON 시스템을 위한 v4.13 리눅스 커널의 `smp_mb()` 구현에 사용된, 메모리 참조를 포함해 여러 하드웨어 오퍼레이션을 위한 완전한 배리어.

SYNC_WMB

OCTEON 시스템에서 `syncw` 를 통해 v4.13 리눅스 커널의 `smp_wmb()` 구현을 위해 사용될 수 있는 쓰기 메모리 배리어. 다른 시스템은 평범한 `sync` 를 사용합니다.

SYNC_MB

전체 메모리 배리어이나 메모리 오퍼레이션만을 위해 작동함. 이는 C++ `atomic_thread_fence(memory_order_seq_cst)` 를 구현하는데 사용될 수 있습니다.

SYNC_ACQUIRE

C++ 의 `atomic_thread_fence(memory_order_acquire)` 를 구현하는데 사용될 수 있는 acquire 메모리 배리어. 이론상, 이는 v4.13 리눅스 커널의 `smp_load_acquire()` 를 구현하는 데에 사용될 수 있는데, 실전에서는 `sync` 가 대신 사용됩니다.

SYNC_RELEASE

C++ 의 `atomic_thread_fence(memory_order_release)` 를 구현하는데에 사용될 수 있는 release

메모리 배리어. 이론상, v4.13 리눅스 커널의 `smp_store_release()` 기능을 구현하는 데에 사용될 수 있으나, 실제로는 `sync` 가 사용됩니다.

SYNC_RMB

v4.13 리눅스 커널에 의해 지원되는 현재 MIPS 구현들은 순서를 강제하기 위한 명시적 명령을 필요로 하지 않는다는 점을 제외하면 이론상 리눅스 커널의 `smp_rmb()` 구현에 사용될 수 있는 읽기 메모리 배리어.

SYNCL

Just-in-time (JIT) 컴파일러에 의해 생성되는 것과 같은 스스로를 수정하는 코드를 허용하기 위해 다른 명령들과 함께 사용되는 명령 캐쉬 동기화 기능.

MIPS 아키텍쳐를 위한 비정형적 논의는 MIPS 가 Arm 과 POWER 의 그것과 유사한 *transitivity* 와 *cumulativity* 정의를 가짐을 알게 합니다. 그러나, 다른 MIPS 구현은 다른 메모리 순서 속성을 가질 수 있는 것으로 보이므로 여러분이 사용하는 특정 MIPS 구현을 위한 문서를 보는게 중요합니다.

15.5.6 POWER / PowerPC

POWER 와 PowerPC CPU 제품군은 다양한 메모리 배리어 명령을 [IBM94, LHF05] 갖습니다:

`sync` 는 모든 앞의 오퍼레이션이 모든 뒤따르는 오퍼레이션이 시작하기 전에 완료된 것으로 보이게 만듭니다. 따라서 이 명령은 매우 비용이 높습니다.

`lwsync` (lightweight sync) 는 로드를 뒤따르는 로드 스토어에 대해 순서 잡으며, 스토어에 대해서도 순서를 잡습니다. 그러나, 이는 스토어를 뒤따르는 로드에 대해 순서잡지는 않습니다. `lwsync` 명령은 *load-acquire* 와 *store-release* 오퍼레이션을 구현하는데 사용될 수 있습니다. 흥미롭게도, `lwsync` 명령은 x86, z System, 그리고 우연히도 SPARC TSO 에서의 것과 동일한 CPU 내 순서를 강제합니다. 그러나, `lwsync` 명령을 각 메모리 참조 명령 사이에 위치시키는 것은 x86, z System, 또는 SPARC TSO 메모리 순서 규칙을 만들지는 않을 겁니다. 이 시스템들에서는 한쌍의 CPU 가 개별적으로 다른 변수에 스토어를 수행하면 모든 다른 CPU 가 이 스토어들의 순서에 동의합니다. PowerPC 에서는 `lwsync` 명령이 각 메모리 참조 명령 쌍 사이에 있다고 해도 그렇지 않은데, PowerPC 는 *multicopy atomic* 하지 않기 때문입니다.

`eieio` (뭐의 약자인지 궁금하시다면, *enforce in-order execution of I/O*) 는 모든 앞의 캐쉬될 수 있는 스토어가 모든 뒤따르는 스토어보다 먼저 완료된 것으로 나타나게 합니다. 그러나, 캐쉬될 수 있는 메모리로의 스토어는 캐쉬될 수 없는 메모리로의 스토어와 별개로 순서지어지는데, 이는 `eieio` 가 MMIO 스토어가 스팬락 해제를 앞서는 걸 강제하지 않음을 의미합니다.

`isync` 는 모든 앞의 명령들이 모든 뒤따르는 명령이 수행을 시작하기 전에 완료된 것으로 나타나게 합니다. 이는 앞의 명령들이 그것들이 생성할 수 있는 모든 trap 이 일어났거나 일어나지 않을 것이 보장되었을 정도로, 그리고 이 명령의 모든 부가작용(예를 들면, page-table 변경) 뒤따르는 명령에 의해 보일 정도로 충분히 진행되었어야만 함을 의미합니다. 그러나, 이는 모든 메모리 참조가 순서잡힐 것을 강제하지 않고, 오로지 그 명령 자체의 실제 수행만을 강제합니다. 따라서, 로드는 여전히 캐쉬되어 있는 오래된 값을 반환할 수 있으며 `isync` 명령은 앞서 저장된 값을 스토어 버퍼로부터 비워지게 강제하지 않습니다.

불행히도, 이 명령들 중 어느 것도 모든 스토어가 순서지어져야 하지만 `sync` 명령의 높은 오버헤드를 요구하지는 않는 리눅스의 `wmb()` 명령에 정확히 맞아떨어지지 않습니다. 그러나 다른 선택지는 없습니다: ppc64 버전의 `wmb()` 와 `mb()` 는 비용이 높은 `sync` 명령으로 정의되었습니다. 그러나, 리눅스의 `smp_wmb()` 명령은 MMIO 에 결코 쓰이지 않아서 (드라이버가 MMIO 들을 UP 는 물론 SMP 커널에서도 순서 잡아야 하기 때문), 더 가벼운 `eieio` 나 `lwsync` 명령으로 정의되었습니다 [MDR16]. 이 명령은 다섯개 모음으로 구성된 이름으로 독특합니다. `smp_mb()` 명령 또한 `sync` 명령으로 정의되었습니다만, `smp_mrb()` 와 `rmb()` 는 모두 더 가벼운 `lwsync` 명령으로 정의되었습니다.

POWER 는 *transitivity* 를 관측하는데 사용될 수 있는 “*cumulativity*” 를 제공합니다. 올바르게 사용되면 앞의 코드 조각의 결과를 보는 모든 코드는 이 앞의 코드 조각 그 자체가 본 것들에 대한 액세스도 보게 됩니다. 훨씬 많은 세부사항이 McKenney 와 Silvera 에 의해 [MS09] 정리되어 있습니다.

POWER 는 Arm 과 거의 같은 방식으로 제어 종속성을 지킵니다만 POWER `isync` 명령은 Arm ISB 명령으로 대체되어야 합니다.

Armv8 과 같이, POWER 는 `smp_mb__after_spinlock()` 이 완전한 메모리 배리어가 될 것을 필요로 합니다. 또한, POWER 는 `smp_mb__after_unlock_lock()` 이 완전한 메모리 배리어가 될 것을 필요로 하는 유일한 아키텍쳐입니다. 두 경우 모두, 이는 POWER 의

락킹 기능의 완화된 순서 속성 때문이며 `lwsync` 명령의 사용이 획득과 해제 둘 다에 순서를 제공해야 하기 때문입니다.

POWER 제품군의 많은 제품들이 비일관적 인스트럭션 캐시를 가져서 메모리로의 스토어는 인스트럭션 캐시에 반영되지 않을 수 있습니다. 고맙게도 몇몇 사람들이 스스로를 수정하는 코드를 요즘 작성하지만, JIT와 컴파일러는 항상 그러지는 않습니다. 더 나아가, 최근에 수행된 프로그램을 다시 컴파일하는 건 CPU의 시점에서는 스스로를 수정하는 코드처럼 보입니다. `icbi` 명령 (instruction cache block invalidate)은 인스트럭션 캐시로부터 특정 캐시 라인을 무효화 시키며 이 상황에 쓰일 수 있을 겁니다.

15.5.7 SPARC TSO

SPARC의 TSO (total-store order)가 리눅스와 Solaris 양쪽에서 사용되지만, 이 아키텍처는 PSO (partial store order)와 RMO (relaxed-memory order) 또한 정의합니다. RMO에서 수행되는 모든 프로그램은 PSO나 TSO에서도 수행될 것이며, 비슷하게, PSO에서 수행되는 프로그램은 TSO에서도 수행될 겁니다. 공유 메모리 병렬 프로그램을 이와 다른 방향으로 읊기는 것은 주의 깊은 메모리 배리어 삽입을 필요로 할 겁니다.

SPARC의 PSO와 RMO 모드는 오늘날 그렇게 널리 사용되지는 않지만 그것들은 세밀한 순서 제어를 가능하게 하는 매우 유연한 메모리 배리어 명령을 크게 늘렸습니다:

StoreStore는 앞의 스토어를 뒤의 스토어에 대해 순서 잡습니다. (이는 리눅스 `smp_wmb()` 기능에서 사용됩니다.)

LoadStore는 앞의 로드를 뒤의 스토어에 대해 순서 잡습니다.

StoreLoad는 앞의 스토어를 뒤의 로드에 대해 순서 잡습니다.

LoadLoad는 앞의 로드를 뒤의 로드에 대해 순서 잡습니다. (이는 리눅스 `smp_rmb()` 기능에서 사용됩니다.)

Sync는 모든 앞의 오퍼레이션을 뒤의 오퍼레이션 전에 완료시킵니다.

MemIssue는 앞의 메모리 오퍼레이션을 뒤의 메모리 오퍼레이션 전에 완료시키는데, 일부 memory-mapped I/O에 중요합니다.

Lookaside는 MemIssue와 같은 일을 하지만 앞의 스토어와 뒤따르는 로드에만 적용되며 거기서도

같은 메모리 위치에 대해 액세스하는 스토어와 로드에 대해서만입니다.

그래서, 왜 “`membar #MemIssue`”가 필요할까요? “`membar #StoreLoad`”는 뒤의 로드가 그 값을 스토어 베퍼로부터 가져올 수 있어서 그 쓰기가 MMIO 레지스터로의 것이었다면 읽일 값에 부가작용을 일으킬 수 있는 경우라면 문제가 될 수 있기 때문입니다. 반대로, “`membar #MemIssue`”는 이 로드가 수행되기 전에 스토어 베퍼가 비워지길 기다리고, 따라서 이 로드는 그 값을 MMIO 레지스터에서 정말로 읽어오게 합니다. 드라이버는 “`membar #Sync`”를 대신 사용할 수 있으나, 보다 비용이 높은 “`membar #Sync`”의 추가적 기능이 필요치 않을 때에는 더 저렴한 “`membar #MemIssue`”가 선호됩니다.

“`membar #Lookaside`”는 “`membar #MemIssue`”의 저렴한 버전으로, 특정 MMIO 레지스터로의 쓰기가 그 레지스터로부터 다음에 읽혀질 값에 영향을 끼치는 경우 유용합니다. 그러나, 특정 MMIO 레지스터로의 쓰기가 다른 MMIO 레지스터로부터 다음에 읽을 값에 영향을 끼치는 경우엔 더 비용이 높은 “`membar #MemIssue`”가 반드시 사용되어야 합니다.

SPARC는 인스트럭션 흐름이 수정되었을 때와 이 인스트럭션 중 무언가가 수행되는 시점 사이에 `flush` 명령이 사용될 것을 필요로 합니다 [SPA94]. 이는 해당 위치의 이전 값을 SPARC의 인스트럭션 캐시로부터 비워내기 위해 필요합니다. `flush`는 주소를 취하며, 인스트럭션 캐시로부터 그 주소만을 비움을 주의하십시오. SMP 시스템에서는 모든 CPU의 캐시가 비워지나 언제 CPU 밖의 비우기가 완료되는지를 아는 편리한 방법이 없으나, 구현 메모에 참고사항이 있긴 합니다.

그러나 다시 말하지만, 리눅스 커널은 TSO 모드에서 SPARC를 수행하므로, 앞의 모든 `membar` 변종은 역사적 흥미거리일 뿐입니다. 특히, `smp_mb()` 기능은 다른 세개의 재배치가 TSO에 의해 금지되므로 `#StoreLoad`만을 필요로 합니다.

15.5.8 x86

역사적으로, x86 CPU는 모든 CPU가 특정 CPU의 메모리 쓰기 순서에 동의하게끔 “process ordering”을 제공했습니다. 이는 `smp_wmb()` 기능이 no-op이 될 수 있게 했습니다 [Int04b]. 물론, `smp_wmb()` 기능 전후로 재배치 될 수 있는 최적화를 막기 위한 컴파일러 지시어는 필요했습니다. 아주 옛날에는, 특정 x86 CPU는 로드를 위한 순서 보장을 제공하지 않아서 `smp_mb()`와 `smp_rmb()` 기능이 `lock; addl`로 확장되었습니다. 이 어토믹 인스트럭션은 로드와 스토어 둘 다에 대한 배리어로 동작합니다.

하지만 그건 과거의 일입니다. 최근에 Intel은 x86을 위한 메모리 모델을 출판했습니다 [Int07]. Intel의 현대 CPU는 이전의 명세서에서 이야기된 것보다 더 강화된 순서를 제공하는 것으로 드러났으며, 따라서 이 모델은 이 현대 동작만을 요합니다. 더 최근에는 Intel이 x86을 위한 업데이트된 메모리 모델 [Int11, Section 8.2]을 출판했는데, 여기선 스토어에 대한 전체 전역 순서를 강요합니다만, 개별 CPU는 여전히 스스로의 스토어가 전체 전역 순서가 파악하는 것보다 일찍 볼 수 있게 합니다. 이 전체 순서에 대한 예외는 스토어 베파와 연관된 중요한 하드웨어 최적화를 허용하기 위해 필요합니다. 또한, x86은 다른 multicity 원자성을 제공하는데, 예를 들면 CPU 0이 CPU 1의 스토어를 봤다면 CPU 0은 CPU 1이 그 스토어 전에 봤던 모든 스토어를 볼 것이 보장됩니다. 소프트웨어는 이 하드웨어 최적화를 덮기 위해 어토믹 오퍼레이션을 사용할 수 있는데, 어토믹 오퍼레이션이 비 어토믹 베파의 그것들에 비해 비용이 높은 이유 중 하나입니다.

특정 메모리 위치에 가해지는 어토믹 명령들은 모두 같은 크기여야 한다는 것을 [Int16, Section 8.1.2.2] 알아 두는게 중요합니다. 예를 들어, 한 CPU가 원자적으로 한 바이트를 증가시키는 사이 다른 CPU는 같은 위치에 4-바이트 원자적 값 증가를 수행하는 프로그램을 작성한다면, 알아서 하세요.

어떤 SSE 명령은 완화된 순서를 갖습니다 (clflush와 비임시적 이동 명령 [Int04a]). 이 비임시적 이동 명령을 사용하는 코드는 `smp_mb()`를 위해 `mfence`를, `smp_rmb()`를 위해 `lfence`를, 그리고 `smp_wmb()`를 위해 `sfence`를 사용할 수 있습니다. x86 CPU의 일부 오래된 변종들은 비순차적 스토어를 가능하게 하기 위해 모드 bit를 가지며 이런 CPU에서 `smp_wmb()`는 `lock; addl`로 정의되어야만 합니다.

최근의 x86 구현이 특별한 명령 없이 스스로를 수정하는 코드를 수용하지만, 과거와 잠재적 미래 x86 구현에 완전히 호환되려면, CPU는 `jump` 명령이나 순차화 명령 (예: `cpuid`)을 코드 수정과 수행 사이에 수행해야만 합니다 [Int11, Section 8.1.3].

15.5.9 z Systems

z Systems 기계는 앞서 360, 370, 390 그리고 Zseries [Int04c] 라 알려진 IBM mainframe 제품군을 만드는데 쓰입니다. 병렬성은 z Systems에 뒤늦게 도입되었습니다만 이 mainframe들이 1960년대 중반에 처음 선적되었음을 생각하면 그렇게 늦은 건 아닙니다. “`bcr 15,0`” 명령이 리눅스 `smp_mb()` 기능을 위해 사용됩니다만 `smp_rmb()`와 `smp_wmb()` 기능들을 위해선 컴파일러 제약만으로 충분합니다. 이 제품군은 또한 Table 15.5에 보인 것과 같이 강력한 메모리 순서 규칙을 제공합니다. 특히, 모든 CPU가 다른 CPU들로부터의

관계없는 스토어들의 순서에 대해 동의하게 되어 있는 데, 즉, z Systems CPU 제품군은 완전히 multicity atomic 하며, 이 속성을 제공하는 유일한 상용 가능한 시스템입니다.

다른 대부분의 CPU와 같이 z Systems 아키텍처는 캐쉬 일관성 인스트럭션 흐름을 보장하지 않으며, 따라서 스스로를 수정하는 코드는 인스트럭션의 업데이트와 수행 사이에 인스트럭션 직렬화 명령을 수행해야 합니다. 그렇다고는 하나, 많은 실제 z Systems 기계들이 인스트럭션 직렬화 없이 스스로를 수정하는 코드를 받아들이고 있습니다. z Systems 명령 집합은 `compare-and-swap`, 일부 종류의 분기문 (예를 들어, 앞서 이야기된 “`bcr 15,0`” 명령), 그리고 `test-and-set` 같은 많은 직렬화 명령 집합을 제공합니다.

15.6 Where is Memory Ordering Needed?

Almost all people are intelligent. It is method that they lack.

F. W. Nichol

이 섹션은 Table 15.3와 Section 15.1.3를 다시 방문해 더 정교한 경험적 법칙과 함께 그 사이의 논의를 요약합니다.

그 첫번째 경험적 규칙은 메모리 순서 오퍼레이션들이 최소 두개의 쓰레드에서 최소 두개의 공유변수 사이의 상호작용을 가질 가능성이 있을 때에만 필요하다는 겁니다. 그 사이의 것들에 의해, 이 단일 문장은 Section 15.1.3의 기본적 경험적 법칙의 만흔 것을 담아내는데, 예를 들어 “memory-barrier pairing”은 “사이클”의 두개 쓰레드에서의 특수 경우임을 명심하십시오. 또한, 항상 그렇듯 싱글쓰레드 프로그램이 충분한 성능을 제공한다면, 왜 병렬성을 고려하겠습니까?²⁰ 어쨌건, 병렬성을 막는 것은 메모리 순서 오퍼레이션의 추가 비용을 막습니다.

그 두번째 경험적 법칙은 로드-베파링 상황에 연관됩니다: 특정 사이클에서의 모든 쓰레드간 통신이 `store-to-load` 연결을 사용한다면 (즉, 다음 쓰레드의 로드는 앞의 쓰레드가 저장한 값을 반환한다면), 최소한의 순서 짓기로 충분합니다. 최소한의 순서 짓기는 종속성과 `acquire`는 물론 더 강한 순서 오퍼레이션들도 포함합니다.

그 세번째 경험적 법칙은 `release-acquire` 연결에 연관됩니다: 주어진 사이클의 모든 링크가 하나만 제외하고는 모두 `store-to-load` 연결이라면, Listings 15.23

²⁰ 취미가들과 연구자들은 물론 여러 이유로 이를 무시하셔도 무방하겠습니다.

and 15.24 에 보인 것처럼 그 store-to-load 연결의 각 짹에 release-acquire 짹을 사용하는 것으로 충분합니다. 여러분은 이 acquire 를 허용되는 환경에서는 종속성으로 대체할 수 있는데, C11 표준의 메모리 모델은 종속성을 완전히 존중하지 않음을 명시하시기 바랍니다. 따라서, 로드를 앞서는 종속성은 `READ_ONCE()` 나 `rcu_dereference()` 를 통해 이루어져야 합니다: 평범한 C 언어 로드는 충분치 않습니다. 또한, 여러분의 컴파일러에 의해 부서진 종속성은 어떤 것도 순서잡지 않을테니 Sections 15.3.2 and 15.3.3 을 주의 깊게 다시 보세요. 하나의 store-to-load 가 아닌 링크를 공유하는 두개의 쓰레드는 보통 `WRITE_ONCE()` 더하기 `smp_wmb()` 를 `smp_store_release()` 로, `READ_ONCE()` 더하기 `smp_rmb()` 는 `smp_load_acquire()` 로 교체할 수 있습니다. 그러나, 현명한 개발자는 그런 교체를 주의 깊게 검사할 텐데, 예를 들면 Section 12.3 에 설명된 것처럼 herd 도구를 사용하는 겁니다.

Quick Quiz 15.43: 왜 store-to-load 가 아닌 load-to-store 와 store-to-store 링크에는 보다 무거운 순서 오퍼레이션을 사용해야만 하죠? 대체 무엇이 store-to-load 링크를 그렇게 특별하게 합니까???

마지막인 그 네번째 경험적 규칙은 어디에 전체 메모리 배리어가 (또는 그보다 강력한 것이) 필요한지 정의 합니다: 주어진 사이클이 두개 이상의 store-to-load 링크가 아닌 링크를 갖는다면 (즉, 총 두개 이상의 링크가 load-to-store 나 store-to-store 링크라면), 여러분은 이 사이클의 각 store-to-load 가 아닌 링크들 사이에 최소 하나의 전체 메모리 배리어를 필요로 하며, 이는 Listing 15.19 와 Quick Quiz 15.24 의 답변에 설명되어 있습니다. 전체 배리어는 `smp_mb()`, 성공한 전체 강력도의 `void` 가 아닌 atomic RMW 오퍼레이션, 그리고 `smp_mb__before_atomic()` 또는 `smp_mb__after_atomic()` 와 함께 사용된 다른 atomic RMW 오퍼레이션들을 포함합니다. 모든 RCU 의 grace-period 대기 기능 (`synchronize_rcu()` 와 그 친구들) 또한 전체 배리어로 동작합니다만 `smp_mb()` 보다 훨씬 비용이 높습니다. 전체 배리어가 확장성을 손상시키는 것보다는 성능을 보통 덜 손상시키긴 합니다만 힘은 비용과 함께 옵니다.

이 규칙들을 다시 정리해 봅니다:

1. 메모리 순서 오퍼레이션들은 최소한 두개의 변수들이 최소 두개의 쓰레드에 의해 공유될 때에만 필요합니다.
2. 사이클의 모든 링크가 store-to-load 링크라면, 최소한의 순서 보장으로 충분합니다.
3. 만약 사이클의 링크들 중 하나를 제외한 모두가 store-to-load 링크라면, 각 store-to-load 링크는 release-acquire 쌍을 사용할 수 있습니다.

4. 그렇지 않다면, 최소 하나의 전체 배리어가 store-to-load 가 아닌 각 쌍 사이에 필요합니다.

이 네개의 경험적 규칙은 최소한의 보장을 다룸을 명심하세요. 특정 아키텍쳐는 Section 15.5 에서 설명한 것처럼 보다 상당한 보장을 제공할 수도 있습니다만 이 보장은 그 아키텍쳐에서만 수행되는 코드에만 의존할 수도 있습니다. 또한, 더 연관된 메모리 모델은 보다 높은 복잡도를 대가로 더 강한 보장을 제공할 수도 있습니다 [AMM⁺18]. 이 보다 정형적인 메모리 순서 논문에서, store-to-load 링크는 readers-from (rf) 링크의 예이고, load-to-store 링크는 from-reads (fr) 링크의 예이며, store-to-store 링크는 coherence (co) 링크의 예입니다.

마지막 조언입니다: 날것의 메모리 순서 기능을 사용하는 것은 최후의 수단입니다. 거의 항상 이미 존재하는 락킹이나 RCU 같은 기능을 사용해서 그 기능들이 여러분을 위한 메모리 순서 보장을 하게 하는 것이 더 낫습니다.

Chapter 16

Ease of Use

16.1 What is Easy?

When someone says “I want a programming language in which I need only say what I wish done,” give them a lollipop.

Alan J. Perlis, updated

사용 편의성 요구사항을 들여다보고자 한다면, 리눅스 커널 RCU 의 사용 편의성 버그가 RCU 사용을 통해 이용될 수 있는 리눅스 커널 보안 버그를 초래했음을 고려해 주십시오.

불행히도, “편하다”는 상대적인 용어입니다. 예를 들어, 많은 사람들이 15 시간의 비행기 여행을 약간의 고난으로 여깁니다—이동의 대안적인 모드, 특히 수영을 생각하하는 걸 멈추지 않는다면. 이는 사용하기 편한 API를 만드는 것은 여러분이 여러분의 의도된 사용자들이 사용하기에 무엇이 편한지를 충분히 이해할 것을 요구함을 의미합니다. 여러분에게 편의를 위해 무얼 하고 뭘 하지 않아야 할까요.

다음 질문이 그 요지를 보입니다: “오늘날 살아있는 인류 가운데 무작위로 골라진 사람에게 어떤 변화가 그 사람의 인생을 개선시킬까?”

모두의 삶을 도울 것이 보장된 하나의 변화는 존재하지 않습니다. 어쨌건, 엄청나게 다양한 사람이 존재하며, 그들의 필요한 것, 원하는 것, 소망, 그리고 포부도 모두 다양합니다. 기아로 굶주리는 사람은 음식을 필요로 할테지만 병적으로 비만인 사람에게 추가적인 음식은 죽음을 앞당길 겁니다. 많은 젊은 사람들에게 열렬히 소망될 높은 단계의 즐거움은 어떤 사람에게는 심장 마비에서 회복되기에 치명적일 겁니다. 어떤 사람에게는 성공을 위해 치명적인 정보가 정보 과다로 고통받는 누군가에게는 문제가 될 수도 있을 겁니다. 요약하자면, 여러분이 아무것도 모르는 사람들을 돋고자 기획된 소프트웨어 프로젝트를 위해 일하고 있다면, 여러분은 그

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.

사람들이 여러분의 프로젝트에 문제를 발견해도 놀라지 않아야 합니다.

여러분이 정말로 해당 사람들을 돋고자 한다면 추가된 기간, 거의 수년 동안을 그들과 가깝게 일하는 것 외에는 간단한 대안이 없습니다. 그러나, 여러분의 이상한 사용자들이 여러분의 소프트웨어에 행복해지는 경우를 증가시키기 위해 여러분이 할 수 있는 몇 가지 간단한 것들이 있으며, 이 간단한 것들이 다음 섹션에서 다루어집니다.

16.2 Rusty Scale for API Design

Finding the appropriate measurement is thus not a mathematical exercise. It is a risk-taking judgment.

Peter Drucker

이 섹션은 Rusty Russell 의 2003년 Ottawa Linux 심포지엄 키노드의 일부 [Rus03, Slides 39–57] 를 수정했습니다. Rusty 의 요점은 목적은 단순히 사용하기 편한 API를 만드는 게 아니라 API를 잘못 사용하기 어렵게 해야 한다는 겁니다. 그 결과, Rusty 는 그의 “Rusty Scale” 을 이 오용하기 어렵기 속성의 중요도의 내림차순으로 제안합니다.

다음 리스트는 Rusty Scale 을 리눅스 커널에 맞춰 일반화 해보려는 시도입니다:

1. 잘못되기 불가능할 것. 이는 모든 API 설계자드리 추구해야 하는 표준이지만 상상의 `dwim()`¹ 커맨드만이 근접할 수 있었습니다.
2. 컴파일러나 링커가 여러분이 잘못을 저지르게 못하게 할 것.

¹ `dwim()` 함수는 “do what I mean” 으로 펼쳐지는 축약어입니다.

3. 컴파일러나 링커는 여러분이 뭔가 잘못하면 경고 할 것. `BUILD_BUG_ON()`은 여러분의 사용자의 친구가 됩니다.
4. 가장 간단한 사용이 올바른 방법일 것.
5. 이름이 사용법을 설명하게 하기. 하지만 이름은 양 날의 검입니다. 비록 `rcu_read_lock()`이 코드를 reader-writer 락킹으로부터 변환하고자 하는 누군가에겐 충분히 평범하지만, 레퍼런스 카운팅에서 코드를 변환하는 사람에겐 어떤 경악을 금치 못하게 할 수도 있을 겁니다.
6. 올바르게 하지 않으면 수행시간에 항상 고장나게 하기. `WARN_ON_ONCE()`은 여러분의 사용자의 친구가 됩니다.
7. 일반적인 관습을 따르면 올바르게 되기. `malloc()` 라이브러리 함수가 좋은 예입니다. 메모리 할당을 잘못하기는 쉽지만 수많은 프로젝트가 적어도 대부분의 경우 이를 올바르게 관리합니다. `malloc()`을 Valgrind [The11] 와 함께 사용하는 것은 `malloc()`을 거의 “올바르게 하지 않으면 수행시간에 항상 고장나게 하기” 지점까지 옮깁니다.
8. 문서를 읽으면 올바르게 사용할 수 있게 하기.
9. 구현을 읽으면 올바르게 사용할 수 있게 하기.
10. 올바른 메일링 리스트 기록을 읽으면 올바르게 사용할 수 있게 하기.
11. 올바른 메일링 리스트 기록을 읽으면 잘못 사용하게 하기.
12. 구현을 읽으면 잘못 사용하게 하기. `rcu_read_lock()`의 최초 비 `CONFIG_PREEMPT` 구현은 [McK07a] 이 지점의 확장성에 대한 악명 높은 예입니다.
13. 문서를 읽으면 잘못 사용하게 하기. 예를 들어, DEC Alpha `wmb` 명령의 문서는 [Cor02] 많은 개발자들로 하여금 이 명령이 실제보다 훨씬 강력한 메모리 순서 규칙을 갖는 걸로 오해하게 만들었습니다. 나중의 문서는 이 지점을 분명케 했고 [Com01, Pug00], `wmb` 명령을 “문서를 읽으면 올바르게 사용하게 하기” 지점까지 옮겨놓았습니다.
14. 혼한 관례를 따르면 잘못 사용하게 하기. `printf()` 문은 이 지점의 한 예인데, 개발자들은 거의 항상 `printf()`의 반환 오류를 검사하지 않기 때문입니다.
15. 올바르게 사용하면 수행 시간에 고장나게 하기.
16. 이름이 그것을 어떻게 사용하면 안되는지 말하게 하기.
17. 명백한 사용이 잘못되기. 리눅스 커널의 `smp_mb()` 함수는 이 지점의 한 예입니다. 많은 개발자들이 이 함수가 실제로 수행하는 것보다 훨씬 강한 순서 규칙을 갖는다고 가정합니다. Chapter 15 가 이 실수를 막기 위한 정보를 리눅스 커널 소스 트리의 Documentation 과 tools/memory-model 디렉토리들처럼 담고 있습니다.
18. 컴파일러나 링커가 여러분이 올바른 사용을 했을 때 경고를 하기.
19. 컴파일러나 링커가 여러분이 올바른 사용을 할 수 없게 하기.
20. 올바르게 사용하기가 불가능하기. `gets()` 함수는 이 지점의 유명한 예입니다. 사실, `gets()`는 무조건적 버퍼 오버플로우 보안 구멍으로 가장 잘 설명될 수도 있을 겁니다.

16.3 Shaving the Mandelbrot Set

Simplicity does not precede complexity,
but follows it.

Alan J. Perlis

유용한 프로그램들의 집합은 깨끗하게 구분되는 선이 없다는 점에서 Mandelbrot 집합 (Figure 16.1에 보여 있습니다)을 닮았습니다—만약 그런 선이 있다면, 이 문제는 해결 가능할 겁니다. 그러나 우리는 실제 사람들이 사용할 수 있는 API가 필요하지, 모든 잠재적 사용을 위해 박사 학위를 필요로 하는 API를 필요로 하지 않습니다. 따라서, 우린 “Mandelbrot set 을 깎아내어”² API의 사용을 잠재적 사용의 완벽한 집합의 쉽게 설명될 수 있는 부분집합으로 제한해야 합니다.

그런 깎아냄은 반생산적으로 보일 수 있습니다. 어쨌건, 어떤 알고리즘이 동작한다면, 왜 사용하지 않습니까?

왜 적어도 일부의 깎아냄은 반드시 필요한지 이해하기 위해, 데드락을 방지하지만 아마도 가능한 가장 나쁜 방법으로 그걸 하는 락킹 설계를 생각해 봅시다. 이 설계는 순환형 이중 링크드 리스트를 사용하는데, 이 리스트는 헤더 원소와 함께 시스템의 각 쓰레드를 위해 하나의 원소를 가집니다. 새로운 쓰레드가 생성될 때, 그

² Josh Triplett 덕입니다.

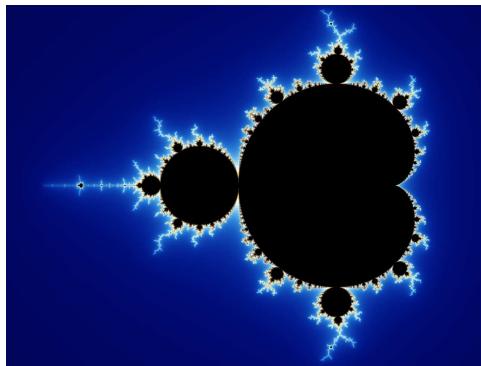


Figure 16.1: Mandelbrot Set (Courtesy of Wikipedia)

부모 쓰레드는 이 리스트에 새 원소를 삽입해야 하는데, 이는 어떤 종류의 동기화를 필요로 합니다.

이 리스트를 보호하기 위해 전역 락을 사용하는 방법이 있습니다. 그러나, 이는 쓰레드가 빈번히 생성되고 삭제된다면 병목이 될 수 있습니다.³ 또 다른 방법은 해쉬 테이블을 사용하고 각 해쉬 베킷을 락킹하는 것입니다만 이는 리스트를 순서대로 수캐닝 하는 성능을 나쁘게 할 수 있습니다.

세 번째 방법은 개별 리스트 원소를 락킹하고 삽입 시에 그 앞과 뒤 원소들의 락도 잡게끔 하는 겁니다. 두 락이 모두 잡혀야 하므로, 우린 이걸 어느 순서로 잡을지 정해야 합니다. 두개의 관례적 방법은 이 락을 주소 순서대로 잡는 것, 또는 리스트에 나타나는 순서대로 잡아서 락킹 되는 두개의 원소 중 하나인 경우엔 그 헤더가 항상 먼저 잡히게 하는 겁니다. 그러나, 이 방법들 둘 다 특수한 검사와 분기를 필요로 합니다.

깎아내려야 하는 해결책은 리스트 순서대로 무조건적으로 락을 잡는 겁니다. 하지만 데드락은 어떡하죠? 데드락은 일어날 수 없습니다.

이를 보기 위해, 0에서 시작해 이 리스트의 마지막 원소에 (이 리스트가 순환형이므로, 헤더의 앞 원소가 됩니다) N 까지 번호를 매겨봅시다. 비슷하게, 시스템의 쓰레드는 0부터 $N - 1$ 까지 번호를 매깁니다. 각 쓰레드가 어떤 연속적인 한쌍의 원소를 락킹하려 한다면, 그 쓰레드 중 하나는 두 락을 모두 획득할 수 있을 것이 보장됩니다.

왜일까요?

이 리스트의 전체에 달기에는 쓰레드의 수가 충분치 않기 때문입니다. 쓰레드 0이 원소 0의 락을 잡으려 한다고 해봅시다. 블록되기 위해서는, 어떤 다른 쓰레드가 이미 원소 1의 락을 잡았어야 하는데 쓰레드 1이 그랬다고 해봅시다. 비슷하게, 쓰레드 1이 블락되기 위

³ 운영체제에 대한 깊은 이해가 있는 분들이라면 부디 불신을 멈추십시오. 불신을 멈출 수 없는 분들은 더 나은 예를 제공해주시면 좋겠습니다.

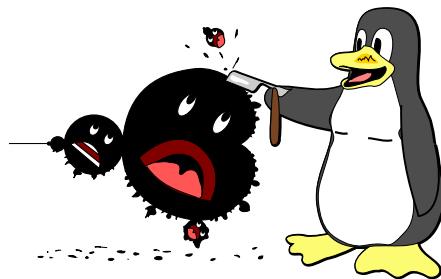


Figure 16.2: Shaving the Mandelbrot Set

해선 어떤 다른 쓰레드가 원소 2의 락을 잡았어야 하며, 그런식으로 원소 $N - 1$ 의 락을 잡고 있을 쓰레드 $N - 1$ 까지 갑니다. 하지만 더이상의 쓰레드가 없으므로 쓰레드 $N - 1$ 은 블록될 수 없습니다. 따라서, 데드락은 일어날 수 없습니다.

그러니 이 즐겁고 작은 알고리즘을 사용하지 않을 이유가 어디 있겠습니까?

사실 여러분이 정말로 그걸 사용하고 싶어한다면 우린 여러분을 막을 수 없습니다. 그러나, 우리는 그런 코드가 우리가 신경쓰는 프로젝트에 포함되는 걸 반대할 수 있습니다.

그러나, 이 알고리즘을 사용하기 전에, 다음 Quick Quiz를 생각해 보세요.

Quick Quiz 16.1: 원소의 삭제에도 비슷한 알고리즘을 사용할 수 있나요?

■

여기서의 중요한 사실은 이 알고리즘은 극단적으로 특수화 되었으며 (특정한 크기의 리스트에서만 동작합니다), 잘못되기도 쉽다는 겁니다. 리스트에 노드를 추가하는데 사고로 실패하는 모든 버그는 데드락을 초래할 수 있습니다. 실제로, 노드를 약간 늦게 추가하는 것만으로도 쓰레드의 수를 늘리는 게 그럴 수 있듯 데드락을 초래할 수 있습니다.

또한, 앞에서 설명된 다른 알고리즘은 “좋고 충분합니다”. 예를 들어, 주소 순서로 락을 획득하는 건 충분히 간단하고 빠르며, 모든 크기의 리스트의 사용을 허용합니다. 다만 빈 리스트와 하나의 원소만 포함하는 리스트의 특수 경우에 주의하세요!

Quick Quiz 16.2: 와우! 도대체 무엇이 누군가로 하여금 이것 만큼 깎아낼 가치가 있는 알고리즘을 생각해내게 할까요???

■

요약하자면, 우린 어떤 알고리즘을 그게 동작한다는 이유만으로 사용하지 않습니다. 우린 그대신 그걸 배울 가치가 충분할 정도로 충분히 유용한 알고리즘들만 사용합니다. 그 알고리즘이 더 어렵고 복잡하다면, 그것

은 더 일반적으로 유용해서 그걸 배우고 그것의 버그를 고치는 고통이 가치있게 해야 합니다.

Quick Quiz 16.3: 이 규칙에 대한 예외를 하나 주세요.

예외들을 제외하고도, 우린 Figure 16.2 에 보인 것처럼 우리의 프로그램이 유지가능하게 지속되게끔 소프트웨어 “Mandelbrot set” 을 깎아내기를 계속해야 합니다.

Prediction is very difficult, especially about the future.

Niels Bohr

Chapter 17

Conflicting Visions of the Future

이 챕터는 병렬 프로그래밍의 미래에 대한 엇갈리는 예측들을 보입니다. 이것들 중 어느것이 진실이 될지는 명확치 않으며, 실제로 이 중 어느거라도 진실이 될지 명확치 않습니다. 그러나 각 예측은 거기 헌신하는 지지자들이 있으며 충분한 수의 사람이 충분히 열렬히 지지를 한다면 여러분은 그것이 그 지지자들의 생각, 발언, 그리고 행위에 영향을 줄 수 있는 형태로 그것의 존재를 다루어야 할 것이기에 중요합니다. 게다가 이 예측들 중 하나 또는 여러개가 정말로 이루어질 수 있습니다. 하지만 대부분은 그렇지 않습니다. 뭐가 뭔지 이야기하면 부자가 될겁니다 [Spi77]!

따라서, 다음 세션들은 트랜잭션 메모리, 하드웨어 트랜잭션 메모리, 회귀 테스트에서의 정형적 검증, 그리고 병렬 함수형 프로그래밍의 개론을 다룹니다. 그러나 먼저, 2000년대 초에서 가져온 예언에 대한 조심스러운 이야기로 시작합니다.

17.1 The Future of CPU Technology Ain't What it Used to Be

A great future behind him.

David Maraniss

과거의 것들은 수많은 해의 경험을 거친 렌즈로 보면 무척 간단하고 순수해 보입니다. 그리고 2000년대 초는 Moore's Law 가 그때는 전통적이던 CPU 클럭 주파수의 증가를 가져다 주는 것에 대한 임박한 실패로부터 결백했습니다. 오, 기술의 한계에 대한 가끔의 경고가 있긴 했습니다만 그런 경고는 수십년째 들려오고 있었습니다. 그걸 고려하고 다음 시나리오들을 생각해 봅시다:

1. Uniprocessor Über Alles (Figure 17.1),
2. Multithreaded Mania (Figure 17.2),

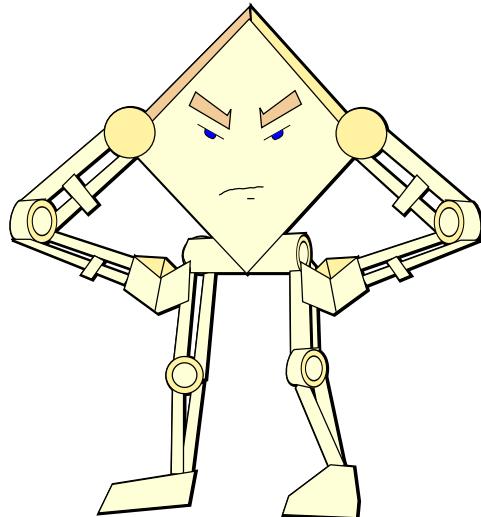


Figure 17.1: Uniprocessor Über Alles

3. More of the Same (Figure 17.3), and
4. Crash Dummies Slamming into the Memory Wall (Figure 17.4).
5. Astounding Accelerators (Figure 17.5).

이 시나리오 각각은 다음 세션들에서 다루어집니다.

17.1.1 Uniprocessor Über Alles

2004년에 다음과 같이 말해진 바와 같이 [McK04]:

이 시나리오에서, Moore's-Law 가 CPU 클럭 비율을 늘려가는 현상과 수평적으로 확장되는 컴퓨팅의 계속된 진보의 결합으로 인해

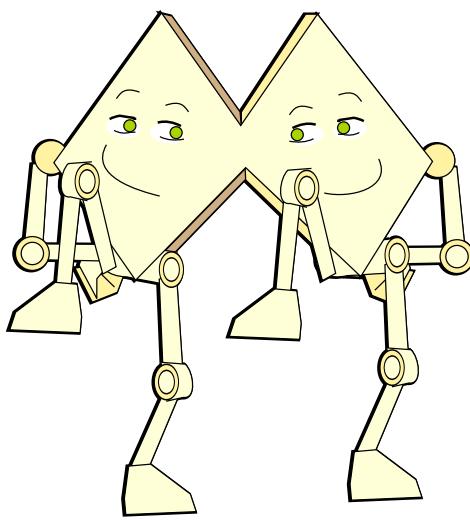


Figure 17.2: Multithreaded Mania

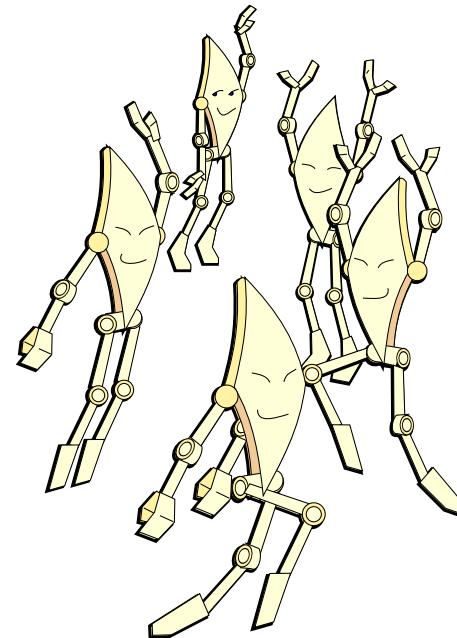


Figure 17.3: More of the Same

SMP 시스템을 무의미하게 만듭니다. 이 시나리오는 따라서 “Uniprocessor Über Alles”라고 불리게 되는데, 말 그대로 모든 것들보다도 유니프로세서를 사용하게 되는 시나리오입니다.

이 유니프로세서 시스템은 인스트럭션 오버헤드만을 겪게 될텐데, 메모리 배리어, 캐쉬 쓰래싱, 경쟁은 단일 CPU 시스템에 영향을 끼치지 않기 때문입니다. 이 시나리오에서, RCU는 NMI 와의 상호작용 같은 일부 어플리케이션에서만 유용합니다. RCU를 제공하지 않는 운영체제가 이를 도입할 필요를 겪을지는 명확치 않은데, 이미 RCU를 구현한 운영체제는 계속 그럴 수 있긴 합니다.

그러나, 최근의 멀티쓰레드 CPU의 발전은 이 시나리오는 실현 가능성에 상당히 적을 것을 암시하는 것 같습니다.

실제로 실현가능성이 적습니다! 그러나 더 많은 소프트웨어 커뮤니티가 그들이 병렬성을 받아들여야 한다는 사실을 꺼려했고, 그건 이 커뮤니티가 Moore's-Law가 제공한 CPU 코어 클럭 주파수 증가로 인한 “공짜 점심”이 정말로 끝났음으로 결론짓기 전이었습니다. 잊지 마세요: 믿음은 감정이지, 합리적 기술 사고 과정의 결과가 아닐 수 있습니다!

17.1.2 Multithreaded Mania

역시 2004년으로부터 [McK04]:

Uniprocessor Über Alles의 덜 극단적인 변종은 하드웨어 멀티쓰레딩을 제공하는 유니프로세서를 갖추는데, 실제로 멀티쓰레딩 제공 CPU는 많은 데스크탑과 랩톱 컴퓨터 시스템에서 오늘날의 표준입니다. 가장 공격적인 멀티쓰레딩 제공 CPU는 모든 캐쉬 계층을 공유하며, 따라서 CPU 대 CPU 메모리 응답 시간을 제거하고, 그 결과 전통적인 동기화 메커니즘에서의 성능 제약을 크게 줄입니다. 그러나, 멀티쓰레딩 제공 CPU는 여전히 메모리 배리어에 의해 발생하는 경쟁과 파이프라인 정지로 인한 오버헤드를 일으킬 겁니다. 더 나아가, 모든 하드웨어 쓰레드가 모든 캐쉬를 공유하므로, 특정 하드웨어 쓰레드에게 사용 가능한 캐쉬는 동일한 싱글 쓰레드 CPU에서 가능할 것의 한 부분에 불과할 텐데, 이는 큰 캐쉬 사용량을 갖는 어플리케이션의 성능을 하락시킬 겁니다. 또한 제한된 캐쉬 양이 RCU 기반 알고리즘이 그것의 grace period에 의한 추가적 메모리 소모로 인한 성능 문제를 일으킬 가능성도 있습니다. 이 가능성을 조사하는 건 미래의 일입니다.

그러나, 그런 성능 하락을 막기 위해선 여러 멀티 쓰레드 제공 CPU와 멀티 CPU 칩들이 최소한 캐쉬의 일부 계층을 하드웨어 쓰레드

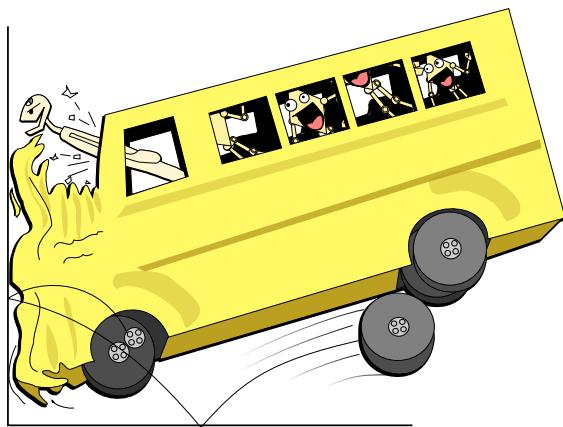


Figure 17.4: Crash Dummies Slamming into the Memory Wall

별로 분리시켜야 합니다. 이는 각 하드웨어 쓰레드에게 사용 가능한 캐쉬의 양을 늘릴 것입니다만, 다시 하드웨어 쓰레드 간에 전달되어야 하는 캐쉬라인들을 위한 메모리 응답시간 증가를 다시 일으킵니다.

그리고 우리 모두 이 이야기가 단일 소켓에 끼워진 단일 판 위에 여러개의 멀티쓰레딩 코어들이 올라가고 다양한 수준의 코어당 더 적은 수의 활성 쓰레드 수를 위한 최적화와 함께 어떻게 흘러왔는지 압니다. 그러면 질문은 미래의 공유 메모리 시스템은 항상 단일 소켓에 적합할 것인가 됩니다.

17.1.3 More of the Same

또다시 2004년으로부터 [McK04]:

더-많은-같은것들 시나리오는 메모리 응답시간 비율이 오늘날의 것과 대략적으로 비슷하게 유지될 것을 가정합니다.

이 시나리오는 실제로 변화를 가져오는데, 더 많은 같은 것들을 위해, interconnect 성능이 Moore's-Law로 인한 코어 CPU 성능의 증가와 함께 증가를 계속해야 하기 때문입니다. 이 시나리오에서, 파이프라인 중단, 메모리 응답시간, 그리고 경쟁으로 인한 오버헤드는 심각할 것으로 유지되지만 RCU는 그것이 오늘날 즐기는 것과 같은 높은 수준의 적용성을 유지하게 됩니다.

그리고 이 변화는 Moore's Law 가 여전히 제공하고 있는 통합의 여지껏 증가하는 수준의 것입니다. 하지만

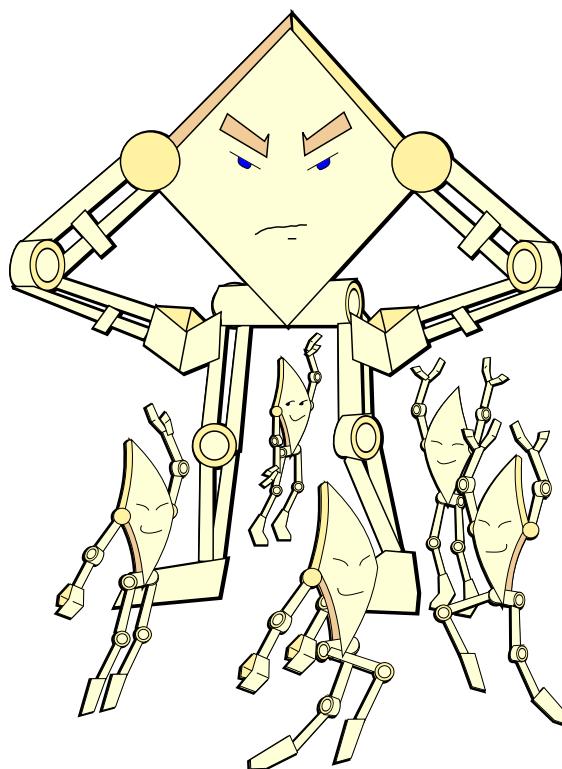


Figure 17.5: Astounding Accelerators

더 장기적으로는 어떻게 될까요? 판당 더 많은 CPU? 또는 더 많은 I/O, 캐쉬, 그리고 메모리?

서버는 앞의 전략을 선택한 듯 보이고 칩 위의 임베디드 시스템 (SoC)은 뒤의 것을 선택해 가고 있는 것으로 보입니다.

17.1.4 Crash Dummies Slamming into the Memory Wall

그리고 2004년으로부터 하나 더 인용합니다 [McK04]:

만약 Figure 17.6에 보인 메모리 응답시간 경향이 지속된다면, 메모리 응답시간은 인스트럭션 수행 오버헤드에 비해 상대적으로 증가 하길 계속할 겁니다. RCU를 상당히 사용하는 리눅스와 같은 시스템은 Figure 17.7에 보인 것처럼 RCU가 이득이 되는 더 많은 사용처를 찾을겁니다. 이 그림에서 보여지듯, RCU가 널리 사용되면 증가하는 메모리 응답시간 비율이 RCU에게 다른 동기화 메커니즘 대비 증가하는 장점을 제공합니다. 대조적으로, RCU를 적게 사용하는 시스템은 Figure 17.8

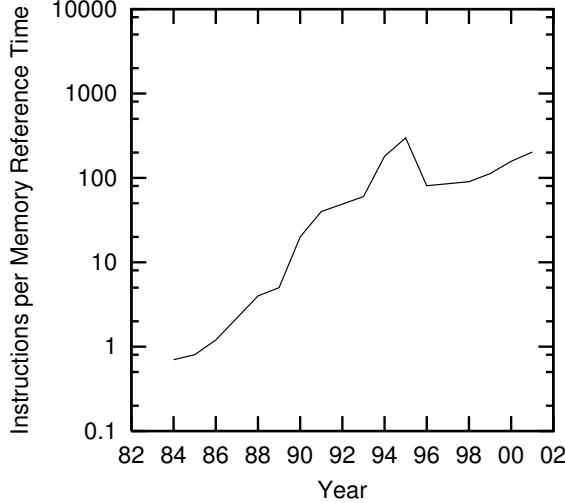


Figure 17.6: Instructions per Local Memory Reference for Sequent Computers

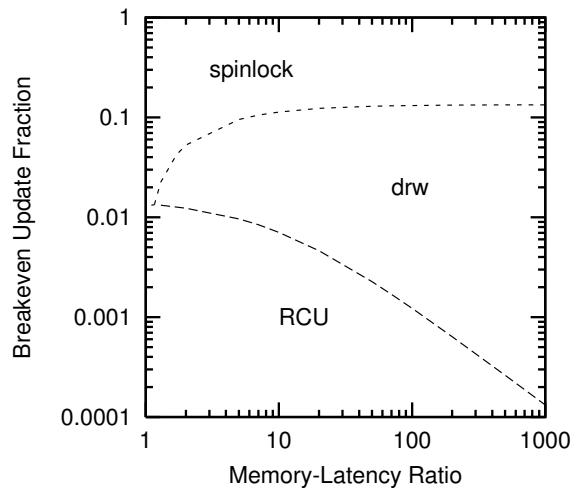


Figure 17.8: Breakevens vs. r, λ Small, Four CPUs

에 보인 것처럼 RCU 의 사용이 비용을 절감 할 수 있게끔 더 많은 읽기 비율을 필요로 합니다. 이 그림에서 볼 수 있듯, RCU 가 적게 사용된다면 증가하는 메모리 응답시간 비율은 RCU 를 다른 동기화 메커니즘 대비 단점이 더 많게 만듭니다. 리눅스는 높은 부하 시에 grace period 당 1,600 개가 넘는 콜백을 보였으므로, 리눅스는 앞의 카테고리에 속한다고 말해도 안전할 겁니다.

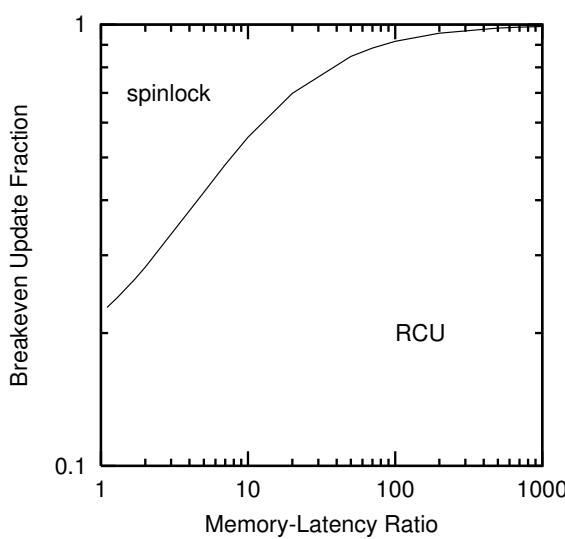


Figure 17.7: Breakevens vs. r, λ Large, Four CPUs

한편으로, 이 구절은 상당한 업데이트가 있는 워크로드에서는 RCU 가 겪을 수 있는 캐시 온도 문제를 예측하지 못했는데, 이는 부분적으로 RCU 가 그런 워크로드에서 정말로 사용될 거라고는 생각되지 않았기 때문입니다. 때문에, sequence locking 이 그렇듯 이런 캐시 온도 문제가 심각할 수 있는 경우에 SLAB_TYPESAFE_BY_RCU 가 사용되게 되었습니다. 다른 한편, 이 구절은 또한 RCU 가 스케줄링 응답시간 절감이나 보안을 위해 서도 사용될 것을 예측하지 못했습니다.

이 책을 위해 생성된 많은 데이터가 8개 소켓, 소켓 당 28개 코어, 코어당 두개 하드웨어 쓰레드를 장착해 총 448개 하드웨어 쓰레드를 갖는 시스템에서 수집되었습니다. 유휴 시스템 메모리 응답시간은 1 마이크로 세컨드 미만인데, 이는 비슷한 크기의 2004년도 시스템 대비 더 나쁘지 않습니다. 어떤 사람들은 이 응답시간이 x86 CPU 제품군의 상대적으로 강한 메모리 순서 규칙 때문에 마이크로세컨드에 가깝다고 주장합니다만, 그 특정 주장이 받아들여지기 전까진 시간이 좀 더 필요할 겁니다.

17.1.5 Astounding Accelerators

하드웨어 가속기의 잠재성은 2021년 만큼이나 2004년에도 분명치 않아 보였고, 그래서 이 섹션은 인용문이 없습니다. 그러나, 2020년 11월의 Top 500 목록 [MDSS20]은 매우 많은 가속기를 담고 있어서, 어떤 사람은 이 섹션이 미래가 아니라 현재에 대한 시각이라고 주장할 수도 있겠습니다. 같은 주장이 앞의 섹션들 대부분에도 통합됩니다.

하드웨어 가속기는 암호화, 압축, 기계 학습을 포함한 많은 경우에 사용되고 있습니다.

요약하자면, 이 챕터의 나머지 부분을 포함한 예언들에 조심하세요.

17.2 Transactional Memory

Everything should be as simple as it can be, but not simpler.

Albert Einstein, by way of Louis Zukofsky

데이터베이스 외 영역에서 트랜잭션을 사용하려는 아이디어는 수십년 전부터 시작되었는데 [Lom77, Kni86, HM93], 데이터베이스와 비 데이터베이스 트랜잭션 사이의 중요 차이점은 비 데이터베이스 트랜잭션은 데이터베이스 트랜잭션을 정의하는 속성인 “ACID”¹ 중 “D”를 제거한다는 겁니다. 메모리 기반 트랜잭션, 또는 “transactional memory (트랜잭션 메모리)” (TM)을 지원한다는 아이디어는 하드웨어에서는 비교적 최근의 일이나 [HM93], 불행히도 일반 상용 하드웨어에서 그런 트랜잭션을 지원하는건 비슷한 다른 제안들은 진행되었음에도 [SSHT93] 곧바로 진행되지 않았습니다. 그리 오래지 않아, Shavit과 Touitou는 일반 상용 하드웨어에서 돌릴 수 있으며 메모리 순서 문제를 주거나 받을 수 있는 [ST95] 트랜잭션 메모리의 소프트웨어 구현 (STM)을 제안했습니다. 이 제안은 여러 해동안 차차 시들어갔는데, 아마도 연구자 커뮤니티의 관심은 non-blocking 동기화에 흡수되었기 때문일 겁니다 (Section 14.2를 참고하세요).

하지만 세기가 바뀌며, TM은 더 많은 관심을 받기 시작했으며 [MT01, RG01], 그 십년의 중반에 이르러서는 관심의 정도가 “작열했다” [Her05, Gro07] 고 말할 수 있으며, 약간의 경고 목소리만 존재했습니다 [BLM05, MMW07].

TM의 기본 아이디어는 코드의 일부를 원자적으로 수행하여 다른 쓰레드는 중간 상태를 볼 수 없게 하자는 겁니다. 보통 그렇듯, TM의 의미 규칙은 상당한 성능과

¹ Atomicity (원자성), consistency (일관성), isolation (격리), 그리고 durability (지속성).

확장성 하락을 초래할 수 있지만 각 트랜잭션을 회귀적으로 획득될 수 있는 전역 락 획득과 해제로 대체될 수 있습니다. 하드웨어는 소프트웨어는 TM 구현에서 피할 수 없는 복잡도의 대부분은 동시에 트랜잭션이 안전하게 병렬로 수행될 수 있을 때 효율적으로 파악될 수 있습니다. 이 파악은 동적으로 이루어지므로, 충돌하는 트랜잭션은 취소되거나 “롤백 (rolled back)” 될 수 있으며, 일부 구현에서는 이 실패 모드가 프로그래머에게 보여집니다.

트랜잭션 룰백은 트랜잭션 크기가 작아질수록 더욱 이뤄지기 어려우므로, 스택, 큐, 해쉬 테이블, 그리고 탐색 트리에서 사용되는 링크드 리스트 조정 같은 작은 메모리 기반 오퍼레이션에 상당히 매력적일 겁니다. 그러나, 특히 I/O와 프로세스 생성과 같은 비 메모리 오퍼레이션을 포함하는 큰 트랜잭션을 위한 경우는 현재로썬 훨씬 어렵습니다. 다음 섹션들은 “어디에나 트랜잭션 메모리를” [McK09b] 이라는 큰 비전에의 현재의 문제들을 알아봅니다. Section 17.2.1은 외부의 세계와 상호작용하는 과정에서 부딪히는 문제들을 알아보고, Section 17.2.2는 프로세스 수정 기능들과의 상호작용을 알아보며, Section 17.2.3은 다른 동기화 기능들과의 상호작용을 알아보고, 마지막으로 Section 17.2.4는 일부 토론과 함께 장을 닫습니다.

17.2.1 Outside World

Donald Knuth의 혁명한 말을 빌리자면:

많은 컴퓨터 사용자들이 입출력이 “진짜 프로그래밍”의 실제 부분이 아니라 기계 안팎에서 정보를 얻기 위해 (불행히도) 이루어져야만 하는 것일 뿐이라고 느낍니다.

우리가 이 입출력을 “진짜 프로그래밍”이라고 믿는 만큼, 사실은 대부분의 컴퓨터 시스템에서 외부와의 상호작용은 첫번째 요구사항이라는 겁니다. 따라서 이 섹션은 트랜잭션 메모리의 I/O 오퍼레이션, 시간 지연, 또는 영구 저장장치를 통해서 이루어질 수 있는 그런 상호작용 능력을 비평해 봅니다.

17.2.1.1 I/O Operations

우린 I/O 오퍼레이션을 필요하다면 락 기반 크리티컬 섹션 내에서 해저드 포인터를 잡은 채, 시퀀스 락킹 read-side 크리티컬 섹션에서, 또는 userspace-RCU read-side 크리티컬 섹션에서, 그리고 심지어는 이 모든 것을 한꺼번에 한 채로 수행할 수 있습니다. 여러분이 I/O 오퍼레이션을 트랜잭션 형태로 수행하려 하면 어떤 일이 벌어질까요?

여기서의 문제는 트랜잭션은 예를 들면 충돌 때문에 룰백될 수 있다는 겁니다. 대략적으로 말하자면, 이는 해당 트랜잭션 내의 모든 오퍼레이션이 취소 가능해서 이 오퍼레이션을 두번 수행하는 것이 한번 수행되었을 때와 같은 효과를 내야 할 것을 필요로 합니다. 불행히도, I/O 는 일반적으로 취소 불가한 오퍼레이션의 원형이어서, 트랜잭션 내에 일반 I/O 오퍼레이션을 포함하기 어렵게 합니다. 실제로, 일반 I/O 는 취소 불가합니다: 일단 핵전쟁을 시작하는 버튼을 누르고 나면, 되돌릴 수 없습니다.

여기 트랜잭션 내에서 I/O 를 처리하기 위한 일부 선택사항이 있습니다:

1. 트랜잭션 내에서의 I/O 를 메모리 내 버퍼와 함께 이뤄지는 버퍼 기반 I/O 로 제한합니다. 그럼 이 버퍼는 모든 다른 메모리 위치가 포함될 수 있는 것과 같은 방법으로 트랜잭션에 포함될 수 있을 겁니다. 이는 선택되는 메커니즘이 될 것으로 보이며, stream I/O 와 대용량 저장장치 I/O 같은 많은 혼한 상황에서 잘 동작합니다. 그러나, 여러 기록 기반 출력 스트림이 여러 프로세스로부터의 단일 파일에 머지되는, fopen() 에의 “a+” 옵션이나 open() 에의 O_APPEND 플래그를 통해 이루어질 수 있는 경우에는 특수한 처리가 필요합니다. 또한, 다음 섹션에서도 알아보겠지만 일반적인 네트워킹 오퍼레이션은 버퍼링을 통해 처리될 수 없습니다.
2. 트랜잭션 내에서의 I/O 를 금지해서 I/O 오퍼레이션을 수행하려는 모든 시도가 그를 감싼 트랜잭션을 (그리고 중첩된 여러 트랜잭션을) 취소시키게 합니다. 이 시도는 버퍼링 불가한 I/O 를 위한 일반적인 TM 방법으로 보이나, TM 이 I/O 를 제어하기 위한 다른 동기화 기능과 함께 사용되어야 하게 합니다.
3. 트랜잭션 내에서의 I/O 를 금지하나, 이 금지를 강제하기 위해 컴파일러의 도움을 받습니다.
4. 한번에 단 하나만 수행될 수 있는 특수한 취소 불가한 트랜잭션 [SMS08] 을 허용해서 취소 불가한 트랜잭션이 I/O 오퍼레이션을 포함할 수 있게 합니다.² 이는 일반적으로 동작합니다만, I/O 오퍼레이션의 확장성과 성능을 상당하게 제한합니다. 확장성과 성능이 병렬성의 첫번째 목적인 만큼, 이 방법은 일반적으로 스스로를 제한하는 것으로 보여지기도 합니다. 더 나쁜 것이, I/O 오퍼레이션을 제어하기 위한 취소 불가성의 사용은 직접적인 트랜잭션 취소 오퍼레이션의 사용을 크게 제한하는

² 이전의 출판물에서, 취소 불가한 트랜잭션은 피할 수 없는 트랜잭션이라 불리었습니다.

듯 합니다.³ 마지막으로, 특정 데이터 항목을 조정하는 취소 불가한 트랜잭션이 있다면 같은 데이터 아이템을 조정하는 모든 다른 트랜잭션은 non-blocking 될 수 없습니다.

5. I/O 오퍼레이션이 트랜잭션 가능한 하위 영역에 들어갈 수 있는 새로운 하드웨어와 프로토콜을 만듭니다. 입력 오퍼레이션의 경우, 하드웨어는 그 오퍼레이션의 결과를 올바르게 예측할 수 있고 그 예측이 틀리면 그 트랜잭션을 취소할 수 있어야 합니다.

I/O 오퍼레이션은 TM 의 널리 알려진 약점이며, 트랜잭션에서 I/O 를 지원하는데에서의 문제가 합리적이고 일반적인 해결책을 가지고 있는지는 최소한 “합리적인” 이 쓸만한 성능과 확장성을 포함한다면 불명확 합니다. 그러나, 이 문제에 대한 지속적인 관심과 시간은 추가적인 진보를 만들어 낼 겁니다.

17.2.1.2 RPC Operations

우린 RPC 들을 락 기반 크리티컬 섹션에서, 해저드 포인터를 쥔 상태에서, 시퀀스 락킹 read-side 크리티컬 섹션에서, userspace-RCU read-side 크리티컬 섹션에서, 그리고 심지어 필요하다면 이 모든 것과 함께 할 실행할 수도 있습니다. 트랜잭션 내에서 RPC 를 실행하려 하면 어떻게 될까요?

RPC 요청과 응답이 모두 이 트랜잭션 내에 포함된다면, 그리고 트랜잭션의 어떤 부분이 응답으로 반환되는 결과에 의존적이라면, buffered I/O 의 경우에 사용될 수 있는 메모리 버퍼 트릭은 사용될 수 없습니다. 이 버퍼링 방법을 사용하려는 모든 시도는 아래의 예에서처럼 그 요청은 이 트랜잭션이 성공될 거라고 보장되기 전까지는 전송될 수 없지만 이 트랜잭션의 성공 여부는 그 응답이 도착할 때까지 알 수 없으므로, 이 트랜잭션을 데드락에 빠뜨릴 겁니다.

```

1 begin_trans();
2 rpc_request();
3 i = rpc_response();
4 a[i]++;
5 end_trans();

```

이 트랜잭션의 메모리 사용량은 이 RPC 응답을 받기 전까지는 파악될 수 없으며, 이 트랜잭션의 메모리 사용량이 파악되기 전까지는 이 트랜잭션이 커밋될 수 있는지 파악될 수 없습니다. 따라서 트랜잭션의 규칙에서 일관적인 단 하나의 행동은 무조건적으로 트랜잭션을 취소하는 것으로, 달리 말하면 최소한 도움이 되지 않습니다.

³ 이 어려움은 Michael Factor 에 의해 지적되었습니다. 이 문제를 이해하기 위해, 어떤 트랜잭션이 취소 불가한 오퍼레이션을 수행한 후 취소되려 할 때 TM 은 어떻게 반응해야 할지 생각해 봅시다.

여기 TM에서 가능할 법한 옵션들이 있습니다:

1. 트랜잭션 내에서의 RPC를 금지해서 RPC 오퍼레이션을 수행하려는 모든 시도는 그를 감싼 트랜잭션을 취소시키게 합니다(그리고 아마도 여러개의 중첩된 트랜잭션도). 대안적으로, 컴파일러가 RPC가 없는 트랜잭션만을 강제하게 하는 도움을 받을 수도 있습니다. 이 방법은 동작하나, TM이 다른 동기화 기능들과 상호작용하게 해야 할 겁니다.
2. 한번에 단 하나의 취소 불가한 특수한 트랜잭션 [SMS08] 만을 허용해서 취소 불가한 트랜잭션들이 RPC 오퍼레이션을 포함할 수 있게 합니다. 이는 일반적으로 동작하지만, 심각하게 RPC 오퍼레이션의 확장성과 성능을 제한합니다. 확장성과 성능이 병렬성의 첫번째 목표임을 감안하면, 이 방법의 범용성은 스스로를 제한하는 듯 보입니다. 더 나아가, RPC 오퍼레이션을 허용하기 위해 취소 불가한 트랜잭션을 사용하는 것은 일단 RPC 오퍼레이션이 시작되면 직접적인 트랜잭션 취소 오퍼레이션이 제한됩니다. 마지막으로, 어떤 데이터 항목을 조정하는 취소 불가한 트랜잭션이 존재한다면, 같은 데이터 항목을 조정하는 모든 다른 트랜잭션은 블록되어야 합니다.
3. 트랜잭션의 성공이 RPC 응답이 받아지기 전에 정해질 수 있는 특수한 경우를 정의하고 자동으로 이것들을 RPC 요청을 보내기 전에 취소 불가한 트랜잭션으로 변환시킵니다. 물론, 동시의 여러 트랜잭션이 이 방법으로 RPC 호출을 시도한다면 하나만 남기고 모든 트랜잭션을 롤백해야 할텐데 이는 성능과 확장성의 저하를 초래할 겁니다. 그러나 이 방법은 RPC와 함께 종료되는 오랜 시간 수행되는 트랜잭션에는 가치 있을 수도 있습니다. 이 방법은 여전히 직접적인 트랜잭션 취소 오퍼레이션을 제약할 수 있습니다.
4. RPC 응답이 트랜잭션 밖으로 빠질 수 있는 특수한 경우들을 정의하고 그 경우엔 buffered I/O와 비슷한 방법을 사용해 진행합니다.
5. 트랜잭션의 범위를 RPC 서버에 이어 클라이언트 까지 포함하도록 확장합니다. 이는 이론상 가능하며 분산 데이터베이스를 통해 시범되었습니다. 그러나, 요청되는 성능과 확장성 요구사항이 분산 데이터베이스 기법으로 이루어질 수 있는지는 불명확한데, 메모리 기반 TM은 그런 응답시간을 가릴 수 있는 뒷단의 느린 디스크 드라이브를 갖지 않기 때문입니다. 물론, solid-state 디스크의 발전을 놓고 보면 데이터베이스가 응답시간 감추기 기법을 재설계 해야 할 것으로 보이긴 합니다.

앞의 섹션에서 이야기 되었듯, I/O는 TM의 알려진 약점이며, RPC는 I/O의 특별히 문제가 되는 경우일 뿐입니다.

17.2.1.3 Time Delays

트랜잭션 외 액세스와의 상호 작용의 중요한 특수 경우는 트랜잭션 내에서의 시간 지연을 포함합니다. 물론, 트랜잭션 내에서의 시간 지연이라는 아이디어는 TM의 원자성에 적용되지만 이런 종류의 것은 논쟁의 소지가 있지만 약한 원자성이란 무엇인가에 대한 것이 됩니다. 더 나아가, memory-mapped I/O와의 올바른 상호작용은 때때로 주의깊게 제어된 타이밍을 필요로 하며, 어플리케이션은 종종 여러 목적으로 시간 지연을 사용합니다. 마지막으로, 어떤 사람들은 시간 지연을 락 기반의 크리티컬 섹션에서, 해저드 포인터를 전 채, 시퀀스 락킹 read-side 크리티컬 섹션 내에서, 그리고 심지어는 필요하다면 이 모든 것들과 함께 수행할 수도 있습니다. 그렇게 하는건 경쟁이나 확장성 시점에서 보면 현명하지 않을 테지만, 다시 말하지만 그렇게 하는건 근본적 컨셉상의 문제를 일으키지는 않습니다.

그러니, TM은 트랜잭션 내에서의 시간 지연을 위해 뭘 할 수 있을까요?

1. 트랜잭션 내에서의 시간 지연을 무시합니다. 이는 우아해 보일 수 있지만 다른 너무 많은 “우아한” 해결책들처럼, 기존 코드와의 접촉에서 살아남지 못합니다. 크리티컬 섹션 내에 중요한 시간 지연을 가질 그런 코드는 트랜잭션화 되는데 실패할 겁니다.
2. 시간 지연 오퍼레이션을 만나는 순간 트랜잭션들을 취소시킵니다. 이는 매력적이지만 불행히도 시간 지연 오퍼레이션을 항상 자동으로 탐지할 수는 없습니다. 그 반복문은 중요한 계산을 하는 걸까요, 아니면 시간이 지나가길 기다릴 뿐일까요?
3. 컴파일러가 트랜잭션 내에서의 시간 지연을 금지하게 부탁합니다.
4. 시간 지연이 평범히 수행되게 합니다. 불행히도, 일부 TM 구현은 수정사항을 커밋 시점에만 외부에 노출하는데, 이는 시간 지연의 목적을 의미없게 할 수 있습니다.

하나의 올바른 답이 있는지는 명확치 않습니다. 약화된 원자성을 가져서 변경사항을 트랜잭션 내에서 즉각적으로 외부에 노출시키는(취소 시에는 이 변경들을 롤백시킵니다) TM 구현은 마지막 대안으로 잘 처리될 수도 있을 겁니다. 이 경우에조차, 이 트랜잭션의 다른쪽

끝의 코드 (또는 심지어 하드웨어) 는 취소된 트랜잭션을 처리하기 위해 상당한 재설계가 필요할 겁니다. 이 재설계 필요는 트랜잭션 메모리를 기준 코드에 적용하기 더 어렵게 할 겁니다.

17.2.1.4 Persistence

많은 다른 종류의 락킹 기능들이 있습니다. 한가지 흥미로운 차이점은 지속성으로, 달리 말하면 어떤 락이 그 락을 사용하는 프로세스의 주소 공간에 무관하게 존재할 수 있는가입니다.

비 지속적 락은 `pthread_mutex_lock()`, `pthread_rwlock_rdlock()`, 그리고 대부분의 커널 단계 락킹 기능들을 포함합니다. 비 지속적 락의 데이터 구조체를 만들어내는 메모리 위치가 사라진다면 락도 사라집니다. 일반적인 `pthread_mutex_lock()` 사용에서 이는 프로세스가 종료될 때 그것의 모든 락도 사라짐을 의미합니다. 이 속성은 프로그램 종료 시점의 락 정리를 간단하게 하기 위해 사용될 수 있으나, 관계 없는 어플리케이션들이 락을 공유하기는 그 어플리케이션들이 메모리를 공유하게 해야 하므로 어렵게 만들니다.

Quick Quiz 17.1: 하지만 어떤 어플리케이션이 파일로 매핑된 메모리 지역에 있는 `pthread_mutex_lock()` 을 전 채로 종료된다면요?

■
지속적 락은 관계 없는 어플리케이션 간에 메모리를 공유할 필요를 제거하는데 도움이 됩니다. 지속적 락킹 API 는 `flock` 부류들, `lockf()`, System V 세마포어, `open()` 에의 `O_CREAT` 플래그 등을 포함합니다. 이 지속적 API 들은 여러 어플리케이션을 수행하는 거대 규모 오퍼레이션들을 보호하는데 사용될 수 있으며, `O_CREAT` 의 경우는 심지어 운영체제 리부팅 이후에도 살아남습니다. 필요하다면, 락은 심지어 분산 락 관리자와 분산 파일시스템을 통해 여러 컴퓨터 시스템을 아우를 수도 있습니다—그리고 이 컴퓨터 시스템들 모든 것의 리부팅에도 지속될 수 있습니다.

지속적 락은 어떤 어플리케이션에 의해서도 사용될 수 있는데, 여러 언어와 소프트웨어 환경으로 쓰여진 어프리케이션들도 포함됩니다. 실제로, 어떤 지속적 락은 C 로 쓰여진 어플리케이션에 의해 획득되고 Python 으로 쓰인 어플리케이션에 의해 해제될 수 있습니다.

비슷한 지속적 기능이 TM 에는 어떻게 제공될 수 있을까요?

1. 지속적 트랜잭션을 예를 들면 SQL 과 같은, 그것을 지원하기 위해 설계된 특수 목적 환경에 제약 합니다. 이는 수십년의 데이터베이스 시스템의 긴 역사를 놓고 볼 때 분명 동작하지만, 지속적 락이 제공하는 것 만큼의 유연성을 제공하진 않습니다.

2. 일부 저장 장치나 파일시스템을 통해 제공되는 스냅샷 기능을 사용합니다. 불행히도, 이는 네트워크 통신을 처리하지 않으며, 예를 들면 메모리 스톱 같은 스냅샷 기능을 제공하지 않는 기기로의 I/O 역시 처리하지 않습니다.

3. 타임머신을 만듭니다.

4. 현존하는 지속적 기능을 사용해 트랜잭션 내에서의 그런 사용을 막음으로써 문제를 완전히 막습니다.

물론, 이게 트랜잭션 메모리라고 불린다는 사실은 우릴 잠깐 정지시킬 텐데, 그 이름 자체가 지속적 트랜잭션의 컨셉과 충돌하기 때문입니다. 그러나 이 가능성을 트랜잭션 메모리의 고유의 한계점을 보이는 중요한 테스트 케이스로 생각할 가치는 있습니다.

17.2.2 Process Modification

프로세스는 영원하지 않습니다: 그것들은 생성되고 파괴되며, 그것들의 메모리 매핑은 수정되고, 동적 라이브러리에 연결되며, 디버깅 됩니다. 이 세션들은 트랜잭션 메모리가 어떻게 계속 변화하는 수행 환경을 처리하는지 알아봅니다.

17.2.2.1 Multithreaded Transactions

락을 전채, 또는 필요하다면 해저드 포인터를 전 채, 시퀀스 락킹 read-side 크리티컬 섹션 내에서, userspace-RCU read-side 크리티컬 섹션 내에서, 그리고 필요하다면 이 모든 것을 한채로 프로세스와 쓰레드를 생성하는 게 완벽하게 합법적입니다. 이는 합법일 뿐 아니라 매우 간단한데, 다음 코드 조각으로 보이는 바와 같습니다:

```

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 슈도코드 조각은 CPU 당 하나의 쓰레드를 생성하기 위해 `pthread_create()` 를 사용하고 이어서 각 쓰레드가 완료되기를 기다리기 위해 `pthread_join()` 을 사용하며 이 모든 행동은 `pthread_mutex_lock()` 으로 보호됩니다. 그 효과는 락 기반의 크리티컬 섹션을 병렬로 수행하는 것이며, 비슷한 효과를 `fork()` 와 `wait()` 로 얻을 수도 있습니다. 물론, 이 크리티컬 섹션은 쓰레드 생성 오버헤드를 정당화 할 수 있을 만큼 커야 할테지만, 제품 소프트웨어에는 커다란 크리티컬 섹션의 예가 많습니다.

TM은 트랜잭션 내에서 쓰레드를 생성하는 것에 대해 어떻게 할까요?

1. 트랜잭션 내에서의 `pthread_create()` 를 불법으로 규정하여 그 경우 트랜잭션을 취소시킵니다. 대안적으로, 컴파일러가 `pthread_create()` 가 없는 트랜잭션만을 강제하게 도움을 구합니다.
2. `pthread_create()` 가 트랜잭션 내에서 수행될 수 있게 하지만 그 부모 쓰레드만이 트랜잭션의 부분으로 여겨지게 합니다. 이 방법은 존재하는 그리고 가정되는 TM 구현들과 합리적으로 호환된다고 보이겠지만 그건 부주의한 사람을 위한 속임수인 듯 합니다. 이 방법은 더 많은 질문을 던지게 하는데, 충돌하는 자식 쓰레드 액세스를 어떻게 처리할 것인지 등입니다.
3. `pthread_create()` 들을 함수 호출로 변환합니다. 이 방법 또한 매력적인데, 자식 쓰레드들이 서로 통신하는 희귀하지 않은 경우들을 처리하지 않기 때문입니다. 또한, 트랜잭션의 몸체의 동시 수행을 허용하지 않습니다.
4. 부모와 모든 자식 쓰레드를 감싸게끔 트랜잭션을 확장합니다. 이 방법은 충돌하는 액세스들의 본성에 대한 흥미로운 질문을 자아내는데, 부모와 자식이 서로 충돌하는게 허용되어 있지만 다른 쓰레드와는 그렇지 않기 때문입니다. 이는 또한 부모 쓰레드가 트랜잭션을 커밋하기 전에 자식들을 기다리지 않으면 무슨 일이 벌어질 것인가 같은 흥미로운 질문들을 자아냅니다. 더 흥미로운건, 부모가 트랜잭션 내에서 참여하는 변수들의 값에 기반해 조건적으로 `pthread_join()` 을 수행한다면 무슨 일이 벌어질까요? 이 질문들에 대한 답은 락킹의 경우 합리적으로 단순합니다. TM에서의 답은 독자 여러분의 뜻으로 남겨둡니다.

트랜잭션의 병렬 수행은 데이터베이스 세계에서 흔하다는 걸 놓고 보면, 현재의 TM 제안들이 그걸 제공하지 않는다는 건 놀라울 수 있습니다. 다른 한편, 앞의 예는 간단한 교재의 예에서는 일반적이지 않은 상당히 발전된 락킹의 사용들이므로, 아마도 그 누락은 예상되었을 수 있습니다. 그렇다고 하나, 어떤 연구자들은 코드를 자동으로 병렬화 하기 위해 트랜잭션을 사용하며 [RKM⁺10], 다른 TM 연구자들은 트랜잭션 내에서의 `fork/join` 병렬성을 조사하고 있다는 소식이 있으므로, 이 주제는 더 자세히 다뤄질 수도 있습니다.

17.2.2.2 The `exec()` System Call

우린 락 기반 크리티컬 섹션에서, 해저드 포인터를 친 채, 시퀀스 락킹 `read-side` 크리티컬 섹션 내에서, userspace-RCU `read-side` 크리티컬 섹션 내에서, 심지어 필요하다면 이 모든 것과 함께 `exec()` 시스템 콜을 실행할 수 있습니다. 그 정확한 의미는 기능의 종류에 의존합니다.

지속적이지 않은 기능 (`pthread_mutex_lock()`, `pthread_rwlock_rdlock()`, 그리고 userspace RCU가 포함됩니다)의 경우, `exec()` 이 성공한다면, 어떤 락도 쥐어지지 않은채 전체 주소 공간이 사라집니다. 물론, `exec()` 이 실패한다면 그 주소공간은 여전히 살아있으므로 연관된 락들 또한 여전히 살아 있습니다. 약간 이상할 수 있지만, 잘 정의되어 있습니다.

다른 한편, 지속적 기능 (`flock` 무리, `lockf()`, System V 세마포어, 그리고 `open()` 의 `O_CREAT` 플래그 등이 포함됩니다)의 경우 `exec()` 의 성공과 실패에 무관하게 생존할 것이므로 `exec()` 된 프로그램은 그걸 해제할 수도 있을 겁니다.

Quick Quiz 17.2: `mmap()` 메모리 영역의 데이터 구조로 표현되는 비지속적 기능은 어떻습니까? 크리티컬 섹션이나 그런 기능 내에서의 `exec()` 이 존재하면 어떻게 되죠?



트랜잭션 내에서 `exec()` 시스템 콜을 수행하면 어떻게 될까요?

1. 트랜잭션 내에서의 `exec()` 을 불허하여 이를 감싸는 트랜잭션이 `exec()` 을 만나는 순간 취소되게 합니다. 이는 괜찮은 정의지만, `exec()` 을 위한 비 TM 동기화 기능의 사용을 분명 필요로 합니다.
2. 트랜잭션 내에서의 `exec()` 을 불허하며 컴파일러가 이를 강제하게 합니다. 이 방법을 취하는 C++에서의 TM 명세 작업이 있는데, 여기선 함수들을 `transaction_safe` 와 `transaction_unsafe` 속성으로 장식될 수 있게 합니다.⁴ 이 방법은 수행시간에 트랜잭션을 취소하는 것보다 일부 장점을 갖지만 역시 `exec()` 을 위한 비 TM 동기화 기능의 사용을 필요로 합니다. 한가지 단점은 수많은 라이브러리 함수를 `transaction_safe` 와 `transaction_unsafe` 속성으로 장식해야 한다는 겁니다.
3. 트랜잭션을 비 지속적 락킹 기능과 비슷하게 취급하여 `exec()` 이 실패하면 트랜잭션이 생존하게 하고, `exec()` 이 성공하면 조용히 커밋합니다. 이 트랜잭션에 의해 `mmap()` 된 메모리 내의 변수들만이 영향을 받게 되는 (그리고 따라서 성공적 `exec()` 시스템 콜에도 살아남는) 경우는 독자 여러분의 연습 문제로 남겨둡니다.
4. `exec()` 시스템 콜이 성공할 것이라면 트랜잭션을 (그리고 `exec()` 시스템 콜을) 취소시키지만 `exec()` 시스템 콜이 실패할 것이라면 트랜잭션을 지속되게 합니다. 이는 어떤 점에선 “올바른”

⁴ 제게 이 명세를 알려준 Mark Moir 에게, 또한 그로부터 전에 더 이전의 버전을 알려준 Michael Wong 에게 감사를 전합니다.

방법이지만 만족스럽지 않은 결과를 위한 상당한 작업을 필요로 할 겁니다.

`exec()` 시스템 콜은 아마도 완벽한 TM 사용에 대한 장애물의 가장 기묘한 예일 수 있겠는데, 어떤 방법이 말이 되는지 명확하지 않으며, 어떤 사람들은 이게 `exec()` 과 함께 벌어지는 실제 세상에서의 위험의 단순한 반영이라고 주장할 수도 있습니다. 그렇다고는 하나, 트랜잭션 내에서의 `exec()` 을 금지하는 두 가지 옵션은 아마도 가장 논리적입니다.

비슷한 문제가 `exit()` 과 `kill()` 시스템 콜, 그리고 `longjmp()` 또는 트랜잭션을 빠져나가야 하는 예외에 존재합니다. (`longjmp()` 나 예외는 어디서 나왔을까요?)

17.2.2.3 Dynamic Linking and Loading

락 기반 크리티컬 섹션, 해저드 포인터를 쥐는 코드, 시퀀스 락킹 read-side 크리티컬 섹션, 그리고 userspace-RCU read-side 크리티컬 섹션은 C/C++ 공유 라이브러리와 Java 클래스 라이브러리를 포함한 동적으로 링크되고 로드되는 함수들을 호출하는 코드를 포함하는게 합법적입니다. 물론, 이 라이브러리에 있는 코드는 정의상으로 컴파일 시점엔 알려지지 않았습니다. 따라서, 동적으로 로드된 함수가 트랜잭션 내에서 호출되면 무슨 일이 벌어질까요?

이 질문은 두 부분을 갖습니다: (a) 트랜잭션 내에서 어떻게 동적으로 함수를 링크하고 로드하는가 그리고 (b) 이 함수 내의 알려지지 않은 본성의 코드에 대해 뭘 할 것인가? 공정을 위해 말하자면, 항목 (b)는 최소한 이론상으로는 락킹과 userspace-RCU 에도 일부 문제를 일으킬 수 있습니다. 예를 들어, 이 동적으로 링크된 함수는 락킹에 대한 데드락을 일으키거나(문제가 되게도) userspace-RCU read-side 크리티컬 섹션 내로 quiescent state 를 추가시킬 수도 있습니다. 차이점이라면 락킹과 userspace-RCU 크리티컬 섹션 내에서 허용된 오퍼레이션들의 종류는 잘 이해되어 있는 반면, TM의 경우엔 상당한 불확실성이 있는듯 하다는 겁니다. 실제로, TM의 다른 구현들은 다른 제약들을 갖는듯 합니다.

그러니 TM은 동적으로 링크되고 로드되는 라이브러리 함수에 대해 뭘 할까요? 실제로 코드를 로딩하는 것인 항목 (a)를 위한 옵션들에는 다음이 포함됩니다:

1. 동적 링킹과 로딩을 페이 풀트와 비슷한 방식으로 취급해서 그 함수가 로드되고 링크되지만 그 트랜잭션은 취소되게 합니다. 그 트랜잭션이 어보트되었다면, 재시도는 그 함수가 이미 존재함을 찾게 될 것이고 그 트랜잭션은 평범하게 처리될 수 있을 겁니다.

2. 트랜잭션 내에서의 함수의 동적 링킹과 로딩을 금지합니다.

아직 로드되지 않은 함수 내의 TM에 친화적이지 않은 오퍼레이션들을 탐지할 수 없는 경우인 (b)를 위한 선택 사항들은 다음의 것들이 포함됩니다:

1. 그냥 그 코드를 수행합니다: 그 함수 내에 TM에 친화적이지 않은 오퍼레이션이 있다면, 단순히 그 트랜잭션을 취소합니다. 불행히도, 이 방법은 컴파일러가 언제 어떤 트랜잭션들이 안전히 만들어질 수 있는지 파악할 수 없게 합니다. 그에 관계 없이 안전한 만들어질 수 있는 능력을 허용하는 한가지 방법은 취소 불가한 트랜잭션입니다만, 현재의 구현들은 한번에 단 하나의 취소 불가 트랜잭션만 진행될 수 있게 하는데, 이는 상당히 성능과 확장성을 제한합니다. 취소 불가한 트랜잭션은 또한 직접적인 트랜잭션 취소 오퍼레이션의 사용을 제한합니다. 마지막으로, 특정 데이터 항목을 조정하는 취소 불가한 트랜잭션이 있다면, 같은 데이터 항목을 조정하는 모든 다른 트랜잭션은 non-blocking 될 수 없습니다.
2. 어떤 함수가 TM 친화적인지 알리는 장식을 함수 선언에 합니다. 이 장식은 컴파일러의 타입 시스템에 의해 강제될 수 있습니다. 물론, 많은 언어에 있어서 이는 언어 확장이 제안되고 표준화 되고 구현될 것을 필요로 하는데 이는 연관된 시간 지연을 내포하며 그렇지 않았다면 관계 없었을 많은 라이브러리 함수들에의 연관된 장식을 필요로 합니다. 그러나, 이 표준화 노력은 이미 진행 중입니다 [ATS09].
3. 앞에서와 같이, 트랜잭션 내에서의 함수 동적 링킹과 로딩을 금지합니다.

I/O 오퍼레이션은 물론 알려진 TM의 약점이며, 동적 링킹과 로딩은 I/O의 또 다른 특수 경우라고 생각될 수 있습니다. 그러나, TM 제안자는 이 문제를 풀거나 TM이 병렬 프로그래머의 도구상자의 여러 도구 중 하나일 뿐인 세계에 체념해야 합니다. (공정을 위해 말하자면, 여러 TM 제안자들이 TM 외에도 여러개를 갖는 세계에 체념했습니다.)

17.2.2.4 Memory-Mapping Operations

락 기반의 크리티컬 섹션, 해저드 포인터를 친체, 시퀀스 락킹 read-side 크리티컬 섹션 내에서, 그리고 userspace-RCU read-side 크리티컬 섹션 내에서, 심지어 필요하다면 이 모든 것을 함께 한 상태에서 메모리 매핑 오퍼레이션을 (`mmap()`, `shmat()`, 그리고 `munmap()`을

포함 [Gro01]) 수행하는게 완벽히 합법입니다. 그런 오퍼레이션을 트랜잭션 내에서 수행하려 하면 어떻게 될까요? 더 자세히는, 다시 매핑되는 메모리 영역이 현재 쓰레드의 트랜잭션에 참여하는 공유 변수를 포함하면 어떻게 될까요? 그리고 이 메모리 영역이 다른 쓰레드의 트랜잭션에 참여하는 변수들을 포함하면 어떨까요?

이 TM 시스템의 메타데이터가 다시 매핑되는 경우는 고려할 필요가 없을텐데, 대부분의 락킹 기능은 그들의 락 변수들을 재 매핑하는 행위의 결과를 정의하지 않기 때문입니다.

여기 TM에서의 메모리 매핑에 대한 옵션들이 일부 있습니다:

1. 트랜잭션 내에서의 메모리 재 매핑은 불법이며 모든 감싼 트랜잭션들이 취소되게 합니다. 이는 일을 어떤식으로 간단하게 만들지만 TM이 그들의 크리티컬 섹션에서의 재 매핑을 처리하기 위한 동기화 기능들과 상호작용할 것을 필요로 합니다.
2. 메모리 재 매핑은 트랜잭션 내에서 불법이며, 컴파일러가 이 금지를 강제하는 것을 돋습니다.
3. 메모리 매핑은 트랜잭션 내에서 합법이나, 매핑되는 영역에 변수를 갖는 모든 다른 트랜잭션들을 취소시킵니다.
4. 메모리 매핑은 트랜잭션 내에서 합법이나, 매핑되는 영역이 현재 트랜잭션의 사용중인 영역과 겹치면 그 매핑 오퍼레이션이 실패합니다.
5. 모든 메모리 매핑 오퍼레이션은 그게 트랜잭션 안에서든 밖에서든 시스템의 모든 트랜잭션의 사용 중인 메모리에 대해 매핑을 하는지 검사하게 합니다. 거기 겹치는 구간이 있다면 그 메모리 매핑 오퍼레이션은 실패합니다.
6. 시스템의 어떤 트랜잭션과라도 메모리 사용 부분이 겹치는 메모리 매핑 오퍼레이션의 효과는 TM 총돌 매니저에 의해 탐지되는데, 이는 이 메모리 매핑 오퍼레이션이 실패해야 할지 아니면 총돌하는 트랜잭션을 취소할지 동적으로 결정합니다.

`munmap()`은 연관된 메모리 영역을 매핑되지 않은 상태로 남겨두어서 추가적인 의미를 가질 수도 있다는 점을 알려 듭니다.⁵

17.2.2.5 Debugging

브레이크포인트 같은 일반적인 디버깅 오퍼레이션은 락 기반의 크리티컬 섹션과 userspace-RCU read-side 크리티컬 섹션 내에서 평범하게 동작합니다. 그러나, 초기의 트랜잭션 메모리 하드웨어 구현에서는 [DLMN09]

⁵ 매핑과 언매핑의 이 차이는 Josh Triplett에 의해 알려졌습니다.

트랜잭션 내에서의 예외가 트랜잭션을 취소시켰는데 이는 브레이크포인트가 모든 감싼 트랜잭션을 취소시킴을 의미합니다.

그러니 어떻게 트랜잭션을 디버깅 할 수 있을까요?

1. 브레이크포인트를 포함하는 트랜잭션을 갖는 소프트웨어 애플리케이션 기술을 사용합니다. 물론, 모든 트랜잭션의 범위에서 언제든 브레이크포인트를 갖는 모든 트랜잭션을 애플리케이션 할 필요가 있을 수 있습니다. 만약 런타임 시스템이 어떤 트랜잭션의 범위 내에 어떤 브레이크포인트가 있는지 없는지 탐지할 수 없다면, 안전을 위해 모든 트랜잭션을 애플리케이션 해야 할 수도 있습니다. 그러나, 이 방법은 상당한 오버헤드를 일으킬 수도 있는데, 이는 결국 찾고자 하는 버그를 숨길 수도 있습니다.
2. 브레이크포인트 예외를 처리할 수 있는 하드웨어 TM 구현만을 사용합니다. 불행히도, 지금 시점에서 (2021년 3월), 모든 그런 구현은 연구용 프로토타입입니다.
3. 다른 간단한 하드웨어 TM 구현들보다 (매우 개략적으로 말해서) 예외를 더 잘 다룰 수 있는 소프트웨어 TM 구현만을 사용합니다.
4. 더 주의 깊게 프로그래밍 해서 트랜잭션 내에 버그를 남겨둘 여지를 없앱니다. 이걸 어떻게 할 수 있는지 파악하자마자 모두가 그 비밀을 알 수 있게 해주세요!

트랜잭션 메모리는 다른 동기화 메커니즘들에 비해 생산성의 개선을 제공할 거라 믿을 이유가 있습니다만, 트랜잭션적 디버깅 기술이 트랜잭션들에 적용될 수 없다면 그 개선은 쉽게 사라질 수 있습니다. 이는 트랜잭션 메모리가 거대한 트랜잭션들에서 초보자들에 의해 사용될 경우 특히 사실일 것으로 보입니다. 대조적으로, 마초 “일류” 프로그래머들은 그런 디버깅 도구 없이도 일을 해낼 수 있을 수도 있는데, 특히 작은 트랜잭션들이라면 그럴 수도 있습니다.

따라서, 트랜잭션 메모리가 그 생산성 약속을 초보자 프로그래머들에게 전달하려면 이 디버깅 문제가 해결되어야 합니다.

17.2.3 Synchronization

언젠가 트랜잭션 메모리가 모두를 위한 모든것이 될 수 있음을 증명하는 날이 온다면 어떤 다른 동기화 메커니즘도 사용할 필요가 없을 겁니다. 그 전까지는, 트랜잭션 메모리가 할 수 없는 것을 할 수 있는데, 또는 특정 상황에선 더 자연스러운 일을 하는 동기화 메커니즘들과 함께 사용되어야 할 겁니다. 다음 섹션들은 이 영역에서의 현재의 도전사항들을 정리합니다.

17.2.3.1 Locking

락을 잡고 있으면서 다른 락을 잡는 건 흔한데, 최소한 데드락을 막기 위한 잘 알려진 소프트웨어 엔지니어링 기법이 사용되는 동안은 잘 동작합니다. RCU read-side 크리티컬 섹션 내에서 락을 잡는 건 드물지 않은데, RCU read-side 기능은 락 기반의 데드락 사이클에 참여할 수 없기 때문에 데드락 걱정을 덜어줍니다. 해저드 포인터를 잡고 있는 동안이나 시퀀스 락 read-side 크리티컬 섹션 내에서 락을 잡는 것도 가능합니다. 하지만 트랜잭션 내에서 락을 잡으려 하면 어떻게 될까요?

이론상, 답은 간단합니다: 그 락을 표현하는 데이터 구조를 트랜잭션의 한 부분으로 수정하며, 모든 것이 완벽하게 동작합니다. 실제로는, TM 시스템의 구현상 세부사항에 따라 여러 분명치 않은 복잡성이 [VGS08] 일어날 수 있습니다. 이 복잡성은 해결될 수 있으나, 트랜잭션 바깥에서 잡히는 락을 위한 오버헤드의 45% 증가와 트랜잭션 내에서 획득되는 락을 위한 오버헤드의 300% 증가의 비용을 초래합니다. 이 오버헤드는 작은 양의 락킹을 갖는 트랜잭션화 가능한 프로그램에서는 수용될 수 있을 수도 있겠으나, 가끔 트랜잭션을 사용하고자 하는 제품 품질의 락 기반 프로그램에서는 종종 수용 불가합니다.

TM을 위한 일부 옵션이 여기 있습니다:

1. 락킹 친화적 TM 구현만 사용합니다. 불행히도, 락킹에 비친화적인 구현은 매력적인 속성을 갖는데, 성공한 트랜잭션을 위한 낮은 오버헤드와 극단적으로 큰 트랜잭션의 수용 가능성이 포함됩니다.
2. 락 기반의 프로그램에 TM을 도입할 때 “작은 단계”에서만 TM을 사용해서 락킹 친화적 TM 구현의 한계를 수용합니다.
3. 락킹 기반 기존 시스템을 완전히 무시하고 모든 것을 트랜잭션으로 재구현합니다. 이 방법은 옹호자가 많으나, 이 시리즈에서 설명된 모든 문제가 해결될 것을 필요로 합니다. 이 문제들을 해결하는데 필요한 시간 동안, 경쟁 동기화 메커니즘들은 개선될 기회를 가질 겁니다.
4. TxLinux [RHP+07] 그룹과 수많은 트랜잭션 기반 락 제거 프로젝트들에서 [PD11, Kle14, FIMR16, PMDY20] 행해진 것처럼 TM을 락 기반 시스템의 최적화로만 사용합니다. 이 방법은 말이 되는 듯 하나, 락킹 설계 제한을 (데드락 회피를 위한 필요 같은) 그대로 둡니다.
5. 락킹 기능에 의해 발생하는 오버헤드를 제거하려 노력합니다.

TM과 락킹을 상호작용하게 하는데 문제가 있을 수 있다는 사실은 많은 사람에게 놀라움으로 다가오는데, 실제 세계의 제품 소프트웨어에서 새로운 메커니즘과 기능을 시도해야 하는 필요성을 강조합니다. 다행히도, 오픈소스의 발전은 그런 소프트웨어의 많은 수가 연구자들을 포함한 모두가 자유로이 사용 가능함을 의미합니다.

17.2.3.2 Reader-Writer Locking

락을 잡고 있으면서 reader-writer 락을 읽기 모드로 잡는 건 흔한데, 최소한 데드락을 막기 위한 잘 알려진 흔한 소프트웨어 엔지니어링 기법들이 사용되는 동안은 잘 동작합니다. RCU read-side 크리티컬 섹션 내에서 reader-writer 락을 읽기 모드로 잡는 것 또한 잘 동작하며, RCU read-side 기능들은 락 기반 데드락 사이클에 참여할 수 없기 때문에 데드락에 대한 걱정을 덜어줍니다. 해저드 포인터를 잡고 있는 동안 또는 시퀀스 락 read-side 크리티컬 섹션 내에서 락을 잡는 것 또한 가능합니다. 하지만 여러분이 트랜잭션 내에서 reader-writer 락을 읽기 모드로 획득하려 하면 무슨 일이 벌어질까요?

불행히도, 트랜잭션 내에서 전통적인 가운데 기반 reader-writer 락을 읽기 모드로 잡으려는 시도는 reader-writer 락의 목적을 의미없게 합니다. 이를 보기 위해, 같은 reader-writer 락을 읽기 모드로 획득하려는 동시에 트랜잭션 한 쌍을 생각해 보세요. 읽기 모드 획득은 이 reader-writer 락의 데이터 구조의 수정을 필요로 하므로, 충돌이 발생하고, 이는 두 트랜잭션 중 하나의 롤백을 일으킵니다. 이 동작은 동시에 읽기 쓰레드들을 허용하려는 reader-writer 락의 목표와 완전히 불일치합니다.

TM을 위한 일부 옵션이 여기 있습니다:

1. 특정 CPU(또는 쓰레드)가 락을 읽기 모드 획득할 때 지역 데이터만을 수정하는 CPU 별, 또는 쓰레드 별 reader-writer 락킹 [HW92]을 사용합니다. 이는 두 트랜잭션이 동시에 락을 읽기 모드로 획득하는 것을 충돌을 허용하며, 의도될 대로 둘 모두 진행되는 걸 허용합니다. 불행히도, (1) CPU/쓰레드 별 락킹에서의 쓰기 모드 락 획득 오버헤드는 상당히 높으며, (2) CPU/스레드 별 락킹의 메모리 오버헤드는 위험할 수 있으며, (3) 이 변형은 여러분이 문제의 소스코드에의 접근이 허용될 때에만 가능합니다. 다른 더 최근의 확장 가능한 reader-writer 락 [LLO09]은 이 문제들의 일부 또는 전부를 해결할 수도 있습니다.
2. 락 기반의 프로그램에 TM을 도입할 때 “작은 단계”에서만 TM을 사용해서 트랜잭션 내에서 reader-writer 락을 읽기 모드로 획득할 때의 문제를 회피합니다.

3. 락킹 기반 기존 시스템을 완전히 버리고, 트랜잭션으로 모든 것을 재구현합니다. 이 방법은 많은 지지자가 있으나, 이 시리즈에서 설명된 모든 문제가 해결될 것을 필요로 합니다. 이 문제들을 해결하는 동안 경쟁 동기화 메커니즘들은 물론 개선될 기회를 가질 겁니다.
4. TxLinux [RHP⁰⁷] 그룹과 더 최근의 TM을 이용해 reader-writer 락을 제거한 작업들 [FIMR16]처럼 락 기반 시스템의 최적화에만 TM을 사용합니다. 이 방법은 최소한 POWER8 CPU에서는 말이 되는 듯 하나 [LGW¹⁵], 락킹 설계 제한(데드락 회피의 필요 등)은 그대로 둡니다.

물론, 배타적 락킹에서 그랬듯 TM을 reader-writer 락킹과 조합하는데 따르는 불명확한 문제들이 여럿 있을 수 있습니다.

17.2.3.3 Deferred Reclamation

이 섹션은 RCU에 주 초점을 맞춥니다. TM을 레퍼런스 카운터나 해저드 포인터 같은 다른 deferred-reclamation 메커니즘과 결합할 때에도 비슷한 문제와 가능한 해결법이 나타납니다. 아래의 글에서는 알려진 차이점을 특별히 이야기 합니다.

레퍼런스 카운팅, 해저드 포인터, 그리고 RCU는 Sections 9.5.5 and 9.6.3에서 이야기 되었듯 모두 널리 사용됩니다. 이는 이 섹션에서 이야기된 모든 문제를 극복하지 않기로 한 모든 TM 구현은 이 동기화 메커니즘들과 깨끗하고 효율적으로 상호작용해야 함을 의미합니다.

오스틴의 텍사스 대학교의 TxLinux 그룹은 RCU/TM 상호작용의 문제를 취한 그룹으로 보입니다 [RHP⁰⁷]. 그들은 TM을 Linux 2.6 커널에 적용했는데, 이 커널은 RCU를 사용하므로 그들은 TM이 RCU 업데이트 락킹을 대체하는 형태로 TM과 RCU를 상호작용하게 하는 것밖에 다른 선택지가 없었습니다. 불행히도, 논문은 RCU 구현의 락이 (예: `rcu_ctrlblk.lock`) 트랜잭션으로 변환되었다고 명시했지만, RCU 기반 업데이트에 의해 사용된 락들에(예: `dcache_lock`) 대해선 무슨 일이 일어났는지는 설명하지 않습니다.

더 최근에는, Dimitrios Siakavaras 등이 HTM과 RCU를 탐색 트리에 적용했고 [SNGK17, SBN²⁰], Christina Giannoula 등은 HTM과 RCU를 그래프에 색을 지정하는데 사용했으며 [GGK18], SeongJae Park 등은 HTM과 RCU를 경쟁이 심한 NUMA 시스템에서의 락킹을 최적화하는데 사용했습니다 [PMDY20].

RCU는 읽기 쓰레드와 업데이트 쓰레드를 동시에 수행될 수 있게 허용하며 더 나아가 RCU 읽기 쓰레드가 업데이트 중인 데이터에 접근하는 것도 허용함을 알아두는게 중요합니다. 물론, 성능, 확장성, 그리고 리얼타임 응답시간 이득이 될 수 있는 이 RCU의 속성은

TM의 원자적 속성에도 적용될 수 있겠지만, POWER8 CPU 군의 지연된 트랜잭션 설비 [LGW¹⁵]는 이를 이 규칙에서의 예외로 만듭니다.

그래서 TM 기반 업데이트는 어떻게 동시의 RCU 읽기 쓰레드와 상호작용할 수 있을까요? 일부 가능성은 다음과 같습니다:

1. RCU 읽기 쓰레드는 동시의 충돌하는 TM 업데이트를 취소시킵니다. 이는 TxLinux 프로젝트에 의해 실제로 취해진 방법입니다. 이 방법은 RCU 의미를 지키며, 또한 RCU의 read-side 성능, 확장성, 그리고 리얼타임 응답시간 속성을 지킵니다만 불행히도 충돌하는 업데이트를 불필요하게 취소시키는 부작용을 갖습니다. 최악의 경우, 길게 유지되는 RCU 읽기 쓰레드는 모든 업데이트 쓰레드를 기아에 빠뜨릴 가능성이 있으며, 이는 이론상 시스템 정지를 초래할 수 있습니다. 또한, 모든 TM 구현이 이 방법을 구현하는데 필요한 강력한 원자성을 제공하는건 아닙니다.
2. 충돌하는 TM 업데이트와 동시에 수행되는 RCU 읽기 쓰레드는 기존(트랜잭션 전) 값을 받습니다. 이는 RCU 의미와 성능을 지키고 RCU-update 기아도 방지합니다. 그러나, 모든 TM 구현이 현재 수행중인 트랜잭션에 의해 업데이트 되고 있을지도 모르는 변수에의 기존 값을 즉시 접근할 수 있게 하는건 아닙니다. 특히, 이전 값을 로그에 유지하는(따라서 훌륭한 TM 커밋 성능을 제공하는) 로그 기반 TM 구현은 이 방법에 행복하지 않을 겁니다. 아마도 `rcu_dereference()` 기능은 RCU가 TM 구현의 더 큰 범위에서 기존 값에 RCU가 접근하는 걸 도울 수 있겠지만, 성능이 문제가 될 수도 있을 겁니다. 그렇다고 하나, 이 방법으로 RCU와 상호작용한 유명한 TM 구현도 있습니다 [PW07, HW11, HW13].
3. RCU 읽기 쓰레드가 수행중인 트랜잭션과 충돌하는 액세스를 수행한다면, 그 RCU 액세스는 이 충돌하는 트랜잭션이 커밋하거나 취소된 뒤로 지연됩니다. 이 방법은 RCU 의미를 지키나 RCU의 성능이나 리얼타임 응답시간을 지키지 않는데, 특히 긴 시간 수행되는 트랜잭션의 존재 시 그렇습니다. 또한, 모든 TM 구현이 충돌하는 액세스를 지연시킬 수 있는 건 아닙니다. 그러나, 이 방법은 작은 트랜잭션만 지원하는 하드웨어 TM 구현에서는 상당히 합리적인 듯 합니다.
4. RCU 읽기 쓰레드가 트랜잭션으로 변환됩니다. 이 방법은 RCU가 어떤 TM 구현과도 호환될 것을 상당히 보장합니다만 RCU read-side 크리티컬 섹션의 TM에 의한 취소 가능성은 암시해 RCU의 리얼타임 응답시간 보장을 파괴하며, 또한 RCU의

read-side 성능을 떨어뜨립니다. 더 나아가, 이 방법은 어떤 RCU read-side 크리티컬 섹션이 해당 TM 구현은 처리할 수 없는 오퍼레이션을 포함하는 경우에 적합치 못합니다. 이 방법은 해저드 포인터와 레퍼런스 카운터에 적용하긴 더 어려우데, 코드의 부분에 대한 명확히 정의된 읽기 쓰레드 부분 공지가 없기 때문입니다.

5. 많은 업데이트 쪽 RCU 사용이 새 데이터 구조를 발행하기 위해 단일 포인터를 수정합니다. 이 경우들 중 일부에서 RCU는 뒤이어 취소되는 트랜잭션 상의 포인터 업데이트를 보는게 안전히 허용되는데, 이 트랜잭션이 메모리 순서 규칙을 지키고 취소 과정이 연관된 구조체를 메모리 해제하는데 `call_rcu()`를 사용하는 동안은 그렇습니다. 불행히도, 모든 TM 구현이 트랜잭션 내에서 메모리 배리어를 존중하지는 않습니다. 아마도, 이는 트랜잭션이 원자적일 것이라 여겨지며, 트랜잭션 내에서의 액세스의 순서는 아무래도 상관 없다 생각했기 때문인 듯 합니다.
6. RCU 업데이트 내에서의 TM 사용을 금지합니다. 이는 동작할 것이 보장되지만, TM의 사용을 제한합니다.

특히 user-level RCU 와 해저드 포인터 구현의 발전을 놓고 보면 더 많은 방법들이 나타날 것으로 보입니다.⁶ 더 나은 성능과 확장성의 STM 구현들 중 다수가 RCU 비슷한 기법을 내부적으로 사용한다는 것은 흥미롭습니다 [Fra04, FH07, GYW⁺19, KMK⁺19].

Quick Quiz 17.3: MV-RLU는 상당히 좋아 보입니다! 이게 RCU를 무찌를 수 있을까요?

17.2.3.4 Extra-Transactional Accesses

락 기반 크리티컬 섹션 내에서는 동시에 액세스 되거나 심지어 이 락의 크리티컬 섹션 외부에서 수정되는 여러 변수들을 조정하는게 완전히 합법적인데 흔한 한 가지 예는 통계적 카운터입니다. 같은 일이 RCU read-side 크리티컬 섹션 내에서 가능하며, 이는 실제로 흔한 경우입니다.

“Dirty reads”라고 흔히 불리는, 제품 데이터베이스 시스템에서 널리 상용되는 메커니즘을 놓고 볼 때, 엑스트라-트랜잭션 (extra-transactional) 액세스가 TM의 부분에서 완화된 atomicity [BLM06] 등의 지점에서 상당한 관심을 받았음을 놀랍지 않습니다.

여기 엑스트라-트랜잭션 옵션이 몇개 있습니다:

⁶ 앞의 여러 대안들을 가져와 준 TxLinux group, Maged Michael, 그리고 Josh Triplett에게 감사드립니다.

1. 엑스트라-트랜잭션 액세스로 인한 충돌은 항상 트랜잭션을 취소시킨다. 이는 강한 atomicity입니다.
2. 엑스트라-트랜잭션 액세스로 인한 충돌은 무시되어, 트랜잭션을 통한 충돌만이 트랜잭션들을 취소시킬 수 있습니다. 이는 완화된 atomicity입니다.
3. 트랜잭션은 비 트랜잭션 오퍼레이션을 특수한 경우에 사용하는 것이 허용되는데, 메모리를 할당받거나 락 기반 크리티컬 섹션과 상호작용할 때가 그 예입니다.
4. 여러 트랜잭션에서 하나의 변수에 동시에 취해질 수 있는 일부 오퍼레이션을 (예를 들면, 더하기) 허용하는 하드웨어 확장을 만듭니다.
5. 트랜잭션 메모리에 완화된 의미를 도입합니다. 한가지 방법은 Section 17.2.3.3에서 설명된 대로 RCU 와의 결합을 하는 것으로, Gramoli 와 Guer-raoui는 다른 완화된 트랜잭션 방법을 여러가지 연구했는데 [GG14], 예를 들면 거대한 “우아한” 트랜잭션의 더 작은 트랜잭션으로의 제한된 조각내기로, 충돌 가능성 줄입니다(적은 성능과 확장성에도 불구하고). 아마도 더 많은 경험은 엑스트라-트랜잭션 액세스의 어떤 사용이 완화된 트랜잭션을 대체할 수 있음을 보일 겁니다.

트랜잭션은 진공 상태에서 다른 동기화 메커니즘과의 상호작용 필요성 없이 생각된 것처럼 보입니다. 그렇다면, 트랜잭션을 비 트랜잭션 액세스와 결합하려 할 때 나타나는 혼란과 복잡성은 놀랍지 않습니다. 그러나 트랜잭션이 격리된 데이터 구조로의 작은 업데이트로 국한될 것이 아니라면, 또는 현재 존재하는 수많은 병렬 코드와 상호작용하지 않는 새로운 프로그램에만 국한될 것이 아니라면, 트랜잭션은 가까운 미래에 거대 규모의 실용적 효과를 내기 위해선 그것들과 결합되어야 합니다.

17.2.4 Discussion

전체적인 TM 도입에의 장애물들은 다음 결론을 이끌어냅니다:

1. TM의 흥미로운 속성 중 하나는 트랜잭션이 취소되고 재시도 될 수 있다는 사실입니다. 이 속성은 버퍼링 되지 않은 I/O, RPC, 메모리 매핑 오퍼레이션, 시간 지연, 그리고 `exec()` 시스템 콜을 포함한 취소 불가한 오퍼레이션과 TM의 관계에서의 어려움의 근원을 이룹니다. 이 속성은 또한 실패의 가능성에서 피할 수 없는 모든 복잡도를 개발자에게 보이는 형태로 도입하게 되어버린다는 결론을 갖습니다.

- TM 의 또 다른 흥미로운 속성은 Shpeisman 등 [SATG⁰⁹]에 의해 이야기 되었듯, TM은 그것이 보호하는 데이터와 동기화를 관련지운다는 겁니다. 이 속성은 TM의 I/O, 메모리 매핑 오퍼레이션, 엑스트라-트랜잭션 액세스, 그리고 디버깅 브레이크포인트와의 문제의 근원을 이룹니다. 대조적으로, 락킹과 RCU를 포함한 전통적인 동기화 기능들은 동기화 기능과 그것이 보호하는 데이터 사이에 분명한 분리를 유지합니다.
- TM 분야에서 종사하는 많은 사람들의 선언된 목표들 중 하나는 거대한 순차적 프로그램의 병렬화를 쉽게 한다는 겁니다. 그러므로 개별 트랜잭션은 보통 순차적으로 수행될 것으로 예상되는데, 이는 TM의 멀티쓰레드 기반 트랜잭션에서의 문제의 많은 부분을 설명할 수도 있습니다.

Quick Quiz 17.4: `spin_trylock()` 같은게 존재하는데 TM이 실패의 컨셉을 가져왔다고 말하는게 말이 됩니까???

■ TM 연구자들과 개발자들은 이 모든 것에 대해 뭘 해야 할까요?

한가지 방법은 작은 영역부터 집중하는 것으로, 하드웨어의 잠재적 도움이 다른 동기화 기능에 비해 큰 도움을 줄 수 있는 작은 트랜잭션에 집중하거나 TM과 락킹의 결합된 접근법이 [PAT11] 향상된 생산성을 제공한다는 증거를 보이는 작은 프로그램에 집중하는 겁니다. Sun은 Rock 연구용 CPU에서 작은 트랜잭션 접근법을 취했습니다 [DLMN09]. 일부 TM 연구자들은 이 두개의 작은게-아름답다 접근법에 동의하는 듯하고 [SSHT93], 어떤 사람들은 TM에 훨씬 큰 희망을 가지며, 어떤 사람들은 높은 TM에 대한 기대는 TM의 가장 큰 적이 될 수도 있다고 [Att10, Section 6] 생각하는 듯 합니다. 그러나 TM이 더 큰 문제에 취해질 가능성도 상당히 있으며, 이 섹션은 TM이 이 높은 목표를 이루기 위해서는 해결해야만 할 문제들 몇가지를 나열했습니다.

물론, 연관된 모두는 이를 배움의 기회로 여겨야 합니다. TM 연구자들은 전통적 동기화 기능을 통해 성공적으로 거대 소프트웨어 시스템을 구축한 실무자들로부터 배울 것들이 많을 듯 합니다.

그리고 반대도 마찬가지죠.

Quick Quiz 17.5: 무얼 배우나요? 이 웃긴 섹션에 나열된 웃긴 많은 특수 경우들 같은 드문 경우들을 위한 메모리 기반 데이터 구조와 락킹에 TM을 그냥 사용하는게 어떤가요???

■ 그러나 지금으로썬, STM의 현 상태는 여러 만화로 요약될 수 있습니다. 먼저, Figure 17.9 이 STM의

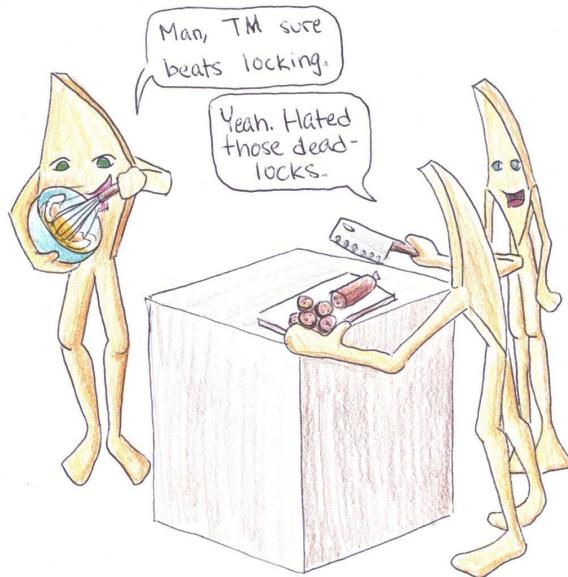


Figure 17.9: The STM Vision



Figure 17.10: The STM Reality: Conflicts



Figure 17.11: The STM Reality: Irrevocable Operations



Figure 17.12: The STM Reality: Realtime Response

비전을 보입니다. 항상 그렇듯, 실제는 약간 더 미묘하여, Figures 17.10, 17.11, 그리고 17.12에 보여진 것과 같습니다.⁷ 더 진중한 버전의 STM 회고도 있습니다 [Duf10a, Duf10b].

제한된 버전의 HTM을 제원하는 하드웨어가 상용으로 접근 가능한 한데, 다음 섹션에서 다릅니다.

17.3 Hardware Transactional Memory

Make sure your report system is reasonably clean and efficient before you automate. Otherwise, your new computer will just speed up the mess.

Robert Townsend

2021년 기준, 하드웨어 트랜잭션 메모리 (HTM)은 상업적으로 접근 가능한 상용 컴퓨터 시스템의 여러 형태로 수년째 사용해왔습니다 [YHLR13, Mer11, JSG12, Hay20]. 이 섹션은 병렬 프로그래머의 도구상자에서 HTM의 위치를 정의해 보려 합니다.

컨셉적 관점에서, HTM은 선정된 선언문들이 (“트랜잭션”) 다른 프로세서에서 수행중인 다른 트랜잭션의 관점에서 볼 때 어도막하게 효과를 발휘하도록 하기 위해 프로세서 캐시와 예측적 수행을 사용합니다. 이 트랜잭션은 begin-transaction 기계 명령에 의해 시작되고 commit-transaction 기계 명령에 의해 완료됩니다. 일반적으로 abort-transaction 기계 명령도 존재하는데, 예측을 찌그러뜨리고 (begin-transaction 명령과 뒤따르는 명령들이 수행되지 않은 것처럼) 실패 처리기에서의 수행을 시작합니다. 실패 처리기의 위치는 일반적으로 begin-transaction 명령에 의해 명시되는데, 명시적 실패 처리기 주소가 주어지거나 명령 자체에 의해 설정되는 조건 코드를 통해 주어집니다. 각 트랜잭션은 모든 다른 트랜잭션에 대해 원자적으로 수행됩니다.

HTM은 여러 중요한 이득을 갖는데, 데이터 구조의 자동화된 동적 파티셔닝, 동기화 기능의 캐시 미스 감소, 상당한 수의 실용적 어플리케이션의 지원이 포함됩니다.

그러나, 제대로 된 문서를 읽을 것이 항상 필요하며, HTM도 예외가 아닙니다. 이 섹션의 주요 지점은 어떤 조건에서 HTM의 이득이 그것의 제대로 된 문서에 숨어 있는 복잡도를 이겨내는가를 알아내는 것입니다. 따라서, Section 17.3.1은 HTM의 이득을 이야기하고,

⁷ 최근의 진행중인 학계 작업들은 리얼타임에서의 사용을 위한 기반 STM 시스템을 조사했는데 [And19, NA18], 성능 결과가 없음에도 불구하고, 그리고 리얼타임에서의 STM/HTM 결합 시스템은 빠른 일반적 경우 성능과 최악의 경우 진행 보장 사이에서의 선택이 필요함을 일부 보였습니다 [AKK⁺14, SBV10].

Section 17.3.2 는 그 약점을 이야기 합니다. 이는 이전의 섹션들과 논문들에서와 [MMW07, MMTW10] 같은 방법입니다.⁸

이어서 Section 17.3.3 는 리눅스 커널 (그리고 많은 유저 스페이스 어플리케이션)에서 사용되는 동기화 기능들과의 조합 관점에서의 HTM 의 약점을 설명합니다. Section 17.3.4 는 병렬 프로그래머의 도구상자 내에서 어디에 HTM 이 가장 잘 맞을지 알아보고, Section 17.3.5 는 HTM 의 영역과 인상을 크게 개선시킬 수도 있을 이벤트를 일부 나열해 봅니다. 마지막으로, Section 17.3.6 는 결론을 짓습니다.

17.3.1 HTM Benefits WRT Locking

HTM 의 주요 장점은 (1) 다른 동기화 기능에 의해 종종 일어나는 캐쉬 미스의 회피, (2) 동적으로 데이터 구조를 파티셔닝하는 능력, 그리고 (3) 상당한 수의 실용적 어플리케이션을 갖는다는 사실입니다. 저는 TM 에 대한 전통과 달리 두 가지 이유로 쉬운 사용성을 여기에 포함시키지 않습니다. 첫째로, 쉬운 사용성은 HTM 의 주요 장점으로부터 생겨나는 것인데, 그게 이 섹션이 주목하는 부분입니다. 둘째로, 기본적 프로그래밍 재능에 대한 검사와 [Bor06, DBA09, PBCE20] 심지어 취업 면접에서의 작은 프로그래밍 테스트의 사용에 대한 [Bra07] 상당한 노란이 있었습니다. 이는 무엇이 프로그래밍을 쉽게 하는지 어렵게 하는지에 대한 확고한 이해를 정말로 갖지 못하고 있음을 보입니다. 따라서, 이 섹션의 나머지 부분은 앞에 나열된 세 가지 장점에 집중합니다.

17.3.1.1 Avoiding Synchronization Cache Misses

대부분의 동기화 메커니즘은 어토믹 명령에 의해 운영되는 데이터 구조에 기반합니다. 이 어토믹 명령들은 일반적으로 일단 연관된 캐쉬라인이 그들이 수행중인 CPU 에 의해 소유되게 하여, 다른 CPU 에서의 같은 동기화 기능의 사용은 캐쉬 미스를 일으키게 합니다. 이 통신을 위한 캐쉬 미스는 전통적 동기화 메커니즘의 성능과 확장성을 크게 악화시킵니다 [ABD⁺97, Section 4.2.3].

대조적으로, HTM 은 해당 CPU 의 캐쉬를 이용해 동기화를 하므로 별도의 동기화 데이터 구조와 그로 인한 캐쉬 미스를 회피합니다. HTM 의 장점은 락 데이터 구조가 별도의 캐쉬 라인에 위치하는 경우 가장 커지는데, 이 경우 크리티컬 섹션을 HTM 트랜잭션으로 변환시키는 것은 크리티컬 섹션의 캐쉬 미스로 인한 오버헤드를 완전히 줄일 수 있습니다. 이로 인한 절약은 짧은 크리티컬 섹션이라는 흔한 경우 상당히 클 수 있는데, 최소한

⁸ 다른 저자들, Maged Michael, Josh Triplett, Jonathan Walpole, Andi Kleen 등과의 많은 자극적 토론에 소중한 감사 to 표합니다.

제거된 락이 그 락에 의해 보호되는 변수의 캐쉬 라인을 공유하는 많은 경우가 아닐 경우에는 그렇습니다.

Quick Quiz 17.6: 왜 락 변수와 보호되는 변수가 캐쉬 라인을 공유하는 일반적 경우가 중요하죠?



17.3.1.2 Dynamic Partitioning of Data Structures

일부 전통적인 동기화 메커니즘의 사용에 있어서의 주요 장애물은 정적으로 데이터 구조를 분할해야 한다는 필요성입니다. 가장 흔한 예로는 각 해쉬 체인의 조각이 되는 해쉬 테이블과 같은 쉽게 분할될 수 있는 데이터 구조도 여럿 있습니다. 각 해쉬 체인을 위한 락을 할당하는 것은 간단히 이 해쉬 테이블을 해당 체인으로 국한된 오퍼레이션에 대해 분할할 수 있습니다.⁹ 분할하기는 비슷하게 배열, 래디스 트리, 스kip리스트, 그리고 여러 다른 데이터 구조에 있어서도 간단합니다.

그러나, 여러 종류의 트리와 그래프에 있어 분할하기는 상당히 어려우며, 그 결과물은 종종 복잡합니다 [Ell80]. 일반적 데이터 구조를 분할하기 위해선 Δ -wo-phased 락킹과 락의 해싱된 배열을 사용하는게 가능하지만, Section 17.3.3 에서 곧 논의할 것인지만 다른 기법이 선호된다는게 밝혀졌습니다 [Mil06]. 동기화 캐쉬 미스의 회피 덕에, HTM 은 거대한 분할 불가능한 데이터 구조에 대해 최소한 상대적으로 적은 업데이트 를 설정하면 사용 가능성이 있습니다.

Quick Quiz 17.7: HTM 의 성능과 확장성에 있어 상대적으로 적은 업데이트가 왜 중요한가요?



17.3.1.3 Practical Value

HTM 의 실용적 가치에 대한 일부 증거가 여러 하드웨어 플랫폼에서 선보였는데, Sun Rock [DLMN09], Azul Vega [Cli09], IBM Blue Gene/Q [Mer11], Intel Haswell TSX [RD12], 그리고 IBM System z [JSG12] 이 포함됩니다.

예상되는 실용적 이득은 다음을 포함합니다:

1. 메모리 내 데이터 접근과 업데이트 시의 락 제거 [MT01, RG02].
2. 거대한 분할 불가한 데이터 구조로의 동시의 액세스와 작은 무작위 업데이트.

그러나, HTM 은 또한 매우 실제적인 단점을 갖는데, 다음 섹션에서 다룹니다.

⁹ 그리고 이 방법을 여러 해쉬 체인을 접근하는 오퍼레이션에 대해서는 연관된 모든 락을 해쉬 순서로 획득하게 하는 식으로 쉽게 확장할 수 있습니다.

17.3.2 HTM Weaknesses WRT Locking

HTM의 컨셉은 상당히 간단합니다: 한 그룹의 액세스와 업데이트가 원자적으로 이루어집니다. 그러나, 많은 간단한 아이디어의 경우처럼, 실제 세계의 실제 시스템에 이를 적용할 때 복잡도가 나타납니다. 이 복잡도는 다음과 같습니다:

1. 트랜잭션 크기 제한.
2. 충돌 처리.
3. 취소하기와 되돌리기.
4. 진행 보장의 부재.
5. 취소 불가한 오퍼레이션.
6. 의미상의 차이.

이 복잡성 각각이 다음 섹션에서 다루어지며, 그 뒤에 요약이 나옵니다.

17.3.2.1 Transaction-Size Limitations

현재 HTM 구현의 트랜잭션 크기 제한은 트랜잭션에 의해 영향받는 데이터를 프로세서 캐쉬에 둔다는 점에서 기인합니다. 이게 특정 CPU가 트랜잭션을 자신의 캐쉬에 국한시켜서 수행함으로써 이를 원자적으로 행하는 것으로 다른 CPU에게 보이게 하지만, 이는 또한 여기 들어맞지 않는 트랜잭션은 커밋될 수 없음을 의미합니다. 더 나아가, 인터럽트, 시스템콜, 예외, 트랩, 그리고 컨텍스트 스위치 같은 수행 컨텍스트를 바꾸는 이벤트는 해당 CPU의 진행중인 트랜잭션을 취소시키거나 다른 수행 문맥의 캐쉬 사용량으로 인해 트랜잭션 크기를 더 제한해야 합니다.

물론, 현대의 CPU는 큰 캐쉬를 갖는 경향을 보이며, 많은 트랜잭션에 필요한 데이터는 쉽게 1메가바이트 캐쉬에 담길 겁니다. 불행히도, 캐쉬에 있어 작은 크기가 전부는 아닙니다. 문제는 대부분의 캐쉬는 하드웨어로 구현된 해쉬 테이블로 생각될 수 있습니다. 그러나, 하드웨어 캐쉬는 버킷을 (일반적으로 *set* 이라 불립니다) 연결시키지 않고, 고정된 수의 *set* 별 캐쉬라인을 제공합니다. 각 *set*에 의해 제공된 원소의 갯수를 해당 캐쉬의 *associativity* 라 불립니다.

캐쉬 *associativity*는 다양하지만, 제가 이를 타이핑하는 랩탑에 있는 8-way *associativity* 레벨 0 캐쉬는 드물지 않습니다. 이게 의미하는 바는 특정 트랜잭션에 아홉개의 캐쉬 라인을 건드려야 하며 그 아홉개의 캐쉬 라인이 모두 같은 *set*으로 매핑된다면 해당 캐쉬에 얼마나 많은 추가적 공간이 존재하는가와 관계없이 이 트랜잭션은 결코 완료될 수 없다는 겁니다. 그렇습니다, 특정 데이터 구조에서 데이터 원소가 무작위적으로 선택된다면 그

트랜잭션이 커밋될 가능성은 상당히 높습니다, 그러나 보장은 없습니다 [McK11c].

이 제한을 완화시키려는 일부 연구가 있습니다. 완전한 *associativity*의 *victim cache*는 이 *associativity* 제한을 완화시킬 테지만, *victim* 캐쉬의 크기에 대한 까다로운 성능과 에너지 효율 제한이 있습니다. 그렇다고 하나, 수정되지 않은 캐쉬 라인을 위한 HTM *victim* 캐쉬는 상당히 작을 수 있는데, 주소만 가지면 되기 때문입니다: 데이터 자체는 메모리에 저장되거나 다른 캐쉬에 의해 따라갈 수 있으며, 주소 자체만으로도 쓰기와의 충돌을 탐지하기 충분합니다 [RD12].

Unbounded transactional memory (UTM) 방법은 [AAKL06, MBM⁺06] DRAM을 극단적으로 큰 *victim* 캐쉬로 사용합니다만, 그런 방법을 제품 품질의 캐쉬 일관성 메커니즘과 결합하는 것은 여전히 해결되지 않은 문제입니다. 또한, DRAM을 *victim* 캐쉬로 사용하는 것은 불행한 성능과 에너지 효율성 결론을 가질 수 있으며, 특히 *victim* 캐쉬가 완전한 *associativity*를 갖는다면 그렇습니다. 마지막으로, “unbounded”라는 UTM의 속성은 모든 DRAM이 *victim* 캐쉬로 사용될 수 있음을 가정하는데, 실제로는 크지만 여전히 고정된 양의 DRAM이 특정 CPU에 할당되므로 해당 CPU의 트랜잭션의 크기는 제한될 겁니다. 다른 방법들은 하드웨어와 소프트웨어 트랜잭션 메모리의 결합을 사용하며 [KCH⁺06] 여기서 STM은 HTM의 fallback 메커니즘으로 생각될 수 있습니다.

그러나, 제가 알기로는 TM 읽기 집합의 표현의 간략화라는 예외가 있지만 현재 사용 가능한 시스템들은 이런 연구 아이디어들을 구현하지 않았는데, 아마도 좋은 이유가 있었을 겁니다.

17.3.2.2 Conflict Handling

첫번째 복잡성은 충돌 가능성입니다. 예를 들어, 트랜잭션 A와 B가 다음과 같이 정의되었다고 해봅시다.

Transaction A	Transaction B
$x = 1;$	$y = 2;$
$y = 3;$	$x = 4;$

각 트랜잭션이 각자의 프로세서에서 동시에 수행된다고 해봅시다. 트랜잭션 A *x*로의 스토어를 트랜잭션 B의 *y* 스토어와 동시에 행하면, 어떤 트랜잭션도 진행될 수 없습니다. 이를 자세히 보기 위해, 트랜잭션 A가 *y*로의 스토어를 하는 걸 봅시다. 그럼 트랜잭션 A는 트랜잭션 B와 엮이게 되어, 각자에 대해 트랜잭션에 원자적으로 수행되어야 한다는 요구사항을 여기게 됩니다. 트랜잭션 B가 *x*로의 스토어를 하게 하는 것 역시 비슷하게 원자적 수행 요구사항을 여기게 합니다. 이

상황은 충돌 (*conflict*) 라 명명되었는데, 두 동시에 트랜잭션이 같은 변수에 액세스 하며 그 액세스 중 최소 하나는 스토어일 때 발생합니다. 따라서 이 시스템은 수행이 진행될 수 있게 하기 위해 트랜잭션 중 하나 또는 둘 다를 중지시켜야 합니다. 어떤 트랜잭션을 중지시킬지의 선택은 박사 학위논문을 만들기 충분할 만큼의 능력을 얻게 할만큼 흥미로운 주제일텐데 그런 예도 있습니다 [ATC^{11]}.¹⁰ 이 섹션의 목표에 집중하기 위해, 우린 시스템이 무작위적 선택을 있다고 가정할 수 있습니다.

또 다른 복잡성은 충돌 탐지로, 최소한 가장 간단한 경우에는 비교적 간단합니다. 프로세서는 트랜잭션을 수행할 때 이 트랜잭션에 의해 만져진 모든 캐쉬라인을 표시합니다. 만약 이 프로세서의 캐쉬가 현재 트랜잭션에 의해 접촉된 것으로 표시된 캐쉬 라인에 연관된 요청을 받는다면, 잠재적 충돌이 일어난 겁니다. 더 정교한 시스템은 현재 프로세서의 트랜잭션이 그 요청을 보낸 프로세서의 것보다 앞서 수행되도록 노력할 것이고, 이 과정을 최적화 하는 것은 역시나 박사 학위논문을 만들 능력을 얻을 수 있게 할 겁니다. 그러나 이 섹션은 매우 간단한 충돌 탐지 전략을 가정합니다.

그러나, HTM이 효과적으로 동작하기 위해선 충돌의 가능성성이 상당히 낮아야 하는데, 이는 결국 데이터 구조가 충분히 낮은 충돌 확률을 유지하게끔 짜여져야 할 것을 필요로 합니다. 예를 들어, 간단한 삽입, 삭제, 그리고 탐색을 하는 red-black 트리는 이 경우에 들어맞습니다만, 트리의 원소들의 정확한 수를 유지하는 red-black 트리는 그렇지 않습니다.¹¹ 또 다른 예로, 하나의 트랜잭션 내에서 트리의 모든 원소를 접근하는 red-black 트리는 높은 충돌 가능성을 가져서 성능과 확장성을 떨어뜨립니다. 그 결과, 많은 순차적 프로그램들은 HTM이 효과적으로 동작하기 전에 일부 재구축을 필요로 할 겁니다. 어떤 경우에는, 실무자들은 추가적 단계들을 취하고 (red-black 트리의 경우, radix 트리나 해쉬 테이블 같은 분할 가능한 데이터 구조로의 변경 같은) 간단히 락킹을 하는 걸 선호할 텐데, 특히 HTM이 모든 관련 아키텍처에서 사용될 준비가 될 때까지는 그럴 겁니다 [Cli09].

Quick Quiz 17.8: 동기화 메커니즘의 선택에 관계 없이 red-black 트리가 어떻게 모든 원소를 효율적으로 접근할 수 있죠???

더 나아가, 동시에 트랜잭션 사이에서의 충돌하는 액세스의 가능성이 실패를 초래할 수 있습니다. 그런 실패를 처리하는 것에 대해 다음 섹션에서 다릅니다.

¹⁰ Liu 와 Spear 의 “Toxic Transactions” [LS11] 는 특히 교훈적입니다.

¹¹ 이 카운트를 업데이트 해야 한다는 필요성이 트리에의 삽입과 삭제가 서로 충돌하게 만들어, 강력한 non-commutativity 를 초래합니다 [AGH^{11a}, AGH^{11b}, McK11b].

17.3.2.3 Aborts and Rollbacks

어떤 트랜잭션이든 언제든 중단될 수도 있으므로, 트랜잭션이 취소될 수 없는 문장을 포함하지 않는게 중요합니다. 이는 트랜잭션은 I/O, 시스템콜, 또는 디버깅 브레이크포인트를 수행할 수 없음을 (HTM 트랜잭션을 위한 단계별 디버거가 없습니다!!!) 의미합니다. 대신, 트랜잭션은 스스로를 평범한 캐쉬된 메모리에의 접근으로 국한시켜야 합니다. 더 나아가, 일부 시스템에서는 인터럽트, 예외, 트랩, TLB 미스, 그리고 다른 이벤트들도 트랜잭션을 중단시킬 겁니다. 오류 조건에 대한 적절치 않은 처리로 초래된 수많은 버그를 생각해 보면, 중단과 되돌림이 사용성에 어떤 영향을 주는지 묻는게 공정합니다.

Quick Quiz 17.9: 하지만 디버거가 트랜잭션의 성공한 라인에 브레이크포인트를 설정하고 앞의 트랜잭션 수행의 단계를 다시 추적하길 재시도하는 방법으로 단계별 수행을 에뮬레이션하는 건 왜 안되죠?

물론, 중단과 되돌리기는 HTM이 하드 리얼타임 시스템에서 유용할지 질문하게 합니다. HTM의 성능 이익은 중단과 되돌리기의 비용을 넘어서며, 그렇다면 어떤 조건에서 그럴까요? 트랜잭션은 우선순위 높이기를 할 수 있을까요? 또는 높은 우선순위 쓰레드를 위한 트랜잭션은 낮은 우선순위 쓰레드를 중단시켜야 할까요? 만약 그렇다면, 하드웨어는 어떻게 효율적으로 우선순위를 전달받나요? HTM의 리얼타임에서의 사용에 대한 글들은 상당히 적은데, 아마도 HTM을 비 리얼타임 환경에서 잘 동작하게 만드는데도 충분한 정도를 넘어서는 문제들이 산적했기 때문일 겁니다.

현재의 HTM 구현은 결정론적으로 특정 트랜잭션을 중단시키므로, 소프트웨어는 fallback 코드를 제공해야만 합니다. 이 fallback 코드는 예를 들어 락킹 같은 다른 형태의 동기화를 사용해야만 합니다. 락 기반 fallback이 사용된다면, 데드락의 가능성을 포함한 모든 락킹의 한계들이 다시 나타납니다. 물론 이 fallback이 자주 사용되지 않길 바랄 수 있겠는데, 그럼 더 간단하고 덜 데드락에 취약한 락킹 설계가 사용될 수 있을 겁니다. 그러나 이는 시스템이 어떻게 락 기반 fallback 사용에서 트랜잭션으로 전환하는지 질문을 일으킵니다.¹² 한가지 방법은 test-and-test-and-set 방법 [MT02] 으로, 락이 해제되기 전까지는 모두가 뒤로 물러서서 시스템이 트랜잭션 모드의 깔끔한 상태에서 시작할 수 있게 하는 겁니다. 그러나, 이는 상당한 스피닝을 초래할 수 있는데, 락을 뜯 쓰레드가 블록되거나 preemption 당한다면 현명하지 않은 선택일 수도 있습니다. 또 다른 방법은 락을 뜯 쓰레드와 병렬로 트랜잭션이 진행될 수 있게 하는 것인데 [MT02], 이는 원자성 유지에 어려움을 자아내

¹² Fallback 모드에서 멈춰있는 어플리케이션의 가능성은 “lemming effect” 라 명명되었습니다.

며, 이 쓰레드가 락을 쥐는 이유가 연관된 트랜잭션이 캐쉬 크기에 맞지 않기 때문이라면 특히 그렇습니다.

마지막으로, 중단과 되돌리기의 가능성을 처리하는 것은 가능한 오류 조건을 모두 올바르게 처리해야하는 개발자에게 추가적 짐을 지우는 것으로 보입니다.

HTM 사용자들은 fallback 코드 경로와 fallback 코드에서 트랜잭션 사용 코드로의 전환 두 경우에 상당한 검증 노력을 들여야 함이 분명합니다. HTM 하드웨어의 검증 요구사항이 덜 힘들 거라 믿을 이유 또한 없습니다.

17.3.2.4 Lack of Forward-Progress Guarantees

트랜잭션 크기, 충돌, 그리고 중단/되돌리기가 모두 트랜잭션들을 중단되게 할 수 있지만, 누군가는 충분히 작고 짧은 트랜잭션은 결국 성공할 것이 보장될 것을 희망할 수도 있겠습니다. 이는 어토믹 오퍼레이션을 구현하기 위해 이 명령을 사용하는 코드가 compare-and-swap (CAS) 와 load-linked/store-conditional (LL/SC) 오퍼레이션을 무조건적으로 재시도 하듯 트랜잭션이 무조건적으로 재시도 되는 것을 허용할 겁니다.

불행히도, low-clock-rate 학술 연구 프로토타입 [SBV10] 외에는 현재 사용 가능한 HTM 구현들은 어떤 종류의 진행 보장도 하지 않습니다. 앞서 언급되었듯, 따라서 HTM은 시스템의 데드락을 막는데 사용될 수 없습니다. 희망컨대 미래의 HTM 구현은 어떤 종류의 진행 보장을 제공할 겁니다. 그 전까지는, HTM은 리얼타임 어플리케이션에서 상당한 주의와 함께 사용되어야 할 겁니다.

2021년 기준으로 이 암울한 그림에서의 한가지 예외는 제한된 트랜잭션 [JSG12] 을 제공하는 IBM 메인프레임입니다. 이 제약은 상당히 강력하며 Section 17.3.5.1에 보여져 있습니다. HTM 진행 보장이 메인프레임에서 일반 CPU 제품군까지 퍼질 것인지 보는 것도 흥미롭겠습니다.

17.3.2.5 Irrevocable Operations

중단과 되돌리기의 또 다른 결말은 HTM 트랜잭션은 되돌릴 수 없는 오퍼레이션을 수용할 수 없다는 겁니다. 현재의 HTM 구현은 일반적으로 트랜잭션 내의 모든 액세스가 캐쉬될 수 있는 메모리로의 것일 것으로 (따라서 MMIO 액세스를 금지합니다) 제한하고 인터럽트, 트랩, 그리고 예외의 경우 트랜잭션을 중단시킵니다 (따라서 시스템콜을 금지합니다).

Buffered I/O 는 해당 버퍼 채우기/비우기 오퍼레이션이 트랜잭션 외에서 일어나는 경우 HTM 트랜잭션에 들어갈 수 있음을 알아두시기 바랍니다. 이게 가능한 이유는 데이터를 버퍼에 더하고 제거하는 건 되돌릴 수 있기 때문입니다: 실제 버퍼 채우기/비우기 오퍼레이션 만이 취소 불가합니다. 물론, 이 buffered-I/O 방법은 이

I/O 를 트랜잭션의 크기에 포함시켜서 트랜잭션의 크기를 키우고 실패의 가능성은 높이는 효과를 일으킵니다.

17.3.2.6 Semantic Differences

HTM 이 많은 경우에 락킹의 즉시 대체제로 사용될 수 있지만 (따라서 Transactional lock elision [DHL⁺08] 이라는 명칭이 나왔습니다), 그 의미에는 작은 차이들이 있습니다. 트랜잭션으로 수행될 때 데드락이나 라이브락을 초래할 수 있는 조합된 락 기반 크리티컬 섹션을 포함시키는 특히 나쁜 예가 Blundell [BLM06]에 의해 알려졌습니다만, 훨씬 간단한 예는 텅빈 크리티컬 섹션입니다.

락 기반 프로그램에서, 텅 빈 크리티컬 섹션은 이 락을 잡았던 앞선 프로세스들이 이제 이를 해제했음을 보장합니다. 이 용법은 2.4 리눅스 커널의 네트워킹 스택에서 구성 변경을 처리하기 위해 사용되었습니다. 그러나 이 텅 빈 크리티컬 섹션은 트랜잭션으로 변환된다면, 그 결과는 no-op입니다. 달리 말하자면, transactional lock elision은 락킹의 데이터 보호 의미를 유지하나, 락킹의 시간 기반 메세징 의미는 잃습니다.

Quick Quiz 17.10: 하지만 누가 빙 락 기반 크리티컬 섹션을 필요로 합니까???

Quick Quiz 17.11: Transactional lock elision은 간단히 빙 락 기반 크리티컬 섹션을 제거하지 않음으로써 락킹의 시간 기반 메세징 의미를 다룰 수 있지 않을까요?

Quick Quiz 17.12: 현대의 하드웨어에서 [MOZ09] 어떻게 시간에 기반한 병렬 소프트웨어가 동작할 거라 생각될 수 있습니까?

락킹과 트랜잭션 사이의 한가지 중요한 의미상 차이는 락 기반 리얼타임 프로그램에서 우선순위 역전을 막기 위해 사용되는 우선순위 높이기입니다. 우선순위 역전이 일어날 수 있는 경우 중 하나는 락을 잡은 낮은 우선순위 쓰레드가 중간 우선순위 CPU 사용 쓰레드에 preemption 당할 때입니다. 그런 중간 우선순위 쓰레드가 CPU 당 최소 하나 있다면, 이 낮은 우선순위 쓰레드는 다시 동작할 기회를 얻지 못합니다. 이제 어떤 높은 우선순위 쓰레드가 이 락을 얻으려 하면 블록됩니다. 이 쓰레드는 이 낮은 우선순위 쓰레드가 이 락을 해제하기 전까지 그 락을 얻을 수 없고, 낮은 우선순위 쓰레드는 동작할 기회를 얻기 전까지는 이 락을 놓을 수 없으며, 중간 우선순위 쓰레드가 CPU를 놓기 전까지는 동작할 기회를 얻지 못합니다. 따라서, 이 중간 우선순위 쓰레드는 실질적으로 높은 우선순위 프로세스를 블록하고 있으며, 따라서 “우선순위 역전”이라는 이름이 붙었습니다.

Listing 17.1: Exploiting Priority Boosting

```

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(500); /* sleep 0.5 ms. */
20         acquire_lock(&boost_lock[i]);
21         release_lock(&boost_lock[i]);
22         i = i ^ 1;
23     }
24 }

```

우선순위 역전을 막는 한가지 방법은 우선순위 상속으로, 락에 블록되는 높은 우선순위 쓰레드가 임시적으로 자신의 우선순위를 락을쥔 쓰레드에게 주는 것인데, 우선순위 높이기 라고도 불립니다. 그러나, 우선순위 높이기는 Listing 17.1에 보인 것처럼 우선순위 역전을 막는 것 이외의 것을 위해서도 사용될 수 있습니다. 이 리스트의 라인 1-12는 매 밀리세컨드마다 수행되어야만 하는 낮은 우선순위 프로세스를 보이고, 라인 14-24는 `boostee()`가 필요에 따라 주기적으로 수행됨을 보장하기 위해 우선순위 높이기를 사용하는 높은 우선순위 프로세스를 보입니다.

`boostee()` 함수는 두개의 `boost_lock[]` 락들 중 하나를 항상 잡아서 `booster()`의 라인 20-21은 필요에 따라 우선순위를 높일 수 있게 함으로써 이를 가능하게 합니다.

Quick Quiz 17.13: 하지만 Listing 17.1의 `boostee()` 함수는 그 대신 락을 반대 순서로 잡습니다! 이는 데드락을 초래할 수 있지 않나요?



이 조정은 `boostee()`가 시스템이 바빠지기 전에 라인 5에서 첫번째 락을 잡을 것을 필요로 하지만, 이는 현대의 하드웨어에서 조차 쉽게 해결됩니다.

불행히도, 이 조정은 transactional lock elision의 존재에서 쉽게 부서질 수 있습니다. `boostee()` 함수의 겹치는 크리티컬 섹션은 하나의 무한한 트랜잭션이 되는데, 이는 언젠가 중단될 것인데, 예를 들어 `boostee()` 함수를 수행하는 쓰레드가 `preemption` 당하는 첫번째 경우가 있겠습니다. 이 시점에서, `boostee()`는 락킹으로 되돌아가지만, 낮은 우선순위와 조용한 초기화 단계는 이미 끝났음을 놓고 보면 (애초에 `boostee()`가 `preemption`

당한게 그 이유입니다), 이 쓰레드는 다시 수행될 기회를 영원히 잡지 못할 수도 있습니다.

그리고 `boostee()` 쓰레드가 락을 잡고 있지 않다면, `booster()` 쓰레드의 Listing 17.1의 라인 20와 21에서의 텅 빈 크리티컬 섹션은 아무 효과 없는 텅 빈 트랜잭션이 되어, `boostee()`는 결코 수행되지 않습니다. 이 예는 transactional memory의 되돌리고 재시도 하기 방법의 미묘한 결과를 보입니다.

경험은 추가적인 미묘한 의미적 차이들을 드러낼 거라는 점을 생각해 보면, HTM 기반 락 제거를 거대한 프로그램에 적용하는 것은 주의와 함께 이루어져야 합니다. 그러나, 그게 적용되는 곳에서라면 HTM 기반 락 제거는 락 변수와 연관된 캐쉬 미스를 제거할 수 있으며, 이는 거대한 실제 세계 소프트웨어 시스템에서 2015년 초 기준으로 수십 퍼센트의 성능 향상을 가져왔습니다. 따라서 우리는 안정적인 지원을 제공하는 하드웨어에서의 이 기술의 상당한 사용을 예상할 수 있습니다.

Quick Quiz 17.14: 그러니까 여러 사람이 락킹을 대체하려고 했고, 그들은 대부분 그저 락킹을 최적화하는데 그치는 건가요???

**17.3.2.7 Summary**

HTM이 강력한 사용처를 가지게 될 것으로 보이긴 합니다만, 현재의 구현은 주의 깊은 처리를 필요로 할 심각한 트랜잭션 크기 제한, 충돌 처리 복잡성, 중단하고 되돌리기 문제, 그리고 의미적 차이를 갖고 있습니다. 락킹 대비 HTM의 현재 상황은 Table 17.1으로 요약될 수 있습니다. 여기서 볼 수 있듯, HTM의 현재 상황은 락킹의 심각한 단점 일부를 완화시키지만,¹³ 이를 위해 그 자체의 상당한 수의 단점을 가져옵니다. 이 단점들은 TM 커뮤니티의 리더에 의해 인정되었습니다 [MS12].¹⁴

또한, 이게 전부가 아닙니다. 락킹은 보통 그 자체로 사용되지 않고, 일반적으로 레퍼런스 카운팅, 어토믹 오퍼레이션, non-blocking 데이터 구조, 해저드 포인터 [Mic04, HLM02], 그리고 RCU [MS98a, MAK⁺01, HMBW07, McK12b]를 포함한 다른 동기화 메커니즘들과 결합되어 사용됩니다. 다음 섹션은 어떻게 그런 결합이 그림을 바꾸는지 보입니다.

¹³ 공정을 위해, 락킹의 단점들은 Section 17.3.3에서 논의되듯 데드락 탐지기 [Cor06a], 락킹을 위해 조정된 여러 데이터 구조들, 그리고 중간의 긴 역사를 포함해 잘 알려지고 널리 사용되는 공학적 해결책이 존재함을 강조해 두는게 중요하겠습니다. 또한, 락킹이 많은 학술 논문을 살짝 보는 것만으로도 사람들을 납득시킬 수 있을만큼 금박한 것이었다면, 그 많은 거대한 락 기반 병렬 프로그램은 (FOSS와 독점 소프트웨어 양쪽) 어디서 나왔겠습니까?

¹⁴ 또한, 2011년 초, 저는 transactional memory의 기본 가정에 대한 비평을 전하기 위해 초대된 바 있습니다 [McK11e]. 청중들은 놀랍게도 적대적이지 않았지만, 제가 발표 중에도 시차로 인해 괴로워하고 있었기 때문에 제게 편하게 대해준 것일 수도 있습니다.

Table 17.1: Comparison of Locking and HTM (**Advantage** , **Disadvantage** , **Strong Disadvantage**)

	Locking	Hardware Transactional Memory
Basic Idea	Allow only one thread at a time to access a given set of objects.	Cause a given operation over a set of objects to execute atomically.
Scope	Handles all operations.	Handles revocable operations. Irrevocable operations force fallback (typically to locking).
Composability	Limited by deadlock.	Limited by irrevocable operations, transaction size, and deadlock (assuming lock-based fallback code).
Scalability & Performance	Data must be partitionable to avoid lock contention.	Data must be partitionable to avoid conflicts.
	Partitioning must typically be fixed at design time.	Dynamic adjustment of partitioning carried out automatically down to cacheline boundaries. Partitioning required for fallbacks (less important for rare fallbacks).
	Locking primitives typically result in expensive cache misses and memory-barrier instructions.	Transactions begin/end instructions typically do not result in cache misses, but do have memory-ordering and overhead consequences.
	Contention effects are focused on acquisition and release, so that the critical section runs at full speed.	Contention aborts conflicting transactions, even if they have been running for a long time.
Hardware Support	Privatization operations are simple, intuitive, performant, and scalable.	Privatized data contributes to transaction size.
	Commodity hardware suffices.	New hardware required (and is starting to become available).
Software Support	Performance is insensitive to cache-geometry details.	Performance depends critically on cache geometry.
	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.

17.3.3 HTM Weaknesses WRT Locking When Augmented

실무자들은 락킹의 단점을 일부를 회피하기 위해 레퍼런스 카운팅, 어토믹 오퍼레이션, non-blocking 데이터 구조, 해저드 포인터, 그리고 RCU 를 오랫동안 사용해 왔습니다. 예를 들어, 데드락은 레퍼런스 카운트, 해저드 포인터, 또는 RCU 를 데이터 구조 보호에 사용함으로써 막아질 수 있는데, 특히 읽기 전용 크리티컬 섹션의 경우 그렇습니다 [Mic04, HLM02, DMS⁺12, GMTW08, HMBW07]. 이 방법은 또한 데이터 구조를 쪼개야 하는 필요를 줄이는데, Chapter 10 에서 이를 다루었습니다. RCU 는 더 나아가 경쟁에서 자유로운 bounded wait-free read-side 기능들 [MS98a, DMS⁺12] 을 제공하며, 해저드 포인터는 lock-free read-side 기능들 [Mic02, HLM02, Mic04] 을 제공합니다. 이 점들을 Table 17.1 에 더한, 증강된 락킹과 HTM 사이의 비교가 Table 17.2 에 있습니다. 두 표 사이의 차이에 대한 요약은 다음과 같습니다:

1. Non-blocking read-side 미커니즘의 사용은 데드락 문제를 완화시킵니다.
2. 해저드 포인터와 RCU 같은 read-side 커니즘은 쪼개질 수 없는 데이터에 효율적으로 사용될 수 있습니다.
3. 해저드 포인터와 RCU 는 서로간이나 업데이트 쓰레드와 경쟁하지 않아서 읽기가 대부분인 워크로드에 훌륭한 성능과 확장성을 제공합니다.
4. 해저드 포인터와 RCU 는 진행 보장을 제공합니다 (각각 lock freedom 과 bounded wait-freedom 을).
5. 해저드 포인터와 RCU 를 위한 사설화 오퍼레이션은 간단합니다.

시력이 좋은 분들을 위해, Table 17.3 는 Tables 17.1 and 17.2 를 합쳤습니다.

물론, 다음 섹션에서 이야기 하듯 HTM 의 증강도 가능합니다.

17.3.4 Where Does HTM Best Fit In?

HTM 의 적용 영역이 page 166 의 Figure 9.30 에서의 RCU 로 보이는 것처럼 잘 파악되기까지는 시간이 걸릴 것으로 보이지만, 그게 그 방향으로의 이동을 시작하지 않을 이유는 아닙니다.

거대한 멀티프로세서에서 돌아가는 거대한 메모리상의 데이터 구조에서의 상대적으로 다른 부분들에 상대적으로 작은 변화를 포함하는 업데이트가 많은 워크로드에서 HTM 은 가장 잘 맞을 것으로 보이는데, 이는

현재 HTM 구현의 크기 제한과 맞고 충돌과 그에 따른 중단 및 되돌리기의 가능성을 최소화 하기 때문입니다. 이 시나리오는 현재의 동기화 기능을 가지고는 처리하기가 상대적으로 어려운 것입니다.

락킹을 HTM 과 함께 사용하는 것은 HTM 의 취소 불가 오퍼레이션에서의 문제를 극복할 것으로 보이며, RCU 나 해저드 포인터의 사용은 HTM 의 트랜잭션 크기 제한을 데이터 구조의 큰 부분을 움직이는 읽기 전용 오퍼레이션에서 완화할 수 있을 수도 있습니다 [PMDY20]. 현재의 HTM 구현은 무조건적으로 RCU 나 해저드 포인터 읽기 쓰레드와 충돌하는 업데이트 트랜잭션을 중단시키지만, 미래의 HTM 구현은 이 동기화 메커니즘들과 더 부드럽게 상호작용할지도 모릅니다. 그때까지는, 거대한 RCU 나 해저드 포인터 read-side 크리티컬 섹션과의 업데이트 충돌 확률은 동일한 읽기 전용 트랜잭션과의 충돌의 확률보다 훨씬 작아야 할 겁니다.¹⁵ 그러나, RCU 나 해저드 포인터 읽기 쓰레드의 지속적인 등장은 연관된 충돌의 지속적 등장으로 인해 업데이트 쓰레드를 기아에 빠뜨릴 수도 있습니다. 이 취약점은 extra-transactional 읽기에게 로드되는 메모리 위치의 트랜잭션 전 복사본을 제공하는 식으로 제거될 수 있습니다 (상당한 하드웨어 비용과 복잡도가 들겠지만).

HTM 트랜잭션이 fallback 을 가져야만 한다는 사실은 어떤 경우엔 데이터 구조의 정적 분할 가능성 강요를 HTM 에 다시 불러올 수도 있습니다. 미래의 HTM 구현이 진행 보장을 제공한다면 어떤 경우에는 fallback 코드의 필요를 제거할 수도 있으므로 이 한계는 완화될 수도 있겠는데, 이는 결국 HTM 이 높은 충돌 확률을 갖는 환경에서 효율적으로 사용될 수 있게 할 수도 있습니다.

요약하자면, HTM 은 중요한 사용처와 어플리케이션을 가질 수 있을 것 같지만, 이는 병렬 프로그래머의 도구상자의 또 하나의 도구일뿐, 도구상자 전체의 대체제는 아닙니다.

17.3.5 Potential Game Changers

HTM 의 필요를 크게 증가시킬 게임 체인저는 다음을 포함합니다:

1. 진행 보장.
2. 트랜잭션 크기 증가.
3. 디버깅 지원 개선.

¹⁵ NoSQL 데이터베이스가 전통적인 데이터베이스의 엄격한 트랜잭션을 완화시키는 때에 공유 메모리 시스템에 엄격한 트랜잭션 메커니즘이 나오는 건 꽤 아이러닉합니다. 리눅스 커널의 주소 공간 무작위화 방어 메커니즘 [JLK16a, JK16b] 에서의 black-hat 공격에도 불구하고 HTM 은 은 실제로 TM 의 편한 사용성 약속을 알고 있습니다.

Table 17.2: Comparison of Locking (Augmented by RCU or Hazard Pointers) and HTM (**Advantage** , **Disadvantage** , **Strong Disadvantage**)

	Locking with Userspace 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 and overhead 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 bounded wait-free with low overhead. (Lock-free with low overhead for hazard pointers.)	Read-only transactions subject to conflicts and rollbacks. No forward-progress guarantees other than those supplied by fallback code.
Hardware Support	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.	
	Commodity hardware suffices.	New hardware required (and is starting to become available).
Software Support	Performance is insensitive to cache-geometry details.	Performance depends critically on cache geometry.
	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.

Table 17.3: Comparison of Locking (Plain and Augmented) and HTM (Advantage , Disadvantage , Strong Disadvantage)

	Locking	Locking with Userspace RCU or Hazard Pointers	HTM
Basic Idea	Allow only one thread at a time to access a given set of objects.	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 all operations.	Handles revocable operations.
Composability	Limited by deadlock.	Readers limited only by grace-period-wait operations. Updaters limited by deadlock. Readers reduce deadlock.	Irrevocable operations force fallback (typically to locking). 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 lock contention among updaters.	Data must be partitionable to avoid conflicts.
	Partitioning must typically be fixed at design time.	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).
	Locking primitives typically result in expensive cache misses and memory-barrier instructions.	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 and overhead consequences.
	Contention effects are focused on acquisition and release, so that the critical section runs at full speed.	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 bounded wait-free with low overhead. (Lock-free with low overhead for hazard pointers.)	Read-only transactions subject to conflicts and roll-backs. No forward-progress guarantees other than those supplied by fallback code.
	Privatization operations are simple, intuitive, performant, and scalable.	Privatization operations are simple, intuitive, performant, and scalable when data is visible only to updaters.	Privatized data contributes to transaction size.
Hardware Support	Commodity hardware suffices.	Commodity hardware suffices.	New hardware required (and is starting to become available).
	Performance is insensitive to cache-geometry details.	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 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.	Long experience of successful interaction.	Just beginning investigation of interaction.
Practical Apps	Yes.	Yes.	Yes.
Wide Applicability	Yes.	Yes.	Jury still out.

4. 완화된 원자성.

이것들이 다음 섹션에서 다루어집니다.

17.3.5.1 Forward-Progress Guarantees

Section 17.3.2.4 에서 이야기된 것처럼, 현재의 HTM 구현은 진행 보장을 제공하지 않는데, 따라서 HTM 실패를 다루기 위해 fallback 소프트웨어를 필요로 합니다. 물론, 보장을 요구하기는 쉽지만 그걸 제공하기는 항상 쉽지는 않습니다. HTM 의 경우, 보장에 대한 장애물은 캐쉬 크기와 associativity, TLB 크기와 asociativity, 트랜잭션 길이와 인터럽트 빈번도, 그리고 스케줄러 구현이 포함될 수 있습니다.

캐쉬 크기와 associativity 는 Section 17.3.2.1 에서 현재의 한계를 회피하기 위한 일부 연구와 함께 이야기 되었습니다. 하지만, HTM 진행 보장은 크기 제한과 함께 올 것이며, 그게 크더라도 그 한계는 언젠가 닥칠 겁니다. 그런데 왜 현재의 HTM 구현은 작은 트랜잭션에 대해서는 진행 보장을 제공하지 않을까요, 예를 들면 캐쉬의 associativity 로 제한을 걸어서라도? 한가지 잠재적 이유는 하드웨어 실패 처리 필요일 수도 있습니다. 예를 들어, 한 고장난 cache SRAM cell 은 이 고장난 cell 을 비활성화 함으로써 처리될 수 있으며, 따라서 캐쉬의 associativity 를 줄이고 진행 보장이 주어지는 트랜잭션의 최대 크기 또한 줄어듭니다. 이는 보장된 트랜잭션 크기를 줄일 뿐이므로, 다른 이유도 있을 것 같습니다. 아마도 진행 보장을 제품 수준 하드웨어에서 제공하는 것은 사람들이 생각하는 것보다 더 어려울지고 모르고, 소프트웨어에서 진행 보장을 만드는 것의 어려움을 생각하면 완전히 그럴싸한 설명입니다. 문제를 소프트웨어에서 하드웨어로 옮기는 것은 그걸 꼭 쉽게 만드는 것만은 아닙니다 [JSG12].

물리적으로 태깅되고 인덱싱 된 캐쉬에서라면 트랜잭션이 캐쉬에 들어가는 것만으로는 충분치 않습니다. 그것의 주소 변환 또한 TLB 에 들어맞아야만 합니다. 따라서 모든 진행 보장은 TLB 크기와 associativity 도 계산에 넣어야 합니다.

인터럽트, 트랩, 그리고 예외가 현재의 HTM 구현에서는 트랜잭션을 중단시키므로, 트랜잭션의 수행 시간은 예상되는 인터럽트간 시간 간격보다 짧아야 합니다. 어떤 트랜잭션이 얼마나 작은 데이터를 건드는가와 관계 없이, 이게 너무 오래 돌아간다면, 중단당할 겁니다. 따라서, 모든 진행 보장은 트랜잭션 크기만이 아니라 수행시간에도 조건적이 되어야만 합니다.

진행 보장은 여러 충돌하는 트랜잭션 중 어느 것을 중단시킬지 결정하는 능력에 치명적으로 의존합니다. 앞의 트랜잭션을 중단시키고는 나중의 트랜잭션에 스스로가 중단당하는, 따라서 어떤 트랜잭션도 실제로 커밋하지 못하는 끝없는 트랜잭션의 연속을 상상하기는

너무 쉽습니다. 충돌 처리의 복잡성은 많은 HTM 충돌 처리 정책 제안들 [ATC⁺11, LS11] 에 의해 증명되었습니다. 추가적인 복잡도가 Blundell 에 의해 이야기 되었듯 [BLM06] 트랜잭션 외 액세스에 의해 더해집니다. 트랜잭션 외 액세스를 이 모든 문제에 대해 비난하기는 쉬우나, 이런 생각의 어리석음은 트랜잭션 외 액세스 각각을 각자의 단일 액세스 트랜잭션에 위치시킴으로써 쉽게 선보여질 수 있습니다. 문제는 액세스의 패턴이지, 그것이 트랜잭션에 감싸여서 수행되는가가 아닙니다.

마지막으로, 모든 트랜잭션 진행 보장은 트랜잭션을 수행하는 쓰레드가 성공적으로 커밋하기 충분하게 오래 수행되게 허용해야 하는 스케줄러에 의존적입니다.

따라서 HTM 제조사가 진행 보장을 제공하는데에는 상당한 장애물이 존재합니다. 그러나, 그것들 중 하나라도 처리하는 것의 효과는 거대할 겁니다. 이는 HTM 트랜잭션은 더이상 소프트웨어 fallback 을 필요로 하지 않을 것을 의미하는데, 이는 HTM 이 마침내 TM 의 데드락 제거 약속을 지키는 것을 의미합니다.

그러나, 2012년 말, IBM Mainframe 은 일반적인 best-effort HTM 구현 [JSG12] 에 더해 제한된 트랜잭션 을 포함하는 HTM 구현을 발표했습니다. 제한된 트랜잭션은 best-effort 트랜잭션에 사용되었던 tbEGIN 을 대신해 tbEGINC 명령으로 시작합니다. 제한된 트랜잭션은 항상 (결국은) 완료될 것이 보장되어서, 트랜잭션이 중단되면 fallback 경로로 분기하는 대신 (best-effort 트랜잭션에선 이렇게 되었습니다), 하드웨어가 tbEGINC 명령부터 트랜잭션을 재시작 합니다.

Mainframe 설계자는 이 진행 보장을 위한 극한적인 측정을 해야 했습니다. 만약 제한된 트랜잭션이 반복적으로 실패한다면, CPU 는 분기 예측을 비활성화 하고, 순차적 수행을 강제하고, 파이프라이닝을 비활성화하기까지 할 수도 있습니다. 반복된 실패가 높은 경쟁 때문에, CPU 는 예측적 읽기를 비활성화하고, 무작위적 자연을 넣고, 충돌하는 CPU 의 순차적 수행을 시킬 수 조차 있습니다. “흥미로운” 진행 시나리오는 최소 두개의 CPU 로부터 백개의 CPU 까지 포함시킬 수 있습니다. 이 극한의 측정은 왜 다른 CPU 들이 지금껏 제한된 트랜잭션을 제공하는 것을 자제했는지 어떤 통찰을 제공합니다.

이름이 의미하듯, 제한된 트랜잭션은 실제로 상당히 제한되어 있습니다:

1. 최대 데이터 사용량은 메모리의 네개 블록으로, 각 블록은 32 바이트가 최대입니다.
2. 최대 코드 사용량은 256 바이트입니다.
3. 만약 4K 페이지가 제한된 트랜잭션의 코드를 포함한다면, 그 페이지는 그 트랜잭션의 데이터를 포함하지 않습니다.

4. 수행될 수 있는 어셈블리 명령의 최대 수는 32 입니다.

5. 후방으로의 분기는 금지되어 있습니다.

그러나, 이 제약들은 링크드 리스트, 스택, 큐, 그리고 배열 같은 많은 중요한 데이터 구조들을 지원합니다. 따라서 제약된 HTM은 병렬 프로그래머의 도구상자에 중요한 도구가 될 것 같습니다.

이 진행 보장은 절대적일 필요는 없음을 알아두십시오. 예를 들어, 어떤 HTM의 사용이 fallback으로 전역 락을 사용한다고 해봅시다. 이 fallback 메커니즘은 Section 17.3.2.3에서 설명된 “lemming effect”를 막기 위해 조심스레 설계되었다고 가정하면, HTM 되돌리기가 충분히 빈번하지 않을 경우, 이 전역 락은 병목이 되지 않을 겁니다. 그렇다고는 하나, 시스템이 거대할수록, 크리티컬 섹션이 길수록, 그리고 “lemming effect”로부터 회복되는데 필요한 시간이 길수록, “충분히 빈번하지 않음”을 더 드물어져야 할 겁니다.

17.3.5.2 Transaction-Size Increases

진행 보장은 중요하지만, 앞서 봤듯 그건 트랜잭션 크기와 시간에 기반한 조건적 보장일 겁니다. 거기에 일부 진보가 있었는데, 예를 들어 어떤 상용 HTM 구현은 극단적으로 큰 HTM 읽기 집합을 지원하기 위한 추정 기법을 사용합니다 [RD12]. 또 다른 예로, POWER8 HTM은 중단된 트랜잭션을 지원하는데, 이는 관계 없는 액세스를 중단된 트랜잭션의 읽기와 쓰기 집합에 넣는 것을 막습니다 [LGW⁺15]. 이 능력은 고성능 reader-writer 락을 만드는데 사용되었습니다 [FIMR16].

작은 크기에 대한 보장조차도 굉장히 유용할 것을 알아두는 게 중요합니다. 예를 들어, 두 개 캐시 라인에 대한 보장은 스택, 큐, 또는 `dequeue`를 지원하는데 충분합니다. 그러나, 더 큰 데이터 구조는 더 큰 보장을 필요로 하는데, 예를 들어 트리를 순서대로 순회하는 것은 트리의 노드 수만큼의 크기에 대한 보장을 필요로 합니다. 따라서, 보장되는 크기의 약간의 증가라도 HTM의 유용성을 증가시킬 수 있으며, 이는 CPU가 그걸 제공하거나 충분히 좋은 대안을 제공할 필요를 증가시킵니다.

17.3.5.3 Improved Debugging Support

트랜잭션 크기에의 또 다른 억제자는 트랜잭션 디버깅의 필요입니다. 현재 메커니즘의 문제는 단계별 수행 예외가 이를 감싼 트랜잭션을 중단시킨다는 겁니다. 이 문제를 회피하는 여러 방법이 있는데, 프로세서를 에뮬레이션 하는 것 (느려요!), HTM을 대체하는 STM의 사용 (느리고 약간 다른 의미를 가져옵니다!), 진행을 에뮬레이션하기 위한 반복된 재시도를 사용하는 재생

기법 (이상한 실패 모드!), 그리고 HTM 트랜잭션 디버깅의 완전한 지원 (복잡해요!) 등이 포함됩니다.

브레이크포인트, 단계별 수행, 그리고 문자 출력 등을 포함하는 고전적 디버깅 기법의 간단한 사용을 허용하는 HTM 시스템을 만드는 HTM 제조사는 HTM을 훨씬 더 강력하게 만들 겁니다. 일부 트랜잭션 메모리 연구자들은 이 문제를 2013년에 알아차리기 시작했는데, 최소한 하드웨어가 돋는 디버깅 장치를 포함하는 제안 한가지를 냈습니다 [GKP13]. 물론, 이 제안은 사용 가능한 하드웨어가 그런 장치를 얻는 것에 의존적입니다 [Hay20, Int20]. 더 나쁜게, 일부 최첨단 디버깅 기능은 HTM과 호환되지 않습니다 [OHOC20].

17.3.5.4 Weak Atomicity

HTM이 가까운 미래에는 어떤 종류의 크기 제한을 마주하게 될 거라는 점을 놓고 보면, HTM이 다른 메커니즘과 부드럽게 상호작용할 수 있을 필요가 있을 겁니다. HTM의 해저드 포인터나 RCU와 같은 읽기가 대부분인 경우를 위한 메커니즘과의 상호작용은 트랜잭션 외 읽기가 충돌하는 쓰기를 갖는 트랜잭션을 무조건적으로 중단시키지 않는다면—대신, 그 읽기는 간단히 트랜잭션 전의 값을 제공받는다면—개선될 겁니다. 이 방법으로, 해저드 포인터와 RCU는 HTM이 더 큰 데이터 구조를 다룰 수 있으면서 충돌 가능성을 줄일 수 있을 겁니다.

그러나, 이는 꼭 간단하지는 않습니다. 이를 구현하는 가장 간단한 방법은 각 캐시 라인과 버스에 추가적인 상태를 필요로 하는데, 이는 사소하지 않은 비용 추가입니다. 이 비용을 통한 이득은 거대한 메모리 사용량을 갖는 읽기 쓰레드가 계속되는 충돌로 업데이트 쓰레드를 기아에 빼뜨리는 위험을 줄이는 겁니다. Siakavaras 등 [SNGK17]에 의해 이진 검색 트리에 큰 효과를 낸 대안적인 방법은 RCU를 읽기 전용 순회에 사용하고 HTM은 실제 업데이트 자체에만 사용하는 겁니다. 이 조합은 다른 트랜잭션 메모리 기법을 220%, 가량 상회하는데, Howard와 Walpole [HW11]에 의해 관측된 것과 비슷한 속도 향상입니다. 두 경우 모두, 완화된 원자성이 하드웨어가 아닌 소프트웨어에 구현되었습니다. 그러나 하드웨어와 소프트웨어 양쪽에 완화된 원자성이 구현됨으로써 어떤 추가적 속도 향상이 이루어질지 알아보는 것도 흥미로울 겁니다.

17.3.6 Conclusions

현재의 HTM 구현이 일부 상황에 실제 성능 이득을 가져왔지만, 상당한 단점도 가지고 있습니다. 가장 심각한 단점은 제한된 트랜잭션 크기, 충돌 처리의 필요, 중단하고 되돌리기의 필요, 진행 보장의 부재, 되돌릴 수 없는

오퍼레이션의 처리 기능 부재, 그리고 락킹과의 미묘한 의미 차이로 보입니다.

이 단점들 중 일부는 미래의 구현에서 완화될 수 있겠으나, 앞서 언급되었듯 [MMW07, MMTW10] 다른 종류의 여러 동기화 메커니즘들과 HTM이 잘 동작하게끔 만들 필요는 계속해서 클립니다. RCU를 HTM과 함께 사용하는 일부 작업이 있었지만 [SNGK17, SBN⁺20, GGK18, PMDY20], HTM을 향한 진보가 RCU와 다른 deferred-reclamation 메커니즘들과 더 잘 동작한다는 약간의 증거도 있었습니다.

요약하자면, 현재의 HTM 구현은 병렬 프로그래머의 도구상자를 위한 유용한 환영받는 추가물이 될 수 있겠고, 그걸 사용하기 위해 많은 흥미롭고 도전적인 일이 존재할 겁니다. 그러나, 모든 병렬 프로그래밍 문제를 사라지게 할 마법지팡이로 여겨지진 못할 겁니다.

17.4 Formal Regression Testing?

Theory without experiments: Have we gone too far?

Michael Mitzenmacher

정형적 검증은 여러 제품 환경에서 유용한 것으로 증명되었습니다 [LBD⁺04, BBC⁺10, Coo18, SAE⁺18, DFLO19]. 그러나, 리눅스 커널과 같은 복잡한 동시성 코드베이스에 결합된 지속적 통합에 사용되는 자동화된 회귀 테스트 집합에 하드코어 정형 검증이 포함될 수 있기는 할지는 의문입니다. 리눅스 커널 SRCU를 위한 컨셉 증명은 존재하지만 [Roy17], 이 테스트는 가장 간단한 RCU 구현 중 하나의 작은 부분을 위한 것이고, 계속 변화하는 리눅스 커널과 함께 유지하기는 어려운 것으로 증명되었습니다. 따라서 리눅스 커널의 회귀 테스트에 정형 검증을 첫번째 멤버로 포함시키기 위해선 무엇이 필요할지 물어볼 가치가 있습니다.

다음 리스트는 좋은 시작이 될 수 있을 겁니다 [McK15a, slide 34]:

1. 모든 필요한 변환은 자동화 되어야 합니다.
2. 환경은 (메모리 순서 규칙 포함) 올바르게 처리되어야만 합니다.
3. 메모리와 CPU 오버헤드는 받아들여질 수 있을 만큼 적어야 합니다.
4. 버그의 위치를 향하는 구체적 정보가 주어져야 합니다.
5. 소스코드와 입력 이상의 정보의 규모는 작아야만 합니다.

6. 발견된 버그는 코드의 사용자에게 적합해야만 합니다.

이 리스트는 Richard Bornat의 명언에 기반하지만 그보다 더 수수합니다: “정형적 검증 연구자들은 개발자들이 작성하는 코드를 그들이 작성하는 언어로 그것이 수행되는 환경에서 그들이 작성하는 방법으로 검증해야 한다.; 다음 섹션들은 앞의 요구사항 각자를 논하며, 이 필요성에 몇가지 도구들은 얼마나 잘 화답하고 있는지 보이는 섹션이 그를 뒤따릅니다.

17.4.1 Automatic Translation

Promela와 spin은 귀중한 설계상의 도움이 되지만, 여러분이 C 언어 프로그램을 정형적으로 회귀 테스트 하려 한다면 여러분은 여러분이 코드를 다시 검증하고 싶을 때마다 그 코드를 Promela로 직접 변환해야 합니다. 여러분의 코드가 매 60-90 일마다 릴리즈 되는 리눅스 커널이라면, 여러분은 매년 네번에서 여섯번 가량 직접 변환을 해야할 겁니다. 시간이 흐름에 따라, 사람의 오류가 발생할 것인데, 이는 이 검증이 소스코드와 맞지 않음을 의미해서 이 검증을 쓸모없게 만들 겁니다. 반복된 검증은 이 정형적 검증 도구이 여러분의 코드를 직접 입력받게 하거나 버그로부터 자유로운 여러분의 코드를 검증에 필요한 형태로 자동으로 변환하는 도구를 가지게 해야 할 겁니다.

PPCMEM과 herd는 이론상 입력 어셈블리와 C++ 코드를 입력받을 수 있으나, 이 도구들은 매우 작은 리트머스 테스트에만 동작해서, 일반적으로는 여러분이 여러분의 메커니즘의 핵심 부분을 직접 노출시켜야만 합니다. Promela와 spin에서처럼, PPCMEM과 herd는 매우 유용하나, 회귀 테스트에 잘 맞지는 않습니다.

대조적으로, cbmc와 Nidhugg는 합리적 (여전히 상당히 제한되어 있지만) 크기의 C 프로그램을 입력받으며, 그 능력이 계속 성장한다면, 회귀 테스트 도구에 훌륭한 추가품이 될 수 있을 겁니다. Coverity 정적 분석 도구 또한 리눅스 커널을 포함한 상당히 큰 크기의 C 프로그램을 입력으로 받습니다. 물론, Coverity의 정적 분석은 cbmc와 Nidhugg의 것에 비하면 상당히 단순합니다. 다른 한편, Coverity는 특수한 도전을 [BBC⁺10] 갖는 총괄적인 “C 프로그램”에 대한 정의를 가졌습니다. Amazon Web Services는 cbmc를 포함한 다양한 정형적 검증 도구를 사용하며, 이 도구들 중 일부를 회귀 테스팅에 사용합니다 [Coo18]. Google은 상대적으로 간단한 정적 분석 도구를 논란의 여지가 있지만 C 코드베이스보다 덜 다양한 거대한 Java 코드베이스에 사용합니다 [SAE⁺18]. Facebook은 동시성 분석을 포함해 그들의 코드베이스에 보다 공격적인 형태의 정형 검증을 사용합니다만 [DFLO19, O'H19] 아직 리눅스 커널에까지 적용하진 않았습니다. 마지막으로, 마이크로

소프트는 그들의 코드베이스에 정적 분석을 오랜 기간 사용해 왔습니다 [LBD⁺04].

이 리스트를 놓고 보면, 제품 수준 소스 코드를 직접 사용할 수 있는 잘 만들어진 정형적 검증 도구를 만들 수 있을 것이 분명합니다.

그러나, C 코드를 입력으로 받는 것의 한가지 단점은 컴파일러가 올바를 거라 가정한다는 겁니다. 한가지 대안은 C 컴파일러가 생성한 바이너리를 입력으로 받아서 모든 연관된 컴파일러 버그를 처리하게 하는 겁니다. 이 방법은 여러 검증 노력에서 사용되었는데, 가장 두곽을 드러내는 것은 SEL4 프로젝트 [SM13] 일 겁니다.

Quick Quiz 17.15: SEL4 프로젝트에서 사용된 여러 검증 도구의 놀라운 본성을 놓고 보면, 왜 이 챕터는 그걸 더 다루지 않는지 궁금하군요?

그러나, 소스나 바이너리에서 직접 검증을 하는 것은 사람에 의한 변환 과정에서의 오류를 제거하는 장점을 갖는데, 안정적인 회귀 테스팅에 치명적으로 중요합니다.

이는 특수 목적 언어를 사용하는 도구가 쓸모없다는 말이 아닙니다. 그와 반대로, 그것들은 Chapter 12에서 논의 했듯 설계 시점 검증에 매우 도움이 될 수 있습니다. 그러나, 그런 도구는 이 섹션의 주제인 자동화된 회귀 테스트에서 특별히 도움되진 않습니다.

17.4.2 Environment

정형적 검증 도구들이 올바르게 그들의 환경을 모델링하는 것은 치명적으로 중요합니다. 한가지 너무 흔한 누락은 메모리 모델로, Promela/spin을 포함한 수많은 정형적 검증 도구들이 sequential consistency로의 제약을 갖습니다. Section 12.1.4.6와 연관된 QRCU 경험은 중요한 주의를 기울여야 할 이야기입니다.

Promela와 spin은 Chapter 15에서 알아봤듯 현대의 컴퓨터 시스템과는 잘 들어맞지 않는 sequential consistency를 가정합니다. 대조적으로, PPCMEM과 herd의 강력한 장점 중 하나는 x86, Arm, Power, 그리고 herd의 경우 리눅스 커널 버전 v4.17에서 받아들여진 리눅스 커널 메모리 모델 [AMM⁺18]을 포함한 다양한 CPU 제품군들의 메모리 모델에 대한 자세한 모델링입니다.

cmbc와 Nihugg 도구들은 메모리 모델을 선택할 수 있는 어떤 능력을 제공하나, PPCMEM과 herd 만큼의 다양성은 아닙니다. 그러나, 시간이 갈수록 거대 규모 도구들이 상당히 다양한 메모리 모델을 수용할 겁니다.

장기적으로는, 정형적 검증 도구가 I/O를 포함하는 것이 도움이 될테지만 [MDR16] 그러기까진 시간이 좀 걸릴 겁니다.

그렇다고 하나, 환경을 맞추는데 실패하는 도구들도 여전히 유용할 수 있습니다. 예를 들어, 상당히 많은 동

시성 버그가 가상의 sequential consistency 제공 시스템에서도 버그일 것이며, 이 버그들은 이 시스템의 메모리 모델을 sequential consistency로 지나치게 간략화하는 도구에 의해서도 발견될 수 있습니다. 그러나, 이런 도구들은 누락된 메모리 순서 지시어에 연관된 버그들은 찾지 못할 텐데, 이 점은 Section 12.1.4.6의 주의를 요하는 이야기에서 언급되었습니다.

17.4.3 Overhead

모든 하드코어 정형적 검증 도구들은 본성적으로 폭발적인데, 흥미로운 소프트웨어 질문 중 대부분이 실제로 비결정적이라 생각하기 전까지는 실망스러워 보일 겁니다. 그러나, 폭발적인 성질에도 정도의 차이가 있습니다.

PPCMEM은 설계상 최적화되어 있지 않은데, 문제의 메모리 모델이 올바르게 표현되었음을 확실히 보장하기 위함입니다. herd는 Section 12.3에서 언급되었듯 더 적극적인 최적화를 하는데, 따라서 PPCMEM보다 수십 수백배 빠릅니다. 그렇다고 하나, PPCMEM과 herd 둘다 커다란 코드의 몸통보다는 작은 리트머스 테스트를 목표로 합니다.

대조적으로, Promela/spin, cmbc 그리고 Nidhugg는 (어떤) 더 큰 분량의 코드를 위해 설계되었습니다. Promela/spin은 Curiosity rover의 파일시스템을 검증하는데 사용되었고 [GHH⁺14], 앞서 언급되었듯 cmbc와 Nidhugg는 리눅스 커널 RCU에 적용되었습니다.

만약 휴리스틱 상의 발전이 지난 30년간의 속도와 비슷하게 계속된다면 우린 정형적 검증의 오버헤드가 크게 감소할 것을 기대할 수 있습니다. 그러나, 조합상의 폭발 (combinatorial explosion)은 여전히 조합상의 폭발이므로, 휴리스틱의 계속된 개선과 관계없이 검증될 수 있는 프로그램의 크기를 분명하게 제한할 것입니다.

그러나, 조합상의 폭발의 이면은 마케도니아의 Philip II의 조언입니다: “분할하고 정복하라.” 만약 큰 프로그램이 분할될 수 있고 그 조각들이 검증된다면, 그 결과는 조합상의 내파 (implosion) [McK11e]입니다. 분할을 할 자연스러운 장소는 API 경계인데, 예를 들어 락킹 기능의 그것들입니다. 그러면 하나의 검증 경로는 이 락킹 구현이 올바른지를 검증하고, 추가적인 검증 경로는 락킹 API의 올바른 사용을 검증할 수 있습니다.

Table 17.4: Emulating Locking: Performance (s)

# Threads	Locking	cmpxchg_acquire
2	0.004	0.022
3	0.041	0.743
4	0.374	59.565
5	4.905	

Listing 17.2: Emulating Locking with `cmpxchg_acquire()`

```

1 C C-SB+1-o-o-u+1-o-o-u-C
2
3 {}
4
5 P0(int *sl, int *x0, int *x1)
6 {
7     int r2;
8     int r1;
9
10    r2 = cmpxchg_acquire(sl, 0, 1);
11    WRITE_ONCE(*x0, 1);
12    r1 = READ_ONCE(*x1);
13    smp_store_release(sl, 0);
14 }
15
16 P1(int *sl, int *x0, int *x1)
17 {
18     int r2;
19     int r1;
20
21    r2 = cmpxchg_acquire(sl, 0, 1);
22    WRITE_ONCE(*x1, 1);
23    r1 = READ_ONCE(*x0);
24    smp_store_release(sl, 0);
25 }
26
27 filter (0:r2=0 /\ 1:r2=0)
28 exists (0:r1=0 /\ 1:r1=0)

```

이 방법의 성능상 이득은 리눅스 커널 메모리 모델 [AMM⁺18]을 사용해 선보일 수 있습니다. 이 모델은 `spin_lock()`과 `spin_unlock()` 기능을 제공합니다만 이 기능들은 또한 Listing 17.2 (C-SB+1-o-o-u+1-o-o-*u.litmus 와 C-SB+1-o-o-u+1-o-o-u*-C.litmus)에서 보이듯 `cmpxchg_acquire()` 와 `smp_store_release()`를 사용해 에뮬레이션 될 수 있습니다. Table 17.4는 이 모델의 `spin_lock()`과 `spin_unlock()`을 사용하는 것의 성능과 확장성을 이 기능들을 에뮬레이션 하는 것과 비교합니다. 차이는 사소하지 않습니다: 네개의 프로세스에서, 이 모델은 에뮬레이션 보다 수백배 이상 빠릅니다!

Quick Quiz 17.16: Listing 17.2의 라인 27에서는 왜 단순하게 그 조건을 `exists` 절에 추가하는 대신 별도의 `filter` 커맨드를 사용하나요? 그리고 `cmpxchg_acquire()` 대신 `xchg_acquire()`를 사용하는게 더 간단하지 않을까요?

■
도구들이 자동으로 커다란 프로그램을 분할하고 그 조각들을 검증하고 그 조각들의 조합들을 검증할 수 있다면 매우 유용할 겁니다. 그 전까지는, 커다란 프로그램의 검증은 상당한 직접적 개입을 필요로 할 겁니다. 이 개입은 스크립트를 사용하는 것으로 중개되는게 선호될 것이고, 각 릴리즈마다 반복된 검증을 안정적으로 이끌어 갈 수 있을수록 좋을 것이고, 종국적으로는 잘 짜여진 지속적 통합의 형태면 좋을 겁니다. 그리고 Facebook의 Infer 도구는 compositionality와 abstraction

을 통해 [BGOS18, DFLO19] 그걸 위한 중요한 단계를 밟고 있습니다.

어떤 경우든, 우린 정형적 검증 기능이 시간에 따라 증가하길, 그리고 그런 증가가 결국 정형적 검증의 회귀 테스트에서의 적합도를 증가시킬 것을 예상할 수 있습니다.

17.4.4 Locate Bugs

모든 크기의 모든 소프트웨어 작품은 버그를 갖습니다. 따라서, 버그의 존재나 부재만을 알리는 정형적 검증 도구는 특별히 유용하지 않습니다. 필요한 건 그 버그가 어디에 있는지와 그 버그의 본성에 대한 최소한의 어떤 정보를 제공하는 도구입니다.

`cbmc`의 출력은 소스코드로 매핑되는 추적 기록을 포함하는데 Promela/spin과 Nidhugg의 것과 유사합니다. 물론, 이 기록은 무척 길 수 있으며, 그걸 분석하는 건 매우 귀찮을 수 있습니다. 그러나, 그러는게 과거 방법으로 버그를 찾는 것보다는 보통 훨씬 빠르고 즐겁습니다.

또한, 정형적 검증 도구의 가장 간단한 테스트 중 하나는 버그 주입입니다. 어쨌건, 아무나 `printf("VERIFIED\n")`을 작성할 수 있는 건 아니지만, 정형적 검증 도구의 개발자들 역시 나머지 사람들 만큼이나 버그에 취약하다는 게 사실입니다. 따라서, 버그가 존재한다고 주장하기만 하는 정형적 검증 도구는 그걸 실제 세계의 코드에서 검증하기가 더 어려우므로 기본적으로 덜 믿음직합니다.

이 모든 것을 차치하고, 정형적 검증 도구를 작성하는 사람들은 존재하는 도구들을 사용할 수 있습니다. 예를 들어, 심각하고 드문 버그의 존재와 부재를 탐지하도록 설계된 도구는 `bisection`을 도울 수도 있습니다. 만약 이 프로그램의 어떤 과거 버전이 버그를 포함하지 않았지만 새 버전은 포함한다면, `bisection`은 이 버그를 주입한 커밋을 빠르게 찾는데 사용될 수 있으며, 그 정보는 버그를 찾고 고치는데 충분할 수도 있습니다. 물론, 이런 종류의 전략은 흔한 버그에서는 잘 동작하지 않을 텐데, 이 경우 `bisection`은 모든 커밋이 최소 하나의 흔한 버그는 가질 것이기 때문입니다.

따라서, 많은 정형적 검증 도구들에 의해 제공되는 수행 기록은 계속해서 가치있을 것이며, 특히 복잡하고 이해하기 어려운 버그에서 그럴 겁니다. 또한, 최근의 작업은 full-up 정확성 증명을 위해 사용되던 전통적 Hoare 로직을 생각나게 하는 *incorrectness-logic* 정형화를, 그러나 버그를 찾는 목적으로 [O'H19] 적용합니다.

17.4.5 Minimal Scaffolding

과거, 정형적 검증 연구자들은 소프트웨어의 무엇이 검증될 것인지에 대한 전체 명세서를 요구했습니다. 불행히도, 수학적으로 엄격한 명세서는 실제 코드보다 클

수도 있고, 명세의 각 행은 코드의 각 행이 그렇듯 버그를 포함하고 있을 수 있습니다. 코드가 명세를 올바르게 구현했다는 걸 증명하기 위한 정형적 검증 노력은 둘 사이의 버그 대비 버그 비교의 증명이 될텐데, 이는 그다지 도움되지 않을 수 있습니다.

더 나쁜게, 리눅스 커널 RCU 를 포함한 여러 소프트웨어 작품에 대한 그 요구사항은 본성적으로 경험에 편중되어 있습니다 [McK15g, McK15d, McK15e].¹⁶ 이런 흔한 종류의 소프트웨어에서, 완전한 명세는 세련된 허구입니다. 완전한 명세란 2017년 말의 Meltdown 과 Spectre 사이드 채널 공격으로 분명해진 하드웨어에서의 허구보다 덜하지도 않습니다 [Hor18].

이 상황은 실제 세계의 소프트웨어와 하드웨어 작품에 대한 정형적 검증의 희망을 포기하게 할 수도 있겠으나, 할 수 있는게 상당히 있다는 것이 드러났습니다. 예를 들어, 설계와 코딩 규칙은 코드에 포함된 단정문들처럼 부분적 명세로 동작할 수 있습니다. 그리고 실제로 cbmc 와 Nidhugg 같은 정형적 검증 도구는 발동될 수 있는 단정문들을 검사함으로써 암묵적으로 이 단정문들을 명세의 부분으로 취급합니다. 그러나, 단정문들도 코드의 한 부분인데, 이는 특히나 그 코드가 스트레스 테스트에 적합하다면 그게 쓸모없어질 확률을 낮춥니다.¹⁷ cbmc 도구 역시 array-out-of-bound 참조를 검사하며, 따라서 암묵적으로 그걸 명세에 포함시킵니다. 앞서 언급된 비정확성로직 또한 명세를 사용하는 걸로 생각될 수 있습니다 [O'H19].

이 암묵적 명세 접근법은 상당히 말이 되는데, 특히 여러분이 정형적 검증을 완전한 올바름의 증명이 아니라 일반적인 경우보다 다른 강점과 약점을 갖는 검증의 대안적 형태, 즉 테스트로 생각한다면 그렇습니다. 이 관점에서, 소프트웨어는 항상 버그를 가질 것이며, 따라서 버그를 찾는 걸 돋는 모든 종류의 모든 도구는 실제로 아주 좋은 것입니다.

17.4.6 Relevant Bugs

버그를 발견하는 것—그리고 고치는 것—은 물론 모든 종류의 검증 노력의 모든 요점입니다. 분명, 위양성 (false positive) 는 막아져야 합니다. 그러나 위양성의 부재 하에서 조차도, 보기는 존재합니다.

예를 들어, 어떤 소프트웨어 작품이 정확히 100개의 버그를 가지고 있으며, 이것들 각자는 평균적으로 백만년의 수행 시간 중 한번 발생한다고 해봅시다. 더 나아가서 박식한 정형 검증 도구가 개발자들은 마땅히 고쳐야 할 이 모든 100개의 버그를 찾아냈다고 해 봅시다. 이

¹⁶ 또는, 정형적 검증의 말투로 말하자면, 리눅스 커널 RCU 는 불완전한 명세를 가겠습니다.

¹⁷ 그리고 여러분은 여러분의 코드를 스트레스 테스트 합니다. 그렇지 않습니까?

소프트웨어 제품의 안정성에는 무슨 일이 벌어지겠습니까?

답은 안정성의 하락입니다.

이를 보기 위해, 역사적 경험은 약 7% 의 수정이 새로운 버그를 불러들인다고 [BJ12] 함을 기억하십시오. 따라서, 조합된 실패까지의 중간 시간 (meat time to failure: MTBF) 약 10,000년을 갖는 이 100개의 버그를 고치는 것은 일곱개의 버그를 더 불러들입니다. 역사적 통계는 새 버그 각각은 70,000년보다 훨씬 적은 MTBF 를 가질 것이라 말합니다. 이는 결국 이 일곱개의 새 버그의 조합된 MTBF 는 10,000년보다 훨씬 적을 것임을 의미하며, 이는 결국 좋은 의도로 만들어진 원래의 100개의 버그의 수정이 실제로는 전체 소프트웨어의 안정성을 실제로는 하락시켰음을 의미하게 됩니다.

Quick Quiz 17.17: 알려진 버그의 MTBF 들이 아직 발견되지 않은 버그의 MTBF 를 예측하기 좋은 정보임을 어떻게 아나요?

■

Quick Quiz 17.18: 하지만 정형적 검증 도구는 그 수정에 의해 만들어진 버그를 곧바로 찾아낼텐데 왜 이게 문제인가요?

■

더 나쁜게, 평균적으로 매일 한번씩 나타나는 하나의 버그와 백만년마다 나타나는 99개의 버그를 갖는 소프트웨어 작품을 생각해 봅시다. 어떤 정형적 검증 도구가 이 99 백만년짜리 버그를 발견했지만, 이 하루짜리 버그를 찾지 못했다고 해봅시다. 이 99개의 발견된 버그를 고치는데에는 시간과 노력을 필요로 할 것이고 안정성을 떨어뜨리며, 매일 나타나는 버그를 고치기 위해서는 아무것도 하지 않게 될 것인데, 이는 부끄럽고 훨씬 나쁜 일입니다.

따라서, 가장 문제가 되는 버그를 찾는 걸 우선시하는 검증 도구를 갖는게 최선일 겁니다. 그러나, Section 17.4.4 에서 이야기된 바와 같이, 추가적인 도구를 사용하는게 가능합니다. 그런 한가지 강력한 도구는 평범한 과거의 테스트 기법입니다. 버그에 대한 정보를 가졌다면, 이를 위한 구체적 테스트를 만드는게 가능할 것이고, 버그가 드러날 확률을 높이기 위해 Section 11.6.4 에서 설명된 기법들을 사용할 수 있을 겁니다. 이 기법들은 버그의 평소 실패 확률을 대략적으로 추정하는 계산을 가능하게 할 것이고, 이는 결국 버그 수정 노력의 우선순위를 정하는데 사용될 수 있을 겁니다.

Quick Quiz 17.19: 하지만 많은 정형적 검증 도구는 한번에 하나의 버그만 찾을 수 있으므로, 이 도구가 다음 버그를 찾기 전에 각 버그가 고쳐져야만 합니다. 그런 도구가 주어졌을 때 어떻게 버그 수정 노력의 우선순위를 정할 수 있겠습니까?

■

더 적은 수의 *preemption* 이 더 발생 가능성 높다는 합리적 가정 하에 더 적은 *preemption* 을 갖는 수행의 우선순위를 높이는 정형 검증 작업이 최근에 있었습니다.

연관된 버그를 찾는 것은 너무 큰 요구로 들릴 수도 있겠으나, 우리가 소프트웨어의 안정성을 정말로 높이고자 한다면 정말로 필요한 것입니다.

17.4.7 Formal Regression Scorecard

Table 17.5는 이 챕터에서 다루어진 정형 검증 도구들에 대한 대략적인 점수판을 보입니다. 더 짧은 파장이 더 긴 파장보다 낫습니다.

*Promela*는 수동 변환을 필요로 하고 *sequential consistency*만을 지원하므로 첫번째 두 셀이 빨간색입니다. 이 도구는 합리적인 오버헤드를 가졌고(정형 검증을 위해서라면요) 실행 순서 기록을 제공하므로, 다음 두 셀은 노란색입니다. 수동 변환을 필요로 함에도 불구하고, *Promela*는 단정문을 자연스러운 방법으로 처리하므로, 다섯번째 셀은 초록색입니다.

*PPCMEM*은 지원하는 리트머스 테스트의 작은 크기 때문에 보통 수동 변환을 필요로 하므로 첫번째 셀은 오렌지색입니다. 이 도구는 여러 메모리 모델을 처리하므로 두번째 셀은 초록입니다. 이것의 오버헤드는 상당히 높으므로 세번째 셀은 빨간색입니다. 이 도구는 오퍼레이션 간의 관계를 그림으로 표시해 주는데, 이는 실행 순서 기록만큼이나 도움이 되진 않지만 여전히 크게 유용하므로, 네번째 셀은 노란색입니다. 이 도구는 *exists* 절을 작성할 것을 필요로 하고 프로세스간 단정문을 받지 못하므로 다섯번째 셀은 노란색입니다.

*herd*는 *PPCMEM*과 비슷한 크기 제한을 가지므로 *herd*의 첫번째 셀은 역시 오렌지색입니다. 이 도구는 여러 메모리 모델을 지원하므로 두번째 셀은 파랑입니다. 이 도구는 합리적인 오버헤드를 가지므로 세번째 셀은 노란색입니다. 이 도구의 버그 탐지 능력과 단정문 기능은 *PPCMEM*의 그것과 상당히 유사하므로, *herd* 또한 나머지 두 셀에 노란색을 갖습니다.

*cbmc*는 C 코드를 직접 입력받으므로, 첫번째 셀은 파란색입니다. 이 도구는 몇가지 메모리 모델을 지원하므로, 두번째 셀은 노란색입니다. 이 도구는 합리적인 오버헤드를 가지므로 세번째 셀 또한 노란색입니다만, 아마도 SAT-solver 성능은 계속해서 개선될 겁니다. 이 도구는 실행 순서 기록을 제공하므로 네번째 셀은 초록입니다. 이 도구는 C 코드로부터 단정문을 직접 가져오므로, 다섯번째 셀은 파란색입니다.

Nidhugg 또한 C 코드를 직접 입력으로 받으므로, 첫 번째 셀은 파란색입니다. 이 도구는 두개의 메모리 모델만을 지원하므로, 두번째 셀은 오렌지색입니다. 이 도구의 오버헤드는 무척 낮으므로(정형적 검증 치고는), 세번째 셀은 초록색입니다. 이 도구는 실행 순서 기록을 제공하므로 네번째 셀은 초록색입니다. 이 도구는

C 코드로부터 단정문을 직접 받으므로 다섯번째 셀은 파란색입니다.

그런데 여섯번째의 마지막 열은 어떻습니까? 이 도구들 중 어느 것이라도 올바른 버그를 찾는데 어떤지 이야기하기엔 너무 이르므로 모두 물음표와 함께 노란색입니다.

Quick Quiz 17.20: Table 17.5에 보인 점수판에서 테스트는 어떤 모습을 갖게 될까요?

Quick Quiz 17.21: 하지만 Table 17.5에 보인 것보다 훨씬 많은 정형 검증 시스템이 존재하지 않나요?

다시 말하지만, 이 표는 이 도구들을 회귀 테스트에 사용하는 데에 있어서의 평가임을 알아 두시기 바랍니다. 이것들 중 다수가 회귀 테스트에는 잘 맞지 않음은 이것들이 쓸모없음을 말하는게 전혀 아니며, 실제로 이것들 중 많은 것들이 여러 차례 그 가치를 증명했습니다.¹⁸ 단지 회귀 테스트에는 맞지 않다는 겁니다.

그러나, 이 또한 바뀔 겁니다. 무엇보다, 정형 검증 도구들은 2010년대에 인상적인 진보를 했습니다. 이 진보가 지속된다면, 정형 검증은 병렬 프로그래머의 검증 도구상자의 필수적인 도구가 될 겁니다.

17.5 Functional Programming for Parallelism

The curious failure of functional programming for parallel applications.

Malte Skarupke

1980년대 초에 제가 처음으로 함수형 프로그래밍 수업을 들었을 때, 교수님은 부작용으로부터 자유로운 함수형 프로그래밍 스타일은 사소한 병렬화와 분석에 잘 맞는다고 단정하셨습니다. 30년 후, 이 단정은 남아있습니다만, 주류 제작사에서의 병렬 함수형 언어의 사용은 매우 적으며, 이는 교수님의 프로그램은 상태를 유지하지 않고 I/O를 하지 않아야 한다고 하셨던 추가적인 말씀과 관계가 아주 없지는 않을 겁니다. Erlang과 같은 함수형 언어의 틈색에서의 사용이 있고, 여러 다른 함수형 언어에 멀티쓰레드 지원이 추가되었습니다만, 주류 제조 환경은 C, C++, Java, 그리고 Fortran 같은 절차형 언어의 (보통 OpenMP, MPI, 또는 coarrays 와 같은 형태의) 영역으로 남아 있습니다.

¹⁸ 한가지만 예를 들자면, *Promela*는 다른 아닌 Curiosity Rover의 파일시스템을 검증하는데 사용되었습니다. 여러분의 정형 검증 도구는 현재 화성에서 동작하는 소프트웨어에 사용되었습니까???

Table 17.5: Formal Regression Scorecard

	Promela	PPCMEM	herd	cbmc	Nidhugg
(1) Automated	Red	Orange	Orange	Blue	Blue
(2) Environment	(MM)	Green	Blue	(MM)	(MM)
(3) Overhead	Yellow	Red	Yellow	(SAT)	Green
(4) Locate Bugs	Yellow	Yellow	Yellow	Green	Green
(5) Minimal Scaffolding	Green	Yellow	Yellow	Blue	Blue
(6) Relevant Bugs	???	???	???	???	???

이 상황은 자연스레 “분석이 목표라면, 분석 전에 왜 이 절차형 언어를 함수형 언어로 변환하면 어떨까?”라는 질문을 일으키게 합니다. 이 방법에 대한 여러 반대들이 물론 존재하는데, 이 중 세개만 나열하자면:

1. 절차형 언어는 다른 독립된 함수들에 의해, 또는 더 나쁘게도 여러 쓰레드에 의해 수정될 수 있는 전역 변수들을 종종 많이 사용합니다. Haskell의 *monads*는 싱글 쓰레드 전역 상태를 처리하기 위해 발명되었으며, 전역 상태로의 멀티 쓰레드에 의한 액세스는 함수형 모델에 추가적인 위반을 가함을 알아 두시기 바랍니다.
2. 멀티쓰레드 기반 절차형 언어는 종종 락, 어토믹 오퍼레이션, 그리고 트랜잭션 같은 동기화 기능을 사용하는데, 이는 함수형 모델에 위반을 더합니다.
3. 절차형 언어는 함수 인자에 별명을 붙일 (*alias*) 수 있는데, 예를 들어 같은 구조체로의 포인터를 어떤 함수의 같은 호출에 두개의 다른 인자로 넘길 수 있습니다. 이는 이 함수가 이 구조체를 두개의 다른 (그리고 아마도 겹치는) 코드 순서에서 알지 못한 업데이트를 만들 수 있게 하는데, 이는 분석을 무척 복잡하게 만듭니다.

물론, 전역 상태, 동기화 기능, 그리고 aliasing의 중요성을 감안하여 현명한 함수형 프로그래밍 전문가들은 함수형 프로그래밍 모델을 그것들에 조정하기 위한 여러 시도를 제안했으면, monads 그런 것들 중 하나입니다.

또 다른 방법은 병렬 절차적 프로그램을 함수형 그로그램으로 변환하고, 그 결과를 분석하기 위해 함수형 프로그래밍 도구를 사용하는 것입니다. 하지만 이것보다 훨씬 더 잘 할 수 있는데, 모든 실제 계산은 유한한 입력을 갖는 거대한 유한 상태 기계 (finite-state machine) 이기 때문입니다. 이는 모든 실제 프로그램은 하나의 표현으로 변환될 수 있다는 것을 의미하는데, 비실용적으로 큰 것일 수도 있긴 합니다 [DHK12].

하지만, 병렬 알고리즘의 여러 저수준 핵심 부분들이 현대의 컴퓨터의 메모리에 쉽게 들어갈 수 있을 만큼

충분히 작은 표현으로 변환됩니다. 만약 그런 표현이 단정과 결합된다면, 그 단정이 격발될 수 있을지 확인하는 것은 satisfiability 문제가 됩니다. Satisfiability 문제는 NP-complete 하긴 하나, 전체 상태 공간을 생성하는데 필요한 것보다 훨씬 짧은 시간에 종종 해결될 수 있습니다. 또한, 이 해결 시간은 기저의 메모리 모델에 덜 종속적인 것으로 보이므로, 완화된 순서 규칙의 시스템에서 돌아가는 알고리즘도 검사될 수 있습니다 [AKT13].

일반적인 방법은 프로그램을 single-static-assignment (SSA) 형태로 변환하여서 각 변수로의 할당문이 그 변수의 분리된 버전을 만들게 하는 겁니다. 이는 모든 활성화된 쓰레드로부터의 할당문에 적용되어 그 결과 만들어지는 표현은 해당 코드의 모든 가능한 수행을 갖게 됩니다. 할당문을 하나 추가하는 것은 입력과 초기 값의 어떤 조합이 어떤 단정문을 격발시키는 결과를 낳을 수 있는지 질문하게 만드는데, 이는 앞에서도 언급된 것처럼 satisfiability 문제입니다.

한 가지 가능한 반대는 이게 임의의 반복문을 우아하게 처리하지 않는다는 겁니다. 그러나, 많은 경우 이는 그 반복문을 유한한 횟수만큼 unrolling 함으로써 처리될 수 있습니다. 또한, 어떤 반복문은 귀납법으로 풀어질 수 있는 것으로 증명될 겁니다.

또 다른 가능한 반대는 spinlock 이 임의의 긴 반복문을 갖고, 모든 유한 횟수 unrolling 은 이 spinlock 의 전체 행동을 포착하지 못할 거라는 겁니다. 이 반대는 쉽게 극복될 수 있는 것으로 드러났습니다. 전체 spinlock 을 모델링하는 대신, 그 락을 획득하려 시도하는 trylock 을 모델링하고 즉각 그려지 못한다면 중단하는 겁니다. 그러면 이 단정문은 spinlock 이 그 락을 즉각 얻을 수 없기 때문에 중단되었을 때에는 격발되지 않도록 조정되어야만 합니다. 이 논리 표현은 시간에 무관하므로, 모든 가능한 동시성 동작은 이 방법으로 포착될 겁니다.

마지막 반대는 이 기법이 리눅스 커널을 만드는 수백만 행의 코드와 같은 완전한 크기의 소프트웨어 작품은 처리하지 못할 거라는 겁니다. 이는 그럴 수 있는 이야 기이나, 리눅스 커널의 많은 작은 병렬 기능들 각각을 세세히 검증할 수 있다는 것은 상당히 가치있을 겁니다. 그리고 이 방법의 선봉에 선 연구자들은 실제로 리눅스

커널의 Tree RCU 구현을 포함한 실제 세계의 사소하지 않은 코드에 적용했습니다 [LMKM16, KS17a].

이 기법이 얼마나 널리 적용가능할지 두고 봐야겠습니다만, 이는 정형 검증 영역에서의 흥미로운 혁신들 중 하나일 뿐입니다. 병렬 프로그래밍 대변자들은 그들의 피할 수 없는 함수형 프로그래밍의 세계 정복에 대한 단정에 대해 긴 시간 후에는 옳았음으로 드러날 수도 있겠으나, 이 오랜 시간 있어온 방법론은 그것의 정형 검증 영역에서는 믿을 수 있는 결과를 보이기 시작하고 있습니다. 따라서 함수형 프로그래밍의 피할 수 없는 세계 정복에 대한 의심을 계속할 이유가 있습니다.

17.6 Summary

이 챕터는 멀티코어, 트랜잭션 메모리, 회귀 테스트로써의 정형 검증, 그리고 동시적 함수형 프로그래밍을 포함한 가능한 미래들을 빠르게 훑어봤습니다. 이 미래들 중 어느 것이든 실제가 될 수 있습니다만 과거에서와 같이, 미래는 우리가 상상할 수 있는 것보다 훨씬 더 이상할 겁니다.

History is the sum total of things that could have been avoided.

Konrad Adenauer

Chapter 18

Looking Forward and Back

여러분은 이 책의 마지막에 도착했습니다. 수고하셨어요! 여러분의 여정이 즐겁지만 도전적이고 가치있는 것 이었길 바랍니다.

여러분의 편집자와 기여자에게, 이것은 Second Edition으로의 여정의 마지막입니다만, 함께하고자 하는 분들을 위해선 Third Edition으로의 여정의 시작입니다. 어떻든, 이 여정을 정리해 보는게 좋겠습니다.

Chapter 1은 이 책이 무엇에 관한 것인지 다루었고, 저수준 병렬 프로그래밍 외의 관심있을만한 무언가에 대한 대안도 이야기 했습니다.

Chapter 2은 병렬 프로그래밍에서의 도전 과제와 그걸 해결하는 고수준 접근법들을 다뤘습니다. 이 챕터는 또한 여전히 병렬성의 이익을 얻으면서도 이 도전과제들을 피하는 방법들을 다뤘습니다.

Chapter 3는 멀티코어 하드웨어에 대한 높은 단계에서의 개론을 제공했는데, 특히 동시성 소프트웨어에 도전과제를 던져주는 것들에 대해서 그랬습니다. 이 chapter는 이 도전과제들이 기인하는 것들에 대한 책임을 물었는데, 완고한 하드웨어 설계자들 보다는 많은 것이 물리 법칙에 대한 것이었습니다. 그러나, 하드웨어 설계자와 공학자들이 할 수 있을 것들도 있을 수 있으며, chapter가 그것들 약간을 다뤘습니다. 그 사이, 소프트웨어 설계자와 공학자들은 이 도전과제를 극복하기 위한 그들의 작업을 해야만 하는데, 이 책의 나머지 부분에서 이야기 된 것들입니다.

Chapter 4는 저수준 동시성 거래를 위한 도구들에 대해 짧게 알아봤습니다. Chapter 5은 이어서 그런 도구들의 사용법을—그리고, 보다 중요하게, 병렬 프로그래밍 설계 기법의 사용을—간단하지만 놀랍도록 어려운 동시성 카운팅 작업에 대해서—선보였습니다. 실제로 이는 매우 어려워서 여러 동시적 카운팅 알고리즘이 일반적으로 사용되고 있으며, 이것들 각각은 다른 사용처에 특화되어 있습니다.

Chapter 6는 가장 중요한 병렬 프로그래밍 설계 기법인, 문제를 가능한 가장 높은 단계에서 분할하는 방법을

깊이 들여다 보았습니다. 이 chapter는 또한 이 설계 공간에서의 요점 여럿을 살펴봤습니다.

Chapter 7는 병렬 프로그래밍의 일꾼 (그리고 악당)인 락킹을 상세히 알아봤습니다. 이 chapter은 여러 종류의 락킹을 알아보고 널리 알려졌고 공격적으로 알려진 락킹의 단점들을 위한 공학적 해결법들을 선보였습니다.

Chapter 8는 주어진 데이터 항목과 실제 쓰레드의 연관성 의해 동기화가 제공되는 데이터 소유권의 사용을 논했습니다. 적용될 수 있는 곳이라면 이 방법은 상당한 단순성을 가지고 훌륭한 성능과 확장성을 제공합니다.

Chapter 9는 어떻게 약간의 자연이 놀랍도록 많은 경우에 코드를 단순화하면서도 성능과 확장성에 거대한 개선을 가져올 수 있는지 보였습니다. 이 chapter에서 보인 여러 메커니즘들은 CPU 캐시가 읽기 전용 데이터를 복사하는 기능의 장점을 취해서 빛의 속도와 원자의 크기를 크게 제한하는 물리 법칙을 회피합니다. Chapter 10은 길고 영광스런 병렬 프로그램의 역사를 갖는 해쉬 테이블에 집중해서 동시성 데이터 구조를 살펴봤습니다.

Chapter 11은 코드 리뷰와 테스트 방법에 대해 살펴봤으며, Chapter 12는 정형적 검증을 살펴봤습니다. 정형적 검증/테스트의 사이 어디에 여러분이 있든, 코드가 자세히 검증되지 않았다면 이는 동작하지 않습니다. 그리고 이는 동시성 코드에서는 최소 두배는 그립니다.

Chapter 13은 동시성 메커니즘들을 그것들끼리 또는 다른 설계 묘수들과 조합하는 것이 병렬 프로그래머의 삶을 무척 쉽게 해줄 수 있는 여러 상황들을 보였습니다. Chapter 14는 고급 동기화 방법들을 알아봤는데, lockless 프로그래밍, non-blocking synchronization, 그리고 병렬 리얼타임 컴퓨팅을 포함했습니다. Chapter 15는 치명적으로 중요한 주제인 메모리 순서 규칙을 알아봤는데, 여러분이 메모리 순서 규칙 문제를 풀 뿐만이 아니라 그걸 완전히 회피하기 위한 기법과 도구들을 보였습니다. Chapter 16는 놀랍도록 중요한 사용성이라는 주제를 짧게 알아봤습니다.

마지막이지만 절대 빼놓을 수 없는 Chapter 17는 여러 충돌하는 미래에 대한 상상을 알아보았는데, CPU 기술 트렌드, 트랜잭션 메모리, 하드웨어 트랜잭션 메모리, 회귀 테스트에서의 정형적 검증의 사용, 그리고 미래의 병렬 프로그래밍이 함수형 프로그래밍 언어로 이루어질 거라는 오래된 예측을 포함했습니다.

이렇게 이 Second Edition의 내용을 요약했습니다. 그런데 이 책은 어떻게 시작되었을까요?

Paul의 병렬 프로그래밍 여정은 1990년대에 그가 Sequent Computer Systems, Inc에 입사하면서 시작되었습니다. Sequent는 새로 고용된 엔지니어들이 그들을 멘토링하고 코드를 리뷰하고 여러 주제에 대해 풍부한 양의 조언을 주는 경험 많은 엔지니어들로 둘러싸인 작은 방에 위치하게 하는 견습 프로그램 같은 걸 사용했습니다. 그 결과 이 새로 고용된 엔지니어들은 두세 달 만에 생산적인 병렬 프로그래머가 되었고, 그들 중 여럿은 일 년 만에 놀라운 일들을 해냈습니다.

Sequent는 병렬성의 미스테리에서 새로운 엔지니어들을 빠르게 훈련시키는 그들의 능력이 흔하지 않음을 알아냈고, 따라서 그 회사의 병렬 프로그래밍 지혜를 기록한 짧은 책을 만들어 냈는데 [Seq88], 그보다 몇 년 전에 쓰여진 두개의 놀라운 논문을 [BK85, Inm85] 포함했습니다. 이미 이 미스테리에 빠져 있던 사람들은 이 책과 논문들에 경의를 표했으나, 초심자들은 보통 이로 부터 큰 이익을 얻기는 어려워서, 불가피하게도 그 책이나 논문들에 의해 명시적으로 금지되지 않은 고도로 창의적이고 상당히 파괴적인 에러를 만들어 냈습니다.¹ 물론 이 상황은 Paul이 어떤 개선된 책을 쓰는 것을 생각하기 시작하게 했으나, 이 때의 그의 노력은 내부 교육 자료와 출간된 논문들로 제한되어 있었습니다.

1999년에 Sequent가 IBM에 인수되었을 때, 세계의 가장 큰 데이터베이스 인스턴스들은 Sequent 하드웨어에서 돌아가고 있었습니다. 그러나 시간은 바뀌며, 2001년에 이르러선 많은 Sequent의 병렬 프로그래머가 리눅스 커널로 그들의 주의를 돌렸습니다. 초기의 약간의 꺼림 후, 리눅스 커널 커뮤니티는 동시성을 커뮤니티를 통한 많은 훌륭한 혁신과 개선과 함께 효과적이고 열광적으로 받아들였습니다 [BWC^M+10, McK12a]. 책을 써야겠다는 생각은 Paul에게 때때로 떠올랐으나, 삶은 빠르게 흘러갔고, 따라서 그는 이 프로젝트에 진척을 내지 못하고 있었습니다.

2006년, Paul은 리눅스 확장성에 대한 컨퍼런스에 초대받았으며, 존경받는 병렬 프로그래밍 전문가들의 패널에 마지막 질문을 던지는 특권을 얻었습니다. Paul은 1991년부터 2006년의 15년 동안 병렬 시스템의 가격은 집 한채 가격에서 중간대 자전거 정도까지 떨어졌음을, 그리고 다음 15년을 거쳐 2021년에 이르러서는 가격이

더욱 극적으로 떨어질 것이 분명하다고 이야기 하며 시작했습니다. 그는 또한 가격을 떨어뜨리는 것은 병렬 프로그래밍 문제를 해결하는데 상당한 친화성과 빠른 진척을 가져올 것이라고 이야기 했습니다. 이것이 그의 질문을 이끌었습니다: “2021년에도 병렬 프로그래밍이 루틴화 되지 않는다면 그 이유는 뭘까요?”

첫번째 패널리스트는 그런 황당한 질문을 던지는 누구에게나 경멸을 하는 듯 보였으며, 빠르게 짧은 답을 내놓았습니다. 이는 역시 Paul에게 짧은 답을 내놓게 했습니다. 그들은 말을 한동안 주고받았는데, 예를 들어 그 패널리스트의 짧은 답 “데드락”은 Paul의 “락의 존성 검사기”라는 짧은 답을 유발했습니다.

그 패널리스트는 결국 짧은 답이 떨어졌고, 임시변통으로 마지막 답을 내놓았습니다, “당신 같은 사람들은 망치에 머리를 맞아야 한다!”

Paul의 답은 물론 “당신도 거기 줄어야 할겁니다!”였습니다.

Paul은 그의 주의를 다음 패널리스트에게로 돌렸는데, 그는 첫번째 패널리스트에게 어느정도 동의하고 있으며 Paul의 답변들을 처리하고 싶지 않아 하는 듯 보였습니다. 따라서 그는 짧은 이도저도 아닌 말을 했습니다. 그리고 그렇게 나머지 패널들에게도 마이크가 넘어갔습니다.

여러분이 그 이름을 들어봤을지도 모르는 마지막 패널리스트, Linus Torvalds의 차례가 올 때까지는 그랬습니다. Linus는 3년 전 (즉, 2003) 모든 동시성에 연관된 패치들의 첫번째 버전은 보통 무척 조악했으며, 설계상의 취약점과 많은 버그를 가지고 있었다고 이야기 했습니다. 그리고 그게 받아들여지기 충분할 정도로 정리되었을 때까지도 버그들은 여전히 남아있었습니다. Linus는 2006년 기준으로 그때와 지금의 상황을 비교했는데, 그는 잘 설계되었고 버그는 약간만 있거나 아예 없는 동시성 연관 패치의 첫번째 버전은 그리 드물지 않다고 했습니다. 그는 이어서 만약 도구들이 계속해서 개선된다면, 아마도 병렬 프로그래밍은 2021년에 이르러서는 일상적인 것이 될 거라고 제안했습니다.²

그리고는 그 컨퍼런스는 마무리 되었습니다. Paul은 특히나 첫번째 패널리스트가 그랬던 것과 같은 방법으로 세계를 보던 사람들을 포함한 청중들에게 둘러싸였는데 놀라지 않았습니다. Paul은 또한 일부 청중들은 그에게 그 질문을 던져줘 고맙다고 하는데에도 놀라지 않았습니다. 하지만, 어떤 사람이 다가와 말을 하기도 힘들 정도로 흐느끼며 그의 얼굴에 눈물을 흘리며 “고맙습니다”라고 하는데에는 무척 놀랐습니다.

이 사람은 Sequent에서 수년간을 일했으며, 따라서 병렬 프로그래밍을 매우 잘 이해하고 있었습니다. 더 나아가, 그는 현재 병렬 코드를 작성하는 직업을 갖는

¹ “하지만 대체 왜 그걸 한 거죠???” “글쎄요, 안그럴 이유가 뭐죠?”

² 2021년에 이르러 병렬 프로그래밍이 일상화 되지 않았다고 단정하려는 사람은 Chapter 2의 명구를 읽어보셔야 합니다.



Figure 18.1: The Most Important Lesson

그룹에 할당되어 있었습니다. 그리고 그 일은 잘 진행되지 못하고 있었죠. 아시겠죠, 그들이 그의 병렬 프로그래밍에 대한 설명의 이해에 문제를 가지고 있던 게 아닙니다.

문제는 그들의 그의 말을 전혀 들으려 하지 않고 있었다는 겁니다.

그 순간, Paul은 “언젠가는 책을 써야지”에서 “이 책을 쓰기 위한 뭐든 해야겠다”라고 생각을 바꿨습니다. Paul은 이 남자의 이름을 기억하지 못하고 있음을 인정해야 함에 창피함을 느낍니다, 실제로 그의 이름을 알았던 적이 있긴 하다면요.

이 책은 더도 덜도 아니고 그 남자를 위한 겁니다.

그리고 이 책은 또한 그들의 기술 목록에 저수준 동작을 추가하고 싶은 모두를 위한 것이기도 합니다. 여러분이 이 책에서 그 외에는 어떤 것도 기억하지 못한다면, Figure 18.1의 교훈을 생각하십시오.

나머지 우리들을 위해선, 누군가가 우리에게 어려운 문제를 어떻게 해결하는지 보여주려 한다면, 우린 최소한 듣기를 하는 선행을 베풀 수 있을 겁니다.

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 연관된 일부 중요한 질문들을 논합니다. 각 섹션은 또한 연관된 질문에 대해 걱정하는 걸 어떻게 막는지 보이는데, 여러분의 목표가 단지 여러분의 SMP 코드가 가능한한 빨리 고통 없이 동작하게 하는 것이라면—그나저나, 이는 훌륭한 목표죠!—무척 중요할 수 있습니다.

이 질문들에 대한 답은 종종 단일 쓰레드 환경에서의 것에 비해 덜 직관적이지만, 약간의 노력을 한다면 이해하기 그렇게 어렵지 않습니다. 여러분이 재귀를 이해했다면, 여기에 엄청난 도전을 일으키는 건 없습니다.

A.1 What Does “After” Mean?

“나중”은 직관적이지만 놀랍게도 어려운 개념입니다. 중요하고 비직관적인 문제 하나는 코드는 언제든 얼마큼이든 지연될 수 있다는 겁니다. 전역 상태를 타임스탬프 “t”와 정수형 필드 “a”, “b”, 그리고 “c”로 통신하는 생성 쓰레드와 소비 쓰레드를 생각해 봅시다. 생성 쓰레드는 Listing A.1에 보인 것처럼 현재 시간을 (1970년 아래로의 시간을 초 단위로) 기록하고 “a”, “b”, 그리고 “c”를 업데이트 하는 반복문을 수행합니다. 소비 쓰레드는 Listing A.2에 보인 것처럼 역시 현재 시간을 기록하지만 생성 쓰레드의 타임스탬프를 필드 “a”, “b”, 그리고 “c”와 함께 복사하는 반복문을 수행합니다. 수행의 마지막에서, 이 소비 쓰레드는 예를 들면 시간이 뒤로 흐른 것으로 보이는 것 같은 아래적인 기록들의 리스트를 출력합니다.

Quick Quiz A.1: 이 예들에서 어떤 SMP 코딩 오류를 당신은 찾았나요? 전체 코드를 위해선 `time.c`를 보세요.

직관적으로는 생성 쓰레드가 타임스탬프나 값을 기록하는 데에는 시간이 많이 들지 않을 것이므로 생성 쓰레드와 소비 쓰레드 사이의 타임스탬프 차이는 상당히 작을 것이라 예상할 수 있겠습니다. 듀얼코어 1GHz

Listing A.1: “After” Producer Function

```
1 /* WARNING: BUGGY CODE. */
2 void *producer(void *ignored)
3 {
4     int i = 0;
5
6     producer_ready = 1;
7     while (!goflag)
8         sched_yield();
9     while (goflag) {
10         ss.t = dgettimeofday();
11         ss.a = ss.c + 1;
12         ss.b = ss.a + 1;
13         ss.c = ss.b + 1;
14         i++;
15     }
16     printf("producer exiting: %d samples\n", i);
17     producer_done = 1;
18     return (NULL);
19 }
```

x86에서의 샘플 결과가 Table A.1에 보여져 있습니다. 여기서, “seq” 행은 이 반복문을 통과한 횟수이고, “time” 행은 초 단위에서의 이상현상 횟수이며, “delta” 행은 생성 쓰레드의 타임스탬프가 소비 쓰레드의 그것을 뒤따르는 초 수이며 (음수는 소비 쓰레드가 타임스탬프를 생성 쓰레드가 취득하기 전에 취득했음을 의미합니다), “a”, “b”, 그리고 “c”로 라벨링 된 행들은 이 값들이 소비 쓰레드가 얻어온 앞의 스냅샷 대비 얼마나 증가했는지를 보입니다.

Table A.1: “After” Program Sample Output

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

Listing A.2: “After” Consumer Function

```

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 loop, collected %d items %d\n",
37          i, curseq);
38      if (ssc[0].iserror)
39          printf("0/%ld: %.6f %.6f (%.3f) %ld %ld %ld\n",
40              ssc[0].sequence,
41              ssc[j].t, ssc[j].tc,
42              (ssc[j].tc - ssc[j].t) * 1000000,
43              ssc[j].a, ssc[j].b, ssc[j].c);
44      for (j = 0; j <= i; j++)
45          if (ssc[j].iserror)
46              printf("%d/%ld: %.6f (%.3f) %ld %ld %ld\n",
47                  j, ssc[j].sequence,
48                  ssc[j].t, (ssc[j].tc - ssc[j].t) * 1000000,
49                  ssc[j].a - ssc[j - 1].a,
50                  ssc[j].b - ssc[j - 1].b,
51                  ssc[j].c - ssc[j - 1].c);
52      consumer_done = 1;
53  }

```

왜 시간이 거꾸로 흐르는 걸까요? 팔호 안의 수는 마이크로세컨드 단위로, 큰 수 하나는 10 마이크로세컨드를 넘어서며, 어떤건 100 마이크로세컨드까지 넘집니다! 이 CPU는 잠재적으로 그 동안 100,000 개의 인스트러션을 수행할 수 있음을 알아두시기 바랍니다.

한가지 가능한 경우의 수가 다음 사건의 연속 시에 가능합니다:

1. 소비 쓰레드가 타임스탬프를 얻습니다 (Listing A.2, 라인 13).
2. 소비 쓰레드가 preemption 당합니다.
3. 임의의 시간이 흐릅니다.
4. 생성 쓰레드가 타임스탬프를 얻습니다 (Listing A.1, 라인 10).
5. 소비 쓰레드가 수행을 재개하고, 이 생성 쓰레드의 타임스탬프를 가져옵니다 (Listing A.2, 라인 14).

이 시나리오에서, 생성 쓰레드의 타임스탬프는 소비 쓰레드가 타임스탬프를 가져온 시간보다 임의의 양만큼 뒤일 겁니다.

여러분의 SMP 코드에서 “after”의 의미로 괴롭힘 당하는 걸 어떻게 막을까요?

단순히 SMP 기능들을 설계된 대로 사용하면 됩니다. 이 예에서, 가장 쉬운 방법은 락킹을 사용하는 것으로, 예를 들어 Listing A.1의 라인 10 전에 생성 쓰레드에서 락을 잡고 Listing A.2의 라인 13 전에 소비 쓰레드에서 락을 잡는 겁니다. 이 락은 또한 Listing A.1의 라인 13 뒤, 그리고 Listing A.2의 라인 17 뒤에서 해제되어야 합니다. 이 락은 Listing A.1의 라인 10-13와 Listing A.2의 라인 13-17가 서로를 배제하게 하는데, 달리 말하면 각자에 대해 어토믹하게 수행됩니다. 이 점이 Figure A.1에 표현되어 있습니다: 이 락킹은 코드 상자 모두가 시간상 겹쳐지는 것을 막아서 소비 쓰레드의 타임스탬프는 앞의 생성 쓰레드의 타임스탬프 뒤에 얻어져야만 하게 합니다. 이 그림의 각 상자의 코드 조각은 “크리티컬 섹션”이라 명명됩니다; 한번에 그런 크리티컬 섹션 중 오직 하나만이 수행됩니다.

이 락킹의 추가는 Table A.2에 보인 결과를 초래합니다. 여기엔 시간이 뒤로 흐르는 경우는 사라졌고, 그대신 소비 쓰레드에 의한 뒤따르는 읽기 사이에 1,000 회의 추가가 만들어져 있습니다.

Table A.2: Locked “After” Program Sample Output

seq	time (seconds)	delta	a	b	c
58597:	1156521.556296	(3.815)	1485	1485	1485
403927:	1156523.446636	(2.146)	2583	2583	2583

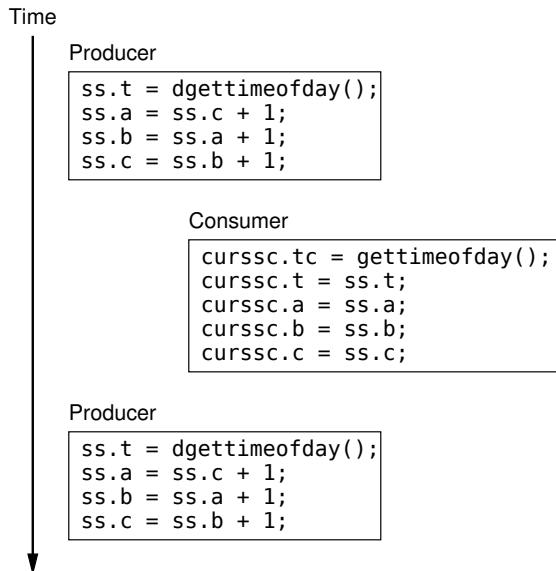


Figure A.1: Effect of Locking on Snapshot Collection

Quick Quiz A.2: 연속된 소비 쓰레드의 읽기 사이에 어떻게 그렇게 큰차이가 발생하죠? 전체 코드를 위해선 `timelocked.c`를 보시기 바랍니다.

요약하자면, 여러분이 배타적 락을 획득했다면, 여러분은 여러분이 그 락을쥔 상태에서 행한 모든 일이 그 락을 전에쥔 쓰레드가 행한 모든 일 뒤에 일어나는 것을 압니다. 최소한 transactional lock elision (Section 17.3.2.6 을 참고하세요) 을 주고받는다면요. 어떤 CPU 가 메모리 배리어를 수행하거나 하지 않았는지 걱정할 필요도, CPU 나 컴파일러의 코드 재배치에 걱정할 필요도 없습니다—삶은 단순합니다. 물론, 이 락킹이 이 두 조각의 코드가 동시에 수행되는 것을 막는다는 것은 이 프로그램의 멀티프로세서에서의 성능 증가를 제한하는, “안전하지만 느린” 상황을 초래할 수 있음을 의미합니다. Chapter 6 는 많은 경우에서 성능과 확장성을 증가시키는 방법들을 설명합니다.

그러나, 대부분의 경우, 스스로가 코드 조각들이 전이나 후에 수행되는지에 대해 걱정하고 있음을 발견한다면, 여러분은 표준 기능의 사용으로 상황을 낫게 하기 위한 힌트를 여기서 얻어야 합니다. 이 기능들이 그 걱정을 여러분 대신 하세요.

A.2 What is the Difference Between “Concurrent” and “Parallel”?

고전적 컴퓨팅 관점에서, “동시 (concurrent)” 와 “병렬 (parallel)” 은 분명 유의어입니다. 그러나, 이는 많은 사람들이 그 둘 사이의 차이를 그리는 걸 멈추게 하지 않았으며, 이 차이는 두개의 다른 관점에서 이해될 수 있는 것으로 드러났습니다.

첫번째 관점은 “병렬” 을 “데이터 병렬” 의 약자로 여기고 “동시”를 그 외의 모든것으로 여기게 하는 겁니다. 이 관점에서, 병렬 컴퓨팅에서 전체 문제의 각 부분은 다른 부분과의 소통 없이 완전히 독립적으로 진행될 수 있습니다. 이 경우, 부분들간의 조정은 조금만, 또는 아예 요구되지 않을 수 있습니다. 대조적으로, 동시 컴퓨팅은 경쟁되는 락, 트랜잭션, 또는 다른 동기화 메커니즘등의 형태로 더 상호 의존성을 가질 수도 있습니다.

Quick Quiz A.3: 프로그램의 한 부분이 RCU read-side 기능을 유일한 동기화 메커니즘으로 사용한다고 해봅시다. 이는 병렬성입니까 동시성입니까?

이는 또한 왜 그런 차이가 중요한지 질문을 던지게 하는데, 이는 아랫단의 스케줄러에 적용되는 두번째 관점을 가져오게 합니다. 스케줄러는 상당한 복잡도와 능력을 가지는데, 대략적인 경험적 규칙으로, 병렬 프로세스들 여럿이 더 긴밀하고 비정규적으로 통신할수록, 스케줄러에 더 높은 수준의 정교성이 필요시 됩니다. 따라서, 병렬 컴퓨팅의 상호의존성 제거는 병렬 컴퓨팅 프로그램이 가장 단순한 스케줄러에서도 잘 동작함을 의미합니다. 실제로, 순수한 병렬 컴퓨팅 프로그램은 임의로 쪼개지고 단일 프로세스에 섞여지더라도 성공적으로 수행될 수 있습니다.¹ 반대로, 동시 컴퓨팅 프로그램은 스케줄러의 그 부분에 상당한 미묘한 트릭을 필요로 하게 할 수 있습니다.

어떤 사람은 단순히 스케줄러로부터의 경쟁의 합리적 수준을 요구함으로써 병렬성과 동시성 사이의 차이를 단순히 무시할 수 있게 해야 한다고 주장할 수 있습니다. 이게 종종 좋은 전략이지만, 효율성, 성능, 그리고 확장성에 대한 염려가 스케줄러가 합리적으로 제공할 수 있는 경쟁의 수준을 크게 제한하는 중요한 환경들이 있습니다. 한가지 중요한 예는 스케줄러가 SIMD 유닛이나 GPGPU 처럼 하드웨어에서 구현되었을 때입니다. 또 다른 예는 일의 단위들이 상당히 짧은 워크로드로, 이 경우 소프트웨어 기반의 스케줄러는 정교성과 효율성 사이에서 힘든 결정을 해야만 합니다.

이제, 이 두번째 관점은 워크로드가 사용 가능한 스케줄러와 맞게끔, 즉 병렬 워크로드가 간단한 스케줄러를

¹ 그래요, 이는 데이터 병렬 컴퓨팅 프로그램은 순차적 수행에 잘 맞음을 의미합니다. 왜 물어보시죠?

사용하고 동시성 워크로드는 정교한 스케줄러를 요구하는 식으로 만드는 것으로 생각될 수 있습니다.

불행히도, 이 관점은 첫번째 관점에 의해 놓여진 종속성 기반 차이와 항상 잘 맞지는 않습니다. 예를 들어, 고도로 상호의존적인 CPU 당 하나의 쓰레드를 갖는 락 기반 워크로드는 스케줄러의 결정이 필요치 않기 때문에 간단한 스케줄러와도 잘 동작할 수 있습니다. 사실, 이런 종류의 일부 워크로드는 순차적 기계에서도 순서대로 수행되는 식으로 돌아갈 수 있습니다. 따라서, 그런 워크로드는 첫번째 관점에 의해선 “동시적”이라 라벨링 되지만 두번째 관점을 취하는 많은 사람들에 의해선 “병렬적”이라 라벨링 됩니다.

Quick Quiz A.4: 두번째 (스케줄러 기반) 관점의 어떤 부분에서 락 기반 CPU 별 싱글쓰레드 워크로드가 “동시적”으로 여겨질 수 있을까요?

■
이건 아무 문제 없습니다. 인간이 작성한 어떤 규칙도 목적 우주에 대해 어떤 가치도 갖지 않습니다, 멀티프로세서 프로그램을 “동시적”과 “병렬적” 같은 카테고리로 나누는 규칙조차도요.

이 구분하려는 시도의 실패는 그런 규칙들이 쓸모없음을 의미하는게 아니라 이를 새로운 환경에 적용하고자 할 때 회의적인 마음자세를 가져야 함을 의미합니다. 항상 그렇듯, 그런 규칙을 적용되는 곳에 사용하고 그렇지 않은 곳에선 무시하세요.

사실, 새 카테고리는 병렬성, 동시성, 맵리듀스, 태스크 기반, 등등으로 더 나타날 겁니다. 어떤 것들은 시간의 시험에 들 것이니, 잘 추측하시기 바랍니다!

A.3 What Time Is It?

멀티코어 컴퓨터 시스템에서의 시간 관리에서의 핵심적 문제가 Figure A.2로 그려져 있습니다. 한가지 문제는 시간을 읽는데 시간이 걸린다는 겁니다. 어떤 명령은 하드웨어 시계를 읽을거고, 이 읽기 오퍼레이션을 완료하기 위해 off-core (더 나쁜 경우 off-socket)으로 나가야 할 수도 있습니다. 읽혀진 값에 대한 어떤 연산을 해야할 수도 있는데, 예를 들어 요구된 포맷으로 변환하기 위해, network time protocol (NTP) 조정을 위해, 등등이 있겠습니다. 그러니 결국 반환된 시간은 초래된 시간 간격의 시작, 끝, 또는 그 사이 어디에 맞춰져 있을까요?

더 나쁜게, 시간을 읽는 쓰레드는 인터럽트 당하거나 preemption 당할 수도 있습니다. 더 나아가, 시간을 읽는 시점과 그 시간의 실제 사용 사이에 어떤 연산이 있을 겁니다. 이 두개의 가능성 모두 이 불확정 시간을 늘립니다.

한가지 방법은 시간을 두번 읽고 타임스탬프가 만들어지는 이 오퍼레이션의 각 방향 중 하나에 있는, 이 두 읽기의 수치적 중간값을 사용하는 것입니다. 그럼

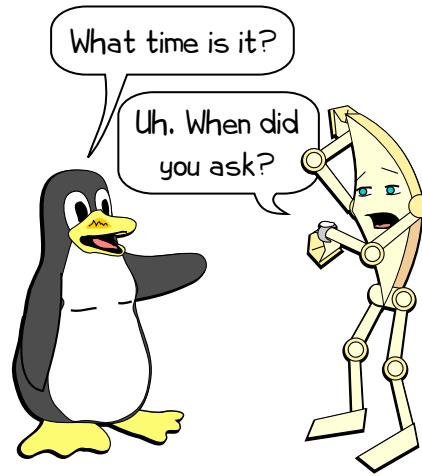


Figure A.2: What Time Is It?

이 두 읽기 사이의 차이는 중간의 오퍼레이션이 수행된 시간의 불확정성의 측정값이 됩니다.

물론, 많은 경우 정확한 시간이 필요합니다. 예를 들어, 인간 사용자의 이익을 위한 시간을 출력하는 경우, 우린 내부의 하드웨어와 소프트웨어 지연이 쓸모없어지는 인간의 느린 반사속도에 기댈 수 있습니다. 비슷하게, 어떤 서버가 클라이언트로의 응답에 시간을 기록해야 한다면, 요청의 도착과 응답의 전송 사이 어느 시간이든 잘 동작할 겁니다.

A.4 How Much Ordering?

얼만큼의 순서가 충분할까요?

여러분은 잘 순서 잡힌 동시적 시스템을 주의 깊게 구축했고 그건 성능도 확장성도 좋지 않을 뿐이란 걸 발견했을 수도 있습니다. 또는 주의는 날려버리고 여러분의 놀랍도록 빠르고 확장성 좋은 소프트웨어는 또한 불안정하다는 것을 발견했을 수도 있습니다. 강력한 안정성과 확장성에 결합된 성능 사이의 행복한 중간은 존재할까요?

정답은, 많은 경우 그렇듯, “경우에 따라 다르다”입니다.

한가지 방법은 강력하게 순서잡힌 시스템을 구축하고, 그것의 성능과 확장성을 조사하는 겁니다. 이게 충분하다면, 이 시스템은 충분히 좋으며, 더이상의 일은 필요 없습니다. 그렇지 않다면, 주의 깊은 조사를 취하고 (Section 11.7를 참고하세요) 시스템의 성능이 충분히 좋아질 때까지 각 병목지점을 격파하는 겁니다.

이 방법은 매우 잘 동작할 수 있는데, 특히 상당한 시스템 전반적 이익을 얻고자 하는 희망으로 시스템의

무작위적인 컴포넌트들을 최적화 하는 너무나도 흔한 방법에 비해 그렇습니다. 그러나, 강력한 순서로부터 시작하는 것은 낭비가 심할 수 있는데, 시스템의 병목의 순서를 완화시키는 것은 그 시스템의 나머지 커다란 부분이 그 완화된 부분에 맞춰 재설계 되고 재작성 되어야하기 때문입니다. 더 나쁜게, 하나의 병목을 제거하는 것은 종종 다른 것을 등장시키는데, 그것 역시 결국 완화되어야 하는 것으로 드러나고, 결국 시스템의 다른 모든 부분들의 재설계와 재작성을 초래하기 때문입니다. 아마도 이보다도 나쁜, 그리고 역시 흔한 방법은 빠르지만 불안정한 시스템으로 시작해서 끝없는 동시성 버그의 반복에 대해 두더지 잡기를 하는 것인데, 이 나중의 경우는 Chapters 11 and 12 이 항상 여러분과 함께하길 겁니다.

시스템의 어떤 부분이 완화된 순서 규칙을 사용할 수 있는지, 그리고 어떤 부분이 완화된 순서로부터 실제 이득을 얻는지 파악하는 설계 시점 도구를 사용하는 게 나을 겁니다. 이 작업들이 다음 섹션들에서 다루어집니다.

A.4.1 Where is the Defining Data?

이를 위한 한가지 방법은 강력한 순서규칙에 의해 발생된 일관성의 영역은 이 시스템의 영역² 바깥으로 확장되지 못한다는 것을 마음에 굳혀두는 겁니다. 바깥 세상의 상태를 추적하기 위한 역할을 하는 시스템의 부분은 보통 완화된 순서규칙을 가질 수 있고, 빛의 속도의 지연은 시스템 내 상태가 바깥 세상의 것을 뒤따르게 강제할 겁니다. 본성적으로 시간에 뒤쳐진 데이터에 대한 일관적 시각을 강제하기 위해 큰 오버헤드를 일으키는 건 많은 경우 의미가 없습니다. 이 경우, Chapter 9 의 방법은 상당히 도움이 될 수 있으며, Chapter 10 에 설명된 데이터 구조들 일부도 마찬가지입니다.

그러나, 그 데이터에 접근하는 것들에게 보이는 의미 있는 어떤 규칙들을 갖는게 현명한데, 예를 들면 특정 함수의 반환값은 다음과 같을 수도 있을 겁니다:

1. 해당 함수 호출 시점에서의 어떤 추상적 값과 그 함수로부터의 리턴 시점에서의 그 추상적 값 사이의 어떤 값. 예를 들어, Section 5.2 에서 이야기 된 통계적 카운터를 그런 카운터는 일반적으로 최소한 연속적인 오버플로 사이에서는 단조증가함을 가슴속에 새기고 보십시오.
2. 해당 함수로의 호출과 리턴 사이의 실제 값. 예를 들어, Listing 5.2 에 보인 단일 변수 어토믹 카운터를 보십시오.
3. 해당 함수에서 사용된 어떤 값이 이 함수의 호출과 리턴 사이 시간 동안 바뀌지 않았다면, 예상된

² 분산 시스템이 될수도 있습니다.

값, 그렇지 않다면 예상된 값에 대한 어떤 근사값. 이 근사치에 대한 정확한 경계의 명세는 상당히 도전적일 수 있습니다. 예를 들어, Section 10.3 에 보인 것처럼 RCU 로 보호되는 연결된 데이터 구조에서의 다른 원소들의 값을 합하는 함수를 생각해 보세요.

요약하자면, 완화된 순서 규칙은 보통 완화된 일관성을 수반하며, 여러분은 어떻게 이 완화가 그들에게 영향을 미치는지에 대한 어떤 약속을 여러분의 사용자에게 제공할 수 있어야 합니다. 동시에, 호출자가 그 함수 호출과 그 함수에 의해 계산된 값의 사용 사이에 띄울 잡고 있지 않았다면, 완전히 순서 잡힌 구현도 일반적으로 앞의 선택사항들에 의해 제공되는 의미 규칙보다 나은 것을 하진 못합니다.

Quick Quiz A.5: 하지만 완전히 순서잡힌 구현이 성능도 확장성도 더 좋은 완화된 순서의 구현보다 더 강한 보장을 제공하지 못한다면, 완전한 순서를 사용할 이유가 뭡니까?



어떤 사람은 유용한 컴퓨팅 처리는 바깥 세계와의 상호작용에서만 일어나며, 따라서 모든 컴퓨팅은 완화된 순서규칙을 사용할 수 있다고 주장할 수도 있겠습니다. 그런 주장은 틀렸습니다. 예를 들어, 은행 계좌의 값은 여러분의 은행의 컴퓨터에 의해 정의되었으며, 살마들은 종종 그들의 계좌 잔고에 연관된 정확한 계산을 선호하는데, 특히 그런 근사화가 그 은행의 선호에 의해 이루어질 거라 의심하는 살마들은 그럴 겁니다.

짧게 말하자면, 외부의 상태를 추적하는 데이터는 완화된 순서규칙 액세스의 매력적인 후보가 될 수 있지만, 정확히 무엇이 추적되고 있으며 무엇이 추적을 하고 있는지에 대해 주의 깊게 생각하십시오.

A.4.2 Consistent Data Used Consistently?

순서규칙 완화가 안전하다는 또 다른 힌트는 띄울 잡고 있는 동안 계산되지만 그 띄이 해제된 후에 사용되는 데이터의 모습에 있습니다. 이 계산된 결과는 그 띄이 해제되자마자 최선의 경우라도 근사화된 값이 되어버리는게 분명한데, 이는 애초에 근사화된 결과를 계산할 것을 생각해 보게 하며, 이는 아마도 완화된 순서 규칙을 허용할 수 있을 겁니다. 이런 이유로, Chapter 5 는 다양한 카운팅의 근사화 방법을 다뤘습니다.

그러나, 상당한 주의가 필요합니다. 띄 해제를 뒤따르는 데이터의 사용이 완화된 순서규칙 최적화가 도움이 될거라는 힌트일까요? 또는, 띄이 너무 빨리 해제되는 버그일까요?

A.4.3 Is the Problem Partitionable?

시스템이 데이터의 정의되는 인스턴스를 잡고 있거나 락 해제 후에 계산된 값을 사용하는 것은 버그라고 증명되었다고 해봅시다. 이제 어떡할까요?

한가지 방법은 Chapter 6에서 이야기 된 것처럼 시스템을 쪼개는 겁니다. 분할하기는 상당한 확장성을 제공할 수 있으며, 그것의 보다 극단적인 형태에서는 Chapter 8에서 이야기된 것처럼 CPU 별 성능이 순차적 프로그램의 그것과도 견주어질 수 있습니다. 부분적 쪼개기는 종종 락킹에 의해 조정되는데, Chapter 7의 주제입니다.

A.4.4 None of the Above?

앞의 세션들은 때로는 완화된 순서규칙을 사용하고 때로는 그렇지 않은 방법으로 성능과 확장성을 높이는 쉬운 방법들을 설명했습니다. 하지만 평범한 사실은 멀티 코어 시스템은 삶을 쉽게 만들어줘야 했다는 회한 따위 갖지 않는다는 겁니다. 그러나 Chapters 14 and 15에서 다루어진 고급 주제들은 도움이 될 수도 있을 겁니다.

하지만 여러분의 코드베이스를 병목이 아닌 것에 대한 최적화를 함으로써 불안정하게 만들기는 너무 쉬우니 주의를 가지고 진행하십시오. 다시 말하지만 Section 11.7이 도움이 될 수 있습니다. 이 책은 여러 교묘한 상황들을 처리하기 위한 많은 정보를 담고 있으니, 이 책의 다른 부분들을 다시 보는 데에도 시간을 쓸 가치가 있을 수도 있습니다.

The only difference between men and boys is the price of their toys.

M. Hébert

Appendix B

“Toy” RCU Implementations

이 부록의 toy RCU 구현은 높은 성능, 실용성, 또는 어떤 종류의 제품 단계에서의 사용을 위해서가 아닌¹ 명료성을 위해 설계되었습니다. 그러나, 여러분은 이 toy RCU 구현을 쉽게 이해하기 위해서라도 Chapters 2, 3, 4, 6, 그리고 9에 대한 깊은 이해를 필요로 할 겁니다.

이 부록은 RCU 구현 시리즈를 존재 보장 문제 해결의 관점에서 정교도를 높여가는 순서로 제공합니다. Appendix B.1은 간단한 락킹에 기반한 간단한 RCU 구현을 보이고, Appendices B.2부터 B.9

에서는 락킹, reference counters, 그리고 free-running 카운터에 기반한 RCU 구현 시리즈를 제공합니다. 마지막으로, Appendix B.10은 요약과 바람직한 RCU 속성의 나열을 제공합니다.

B.1 Lock-Based RCU

Listing B.1: Lock-Based RCU Implementation

```
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 }
```

아마도 가장 간단한 RCU 구현은 Listing B.1 (`rcu_lock.h`와 `rcu_lock.c`)에 보인 것처럼 락을 사용하는 것일 겁니다. 이 구현에서, `rcu_read_lock()`은 전역 `spinlock` 하나를 획득하고 `rcu_read_unlock()`은 이를

¹ 그러나, 제품 품질의 사용자 레벨 RCU 구현이 있습니다 [Des09b, DMS¹²].

해제하며, `synchronize_rcu()`는 이 락을 획득하고는 곧바로 내려놓습니다.

`synchronize_rcu()`는 락을 획득하기 (그리고 내려놓기) 전까지는 리턴하지 않으므로, 앞의 모든 RCU read-side 크리티컬 섹션이 완료되기 전까지는 리턴될 수 없고, 따라서 RCU semantic을 완전히 구현합니다. 물론, 한번에 하나의 RCU 읽기 쓰레드만이 read-side 크리티컬 섹션에 들어가 있을 수 있는데, 이는 RCU의 목적을 완전히 놓치는 것입니다. 또한, `rcu_read_lock()`과 `rcu_read_unlock()`에서의 락 오퍼레이션은 굉장히 무거워서, read-side 오퍼레이션은 한개의 POWER5 CPU에서의 100 나노세컨드에서 64-CPU 시스템에서의 17 마이크로세컨드 까지를 오릅니다. 더 나쁜게, 이 락 오퍼레이션은 `rcu_read_lock()`이 데드락 사이클에 끌 수 있게 합니다. 더 나아가, 회귀적 락의 부재 시, RCU read-side 크리티컬 섹션은 중첩될 수 없고, 마지막으로, 동시의 RCU 업데이트는 흔한 grace period에 의해 완료되겠지만 이 구현은 grace period들을 순차화 시켜서 grace-period 공유를 금지합니다.

Quick Quiz B.1: 왜 Listing B.1에서의 RCU 구현 내의 모든 데드락은 다른 RCU 구현에서도 데드락이지 않을까요?

Quick Quiz B.2: Listing B.1의 RCU 구현에서는 왜 RCU 읽기 쓰레드들이 병렬로 수행될 수 있게끔 단순히 reader-writer 락을 사용하지 않죠?

이 구현이 제품 단계 환경에서 유용할 거라 상상하긴 어렵지만 거의 모든 사용자 단계 어플리케이션에서 구현될 수 있다는 장점을 갖추고 있긴 합니다. 더 나아가, CPU 별 락 또는 reader-writer 락을 사용하는 비슷한 구현이 2.4 리눅스 커널에 의해 제품 단계에서 사용된 바 있습니다.

이 CPU 별 락 방법의 쓰레드 별 락으로 수정된 버전이 다음 섹션에서 설명됩니다.

Listing B.2: Per-Thread 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 }

```

B.2 Per-Thread Lock-Based RCU

Listing B.2 (`rcu_lock_percpu.h` 와 `rcu_lock_percpu.c`) 가 쓰레드별 락에 기반한 구현을 보입니다. `rcu_read_lock()` 과 `rcu_read_unlock()` 함수는 현재 쓰레드의 락을 각각 획득하고 해제합니다. `synchronize_rcu()` 함수는 각 쓰레드의 락을 순서대로 획득하고 해제합니다. 따라서, `synchronize_rcu()`가 시작할 때 수행되고 있던 모든 RCU read-side 크리티컬 섹션은 `synchronize_rcu()` 가 리턴하기 전에 완료되어야만 합니다.

이 구현은 동시의 RCU 읽기 쓰레드를 허용하는 장점을 갖추고 있으며, 단일 전역 락에서는 일어날 수 있는 데드락 조건을 막습니다. 더 나아가, read-side 오버헤드는 약 140 나노세컨드로 높지만, CPU 의 수에 상관 없이 약 140 나노세컨드로 유지됩니다. 그러나, update-side 오버헤드는 단일 POWER5 CPU 에서의 600 나노세컨드에서 64 CPU 에서의 100 마이크로세컨드 까지를 오릅니다.

Quick Quiz B.3: Listing B.2 의 라인 15–18 에서의 반복문에서는 모든 락을 획득한 후 한번에 해제하는게 더 깔끔하지 않을까요? 어쨌건, 이 변경으로 인해 어떤 읽기 쓰레드도 존재하지 않는 시점이 존재하게 되어서 모든 것을 훨씬 간단하게 만들 겁니다.



Quick Quiz B.4: Listing B.2 에 보인 구현은 데드락으로부터 자유로운가요? 이유는 뭐죠?



Quick Quiz B.5: Listing B.2 에 보인 RCU 알고리즘의 한가지 장점은 예를 들면 POSIX pthreads 같은 곳에서 널리 사용 가능한 기능만을 사용한다는 점일까요?

**Listing B.3:** RCU Implementation Using Single Global Reference Counter

```

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 }

```

이 방법은 비슷한 방법이 리눅스 2.4 커널에서 사용되었다는 점을 볼 때 [MM00] 일부 상황에서 유용할 수 있습니다.

이어서 소개될 카운터 기반 RCU 구현은 락 기반 구현의 단점들 일부를 극복합니다.

B.3 Simple Counter-Based RCU

약간 더 진보된 RCU 구현이 Listing B.3 (`rcu_rcg.h` 와 `rcu_rcg.c`)에 보여 있습니다. 이 구현은 라인 1에 정의된 전역 레퍼런스 카운터 `rcu_refcnt`를 사용합니다. `rcu_read_lock()` 기능은 이 카운터를 어토믹하게 증가시키고, RCU read-side 크리티컬 섹션이 이 어토믹 값 증가 뒤로 순서잡혔음을 보장하기 위해 메모리 배리어를 수행합니다. 비슷하게, `rcu_read_unlock()`은 RCU read-side 크리티컬 섹션을 가두기 위해 메모리 배리어를 수행하고 이어서 이 카운터를 어토믹하게 감소시킵니다. `synchronize_rcu()` 기능은 이 레퍼런스 카운터가 0이 되기를 메모리 배리어로 감싸진 가운데에서 기다립니다. 라인 19의 `poll()`은 그저 순수한 지연을 제공하며, 순수한 RCU semantic 관점에서는 없어져도 됩니다. 다시 말하지만, 일단 `synchronize_rcu()` 가 리턴하면, 모든 앞의 RCU read-side 크리티컬 섹션은 완료되었을 것이 보장됩니다.

Appendix B.1에 보인 락 기반 구현과의 행복한 대조로, 이 구현은 RCU read-side 크리티컬 섹션의 병렬 수행을 허용합니다. Appendix B.2에 보인 쓰레드별 락 기반 구현과의 행복한 대조로, 이 구현은 또한 중첩을 허용합니다. 또한, `rcu_read_lock()` 기능은 데드락

사이클에 참여될 수 없는데, spin 도 block 도 하지 않기 때문입니다.

Quick Quiz B.6: 하지만 여러분이 `synchronize_rcu()` 호출 전반에 걸쳐 락을 잡고 있고, 어떤 RCU read-side 크리티컬 섹션 내에서 같은 락을 잡으면 어떻게 될까요?



그러나, 이 구현은 여전히 심각한 단점이 있습니다. 첫째로, `rcu_read_lock()` 과 `rcu_read_unlock()` 의 어토믹 오퍼레이션은 여전히 무거워서, read-side 오버헤드를 단일 POWER5 CPU에서의 100 나노세컨드에서 64-CPU 시스템에서의 40 마이크로세컨드를 오가게 합니다. 이는 RCU read-side 크리티컬 섹션이 실제 read-side 병렬성을 얻기 위해서는 굉장히 길어야 함을 의미합니다. 다른 한편, 읽기 쓰레드가 부재할 경우, grace period 는 40 나노세컨드 가량에 끝나는데, 리눅스 커널의 제품 품질 구현보다 수십수백배 빠릅니다.

Quick Quiz B.7: `synchronize_rcu()` 가 10-밀리세컨드 지연을 갖는데 어떻게 grace period 는 40 나노세컨드에 끝날 수 있죠?



둘째로 수많은 동시의 `rcu_read_lock()` 과 `rcu_read_unlock()` 오퍼레이션이 존재한다면, `rcu_refcnt`에 상당한 메모리 경쟁이 존재할 것이어서 비싼 캐쉬 미스를 초래할 겁니다. 이 두개의 단점은 RCU의 주요 목적인 낮은 오버헤드의 read-side 동기화 기능 제공하기를 크게 해칩니다.

마지막으로, 긴 read-side 크리티컬 섹션을 갖는 큰 수의 RCU 읽기 쓰레드는 `synchronize_rcu()` 가 평생 완료되지 못하게 할 수 있는데, 이 전역 카운터가 평생 0이 되지 않을 수도 있기 때문입니다. 이는 RCU 업데이트 쓰레드의 기아를 초래할 수 있는데, 이 역시 제품 환경에서는 받아들여질 수 없는 일입니다.

Quick Quiz B.8: Listing B.3 의 RCU 구현에서 `synchronize_rcu()` 가 너무 오래 기다리고 있었다면 `rcu_read_lock()` 이 기다리게 하는건 어떤가요? 이게 `synchronize_rcu()` 가 기아에 빠지는 걸 방지하지 않을까요?



따라서, 이 구현 역시 제품 환경에서 유용할 거라 상상하기는 어렵지만 락 기반 메커니즘에 비해서는 예를 들면 높은 부하에서의 디버깅 환경에 적합한 RCU 구현으로는 약간 더 나은 잠재력을 가지고 있습니다. 다음 섹션은 읽기 쓰레드에 더 선호될 레퍼런스 카운팅 방법의 변주를 설명합니다.

Listing B.4: RCU Global Reference-Count Pair Data

```

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);

```

Listing B.5: RCU Read-Side Using Global Reference-Count Pair

```

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 }

```

B.4 Starvation-Free Counter-Based RCU

Listing B.5 (`rcu_rcpg.h`) 가 Listing B.4 에 보인 한 쌍의 레퍼런스 카운터와 (`rcu_refcnt[]`) 이 쌍 중에서 하나의 카운터를 선택하는 전역 인덱스 (`rcu_idx`), 쓰레드별 중첩 카운터 (`rcu_nesting`), 쓰레드별 전역 인덱스의 스냅샷 (`rcu_read_idx`), 그리고 전역 락 (`rcu_gp_lock`) 을 사용하는 RCU 구현의 read-side 기능을 보이고 있습니다.

Design 기아로부터의 자유를 제공하는 건 `rcu_refcnt[]` 배열의 두 원소입니다. 핵심은 `synchronize_rcu()` 가 앞서 존재한 읽기 쓰레드들만을 기다리면 된다는 겁니다. 만약 어떤 새로운 읽기 쓰레드가 특정 `synchronize_rcu()` 호출이 수행을 시작한 후에 시작된다면, 이 `synchronize_rcu()` 수행은 이 새 읽기 쓰레드를 기다리지 않아도 됩니다. 언제든, 특정 읽기 쓰레드가 자신의 RCU read-side 크리티컬 섹션을 `rcu_read_lock()` 을 통해 시작했을 때, 이

쓰레드는 `rcu_idx` 변수로 가리키는 `rcu_refcnt[]` 배열의 원소를 값 증가시킵니다. 이 똑같은 읽기 쓰레드가 `rcu_read_unlock()`을 통해 자신의 RCU read-side 크리티컬 섹션을 종료할 때, 이 쓰레드는 모든 가능한 `rcu_idx` 값으로의 뒤따르는 변화를 무시하고 무엇이든 자신이 증가시킨 원소의 값을 감소시킵니다.

이 조정은 `synchronize_rcu()`는 `rcu_idx`의 값을 `rcu_idx = !rcu_idx`로 조정함으로써 기아를 막을 수 있음을 의미합니다. `rcu_idx`의 기존 값이 0이었고, 따라서 새 값은 1이라고 해봅시다. 이 값 조정 오퍼레이션 뒤에 도착한 새 읽기 쓰레드는 `rcu_refcnt[1]`의 값을 증가시킬 것이며, 앞서 `rcu_refcnt[0]`의 값을 증가시킨 기존 읽기 쓰레드들은 자신의 RCU read-side 크리티컬 섹션을 종료할 때 `rcu_refcnt[0]`의 값을 감소시킬 겁니다. 이는 `rcu_refcnt[0]`의 값이 더이상 증가되지 않을 것이며, 따라서 단조 감소할 것을 의미합니다.² 이는 `synchronize_rcu()`가 해야 할 모든 일은 `rcu_refcnt[0]`의 값이 0이 되길 기다리는 것 뿐임을 의미합니다.

이 배경지식과 함께라면, 실제 기능의 구현을 들여다 볼 준비가 되었습니다.

Implementation `rcu_read_lock()` 기능은 `rcu_idx`에 의해 인덱스되는 `rcu_refcnt[]` 쌍의 멤버를 값 증가시키고, 이 인덱스의 스냅샷을 쓰레드별 변수 `rcu_read_idx`에 보관합니다. `rcu_read_unlock()` 기능은 이어서 그게 무엇이든 연관된 `rcu_read_lock()` 이 값 증가시켰던 카운터의 값을 원자적으로 감소시킵니다. 그러나, 쓰레드당 단 하나의 `rcu_idx` 값만이 기억되었으므로, 중첩을 허용하기 위해선 추가적인 방법이 필요합니다. 이 추가적인 방법은 중첩을 추적하는 쓰레드별 `rcu_nesting` 변수를 사용합니다.

이 모든게 동작하게 하기 위해, Listing B.5의 `rcu_read_lock()`의 라인 6은 현재 쓰레드의 `rcu_nesting`을 가져오고, 라인 7 가 이게 가장 바깥 `rcu_read_lock()`임을 확인하면, 라인 8-10는 `rcu_idx`의 현재 값을 가져오고, 그것을 이 쓰레드의 `rcu_read_idx`에 저장한 후, `rcu_refcnt`의 선택된 원소의 값을 어토믹하게 증가시킵니다. `rcu_nesting`의 값에 관계 없이, 라인 12는 이를 값 증가시킵니다. 라인 13은 이 RCU read-side 크리티컬 섹션의 `rcu_read_lock()` 코드 앞으로 새어나가지 않음을 보장하기 위해 메모리 배리어를 수행합니다.

비슷하게, `rcu_read_unlock()` 함수는 RCU read-side 크리티컬 섹션의 `rcu_read_unlock()` 코드 뒤로 새어나가지 않게끔 라인 21에서 메모리 배리어를 수

² 이 “단조 감소”를 무시하는 경주 조건이 존재합니다. 이 경주 조건은 `synchronize_rcu()` 코드를 통해 처리될 겁니다. 그 전까지는 불신을 미뤄두길 제안합니다.

Listing B.6: RCU Update Using Global Reference-Count Pair

```

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 }
```

행합니다. 라인 22는 이 쓰레드의 `rcu_nesting` 인스턴스를 가져오고, 라인 23에서 이것이 가장 바깥 `rcu_read_unlock()`임을 확인하면, 라인 24와 25에서 이 쓰레드의 `rcu_read_idx` 인스턴스를 가져오고 (이는 가장 바깥의 `rcu_read_lock()`에 의해 저장되었음) `rcu_refcnt`의 선택된 원소의 값을 원자적으로 감소시킵니다. 중첩 단계에 관계 없이, 라인 27는 이 쓰레드의 `rcu_nesting` 인스턴스의 값을 감소시킵니다.

Listing B.6 (`rcu_rcp.c`)는 연관된 `synchronize_rcu()` 구현을 보입니다. 라인 6 와 19은 하나 이상의 동시의 `synchronize_rcu()`를 방지하기 위해 `rcu_gp_lock`을 획득하고 해제합니다. 라인 7 와 8은 각각 `rcu_idx` 값을 가져오고 이를 보상하여 뒤따르는 `rcu_read_lock()` 인스턴스는 앞의 인스턴스와 다른 `rcu_refcnt`를 사용하게끔 만듭니다. 라인 10-12는 이어서 `rcu_refcnt`의 앞의 원소가 0이 되기를 라인 9에서의 `rcu_refcnt`의 검사가 `rcu_idx` 보상을 앞지르게끔 재배치 되지 않게끔 보장하는 메모리 배리어와 함께 기다립니다. 라인 13-18는 이 과정을 반복하며, 라인 20는 모든 뒤따르는 회수 오퍼레이션이 `rcu_refcnt`의 검사를 앞지르게끔 재배치 되지 않게끔 보장합니다.

Quick Quiz B.9: Listing B.6의 `synchronize_rcu()`에서의 라인 5에서의 메모리 배리어는 바로 뒤에 스판락 획득이 있는데 왜 필요하죠?

Quick Quiz B.10: 왜 이 카운터는 Listing B.6에서 두 번 뒤집히나요? 한번 뒤집고 기다리기만으로도 충분하지 않나요?

이 구현은 Listing B.3에 보인 단일 카운터 기반 구현에서 일어날 수 있는 업데이트 starvation 문제를 막습니다.

Discussion 여전히 심각한 단점들이 있습니다. 첫째, `rcu_read_lock()` 과 `rcu_read_unlock()` 의 어토믹 오퍼레이션들은 여전히 꽤 무겁습니다. 실제로, 이건 Listing B.3 에 보인 단일 카운터 변종의 것들보다 더 복잡해서 읽기 쪽 기능들은 단일 POWER5 CPU 에서 약 150 나노세컨드, 64-CPU 시스템에서는 거의 40 마이크로세컨드 의 시간을 소비합니다. 업데이트 쪽 `synchronize_rcu()` 기능은 이보다 더 비싼데, 단일 POWER5 CPU 에서 약 200 나노세컨드에 64-CPU 시스템에서 40 마이크로세컨드 를 소비합니다. 이는 RCU read-side 크리티컬 섹션이 실제 읽기 쪽의 병렬성을 얻기 위해선 굉장히 길어야 함을 의미합니다.

둘째, 많은 동시의 `rcu_read_lock()` 과 `rcu_read_unlock()` 오퍼레이션이 존재한다면, `rcu_refcnt` 원소로의 상당한 메모리 경쟁이 있어서 비싼 캐쉬 미스를 초래합니다. 이는 병렬 read-side 액세스를 제공하기 위해 필요한 RCU read-side 크리티컬 섹션의 시간을 더 늘립니다. 이 두개의 단점들은 RCU 의 목적을 대부분의 상황에서 좌절시킵니다.

셋째, `rcu_idx` 를 두번 뒤집어야 하는 필요성은 업데이트에의 상당한 오버헤드를 부과하며, 특히 큰 수의 쓰레드가 있을때 그렇습니다.

마지막으로, 동시의 RCU 업데이트는 이론상 공통의 grace period 에 의해 처리될 수 있다는 사실에도 불구하고, 이 구현은 grace period 들을 순차화 시켜서 grace-period 공유를 막습니다.

Quick Quiz B.11: 어토믹 값 증가와 감소가 그렇게 비싸면, Listing B.5 의 라인 10 에서 어토믹하지 않은 값 증가를 사용하고 라인 25 에서 어토믹하지 않은 값 감소를 사용하는 건 어떻습니까?

■ 이런 단점들에도 불구하고, 어떤 이는 이 RCU 변종이 작은 단단하게 결합된 멀티프로세서에서, 아마도 더 복잡한 구현과의 API 호환성을 유지하는 메모리 보호 구현으로 사용되는 걸 상상해 볼 수 있을 겁니다. 그러나, 이는 몇 CPU 이상으로는 확장되지 못할 가능성이 큽니다.

다음 섹션은 크게 향상된 read-side 성능과 확장성을 제공하는 레퍼런스 카운팅 방법의 또다른 변종을 설명합니다.

B.5 Scalable Counter-Based RCU

Listing B.8 (`rcu_rcpl.h`) 는 쓰레드별 한쌍의 레퍼런스 카운터를 사용하는 RCU 구현의 read-side 기능들을 보입니다. 이 구현은 Listing B.5 에 보인 구현과 상당히 비슷하지만 유일한 차이점은 `rcu_refcnt` 가 이제 쓰레드별 배열이라는 겁니다 (Listing B.7 에 보인 바와 같습니다). 앞의 섹션에서의 알고리즘과 같이, 이 두

Listing B.7: RCU Per-Thread Reference-Count Pair Data

```
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);
```

Listing B.8: RCU Read-Side Using Per-Thread Reference-Count Pair

```
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 }
```

원소의 배열을 사용하는 것은 읽기 쓰레드가 업데이트 쓰레드를 기아상태에 빠뜨리는 걸 방지합니다. 쓰레드별 `rcu_refcnt[]` 배열의 한가지 장점은 `rcu_read_lock()` 과 `rcu_read_unlock()` 기능이 더이상 어토믹 오퍼레이션을 사용하지 않는다는 겁니다.

Quick Quiz B.12: 나와요! `rcu_read_lock()` 의 `atomic_read()` 가 보인다구요!!! 왜 `rcu_read_lock()` 이 어토믹 오퍼레이션을 가지고 있지 않은 척 하는 거죠???

■ Listing B.9 (`rcu_rcpl.c`) 는 `flip_counter_and_wait()` 라 이름지어진 헬퍼 함수와 함께 `synchronize_rcu()` 의 구현을 보입니다. `synchronize_rcu()` 함수는 Listing B.6 에 보인 것과 닮았지만, 반복되는 카운터 뒤집기가 라인 22 와 23 의 새로운 헬퍼 함수로의 호출 한쌍으로 대체된 게 차이점입니다.

이 새로운 `flip_counter_and_wait()` 함수는 `rcu_idx` 변수를 라인 5 에서 업데이트 하고, 라인 6에서 메모리 배리어를 수행하고, 라인 7-11에서 각 쓰레드의 기존 `rcu_refcnt` 원소에서 스핀하며 이것이 0이 되기

Listing B.9: RCU Update Using Per-Thread Reference-Count Pair

```

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 }

```

를 기다립니다. 일단 모든 원소가 0이 되면 라인 12에서 메모리 배리어를 수행하고 리턴합니다.

이 RCU 구현은 그것의 소프트웨어 환경에 새로운 중요한 요구사항을 부과하는데, 그것은 (1) 쓰레드별 변수를 선언할 수 있어야 할 것, (2) 이 쓰레드별 변수들은 다른 쓰레드에서 액세스 가능해야 할 것, (3) 모든 쓰레드를 순회하는게 가능할 것 입니다. 이 요구사항들은 거의 모든 소프트웨어 환경에서 갖추어질 수 있습니다만, 쓰레드 수의 최대 한계를 종종 고정시킵니다. 더 복잡한 구현은 그런 한계를 막을텐데, 예를 들면 확장 가능한 해쉬 테이블을 쓰는 방식입니다. 그런 구현은 동적으로 쓰레드를 추적할 수도 있을텐데, 예를 들면 `rcu_read_lock()`의 첫번째 호출 때 그것들을 추가시키는 겁니다.

Quick Quiz B.13: 훌륭하군요, 우리가 N 쓰레드를 갖는다면 우린 $2N$ 수십 밀리세컨드 (`flip_counter_and_wait()`) 호출당 한 세트씩, 쓰레드별로 한번씩만 기다린다(가정하더라도) 대기할 수 있군요. 우린 grace period 가 훨씬 더 빨리 끝나길 필요로 하지 않나요?

■
이 구현은 여전히 여러 단점을 가지고 있습니다. 첫째로, `rcu_idx` 를 두번 뒤집어야 하는 필요성은 업데이트에 상당한 오버헤드를 부과하는데, 특히 큰 수의 쓰레드가 있을때 그렇습니다.

둘째로, `synchronize_rcu()` 는 이제 쓰레드의 수와 함께 선형적으로 증가하는 수의 변수들을 검사해야하는데, 큰 수의 쓰레드를 갖는 어플리케이션이 상당한 오버헤드를 부과합니다.

Listing B.10: RCU Read-Side Using Per-Thread Reference-Count Pair and Shared Update Data

```

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);

```

셋째로, 이전과 같이, 동시의 RCU 업데이트는 원칙상 공통의 grace period 에 의해 처리될 수 있어야 하지만, 이 구현은 grace period 들을 순차화 시켜서 grace-period 공유를 막습니다.

마지막으로, 앞서 설명했듯 쓰레드별 변수와 쓰레드들을 순회해야 하는 필요성은 일부 소프트웨어 환경에서는 문제가 될 수 있습니다.

그렇다고는 하나, 이 read-side 기능은 매우 잘 확장되어서, 단일 CPU 나 64-CPU POWER5 시스템에서 수행될 때나 상관 없이 약 115 나노세컨드를 소비합니다. 앞에서 언급되었듯, `synchronize_rcu()` 기능은 확장되지 못하는데, 단일 POWER5 CPU에서의 약 1마이크로세컨드에서부터 64-CPU 시스템에서 약 200 마이크로세컨드의 오버헤드를 갖습니다. 이 구현은 제품 품질 사용자 레벨 RCU 구현의 기초를 형성할 수 있다고 생각될 수 있습니다.

다음 섹션은 더 효율적인 동시의 RCU 업데이트를 허용하는 알고리즘을 설명합니다.

B.6 Scalable Counter-Based RCU With Shared Grace Periods

Listing B.11 (`rcu_rcpls.h`)는 앞에서와 같이 쓰레드별 레퍼런스 카운트 쌍을 이용하지만 업데이트들이 grace period 를 공유하는 걸 허용하는 RCU 구현의 read-side 기능을 보입니다. Listing B.8 에 보인 앞의 구현과의 주요 차이점은 `rcu_idx` 가 이제 `long` 이어서 Listing B.11의 라인 8 는 아래쪽 비트를 마스킹 해 제거해야만 한다는 겁니다. 우린 또한 `atomic_read()` 와 `atomic_set()` 사용을 `READ_ONCE()` 사용으로 바꿨습니다. 데이터는 상당히 유사한데, Listing B.10 에 보였듯 `rcu_idx` 가 `atomic_t` 대신 `long` 이 되었을 뿐입니다.

Listing B.12 (`rcu_rcpls.c`)는 `synchronize_rcu()` 와 그 헬퍼 함수 `flip_counter_and_wait()` 를 보입니다. 이것들은 Listing B.9 의 것들과 비슷합니다. `flip_counter_and_wait()` 의 차이점은 다음을 포함합니다:

1. 라인 6 는 `atomic_set()` 대신 `WRITE_ONCE()` 를 사용하고, 보상하는 대신 값을 증가시킵니다.

Listing B.11: RCU Read-Side Using Per-Thread Reference-Count Pair and Shared Update

```

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 = READ_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 }

```

2. 새로운 라인 7는 이 카운터를 바닥 비트로 마스킹 합니다.

`synchronize_rcu()`로의 변경은 더 폭넓습니다:

- 라인 20에서의 락 획득 전 `rcu_idx` 값을 잡아두는 지역 변수 `oldctr`이 생겼습니다.
- 라인 23는 `atomic_read()` 대신 `READ_ONCE()`를 사용합니다.
- 라인 27-30는 락이 잡혀있는 사이 다른 쓰레드에 의해 최소 세번의 카운터 뒤집기가 행해졌는지 보고, 그렇다면 락을 내려놓고, 메모리 배리어를 수행하고, 리턴합니다. 이 경우, 카운터가 0이 되기 까지의 완전한 기다림이 두번 있으므로 다른 쓰레드들은 필요한 일을 이미 다 끝냈을 겁니다.
- 라인 33-34에서, `flip_counter_and_wait()`는 락이 획득된 사이 두번 미만의 카운터 뒤집기가 행해졌다면 두번째에만 수행됩니다. 다른 한편, 두번의 카운터 뒤집기가 있었다면, 어떤 다른 쓰레드가 모든 카운터가 0이 되기를 완전히 한번 기다렸으므로, 한번만 더 기다리면 됩니다.

이 방법과 함께, 임의의 큰 수의 쓰레드가 쓰레드당 CPU 하나를 가진 채 동시에 `synchronize_rcu()`를 수행했다면, 카운터가 0이 되기를 기다리는 횟수는 세 번만이 될겁니다.

Listing B.12: RCU Shared Update Using Per-Thread Reference-Count Pair

```

1 static void flip_counter_and_wait(int ctr)
2 {
3     int i;
4     int t;
5
6     WRITE_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 = READ_ONCE(rcu_idx);
24     smp_mb();
25     spin_lock(&rcu_gp_lock);
26     ctr = READ_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 }

```

이 개선에도 불구하고, 이 RCU 구현은 여전히 단점이 여럿 있습니다. 첫째로, 이전과 같이 `rcu_idx`를 두번 뒤집어야 하는 필요성은 업데이트에 상당한 오버헤드를 부과하는데, 특히 큰 수의 쓰레드가 있을 때 그렇습니다.

둘째, 각 업데이트 쓰레드는 여전히 `rcu_gp_lock`을 획득해야만 하며, 해야 할 일이 없을 때조차도 그렇습니다. 이는 큰 수의 동시 업데이트가 존재하는 경우 상당한 확장성 한계를 초래할 수 있습니다. 리눅스 커널의 제품 품질 리얼타임 RCU 구현에서 이루어졌던 것처럼 [McK07a] 이를 막는 방법들이 있습니다.

셋째, 이 구현은 쓰레드별 변수와 쓰레드를 순회하는 기능을 필요로하는데, 다시 말하지만 어떤 소프트웨어 환경에서는 문제가 될 수 있습니다.

마지막으로, 32-bit 기계에서는 `rcu_idx` 카운터가 오버플로우되기까지 긴 시간 업데이트 쓰레드가 preemption 당해 있을 수도 있습니다. 이는 그런 쓰레드가 불필요한 카운터 뒤집기 두번을 강제하게 할 수 있습니다. 그러나, 각 grace period가 1マイ크로세컨드만 소비했다고 하더라도, 공격하는 쓰레드는 한시간 이상을 preemption 당해야 할 것인데, 이 경우 추가적인 한쌍의

Listing B.13: Data for Free-Running Counter Using RCU

```

1 DEFINE_SPINLOCK(rcu_gp_lock);
2 long rcu_gp_ctr = 0;
3 DEFINE_PER_THREAD(long, rcu_reader_gp);
4 DEFINE_PER_THREAD(long, rcu_reader_gp_snap);

```

카운터 뒤집기가 여러분의 걱정거리 중 최소한의 것임을 겁니다.

Appendix B.3에서 설명한 구현에서와 같이, 이 read-side 기능은 매우 잘 확장하여, CPU의 수에 관계 없이 약 115 나노세컨드의 오버헤드만을 일으킵니다. `synchronize_rcu()` 기능은 여전히 비용이 높아서, 1 마이크로세컨드에서 16 마이크로세컨드까지의 오버헤드를 초래합니다. 그렇다고는 하나 이는 Appendix B.5의 구현에서 발생하는 약 200 마이크로세컨드에 비하면 훨씬 낮은 것입니다. 따라서, 그 단점에도 불구하고, 어떤 사람들은 이 RCU 구현이 실제 삶의 제품 단계 어플리케이션에서 사용되는 걸 상상해 볼 수 있을 겁니다.

Quick Quiz B.14: 이 모든 toy RCU 구현은 `rcu_read_lock()`과 `rcu_read_unlock()`, 또는 `synchronize_rcu()`에서 어토믹 오퍼레이션을 가지고 있어서 쓰레드의 수에 따라 오버헤드가 선형적으로 증가합니다. 어떤 환경에서 어떤 RCU 구현은 이 세 개의 모든 기능이 결정적인 $O(1)$ 오버헤드와 응답시간을 제공하는 가벼운 구현을 즐길 수 있을까요?

■ Listing B.11를 다시 참고해보면, 우린 전역 변수로의 한번의 액세스가 있고 쓰레드 지역 변수로의 네번 이상의 액세스가 있음을 볼 수 있습니다. POSIX 쓰레드를 구현하는 시스템에서의 쓰레드 지역 액세스의 상대적으로 높은 비용을 놓고 생각해 보면, 이 세 개의 쓰레드 지역 변수들을 하나의 구조체로 만들어 `rcu_read_lock()`과 `rcu_read_unlock()`이 각자의 쓰레드 지역 데이터를 하나의 쓰레드 지역 저장소 액세스로 액세스 할 수 있게 하고 싶을 겁니다. 그러나, 그보다도 나은 방법은 쓰레드 지역 액세스를 한번으로 줄이는 것일텐데, 이를 다음 섹션에서 합니다.

B.7 RCU Based on Free-Running Counter

Listing B.14 (`rcu.h` and `rcu.c`)는 Listing B.13에 보인 데이터와 함께 짹수 값만을 취하는 전역 free-running 카운터 하나에 기반한 RCU 구현을 보입니다.

이로 인한 `rcu_read_lock()` 구현은 굉장히 간단합니다. 라인 3와 4는 전역 free-running `rcu_gp_ctr` 변수에 1을 더하고 그 결과 만들어지는 훌수 값을 쓰레드별 변수 `rcu_reader_gp`에 저장합니다. 라인 5는

Listing B.14: Free-Running Counter Using RCU

```

1 static inline void rcu_read_lock(void)
2 {
3     __get_thread_var(rcu_reader_gp) =
4         READ_ONCE(rcu_gp_ctr) + 1;
5     smp_mb();
6 }
7
8 static inline void rcu_read_unlock(void)
9 {
10    smp_mb();
11    __get_thread_var(rcu_reader_gp) =
12        READ_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     WRITE_ONCE(rcu_gp_ctr, 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                  rcu_gp_ctr) < 0)) {
27             poll(NULL, 0, 10);
28         }
29     }
30     spin_unlock(&rcu_gp_lock);
31     smp_mb();
32 }

```

뒤따르는 RCU read-side 크리티컬 섹션의 내용이 “밖으로 새어나가는” 것을 방지하기 위해 메모리 배리어를 수행합니다.

`rcu_read_unlock()` 구현도 비슷합니다. 라인 10은 다시 앞의 RCU read-side 크리티컬 섹션이 “새어 나오는” 것을 막기 위해 메모리 배리어를 수행합니다. 라인 11과 12는 이어서 `rcu_gp_ctr` 전역 변수를 쓰레드별 변수 `rcu_reader_gp`에 복사하고, 이 쓰레드별 변수를 짹수 값으로 남겨두어 동시에 `synchronize_rcu()`가 이를 알고 무시할 수 있게 해줍니다.

Quick Quiz B.15: 어떤 짹수 값이든 `synchronize_rcu()`에게 그 테스크를 무시하라고 말하기 충분하다면, 왜 Listing B.14의 라인 11과 12는 간단히 `rcu_reader_gp`에 0을 할당하지 않나요?

■ 따라서, `synchronize_rcu()`는 모든 쓰레드별 `rcu_reader_gp` 변수가 짹수 값을 가지길 기다릴 수 있습니다. 그러나, `synchronize_rcu()`는 앞에서부터 존재해온 RCU read-side 크리티컬 섹션만을 기다리면 되기 때문에 이보다 훨씬 더 잘할수 있습니다. 라인 19은 앞의 RCU로 보호되는 데이터 구조의 조정이 라인 21 뒤로 순서 재배치 되는 걸 (CPU에 의해서든 컴파일러에 의해서든) 막기 위해 메모리 배리어를 수행합니다. 라인 20은 여러 `synchronize_rcu()` 인스턴스가 동시에 수행되는 것을 막기 위해 `rcu_gp_lock`을 획득합니다

(그리고 라인 30에서 이를 해제합니다). 라인 21는 이어서 전역 `rcu_gp_ctrl` 변수를 2증가시켜서, 모든 앞서서부터 존재한 RCU read-side 크리티컬 섹션이 연관된 쓰레드별 `rcu_reader_gp` 변수에 `rcu_gp_ctrl`의 것에 해당 기계의 워드 크기로 modulo 연산한 값보다 작은 값을 갖게 합니다. 또한 `rcu_reader_gp`의 짹수 값의 쓰레드는 RCU read-side 크리티컬 섹션에 있지 않으므로, 라인 23-29는 `rcu_reader_gp` 값을 그 값이 모두 짹수이거나 (라인 24) 전역 `rcu_gp_ctrl` (라인 25-26) 보다 큰 값일 때까지 스캐닝 합니다. 라인 27은 앞서서부터 존재한 RCU read-side 크리티컬 섹션을 기다리기 위해 짧은 시간동안 블록됩니다만, 이는 grace-period 응답시간이 중요하다면 spin-반복문으로 대체될 수 있습니다. 마지막으로, 라인 31에서의 메모리 배리어는 모든 뒤따르는 제거가 앞의 반복문 안으로 재배치 되지 않게 보장합니다.

Quick Quiz B.16: Listing B.14의 라인 19와 31의 메모리 배리어는 왜 필요하죠? 라인 20와 30의 락킹 기능에 내재된 메모리 배리어로 충분하지 않나요?

■ 이 방법은 훨씬 나은 read-side 성능을 달성해서, POWER5 CPU의 갯수에 관계 없이 약 63나노세컨드의 오버헤드를 일으킵니다. 업데이트는 더 큰 오버헤드를 일으키는데, 단일 POWER5 CPU에서의 500나노세컨드에서 64개의 CPU에서의 100마이크로세컨드를 오릅니다.

Quick Quiz B.17: Appendix B.6에 설명된 업데이트 쪽 batching 최적화는 Listing B.14에 보인 구현에 적용될 수 없나요?

■ 이 구현은 앞서 언급된 업데이트 쪽의 높은 오버헤드에 더해 일부 심각한 단점을 가지고 있습니다. 첫째, 더이상 RCU read-side 크리티컬 섹션의 중첩을 허용하지 않는다는, 다음 섹션의 주제입니다. 둘째, 어떤 읽기 쓰레드가 Listing B.14의 라인 3에서 `rcu_gp_ctrl`에서 읽기를 했지만 `rcu_reader_gp`에서 저장을 하기 전에 preemption 당한다면, 그리고 `rcu_gp_ctrl` 카운터가 그것의 가능한 값들의 전부는 아니라도 절반 이상을 세어버린다면, `synchronize_rcu()`는 뒤따르는 RCU read-side 크리티컬 섹션을 무시할 겁니다. 마지막이자셋째, 이 구현은 소프트웨어 환경이 쓰레드를 순회할 수 있고 쓰레드별 변수를 유지할 수 있게 해야 합니다.

Quick Quiz B.18: Listing B.14의 라인 3-4에서 읽기 쓰레드가 preemption 될 수 있다는 가능성이 잔짜로 문제인가요? 달리 말하자면, 문제를 일으킬 수 있는 실제 이벤트 순서가 존재합니까? 아니라면 왜 그렇습니까? 그렇다면 그 이벤트 순서는 무엇이며 어떻게 그 문제가 처리될 수 있습니까?

Listing B.15: Data for Nestable RCU Using a Free-Running Counter

```

1 DEFINE_SPINLOCK(rcu_gp_lock);
2 #define RCU_GP_CTR_SHIFT 7
3 #define RCU_GP_CTR_BOTTOM_BIT (1 << RCU_GP_CTR_SHIFT)
4 #define RCU_GP_CTR_NEST_MASK (RCU_GP_CTR_BOTTOM_BIT - 1)
5 #define MAX_GP_ADV_DISTANCE (RCU_GP_CTR_NEST_MASK << 8)
6 unsigned long rcu_gp_ctrl = 0;
7 DEFINE_PER_THREAD(unsigned long, rcu_reader_gp);

```

B.8 Nestable RCU Based on Free-Running Counter

Listing B.16(`rcu_nest.h`와 `rcu_nest.c`)는 단일 전역 free-running 카운터에 기반하나 RCU read-side 크리티컬 섹션의 중첩을 허용하는 RCU 구현을 보입니다. 이 중첩가능성은 Listing B.15에 보인 정의를 이용해 전역 `rcu_gp_ctrl`의 아래쪽 비트를 중첩 정도 세기를 위해 예약함으로써 이루어집니다. 이는 아래쪽의 한개 비트를 중첩 깊이를 세는데 예약해두는 것으로 생각될 수 있는, Appendix B.7의 방법의 범용화입니다. 이를 위해 두개의 C 전처리기 매크로 `RCU_GP_CTR_NEST_MASK`와 `RCU_GP_CTR_BOTTOM_BIT`이 사용됩니다. 이것들은 연관되어 있습니다: `RCU_GP_CTR_NEST_MASK=RCU_GP_CTR_BOTTOM_BIT-1`. 이 `RCU_GP_CTR_BOTTOM_BIT` 매크로는 중첩을 세기 위해 예약된 비트들의 바로 앞에 위치하는 단일 비트를 담고 있고, `RCU_GP_CTR_NEST_MASK`는 중첩을 세는데 사용되는 `rcu_gp_ctrl`의 영역을 덮는 모든 1비트들을 갖습니다. 분명, 이 두개의 C 전처리기 매크로는 최대 필요한 RCU read-side 크리티컬 섹션 중첩을 허용하기 충분한 아래쪽 비트를 예약해야 하며, 이 구현은 최대 RCU read-side 크리티컬 섹션 중첩 깊이 127을 위해 일곱 비트를 예약하는데, 대부분의 어플리케이션이 필요한 정도를 넘어서설 겁니다.

그 결과로 나오는 `rcu_read_lock()` 구현은 여전히 합리적 수준으로 간단합니다. 라인 6은 이 쓰레드의 `rcu_reader_gp`로의 포인터를 지역 변수 `rrgp`에 위치시켜서 비싼 pthreads thread-local-state API로의 호출 횟수를 최소화 합니다. 라인 7은 `rcu_reader_gp`의 현재 값을 또 다른 지역 변수 `tmp`에 저장하며, 라인 8는 이 아래쪽 비트들이 0인지, 즉 이게 가장 바깥 `rcu_read_lock`인지, 검사합니다. 그렇다면 라인 9는 전역 `rcu_gp_ctrl`을 `tmp`에 위치시키는데, 라인 7에 의해 앞서 읽은 현재값은 더이상 유효하지 않을 수 있기 때문입니다. 어느 경우든, 라인 10는 이 중첩 깊이를 증가시키는데, 즉 여러분이 기억해야 할 것은 이제 이 카운터의 아래쪽 7개 비트에 저장되어 있습니다. 라인 11은 업데이트된 카운터를 이 쓰레드의 `rcu_reader_gp`에 저장하고, 마지막으로 라인 12은 이 RCU read-side 크리티컬 섹션이

Listing B.16: Nestable RCU Using a Free-Running Counter

```

1 static void rcu_read_lock(void)
2 {
3     unsigned long tmp;
4     unsigned 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 = READ_ONCE(rcu_gp_ctrl);
10    tmp++;
11    WRITE_ONCE(*rrgp, tmp);
12    smp_mb();
13 }
14
15 static void rcu_read_unlock(void)
16 {
17     smp_mb();
18     __get_thread_var(rcu_reader_gp)--;
19 }
20
21 void synchronize_rcu(void)
22 {
23     int t;
24
25     smp_mb();
26     spin_lock(&rcu_gp_lock);
27     WRITE_ONCE(rcu_gp_ctrl, rcu_gp_ctrl +
28                 RCU_GP_CTR_BOTTOM_BIT);
29     smp_mb();
30     for_each_thread(t) {
31         while (rcu_gp_onGoing(t) &&
32                 ((READ_ONCE(per_thread(rcu_reader_gp, t)) -
33                  rcu_gp_ctrl) < 0)) {
34             poll(NULL, 0, 10);
35         }
36     }
37     spin_unlock(&rcu_gp_lock);
38     smp_mb();
39 }

```

`rcu_read_lock()` 호출을 앞서는 코드로 새어나가는 걸 막기 위해 메모리 배리어를 수행합니다.

달리 말하자면, 이 `rcu_read_lock()` 구현은 현재 `rcu_read_lock()` 이 다른 RCU read-side 크리티컬 섹션의 안에 중첩되어 있지 않은 전역 `rcu_gp_ctrl`의 복사본을 가져오고, 중첩되어 있다면 현재 쓰레드의 `rcu_reader_gp` 인스턴스의 내용을 가져온다는 겁니다. 어느 쪽이던, 추가적인 중첩 수준을 기록하기 위해 읽어온 값을 증가시키고, 그 결과를 현재 쓰레드의 `rcu_reader_gp` 인스턴스에 저장합니다.

흥미롭게도, `rcu_read_lock()`의 차이에도 불구하고 `rcu_read_unlock()` 구현은 Appendix B.7에 보인 것과 크게 유사합니다. 라인 17은 RCU read-side 크리티컬 섹션의 `rcu_read_unlock()`을 뒤따르는 코드로 새어나가는 걸 막기 위해 메모리 배리어를 수행하고, 라인 18은 이 쓰레드의 `rcu_reader_gp` 인스턴스의 값을 감소시키는데, 이는 `rcu_reader_gp`의 아래쪽 비트에 저장된 중첩 횟수를 감소시키는 효과를 낳습니다. 이 기능의 디버깅 버전은 이 아래쪽 비트들이 0이 아니었는지(값 감소 전에!) 검사할 겁니다.

`synchronize_rcu()`의 구현은 Appendix B.7에 보인 것과 상당히 유사합니다. 두개의 차이점이 있습니다. 첫째는 라인 27와 28가 전역 `rcu_gp_ctrl`에 상수 “2” 대신 `RCU_GP_CTR_BOTTOM_BIT`을 더한다는 것이고, 두번째는 라인 31에서의 비교가 별개 함수로 추상화 되었으며, 그 함수는 무조건적으로 아래쪽 비트를 검사하는게 아니라 `RCU_GP_CTR_BOTTOM_BIT`에 의해 가리켜지는 비트를 검사한다는 겁니다.

이 방법은 Appendix B.7에 보인 것과 거의 똑같은 read-side 성능을 달성하여서 POWER5 CPU 갯수에 관계 없이 약 65 나노세컨드를 소모합니다. 업데이트는 역시 더 큰 오버헤드를 일으키는데, 단일 POWER5 CPU에서의 600 나노세컨드에서 64 CPU에서의 100 마이크로세컨드를 오갑니다.

Quick Quiz B.19: 왜 이 복잡한 비트 조정 대신 앞의 섹션에서처럼 가난하게 별개의 쓰레드별 중첩 수준 변수를 유지하지 않나요?

이 구현은 이제 RCU read-side 크리티컬 섹션의 중첩이 허용되었다는 것만 제외하면 Appendix B.7의 것과 동일한 단점을 갖습니다. 또한, 32-bit 시스템에서 이 방법은 전역 `rcu_gp_ctrl` 변수를 오버플로우시키는데 걸리는 시간을 단축시킵니다. 다음 섹션은 오버플로우에 걸리는 시간을 크게 증가시키면서 read-side 오버헤드를 크게 줄이는 방법을 보입니다.

Quick Quiz B.20: Listing B.16에 보인 알고리즘을 가지고 어떻게 전역 `rcu_gp_ctrl`의 오버플로우에 걸리는 시간을 두배로 늘릴 수 있죠?

Quick Quiz B.21: 다시 Listing B.16의 알고리즘에서 카운터 오버플로우는 치명적인가요? 왜 그렇죠? 그게 치명적이라면 이를 고치기 위해 뭘 할 수 있을까요?

B.9 RCU Based on Quiescent States

Listing B.18 (`rcu_qs.h`)는 Listing B.17에 보인 데이터와 함께 quiescent state에 기반한 RCU의 사용자 수준 구현에 사용된 read-side 기능들을 보입니다. 이 리스트의 라인 1-7에 보인 것처럼 `rcu_read_lock()`과 `rcu_read_unlock()` 기능은 아무 것도 하지 않으며, 실제로 inline화 되고 최적화되어 사라질 것으로 기대될 수 있는데, 리눅스 커널의 서버 빌드에서도 그렇습니다. 이는 quiescent-state 기반 RCU 구현은 RCU read-side 크리티컬 섹션의 길이를 앞서 언급한 quiescent state를 사용해 추정하기 때문입니다. 이 quiescent state 각각은 이 리스트의 라인 9-15에 보인 `rcu_quiescent_state()` 호출을 포함합니다. 연장된 quiescent state(예를 들면, 블록킹)에 진

Listing B.17: Data for Quiescent-State-Based RCU

```

1 DEFINE_SPINLOCK(rcu_gp_lock);
2 long rcu_gp_ctr = 0;
3 DEFINE_PER_THREAD(long, rcu_reader_qs_gp);

```

Listing B.18: Quiescent-State-Based RCU Read Side

```

1 static void rcu_read_lock(void)
2 {
3 }
4
5 static void rcu_read_unlock(void)
6 {
7 }
8
9 static void rcu_quiescent_state(void)
10 {
11     smp_mb();
12     __get_thread_var(rcu_reader_qs_gp) =
13         READ_ONCE(rcu_gp_ctr) + 1;
14     smp_mb();
15 }
16
17 static void rcu_thread_offline(void)
18 {
19     smp_mb();
20     __get_thread_var(rcu_reader_qs_gp) =
21         READ_ONCE(rcu_gp_ctr);
22     smp_mb();
23 }
24
25 static void rcu_thread_online(void)
26 {
27     rcu_quiescent_state();
28 }

```

입하는 쓰레드는 이대신 `rcu_thread_offline()` (라인 17-23)를 연장된 quiescent state에 진입할 때, 그리고 여기서 나갈 땐 `rcu_thread_online()` (라인 25-28)을 호출합니다. `rcu_thread_online()`은 `rcu_read_lock()` 과, `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.³ RCU read-side 크리티컬 섹션에서 `rcu_quiescent_state()`, `rcu_thread_offline()`, 또는 `rcu_thread_online()`을 호출하는 것은 불법입니다.

`rcu_quiescent_state()`에서, 라인 11은 이 quiescent state 앞의 코드가 (RCU read-side 크리티컬 섹션 포함) quiescent state 안으로 재배치 되는 걸 막기 위해 메모리 배리어를 수행합니다. 라인 12-13은 전역 `rcu_gp_ctr`의 복사본을 가져오는데, `rcu_gp_ctr`이 여러번 읽혀지는 결과를 가져올 수 있는 어떤 컴파일러 최적화도 막기 위해 `READ_ONCE()`를 사용하며, 이어서 읽혀진 값에 1을 더하고 이를 쓰레드별 `rcu_reader_qs_gp` 변수에 저장해서 모든 동시의 `synchronize_rcu()` 수행은 할수 값을 읽고, 따라서 새 RCU read-side 크리티컬 섹션이 시작되었음을 알 수 있게 합니다. 이전의 RCU read-side 크리티컬 섹션을 기다리는 `synchronize_rcu()` 인스턴스는 따라서 이 새 것을 무시해도 됨을 알게 됩니다. 마지막으로, 라인 14는 메모리 배리어를 수행하는데, 이는 뒤따르는 코드가 (RCU read-side 크리티컬 섹션 포함) 라인 12-13 와 순서를 바꾸는 걸 방지합니다.

Listing B.19: RCU Update Side Using Quiescent States

```

1 void synchronize_rcu(void)
2 {
3     int t;
4
5     smp_mb();
6     spin_lock(&rcu_gp_lock);
7     WRITE_ONCE(rcu_gp_ctr, rcu_gp_ctr + 2);
8     smp_mb();
9     for_each_thread(t) {
10         while (rcu_gp_ongoing(t) &&
11                ((per_thread(rcu_reader_qs_gp, t) -
12                  rcu_gp_ctr) < 0)) {
13             poll(NULL, 0, 10);
14         }
15     }
16     spin_unlock(&rcu_gp_lock);
17     smp_mb();
18 }

```

`qs_gp` 변수에 저장해서 모든 동시의 `synchronize_rcu()` 수행은 할수 값을 읽고, 따라서 새 RCU read-side 크리티컬 섹션이 시작되었음을 알 수 있게 합니다. 이전의 RCU read-side 크리티컬 섹션을 기다리는 `synchronize_rcu()` 인스턴스는 따라서 이 새 것을 무시해도 됨을 알게 됩니다. 마지막으로, 라인 14는 메모리 배리어를 수행하는데, 이는 뒤따르는 코드가 (RCU read-side 크리티컬 섹션 포함) 라인 12-13 와 순서를 바꾸는 걸 방지합니다.

Quick Quiz B.22: Listing B.18의 라인 14에 보인 추가적인 메모리 배리어는 `rcu_quiescent_state()`의 오버헤드를 크게 늘리지 않나요?

■

어떤 어플리케이션들은 RCU를 가끔만 사용하나, 사용할 때에는 많이 사용할 수도 있습니다. 그런 어플리케이션은 RCU를 사용하기 시작할 때 `rcu_thread_online()`을 사용하고 더이상 RCU를 사용하지 않을 때 `rcu_thread_offline()`을 사용할 수도 있을 겁니다. `rcu_thread_offline()` 호출과 뒤따르는 `rcu_thread_online()` 호출 사이의 시간은 연장된 quiescent state 이므로, RCU는 이 사이에 명시적인 quiescent state가 등록되지 않을 것으로 예상할 겁니다.

`rcu_thread_offline()` 함수는 단순히 쓰레드별 `rcu_reader_qs_gp` 변수를 짹수 값을 가지고 있을 `rcu_gp_ctr`의 현재 값으로 설정합니다. 따라서 모든 동시의 `synchronize_rcu()` 인스턴스는 이 쓰레드를 무시해도 됨을 알 겁니다.

Quick Quiz B.23: Listing B.18의 라인 11 와 14의 메모리 배리어 한쌍은 왜 필요한가요?

■

`rcu_thread_online()` 함수는 단순히 `rcu_quiescent_state()`를 호출하는데, 따라서 연장된 quiescent state의 끝을 표시합니다.

³ 이 리스트의 코드가 `rcu_quiescent_state()`는 `rcu_thread_online()` 뒤에 곧바로 `rcu_thread_offline()`이 따라오는 것으로 동일하게 일관적이지만, 이 관계는 성능 최적화에 의해 사라집니다.

Listing B.19 (`rcu_qs.c`) 는 앞 섹션의 것과 상당히 비슷한 `synchronize_rcu()` 구현을 보입니다.

이 구현은 `rcu_read_lock()`-`rcu_read_unlock()` 왕복이 약 50 피코세컨드의 오버헤드를 가지는, 놀랍도록 빠른 read-side 기능을 가집니다. `synchronize_rcu()` 오버헤드는 단일 CPU POWER5 시스템에서의 600 나노세컨드에서 64-CPU 시스템에서의 100 마이크로세컨드 사이를 오갑니다.

Quick Quiz B.24: 분명히 해두자면, 2008년의 POWER 시스템의 클럭 주파수는 상당히 높았지만, 5 GHz 클럭 주파수조차도 반복문이 50 피코세컨드 내에 수행되게 하는데에는 불충분해요! 뭐가 어떻게 된거죠?

■
하지만, 이 구현은 각 쓰레드가 `rcu_quiescent_state()` 를 주기적으로 호출하거나 연장된 quiescent state 를 위해 `rcu_thread_offline()` 을 호출할 것을 필요로 합니다. 이 함수들을 주기적으로 호출해야 하는 필요성은 이 구현을 특정한 종류의 라이브러리 함수 같은 일부 상황에서는 사용하기 어렵게 할 수 있습니다.

Quick Quiz B.25: 왜 그 코드가 라이브러리에 있다는 사실이 Listings B.18 and B.19 에 보인 RCU 구현의 사용 편의성에 차이를 끼칠 수 있죠?

■
Quick Quiz B.26: 하지만 `synchronize_rcu()` 호출 동안 락을 잡고 있고, 똑같은 락을 어떤 RCU read-side 크리티컬 섹션 내에서 잡는다면 어떻게 될까요? 이는 데드락일 겁니다만, 어떤 코드도 만들어내지 않는 기능이 어떻게 데드락 사이클에 참여할 수 있을까요?

■
또한, 이 구현은 동시의 `synchronize_rcu()` 호출이 grace period 를 공유하는 걸 허용하지 않습니다. 그러나, 이 버전의 RCU 에 기반한 제품 품질 RCU 구현을 쉽게 상상할 수 있을 겁니다.

B.10 Summary of Toy RCU Implementations

여기까지 오셨다면, 축하합니다! 여러분은 이제 RCU 자체만이 아니라 이를 둘러싼 소프트웨어 환경과 어프리케이션들의 요구사항에 대해서도 훨씬 선명한 이해를 가졌을 겁니다. 이보다도 깊은 이해를 원하는 분들은 제품 수준 RCU 구현 [DMS⁺12, McK07a, McK08a, McK09a] 을 읽어보시기 바랍니다.

앞의 섹션들은 다양한 RCU 기능의 요구되는 속성들을 나열했습니다. 다음 리스트는 새로운 RCU 구현을 만들고자 하는 분들을 위한 쉬운 참고사항을 제공합니다.

1. 어떤 grace period 의 시작 시점에 존재했던 모든 RCU read-side 크리티컬 섹션은 이 grace period 의 종료 시점에는 완료되어 있어야 하는 read-side 기능들 (`rcu_read_lock()` 과 `rcu_read_unlock()` 같은) 그리고 grace-period 기능들 (`synchronize_rcu()` 과 `call_rcu()` 같은) 이 있어야만 합니다.
2. RCU read-side 기능들은 최소한의 오버헤드를 가져야만 합니다. 특히, 캐쉬 미스, 어토믹 명령, 메모리 배리어, 그리고 분기문은 회피되어야 합니다.
3. RCU read-side 기능은 리얼타임 사용을 가능하게 하기 위해 $O(1)$ 계산 복잡도를 가져야 합니다. (이는 읽기 쓰레드가 업데이트 스레드와 동시에 수행됨을 암시합니다.)
4. RCU read-side 기능은 모든 컨텍스트에서 사용이 가능해야 합니다 (리눅스 커널의 경우, idle loop 를 제외한 모든 곳에서 가능합니다). 한 가지 중요한 특수 경우는 RCU read-side 기능이 RCU read-side 크리티컬 섹션 내에서 사용 가능한 경우로, 달리 말하자면, RCU read-side 크리티컬 섹션은 중첩될 수 있다는 겁니다.
5. RCU read-side 기능은 무조건적으로 실패하지 않아야 합니다. 이 속성은 무척 중요한데, 실패 검사는 복잡도를 높이고 테스트와 검증을 복잡하게 할 것이기 때문입니다.
6. Quiescent state (그리고 따라서 grace period) 외의 모든 오퍼레이션은 RCU read-side 크리티컬 섹션 내에서 허용되어야 합니다. 특히, I/O 같은 취소 불가능한 오퍼레이션도 허용되어야 합니다.
7. RCU read-side 크리티컬 섹션 내에서 수행 중인 동안 RCU 로 보호되는 데이터 구조를 업데이트 할 수 있어야 합니다.
8. RCU read-side 와 update-side 기능들은 메모리 할당자 설계와 구현과 독립적이어야 하는데, 달리 말하자면 같은 RCU 구현이 어떤 데이터 구조를 보호하는데 있어 그 데이터 원소가 어떻게 할당되고 해제되는지는 관계 없어야 합니다.
9. RCU grace period 는 RCU read-side 크리티컬 섹션 밖에서 중단되는 쓰레드에 의해 블록되지 않아야 합니다. (그러나 대부분의 quiescent-state 기반 구현은 이 요구사항을 위반함을 알아두시기 바랍니다.)

Quick Quiz B.27: RCU read-side 크리티컬 섹션 내에서 grace period 가 금지된다면, 어떻게 RCU 로 보호되는 데이터 구조가 RCU read-side 크리티컬 섹션 내에서 업데이트 될 수 있죠?

■

Appendix C

Why Memory Barriers?

그래서 뭐가 CPU 설계자들을 불쌍한 의심없던 SMP 소프트웨어 설계자들에게 memory barrier를 가하게 훔쳤을까요?

짧게 말하자면, 메모리 참조를 재배치 하는 것은 훨씬 나은 성능을 가능하게 하며, 따라서 메모리 배리어는 올바른 오퍼레이션이 순서 맞춰진 메모리 참조에 의존적인 동기화 기능들 같은 것을 위해 순서를 강제하기 위해 필요합니다.

이 질문에 대한 더 자세한 답은 CPU 캐시가 어떻게 동작하는지, 특히 캐시가 정말로 잘 동작하기 위해 무엇을 필요로 하는지에 대한 좋은 이해가 필요합니다. 다음 섹션들은:

1. 캐시의 구조를 보이고,
2. 캐시 일관성 프로토콜이 어떻게 CPU들이 각 메모리 위치의 값에 동의를 하는 것을 보장하는지 설명하며, 마지막으로
3. 스토어 베피와 무효화 큐가 어떻게 캐시와 캐시 일관성 프로토콜에게 높은 성능을 이루게 돋는지 설명합니다.

우린 메모리 배리어가 좋은 성능과 확장성을 가능하게 하기 위해 필요하며 CPU는 그것들과 그것들이 접근하고자 하는 메모리 사이의 연결부보다 수십수백배 빠르다는 사실에서 기인한 필요한 악마임을 보게 될 겁니다.

C.1 Cache Structure

현대의 CPU는 현대의 메모리 시스템보다 훨씬 빠릅니다. 2006년의 CPU는 나노세컨드당 열개의 명령을 수행할 수도 있으나, 메인 메모리에서 데이터 항목을 가져오는데에는 수십 나노세컨드를 필요로 합니다. 이 속도에서의 괴리—수백배 이상의—는 현대 CPU에서 찾아볼 수 있는 수백 메가바이트 캐시를 초래했습니다.

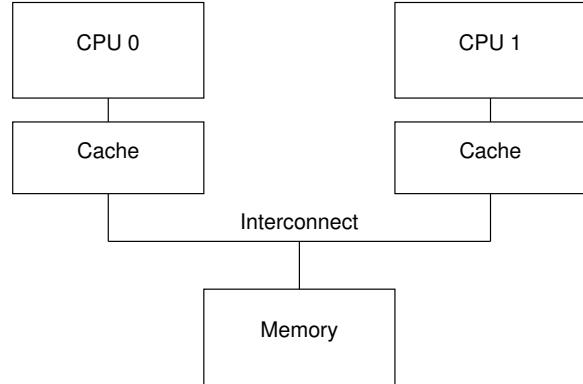


Figure C.1: Modern Computer System Cache Structure

이 캐시들은 Figure C.1에 보인 것처럼 CPU들과 연관되어지며, 보통 수 사이클만에 접근될 수 있습니다.¹

데이터는 “캐시 라인”이라 불리는 고정 길이 블록으로 CPU들의 캐시와 메모리 사이를 떠다니는데, 그 크기는 보통 2의승수이며, 16에서 256 바이트 사이를 오갑니다. 어떤 데이터 항목이 어떤 CPU에 의해 처음 액세스 될 때, 이는 해당 CPU의 캐시에 존재하지 않을 텐데, 이는 “캐시 미스 (cache miss)”(또는, 더 구체적으로는 “startup” 또는 “warmup” 캐시 미스)가 일어났음을 의미합니다. 이 캐시 미스는 이 CPU가 해당 항목이 메모리로부터 읽어들여지기까지 수백 사이클을 기다려야 (또는 “stall” 되어야) 함을 의미합니다. 그러나, 이 항목은 이 CPU의 캐시로 읽혀들여질 것이므로, 뒤따르는 액세스들은 이 항목을 캐시에서 발견하고 따라서 완전한 속도를 낼 수 있을 겁니다.

¹ 작은 레벨 1의 캐시를 1 사이클 액세스 시간을 가질 만큼 CPU에 가깝게, 더 큰 레벨 2의 캐시를 더 긴 액세스 속도, 대략 10 클락 사이클로 위치하는 식의 다중 레벨의 캐시를 사용하는 것은 표준적인 방법입니다. 고성능 CPU들은 종종 3 또는 4 레벨의 캐시조차 같습니다.

	Way 0	Way 1
0x0	0x12345000	
0x1	0x12345100	
0x2	0x12345200	
0x3	0x12345300	
0x4	0x12345400	
0x5	0x12345500	
0x6	0x12345600	
0x7	0x12345700	
0x8	0x12345800	
0x9	0x12345900	
0xA	0x12345A00	
0xB	0x12345B00	
0xC	0x12345C00	
0xD	0x12345D00	
0xE	0x12345E00	0x43210E00
0xF		

Figure C.2: CPU Cache Structure

어느정도 시간이 지나면 이 CPU의 캐시는 가득찰 테고, 뒤따르는 미스들은 새로 읽어온 항목을 위한 공간을 마련하기 위해 존재하는 항목을 내버려야 할 겁니다. 이런 캐시 미스는 “capacity miss”라고 명명되는데, 캐시의 제한된 용량 때문에 일어났기 때문입니다. 그러나, 대부분의 캐시는 완전히 가득 차지 않았을 때 조차도 새 항목을 위한 공간을 위해 오래된 항목을 버려야 할 수 있습니다. 이는 커다란 캐시는 Figure C.2에 보인 것처럼 고정 크기 해쉬 버킷과 (또는 CPU 설계자들이 부르는 대로라면 “set”) 체이닝 없는 하드웨어 해쉬 테이블로 구현되어 있다는 사실 때문입니다.

이 캐시는 16개의 “set”과 두개의 “way”를 가진 총 32개의 “라인”을 가지며, 각 항목은 256 바이트의 정렬된 메모리 블록인 256-바이트 “캐시 라인”을 갖습니다. 이 캐시 라인 크기는 약간 큰 크기이지만 16 진수 계산을 훨씬 간단하게 해줄 겁니다. 하드웨어 용어에서, 이는 2-way set-associative 캐시라 불리며, 16개의 버킷을 갖고 각 버킷의 해쉬 체인은 최대 두개의 원소로 제한된 소프트웨어 해쉬 테이블과 유사합니다. 이 크기 (이 경우 32 캐시 라인)와 associativity (이 경우 2)는 합쳐져서 이 캐시의 “geometry”라고 불립니다. 이 캐시는 하드웨어로 구현되었으므로, 그 해쉬 함수는 굉장히 간단합니다: 메모리 주소에서 4개 비트를 꺼냅니다.

Figure C.2에서, 각 상자는 256-바이트 캐시 라인을 가질 수 있는 캐시 항목에 연관됩니다. 그러나, 캐시 항목은 이 그림의 빈 상자로 표시되어 있듯 비어있을 수 있습니다. 나머지 상자들은 그것들이 가진 캐시 라인의 메모리 주소로 표시되어 있습니다. 캐시 라인은 256 바이트로 정렬되어 있어야 하므로, 각 주소의 아래쪽 8비트는 0이며, 하드웨어 해쉬 함수는 그보다 높은 쪽 4개의 비트를 해쉬 라인 숫자와 매치되게 합니다.

이 그림에 보여진 상황은 프로그램의 코드가 주소 0x43210E00부터 0x43210EFF에 위치하고, 이 프로그램이 0x12345000부터 0x12345EFF의 데이터를 순차적으로 액세스 했을 때 일어날 수도 있을 겁니다. 이 프로그램이 이제 0x12345F00을 액세스 하려 한다고 해봅시다. 이 위치는 라인 0xF로 해쉬 되며, 이 라인의 두 way는 비어있으므로, 연관된 256-바이트 라인이 수용될 수 있습니다. 이 프로그램이 라인 0x0으로 해쉬 되는 0x1233000 위치를 접근하려 한다면, 연관된 256-바이트 캐시 라인이 way 1에 수용될 수 있습니다. 256-바이트 캐시 라인이 way 1에 수용될 수 있습니다. 그러나, 프로그램이 라인 0xE로 해쉬 되는 0x1233E00 위치를 액세스 하려 하면, 이 새로운 캐시 라인을 위한 고간을 만들기 위해 존재하는 라인 중 하나가 버려져야 합니다. 이 버려진 라인이 나중에 다시 액세스 된다면 캐시 미스가 일어날 겁니다. 그런 캐시 미스는 “associativity miss”라 불립니다.

지금까지는 CPU가 데이터를 읽는 경우만 생각해 봤습니다. 쓰기를 할 때는 무슨 일이 일어날까요? 모든 CPU가 어떤 데이터 항목의 값에 대해 동의한느 것은 중요하므로, 어떤 CPU는 어떤 데이터 항목에 쓰기를 하기 전에 먼저 그것이 다른 CPU들의 캐시에서 없어지거나 “무효화 (invalidate)” 되게 해야만 합니다. 이 무효화가 완료되면, 이 CPU는 안전하게 데이터 항목을 수정할 수 있습니다. 이 데이터 항목이 이 CPU의 캐시에 존재하지만 읽기 전용이었다면, 이 과정은 “write miss”라고 불립니다. 특정 CPU가 다른 CPU들의 캐시로부터 특정 데이터 항목을 완전히 무효화 하면, 해당 CPU는 이 데이터 항목을 반복적으로 쓸 수 (그리고 읽을 수) 있을 겁니다.

나중에 다른 CPU들 중 하나가 이 데이터 항목을 접근하려 하면 캐시 미스가 일어날 텐데, 이번엔 이 앞의 CPU가 그 항목에 쓰기를 하기 위해 무효화를 했기 때문입니다. 이런 종류의 캐시 미스는 “communication miss”라 불리는데, 이는 보통 CPU들이 이 데이터 항목을 통신을 위해 사용하기 때문입니다 (예를 들면, 같은 CPU들 사이에 상호 배타 알고리즘을 통신하기 위해 사용되는 데이터 항목입니다).

분명하게, 모든 CPU가 데이터의 일관된 시선을 유지하게 보장하는 데에는 큰 주의가 필요합니다. 이 모든 읽기, 무효화, 그리고 쓰기를 놓고 볼 때, 데이터가 손실되거나 (아마도 더 나쁘게도) 다른 CPU들이 같은 데이터 항목에 대해 그들의 캐시에서는 다른 값을 갖고 있는 경우를 상상하기는 쉽습니다. 이 문제들은 “캐시 일관성 프로토콜”에 의해 방지되는데, 다음 섹션에서 설명합니다.

C.2 Cache-Coherence Protocols

캐시 일관성 프로토콜은 비일관적이거나 손실된 데이터를 막을 수 있게끔 캐시 라인 상태들을 관리합니다. 이 프로토콜은 수십 개의 상태를 가질 정도로² 매우 복잡할 수 있으나 우리의 목적을 위해선 4개 상태의 MESI 캐시 일관성 프로토콜에 대해서만 걱정하면 됩니다.

C.2.1 MESI States

MESI는 “modified”, “exclusive”, “shared”, 그리고 “invalid”라는 각 캐시 라인이 이 프로토콜을 사용해 취할 수 있는 네 개의 상태를 의미합니다. 따라서 이 프로토콜을 사용하는 캐쉬는 각 캐시 라인에 해당 라인의 물리 주소와 데이터에 대해 2 비트의 상태 “tag”를 유지합니다.

“modified” 상태의 라인은 연관된 CPU로부터의 최근의 메모리 스토어를 의미하며, 이 연관된 메모리는 다른 CPU의 캐쉬에는 보이지 않을 것이 보장됩니다. 따라서 “modified” 상태의 캐쉬 라인은 이 CPU에 “owned (소유되어 있다)”라고 말해집니다. 이 캐쉬는 데이터의 최신 복사본만을 쥐고 있으므로, 이 캐쉬는 궁극적으로는 이를 메모리에 다시 쓰거나 다른 캐쉬에게 넘겨줄 책임을 가지며, 이는 이 라인이 다른 데이터를 잡아두기 위해 재사용되기 전에 행해져야만 합니다.

“exclusive” 상태는 “modified” 상태와 매우 비슷한데, 한 가지 예외는 이 캐쉬 라인은 아직 연관된 CPU에 의해 수정되지 않았다는 것으로, 이는 결국 메모리에 위치해 있는 이 캐쉬 라인의 데이터의 복사본도 최신의 것이라는 겁니다. 그러나, 이 CPU는 언제든 이 라인에 스토어를 행할 수 있으므로, 다른 CPU에게 물어보지 않고서는 “exclusive” 상태의 라인은 연관된 CPU에게 소유되어 있다고 말해질 수 있습니다. 그러나, 메모리의 연관된 값이 최신의 것이기 때문에, 이 캐쉬는 이 데이터를 메모리에 다시 쓰거나 다른 CPU에게 넘겨주지 않고도 버릴 수 있습니다.

“shared” 상태의 라인은 최소 하나의 다른 CPU의 캐쉬에 복사되어 있을 수도 있어서, 이 CPU는 다른 CPU에게 자문을 먼저 구하지 않고는 이 라인에 스토어를 할 수 없습니다. “exclusive” 상태에서와 마찬가지로, 메모리에 있는 이 연관된 값은 최신의 것이어서, 이 캐쉬는 이 데이터를 메모리에 다시 쓰거나 다른 CPU에게 넘겨주지 않고도 버릴 수 있습니다.

“invalid” 상태의 라인은 비어 있는데, 달리 말해 데이터를 쥐고 있지 않습니다. 캐쉬에 새 데이터가 들어오면,

² Culler 등의 책 [CS99]의 670과 671 페이지에서 각각 SGI Origin2000과 Sequent(지금은 IBM) NUMA-Q를 위한 9개 상태와 26개 상태 다이어그램들을 보시기 바랍니다. 두 다이어그램은 실제의 것보다 훨씬 간단합니다.

이는 가능하다면 “invalid” 상태의 캐쉬 라인에 위치하게 됩니다. 다른 상태의 라인을 교체하는 것은 교체된 라인이 미래에 참조될 때 비용이 높은 캐쉬 미스를 초래하기 때문에 이 방법이 선호됩니다.

모든 CPU는 캐쉬 라인들로 운반되는 데이터에 대해 일관된 모습을 봐야만 하므로, 이 캐쉬 일관성 프로토콜은 시스템을 돌아다니는 캐쉬 라인들의 움직임을 조정하는 메세지를 제공합니다.

C.2.2 MESI Protocol Messages

앞의 섹션에 설명된 많은 변환은 CPU들 사이의 통신을 필요로 합니다. CPU들이 하나의 공유 버스에 있다면, 다음 메세지가 충분합니다:

Read:

“Read” 메세지는 읽혀질 캐쉬 라인의 물리 주소를 담습니다.

Read Response:

“Read response” 메세지는 앞의 “read” 메세지에 의해 요청된 데이터를 담습니다. 이 “read response” 메세지는 메모리 또는 다른 캐쉬들 가운데 하나에 의해 제공될 수도 있습니다. 예를 들어, 캐쉬들 가운데 하나가 “modified” 상태로 해당 데이터를 가지고 있다면, 그 캐쉬는 “read response” 메세지를 제공해야만 합니다.

Invalidate:

“Invalidate” 메세지는 무효화 될 캐쉬 라인의 물리 주소를 담습니다. 모든 다른 캐쉬들은 그들의 캐쉬에서 연관된 데이터를 제거하고 응답해야만 합니다.

Invalidate Acknowledge:

“Invalidate” 메세지를 받는 CPU는 자신의 캐쉬에서 명시된 데이터를 제거한 후 “invalidate acknowledge” 메세지를 가지고 응답해야만 합니다.

Read Invalidate:

“Read invalidate” 메세지는 읽혀질 캐쉬 라인의 물리 주소를 담고 있으며, 동시에 다른 캐쉬들이 이 데이터를 제거하게 지시합니다. 따라서, 이는 그 이름이 암시하듯 “read”와 “invalidate”的 조합입니다. “Read invalidate” 메세지는 그에 대한 응답으로 “read response”와 “invalidate acknowledge” 메세지의 집합을 모두 요구합니다.

Writeback:

“Writeback” 메세지는 메모리에 도로 쓰여질 (그리고 아마도 다른 CPU의 캐쉬에도 기어들어가게 될 “snooped”) 주소와 데이터를 포함합니다. 이

메세지는 캐쉬들이 다른 데이터를 위한 공간을 만들기 위해 “modified” 상태의 라인을 제거하는 걸 가능하게 합니다.

Quick Quiz C.1: Writeback 메세지는 어디서 생겨나서 어디로 향하게 되나요?

흥미롭게도, 공유 메모리 멀티프로세서 시스템은 장막을 걷어보면 실제로 메세지 전달(message-passing) 컴퓨터입니다. 이는 분산된 공유 메모리를 사용하는 SMP 머신들의 클러스터는 시스템 구조의 두개의 다른 수준에서의 공유 메모리를 구현하기 위해 메세지 전달을 사용하고 있음을 의미합니다.

Quick Quiz C.2: 두개의 CPU 가 동시에 같은 캐쉬 라인을 무효화 하려 하면 어떻게 되나요?

Quick Quiz C.3: 거대 멀티프로세서에서 “invalidate” 메세지가 나타날 때, 모든 CPU 는 “invalidate acknowledge” 응답을 보내야만 합니다. 이로 인한 “invalidate acknowledge” 응답의 “폭풍”은 시스템 버스를 완전히 포화시키지 않을까요?

Quick Quiz C.4: SMP 머신이 정말로 메세지 전달을 어쨌든 사용한다면, 왜 SMP 를 신경쓰죠?

C.2.3 MESI State Diagram

특정 캐쉬 라인의 상태는 Figure C.3 에 보인 것처럼 프로토콜 메세지가 보내지고 받아짐에 따라 바뀝니다.

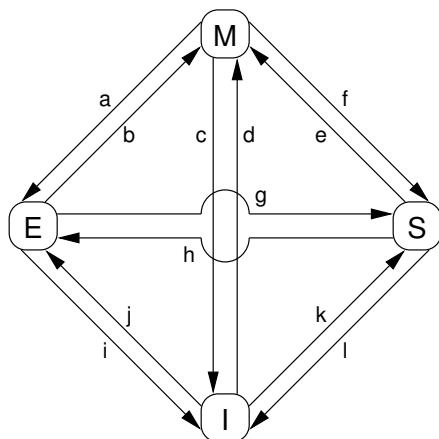


Figure C.3: MESI Cache-Coherency State Diagram

이 그림에서의 전환 움직임들은 다음과 같습니다:

Transition (a):

캐쉬 라인이 메모리에 도로 쓰여지나, 이 CPU 는 자신의 캐쉬에 이를 유지하며 더 나아가 이를 수정할 권리를 유지합니다. 이 전환은 “writeback” 메세지를 필요로 합니다.

Transition (b):

이 CPU 가 이미 배타적 액세스를 가지고 있는 캐쉬 라인에 쓰기를 합니다. 이 전환은 어떤 메세지도 보내지거나 받아질 필요를 갖지 않습니다.

Transition (c):

이 CPU 가 수정한 캐쉬 라인을 위한 “read invalidate” 메세지를 받습니다. 이 CPU 는 자신의 지역 복사본을 무효화하고, 이 데이터를 요청한 CPU 에게 보내며 자신이 더이상 그 지역 복사본을 가지고 있지 않음을 알리는 “read response” 와 “invalidate acknowledge” 메세지를 응답해야 합니다.

Transition (d):

이 CPU 가 자신의 캐쉬에 존재하지 않던 데이터 항목에 대해 어토믹 read-modify-write 오퍼레이션을 행합니다. 이는 “read invalidate” 를 보내고, “read response” 를 통해 그 데이터를 받습니다. 이 CPU 는 완전한 “invalidate acknowledge” 응답 집합을 받으면 이 전환을 완료할 수 있습니다.

Transition (e):

이 CPU 가 자신의 캐쉬에 read-only 로 가지고 있던 데이터 항목에 어토믹 read-modify-write 오퍼레이션을 행합니다. 이는 “invalidate” 메세지를 보내야 하며, 이 전환을 완료하기 전에 완전한 “invalidate acknowledge” 응답 집합을 기다려야 합니다.

Transition (f):

어떤 다른 CPU 가 이 캐쉬 라인을 읽고, 그 데이터가 이 CPU 의 캐쉬에서 제공되어서, 읽기 전용 복사본을 유지하며, 이를 메모리에 도로 쓰기 할 수도 있습니다. 이 전환은 “read” 메세지의 도착으로 시작되고, 이 CPU 는 요청된 데이터를 포함한 “read response” 메세지로 응답합니다.

Transition (g):

어떤 다른 CPU 가 이 캐쉬 라인의 데이터 항목을 읽고, 그 데이터가 이 CPU 의 캐쉬 또는 메모리에서 제공됩니다. 어느 경우든, 이 CPU 는 읽기 전용 복사본을 유지합니다. 이 전환은 “read” 메세지의 도착으로 시작되며, 이 CPU 는 요청된 데이터를 담은 “read response” 메세지를 응답합니다.

Transition (h):

이 CPU 는 이 캐쉬 라인에 어떤 데이터 항목을

곧 써야 할 것임을 깨닫고 따라서 “invalidate” 메세지를 보냅니다. 이 CPU 는 완전한 “invalidate acknowledge” 응답 집합을 받기 전까지는 이 전환을 끝내지 못합니다. 대안적으로, 모든 다른 CPU 들이 “writeback” 메세지를 통해 이 캐쉬 라인을 자신들의 캐쉬에서 제거하여 (아마도 다른 캐쉬 라인을 위한 공간을 만들기 위해) 이 CPU 가 이를 캐쉬에 두는 마지막 CPU 가 되게 할수도 있습니다.

Transition (i):

어떤 다른 CPU 가 이 CPU 의 캐쉬에만 있는 캐쉬 라인의 데이터 항목에 대해 어토믹 read-modify-write 오피레이션을 수행해서, 이 CPU 는 자신의 캐쉬에서 이 라인을 무효화 합니다. 이 전환은 “read invalidate” 메세지의 도착으로 시작되며, 이 CPU 는 “read response” 와 “invalidate acknowledge” 메세지를 모두 응답합니다.

Transition (j):

이 CPU 는 자신의 캐쉬에 있지 않은 캐쉬 라인의 데이터 항목에 스토어를 행하고, 따라서 “read invalidate” 메세지를 보냅니다. 이 CPU 는 “read response” 와 완전한 “invalidate acknowledge” 메세지 집합을 받기 전까지 이 전환을 완료할 수 없습니다.

Transition (k):

이 CPU 가 자신의 캐쉬에 없던 캐쉬 라인의 데이터 항목을 로드합니다. 이 CPU 는 “read” 메세지를 보내며, 연관된 “read response” 를 받으면 전환을 완료합니다.

Transition (l):

어떤 다른 CPU 가 이 캐쉬 라인의 데이터 항목으로 스토어를 하지만, 이 캐쉬 라인은 다른 CPU 의 캐쉬에 (이 현재 CPU 의 캐쉬 같은) 있으므로 이 캐쉬 라인을 읽기 전용 상태로 유지합니다. 이 전환은 “invalidate” 메세지의 수신으로 시작되며, 이 CPU 는 “invalidate acknowledge” 메세지를 응답합니다.

Quick Quiz C.5: 하드웨어는 앞서 설명된 지연된 전환을 어떻게 처리하나요?



C.2.4 MESI Protocol Example

이제 초기에는 메모리의 address 0 에 위치해 있는 데이터의 캐쉬 라인이 네개의 CPU 시스템에서의 단일 캐쉬 라인만 갖는 여러 캐쉬들을 이동하는 모습을 캐쉬 라인의 관점에서 봅시다. Table C.1 이 데이터의 흐름을 보이는데, 첫번째 열은 오피레이션의 순서를, 두번째

열은 그 오피레이션을 행하는 CPU 를, 세번째 열은 수행되는 오피레이션을, 나머지 네개의 열은 각 CPU 의 캐쉬 라인의 상태를 (메모리 주소에 이어 MESI 상태), 그리고 마지막 두개의 열은 연관된 메모리 내용이 최신인지 (“V”) 아닌지 (“I”) 보입니다.

처음에는 이 데이터가 위치해 있는 CPU 캐쉬 라인들은 “invalid” 상태에 있으며, 그 데이터는 메모리에서 유효합니다. CPU 0 이 address 0 에서 데이터를 로드하면, 이는 CPU 0 의 캐쉬에 “shared” 상태로 들어가게 되며, 이는 여전히 메모리에서 유효합니다. CPU 3 역시 address 0 에서 이 데이터를 로드해서, 두 CPU 의 캐쉬에서 “shared” 상태로 있게 하며, 이 데이터는 여전히 메모리 상에서 유효합니다. 이어서 CPU 0 이 다른 캐쉬 라인을 (address 8) 로드하는데, 이는 address 0 의 데이터를 무효화를 통해 자신의 캐쉬에서 밖으로 내보내고 address 8 의 데이터로 교체합니다. CPU 2 가 이제 address 0 으로부터의 로드를 합니다만, 이 CPU 는 자신이 곧 거기에 스토어를 해야 할 것임을 깨달아 배타적 복사본을 얻기 위해 “read invalidate” 메세지를 사용하여 이를 CPU 3 의 캐쉬에서 무효화 시킵니다 (메모리의 복사본은 여전히 최신의 것으로 남아있지만). 이어서 CPU 2 는 예상된 스토어를 행하여 이 상태를 “modified” 로 바꿉니다. 메모리에 있는 이 데이터의 사본은 이제 최신이 아닙니다. CPU 1 은 원자적 값 증가를 행하는데, CPU 2 의 캐쉬로부터 데이터를 얻고 이를 무효화 하기 위해 “read invalidate” 메세지를 사용하며, 따라서 CPU 1 의 캐쉬는 “modified” 상태가 됩니다 (그리고 메모리의 복사본은 최신이 아닌 상태로 유지됩니다). 마지막으로, CPU 1 이 address 8 의 캐쉬 라인을 읽는데, address 0 의 데이터를 메모리로 도로 내보내기 위해 “writeback” 메세지를 사용합니다.

우린 데이터가 이 CPU 의 캐쉬들 중 일부에 남겨져 있는 채로 이를 끝냄을 알아두시기 바랍니다.

Quick Quiz C.6: 어떤 순서의 오피레이션들이 이 CPU 의 캐쉬들을 “invalid” 상태로 되돌릴까요?



C.3 Stores Result in Unnecessary Stalls

Figure C.1 에 보인 캐쉬 구조가 특정 CPU 에서 특정 데이터 항목으로의 반복된 읽기와 쓰기에 대해 좋은 성능을 제공하지만, 해당 캐쉬 라인으로의 첫번째 쓰기는 성능이 상당히 떨어집니다. 이를 이해하기 위해, CPU 0 에 의한 CPU 1 의 캐쉬에 있는 캐쉬 라인으로의 쓰기에 서의 시간 흐름을 보이는 Figure C.4 를 생각해 봅시다. CPU 0 는 이 캐쉬 라인에 쓰기를 할 수 있게 되기 전에

Table C.1: Cache Coherence Example

Sequence #	CPU #	Operation	CPU Cache				Memory	
			0	1	2	3	0	8
0		Initial State	-/I	-/I	-/I	-/I	V	V
1	0	Load	0/S	-/I	-/I	-/I	V	V
2	3	Load	0/S	-/I	-/I	0/S	V	V
3	0	Invalidation	8/S	-/I	-/I	0/S	V	V
4	2	RMW	8/S	-/I	0/E	-/I	V	V
5	2	Store	8/S	-/I	0/M	-/I	I	V
6	1	Atomic Inc	8/S	0/M	-/I	-/I	I	V
7	1	Writeback	8/S	8/S	-/I	-/I	V	V

이 캐시 라인이 도착하기 전에 CPU 0은 연장된 시간 동안 멈춰있어야만 합니다.³

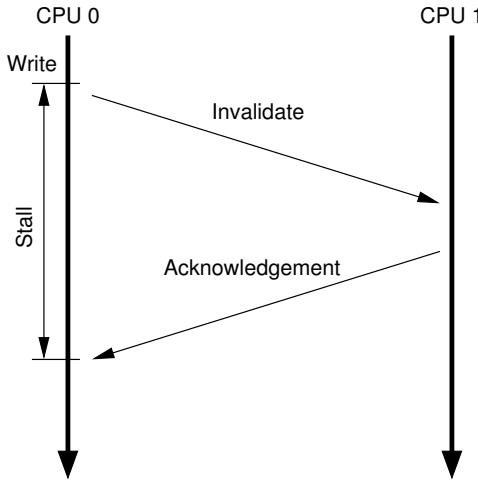


Figure C.4: Writes See Unnecessary Stalls

하지만 CPU 0이 그렇게 오래 멈춰있게 할 진짜 이유는 없습니다—어쨌건, 어떤 데이터가 CPU 1이 보내는 캐시 라인에 있건, CPU 0은 무조건적으로 이를 덮어쓸 겁니다.

C.3.1 Store Buffers

이 불필요한 쓰기에서 멈춰있음을 방지하는 한가지 방법은 Figure C.5에 보인 것처럼 각 CPU와 그들의 캐시 사이에 “store buffer (스토어 버퍼)”를 두는 겁니다. 이

³ 한 CPU의 캐시에서 다른 캐시로 캐시 라인을 이동시키는데 걸리는 시간은 일반적으로 간단한 레지스터에서 레지스터로의 명령을 수행하는 것보다 수십 수백배 더 깁니다.

스토어 버퍼가 추가되면 CPU 0은 자신의 쓰기를 자신의 스토어 버퍼에 기록하고 수행을 계속할 수 있습니다. 마침내 이 캐시 라인이 CPU 1에서 CPU 0으로 넘어가게 되면, 이 데이터는 스토어 버퍼에서 이 캐시 라인으로 이동하게 됩니다.

Quick Quiz C.7: 하지만 스토어 버퍼의 주요 목적이 멀티프로세서 캐시 일관성 프로토콜에서의 응답 지연을 감추기 위함이라면, 단일프로세서는 왜 스토어 버퍼를 갖죠?

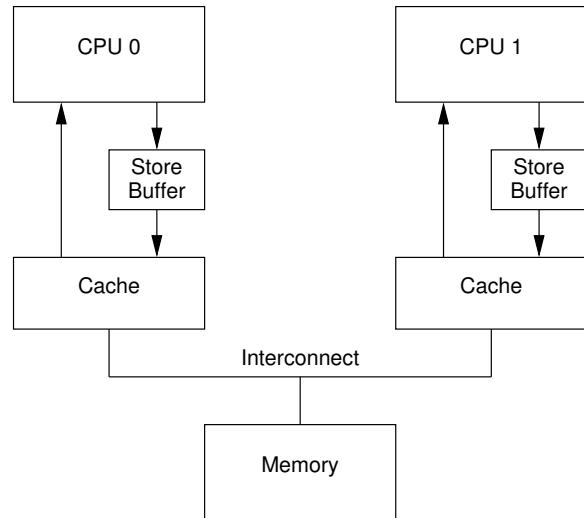


Figure C.5: Caches With Store Buffers

스토어 버퍼는 특정 CPU에, 또는 하드웨어 멀티쓰레딩에서는 코어에 지역적입니다. 어느 쪽이던, 특정 CPU는 자신에게 할당된 스토어 버퍼에만 액세스 할 수 있습니다. 예를 들어, Figure C.5에서 CPU 0은 CPU 1

의 스토어 버퍼에 접근할 수 없고 반대도 마찬가지입니다. 이 제한은 문제를 분리함으로써 하드웨어를 단순화 시킵니다: 스토어 버퍼는 연속되는 쓰기를 위한 성능을 개선시키며 CPU 사이의 (또는 코어들 사이의, 경우에 따라) 통신을 위한 책임은 캐시 일관성 프로토콜에 의해 처리됩니다. 그러나, 이 제한 아래에서 조차 처리되어야만 하는 복잡성이 존재하는데 이는 다음 두 섹션에서 다룹니다.

C.3.2 Store Forwarding

첫번째 복잡성인 자기 일관성의 위배를 보기 위해 0으로 초기화 되는 변수 “a” 와 “b” 를 갖고 변수 “a” 를 담는 캐시 라인은 초기에 CPU 1 에 소유되며 “b” 를 담는 캐시 라인은 초기에 CPU 0 에 소유되는 다음 코드를 생각해 봅시다:

```
1 a = 1;
2 b = a + 1;
3 assert(b == 2);
```

이 단정은 실패할 거라 예상할 겁니다. 그러나, Figure C.5 에 보인 매우 단순한 구조를 사용할 만큼 어리석은 사람이 있다면 놀랄 겁니다. 그런 시스템은 다음 이벤트 집합을 볼 잠재성이 있습니다:

- 1 CPU 0 이 $a = 1$ 을 수행합니다.
- 2 CPU 0 이 “a” 가 캐시에 있는지 확인하고, 그렇지 않음을 발견합니다.
- 3 따라서 CPU 0 이 “a” 를 담는 캐시 라인의 배타적 소유권을 갖기 위해 “read invalidate” 메세지를 보냅니다.
- 4 CPU 0 이 “a” 로의 스토어를 스토어 버퍼에 기록합니다.
- 5 CPU 1 이 “read invalidate” 메세지를 받고, 이 캐시 라인을 보내고 자신의 캐시로부터 이 캐시 라인을 제거함으로써 응답합니다.
- 6 CPU 0 이 $b = a + 1$ 을 수행합니다.
- 7 CPU 0 이 CPU 1 로부터 여전히 값이 0인 “a” 를 담은 캐시 라인을 받습니다.
- 8 CPU 0 이 자신의 캐시에서 “a” 를 로드하고 그 값이 0임을 보게 됩니다.
- 9 CPU 0 이 자신의 스토어 버퍼로부터의 항목을 새로 도착한 캐시 라인에 적용해 자신의 캐시의 “a” 의 값을 1 로 만듭니다.

10 CPU 0 이 앞의 “a” 에서 로드한 값 0에 1 을 더하고 이를 “b” 를 담는 캐시 라인에 (이미 CPU 0 에 소유되어 있다고 가정합니다) 저장합니다.

11 CPU 0 이 `assert(b == 2)` 를 수행하고, 실패합니다.

문제는 우리가 하나는 캐시에 그리고 다른 하나는 스토어 버퍼에, “a” 의 두 복사본을 갖는다는 겁니다.

이 예는 각 CPU 가 자신의 오퍼레이션들을 프로그램 순서대로 행해지는 것으로 본다는 매우 중요한 보장을 깨버립니다. 이 보장을 깨는 것은 소프트웨어 종류에 있어 강력한 반직관이어서 하드웨어 사람들은 이에 공감하고 Figure C.6 에 보인 것처럼 각 CPU 가 로드할 때 캐시만이 아니라 스토어 버퍼도 참조하는 (또는 “snoop” 하는) “store forwarding” 을 구현했습니다. 달리 말하면, 특정 CPU 의 스토어는 뒤따르는 로드에 캐시를 거칠 필요 없이 곧바로 전달됩니다.

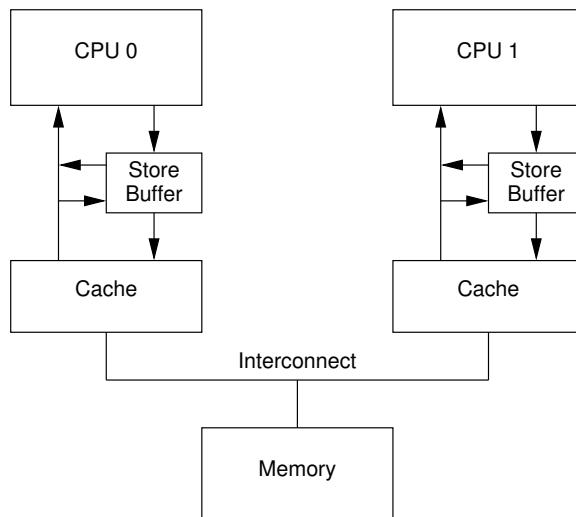


Figure C.6: Caches With Store Forwarding

Store forwarding 이 있다면, 앞의 흐름에서의 항목 8 은 스토어 버퍼 안의 “a” 의 값 1 을 보고, “b” 의 마지막 값은 사람들이 바라는대로 2 가 될 겁니다.

C.3.3 Store Buffers and Memory Barriers

두번째 복잡성인 전역 메모리 순서의 위배를 보기 위해 “a” 와 “b” 가 0 으로 초기화된 다음 코드 흐름을 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 이 `foo()` 를, CPU 1 은 `bar()` 를 수행한다고 해봅시다. 더 나아가 “a” 를 담는 캐쉬 라인은 CPU 1 의 캐쉬에만 있고 “b” 를 담는 캐쉬 라인은 CPU 0 에 의해 소유되어 있다고 해봅시다. 그러면 오퍼레이션들의 흐름은 다음과 같을 수도 있습니다:

- 1 CPU 0 이 `a = 1` 을 수행합니다. 이 캐쉬 라인은 CPU 0 의 캐쉬에 없으므로 CPU 0 은 “a” 의 새 값을 자신의 스토어 버퍼에 저장하고 “read invalidate” 메세지를 날립니다.
- 2 CPU 1 이 `while (b == 0) continue` 를 수행하지만, “b” 를 담는 캐쉬 라인은 자신의 캐쉬에 없습니다. 따라서 “read” 메세지를 보냅니다.
- 3 CPU 0 이 `b = 1` 을 수행합니다. 이 캐쉬 라인은 이미 소유하고 있으므로(달리 말하면 이 캐쉬 라인은 이미 “modified” 또는 “exclusive” 상태이므로), “b” 의 새 값을 캐쉬 라인에 저장합니다.
- 4 CPU 0 이 “read” 메세지를 받고, 이제 업데이트된 “b” 의 값을 CPU 1 에게 보내며, 자신의 캐쉬의 이 캐쉬 라인을 “shared” 상태로 만듭니다.
- 5 CPU 1 이 “b” 를 담는 캐쉬 라인을 받고 자신의 캐쉬에 이를 설치합니다.
- 6 CPU 1 은 이제 “b” 의 값이 1 임을 보게 되므로, `while (b == 0) continue` 를 마치고 다음 명령으로 넘어갑니다.
- 7 CPU 1 이 `assert(a == 1)` 을 수행하고, CPU 1 은 “a” 의 기존 값을 가지고 동작하고 있으므로 이 단정이 실패합니다.
- 8 CPU 1 이 “read invalidate” 메세지를 받고, “a” 를 담는 캐쉬 라인을 CPU 0 에게 보내고 자신의 캐쉬에서 이 캐쉬 라인을 무효화 합니다. 하지만 너무 늦었습니다.
- 9 CPU 0 은 “a” 를 담는 캐쉬 라인을 받고 버퍼에 저장된 스토어를 CPU 1 의 실패한 단정의 희생자가 되기 직전에 적용합니다.

Quick Quiz C.8: 앞의 step 1 에서, CPU 0 는 왜 단순한 “invalidate” 가 아닌 “read invalidate” 를 보내야 하죠?

■ 하드웨어 설계자는 이를 직접적으로 도울수는 없는 데, CPU 는 어떤 변수들이 연관되어 있는지 알 수 없기 때문입니다. 따라서, 하드웨어 설계자들은 소프트웨어가 CPU 에게 그런 관계를 말해줄 수 있도록 메모리 배리어 명령을 제공합니다. 이 프로그램은 메모리 배리어를 포함하게끔 업데이트 되어야만 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 }

```

이 메모리 배리어 `smp_mb()` 는 CPU 가 뒤따르는 스토어를 그 변수의 캐쉬 라인에 적용하기 전에 스토어 버퍼를 비우게 합니다. CPU 는 진행하기 전에 스토어 버퍼가 비워질 때까지 단순히 멈춰 있거나 스토어 버퍼의 기존 항목들이 적용되기 전까지는 스토어 버퍼가 뒤따르는 스토어를 중지시키게 할 수도 있습니다.

뒤의 방법의 경우 오퍼레이션들의 흐름은 다음과 같을 수도 있을 겁니다:

- 1 CPU 0 가 `a = 1` 을 수행합니다. 이 캐쉬 라인은 CPU 0 의 캐쉬에 없으므로, CPU 0 는 “a” 의 새 값을 자신의 스토어 버퍼에 넣고 “read invalidate” 메세지를 보냅니다.
- 2 CPU 1 이 `while (b == 0) continue` 를 수행합니다만, “b” 를 담는 캐쉬 라인은 자신의 캐쉬에 없습니다. 따라서 “read” 메세지를 보냅니다.
- 3 CPU 0 이 `smp_mb()` 를 수행하고, 모든 현재 스토어 버퍼 항목에 (구체적으로, `a = 1`) 표시를 합니다.
- 4 CPU 0 이 `b = 1` 을 수행합니다. 이는 자신의 캐쉬 라인에 이미 소유되어 있으나(달리 말하면, 이 캐쉬 라인은 “modified” 또는 “exclusive” 상태이므로) 스토어 버퍼에 표시된 항목이 있습니다. 따라서, “b” 의 새 값을 캐쉬라인에 저장하는 대신 이를 스토어 버퍼에 (그러나 표시 없는 항목으로) 넣습니다.
- 5 CPU 0 이 “read” 메세지를 받고 “b” 의 원래 값을 담는 캐쉬 라인을 CPU 1 에게 보냅니다. 이 CPU

- 는 또한 이 캐쉬 라인의 자신의 복사본을 “shared”로 표시합니다.
- 6 CPU 1 이 “b” 를 담는 캐쉬 라인을 받고 자신의 캐쉬라인에 설치합니다.
- 7 CPU 1 은 이제 “b” 의 값을 로드합니다만, “b” 의 값이 여전히 0 임을 확인하고 while 문을 반복합니다. “b” 의 새 값은 CPU 0 의 스토어 버퍼 안에 안전히 감춰져 있습니다.
- 8 CPU 1 은 “read invalidate” 메세지를 받고, “a” 를 담는 캐쉬 라인을 CPU 0 에게 보내고 자신의 캐쉬에서 이 캐쉬 라인을 무효화 합니다.
- 9 CPU 0 이 “a” 를 담는 캐쉬 라인을 받고 버퍼링 된 스토어를 적용하고, 이 라인을 “modified” 상태로 전환합니다.
- 10 “a” 로의 스토어가 이 스토어 버퍼 내의 `smp_mb()`에 의해 표시된 유일한 항목이었으므로, CPU 0 은 “b” 의 새 값도 저장할 수 있습니다—“b” 를 담는 캐쉬 라인이 지금은 “shared” 상태라는 사실을 제외하면요.
- 11 따라서 CPU 0 은 “invalidate” 메세지를 CPU 1 에게 보냅니다.
- 12 CPU 1 은 “invalidate” 메세지를 받고 “b” 를 담는 캐쉬라인을 자신의 캐쉬에서 무효화 시키고, CPU 0 에게 “acknowledgement” 메세지를 보냅니다.
- 13 CPU 1 이 `while (b == 0) continue` 를 수행합니다만, “b” 를 담는 캐쉬 라인은 자신의 캐쉬에 없습니다. 따라서 CPU 0 에게 “read” 메세지를 보냅니다.
- 14 CPU 0 이 “acknowledgement” 메세지를 받고, “b” 를 담는 캐쉬라인을 “exclusive” 상태로 놓습니다. CPU 0 은 이제 “b” 의 새 값을 캐쉬 라인에 저장합니다.
- 15 CPU 0 이 “read” 메세지를 받고, “b” 의 새 값을 담은 캐쉬 라인을 CPU 1 에게 보냅니다. 또한 이 캐쉬 라인의 자신의 복사본을 “shared” 로 표시합니다.
- 16 CPU 1 이 “b” 를 담는 캐쉬 라인을 받고 자신의 캐쉬에 설치합니다.
- 17 CPU 1 은 이제 “b” 의 값을 로드할 수 있으며, “b” 의 값이 1 임을 확인하므로, while 반복문을 종료하고 다음 문장으로 진행합니다.
- 18 CPU 1 은 `assert(a == 1)` 을 수행하지만, “a” 를 담는 캐쉬라인은 자신의 캐쉬에 더이상 존재하지 않습니다. 이 캐쉬라인을 CPU 0 으로부터 받으면, 이는 최신의 “a” 의 값을 가지고 있을 것이며, 따라서 이 단정은 통과됩니다.

Quick Quiz C.9: Page 409 의 Appendix C.3.3 의 step 15 뒤에, 두 CPU 가 “b” 의 새 값을 담은 캐쉬라인을 버릴 수도 있을 겁니다. 이게 이 새로운 값을 잃어버리게 할 수 있을까요?

■ 이로써 알 수 있듯, 이 프로세스는 적지 않은 표시해두기를 필요로 합니다. 어떤건 직관적으로 간단할지라도, “a 의 값을 로드하라” 같은게 실리콘 내에선 복잡한 여러 단계를 필요로 할 수 있습니다.

C.4 Store Sequences Result in Unnecessary Stalls

불행히도, 각 스토어 버퍼는 상대적으로 작아야 하는데, 이는 약간의 스토어들을 수행하는 CPU 도 자신의 스토어 버퍼를 꽉 채울 수 있음을 의미합니다(예를 들어, 그 것들 모두가 캐쉬 미스를 초래한다면). 이 경우, CPU 는 수행을 계속하기 전에 자신의 스토어 버퍼가 비워지게끔 캐쉬라인 무효화가 완료되기를 기다려야만 합니다. 이 똑같은 상황이 메모리 배리어 뒤에 일어날 수 있는데, 모든 뒤따르는 스토어 명령이 이 스토어들이 캐쉬 미스를 초래하는가에 관계 없이 무효화가 완료되기를 기다려야만 하는 겁니다.

이 상황은 무효화 응답 메세지가 더 빨리 도착하게 함으로써 개선될 수 있습니다. 이를 달성하는 한가지 방법은 무효화 메세지의 per-CPU queue, 또는 “invalidate queue” 를 사용하는 겁니다.

C.4.1 Invalidate Queues

무효화 응답 메세지가 긴 시간을 취하게 되는 한가지 이유는 연관된 캐쉬 라인이 실제로 무효화 되었음을 보장해야 하며, 예를 들어 이 CPU 가 캐쉬에 존재하는 데이터의 상당한 로드와 스토어를 해서 캐쉬가 바쁘면 이 무효화는 지연될 수 있다는 것입니다. 또한, 큰 수의 무효화 메세지가 짧은 시간에 도착하면, 해당 CPU 는 그것을 처리하는데 바빠져서 다른 CPU 들을 멈춰있게 할 수 있습니다.

그러나, CPU 는 응답을 보내기 전에 정말로 캐쉬 라인을 무효화 할 필요는 없습니다. 그대신 이 CPU 가 이 캐쉬 라인에 연관된 어떤 메세지를 나중에 보내기 전에 이 무효화 메세지는 처리될 거라는 이해 하에 이 무효화 메세지를 queue 에 저장해 둘 수 있습니다.

C.4.2 Invalidate Queues and Invalidate Acknowledge

Figure C.7 는 invalidate queue 를 갖는 시스템을 보입니다. Invalidate queue 를 갖는 CPU 는 invalidate 메세지에 그에 연관된 라인이 정말로 무효화 되기를 기다리는 대신 그 메세지가 queue 에 들어가자마자 응답을 할 수 있습니다. 물론, 이 CPU 는 invalidate 메세지 송신을 준비할 때 자신의 invalidate queue 를 참조해야만 합니다—만약 연관된 캐시 라인을 위한 항목이 invalidate queue 에 있다면 그 CPU 는 이 invalidate message 를 즉각 보낼 수 없습니다; 그대신 이 invalidate queue 항목이 처리될 때까지 기다려야만 합니다.

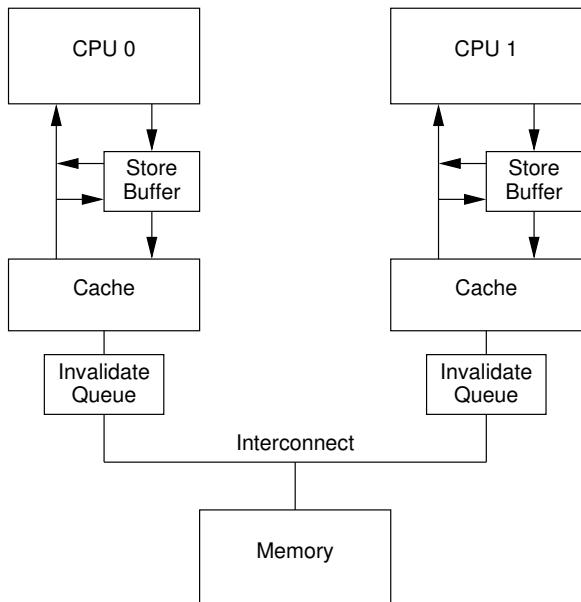


Figure C.7: Caches With Invalidate Queues

Invalidate queue 내에 항목을 넣는 것은 기본적으로 이 CPU 가 그 항목을 거기 연관된 어떠한 MESI 프로토콜 메세지를 보내기 전에는 처리할 것이라는 약속입니다. 연관된 데이터 구조에 대한 경쟁이 치열한 동안은 그 CPU 는 그런 약속으로 인한 문제를 드물게만 겪을 겁니다.

그러나, invalidate message 가 invalidate queue 내에 있을 수 있다는 사실은 추가적인 메모리 순서 잘못 맞추기 기회를 자아내는데, 다음 섹션에서 이를 다룹니다.

C.4.3 Invalidate Queues and Memory Barriers

CPU 들이 invalidate 요청을 queue 에 넣지만 즉각 응답한다고 해봅시다. 이 방법은 스토어를 하는 CPU 에 보이는 캐시 무효화 응답시간을 최소화 시키지만, 다음 예에서 볼 수 있듯 메모리 배리어를 망가뜨릴 수 있습니다.

“a” 와 “b” 의 값이 초기에 0이며, “a” 는 읽기 전용 복사본이며 (MESI “shared” 상태), “b” 는 CPU 0 에 의해 소유되어 있다고 (MESI “exclusive” 또는 “modified” 상태) 해봅시다. 그리고 CPU 0 이 다음 코드의 `foo()` 를, CPU 1 은 `bar()` 를 수행한다고 해봅시다:

```

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 }
```

그러면 오퍼레이션의 흐름은 다음과 같을 겁니다:

- 1 CPU 0 이 `a = 1` 을 수행합니다. 연관된 캐시라인은 CPU 0 의 캐시에 읽기전용으로 있으므로, CPU 0 은 “a” 의 새 값을 자신의 스토어 버퍼에 넣고 연관된 캐시 라인을 CPU 1 의 캐시에서 비워내기 위해 “invalidate” 메세지를 보냅니다.
- 2 CPU 1 이 `while (b == 0) continue` 를 수행합니다만, “b” 를 담는 캐시 라인은 자신의 캐시에 없습니다. 따라서 “read” 메세지를 보냅니다.
- 3 CPU 1 이 CPU 0 의 “invalidate” 메세지를 받고 queue 에 이를 넣고, 즉각 응답합니다.
- 4 CPU 0 이 CPU 1 로부터의 응답을 받고, 따라서 라인 4 의 `smp_mb()` 로 진행할 수 있으므로, “a” 의 값을 스토어 버퍼에서 자신의 캐시라인으로 옮깁니다.
- 5 CPU 0 이 “read” 메세지를 받고, 이제 업데이트 된 “b” 의 값을 담는 캐시 라인을 CPU 1 에게 보내며, 이 라인을 자신의 캐시에 “shared” 로 표시합니다.
- 6 CPU 1 이 “b” 를 담는 캐시라인을 받고 자신의 캐시에 이를 설치합니다.

- 7 CPU 1 은 이제 `while (b == 0) continue` 를 끌 낼 수 있고, “b” 의 값이 1 임을 확인하므로, 다음 문장으로 넘어갑니다.
- 8 CPU 1 이 `assert(a == 1)` 을 수행하고, “a” 의 기존 값이 여전히 CPU 1 의 캐쉬에 있으므로, 이 단정은 실패합니다.
- 9 이 단정의 실패에도 불구하고 CPU 1 은 `queue` 의 “invalidate” 메세지를 수행하고, (느리게) “a” 를 담는 캐쉬 라인을 자신의 캐쉬에서 무효화 시킵니다.

Quick Quiz C.10: Appendix C.4.3 의 첫번째 시나리오에서의 step 1 에서, 왜 “read invalidate” 대신 “invalidate” 메세지가 보내지죠? CPU 0 은 “a” 와 캐쉬 라인을 공유하는 다른 변수의 값이 필요하지 않을까요?

■ 무효화 응답을 빨리 하는게 메모리 배리어를 실질적으로 무시되게 할 수 있다면 그 가속화는 의미가 없습니다. 그러나, 메모리 배리어 명령은 특정 CPU 가 메모리 배리어를 수행했을 때 `invalidate queue` 에 있는 모든 항목을 표시하고 뒤따르는 모든 로드가 이 표시된 항목들이 이 CPU 의 캐쉬에 적용되기까지 기다리게끔 `invalidate queue` 와 상호작용할 수 있습니다. 따라서, 우린 `bar` 에 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 C.11: 뭐라고요??? 이 CPU 는 `while` 반복문이 완료되기 전까지는 `assert()` 를 수행할 수 없는데 메모리 배리어가 왜 필요하죠?

■ 이 변경이 있으면, 오퍼레이션의 흐름은 다음과 같아집니다:

- 1 CPU 0 이 `a = 1` 을 수행합니다. 연관된 캐쉬 라인은 CPU 0 의 캐쉬에 읽기 전용으로 있으므로, CPU 0 은 “a” 의 새 값을 스토어 버퍼에 넣고 연관된 캐쉬 라인을 CPU 1 의 캐쉬에서 비워내기 위해 “invalidate” 메세지를 보냅니다.

- 2 CPU 1 이 `while (b == 0) continue` 를 수행하지만, “b” 를 담는 캐쉬 라인은 자신의 캐쉬에 없습니다. 따라서 “read” 메세지를 보냅니다.
- 3 CPU 1 이 CPU 0 의 “invalidate” 메세지를 받고 `queue` 에 이를 넣고, 곧바로 응답합니다.
- 4 CPU 0 이 CPU 1 의 응답을 받고, 따라서 라인 4 의 `smp_mb()` 뒤로 수행을 이어갈 자유를 얻게 되어 “a” 의 값을 스토어 버퍼에서 캐쉬라인으로 옮깁니다.
- 5 CPU 0 이 `b = 1` 을 수행합니다. 이 캐쉬 라인을 이미 소유하고 있으므로 (달리 말하자면, 이 캐쉬 라인은 “modified” 또는 “exclusive” 상태입니다), “b” 의 새 값을 자신의 캐쉬라인에 저장합니다.
- 6 CPU 0 이 “read” 메세지를 받고, 이제 업데이트된 “b” 의 값을 담는 캐쉬라인을 CPU 1 에게 보내며, 이 라인을 자신의 캐쉬에 “shared” 로 표시합니다.
- 7 CPU 1 이 “b” 를 담는 캐쉬 라인을 받고 자신의 캐쉬에 이를 설치합니다.
- 8 CPU 1 은 이제 `while (b == 0) continue` 의 수행을 마치고, “b” 의 값이 1 임을 확인하므로, 이제는 메모리 배리어인 다음 문장으로 진행합니다.
- 9 CPU 1 은 이제 자신의 `invalidation queue` 에 있던 기존부터 존재한 모든 메세지가 처리될 때까지 멈춰 있어야만 합니다.
- 10 CPU 1 이 이제 `queue` 에 있던 “invalidate” 메세지를 처리해서 “a” 를 담고 있는 캐쉬 라인을 자신의 캐쉬에서 무효화 합니다.
- 11 CPU 1 이 `assert(a == 1)` 을 수행하는데, “a” 를 담는 이 캐쉬라인은 더이상 CPU 1 의 캐쉬에 없으므로 “read” 메세지를 보냅니다.
- 12 CPU 0 이 이 “read” 메세지에 “a” 의 새 값을 담는 캐쉬 라인으로 응답합니다.
- 13 CPU 1 이 “a” 의 값 1 을 담는 이 캐쉬라인을 받고, 따라서 이 단정문은 격발되지 않습니다.

훨씬 많은 MESI 메세지와 함께, 이 CPU 는 마침내 올바른 답을 찾았습니다. 이 셋션은 왜 CPU 설계자들이 그들의 캐쉬 일관성 최적화에 매우 조심스러워야 하는지 보입니다.

C.5 Read and Write Memory Barriers

앞의 섹션에서, 메모리 배리어는 스토어 버퍼와 invalidate queue 모두의 항목들을 표시하는데 사용되었습니다. 하지만 우리의 코드에서, `foo()` 는 invalidate queue에 어떤 일을 할 이유가 없고, `bar()` 는 비슷하게 스토어 버퍼에 무슨 일을 할 이유가 없습니다.

따라서 많은 CPU 구조는 이 두개 중 하나에만 일을 하는 더 완화된 메모리 배리어 명령을 제공합니다. 거칠게 말해서, “읽기 메모리 배리어 (read memory barrier)”는 invalidate queue 만을 표시하고 “쓰기 메모리 배리어 (write memory barrier)”는 스토어 버퍼만을 표시하며, 완전한 메모리 배리어는 두 일을 모두 합니다.

이것의 효과는 읽기 메모리 배리어는 로드들만이 이를 수행한 CPU 들에서는 순서가 잡히게 만들어줘서 읽기 메모리 배리어를 앞서는 로드들은 이 읽기 메모리 배리어를 뒤따르는 로드들보다 먼저 완료된 것으로 보이게 하는 겁니다. 비슷하게, 쓰기 메모리 배리어는 스토만을 순서잡는데, 역시 이를 수행한 CPU 에서만 그려하며, 역시 쓰기 메모리 배리어를 앞서는 스토어들이 이를 뒤따르는 스토어들보다 먼저 완료된 것으로 보이게 합니다. 완전한 메모리 배리어는 로드와 스토어를 모두 순서잡지만, 이 역시 이 메모리 배리어를 수행하는 CPU 에서만 그렇습니다.

`foo()` 와 `bar()` 를 읽기/쓰기 메모리 배리어를 사용하게 바꾸면 다음과 같을 겁니다:

```

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 }
```

어떤 컴퓨터들은 이보다도 많은 종류의 메모리 배리어를 갖습니다만, 이 세개의 변종에 대한 이해는 일반적인 메모리 배리어에 대한 좋은 소개가 될 겁니다.

C.6 Example Memory-Barrier Sequences

이 섹션은 매력적이지만 묘하게 망가진 메모리 배리어 사용들을 선보입니다. 이들 중 대다수가 대부분의 경우

동작하지만, 그리고 일부는 특정 CPU 에서는 항상 동작하지만, 목표가 모든 CPU 에서 안정적으로 동작하는 거라면 이런 사용은 금지되어야만 합니다. 이 교묘한 망가짐을 더 잘 이해하기 위해, 먼저 순서 잡기에 적대적인 구조에 집중합니다.

C.6.1 Ordering-Hostile Architecture

여러 순서잡기에 적대적인 컴퓨터 시스템들이 수십년간 만들어졌습니다만, 적대감의 본질은 항상 극단적으로 교묘했고, 그것을 이해하는 데에는 특정 하드웨어에 대한 자세한 이해가 필요했습니다. 특정 하드웨어 벤더를 고르는 대신, 그리고 아마도 독자 여러분을 자세한 기술 명세로 의사시키는 것에 대한 매력적인 대안으로써, 가상의 것이지만 최대한 메모리 순서잡기에 적대적인 컴퓨터 구조를 설계합시다.⁴

이 하드웨어는 다음 순서잡기 제한을 따라야만 합니다 [McK05a, McK05b]:

1. 각 CPU 는 자신의 메모리 액세스를 항상 프로그램 순서대로 인지합니다.
2. CPU 는 스토어와 어떤 오퍼레이션의 순서를 바꿀 수 있는데, 두 오퍼레이션이 다른 지역을 참조할 때만 그렇습니다.
3. 특정 CPU 의 읽기 메모리 배리어 (`smp_rmb()`) 를 앞서는 모든 로드는 모든 CPU 에게 이 읽기 메모리 배리어를 뒤따르는 로드를 앞서는 것으로 보이게 됩니다.
4. 특정 CPU 의 쓰기 메모리 배리어 (`smp_wmb()`) 를 앞서는 모든 스토어는 모든 CPU 에게 이 쓰기 메모리 배리어를 뒤따르는 스토어를 앞서는 것으로 보이게 됩니다.
5. 특정 CPU 의 완전한 메모리 배리어 (`smp_mb()`) 를 앞서는 모든 액세스 (로드와 스토어) 는 모든 CPU 에게 이 메모리 배리어를 뒤따르는 모든 액세스를 앞서는 것으로 보이게 됩니다.

Quick Quiz C.12: 각 CPU 각 자신의 메모리 액세스를 순서대로 본다는 보장은 각 사용자 수준 쓰레드가 자신의 메모리 액세스를 순서대로 본다는 것을 보장하나요? 그 이유는요?



⁴ 실제 하드웨어 구조에 대한 자세한 이해를 선호하는 독자 여러분은 CPU 제조사의 매뉴얼들 [SW95, Adv02, Int02b, IBM94, LHF05, SPA94, Int04b, Int04a, Int04c], Charachorloo 의 학위 논문 [Gha95], Peter Sewell 의 작업물 [Sew], 또는 Sorin, Hill, 그리고 Wood 에 의해 작성된 훌륭한 하드웨어 기반 입문서 [SHW11] 를 보시기 바랍니다.

거대한 non-uniform cache architecture (NUCA) 시스템을 상상해 봅시다. 이 시스템은 공정한 접합부 (inter-connect) 대역폭을 동일 노드 내의 CPU들에게 공정하게 할당하기 위해서 Figure C.8에 보인 것처럼 각 노드의 접합부 인터페이스에 per-CPU queue를 제공합니다. 특정 CPU의 액세스는 그 CPU에 의해 수행된 메모리 배리어에 의해 명세된 순서대로 일어나지만, CPU들의 액세스들의 상대적인 순서는 상당히 재배치 될 수 있는데, 곧이어 보게 될 겁니다.⁵

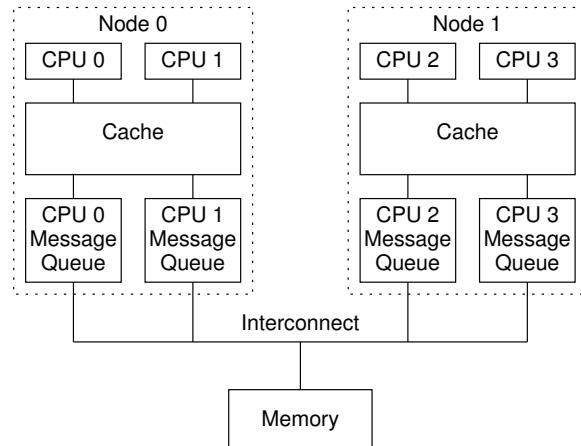


Figure C.8: Example Ordering-Hostile Architecture

C.6.2 Example 1

Listing C.1은 CPU 0, 1, 그리고 2에 의해 동시에 수행되는 코드를 보입니다. “a”, “b”, 그리고 “c”는 초기에 0 값을 갖습니다.

Listing C.1: Memory Barrier Example 1

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>

CPU 0이 최근에 많은 캐시 미스를 경험했으며, 따라서 그것의 메세지 queue는 꽉 차 있지만, CPU 1은 이 캐시에 배타적으로 동작했으며, 따라서 그것의 메세지 queue는 비어 있다고 해봅시다. 그러면 CPU 0의 “a”와

⁵ 모든 실제 하드웨어 설계자는 필사적으로 이에 반발할 텐데, 어떤 queue가 두 CPU가 액세스하는 캐시라인에 연관된 메세지를 처리해야 할지에 대한 가능성이 그들을 화나게 할 것이기 때문입니다. 제가 말할 수 있는건 “더 나은 예를 주세요”입니다.

“b”에의 할당은 노드 0의 캐시에 즉각적으로 나타나지만 (그리고 따라서 CPU 1에게 보이게 됨), CPU 0의 앞의 교통량에 의해 막혀있게 될 겁니다. 따라서, CPU 2는 CPU 1의 “c”에의 할당을 CPU 0의 “a”로의 할당 전에 볼 수 있어서 메모리 배리어에도 불구하고 단정문이 격발되게 할 수 있습니다.

따라서, 이식성 있는 코드는 이 단정문이 격발되지 않을 것이라 가정해선 안되는데 컴파일러와 CPU는 이 코드를 재배치 할 수 있기 때문입니다.

Quick Quiz C.13: 이 코드는 CPU 1의 “while”과 “c”에의 값 할당 사이에 메모리 배리어를 넣음으로써 고쳐질 수 있을까요? 왜죠?

■

C.6.3 Example 2

Listing C.2는 CPU 0, 1, 그리고 2에 의해 동시에 수행되는 세개의 코드 조각을 보입니다. “a”와 “b”는 둘다 초기에 0입니다.

Listing C.2: Memory Barrier Example 2

CPU 0	CPU 1	CPU 2
<pre>a = 1;</pre>	<pre>while (a == 0); smp_mb();</pre>	<pre>y = b;</pre>

```
b = 1;
```

```
smp_rmb();
```

```
x = a;
```

```
assert(y == 0 || x == 1);
```

역시, CPU 0이 최근에 많은 캐시 미스를 경험했고, 따라서 message queue가 꽉 차 있지만 CPU 1은 자신의 캐시에서 배타적으로 수행되었기 때문에 message queue가 비어 있다고 해봅시다. 그러면 CPU 0의 “a”에의 값 할당은 Node 0의 캐시에 즉각적으로 나타나지만 (그리고 따라서 CPU 1에게 보입니다), CPU 0의 앞의 교통량에 의해 막힐 겁니다. 대조적으로, CPU 1의 “b” 값 할당은 CPU 1의 비어있는 queue를 통해 나아갈 겁니다. 따라서, CPU 2는 CPU 1의 “b”로의 값 할당을 CPU 0의 “a”에의 값 할당보다 먼저 봐서 메모리 배리어에도 불구하고 이 단정문이 격발되게 할 수 있습니다.

이론상으로, 이식 가능한 코드는 이 예제 코드 조각에 기대선 안됩니다만, 앞에서와 같이 실전에서는 대부분의 주류 컴퓨터 시스템에서 동작합니다.

C.6.4 Example 3

Listing C.3는 CPU 0, 1, 그리고 2에 의해 동시에 수행되는 세개의 코드 조각을 보입니다. 모든 변수는 초기에 0입니다.

CPU 1도 CPU 2도 CPU 0의 라인 3에서의 “b”로의 값 할당을 보기 전까지는 라인 5로 진행하지 못합니다.

Listing C.3: Memory Barrier Example 3

	CPU 0	CPU 1	CPU 2
1	a = 1;		
2	smp_wmb();		
3	b = 1;	while (b == 0);	while (b == 0);
4		smp_mb();	smp_mb();
5		c = 1;	d = 1;
6	while (c == 0);		
7	while (d == 0);		
8	smp_mb();		
9	e = 1;		assert(e == 0 a == 1);

일단 CPU 1 과 2 가 라인 4에서 각자의 메모리 배리어를 수행하면, 그들은 라인 2에서의 메모리 배리어를 앞서는 CPU 0에 의한 모든 값 할당을 볼 것이 보장됩니다. 비슷하게, CPU 0의 라인 8에서의 메모리 배리어는 라인 4의 CPU 1과 2의 것들과 짹을 이루므로, CPU 0은 라인 9에서의 “e”로의 값 할당을 그것의 “b”로의 값 할당이 다른 두 CPU에게 보여지기 전까지는 수행하지 않을 겁니다. 따라서, CPU 2의 라인 9에서의 단정문은 격발되지 않을 것이 보장됩니다.

Quick Quiz C.14: Listing C.3의 CPU 1과 2에 의한 라인 3~5가 인터럽트 핸들러이고, CPU 2의 라인 9는 프로세스 수준에서 수행된다고 해봅시다. 달리 말하자면, 이 표의 세 열의 코드는 같은 CPU에서 수행되나, 첫번째 두 열은 인터럽트 핸들러에서, 그리고 세번째 열은 프로세스 수준에서 수행되므로, 세번째 열의 코드는 첫번째 열의 코드에 의해 인터럽트 당할 수 있습니다. 이 코드가 올바르게 동작하려면, 달리 말해 이 단정문이 격발되는 걸 막기 위해 무언가가 필요할까요?

Quick Quiz C.15: Listing C.3의 예에서 CPU 2가 `assert(e==0 || c==1)`를 수행했다면, 이 단정문은 격발될까요?

리눅스 커널의 `synchronize_rcu()` 기능은 이 예에서 보인 것과 비슷한 알고리즘을 사용합니다.

C.7 Are Memory Barriers Forever?

일반적으로 순서 외 수행(out-of-order)에, 특히 메모리 참조 재배치에 있어서 무척 떨 적극적인 최근의 시스템이 여럿 있었습니다. 이 추세는 메모리 배리어가 과거의 것이 되는 지점까지 계속될까요?

이에 긍정하는 주장은 각 쓰레드는 메모리가 준비되기까지 수십, 수백, 또는 심지어 수천의 다른 쓰레드가 그사이 진행을 할동안 기다리는, 제안된 상당한 규모의 멀티 쓰레드 하드웨어 구조를 인용할 겁니다. 그런 구조에서는 메모리 배리어가 필요치 않을 텐데, 특정 쓰레드는 다음 명령을 수행하기 전에 그저 모든 다른 오퍼레이션의 완료되기를 기다리면 되기 때문입니다. 잠재적으로

로 수천의 다른 쓰레드가 있으므로, 이 CPU는 완전히 사용되므로 어떤 CPU 시간도 낭비되지 않습니다.

이에 대항하는 주장은 수천 쓰레드까지 확장되기 적합한 어플리케이션의 굉장히 제한된 수와 일부 어플리케이션에서는 수십 마이크로세컨드에 달하는, 점점 증가하는 리얼타임 요구사항을 인용할 겁니다. 리얼타임 응답 요구사항은 그대로 맞추기가 충분히 어려우며, 거대 멀티 쓰레드 시나리오에서 암시되는 무척 낮은 단일 쓰레드 처리량에서는 더욱 맞추기 어려울 겁니다.

또 다른 긍정하는 주장은 CPU가 순서 외 수행의 성능 장점을 거의 모두 제공하면서도 완전히 순차적인 일관된 수행의 환상을 제공하는 CPU를 가능하게 할수도 있는 점점 증가하는 세련된 응답시간을 감추는 하드웨어 구현을 인용할 겁니다. 이에 대항하는 주장은 배터리로 동작하는 기기들과 환경에 대한 책임에서 비롯되는 점점 증가하는 강력한 전력 효율성 요구사항을 인용할 겁니다.

누가 맞을까요? 우린 증거가 없으므로, 모든 시나리오에서도 살아남기 위해 준비합니다.

C.8 Advice to Hardware Designers

소프트웨어 사람들의 삶을 어렵게 하기 위해 하드웨어 설계자들이 할 수 있는 일들이 여럿 있습니다. 여기 우리가 과거에 마주한 그런 것들 일부를 미래에는 그런 문제를 방지하기 둑길 바라는 마음으로 나열합니다:

1. 캐쉬 일관성을 무시하는 I/O 기기들.

이 매력적인 잘못된 기능은 메모리로부터의 DMA들이 출력 버퍼로의 최근의 변경을 놓치게하거나 그만큼이나 나쁘게도 입력 버퍼가 DMA 완료 직후의 CPU 캐쉬의 내용으로 덮어써지게 할 수 있습니다. 여러분이 그런 잘못된 동작에서도 동작하게 하려면, 여러분은 주의 깊게 DMA 버퍼의 모든 위치에 대한 CPU 캐쉬를 그 버퍼를 I/O 기기에게 보여주기 전에 비워야만 합니다. 비슷하게, 모든 DMA 버퍼의 모든 위치에 대한 CPU 캐쉬를 그 버퍼로의 DMA가 완료된 후에 비워내야 합니다.

심지어 그리고 나서도, 여러분은 포인터 버그를 매우 조심해야 하는데, 잘못 위치된 입력 버퍼로의 읽기는 데이터 입력을 오염시키는 결과를 초래할 수 있기 때문입니다!

2. 캐쉬 일관성 데이터를 전달하는데 실패하는 외부 버스들.

이는 앞의 문제보다도 더 고통스러운, 그러나 여러 기기들을—그리고 심지어 메모리 자체를—캐쉬 일관성을 무시하게 만들 수 있는 변종입니다. 임베디드 시스템들이 멀티코어 구조로 옮겨가고 있으므로 우리는 이런 문제들을 상당후 보게 될 것에 의심의 여지가 없음을 알리는 건 저의 고통스러운 의무입니다. 2021년 기준, 이런 문제를 새로 운 인터컨넷 표준으로 해결하려는 노력이 일부 있었는데, 이 표준이 정말로 얼마나 효과적일지에 대해서는 일부 논란이 있습니다 [Won19].

3. 캐쉬 일관성을 무시하는 기기 인터럽트.

이는 충분히 문제없는 듯 들릴 수도 있습니다—어쨌건, 인터럽트는 메모리 참조가 아닙니다, 안그래요? 하지만 분리된 캐쉬를 가지며 한쪽은 무척 바빠서 입력 버퍼의 마지막 캐쉬라인을 잡고 있는 CPU를 떠올려 봅시다. 연관된 I/O-완료 인터럽트가 이 CPU에 도착하면, 이 CPU의 마지막 캐쉬라인으로의 메모리 참조는 예전 데이터를 반환할 수 있어서, 다시 데이터 오염을 초래하지만 뒤따르는 crash dump에는 보이지 않을 겁니다. 시스템이 문제된 입력 버퍼를 덤플링할 시점에는 이 DMA는 대부분의 경우 이미 완료되었을 겁니다.

4. 캐쉬 일관성을 무시하는 프로세서간 인터럽트 (inter-processor interrupts: IPIs).

이는 IPI가 연관된 메세지 버퍼의 캐쉬 라인이 모두 메모리에 넘어가기 전에 목적지에 도착하면 문제 가 될 수 있습니다.

5. 캐쉬 일관성보다 먼저 이루어지는 컨텍스트 전환.

메모리 액세스가 지나치게 순서 없이 완료된다면, 컨텍스트 전환은 상당히 비참해질 수 있습니다. 태스크가 한 CPU에서 다른 쪽으로 원천 CPU에게 보이는 메모리 액세스가 목적 CPU에도 보이게 하기 전에 옮겨간다면 이 태스크는 연관된 변수들이 이전 값으로 돌아간 것을 보게 될텐데, 이는 대부분의 알고리즘을 크게 혼란시킬 수 있습니다.

6. 지나치게 친절한 시뮬레이터와 에뮬레이터.

메모리 재배치를 강제하는 시뮬레이터나 에뮬레이터를 작성하길 어려우므로, 이 환경에서 그냥 잘 동작하는 소프트웨어는 실제 하드웨어에서 처음

수행될 때 불쾌한 놀랄을 받을 수 있습니다. 불행히도, 하드웨어는 시뮬레이터나 에뮬레이터보다 더 일탈적인 규칙입니다만, 이 상태가 바뀌길 바랍니다.

반복합니다, 우린 하드웨어 설계자가 이런 일을 하지 않길 장려합니다!

Appendix D

Style Guide

This appendix is a collection of style guides which is intended as a reference to improve consistency in perfbook. It also contains several suggestions and their experimental examples.

Appendix D.1 describes basic punctuation and spelling rules. Appendix D.2 explains rules related to unit symbols. Appendix D.3 summarizes \LaTeX -specific conventions.

D.1 Paul’s Conventions

Following is the list of Paul’s conventions assembled from his answers to Akira’s questions regarding perfbook’s punctuation policy.

- (On punctuations and quotations) Despite being American myself, for this sort of book, the UK approach is better because it removes ambiguities like the following:

Type “`ls -a`,” look for the file “`..`,” and file a bug if you don’t see it.

The following is much more clear:

Type “`ls -a`”, look for the file “`..`”, and file a bug if you don’t see it.

- American English spelling: “color” rather than “colour”.
- Oxford comma: “a, b, and c” rather than “a, b and c”. This is arbitrary. Cases where the Oxford comma results in ambiguity should be reworded, for example, by introducing numbering: “a, b, and c and d” should be “(1) a, (2) b, and (3) c and d”.
- Italic for emphasis. Use sparingly.

- `\code{}` for identifiers, `\url{}` for URLs, `\path{}` for filenames.
- Dates should use an unambiguous format. Never “mm/dd/yy” or “dd/mm/yy”, but rather “July 26, 2016” or “26 July 2016” or “26-Jul-2016” or “2016/07/26”. I tend to use `yyyy . mm . ddA` for filenames, for example.
- North American rules on periods and abbreviations. For example neither of the following can reasonably be interpreted as two sentences:

- Say hello, to Mr. Jones.
- If it looks like she sprained her ankle, call Dr. Smith and then tell her to keep the ankle iced and elevated.

An ambiguous example:

If I take the cow, the pig, the horse, etc.
George will be upset.

can be written with more words:

If I take the cow, the pig, the horse, or
much of anything else, George will be
upset.

or:

If I take the cow, the pig, the horse, etc.,
George will be upset.

- I don’t like ampersand (“&”) in headings, but will sometimes use it if doing so prevents a line break in that heading.
- When mentioning words, I use quotations. When introducing a new word, I use `\emph{}`.

Following is a convention regarding punctuation in L^AT_EX sources.

- Place a newline after a colon (:) and the end of a sentence. This avoids the whole one-space/two-space food fight and also has the advantage of more clearly showing changes to single sentences in the middle of long paragraphs.

D.2 NIST Style Guide

D.2.1 Unit Symbol

D.2.1.1 SI Unit Symbol

NIST style guide¹ states the following rules (rephrased for perfbook).

- When SI unit symbols such as “ns”, “MHz”, and “K” (kelvin) are used behind numerical values, narrow spaces should be placed between the values and the symbols.

A narrow space can be coded in L^AT_EX by the sequence of “\,”. For example,

“2.4 GHz”, rather than “2.4GHz”.

- Even when the value is used in adjectival sense, a narrow space should be placed. For example,

“a 10 ms interval”, rather than “a 10-ms interval” nor “a 10ms interval”.

The symbol of micro (μ : 10^{-6}) can be typeset easily by the help of “gensymb” L^AT_EX package. A macro “\micro” can be used in both text and math modes. To typeset the symbol of “microsecond”, you can do so by “\micro s”. For example,

10 μ s

Note that math mode “\mu” is italic by default and should not be used as a prefix. An improper example:

10 μ s (math mode “\mu”)

D.2.1.2 Non-SI Unit Symbol

Although NIST style guide does not cover non-SI unit symbols such as “KB”, “MB”, and “GB”, the same rule should be followed.

Example:

“A 240 GB hard drive”, rather than “a 240-GB hard drive” nor “a 240GB hard drive”.

Strictly speaking, NIST guide requires us to use the binary prefixes “Ki”, “Mi”, or “Gi” to represent powers of 2^{10} . However, we accept the JEDEC conventions to use “K”, “M”, and “G” as binary prefixes in describing memory capacity.²

An acceptable example:

“8 GB of main memory”, meaning “8 GiB of main memory”.

Also, it is acceptable to use just “K”, “M”, or “G” as abbreviations appended to a numerical value, e.g., “4K entries”. In such cases, no space before an abbreviation is required. For example,

“8K entries”, rather than “8 K entries”.

If you put a space in between, the symbol looks like a unit symbol and is confusing. Note that “K” and “k” represent 2^{10} and 10^3 , respectively. “M” can represent either 2^{20} or 10^6 , and “G” can represent either 2^{30} or 10^9 . These ambiguities should not be confusing in discussing approximate order.

D.2.1.3 Degree Symbol

The angular-degree symbol (°) does not require any space in front of it. NIST style guide clearly states so.

The symbol of degree can also be typeset easily by the help of gensymb package. A macro “\degree” can be used in both text and math modes.

Example:

45°, rather than 45 °.

D.2.1.4 Percent Symbol

NIST style guide treats the percent symbol (%) as the same as SI unit symbols.

50 % possibility, rather than 50% possibility.

¹ <https://www.nist.gov/pml/nist-guide-si-chapter-7-rules-and-style-conventions-expressing-values-quantities>

² <https://www.jedec.org/standards-documents/dictionary/terms/mega-m-prefix-units-semiconductor-storage-capacity>

D.2.1.5 Font Style

Quote from NIST check list:³

Variables and quantity symbols are in italic type. Unit symbols are in roman type. Numbers should generally be written in roman type. These rules apply irrespective of the typeface used in the surrounding text.

For example,

e (elementary charge)

On the other hand, mathematical constants such as the base of natural logarithms should be roman.⁴ For example,

e^x

D.2.2 NIST Guide Yet To Be Followed

There are a few cases where NIST style guide is not followed. Other English conventions are followed in such cases.

D.2.2.1 Digit Grouping

Quote from NIST checklist:⁵

The digits of numerical values having more than four digits on either side of the decimal marker are separated into groups of three using a thin, fixed space counting from both the left and right of the decimal marker. Commas are not used to separate digits into groups of three.

NIST Example: 15 739.012 53 ms

Our convention: 15,739.01253 ms

In LATEX coding, it is cumbersome to place thin spaces as are recommended in NIST guide. The `\num{}` command provided by the “siunitx” package would be of help for us to follow this rule. It would also help us overcome different conventions. We can select a specific digit-grouping style as a default in preamble, or specify an option to each `\num{}` command as is shown in Table D.1.

As are evident in Table D.1, periods and commas used as other than decimal markers are confusing and should

Table D.1: Digit-Grouping Style

Style	Outputs of <code>\num{}</code>		
NIST/SI (English)	12 345	12.345	1 234 567.89
SI (French)	12 345	12,345	1 234 567,89
English	12,345	12.345	1,234,567.89
French	12 345	12,345	1 234 567,89
Other Europe	12.345	12,345	1.234.567,89

be avoided, especially in documents expecting global audiences.

By marking up constant decimal values by `\num{}` commands, the LATEX source would be exempted from any particular conventions.

Because of its open-source policy, this approach should give more “portability” to perfbook.

D.3 LATEX Conventions

Good looking LATEX documents require further considerations on proper use of font styles, line break exceptions, etc. This section summarizes guidelines specific to LATEX.

D.3.1 Monospace Font

Monospace font (or typewriter font) is heavily used in this textbook. First policy regarding monospace font in perfbook is to avoid directly using “`\texttt{}`” or “`\tt`” macro. It is highly recommended to use a macro or an environment indicating the reason why you want the font.

This section explains the use cases of such macros and environments.

D.3.1.1 Code Snippet

Because the “`verbatim`” environment is a primitive way to include listings, we have transitioned to a scheme which uses the “`fancyvrb`” package for code snippets.

The goal of the scheme is to extract LATEX sources of code snippets directly from code samples under `CodeSamples` directory. It also makes it possible to embed line labels in the code samples, which can be referenced within the LATEX sources. This reduces the burden of keeping line numbers in the text consistent with those in code snippets.

Code-snippet extraction is handled by a couple of perl scripts and recipes in `Makefile`. We use the escaping feature of the `fancyvrb` package to embed line labels as comments.

³ #6 in <https://physics.nist.gov/cuu/Units/checklist.html>

⁴ <https://physics.nist.gov/cuu/pdf/typefaces.pdf>

⁵ #16 in <https://physics.nist.gov/cuu/Units/checklist.html>.

Listing D.1: LATEX Source of Sample Code Snippet (Current)

```

1 \begin{listing}[tb]
2 \begin{fcvlabel}[ln:base1]
3 \begin{VerbatimL}[commandchars=\$\[\]]]
4 /*
5 * Sample Code Snippet
6 */
7 #include <stdio.h>
8 int main(void)
9 {
10     printf("Hello world!\n");      $lnlbl[printf]
11     return 0;                      $lnlbl[return]
12 }
13 \end{VerbatimL}
14 \end{fcvlabel}
15 \caption{Sample Code Snippet}
16 \label{lst:app:styleguide:Sample Code Snippet}
17 \end{listing}

```

Listing D.2: Sample Code Snippet

```

1 /*
2 * Sample Code Snippet
3 */
4 #include <stdio.h>
5 int main(void)
6 {
7     printf("Hello world!\n");
8     return 0;
9 }

```

We used to use the “verbbox” environment provided by the “verbatimbox” package. Appendix D.3.1.2 describes how verbbox can automatically generate line numbers, but those line numbers cannot be referenced within the LATEX sources.

Let's start by looking at how code snippets are coded in the current scheme. There are three customized environments of “Verbatim”. “VerbatimL” is for floating snippets within the “listing” environment. “VerbatimN” is for inline snippets with line count enabled. “VerbatimU” is for inline snippets without line count. They are defined in the preamble as shown below:

```

\DefineVerbatimEnvironment{VerbatimL}{Verbatim}%
  {fontsize=scriptsize,numbers=left,numberssep=5pt,%
   xleftmargin=9pt,obeytabs=true,tabsize=2}%
\AfterEndEnvironment{VerbatimL}{\vspace*{-9pt}}%
\DefineVerbatimEnvironment{VerbatimN}{Verbatim}%
  {fontsize=scriptsize,numbers=left,numberssep=3pt,%
   xleftmargin=5pt,xrightmargin=5pt,obeytabs=true,%
   tabsize=2,frame=single}%
\DefineVerbatimEnvironment{VerbatimU}{Verbatim}%
  {fontsize=scriptsize,numbers=none,xleftmargin=5pt,%
   xrightmargin=5pt,obeytabs=true,tabsize=2,%
   samepage=true,frame=single}

```

The LATEX source of a sample code snippet is shown in Listing D.1 and is typeset as shown in Listing D.2.

Labels to lines are specified in “\$lnlbl[]” command. The characters specified by “commandchars” option to

VerbatimL environment are used by the fancyvrb package to substitute “\lnlbl{}” for “\$lnlbl[]”. Those characters should be selected so that they don't appear elsewhere in the code snippet.

Labels “printf” and “return” in Listing D.2 can be referred to as shown below:

```

\begin{fcvref}[ln:base1]
Lines-\lnref{printf} and-\lnref{return} can be referred
to from text.
\end{fcvref}

```

Above code results in the paragraph below:

Lines 7 and 8 can be referred to from text.

Macros “\lnlbl{}” and “\lnref{}” are defined in the preamble as follows:

```

\newcommand{\lnlblbase}{}%
\newcommand{\lnlbl}[1]{%
  \phantomsection\label{\lnlblbase:#1}%
\newcommand{\lnrefbase}{}%
\newcommand{\lnref}[1]{\ref{\lnrefbase:#1}}

```

Environments “fcvlabel” and “fcvref” are defined as shown below:

```

\newenvironment{fcvlabel}[1][]{%
  \renewcommand{\lnlblbase}{\#1}%
  \ignorespaces{\ignorespacesafterend}%
\newenvironment{fcvref}[1][]{%
  \renewcommand{\lnrefbase}{\#1}%
  \ignorespaces{\ignorespacesafterend}%

```

The main part of LATEX source shown on 라인 2-14 in Listing D.1 can be extracted from a code sample of Listing D.3 by a perl script utilities/fcvextract.pl. All the relevant rules of extraction are described as recipes in the top level Makefile and a script to generate dependencies (utilities/gen_snippet_d.pl).

As you can see, Listing D.3 has meta commands in comments of C (C++ style). Those meta commands are interpreted by utilities/fcvextract.pl, which distinguishes the type of comment style by the suffix of code sample's file name.

Meta commands which can be used in code samples are listed below:

- \begin{snippet}<options>
- \end{snippet}
- \lnlbl{<label string>}
- \fcvexclude
- \fcvblank

Listing D.3: Source of Code Sample with “snippet” Meta Command

```

1 //\begin{snippet}[labelbase=ln:base1,keepcomment=yes,commandchars=\$\[\]]
2 /*
3  * Sample Code Snippet
4 */
5 #include <stdio.h>
6 int main(void)
7 {
8     printf("Hello world!\n");           //\lnlbl{printf}
9     return 0;                          //\lnlbl{return}
10 }
11 //\end{snippet}

```

“<options>” to the `\begin{snippet}` meta command is a comma-separated list of options shown below:

- `labelbase=<label base string>`
- `keepcomment=yes`
- `gobbleblank=yes`
- `commandchars=\$[\]\{\}`

The “`labelbase`” option is mandatory and the string given to it will be passed to the `\begin{fcvlabel}[<label base string>]` command as shown on line 2 of Listing D.1. The “`keepcomment=yes`” option tells `fcvextract.pl` to keep comment blocks. Otherwise, comment blocks in C source code will be omitted. The “`gobbleblank=yes`” option will remove empty or blank lines in the resulting snippet. The “`commandchars`” option is given to the `VerbatimL` environment as is. At the moment, it is also mandatory and must come at the end of options listed above. Other types of options, if any, are also passed to the `VerbatimL` environment.

The “`\lnlbl`” commands are converted along the way to reflect the escape-character choice.⁶ Source lines with “`\fcvexclude`” are removed. “`\fcvblank`” can be used to keep blank lines when the “`gobbleblank=yes`” option is specified.

There can be multiple pairs of `\begin{snippet}` and `\end{snippet}` as long as they have unique “`labelbase`” strings.

Our naming scheme of “`labelbase`” for unique name space is as follows:

In:<Chapter/Subdirectory>:<File Name>:<Function Name>

Litmus tests, which are handled by “`herdtools7`” commands such as “`litmus7`” and “`herd7`”, were problematic in this scheme. Those commands have particular rules of

where comments can be placed and restriction on permitted characters in comments. They also forbid a couple of tokens to appear in comments. (Tokens in comments might sound strange, but they do have such restriction.)

For example, the first token in a litmus test must be one of “C”, “PPC”, “X86”, “LISA”, etc., which indicates the flavor of the test. This means no comment is allowed at the beginning of a litmus test.

Similarly, several tokens such as “`exists`”, “`filter`”, and “`locations`” indicate the end of litmus test’s body. Once one of them appears in a litmus test, comments should be of OCaml style (“`(* ... *)`”). Those tokens keep the same meaning even when they appear in comments!

The pair of characters “`{`” and “`}`” also have special meaning in the C flavour tests. They are used to separate portions in a litmus test.

First pair of “`{`” and “`}`” encloses initialization part. Comments in this part should also be in the ocaml form.

You can’t use “`{`” and “`}`” in comments in litmus tests, either.

Examples of disallowed comments in a litmus test are shown below:

```

1 // Comment at first
2 C C-sample
3 // Comment with { and } characters
4 {
5 x=2; // C style comment in initialization
6 }
7
8 P0(int **x)
9 {
10     int r1;
11
12     r1 = READ_ONCE(*x); // Comment with "exists"
13 }
14
15 [...]
16
17 exists (0:r1=0) // C++ style comment after test body

```

To avoid parse errors, meta commands in litmus tests (C flavor) are embedded in the following way.

⁶ Characters forming comments around the “`\lnlbl`” commands are also gobbled up regardless of the “`keepcomment`” setting.

```

1 C C-SB+o-o+o-o
2 //\begin{snippet}[labelbase=ln:base,commandchars=\%\@\$]
3
4 {
5 1:r2=0          (*\lnlbl{initr2}*)
6 }
7
8 P0(int *x0, int *x1)      //\lnlbl{P0:b}
9 {
10    int r2;
11
12    WRITE_ONCE(*x0, 2);
13    r2 = READ_ONCE(*x1);
14 }                      //\lnlbl{P0:e}
15
16 P1(int *x0, int *x1)
17 {
18    int r2;
19
20    WRITE_ONCE(*x1, 2);
21    r2 = READ_ONCE(*x0);
22 }
23
24 //\end{snippet}
25 exists (1:r2=0 /\ 0:r2=0)  (* \lnlbl{exists_*} *)

```

Example above is converted to the following intermediate code by a script `utilities/reorder_ltms.pl`.⁷ The intermediate code can be handled by the common script `utilities/fcvextract.pl`.

```

1 // Do not edit!
2 // Generated by utilities/reorder_ltms.pl
3 //\begin{snippet}[labelbase=ln:base,commandchars=\%\@\$]
4 C C-SB+o-o+o-o
5
6 {
7 1:r2=0          //\lnlbl{initr2}
8 }
9
10 P0(int *x0, int *x1)      //\lnlbl{P0:b}
11 {
12    int r2;
13
14    WRITE_ONCE(*x0, 2);
15    r2 = READ_ONCE(*x1);
16 }                      //\lnlbl{P0:e}
17
18 P1(int *x0, int *x1)
19 {
20    int r2;
21
22    WRITE_ONCE(*x1, 2);
23    r2 = READ_ONCE(*x0);
24 }
25
26 exists (1:r2=0 /\ 0:r2=0)  \lnlbl{exists_*}
//\end{snippet}

```

Note that each litmus test's source file can contain at most one pair of `\begin{snippet}` and `\end{snippet}` because of the restriction of comments.

⁷ Currently, only C flavor litmus tests are supported.

Listing D.4: L^AT_EX Source of Sample Code Snippet (Obsolete)

```

1 \begin{listing}[tb]
2 { \scriptsize
3 \begin{verbbox}[\LstLineNo]
4 /*
5  * Sample Code Snippet
6 */
7 #include <stdio.h>
8 int main(void)
9 {
10   printf("Hello world!\n");
11   return 0;
12 }
13 \end{verbbox}
14 }
15 \centering
16 \theverbbox
17 \caption{Sample Code Snippet (Obsolete)}
18 \label{lst:app:styleguide:Sample Code Snippet (Obsolete)}
19 \end{listing}

```

Listing D.5: Sample Code Snippet (Obsolete)

```

1 /*
2  * Sample Code Snippet
3 */
4 #include <stdio.h>
5 int main(void)
6 {
7   printf("Hello world!\n");
8   return 0;
9 }

```

D.3.1.2 Code Snippet (Obsolete)

Sample L^AT_EX source of a code snippet coded using the “`verbatimbox`” package is shown in Listing D.4 and is typeset as shown in Listing D.5.

The auto-numbering feature of `verbbox` is enabled by the “`\LstLineNo`” macro specified in the option to `verbbox` (line 3 in Listing D.4). The macro is defined in the preamble of `perfbook.tex` as follows:

```
\newcommand{\LstLineNo}{\makebox[5ex][r]{\arabic{VerbboxLineNo}\hspace{2ex}}}
```

The “`verbatim`” environment is used for listings with too many lines to fit in a column. It is also used to avoid overwhelming L^AT_EX with a lot of floating objects. They are being converted to the scheme using the `VerbatimN` environment.

D.3.1.3 Identifier

We use “`\co{}`” macro for inline identifiers. (“`co`” stands for “code”.)

By putting them into `\co{}`, underscore characters in their names are free of escaping in L^AT_EX source. It is convenient to search them in source files. Also, `\co{}` macro has a capability to permit line breaks at particular

sequences of letters. Current definition permits a line break at an underscore (`_`), two consecutive underscores (`__`), a white space, or an operator `->`.

D.3.1.4 Identifier inside Table and Heading

Although `\co{}` command is convenient for inlining within text, it is fragile because of its capability of line break. When it is used inside a “`tabular`” environment or its derivative such as “`tabularx`”, it confuses column width estimation of those environments. Furthermore, `\co{}` can not be safely used in section headings nor description headings.

As a workaround, we use “`\tco{}`” command inside tables and headings. It has no capability of line break at particular sequences, but still frees us from escaping underscores.

When used in text, `\tco{}` permits line breaks at white spaces.

D.3.1.5 Other Use Cases of Monospace Font

For URLs, we use “`\url{}`” command provided by the “`hyperref`” package. It will generate hyper references to the URLs.

For path names, we use “`\path{}`” command. It won’t generate hyper references.

Both `\url{}` and `\path{}` permit line breaks at “`/`”, “`-`”, and “`.`”.⁸

For short monospace statements not to be line broken, we use the “`\nbco{}`” (non-breakable co) macro.

D.3.1.6 Limitations

There are a few cases where macros introduced in this section do not work as expected. Table D.2 lists such limitations.

Table D.2: Limitation of Monospace Macro

Macro	Need Escape	Should Avoid
<code>\co, \nbco</code>	<code>\, %, {, }</code>	
<code>\tco</code>	<code>#</code>	<code>%, {, }, \</code>

While `\co{}` requires some characters to be escaped, it can contain any character.

On the other hand, `\tco{}` can not handle “`%`”, “`{`”, “`}`”, nor “`\`” properly. If they are escaped by a “`\`”, they

⁸ Overfill can be a problem if the URL or the path name contains long runs of unbreakable characters.

appear in the end result with the escape character. The “`\verb`” command can be used in running text if you need to use monospace font for a string which contains many characters to escape.⁹

D.3.2 Cross-reference

Cross-references to Chapters, Sections, Listings, etc. have been expressed by combinations of names and bare `\ref{}` commands in the following way:

```

1 Chapter~\ref{chp:Introduction},
2 Table~\ref{tab:app:styleguide:Digit-Grouping Style}

```

This is a traditional way of cross-referencing. However, it is tedious and sometimes error-prone to put a name manually on every cross-reference. The `cleveref` package provides a nicer way of cross-referencing. A few examples follow:

```

1 \Cref{chp:Introduction},
2 \cref{sec:intro:Parallel Programming Goals},
3 \cref{chp:app:styleguide:Style Guide},
4 \cref{tab:app:styleguide:Digit-Grouping Style}, and
5 \cref{lst:app:styleguide:Source of Code Sample} are
6 examples of cross-\-references.

```

Above code is typeset as follows:

Chapter 2, Section 2.2, Appendix D, Table D.1, and Listing D.3 are examples of cross-references.

As you can see, naming of cross-references is automated. Current setting generates capitalized names for both of `\Cref{}` and `\cref{}`, but the former should be used at the beginning of a sentence.

We are in the middle of conversion to `cleveref`-style cross-referencing.

Cross-references to line numbers of code snippets can be done in a similar way by using `\Clnref{}` and `\clnref{}` macros, which mimic `cleveref`. The former puts “Line” as the name of the reference and the latter “line”.

Please refer to `cleveref`’s documentation for further info on its cleverness.

⁹ The `\verb` command is not almighty though. For example, you can’t use it within a footnote. If you do so, you will see a fatal L^AT_EX error. A workaround would be a macro named `\VerbatimFootnotes` provided by the `fancyvrb` package. Unfortunately, `perfbook` can’t employ it due to the interference with the `footnotebackref` package.

D.3.3 Non Breakable Spaces

In \LaTeX conventions, proper use of non-breakable white spaces is highly recommended. They can prevent widow-ing and orphaning of single digit numbers or short variable names, which would cause the text to be confusing at first glance.

The thin space mentioned earlier to be placed in front of a unit symbol is non breakable.

Other cases to use a non-breakable space (“~” in \LaTeX source, often referred to as “nbsp”) are the following (inexhaustive).

- Reference to a Chapter or a Section:

Please refer to Appendix D.2.

- Calling out CPU number or Thread name:

After they load the pointer, CPUs 1 and 2 will see the stored value.

- Short variable name:

The results will be stored in variables a and b.

D.3.4 Hyphenation and Dashes

D.3.4.1 Hyphenation in Compound Word

In plain \LaTeX , compound words such as “high-frequency” can be hyphenated only at the hyphen. This sometimes results in poor typesetting. For example:

High-frequency radio wave, high-frequency radio wave.

By using a shortcut “\-/” provided by the “extdash” package, hyphenation in elements of compound words is enabled in perfbook.¹⁰

Example with “\-/”:

High-frequency radio wave, high-frequency radio wave, high-frequency radio wave, high-frequency radio wave, high-frequency radio wave.

¹⁰ In exchange for enabling the shortcut, we can't use plain \LaTeX 's shortcut “\-” to specify hyphenation points. Use `pfhyphex.tex` to add such exceptions.

D.3.4.2 Non Breakable Hyphen

We want hyphenated compound terms such as “x-coordinate”, “y-coordinate”, etc. not to be broken at the hyphen following a single letter.

To make a hyphen unbreakable, we can use a short cut “\=/” also provided by the “extdash” package.

Example without a shortcut:

x-, y-, and z-coordinates; x-, y-, and z-coordinates; x-, y-, and z-coordinates; x-, y-, and z-coordinates; x-, y-, and z-coordinates;

Example with “\-/”:

x-, y-, and z-coordinates; x-, y-, and z-coordinates; x-, y-, and z-coordinates; x-, y-, and z-coordinates; x-, y-, and z-coordinates;

Example with “\=/”:

x-, y-, and z-coordinates; x-, y-, and z-coordinates; x-, y-, and z-coordinates; x-, y-, and z-coordinates; x-, y-, and z-coordinates;

Note that “\=/” enables hyphenation in elements of compound words as the same as “\-/” does.

D.3.4.3 Em Dash

Em dashes are used to indicate parenthetical expression. In perfbook, em dashes are placed without spaces around it. In \LaTeX source, an em dash is represented by “---”.

Example (quote from Appendix C.1):

This disparity in speed—more than two orders of magnitude—has resulted in the multi-megabyte caches found on modern CPUs.

D.3.4.4 En Dash

In \LaTeX convention, en dashes (–) are used for ranges of (mostly) numbers. Past revisions of perfbook didn't follow this rule and used plain dashes (-) for such cases.

Now that `\clnrefrange`, `\crefrange`, and their variants, which generate en dashes, are used for ranges of cross-references, the remaining couple of tens of simple

dashes of other types of ranges have been converted to en dashes for consistency.

Example with a simple dash:

Lines 4–12 in Listing D.4 are the contents of the `verbbox` environment. The box is output by the `\theverbbox` macro on 라인 16.

Example with an en dash:

Lines 4–12 in Listing D.4 are the contents of the `verbbox` environment. The box is output by the `\theverbbox` macro on 라인 16.

D.3.4.5 Numerical Minus Sign

Numerical minus signs should be coded as math mode minus signs, namely “ $-$$ ”.¹¹ For example,

–30, rather than -30.

D.3.5 Punctuation

D.3.5.1 Ellipsis

In monospace fonts, ellipses can be expressed by series of periods. For example:

Great ... So how do I fix it?

However, in proportional fonts, the series of periods is printed with tight spaces as follows:

Great ... So how do I fix it?

Standard LATEX defines the `\dots` macro for this purpose. However, it has a kludge in the evenness of spaces. The “ellipsis” package redefines the `\dots` macro to fix the issue.¹² By using `\dots`, the above example is typeset as the following:

Great ... So how do I fix it?

Note that the “`xspace`” option specified to the “ellipsis” package adjusts the spaces after ellipses depending on what follows them.

For example:

¹¹ This rule assumes that math mode uses the same upright glyph as text mode. Our default font choice meets the assumption.

¹² To be exact, it is the `\textellipsis` macro that is redefined. The behavior of `\dots` macro in math mode is not affected. The “`amsmath`” package has another definition of `\dots`. It is not used in `verbbox` at the moment.

- He said, “I ... really don’t remember ...”
- Sequence A: (one, two, three, ...)
- Sequence B: (4, 5, ..., n)

As you can see, extra space is placed before the comma. `\dots` macro can also be used in math mode:

- Sequence C: (1, 2, 3, 5, 8, ...)
- Sequence D: (10, 12, ..., 20)

The `\ldots` macro behaves the same as the `\dots` macro.

D.3.5.2 Full Stop

LATEX treats a full stop in front of a white space as an end of a sentence and puts a slightly wider skip by default (double spacing). There is an exception to this rule, i.e. where the full stop is next to a capital letter, LATEX assumes it represents an abbreviation and puts a normal skip.

To make LATEX use proper skips, one need to annotate such exceptions. For example, given the following LATEX source,

```
\begin{quote}
Lock-1 is owned by CPU-A.
Lock-2 is owned by CPU-B. (Bad.)

Lock-1 is owned by CPU-A\@.
Lock-2 is owned by CPU-B\@. (Good.)
\end{quote}
```

the output will be as the following.

Lock 1 is owned by CPU A. Lock 2 is owned by CPU B. (Bad.)

Lock 1 is owned by CPU A. Lock 2 is owned by CPU B. (Good.)

On the other hand, where a full stop is following a lower case letter, e.g. as in “Mr. Smith”, a wider skip will follow in the output unless it is properly hinted. Such hintings can be done in one of several ways.

Given the following source,

```
\begin{itemize}[nosep]
\item Mr. Smith (bad)
\item Mr.~Smith (good)
\item Mr.\ Smith (good)
\item Mr.\@ Smith (good)
\end{itemize}
```

the result will look as follows:

- Mr. Smith (bad)
- Mr. Smith (good)
- Mr. Smith (good)
- Mr. Smith (good)

D.3.6 Floating Object Format

D.3.6.1 Ruled Line in Table

They say that tables drawn by using ruled lines of plain \LaTeX look ugly.¹³ Vertical lines should be avoided and horizontal lines should be used sparingly, especially in tables of simple structure.

Table D.3 (corresponding to a table from a now-deleted section) is drawn by using the features of “booktabs” and “xcolor” packages. Note that ruled lines of booktabs can not be mixed with vertical lines in a table.¹⁴

Table D.3: Refrigeration Power Consumption

Situation	T (K)	C_p	Power per watt waste heat (W)
Dry Ice	195	1.990	0.5
Liquid N ₂	77	0.356	2.8
Liquid H ₂	20	0.073	13.7
Liquid He	4	0.0138	72.3
IBM Q	0.015	0.000051	19,500.0

D.3.6.2 Position of Caption

In \LaTeX conventions, captions of tables are usually placed above them. The reason is the flow of your eye movement when you look at them. Most tables have a row of heading at the top. You naturally look at the top of a table at first. Captions at the bottom of tables disturb this flow. The same can be said of code snippets, which are read from top to bottom.

For code snippets, the “ruled” style chosen for listing environment places the caption at the top. See Listing D.2 for an example.

As for tables, the position of caption is tweaked by `\floatstyle{}` and `\restylefloat{}` macros in preamble.

¹³ <https://www.inf.ethz.ch/personal/markuspt/teaching/guides/guide-tables.pdf>

¹⁴ There is another package named “arydshln” which provides dashed lines to be used in tables. A couple of experimental examples are presented in Appendix D.3.7.2.

Vertical skips below captions are reduced by setting a smaller value to the `\abovecaptionskip` variable, which would also affect captions to figures.

In the tables which use horizontal rules of “booktabs” package, the vertical skips between captions and tables are further reduced by setting a negative value to the `\abovetopsep` variable, which controls the behavior of `\toprule`.

D.3.7 Improvement Candidates

There are a few areas yet to be attempted in `perfbook` which would further improve its appearance. This section lists such candidates.

D.3.7.1 Grouping Related Figures/Listings

To prevent a pair of closely related figures or listings from being placed in different pages, it is desirable to group them into a single floating object. The “subfig” package provides the features to do so.¹⁵

Two floating objects can be placed side by side by using `\parbox` or `minipage`. For example, Figures 14.10 and 14.11 can be grouped together by using a pair of `minipages` as shown in Figures D.1 and D.2.

By using `subfig` package, Listings 15.4 and 15.5 can be grouped together as shown in Listing D.6 with sub-captions (with a minor change of blank line).

Note that they can not be grouped in the same way as Figures D.1 and D.2 because the “ruled” style prevents their captions from being properly typeset.

The sub-caption can be cited by combining a `\ref{}` macro and a `\subref{}` macro, for example, “Listing D.6(a)”.

It can also be cited by a `\ref{}` macro, for example, “Listing D.6b”. Note the difference in the resulting format. For the citing by a `\ref{}` to work, you need to place the `\label{}` macro of the combined floating object ahead of the definition of subfloats. Otherwise, the resulting caption number would be off by one from the actual number.

D.3.7.2 Table Layout Experiment

This section presents some experimental tables using booktabs, xcolors, and arydshln packages. The corresponding tables in the text have been converted using one of the format shown here. The source of this section can be

¹⁵ One problem of grouping figures might be the complexity in \LaTeX source.



Figure D.1: Timer Wheel at 1 kHz



Figure D.2: Timer Wheel at 100 kHz

Listing D.6: Message-Passing Litmus Test (by subfig)

(a) Not Enforcing Order

```

1 C C-MP+o-wmb-o+o-o.litmus
2
3 {
4 }
5
6 P0(int* x0, int* x1) {
7
8   WRITE_ONCE(*x0, 2);
9   smp_wmb();
10  WRITE_ONCE(*x1, 2);
11
12 }
13
14 P1(int* x0, int* x1) {
15
16   int r2;
17   int r3;
18
19   r2 = READ_ONCE(*x1);
20   r3 = READ_ONCE(*x0);
21
22 }
23
24
25 exists (1:r2=2 /\ 1:r3=0)

```

(b) Enforcing Order

```

1 C C-MP+o-wmb-o+o-rmb-o.litmus
2
3 {
4 }
5
6 P0(int* x0, int* x1) {
7
8   WRITE_ONCE(*x0, 2);
9   smp_wmb();
10  WRITE_ONCE(*x1, 2);
11
12 }
13
14 P1(int* x0, int* x1) {
15
16   int r2;
17   int r3;
18
19   r2 = READ_ONCE(*x1);
20   smp_rmb();
21   r3 = READ_ONCE(*x0);
22
23 }
24
25 exists (1:r2=2 /\ 1:r3=0)

```

regarded as a reference to be consulted when new tables are added in the text.

In Table D.4 (corresponding to Table 3.1), the “S” column specifiers provided by the “siunitx” package are used to align numbers.

Table D.5 (corresponding to Table 13.1) is an example of table with a complex header. In Table D.5, the gap in the mid-rule corresponds to the distinction which had been represented by double vertical rules before the conversion. The legends in the frame box appended here explain the abbreviations used in the matrix. Two types of memory barrier are denoted by subscripts here. The legends and subscripts are not present in Table 13.1 since they are redundant there.

Table D.6 (corresponding to Table C.1) is a sequence diagram drawn as a table.

Table D.7 is a tweaked version of Table 9.2. Here, the “Category” column in the original is removed and the categories are indicated in rows of bold-face font just below the mid-rules. This change makes it easier for \rowcolors{} command of “xcolor” package to work properly.

Table D.8 is another version which keeps original columns and colors rows only where a category has multiple rows. This is done by combining \rowcolors{} of “xcolor” and \cellcolor{} commands of the “colortbl” package (\cellcolor{} overrides \rowcolors{}).

Table D.4: CPU 0 View of Synchronization Mechanisms on 8-Socket System With Intel Xeon Platinum 8176 CPUs @ 2.10GHz

Operation	Cost (ns)	Ratio (cost/clock)
Clock period	0.5	1.0
Best-case CAS	7.0	14.6
Best-case lock	15.4	32.3
Blind CAS	7.2	15.2
CAS	18.0	37.7
Blind CAS (off-core)	47.5	99.8
CAS (off-core)	101.9	214.0
Blind CAS (off-socket)	148.8	312.5
CAS (off-socket)	442.9	930.1
Comms Fabric	5,000	10,500
Global Comms	195,000,000	409,500,000

In Table 9.2, the latter layout without partial row coloring has been chosen for simplicity.

Table D.9 (corresponding to Table 15.1) is also a sequence diagram drawn as a tabular object.

Table D.10 shows another version of Table D.3 with dashed horizontal and vertical rules of the `arydshln` package.

Table D.10: Refrigeration Power Consumption

Situation	T (K)	C_P	Power per watt waste heat (W)
Dry Ice	195	1.990	0.5
Liquid N ₂	77	0.356	2.8
Liquid H ₂	20	0.073	13.7
Liquid He	4	0.0138	72.3
IBM Q	0.015	0.000051	19,500.0

In this case, the vertical dashed rules seems unnecessary. The one without the vertical rules is shown in Table D.11.

Table D.5: Synchronization and Reference Counting

Acquisition	Release			
	Locks	Reference Counts	Hazard Pointers	RCU
Locks	—	CAM _R	M	CA
Reference Counts	A	AM _R	M	A
Hazard Pointers	M	M	M	M
RCU	CA	M _A CA	M	CA

Key:

- A: Atomic counting
- C: Check combined with the atomic acquisition operation
- M: Full memory barriers required
- M_R: Memory barriers required only on release
- M_A: Memory barriers required on acquire

Table D.11: Refrigeration Power Consumption

Situation	T (K)	C_P	Power per watt waste heat (W)
Dry Ice	195	1.990	0.5
Liquid N ₂	77	0.356	2.8
Liquid H ₂	20	0.073	13.7
Liquid He	4	0.0138	72.3
IBM Q	0.015	0.000051	19,500.0

D.3.7.3 Miscellaneous Candidates

Other improvement candidates are listed in the source of this section as comments.

Table D.6: Cache Coherence Example

Sequence #	CPU #	Operation	CPU Cache				Memory	
			0	1	2	3	0	8
0		Initial State	-/I	-/I	-/I	-/I	V	V
1	0	Load	0/S	-/I	-/I	-/I	V	V
2	3	Load	0/S	-/I	-/I	0/S	V	V
3	0	Invalidation	8/S	-/I	-/I	0/S	V	V
4	2	RMW	8/S	-/I	0/E	-/I	V	V
5	2	Store	8/S	-/I	0/M	-/I	I	V
6	1	Atomic Inc	8/S	0/M	-/I	-/I	I	V
7	1	Writeback	8/S	8/S	-/I	-/I	V	V

Table D.7: RCU Publish-Subscribe and Version Maintenance APIs

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>	2.5.44	Memory barrier
<code>list_add_tail_rcu()</code>	2.5.44	Memory barrier
<code>list_del_rcu()</code>	2.5.44	Simple instructions
<code>list_replace_rcu()</code>	2.6.9	Memory barrier
<code>list_splice_init_rcu()</code>	2.6.21	Grace-period latency
Hlist traversal		
<code>hlist_for_each_entry_rcu()</code>	2.6.8	Simple instructions (memory barrier on Alpha)
Hlist update		
<code>hlist_add_after_rcu()</code>	2.6.14	Memory barrier
<code>hlist_add_before_rcu()</code>	2.6.14	Memory barrier
<code>hlist_add_head_rcu()</code>	2.5.64	Memory barrier
<code>hlist_del_rcu()</code>	2.5.64	Simple instructions
<code>hlist_replace_rcu()</code>	2.6.15	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 D.8: RCU Publish-Subscribe and Version Maintenance APIs

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>	2.5.44	Memory barrier
	<code>list_add_tail_rcu()</code>	2.5.44	Memory barrier
	<code>list_del_rcu()</code>	2.5.44	Simple instructions
	<code>list_replace_rcu()</code>	2.6.9	Memory barrier
	<code>list_splice_init_rcu()</code>	2.6.21	Grace-period latency
Hlist traversal	<code>hlist_for_each_entry_rcu()</code>	2.6.8	Simple instructions (memory barrier on Alpha)
Hlist update	<code>hlist_add_after_rcu()</code>	2.6.14	Memory barrier
	<code>hlist_add_before_rcu()</code>	2.6.14	Memory barrier
	<code>hlist_add_head_rcu()</code>	2.5.64	Memory barrier
	<code>hlist_del_rcu()</code>	2.5.64	Simple instructions
	<code>hlist_replace_rcu()</code>	2.6.15	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 D.9: Memory Misordering: Store-Buffering Sequence of Events

CPU 0			CPU 1		
Instruction	Store Buffer	Cache	Instruction	Store Buffer	Cache
1 (Initial state)		$x1==0$	(Initial state)		$x0==0$
2 $x0 = 2;$	$x0==2$	$x1==0$	$x1 = 2;$	$x1==2$	$x0==0$
3 $r2 = x1; (0)$	$x0==2$	$x1==0$	$r2 = x0; (0)$	$x1==2$	$x0==0$
4 (Read-invalidate)	$x0==2$	$x0==0$	(Read-invalidate)	$x1==2$	$x1==0$
5 (Finish store)		$x0==2$	(Finish store)		$x1==2$

The Answer to the Ultimate Question of Life, The Universe, and Everything.

“The Hitchhikers Guide to the Galaxy”,
Douglas Adams

Appendix E

Answers to Quick Quizzes

E.1 How To Use This Book

Quick Quiz 1.1:

Quick Quiz 의 답은 어디 있나요?



Answer:

Appendix E 의 page 431 에 있습니다.

이봐요, 쉽게 설명드린 것 같군요!



Quick Quiz 1.2:

일부 Quick Quiz 의 질문은 저자의 시선보다는 독자의 시선에서 쓰여진 것 같습니다. 의도된 바인가요?



Answer:

그렇습니다! 많은 것들이 Paul E. McKenney 가 이 주제를 다루는 수업을 듣는 무직의 학생이었다면 물어봤음직한 질문들입니다. Paul 은 이것들을 교수들이 아니라 병렬 하드웨어와 소프트웨어에서 배웠음을 말해줄 가치가 있습니다. Paul 의 경험에 따르면, 교수들은 병렬 시스템보다는 구두의 질문에 대한 답을 줄 가능성 이 무척 큽니다, 최근의 음성 어시스턴트들의 발달에도 불구하고 말이죠. 물론, 교수들과 병렬 시스템들 중 어느 쪽이 이런 종류의 질문에 더 유용한 답을 주는가는 긴 논쟁을 할 수도 있겠지만, 지금으로썬 교수들과 병렬 시스템들 모두 세대에 따라 그 답변의 유용도가 상당히 다르다는 정도로 동의하고 넘어갑시다.

다른 퀴즈들은 컨퍼런스에서의 발표와 이 책의 것들을 다루는 수업에서 받은 실제 질문들과 상당히 비슷합니다. 그 외의 일부는 저자의 시점에서 쓰였습니다.



Quick Quiz 1.3:

○] Quick quiz 는 제게 맞지 않는 것 같아요. 어떡하죠?



Answer:

여기 가능한 몇가지 전략이 있습니다:

1. 그냥 Quick quiz 를 무시하고 이 책의 나머지를 읽으세요. Quick quiz 의 일부 흥미로운 것들을 놓칠 수 있겠지만, 이 책의 나머지 부분에도 좋은 것들이 많습니다. 이는 여러분의 주요 목적이 이것들에 대한 일반적 이해를 얻고자 함이거나 이 책에서 특정 문제에 대한 해결책을 찾기 위함이라면 이게 상당히 합리적인 접근법입니다.
2. 스스로의 답을 내기 위해 많은 시간을 들이지 말고 곧바로 답을 찾아보세요. 이 접근법은 해당 Quick quiz 의 답이 여러분이 풀고자 하는 특정 문제의 답을 위한 열쇠를 가지고 있다면 합리적입니다. 이 접근법은 또한 여러분이 이 주제에 대한 더 깊은 이해를 원하지만 백지 상태부터 병렬 해결책을 만들어내라고 요구받지 않을 것이라 예상된다면 합리적입니다.
3. Quick quiz 가 방해되지만 무시할 수는 없다면, 이 책의 git 저장소로부터 L^AT_EX 소스를 복사할 수 있습니다. 그 후 `make nq` 커맨드를 실행할 수 있는데, 이는 `perfbook-nq.pdf` 를 만들 겁니다. 이 PDF 는 Quick quiz 가 있어야 할 자리에 거슬리지 않는 상자에 담긴 태그들을 포함하고 있고, 각 챕터의 Quick quiz 와 그 답들을 해당 챕터의 끝에 일반적 교과서 스타일로 담고 있을 겁니다.
4. Quick Quizz 를 좋아하도록 (아니면 적어도 다룰 수 있도록) 노력하세요. 경험상 글을 읽는 와중에 주기적으로 퀴즈를 푸는건 이해의 깊이를 상당히 개선시킵니다.

Quick quiz 는 그 답으로 하이퍼링크 되어 있고 그 반대 역시 마찬가지임을 알아 두세요. 답으로 이동하려면 “Quick Quiz” 부분이나 작은 검정 사각형을 클릭하세요. 답에서 해당 퀴즈의 시작으로 돌아가려면 답의 시작 부분이나 작은 검정 사각형을 클릭하세요. 또는, 연관된 퀴즈의 끝으로 돌아가려면 답 끝의 작은 하얀색 사각형을 클릭하세요.

□

E.2 Introduction

Quick Quiz 2.1:

이봐요!!! 병렬 프로그래밍은 수십년동안 엄청나게 어렵다고 알려져왔어요. 당신은 그게 그렇게 어렵지 않고 하는 것 같군요. 뭘가 게임이라도 하는 건가요?

■

Answer:

병렬 프로그래밍이 정말 엄청나게 어렵다고 생각한다면, “왜 병렬 프로그래밍은 어려운가?”라는 질문에 대한 답을 가져야 합니다. 어떤 사람은 데드락, 레이스 컨디션, 테스트 커버리지 등의 많은 이유를 이야기 할 수 있겠지만 진짜 답은 그게 정말로 그렇게 어렵지는 않다 입니다. 무엇보다, 병렬 프로그래밍이 정말 그렇게 소름끼치도록 어렵다면, Apache, MySQL, 그리고 리눅스 커널을 포함하는 많은 수의 오픈소스 프로젝트들은 그걸 해냈겠습니까?

더 나은 질문은 아마도 이걸겁니다: “왜 병렬 프로그래밍은 그렇게 어려울거라 여겨지는가?” 그 답을 알기 위해, 1991년도로 돌아가 봅시다. Paul McKenney는 Sequent 의 벤치마킹 센터로 여섯개의 dual-80486 Sequent Symmetry CPU 보드를 들고 주차장을 건너 걸어가고 있었는데, 그는 순간 자신이 그가 최근 구입한 집보다 몇배가 되는 가격의 물건을 들고 있음을 깨달았습니다.¹ 이런 병렬 시스템의 높은 가격은 병렬 프로그래밍이 \$100,000—1991년 US 달러—까지 달하는 기계를 제조하거나 구매할 수 있는 고용주를 위해 일하는, 권한을 받은 일부에게로 제한되었음을 의미합니다.

대조적으로, 2020년, Paul은 자신이 이 글을 six-core x86 랩톱에서 타이핑하고 있음을 압니다. 그 dual-80486 CPU 보드와 달리, 이 랩톱은 또한 64 GB 메인 메모리, 1 TB SSD, 디스플레이, 이더넷, USB 포트, 무선인터넷, 블루투스를 포함하고 있습니다. 그리고 이 랩톱은 그 dual-80486 CPU 보드들 중 하나보다도 열배 넘게 싹니다, 물가 상승을 계산하지 않아도 말이죠.

¹ 그래요, 이 순간적 깨달음은 정말 그를 훨씬 조심스럽게 견제 만들었습니다. 왜 묻는 거죠?

병렬 시스템은 정말 도착했습니다. 그것들은 더이상 선택받은 일부에게만 허용된 도메인이 아니라 거의 모두에게 사용가능한 무언가입니다.

이전의 병렬 하드웨어의 제한된 접근성이 병렬 프로그래밍은 그렇게 어렵다고 여겨지게 한 진짜 이유입니다. 어쨌건, 접근할 수가 없다면 가장 간단한 기계조차도 프로그래밍 하는 법을 배우기가 무척 어렵습니다. 흔치 않고 비싼 병렬 기계의 시대는 거의 과거가 되고 있으므로, 병렬 프로그래밍이 미치도록 어렵다고 여겨지던 시대는 끝나가고 있습니다.²

□

Quick Quiz 2.2:

병렬 프로그래밍이 순차적 프로그래밍만큼 쉬워지는게 가능은 할까요?

■

Answer:

그건 프로그래밍 환경에 달려 있습니다. SQL [Int92] 은 과소평가된 성공사례로, 병렬성을 전혀 모르는 프로그래머가 거대한 병렬 시스템을 생산적으로 바쁘게 만들 수 있게 합니다. 병렬 컴퓨터가 저렵해지고 더 접근 가능하게 될수록 이런 주제가 다양해질 거라 예상해 볼 수 있습니다. 예를 들어, 과학 기술 컴퓨팅 쪽에서의 가능한 대적자는 일반적 행렬 계산을 자동으로 병렬화 시키는 MATPAB*P 가 될겁니다.

마지막으로, 리눅스와 유닉스 시스템에서는 다음 셸 커맨드를 생각해 봅시다:

```
get_input | grep "interesting" | sort
```

이 셸 파이프라인은 get_input, grep, 그리고 sort 를 병렬로 수행합니다. 봐요, 그렇게 어렵지 않았어요, 이제 어때요?

짧게 요약해서, 병렬 프로그래밍은 순차적 프로그래밍만큼 간단합니다—최소한 병렬성을 사용자로부터 감추는 환경에서라면요!

□

Quick Quiz 2.3:

오, 정말로요??? 정확성, 유지가능성, 견고성, 등등은 어찌고요?

■

Answer:

² 병렬 프로그래밍은 어떤 면에서는 순차적 프로그래밍보다 어려운데, 예를 들어 병렬 검증은 더 어렵습니다. 하지만 더이상 미치도록 어렵진 않습니다.

이것들은 중요한 목표입니다만, 병렬 프로그램에 그 렇듯 순차적 프로그램에도 중요합니다. 따라서 중요하긴 하지만 병렬 프로그래밍에 국한된 리스트에 들어갈 수는 없습니다.



Quick Quiz 2.4:

그리고 정확성, 유지가능성, 견고성이 그 리스트에 들어 가지 못한다면, 생산성과 범용성은 왜 들어가는거죠?



Answer:

병렬 프로그래밍이 순차적 프로그래밍에 비해 훨씬 어렵다고 여겨진다는 점을 고려하면, 생산성 역시 그러하며 따라서 무시되지 말아야 합니다. 더 나아가서, SQL과 같은 높은 생산성의 병렬 프로그래밍 환경은 특수 목적을 처리하며, 따라서 범용성 역시 리스트에 추가되어야 합니다.



Quick Quiz 2.5:

병렬 프로그램은 순차적 프로그램보다 정확성을 입증하기가 훨씬 어렵다는 점을 고려하면, 정확성도 정말 이 리스트에 들어가야 하지 않습니까?



Answer:

엔지니어링 관점에서 보면, 정식으로든 비정식으로든 정확성을 검증하는데 드는 어려움은 생산성이라는 주요 목표에 영향 끼치는 만큼만 중요합니다. 따라서, 정확성 검증이 중요한 경우라면, “생산성” 항목에 포함되어 있습니다.



Quick Quiz 2.6:

그냥 재미를 목표로 하는건 어때요?



Answer:

재미를 주는 것도 물론 중요합니다만, 여러분이 취미가가 아니라면, 주요 목표가 되진 못할 겁니다. 한편, 여러분이 취미가가 맞다면, 날뛰어 보십시오!



Quick Quiz 2.7:

병렬 프로그래밍이 성능 외의 것을 위한 경우는 없나요?



Answer:

해결해야 하는 문제가 본질적으로 병렬적인 경우가 있는데, 예를 들어 Monte Carlo 방법론과 일부 수학 연산들입니다. 하지만, 이 경우들에도 병렬성을 관리하기 위한 추가 작업이 일부 있을 겁니다.

병렬성은 또한 때로 안전성을 위해 사용됩니다. 한 가지만 예를 들어보자면, triple-modulo redundancy는 세개의 시스템을 병렬로 돌리고 그 결과를 투표해서 정합니다. 극단적 경우에는, 이 세개의 시스템이 다른 알고리즘과 기술을 가지고 독립적으로 구현될 겁니다.



Quick Quiz 2.8:

프로그램을 비효율적인 스크립트 언어에서 C나 C++로 재작성하는 건 어떨까요?



Answer:

그런 재작성을 위한 개발자들, 예산, 그리고 시간이 충분하다면, 그리고 그게 단일 CPU에서도 필요한 수준의 성능을 얻을 수 있다면, 합리적 접근법이 될 수 있습니다.



Quick Quiz 2.9:

왜 이건 기술적이지 않은 문제들에 대해 떠드는 거죠??? 단순히 아무 기술적이지 않은 문제가 아니라, 생산성이 라구요? 누가 이걸 신경씁니까?



Answer:

여러분이 순수한 취미가라면, 아마 신경쓸 필요 없을 겁니다. 하지만 순수한 취미가라 할지라도 얼마나 빨리 얼마나 많이 일을 해낼 수 있는가에 종종 신경씁니다. 어쨌건, 가장 대중적인 취미가용 도구는 그 일을 하는데 최적인 것들이 경우가 많고, “최적”의 정의의 중요한 부분은 생산성과 연관되어 있습니다. 그리고 만약 누군가가 여러분이 병렬 코드를 작성하는데 대해 돈을 지불하고 있다면, 그들은 여러분의 생산성에 무척 신경쓰고 있을 가능성이 높습니다. 그리고 여러분에게 돈을 지불하는 사람이 뭔가에 신경쓰고 있다면, 여러분도 그것에 최소 어느 정도는 신경을 쓰는게 현명할 겁니다!

그 외에도, 여러분이 정말로 생산성을 신경쓰지 않는다면, 컴퓨터를 사용하지 않고 수작업으로 직접 일을 하시겠죠!



Quick Quiz 2.10:

병렬 시스템이 그렇게 싸졌는데, 그걸 프로그램하라고 누가 돈을 줄까요?



Answer:

이 질문에 여러 답이 가능합니다:

1. 병렬 기계의 클러스터가 있다면, 이 클러스터의 전체 가격은 상당한 수준의 개발 노력을 정당화 할 수 있을텐데, 개발 비용이 이 많은 수의 기계들로 나뉘어질 수 있기 때문입니다.
2. 수천만명의 사용자들에 의해 돌아가는 대중적인 소프트웨어는 상당한 개발 노력을 쉽게 정당화 할 수 있는데, 이 개발 비용은 수천만명의 사용자로 나뉘어질 수 있기 때문입니다. 이는 커널과 시스템 라이브러리 같은 것들을 포함함을 알아두십시오.
3. 저렴한 병렬 머신이 중요한 장비의 한 부품의 운영을 조절한다면, 이 장비의 한 부품의 가격은 상당한 개발 노력을 쉽게 정당화 시킬 수도 있습니다.
4. 저렴한 병렬 기계의 소프트웨어가 극단적으로 중요한 결과 (예: 전력 절약) 를 만들어 낸다면, 이 중요한 결과가 역시 상당한 개발 비용을 정당화 시킬 수도 있습니다.
5. 안전성이 중요한 시스템은 목숨을 보호하므로, 분명히 매우 큰 개발 노력을 정당화 할 수 있습니다.
6. 취미가들과 연구자들은 대신 지식, 경험, 재미, 그리고 영광을 추구할 겁니다.

그러니 줄어드는 하드웨어 가격이 소프트웨어를 가치없게 하지는 않으며, 오히려 소프트웨어 개발 비용을 하드웨어 가격 아래 “감출” 수 없어졌습니다, 엄청나게 커다란 하드웨어 단위가 존재하지 않는 이상 말이죠.

□

Quick Quiz 2.11:

이건 이루어질 수 없는 이상이예요! 왜 실제로 이를 수 있는 것들에 집중하지 않는 거죠?

■

Answer:

이건 대단히 이를 수 있는 것입니다. 휴대폰은 최종 사용자 측에서의 프로그래밍이나 설정을 매우 적게 하거나 아예 없이도 전화를 하고 문자 메세지를 주고받을 수 있는 컴퓨터입니다.

이는 사소한 예로 보일수도 있겠지만, 주의깊게 들여다보면 이게 간단하거나와 심오하기도 함을 알게 될겁니다. 범용성을 희생시키고자 할 때, 우린 상당한 생산성 증가도 얻을 수 있습니다. 따라서 지나친 범용성에 빠져든 사람들은 소프트웨어 스택의 꼭대기 근처에 도달하기 충분한 정도로 생산성 목표를 잡지 못할 겁니다. 이 사실은 약자도 있습니다: YAGNI, 또는 “You Ain’t Gonna Need It.”

□

Quick Quiz 2.12:

잠깐만요! 이 방법은 개발 노력을 당신으로부터 당신이 사용하는 병렬 소프트웨어를 개발한 누군가에게 떠넘길 뿐인 것 아닌가요?

■

Answer:

바로 그렇습니다! 그리고 그게 이미 존재하는 소프트웨어를 사용하는 것의 핵심입니다. 한 팀의 작업물이 다른 여러 팀에 의해 사용될 수 있고, 결국 모든 팀이 필요없이 바퀴를 재발명하는 것에 비교해 전체적 노력을 크게 줄이게 됩니다.

□

Quick Quiz 2.13:

어떤 다른 병목지점이 추가된 CPU 가 성능을 개선하는 걸 막을 수 있을까요?

■

Answer:

여러 잠재적 병목이 있을 수 있습니다:

1. 메인 메모리. 하나의 쓰레드가 모든 가용한 메모리를 소비한다면, 추가된 쓰레드는 그저 우습게도 스스로를 페이징 인/아웃 시킬 뿐일 겁니다.
2. 캐시. 하나의 쓰레드의 캐시 사용량이 모든 공유 CPU 캐시(들)을 완전히 채울 정도라면 더 많은 쓰레드를 추가하는 것은 그저 그 캐시를 쓰래싱하게 만들 텐데, 이걸 Chapter 10에서 보게 될 겁니다.
3. 메모리 대역폭. 하나의 쓰레드가 모든 가용한 메모리 대역폭을 소비한다면, 추가된 쓰레드는 그저 시스템 인터컨넥트에 일감이 더 쌓이는 결과가 될 겁니다.
4. I/O 대역폭. 하나의 쓰레드가 I/O 에 매여 있다면, 더 많은 쓰레드를 추가하는건 그 모든 쓰레드가 영향받는 I/O 자원에 줄어서 기다리는 결과를 초래할 겁니다.

특정한 하드웨어 시스템은 추가적인 병목을 얼마든 가지고 있을 수 있습니다. 중요한 사실은 여러 CPU 또는 쓰레드 사이에 공유되는 모든 자원이 잠재적 병목이라는 겁니다.

□

Quick Quiz 2.14:

CPU 캐쉬 용량 외에, 어떤 것이 동시에 수행되는 쓰레드의 수를 제한할 수 있을까요?

**Answer:**

쓰레드의 수를 제한할 수 있는 잠재적인 것들은 얼마든지 있습니다:

1. 메인 메모리. 각 쓰레드가 메모리를 일부 소비하므로 (다른게 전혀 없다면 스택을 위해서라도), 극단적으로 많은 쓰레드는 메모리를 고갈시켜서 극단적인 페이징이나 메모리 할당 실패를 초래합니다.
2. I/O 대역폭. 각 쓰레드가 특정량의 대용량 저장장치 I/O 또는 네트워킹 트래픽을 만든다면, 극단적으로 많은 수의 쓰레드는 극단적인 I/O 대기 지연을 초래해, 역시 성능을 하락시킵니다. 어떤 네트워킹 프로토콜은 쓰레드가 너무 많아 어떤 네트워킹 이벤트가 특정 시간 내에 응답을 받지 못한다면 타임아웃 또는 다른 실패를 할 수 있습니다.
3. 동기화 오버헤드. 많은 동기화 프로토콜에 있어, 극단적으로 많은 수의 쓰레드는 극단적인 스플팅, 블락킹, 롤백을 초래해 성능을 하락시킬 수 있습니다.

구체적인 어플리케이션과 플랫폼에 따라서 추가적인 제한적 요소가 얼마든지 있을 수 있습니다.

**Quick Quiz 2.15:**

“명시적 타이밍”이 정확히 뭔가요???

**Answer:**

각 쓰레드가 특정 자원 접근으로의 접근을 서로 동의된 시간 동안 허락받는 것입니다. 예를 들어, 여덟개의 쓰레드를 갖는 병렬 프로그램은 8밀리세컨드 시간 간격으로 조직되어서, 첫번째 쓰레드는 각 간격의 첫번째 밀리세컨드 동안 접근 권한을 갖고, 두번째 쓰레드는 두 번째 밀리세컨드 동안, 그런 식입니다. 이 방법은 주의 깊게 동기화된 시계와 수행 시간에 대한 주의를 분명히 필요로 하며, 따라서 상당히 조심해야 합니다.

사실, 하드 리얼타임 환경 외에서는 여러분은 이것 대신 다른 걸 사용하고자 할 겁니다. 명시적 타이밍은 그래도 언급할 가치가 있는데, 여러분이 필요로 할 때엔 항상 거기 있을 것이기 때문입니다.

**Quick Quiz 2.16:**

병렬 프로그래밍에 어떤 다른 장애물들도 있을까요?

**Answer:**

병렬 프로그래밍의 다른 잠재적 장애물이 굉장히 많습니다. 여기 그 중 몇개가 있습니다:

1. 주어진 프로젝트를 위해 알려진 유일한 알고리즘 이 근본적으로 순차적일 수 있습니다. 이 경우, 병렬 프로그래밍을 하지 않거나 (여러분의 프로젝트가 병렬로 수행되어야만 한다는 법이 없습니다) 새로운 병렬 알고리즘을 발명해야 합니다.
2. 같은 주소공간을 공유하는 바이너리 플러그인만을 지원해서, 어떤 개발자도 이 프로젝트의 모든 소스 코드로의 접근 권한이 없는 경우. 데드락을 포함한 많은 병렬 프로그램의 버그들이 흔하므로, 그런 바이너리로만 되어 있는 플러그인들은 현재 소프트웨어 개발 방법론에 상당한 어려움을 끼칩니다. 이 또한 바뀔 수도 있겠지만, 현재로써는, 주소공간을 공유하는 병렬 코드의 모든 개발자가 해당 주소공간에서 수행되는 모든 코드를 볼 수 있어야 합니다.
3. 프로젝트가 병렬성을 고려하지 않고 설계된 API 를 무척 많이 사용하는 경우 [AGH⁺11a, CKZ⁺13]. System V 메세지 큐 API의 일부 화려한 기능이 이 경우에 속합니다. 물론, 여러분의 프로젝트가 수십년 된 것이라면, 그리고 그 개발자가 병렬 하드웨어로의 접근권한을 갖지 못했다면, 의심의 여지 없이 그런 API가 제법 있을 겁니다.
4. 프로젝트가 병렬성에 대한 고려 없이 구현된 경우. 순차적 환경에서 극단적으로 잘 동작하지만 병렬 환경에서는 그러지 못하는 기술이 엄청나게 많음을 놓고 생각해 보면, 여러분의 프로젝트가 순차적 하드웨어 위에서만 대부분의 시간을 수행되어 왔다면, 여러분의 프로젝트는 의심의 여지 없이 병렬성에 친화적이지 않은 코드를 많이 가지고 있을 겁니다.
5. 좋은 소프트웨어 환경 실천법에 대한 고려 없이 프로젝트가 구현된 경우. 잔인한 사실은 공유 메모리 병렬 환경이 종종 순차적 환경에 비해 부주의한 개발 실천에 훨씬 덜 자비롭다는 사실입니다. 병렬화를 시도하기 전에 현재의 설계와 코드를 정리하는게 훨씬 좋을 겁니다.
6. 여러분의 프로젝트의 개발을 시작했던 사람들이 그 후 다른 곳으로 자리를 옮겨서, 남아있는 사람들이 그걸 유지보수하거나 작은 기능들을 추가하는

건 잘 하지만, “커다란 야수같은” 변경을 만들긴 불 가능한 경우. 이 경우, 여러분의 프로젝트를 병렬화하기 위한 매우 간단한 방법을 찾아내지 못한다면 여러분은 그걸 순차적으로 놔두는 게 최선일 겁니다. 그러나, 여러분의 프로젝트를 병렬화하기 위한 다양한 간단한 방법들이 있는데, 여러 인스턴스를 수행하는 것, 많이 사용되는 라이브러리 함수의 병렬 버전을 사용하는 것, 데이터베이스와 같은 다른 병렬 프로젝트를 사용하는 것 등이 여기 포함됩니다.

이러한 장애물 중 다수는 근본적으로 기술적인 것이 아니라 주장할 수도 있겠습니다만, 그게 그걸 덜 현실적으로 만들지 않습니다. 짧게 말해서, 거대한 코드를 병렬화하는 것은 거대하고 복잡한 일이 될 수 있습니다. 다른 거대하고 복잡한 일들과 마찬가지로, 그걸 시작하기 전에 여러분의 숙제를 끝마쳐 두는 게 좋습니다.



E.3 Hardware and its Habits

Quick Quiz 3.1:

병렬 프로그래머는 왜 하드웨어의 저수준 특성을 배우는 걸 신경써야 하죠? 추상화의 윗단에 머무르는 게 더 쉽고, 낫고, 더 우아하지 않을까요?



Answer:

하드웨어의 자세한 성격을 이해하는 것 또한 일을 쉽게 만들겠지만, 대부분의 경우 그건 바보같은 일이 됩니다. 병렬성의 유일한 목적이 성능을 높이는 것임을 받아들이신다면, 그리고 성능은 하드웨어의 자세한 특성에 의존적임을 받아들이신다면, 병렬 프로그래머는 최소 몇가지의 하드웨어 특성을 알아야 함을 논리적으로 받아들이실 겁니다.

이건 대부분의 엔지니어링 격언에 있는 내용입니다. 다리를 만드는데 사용되는 콘크리트와 강철의 특성을 이해하지 못하는 엔지니어가 설계한 다리를 여러분은 사용하고 싶은가요? 아니라면, 병렬 프로그래머가 최소 한 약간의 하드웨어에 대한 이해조차 없이 훌륭한 병렬 소프트웨어를 개발할 수 있을 거라고 생각하십니까?



Quick Quiz 3.2:

대체 어떤 기계가 다양한 데이터 원소에 대한 어토믹 오퍼레이션을 허용할 수 있죠?



Answer:

이 질문에 대한 답변 중 하나는 데이터의 여러 원소들을 하나의 기계 단위에 답을 수 있으며, 그렇게 되면 그것들은 원자적으로 처리될 수 있다는 것입니다.

좀 더 시대에 맞춘 답변은 트랜잭션 메모리 (transactional memory) 를 지원하는 기계가 존재한다는 것입니다 [Lom77, Kni86, HM93]. 2014년 초 기준으로, 여러 주류 시스템이 제한적인 하드웨어 트랜잭션 메모리 구현을 제공하고 있는데, Section 17.3 에서 좀 더 자세한 내용을 다룹니다. 소프트웨어 트랜잭션 메모리의 적용성에 대해선 여전히 평가가 유보적 인데 [MMW07, PW07, RHP⁺07, CBM⁺08, DFGG11, MS12], Section 17.2 에서 다루겠습니다.



Quick Quiz 3.3:

그래서 CPU 설계자들은 캐시 미스의 오버헤드도 많이 줄였나요?



Answer:

불행히도, 별로 그렇지 못합니다. 일부 CPU 에서는 오버헤드가 조금 줄었지만, 빛의 유한한 속도와 물질의 원자적 특성이 거대한 시스템에서의 캐시 미스 오버헤드 감소를 제한합니다. Section 3.3 에서 일부 가능할법한 미래의 발전에 대해 논합니다.



Quick Quiz 3.4:

이게 단순화된 거라구요? 어떻게 이보다 더 복잡한 일이 가능한가요?



Answer:

이건 여러 가능한 복잡 요소를 무시하고 있는데, 다음과 같은 것들입니다:

1. 다른 CPU 들이 이 캐시라인이 연관되는 메모리 참조 오퍼레이션을 동시에 수행할 수도 있습니다.
2. 이 캐시라인은 여러 CPU 의 캐시에 읽기 전용으로 복사되어 있을 수도 있는데, 이 경우는 모든 캐시로부터 이 캐시라인이 제거되어야 합니다.
3. CPU 7은 자신의 캐시로부터 이 캐시라인을 제거해서 (예를 들어, 다른 데이터를 위한 공간을 만들기 위해), 이 요청이 도착한 시점에서는 이 캐시라인이 메모리로 향하고 있을 수 있습니다.
4. 고쳐질 수 있는 어떤 에러가 이 캐시라인에 발생했을 수 있는데, 그렇다면 데이터가 사용되기 전 언젠가는 이 에러가 고쳐져야 합니다.

제품 수준의 캐시 일관성 메커니즘은 이런 종류의 고려할 부분들 때문에 굉장히 복잡합니다 [HP95, CSG99, MHS12, SHW11].



Quick Quiz 3.5:

왜 이 캐시라인을 CPU 7 의 캐시로부터 제거해야 하나요?



Answer:

이 캐시라인이 CPU 7 의 캐시로부터 제거되지 않았다면, CPU 0 과 7 은 이 캐시라인의 변수들에 다른 값을 가질 수 있습니다. 이런 종류의 비일관성은 병렬 소프트웨어를 무척 복잡하게 만들 수 있으므로, 현명한 하드웨어 설계자들은 이를 막습니다.



Quick Quiz 3.6:

Table 3.1 는 CPU 0 가 CPU 224 와 코어를 공유한다고 하는데요. 그건 CPU 1 이 되어야 하는거 아닌가요???



Answer:

이것에 대해 이상하다고 생각하기 쉽지만, `/sys/devices/system/cpu/cpu0/cache/index0/shared_cpu_list` 파일은 정말로 0, 224 라는 문자열을 가지고 있습니다. 따라서, CPU 0 의 하이퍼쓰레드 쌍둥이는 정말로 CPU 224 입니다. 어떤 사람들은 많은 워크로드에서 두번째 하이퍼쓰레드는 추가적 성능을 크게 내지는 못함을 인용하며 이 숫자 규칙이 순진한 어플리케이션과 스케줄러가 더 나은 성능을 낼 수 있을 거라 예상합니다. 이 예상은 순진한 어플리케이션과 스케줄러는 CPU 를 숫자 순서로 활용해서, 더 약한 하이퍼쓰레드 쌍둥이 CPU 들을 다른 모든 코어들이 사용되기 전까지는 사용되지 않게 놔둘거라 가정합니다.



Quick Quiz 3.7:

분명 하드웨어 설계자들은 이 상황을 개선하려 노력했을 수 있을 거예요! 왜 그들은 이 단일 명령 오퍼레이션의 끔찍한 성능에 만족하고 있는거죠?



Answer:

하드웨어 설계자들은 이 문제를 위해 일을 해왔습니다, 그리고 물리학자 Stephen Hawking 같은 선각자들에게 자문을 구했습니다. Hawking 의 발견은 하드웨어 설계자들이 두개의 기본적 문제를 [Gar07] 가지고 있다는 것이었습니다.

1. 빛의 유한한 속도, 그리고

Table E.1: Performance of Synchronization Mechanisms on 16-CPU 2.8 GHz Intel X5550 (Nehalem) System

Operation	Cost (ns)	Ratio (cost/clock)
Clock period	0.4	1.0
Same-CPU CAS	12.2	33.8
Same-CPU lock	25.6	71.2
Blind CAS	12.9	35.8
CAS	7.0	19.4
Off-Core		
Blind CAS	31.2	86.6
CAS	31.2	86.5
Off-Socket		
Blind CAS	92.4	256.7
CAS	95.9	266.4
Off-System		
Comms Fabric	2,600	7,220
Global Comms	195,000,000	542,000,000

2. 물질의 원자적 본성.

이 첫번째 문제는 기본 속도를 제한하며, 두번째 문제는 소형화를 제한해서, 결국 빈도를 제한합니다. 그리고 이는 현재 제품들의 주파수를 10 GHz 아래로 제한하고 있는 에너지 소비 이슈를 회피하기도 합니다.

또한, page 23 의 Table 3.1 은 448 하드웨어 쓰레드의 거대 시스템을 보입니다. 16 하드웨어 쓰레드를 갖는 훨씬 작은 시스템을 보이는 Table E.1 에 나타난 것처럼 더 작은 시스템은 종종 더 나은 응답시간을 갖습니다. 비슷한 모습이 Table 3.1 의 두개의 “Off-core” 열까지에 보여져 있습니다.

더 나아가서, 제가 이걸 타이핑 하고 있는 랩톱과 같은 더 작은 새로운 단일 소켓 시스템은 더욱 합리적인 응답시간을 갖는데, Table E.2 에 이 점이 보여져 있습니다.

대안적으로, 1990년대 중반의 64-CPU 시스템은 5 마이크로세컨드를 넘는 인터컨넥트간 응답시간을 가졌고, 따라서 Table 3.1 에 보인 8 소켓 448 하드웨어 쓰레드의 괴물 같은 머신은 25년전의 비슷했던 것에 비하면 5 배가 넘는 개선을 이뤘음을 보입니다.

하드웨어 쓰레드를 하나의 다이 위의 단일 코어와 여러 코어들에 병합시키는게 응답시간을 상당히 개선시켰는데, 최소한 단일 코어 또는 단일 다이 내로 국한해서는 그렇습니다. 전체 시스템 응답시간에 있어서도 약간 개선이 있었습니다만, 대략 100배 정도에 그쳤습니다. 불행히도, 빛의 속도도 물질의 원자적 본성도 지난 수년간 크게 바뀌지 않았습니다 [Har16]. 따라서, 공간적/

Table E.2: CPU 0 View of Synchronization Mechanisms on 12-CPU Intel Core i7-8750H CPU @ 2.20 GHz

Operation	Cost (ns)	Ratio (cost/clock)	CPUs
Clock period	0.5	1.0	
Same-CPU CAS	6.2	13.6	0
Same-CPU lock	13.5	29.6	0
In-core blind CAS	6.5	14.3	6
In-core CAS	16.2	35.6	6
Off-core blind CAS	22.2	48.8	1–5,7–11
Off-core CAS	53.6	117.9	1–5,7–11
Off-System			
Comms Fabric	5,000	11,000	
Global Comms	195,000,000	429,000,000	

시간적 지역성은 동시성 소프트웨어의 최우선 걱정거리입니다, 상대적으로 작은 시스템에서 돌아가고 있다 해도 말이죠.

Section 3.3은 병렬 프로그래머들의 역겨울 완화시키기 위해 하드웨어 설계자들이 다른 어떤 걸 할 수 있을지 알아봅니다.



Quick Quiz 3.8:

이 숫자들은 미친듯이 크군요! 이걸 어떻게 제 머리로 이해할 수 있을까요?



Answer:

두루마리 휴지 한개를 가져와 보세요. 미국에서, 두루마리 휴지는 일반적으로 350–500 칸으로 되어 있습니다. 하나의 클락 사이클을 나타내기 위해 한 칸을 떼어서 한쪽에 두세요. 이제 두루마리 휴지의 나머지를 펼쳐보세요.

그 결과 나오는 두루마리 휴지의 더미가 하나의 CAS 캐쉬 미스를 나타낼 겁니다.

더 비싼 시스템간 통신 응답시간을 위해선, 그 응답시간을 나타내기 위해 두루마리 휴지 여러개 (또는 여러 상자)를 사용하세요.

중요한 절약 팁: 두루마리 휴지를 준비할 때에는 그것들이 여러분의 일생 동안 모두 사용될 양인지도 생각해 두세요!¹³



Quick Quiz 3.9:

하지만 전자 각각은 그렇게 빠르게 전혀 움직이지 못해요, 컨덕터 (conductor) 내에서도요!!! 세미컨덕터 전압 수준에서의 컨덕터 내 전자의 속도는 초당 오직 밀리미터 수준이라고요. 어떻게 된거죠???



Answer:

전자의 표류 속도는 개별 전자의 장시간 이동을 추적해서 나옵니다. 개별 전자들은 상당히 무작위적으로 튀어다니며, 따라서 그들의 순간적 속도는 매우 높지만, 장시간으로 보면 그리 많이 움직이지 않았음이 드러납니다. 여기서 전자는 그들의 시간 대부분을 완전한 고속도로에서의 속도로 여행하지만 장기적으로는 어디로도 가지 않는 장거리 통근자를 닮았습니다. 이런 컴퓨터의 속도는 시간당 70마일 (시속 113 킬로미터)는 될테지만, 이 행성의 표면에 상대적으로 측정한 그들의 장기간 표류 속도는 0에 수렴할 겁니다.

따라서, 우린 전자의 표류 속도가 아니라 순간 속도에 주목해야 합니다. 하지만, 그 순간 속도조차 빛의 속도의 발끝에도 가지 못합니다. 하지만, 컨덕터 내에서 측정된 전자기파의 속도는 빛의 속도에 조금은 근접하며, 따라서 여전히 미스터리가 남아있습니다.

또 다른 트릭은 전자가(원자적 관점에서 보면) 상당한 거리에 있는 다른 전자들과 상호작용을 한다는 겁니다, 그것들의 음전하 덕분에요. 이 상호작용은 빛의 속도로 움직이는 포톤 (photon)에 의해 행해집니다. 따라서 전자기파의 전자가 있지만, 빠른 일의 대부분을 하는 건 포톤입니다.

통근자 비유를 연장해 보자면, 운전자는 사고나 교통체증을 다른 운전자에게 알리기 위해 스마트폰을 사용할 수도 있고, 따라서 교통 흐름 상의 변화는 개별 차의 순간 속도보다도 훨씬 빠르게 전파될 수 있습니다. 전자기파와 교통 흐름간의 비유를 요약하자면:

¹³ 특히나 2020년 초, 코로나 바이러스가 창궐하는 한가운데 상점들에 두루마리 휴지가 잘 남아있지 않은 상황에서는요!

- (무척 느린) 전자의 표류 속도는 통근자의 장기적 속도에 유사해서, 둘 다 거의 0에 가깝습니다.
- (여전히 느린) 전자의 순간 속도는 도로위의 자동차의 속도와 유사합니다. 둘 다 표류 속도에 비해선 무척 빠르지만, 변화가 전파되는 속도에 비하면 상당히 느립니다.
- (훨씬 빠른) 전자기파의 전파 속도는 기본적으로 표토니 전자기력을 전자들 사이에 전달하기 때문입니다. 비슷하게, 교통 상황은 운전자간의 소통 때문에 상당히 빨리 변화될 수 있습니다. 이게 이미 교통 체증에 빠진 운전자에겐 특정 커패시터 (capacitor)에 빠진 전자에게만큼이나 도움이 되진 못하겠지만요.

물론, 이 주제를 완전히 이해하려면, 전기역학을 읽어보셔야 합니다.



Quick Quiz 3.10:

분산시스템에서의 통신이 그렇게 무섭도록 비용이 높다면, 왜 누군가는 그런 시스템을 사용하려 하나요?



Answer:

여러 이유가 있습니다:

- 공유메모리 멀티프로세서 시스템은 엄격한 크기 제한이 있습니다. 수천개 이상의 CPU 가 필요하면, 분산 시스템을 사용하는 것 외에는 선택의 여지가 없습니다.
- 거대한 공유메모리 시스템은 작은 것들보다 단위 연산별 비용이 비싼 경향이 있습니다.
- 거대한 공유메모리 시스템은 작은 것들보다 훨씬 긴 캐쉬 미스 지연시간을 갖는 경향이 있습니다. 이를 자세히 보기 위해, page 23 의 Table 3.1 를 Table E.2 와 비교해 보시기 바랍니다.
- 분산시스템에서의 통신 오퍼레이션은 많은 CPU 시간을 사용할 이유가 없으므로, 계산 작업은 메세지가 전송되는 동안 병렬로 진행될 수 있습니다.
- 많은 중요한 문제들은 “당황스럽도록 병렬적” 이어서 매우 적은 수의 메세지들로 무척 거대한 양의 일처리가 가능해 집니다. SETI@HOME [Uni08b]은 그런 어플리케이션의 한가지 예일 뿐입니다. 이런 종류의 어플리케이션은 극단적으로 긴 통신 지연시간에도 불구하고 컴퓨터 네트워크를 잘 사용할 수 있습니다.

따라서, 거대한 공유메모리 시스템은 분산컴퓨팅에서 제공되는 것보다 빠른 지연시간을 가질 때 이득을 보는, 그리고 거대한 공유 메모리에서 이득을 보는 어플리케이션들을 위해 사용되는 경향을 가집니다.

병렬 어플리케이션에 대한 노력이 계속됨에 따라 긴 통신 지연시간을 갖는 기계 또는 클러스터에서 잘 돌아가는 당황스럽도록 병렬적인 어플리케이션의 수는 늘어날 가능성이 큰데, 비용 절감이 이를 이끄는 역할을 할겁니다. 그렇다고는 하나, 크게 줄어든 하드웨어 지연시간은 크게 환영 받는 발전이 될텐데, 단일 시스템에도 분산 컴퓨팅에도 그렇습니다.



Quick Quiz 3.11:

좋아요, 우리가 분산 프로그래밍 기술을 공유메모리 병렬 프로그램에 적용해야 할 거라면, 그냥 항상 분산 기술을 사용하고 공유 메모리는 생략하는게 어떤가요?



Answer:

프로그램의 작은 부분만이 성능에 중요한 경우가 자주 있기 때문입니다. 공유 메모리 병렬성은 우리가 그 작은 부분에 대해서만 분산 프로그래밍 기술에 집중할 수 있도록 해서, 프로그램의 성능에 중요하지 않은 많은 부분에는 더 간단한 공유 메모리 기술을 사용할 수 있게 합니다.



E.4 Tools of the Trade

Quick Quiz 4.1:

이것들을 도구라고 부르시나요??? 이것들은 제게는 그보다는 저수준 (low-level) 동기화 도구들처럼 보이는데요!



Answer:

그것들은 실제로 저수준 동기화 도구들이기 때문에 그렇게 보일 겁니다. 그리고 그것들은 실제로 저수준 동시성 소프트웨어를 만들기 위한 기본적 도구들입니다.



Quick Quiz 4.2:

하지만 이 웃긴 셀 스크립트는 진짜 병렬 프로그램이 아니예요! 왜 이런 사소한 걸 신경쓰죠???



Answer:

여러분은 절대 간단한 것을 잊지 말아야 하기 때문입니다!

이 책의 제목은 “Is Parallel Programming Hard, And, If So, What Can You Do About It?” 임을 명심하시기 바랍니다. 이를 위해 여러분이 할 수 있는 가장 효과적인 일은 간단한 걸 있는 걸 예방하는 겁니다! 어쨌건, 여러분이 병렬 프로그래밍을 어려운 방식으로 할 것을 선택한다면, 여러분 스스로밖에는 욕할 사람이 없을 겁니다.



Quick Quiz 4.3:

병렬 셀 스크립트를 작성할 수 있는 더 간단한 방법이 있을까요? 있다면, 어떻게 하죠? 없다면, 왜 없죠?



Answer:

간단한 한가지 방법은 셀 파이프라인을 사용하는 겁니다:

```
grep $pattern1 | sed -e 's/a/b/' | sort
```

충분히 큰 입력 파일에 대해서, grep은 패턴 매칭을 sed가 수정을 하고 sort가 입력을 처리하는 동안 병렬로 수행할 겁니다. 셀 스크립트의 병렬성과 파이프라인 방식에 대한 데모를 위해 `parallel.sh` 파일을 보시기 바랍니다.



Quick Quiz 4.4:

하지만 스크립트 기반의 병렬 프로그래밍이 그렇게 쉽다면, 다른 것들에 신경쓰는 이유가 뭔가요?



Answer:

사실, 오늘날 사용되는 병렬 프로그램들 중 많은 것들이 스크립트 기반일 가능성이 무척 높습니다. 하지만, 스크립트 기반 병렬성은 나름의 한계를 가지고 있습니다:

1. 새로운 프로세스의 생성은 `fork()` 와 `exec()` 이라는 비싼 시스템 콜을 사용하기 때문에 무척 무거운 작업입니다.
2. 데이터 공유와 파이프라이닝의 포함은 보통 비싼 파일 I/O 를 사용되게 합니다.
3. 스크립트에서 사용할 수 있는 안정적인 동기화 도구들은 일반적으로 비싼 파일 I/O 를 사용되게 합니다.
4. 스크립트 언어는 너무 느린 경우가 많습니다. 다만 저수준 프로그래밍 언어로 쓰여진 오랫동안 수행되는 프로그램들을 실행시키는데 사용되기에에는 무척 유용한 경우가 많습니다.

이 한계들은 스크립트 기반의 병렬성이 일의 각 단위가 최소 수십 밀리세컨드, 가능하다면 그보다 더 긴 실행시간을 가지는 경우에 잘 동작하는 간결한 방식을 사용해야 하게 합니다.

더 세밀한 병렬성이 필요한 곳에는 그게 더 간결한 형태로 그 문제가 표현될 수 있을지 그 문제에 대해 고민해보는 것이 좋습니다. 그게 불가능하다면, Section 4.2에서 이야기 되는 다른 병렬 프로그래밍 환경을 사용하는 것을 고려해 봐야 합니다.



Quick Quiz 4.5:

이 `wait()` 함수는 왜 그리 복잡하죠? 그냥 셀 스크립트의 `wait`처럼 동작하게 만드는게 어떤가요?



Answer:

일부 병렬 어플리케이션은 특정 자식 프로세스가 종료되었을 때 특수한 행동을 취해야 하며, 따라서 각 자식을 개별적으로 기다릴 수 있어야 합니다. 또한, 일부 병렬 어플리케이션은 해당 자식이 종료된 이유를 감지해야 합니다. Listing 4.2에서 봤듯이, `wait()` 함수를 가지고 `waitall()` 함수를 만드는 건 어렵지 않지만, 그 반대는 불가능할 겁니다. 특정 자식에 대한 정보가 일단 사라지면, 그건 사라지는 겁니다.



Quick Quiz 4.6:

`fork()` 와 `wait()` 에 대해서 이야기할 게 더 많지 않나요?



Answer:

실제로 그렇습니다. 그리고 이 섹션은 메세징 기능 (UNIX 파일, TCP/IP, 공유 파일 I/O 등) 과 메모리 매핑 기능 (`mmap()` 과 `shmget()` 등) 등을 추가한 버전으로 미래에 확장될 수도 있습니다. 그전까지는 이 기능들을 상당히 자세하게 다루는 책이 많이 있으며, 정말 동기가 부여된 사람은 `manpage`, 이 기능들을 사용하는 현존하는 병렬 어플리케이션들, 뿐만 아니라 이것들을 구현하는 리눅스 커널의 소스 코드를 읽을 수 있을 겁니다.

Listing 4.3의 부모 프로세스는 자식 프로세스가 `printf()` 를 실행한 후에 종료될 때까지 기다림을 알아두시는 게 중요합니다. `printf()` 의 버퍼 I/O 동시성을 여러 프로세스에서 같은 파일에 사용하는 것은 간단하지 않으며, 그렇게 하지 않는 게 제일 좋습니다. 여러분이 정말로 동시적 버퍼 I/O 를 해야만 한다면, 여러분의 OS의 문서를 참고하십시오. UNIX/Linux 시스템에서는 Stewart Weiss의 강의 노트가 다양한 내용의 예제와 함께 좋은 개요를 제공합니다 [Wei13].



Quick Quiz 4.7:

Listing 4.4 의 `mythread()` 함수가 그냥 리턴해도 된다면, 왜 `pthread_exit()` 을 신경쓰나요?

**Answer:**

이 간단한 예제에서는 그럴 이유가 없습니다. 하지만, `mythread()` 가 별도로 컴파일 된 다른 함수들을 호출하는 좀 더 복잡한 예를 생각해 봅시다. 그런 경우, `pthread_exit()` 은 이 다른 함수들이 `mythread()` 에게 여러 같은 것들을 리턴하지 않고도 이 쓰레드의 수행을 중지시키게 할 수 있습니다.

**Quick Quiz 4.8:**

데이터 레이스의 존재 하에 C 언어는 어떤 보장도 하지 않는다면, 리눅스 커널은 왜 그렇게 많은 데이터 레이스를 갖는 거죠? 리눅스 커널은 완전히 망가져 있다는 말을 하고 싶은 건가요???

**Answer:**

아, 하지만 리눅스 커널은 데이터 레이스의 존재 하에서도 안전한 수행을 가능하게 하는 asm 같은 특수한 GNU 확장 기능을 포함하는, 주의 깊게 선택된 C 언어의 상위 집합을 사용합니다. 거기에 더해서, 리눅스 커널은 데이터 레이스가 특히 문제시 될 수 있는 다수의 플랫폼 위에서는 동작하지 않습니다. 예를 하나 들자면, 32비트 포인터와 16비트 버스를 갖는 임베디드 시스템을 생각해 보세요. 그런 시스템에서는 어떤 포인터로의 저장과 읽기에 연관된 데이터 레이스는 해당 포인터의 기존 값의 아래쪽 16비트와 새 값의 위쪽 16비트가 결합된 값을 리턴하는 읽기가 생기게 할 수도 있을 겁니다.

그러나, 리눅스 커널에서 조차, 데이터 레이스는 상당히 위험할 수 있고 가능한 곳에서는 막아져야만 합니다 [Cor12].

**Quick Quiz 4.9:**

동시에 여러 쓰레드가 같은 락을 쥘 수 있게 하고 싶으면 어떡하죠?

**Answer:**

여러분이 해야 할 첫번째 일은 왜 여러분이 그런 일을 하고 싶은지 스스로에게 묻는 것입니다. 만약 그 대답이 “많은 쓰레드에 의해 읽히지만 가끔씩만 업데이트 되는 데이터가 많기 때문” 이라면, POSIX reader-writer 락이 여러분이 찾는 것일 수 있습니다.

여러 쓰레드가 동일한 락을 잡고 있는 효과를 내는 또 다른 방법은 한 쓰레드가 그 락을 획득하게 하고,

`pthread_create()` 를 사용해 다른 쓰레드를 생성하는 것입니다. 이게 왜 좋은 생각인지에 대한 답은 독자 여러분의 몫으로 남겨둡니다.

**Quick Quiz 4.10:**

왜 Listing 4.5 의 line 6 에 있는 `lock_reader()` 의 인자 를 `pthread_mutex_t` 포인터로 만들지 않는 거죠?

**Answer:**

`pthread_create()` 로 `lock_reader()` 를 전달해야 하기 때문입니다. `pthread_create()` 로 넘길 때 이 함수를 캐스팅 할 수도 있겠지만, 함수 캐스팅은 상당히 보기에도 안좋고 간단한 포인터 캐스팅보다 올바르게 하기가 어렵습니다.

**Quick Quiz 4.11:**

Listing 4.5 의 라인 20 와 47 의 `READ_ONCE()` 와 라인 47 의 `WRITE_ONCE()` 는 왜 있는 거죠?

**Answer:**

이 매크로들은 컴파일러가 동시적으로 접근되는 공유 변수에 문제를 일으킬 수 있는 종류의 최적화를 행하는 것을 방지합니다. 이것들은 특정 단일 변수로의 접근 순서를 바꾸는 것을 막는 것을 제외하고는 CPU에 대해서는 어떤 제약도 가지지 않습니다. 이 단일 변수 제약은 Listing 4.5 의 코드에도 적용되는데, x 변수만이 접근되기 때문입니다.

`READ_ONCE()` 와 `WRITE_ONCE()` 에 대한 더 많은 정보를 위해선, Section 4.2.5 을 읽어 주시기 바랍니다. 여러 쓰레드에 의한 여러 변수로의 접근 순서를 잡기 위한 더 많은 내용을 위해선, Chapter 15 를 읽어 주시기 바랍니다. 그 사이, `READ_ONCE(x)` 는 GCC intrinsic `__atomic_load_n(&x, __ATOMIC_RELAXED)` 와, `WRITE_ONCE(x, v)` 는 GCC intrinsic `__atomic_store_n(&x, v, __ATOMIC_RELAXED)` 와 유사한 부분이 많다는 정도만 알아 두셔도 좋겠습니다.

**Quick Quiz 4.12:**

`pthread_mutex_t` 를 획득하고 해제할 때마다 네 줄의 코드를 써야 하는 건 분명 고통스러울 것 같군요! 더 나은 방법은 없을까요?

**Answer:**

맞습니다! 그리고 그런 이유로, `pthread_mutex_lock()` 과 `pthread_mutex_unlock()` 기능은 이 애러

체크를 하는 함수로 보통 싸여져서 사용됩니다. 나중에는, 우린 이것들을 리눅스 커널의 `spin_lock()` 과 `spin_unlock()` API 와 함께 감쌀 겁니다.

□

Quick Quiz 4.13:

“`x = 0`”는 listing 4.6 의 코드 조각이 낼 수 있는 유일한 결과일까요? 그렇다면, 왜죠? 아니라면, 어떤 다른 결과가 나올 수 있고, 그건 왜일까요?

■

Answer:

아닙니다. “`x = 0`” 가 출력된 이유는 `lock_reader()` 가 락을 먼저 잡았기 때문입니다. `lock_wrietr()` 가 해당 락을 먼저 획득했다면, 결과는 “`x = 3`” 이었을 겁니다. 하지만, 이 코드 조각이 `lock_reader()` 를 먼저 시작했기 때문에, 그리고 멀티프로세서에서 수행되었기 때문에, 보통은 `lock_reader()` 가 락을 먼저 획득할 거라 생각할 겁니다. 하지만, 그런 보장은 존재하지 않습니다, 특히 바쁜 시스템에서는요.

□

Quick Quiz 4.14:

다른 락을 사용하는 건 서로의 중간 상태를 어떤 락을 가지고 바라보는지에 대해 상당한 혼란을 야기할 수 있습니다. 그러니 이런 종류의 혼란을 막기 위해 잘 쓰여진 병렬 프로그램은 같은 락을 사용하도록 강제되어야 할까요?

■

Answer:

동작하고 잘 확장되는 프로그램을 단일 전역 락을 사용해 작성하는게 가끔은 가능하지만, 그런 프로그램들은 규칙에서의 예외입니다. 좋은 성능과 확장성을 위해선 일반적으로는 여러 락을 사용해야 할 겁니다.

이 규칙에서의 한가지 가능한 예외는 “트랜잭션 메모리”로, 이것은 현재 하나의 연구 주제입니다. 트랙잭션 메모리 기능은 롤백이 추가된 상태에서 최적화가 허용된 단일 전역 락으로 대략적으로 생각될 수 있습니다 [Boe09].

□

Quick Quiz 4.15:

Listing 4.7 에 보여진 코드에서, `lock_reader()` 는 `lock_writer()` 가 만들어내는 값들을 모두 볼 것이 보장될까요? 그렇다면 왜일까요, 아니라면 또 왜일까요?

■

Answer:

아닙니다. 바쁜 시스템에서는, `lock_reader()` 는 `lock_writer()` 의 수행 동안 프리엠션 (preemption) 당할 수도 있고, 이 경우에는 `lock_writer()` 가 만들어내는 `x` 의 중간값을 어떤 것도 보지 못할 겁니다.

□

Quick Quiz 4.16:

기다려봐요!!! Listing 4.6 은 공유 변수 `x` 를 초기화 하지 않았는데, 왜 Listing 4.7 에서는 초기화 되어야 했죠?

■

Answer:

Listing 4.5 의 라인 4 을 보시기 바랍니다. Listing 4.6 의 코드가 먼저 수행되므로, 이는 `x` 의 컴파일 타임 초기화에 의존할 수 있습니다. Listing 4.7 의 코드는 그다음에 수행되므로, `x` 의 재초기화가 필요합니다.

□

Quick Quiz 4.17:

모든 곳에서 `READ_ONCE()` 를 사용하는 대신에 간단히 Listing 4.8 의 라인 10 에서 `goflag` 를 `volatile` 로 선언하는 것은 어떤가요?

■

Answer:

이 특정한 경우에 있어서는 `volatile` 선언이 실제로 합리적인 대안입니다. 하지만, `READ_ONCE()` 의 사용은 `goflag` 가 동시적인 읽기와 업데이트의 대상임을 선명하게 나타내는 장점을 갖습니다. `READ_ONCE()` 는 대부분의 액세스가 락으로 보호되지만 (그리고 따라서 변화되지 않을 때) 몇몇 액세스는 락 바깥에서 이루어질 때 특히 유용함을 알아두시기 바랍니다. 여기서 `volatile` 선언은 락 바깥에서의 특수한 액세스를 코드를 읽는 사람이 알기 어렵게 만들 것이며, 락 내에서 컴파일러가 좋은 코드를 만들기 어렵게 하기도 할 겁니다.

□

Quick Quiz 4.18:

`READ_ONCE()` 는 컴파일러에만 영향을 끼치고 CPU 는 건들지 않습니다. Listing 4.8 의 `goflag` 의 값의 변화가 빠르게 각 CPU 로 전파됨을 보장하기 위해 메모리 배리어를 사용해야 하지 않나요?

■

Answer:

아니오, 메모리 배리어는 필요하지 않을 뿐더러 여기서 도움을 주지도 않습니다. 메모리 배리어는 여러 메모리 참조들 사이의 순서를 강제할 뿐입니다. 그것들은 시스템의 한 부분에서 다른 부분으로의 데이터 전파를 촉진시킨다는 보장은 전혀 갖지 않습니다.⁴ 이는 빠른 경험적 법칙을 이깁니다: 여러 쓰레드 사이에 여러 변수를 사용해 통신하고 있는게 아니라면 여러분은 메모리 배리어가 필요 없습니다.

⁴ 메모리 배리어가 데이터 전파를 촉진시킨다는 하드웨어 루머가 지속적으로 있어왔습니다만 확인된 바는 없습니다.

하지만 `nreadersrunning`은 어떨까요? 그건 통신을 위해 사용되는 두번째 변수가 아닌가요? 실제로 그렇습니다, 그리고 해당 쓰레드가 자신이 시작해야 하는지 알기 위한 체크 전에 자신의 존재를 나타냄을 확실히 할 수 있게끔 `__sync_fetch_and_add()` 안에 내포된 메모리 배리어 명령이 실제로 필요합니다.



Quick Quiz 4.19:

예를 들어 GCC의 `__thread` 저장소 클래스를 사용해 선언되는 것 같은 쓰레드별 변수를 접근할 때에도 `READ_ONCE()`를 사용할 필요가 있을까요?



Answer:

상황에 따라서요. 해당 쓰레드별 변수가 그 쓰레드에 의해서만 접근된다면, 그리고 시그널 핸들러에서 접근되지 않는다면, 필요 없습니다. 그렇지 않다면 `READ_ONCE()`가 필요할 가능성이 높습니다. 이 두 상황에 대한 예를 Section 5.4.4에서 보이겠습니다.

이는 어떻게 한 쓰레드가 다른 쓰레드의 `__thread` 변수에 접근할 수 있느냐는 질문을 이끄는데, 그에 대한 답은 이 두번째 쓰레드가 자신의 `__thread` 변수로의 포인터를 첫번째 쓰레드가 접근할 수 있는 어딘가에 저장해 두어야만 한다는 것입니다. 혼한 방법 중 하나는 쓰레드당 하나의 원소를 갖는 링크드 리스트를 유지하며 각 쓰레드의 `__thread` 변수의 주소를 연관된 원소에 저장하는 것입니다.



Quick Quiz 4.20:

단일 CPU의 처리량과 비교하는 건 좀 너무한 거 아닌가요?



Answer:

전혀요. 실제로, 이 비교는 무척 관대한 겁니다. 좀 더 균형잡힌 비교는 락킹 기능이 제거된 상태에서의 단일 CPU 처리량에 대한 것일 겁니다.



Quick Quiz 4.21:

하지만 1마이크로세컨드는 크리티컬 섹션의 크기로 특별히 작은 건 아닙니다. 예를 들어 몇개의 명령만이 들어있는 것 같은, 훨씬 더 작은 크리티컬 섹션이 필요할 땐 어떡해야 하죠?



Answer:

읽혀진 데이터가 절대 바뀌지 않는다면, 그걸 접근하는 동안 어떤 락도 잡고 있을 필요가 없습니다. 해당

데이터가 충분히 가끔씩만 바뀐다면, 수행을 체크포인트하고, 모든 쓰레드를 종료하고, 데이터를 변경한 후, 해당 체크포인트부터 재시작할 수 있습니다.

또 다른 접근법은 쓰레드당 하나의 배타적 락을 두어서, 자신의 락을 획득하는 것으로 거대한 reader-writer 락을 읽기 모드로 획득하게 하고, 모든 쓰레드당 락을 획득하는 것으로 쓰기 모드 획득을 하도록 구현하는 겁니다 [HW92]. 이는 읽기 쓰레드에게는 상당히 잘 동작하지만, 쓰기 쓰레드에게는 쓰레드의 수가 늘어날수록 더 큰 오버헤드를 야기시킬 겁니다.

매우 작은 크리티컬 섹션을 처리하는 다른 효율적 방법들이 Chapter 9에 설명되어 있습니다.



Quick Quiz 4.22:

여기 사용된 시스템은 몇년 이상 되었고, 새로운 하드웨어는 더 빠를 겁니다. 그러나 누가 reader-writer 락이 느려짐에 대해 걱정하겠습니까?



Answer:

일반적으로 새로운 하드웨어는 개선됩니다. 하지만, reader-writer 락이 448 CPU에서 이상적 성능을 내게끔 하기 위해선 수천수만배의 개선을 필요로 할 겁니다. 더 나쁜 것이, CPU의 갯수가 늘어날수록 필요한 성능 향상이 더 커집니다. 따라서 reader-writer 락킹의 성능 문제는 상당한 시간 동안 우리 곁에 있을 겁니다.



Quick Quiz 4.23:

그런 두 종류의 기능이 정말로 필요한가요?



Answer:

엄밀히 말하면, 필요 없습니다. 두번째 집합의 어느 것 이든 그에 연관된 첫번째 집합 내의 것을 사용해 구현될 수 있습니다. 예를 들어, `__sync_nand_and_fetch()`를 다음과 같이 `__sync_fetch_and_nand()`로 구현할 수 있습니다.

```
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 나중값을 리턴하는 것들을 이용해 구현할 수도 있습니다.

하지만, 그 대안인 앞의 형태가 프로그래머에게도 컴파일러/라이브러리 구현자에게도 상당히 편리할 수 있습니다.



Quick Quiz 4.24:

이 어토믹 오퍼레이션들은 밑바닥의 인스트럭션 셋을 이용해 직접적으로 지원되는 단일 어토믹 인스트럭션을 생성하는 경우가 많을 텐데, 그것이 일을 하는데 가장 빠른 방법일까요?

**Answer:**

불행히도, 그렇지 않습니다. 극명한 반대 예제를 위해선 Chapter 5를 읽어보시기 바랍니다.

**Quick Quiz 4.25:**

ACCESS_ONCE()에는 무슨 일이 벌어진 건가요?

**Answer:**

2018년 v4.15 릴리즈에서, 리눅스 커널의 ACCESS_ONCE()는 읽기와 쓰기를 위해 각각 READ_ONCE()와 WRITE_ONCE()로 교체되었습니다 [Cor12, Cor14a, Rut17]. ACCESS_ONCE()는 RCU 코드에 사용되기 위해 만들어졌습니다만, 곧 코어 API가 되었습니다 [McK07b, Tor08]. 리눅스 커널의 READ_ONCE()와 WRITE_ONCE()는 원래의 ACCESS_ONCE() 구현과는 상당히 다른 복잡한 형태로 진화했는데 큰 구조체에도 access-once 기능을 지원하지만 해당 구조체가 하나의 기계 명령어로 로드되고 스托어 될 수 없을 때 로드/스토어가 찢겨지는 가능성을 막기 위해서였습니다.

**Quick Quiz 4.26:**

리눅스 커널의 fork()와 wait() 비슷한 것들엔 무슨 일이 일어났나요?

**Answer:**

그것들은 정말로 존재하지는 않습니다. 리눅스 커널 내에서 수행되는 모든 태스크를 메모리를 공유하는데, 최소한 여러분이 상당한 양의 메모리 매핑 작업을 일일이 하고 싶지 않다면 그렇습니다.

**Quick Quiz 4.27:**

변수 counter가 mutex의 보호 없이 증가되면 어떤 문제가 벌어지나요?

**Answer:**

Load-store 구조를 갖는 CPU에서라면 counter 증가는 아래와 같은 형태로 컴파일 될 겁니다:

```
LOAD counter,r0
INC r0
STORE r0,counter
```

그런 기계에서라면, 두 쓰레드가 동시에 counter의 값을 로드하고, 각자 증가시킨 후, 그 결과를 저장합니다. 그러면 counter의 새 값은 두 쓰레드가 증가시켰음에도 불구하고 이전 값보다 1만큼만 클 겁니다.

**Quick Quiz 4.28:**

Listing 4.14의 global_ptr를 세번 로드하는데 문제가 뭐죠?

**Answer:**

global_ptr가 최초에는 NULL이 아니었지만 어떤 다른 쓰레드가 global_ptr을 NULL로 만들었다고 해봅시다. 더 나아가서 변경된 코드 (Listing 4.15)의 라인 1이 global_ptr이 NULL이 되기 전, 그리고 라인 2 직전에 수행되었다고 생각해 봅시다. 그러면 라인 1은 global_ptr이 NULL이 아니라고 결론을 내어서, 라인 2는 이것이 high_address 보다 낮을 것이라고 결론내리고, 따라서 라인 3가 do_low()에 NULL 포인터를 넘길 텐데, do_low()는 NULL을 처리할 준비가 안되어 있을 수도 있습니다.

이 책의 편집자는 1990년대 초에 DYNIX/ptx 커널의 메모리 할당자에서 정확히 이와 같은 실수를 했습니다. 이 버그를 추적하는데에는 이 책의 편집자만이 아니라 그의 여러 동료들의 휴일 주말을 써야만 했습니다. 짧게 말해서, 이건 새로운 문제도 아니고 스스로 사라질 것도 아닙니다.

**Quick Quiz 4.29:**

Listing 4.18 내의 do_something()과 do_something_else()가 인라인 함수라는 것이 왜 중요한가요?

**Answer:**

gp는 정적 (static) 변수가 아니므로, do_something() 또는 do_something_else()가 각각 컴파일 된다면, 컴파일러는 이 두 함수가 gp의 값을 바꿀 수도 있다고 가정해야 합니다. 이 가능성은 컴파일러가 라인 15에서 gp를 다시 로드하게 만들어서서, NULL 포인터 역참조를 방지합니다.

**Quick Quiz 4.30:**

이런! 그러니까 컴파일러는 언제든 원하면 평범한 변수로의 스托어를 만들어낼 수 없나요?

**Answer:**

감사하게도, 그에 대한 답은 아니오입니다. 이는 컴파일러가 데이터 레이스로부터 숨겨져 있기 때문입니다.

일반적 스토어 직전에 스토어를 만드는 것은 상당히 특수한 경우입니다: 이는 일부 다른 경우들, CPU, 쓰레드, 시그널 핸들러, 또는 인터럽트 핸들러 같은 것들이 만들어진 스토어를 볼 수 있게 되는 것은 해당 코드에 이미 만들어진 스토어 없이도 데이터 레이스가 존재하지 않는다면 불가능합니다. 그리고 해당 코드에 데이터 레이스가 이미 존재한다면, 이는 개발자의 소망과는 관계 없이 컴파일러가 뭐가 됐든 원하는 코드를 만들어낼 수 있게 되는 정의되지 않은 행위의 악령을 이미 호출했을 것입니다.

하지만 원래의 스토어가 `volatile` 이라면, `WRITE_ONCE()`에서처럼, 모든 컴파일러가 여기에는 어떤 다른 쓰레드에게 신호를 주거나 하는 이 스토어에 연관된 부수 작용이 있을 수도 있음을 알게 되어서 해당 변수로의 데이터 레이스로부터 자유로운 액세스를 허용합니다. 스토어를 만들어내는 것을 통해, 컴파일러는 그러지 말아야 하는데 데이터 레이스를 만들어낼 수도 있습니다.

`volatile` 과 어토믹 변수의 경우에, 컴파일러는 쓰기로 만들어내는 것을 명시적으로 금지당합니다.



Quick Quiz 4.31:

하지만 완전한 메모리 배리어는 무척 무겁지 않나요?
Listing 4.29에 필요한 순서만을 강제하는 더 가벼운 방법이 있지 않나요?



Answer:

많은 경우 그렇듯, 답은 “상황에 따라 다르다”입니다. 하지만, 두개의 쓰레드만이 `status` 와 `other_task_ready` 변수에 접근한다면, Section 4.3.5에서 이야기되는 `smp_store_release()` 와 `smp_load_acquire()` 함수만으로도 충분할 겁니다.



Quick Quiz 4.32:

인터럽트나 시그널 핸들러가 그 스스로도 인터럽트 당할 수 있다면 뭘 해야 하나요?



Answer:

그렇다면 해당 인터럽트 핸들러는 다른 인터럽트된 코드가 따르는 규칙을 그대로 따라야만 합니다. 스스로 인터럽트 될 수 없거나 인터럽트하는 핸들러와 어떤 변수도 공유하지 않는 핸들러만이 안전하게 평범한 액세스를 사용할 수 있을 것이며, 그렇다 해도 해당 변수들은 다른 CPU나 쓰레드에 의해 동시에 액세스될 수는 없습니다.



Quick Quiz 4.33:

Per-thread-변수 API가 존재하지 않는 시스템에선 이걸 어떻게 할 수 있을까요?



Answer:

한가지 방법은 `smp_thread_id()`로 인덱스 되는 배열을 만드는 것이고, 또 다른 방법은 `smp_thread_id()`를 배열 인덱스로 매핑하는 해쉬 테이블을 만드는 것이겠습니다—사실 이 API 집합이 pthread 환경에서 사용하는 방법입니다.

또 다른 방법은 부모가 각 per-thread 변수를 위한 필드를 갖는 구조체를 할당하고, 이를 자식 쓰레드에게 쓰레드 생성 시에 넘기는 것입니다. 하지만, 이 방법은 큰 시스템에선 거대한 소프트웨어 엔지니어링 비용을 부과할 수 있습니다. 이를 이해하기 위해, 어떤 커다란 시스템의 모든 전역 변수가 그것들이 C static 변수인가 아닌가에 관계 없이 하나의 파일에 모두 선언되어야 한다고 생각해 보세요!



Quick Quiz 4.34:

셀은 `fork()` 대신 `vfork()`를 사용하지 않을까요?



Answer:

그럴 수도 있습니다만, 그걸 확인하는 건 독자 여러분의 할일로 남겨져 있습니다. 하지만 그동안, 전 우리가 `vfork()`는 `fork()`의 변종이며, 따라서 우린 이 둘을 모두 의미하는 범용적 용어로 `fork()`를 사용할 수 있음에 동의했으면 합니다.



E.5 Counting

Quick Quiz 5.1:

효율적이고도 확장성 있는 카운팅이 왜 어려워야 하죠??? 무엇보다도, 컴퓨터는 카운팅이라는 하나의 목적을 위한 특수 하드웨어를 가지고 있잖아요!!!



Answer:

Section 5.1에서 다루겠지만, 예를 들어 공유된 카운터에 대한 어토믹 오퍼레이션과 같은 간단한 카운팅 알고리즘은 느리고 확장성이 나쁘거나, 부정확합니다.



Quick Quiz 5.2:

네트워크 패킷 카운팅 문제. 여러분이 송수신된 네트워크 패킷의 갯수에 대한 통계를 수집해야 한다고 해봅시다. 패킷은 시스템 상의 어떤 CPU 를 통해서든 송수신 될 수도 있습니다. 더 나아가서 여러분의 시스템이 CPU 마다 초당 수백만개 이상의 패킷을 처리할 수 있다고, 그리고 그 수를 5초마다 세는 시스템 모니터링 패키지가 있다고 해봅시다. 여러분은 이 카운터를 어떻게 구현하겠습니까?

**Answer:**

힌트: 이 카운터를 업데이트 하는 행위는 무척 빨라야 합니다만, 이 카운터는 오백만번의 업데이트에 한번 정도만 읽혀지므로, 이 카운터를 읽는 행위는 상당히 느려도 괜찮습니다. 추가로, 읽혀진 값은 보통 완전히 정확할 필요는 없습니다—어쨌건, 이 카운터는 밀리세컨드당 천번 가량 업데이트 되므로, “실제 값”으로부터 수천 정도는 차이를 갖는 값을 가지고 작업할 수 있어야 합니다. “실제 값”이 이 맥락에서 무엇을 의미하는가에 관계없어요. 하지만, 읽혀지는 값은 일정한 오류만을 가져야 합니다. 예를 들어, 이 수가 수백만 이상의 값이라면 1% 에러는 문제 없을 겁니다만, 이 카운트가 조 이상이 된다면 허용되지 못할 수도 있을 겁니다. Section 5.2 을 읽어 보시기 바랍니다.

**Quick Quiz 5.3:**

대략적 구조체 할당 한계 문제. 어떤 구조체의 할당이 그 구조체의 수가 어떤 한계 (예를 들어 10,000) 를 넘어 셨을 때 실패하도록 하거나 하기 위해 할당된 구조체의 갯수를 유지해야 한다고 생각해 봅시다. 더 나아가서 이 구조체는 수명이 짧아서, 이 한계는 아주 가끔만 넘어서게 되며, “약간 허술한” 대략적 한계가 허용된다고 생각해 봅시다.

**Answer:**

힌트: 이 카운터를 업데이트 하는 행위는 이번에도 무척 빨라야 하지만, 이 카운터는 값이 증가될 때마다 읽혀집니다. 하지만, 읽혀지는 값은 대략적인 정도를 넘어서는 차이는 구분할 수 있어야 한다는 점을 제외하고는 아주 정확하지 않아도 됩니다. Section 5.3 을 읽어보시기 바랍니다.

**Quick Quiz 5.4:**

정확한 구조체 할당 한계 문제. 어떤 구조체의 할당이 그 구조체의 수가 정확한 한계 (예를 들어 10,000) 를 넘어 셨을 때 반드시 실패하도록 하기 위해 할당된 구조체의 갯수를 유지해야 한다고 생각해 봅시다. 더

나아가서 이 구조체는 수명이 짧고, 이 한계는 아주 가끔만 초과되어서, 거의 항상 하나의 구조체만이 실제로 사용되고 있고, 더 나아가서 이 카운터가 정확히 언제 0이 되는지, 예를 들어 그 구조체가 단 하나도 사용되고 있지 않다면 어떤 메모리를 해제하거나 하기 위해 이 카운터가 정확히 언제 0이 되는지 알아야 한다고 해봅시다.

**Answer:**

힌트: 이 카운터의 업데이트는 이번에도 무척 빨라야 하지만 이 카운터가 증가될 때마다 읽혀집니다. 하지만, 읽혀지는 값은 이 값이 이 한계와 0 사이인지, 0이하인지, 또는 한계 이상인지 를 완벽하게 구분해야 한다는 점을 제외하고는 정확하지 않아도 됩니다. Section 5.4 을 읽어 보시기 바랍니다.

**Quick Quiz 5.5:**

제거 가능한 I/O 기기 액세스 카운트 문제. 많이 사용되는 제거 가능한 대용량 저장 장치의 레퍼런스 카운트를 유지해야 한다고 해봅시다, 여러분이 사용자에게 이 기기를 제거하는게 언제 안전한지 말하기 위해서요. 평범한 경우처럼, 이 사용자는 이 기기를 제거하고자 하는 의도를 알릴 것이고, 시스템은 이 사용자에게 그게 언제 안전한지 알려줘야 합니다.

**Answer:**

힌트: 또다시, 이 카운터의 업데이트는 I/O 오퍼레이션의 속도를 낮추지 않기 위해 무척 빠르고 확장성 있어야 합니다만, 이 카운터는 사용자가 이 기기를 제거하고자 할때만 읽혀지므로, 이 카운터의 읽기는 무척 느려도 괜찮습니다. 더 나아가서, 이 사용자가 이미 이 기기를 제거하고자 한다고 알리기 전까지는 이 카운터를 읽을 필요 자체가 없습니다. 또한, 읽혀지는 값은 0과 0이 아닌 값을 완전히 구분할 수 있어야 한다는 점, 그리고 이 기기가 제거되는 중일 때를 제외하고는 정확하지 않아도 괜찮습니다. 하지만, 일단 0이라는 값이 읽혀진다면, 뒤따르는 쓰레드가 이 제거중인 기기의 액세스를 얻게 되는 일을 막기 위한 행동이 이뤄지기 전까지는 그 값을 0으로 유지해야 합니다. Section 5.4.6 을 읽어 보시기 바랍니다.

**Quick Quiz 5.6:**

한가지 더 간단할 수 있는 것은 READ_ONCE() 와 WRITE_ONCE() 를 함께 사용하는 대신 더 간단한 ++ 를 사용하는 것일 수 있습니다. 무엇 때문에 그렇게 추가적인 타이핑을 하죠???



Answer:

컴파일러가 어떻게 문제를 일으킬 수 있는지, 그리고 `READ_ONCE()` 와 `WRITE_ONCE()` 가 이 문제를 어떻게 막을 수 있는지에 대한 내용을 위해 페이지 40 의 Section 4.3.4.1 를 보시기 바랍니다.

□

Quick Quiz 5.7:

하지만 영리한 컴파일러는 Listing 5.1 의 라인 5 이 `++` 연산자와 동일함을 알아차리고 x86 add-to-memory 인 스트럭션을 생성하지 않을까요? 그리고 CPU 캐쉬는 이를 어토믹하게 만들지 않을까요?

■

Answer:

`++` 연산자는 어토믹할 수도 있지만, 그게 C11 `_Atomic` 변수에 행해지는게 아니라면 그래야 한다는 요구사항은 없습니다. 그리고 실제로, `_Atomic` 이 없다면, GCC 는 종종 이 값을 레지스터에 로드하고, 이 레지스터의 값을 증가시킨 후, 그 값을 메모리에 저장해서, 분명히 어토믹하지 않게 합니다.

더 나아가, 컴파일러에게 이 위치는 MMIO 디바이스 레지스터일 수도 있다고 이야기하는 `READ_ONCE()` 와 `WRITE_ONCE()` 내에서의 `volatile` 캐스팅을 알아두시기 바랍니다. MMIO 레지스터는 캐쉬되지 않으므로, 컴파일러에게 있어 이 값 증가 연산이 어토믹하다는 가정은 현명하지 못한 것일 겁니다.

□

Quick Quiz 5.8:

실패 횟수의 8-figure accuracy 는 당신이 이걸 진짜로 테스트 했음을 알립니다. 특히나 버그가 검사하는 것만으로 쉽게 보일 수 있는 이런 경우에 이런 사소한 프로그램을 테스트할 필요가 있을까요?

■

Answer:

간단한 병렬 프로그램은 적으며, 대부분의 날에 저는 간단한 순차적 프로그램도 많지 않을 수 있다고 생각합니다.

그 프로그램이 얼마나 작고 간단한지에 관계 없이, 여러분이 그걸 테스트 하지 않았다면, 그것은 동작하지 않는 겁니다. 그리고 여러분이 그걸 테스트 했더라도, 머피의 법칙은 여전히 몇개의 버그는 숨어 있을 거라 말합니다.

더 나아가, 정확성의 증명은 그 가치가 있지만, 그것이 여기서 사용된 `counttorture.h` 테스트 셋업을 포함해 테스팅을 대체하지는 않을 겁니다.

Not only are there very few trivial parallel programs, and most days I am not so sure that there are many trivial sequential programs, either.

No matter how small or simple the program, if you haven't tested it, it does not work. And even if you have tested it, Murphy's Law says that there will be at least a few bugs still lurking.

Furthermore, while proofs of correctness certainly do have their place, they never will replace testing, including the `counttorture.h` test setup used here. After all, proofs are only as good as the assumptions that they are based on. Finally, proofs can be every bit as buggy as are programs! □

Quick Quiz 5.9:

x 축 상의 가로의 점선은 왜 $x = 1$ 에서 대각선에 붙지 않나요?

■

Answer:

어토믹 오퍼레이션의 오버헤드 때문입니다. x 축 상의 점선은 단일 *non-atomic* 값 증가의 오버헤드를 나타냅니다. 어쨌건, 이상적인 알고리즘은 선형으로 확장하기만 하는게 아니라, 싱글쓰레드 코드에 비교해서도 성능 페널티를 일으키지 않을 겁니다.

이 수준의 이상성은 지나치게 느껴질 수도 있겠으나, 이게 Linus Torvalds 에게 충분히 좋다면, 여러분에게도 충분히 좋을 겁니다.

□

Quick Quiz 5.10:

하지만 어토믹 값 증가는 여전히 무척 빠릅니다. 그리고 짧은 반복문 내에서 하나의 변수를 값 증가시키는 것은 제게 굉장히 비현실적으로 들리는데, 어쨌건, 대부분의 프로그램의 실행은 진짜 일을 하는데 사용되어야지, 자신이 한 일을 세는데 쓰이면 안됩니다! 왜 제가 이걸 빠르게 하는데 신경을 써야 하죠?

■

Answer:

많은 경우에, 어토믹 값 증가는 실제로 여러분에게 충분히 빠를 겁니다. 그런 경우에, 여러분은 어토믹 값 증가를 사용해야 합니다. 그러나, 더 나은 카운팅 알고리즘이 필요한 실제 세계의 상황들이 많이 있습니다. 그런 상황의 예 중 하나는 상당히 최적화 된 네트워킹 스택에서 패킷과 바이트를 세는 것으로, 특히 거대한 멀티프로세서에서라면 이런 종류의 카운팅 작업에 상당히 많은 수행 시간이 사용되기 쉽습니다.

또한, 이 챕터의 시작에서 이야기 했듯이, 카운팅은 공유 메모리 병렬 프로그램에서 마주칠 수 있는 문제들을 잘 보여줍니다.

□

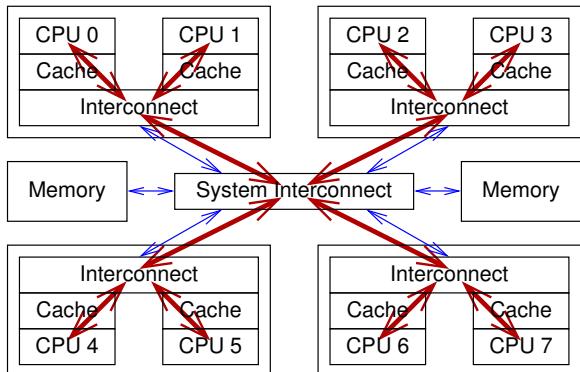


Figure E.1: Data Flow For Global Combining-Tree Atomic Increment

Quick Quiz 5.11:

하지만 왜 CPU 설계자들은 값이 증가되어야 하는 전역 변수를 담고 있는 캐시 라인이 순환될 필요를 없애는 추가적인 기능을 제공하지 않는 거죠?

Answer:

어떤 경우들에는 그렇게 하는게 가능할 수도 있습니다. 하지만, 좀 복잡한 부분이 있습니다:

1. 이 변수의 값이 필요하다면, 이 쓰레드는 이 오퍼레이션이 이 데이터에 도달할 때까지, 그리고 나서는 그 결과가 돌아올 때까지 기다려야 합니다.
2. 이 어토믹 값 증가가 앞 또는 뒤의 오퍼레이션들에 대해 순서지어져야 한다면, 이 쓰레드는 이 오퍼레이션이 이 데이터에 도달할 때까지, 그리고 이 오퍼레이션이 완료되었다는 통지가 돌아올 때까지 기다려야 합니다.
3. CPU 들 사이에서 오퍼레이션을 전달하는 것은 시스템 인터커넥트 상에 더 많은 선로를 필요로 할텐데, 이는 더 많은 디자인 영역과 전력을 소모할 겁니다.

하지만 첫번째 두 조건들이 없다면 어떨까요? 그렇다면 여러분은 평범한 하드웨어에서 이상에 가까운 성능을 달성하는, Section 5.2에서 이야기 하는 알고리즘을 주의 깊게 생각해 봐야 합니다.

앞의 두 조건 중 하나라도 성립된다면, 개선된 하드웨어를 위한 어떤 희망이 있습니다. 하드웨어가 콤바이닝 트리를 구현해서 여러 CPU로부터의 값 증가 요청이 하드웨어에 의해 하나의 값 추가로 결합되는 걸 상상해 볼 수 있겠습니다. 이 하드웨어는 또한 이 요청들에 순서를 적용시켜서, 각 CPU에게 각자의 어토믹 값 증가에 해당하는 값을 리턴시켜줄 수도 있을 겁니다. 이는

이 명령의 응답시간을 $O(\log N)$ 으로 만들텐데, N 은 Figure E.1에 보인 것과 같이 CPU 갯수입니다. 그리고 이런 종류의 하드웨어 최적화를 가진 CPU들이 2011년부터 등장하기 시작했습니다.

이는 Figure 5.2에 보인 현재 하드웨어의 $O(N)$ 성능에 비해 엄청난 개선이며, 3차원 구조 같은 혁신이 실용적인 것으로 증명된다면 추가적인 하드웨어 응답 시간 감소도 가능합니다. 그러나, 어떤 중요한 특수한 경우에 있어서는 소프트웨어가 훨씬 잘 할 수 있음을 볼 겁니다.

□

Quick Quiz 5.12:

하지만 C의 “정수형”이 크기 제한이 있다는 사실이 이를 복잡하게 만들지는 않나요?

■

Answer:

아닙니다, 왜냐하면 더하기는 여전히 상호적이고 결합적입니다. 최소한 unsigned integer 를 사용하는 동안은요. C 표준에서, signed integer 의 오버플로우는 undefined behavior 를 초래함을 기억하시고, 오버플로우 시에 래핑 외의 어떤 일을 하는 기계는 요즘 굉장히 드물다는 사실은 신경쓰지 마세요. 불행히도, 컴퓨터에는 signed integer 가 절대 오버플로우 되지 않을 거라는 가정 하에 최적화를 종종 행하여서, 여러분의 코드가 signed integer 를 오버플로우 나게 한다면, 현대의 two's-complement 하드웨어에서라도 문제를 일으킬 수 있습니다.

그렇다고는 하나, (예를 들어) 32비트 쓰레드당 카운터의 합을 64비트로 만들려 할 때 추가적인 복잡도의 요인이 존재합니다. 이런 추가적 복잡성을 처리하는 것은 독자 여러분의 몫으로 남겨두겠습니다. 이 챕터의 뒷부분에서 소개되는 일부 기법들이 상당히 도움이 될 겁니다.

□

Quick Quiz 5.13:

배열이요??? 하지만 그럼 쓰레드의 수가 제한되지 않나요?

■

Answer:

그럴 수 있습니다, 그리고 이 장난감 구현에서는, 그렇습니다. 하지만 임의의 수의 쓰레드를 허용하는 대안적 구현은 그렇게 어렵지 않은데, 예를 들면 Section 5.2.3에 보여진 GCC의 __thread 기능을 사용하는 겁니다.

□

Quick Quiz 5.14:

GCC 는 이것 외에 어떤 못된 최적화를 할 수 있나요?

**Answer:**

더 많은 정보를 위해 Sections 4.3.4.1 and 15.3 를 참고하시기 바랍니다. 한가지 못된 최적화는 뒤따르는 `read_count()` 함수에 흔한 부가적 표현 제거를 적용하는 것으로, 이는 이 값의 변화가 이 함수로의 이어지는 호출로부터 리턴될 거라고 믿는 코드를 놀라게 할 수도 있을 겁니다.

**Quick Quiz 5.15:**

Listing 5.3 의 `counter` per-thread 변수는 어떻게 초기화되나요?

**Answer:**

C 표준은 전역 변수의 초기값은 명시적으로 초기화되지 않는다면 0이라고 명시하고 있으므로, `counter` 의 모든 인스턴스는 묵시적으로 0으로 초기화 됩니다. 이와는 별개로, 사용자가 통계적 카운터들로부터의 연속된 읽기 사이의 차이에 대해서만 관심이 있다면 이 초기값은 의미 없을 겁니다.

**Quick Quiz 5.16:**

Listing 5.3 의 코드는 복수의 카운터를 어떻게 허용할까요?

**Answer:**

실제로, 이 장난감 예제는 복수의 카운터를 허용하지 않습니다. 복수의 카운터를 제공할 수 있도록 이걸 수정하는 것은 독자 여러분의 과제로 남겨둡니다.

**Quick Quiz 5.17:**

이 읽기 오퍼레이션은 쓰레드당 값을 합하는데 시간을 요하며, 이 시간 동안 이 카운터의 값은 변화될 수 있습니다. 이는 Listing 5.3 의 `read_count()` 를 통해 리턴되는 값은 정확하지 않을 것을 의미합니다. 이 카운터는 단위 시간당 r 카운트의 속도로 증가된다고, 그리고 `read_count()` 의 수행은 Δ 단위 시간을 소모한다고 가정해 봅시다. 리턴되는 값에 예상되는 에러는 얼마정도일까요?

**Answer:**

최악의 경우에 대한 분석을 먼저 하고, 이어서 덜 보수적인 분석을 해봅시다.

최악의 경우, 읽기 오퍼레이션은 순식간에 완료되지만, 리턴하기 전에 Δ 시간의 지연을 가져서, 최악의 경우의 에러는 단순히 $r\Delta$ 가 됩니다.

이 최악의 경우 행동은 가능성이 적으므로, N 개의 카운터 각각으로부터의 읽기가 Δ 시간 동안 동일한 시간을 사용한다고 생각해 봅시다. N 개의 읽기 사이에는 $\frac{4}{N+1}$ 시간의 $N+1$ 개 시간 간격이 존재할 겁니다. 마지막 쓰레드의 카운터로부터의 읽기 이후의 이 지연으로 인한 에러는 $\frac{r\Delta}{N(N+1)}$ 이 될 것이며, 뒤에서 두번째 쓰레드의 카운터로부터의 에러는 $\frac{2r\Delta}{N(N+1)}$, 뒤에서 세번째 쓰레드는 $\frac{3r\Delta}{N(N+1)}$, 이런 식으로 될 겁니다. 총 에러는 각 쓰레드의 카운터로부터의 읽기로 인한 에러들의 합이 되므로:

$$\frac{r\Delta}{N(N+1)} \sum_{i=1}^N i \quad (E.1)$$

이 합을 닫힌 형태로 정리해 보면:

$$\frac{r\Delta}{N(N+1)} \frac{N(N+1)}{2} \quad (E.2)$$

중복을 제거하면 다음과 같이 직관적으로 예상된 결과가 나옵니다:

$$\frac{r\Delta}{2} \quad (E.3)$$

호출자가 이 읽기 오퍼레이션을 통해 리턴된 카운트를 사용하는 코드를 수행하는 동안에도 이 에러는 증가함을 기억해 두는 것이 중요합니다. 예를 들어, 리턴된 카운트의 값에 기반한 어떤 계산을 수행하는 데에 t 시간을 소모했다면, 최악의 경우의 에러는 $r(\Delta + t)$ 로 증가할 겁니다.

예상되는 에러 역시 비슷하게 다음과 같이 증가합니다:

$$r \left(\frac{\Delta}{2} + t \right) \quad (E.4)$$

물론, 어떤 경우에는 읽기 오퍼레이션 동안 카운터의 값이 계속 증가하는게 받아들여질 수 없을 때도 있습니다. Section 5.4.6 은 이 상황을 처리하는 방법을 이야기합니다.

지금까지, 우리는 값이 증가하기만 하지 감소하지는 않는 카운터를 생각해 봤습니다. 만약 이 카운터 값이 단위 시간당 r 카운트 만큼만 바뀌지만, 어떤 방향에서든 그렇다면, 우린 이 에러가 줄어들 것이라고 예상해야 합니다. 하지만, 최악의 경우는 카운터가 양방향으로 움직일 수 있다고 해도 이 최악의 경우는 읽기 오퍼레이션이

순식간에 완료되지만 4 시간 단위 동안 지연되고, 그동안 이 카운터의 값이 같은 방향으로 변화해 절대적인 에러는 rA 가 되므로 변하지 않습니다.

증가와 감소의 패턴에 대한 가정에 기반해 평균 에러를 계산하기 위한 여러 방법이 있습니다. 단순하게 하기 위해, 오퍼레이션들 중 f 정도가 감소 오퍼레이션의 비율이고, 관심 있는 에러는 카운터의 장시간 경향선으로부터의 초이라고 생각해 봅시다. 이 가정 하에서라면, f 가 0.5 이하일 때, 각 값 감소는 증가에 의해 취소되어서, $2f$ 의 오퍼레이션이 서로를 취소시킬 것이어서, $1 - 2f$ 의 오퍼레이션이 취소되지 않는 값 증가로 남게 됩니다. 반면에, f 가 0.5 보다 크다면, $1 - f$ 의 카운터 이동이 최소되어서 $-1 + 2(1 - f)$ 만큼 음의 방향으로 카운터가 이동하는데, 이는 $1 - 2f$ 로 간략화 되므로, 어떤 경우든 이 카운터는 오퍼레이션 당 평균 $1 - 2f$ 만큼 움직이게 됩니다. 따라서, 카운터의 장시간 움직임은 $(1 - 2f)r$ 이 됩니다. 이를 수식 E.3에 대입해 보면:

$$\frac{(1 - 2f)rA}{2} \quad (E.5)$$

이 모든 것을 뒤로 하고, 대부분의 통계적 카운터 사용의 경우, `read_count()`에 의해 리턴되는 값의 에러는 상관없습니다. 이 관계 없음은 `read_count()`이 수행되는 데에 필요한 시간은 일반적으로 연속되는 `read_count()` 호출 사이의 시간 간격에 비해 무척 짧기 때문입니다.

□

Quick Quiz 5.18:

Listing 5.4의 명시적인 `counterp` 배열은 쓰레드 수에 대한 임의의 제한을 다시 암시하지 않나요? 왜 GCC는 쓰레드들이 서로의 쓰레드별 변수를 쉽게 접근할 수 있게끔 리눅스 커널의 `per_cpu()` 기능과 비슷한 `per_thread()` 인터페이스를 제공하지 않나요?

■

Answer:

정말로, 왜그럴까요?

공정하게 말하자면, GCC는 리눅스 커널이 무시할 수 있는 일부 어려움을 겪게 됩니다. 사용자 수준 쓰레드가 종료될 때, 그것의 쓰레드별 변수는 모두 사라져서, 쓰레드별 변수 접근 문제를 복잡하게 만드는데, 특히 사용자 수준 RCU의 발전 전에 그랬습니다 (Section 9.5을 참고하세요). 대조적으로, 리눅스 커널에서는 CPU가 오프라인이 될 때 해당 CPU의 CPU별 변수는 매핑된 채로 유지되고 액세스될 수 있습니다.

비슷하게, 새로운 사용자 레벨 쓰레드가 생성될 때, 그것의 쓰레드별 변수는 갑자기 존재하게 됩니다. 대조적으로 리눅스 커널에서는 모든 CPU별 변수가 부팅 시점에 매핑되고 초기화 됩니다. 연관된 CPU가 아직

존재하지 않는지에 관계 없이, 또는 연관된 CPU가 존재하기 할지에 관계 없이요.

리눅스 커널이 암시하는 핵심적 한계점은 컴파일 시점에 정해지는 CPU 갯수의 최대 한계, 즉 `CONFIG_NR_CPUS`와 일반적으로 보다 타이트한 부팅 시점의 한계인 `nr_cpu_ids`입니다. 대조적으로, 유저 스페이스에는 쓰레드의 수에 대한 하드코딩된 상한선이 존재할 이유가 없습니다.

물론, 두 환경 모두 동적으로 로드된 코드를 다뤄야 하는데 (유저 스페이스의 경우 동적 라이브러리, 리눅스 커널의 경우 커널 모듈), 이는 쓰레드별 변수의 복잡도를 증가시킵니다.

이런 복잡성이 유저 스페이스 환경이 다른 쓰레드의 쓰레드별 변수를 접근할 수 있게 하는 것을 상당히 어렵게 만듭니다. 그러나, 그런 액세스는 상당히 유용하며, 언젠가는 지원되길 희망되고 있습니다.

그전까지는, 이것과 같은 교재의 예제는 그 한계가 사용자에 의해 쉽게 조정될 수 있는 배열을 사용할 수 있습니다. 대안적으로, 그런 배열은 실행 시점에 필요한 대로 동적으로 할당되고 확장될 수 있습니다. 마지막으로, 링크드 리스트와 같은 변동되는 길이의 자료구조도 유저스페이스 RCU 라이브러리 [Des09b, DMS⁺¹²]에서 그렇듯 사용될 수 있습니다. 이 마지막 방법은 어떤 경우에는 거짓 공유를 줄일 수 있습니다.

□

Quick Quiz 5.19:

Listing 5.4의 라인 19에서의 NULL 체크는 추가적인 브랜치 예측 실패를 일으키지 않나요? 영구히 0 값을 갖는 변수 집합을 갖고 사용되지 않은 카운터 포인터를 NULL로 만드는 대신 그 변수를 가리키게 하는건 어떤가요?

■

Answer:

이건 합리적인 전략입니다. 성능이 어떻게 달라지는지 확인하는 건 독자 여러분에게 남겨두겠습니다. 하지만, 성능을 위한 빠른 경로는 `read_count()`가 아니라 `inc_count()` 임을 명심해 두시기 바랍니다.

□

Quick Quiz 5.20:

Listing 5.4의 `read_count()`에서의 합산을 보호하는 `lock` 같이 무거운 게 도대체 왜 필요한거죠?

■

Answer:

기억하세요, 한 쓰레드가 종료될 때, 그것의 쓰레드별 변수는 사라집니다. 따라서, 특정 쓰레드의 쓰레드별 변수를 그 쓰레드가 종료된 후에 접근하려 하면, `segmentation fault`가 날 겁니다. 이 락은 합산과 쓰레드 종료 간의 순서를 조정해서 이 시나리오를 방지합니다.

물론, 이 대신 reader-writer 락을 read-acquire 할 수도 있습니다만, Chapter 9는 여기서 필요한 조정을 구현하기 위한, 이보다도 가벼운 메커니즘을 소개할 겁니다.

또 다른 방법은 쓰레드별 변수 대신 배열을 사용하는 것으로, Alexey Roystman 이 노트한 바에 따르면 NULL 테스트를 제거할 겁니다. 하지만, 배열로의 액세스는 쓰레드별 변수로의 액세스보다 많은 경우 느리며 배열의 사용은 쓰레드의 수에 대한 고정된 상한 한계를 암시합니다. 또한, `inc_count()` 라는 빠른 경로 상에는 테스트도 락도 필요치 않음을 알아 두시기 바랍니다.



Quick Quiz 5.21:

Listing 5.4의 `count_register_thread()`에서 대체 왜 락을 잡아야만 하죠? 이것은 어떤 다른 쓰레드도 수정하지 않는 위치에 적절히 정렬되어 저장된 하나의 기계 단어이니, 어쨌건 어토믹할 거예요, 그렇죠?



Answer:

이 락은 실제로 없어질 수 있습니다. 그렇지만 이 함수가 쓰레드의 시작 때에만 수행된다는 것, 그리고 따라서 어떤 성능에 중요한 지점도 아니라는 것을 생각하면 나중에 사과하기보단 안전을 중시하는게 낫습니다. 이제, 우리가 이걸 수천개의 CPU를 갖는 기계 위에서 테스트 한다면, 우린 이 락을 없애야 할 수도 있습니다. 그러나 “겨우” 수백개의 CPU를 갖는 기계 위에서라면, 그렇게 까지 할 필요는 없습니다.



Quick Quiz 5.22:

좋아요, 하지만 리눅스 커널은 CPU 별 카운터의 값의 합을 읽을 때 락을 잡을 필요가 없죠. 그런데 왜 유저 스페이스 코드는 이걸 해야만 하죠???



Answer:

기억하세요, 리눅스 커널의 CPU 별 변수는 항상 접근할 수 있습니다, 연관된 CPU가 오프라인일 때 조차도요—이 연관된 CPU가 결코 존재하지 않았고 결코 존재하지 않을 것이라 할지라도요.

한 가지 해결책은 Listing E.1 (`count_tstat.c`)에 보인 것처럼 각 쓰레드가 모든 쓰레드가 종료될 때까지 존재할 것을 보장하는 것입니다. 이 코드의 해석은 독자 여러분의 몫으로 남겨둡니다만, 이는 `countertorture.h` 카운터 평가 프로그램에 약간의 수정을 요함을 알아 두시기 바랍니다. (힌트: `#ifndef KEEP_GCC_THREAD_LOCAL`을 참고하세요.) Chapter 9는 이 상황을 훨씬 우아한 방식으로 처리하는 동기화 메커니즘을 소개합니다.



Listing E.1: Per-Thread Statistical Counters With Lockless Summation

```

1 unsigned long __thread counter = 0;
2 unsigned long *counterp[NR_THREADS] = { NULL };
3 int finalthreadcount = 0;
4 DEFINE_SPINLOCK(final_mutex);
5
6 static __inline__ void inc_count(void)
7 {
8     WRITE_ONCE(counter, counter + 1);
9 }
10
11 static __inline__ unsigned long read_count(void)
12     /* need to tweak counttorture! */
13 {
14     int t;
15     unsigned long sum = 0;
16
17     for_each_thread(t)
18         if (READ_ONCE(counterp[t]) != NULL)
19             sum += READ_ONCE(*counterp[t]);
20     return sum;
21 }
22
23 void count_register_thread(unsigned long *p)
24 {
25     WRITE_ONCE(counterp[smp_thread_id()], &counter);
26 }
27
28 void count_unregister_thread(int nthreadsexpected)
29 {
30     spin_lock(&final_mutex);
31     finalthreadcount++;
32     spin_unlock(&final_mutex);
33     while (READ_ONCE(finalthreadcount) < nthreadsexpected)
34         poll(NULL, 0, 1);
35 }
```

Quick Quiz 5.23:

Listing 5.5의 `inc_count()`는 왜 어토믹 인스트럭션을 사용해야 하나요? 어쨌건, 우린 쓰레드별 카운터를 접근하는 여러 쓰레드가 있는 거잖아요!



Answer:

두 쓰레드 중 하나는 읽기만을 하고 있고, 이 변수는 정렬되었고 기계 단어 크기이므로, 어토믹하지 않은 인스트럭션으로도 충분합니다. 그러나, 카운터 업데이트가 `eventual()`에게 보이는 것을 막을 수도 있는 컴파일러 최적화를 방지하기 위해 `READ_ONCE()` 매크로가 사용됩니다.⁵

이 알고리즘의 이전 버전은 실제로 어토믹 인스트럭션을 사용했습니다, 그것들이 실제로는 불필요했음을 지적해준 Ersoy Bayramoglu에게 감사를 드립니다. 하지만, 32비트 시스템에서 이 쓰레드별 `counter` 변수는 그것들을 올바르게 합산하기 위해 32비트로 제한되어야 할 수도 있습니다만, 64비트 `global_count` 변수가 있어야 오버플로우를 방지할 수 있을 겁니다. 이 경우, 오버플로우를 방지하기 위해 쓰레드별 `counter` 변수를 주기적으로 0으로 초기화 시켜줄 필요가 있는데, 이것은

⁵ `READ_ONCE()`의 간단한 정의가 Listing 4.9에 보여져 있습니다.

어토믹 인스트럭션을 필요로 합니다. 이 초기화는 너무 오래 지연되면 작은 쓰레드별 변수에서의 오버플로우가 초래될 것임을 알아두는게 정말 중요합니다. 따라서 이 방법은 시스템에 리얼타임 요구사항이 존재해야 함을 암시하며, 따라서 각별한 주의가 필요합니다.

반면에, 만약 모든 변수가 같은 크기라면, 어떤 변수의 오버플로우도 위험하지 않은데 결과적인 합은 이 워드 크기로 모듈로 연산될 것이기 때문입니다.



Quick Quiz 5.24:

Listing 5.5 의 `eventual()` 함수에서의 단일 글로벌 쓰레드는 전역 락 만큼이나 심각한 병목 아닐까요?



Answer:

이 경우에는, 아니오. 여기서 일어날 일은 쓰레드의 수가 늘어난다면 `read_count()` 가 반환하는 값이 더욱 부정확해 진다는 것 뿐입니다.



Quick Quiz 5.25:

Listing 5.5 의 `read_count()` 가 반환하는 예상값은 쓰레드의 수가 늘어날수록 더욱 부정확해지지 않나요?



Answer:

그렇습니다. 이게 문제가 된다면, 한가지 수정방법은 여러 `eventual()` 쓰레드를 제공해서, 각자가 자신의 할당량만 처리하는 것입니다. 더 극단적인 경우에는 `eventual()` 쓰레드들의 트리 같은 계층 구조 사용이 필요할 수도 있습니다.



Quick Quiz 5.26:

Listing 5.5 에 보인 결과적으로 일관되는 알고리즘에 서는 읽기도 업데이트도 극단적으로 낮은 오버헤드를 가지고 극단적인 확장성을 갖는데, 왜 어떤 사람들은 읽기 성능이 나쁜, Section 5.2.2 에 보인 구현을 신경쓸까요?



Answer:

`eventual()` 을 수행하는 쓰레드는 CPU 시간을 소비합니다. 이 결과적으로 일관되는 카운터가 계속해서 추가되면 그로 인한 `eventual()` 쓰레드들은 결과적으로 모든 사용 가능한 CPU를 소비할 수도 있습니다. 따라서 이 구현은 또 다른 종류의, 쓰레드나 CPU 의수가 아니라 결과적으로 일관적인 카운터의 수의 의미에서 확장성 제한을 갖습니다.

물론, 다른 트레이드오프를 만드는 것도 가능합니다. 예를 들어, 모든 결과적으로 일관적인 카운터들을 처리하는 단 하나의 쓰레드를 만들어, 이 오버헤드를 단일

CPU 로 제한할 수도 있겠으나, 카운터의 수가 늘어날 수록 업데이트에서 읽기까지의 지연시간이 늘어나는 결과를 초래할 겁니다. 대안적으로, 이 단일 쓰레드는 이 카운터의 업데이트 비율을 추적해서 자주 업데이트 되는 카운터를 더 자주 접근하는 것도 가능하겠습니다. 또한, 이 카운터를 처리하는 쓰레드의 수는 전체 CPU 수의 일정 부분만큼만 되도록 하고, 이를 수행 시간에 조정할 수도 있겠습니다. 마지막으로, 각 카운터는 자신의 지연시간을 명시할 수도 있고, 데드라인 스케줄링 기법이 각 카운터에게 필요한 지연시간을 제공하도록 사용될 수도 있겠습니다.

다른 많은 트레이드오프도 만들어질 수 있음이 분명합니다.



Quick Quiz 5.27:

Listing 5.5 의 `read_count()` 에 의해 반환되는 추정값의 정확도는 어떻게 되나요?



Answer:

이 추정값의 정확도를 평가하는 한가지 간단한 방법은 Quick Quiz 5.17에 설명된 분석 기법을 사용하되 Δ 를 `eventual()` 쓰레드의 다음 수행의 시작까지의 기간으로 설정하는 것입니다. 특정 카운터가 여러 `eventual()` 쓰레드를 갖는 경우도 다루는 것은 독자 여러분의 연습 문제로 남겨두겠습니다.



Quick Quiz 5.28:

패킷의 크기는 다양하다는 점을 놓고 생각할 때, 패킷의 수를 세는 것과 패킷들의 전체 바이트 수를 세는 것 사이에는 어떤 근본적 차이가 있을까요?



Answer:

패킷의 수를 셀 때, 이 카운터는 값 1만큼씩만 증가될 겁니다. 반면, 바이트 수를 셀 때, 이 카운터는 큰 수만큼 증가될 수도 있을 겁니다.

이게 왜 문제가 될까요? 값 1만큼만 증가하는 경우에는, 반환되는 값은 그 카운터가 언젠가는 가졌을 정확한 값일 겁니다. 정확히 그게 언제인지는 말할 수 없지만요. 반면, 바이트를 셀 때에는 두개의 다른 쓰레드가 어떤 전역적 오퍼레이션들의 순서로 일관되지 않은 값을 반환할 수도 있습니다.

이걸 자세히 보기 위해, 쓰레드 0 이 값 3 을 자신의 카운터에 더하고, 쓰레드 1 은 값 5를 자신의 카운터에 더하며, 쓰레드 2 와 3 이 이 카운터를 더한다고 해봅시다. 이 시스템이 “완화된 순서 규칙”을 가지고 있거나 컴파일러가 강력한 최적화를 사용한다면, 쓰레드 2 는 이 합이 2라고 보고 쓰레드 3 은 이 합이 5라고 볼 수 있습니다. 이 카운터의 값의 순서의 가능할 법한 전역적

순서는 0,3,8 과 0,5,8 이고 이 중 어떤 것도 이 얻어진 결과와 일관되지 못합니다.

여러분이 이걸 놓치셨다면, 여러분은 혼자가 아닙니다. Michael Scott 은 Paul E. McKenney 의 박사학위 시험 때 이 질문으로 Paul 에게 한방을 먹였습니다.



Quick Quiz 5.29:

읽기 쓰레드는 모든 쓰레드의 카운터를 합해야 한다는 점을 놓고 보면, 이 카운터 읽기 오퍼레이션은 큰 수의 쓰레드를 가졌을 때 긴 시간을 요할 수 있습니다. 값 증가 오퍼레이션이 빠르고 확장성 있게 유지하면서 읽기 쓰레드들 역시 합리적인 성능과 확장성만이 아니라 좋은 정확도를 취할 수 있는 방법은 없을까요?



Answer:

한가지 방법은 Section 5.2.4 에서 설명된 방법과 비슷하게 전역적 추정치를 유지하는 것입니다. 업데이트 쓰레드는 각자의 쓰레드별 변수 값을 증가시키지만, 그에 어떤 미리 정의된 한계에 다다르면, 어토믹하게 그걸 전역 변수에 더하고 자신의 쓰레드별 변수를 0으로 초기화 합니다. 이는 평균적 값 증가 오버헤드와 읽혀지는 값의 정확성 사이의 트레이드오프를 가능하게 할 겁니다. 특히, 이는 읽기 쪽 비정확성의 바운더리를 명확하게 해 줄 겁니다.

또 다른 방법은 읽기 쓰레드가 종종 값의 특정 변화에 대해서만 신경을 쓰지, 구체적 값에는 별로 신경 쓰지 않는다는 사실을 사용하는 것입니다. 이 방법은 Section 5.3 에서 알아봅니다.

독자 여러분은 다른 방법들을 생각하고 시도해 볼 것이 장려되는데, 예를 들면 컴바이닝 트리를 사용하는 것입니다.



Quick Quiz 5.30:

Listing 5.7 는 왜 Section 5.2 에서 보인 inc_count() 와 dec_count() 인터페이스 대신 add_count() 와 sub_count() 를 제공하는 거죠?



Answer:

구조체는 각자 다른 크기를 가지기 때문입니다. 물론, 특정 크기의 구조체에만 연관된 한계선 카운터는 inc_count() 와 dec_count() 를 사용할 수도 있을 겁니다.



Quick Quiz 5.31:

Listing 5.7 의 라인 3 에서의 이상한 형태의 조건은 뭐죠? Listing 5.8 에서 보인 더 직관적인 형태의 fastpath 는 어떤가요?



Answer:

두 단어로 말할 수 있습니다. “정수형 오버플로우.”

Listing 5.8 의 공식에 counter 가 10이고 delta 가 ULONG_MAX 라고 대입해 보세요. 그리고 나서 Listing 5.7 에 보인 코드를 다시 실행해 보세요.

이 예의 나머지 부분을 위해선 정수형 오버플로우에 대한 좋은 이해가 필요할 것이며, 따라서 여러분이 정수형 오버플로우를 이전에 다뤄본 경험이 없다면, 그걸 위해 여러 예제를 시도해 보시기 바랍니다. 정수형 오버플로우는 가끔 병렬 알고리즘보다도 옳게 처리하기가 어렵습니다!



Quick Quiz 5.32:

Listing 5.7 에서 globalize_count() 는 왜 나중에 balance_count() 를 호출해 그것을 다시 채우기 위한 이유만으로 쓰레드별 변수를 0으로 초기화 시키나요? 왜 그 쓰레드별 변수를 그냥 0이 아닌 채로 두지 않는 거죠?



Answer:

실제로 이 코드의 이전 버전이 그렇게 했습니다. 하지만 더하기와 빼기는 무척 비용이 낮고, 도사리고 있는 모든 특수 경우를 처리하는 건 상당히 복잡합니다. 다시 말하지만, 스스로 해보세요, 하지만 정수형 오버플로우를 조심하세요!



Quick Quiz 5.33:

add_count() 에서는 globalreserve 가 우리를 위해 세어졌는데, Listing 5.7 의 sub_count() 에서는 왜 그 렇지 않나요?



Answer:

globalreserve 변수는 모든 쓰레드의 countermax 변수의 합을 따라갑니다. 이 쓰레드들의 counter 변수의 합은 0과 globalreserve 사이 어디엔가 있을 수 있습니다. 따라서 우리는 모든 쓰레드의 counter 변수가 add_count() 시점에서 꽉 차있다고 가정하고 sub_count() 시점에서 모두 비어 있다고 가정하는 보수적 전략을 취합니다.

하지만 우린 나중에 여기로 돌아올테니, 이 질문을 기억해 두세요.



Quick Quiz 5.34:

Listing 5.7에 보인 `add_count()`를 한 쓰레드가 호출하고, 다른 쓰레드가 `sub_count()`를 호출한다고 해봅시다. 카운터의 값은 0이 아님에도 불구하고 `sub_count()`는 실패를 리턴하지 않을까요?

**Answer:**

실제로 그럴 겁니다! 많은 경우, 이는 Section 5.3.3에서 이야기한 대로 문제가 될 것이며, 그런 경우엔 Section 5.4에서 이야기한 알고리즘이 도움이 될 겁니다.

**Quick Quiz 5.35:**

Listing 5.7에는 왜 `add_count()`와 `sub_count()`가 모두 있죠? 왜 단순히 `add_count()`에 음수를 넘기지 않는 건가요?

**Answer:**

`add_count()`가 그 인자로 `unsigned long`을 받음을 생각해 보면, 음수를 넘기기는 좀 어려울 겁니다. 그리고 여러분이 반물질 메모리를 가지고 있는게 아니라면, 사용 중인 구조체의 수를 세는데 음수를 허용하는 건 큰 의미가 없을 수 있습니다.

이 장난은 뒤로 하고 이야기 하자면, `add_count()`와 `sub_count()`를 합치는 것도 가능할 겁니다만, 이 합쳐진 함수의 `if` 조건은 현재의 함수들 한쌍보다 더 복잡할 것이었, 결국 이 fast path의 더 느린 수행을 의미하게 될 겁니다.

**Quick Quiz 5.36:**

Listing 5.9의 라인 15에서 왜 `counter`를 `countermax / 2`로 설정하는 거죠? 그냥 `countermax` 카운트를 취하는 게 더 간단하지 않을까요?

**Answer:**

첫째로, 그건 실제로는 `countermax` 카운트를 예약하는 것입니다만 (라인 14를 참고하세요), 이 부분은 이것들의 절반만이 실제로 이 순간에 이 쓰레드에 의해 사용되도록 조정합니다. 이는 이 쓰레드가 `globalcount`를 다시 참조하기 전에 최소한 `countermax / 2` 만큼의 값 증가나 감소를 행할 수 있게 합니다.

`globalcount`에서의 계산은 라인 18에서의 조정 덕분에 정확하게 유지됨을 알아 두시기 바랍니다.

**Quick Quiz 5.37:**

Figure 5.6에서, 비록 남아 있는 카운트의 최대 한계까지의 4분의 1이 쓰레드 0에 할당되어 있다고는 해도, 가운데와 오른쪽 구성을 있는 위쪽의 점선이 보이듯 남아있는 카운트의 8분의 1만이 소모됩니다. 왜 그런거죠?

**Answer:**

이런 일이 일어나는 이유는 쓰레드 0의 `counter`가 그것의 `countermax`의 절반으로 설정되었기 때문입니다. 따라서, 쓰레드 0에 할당된 4분의 1 가운데, 절반은 (8분의 1) `globalcount`에서 오고, 나머지 절반 (다시 말하지만, 8분의 1)은 남아있는 카운트로부터 옵니다.

이 방법을 취하는 두가지 목적이 있습니다: (1) 쓰레드 0이 증가는 물론 감소에서도 fastpath를 이용할 수 있게 하는 것, 그리고 (2) 모든 쓰레드가 그 한계까지 단조롭게 값을 증가시키기만 할 때의 부정확도를 줄이는 것. 이 마지막 부분을 더 자세히 알아보려면, 이 알고리즘을 좀 더 자세히 들여다 보시기 바랍니다.

**Quick Quiz 5.38:**

왜 이 쓰레드의 `counter`와 `countermax` 변수들을 하나의 단위로 원자적 조정해야 하죠? 그것들 각자를 원자적으로 조정하는 것으로 충분하지 않을까요?

**Answer:**

이것도 어쩌면 가능할지도 모르지만, 상당한 주의가 필요합니다. 먼저 `countermax`를 0으로 만들지 않고 `counter`를 제거하는 것은 연관된 쓰레드가 `counter`를 0이 된 직후에 증가시켜서 이 카운터를 0으로 만드는 효과를 완전히 무효화 하는 결과를 초래할 수 있습니다.

반대의 순서, 즉 `countermax`를 0으로 만들고 나서 `counter`를 제거하는 것은 역시 `counter`가 0이 아니게 할 수 있습니다. 이걸 알아보기 위해, 다음 순서의 이벤트들을 생각해 봅시다:

1. 쓰레드 A가 자신의 `countermax`를 읽어오고, 그게 0이 아님을 확인합니다.
2. 쓰레드 B가 쓰레드 A의 `countermax`를 0으로 만듭니다.
3. 쓰레드 B가 쓰레드 A의 `counter`를 제거합니다.
4. 자신의 `countermax`가 0이 아님을 확인한 쓰레드 A는 자신의 `counter`에 값을 더하여서, 0이 아닌 값을 갖는 `counter` 변수를 초래합니다.

다시 말하지만, `counterandmax` 와 `counter` 를 별도의 변수들로 두고 원자적으로 조정하는 것도 가능할 수 있습니다만 상당한 주의가 필요함은 분명합니다. 또한 그렇게 하는 것이 fastpath 를 느리게 만들 것도 분명해 보입니다.

이 가능성들을 더 탐험해 보는 것은 독자 여러분의 몫으로 남겨둡니다.



Quick Quiz 5.39:

Listing 5.12 의 라인 7 는 어떻게 C 표준을 위반하나요?



Answer:

이것은 바이트당 여덟 비트가 사용된다고 가정합니다. 이 가정은 쉽게 공유 메모리 멀티프로세서로 조립될 수 있는 현재의 일반 상용 마이크로프로세서에 성립합니다만, C 코드를 수행해 본 적 있는 모든 컴퓨터 시스템에 대해 성립하지는 않습니다. (C 표준에 맞추기 위해선 대신 어떤 일을 할 수 있을까요? 그 단점은 무엇일까요?)



Quick Quiz 5.40:

단 하나의 `counterandmax` 변수만 있는데, 왜 Listing 5.12 의 line 18 에서는 포인터를 넘기는 거죠?



Answer:

쓰레드당 단 하나의 `counterandmax` 변수가 있습니다. 나중에 우리는 다른 쓰레드의 `counterandmax` 변수를 `split_counterandmax()` 로 넘기는 코드를 보게 될 겁니다.



Quick Quiz 5.41:

Listing 5.12 의 `merge_counterandmax()` 는 직접 `atomic_t` 에 저장을 하는 대신 `int` 를 리턴하나요?



Answer:

뒤에서 우린 `atomic_cmpxchg()` 기능에 넘기기 위한 `int` 를 반환받을 필요가 있음을 알게 될 겁니다.



Quick Quiz 5.42:

우웩! Listing 5.13 의 라인 11 의 추한 `goto` 는 웬말이죠?
`break` 문 모르세요???



Answer:

이 `goto` 를 `break` 로 교체하려면 라인 15 이 리턴해야 하는지를 알기 위한 플래그를 유지할 것을 필요로 할 텐데, 이건 fastpath 에서 하고 싶은 일이 아닐 겁니다. 여러분이 정말로 `goto` 를 그렇게나 싫어한다면, 여러분이 할 일은 이 fastpath 를 성공 또는 실패를 리턴해서 “실패”는 slowpath 의 필요를 의미하는 별개의 함수로 만드는 것일 겁니다. 이건 `goto` 를 싫어하는 독자 여러분의 연습문제로 남겨두겠습니다.



Quick Quiz 5.43:

Listing 5.13 의 라인 13-14 의 `atomic_cmpxchg()` 기능은 왜 실패할 수 있죠? 어쨌건, 우린 기존 값을 라인 9 에서 가져온 후 바꾸지 않았잖아요!



Answer:

나중에, 우린 Listing 5.15 의 `flush_local_count()` 함수가 어떻게 이 쓰레드의 `counterandmax` 변수를 Listing 5.13 의 라인 8-14 의 fastpath 수행과 동시에 업데이트 할 수도 있는지 볼 겁니다.



Quick Quiz 5.44:

쓰레드가 간단하게 `counterandmax` 변수를 Listing 5.15 의 라인 14 에서의 `flush_local_count()` 가 비운 후에 곧바로 다시 채우는 건 왜 안되나요?



Answer:

이 다른 쓰레드는 `flush_local_count()` 호출자가 `gblcnt_mutex` 를 해제하기 전에는 자신의 `counterandmax` 를 도로 채울 수 없습니다. 그 때에는, `flush_local_count()` 의 호출자는 이 카운트의 사용을 끝냈을 것이며, 따라서 이 다른 쓰레드가 값을 도로 채우는데 문제는 없을 겁니다—`globalcount` 의 값이 재충전을 허용할 만큼 충분히 크다는 가정 하에서요.



Quick Quiz 5.45:

Listing 5.15 의 line 27 에서 `flush_local_count()` 가 `counterandmax` 변수를 액세스 하는 동안 `add_count()` 와 `sub_count()` 의 동시에 수행되는 fastpath 들이 해당 변수를 간섭하는 것은 무엇이 막고 있습니까?



Answer:

그런 건 없습니다. 다음 세가지 경우를 고려해 보세요:

- 만약 `flush_local_count()` 의 `atomic_xchg()` 각 이 fastpath 들의 `split_counterandmax()` 전에 수행되었다면, 이 fastpath 는 0 값의 counter 와 `countermax` 를 보게 될 것이며, 따라서 slowpath 로 이동할 겁니다(물론 `delta` 가 0이 아니라면요).
- `flush_local_count()` 의 `atomic_xchg()` 가 각 fastpath 의 `split_counterandmax()` 후에, 그러나 이 fastpath 의 `atomic_cmpxchg()` 전에 수행되었다면, `atomic_cmpxchg()` 는 실패하여, fastpath 가 재시작하게 하고, 따라서 앞의 case 1 을 줄입니다.
- `flush_local_count()` 의 `atomic_xchg()` 가 각 fastpath 의 `atomic_cmpxchg()` 후에 수행된다면, 이 fastpath 는 (대부분의 경우) `flush_local_count()` 가 이 쓰레드의 `counterandmax` 변수를 0 으로 만들기 전에 성공적으로 완료될 겁니다.

어떤 경우든, 이 레이스는 올바르게 해결되었습니다.



Quick Quiz 5.46:

`atomic_set()` 기능은 명시된 `atomic_t` 에 간단한 값 할당을 할 뿐인데, Listing 5.16 의 `balance()` 의 라인 21 은 어떻게 이 변수를 업데이트 할 수 있죠?



Answer:

`balance_count()` 와 `flush_local_count()` 의 호출자 모두 `gblcnt_mutex` 를 쥐며, 따라서 한번에 하나만 수행될 수 있습니다.



Quick Quiz 5.47:

하지만 시그널 핸들러는 수행되는 동안 다른 CPU 로 옮겨질 수 있습니다. 이 가능성은 한 쓰레드와 이 쓰레드를 언터럽트 한 시그널 핸들러 사이에서의 안정적인 통신 을 위해 어토믹 인스트럭션과 메모리 배리어를 필요로 하게 하지 않나요?



Answer:

아뇨. 이 시그널 핸들러가 다른 CPU 로 옮겨진다면, 이 언터럽트 당한 쓰레드 역시 함께 이동합니다.



Quick Quiz 5.48:

Figure 5.7 에서, REQ `theft` 상태는 왜 빨간색으로 칠해졌나요?



Answer:

Fastpath 만이 `theft` 상태를 바꿀 수 있음을, 그리고 이 쓰레드가 이 상태에 너무 오래 머무른다면 이 slowpath 를 수행하고 있는 쓰레드는 POSIX 시그널을 다시 보낼 것임을 알리기 위해서입니다.



Quick Quiz 5.49:

Figure 5.7 에서, 분리된 REQ 와 ACK `theft` 상태를 갖는 것의 요지가 무엇인가요? 왜 이것들을 하나의 REQACK 상태로 만들어서 이 상태 머신을 더 간단하게 만들지 않죠? 그러면 시그널 핸들러든 fastpath 든 먼저 그 상태에 도달한 사람이 상태를 READY 로 만들면 될텐데요.



Answer:

REQ 와 ACK 상태를 하나로 만드는 것이 나쁜 생각인 이유는 여러가지가 있는데 다음의 것들이 포함됩니다:

- 이 slowpath 는 시그널이 어디서 다시 전송되어야 하는지 파악하는데 REQ 와 ACK 상태를 사용합니다. 만약 이 상태가 합쳐져 있다면, slowpath 는 시그널들을 무작정 계속해서 보내는 것밖에 선택의 여지가 없을텐데, 이는 불필요하게 fastpath 를 느리게 만드는 도움 안되는 효과만 낼 겁니다.
- 다음의 레이스가 초래될 겁니다:
 - 이 slowpath 가 특정 쓰레드의 상태를 REQACK 으로 설정합니다.
 - 해당 쓰레드가 fastpath 를 끝내고 REQACK 상태를 발견합니다.
 - 이 쓰레드가 시그널을 받고, 이 역시 REQACK 상태를 발견하고, 현재 진행중인 fastpath 가 없으므로, 이 상태를 READY 로 만듭니다.
 - 이 slowpath 는 이 READY 상태를 발견하고, 카운트를 가져오고, 상태를 IDLE 로 만들고, 완료됩니다.
 - 이 fastpath 는 상태를 READY 로 만들어서, 이 쓰레드의 다음 fastpath 수행을 막습니다.

여기서의 기본적인 문제는 합쳐진 REQACK 상태 가 시그널 핸들러에 의해서도 fastpath 에 의해서도 보여질 수 있다는 겁니다. 네개의 상태에 의해 관리되는 깔끔한 분리가 제대로 순서지어진 상태 전환을 보장합니다.

하지만, 여러분이 세개 상태의 셋업을 올바르게 만드는 것도 가능할 수도 있습니다. 만약 성공한다면, 그것을 네 개 상태 셋업과 주의 깊게 비교해 보시기 바랍니다. 그 세개의 상태 해법은 정말로 선호될 만 한가요, 그리고 그렇다면 왜, 안그렇다면 왜 그런가요?



Quick Quiz 5.50:

Listing 5.18 의 flush_local_count_sig() 함수에서, 쓰레드별 변수인 theft 의 사용은 왜 READ_ONCE() 와 WRITE_ONCE() 로 짜여있나요?



Answer:

첫번째 것 (라인 11)은 불필요하지 않은가 논쟁이 될 수 있습니다. 마지막의 두개 (라인 14 과 16)는 중요합니다. 이것들이 없어지면, 컴파일러는 라인 14-16 를 다음과 같이 바꿀 수 있습니다:

```
14 theft = THEFT_READY;
15 if (counting) {
16     theft = THEFT_ACK;
17 }
```

이는 치명적인 일이 될텐데, slowpath 는 THEFT_READY 의 변경 중인 값을 보게 될수도 있고, 따라서 연관된 쓰레드가 준비되기 전에 값 가져오기를 시작할 수 있기 때문입니다.



Quick Quiz 5.51:

Listing 5.18 에서, 왜 line 28 이 다른 쓰레드의 countermax 변수에 직접 접근하는게 안전하죠?



Answer:

그 다른 쓰레드는 gblcnt_mutex 락을 잡지 않고는 자신의 countermax 변수의 값을 바꾸는 게 허락되지 않았기 때문입니다. 하지만 그 호출자가 이 락을 쥐었고, 따라서 다른 쓰레드가 이 락을 잡는게 불가능하며, 그러니 그 다른 쓰레드는 countermax 변수를 바꿀 수 없습니다. 그러므로 우리는 안전하게 이걸 접근할 수 있습니다—하지만 바꿀 순 없습니다.



Quick Quiz 5.52:

Listing 5.18 에서, line 33 은 왜 현재 쓰레드가 자신에게 시그널을 보내는지 체크하지 않는 거죠?



Answer:

추가적인 검사가 필요치 않습니다. flush_local_count() 는 이미 globalize_count() 를 호출했으며,

따라서 라인 28 의 체크는 성공하고 뒤따르는 pthread_kill() 을 건너뛰었을 겁니다.



Quick Quiz 5.53:

Listings 5.17 and 5.18 에 보인 코드는 GCC 와 POSIX 에서 동작합니다. 이게 ISO C 표준도 따르게 하기 위해선 뭐가 필요할까요?



Answer:

theft 변수는 시그널 핸들러와 그 시그널에 의해 인터럽트된 코드 사이에서 안전하게 공유됨을 보장하기 위해 sig_atomic_t 타입이어야만 합니다.



Quick Quiz 5.54:

Listing 5.18 에서, 라인 41 은 왜 시그널을 다시 보낼까요?



Answer:

많은 운영체제가 수십년에 걸쳐 간혹 시그널을 잃어버리는 성격을 가져버렸기 때문입니다. 이게 가능이냐 버그냐에 대해서는 논쟁이 가능하겠습니까만 관계 없습니다. 사용자의 시점에서 분명한 증상은 커널 버그가 아니라 사용자 어플리케이션이 멈춰있는 것일 겁니다.

여러분의 사용자 어플리케이션이 멈춰있다구요!



Quick Quiz 5.55:

POSIX 시그널은 느리기만 한 게 아니고, 시그널을 각 쓰레드에 보내는 것은 확장이 안됩니다. 만약 여러분이 (예를 들어) 10,000 쓰레드를 가지고 있고 이 읽기 쪽이 빠르게 만들어야 한다면 어떻게 하겠습니까?



Answer:

한가지 방법은 Section 5.2.4 에 보인 기법으로, 전체 카운터 값에 대한 근사치를 단일 변수에 요약하는 겁니다. 또 다른 방법은 여러 쓰레드가 읽기를 수행하게끔 해서, 각 쓰레드가 업데이트를 하는 쓰레드 중 일부 집합들과만 상호작용하게 하는 겁니다.



Quick Quiz 5.56:

만약 여러분이 원하는 건 이 정확한 리미트 카운터가 아래쪽 한계에는 정확하지만 위쪽 한계에는 정확하지 않아도 되는 것이라면 어떻게 하시겠습니까?

Answer:

한가지 간단한 해결책은 위쪽 한계를 원하는 만큼 실제보다 크게 잡는 겁니다. 그런 부풀리기의 제약은 이 위쪽 한계가 이 카운터가 표현할 수 있는 가장 큰 값이 되는 경우입니다.

**Quick Quiz 5.57:**

편향된 카운터를 사용할 때 상황을 더 좋게 하기 위해 여러분이 해보았을 법한 더 나은 방법은 무엇이 있을까요?

**Answer:**

이 위쪽 한계를 이 편향치, 예상되는 최대 액세스 횟수, 그리고 “엎질러짐”을 충분히 받아들일 수 있을 만큼 크게 설정해서 액세스 횟수가 그 최대치일 때에도 일이 효율적으로 진행되게 하는 것이 좋을 겁니다.

**Quick Quiz 5.58:**

웃기네요! 이 카운터를 업데이트 하기 위해 reader-writer 락을 읽기 모드로 획득한다구요? 뭐하는 짓이예요???

**Answer:**

아마도, 이상하겠죠, 하지만 정말입니다! 이건 “Reader-writer lock” 이란 이름은 생각 없이 붙여진 것이라 여러분이 생각하게 하기 충분할 거예요, 그렇지 않나요?

**Quick Quiz 5.59:**

실제 시스템에서는 어떤 다른 문제들이 해결되어야 할까요?

**Answer:**

무척 많은 것들이 있습니다!

여기 몇가지 생각을 시작해 볼만한 것들이 있습니다:

1. 기기가 얼마든지 있을 수 있으며, 따라서 전역 변수들은 적절치 않을 것이고, `do_io()` 같은 함수로의 인자의 부재도 그렇습니다.
2. 루프에서의 폴링은 실제 시스템에서는 CPU 시간과 전력을 낭비하므로 문제가 될 수 있습니다.
3. I/O는 실패할 수도 있으며, 따라서 `do_io()`는 리턴 값을 가져야 할 겁니다.

4. 만약 이 기기가 고장나면, 마지막 I/O는 결코 완료되지 않을 겁니다. 그런 경우라면, 에러로부터의 회복을 위해 시간 제한 같은 게 필요할 수도 있습니다.

5. `add_count()`과 `sub_count()`은 실패할 수 있습니다만, 이것들의 리턴값이 체크되지 않고 있습니다.

6. Reader-writer 락은 잘 확장되지 못합니다. Reader-writer 락의 이 높은 읽기 모드 획득 쪽 비용을 제거하기 위한 한가지 방법이 Chapters 7 and 9에 소개됩니다.

**Quick Quiz 5.60:**

Table 5.1의 `count_stat.c` 열에서, 우린 읽기쪽의 쓰레드 수에 따른 선형적 확장성을 볼 수 있습니다. 더 많은 쓰레드가 존재할수록 더 많은 쓰레드별 카운터가 합산되어야 하는데 이게 어떻게 가능하죠?

**Answer:**

읽기쪽 코드는 쓰레드 수에 관계 없이 고정된 크기의 배열 전체를 스캔해야 하며, 따라서 성능에 차이가 없습니다. 대조적으로, 마지막 두개의 알고리즘에서, 읽기 쓰레드는 더 많은 쓰레드가 있을 때 더 많은 일을 해야 합니다. 또한, 마지막 두개의 알고리즘은 쓰레드 ID라는 정수로부터 연관된 `_thread` 변수로의 매핑을 가지므로 추가적인 간접 단계를 갖습니다.

**Quick Quiz 5.61:**

Table 5.1의 네번째 열에서조차도, 이 통계적 카운터 구현의 읽기쪽 성능은 상당히 무섭네요. 그런데도 왜 이걸 신경쓰죠?

**Answer:**

“주어진 일에 적합한 도구를 사용하라.”

Figure 5.1에서 볼 수 있듯이, 병렬 업데이트의 많은 사용이 필요한 일에 단일 변수 어토믹 값 증가는 사용될 필요 없습니다. 반면, Table 5.1의 위쪽 절반에 보여진 알고리즘들은 업데이트가 많은 상황의 일을 훌륭하게 해냅니다. 물론, 읽기가 대부분인 상황이라면, 여러분은 다른 무언가, 예를 들면 Section 5.2.4에서 사용된 방법과 비슷한, 단일 로드를 이용해 읽어질 수 있는 단일 어토믹하게 값 증가되는 변수를 사용한 결과적 일관성 설계를 사용해야 합니다.



Quick Quiz 5.62:

Table 5.1 의 아래쪽 절반에 보여진 성능 데이터를 놓고 보면, 우린 항상 어토믹 오퍼레이션보다 시그널을 선호해야 하겠군요, 그렇죠?

**Answer:**

그건 워크로드에 달려 있습니다. 64 코어 시스템에서라면, 단 하나의 시그널을 (거의 5 마이크로세컨드 성능 저하를 일으키는) 위해서도 100개가 넘는 어토믹이 아닌 오퍼레이션이 필요함을 (대략 40 나노세컨드 성능 이득을 위해) 알아 두시기 바랍니다. 훨씬 더 읽기 집약도가 높은 워크로드가 많이 있긴 하지만, 여러분은 여러분의 워크로드를 고려해야 할 겁니다.

또한, 역사적으로 메모리 배리어는 평범한 인스트럭션들보다 비쌌지만, 여러분이 사용할 실제 하드웨어 위에서 이걸 체크해 보셔야 합니다. 컴퓨터 하드웨어의 특성은 시간에 따라 변하며, 알고리즘은 그에 따라 변해야만 합니다.

**Quick Quiz 5.63:**

Table 5.1 의 아래쪽 절반에 보여진 읽기 쓰레드의 락 경쟁을 해결하기 위해 진보된 기법들이 사용될 수 있을까요?

**Answer:**

한가지 방법은 scalable non-zero indicators (SNZI) [ELLM07] 에서처럼 업데이트 쪽 성능을 포기하는 것입니다. 이에 대해선 해볼만한 여러 방법들이 있으며, 이것들은 독자 여러분의 연습문제로 남겨 두겠습니다. 빈번한 전역 락 획득을 연관된 아래 계층의 지역 락 획득으로 대체하는 모든 방법들 역시 상당히 잘 동작할 겁니다.

**Quick Quiz 5.64:**

++ 오퍼레이터는 1,000 자리 숫자에도 잘 동작해요! 오퍼레이터 오버로딩이라고 뭘 들어 보셨나요???

**Answer:**

C++ 언어에서는, 1,000 자리 숫자를 위해서도 여러분이 그 숫자를 구현하는 클래스로의 접근 권한이 있다는 규정 하에 ++ 를 사용할 수도 있을 겁니다. 하지만 2021년 기준으로, C 언어는 오퍼레이터 오버로딩을 허용하지 않습니다.

**Quick Quiz 5.65:**

하지만 우리가 모든 것을 분할해야 한다면, 왜 공유 메모리 멀티쓰레딩을 고려하죠? 문제를 완전하게 분할하고 각각 각자의 주소공간을 가지는 여러 프로세스로 돌리는 게 어떤가요?

**Answer:**

실제로, 별도의 주소 공간을 갖는 여러 프로세스는 병렬성을 노출시키는 훌륭한 방법이 될 수 있습니다, fork-join 방법론과 Erlang 언어의 제안자도 곧바로 여러분에게 그렇게 말할 겁니다. 하지만, 공유메모리 병렬성에는 몇 가지 장점도 있습니다:

1. 어플리케이션의 가장 성능에 중요한 부분만이 분할되어야 하며, 그런 부분은 보통 어플리케이션의 작은 부분입니다.
2. 캐쉬 미스는 개별 레지스터간 인스트럭션에 비교하면 상당히 느리지만, TCP/IP 네트워킹 같은 것 보다는 무척 빠른 프로세스간 통신 기능들보다도 일반적으로 상당히 빠릅니다.
3. 공유메모리 멀티프로세서는 이미 사용 가능하고 무척 저렴합니다, 따라서 1990년대와는 다르게, 공유메모리 병렬성의 사용에 대한 비용 페널티가 무척 작습니다.

항상 그렇듯, 해당 일에 적합한 도구를 사용하세요!



E.6 Partitioning and Synchronization Design

Quick Quiz 6.1:

이 Dining Philosophers 문제에 더 나은 해결책이 있을까요?

**Answer:**

그런 향상된 해결책 하나가 Figure E.2 에 보여져 있는데, 단순히 철학자들에게 추가의 다섯개 포크를 제공하는 것입니다. 모든 다섯명의 철학자가 이제 동시에 식사를 할 수 있고, 어떤 철학자도 다른 누군가를 기다릴 필요가 없습니다. 또한, 이 방법은 무척 향상된 재앙 통제를 제공합니다.

이 해결책은 누군가에겐 속임수처럼 보일 수도 있겠습니다만, 이런 종류의 “속임수”가 많은 동시성 문제에 있어 좋은 해결책을 찾기 위한 열쇠입니다.



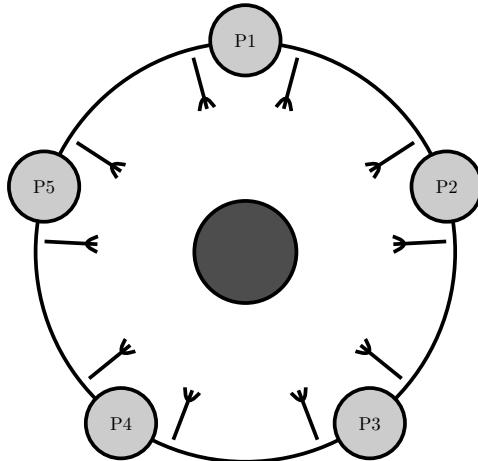


Figure E.2: Dining Philosophers Problem, Fully Partitioned

Quick Quiz 6.2:

그리고 어떤 의미에서 이 “수평적 병렬성”은 “수평적”이라고 불릴 수 있는 건가요?

■

Answer:

Inman은 일반적으로 어플리케이션이 꼭대기에, 그리고 하드웨어 연결부가 바닥에, 수직으로 그려지는 프로토콜 스택을 가지고 일하고 있었습니다. 데이터는 이 스택의 위에서 아래로 흐릅니다. “수평적 병렬성”은 패킷을 다른 네트워크 연결부로부터 병렬로 처리하는 반면, “수직 병렬성”은 주어진 패킷을 다른 프로토콜 처리 단계로 동시에 처리합니다.

“수직 병렬성”은 또한 “파이프라이닝”이라고도 불립니다.

□

Quick Quiz 6.3:

이 compound double-ended queue 구현에서는, 이 락을 해제하고 재획득하는 동안에 이 queue 가 비어있지 않게 되면 어떡해야 하죠?

■

Answer:

이 경우에는, 그냥 이 비어있지 않은 큐에서 원소를 제거하고, 두 락을 해제하고, 리턴하면 됩니다.

□

Quick Quiz 6.4:

해쉬 기반의 double-ended queue는 좋은 해결책일까요? 왜 그렇고 왜 그렇지 않을까요?

■

Answer:

여기 답변하는 최선의 방법은 여러개의 다른 멀티프로세서 시스템에서 `lockhdeq.c`를 수행해 보는 것이며, 여러분이 가능한 시간 내에 그렇게 할 것을 장려합니다. 걱정을 할 수 있는 한가지 이유는 이 구현에서 각 오퍼레이션은 하나가 아닌 두개의 락을 획득해야 한다는 것입니다.

잘 설계된 첫번째 성능 연구가 예로 들어질 겁니다.⁶ 순차적 구현과의 비교도 잊지 마십시오!

□

Quick Quiz 6.5:

빈 queue로 모든 원소를 옮긴다구요? 이 미친 해결책이 대체 어떤 세상에서는 최적인가죠???

■

Answer:

데이터 흐름의 방향이 가끔씩만 바뀌는 경우에 최적입니다. 이 double-ended queue가 동시에 양 끝에서 비워지는 경우엔 물론 무척 안좋은 선택일 겁니다. 이는 물론 다른 질문을 떠오르게 하는데, 양 끝에서 동시에 비우기를 하는건 대체 어떤 세상에서 합리적인 일이겠냐는 겁니다. 이 질문에 대한 가능한 답 중 하나는 work stealing queue입니다.

□

Quick Quiz 6.6:

조합된 병렬 double-ended queue 구현은 왜 대칭적일 수 없죠?

■

Answer:

락 계층을 사용해 데드락을 회피해야한다는 필요성이 비대칭성을 강제하는데, 식사하는 철학자들 문제의 포크 번호 매기기 해결책과 같은 것입니다 (Section 6.1.1을 참고하세요).

□

Quick Quiz 6.7:

Listing 6.3의 라인 28에서의 오른쪽 `dequeue` 재시도가 왜 필요하죠?

■

Answer:

이 쓰레드가 라인 25에서 `d->rlock`을 내려놓고 라인 27에서 이 락을 다시 획득하는 사이에 다른 쓰레드가 원소를 하나 `enqueue` 했을 수도 있기 때문에 이 재시도가 필요합니다.

□

⁶ Dalessandro 등의 연구 [DCW⁺11] 와 Dice 등의 연구 [DLM⁺10]는 훌륭한 시작 지점이 될 겁니다.

Quick Quiz 6.8:

분명 왼쪽 락은 가끔은 획득 가능할 거예요!!! 그런데도 왜 Listing 6.3 의 라인 25 에서는 무조건적으로 오른쪽 락을 해제해야 하는 거죠?

**Answer:**

왼쪽 락을 획득 가능할 때 획득하기 위해서 `spin_trylock()` 을 사용하는 것도 가능할 겁니다. 하지만, 그 실패의 경우에는 여전히 이 오른쪽 락을 내려놓고 두 락을 제대로 된 순서로 다시 획득해야 할 겁니다. 이 변화를 만드는 것은 (그리고 이게 그럴 가치가 있는지 알아내는 것은) 독자 여러분의 연습문제로 남겨두겠습니다.

**Quick Quiz 6.9:**

하지만 데이터가 한 방향으로만 흐르는 경우에, Listing 6.3 에 보인 알고리즘은 마지막 원소를 가져가서 아랫단의 double-ended queue 를 비울 때마다 양 끝단이 같은 락을 획득하려 할 겁니다. 이는 이 알고리즘이 양 끝단으로 동시의 액세스를 제공하는 것이 이 queue 가 상당히 큰 수의 원소를 가지고 있을 때조차 실패할 수 있음을 의미하지 않나요?

**Answer:**

실제로 그렇습니다!

하지만 같은 속성을 가지고 있다고 하는 다른 알고리즘들도 마찬가지입니다. 예를 들어, 락들의 해쉬 배열을 사용하는 소프트웨어 트랜잭션 메모리 메커니즘을 사용하는 해결책들에서는 가장 왼쪽과 가장 오른쪽 원소들의 주소가 가끔은 같은 락에 해제되는 경우가 있을 겁니다. 이런 해쉬 충돌 역시 동시의 액세스를 방지할 겁니다. 다른 예를 하나 들어보자면, 소프트웨어 `fallback` 을 갖추고 하드웨어 트랜잭션 메모리 메커니즘을 사용하는 해결책에서는 [YHLR13, Mer11, JSG12] 이 소프트웨어 `fallback` 내에서 락킹을 종종 사용할 것이고, 따라서 무엇이 되었든 이 락킹 해결책이 겪는 한계만큼 한계를 겪을 겁니다(비록 뜯어하긴 하길 바랍니다만).

따라서 2021년 기준으로, compound double-ended queue 를 포함한 동시의 double-ended queue 문제를 위한 모든 실용적 해결책은 적어도 어떤 환경에서는 완전한 동시성을 제공하지 못합니다.

**Quick Quiz 6.10:**

Double-ended queue 문제에는 왜 한개가 아니라 두개의 해결책이 있는 거죠?

**Answer:**

사실은 최소 세개의 해결책이 있습니다. Dominik Dingel 의 것인 그 세번째 해결책은 reader-writer 락킹을 흥미로운 방식으로 사용하며, `lockrwdeq.c` 에서 찾을 수 있을 겁니다.

**Quick Quiz 6.11:**

연결 기반 double-ended queue 는 해쉬 기반 double-ended queue 보다 두배나 빠르게 동작합니다. 제가 이 해쉬 테이블의 크기를 미친듯이 크게 늘려도 말이죠. 왜 그런거죠?

**Answer:**

이 해쉬 기반의 double-ended queue 의 락킹 설계는 한번에 한 쓰래드만이 각 끝에 있을 수 있게 하며, 따라서 각 오퍼레이션을 위해 두개의 락 획득을 필요로 합니다. 연결 기반 double-ended queue 또한 한번에 각 끝에 하나의 쓰래드만 있을 수 있게 하고, 대부분의 경우 오퍼레이션 당 하나의 락 획득만을 필요로 합니다. 따라서, 이 연결 기반 double-ended queue 는 해쉬 기반 double-ended queue 보다 빠르게 동작할 것으로 기대됩니다.

각 끝에서 한번에 여러 동시의 오퍼레이션을 가능하게 하는 double-ended queue 를 만들 수 있겠습니까? 만약 그렇다면, 또는 그렇지 않다면, 왜입니까?

**Quick Quiz 6.12:**

Double-ended queue 를 위해 동시성을 제어하는 훨씬 나은 방법이 있을까요?

**Answer:**

한가지 가능한 방법은 여러 double-ended queue 가 병렬로 사용될 수 있게 해서 더 간단한 단일 락 기반 double-ended queue 가 사용될 수 있도록, 그리고 어쩌면 각 double-ended queue 또한 한쪽의 평범한 단일 끝단 queue 로 바꿀 수 있게끔, 문제 자체를 해결 가능하게 바꾸는 것입니다. 그런 “수평 확장” 없이는 속도 향상은 2.0 으로 제한됩니다. 반면에, 수평-확장 설계는 매우 큰 속도 향상을 얻을 수 있으며, 이 queue 의 양 끝에서 동작하는 여러 쓰래드가 있을 때 특히 매력적인데, 이 멀티 쓰래드 경우에는 `dequeue` 가 강한 순서 보장을 할 수 없게 되기 때문입니다. 어쨌건, 어떤 쓰래드가 어떤 아이템을 첫번째로 제거했다는 사실은 그게 그 쓰래드가 그 아이템을 첫번째로 처리할 것이라는 의미를 갖지 않습니다 [HKLP12]. 그리고 보장이 없다면, 이 보장을 제공하는 걸 거부하는데서 오는 성능 이득을 얻을 수도 있을 겁니다.

이 문제가 여러 queue 를 사용할 수 있게끔 변환이 가능한가에 관계없이, 각 enqueue 와 dequeue 오퍼레이션이 더 큰 단위의 일에 연관될 수 있게끔 일을 뭉쳐서 (batching) 할 수 있는지도 물어볼 가치가 있습니다. 이 batching 방법은 이 queue 자료 구조로의 경쟁을 줄 이는데, 이는 Section 6.3 에서 보게 되겠지만 성능도 확장성도 증가시킵니다. 어쨌건, 여러분이 높은 동기화 오버헤드를 일으켜야만 한다면, 여러분은 여러분의 돈의 가치를 얻게 됨을 분명히 해두십시오.

다른 연구자들은 queue 에서의 제한된 순서 보장으로부터 장점을 취하는 다른 방법들에 대해 연구하고 있습니다 [KLP12].

□

Quick Quiz 6.13:

이 모든 크리티컬 섹션에 대한 문제들은 우리가 단순히 크리티컬 섹션을 갖지 않는 non-blocking 동기화 [Her90] 를 항상 사용해야 함을 의미하지 않나요?

■

Answer:

Non-blocking 동기화는 일부 상황에서는 매우 유용할 수 있긴 합니다만, Section 14.2 에서 이야기 되었듯 만병통치약은 아닙니다. 또한, non-blocking 동기화는 Josh Triplett 에 의해 이야기 되었듯 실제로는 크리티컬 섹션을 갖습니다. 예를 들어, compare-and-swap 오퍼레이션에 기반한 non-blocking 알고리즘에서 최초의 로드로부터 시작해서 compare-and-swap 으로 이어가는 코드는 락 기반의 크리티컬 섹션과 비슷합니다.

□

Quick Quiz 6.14:

어떤 구조체를 그것의 락이 획득되어 있는 동안 해제되지 않게 하는 방법들로는 어떤 게 있나요?

■

Answer:

여기 존재 보장 문제에 대한 몇가지 해결책이 있습니다:

1. 구조체별 락이 획득되어 있는 동안 잡히는 정적으로 할당된 락을 제공하는 것으로, 이는 계층적 락킹의 한 예입니다 (Section 6.4.2 을 참고하세요). 물론, 이 목적으로 하나의 전역 락을 사용하는 것은 받아들이기 어려울 정도로 높은 수준의 락 경쟁을 일으켜서 성능과 확장성을 극적으로 떨어뜨릴 겁니다.
2. Chapter 7 에서 보인 것처럼 정적으로 할당된 락들의 배열을 제공하고, 이 구조체의 주소를 획득할 락을 선택하기 위해 해싱하는 것. 충분히 높은 품질의 해시 함수가 있다면, 이는 단일 전역 락의 확장성 제한을 막아줍니다만, 읽기가 대부분인 상황에서는

이 락 획득 오버헤드가 받아들이기 어려울 정도의 성능 하락을 초래할 수 있습니다.

3. 가비지 콜렉터를 제공하는 소프트웨어 환경이라면 그것을 사용해서 구조체가 참조되는 동안은 할당해제되지 못하게 하는 방법이 있습니다. 이는 무척 잘 동작하고, 개발자들의 어깨로부터 존재 보장 부담을 (그 외에도 많은 것을) 제거합니다만, 프로그램에 가비지 콜렉션 오버헤드를 부과합니다. 가비지 콜렉션 기술은 지난 수십년간 상당히 발전했지만, 그 오버헤드는 어떤 어플리케이션들에게는 받아들이기 어려울 정도로 높을 수 있습니다. 또한, 일부 어플리케이션은 개발자들이 대부분의 가비지 콜렉션 기반 환경들이 허용하는 것에 비해 데이터 구조의 레이아웃과 위치에 더 많은 제어를 필요로 하기도 합니다.
4. 가비지 콜렉터의 한 특수한 경우로, 전역 레퍼런스 카운터를, 또는 레퍼런스 카운터의 전역적 배열을 사용하는 것. 이는 앞의 락을 사용하는 방법에서 이야기 된 것과 유사한 강점과 한계점을 갖습니다.
5. 내부에서 바깥으로의 레퍼런스 카운트라고 생각될 수 있는, 해저드 포인터를 사용하는 것. 해저드 포인터 기반 알고리즘은 쓰레드별 포인터 리스트를 유지하여서, 이 리스트에 특정 포인터가 보이는 것은 해당 구조체로의 레퍼런스처럼 동작합니다. 해저드 포인터는 프로덕션에서의 상당한 사용을 보이기 시작하고 있습니다 (Section 9.6.3.1 를 참고하세요).
6. 트랜잭션 메모리 (Transactional Memory: TM) [HM93, Lom77, ST95] 를 사용해서 이 데이터 구조로의 모든 참조와 수정이 어토믹하게 수행되도록 하는 것. TM 이 최근 몇년간 많은 흥분을 일으켰고 프로덕션 소프트웨어에서 일부 사용될 가능성이 보이긴 합니다만, 개발자들은 일부 주의를 해야하는데 [BLM05, BLM06, MMW07], 특히 성능에 중요한 코드에서 그렇습니다. 특히, 존재 보장은 이 트랜잭션인 전역 레퍼런스로부터 업데이트 되는 데이터 항목으로의 전체 과정을 커버할 것을 필요로 합니다. 그것을 다른 동기화 메커니즘들과 결합해 그 취약성을 극복하는 방법들을 포함한 TM 에 대한 더 많은 정보를 위해선 Sections 17.2 and 17.3 를 참고하세요.
7. 극단적으로 가벼운 가비지 콜렉터 같은 것이라 생각될 수 있는 RCU 를 사용하는 것. 업데이트 쓰레드는 RCU 읽기 쓰레드가 아직 참조하고 있을 수 있는 RCU 로 보호되는 데이터 구조체를 할당해제할 수 없게 되어 있습니다. RCU 는 읽기가

대부분인 데이터 구조체에서 무척 많이 사용되고 있으며, Section 9.5에서 길게 이야기 됩니다.

존재 보장의 제공에 대한 더 많은 내용을 위해선, Chapters 7 와 9 를 읽어보시기 바랍니다.



Quick Quiz 6.15:

어떻게 싱글쓰레드 기반 64-by-64 행렬 곱셈이 1.0 보다 낮은 효율성을 가질 수 있죠? Figure 6.17 의 모든 선들은 하나의 쓰레드만 사용했을 때에는 1.0의 효율성을 보여야 하는 거 아닌가요?



Answer:

matmul.c 프로그램은 지정된 수의 작업 쓰레드를 만드는데, 하나의 작업 쓰레드의 경우에 조차도 쓰레드 생성 오버헤드를 일으킵니다. 단일 작업 쓰레드의 경우를 위한 쓰레드 생성 오버헤드를 제거하는 데 필요한 변경 작업은 독자 여러분의 연습문제로 남겨둡니다.



Quick Quiz 6.16:

데이터 병렬 기법이 어떻게 행렬 곱셈을 도울 수 있죠?
이건 이미 데이터 병렬이라구요!!!



Answer:

주의를 기울여주신 것에 감사합니다! 이 예제는 데이터 병렬성이 매우 좋은 것일 수 있긴 하지만, 그것이 모든 비효율성의 원인을 자동으로 사라지게 하는 어떤 마법 지팡이는 아니라는 것을 보이기 위함입니다. 완전한 성능으로의 선형적 확장은 “오직” 64 쓰레드에 대해서라도 설계와 구현의 모든 단계에 주의를 필요로 합니다.

특히, 여러분은 각 조각의 크기에 주의를 기울여야 합니다. 예를 들어, 여러분이 64-by-64 행렬 곱셈을 64 쓰레드로 쪼갠다면, 각 쓰레드는 오직 64개의 부동소수 점 곱셈만을 합니다. 부동소수점 곱셈의 비용은 쓰레드 생성의 오버헤드에 비해 아주 작으며, 캐시 미스 오버헤드 또한 이론적으로 완벽한 확장성을 망치는 (그리고 이 선을 그렇게나 흔들리게 하는) 역할을 합니다. 448 하드웨어 쓰레드의 완전한 사용의 경우에는 좋은 확장성을 얻기 위해 수십만개의 열과 행을 갖는 행렬을 필요로 할텐데, 그 지점부터는 GPGPU 가 상당히 매력적인 선택이 되는데, 특히 가격/성능의 관점에서 그렇습니다.

여러분이 다양한 입력을 받는 병렬 프로그램을 가지고 있다면, 입력의 크기가 병렬화를 하기에 가치가 있으면 너무 작지 않은지에 대한 체크를 항상 포함시키십시오. 그리고 병렬화가 도움이 되지 않을 때에는, 쓰레드를 만들기 위한 오버헤드를 일으키는 건 도움이 되지 않습니다.



Quick Quiz 6.17:

어떤 경우에 계층적 락킹이 잘 동작하나요?



Answer:

Listing 6.8 의 line 31에서의 비교가 훨씬 무거운 오퍼레이션으로 대체되었다면, `bp->bucket_lock` 의 해제는 어쩌면 `cur->node_lock` 의 추가적인 획득과 해제의 오버헤드를 넘어설 정도로 락 컨텐션을 줄였을 수도 있습니다.



Quick Quiz 6.18:

이 리소스 할당자 설계는 Section 5.3에서 이야기한 대략적 리미트 카운터의 그것과 닮지 않았나요?



Answer:

실제로 그렇습니다! 우린 메모리를 할당하고 해제하는 것을 생각하고 있었습니다만, Section 5.3의 알고리즘은 “카운트”를 할당하고 해제하는 매우 비슷한 일을 하고 있었습니다.



Quick Quiz 6.19:

Figure 6.21에서, 수행 길이를 증가시킴에 따라 성능이 올라감에는 세개의 샘플 그룹에 의하는 패턴이 있는데 예를 들면 수행 길이 10, 11, 12가 그렇습니다. 왜죠?



Answer:

이는 CPU 별 타겟 값이 셋이기 때문입니다. 수행 길이 12의 경우는 글로벌 풀 락을 두번 획득해야만 하는 반면, 수행 길이 13의 경우는 이 글로벌 풀 락을 세번 획득해야만 합니다.



Quick Quiz 6.20:

할당 실패는 두개 쓰레드 테스트에서 수행 길이가 19 이상일 경우에 관측되었습니다. 글로벌 풀의 크기가 40이고 쓰레드별 타겟 풀 크기 s 가 셋이고, 쓰레드 수 n 이 2이며, 이 쓰레드별 풀이 초기에는 어떤 메모리도 사용되지 않고 있으므로 비어있음을 놓고 생각해 보면, 실패가 일어날 수 있는 최소한의 할당 수행 길이 m 은 무엇일까요?(각 쓰레드는 반복적으로 m 블록의 메모리를 할당하고, 이어서 m 블록의 메모리를 해제함을 기억하시기 바랍니다.) 달리 생각하면, n 쓰레드가 풀 크기 s 를 가지고 있으며 각 쓰레드가 반복적으로 m 블록의 메모리를 할당받고 이어서 m 블록의 메모리를 해제함을 고려할 때, 글로벌 풀의 크기는 얼마나 커야 할까요?
Note: 정확한 답을 얻기 위해선 여러분이 `smpalloc.c` 소스 코드를 들여다 보고 그걸 한 단계씩 수행해 보기도 할 것을 필요로 할 겁니다. 경고 했어요!

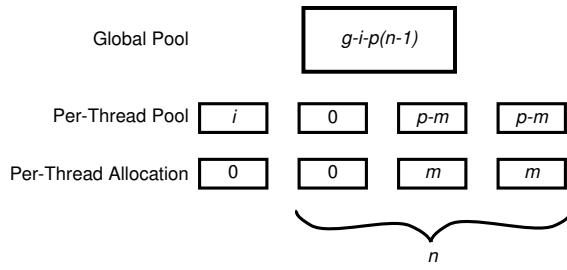


Figure E.3: Allocator Cache Run-Length Analysis

■

Answer:

이 해결책은 Alexey Roystman 이 제공한 것에서 채택되었습니다. 이는 다음의 정의들에 기반합니다:

g 전역적으로 사용 가능한 블록들의 갯수.

i 초기화 쓰레드의 쓰레드별 풀에 남아있는 블록의 갯수. (이게 여러분이 코드를 봐야 하는 이유 중 하나입니다!)

m 할당/해제 수행 길이.

n 초기화 쓰레드를 제외한 쓰레드들의 수.

p 실제로 할당된 블록들과 쓰레드별 풀에 남아있는 블록들을 포함해 쓰레드별 최대 블록 소비.

g, m , 그리고 n 의 값은 주어졌습니다. p 의 값은 m 을 다음 s 의 배까지 올립되는데, 다음과 같습니다:

$$p = s \left\lceil \frac{m}{s} \right\rceil \quad (E.6)$$

i 의 값은 다음과 같습니다:

$$i = \begin{cases} g & (\text{mod } 2s) = 0 : 2s \\ g & (\text{mod } 2s) \neq 0 : g \quad (\text{mod } 2s) \end{cases} \quad (E.7)$$

이 크기들 간의 관계가 Figure E.3에 보여져 있습니다. 글로벌 풀은 이 그림의 꼭대기에 보여져 있으며, “나머지” 초기화 쓰레드의 쓰레드별 풀과 쓰레드별 할당은 왼쪽 한쌍의 박스들로 그려져 있습니다. 이 초기화 쓰레드는 할당한 블록이 없습니다만, 자신의 쓰레드별 풀에 남겨진 i 블록들을 가지고 있습니다. 오른쪽 두쌍의 상자는 최대로 가능한 수의 블록을 잡아두고 있는 쓰레드들의 쓰레드별 풀들과 쓰레드별 할당들인데, 왼쪽으로부터 두번째 한쌍의 상자들은 현재 할당을 시도하고 있는 쓰레드를 나타냅니다.

전체 블록들의 갯수는 g 이며, 쓰레드별 할당과 쓰레드별 풀을 더하면, 우리는 이 글로벌 풀이 $g - i - p(n - 1)$ 블록을 포함하고 있음을 알 수 있습니다. 이 할당 쓰레드가 성공하게 되려면, 글로벌 풀에 최소 m 블록들이 있어야 하는데, 달리 말하면:

$$g - i - p(n - 1) \geq m \quad (E.8)$$

이 질문에서 $g = 40$, $s = 3$, 그리고 $n = 2$ 입니다. 식 E.7은 $i = 4$ 를 제공하고, and 식 E.6은 $m = 18$ 을 위해 $p = 18$ 이라는 결과를, $m = 19$ 를 위해 $p = 21$ 이라는 결과를 제공합니다. 이를 식 E.8에 끼워넣으면, $m = 18$ 은 넘쳐나지만, $m = 19$ 는 잘 될 수 있음을 보입니다.

i 의 존재는 버그로 여겨질 수도 있습니다. 어쨌건, 초기화 쓰레드의 캐쉬에 남아있기만을 위해서 왜 메모리를 할당하겠습니까? 이를 고치는 한가지 방법은 현재 쓰레드의 풀의 것을 글로벌 풀로 비워버리는 `memblock_flush()` 함수를 제공하는 것일 겁니다. 그러면 이 초기화 쓰레드는 모든 블록을 해제한 후에 이 함수를 호출할 수 있을 겁니다.

□

E.7 Locking

■

Quick Quiz 7.1:

희생양 역할을 하는게 어떻게 영광스러운 것일 수 있죠???

■

Answer:

락킹이 연구 논문의 희생양이 된 이유는 그것이 실제 세계에서 굉장히 많이 사용되었기 때문입니다. 반면에, 누구도 락킹을 사용하지 않거나 신경쓰지 않았다면, 대부분의 연구 논문은 그걸 언급조차 하지 않았을 겁니다.

□

Quick Quiz 7.2:

하지만 락 기반의 deadlock의 정의는 각 쓰레드가 최소 하나의 락을쥔 채 어떤 다른 쓰레드가 쥐고 있는 락을 기다린다였습니다. 사이클이 존재하는지 어떻게 알 수 있죠?

■

Answer:

이 그래프에 사이클이 존재하지 않는다고 생각해 봅시다. 그러면 우린 방향성을 directed acyclic graph (DAG)를 갖게 될텐데, 여기엔 최소 하나의 leaf 노드가 존재할 겁니다.

만약 이 leaf 노드가 락이라면, 우린 어떤 쓰레드에 대해서도 쥐여지지 않은 어떤 락을 기다리는 쓰레드가

존재한다는 것인데, 이는 앞의 정의에 반하는 일입니다. 이 경우 이 쓰레드는 곧바로 이 락을 획득할 겁니다.

반면, 이 leaf 노드가 쓰레드였다면, 우린 어떤 락도 기다리고 있지 않은 쓰레드를 가지고 있다는 의미이며, 이는 또다시 정의에 반하는 일입니다. 그리고 이 경우, 이 쓰레드는 수행 중이거나 락이 아닌 다른 무언가에 블록되어 있을 겁니다. 첫번째 경우라면, 무한 루프 버그가 존재하지 않는다면, 이 쓰레드는 결과적으로 락을 해제할 겁니다. 두번째 경우, 깨어나기 실패하는 버그가 존재하지 않는다면, 이 쓰레드는 결국은 깨어나서 락을 해제할 겁니다.⁷

따라서, 이 락 기반 deadlock의 정의에 기반하여, 연관된 그래프에는 사이클이 존재해야만 합니다.

□

Quick Quiz 7.3:

이 규칙에 대한 어떤 예외가 있어서, 라이브러리 코드가 호출자의 함수를 전혀 호출하지 않음에도 라이브러리와 호출자 양쪽의 락이 포함된 데드락 사이클이 존재할 수도 있나요?

■

Answer:

실제로 그런 예외가 있습니다! 여기 그 중 일부가 있습니다:

1. 이 라이브러리 함수의 인자 중 하나가 이 라이브러리 함수가 획득해야 하는 어떤 락으로의 포인터이고 이 라이브러리 함수가 이 호출자의 락을 잡는 동안 자신의 락을 잡고 있다면, 호출자와 라이브러리의 락 모두가 포함된 데드락 사이클을 가질 수 있습니다.
2. 이 라이브러리 함수 중 하나가 호출자에 의해 획득된 락으로의 포인터를 리턴한다면, 그리고 이 호출자가 이 라이브러리의 락이 잡혀 있는 사이 자신의 락을 잡았다면, 우린 호출자와 라이브러리의 락들이 포함된 데드락 사이클을 가질 수 있습니다.
3. 이 라이브러리 함수들 중 하나가 락을 획득하고 그걸 잡고 있는 채로 리턴한다면, 그리고 이 호출자가 자신의 락 중 하나를 잡는다면, 우린 호출자와 라이브러리의 락들이 모두 포함된 데드락 사이클을 만들 수 있는 또 다른 방법을 가지고 있게 됩니다.
4. 이 호출자가 락을 잡는 시그널 핸들러를 가지고 있다면, 데드락 사이클은 호출자와 라이브러리 락

⁷ 물론, 깨어나기 실패하는 버그의 한 종류는 락만이 아니라 다른 락이 아닌 자원이 연관되는 deadlock입니다. 하지만 여기서 질문은 “락 기반의 deadlock”이었습니다!

모두를 가질 수 있습니다. 하지만, 이 경우 라이브러리의 락은 이 데드락 사이클에서 결백한 방관자입니다. 그러나, 시그널 핸들러에서 락을 획득하는 것은 대부분의 경우 하면 안될 일임을 알아두시기 바랍니다—그건 그냥 나쁜 생각이 아니라, 하면 안되는 행동입니다. 하지만 여러분이 시그널 핸들러 내에서 명확히 락을 잡아야만 한다면, 해당 쓰레드 컨텍스트 내에서 같은 락을 잡는 것을 막기 위해 그 시그널을 막아둘 것을 분명히 하십시오.

□

Quick Quiz 7.4:

하지만 `qsort()` 가 비교 함수를 호출하기 전에 자신의 락을 모두 내려놓는다면, 다른 `qsort()` 쓰레드와의 레이스로부터 어떻게 자신을 보호하나요?

■

Answer:

비교되는 데이터 원소들의 소유권을 자신의 것으로 하거나 (Chapter 8에서 이야기 되었습니다) 레퍼런스 카운팅과 같은 미루기 메커니즘을 사용해서 (Chapter 9에서 이야기 되었습니다) 그럴 수 있습니다. 또는 Section 7.1.1.3에서 이야기 되었던 레이어 기반 락킹 계층을 사용할 수 있습니다.

다른 한편, 정렬되고 있는 키를 변경하는 것은 아무리 좋게 이야기 해도 용감하다고 밖에 할 도리가 없습니다.

□

Quick Quiz 7.5:

락으로의 포인터가 함수에 넘겨지는 혼란 경우 하나를 이야기해 보시죠.

■

Answer:

물론 락킹 기능들이죠!

□

Quick Quiz 7.6:

`pthread_cond_wait()` 이 먼저 이 뮤텍스를 해제하고 다시 그 뮤텍스를 획득한다는 사실은 데드락의 가능성은 제거하지 않나요?

■

Answer:

결코 아닙니다!

`mutex_a` 를 획득하고 이어서 `mutex_b` 를 획득하는, 그리고 나서는 `mutex_a` 를 `pthread_cond_wait()`에 넘기는 프로그램을 생각해 봅시다. 이제, `pthread_cond_wait()` 은 `mutex_a` 를 해제하지만 리턴하기 전에 이를 다시 획득할 겁니다. 만약 어떤 다른 쓰레드가 그 사이에 `mutex_a` 를 획득하고 `mutex_b` 를 기다리며 블락된다면, 이 프로그램은 데드락에 걸립니다.

□

Quick Quiz 7.7:

Listing 7.3에서 Listing 7.4로의 변환은 어디서든 적용될 수 있나요?

**Answer:**

결코 아닙니다!

이 변환은 `layer_2_processing()` 함수가 멱등하다고 가정하는데, `layer_1()` 라우팅 결정이 변화될 때 같은 함수에 대해서도 여러번 실행될 수 있다는 사실 때문입니다. 따라서, 실제 삶에서는, 이 변환이 무척 복잡할 수 있습니다.

**Quick Quiz 7.8:**

하지만 Listing 7.4의 복잡도는 그게 데드락을 방지한다는 점을 생각하면 가치있죠, 그렇죠?

**Answer:**

아마도요.

`layer_1()`에서의 라우팅 결정이 충분히 자주 바뀐다면, 이 코드는 항상 재시도를 하고 결코 진행을 못낼 겁니다. 이는 어떤 쓰레드도 진행을 못낸다면 “livelock”이라고, 그렇지 않고 일부 쓰레드는 진행을 내지만 나머지는 그러지 못한다면 “starvation”이라고 불립니다 (Section 7.1.2을 참고하세요).

**Quick Quiz 7.9:**

Section 7.1.6에서 설명된 “필요한 락들을 먼저 획득하기” 접근법을 사용할 때 livelock은 어떻게 회피할 수 있나요?

**Answer:**

추가적인 전역 락을 제공합니다. 어떤 쓰레드가 반복적으로 필요한 락을 획득하는 걸 시도하고 실패한다면, 해당 쓰레드가 조건 없이 새로운 전역 락을 획득하고, 이어서 조건적으로 모든 필요한 락들을 획득합니다. (Doug Lea에 의해 제안되었습니다.)

**Quick Quiz 7.10:**

락 A가 어떤 시그널 핸들러의 내부에선 결코 잡히지 않았지만, 락 B는 쓰레드 컨텍스트에서도 시그널 핸들러에서도 잡혔다고 해봅시다. 나아가서 락 A는 가끔 시그널이 블록되지 않은 상태에서도 잡힌다고 해봅시다. 락 B를 잡고 있는 사이에 락 A를 획득하는 건 왜 불법인가요?

**Answer:**

이는 데드락을 초래할 테니까요. 락 A는 가끔 시그널

핸들러 바깥에서 시그널을 블록하지 않은 상태에서 잡힌다는 걸 놓고 보면, 이 락을 잡는 사이 시그널이 처리될 수 있습니다. 여기 연관된 시그널 핸들러는 락 B를 잡을 수도 있고, 따라서 락 B는 락 A를쥔 상태에서 획득될 수 있습니다. 따라서, 우리가 락 B를 잡은 상태에서 락 A도 잡으면, 우린 deadlock 사이클을 갖게 됩니다. 이 문제는 락 B를 잡고 있는 사이 시그널이 블록되어 있다 해도 존재함을 알아두시기 바랍니다.

이는 인터럽트나 시그널 핸들러 내에서 획득되는 락에 무척 조심스러워야 하는 또다른 이유입니다. 하지만 리눅스 커널의 락의존성 검사기는 이 상황을 포함해 많은 걸 알고 있으니, 부디 그걸 항상 사용하십시오!

**Quick Quiz 7.11:**

시그널 핸들러 내에서 어떻게 시그널들을 합법적으로 블록할 수 있죠?

**Answer:**

그렇게 할 수 있는 가장 간단하고 빠른 방법은 해당 시그널을 설정할 때 `sigaction()`에 넘기는 `struct sigaction`의 `sa_mask` 필드를 사용하는 겁니다.

**Quick Quiz 7.12:**

시그널 핸들러 내에서 락을 잡는게 그렇게 나쁜 생각이라면, 그걸 안전하게 하는 방법을 이야기 하는 이유는 뭐죠?

**Answer:**

똑같은 규칙이 운영체제 커널과 일부 임베디드 어플리케이션의 인터럽트 핸들러에도 적용되기 때문입니다.

많은 어플리케이션 환경에서, 시그널 핸들러 내에서 락을 잡는 건 나쁜 행위로 알려져 있습니다 [Ope97]. 하지만, 이는 영리한 개발자들이 (아마도 혼명치 못하게도) 어토믹 오퍼레이션을 가지고 집에서 만든 락 기능을 사용하는 걸 막지는 못합니다. 그리고 어토믹 오퍼레이션들은 많은 경우 시그널 핸들러 내에서 사용이 합법입니다.

**Quick Quiz 7.13:**

간단한 락킹 계층, 레이어나 다른 것들이 존재하지 않게끔 제어를 객체 그룹 간에 자유로이 넘겨대는 객체지향 어플리케이션에서⁸는 이 어플리케이션을 어떻게 병렬화 시킬 수 있을까요?



⁸ “객체 지향 스파게티 코드”라고도 알려져 있습니다.

Answer:

여러 방법들이 있습니다:

1. 많은 수의 시뮬레이션이 (예를 들면) 기계적이나 전자적 기기의 좋은 설계로 수렴되게 하게끔 하는 시뮬레이션을 통한 패러메트릭 탐색의 경우, 시뮬레이션을 싱글쓰레드로 두되, 시뮬레이션의 각 인스턴스를 병렬로 돌립니다. 이는 객체 지향 설계를 유지하며, 높은 단계에서의 병렬성을 얻게 하며, 또한 deadlock 과 동기화 오버헤드를 회피합니다.
2. 객체를 한번에 하나의 그룹 이상의 객체들과는 작업하지 않게끔 여러 그룹으로 나눕니다. 그리고 나면 각 그룹에 락 하나씩을 연관짓습니다. 이는 Section 7.1.1.7에서 이야기 된 한번에 하나의 락 설계의 예입니다.
3. 객체를 어떤 그룹 기반 순서대로 각 그룹의 객체들과 작업하게끔 그룹짓습니다. 그리고 각 그룹별로 락을 연관짓고, 그룹들 간의 락킹 계층을 넣습니다.
4. 임의로 선택된 계층을 락들에 넣고, 락을 역순서로 획득해야 하면 Section 7.1.1.5에서 이야기한 조건적 락킹을 사용합니다.
5. 주어진 오퍼레이션 그룹을 이어가기 전에, 어떤 락이 획득될지 예측하고, 실제로 어떤 업데이트를 하기 전에 그것들을 획득하려 합니다. 만약 이 예측이 올바르지 않았음이 드러난다면, 모든 락을 내려놓고 경험의 이익을 포함하는 업데이트 된 예측을 가지고 재시도 합니다. 이 방법은 Section 7.1.1.6에서 이야기 되었습니다.
6. 트랜잭션 메모리를 사용합니다. 이 방법은 Sections 17.2–17.3에서 이야기 될 여러 장단점을 가지고 있습니다.
7. 어플리케이션을 더욱 동시성 친화적으로 리팩토링 합니다. 이는 싱글쓰레드로 돌아갈 때 조차도 더 빨리 수행되도록 어플리케이션을 만드는 부작용 또한 가질 확률이 높으나, 어플리케이션을 수정하기 더 어렵게 할 수도 있습니다.
8. 락킹에 추가적으로 뒤 챕터들에서 다룰 기법들을 사용합니다.

□

Quick Quiz 7.14:

Listing 7.5에 보인 livelock은 어떻게 회피될 수 있나요?

■

Answer:

Listing 7.4은 몇가지 좋은 힌트를 제공합니다. 많은

경우, livelock은 여러분이 락킹 설계를 다시 고려해야 한다는 힌트입니다. 또는 여러분의 락킹 설계가 “막 자랐다면” 처음으로 방문해야 하는 곳입니다.

그러나, Doug Lea가 제안한 충분히 좋은 한가지 방법은 Section 7.1.1.5에 이야기된 것과 같이 조건적 락킹을 사용하되 이를 Section 7.1.1.6에서 이야기한 것처럼 공유된 데이터를 수정하기 전에 모든 필요한 락들을 먼저 잡는 것입니다. 특정 크리티컬 섹션이 너무 여러번 재시도 된다면, 무조건적으로 전역 락을 잡고, 무조건적으로 모든 필요한 락들을 잡습니다. 이는 deadlock과 livelock을 모두 회피하며, 이 전역 락은 너무 자주 잡히지 않는다는 가정 하에 합리적 수준으로 확장합니다.

□

Quick Quiz 7.15:

Listing 7.6의 코드에는 어떤 문제들이 있나요?

■

Answer:

여기 두가지 문제가 있습니다:

- 1초의 대기는 대부분의 경우 너무 깁니다. 대기 기간은 대략 이 크리티컬 섹션을 수행하는데 걸리는 시간으로 시작되어야 하는데, 이는 보통 마리크로 세컨드나 밀리세컨드 범위가 될 겁니다.
2. 이 코드는 오버플로우를 검사하지 않습니다. 한편, 이 버그는 앞의 버그에 의해 제거될 수 있습니다: 32비트로 초를 저장하면 50년도 넘는 시간을 표현할 수 있습니다.

□

Quick Quiz 7.16:

Unfairness를 회피할 정도로 락 경쟁을 낮게 유지하는 좋은 병렬 설계를 사용하는게 낫지 않을까요?

■

Answer:

어떤 면에선 그게 낫겠습니다만, 때로는 높은 락 경쟁을 초래하는 설계를 사용하는게 적절한 상황이 있습니다.

예를 들어, 드문에러 조건을 갖는 시스템을 상상해 보세요. 그런 드문에러 조건 중 하나가 발동되었을 때에만 도움이 되는 복잡하고 디버깅하기 어려운 설계보다는 이 드문에러 상황의 기간 동안만 낮은 성능과 확장성을 갖는 간단한 에러 처리 설계가 최고일 수도 있습니다.

그러나, 예를 들면 문제 자체를 조개는 것과 같은, 간단하면서도 에러 조건 하에서도 효율적인 설계를 시도하는 노력을 할 가치는 있습니다.

□

Quick Quiz 7.17:

락을 잡은 쓰레드는 어떻게 간접받을 수 있나요?

**Answer:**

이 락에 의해 보호되는 데이터가 락 자체와 같은 캐시 라인에 존재한다면 다른 CPU들의 이 락을 획득하려는 시도들은 이 락을 쥐고 있는 CPU에게 비싼 캐시 미스를 일으킬 겁니다. 이는 거짓 공유 (false sharing)의 특수한 경우로, 다른 락들로 보호되는 한쌍의 변수가 같은 캐시 라인에 있을 때에도 일어날 수 있습니다. 반대로, 이 락이 자신이 보호하는 데이터와 다른 캐시 라인에 있다면, 이 락을 쥐고 있는 CPU는 해당 변수로의 첫번째 액세스 때에만 캐시 미스를 낼 겁니다.

물론, 락과 데이터를 별도의 캐시 라인에 두는 것의 단점은 경쟁되지 않은 경우에 단 하나가 아니라 두개의 캐시 미스를 일으킬 것이라는 겁니다. 항상 그렇듯, 현명한 선택을 하세요!

**Quick Quiz 7.18:**

배타적 락을 잡자마자 곧바로 놔버리는, 즉 텅 빈 크리티컬 섹션 같은 것을 갖는 것은 말이 될까요?

**Answer:**

텅 빈 락 기반 크리티컬 섹션은 드물게만 사용되지만, 쓰임새가 있습니다. 핵심은 배타적 락의 의미는 두개의 컴포넌트를 갖는다는 것입니다: (1) 친숙한 데이터 보호 의미 그리고 (2) 어떤 락을 해제한다는 것이 같은 락의 획득을 기다리는 쓰레드에게 보내는 메세지의 의미. 텅 빈 크리티컬 섹션은 데이터 보호 컴포넌트 없이 이 메세지 컴포넌트를 사용합니다.

이 답의 나머지 부분은 텅 빈 크리티컬 섹션의 사용 예를 제공합니다만, 이 예들은 “회색 마법”으로 여겨져야 합니다.⁹ 그처럼, 텅 빈 크리티컬 섹션은 실제 환경에서는 거의 사용되지 않습니다. 그러나, 이 회색 영역으로 들어가 봅시다...

텅 빈 크리티컬 섹션의 역사적 사용 중 하나는 2.4 리눅스 커널의 네트워킹 스택에 “big reader lock”의 약자로 brlock 이라 불린, 읽기 쓰레드쪽 확장성 있는 reader-writer 락에 의해 등장했습니다. 이 사용 예는 Section 9.5에서 설명되는 read-copy update (RCU)의 의미를 간략화 하는 방법이었습니다. 그리고 실제로 이 리눅스 커널의 사용 예는 RCU로 대체되었습니다.

이 텅 빈 락 기반 크리티컬 섹션 사용법은 일부 상황에서 락 컨텐션을 줄이기 위해 사용될 수도 있습니다. 예를 들어, 각 쓰레드가 쓰레드별 리스트에 관리되는 일의 단위를 처리하며, 쓰레드들은 각자의 리스트를 만지는 게

금지된 [McK12e] 멀티쓰레드 기반 유저 스페이스 어플리케이션을 생각해 봅시다. 앞서 스케줄된 모든 일 단위들이 업데이트가 진행되기 전에 완료되어야 함을 필요로 하는 업데이트도 있을 수 있습니다. 이를 처리하는 한가지 방법은 각 쓰레드에 하나의 일 단위를 스케줄 해서, 이 모든 일 단위들이 완료되었을 때, 업데이트가 진행될 수 있게 하는 것입니다.

일부 어플리케이션에서, 쓰레드는 오고 갈 수 있습니다. 예를 들어, 각 쓰레드는 어플리케이션의 한 사용자에 연관될 수도 있고, 따라서 해당 유저가 로그아웃하거나 연결이 끊겼을 때 제거될 수 있습니다. 많은 어플리케이션에서, 쓰레드는 원자적으로 떠날 수 없습니다: 그대신 명시적으로 스스로를 특정 순서의 액션들을 사용해 어플리케이션의 다양한 부분으로부터 제거해야 합니다. 그런 특정 액션 하나는 다른 쓰레드로부터 더이상 요청을 받는 것을 거부하는 것, 그리고 다른 특정 액션은 자신의 리스트의 남아있는 일 단위들을 폐기하는 것, 예를 들어 이 일 단위들을 전역 일 항목 폐기 리스트에 넣어 남아있는 쓰레드 중 하나가 이를 취하도록 하는 것일 겁니다. (왜 그 일들을 직접 함으로써 이 쓰레드의 일 항목 리스트를 비우지 않을까요? 주어진 일 항목이 더 많은 일 항목을 생성해서 이 리스트가 빠른 시간 내에 비워지지 않을 수 있기 때문입니다.)

이 어플리케이션이 성능도 좋고 확장성도 좋아야 한다면, 좋은 락킹 설계가 필요합니다. 한가지 흔한 해결책은 쓰레드 제거 전체 프로세스를 위한 하나의 전역 락을 (G 라 부릅시다) 개별 제거 오퍼레이션을 보호하는 잔규모의 락과 함께 사용하는 것입니다.

이제, 제거되는 쓰레드는 자신의 리스트의 일을 폐기하기 전에 이어지는 요청들을 받는 것을 분명히 거부해야하는데, 그러지 않는다면 이 폐기 액션 후에 추가적인 일이 도착할 수 있으며, 이는 이 폐기 액션이 효과적이지 못하게 할 것이기 때문입니다. 따라서 제거되는 쓰레드를 위한 간략화된 슈도코드는 다음과 같을 겁니다:

1. G 락을 획득합니다.
2. 통신을 보호하는 락을 잡습니다.
3. 다른 쓰레드로부터의 더이상의 통신을 거부합니다.
4. 통신을 보호하는 락을 해제합니다.
5. 전역 일 항목 폐기 리스트를 보호하는 락을 잡습니다.
6. 모든 보류 중인 일 항목을 전역 일 항목 폐기 리스트로 이동시킵니다.
7. 전역 일 항목 폐기 리스트를 보호하는 락을 해제합니다.

⁹ 이 설명을 제공한 Alexey Roytman에게 감사를 표합니다.

8. G 락을 해제합니다.

물론, 모든 미리 존재하던 일 항목을 기다려야 하는 쓰레드는 떠나는 쓰레드를 신경써야 합니다. 이를 자세히 알아보기 위해, 어떤 제거되는 쓰레드가 다른 쓰레드로부터의 더이상의 통신을 거부한 직후에 모든 미리 존재한 일 항목을 기다리는 쓰레드가 시작되었다고 생각해 봅시다. 쓰레드들은 서로의 일 항목 리스트에 접근할 수 없는데 이 쓰레드는 이 제거되는 쓰레드의 일 항목이 완료되기를 기다릴 수 있을까요?

한가지 간단한 방법은 이 쓰레드가 G 를, 그리고 전역 일 항목 폐기 리스트를 보호하는 락을 잡고, 이 일 항목들을 자신의 리스트로 옮기는 것입니다. 이 쓰레드는 이후에 두 락을 해제하고, 일 항목 중 하나를 자신의 리스트의 끝에 위치시키고, 각 쓰레드의 리스트 (자신의 것 포함)에 있는 일 항목들이 모두 완료되기를 기다립니다.

이 방법은 많은 경우에 잘 동작합니다만, 각 일 항목이 전역 일 항목 폐기 리스트에서 가져와지는 만큼 특별한 처리가 필요하다면, G 로의 지나친 컨텐션이 초래될 수 있습니다. 이 컨텐션을 방지하는 한가지 방법은 G 를 획득하고는 곧바로 해제하는 것입니다. 그러면 앞의 모든 일 항목이 완료되기를 기다리는 프로세스는 다음과 같이 됩니다:

1. 한 전역 카운터를 1로 값 설정하고 한 조건 변수를 0으로 초기화 합니다.
2. 모든 쓰레드에게 그것들이 원자적으로 이 전역 카운터의 값을 증가시키도록, 그리고 나서 일 항목을 넣도록 메세지를 보냅니다. 이 일 항목은 원자적으로 이 전역 카운터의 값을 감소시키며, 그 결과 그 값이 0이 되면, 이는 한 조건 변수의 값을 1로 설정합니다.
3. 모든 현재 제거되는 중인 쓰레드가 제거되기를 끝낼 때까지 기다리게끔 G 를 획득합니다. 한번에 하나의 쓰레드만 제거될 것이므로, 모든 남아있는 쓰레드는 이미 앞의 단계에서 보낸 메세지를 받았을 겁니다.
4. G 를 해제함.
5. 전역 일 항목 폐기 리스트를 보호하는 락을 획득합니다.
6. 모든 일 항목을 전역 일 항목 폐기 리스트에서 이 쓰레드의 리스트로 이동시키고, 그것들을 필요한 대로 처리합니다.
7. 전역 일 항목 폐기 리스트를 보호하는 락을 해제합니다.

8. 추가적인 일 항목을 이 쓰레드의 리스트에 넣습니다. (앞에서와 마찬가지로, 이 일 항목은 원자적으로 전역 카운터의 값을 감소시키고, 그 결과 값이 0 이 되면 조건 변수를 1로 설정합니다.)

9. 이 조건 변수가 값 1이 되기를 기다립니다.

이 프로세스가 완료되면, 모든 앞서서부터 존재한 일 항목들은 완료되었을 것이 보장됩니다. 텅 빈 크리티컬 섹션은 락킹을 메세징은 물론이고 데이터의 보호를 위해서도 사용됩니다.

□

Quick Quiz 7.19:

VAX/VMS DLM 이 reader-writer 락을 에뮬레이션 하는 다른 방법도 있을까요?

■

Answer:

실제로 여러가지가 있습니다. 한가지 방법은 null, 보호읽기, 그리고 배타 모드를 사용하는 것입니다. 다른 방법은 null, 보호읽기, 그리고 동시쓰기 모드를 사용하는 것입니다. 세번째 방법은 null, 동시읽기, 그리고 배타모드를 사용하는 것입니다.

□

Quick Quiz 7.20:

Listing 7.7 의 코드는 우습고 복잡합니다! 왜 하나의 전역 락을 조건적으로 획득하지 않죠?

■

Answer:

조건적으로 하나의 글로벌 락을 획득하는 것은 실제로 잘 동작합니다만, 상대적으로 적은 수의 CPU 에 대해서만 그렇습니다. 그게 왜 수백개의 CPU 를 가진 시스템에서는 문제가 되는지 알기 위해선 Figure 5.1 를 보시기 바랍니다.

□

Quick Quiz 7.21:

잠시만요! 우리가 Listing 7.7 의 라인 16 에서 이 토퍼먼트에 승리했다면 우린 do_force_quiescent_state() 의 일을 해야만 하게 됩니다. 이게 정확히 어떻게 승리라고 할 수 있나요?

■

Answer:

진짜 어떻게 그럴까요? 이는 단지 동시성에서는 삶에서와 마찬가지로 게임을 하기 전에 승리가 정확히 무엇을 의미하는지 알고 주의해야 함을 의미합니다.

□

Quick Quiz 7.22:

Listing 7.8 의 라인 2에서 보인 명시적인 초기화를 사용하는 대신 C 언어의 기본 0으로 초기화 기능을 사용하지 않나요?

**Answer:**

이 기본 초기화 기능은 함수의 범위 내에서 auto 변수로 할당된 락에는 적용되지 않기 때문입니다.

**Quick Quiz 7.23:**

Listing 7.8 의 라인 7-8에 있는 안쪽 반복문을 왜 신경쓰나요? 그냥 라인 6에서 원자적 교환 오퍼레이션을 반복하는게 어떤가요?

**Answer:**

이 락이 이미 잡혔고 여러 쓰레드가 이 락을 잡으려 시도한다고 생각해 봅시다. 이 경우, 이 쓰레드들이 모두 어토믹 교환 오퍼레이션을 반복하고 있게 된다면, 이들은 서로들 사이에 이 락을 쥐고 있는 캐쉬 라인을 주고 받기를 반복할 것이어서, 인터컨넥트 부위에 로드를 의미합니다. 반면에, 이 쓰레드들이 라인 7-8의 안쪽 루프에서 수행을 반복하고 있으면, 각자의 캐쉬만 가지고 수행을 반복하므로 인터컨넥트에는 무시할 수 있을만한 부하만을 일으킬 겁니다.

**Quick Quiz 7.24:**

Listing 7.8 의 라인 14에서는 왜 간단히 이 락에 0을 저장하지 않나요?

**Answer:**

이것도 정당한 구현이 될 수 있습니다만, 이 저장이 메모리 배리어에 앞서고 `WRITE_ONCE()`를 사용할 때에만 그렇습니다. 이 메모리 배리어는 `xchg()` 오퍼레이션이 사용될 때에는 요구되지 않는데 이 오퍼레이션은 그것이 값을 리턴한다는 사실 때문에 완전한 메모리 배리어를 내포하기 때문입니다.

**Quick Quiz 7.25:**

카운터의 최대값이 정해져 있는데 어떻게 한 카운터가 다른 것보다 크다고 말할 수 있나요?

**Answer:**

C 언어에서라면 다음 매크로가 이를 올바르게 처리합니다:

```
#define ULONG_CMP_LT(a, b) \
    (ULONG_MAX / 2 < (a) - (b))
```

간단히 두개의 부호 있는 정수를 빼기하고 싶을 수 있겠지만, 부호 있는 변수의 오버플로우는 C 언어에서 정의되지 않아 있으므로 사용되지 않아야 합니다. 예를 들어, 컴파일러가 이 값을 중 하나는 양수이고 나머지는 음수임을 안다면, 이 양수에서 이 음수를 빼는 것이 오버플로우를 일으키고 그 결과 음수가 될지라도 컴파일러는 이 양수가 음수보다 크다고 가정해 버릴 수 있습니다.

컴파일러는 두 숫자의 부호를 어떻게 알 수 있을까요? 이전의 값 할당과 비교로부터 이를 추측할 수도 있습니다. 이 경우, 이 CPU 별 카운터가 부호를 가지고 있었다면, 컴파일러는 그것이 항상 증가만 한다고 추측 할 수 있고, 그러면 이것들이 결코 음수가 되지 않는다고 가정할 수도 있습니다. 이 가정은 이 컴파일러가 불행한 코드를 생성하게 만들 수 있습니다 [McK12d, Reg10].

**Quick Quiz 7.26:**

뭐가 낫나요, 카운터 방법인가요 플래그 방법인가요?

**Answer:**

플래그 방법은 일반적으로 더 적은 캐쉬 미스를 낼 겁니다만, 더 나은 답변은 둘 다 시도해 보고 여러분의 특정 워크로드에 최선의 것이 무엇인지 보라는 것이겠습니다.

**Quick Quiz 7.27:**

묵시적 존재 보장에 기대는게 어떻게 버그에 이를 수 있죠?

**Answer:**

여기 올바르지 않은 묵시적 존재 보장의 사용으로부터 나올 수 있는 버그들이 있습니다:

1. 한 프로그램이 한 전역 변수로의 주소를 어떤 파일에 쓰고, 같은 프로그램의 나중 인스턴스가 이 주소를 읽고 역참조 하려 합니다. 이는 주소 공간 무작위화 (address-space randomization) 때문에 실패할 수 있으며, 이 프로그램의 재 컴파일링에 대해선 말할 것도 없습니다.
2. 어떤 모듈은 자신의 변수로의 주소를 어떤 다른 모듈에 위치한 포인터에 저장하고, 이 모듈이 언로드된 다음에 이 포인터를 역참조 하려 할 수 있습니다.
3. 어떤 함수는 자신의 스택 상의 변수의 주소를 어떤 전역 변수에 저장하고, 이 함수가 리턴된 다음에 어떤 다른 함수가 그 주소를 역참조 할 수도 있습니다.

여러분은 추가적인 가능성을 가져올 수 있으리라 확신합니다.



Quick Quiz 7.28:

우리가 제거해야 하는 원소가 Listing 7.9 의 라인 8 에 있는 리스트의 첫번째 원소가 아니라면 어떻게 되는 걸까요?



Answer:

이건 채이닝이 없는 매우 간단한 해쉬 테이블이며, 따라서 특정 버킷에 들어있는 유일한 원소는 첫번째 원소입니다. 독자 여러분은 이 예를 완전한 채이닝을 갖는 해쉬 테이블에 적용해 보셔도 좋겠습니다.



E.8 Data Ownership

Quick Quiz 8.1:

C 또는 C++ 로 공유 메모리 병렬 프로그램들을 (예를 들면, pthread 를 사용해서) 만들 때 어떤 형태의 데이터 소유권이 제거하기 엄청 어렵나요?



Answer:

함수에서의 auto 변수의 사용입니다. 기본적으로, 이것들은 현재 함수를 수행하는 쓰레드에 사유화 되어 있습니다.



Quick Quiz 8.2:

Section 8.1 에서 보인 이 예에 남아 있는 동기화는 무엇인가요?



Answer:

sh & 오퍼레이터를 통한 쓰레드의 생성과 sh wait 커맨드를 통한 쓰레드 기다리기입니다.

물론, 프로세스가 예를 들어 shmget() 이나 mmap() 시스템 콜을 이용해 명시적으로 메모리를 공유한다면, 이 공유된 메모리를 접근하거나 업데이트 할 때 명시적 동기화가 필요할 겁니다. 프로세스들은 또한 다음 프로세스간 통신 메커니즘 중 무엇을 이용해서든 동기화를 할 수도 있을 겁니다:

1. System V 세마포어.
2. System V 메세지 큐.
3. 유닉스 도메인 소켓.

4. TCP/IP, UDP, 그리고 다른 것들의 모든 호스트를 포함한 네트워킹 프로토콜.

5. 파일 락킹.

6. O_CREAT 와 O_EXCL 플래그를 사용하는 open() 시스템콜.

7. rename() 시스템콜의 사용.

가능한 동기화 메커니즘의 완전한 리스트는 독자 여러분의 연습문제로 남겨두는데, 그것은 무척 긴 리스트가 될 것이라는 경고를 남겨둡니다. 예상치 못한 시스템콜들의 놀랄만큼 큰 수는 서비스로서의 동기화 메커니즘으로 압착될 수 있습니다.



Quick Quiz 8.3:

Section 8.1 에 보인 예에 어떤 공유되는 데이터가 있나요?



Answer:

이건 철학적 질문입니다.

“아니오” 라는 대답을 원하는 사람들은 프로세스들이 정의대로 메모리를 공유하지 않는다고 주장할 수도 있겠습니다.

“네” 라는 대답을 원하는 사람들은 공유 메모리를 필요로 하지 않는 많은 수의 동기화 메커니즘을 나열할 수 있겠고, 커널은 어떤 공유된 상태를 가질 것을 이야기할 수 있으며, 어쩌면 프로세스 ID (PID) 의 할당은 공유 데이터로 간주될 수 있다고까지 주장할 수도 있겠습니다.

그런 논쟁은 무척 지적인 행위이고, 지적이라 느끼고 불행한 학우나 동료에게서 점수를 따기에 좋은 방법입니다만, 유용한 무엇인가가 되는 것을 막는 방법인 경우가 대부분입니다.



Quick Quiz 8.4:

각 쓰레드가 각자의 쓰레드별 변수 인스턴스를 읽지만 다른 쓰레드의 인스턴스에 쓰기를 하는 부분적 데이터 소유권이 말이 되긴 할까요?



Answer:

놀랍게도, 그렇습니다. 한가지 예는 쓰레드들이 다른 쓰레드의 우편함에 메세지를 보내고 각 쓰레드는 그 메세지가 처리되고 나면 그 메세지를 제거할 책임을 갖는 경우와 같은 간단한 메세지 전달 시스템이 되겠습니다. 그런 알고리즘의 구현은 비슷한 소유권 패턴의 다른 알고리즘을 알아보는 것과 함께 독자 여러분의 연습문제로 남겨둡니다.



Quick Quiz 8.5:

POSIX 시그널 외에 어떤 다른 메커니즘이 함수 보내기 위해 사용될 수 있을까요?

**Answer:**

그런 메커니즘들이 매우 많은데, 다음을 포함합니다:

1. System V 메세지 큐.
2. 공유 메모리 디큐 (Section 6.1.2 를 참고하세요).
3. 공유 메모리 메일함.
4. 유닉스 도메인 소켓.
5. TCP/IP 또는 UDP, 그리고 RPC, HTTP, XML, SOAP, 그 외에도 여러 더 높은 단계에서의 프로토콜과 결합된 형태의 것들.

완전한 리스트를 만드는 것은 그 리스트는 굉장히 길 것이라 경고받은, 그럼에도 그것을 원하는 독자 여러분의 연습문제로 남겨둡니다.

**Quick Quiz 8.6:**

하지만 Listing 5.5 의 라인 17–34 에 있는 `eventual()` 함수의 어느 부분도 실제로 `eventual()` 쓰레드에 소유되어 있지 않습니다! 이게 어떻게 데이터 소유권인가요???

**Answer:**

핵심 문구는 “데이터로의 권한을 소유한다” 입니다. 이 경우, 이 권한은 이 리스트의 라인 1에 정의된 쓰레드별 `counter` 변수로의 접근 권한입니다. 이 상황은 Section 8.2 에서 설명한 것과 비슷합니다.

하지만, `eventual()`에 소유되는 데이터가 실제로 있는데, 이 리스트의 라인 19 와 20에 정의된 `t` 와 `sum` 변수들입니다.

특수 쓰레드의 다른 예들을 위해서는, 리눅스 커널의 커널 쓰레드들을 보실 수 있겠는데, 예를 들면 `kthread_create()` 와 `kthread_run()` 에 의해 만들어지는 것들입니다.

**Quick Quiz 8.7:**

쓰레드별 데이터의 완전한 사유화를 유지하면서 높은 정확도를 유지하는게 가능할까요?

**Answer:**

그렇습니다. 한가지 방법은 `read_count()` 각 자신의 쓰레드별 변수의 값을 더하게 하는 겁니다. 이는 완전한 소유권과 성능을 유지하게 하지만 정확도에는 약간의 개선만을 가져다 주는데, 무척 커다란 수의 쓰레드를 갖는 시스템에서는 특히 그렇습니다.

또 다른 방법은 `read_count()` 가 함수 보내기를 사용하게 하는 건데, 예를 들면 쓰레드별 시그널의 형태입니다. 이는 정확도를 크게 개선하지만 `read_count()` 에 심각한 성능 비용을 청구합니다.

하지만, 이 방법들 둘 다 카운터 업데이트라는 일반적인 경우에서의 캐쉬 쓰래싱을 제거하는 장점을 갖습니다.



E.9 Deferred Processing

Quick Quiz 9.1:

메모리 해제 후 사용 체크를 왜 신경쓰죠?

**Answer:**

버그를 발견할 확률을 높이기 위해서입니다. 한 타입의 구조체를 할당하고 해제하기만 하는 작은 고문 테스트 프로그램 (`routetorture.h`)는 놀랍도록 많은 양의 메모리 해제 후 사용 문제를 제어할 수 있습니다. 버그를 찾는 확률을 높이는 것의 중요성에 대해 더 많은 내용을 위해 페이지 207 의 Figure 11.4 와 페이지 208 에서 시작하는 Section 11.6.4 의 관련된 이야기를 보시기 바랍니다.

**Quick Quiz 9.2:**

Listing 9.3 의 `route_del()` 은 메모리 해제될 원소로의 횡단을 보호하기 위해서는 레퍼런스 카운트를 사용하지 않나요?

**Answer:**

이 횡단은 이미 락을 통해 보호되고 있어서 추가적 보호가 필요하지 않기 때문입니다.



Quick Quiz 9.3:

Figure 9.2 의 224 CPU 에서의 “ideal” 선의 끊어짐은 무엇 때문인가요? 곧은 선이어야 하지 않아요?

**Answer:**

그 끊어짐은 하이퍼쓰레딩 때문입니다. 이 시스템에서, 소켓 내의 각 코어의 첫번째 하드웨어 쓰레드는 연속적인 CPU 숫자를 가지며, 이 숫자는 다른 소켓의 각 코어의 첫번째 하드웨어들로 이어지고, 마지막으로 모든 소켓의 각 코어의 두번째 하드웨어 쓰레드로 이어집니다. 이 시스템에서, CPU 숫자 0-27 는 이 첫번째 소켓의 28개 코어의 첫번째 하드웨어 쓰레드이고, 숫자 28-55 는 두번째 소켓의 28개 코어의 첫번째 하드웨어 쓰레드이며, 그렇게 이어져서 196-223의 숫자는 여덟번째 소켓의 28개 코어의 첫번째 하드웨어 쓰레드입니다. 따라서 CPU 숫자 224-251 는 첫번째 소켓의 28개 코어의 두번째 하드웨어 쓰레드이고 252-279 는 두번째 소켓의 28개 코어의 두번째 하드웨어 쓰레드이며, 그렇게 이어져서 420-447 까지는 여덟번째 소켓의 28개 코어의 두번째 하드웨어 쓰레드입니다.

이게 왜 문제일까요?

코어의 두개 하드웨어 쓰레드는 자원을 공유하며, 이 워크로드는 하나의 하드웨어 쓰레드가 그 코어의 자원의 절반 이상을 소비하게 하는 것 같습니다. 따라서, 해당 코어의 두번째 하드웨어 쓰레드를 추가하는 것은 희망하는 것보다 자원을 덜 추가하는 게 됩니다. 다른 워크로드는 각 코어의 두번째 하드웨어 쓰레드에서 더 큰 이득을 얻을 수도 있겠습니다만, 하드웨어와 워크로드 양쪽의 세부사항에 의존적일 겁니다.

**Quick Quiz 9.4:**

Figure 9.2 의 refcnt 트레이스는 최소한 x-axis에서 좀 떨어져야 하는 거 아닌가요???

**Answer:**

“약간” 을 정의해 주세요.

Figure E.4 는 같은 데이터를 로그-로그 플랫 (log-log plot)으로 보입니다. 볼 수 있듯, refcnt 라인은 두개 CPU에서 5,000 아래로 떨어집니다. 이는 두개 CPU에서의 이 refcnt 성능은 Figure 9.2 에서의 첫번째 y-axis tick인 5×10^6 보다 친배 이상 적음을 의미합니다. 따라서, Figure 9.2 에 보인 레퍼런스 카운팅의 성능 묘사는 너무나도 정확합니다.

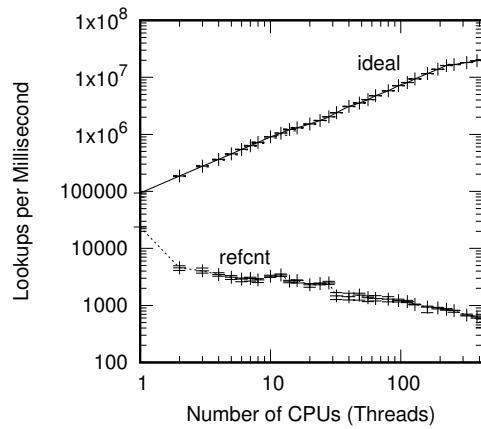


Figure E.4: Pre-BSD Routing Table Protected by Reference Counting, Log Scale

Quick Quiz 9.5:

동시성이 “레퍼런스 카운팅의 유용성을 분명 줄였다”면, 리눅스 커널에는 왜 그렇게 많은 레퍼런스 카운터가 존재하나요?

**Answer:**

해당 문장은 “유용성을 줄였다”고 했지, “유용성을 제거했다”고는 하지 않았죠?

리눅스 커널이 상당히 동시적인 환경에서 레퍼런스 카운팅의 장점을 취하기 위해 사용하는 기법들 중 일부를 이야기 하는 Section 13.2 를 보시기 바랍니다.

**Quick Quiz 9.6:**

해저드 포인터 논문들은 삭제된 원소들을 표시하기 위해 각 포인터의 바닥 비트들을 사용하는데 HAZPTR_POISON 은 왜 사용되나요?

**Answer:**

해저드 포인트의 해당 공개된 구현은 삽입과 삭제에 non-blocking 동기화 기법을 사용했습니다. 이 기법들은 이 데이터 구조를 횡단하는 읽기 쓰레드가 업데이트 쓰레드가 업데이트를 완료하는 것을 “도와줄” 것을 필요로 하는데 이는 읽기 쓰레드들이 삭제된 원소의 다음 원소를 봐야 함을 의미합니다.

대조적으로, 우린 업데이트들을 동기화 하기 위해 락킹을 사용할텐데, 이는 읽기 쓰레드들이 업데이트 쓰레드들의 업데이트 완료를 도울 필요를 없애버려서 결국 우리가 포인터의 바닥 비트를 그대로 둘 수 있게 해줍니다. 이 방법은 읽기 쪽 코드를 더 간단하고 빠르게 해줍니다.



Quick Quiz 9.7:

Listing 9.4 의 `hp_try_record()` 는 왜 데이터 원소로의 이중 간접 경로를 갖나요? `void **` 대신 `void *` 를 쓰는게 어렵습니까?

**Answer:**

`hp_try_record()` 는 동시에 수정을 검사해야만 하기 때문입니다. 이 일을 하기 위해선 이 원소로의 포인터로의 변경을 검사하기 위해 이 원소로의 포인터로의 포인터를 필요로 합니다.

**Quick Quiz 9.8:**

왜 `hp_try_record()` 를 신경쓰죠? 그냥 실패에 내성 있는 `hp_record()` 함수를 그냥 사용하는게 더 쉽지 않나요?

**Answer:**

어떤 측면에선 그게 더 쉬울 수도 있습니다만, Pre-BSD 라우팅 예에서 곧 보겠지만, `hp_record()` 는 단순하게 동작하지 않는 상황이 있습니다.

**Quick Quiz 9.9:**

읽기 쓰레드는 “일반적으로” 재시작해야만 한다구요? 그 예외는 어떤 것들인가요?

**Answer:**

만약 그 포인터가 전역 변수로부터 나오거나 메모리 해제되지 않을 것이라면, `hp_record()` 는 동시에 삭제에도 불구하고 이 해저드 포인터를 기록하는 시도를 반복할 수 있습니다.

그런 경우, 재시작은 Folly 오픈소스 라이브러리에 구현된 `UnboundedQueue` 와 `ConcurrentHashMap` 데이터 구조에서 보여진 것처럼 연결 카운팅을 사용해 회피될 수 있습니다.¹⁰

**Quick Quiz 9.10:**

하지만 이런 해저드 포인터의 제약들은 다른 형태의 레퍼런스 카운팅에도 적용되지 않나요?

**Answer:**

그렇기도 하고 아니기도 합니다. 이 제약들은 참조 획득이 실패할 수도 있는 레퍼런스 카운팅 메커니즘들에만 적용됩니다.

**Quick Quiz 9.11:**

Figure 9.3 는 하이퍼쓰레드가 일으키는 224 쓰레드에서의 성능 하락의 기미를 보이지 않습니다. 왜 그런거죠?

**Answer:**

현대의 마이크로프로세서는 복잡한 야수여서, 모든 간단한 답변에는 상당한 회의주의가 적당합니다. 그건 그렇다 치고, 가장 그럴싸한 이유는 해저드 포인터 읽기 쓰레드에게 완전한 메모리 배리어가 요구된다는 것입니다. 그런 메모리 배리어에서 기인하는 모든 지연은 해당 코어를 공유하는 다른 하드웨어 쓰레드에게 가능한 시간을 주어서 하드웨어 쓰레드당 성능의 비용으로 더 큰 확장성을 초래할 수 있습니다.

**Quick Quiz 9.12:**

“Structured Deferral: Synchronization via Procrastination” [McK13] 이라는 논문은 해저드 포인터가 이상에 가까운 성능을 보임을 이야기 했습니다. Figure 9.3 에는 무슨 일이 벌어진 건가요???

**Answer:**

첫째로, Figure 9.3 는 선형 y 축을 가지고 있는 반면 “Structured Deferral” 논문의 그래프는 로그스케일 y 축을 가지고 있습니다. 다음으로, 해당 논문은 가벼운 부하의 해시 테이블을 사용하는 반면, Figure 9.3 의 것은 10개 원소를 갖는 간단한 링크드 리스트를 사용하는데, 이는 해저드 포인터가 “Structured Deferral” 논문에서의 것보다 이 워크로드에서 더 큰 메모리 배리어 페널티를 받게 됨을 의미합니다. 마지막으로, 해당 논문은 오래된 작은 크기의 x86 시스템을 사용한 반면, Figure 9.3 에 보인 데이터를 만드는데에는 훨씬 새롭고 커다란 시스템이 사용되었습니다.

추가로, 비대칭 배리어의 짙을 맞춘 사용 [Mic08, Cor10b, Cor18] 이 이 방법을 지원하는 시스템에서 읽기 쪽 해저드 포인터 메모리 배리어를 제거하기 위해 제안되었는데 [Gol18], 이는 이 그림에 보인 것보다 해저드 포인터의 성능을 더 개선할 수도 있습니다.

항상 그렇듯, 여러분의 이득은 다양할 수 있습니다. 성능 차이를 놓고 볼 때, 해저드 포인터는 (메모리 배리어 오버헤드가 최소한 캐시 미스 페널티를 넘어설 수 있는) 매우 커다란 데이터 구조와 탐색 오퍼레이션이 최소의 해저드 포인터를 필요로 하는 해시 테이블과 같은 데이터 구조에서는 여러분에게 최고의 성능을 제공할 것이 분명합니다.



¹⁰ <https://github.com/facebook/folly>

Quick Quiz 9.13:

이 시퀀스 락 이야기는 왜 Chapter 7에 있지 않죠, 이건 락킹이잖아요?

**Answer:**

이 시퀀스 락 메커니즘은 실제로 두개의 별개의 동기화 메커니즘인 시퀀스 카운트와 락킹의 조합입니다. 사실, 이 시퀀스 카운트 메커니즘은 리눅스 커널에서 `write_seqcount_begin()` 과 `write_seqcount_end()` 기능을 통해 사용할 수 있습니다.

하지만, `write_seqlock()` 과 `write_sequnlock()`의 조합이 리눅스 커널에서는 훨씬 더 많이 사용됩니다. 더 중요하게, 많은 사람들이 “시퀀스 카운트” 보다는 “시퀀스 락”이라고 말할 때 더 잘 알아들을 겁니다.

따라서 이 섹션은 사람들이 제목으로부터 이게 무엇인지 이해할 수 있게끔 “Sequence Locks”라고 제목지어져 있으며, “Deferred Processing”으로 표현되어 있는 이유는 (1) “시퀀스 락”의 “시퀀스 카운트”를 강조하고 (2) “시퀀스 락”은 간단한 락 그 이상의 무엇이기 때문입니다.

**Quick Quiz 9.14:**

Listing 9.10의 `read_seqbegin()`은 왜 이미 망한 읽기가 시작되게 하는 대신 아래쪽 bit이 셋 되었는지 검사하고 내부적으로 재시도 하지 않나요?

**Answer:**

그건 합법적인 구현이 될 겁니다. 하지만, 워크로드가 읽기가 대부분이라면, 일반적인 경우 성공적 읽기의 오버헤드를 증가시킬 수 있는데, 이는 생산적이지 못할 겁니다. 그러나, 충분히 큰 업데이트 쓰레드의 비율과 충분히 높은 읽기 쓰레드의 오버헤드가 주어진다면, `read_seqbegin()` 내부의 체크를 갖는 것이 선호될 수 있습니다.

**Quick Quiz 9.15:**

Listing 9.10의 line 26의 `smp_mb()`는 왜 필요한가요?

**Answer:**

그게 없다면, 컴파일러도 CPU도 `read_seqretry()` 앞의 크리티컬 섹션을 이 함수 다음으로 재배치 할 권리를 갖습니다. 이는 이 시퀀스 락이 크리티컬 섹션을 보호하는 걸 방해할 겁니다. `smp_mb()` 기능은 그런 재배치를 막습니다.

**Quick Quiz 9.16:**

Listing 9.10의 코드에서는 더 완화된 형태의 메모리 배리어를 사용할 순 없나요?

**Answer:**

오래전 버전의 리눅스 커널에서는 안됩니다.

매우 최신 버전의 리눅스 커널에서는 line 16은 `READ_ONCE()` 대신 라인 17의 `smp_mb()`를 없애도 되게 할 수 있는 `smp_load_acquire()`를 사용할 수도 있습니다. 비슷하게, 라인 41는 예를 들면 다음과 같이 `smp_store_release()`를 사용할 수 있습니다:

```
smp_store_release(&slp->seq, READ_ONCE(slp->seq) + 1);
```

이는 라인 40의 `smp_mb()`를 제거할 수 있게 할 겁니다.

**Quick Quiz 9.17:**

시퀀스 락킹 업데이트 쓰레드가 읽기 쓰레드를 starve시키는 걸 무엇이 방지하나요?

**Answer:**

아무것도요. 이것은 시퀀스 락킹의 약점 중 하나이며, 그 결과, 여러분은 시퀀스 락킹을 읽기가 대부분인 환경에서만 사용해야 합니다. 그렇지 않다면 읽기 쪽의 starvation은 여러분의 환경에서는 허용되는 것이니, 이 경우라면, 시퀀스 락킹 업데이트를 가지고 잘 놀아보시기 바랍니다!

**Quick Quiz 9.18:**

만약 뭔가 다른게 쓰기 쓰레드들을 직렬화 시켜서 이 락이 필요 없게 만들면 어떤가요?

**Answer:**

이 경우, `->lock` 필드는 없어질 수 있는데, 리눅스 커널의 `seqcount_t` 가 그렇습니다.

**Quick Quiz 9.19:**

Listing 9.10의 라인 2의 `seq`는 왜 `unsigned long`이 아니라 `unsigned`인가요? 어쨌건, `unsigned` 가 리눅스 커널에 충분히 좋다면, 모두에게도 충분히 좋아야 하지 않나요?

**Answer:**

전혀 그렇지 않습니다. 리눅스 커널은 다음 순서의 이벤트들을 무시할 수 있게 하는 특별한 속성을 여럿 가지고 있습니다:

- 쓰레드 0 이 `read_seqbegin()` 을 수행해서, line 16 에서 `->seq` 를 가져와 그 값이 짹수임을 알게 되고, 따라서 호출자에게 리턴합니다.
- 쓰레드 0 이 자신의 읽기 쪽 크리티컬 섹션을 수행하기 시작하지만 이어서 오랫동안 `preemption` 당합니다.
- 다른 쓰레드가 반복적으로 `write_seqlock()` 와 `write_sequnlock()` 을 호출해서 `->seq` 의 값이 오버플로우 되어 쓰레드 0 이 읽었던 값으로 되돌려집니다.
- 쓰레드 0 이 수행을 재개해서, 비일관적인 데이터를 가지고 읽기 크리티컬 섹션을 완료합니다.
- 쓰레드 0 이 `read_seqretry()` 를 호출하는데, 이는 쓰레드 0 이 이 시퀀스 락에 의해 보호되는 데 이터의 일관적인 값만을 보았다고 잘못된 결론을 내립니다.

리눅스 커널은 시퀀스 락킹을 가끔만 업데이트 되는 것들을 위해 사용하는데, 하루 중 시간 정보가 그런 경우입니다. 이 정보는 최대 밀리세컨드마다 업데이트 되므로, 이 카운터를 오버플로우 시키려면 7주가 걸립니다. 만약 어떤 커널 쓰레드가 7주간 `preemption` 당한다면, 리눅스 커널의 soft-lockup 코드는 그 동안 매 2분마다 경고를 내보낼 겁니다.

반대로, 64 비트 카운터에서는 매 나노세컨드마다 업데이트가 된다고 해서 오버플로우를 위해 5세기가 넘게 걸립니다. 따라서, 이 구현은 64 비트 시스템에서 `->seq` 를 위해 64 비트 타입을 사용합니다.

□

Quick Quiz 9.20:

이 버그는 고쳐질 수 있나요? 달리 말하자면, 시퀀스 락을 동시에 추가, 삭제, 그리고 탐색을 지원하는 링크드 리스트를 보호하는 동기화 메커니즘으로만 시퀀스 락을 사용할 수 있나요?

■

Answer:

이를 이루는 한가지 간단한 방법은 읽기 전용 액세스를 포함한 모든 액세스를 `write_seqlock()` 과 `write_sequnlock()` 으로 감싸는 것입니다. 물론, 이 해결책은 모든 읽기 쪽의 병렬성을 금지시키기도 해서 상당한 락 contention 을 초래할 수 있으며, 이는 간단한 락킹을 사용하는 구현만큼이나 쉬울 겁니다.

여러분이 읽기 쪽 액세스를 보호하기 위해 `read_seqbegin()` 과 `read_seqretry()` 를 사용하는 해결책을 내놓는다면, 여러분이 다음 순서의 이벤트들을 올바르게 처리함을 분명히 해 두십시오:

- CPU 0 이 링크드 리스트를 순회하며, 이 리스트의 원소 A 로의 포인터를 집어듭니다.
- CPU 1 이 이 리스트로부터 원소 A 를 제거하고 메모리 해제합니다.
- CPU 2 가 연관되지 않은 데이터 구조를 메모리 할당하여, 원소 A 에 의해 앞서 사용되었던 메모리를 얻게 됩니다. 이 연관되지 않은 데이터 구조에서, 원소 A 에 의해 `->next` 포인터로 앞서 사용되었던 메모리는 이제 어떤 부동소수점 숫자를 저장합니다.
- CPU 0 이 원소 A 의 `->next` 포인터로 사용되던 것을 집어들어 무작위적 비트들을 갖게 되고, 따라서 `segmentation fault` 를 받습니다.

이런 종류의 문제로부터의 보호는 Section 9.5.4.8 에서 이야기 될 “타입-안전 메모리 (type-safe memory)” 를 필요로 합니다. 대략적으로 비슷한 해결책이 Section 9.3 에서 이야기 한 해저드 포인터를 사용하는 비슷한 해결책도 사용 가능합니다. 그러나 어느 경우든, 여러분은 시퀀스 락에 대해서 어떤 다른 동기화 메커니즘을 사용해야 할 겁니다!

□

Quick Quiz 9.21:

Figure 9.7 는 NULL 포인터를 저장하는데 왜 `smp_store_release()` 를 사용하나요? NULL 포인터 저장에 대해 순서지울 구조체 초기화가 없다는 점을 감안하면 `WRITE_ONCE()` 만으로도 이 경우에는 팬참지 않을까요?

■

Answer:

맞아요, 그럴 겁니다.

NULL 포인터가 할당되고 있을 뿐, 순서지울 것이 존재하지 않으므로, `smp_store_release()` 를 사용할 필요는 없습니다. 대조적으로, NULL 이 아닌 포인터를 할당할 때에는 그 포인터로 가리키지는 구조체의 초기화가 이 포인터의 할당 전에 행해졌음을 분명히 하기 위해 `smp_store_release()` 가 사용되어야 합니다.

짧게 말해서, `WRITE_ONCE()` 는 동작할 것이고, 어떤 아키텍쳐에서는 약간의 CPU 시간을 아낄 겁니다. 하지만, 뒤에서 보게 되겠지만, 소프트웨어 엔지니어링 관점의 걱정은 `smp_store_release()` 와 상당히 유사한 `rcu_assign_pointer()` 라는 특수한 기능의 사용을 장려할 겁니다.

□

Quick Quiz 9.22:

동시에 수행되는 읽기 쓰레드는 Figure 9.7에 그려진 수행 순서에 따르면 gptr의 값에 대해 동의하지 않을 수 있습니다. 이건 뭔가 문제있지 않나요???

**Answer:**

꼭 그렇진 않습니다.

Section 3.2.3 와 3.3에서 힌트가 주어진 것처럼, 빛의 속도의 지연은 컴퓨터의 데이터는 그 데이터가 실제로 모델링 하고자 의도된 바깥의 사실에 비교해서는 항상 오래되어 있습니다.

따라서 실제 세계의 알고리즘은 외부의 현실과 그 현실을 반영하는 컴퓨터 내부 데이터의 비일관성을 감내해야만 합니다. 이런 알고리즘들 여럿은 컴퓨터 내부 데이터에서의 비일관성도 어느정도는 감내할 수 있습니다. Section 10.3.3이 이 점을 더 자세히 다룹니다.

이런 비일관적이고 오래된 데이터를 감내해야 하는 필요성은 RCU에 국한되지 않는다는 점을 알아두시기 바랍니다. 이는 레퍼런스 카운팅, 해저드 포인터, 시퀀스 락, 그리고 심지어 일부 락킹 사용 예에도 적용됩니다. 예를 들어, 여러분이 락을 잡은 채 어떤 값을 계산하지만 그 값을 해당 락을 해제한 후 사용한다면, 여러분은 오래된 데이터를 사용하고 있을 수 있습니다. 어쨌건, 그 값이 기반하고 있는 데이터는 그 락이 해제되는 순간 어떻게든 변할 수도 있습니다.

그러니, 그렇습니다, RCU 읽기 쓰레드는 오래되고 비일관적인 데이터를 볼 수 있습니다. 하지만 아니요, 이게 문제여야만 할 이유는 없어요. 그리고 필요하다면 그런 오래되고 비일관적인 데이터 문제를 막을 수 있는 RCU 사용 패턴도 있습니다 [ACMS03].

**Quick Quiz 9.23:**

Figure 9.8에서, CPU 3의 이 오래된 데이터 항목으로의 액세스를 했을 수 있는 읽기 쓰레드는 이 grace period가 시작하기도 전에 이미 완료되었어요! 그런데 왜 CPU 3의 마지막 컨텍스트 스위치를 신경쓰나요???

**Answer:**

이 대기는 읽기 쓰레드들이 싱글쓰레드 상황에서나 적절한 것과 똑같은 인스트럭션들을 사용할 수 있게 해주기 때문입니다. 달리 말하자면, 이 추가적인 “과잉의” 대기가 읽기 쪽의 훌륭한 성능, 확장성, 그리고 리얼타임 응답을 가능하게 합니다.

**Quick Quiz 9.24:**

Listing 9.13의 `rcu_read_lock()`과 `rcu_read_unlock()`의 요점이 뭔가요? 이 quiescent state 들이 스스로 자신을 이야기하게 하는건 어떤가요?

**Answer:**

읽기 쓰레드들은 quiescent state를 가로지를 수 없음을 기억하세요. 예를 들어, 리눅스 커널에서 RCU 읽기 쓰레드는 컨텍스트 스위치를 수행할 수 없습니다. `rcu_read_lock()`과 `rcu_read_unlock()`의 사용은 올바르지 않게 위치된 quiescent state들을 디버깅 할 수 있게 하고, 그러지 않으면 찾기 어렵고 간헐적으로 나타나며 무척 파괴적인 버그를 찾기 쉽게 해줍니다.

**Quick Quiz 9.25:**

Listing 9.13의 `rcu_dereference()`, `rcu_assign_pointer()`, 그리고 `RCU_INIT_POINTER()`의 요점이 뭔가요? 그냥 `READ_ONCE()`, `smp_store_release()`, 그리고 `WRITE_ONCE()`를 각각 대신 사용하는 건 어떤가요?

**Answer:**

RCU 특정 API들은 제안된 교체와 실제로 비슷한 의미를 가집니다만, RCU 특정 API가 RCU 포인터가 아닌 것들에 대해 호출되었을 때, 그리고 그 거꾸로인 상황에 문제를 제기하는 정적 분석 디버깅 검사를 가능하게 합니다.

**Quick Quiz 9.26:**

하지만 기존 구조체가 메모리 해제되어야 하지만 `ins_route()`의 호출자가 성능에 대한 고려 때문 또는 이 호출자가 RCU 읽기 크리티컬 섹션 내에서 수행되고 있다던지 해서 블록될 수 없다면 어떻게 하죠?

**Answer:**

Section 9.5.2.2에서 이야기되는 `call_rcu()` 함수가 비동기적 grace period 대기를 허용합니다.

**Quick Quiz 9.27:**

Section 9.4의 seqlock 또한 읽기 쓰레드들과 업데이트 쓰레드들이 유용한 동시의 진행을 할 수 있게 하지 않나요?

**Answer:**

그렇기도 하고 아니기도 합니다. Seqlock 읽기 쓰레드들이 seqlock 쓰기 쓰레드들과 동시에 수행될 수 있기는 하지만 이게 일어날 때마다 `read_seqretry()` 기능은

이 읽기 쓰레드가 일을 다시 하게 강제합니다. 이 말은 seqlock 업데이트 쓰레드와 동시에 수행되는 seqlock 읽기 쓰레드에 의해 행해진 모든 일은 폐기되고 재시도에서 다시 수행될 것을 의미합니다. 따라서 seqlock 읽기 쓰레드들은 업데이트 쓰레드들과 동시에 수행될 수 있지만, 이 경우에는 실제 일은 전혀 하지 못합니다.

대비적으로 RCU 읽기 쓰레드들은 동시에의 RCU 업데이트 쓰레드들의 존재에도 불구하고 의미있는 일을 행할 수 있습니다.

하지만, 레퍼런스 카운터와 (Section 9.2) 해저드 포인터는 (Section 9.3) 정말로 의미있는 동시에의 진행을 업데이트 쓰레드와 읽기 쓰레드 모두에게 가능하게 하지만, 어떤 더 큰 비용을 요구할 뿐입니다. 이런 지연된 메모리 회수 문제의 다른 해결책들을 비교하기 위해선 Section 9.6 을 참고하시기 바랍니다.

□

Quick Quiz 9.28:

RCU 업데이트 쓰레드를 위한 데이터 소유권의 사용은 이 업데이트들이 연관된 단일쓰레드 기반 코드의 것과 정확히 같은 명령들을 사용해 수행될 수 있음을 의미하지 않나요?

■

Answer:

어떤 경우들, 예를 들면 x86이나 IBM 메인프레임과 같이 store-release 오퍼레이션이 하나의 스토어 명령만을 만들어내는 TSO 시스템에서라면요. 하지만, 완화된 순서 규칙의 시스템들은 메모리 배리어나 어찌면 store-release 명령을 수행해야 합니다. 또한, 데이터를 제거하는 것은 이 데이터 제거 전에 앞서서부터 존재해온 읽기 쓰레드들을 기다려야 하므로 상당한 추가적 일을 필요로 합니다.

□

Quick Quiz 9.29:

하지만 어떤 읽기 쓰레드가 어떤 링크드 리스트를 순회하는 동안 업데이트 쓰레드들이 여러 데이터 아이템들을 이 리스트에 넣고 빼고 있다고 생각해 봅시다. 특히, 이 리스트가 초기에는 원소 A, B, 그리고 C 를 가지고 있고 어느 업데이트 쓰레드가 원소 A 를 삭제하고 이어서 새 원소 D 를 이 리스트의 끝에 추가했다고 해봅시다. 읽기 쓰레드는 {A, B, C, D} 를 볼 수 있는데, 그런 원소의 연속은 실제로 존재한 적이 없는 것입니다! 어떤 대안적 우주에서는 그게 “동시의 읽기 쓰레드를 방해하지 않는다” 라고 여겨집니까???

■

Answer:

순회하는 읽기 쓰레드가 순회의 전체 기간을 통틀어 존재한 원소들만 순회할 필요가 있는 우주에서요. 이 예에서, 그건 원소 B 와 C 일겁니다. 원소 A 와 D 는

이 순회의 각 부분에서만 존재했으므로, 이 읽기 쓰레드는 그것들을 순회할 수 있게 허용되지만, 강요당하지는 않습니다. 이는 읽기 쓰레드가 그저 단일 항목을 탐색하고 있는, 다른 항목들의 존재나 부재를 신경쓰지 않는 일반적인 경우를 지원함을 알아두시기 바랍니다.

더 강력한 일관성이 필요하다면, 더 높은 비용의 동기화 메커니즘들이 필요한데, 예를 들면 sequence 락킹이나 reader-writer 락킹입니다. 하지만 더 강한 일관성이 필요하지 않다면 (그리고 아주 많은 경우에 그렇습니다), 왜 더 높은 비용을 냅니까?

□

Quick Quiz 9.30:

r1 과 r2 의 어떤 다른 마지막 값이 Figure 9.11에서 가능한가요?

What other final values of r1 and r2 are possible in Figure 9.11? ■

Answer:

$r1 == 0 \&& r2 == 0$ 가능성이 책에서 이야기 되었습니다. $r1 == 0$ 가 $r2 == 0$ 를 의미함을 생각하면, $r1 == 0 \&& r2 == 1$ 은 불가함을 알 수 있습니다. 뒤이은 부분에선 $r1 == 1 \&& r2 == 1$ 과 $r1 == 1 \&& r2 == 0$ 가능함을 보이겠습니다.

□

Quick Quiz 9.31:

Figure 9.12에서 P0() 의 두 액세스가 반대 순서로 이루어지면 어떻게 될까요?

■

Answer:

분명 아무 일도 일어나지 않습니다. P0() 의 x 와 y 의 로드가 같은 RCU 읽기 크리티컬 섹션에서 이루어진다는 것으로 충분합니다; 그것들 사이의 순서는 관계없습니다.

□

Quick Quiz 9.32:

Figures 9.11-9.13 의 P0() 의 액세스들이 스토어였다면 무슨 일이 일어날까요?

■

Answer:

정확히 동일한 순서 규칙이 적용되는데, 즉, (1) 만약 P0() 의 RCU 읽기 크리티컬 섹션이 P1() 의 grace period 의 시작을 앞섰다면, P0() 의 RCU 읽기 크리티컬 섹션의 모든 부분이 P1() 의 grace period 의 끝을 앞서게 되며, (2) 만약 P0() 의 RCU 읽기 크리티컬 섹션의 어떤 부분이 P1() 의 grace period 의 종료를 뒤따랐다면, P0() 의 RCU 읽기 크리티컬 섹션의 모든 부분이 P1() 의 grace period 의 시작을 뒤따릅니다.

쓰기를 포함하는 RCU 읽기 크리티컬 섹션은 좀 이상해 보일 수도 있겠지만, RCU는 그래도 괜찮습니다. 이 능력은 리눅스 커널에서 빈번하게 사용되는데, 예를 들면 데이터 구조로의 레퍼런스나 락을 잡기 위해서입니다. 락이나 레퍼런스를 획득하는 것은 메모리로의 어떤 쓰기를 초래하며, 이걸 RCU 읽기 크리티컬 섹션 내에서 해도 됩니다.

RCU 읽기 크리티컬 섹션이 쓰기를 갖는 것이 여전히 이상해 보인다면, reader-writer 락킹의 읽기 크리티컬 섹션에서의 쓰기 사용 경우를 선보인 Section 5.4.6를 다시 읽어보시기 바랍니다.

□

Quick Quiz 9.33:

리스트의 두개 이상의 버전이 존재할 수 있게 하기 위해 삭제 예제를 어떻게 수정하시겠습니까?

■

Answer:

이를 위한 한가지 방법이 Listing E.2에 보여져 있습니다.

Listing E.2: Concurrent RCU Deletion

```

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 }
```

이는 여러개의 동시 삭제들이 synchronize_rcu()에서 기다리고 있을 수 있음을 의미함을 알아두시기 바랍니다.

□

Quick Quiz 9.34:

리스트의 RCU 버전은 한번에 몇개까지 존재할 수 있나요?

■

Answer:

이는 동기화 설계에 달려 있습니다. 업데이트를 보호하는 세마포어가 grace period 내내 잡혀 있다면, 전과 후, 최대 두개의 버전만이 존재할 수 있습니다.

하지만, 탐색, 업데이트, 그리고 list_replace_rcu() 만이 락으로 보호되어서 Listing E.2에서 보인 코드와 비슷하게 synchronize_rcu() 가 락 바깥에 있다고 가정해 봅시다. 더 나아가서 매우 많은 수의 쓰레드가 거의 동시에 RCU 교체를 수행했으며 읽기 쓰레드들은 지속적으로 데이터 구조를 순회한다고 생각해 봅시다.

그러면 Figure 9.15의 마지막 상태부터 시작해서 다음의 이벤트들이 일어날 수 있습니다:

1. 쓰레드 A가 리스트를 순회하여 원소 C로의 참조를 얻습니다.
2. 쓰레드 B가 원소 C를 새로운 원소 F로 대체하고 자신의 synchronize_rcu() 호출이 리턴하길 기다립니다.
3. 쓰레드 C가 리스트를 순회하여 원소 F로의 참조를 얻습니다.
4. 쓰레드 D가 원소 F를 새로운 원소 G로 대체하고 자신의 synchronize_rcu() 호출이 리턴하길 기다립니다.
5. 쓰레드 E가 리스트를 순회하여 원소 G로의 참조를 얻습니다.
6. 쓰레드 F가 원소 G를 새로운 원소 H로 대체하고 자신의 synchronize_rcu() 호출이 리턴하길 기다립니다.
7. 쓰레드 G가 리스트를 순회하여 원소 H로의 참조를 얻습니다.
8. 그리고 앞의 두 단계가 추가적인 새 원소와 함께 빠르게 반복되어 그 모든 것이 synchronize_rcu() 호출 리턴 전에 일어납니다.

따라서, 임의의 수의 버전이 존재할 수 있는데, 메모리와 하나의 grace period 내에 얼마나 많은 업데이트가 완료될 수 있느냐에만 제한됩니다. 하지만 그렇게 빈번하게 업데이트 되는 데이터 구조는 RCU를 사용하기 좋은 후보가 아닐 가능성이 높음을 알아두시기 바랍니다. 그러나, RCU는 필요할 때에는 높은 업데이트 비율을 처리할 수 있습니다.

□

Quick Quiz 9.35:

rcu_read_lock()도 rcu_read_unlock()도 스펜이나 블록을 하지 않는데 어떻게 RCU 읽기 쓰레드들을 RCU 업데이트 쓰레드가 지연시킬 수 있죠?

■

Answer:

특정 RCU 업데이트 쓰레드에 의해 가해진 수정은 연관된 CPU가 이 데이터를 가지고 있는 캐시 라인을 무효화 시키게 하여, 동시의 RCU 읽기 쓰레드가 수행되고 있는 CPU가 비싼 캐시 미스를 일으키게 만듭니다. (데이터 구조를 동시에 읽기 쓰레드에 비용이 높은 캐시 미스를 일으키지 않으면서 변경시키는 알고리즘을 설계할 수 있겠습니까? 뒤따르는 읽기 쓰레드에게도?)

□

Quick Quiz 9.36:

Table 9.1 의 일부 셀들은 왜 느낌표를 (“!”) 가지고 있나요?

**Answer:**

느낌표를 가지고 있는 API 멤버들 (`rcu_read_lock()`, `rcu_read_unlock()`, 그리고 `call_rcu()`)은 Paul E. McKenney가 90년대 중반에 인지하고 있던 리눅스 RCU API 멤버의 전부였습니다. 이 시간동안, 그는 그가 RCU에 대해 알아야 할 걸 모두 알고 있다는 잘못된 느낌을 가지고 있었습니다.

**Quick Quiz 9.37:**

엄청나게 많은 RCU read-side 크리티컬 섹션이 무한정 `synchronize_rcu()` 호출을 기다리게 하는 것은 어떻게 예방하나요?

**Answer:**

RCU read-side 크리티컬 섹션들이 무한정 `synchronize_rcu()`를 기다리게 하는 걸 막기 위해 어떤 일도 할 필요가 없는데, `synchronize_rcu()`는 앞서서부터 존재해온 RCU read-side 크리티컬 섹션들만을 기다리면 되기 때문입니다. 따라서 각 RCU read-side 크리티컬 섹션이 한정된 길이인 한, RCU grace period 역시 한정적이게 됩니다.

**Quick Quiz 9.38:**

`synchronize_rcu()` API 는 모든 앞서서부터 존재해온 인터럽트 핸들러가 완료되길 기다립니다, 맞죠?

**Answer:**

v4.20 이 후의 리눅스 커널에서는 그렇습니다 [McK19c, McK19a].

하지만 그 전의 커널에서는, 특히 preemption 가능한 RCU 를 사용할 때는 아닙니다! 여러분은 그대신 `synchronize_irq()` 를 원활 겁니다. 대안적으로, 여러분은 여러분이 `synchronize_rcu()` 로 하여금 기다리게 하고 싶은 특정 인터럽트 핸들러 내에 `rcu_read_lock()` 과 `rcu_read_unlock()` 을 넣을 수 있습니다. 하지만 그럴 때에도, preemption 가능한 RCU 는 `rcu_read_lock()` 앞의, 또는 `rcu_read_lock()` 뒤의 부분은 기다린다는 보장이 없으니 조심하세요.

**Quick Quiz 9.39:**

어떤 조건에서 `synchronize_srcu()` 는 Ssrcu read-side 크리티컬 섹션 내에서 안전하게 사용될 수 있나요?

**Answer:**

원칙적으로, 여러분은 특정 `srcu_struct` 를 가지고 다른 `srcu_struct` 를 사용하는 Ssrcu read-side 크리티컬 섹션 내에서 `synchronize_srcu()` 나 `synchronize_srcu_expedited()` 를 사용할 수 있습니다. 하지만, 실제로는 이러한 전 거의 분명 나쁜 생각입니다. 특히, Listing E.3 에 보인 코드는 여전히 데드락을 초래할 수 있습니다.

**Listing E.3: Multistage SRCU Deadlocks**

```

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);

```

Quick Quiz 9.40:

`CONFIG_PREEMPT_NONE=y` 로 빌드된 커널에서라면 preemption 이 불능화 되었고 trampoline 은 직접적으로든 간접적으로든 `schedule()` 을 호출하지 않으니 `synchronize_rcu()` 는 모든 trampoline 을 기다리지 않나요?

**Answer:**

바로 그렇습니다!

실제로, nonpreemptible 커널에서, `synchronize_rcu_tasks()` 는 `synchronize_rcu()` 의 wrapper 입니다.

**Quick Quiz 9.41:**

일반적으로, `rcu_dereference()` 에 넘겨지는 포인터는 항상 Table 9.2 의 포인터 발행 함수 중 하나, 예를 들면 `rcu_assign_pointer()` 를 사용해 업데이트 되어야만 합니다.

이 규칙에서 예외는 무엇이겠습니까?

**Answer:**

그런 예외 중 하나는 여러 원소를 갖는 연결된 데이터 구조가 다른 CPU 들에 의한 액세스가 불가능한 동안 하나의 단위로 초기화 되었고 이어서 이 데이터 구조에 하나의 전역 포인터를 넣기 위해 `rcu_assign_pointer()`

가 사용되었을 때입니다. 이 초기화 시점 포인터 할당은 `rcu_assign_pointer()` 를 사용할 필요가 없습니다, 그런 구조체가 전역적으로 보여질 수 있게 된 후의 그런 할당은 `rcu_assign_pointer()` 를 사용해야만 하지 만말입니다.

하지만, 이 초기화 코드가 상당히 자주 수행되는 코드 경로에 있지 않다면, `rcu_assign_pointer()` 가 이론 상으로 필요치 않더라도 이 함수를 사용하는게 현명합니다. “사소한” 변경이 초기화가 사적으로 일어난다는 여러분의 소중한 가정을 바꿔놓기는 너무 쉽습니다.

□

Quick Quiz 9.42:

이 순회와 업데이트 기능들이 모든 RCU API 집합 멤버들과 사용될 수 있다는 데에 어떤 단점은 없습니까?

■

Answer:

“sparse” 와 같은 자동화된 코드 검사기가 (또는 실제 사람이) 특정 RCU 순회 기능이 어떤 종류의 RCU read-side 크리티컬 섹션에 연관되어 있는지 알기 어렵습니다. 예를 들어, Listing E.4 에 보인 코드를 생각해 봅시다.

Listing E.4: Diverse RCU Read-Side Nesting

```

1 rCU_read_lock();
2 preempt_disable();
3 p = rCU_dereference(global_pointer);
4
5 /* . . . */
6
7 preempt_enable();
8 rCU_read_unlock();

```

여기서의 `rcu_dereference()` 기능은 바닐라 RCU 크리티컬 섹션 안에 있나요, RCU Sched 크리티컬 섹션 안에 있나요? 이걸 알기 위해 여러분은 뭘 해야 합니까?

하지만 v4.20 리눅스 커널에서의 RCU 변종들의 통합 이후부터는 더이상 신경쓸 필요 없습니다!

□

Quick Quiz 9.43:

하지만 `hlist_nulls` 읽기 쓰레드가 다른 버킷으로 갔다가 다시 돌아오면 어떻게 하죠?

■

Answer:

이를 제어하기 위한 한가지 방법은 항상 노드를 목적 버켓의 시작점으로 옮겨서 읽기 쓰레드가 매치 되는 NULL 포인터를 갖는 리스트의 끝에 닿았을 때에는 이 쓰레드가 이 리스트 전체를 탐색했음을 보장하는 것입니다.

물론, 해쉬 테이블에 버킷당 많은 원소가 있고 너무 많은 옮기기 오퍼레이션이 존재한다면 읽기 쓰레드는 영영 리스트의 끝에 도달할 수 없을 수도 있을 겁니다. 평범한 경우에 이를 막는 한가지 방법은 해쉬 테이블을

잘 튜닝되어서 짧은 리스트들만을 갖게 하는 것입니다. 이 문제를 파악하고 처리하는 한가지 방법은 읽기 쓰레드가 어떤 많은 수의 노드를 순회한 다음에는 탐색을 종료하고 `update-side` 락을 잡은 후 탐색을 다시 하는 것이지만 이는 데드락을 초래할 수 있습니다. 이 문제를 완전히 막는 또 다른 방법은 읽기 쓰레드가 RCU read-side 크리티컬 섹션 내에서 탐색을 하고, 이어지는 업데이트들 사이의 RCU grace period 하나를 기다리는 겁니다. 중간의 위치는 어떤 적당한 N 값을 가지고 매 N 업데이트마다 RCU grace period 를 하나 기다릴 겁니다.

□

Quick Quiz 9.44:

Tasks RCU 를 위한 `rcu_read_lock_tasks_held()` 는 왜 존재하지 않나요?

■

Answer:

Tasks RCU 는 read-side 마커가 존재하지 않기 때문입니다. 대신, Tasks RCU read-side 크리티컬 섹션은 자발적 컨택스트 스위치 사이로 제한됩니다.

□

Quick Quiz 9.45:

잠깐요, 뭐라고요??? 어떻게 RCU QSBR 이 이상적인 것보다 나을 수 있죠? 어떤 쓰래기 같은 이상에 대한 정의가 그것이 모든 가능한 결과 중 최선이 아니게 할 수 있죠???

■

Answer:

훌륭한 질문이고, 이에 대한 답은 현대의 CPU 와 컴파일러는 굉장히 복잡하다는 것입니다. 하지만 거기까지 가기 전에, RCU QSBR 의 성능 이득은 코어당 하나의 하드웨어 쓰래드가 제공되는 세계에서만 나타남을 알아둘 가치가 있습니다. 시스템에 부하가 완전히 걸리고 나면, RCU QSBR 의 성능은 이상적인 것의 수준으로 떨어집니다.

RCU 버전의 `route_lookup()` 탐색 반복문은 실제 순차적 버전의 것보다 x86 명령을 하나더 가지는데, 이는 `cmp`, `je`, `mov`, `cmp`, `lea`, and `jne` 순서에서의 `lea`입니다. 이 추가된 명령은 RCU 버전의 `route_entry` 구조체의 시작지점에 있는 `rcu_head` 구조체 때문으로, 이 때문에 순차적 버전과 달리 RCU 버전의 `->re_next.next` 포인터는 0이 아닌 오프셋을 갖습니다. 1980년대로 돌아가 보면, 이 추가적인 `lea` 명령은 RCU 버전이 느려지는 결과를 낼 수도 있었습니다만, 우린 지금 21세기에 있으며, 1980년대는 한참 지났습니다.

하지만 Section 3.1.1 를 주의 깊게 읽은 여러분들은 이미 이 모든 걸 알고 있을 겁니다!

이런 반직관적인 결과는 물론 현대 마이크로프로세서에서의 모든 성능 결과는 약간 회의적으로 보아져야

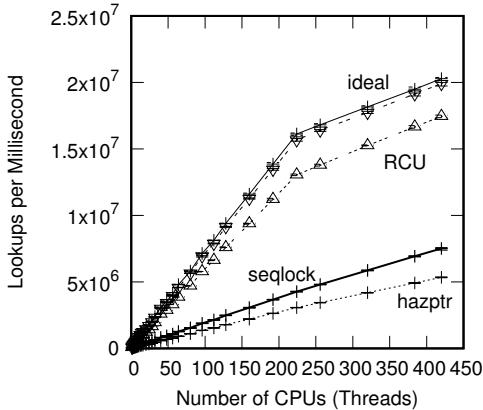


Figure E.5: Pre-BSD Routing Table Protected by RCU QSBR With Non-Initial `rcu_head`

함을 의미합니다. 이론상, 이상적인 것보다 나은 성능을 얻는다는 것은 정말 말이 안됩니다만, 현대의 마이크로 프로세서에서는 정말로 일어날 수 있는 일입니다. 그런 결과는 축복받은 초선형적 성능향상(그런 예 중 하나를 위해 Section 6.5 을 보세요) 비슷한 결로 생각 될 수 있는데, 즉, 흥미롭지만 또한 실용적 중요도는 제한된다는 것입니다. 그러나, RCU 의 강점 중 하나는 그 읽기 쪽 오버헤드가 무척 낮아서 이와 같은 작은 효과가 실제 성능 측정에도 보여질 수 있다는 것입니다.

이는 `rcu_head` 구조체가 `->re_next.next` 포인터가 0 오프셋을 갖게끔 옮겨져서 순차적 버전과 동일하게 되면 어떻게 되는지 질문을 하게 만듭니다. 그에 대한 답은 Figure E.5 에 보여져 있듯, RCU QSBR 의 성능이 여전히 이상적인 것에 근접하지만 이상적인 것을 넘어서지는 않게 한다는 것입니다.

□

Quick Quiz 9.46:

RCU QSBR 의 읽기 성능이 그렇게 좋은데, 왜 다른 userspace 버전을 신경쓰죠?

■

Answer:

RCU QSBR 은 허용될 수 없을 수도 있는 어플리케이션 전체의 제약을 요구하기 때문인데, 예를 들면 어플리케이션 내의 모든 각각의 쓰레드가 정기적으로 quiescent state 를 지나야 한다는 것입니다. 다른 것들 중에서도, 이는 RCU QSBR 이 다른 종류의 userspace RCU [MDJ13c] 에 의해 더 잘 도움받을 수도 있는 라이브러리 작성자에게는 도움이 될 수 있다는 것을 의미합니다.

□

Quick Quiz 9.47:

뭐요? 2.10 GHz 에서의 클락 시간은 약 500 picosecond 인데 RCU 가 300-picosecond 도 안되는 오버헤드를 갖는다는 말을 나더러 믿으라구요?

■

Answer:

우선, 이 측정을 위해 사용된 반복문이 다음과 같음을 고려합시다:

```

1  for (i = nloops; i >= 0; i--) {
2      rCU_read_lock();
3      rCU_read_unlock();
4  }
```

다음으로, 실질적인 `rcu_read_lock()` 과 `rcu_read_unlock()` 의 정의를 생각해 봅시다:

```

1  #define rCU_read_lock()    barrier()
2  #define rCU_read_unlock()  barrier()
```

이 정의들은 메모리 참조에 연관되는 컴파일러의 코드 이동 최적화를 제약하지만, 그것들 자체 내에는 어떤 명령도 넣지 않습니다. 하지만, 이 반복문 변수가 레지스터에 관리된다면, i 로의 액세스는 메모리 참조로 취급되지 않습니다. 더욱이, 컴파일러는 loop unrolling 을 할 수 있어, 결과적인 코드가 단순히 i 값을 1보다 큰 어떤 값으로 증가시키는 것으로 여러 횟수의 반복문 수행을 “해내는” 코드를 만들 수 있습니다.

따라서 267 picoseconds 의 “측정”은 시간 측정을 `rcu_read_lock()` 과 `rcu_read_unlock()` 호출을 포함하는 이 내부 반복문을 통과하는 반복 횟수로 나눈 것에 이 loop-unrolling 으로 i 조정 코드를 나눈 것을 더한 고정된 오버헤드입니다. 그리고 그래서, 이 측정은 실제로 어려를 포함하고 있으며, 실제로 오버헤드를 수십 수백배로 부풀려서 이야기 하고 있습니다. 어쨌건, 만들어진 기계 명령어의 관점에서, `rcu_read_lock()` 과 `rcu_read_unlock()` 의 오버헤드는 정확히 제로입니다.

267 picosecond 의 측정된 시간이 과장된 수치임이 드러나는 건 매일 있는 일은 분명 아닙니다!

□

Quick Quiz 9.48:

이 책의 초기 버전에서는 RCU 읽기 오버헤드가 1 picosecond 미만이라 하지 않았나요? 무슨 일이 있었던 거죠???

■

Answer:

훌륭한 기억력이군요!!! 어떤 초기 버전에서의 오버헤드는 실제로 약 100 femtosecond 이었습니다.

그사이 무슨 일이 있었느냐면, RCU 사용처가 리눅스 커널 내에 더욱 널리 퍼졌는데, page fault 처리 코드도 여기 포함됩니다. 그때로 돌아가면, `rcu_read_lock()` 과 `rcu_read_unlock()` 은 `CONFIG_PREEMPT=n` 커널에서 완전히 아무 일도 하지 않았습니다. 불행히도, 그 상황은 컴파일러가 page fault 되는 메모리 액세스를 RCU read-side 크리티컬 섹션 새로 재배치 할 수 있게 했습니다. 물론, page fault 는 블록될 수 있으며, 따라서 이 크리티컬 섹션을 망가지게 합니다.

이건 이론적 문제만이 아니었습니다: 그로 인한 문제가 실제로 2019년에 이야기 되었습니다. Herber Xu 는 이 문제를 분석했고 Linus Torvalds 는 `rcu_read_lock()` 과 `rcu_read_unlock()` 이 무조건적으로 `barrier()` 호출을 포함하게 하는 커밋을 대기열에 올렸습니다 [Tor19]. 그리고 `barrier()` 가 코드를 추가하진 않지만, 컴파일러 최적화를 제한합니다. 따라서 널리 퍼져있는 RCU 사용처의 비용은 `rcu_read_lock()` 과 `rcu_read_unlock()` 오버헤드보다 약간 더 높아집니다.

물론, 더 오래전의 결과는 Figure 9.23 에서 보인 것과 다른 시스템에서 얻어진 겁니다. 그래서 뭐가 더 영향을 끼쳤을까요, Linus 의 커밋 또는 시스템 변경? 이 질문은 독자 여러분의 연습문제로 남겨둡니다.

□

Quick Quiz 9.49:

Figure 9.23 의 rCU 에는 왜 그리 큰 편차가 존재하는 거죠?

■

Answer:

이는 log-log 그림이며, 따라서 크게 보이는 rCU 편차는 실제로는 수백 picosecond 에 불과함을 명심하세요. 그리고 그건 무엇이든 그것을 일으킬 수 있는 무척 짧은 시간입니다. 하지만, 편차가 CPU 수가 작은 크든 감소된다는 걸로 보아, 한가지 가능한 가설은 이 편차는 하나의 CPU 에서 다른 CPU 로의 migration 때문이라는 것입니다.

그래요, 이 측정은 인터럽트가 불능화 된 채로 취해졌습니다만, 게스트 OS 내에서 취해졌으므로, 하이퍼바이저 단계에서의 preemption 은 가능합니다. 이 게스트 OS 들을 리얼타임 우선순위로 수행함으로써 이런 편차를 줄이려는 노력은 (Joel Fernandes 에 의해 제안되었습니다) 독자 여러분의 연습문제로 남겨두겠습니다.

□

Quick Quiz 9.50:

시스템은 448 개의 하드웨어 쓰레드를 갖는데, 왜 192 CPU 까지만 측정하나요?

■

Answer:

이 데이터를 생성하는데 사용된 스크립트 (`rcusclae.sh`) 는 모아지는 지점들의 각 집합을 위해 게스트 운영체제를 시작하며, 이 특정 시스템에서는 qemu 와 KVM 이 모두 특정 게스트 OS 에 설정될 수 있는 CPU 갯수에 제한을 두기 때문입니다. 그래요, 조금 더 많은 CPU 를 사용하는 것도 가능하겠습니다만 256 은 불가능하다는 점을 생각할 때 이진수 관점에서 192 는 괜찮은 어림수입니다.

□

Quick Quiz 9.51:

Figure 9.25 의 마이크로세컨드 미만 기간에서의 더 큰 에러 범위는 왜 그런거죠?

■

Answer:

더 작은 거리는 더 작은 측정에서 더 큰 상대적 에러를 초래하기 때문입니다. 또한, 리눅스 커널의 `ndelay()` 나 노세컨드 단위 기능은 (2020년 기준으로) 마이크로세컨드 이상의 시간단위 데이터를 위해 사용되는 `udelay()` 보다 덜 정확합니다. Figure 9.23 에 보인 0 의 길이의 경우와 비교해 보는게 유익할 겁니다.

□

Quick Quiz 9.52:

이 데드락 내성에 예외가 있나요? 만약 그렇다면 어떤 이벤트들의 연속이 데드락을 이르게 할 수 있나요?

■

Answer:

RCU read-side 기능이 관여된 데드락 사이클을 만드는 한가지 방법은 다음의 (불법적인) 선언문의 연속입니다:

```
rcu_read_lock();
synchronize_rcu();
rcu_read_unlock();
```

이 `synchronize_rcu()` 는 앞서서부터 존재한 모든 RCU read-side 크리티컬 섹션이 완료되기 전까지 리턴할 수 없지만, `synchronize_rcu()` 가 리턴하기 전까지는 완료될 수 없는 RCU read-side 크리티컬 섹션에 감싸여 있습니다. 그 결과는 고전적인 스스로에 대한 데드락입니다—reader-writer 락의 경우에도 읽기 락을 쥐고 있는 상태에서 쓰기 락을 쥐려고 시도하면 똑같은 결과를 얻을 겁니다.

이 스스로에 대한 데드락 시나리오는 RCU QSBR 에 적용되지 않는데, `synchronize_rcu()` 에 의해 수행되는 컨텍스트 스위치는 이 CPU 를 위한 quiescent state 로 동작하여 grace period 가 종료되게 할 것이기 때문임을 알아두시기 바랍니다. 하지만, 이는 심지어 더 나쁜

것일 뿐일텐데, 이 RCU read-side 크리티컬 섹션에 의해 사용되는 데이터는 이 grace period 완료로 인해 메모리 해제되었을 수도 있기 때문입니다.

요약하자면, Table 9.1에 나열된 synchronous RCU update-side 기능을 RCU read-side 크리티컬 섹션 내에서 사용하지 마십시오.



Quick Quiz 9.53:

데드락과 우선순위 역전에 모두 내성이 있다??? 사실이 라기엔 너무 좋아보이는데요. 이게 가능하다는 걸 제가 왜 믿어야 하죠?



Answer:

진짜 동작합니다. 어쨌건, 그렇지 않다면 리눅스 커널은 동작하지 못할겁니다.



Quick Quiz 9.54:

하지만 잠시만요! 이건 RCU 를 reader-writer 락킹의 대체제로 생각할 때 사용될 수 있는 것과 완전히 똑같은 코드예요! 뭐죠?



Answer:

이건 장난감 예제의 법칙의 효과입니다: 어떤 지점 이후부터는 코드 조각들은 똑같이 보입니다. 유일한 차이점은 우리가 그 코드를 어떻게 생각하느냐에 있습니다. 하지만, 이 차이점은 무척 중요할 수 있습니다. 그 중요성에 대해 한가지만 예를 들어보자면, 우리가 RCU 를 제한된 레퍼런스 카운팅 방법으로 생각한다면, 우린 업데이트가 RCU read-side 크리티컬 섹션을 배제시킬 거라는 바보스러운 생각에 빠지진 않을 겁니다.

그러나 RCU 를 reader-writer 락킹의 대체제로 생각하는 것도 종종 유용한데, 예를 들면 reader-writer 락킹을 RCU 로 대체할 때 그렇습니다.



Quick Quiz 9.55:

우리가 제거하려는 원소가 Listing 9.19 의 라인 9 의 리스트의 첫번째 원소가 아니면 어떻게 되죠?



Answer:

Listing 7.9에서와 같이, 이것은 체이닝이 없는 매우 간단한 해쉬 테이블이며, 따라서 특정 버킷의 유일한 원소는 첫번째 원소입니다. 독자 여러분은 다시 한번 이 예를 완전한 체이닝을 갖는 해쉬 테이블로 조정해 보실 수 있겠습니다. 그럴 기력이 없는 독자 분들은 Chapter 10 을 참고하고 싶으실 수도 있겠네요.



Quick Quiz 9.56:

Listing 9.19 의 라인 15 에서는 라인17에서 락을 내려놓기 전에 RCU read-side 크리티컬 섹션을 빠져나가는게 왜 괜찮나요?



Answer:

첫째로, 라인 14에서의 두번째 검사는 어떤 다른 CPU 가 이 원소를 우리가 락을 획득하여 기다리는 사이에 제거했을지도 모르므로 필요함을 알아 두시기 바랍니다. 그러나, 우리가 이 락을 획득하는 사이 RCU read-side 크리티컬 섹션 내에 있었다는 사실은 이 원소가 재할당되고 이 해쉬 테이블에 재삽입되었을 가능성이 없음을 보장합니다. 더 나아가, 일단 우리가 이 락을 획득했다면, 그 락 자체가 이 원소의 존재를 보장하므로, 우린 더이상 이 RCU read-side 크리티컬 섹션 내에 있을 필요가 없습니다.

이 원소의 키를 재검사 할 필요가 있는가 하는 질문은 독자 여러분의 연습문제로 남겨둡니다.



Quick Quiz 9.57:

Listing 9.19 의 라인 23 에서는 왜 라인 22에서 락을 내려놓기 전에 RCU read-side 크리티컬 섹션을 빠져나가지 않나요?



Answer:

우리가 이 두 라인의 순서를 바꾼다고 생각해 봅시다. 그러면 이 코드는 다음 순서의 이벤트에 취약해 집니다:

1. CPU 0 가 `delete()` 를 호출하고, 삭제될 원소를 찾은 후, 라인 15를 수행합니다. 아직 원소를 삭제하지 않았지만, 곧 할겁니다.
2. CPU 1 이 동시에 `delete()` 를 호출해, 이 동일한 원소를 제거하려 합니다. 하지만, CPU 0 는 여전히 락을 잡고 있으므로, CPU 1 은 라인 13에서 그걸 기다립니다.
3. CPU 0 이 라인 16 와 17 을 수행하며, 라인 18에서 CPU 1 이 RCU read-side 크리티컬 섹션에서 나오길 기다립니다.
4. CPU 1 이 이제 이 락을 잡지만, 라인 14에서의 테스트는 CPU 0 이 이미 이 원소를 제거했으므로 실패합니다. CPU 1 은 이제 (이 quick quiz 를 위해 우리가 라인 23 과 교체한) 라인 22 를 수행하고 자신의 RCU read-side 크리티컬 섹션을 빠져나갑니다.
5. CPU 0 은 이제 `synchronize_rcu()` 로부터 리턴 할 수 있고, 따라서 라인 19 를 수행해 이 원소를 freelist 로 보냅니다.

6. CPU 1 이 이제 메모리 해제된, 어쩌면 이제 다른 종류의 데이터 구조체로 이미 재할당 되었을 수도 있는 원소를 위한 락을 내려놓으려 합니다. 이는 심각한 메모리 오염 에러입니다.

□

Quick Quiz 9.58:

하지만 여러 쓰레드에 임의의 긴 RCU read-side 크리티컬 섹션들이 존재하여 어느 시점에서든 시스템 내에 최소 하나의 RCU read-side 크리티컬 섹션을 수행하는 쓰레드가 존재하면 어떻게 되죠? 그건 SLAB_TYPESAFE_BY_RCU slab의 어떤 데이터든 시스템에 반환되는 것을 막아서 OOM 상황에 이르지 않을까요?

■

Answer:

최소 하나의 쓰레드가 RCU read-side 크리티컬 섹션에 있는 무한정 긴 시간이 분명 존재할 수 있습니다. 하지만, Section 9.5.4.8의 설명 중 핵심 단어는 “사용중” 그리고 “앞서서부터 존재한”입니다. 특정 RCU read-side 크리티컬 섹션은 이론상 그 크리티컬 섹션의 시작 시점에 사용 중이던 데이터 원소로의 참조만을 얻을 수 있습니다. 더 나아가, slab은 그 데이터 원소가 모두 메모리 해제되기 전까지는 시스템에 반환될 수 없으며, 실제로 RCU grace period는 그것들이 모두 메모리 해제되기 전까지는 시작될 수 없음을 기억하십시오.

따라서, slab cache는 해당 slab의 마지막 원소가 메모리 해제되기 시작된 RCU read-side 크리티컬 섹션들만을 기다려야 합니다. 이는 결국 이 마지막 원소가 메모리 해제된 후 시작된 모든 RCU grace period만 카운트 됨을 의미합니다—이 slab은 그 grace period가 종료된 후 시스템으로 반환될 수 있을 겁니다.

□

Quick Quiz 9.59:

nmi_profile() 함수가 preemption 가능했다고 쳐 봅시다. 이 예제가 올바르게 동작하려면 무엇이 바뀌어야 할까요?

■

Answer:

한가지 방법은 nmi_profile()에서 rCU_read_lock()과 rCU_read_unlock()을 사용하고 synchronize_sched()를 synchronize_rcu()로 교체하는 것으로, Listing E.5에 보이는 것과 비슷합니다.

□

Listing E.5: Using RCU to Wait for Mythical Preemptible NMIs to Finish

```

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 }

```

Quick Quiz 9.60:

그냥 grace period를 기다리기 전에 락을 놓거나 grace period를 기다리는 대신 call_rcu() 같은 걸 사용하지 않죠?

■

Answer:

저자들은 linearizable 한 tree 오퍼레이션을 지원해서 동시적인 항목 추가, 제거, 그리고 탐색이 어떤 전역적인 합의된 순서로 수행되는 것처럼 보이길 바랬습니다. 그들의 탐색 tree에서, 이는 grace period에 걸쳐 락을 잡을 것을 필요로 했습니다. (대부분의 경우 linearizability를 요구사항에서 제거하는게 아마 더 낫겠지만, linearizability는 놀랍도록 대중적인 (그리고 비싼!) 요구사항입니다.)

□

Quick Quiz 9.61:

Listing 5.4 (count_end.c)에 보인 통계적 카운터 구현은 read_count()의 합산을 보호하기 위해 전역적 락을 사용했는데, 이는 나쁜 성능과 음의 확장성으로 이어졌습니다. 여러분은 read_count()에 훌륭한 성능과 좋은 확장성을 제공하기 위해 RCU를 어떻게 사용할 수

있겠습니다. (`read_count()`의 확장성은 모든 쓰레드의 카운터를 스캔해야 한다는 필요로 인해 제한되어 할 수 있음을 명심하십시오.)

Answer:

힌트: 전역 변수 `finalcount` 와 배열 `counterp[]` 를 하나의 RCU 로 보호되는 구조체에 위치시키십시오. 초기화 시점에, 이 구조체는 할당되고 0 과 NULL 로 설정됩니다.

`inc_count()` 함수는 변경되지 않습니다.

`read_count()` 함수는 `final_mutex` 를 잡는 대신 `rcu_read_lock()` 을 사용하며, 현재 구조체로의 참조를 얻기 위해 `rcu_dereference()` 를 사용해야 할 겁니다.

`count_register_thread()` 함수는 새로 생성된 쓰레드의 per-thread counter 변수를 참조하기 위해 그 쓰레드에 연관된 배열 내 원소의 값을 설정할 겁니다.

`count_unregister_thread()` 함수는 새 구조체를 할당하고, `final_mutex` 를 잡고, 기존 구조체의 값을 새 것으로 복사하고, 끝나는 쓰레드의 counter 변수를 total 로 더하고, 이 같은 counter 변수로의 포인터를 NULL 값으로 쓰고, 새 구조체로 기존 것을 교체하기 위해 `rcu_assign_pointer()` 를 사용하고, `final_mutex` 를 놓고, 하나의 grace period 를 기다린 후, 마침내 기존 구조체를 메모리 해제할 겁니다.

이게 정말 잘 동작할까요? 왜 일까요 또는 왜 아닐까요?

더 자세한 내용을 위해선 page 265 의 Section 13.5.1 을 보시기 바랍니다.

□

Quick Quiz 9.62:

Section 5.4.6 은 제거될 수 있는 기기로의 I/O 액세스를 세기 위한 두 쌍의 코드 조각을 보였습니다. 이 코드 조각들은 reader-writer 락을 잡아야 하는 이유로 (I/O 를 시작하는) fastpath 에서의 높은 오버헤드로 고생이 심했습니다. 여러분은 훌륭한 성능과 확장성을 제공하기 위해 RCU 를 어떻게 사용하겠습니까? (I/O 액세스를 하는 일반적인 경우를 위한 코드 조각의 성능은 기기 제거 코드 조각의 것보다 훨씬 중요함을 명심하십시오.)

■

Answer:

힌트: reader-writer 락의 read-잡기를 RCU read-side 크리티컬 섹션으로 대체하고, 기기 제거 코드를 그에 맞게 변경하십시오.

이 문제를 위한 한가지 해결책을 위해 page 267 의 Section 13.5.2 을 보십시오.

□

Quick Quiz 9.63:

사용자들은 왜 필요에 따라 동적으로 해저드 포인터를 할당할 수 없나요?

■

Answer:

그럴 수 있습니다만, 메모리 할당 실패를 처리하기 위해 어떤 환경에서는 읽기 쓰레드의 순회에 대한 추가적 오버헤드를 비용으로 지불합니다.

□

Quick Quiz 9.64:

하지만 리눅스 커널의 `kref` 레퍼런스 카운터는 보장된 무조건적 참조 획득을 허용하지 않나요?

■

Answer:

맞아요 그렇습니다, 하지만 그 보장은 참조가 이미 잡혀 있는 경우에만 무조건적으로 적용됩니다. 이를 명심하고, Section 9.6 의 시작 부분의 문장, 특히 “충분히 길어서 읽기 쓰레드가 순회 사이에 참조를 잡고 있지 않는” 부분을 다시 보시기 바랍니다.

□

Quick Quiz 9.65:

하지만 Section 9.3 의 quick quiz 들의 답 중 하나는 꽉을 이룬 비대칭적 배리어로 해저드 포인터의 읽기 쪽 `smp_mb()` 를 제거할 수 있다고 하지 않았던가요?

■

Answer:

맞아요, 그랬습니다. 하지만, 그렇게 하는 것은 해저드 포인터의 “Reclamation Forward Progress” 열 (나중에 설명합니다) 을 lock-free 에서 blocking 으로 바꿀 수 있는데, 커널 내에서 인터럽트를 불능화 시킨 채 spin 하고 있는 CPU 는 업데이트 쪽 비대칭 배리어를 완료되지 못하게 할 수 있기 때문입니다. 리눅스 커널에서, 그런 블록킹은 커널을 `CONFIG_NO_HZ_FULL` 와 함께 빌드하고 부팅 시점에서 연관된 CPU 들을 `nohz_full` 로 지정하고, 한번에 단 하나의 쓰레드만이 하나의 CPU 에서 수행될 수 있음을 보장하며, 커널 내로 들어가는 것을 막음으로써 방지할 수 있습니다. 대안적으로, 여러분은 커널이 CPU 들이 인터럽트가 불능화 된 채 spin 하게 될 수도 있는 어떤 버그로부터도 그 커널이 자유로움을 보장할 수 있을 겁니다.

리눅스 커널에서 인터럽트를 불능화 한 채 spin 하는 CPU 가 드물어 보임을 놓고 보면, 어떤 사람은 비대칭적 배리어 기반 해저드 포인터 업데이트가 이론적으로는 그렇지 않지만 실질적으로는 non-blocking 이라고 주장 할 수도 있겠습니다.

□

E.10 Data Structures

Quick Quiz 10.1:

하지만 체인으로 연결된 해쉬 테이블은 많은 종류 중 하나입니다. 왜 체인으로 연결된 해쉬 테이블에 집중하죠?



Answer:

체인으로 연결된 해쉬 테이블은 완벽히 파티셔닝 가능하며 따라서 동시성 사용에 잘 맞습니다. 예를 들어 split-ordered list [SS06] 같은 다른 완벽하게 파티셔닝 가능한 해쉬 테이블도 존재하지만, 상당히 더 복잡합니다. 그러므로 우리는 체인으로 연결된 해쉬 테이블에서 시작하겠습니다.



Quick Quiz 10.2:

하지만 Listing 10.3의 라인 10–13에서의 두번의 비교는 키가 `unsigned long`에 들어갈 때에는 비효율적이지 않나요?



Answer:

실제로 그렇습니다! 그렇지만, 해쉬 테이블은 상당히 자주 `unsigned long`에 들어맞지는 않는 문자열과 같은 키를 사용해 정보를 저장합니다. 이 해쉬 테이블 구현을 키가 항상 `unsigned long`에 들어가는 경우를 위해 단순화하는 것은 독자 여러분의 연습으로 남겨두겠습니다.



Quick Quiz 10.3:

단순히 해쉬 버킷의 수를 늘리는 대신, 존재하는 해쉬 버킷을 캐싱 열라인 시키는게 낫지 않을까요?



Answer:

그에 대한 답은 많은 것에 의존적입니다. 이 해쉬 테이블이 버킷마다 많은 수의 원소를 가지고 있다면, 해쉬 버킷의 갯수를 늘리는게 분명 나을 겁니다. 반면에, 이 해쉬 테이블이 큰 로드를 가지고 있지 않다면, 그 답은 하드웨어, 해쉬 함수의 효율성, 그리고 워크로드에 종속적일 겁니다. 흥미 있는 독자 여러분은 실험해 보시기 바랍니다.



Quick Quiz 10.4:

소켓을 가로지르는 경우의 Schrödinger의 동물원의 나쁜 확장성을 생각하면, 이 어플리케이션의 여러 복사본을 각 복사본은 동물들의 부분집합만을 가지고 각 복사본을 단일 소켓으로 가둬서 수행시키는 건 어떤가요?



Answer:

그럴 수 있습니다! 실제로, 여러분은 이 아이디어를 이 어플리케이션의 한 복사본을 클러스터의 각 노드에서 수행시키는 커다란 클러스터 시스템으로 확장시킬 수 있습니다. 이 방법은 “sharding”이라 불리며, 커다란 웹 기반 상점들에서 많이 사용되고 있습니다 [DHJ⁺07].

하지만, 여러분이 멀티소켓 시스템에서 소켓별 방식으로 sharding을 할거라면, 분리된 더 작고 저렴한 싱글 소켓 시스템을 여럿 사서 이 데이터 베이스의 한 shard를 각 시스템에서 돌리는게 어떻습니까?



Quick Quiz 10.5:

하지만 해쉬 테이블 내의 원소들이 탐색과 동시에 제거될 수 있다는 것은, 탐색이 찾아진 직후 제거된 데이터 원소로의 참조를 리턴할 수 있다는 의미 아닌가요?



Answer:

맞아요 그럴 수 있습니다! 이게 왜 `hashtab_lookup()`이 RCU read-side 크리티컬 섹션 내에서 호출되어야 하는지에 대한 이유이며, `hashtab_add()`와 `hashtab_del()`이 RCU를 인지하는 리스트 조정 기능을 써야만 하는 이유입니다. 마지막으로, 이는 왜 `hashtab_del()` 호출자가 제거한 원소를 메모리 해제하기 전에 하나의 grace period를 기다려야만 (예: `synchronize_rcu()`를 호출함으로써) 하는 이유입니다. 이는 모든 RCU 읽기 쓰레드가 제거된 원소들을 메모리 해제되기 전에 참조함을 보장할 겁니다.



Quick Quiz 10.6:

Page 178의 Figure 10.4가 해쉬 테이블 크기를 변화시키는게 거의 아무 효과도 내지 못하는 걸 보이는데 해쉬 테이블 크기가 문제라고 여기서 확신할 수 있나요? 문제는 그게 아니라 false sharing 일수도 있지 않을까요?



Answer:

훌륭한 질문입니다!

False sharing은 쓰기를 필요로 하는데, 이 탐색만하는 벤치마크의 동기화되지 않은 수행이나 RCU 수행에서는 사용되지 않습니다. 따라서 문제는 false sharing이 아닙니다.

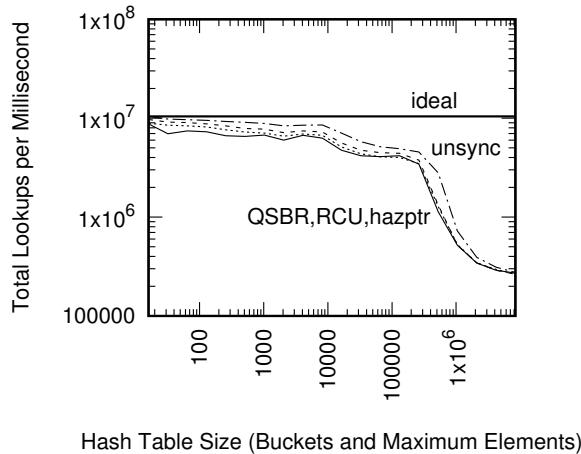


Figure E.6: Read-Only RCU-Protected Hash-Table Performance For Schrödinger's Zoo at 448 CPUs, Varying Table Size

아직 만족스럽지 않나요? 그렇다면 해쉬 테이블 크기, 즉 버킷의 수와 원소의 최대 개수에 대한 함수로써의 448 CPU 성능을 log-log 그림으로 보이는 Figure E.6를 보시기 바랍니다. 크기 1,024의 해쉬 테이블은 1,024 버킷을 가지고 최대 1,024개 원소를 가지며, 평균적으로는 512개 원소를 갖습니다. 이것은 읽기만 하는 벤치마크이므로, 실제 사용량은 항상 평균 사용량과 같습니다.

이 그림은 8,000개 원소 미만에서 거의 이상적인 성능을 보이는데, 즉 해쉬 테이블이 1 MB 미만의 데이터를 가질 때입니다. 이 거의 이상적인 성능은 page 157 의 Figure 9.21에서 보인 pre-BSD 라우팅 테이블에 대해선 심지어 448 CPU에서도 일관적입니다. 그러나, 이 성능은 8,000개 원소에서 상당히 떨어지는데 (이건 log-log 그림입니다), 1,048,576-바이트 L2 캐쉬가 넘쳐날 때입니다. 300,000개 원소에서 성능은 절벽 아래로 떨어지는데 (이 log-log 그림에서 조차), 40,370,176-바이트 L3 캐쉬가 넘치는 지점입니다. 이는 메모리 시스템 병목이 지대하며 성능을 큰 채쉬 테이블의 경우 수십 수백배 넘게 떨어뜨릴 수 있음을 보입니다. 8,338,608 크기의 해쉬 테이블은 1 GB 메모리를 사용하여 L3 캐쉬의 25배 이상이므로 놀라운 일이 아닙니다.

Page 178의 Figure 10.4가 작은 영향을 보이는 이유는 그 데이터가 버킷별로 락킹 되는 해쉬 테이블에서 얻어졌기 때문으로, 락킹 오버헤드와 락킹 경쟁이 캐쉬 용량 효과를 넘어서기 때문입니다. 대조적으로 RCU 와 해저드 포인터 읽기 쓰레드들은 모두 공유된 데이터로의 쓰기를 하지 않아서 캐쉬 용량 효과가 전면에 드러남을 의미합니다.

여전히 만족스럽지 않으신가요? 멀티 소켓 시스템을 찾아서 이 코드를 돌리면서 뭐가 됐든 가능한

performance-counter 하드웨어를 사용하세요. 그 하드웨어는 카운터는 여러분이 특정 시스템에서 나타나는 모든 속도 저하의 자세한 원인을 찾을 수 있게 할 겁니다. 이 연습을 함으로써 얻는 경험은 굉장히 가치 있어서 여러분에게 이 문제를 이론적이라고 생각하는 사람들에 비해 엄청난 이득을 가져다 줄 겁니다.¹¹

□

Quick Quiz 10.7:

이 커다란 시스템에서는 메모리 시스템이 상당한 병목입니다. 유용한 뭔가를 하기 충분한 메모리 대역폭을 주지 않고서 시스템에 448 CPU를 장착해서는 왜 고생할까요?

■

Answer:

이 커다랗고 비싼 시스템을 작은 데이터 원소들로 구성된 간단한 해쉬 테이블 탐색에만 사용하는 것은 실제로 나쁜 생각일 겁니다. 하지만, 이 시스템은 더 많은 계산과 적은 메모리 액세스를 사용하는 무척 많은 워크로드에 매우 유용합니다. 예를 들어, 어떤 인메모리 데이터베이스는 이런 종류의 시스템에서 무척 잘 동작하는데, 이 챕터의 벤치마크에서 수행하는 것에 비해 훨씬 복잡한 쿼리 집합을 사용할 때 뿐이긴 합니다. 예를 들어, 그런 시스템은 각 원소에 저장된 이미지나 비디오를 처리해서 그로 인해 생겨나는 순차적 메모리 액세스가 가능한 메모리 대역폭의 더 나은 사용을 하여서 순전히 포인터 따라가기만 하는 워크로드에 비해 추가적 성능 이득을 얻을 겁니다.

하지만 이걸 여러분의 교훈으로 삼기 바랍니다. 현대의 컴퓨터 시스템은 매우 다양한 형태와 크기로 나오며, 여러분의 어플리케이션에 맞는 것을 고르는데는 상당한 주의가 종종 필요합니다. 그리고 여러분의 어플리케이션을 특정 컴퓨터 시스템에 맞춰 조정하는데는 아마 그보다도 자주 상당한 주의와 작업이 필요합니다.

□

Quick Quiz 10.8:

28 CPU에서의 결과를 448 CPU로 외삽하는 것의 위험성은 Section 10.2.3에서 분명히 보여졌습니다. 하지만 왜 448 CPU에서의 결과로 그 이상을 외삽하는 것은 안전한가요?

■

Answer:

이론상 이는 결코 안전하지 않으며 유용한 행위는 이 프로그램을 더 큰 시스템에서 돌려보는 것일 겁니다. 실용적으로는, 448 CPU 이상의 시스템보다 28 CPU 이상의 것이 훨씬 많습니다. 더해서, 다른 테스트가 RCU

¹¹ 물론, 이론적 이해는 이해하지 못함보다 낫습니다.

read-side 기능들이 1024 CPU 까지의 일관적인 성능과 확장성을 보임을 보인 바 있습니다.



Quick Quiz 10.9:

Listing 10.10 의 코드는 크기 재조정 프로세스가 선택된 버킷을 지나가는 걸 어떻게 보호합니까?



Answer:

그런 보호는 제공하지 않습니다. 그건 이어서 설명될 업데이트 쪽 동시성 제어 기능의 역할입니다.



Quick Quiz 10.10:

한 쓰레드가 크기 재조정 오퍼레이션 중에 이 해쉬 테이블에 원소를 하나 넣는다고 생각해 봅시다. 뒤따르는 크기 재조정 오퍼레이션이 이 삽입 전에 완료됨으로 인해 이 삽입이 사라지는 것은 어떻게 방지됩니까?



Answer:

이 두번째 크기 재조정 오퍼레이션은 이 버킷을 이 삽입이 위치하는 곳으로 옮길 수 없을텐데 이 삽입은 이 해쉬 테이블의 해쉬 버킷 하나 또는 둘 모두의 락을 잡을 것이기 때문입니다. 더 나아가, 이 삽입 오퍼레이션은 RCU read-side 크리티컬 섹션 내에 위치해 있습니다. `hashtab_resize()` 함수를 살펴 볼 때 보게 되겠지만, 이는 각 크기 재조정 오퍼레이션이 이 삽입의 read-side 크리티컬 섹션이 완료되길 기다리기 위해 `synchronize_rcu()` 를 호출함을 의미합니다.



Quick Quiz 10.11:

Listing 10.12 의 `hashtab_lookup()` 함수는 동시의 크기 재조정 오퍼레이션을 무시합니다. 이는 읽기 쓰레드가 크기 재조정 오퍼레이션 중에 추가된 원소를 놓칠 수도 있음을 의미하지 않습니까?



Answer:

아니요. 곧 보게 되겠지만, `hashtab_add()` 와 `hashtab_del()` 함수는 기존 해쉬 테이블을 크기 재조정 오퍼레이션이 진행 중인 동안 최신 상태로 유지합니다.



Quick Quiz 10.12:

Listing 10.12 의 `hashtab_add()` 와 `hashtab_del()` 함수는 크기 재조정 오퍼레이션이 진행 중인 사이에 두 개의 해쉬 버킷을 업데이트 할 수 있습니다. 이는 크기 재조정 오퍼레이션의 빈도가 무시할 만한 수준이 아니라면 성능 저하를 일으킬 수도 있을 겁니다. 그런 경우에 업데이트의 비용을 줄일 수 있지 않습니까?



Answer:

최소한 `hashtab_lookup()` 비용의 약간의 상승이 받아들여질 수 있다면 그렇습니다. 그런 접근법이 Listings E.6 과 E.7 (`hash_resize_s.c`) 에 보여져 있습니다.

Listing E.6: Resizable Hash-Table Access Functions (Fewer Updates)

```

1 struct ht_elem *
2 hashtab_lookup(struct hashtab *htp_master, void *key)
3 {
4     struct ht *htp;
5     struct ht_elem *htep;
6
7     htp = rcu_dereference(htp_master->ht_cur);
8     htp = ht_search_bucket(htp, key);
9     if (htep)
10         return htep;
11     htp = rcu_dereference(htp->ht_new);
12     if (!htp)
13         return NULL;
14     return ht_search_bucket(htp, key);
15 }
16
17 void hashtab_add(struct ht_elem *htep,
18                   struct ht_lock_state *lsp)
19 {
20     struct ht_bucket *htbp = lsp->hbp[0];
21     int i = lsp->hls_idx[0];
22
23     htep->hte_next[!i].prev = NULL;
24     cds_list_add_rcu(&htep->hte_next[i], &htbp->htb_head);
25 }
26
27 void hashtab_del(struct ht_elem *htep,
28                   struct ht_lock_state *lsp)
29 {
30     int i = lsp->hls_idx[0];
31
32     if (htep->hte_next[i].prev) {
33         cds_list_del_rcu(&htep->hte_next[i]);
34         htep->hte_next[i].prev = NULL;
35     }
36     if (lsp->hbp[1] && htep->hte_next[!i].prev) {
37         cds_list_del_rcu(&htep->hte_next[!i]);
38         htep->hte_next[!i].prev = NULL;
39     }
40 }

```

이 버전의 `hashtab_add()` 는 원소를 아직 크기 재조정이 되지 않았다면 기존 버킷에, 이미 크기 재조정이 되었다면 새 버킷에 저장하며 `hashtab_del()` 은 명시된 원소를 어디든 그게 삽입된 버킷에서 제거합니다. `hashtab_lookup()` 함수는 기존 버킷 탐색이 실패하면

Listing E.7: Resizable Hash-Table Update-Side Locking Function (Fewer Updates)

```

1 static void
2 hashtable_lock_mod(struct hashtable *htp_master, void *key,
3                      struct ht_lock_state *lsp)
4 {
5     long b;
6     unsigned long h;
7     struct ht *htp;
8     struct ht_bucket *htbp;
9
10    rcu_read_lock();
11    htp = rcu_dereference(htp_master->ht_cur);
12    htp = ht_get_bucket(htp, key, &b, &h);
13    spin_lock(&htbp->htb_lock);
14    lsp->hbp[0] = htp;
15    lsp->hls_idx[0] = htp->ht_idx;
16    if (b > READ_ONCE(htp->ht_resize_cur)) {
17        lsp->hbp[1] = NULL;
18        return;
19    }
20    htp = rcu_dereference(htp->ht_new);
21    htp = ht_get_bucket(htp, key, &b, &h);
22    spin_lock(&htbp->htb_lock);
23    lsp->hbp[1] = lsp->hbp[0];
24    lsp->hls_idx[1] = lsp->hls_idx[0];
25    lsp->hbp[0] = htp;
26    lsp->hls_idx[0] = htp->ht_idx;
27 }

```

새 버킷을 탐색하는데, 이는 탐색 fastpath에 오버헤드를 더하는 단점을 갖습니다. 대안적인 hashtable_lock_mod()는 크기 재조정 오퍼레이션이 진행 중이라면 새 버킷의 ->hbp[0] 와 ->hls_idx[0] 의 새 버킷으로의 락킹 상태를 아마도 그보다 자연스런 선택일 ->hbp[1] 과 ->hls_idx[1] 대신 리턴합니다. 그러나, 이 덜 자연스러운 선택은 hashtable_add()를 단순화 시키는 이익을 갖습니다.

이 코드의 추가적인 분석은 독자 여러분의 연습으로 남겨두겠습니다.

□

Quick Quiz 10.13:

Listing 10.13의 hashtable_resize() 함수에서, 라인 29에서의 ->ht_new로의 업데이트가 라인 40에서의 ->ht_resize_cur 업데이트보다 전에 일어난 것으로 hashtable_add()와 hashtable_del()의 관점에 보일 것을 무엇이 보장합니까? 달리 말하자면, hashtable_add()와 hashtable_del()이 ->ht_new에서 얻어진 NULL 포인터를 역참조하는 것을 무엇이 방지합니까?

■

Answer:

Listing 10.13의 라인 30에서의 synchronize_rcu()가 모든 앞서서부터 존재해온 RCU 읽기 쓰레드들은 우리가 새 해쉬 테이블 레퍼런스를 라인 29에 설치한 시점과 우리가 라인 40에서 ->ht_resize_cur를 업데이트하는 시점 사이에 완료될 것을 보장합니다. 이는 ->ht_resize_cur의 음수가 아닌 값을 보는 모든 읽기

쓰레드는 ->ht_new 할당 전에 시작했을 수 없음을, 따라서 새 해쉬 테이블로의 레퍼런스를 볼 수 있을 것임을 의미합니다.

그리고 이는 업데이트 쪽 hashtable_add()와 hashtable_del() 함수가 RCU read-side 크리티컬 섹션에 있어야만 하는지에 대한 이유로, Listing 10.11의 hashtable_lock_mod()와 hashtable_unlock_mod() 덕입니다.

□

Quick Quiz 10.14:

Listing 10.13의 라인 40에는 왜 WRITE_ONCE()가 있나요?

■

Answer:

Listing 10.11의 line 16 in hashtable_lock_mod()에서의 READ_ONCE()와 함께 그것이 컴파일러에게 ->ht_resize_cur로의 액세스는 ->ht_resize_cur로부터의 읽기가 실제로 쓰기와 레이스를 벌일 수 있으므로 “if” 조건을 바꾸지 않음으로써 남아있을 수 있게 해달라고 이야기 합니다.

□

Quick Quiz 10.15:

Figure 10.19에 보인 크고 작은 해쉬 테이블들 사이의 성능 차이 중 열만큼이 긴 해쉬 체인 때문이었고 열만큼은 메모리 시스템 병목 때문이었나요?

■

Answer:

이 질문에 답하는 한가지 쉬운 방법은 2,097,152 원소를 가지고 수행하지만 이번엔 2,097,152 버킷을 가져서 버킷당 평균 원소 갯수가 하나로 되도록 되돌리는 것입니다.

그 결과가 Figure E.7 가운데에 있는 세개의 새로운 점선으로 보여져 있습니다. 나머지 여섯개의 선은 page 189의 Figure 10.19에 보인 것들과 동일합니다. 이 새 선과 아래쪽 세 선들의 집합 사이의 차이가 그 성능 차이 중 열만큼이 해쉬 체인 길이 때문이었는지 보이며, 새 선과 위쪽 세 선 집합 사이의 차이가 그 성능 차이 주 열만큼이 메모리 시스템 병목 때문이었는지입니다. 이 새 선은 단일 CPU에서 262,144-원소의 버전의 것보다 조금 낮게 시작하여, 단일 CPU에서 조차 캐시 용량이 성능을 적게나마 떨어뜨립니다.¹² 이는 예견되었을 것으로, 그 작은 버전의과 달리 2,097,152-버킷 해쉬 테이블은 L3 캐시에 들어서지 않기 때문입니다. 이 새 선은 28 CPU를 넘어서야 올라가기 시작하는데 이 역시 예견되었을 것입니다. 이 상승은 29번째 CPU

¹² 그래요, 하드웨어 설계자가 걱정하는 만큼 캐시는 메모리 시스템의 한 부분입니다.

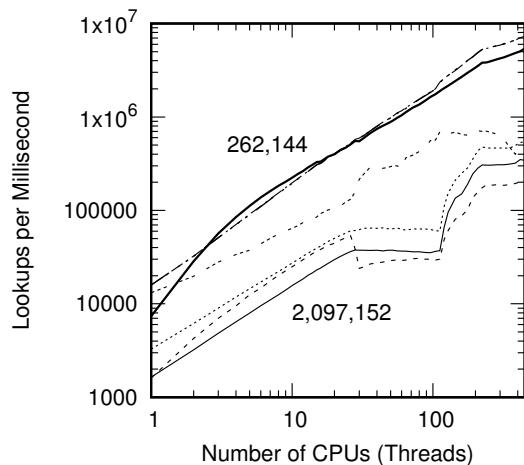


Figure E.7: Effect of Memory-System Bottlenecks on Hash Tables

가 다른 소켓에 있기 때문으로, 추가적인 39 MB 캐시와 함께 추가적인 메모리 대역폭을 가져옵니다.

하지만 이 큰 해쉬 테이블의 524,288 버킷 (하지만 여전히 2,097,152 원소) 해쉬 테이블에 대비한 성능 이득은 추가적인 CPU들과 함께 감소하는데, 메모리 시스템에 존재하는 병목으로 인한 일관적 모습입니다. 400 CPU를 넘어서서는 2,097,152-버킷 해쉬 테이블이 524,288 버킷 해쉬 테이블보다 실제로 약간 뒤집니다. 메모리 시스템이 병목이고 더 큰 수의 버킷은 이 워크로드의 메모리 사용량을 놀랄 일이 아닙니다.

주의 깊은 독자 여러분은 앞의 “대략적”이라는 단어에 주목했을 것이며 더 자세한 분석에 흥미가 있을 수도 있습니다. 그런 독자 여러분은 비슷한 벤치마크를 사용할 수 있는 performance counter 또는 하드웨어 분석 도구 또는 무엇이든 가지고 수행해 보시기 바랍니다. 이는 길고 복잡한 여정이 될 수 있으나, 거기 시간을 들일 용기가 충분한 사람은 하드웨어 성능과 그것의 소프트웨어로의 영향에 대한 자세한 지식으로 보상받을 것입니다.

□

Quick Quiz 10.16:

`hashtorture.h` 코드는 `ht_get_bucket()` 기능을 포함하는 버전의 `hashtab_lock_mod()`를 갖도록 수정될 수 없나요?

■

Answer:

아마 그럴 수 있을 거고, 그렇게 하는건 이 챕터에 보인 버킷별 락킹 기반 해쉬 테이블에 이득을 줄 겁니다. 이 수정을 가하는 것은 독자 여러분의 연습문제로 남겨둡니다.

□

Quick Quiz 10.17:

이 특수화가 얼마나 이득을 제공하나요? 그게 정말 가치있나요?

■

Answer:

첫번째 질문에 대한 대답은 독자 여러분의 연습문제로 남겨둡니다. 이 크기 재조정 가능한 해쉬 테이블을 특수화 시키고 열만큼의 성능 개선이 일어나는지 보십시오. 두번째 질문은 일반적으로 답변될 수 없으며, 특정 사용 예에 맞춰 답변되어야만 합니다. 어떤 사용처는 다른 것들에 비해 성능과 확장성에 무척 예민할 수 있습니다.

□

E.11 Validation

Quick Quiz 11.1:

컴퓨팅에서 파편화된 계획을 진행해야 할 때는 언제죠?

■

Answer:

수많은 상황이 있는데, 가장 중요한 상황은 아마도 개발되어야 할 프로그램을 닮은 무언가를 누구도 만들어 본 적이 없을 때입니다. 이 경우, 믿을 수 있는 계획을 만드는 유일한 방법은 그 프로그램을 구현하고, 계획을 만들고, 두번째로 구현하는 겁니다. 하지만 그 프로그램을 처음 구현하는 사람은 누구든 어떤 파편화된 계획을 따를 수밖에 다른 선택의 여지가 없는데, 무지에서 만들어진 구체적 계획은 실제 세계와의 첫번째 만남에서 생존할 수 없기 때문입니다.

그리고 아마도 이게 파편화된 계획을 따르길 좋아하는, 미친듯 긍정적인 인류가 진화한 이유 중 하나일 겁니다.

□

Quick Quiz 11.2:

누가 조직을 신경씁니까? 어쨌건, 중요한 건 그 프로젝트라구요!

■

Answer:

맞아요, 프로젝트는 중요합니다만, 여러분이 여러분의 작업에 대해 돈을 받고 싶다면, 여러분은 프로젝트에 더해 조직이 필요합니다.

□

Quick Quiz 11.3:

여러분이 다음처럼 보이는 `time` 커맨드의 출력 결과를 처리하는 스크립트를 짠다고 생각해 봅시다:

real	0m0.132s
user	0m0.040s
sys	0m0.008s

그 스크립트는 입력에 에러가 있는지, 그리고 에러가 있는 `time` 결과를 받았다면 적절한 진단 결과를 낼 수 있도록 검사를 해야 합니다. 싱글쓰레드 프로그램에 의해 생성되는 `time` 결과를 사용하기 위해 여러분은 이 프로그램의 테스트를 위해 어떤 테스트 입력을 제공해야 할까요?

■

Answer:

다음의 질문들에 모두 “네”라고 할 수 있습니까?

- 모든 시간이 CPU에 제한되는 프로그램에 의해 사용자 모드에서 소모된 경우를 위한 테스트가 있습니까?
- 모든 시간이 CPU에 제한되는 프로그램에 의해 시스템 모드에서 소모된 경우를 위한 테스트가 있습니까?
- 모든 세 시간이 0인 경우를 위한 테스트가 있습니까?
- `user` 와 `sys` 시간의 합이 `real` 시간보다 큰 경우를 위한 테스트가 있습니까? (이는 멀티쓰레드 프로그램에서 완전히 합법적일 겁니다.)
- 이 시간들 중 하나가 1초 이상일 경우를 위한 테스트가 있습니까?
- 이 시간들 중 하나가 10초 이상일 경우를 위한 테스트가 있습니까?
- 이 시간들 중 하나가 0분이 아닌 경우 (예를 들어, 15m36.342s)를 위한 테스트가 있습니까?
- 이 시간들 중 하나가 60 이상의 초를 갖는 경우를 위한 테스트가 있습니까?
- 이 시간들 중 하나가 32비트의 밀리세컨드를 넘는 경우를 위한 테스트가 있습니까? 64 비트의 밀리세컨드는 어떻습니까?
- 이 시간들 중 하나가 음수인 경우를 위한 테스트가 있습니까?
- 이 시간들 중 하나가 양의 분을 갖지만 음의 초를 갖는 경우를 위한 테스트가 있습니까?

12. 이 시간들 중 하나가 `m` 또는 `s`를 생략하는 경우를 위한 테스트가 있습니까?

13. 시간들 중 하나가 숫자가 아닌 경우를 위한 (예를 들어, `Go Fish`) 테스트가 있습니까?

14. 라인들 중 하나가 생략되는 경우를 위한 (예를 들어, `real`과 `sys` 값은 있지만 `user` 값은 없는) 테스트가 있습니까?

15. 라인들 중 하나가 중복되는 경우를 위한 테스트가 있습니까? 중복되었지만 다른 시간 값을 갖는 경우는 어떻습니까?

16. 특정 라인이 두개 이상의 값을 갖는 경우를 위한 (예를 들어, `real 0m0.132s 0m0.008s`) 테스트가 있습니까?

17. 무작위적 글자들을 담는 경우를 위한 테스트가 있습니까?

18. 무효한 입력을 포함하는 모든 테스트에서, 모든 조합을 만들었습니까?

19. 각 테스트에서, 예상되는 테스트 결과가 있습니까?

여러분이 앞의 경우들의 상당한 수의 테스트 데이터를 만들지 않았다면, 여러분은 높은 품질의 테스트를 만들 기회를 위해 더 파괴적인 태도를 가져야 할 겁니다.

물론, 파괴성에 있어 경제적이 되는 한가지 방법은 테스트 될 소스 코드와 함께 테스트를 만드는 것으로, 화이트 박스 테스팅이라 부립니다 (블랙 박스 테스팅의 반대입니다). 그러나, 이는 만능이 아닙니다: 여러분의 생각이 그 프로그램이 무엇을 다룰 수 있는지에 제한되어 정말 파괴적인 입력을 만드는데 실패하기가 정말 쉽다는 걸 알게 될 겁니다.

□

Quick Quiz 11.4:

제가 코딩을 시작하기도 전부터 이 겸종 일을 하라고 말하는 거예요??? 그건 시작도 안하기에 아주 좋은 방법처럼 들리는데요!!!

■

Answer:

그게 여러분의 프로젝트라면, 예를 들어 취미라면, 하고 싶은대로 하세요. 여러분이 낭비하는 시간은 여러분 스스로의 것일 것이고, 그걸 위해 여러분이 답변을 해야 할 질문을 던지는 사람도 없습니다. 그리고 시간이 완전히 낭비되지 않을 기회도 있습니다. 예를 들어, 여러분이 첫번째 종류의 프로젝트를 한다면, 그 요구사항은 어떤 면에서는 어쨌건 알려지지 않았을 겁니다. 이 경우,

최선의 방법은 여러개의 대략적 해결책들을 빠르게 프로토타이핑하고 그것들을 시도해 본 후, 뭐가 가장 잘 동작하는지 보는 겁니다.

다른 한편, 여러분이 존재하는 시스템들과 상당히 비슷한 시스템을 만들도록 돈을 받고 있다면, 여러분은 여러분의 사용자, 고용자, 그리고 미래의 자신에게 일찍 그리고 자주 검증할 일을 빚지는 겁니다.



Quick Quiz 11.5:

소프트웨어의 정확성을 정말로 테스트 할 수 있다고 말하는 거예요??? 그게 불가능하다는 건 모두가 안다구요!!!



Answer:

이 글은 “테스트” 보다는 “검증”이라는 단어를 사용함에 주의해 주세요. “검증”은 테스팅에 대해 formal한 방법을 포함하는데, 이에 대해 더 자세한 내용을 위해선 Chapter 12를 읽어 주세요.

하지만 모두가 알아야 할 내용을 이야기 하는 동안은, 다원의 진화는 정확도에 대한 것이 아니라 생존에 대한 것임을 기억합시다. 소프트웨어에서도 마찬가지입니다. 개발자로서 저의 목표는 제 소프트웨어가 이론적 관점에서 매력적이게 하는게 아니라 사용자가 무엇을 던져대든 생존하는 것입니다.

정확성이라는 방향도 올바른 사용처가 있지만, 그것의 기본적 한계는 그 정확도가 판정되는 기술서 역시 버그를 가진다는 것입니다. 이는 해당 코드가 의도된 버그들을 가지고 있는지 증명하는 전통적 정확성 증명 이상도 이하도 아님을 의미합니다!

정확성의 대안적 정의는 그 대신 문제가 되는 특성의 부재에 집중하는데, 예를 들어 소프트웨어가 use-after-free 버그, NULL 포인터 역참조, array-out-of-bounds 참조 등이 없음을 증명하는 것입니다. 실수하지 않고 그런 종류의 버그를 찾아서 제거하는건 매우 유용합니다. 그러나 그런 종류의 버그의 부재는 어떤 특정 목표에 적합한지를 보이는 데에는 어떤 일도 하지 않는다는 사실은 여전합니다.

따라서, 사용처 기반 검증이 여전히 중요하게 남아있습니다.

별개로, 여러분이 소프트웨어의 정확성을 검증하는 것 역시 중요한데, 특히 검증기와 명세서의 검증의 필요성을 놓고 보면 더욱 그렇습니다.



Quick Quiz 11.6:

WARN_ON_ONCE() 를 어떻게 구현할 수 있나요?



Answer:

여러분이 WARN_ON_ONCE() 가 가끔은 두번 이상 경고를 하는게 괜찮다면, 간단하게 0으로 초기화 되는 static 변수를 두세요. 그 조건이 일어났을 때, 그 변수를 검사하고 그게 0이 아니면 리턴하세요. 그렇지 않다면, 그것을 1로 바꾸고 메세지를 출력하고 리턴하세요.

그 메세지가 정말로 한번 넘게 발생하지 않아야 한다면 위의 “1로 바꾸고” 대신 어토믹 교환 오퍼레이션을 사용할 수 있습니다. 그 어토믹 교환 오퍼레이션이 0을 리턴할 때에만 메세지를 출력하세요.



Quick Quiz 11.7:

숨겨져 있는 리눅스 커널 해커들의 어떤 잘못된 가정을 당신은 이야기 하는 건가요???



Answer:

이 질문에 대한 완벽한 답을 원하는 사람들은 “Fixes:” 문자를 가지고 있는 커밋들을 리눅스 커널 git 저장소에서 검색해 보세요. 그런 것들이 2020년에는 수천개가 있는데, 아래의 잘못된 가정을 위한 수정들을 포함합니다:

1. 0이 아닌 분모에 대한 검사가 divide-by-zero 에러를 막는다. (힌트: 테스트는 64비트 오퍼레이터를 사용하는데 나누기는 32비트 오퍼레이터를 사용한다고 생각해 보세요.)
2. Userspace 는 커널과 통신하는데 사용되는 버전 관리된 데이터 구조를 0으로 초기화 한다고 신뢰될 수 있다. (힌트: 가끔 userspace 는 그 데이터 구조가 얼마나 큰지 모를 때가 있습니다.)
3. 오래되어 유효하지 않은 TCP duplicate selective acknowledgement (D-SACK) 패킷은 완전히 무시될 수 있다. (힌트: 이 패킷은 다른 정보도 담고 있을 수도 있습니다.)
4. 모든 CPU 는 little-endian 을 사용한다.
5. 데이터 구조가 더이상 필요하지 않아지면 그 메모리는 즉시 해제된다.
6. 모든 기기는 대기 모드인 상태에서 초기화 될 수 있다.
7. 개발자들은 올바른 16진수 산수를 일관적으로 할 거라 신뢰될 수 있다.

이 커밋들을 더 자세히 들여다보는 분들은 잘못된 가정은 예외가 아니라 규칙이라는 결론을 내릴 겁니다.



Quick Quiz 11.8:

왜 존재하는 코드를 종이에 펜으로 복사하죠??? 그건 필사 과정의 애러 가능성을 높일 뿐 아닌가요?



Answer:

필사 과정의 애러를 걱정한다면, `diff`라는 이름의 정말 멋진 도구를 여러분께 처음으로 소개할 수 있게 해주십시오. 또한, 복사를 진행하는건 상당히 가치있습니다:

- 여러분이 많은 코드를 복사한다면, 여러분은 추상화 기회의 이점을 얻지 못할 수 있습니다. 코드를 복사하는 행위는 추상화를 위한 상당한 동기를 부여합니다.
- 코드를 복사하는 것은 여러분에게 이 코드가 새 환경에서 정말 동작할지 생각할 기회를 줍니다. 어떤 분명치 않은 제약, 예를 들어 인터럽트를 불능화 해야하거나 어떤 락을 줘어야 한다거나 하는게 있을까 같은 질문.
- 코드를 복사하는 것은 또한 그 일이 되게 하기 위한 어떤 나은 방법이 있는지 생각할 시간을 줍니다.

그러니까, 그래요, 코드를 복사하세요!



Quick Quiz 11.9:

이 과정은 웃기도록 지나친 엔지니어링이예요! 이 방법으로 작성되는 합리적 양의 소프트웨어를 기대할 수 있습니까???



Answer:

실제로, 손으로 코드를 복사하기 반복하는 것은 노동집약적이고 느립니다. 그러나, 상당한 스트레스 테스트와 정확성 증명과 결합되면 이 방법은 또한 궁극의 성능과 안정성을 필요로 하며 디버깅이 어려운 복잡한 병렬 코드에 굉장히 효과적입니다. 리눅스 커널 RCU 구현이 그 경우입니다.

다른 한편, 여러분이 간단한 싱글쓰레드 셀 스크립트를 작성한다면, 다른 방법이 나을 겁니다. 예를 들어, 대화형 셀에 각 커맨드를 한번에 하나씩 그게 여러분이 원하는 걸 함을 분명히 하기 위한 테스트 데이터들과 함께 수행해보고, 여러분의 스크립트에 성공적인 커맨드를 복사해 붙여넣으세요. 마지막으로, 전체 스크립트를 테스트하세요.

돕고자 하는 동료나 친구가 있다면, 많은 정식적 설계와 코드 리뷰 프로세스를 만큼이나 짹 프로그래밍이 매우 잘 동작할 수 있습니다.

그리고 여러분이 코드를 취미로 작성하고 있다면, 그 땐 뭐든 좋을대로 하세요.

요약하자면, 다른 종류의 소프트웨어는 다른 개발 방법론을 필요로 합니다.



Quick Quiz 11.10:

종이 위에 펜으로 복사하기 후에 결과 코드를 타이핑하는 와중에 버그를 발견하면 어떡하나요?



Answer:

그에 대한 답은 종종 그렇듯 “종속적입니다”. 그 버그가 간단한 오타라면 그 오타를 고치고 타이핑을 계속하세요. 그러나, 그 버그가 디자인 결함을 알린다면, 펜과 종이로 돌아가세요.



Quick Quiz 11.11:

잠깐만요! 대체 왜 소프트웨어의 추상화된 부분이 가끔만 실패하죠???



Answer:

복잡도와 동시성은 무작위와 구분할 수 없는 결과를 내놓기 때문입니다 [MOZ09]. 예를 들어, 리눅스 커널 RCU의 어떤 버그는 그 버그가 발견되기 위해 다음 상황을 필요로 했습니다:

- 커널은 HPC나 리얼타임을 위해 빌드되어서 한 CPU의 RCU 일은 다른 CPU에게 넘겨질 수 있습니다.
- 일을 떠넘긴 CPU는 많은 양의 RCU 일감을 만들어낸 직후 오프라인으로 됩니다.
- 바로 이 시점에서 특수한 `rcu_barrier()` API가 호출됩니다.
- 앞서 오프라인된 CPU의 RCU 일거리는 `rcu_barrier()`가 리턴한 후에도 여전히 수행되는 중입니다.
- 이 남아있는 RCU 일거리는 `rcu_barrier()`를 수행하는 코드와 연관되어 있습니다.

따라서 이 버그를 찾는 데에는 상당한 운과 테스트 기술이 필요했습니다. 그러나 테스트 기술은 그 버그가 알려졌을 때에만 효과적인데 물론 이 경우는 그렇지 않았습니다. 따라서, 이 버그를 찾는 것은 확률적 프로세스로 모델링 되었습니다.



Quick Quiz 11.12:

여러분이 폐기할 수 있는 무척 많은 시스템을 가지고 있다고 해봅시다. 예를 들어, 현재의 클라우드 가격대에서, 여러분은 낮은 가격으로 많은 CPU를 구매할 수 있습니다. 모든 실용적 목적의 충분한 확실성을 위해 이 방법을 사용하는 건 어때요?

**Answer:**

이 방법도 여러분의 검증 무기고에 가치있는 추가가 될 겁니다. 하지만 “모든 실용적 목적”을 불가능하게 하는 한계도 있습니다:

1. 어떤 버그는 극단적으로 낮은 발생확률을 가지지만 고쳐질 필요가 없습니다. 예를 들어, 리눅스 커널의 RCU 구현이 평균 백만년에 한번 발생하는 버그를 가지고 있다고 해봅시다. 백만년의 CPU 시간은 가장 짧은 클라우드 플랫폼에서라 해도 굉장히 비싸지만 2017년 기준 200억 개가 넘는 리눅스 인스턴스가 있으니 하루에 50번도 넘는 실패가 발생할 거라 예상할 수 있습니다.
2. 그 버그는 여러분의 특정 클라우드 컴퓨팅 테스트 환경에서는 발생 확률이 0일 수도 있는데, 이는 여러분이 테스트에 얼마나 많은 기계 시간을 사용하는 그걸 보지 못할 것을 의미합니다. 한가지만 예를 들어보면, preemption 가능한 커널에서만 나타나는 RCU 버그들이 있으며, preemption 불가능한 커널에서만 발생하는 RCU 버그들도 있습니다.

Quick Quiz 11.14:

Equation 11.6에서, 로그의 밑은 10인가요, 2인가요, 아니면 e인가요?

**Answer:**

상관없습니다. 이 결과는 로그값 사이의 비율이므로 어떤 밑을 사용하더라도 같은 답을 얻게 됩니다. 유일한 제약은 여러분이 문자와 분모에 모두 같은 밑을 사용해야 한다는 겁니다.

**Quick Quiz 11.15:**

어떤 버그가 어떤 테스트를 시간당 평균 세번 실패를 하게한다고 해봅시다. 어떤 수정사항이 그 실패 확률을 충분히 줄였음을 99.9% 확신하기 위해선 얼마나 오래 예리 없는 테스트 수행을 해야 할까요?

**Answer:**

Equation 11.11에 n 을 3으로, P 를 99.9로 하면 다음과 같이 됩니다:

$$T = -\frac{1}{3} \ln \frac{100 - 99.9}{100} = 2.3 \quad (\text{E.9})$$

테스트가 실패 없이 2.3 시간을 수행된다면 우린 이 수정사항이 실패 확률을 줄였음을 99.9% 확신할 수 있습니다.

**Quick Quiz 11.16:**

이 모든 factorial과 exponential을 더하는 건 고통입니다. 더 쉬운 방법은 없나요?

**Answer:**

한가지 방법은 “maxima”라는 오픈소스 기호 조작 프로그램을 사용하는 겁니다. 많은 리눅스 배포판의 일부인 이 프로그램을 설치하면 여러분은 그걸 실행하고 `load(distrib);`에 이어 여러 `bfloor(cdf_poisson(m,1));` 커맨드를 사용할 수 있는데, `m`은 원하는 m 값으로 교체하고 (실제 테스트에서의 실제 실패 횟수) 1은 원하는 λ (실제 테스트에서 예상되는 실패 횟수)로 설정해야 합니다.

특히, `bfloor(cdf_poisson(2,24));` 커맨드는 1.181617112359357b-8가 되는데, Equation 11.13에서의 값과 일치합니다.

다른 방법은 이 실제 세계를 인정하는 것으로, 349.2x 안정성 개선의 76.8% 확신을 주기 위해 두번 이하의 오류를 테스트 기간을 계산하는게 전혀 유용하지 않다는 것입니다. 대신, 인간은 특정 값에 집중하여 하는데,

물론, 여러분의 코드는 충분히 작아서 Chapter 12에서 이야기하는 정형적 검증이 도움이 될 겁니다. 하지만 주의하세요: 여러분의 코드의 정형적 검증은 여러분의 가정, 요구사항에 대한 오해, 여러분이 사용하는 소프트웨어 또는 하드웨어 기능에 대한 오해, 혹은 여러분이 증명해야 할 것을 찾지 못한 오류에 대해서는 오류를 찾지 못할겁니다.

**Quick Quiz 11.13:**

뭐라구요??? 앞의 10% 실패율의 다섯번의 테스트 예를 이식에 대입해보면 59,050%를 얻게 되는데 이건 말이 안되잖아요!!!

**Answer:**

그 말이 맞습니다, 말이 안되죠.

확률은 0과 1 사이의 수여서 확률을 구하기 위해선 퍼센트 값을 100으로 나눠야 함을 기억하세요. 따라서 10%는 0.1의 확률이며, 0.4095, 반올림해 41%의 확률을 얻게 되는데 앞의 결과와 잘 들어맞습니다.



Table E.3: Human-Friendly Poisson-Function Display

Certainty (%)	Improvement		
	Any	10x	100x
90.0	2.3	23.0	230.0
95.0	3.0	30.0	300.0
99.0	4.6	46.1	460.5
99.9	6.9	69.1	690.7

예를 들어 10x 개선의 95 % 확신입니다. 사람들은 또한 오류가 없는 테스트 수행을 훨씬 선호하는데 그게 여러분에게 필요한 테스트 시간을 줄이므로 여러분도 그럴겁니다. 따라서, Table E.3 의 값들이 충분한 경우가 많을 겁니다. 간단히 원하는 확신도와 개선 정도를 찾아보면 그 결과 숫자가 한번의 오류가 나타나기 위해 예상되는 시간을 위한 에러 없는 테스트 시간을 보일 겁니다. 따라서 여러분의 수정 전 테스트가 시간당 한번의 실패를 겪었다면, 10x 개선의 95 % 확신을 갖기 위해 여러분은 30시간의 오류 없는 테스트 수행을 해야 합니다.

대안적으로, 여러분은 Section 11.6.2 에 소개된 대략적이고 준비된 방법을 사용할 수 있습니다.

□

Quick Quiz 11.17:

하지만 기다려요!!! 약간의 실패가 있음을 생각하면 (0 번 실패할 확률을 포함해), Equation 11.13 은 m 이 무한으로 감에 따라 1의 값이 되지 않나요?

■

Answer:

맞습니다. 그리고 그렇습니다.

이걸 자세히 보기 위해, $e^{-\lambda}$ 는 i 에 의존하지 않는데 이는 다음과 같이 합으로 나올 수 있음에 주목하시기 바랍니다:

$$e^{-\lambda} \sum_{i=0}^{\infty} \frac{\lambda^i}{i!} \quad (\text{E.10})$$

남아있는 합은 e^{λ} 의 Taylor 시리즈여서 다음이 도출됩니다:

$$e^{-\lambda} e^{\lambda} \quad (\text{E.11})$$

이 두 exponential 은 상통하는 것이며, 따라서 취소될 수 있어서 필요한대로 정확히 1 이 됩니다.

□

Quick Quiz 11.18:

이 오염이 어떤 연관되지 않은 포인터에 영향을 끼쳐서 그게 이후 오염을 일으킨다면 이 방법이 어떻게 도움이 되겠습니까?

■

Answer:

실제로, 그럴 수 있습니다. 많은 CPU 가 그 연관되지 않은 포인터를 찾을 수 있게 돋는 하드웨어 디버깅 기능을 갖습니다. 더 나아가, core dump 가 있다면 여러분은 그 오염된 메모리 영역을 참조하는 포인터를 core dump 에서 찾을 수 있습니다. 또한 여러분은 이 오염된 데이터 배치를 보고 그 배치에 맞는 타입의 포인터를 검사해 볼 수 있습니다.

여러분은 또한 한발 뒤로 물러서서 여러분의 프로그램의 강도를 높게 하는 모듈을 테스트 할 수도 있는데, 이는 오염을 거기 책임 있는 모듈로 국한시킬 확률이 높습니다. 이게 오염을 사라지게 만든다면, 각 모듈에서 노출되는 함수들에 추가적인 인자 검사를 더하는 걸 고려해 보세요.

그러나, 이는 어려운 문제이며 그게 제가 “약간 암흑예술”이라는 말을 사용한 이유입니다.

□

Quick Quiz 11.19:

전 bisect 를 했지만 거대한 커밋을 찾는데 그쳤어요. 이제 어떡하죠?

■

Answer:

거대한 커밋? 안됐군요! 그러나 이는 여러분이 커밋을 작게 유지해야 하는 이유 중 하나이기도 합니다.

그리고 그게 여러분의 답입니다: 그 커밋을 깨물기 좋은 작은 크기로 쪼개고 그 조각들을 bisect 하세요. 제 경험상, 커밋을 쪼개는 것은 버그가 분명해지게 만들기에 종종 충분합니다.

□

Quick Quiz 11.20:

조건적 락킹 기능은 왜 이 가짜 실패 기능을 제공하지 않죠?

■

Answer:

조건적 락킹 기능이 그들에게 사실을 이야기한다는 가정 하의 락킹 알고리즘들이 있습니다. 예를 들어, 만약 조건적 락 실패가 어떤 다른 쓰레드가 이미 어떤 일을 하고 있음을 신호한다면, 가짜 실패는 그 일을 결코 되지 않게 해서 거기 매달려 있는 결과가 나올 겁니다.

□

Quick Quiz 11.21:

웃기네요!!! 어쨌건, 남들보다 늦게 올바른 답을 얻는게 올바르지 않은 답을 얻는것보다는 낫지 않겠습니까???

**Answer:**

이 질문은 아예 어떤 답도 계산하지 않는 선택지를 고려하는 방향까지 가게 되는데 그렇게 함으로써 답을 계산하는 비용을 고려하는데까지 이르릅니다. 예를 들어, 정확한 모델이 존재하며 여러분이 그 모델을 날씨보다는 빠르게 수행하고자 하는 단기적 날씨 예보의 경우를 생각해 봅시다.

이 경우, 이 모델이 실제 날씨보다 빠르게 수행되는 것을 방해하는 모든 성능 버그는 모든 예보를 막습니다. 커다란 슈퍼컴퓨터 클러스터를 구매한 전체 목표는 날씨 예보였음을 놓고 보면, 여러분이 날씨보다 빠르게 그 모델을 돌릴 수 없다면 여러분은 그 모델을 아예 돌리지 않는게 나을 겁니다.

더 가혹한 예들을 안전에 치명적인 리얼타임 컴퓨팅 분야에서 찾아볼 수 있을 겁니다.

**Quick Quiz 11.22:**

하지만 그 모든 어플리케이션 병렬화의 어려운 일을 하 고자 한다면, 왜 그걸 옳게 하지 않죠? 왜 최적의 성능과 선형 확장성 미만의 것에 만족합니까?

**Answer:**

그 정신과 열망에 진심으로 경의를 표하는 바입니다만 당신은 프로그램 완료의 자연에 의한 높은 비용을 잊고 있습니다. 극단적인 예를 들어, 싱글쓰래드 어플리케이션에서의 40 % 성능 하락이 매일 사람 한명을 죽게 하고 있다고 해봅시다. 더 나아가 여덟 개 CPU 시스템에서 순차적 버전에 비해 50 % 빠르게 동작하지만 더러운 병렬 프로그램을 하루만에 만들 수 있지만 최적의 병렬 프로그램을 만드는데에는 4개월의 힘든 설계, 코딩, 디버깅, 그리고 투팅이 필요하다고 해봅시다.

100명 이상의 사람이 빠르고 더러운 버전을 좋아할 거라고 말해도 될겁니다.

**Quick Quiz 11.23:**

예를 들면 캐쉬와 메모리 배치 사이의 상호작용 같은 다른 오류의 원천들은 어떠하죠?

**Answer:**

메모리 배치의 변화는 실제로 비현실적인 수행시간 감소를 초래할 수 있습니다. 예를 들어, 특정 마이크로벤치마크가 거의 항상 L0 캐쉬의 용량을 넘치게 하지만 올바른 메모리 배치에서는 항상 딱 맞는다고 해봅시다.

이게 정말 문제라면, 메모리 배치를 완전히 제어하기 위해 여러분의 마이크로벤치마크를 *huge page* 를 이용해서 (또는 커널에서, 또는 *bare metal* 에서) 수행하는 걸 고려해 보세요.

하지만 다양한 메모리 배치 병목이 존재함을 알아 두시기 바랍니다. 메모리 대역폭에 민감한 벤치마크는 (행렬 계산과 관계되는 것 같은) 메모리 병렬성을 극대화하기 위해 수행되는 쓰레드를 사용 가능한 코어와 소켓들 사이로 흘뿌려야 합니다. 그것들은 또한 데이터를 NUMA 노드, 메모리 컨트롤러, 그리고 DRAM 칩 사이에 가능한 만큼 분배해야 합니다. 대조적으로, 메모리 응답시간에 민감한 벤치마크는 (가장 잘 확장되지 못하는 어플리케이션을 포함해) 그대신 지역성을 최대화해야 해서, 다른 코어와 소켓을 추가하기 전에 개별 코어와 소켓을 최대한 사용해야 합니다.

**Quick Quiz 11.24:**

테스트 되는 코드를 격리하기 위해 추천된 기법들 또한 그 코드의 성능에 영향을 끼치지 않을까요? 특히나 그게 더 큰 어플리케이션 내에서 돌아간다면요.

**Answer:**

실제로 그럴수도 있습니다만, 대부분의 마이크로벤치마크에서는 여러분은 테스트 되는 코드를 그걸 감싸는 어플리케이션으로부터 꺼낼 겁니다. 그러나, 어떤 이유로 여러분이 그 테스트 되는 코드를 어플리케이션 내에 두어야 한다면, Section 11.7.6 에서 이야기되는 기법을 사용해야 할 겁니다.

**Quick Quiz 11.25:**

이 방법은 그저 이상해요! 왜 통계 수업에서 배운대로 중간값과 표준편차를 이용하지 않죠?

**Answer:**

중간값과 표준편차는 이 일을 하기 위해 설계되지 않았기 때문입니다. 이를 보기 위해, 측정에 1 % 상대 오류를 갖는다고 가정하는 다음 데이터 집합에 중간값과 표준편차를 적용해 보시기 바랍니다:

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 중간값과 표준편차가 어떤 종류의 측정 오류 가정을 갖지 않으며, 따라서 그것은 49,500 근처의 값과 49,900 근처의 값들 사이의 차이를 실제로는 추정된

측정 오류 한계 내에 있음에도 통계적으로 심각하다고 본다는 겁니다.

물론, 비슷한 효과를 위해 절대적 차이값 대신 표준편차를 사용하는 Listing 11.2 와 비슷한 스크립트를 만들 수 있겠는데 이는 관심있는 독자 여러분의 연습문제로 남겨두겠습니다. 동일한 데이터 값들의 문자열에 의해 일어날 수 있는 divide-by-zero 오류를 막는데 주의하세요!



Quick Quiz 11.26:

하지만 믿어지는 데이터 그룹의 모든 y 값이 0이면 어떡하죠? 그건 이 스크립트가 모든 0이 아닌 값을 제거하게 하지 않을까요?



Answer:

실제로 그럴겁니다! 하지만 여러분의 성능 측정이 정확히 0 값을 만들어낸다면, 여러분의 성능 측정 코드를 깊이 들여다봐야 할수도 있습니다.

중간값과 표준편차에 기반한 많은 접근법들이 이런 종류의 데이터에 비슷한 문제를 가질 것임을 알아두시기 바랍니다.



E.12 Formal Verification

Quick Quiz 12.1:

락 사용 프로세스에 도달되지 못한 명령문이 있는 이유는 뭐죠? 어쨌건, 이건 전체 상태 공간 탐색 아니었나요?



Answer:

이 락 사용 프로세스는 무한 반복문이므로, 제어는 이 프로세스의 마지막에 결코 도달하지 못합니다. 그러나, 단조증가하는 변수가 없으므로 Promela 는 이 무한 반복문을 작은 수의 상태로 모델링할 수 있습니다.



Quick Quiz 12.2:

이 예에서 Promela 코드 스타일 문제는 무엇이 있을까요?



Answer:

여러개 있습니다:

1. `sum` 의 선언은 초기화 블록으로 옮겨져야 하는데, 어디서도 사용되지 않기 때문입니다.

2. 단정문 코드는 초기화 반복문 바깥으로 옮겨져야 합니다. 이 초기화 반복문은 그러면 어토믹 블록에 위치해서 상태 공간을 크게 줄일 수 있습니다 (얼마나 줄일까요?).

3. 단정문을 감싸는 이 어토믹 블록은 단정문에 더해 `sum` 과 `j` 의 초기화를 포함하도록 확장되어야 합니다. 이 또한 상태 공간을 줄입니다 (다시 말하지만, 얼마나 줄일까요?).



Quick Quiz 12.3:

이 `do-od` 문을 더 간단하게 짜는 방법이 있을까요?



Answer:

그렇습니다. 그걸 `if-fi` 로 대체하고 두개의 `break` 문을 없애세요.



Quick Quiz 12.4:

라인 12–21 와 라인 44–56 내의 오퍼레이션들은 현재의 모든 제품 마이크로프로세서에 원자적 구현이 없음에도 왜 어토믹 블록이 사용되었나요?



Answer:

그 오퍼레이션들은 단정문의 이익만을 위한 것들이기 때문입니다. 그것들은 알고리즘 자체의 부분이 아닙니다. 따라서 그것을 어토믹으로 만드는데 문제가 없으며, 그렇게 표시하는게 Promela 모델에 의해 탐색되어야 하는 상태 공간을 크게 줄여줍니다.



Quick Quiz 12.5:

라인 24–27 에서의 카운터들의 재 합산이 정말로 필요한가요?



Answer:

그렇습니다. 이를 보기 위해, 이 줄들을 지우고 모델을 돌려보세요.

대안적으로, 다음 단계들을 생각해 보십시오:

1. 한 프로세스가 RCU read-side 크리티컬 섹션 내에 있어서 `ctr[0]` 는 0이고 `ctr[1]` 은 2의 값을 갖습니다.
2. 한 업데이트 쓰레드가 수행을 시작하고 이 카운터들의 합이 2임을 보게 되어 빠른 수행경로를 수행하지 못하게 됩니다. 따라서 락을 잡습니다.
3. 두번째 업데이트 쓰레드가 수행을 시작해 `ctr[0]` 의 값을 읽어오는데 0입니다.

4. 첫번째 업데이트 스레드는 `ctr[0]`에 1을 더하고 인덱스를 뒤집고 (이제 0이 됩니다), `ctr[1]`에서 1을 뺍니다 (이제 1이 됩니다).
5. 두번째 업데이트 쓰레드가 이제 1인 `ctr[1]`의 값을 읽습니다.
6. 두번째 업데이트 쓰레드는 원래 읽기 쓰레드가 여전히 완료되지 못했음에도 이제 빠른 수행경로를 수행해도 안전하다고 잘못된 결론을 내립니다.

□

Quick Quiz 12.6:

상태들로 인해 점유되는 메모리에서 200대1 감소에 연관되는 0.48 % 압축률! 이 상태 공간 탐색은 정말로 완벽한가요???

■

Answer:

Spin의 문서에 따르면, 네, 그렇습니다.

간접적인 증거로, `-DCOLLAPSE` 와 `-DMA=88` (두개의 읽기 쓰레드와 세개의 업데이트 쓰레드)에 의한 수행 결과를 비교해 봅시다. 이 수행들에서의 결과물의 차이점이 Listing E.8에 보여져 있습니다. 볼 수 있듯, 상태의 수에 (저장된 것과 매치된 것) 둘 다 동의합니다.

□

Quick Quiz 12.7:

하지만 다른 정형 검증 도구들은 종종 특정 종류의 버그를 찾기 위해 설계됩니다. 예를 들어, 매우 적은 정형 검증 도구들은 명세서 내의 오류를 찾을 겁니다. 그러니 이 “분명 믿지 못할만 하다”는 좀 센 말 아닌가요?

■

Answer:

많은 정형 검증 도구들이 어떤 방법으로 특수화 되었음은 분명한 사실입니다. 예를 들어, Promela는 실제 메모리 모델을 다루지 않고 (그게 Promela에 프로그램이 될 수 있긴 합니다 [DMD13]), CMBC [CKL04]는 확률적 면밀과 데드락을 탐지하지 못하며, Nidhugg [LSLK14]는 데이터 비결정성에 관한 버그를 탐지하지 못합니다. 하지만 이는 이 도구들이 찾게끔 설계되지 않은 버그를 찾을 수 있다고는 믿어질 수 없음을 의미합니다.

그리고 따라서 정형 검증 도구를 작성하는 사람들은 “라벨에 진실을 말해야” 하고, 그들의 도구들이 어떤 종류의 버그를 찾고 못찾는지 분명히 밝혀야 합니다. 그러지 않는다면, 사용자는 그 도구가 어떤 버그를 탐지하지 못함을 처음 발견했을 때 그 도구에 대한 매우 거칠고 공개적인 비난을 할 겁니다. 그래요, 그래요, 최선을 다했다고 할 수 있는 뭔가가 있지만 적절한 면책조항 없이 무언가를 지나치게 이야기 하는 것은 여러분의 도구가

Listing E.8: Spin Output Diff of `-DCOLLAPSE` and `-DMA=88`

```

@@ -1,6 +1,6 @@
(Spin Version 6.4.6 -- 2 December 2016)
+ Partial Order Reduction
- + Compression
+ + Graph Encoding (-DMA=88)

Full statespace search for:
never claim - (none specified)
@@ -9,27 +9,22 @@
invalid end states +
State-vector 88 byte, depth reached 328014, errors: 0
+MA stats: -DMA=77 is sufficient
+Minimized Automaton: 2084798 nodes and 6.38445e+06 edges
1.8620286e+08 states, stored
1.7759831e+08 states, matched
3.6380117e+08 transitions (= stored+matched)
1.3724093e+08 atomic steps
-hash conflicts: 1.1445626e+08 (resolved)

Stats on memory usage (in Megabytes):
20598.919 equivalent memory usage for states
(actual memory usage for states
(compression: 40.87%)
- 8418.559 state-vector as stored =
- state-vector as stored =
19 byte + 28 byte overhead
- 2048.000 memory used for hash table (-w28)
+ 204.907 actual memory usage for states
(actual memory usage for states
(compression: 0.99%)
17.624 memory used for DFS stack (-m330000)
- 1.509 memory lost to fragmentation
-10482.675 total actual memory usage
+ 222.388 total actual memory usage

-nr of templates: [ 0:globals 1:chans 2:procs ]
-collapse counts: [ 0:1021 2:32 3:1869 4:2 ]
unreached in proctype qrcu_reader
(0 of 18 states)
unreached in proctype qrcu_updater
@@ -38,5 +33,5 @@
unreached in init
(0 of 23 states)

-pan: elapsed time 369 seconds
-pan: rate 505107.58 states/second
+pan: elapsed time 2.68e+03 seconds
+pan: rate 69453.282 states/second

```

그로부터 회복될 수도 그러지 못할수도 있는 부정적 반응을 쉽게 야기할 수 있습니다.

경고했어요!



Quick Quiz 12.8:

여기 설명된 QRCU 알고리즘을 위한 두개의 개별적 정확성 증명이 있고 부정확성 증명은 다른 알고리즘에 대한 것이었는데 왜 의심의 여지가 있죠?



Answer:

언제나 의심의 여지는 있습니다. 이 경우, 그 두개의 정확성 증명은 실제 세계의 메모리 순서 모델의 정형화를 사용했으므로 이 두개의 증명들은 올바르지 않은 메모리 순서 규칙 가정에 기반했을 가능성을 높입니다. 더 나아가서, 두 증명 모두 같은 사람에 의해 구축되었으므로 동일한 오류를 가지고 있을 가능성이 꽤 있습니다. 다시 말하지만, 항상 의심의 여지가 있습니다.



Quick Quiz 12.9:

그래요, 그거 훌륭하네요! 이제 제가 40 GB 메인 메모리가 없다면 뭘 하면 될까요???



Answer:

진정하세요, 이 질문에 대한 여러 합리적 답변이 있습니다:

1. 메모리 소모량을 줄이기 위해 컴파일러 플래그 `-DCOLLAPSE` 와 `-DMA=N` 을 시도하세요. Section 12.1.4.1 를 참고하세요.
2. 모델을 더 최적화 해서 메모리 소모량을 줄이세요.
3. 종이와 연필 증명을 진행하세요, 아마 리눅스 커널의 코드 상의 주석들로부터 시작할 수 있을 겁니다.
4. 코드의 정확성을 증명할 순 없지만 숨은 버그를 찾을 수 있는 고문 테스트를 주의 깊게 만드세요.
5. 작은 기계들의 클러스터에서 모델 검사를 하는 도구들을 위한 움직임이 있습니다. 그러나, 저 스스로는 Paul 이 우연히 접근할 수 있었던 큰 기계 때문에 그걸 써볼 일이 없었습니다.
6. 여러분의 문제에 맞는 메모리 사이즈를 갖지만 지불 가능한 가격대의 시스템을 기다리세요.
7. 짧은 시간동안 큰 시스템을 빌릴 수 있는 클라우드 컴퓨팅 서비스들 중 하나를 사용하세요.



Quick Quiz 12.10:

그냥 `rcu_update_flag` 의 값을 증가시키고 `rcu_update_flag` 의 기존 값이 0이었을 때 `dynticks_progress_counter` 의 값을 증가시키지 않는 이유는 뭐죠???



Answer:

NMI의 존재 때문에 그건 안됩니다. `rcu_irq_enter()` 가 `rcu_update_flag` 값을 증가시킨 직후, 그러나 `dynticks_progress_counter` 를 증가시키기 전에 NMI를 받았다고 해봅시다. NMI에 의해 호출된 `rcu_irq_enter()` 의 한 인스턴스가 `rcu_update_flag` 의 원래 값이 0이 아님을 보게 되고, 따라서 `dynticks_progress_counter` 의 값을 증가시키려 할겁니다. 이는 NMI 핸들러가 이 CPU에서 수행중이었음에 대한 흔적을 RCU grace-period 메커니즘에 남기지 않아서 이 NMI 핸들러의 모든 RCU read-side 크리티컬 섹션은 RCU 보호를 잃을 겁니다.

그 정의에 따라 막힐 수 없는 이 NMI 핸들러의 가능성은 이 코드를 복잡하게 만듭니다.



Quick Quiz 12.11:

하지만 라인 7 가 우리가 가장 바깥 인터럽트에 있음을 확인하면 우린 `dynticks_progress_counter` 를 항상 값 증가시켜야 하지 않나요?



Answer:

우리가 수행중인 태스크를 인터럽트 하지 않았다면요! 그 경우, `dynticks_progress_counter` 는 이미 `rcu_exit_nohz()` 에 의해 값이 증가되었을 것이어서 그걸 또다시 값 증가시킬 필요가 없을 겁니다.



Quick Quiz 12.12:

이 섹션의 코드에 있는 어떤 버그를 발견했습니까?



Answer:

여러분이 옳았는지 확인하기 위해 다음 섹션을 읽어보세요.



Quick Quiz 12.13:

`rcu_exit_nohz()` 와 `rcu_enter_nohz()` 에서의 메모리 배리어는 왜 Promela에서 모델링 되지 않죠?



Answer:

Promela는 순차적 일관성을 가정하므로 메모리 배리어를 모델링 할 필요는 없습니다. 실제로, page 227의

Listing 12.13에 보인 것처럼 우린 메모리 배리어의 부재를 명시적으로 모델링 해야 합니다.



Quick Quiz 12.14:

`rcu_exit_nohz()`에 이어 `rcu_enter_nohz()` 가 수행되는 걸 모델링 하는건 좀 이상하지 않나요? 종료 전에 진입하는 걸 모델링하는게 더 자연스럽지 않을까요?



Answer:

그게 더 자연스러울 수 있겠습니다만, 특히 우리가 뒤에 추가할 liveness 검사를 위해 이게 필요합니다.



Quick Quiz 12.15:

잠시만요! 리눅스 커널에서, `dynticks_progress_counter` 와 `rcu_dyntick_snapshot` 모두 CPU 별 변수들입니다. 그런데 왜 그것들이 전역 변수들로 모델링 되는거죠?



Answer:

이 grace-period 코드는 각 CPU 의 `dynticks_progress_counter` 와 `rcu_dyntick_snapshot` 변수들을 따로 처리하여서, 우린 그 상태를 단일 CPU 에 풍질 수 있습니다. 만약 grace-period 코드가 특정 CPU에서의 특정 값에 따라 뭔가 특별한 일을 하기 위한 거였다면 여러 CPU들을 모델링 해야 할수 있습니다. 하지만 다행히도, 우린 우리들을 grace-period 처리를 수행하는 하나와 dynticks-idle 모드를 진입하고 빠져나오는 하나, 총 두개의 CPU 에 안전히 가둘 수 있습니다.



Quick Quiz 12.16:

라인 25와 26의 한쌍의 뒤에서 뒤로의 `grace_period_state` 변경이 있는데, 우린 라인 25의 변경을 놓치지 않을 꺼라고 어떻게 확신하죠?



Answer:

Promela 와 Spin 은 모든 가능한 상태 변경을 기록함을 기억하세요. 따라서, 타이밍은 상관없습니다: Promela/Spin은 이 모델의 모든 나머지 부분을 일부 상태 변수가 특별히 그걸 막지 않는 이상은 그 두 명령문 사이에 섞어낼 겁니다.



Quick Quiz 12.17:

하지만 하나의 `EXECUTE_MAINLINE()` 그룹 내의 명령문들이 비원자적으로 수행되어야 한다면 어떡하죠?



Answer:

가장 쉬운 방법은 그런 명령문 각각을 각자의 `EXECUTE_MAINLINE()`에 넣는 것입니다.



Quick Quiz 12.18:

하지만 `dynticks_nohz()` 프로세스가 조건적인 “if”나 “do” 명령문을 가져서 이 구조의 몸체 명령문들이 비원자적으로 수행되어야 하는 경우는 어떡하죠?



Answer:

뒤 섹션에서 보게 되겠지만 한가지 가능한 방법은 명시적인 라벨과 “goto” 문을 사용하는 겁니다. 예를 들어, 다음 구조는:

One approach, as we will see in a later section, is to use explicit labels and “goto” statements. For example, the construct:

```
if
:: i == 0 -> a = -1;
:: else -> a = -2;
fi;
```

다음과 같이 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;
```

그러나, 이 매크로가 “if” 문의 경우에도 큰 도움이 되는지는 분명치 않으므로, 이런 종류의 상황에는 다음 섹션들에서 직접 코딩하겠습니다.



Quick Quiz 12.19:

라인 46 와 47 (“in_dyntick_irq = 0;” 와 “i++;”) 는 왜 어토믹하게 수행되나요?

**Answer:**

이 부분들은 이 모델을 제어하는데 관계됩니다만, 모델링되는 코드 자체에 대해서는 아니므로 이것들을 어토믹하지 않게 모델링 할 이유가 없습니다. 이것들을 어토믹하게 모델링하는 동기는 상태 공간 크기를 줄이고자 하는 것입니다.

**Quick Quiz 12.20:**

인터럽트의 어떤 특성을 이 dynticks_irq() 프로세스는 모델링할 수 없나요?

**Answer:**

그런 특성 가운데 하나는 중첩된 인터럽트일 것으로, 다음 섹션에서 다룹니다.

**Quick Quiz 12.21:**

Paul은 정말로 항상 그의 코드를 이 고통스럽게 점진적인 방식으로 작성하나요?

**Answer:**

항상 그런건 아니지만, 점점 더 자주 그리고 있습니다. 이 경우, Paul은 인터럽트 핸들러를 포함하는 가장 작은 코드조각으로부터 시작했는데, Promela에서 인터럽트를 가장 잘 모델링하는 방법을 그는 확신할 수 없었기 때문입니다. 일단 그게 동작하게 한 후로 그는 다른 기능들을 추가했습니다. (하지만 그게 이를 다시 하게 된다면, 그는 “장난감” 핸들러로부터 시작했을 겁니다. 예를 들어, 그는 변수를 두번 값 증가시키는 핸들러를 만들고 중앙의 코드는 그 값이 항상 짹수일 것을 검증하게 했을 수도 있습니다.)

왜 이런 점진적 방법을 취할까요? Brian W. Kernighan의 다음 발언을 생각해 보세요:

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

이는 코드의 생성을 최적화 하려는 모든 시도는 코딩에 들어가는 시간과 노력을 늘리는 비용이 있더라도 그 중점 중 최소한 66% 는 디버깅 과정을 최적화 하는데 있어야 함을 의미합니다. 점진적 코딩과 테스팅은 코딩에 드는 노력을 어느정도 늘리는 비용을 들여서 디버깅

과정을 최적화 하는 한가지 방법입니다. Paul은 모든 날들을(약 1주일) 코딩과 디버깅에 전념할 수 있는 사차를 가끔만 누릴 수 있기 때문에 이 방법을 취합니다.

**Quick Quiz 12.22:**

하지만 어떤 IRQ 핸들러가 완료되기 전에 NMI 핸들러가 수행을 시작했고 그 NMI 핸들러는 두번째 IRQ 핸들러가 시작될 때까지 수행을 계속하면 어떻게 되죠?

**Answer:**

이는 단일 CPU 내에서는 일어날 수 없습니다. 첫번째 IRQ 핸들러는 해당 NMI 핸들러가 리턴하기 전까지는 완료될 수 없습니다. 따라서, dynticks와 dynticks-nmi 변수들이 모두 특정 시간 동안 짹수였다면, 연관된 CPU는 그 시간 중 언젠가는 quiescent state에 정말로 있었습니다.

**Quick Quiz 12.23:**

이는 여전히 무척 복잡합니다. 간단하게 CPU 별 비트를 갖는 cpumask_t를 가지고 그 비트를 IRQ나 NMI 핸들러에 진입할 때 지우고 종료할 때 값 설정하는 건 어떨까요?

**Answer:**

이 방법이 기능적으로는 올바르지만 큰 기계에서는 지나친 IRQ 진입/종료 오버헤드를 초래할 수 있습니다. 반면에, 이 섹션에서 보인 방법은 각 CPU가 IRQ와 NMI 진입/종료 시에 CPU 별 데이터만 만져서 특히나 큰 기계에서는 IRQ 진입/종료 오버헤드를 훨씬 낮게 만듭니다.

**Quick Quiz 12.24:**

하지만 x86은 강한 메모리 규칙을 가지고 있는데 왜 그 메모리 모델을 정형화 시키죠?

**Answer:**

사실, 학계에서는 x86 메모리 모델을 완화된 형태로 생각하는데, 앞의 쓰기가 뒤따르는 읽기와 재배치 되는 것을 허용할 수 있기 때문입니다. 학계의 관점에서, 강한 메모리 모델은 어떤 재배치도 허용하지 않아서 모든 쓰레드가 그것들에게 보이는 모든 오퍼레이션의 순서에 동의할 수 있는 것입니다.

또한, 이건 개발자들이 가끔 x86 메모리 순서규칙에 대해 혼란에 빠지는 경우들입니다.



Quick Quiz 12.25:

Listing 12.23 의 라인 8 는 왜 레지스터를 초기화 시키나요? 왜 그대신 라인 4 와 5 에서 초기화 시키지 않죠?

**Answer:**

두 방법 모두 잘 동작합니다. 그러나, 일반적으로는 명시적 명령보다 초기화를 사용하는게 낫습니다. 명시적인 명령은 이 예에서 그 사용법을 보이기 위해 사용되었습니다. 또한, 이 도구의 웹사이트에서 (<https://www.cl.cam.ac.uk/~pes20/ppcmem/>) 얻을 수 있는 많은 리트머스 테스트는 명시적 초기화 명령들을 생성하는 자동화 방법을 사용해 만들어 졌습니다.

**Quick Quiz 12.26:**

하지만 Listing 12.23 의 라인 17, 즉 Fail1: 라벨에 무언가 벌어지길 할까요?

**Answer:**

PowerPC 버전의 `atomic_add_return()` 구현은 `stwcx` 명령이 실패했을 때 반복을 하게 되는데, 조건 코드 레지스터에 0이 아닌 상태를 설정함으로써 이를 통신하며, 이는 결국 `bne` 명령에 의해 검사됩니다. 실제로 반복문을 모델링 하는 것은 상태 공간 폭증을 일으킬 것이므로, 우린 그 대신 Fail1: 라벨로 분기해서, P0 의 `r3` 레지스터의 초기값 2를 가지고 이 모델을 종료시켜서 존재하는 단정문을 깨뜨리지 않게 합니다.

이 속임수가 항상 적용 가능한지에 대해선 논란이 있습니다만, 이게 실패하는 예를 전 아직 보지 못했습니다.

**Quick Quiz 12.27:**

Arm 리눅스 커널도 비슷한 버그를 가지고 있나요?

**Answer:**

Arm 은 `atomic_add_return()` 함수의 어셈블리어 구현 전후에 `smp_bm()` 를 위치하기 때문에 이 버그를 가지고 있지 않습니다. PowerPC 도 이 버그를 더이상 가지고 있지 않습니다; 이건 한참 전에 고쳐졌습니다 [Her11].

**Quick Quiz 12.28:**

Listing 12.23 provide sufficient ordering 의 라인 10 에 있는 `lwsync` 는 충분한 순서 규칙을 제공하나요?

**Answer:**

필요한 의미에 따라 다릅니다. 이 답의 나머지 부분은 Listing 12.23 의 P0 을 위한 어셈블리어가 값을 반환하는 어토믹 오퍼레이션을 구현할 것으로 여긴다고 가정합니다.

Chapter 15 에서 이야기 했듯, 리눅스 커널의 메모리 일관성 모델은 양쪽에서 모두 완전히 순서잡히기 위해 값을 반환하는 어토믹 RMW 오퍼레이션을 필요로 합니다. `lwsync` 로 제공되는 순서는 이 목적에 불충분하며, 따라서 `sync` 가 대신 사용되어야 합니다. 이 변경은 다른 두개의 리트머스 테스트를 [McK15f] 다뤘던 이메일 쓰 레드에 대한 대답의 일환으로 만들어졌습니다 [Fen15]. 리눅스 커널이 가지고 있을 수도 있는 모든 다른 버그의 반면은 독자 여러분의 연습문제로 남겨둡니다.

더 완화된 의미를 제공하는 다른 환경에서는 `lwsync` 가 충분할 수도 있습니다. 하지만 리눅스 커널의 값 반환 어토믹 오퍼레이션에서는 아닙니다!

**Quick Quiz 12.29:**

Listing 12.29 에 보인 것과 같은 리트머스 테스트를 `herd` 로 돌리기 위해선 뭘 해야 하나요?

**Answer:**

v4.17 (또는 그보다 최신의) 리눅스 커널 소스 코드를 가져오고, 필요한 도구들의 설치를 위해 `tools/memory-model/README` 의 설명을 따르세요. 이어서 여러분이 고른 리트머스 테스트를 위해 그 도구들을 수행시키기 위해 설명들을 더 따라가세요.

**Quick Quiz 12.30:**

왜 락킹을 직접 모델링해서 일을 복잡하게 만들죠? 어토믹 오퍼레이션을 이용해 간단히 락킹을 에뮬레이션 하는게 어떤가요?

**Answer:**

한 단어로 말하자면 Table E.4 에 보인 것과 같이 성능입니다. 첫번째 열은 모델링된 `herd` 프로세스의 수를 보입니다. 두번째 열은 `spin_lock()` 과 `spin_unlock()` 을 `herd` 의 `cat` 언어로 직접 모델링 했을 때 `herd` 수행시간을 보입니다. 세번째 열은 `spin_clock()` 을 `cpxchg_acquire()` 로, `spin_unlock()` 을 `smp_store_release()` 로 에뮬레이트 하고 `herd` filter 절을 락을 획득하는데 실패하는 수행을 취소시키기 위해 사용한 경우의 `herd` 수행시간을 보입니다. 네번째 열은 세번째 열과 같지만 `cpxchg_acquire()` 대신 `xchg_acquire()` 를 사용합니다. 다섯번째와 여섯번째 열은 세번째와 네번째 열과 같지만 락을 획득하는데 실패한 수행을 취소시키기 위해 `herd exists` 절을 사용합니다.

`filter` 절의 사용은 `exists` 절의 사용에 비해 두배 가까이 빠름에 역시 유의하시기 바랍니다. `filter` 절은 배제된 수행을 일찍 없애는 것을 가능하게 하므로

Table E.4: Locking: Modeling vs. Emulation Time (s)

# Proc.	Model	Emulate			
		filter		exists	
		cmpxchg	xchg	cmpxchg	xchg
2	0.004	0.022	0.027	0.039	0.058
3	0.041	0.743	0.968	1.653	3.203
4	0.374	59.565	74.818	151.962	500.960
5	4.905				

놀라운 일이 아닌데, 여기서 배제된 수행은 락이 두개 이상의 프로세스에 의해 동시에 획득된 경우들입니다.

더 중요한 것은, `spin_lock()` 과 `spin_unlock()` 을 직접 모델링하는 것은 락킹 에뮬레이션을 모델링 하는 것보다 다섯배에서 수백배 더 빠르다는 겁니다. 이 역시 놀라운 일이 아닐텐데, 직접 모델링 하는 것은 추상화의 수준을 높여서 `herd` 가 모델링 해야 하는 이벤트의 수를 줄이기 때문입니다. `herd` 가 하는 거의 모든 일은 폭발적 계산 복잡도를 가지므로, 이벤트 수의 약간의 감소가 수행시간의 폭발적으로 큰 감소를 가능하게 합니다.

따라서, 정형적 검증에서는 병렬 프로그래밍 그 자체에서보다도 훨씬 더, 분할하고 정복하세요!!!

□

Quick Quiz 12.31:

잠깐요!!! RCU read-side 크리티컬 섹션 바깥으로 포인터를 훌리는 건 치명적인 버그 아닌가요???

■

Answer:

맞아요, 그건 보통 치명적 버그입니다. 하지만, 이 경우 업데이트 쓰레드는 그런 포인터 누출을 적절히 처리하게끔 잘 구축되었습니다. 하지만 이런 일을 하는 버릇을 들이지는 마세요, 그리고 어떤 더 일반적인 접근법이 동작하게 하기 위한 충분한 생각없이는 특히나 이런 행위를 하지 마세요.

□

Quick Quiz 12.32:

Listing 12.32에서, 읽기 쓰레드는 왜 `P1()` 이 라인 45에서 `c` 를 0으로 만든 후 `c` 를 곧바로 읽어오고 나중에 이 같은 값을 `c` 에 그게 0이 된 직후에 저장하여 이 값 0으로 만들기 오퍼레이션을 무효화 하지 못하나요?

■

Answer:

그 읽기 쓰레드는 라인 24에서 다음 원소로 넘어갔으므로, 같은 원소로의 포인터를 읽은대로 저장하는 게 막아집니다.

□

Quick Quiz 12.33:

Listing 12.32에서, 라인 48 바로 전에 `synchronize_rcu()` 를 한번만 호출하지 않는 이유가 뭐죠?

■

Answer:

이는 `P0()` 가 메모리 해제된 원소에 액세스 하기 때문입니다. 하지만 제 말만 믿지 마시고 `herd` 를 써서 시험해 보세요!

□

Quick Quiz 12.34:

역시 Listing 12.32에서, 라인 48 는 `smp_store_release()` 대신 `WRITE_ONCE()` 를 사용할 수 없나요?

■

Answer:

훌륭한 질문입니다. 2021년 기준을, 답은 “아무도 모른다”입니다. 이는 Armv8 의 conditional-move 명령의 의미에 달려있습니다. 이 명령의 의미가 분명해지길 기다리는 동안은 `smp_store_release()` 가 안전한 선택일 겁니다.

□

Quick Quiz 12.35:

하지만 충분히 낫은 단계의 소프트웨어는 모든 의도와 목적에 있어 black hat 에 착취당하는데 면역이 있어야 하지 않나요?

■

Answer:

불행히도, 아닙니다.

한때 Paul E. McKenney 는 리눅스 커널 RCU 가 그런 공격으로부터 면역이 있다고 느꼈으나, Row Hammer 의 발전은 그렇지 않음을 그에게 보였습니다. 어쨌건, black hat 이 시스템의 DRAM 을 공격할 수 있다면, 모든 저수준 소프트웨어를 공격할 수 있는데, RCU 조차도 포함됩니다.

그리고 2018년, 이 가능성은 이론적 예측의 계곡에서 객관적 진실의 어렵고 빠른 계곡으로 옮겨졌습니다 [McK19a].

□

Quick Quiz 12.36:

L4 마이크로 커널의 전체 검증을 놓고 보면, 정형 검증에 대한 이 제한적 시선은 약간 시대에 뒤쳐진 것 아닌가요?

■

Answer:

불행히도, 아닙니다.

L4 마이크로 커널의 전체 검증은 학생별로는 느린 속도로 진행된, 많은 Ph.D. 학생들이 일일이 손을 통해 코드를 검증한 역작이었습니다. 대부분의 소프트웨어 프로젝트는 변경이 너무 빨리 만들어지기 때문에

이런 수준의 노력을 적용할 수 없습니다. 더 나아가서, L4 마이크로커널이 정형 검증의 시선에서 보기에는 거대한 소프트웨어 작품이지만, LLVM, GCC, 리눅스 커널, Hadoop, MongoDB, 그 외에도 수 많은 많은 프로젝트들에 비하면 아주 작습니다. 또한, 이 검증 역시 한계가 있는데, 연구자들도 이를 인정합니다: <https://docs.sel4.systems/projects/sel4/frequently-asked-questions.html#does-sel4-have-zero-bugs>.

정형 검증이 더 큰 수준의 자동화를 적용한 더 최근의 L4 검증을 포함해 어떤 성과를 마침내 거두기 시작하고 있지만 가시적 미래에 테스트를 완전히 대체할 기회는 현재로썬 없습니다. 그리고 전 이에 대해 틀렸기를 감히 바라겠습니다만, 그런 증명은 실제 소프트웨어를 검증하는 진짜 도구에 의한 형태이어야지, 화려한 수사의 몸통 형태여선 안될 겁니다.

아마도 언젠가 정형 검증은 검증을 위해 널리 사용될 텐데, 현재 회귀 테스트라 알린 것들도 포함할 겁니다. Section 17.4은 이 가능성을 현실로 만들기 위해 무엇이 필요할지 알아봅니다.

□

E.13 Putting It All Together

Quick Quiz 13.1:

레퍼런스 카운터가 0이 아닐 때에만 참조를 획득하는 간단한 compare-and-swap 오퍼레이션을 사용해 구현하는 건 어떤가요?

■

Answer:

이게 마지막 참조의 해제와 새 참조의 획득 사이의 경주는 해결할 수 있지만, 데이터 구조가 메모리 해제되고 어쩌면 심지어 어떤 완전 다른 타입의 구조로 재할당되는 것을 막기 위한 일은 전혀 하지 않습니다. “간단한 compare-and-swap 오퍼레이션”이 다른 타입의 구조에 적용되었을 때에는 정의되지 않은 결과를 낼 수 있을 겁니다.

요약하자면, compare-and-swap 같은 어토믹 오퍼레이션의 사용은 타입 안전성 또는 존재 보장을 필요로 합니다.

하지만 타입 변경을 허용할 필요가 있다면 어떨까요?

한가지 방법은 그런 타입 각자가 같은 위치에 레퍼런스 카운터를 갖게 해서, 재할당이 이 타입의 객체 그룹에서 수행된다면 모든게 안전하게 합니다. 이걸 C에서 하려면 각 구조의 레퍼런스 카운터를 사용될 때마다 주석을 다시기 바랍니다. C++에서는 상속과 템플릿을 사용하세요.

□

Quick Quiz 13.2:

한 CPU가 마지막 참조를 해제한 직후에 다른 CPU가 참조를 얻는 경우를 위한 보호는 왜 필요하지 않죠?

■

Answer:

합법적으로 또 다른 참조를 얻기 위해서는 그 CPU는 이미 참조를 가지고 있어야 하니까요. 따라서, 만약 어느 CPU가 마지막 참조를 해제했다면, 새 참조를 얻을 수 있는 CPU는 존재하지 않습니다!

□

Quick Quiz 13.3:

Listing 13.2의 라인 22에서 `atomic_sub_and_test()`가 수행된 직후에, 어떤 다른 CPU가 `kref_get()`을 호출했다고 해봅시다. 다른 CPU는 이제 해제된 객체로의 불법적인 참조를 갖게 되지 않나요?

■

Answer:

이 함수들이 올바르게 사용되면 그런 일은 일어날 수 없습니다. 여러분이 이미 참조를 가지고 있지 않다면 `kref_get()`을 호출하는건 불법인데, `kref_sub()`이 카운터를 0으로 감소시키지 못할 겁니다.

□

Quick Quiz 13.4:

`kref_sub()`이 0을 리턴해서 `release()` 함수가 호출되지 않았음을 알렸다고 해봅시다. 호출자는 어떤 조건에서 객체의 지속되는 존재를 믿을 수 있나요?

■

Answer:

호출자는 최소 하나의 참조가 계속 존재할 걸 아는게 아니라면 지속되는 존재를 믿을 수 없습니다. 일반적으로, 호출자는 이를 알 방법이 없으며, 따라서 `kref_sub()` 호출 후에 객체로의 참조를 하는 것을 주의깊게 막아야만 합니다.

관심 있는 독자 여러분은 이 제한을 RCU, 특히 `call_rcu()`를 사용해 해결해 보시기 바랍니다.

□

Quick Quiz 13.5:

간단하게 해제 함수로 `kfree()`를 넘기지 않는 이유가 뭐죠?

■

Answer:

일반적으로 `kref` 구조체는 더 큰 구조체에 내재되므로, 그리고 `kref` 필드만이 아니라 전체 구조체를 메모리 해제해야 하기 때문입니다. 이는 일반적으로 `container_of()`에 이어 `kfree()`를 호출하는 wrapper 함수로 정의됩니다.

□

Quick Quiz 13.6:

왜 0 레퍼런스 카운트 검사는 `then` 절에서 어토믹 값 증가를 하는 간단한 `if` 문을 통해서 이뤄질 수 없죠?

**Answer:**

`if` 조건이 레퍼런스 카운터 값은 1이라고 확인하고 완료되었다고 해봅시다. 이때 해제 오퍼레이션이 수행되어, 이 레퍼런스 카운터의 값을 0으로 감소시키고 따라서 정리 오퍼레이션을 시작합니다. 하지만 이제 `then` 절은 이 카운터를 다시 1로 증가시켜서 이 객체가 정리된 다음에 사용될 수 있게 합니다.

이 `use-after-cleanup` 버그는 `use-after-free` 버그 만큼이나 나쁩니다.

**Quick Quiz 13.7:**

대체 왜 우린 전역 락을 필요로 했죠?

**Answer:**

특정 쓰레드의 `__thread` 변수는 그 쓰레드가 종료될 때 사라집니다. 따라서 다른 쓰레드의 `__thread` 변수에 접근하는 모든 오퍼레이션을 쓰레드 종료와 동기화 시킬 필요가 있었습니다. 그런 동기화가 없다면 금방 종료된 쓰레드의 `__thread` 변수로의 접근은 segmentation fault 를 초래합니다.

**Quick Quiz 13.8:**

여보세요!!! Listing 13.5 의 라인 48 은 앞서서부터 존재한 `countarray` 구조의 값을 수정하잖아요! 이 구조체는 일단 `read_count()`에게 접근 가능해지면 일관적인 상태로 남는다고 하지 않았나요???

**Answer:**

실제로 그렇게 말씀드렸습니다. 그리고 `count_register_thread()` 가 `count_unregister_thread()` 처럼 새 구조체를 할당받게 하는 것도 가능할 겁니다.

하지만 이는 불필요합니다. `read_count()` 의 오류 범위의 유도는 메모리의 스냅샷에서 기초함을 기억하시기 바랍니다. 새 쓰레드는 값 0인 초기 `counter` 값으로 시작하므로, `read_count()` 의 수행 도중에 새 쓰레드를 넣는다 해도 그 유도식은 유지됩니다. 따라서, 흥미롭게도 새 쓰레드를 더할 때, 이 구현은 실제로 할당을 하지는 않지만 새 구조체를 할당하는 것과 같은 효과를 냅니다.

다른 한편, `count_unregister_thread()` 는 빠져나가는 쓰레드의 일이 중복으로 카운트 되게 할 수 있습니다. 이는 `read_count()` 가 라인 라인 65 와 라인 66

사이에서 호출될 때 발생 가능합니다. 이 중복 카운트를 막는 효율적인 방법들이 있습니다만, 그건 독자 여러분의 연습문제로 남겨두겠습니다.

**Quick Quiz 13.9:**

고정된 크기의 `counterp` 배열을 가지고, 이 코드는 어떻게 쓰레드의 수의 고정된 최대 한계 문제를 어떻게 회피하나요?

**Answer:**

맞아요, 그 배열은 실제로 고정된 최대 한계를 만듭니다. 이 한계는 쓰레드들을 userspace RCU [DMS¹²]에서 그려는 것처럼 링크드 리스트를 사용해 쓰레드를 추적하는 것으로 회피될 수 있습니다. 그와 비슷한 일을 하는건 독자 여러분의 연습문제로 남겨둡니다.

**Quick Quiz 13.10:**

와! Listing 5.4 의 42 라인 코드에 비해 Listing 13.5 는 70 라인 코드를 갖는군요. 이 여분의 복잡도는 그럴만한 가치가 있나요?

**Answer:**

이는 물론 경우에 따라 결정되어야 합니다. 여러분이 선형적으로 확장되는 `read_count()` 구현을 필요로 한다면, Listing 5.4 에 보인 락 기반 구현은 여러분에게 적합치 않을 겁니다. 그러나, `read_count()` 호출이 충분히 드물다면 락 기반 버전이 더 간단하고 따라서 더 나을 수도 있는데, 이 코드 크기 차이는 구조체 정의, 메모리 할당, 그리고 NULL 반환값 검사 때문이긴 합니다.

물론, 더 나은 질문은 “왜 언어 자체적으로 쓰레드간 `__thread` 변수 접근을 지원하지 않나요?” 일 겁니다. 어쨌건, 그런 구현은 락킹과 RCU 의 사용 모두를 불필요하게 할 겁니다. 이는 결국 Listing 5.4 에 보인 것보다도 더 간단한, 그러면서도 Listing 13.5 에 보인 구현의 확장성과 성능 이익을 유지한 구현을 가능하게 할 겁니다!

**Quick Quiz 13.11:**

하지만 Listing 13.9 에 보인 방법은 추가적인 캐쉬 미스를 일으켜서 추가적인 읽기 쪽 오버헤드를 일으키지 않나요?

**Answer:**

실제로 그럴 수 있습니다.

이 캐쉬 미스 오버헤드를 막는 한가지 방법이 Listing E.9 에 보여 있습니다: 단순히 `measurement` 구조체의 인스턴스를 `meas` 라는 이름으로 `animal` 구조체에

Listing E.9: Localized Correlated Measurement Fields

```

1 struct measurement {
2     double meas_1;
3     double meas_2;
4     double meas_3;
5 };
6
7 struct animal {
8     char name[40];
9     double age;
10    struct measurement *mp;
11    struct measurement meas;
12    char photo[0]; /* large bitmap. */
13 };

```

내장시키고 $\rightarrow\text{mp}$ 필드를 이 $\rightarrow\text{meas}$ 필드를 가리키게 하는 겁니다.

그럼 측정값 업데이트는 다음과 같이 될 수 있습니다:

1. 새로운 `measurement` 구조체를 할당하고 새 측정 값을 그 안에 넣습니다.
2. $\rightarrow\text{mp}$ 가 이 새 구조체를 가리키게 `rcu_assign_pointer()`를 사용합니다.
3. 하나의 grace period 가 지나기를 기다리는데, 예를 들어 `synchronize_rcu()` 나 `call_rcu()`를 사용할 수 있습니다.
4. 새 `measurement` 구조체로부터 측정값을 내장 $\rightarrow\text{meas}$ 필드로 복사합니다.
5. `rcu_assign_pointer()`를 사용해 $\rightarrow\text{mp}$ 가 다시 기존대로 내장 $\rightarrow\text{meas}$ 필드를 가리키게 합니다.
6. 또 하나의 grace period 를 기다리고, 새 `measurement` 구조체를 메모리 해제합니다.

이 방법은 일반적인 경우의 추가적 캐시 미스를 제거하기 위해 좀 더 무거운 업데이트 과정을 수행합니다. 이 추가적 캐시 미스는 업데이트가 진행중인 동안에만 일어날 겁니다.

□

Quick Quiz 13.12:

하지만 이 스캐닝은 크기 조절 가능 해쉬 테이블이 크기 재조정인 동안에는 어떻게 동작하나요? 그런 경우, 기존의 해쉬 테이블도 새 해쉬 테이블도 그 해쉬 테이블 내에 모든 원소를 가지고 있다고 보장되지 않습니다!

■

Answer:

사실입니다, Section 10.4 에 소개된 해쉬 테이블은 크기 재조정 중에 전체 스캐닝을 하지 못합니다. 한가지 간단한 해결책은 스캐닝 중에 `hashtab` 구조체의 $\rightarrow\text{ht_lock}$ 을 잡는 것이지만, 이는 두개 이상의 스캐닝이 동시에 진행되는 것을 막습니다.

또 다른 방법은 업데이트가 크기 재조정 진행 중에 새 해쉬 테이블만이 아니라 기존 해쉬 테이블도 수정하게 하는 겁니다. 이는 스캐닝이 기존 해쉬 테이블에서 모든 원소를 찾을 수 있게 합니다. 이를 구현하는 건 독자 여러분의 연습문제로 남겨둡니다.

□

E.14 Advanced Synchronization

Quick Quiz 14.1:

왜 C 와 C++ 같은 고전 언어를 더 현대적인 걸로 대체하지 않죠?

■

Answer:

그 더 현대적인 언어의 제안자가 그들 스스로의 컴파일러 백엔드를 작성할 만큼 에너지가 넘치지 않는다면 도움이 되지 않을 겁니다. 기존에 존재하는 백엔드를 재사용하는 일반적인 방법은 또한 수명이 종료된 객체로의 풍니터를 지원하기 거절하는 것과 같은 매력적인 속성 역시 재사용합니다.

□

Quick Quiz 14.2:

하지만 배터리 기반 시스템은 어떤가요? 그것들은 시스템 내로 흘러들어오는 전원을 필요로 하지 않잖아요.

■

Answer:

금방이든 나중이든 그 배터리는 재충전되어야 하는데 이는 시스템 내로의 전력 흐름을 필요로 합니다.

□

Quick Quiz 14.3:

하지만 queueing 이론에서의 결론을 놓고 보면, 낮은 사용율은 최악의 경우 응답시간을 개선하기보다는 평균 응답시간을 개선할 뿐이지 않나요? 그리고 최악의 경우 응답시간이 리얼타임 시스템에서 정말 신경쓰는 것인지 않나요?

■

Answer:

맞습니다, 그러나 ...

그런 queueing 이론 결과들은 리눅스 커널에서는 무한 태스크에 연관되는 무한의 “호출자”를 가정합니다. 2021년 초 기준으로 어떤 실제 시스템도 무한의 태스크를 지원하지 않으므로 무한의 호출자를 가정하는 결과는 무한보다는 작은 적용성을 가질 것이라 기대되어야 합니다.

다른 queueing 이론 결과들은 유한한 호출자를 갖는데, 깔끔하게 상한이 만들어진 응답시간을 갖습니다 [HL86]. 이 결과들은 실제 시스템을 더 잘 모델링 하며, 이 모델들은 사용율이 줄어들 때 평균과 최악의 경우 응답 시간이 모두 줄어들 것으로 예측합니다. 이 결과들은 락킹과 같은 동기화 메커니즘을 사용하는 동시성 시스템 모델링으로 확장될 수 있습니다 [Bra11, SM04].

요약하자면, 실제 세계의 리얼타임 시스템을 정확하게 묘사하는 queueing 이론은 최악의 경우의 응답시간이 사용율 감소에 따라 줄어듦을 보입니다.

□

Quick Quiz 14.4:

수십년간의 치열한 연구 덕에 정형 검증은 이미 상당히 가능합니다. 추가적인 발전이 정말로 필요한가요, 아니면 이는 그저 정형 검증의 위대한 힘을 게으르게 무시하기를 계속하려는 어떤 실제 일을 하는 사람의 변명인가요?

■

Answer:

이 상황은 어쩌면 실제 소프트웨어의 혼란스러운 세계에 뛰어드기를 막고자 하는 어떤 이론가의 변명일지도요? 어쩌면 좀 더 건설적으로 말해서, 다음과 같은 발전이 필요합니다:

1. 정형 검증은 더 큰 소프트웨어 제품을 처리할 필요가 있습니다. 가장 큰 검증 노력은 고작 10,000 라인의 코드의 시스템에 대해서만 수행되었으며, 그것들은 리얼타임 응답시간보다 훨씬 간단한 속성들에 대해서만 검증을 했습니다.
2. 하드웨어 제조사들은 정형적 타이밍 보장을 제공해야 합니다. 이는 하드웨어가 훨씬 더 간단했던 과거에는 일반적인 일이었으나, 오늘날의 복잡한 하드웨어는 최악의 경우 성능에 대한 무척 복잡한 표현을 초래했습니다. 불행히도, 전력 효율화에 대한 걱정은 제조사들을 이 복잡도를 심지어 더 늘리는 방향으로 떠밀고 있습니다.
3. 타이밍 분석은 개발 방법들과 IDE에 결합되어야 합니다.

그렇게 말했지만, 실제 컴퓨터 시스템의 메모리 모델을 정형화 한 최근의 작업들을 보면 [AMP¹¹, AKNT13] 희망이 있습니다. 다른 한편, 정형 검증은 리눅스의 수만개의 Kconfig 옵션의 조합에 의해 만들어질 수 있는 천문학적 수의 변종들에 대한 테스트에서의 문제만큼이나 많은 문제를 갖고 있습니다. 때로 삶은 어렵습니다!

□

Quick Quiz 14.5:

무엇이 “비 리얼타임 시스템과 어플리케이션에 의해 간단히 얻어질 수 있는가”에 따라 리얼타임을 비 리얼타임과 차별하는 건 우스운 일입니다! 그런 차이에 대한 이론적 기반은 분명 존재하지 않습니다!!! 더 나은 방법은 없을까요???

■

Answer:

이 구별은 인정하건대 엄격한 이론적 시점에서 만족스럽지 않습니다. 하지만 다른 한편으로, 이는 어플리케이션이 표준의 비 리얼타임 접근법을 이용해 싸고 쉽게 개발될 수 있는지, 또는 어렵고 비싼 리얼타임 접근법이 필요한지 결정하기 위해 개발자들이 필요한 것입니다. 달리 말하자면, 이론이 무척 중요하지만, 실용적 프로젝트를 완료해야 하는 우리들에게 이론은 실전을 지원할 뿐입니다.

□

Quick Quiz 14.6:

하지만 한번에 하나의 reader-writer 락 읽기 모드 획득만을 허용한다면, 그건 배타적 락과 동일한 거 아닌가요???

■

Answer:

실제로 그렇습니다, API 만 다를 뿐이죠. 그리고 이 API는 리눅스 커널이 -rt 패치셋을 말도 안되게 커지게 하지 않고도 리얼타임 기능을 제공하게 하므로 중요합니다.

그러나, 이 방법은 분명하고 큰 읽기 쪽 확장성 제한을 갖습니다. 리눅스 커널의 -rt 패치셋은 여러 이유로 이제한을 가지고도 사용될 수 있었습니다: (1) 리얼타임 시스템은 전통적으로 상대적으로 작았고, (2) 리얼타임 시스템은 일반적으로 프로세스 제어에 집중했으므로 I/O 서브시스템에서의 확장성 제한에 영향 받지 않았고, (3) 리눅스 커널의 많은 reader-writer 락은 RCU로 변환되었습니다.

그러나, 이 퀴즈를 뒤따르는 글에서 서령되듯이 동시에 읽기 쓰레드를 허용해야 할 분명한 필요가 드러났습니다.

□

Quick Quiz 14.7:

Listing 14.3 의 라인 12 의 `t->rcu_read_unlock_special.s`의 로드 직후에 `preemption`이 일어났다고 해봅시다. 그럼 그 태스크는 `rcu_read_unlock_special()`을 호출하지 못하게 될 수도 있고, 따라서 현재 `grace period`를 막고 있는 태스크들의 리스트에서 스스로를 제거할 수 없게 되어서 그 `grace period`가 무한히 연장되게 하지 않을까요?

■

Answer:

그건 실제 문제이고, RCU의 스케줄러 후킹을 통해 해결되었습니다. 해당 스케줄러 흑이 `t->rcu_read_lock_nesting` 값이 음수임을 확인하면 해당 컨텍스트 스위치가 완료되기 전에 필요하다면 `rcu_read_unlock_special()` 을 호출합니다.

**Quick Quiz 14.8:**

하지만 fail-stop 버그에도 불구하고 올바른 운영을 하는 게 가치 있는 fault-tolerance 속성 아닌가요?

**Answer:**

그렇기도 하고 아니기도 합니다.

Non-blocking 알고리즘이 fail-stop 버그에 대해 실패 내성을 제공한다는 점에서 맞습니다만, 이는 실용적인 실패 내성이 되기엔 상당히 부족하다는 점에서 아닙니다. 예를 들어, 여러분의 하나의 wait-free queue를 가지고 있으며 하나의 쓰레드가 원소 하나를 dequeue 했다고 해봅시다. 그 쓰레드가 이제 어떤 fail-stop 버그를 당하면, 지금 dequeue 된 그 원소는 실질적으로 잃어버린게 됩니다. 진정한 실패 내성은 non-blocking 속성 이상의 것을 필요로 하며, 이 책의 범위 밖입니다.

**Quick Quiz 14.9:**

이 리스트 앞에 “포함”이라는 단어를 봤습니다. 다른 제약들도 있나요?

**Answer:**

실제로, 그리고 많이 있습니다. 그러나, 그것들은 특정 상황에만 존재하는 경향을 보이며 많은 것들은 앞에 나열된 제약들의 수정본으로 여겨질 수 있습니다. 예를 들어, 데이터 구조의 선택에서의 많은 제약은 “특정 크리티컬 섹션에서 보내는 시간의 제한” 제약을 지키는 걸 도울겁니다.

**Quick Quiz 14.10:**

리얼타임 시스템은 종종 안전에 치명적인 어플리케이션을 위해 사용된다는 점, 그리고 수행시간 메모리 할당은 많은 안전에 치명적인 상황에서 금지되었다는 점을 놓고 볼 때, `malloc()`은 뭘 위한거죠???

**Answer:**

2016년 초, 수행시간 메모리 할당을 금지한 프로젝트들은 멀티쓰레드 컴퓨팅에 관심조차 없었습니다. 따라서 수행시간 메모리 할당은 안전에 치명적인 상황을 위한 추가적인 장애가 아닙니다.

그러나, 2020년에 이르러서는 멀티코어 리얼타임 시스템에서의 수행시간 메모리 할당이 약간 매력을 얻고 있습니다.

**Quick Quiz 14.11:**

`update_cal()`을 보호하기 위한 어떤 동기화가 필요하지 않나요?

**Answer:**

실제로 그렇고, 여러분은 이 책의 앞에서 이야기된 어떤 기법이든 사용할 수 있습니다. 그런 기법 중 하나는 단일 업데이트 쓰레드의 사용으로, Listing 14.6의 `update_cal()`에 보인 코드와 정확히 같은 결과를 낼 겁니다.



E.15 Advanced Synchronization: Memory Ordering

Quick Quiz 15.1:

이 챕터는 첫번째 판본 이후로 다시 작성되었습니다. 메모리 순서 규칙은 2014년 이후로 그렇게나 바뀌었나요?

**Answer:**

이전의 메모리 순서규칙 섹션은 2005년의 Linux Journal 기사 두편 [McK05a, McK05b]에 그 기원을 두고 있었습니다. 그 이후, C와 C++ 메모리 모델이 [Bec11] 정형화 되었고 (그리고 비평되었고 [BS14, BD14, VBC⁺15, BMN⁺15, LVK⁺17, BGV17]), 컴퓨터 시스템을 위한 수행 가능한 정형적 메모리 모델이 표준화 되었으며 [MSS12, McK11d, SSA⁺11, AMP⁺11, AKNT13, AKT13, AMT14, MS14, FSP⁺17, ARM17], 리눅스 커널을 위한 메모리 모델조차 존재하게 되었는데 [AMM⁺17a, AMM⁺17b, AMM⁺18], 이를 바탕으로 C11과 리눅스 메모리 모델의 차이를 설명하는 논문도 나왔습니다 [MWPF18].

2005년 이후의 이 모든 진보를 두고 보면, 완전 다시 작성할 시간이 된게 분명했습니다!

**Quick Quiz 15.2:**

컴파일러 또한 Listing 15.1의 쓰레드 `P0()`와 쓰레드 `P1()`의 메모리 액세스를 재배치 할 수 있습니다, 그렇죠?

**Answer:**

일반적으로 컴파일러 최적화는 CPU가 할 수 있는 것보다 더 확장적이고 심오한 재배치를 진행합니다. 하지만

이 경우, READ_ONCE() 와 WRITE_ONCE() 의 volatile 액세스는 컴파일러의 재배치를 막습니다. 그것 외에도 많은 다른 것들을 막으며, 따라서 이 챕터의 예제들은 READ_ONCE() 와 WRITE_ONCE() 를 널리 사용할 겁니다. READ_ONCE() 와 WRITE_ONCE() 의 필요에 대한 세부사항을 위해선 Section 15.3 를 참고하세요.



Quick Quiz 15.3:

기다려요!!! Table 15.1 의 두번째 열에서 x0 와 x1 둘 동시에 두 값, 즉 0 과 2를 갖습니다. 이게 어떻게 동작할 수 있죠???



Answer:

일을 간단하게 만들어주는 캐시 일관성 프로토콜이 아랫단에 있는데, Appendix C.2 에서 다뤄졌습니다. 하지만 동시에 두개의 값을 갖는 변수가 신기하다면, Section 15.2.1 까지 기다리세요!



Quick Quiz 15.4:

하지만 그 값들은 캐시에서 메인 메모리로도 비워져야 하지 않나요?



Answer:

놀라울 수 있지만, 꼭 그렇진 않습니다! 일부 시스템에서는 그 두 변수가 무척 많이 사용된다면, 그것들은 CPU 들의 캐시들 사이에서만 오갈뿐, 메인 메모리에는 결코 도착하지 않을 수도 있습니다.



Quick Quiz 15.5:

Table 15.3 의 열들은 꽤 무작위적이고 혼란스럽게 느껴지네요. 이 표의 개념적 기반이 있다면 무엇이죠???



Answer:

이 열들은 대략적으로 전력과 오버헤드를 증가시키는 하드웨어 메커니즘에 연관됩니다.

WRITE_ONCE() 열은 단일 변수로의 액세스는 항상 완벽하게 순서잡힌다는 사실을 보이는데, “SV” 행으로 나타내진 바와 같습니다. 여러 변수로의 액세스에 대한 순서를 제공하는 모든 다른 오퍼레이션들도 이 같은 변수 순서규칙을 제공함을 알아두십시오.

READ_ONCE() 열은 (2021년 기준으로) 컴파일러와 CPU 는 사용자에게 보이는 투기적 스토어를 사용하지 않아서 앞의 로드에 주소, 데이터, 또는 수행 측면에서 종속적인 모든 스토어는 이 로드가 완료된 후에 일어나는 것이 보장됩니다. 그러나, 이 보장은 이 의존성이 Sections 15.3.2 and 15.3.3 에서 설명된 것처럼 주의 깊게 구성되었을 것을 가정합니다.

“_relaxed() RMW operation” 열은 값을 반환하는 _relaxed() RMW 가 각각 READ_ONCE() 와 WRITE_ONCE() 만큼 훌륭한 로드와 스토어를 한다는 사실을 보입니다.

*_dereference() 열은 rcu_dereference() 와 그 친구들에 의해 제공되는 주소와 데이터 종속성 순서 규칙을 보입니다. 다시 말하지만 이 종속성은 Section 15.3.2 에서 설명한 것처럼 주의 깊게 구성되어야 합니다.

“Successful *_acquire()” 열은 많은 CPU 가 특별한 “acquire” 형태의 로드와 atomic RMW 인스트럭션을 가지고 있으며 많은 다른 CPU 는 앞의 로드를 뒤의 로드와 스토어에 대해 순서잡는 가벼운 메모리 배리어 명령을 가지고 있음을 보입니다.

“Successful *_release()” 열은 많은 CPU 가 특별한 “release” 형태의 스토어와 atomic RMW 인스트럭션을 가지고 있으며 많은 다른 CPU 들은 앞의 로드와 스토어를 뒤 따르는 스토어에 대해 순서잡는 가벼운 메모리 배리어 명령을 가짐을 보입니다.

smp_rmb() 열은 많은 CPU 가 앞의 로드를 뒤따르는 로드에 대해 순서잡는 가벼운 메모리 배리어 명령을 가짐을 보입니다. 비슷하게, smp_wmb() 열은 많은 CPU 가 앞의 스토어를 뒤따르는 스토어에 대해 순서 잡는 가벼운 메모리 배리어 명령을 가짐을 보입니다.

지금까지 설명한 순서 오퍼레이션들 중 어느 것도 앞의 스토어가 뒤따르는 로드에 대해 순서를 잡을 것을 요구하지 않는데, 이는 이 오퍼레이션은 실제 주요 목적은 앞의 스토어를 뒤따르는 로드에 대해 순서를 바꾸기 위함인 스토어 버퍼와 상관할 필요가 없음을 말합니다. 이 오퍼레이션의 가벼움은 정확히 스토어 버퍼에 상관하지 않는다는 정책 덕입니다. 그러나, 앞에서 이야기 되었듯 앞의 스토어가 뒤의 스토어와 순서 바꾸는 것을 방지하기 위해 스토어 버퍼와 상관해야 할 때가 생기는 데, 이게 이 표의 나머지 열을 필요하게 합니다.

smp_mb() 열은 Itanium 은 예외지만 대부분의 플랫폼에서 존재하는 전체 메모리 배리어에 연관됩니다. 그러나 Itanium 에서 조차 smp_mb() 는 Section 15.5.4 에서 이야기 되듯 READ_ONCE() 와 WRITE_ONCE() 에 대해서는 완전한 순서를 제공합니다.

“Successful full-strength non-void RMW” 열은 일부 플랫폼에서 (x86 등) atomic RMW 명령은 앞과 뒤에 모두 완전한 순서를 제공함을 보입니다. 따라서 리눅스 커널은 full-strength non-void atomic RMW 오퍼레이션이 성공했을 때에는 완전한 순서를 제공할 것을 필요로 합니다. (Full-strength atomic RMW operation 의 이름은 _relaxed, _acquire, 또는 _release 로 끝나지 않습니다.) 앞에서 이야기 되었듯, 이 오퍼레이션이 성공하지 못하는 경우는 “_relaxed() RMW operation” 열에서 다루어집니다.

그러나, 리눅스 커널은 `void` 나 `_relaxed()` `atomic` RMW 오퍼레이션에게 어떤 순서규칙도 요구하지 않으며, 그 표준 예가 `atomic_inc()`입니다. 따라서, 실패하는 non-void atomic RMW operation를 포함해 이 오퍼레이션은 앞이나 뒤의 모든 액세스에 대해 완전한 순서를 제공하기 위해선 앞에 `smp_mb__before_atomic()`를, 뒤에 `smp_mb__after_atomic()`를 배치해야 합니다. `smp_mb__before_atomic()` (또는, 비슷하게 `smp_mb__after_atomic()`)와 atomic RMW 오퍼레이션 사이에는 이 표의 `smp_mb__before_atomic()`과 `smp_mb__after_atomic()` 열의 “a” 항목으로 나타내어졌듯 어떤 순서도 제공될 이유가 없습니다.

요약하자면, 이 표의 구조는 Chapter 3에서 다루어진 물리 법칙에 의해 제한되는 아랫단 하드웨어의 특성에 기반해 만들어졌습니다. 즉, 이 표는 무작위적이지 않지만, 여러분이 혼란스러울 수는 있을 겁니다.



Quick Quiz 15.6:

왜 Table 15.3 는 `smp_mb__after_unlock_lock()` 과 `smp_mb__after_spinlock()` 을 포함하고 있지 않죠?



Answer:

이 두 기능은 특수한 경우를 위한 것이며 Table 15.3에 들어가기엔 조금 다른 것으로 보입니다. `smp_mb__after_unlock_lock()` 기능은 락 획득 직후에 놓여질 목적으로 만들어졌으며, 모든 CPU가 앞의 크리티컬 섹션에서의 액세스를 `smp_mb__after_unlock_lock()` 뒤의 액세스보다, 그리고 뒤의 크리티컬 섹션의 모든 액세스보다 전에 일어난 것으로 보게끔 보장합니다. 여기서 “모든 CPU”는 그 락을 쥐고 있지 않은 CPU 들 역시 포함하며 “앞의 크리티컬 섹션”은 해당 락을 위한 모든 앞의 크리티컬 섹션은 물론이고 `smp_mb__after_unlock_lock()` 을 수행한 CPU 에 의해 해제된 모든 다른 락을 위한 앞의 크리티컬 섹션까지 포함합니다.

`smp_mb__after_spinlock()` 은 `smp_mb__after_unlock_lock()` 과 동일한 보장사항을 제공하지만, `smp_mb__after_spinlock()` 을 수행한 CPU 에 의해 수행된 다른 액세스들에 대한 추가적 시각 보장을 제공합니다. 어떤 스토어 S 가 모든 앞의 락 획득 전에 수행되었고 어떤 로드 L 이 `smp_mb__after_spinlock()` 후에 수행되었다고 하면, 모든 CPU 는 S 를 L 전에 일어난 것으로 보게 됩니다. 달리 말하자면, 어떤 CPU 가 스토어 S 를 수행하고, 락을 획득하고 `smp_mb__after_spinlock()` 을 수행하고, 로드 L 을 수행하면, 모든 CPU 는 S 가 L 전에 수행된 것으로 보게 됩니다.



Quick Quiz 15.7:

하지만 특정 프로젝트가 이 경험적 법칙에 근거해 설계되고 작성될 수 있는지는 어떻게 하나요?



Answer:

이 챕터의 나머지 부분들이 정확히 그 질문에 답변하기 위해 만들어졌습니다!



Quick Quiz 15.8:

특정 사용처에 있어 어떤 메모리 배리어가 충분히 강력한지 어떻게 알 수 있죠?



Answer:

아, 그건 대답하는데 이 챕터의 나머지 대부분이 필요할 정도로 깊은 질문입니다. 하지만 짧게 대답하자면 `smp_mb()` 가 비용이 높긴 하지만 거의 항상 충분히 강합니다.



Quick Quiz 15.9:

기다려요!!! 리트머스 테스트를 자동으로 분석하는 도구를 어디서 구할 수 있나요???



Answer:

거기 필요한 도구들을 설치하기 위해선 v4.17 (또는 그보다 최신) 의 리눅스 커널 소스코드를 구하고 `tools/memory-model/README` 의 내용을 따르세요. 그리고 여러분이 고른 리트머스 테스트를 이 도구들로 수행하기 위해선 나머지 내용을 더 읽으세요.



Quick Quiz 15.10:

Listing 15.3 의 코드 조각이 가지고 있는 실제 하드웨어에서는 옳지 못할 수도 있는 잘못된 가정은 무엇입니까?



Answer:

이 코드는 특정 CPU 가 자신의 값은 보는 걸 멈추게 된다면 모두가 합의한 최종 값을 즉시 보게 될 거라 가정합니다. 실제 하드웨어에서, CPU 중 일부는 마지막 값으로 수렴하기 전의 중간 결과들을 여럿 볼 수도 있습니다. 이 섹션의 뒤에서 이야기될 그림의 데이터를 만드는 게 사용된 실제 코드는 따라서 조금 더 복잡합니다.



Quick Quiz 15.11:

어떻게 CPU들이 같은 변수에서 같은 시각에 다른 값을 볼 수 있습니까?

**Answer:**

Section 15.1.1에서 이야기한 바와 같이, 많은 CPU가 최근 저장된 값을 기록하는 스토어 버퍼를 가지는데, 이는 연관된 캐시 라인이 해당 CPU에 도달하기 전까지는 해당 값이 전역적으로 보이지 않게 합니다. 따라서, 각 CPU가 특정 변수에 대한 (자신의 스토어 버퍼 내에 있는) 자신의 값을 동일한 시각에 보고 있는게—그리고 메인 메모리는 또 다른 값을 가지고 있는게—상당히 가능합니다. 메모리 배리어가 발명된 이후 중 하나는 소프트웨어가 이런 상황을 우아하게 처리할 수 있게 하기 위함이었습니다.

당행히도, 소프트웨어는 여러 CPU가 같은 변수에 대해 여러 값을 볼 수도 있다는 사실에 별 신경을 쓰지 않습니다.

**Quick Quiz 15.12:**

CPU 2와 3은 왜 그리 빨리 동의에 도달했으며, CPU 1과 4는 그리 오래 걸렸나요?

**Answer:**

CPU 2와 3은 같은 코어 내의 한쌍의 하드웨어 쓰레드로, 동일한 캐시 계층을 공유하고 따라서 매우 낮은 통신 응답시간을 갖습니다. 이는 NUMA, 또는, 더 정확히 말하면 NUCA 효과입니다.

이는 왜 CPU 2와 3은 동의하지 않았는지에 대한 답을 이끌어 줍니다. 한가지 가능한 이유는 그것들 각각은 커다란 공유 캐시에 더해 작은 개별 캐시를 가질 수도 있다는 겁니다. 이 코드 내에 메모리 순서 잡기 오퍼레이션이 부재함과 동의하지 않는 짧은 10-나노세컨드 시간을 생각하면 또 다른 가능한 이유는 명령 재배치입니다.

**Quick Quiz 15.13:**

하지만 왜 load-load 재배치를 사용자에게 보이게 하죠? 중간에 스토어가 없는 흔한 경우에는 수행이 진행되게 끔 투기적 수행(speculative execution)을 하게 해서 순서 재배치가 안보이게끔 하는게 어떤가요?

**Answer:**

그럴 수 있고 많은 경우에 그렇습니다, 그러지 않으면 강한 순서규칙의 CPU는 실제로 느려질 거거든요. 그러나, 투기적 수행은 그것만의 단점이 있는데, 특히 투기적

수행이 자주 롤백되어야 하는 경우, 더구나 배터리 기반 시스템이면 그렇습니다. 하지만 미래의 시스템은 이 단점을 극복할 수 있을 겁니다. 그 전까지는, 벤더들이 완화된 순서규칙의 CPU를 계속 생산할 거라 예상할 수 있습니다.

**Quick Quiz 15.14:**

강한 순서규칙의 시스템은 왜 불필요한 `smp_rmb()`와 `smp_wmb()` 수행이라는 성능 비용을 지불해야 하죠? 완화된 순서규칙의 시스템이 그들의 잘못된 순서잡기 선택의 완전한 비용을 책임져야 하지 않나요???

**Answer:**

그게 정확한 실제 상황입니다. 강한 순서규칙의 시스템에서 `smp_rmb()`와 `smp_wmb()`는 명령을 생성하지 않지만 그저 컴파일러에게 제약을 겁니다. 따라서, 이 경우 완화된 순서규칙의 시스템은 실제로 그들의 메모리 순서 규칙 선택의 완전한 비용을 내고 있습니다.

**Quick Quiz 15.15:**

하지만 모든 플랫폼이 Listings 15.10 and 15.11의 `exists` 절을 정말로 발동시키지 않음을 어떻게 알죠?

**Answer:**

여기에 답하기 위해선 세개의 주요 플랫폼 그룹을 정의하는게 필요합니다: (1) Total-store-order (TSO) 플랫폼, (2) Weakly ordered (완화된 순서의) 플랫폼, 그리고 (3) DEC Alpha.

TSO 플랫폼은 모든 메모리 참조 쌍의 순서를 지키는 데 앞의 스토어와 뒤의 로드에 대해서만은 예외입니다. Listing 15.10의 라인 18과 19에서의 주소 종속성은 로드에 뒤이어 다른 로드가 오는 것이므로, TSO 플랫폼은 이 주소 종속성을 지킵니다. 이것들은 또한 Listing 15.11의 라인 17와 18에서의 주소 종속성 역시 지키는데 이는 로드 뒤에 스토어가 따라오는 것이기 때문입니다. 주소 종속성은 로드로부터 시작되어야 하므로, TSO 플랫폼은 암시적이지만 완벽하게 이를 지킵니다만 컴파일러 최적화는 가능하므로 `READ_ONCE()`는 필요합니다.

완화된 순서의 플랫폼은 연관없는 액세스의 순서는 지키지 않을 수 있습니다. 그러나, Listings 15.10 and 15.11의 주소 종속성은 연관없지 않습니다: 주소 종속성이 존재합니다. 하드웨어는 종속성을 추적하고 필요한 순서를 지켜줍니다.

이 완화된 순서규칙 플랫폼의 규칙에 대한 하나의 (유명한) 예외가 있는데, 그 예외는 DEC Alpha의 로드 대 로드 주소 종속성입니다. 그리고 이게 왜 v4.15

전의 리눅스 커널에서 DEC Alpha 는 이제는 사용되지 않는, Listing 15.10 의 라인 18 에 보인 `lockless_dereference()` 에 의해 메모리 배리어를 명시적으로 필요했던 이유입니다. 그러나, DEC Alpha 역시 로드 대 스토어 주소 종속성은 추적하는데, Listing 15.11 의 라인 17 에서 v4.15 전의 리눅스 커널에서도 `lockless_dereference()` 를 필요로 하지 않는 이유입니다.

정리하자면, 현재의 플랫폼들은 TSO 플랫폼들 (x86, mainframe, SPARC, ...) 에서처럼 주소 종속성을 묵시적으로 지켜주거나, 주소 종속성을 위한 하드웨어 추적 기능을 가지거나 (Arm, PowerPC, MIPS, ...), `READ_ONCE()` (v4.15 이후 리눅스 커널에서의 DEC Alpha) 또는 `rcu_dereference()` (v4.14 이전 리눅스 커널에서의 DEC Alpha) 에 의해 제공되는 메모리 배리어를 필요로 합니다.



Quick Quiz 15.16:

SP, MP, LB, 이제는 S 까지. 이 리트머스 테스트 약자들은 어디서 왔고 누가 이걸 따라갈 수 있겠습니까?



Answer:

최고의 일람표는 `test6.pdf` [SSA⁺¹¹] 입니다. 불행히도, 모든 약자가 SB (store buffering), MP (message passing), 그리고 LB (load buffering) 처럼 알기 쉬운 확장을 갖지는 않으나, 최소한 약자들의 리스트는 이미 사용 가능합니다.



Quick Quiz 15.17:

하지만 기다려요!!! Listing 15.12 의 라인 17 는 이 로드를 휘발성으로 기록하는 `READ_ONCE()` 를 사용하는데, 이는 컴파일러가 그 값이 나중에 0 으로 곱해짐을 안다 하더라도 로드 명령을 만들어야 함을 의미합니다. 그런데 어떻게 컴파일러가 이 데이터 종속성을 깨낼 수 있죠?



Answer:

맞습니다, 컴파일러는 휘발성 로드를 위해 로드 명령을 반드시 만들어야 합니다. 하지만 그 값을 0 으로 곱한다면 이 컴파일러는 그 곱셈의 결과를 0 으로 대체할 수가 있는데, 이는 많은 플랫폼에서 데이터 종속성을 깨버릴 겁니다.

더 나쁠 수 있는게, 종속된 스토어가 `WRITE_ONCE()` 를 사용하지 않는다면 컴파일러는 이를 로드 앞으로 옮겨버릴 수 있어서 TSO 플랫폼에서 조차 순서를 제공하지 못하게 할 수 있습니다.



Listing E.10: Litmus Test Distinguishing Multicopy Atomic From Other Multicopy Atomic

```

1 C C-MP-OMCA+o-o-o+o-rmb-o
2
3 {}
4
5 P0(int *x, int *y)
6 {
7     int r0;
8
9     WRITE_ONCE(*x, 1);
10    r0 = READ_ONCE(*x);
11    WRITE_ONCE(*y, r0);
12 }
13
14 P1(int *x, int *y)
15 {
16     int r1;
17     int r2;
18
19     r1 = READ_ONCE(*y);
20     smp_rmb();
21     r2 = READ_ONCE(*x);
22 }
23
24 exists (1:r1=1 /\ 1:r2=0)

```

Quick Quiz 15.18:

제어 종속성은 언어 표준에 의해 강제시 된다면 더 강해지지 않을까요?



Answer:

물론입니다! 그리고 아마도 충분한 미래에는 그렇게 될 겁니다.



Quick Quiz 15.19:

하지만 Listing 15.15 에서, P2() 의 r1 과 r2 가 값 2 와 1 을 각각 관측했고 P3() 의 r2 와 r4 가 값 1 과 2 를 각각 관측한다면 그저 나쁜 결과가 되지 않나요?



Answer:

맞아요, 그럴 겁니다. 그 결과를 확인하기 위해 `exists` 절을 수정하고 어떻게 되는지 한번 보세요.



Quick Quiz 15.20:

Multicopy atomic 과 other-multicopy atomic 에서 다른 행동을 보이는 구체적 예를 들어줄 수 있을까요?



Answer:

Listing E.10 (`C-MP-OMCA+o-o-o+o-rmb-o.litmus`) 이 그런 테스트를 보입니다.

Multicopy-atomic 플랫폼에서, P0() 의 라인 9 에서의 x 스토어는 y 로의 스토어 전에 P1() 을 포함한 모두에게 보여져야 합니다. 따라서, P1() 의 라인 19 에서의 y 로드가 값 1 을 리턴한다면, 라인 21 에서의 그것의 x

로드 그래야 하는데, 라인 20에서의 `smp_rmb()` 가 이 두 로드를 순서대로 수행되게 하기 때문입니다. 그러나, 라인 24의 `exists` 절은 multicopy-atomic 플랫폼에서 발동될 수 없습니다.

반대로, other-multicopy-atomic 플랫폼에서 `P0()` 는 자신의 스토어를 일찍 볼 수 있으며, 따라서 `P1()`에서의 두 스토어의 가시성의 순서에는 어떤 제약도 없어서 결과적으로 `exists` 절이 발동될 수 있습니다.



Quick Quiz 15.21:

그럼 누가 공유된 스토어 버퍼를 갖는 시스템을 설계하려 생각이나 하겠나요???



Answer:

이는 실제로 코어당 여러 하드웨어 쓰레드를 갖는 모든 시스템의 기본적 설계입니다. 하드웨어 관점에서는 자연스럽다는 겁니다!



Quick Quiz 15.22:

그러나 `P0()` 와 `P1()` 은 스토어 버퍼와 캐쉬를 공유하지만 `P2()` 는 자신만의 그것들을 하나씩 갖는다는건 공평하지 않지 않나요???



Answer:

아마도 Figure 15.8에 보인 것처럼 `P2()` 의 스토어 버퍼와 캐쉬를 공유하는 `P3()` 가 존재할 겁니다. 하지만 꼭 그래야 하는건 아니죠. 어떤 플랫폼은 다른 코어가 다른 수의 쓰레드를 불능화 시킬 수 있게 해서 하드웨어가 워크로드의 필요에 맞춰 조정될 수 있게 합니다. 예를 들어, 워크로드의 싱글쓰레드 기반 크리티컬 패쓰 부분은 하나의 쓰레드만 켜져 있는 코어에 할당되어서 이 싱글쓰레드가 이 워크로드의 부분이 해당 코어의 전체 능력을 사용하게 할 수 있습니다. 워크로드의 다른 더 병렬적이지만 캐쉬 미스에 취약한 부분은 모든 하드웨어 쓰레드가 켜져 있는 코어에 할당되어 더 나은 처리량을 제공할 수도 있을 겁니다. 이 향상된 처리량은 하나의 하드웨어 쓰레드는 캐쉬 미스로 인해 멈춰 있는 동안 다른 하드웨어 쓰레드는 진전을 낼 수 있다는 사실 때문일 수 있습니다.

그런 경우, 성능 요구사항이 인간 관점의 공평성을 넘어서합니다.



Listing E.11: R Litmus Test With Write Memory Barrier (No Ordering)

```

1 C C-R+o-wmb-o+o-mb-o
2
3 {}
4
5 P0(int *x0, int *x1)
6 {
7     WRITE_ONCE(*x0, 1);
8     smp_wmb();
9     WRITE_ONCE(*x1, 1);
10 }
11
12 P1(int *x0, int *x1)
13 {
14     int r2;
15
16     WRITE_ONCE(*x1, 2);
17     smp_mb();
18     r2 = READ_ONCE(*x0);
19 }
20
21 exists (1:r2=0 /\ x1=2)

```

Quick Quiz 15.23:

Table 15.4에서 `P1()` 의 스토어에 비해 `P0()` 의 스토어는 대체 왜 그렇게 느렸던 거죠? 달리 말하자면, Listing 15.16의 라인 28에서의 `exists` 절은 실제 시스템에서 정말로 발동되나요?



Answer:

이게 정말 발동될 수 있다는 진실을 받아들이십시오. Akira Yokosawa는 이 리트머스 테스트를 POWER8 시스템에서 수행하기 위해 litmus7 툴을 사용했습니다. 1,000,000,000 회의 수행 중 4회 `exists` 절이 발동되었습니다. 따라서, `exists` 절이 발동되는 것은 백만분의 일 확률이 아니고 일억분의 일 확률입니다. 그러나 이는 정말로 실제 시스템에서 발동됩니다.



Quick Quiz 15.24:

하지만 그 리트머스 테스트에 최소 세개의 쓰레드가 존재하지 않는다면 전파에 대해선 걱정할 필요가 없을 거예요, 그렇죠?



Answer:

틀렸습니다.

Listing E.11 (C-R+o-wmb-o+o-mb-o.litmus) 가 한 쌍의 쓰레드 사이에 store-to-store 와 load-to-store 연결만이 존재하기 때문에 전파를 필요로 하는 두개 쓰레드 리트머스 테스트를 보입니다. `P0()` 가 `smp_wmb()` 를 사용하고 `P1()` 이 `smp_mb()` 를 사용해 완전하게 순서 잡혀 있지만, 일시적이지 않은 연결의 본성은 라인 21의 `exists` 절이 정말 발동될 수 있음을 의미합니다. 이 발동을 막으려면, 라인 8의 `smp_wmb()` 는 `smp_mb()`

가 되어서 각 임시적이지 않은 연결에 한번씩 해서 두번 전파가 이루어지게 해야 합니다.



Quick Quiz 15.25:

`smp_mb()` 가 전파 속성을 갖는데 왜 Listing 15.18 의 라인 25에 있는 `smp_mb()` 는 왜 `exists` 절의 발동을 막지 않죠?



Answer:

대략적 경험적 법칙으로써, `smp_mb()` 배리어의 전파 속성은 프로세스 간의 하나의 load-to-store 연결에만 순서를 유지하기에 충분합니다. 불행히도, Listing 15.18은 하나가 아닌 두개의 load-to-store 링크를 갖는데, 첫 번째는 라인 17에서의 `READ_ONCE()`로부터 라인 24의 `WRITE_ONCE()`로의 것이며 두번째 것은 라인 26의 `READ_ONCE()`로부터 라인 7의 `WRITE_ONCE()`로의 것입니다. 따라서, `exists` 절의 발동을 막기 위해선 하나가 아닌 두개의 `smp_mb()` 가 필요합니다.

이 경험적 법칙에 대한 특수한 예외로, release-acquire 연결은 프로세스 간에 하나의 load-to-store 링크를 가질 수 있어서 여전히 사이클을 막을 수 있습니다.



Quick Quiz 15.26:

하지만 Listing 15.20 (C-2+2W+o-wmb-o+o-wmb-o. litmus)에 보인 것처럼 순서잡힌 스토어만 갖는 리트머스 테스트에서, 연구자들은 Arm과 Power 같은 완화된 순서 규칙의 시스템에서도 사이클이 금지됨을 보였습니다 [SSA¹¹]. 그런데 store-to-store 가 정말 항상 임시적이지 않나요???



Answer:

이 리트머스 테스트는 실제로 매우 흥미롭습니다. 이것의 순서는 일반적인 완화된 순서 규칙 하드웨어 설계에서도 자연스럽게 이루어지는 것으로 보인느데, 이는 연관된 물리 법칙과 캐쉬 일관성 프로토콜 수학으로부터의 소중한 선물로 여겨질 수 있겠습니다.

불행히도, 훨씬 나은 대안적 구현을 갖는 이 선물을 위한 소프트웨어 사용처를 누구도 찾지 못했습니다. 따라서, C11 도 리눅스 커널 메모리 모델도 Listing 15.20에 연관된 어떤 보장을 제공하지 않습니다. 이는 라인 19의 `exists` 절이 발동될 수 있음을 의미합니다.

물론, 배리어 없이는 순서 보장도 없는데 Listing E.12 (C-2+2W+o-o+o-o. litmus)에 보인 것처럼 실제 완화된 순서 규칙의 하드웨어에서조차 그렇습니다.



Listing E.12: 2+2W Litmus Test (No Ordering)

```

1 C C-2+2W+o-o+o-o
2
3 {}
4
5 P0(int *x0, int *x1)
6 {
7     WRITE_ONCE(*x0, 1);
8     WRITE_ONCE(*x1, 2);
9 }
10
11 P1(int *x0, int *x1)
12 {
13     WRITE_ONCE(*x1, 1);
14     WRITE_ONCE(*x0, 2);
15 }
16
17 exists (x0=1 /\ x1=1)

```

Listing E.13: LB Litmus Test With No Acquires

```

1 C C-LB+o-data-o+o-data-o+o-data-o
2
3 {}
4 x1=1;
5 x2=2;
6 }
7
8 P0(int *x0, int *x1)
9 {
10     int r2;
11
12     r2 = READ_ONCE(*x0);
13     WRITE_ONCE(*x1, r2);
14 }
15
16 P1(int *x1, int *x2)
17 {
18     int r2;
19
20     r2 = READ_ONCE(*x1);
21     WRITE_ONCE(*x2, r2);
22 }
23
24 P2(int *x2, int *x0)
25 {
26     int r2;
27
28     r2 = READ_ONCE(*x2);
29     WRITE_ONCE(*x0, r2);
30 }
31
32 exists (0:r2=2 /\ 1:r2=0 /\ 2:r2=1)

```

Quick Quiz 15.27:

Listing 15.21에 보인 것과 같은 종속성 만을 사용하는 리트머스 테스트를 만들 수 있습니까?



Answer:

Listing E.13은 무의미하지만 매우 실제적인 예입니다. 더 유용한(그러나 여전히 실제적인) 리트머스 테스트를 만드는 건 독자 여러분의 몫으로 남겨둡니다.



Quick Quiz 15.28:

Listing 15.25에 보인 것처럼 우리가 하나의 로드에서 스

토어로의 연결과 하나의 스토어에서 스토어로의 링크와 함께 짧은 release-acquire 연결을 갖는다고 해봅시다. 스토어에서 로드로의 연결이 아닌 종류의 것은 하나씩 만 있으므로 `exists`는 발동될 수 없습니다, 맞죠?



Answer:

틀렸습니다. 중요한 건 스토어에서 로드로의 것이 아닌 연결들의 갯수입니다. 단 하나의 스토어에서 로드로의 것이 아닌 연결이 있다면, release-acquire 연결은 `exists` 절이 발동되는 걸 막을 수 있습니다. 그러나, 두개 이상의 스토어에서 로드로의 것이 아닌 연결이 있다면 그게 스토어에서 스토어로든 로드에서 스토어로든 또는 어떤 조합이든간에 각 스토어에서 로드로의 것이 아닌 연결 사이에 완전한 배리어 (`smp_mb()` 또는 그보다 나은 것)가 필요합니다. Listing 15.25에서, `exists` 절이 발동되는 것을 막기 위해선 `P0()` 또는 `P1()` 액세스 사이에 완전한 배리어를 추가할 것이 필요합니다.



Quick Quiz 15.29:

스토어에서 로드로의 연결, 로드에서 스토어로의 연결, 그리고 스토어에서 스토어로의 연결이 있습니다. 하지만 로드에서 로드로의 연결은 어떻습니까?



Answer:

로드에서 로드로의 연결이라는 개념의 문제는 그 두개의 같은 변수로부터의 로드가 같은 값을 반환한다면, 그것들 사이의 순서를 파악할 방법이 없다는 겁니다. 그것들의 순서를 파악할 유일한 방법은 그것들이 다른 값을 반환했을 때로, 그 사이에 중간의 스토어가 있었을 때입니다. 그리고 그 중간의 스토어는 로드에서 로드로의 연결이 아니라 로드에서 스토어로의 연결 뒤에 스토어에서 로드로의 링크가 있었음을 의미합니다.



Quick Quiz 15.30:

컴파일러가 스토어를 만들어내는 걸 막기 위해 `barrier()` 호출을 평범한 스토어 직전에 위치시키는 건 어떤가요?



Answer:

그건 동작하지 않을 겁니다. 컴파일러가 `barrier()` 앞에 스토어를 만들어내는 건 막겠지만, `barrier()` 와 평범한 스토어 사이에 스토어를 만들어내는 건 막지 않습니다.



Listing E.14: Breakable Dependencies With Non-Constant Comparisons

```

1 int *gp1;
2 int *p;
3 int *q;
4
5 p = rcu_dereference(gp1);
6 q = get_a_pointer();
7 if (p == q)
8     handle_equality(p);
9 do_something_with(*p);

```

Listing E.15: Broken Dependencies With Non-Constant Comparisons

```

1 int *gp1;
2 int *p;
3 int *q;
4
5 p = rcu_dereference(gp1);
6 q = get_a_pointer();
7 if (p == q) {
8     handle_equality(q);
9     do_something_with(*q);
10 } else {
11     do_something_with(*p);
12 }

```

Quick Quiz 15.31:

Listing 15.26의 라인 6에서 `&reserve_int` 와 비교하기 전에 포인터를 그냥 역참조하지 않는 이유가 뭐죠?



Answer:

첫째로, `do_something_with()` 전에 `handle_reserve()` 를 호출해야 합니다.

하지만 메모리 순서에 더 밀접하게는, 컴파일러는 비교를 역참조 전으로 옮길 권리를 가져서, 컴파일러가 하드웨어가 종속성을 표시해두는 변수 `p` 대신 `&reserve_int` 를 사용할 수 있습니다.



Quick Quiz 15.32:

하지만 두 포인터 변수를 비교하는건 안전합니다, 그렇죠? 어쨌건, 컴파일러는 두 변수의 값을 모르는데 그 비교에서 뭘 알 수 있겠습니까?



Answer:

불행히도, 컴파일러는 여러분의 종속성 연결을 부수기 충분할 만큼 많은 걸 알 수 있는데, 예를 들어 Listing E.14에 보인 것과 같습니다. 컴파일러는 이 코드를 Listing E.15에 보인 것과 같이 변환할 권리를 가지며, `handle_equality()` 가 인라인화 되었으며 많은 레지스터를 필요로 한다면 레지스터 부족시에 이런 변화를 만들 수 있습니다. 이 변화된 코드의 라인 9는 `q` 를

사용하는데, 이는 `p` 와 동일하지만 하드웨어에 의해 종속성을 가져오는 것으로 표시되지 않았을 수 있습니다. 따라서, 이 변화된 코드는 라인 9 가 라인 5 다음으로 순서잡힐 것을 보장하지 않습니다.¹³

□

Quick Quiz 15.33:

하지만 라인 35에서의 조건은 라인 36를 라인 34 다음으로 순서짓는 제어 종속성을 제공하지 않나요?

■

Answer:

맞아요, 하지만 아닙니다. 맞아요, 제어 종속성이 존재합니다. 그러나 제어 종속성은 나중의 로드가 아니라 스토어만을 순서짓습니다. 여러분이 정말 순서를 필요로 한다면 라인 35 와 36 사이에 `smp_rmb()` 를 둘 수 있습니다. 또는 더 낫게는 `update()` 가 그 구조체들을 재사용하는 대신 할당하게 합니다. 더 많은 정보를 위해선 Section 15.3.3 를 참고하세요.

□

Quick Quiz 15.34:

Listing 15.30 의 `P1()` 에 `smp_mb()` 를 대신 추가할 순 없나요?

■

Answer:

리눅스 커널 메모리 모델에서라면 안됩니다. (시도해 보세요!) 그러나, 여러분은 대신 `P0()` 의 `WRITE_ONCE()` 를 일반적으로 `smp_mb()` 를 더하는 것보단 적은 오버헤드를 갖는 `smp_store_release()` 로 대체할 수 있습니다.

□

Quick Quiz 15.35:

하지만 PowerPC는 리눅스 커널에서 완화된 unlock-lock 순서 속성을 가져서 unlock 전의 쓰기가 lock 뒤의 읽기와 재배치될 수 있게 하지 않나요?

■

Answer:

맞습니다, 그러나 그 락을 잡지 않은 세번째 쓰레드의 관점에만 그렇습니다. 반대로, 메모리 할당자는 이 메모리를 읽기는 두 쓰레드에 대해서만 걱정하면 됩니다. 새로 옮겨진 메모리 블록으로의 액세스를 해야 하는 다른 쓰레드와의 올바른 동기화는 개발자의 책임입니다.

□

Quick Quiz 15.36:

잠깐만요! RCU 의 QSBR 구현에서는 `rcu_read_lock()` 과 `rcu_read_unlock()` 에 어떤 코드도 만들 어지지 않습니다. 이 말은 Listing 15.38 의 RCU read-side 크리티컬 섹션은 그저 비어있음을, 완전히 존재하지 않음을 의미합니다!!! 그런데 어떻게 존재하지도 않는 무언가가 순서에 대해 뭔가 효과를 낼 수 있죠???

■

Answer:

QSBR 에서 RCU read-side 크리티컬 섹션은 정말로 사라지지는 않습니다. 대신, 그것들은 quiescent state 가 만나지는 지점까지 양방향으로 확장됩니다. 예를 들어, 리눅스 커널에서 이 크리티컬 섹션은 최근의 `schedule()` 호출부터 다음 `schedule()` 호출까지 확장될 수 있습니다. 물론, 비 QSBR 구현에서, `rcu_read_lock()` 과 `rcu_read_unlock()` 은 순서를 제공하는 코드를 실제로 만듭니다. 그리고 리눅스 커널에서는 QSBR 구현 조차도 컴파일러 `barrier()` 를 `rcu_read_lock()` 과 `rcu_read_unlock()` 에 내포하는데, 이는 컴파일러가 `page fault` 를 일으킬 수도 있는 메모리 액세스를 RCU read-side 크리티컬 섹션 내로 옮기는 것을 막기 위해 필요합니다.

따라서, 이상하게 보일 수 있겠으나, 텅 빈 RCU read-side 크리티컬 섹션은 정말로 어떤 정도의 순서 규칙을 제공할 수 있고 제공합니다.

□

Quick Quiz 15.37:

Listings 15.36, 15.37, 그리고 15.38 의 리트머스 테스트들에 보인 `P1()` 의 액세스들은 그것들이 Listing 15.31 부터 Listing 15.32 까지에서와 같은 방식으로 재배치 될 수 있나요?

■

Answer:

안됩니다, 이 나중의 리트머스 테스트들 중 어느 것도 RCU read-side 크리티컬 섹션 내에 두개 이상의 액세스를 갖지 않기 때문입니다. 하지만 예를 들면 Listing 15.36에서 `P1()` 의 `WRITE_ONCE()` 를 그 크리티컬 섹션 내로, 그리고 `READ_ONCE()` 를 크리티컬 섹션 전으로 놓는식으로 액세스를 교체하면 어떻게 될까요?

이 액세스들을 교체하는 건 `r2` 의 두 인스턴스가 마지막 값 0을 갖는 걸 허용하는데, 달리 말하자면 RCU read-side 크리티컬 섹션의 순서 속성은 크리티컬 섹션 바깥으로 확장될 수 있으나 그것들의 재배치 속성에 대해서는 같지 않다는 말입니다. 이를 `herd` 로 검사하고 그 이유를 설명하는 건 독자 여러분의 몫으로 남겨 둡니다.

□

¹³ 이 예를 제공한 Linus Torvalds 에게 감사합니다.

Quick Quiz 15.38:

Listing 15.40 의 P2() 의 액세스 사이에 smp_mb() 가 위치되면 어떻게 될까요?

**Answer:**

또다시 사이클은 금지됩니다. 이에 대한 추가적 분석은 독자 여러분의 몫으로 남겨둡니다.

**Quick Quiz 15.39:**

어토믹 오퍼레이션과 smp_mb_after_atomic() 사이의 코드엔 무슨 일이 벌어지나요?

**Answer:**

일단, 그런 일을 벌이지 마세요!

하지만 만약 그런다면, 이 중간의 코드는 smp_mb_after_atomic() 전이나 후로 순서잡히는데, 아키텍쳐에 의존적이나, 둘 다에 배치될 수는 없습니다. 이는 또한 smp_mb_before_atomic() 과 smp_mb_after_spinlock() 에 적용되는데, 즉, 중간의 코드에는 불명확한 순서가 지어지며 그런 코드는 막아야 합니다.

**Quick Quiz 15.40:**

Alpha 의 READ_ONCE() 는 왜 rmb 가 아니라 mb 명령을 내포하나요?

**Answer:**

Alpha 는 mb 와 wmb 명령만을 가져서, 어떻게 하든 smp_rmb() 는 Alpha 의 mb 명령으로 구현될 겁니다. 또한, 리눅스 커널이 종속성 순서에 기대기 시작했던 시점에서는 Alpha 가 종속된 스토어의 순서를 지켜주는지 분명치 않았고, 따라서 smp_mb() 가 안전한 선택이었습니다.

그러나, 앞서 설명한 v5.9 의 READ_ONCE() 변경과 일부 Alpha 의 어토믹 read-modify-write 오퍼레이션들을 놓고 생각하면, 어떤 리눅스 커널 핵심 코드도 DEC Alpha 에 대해 걱정할 필요가 없으며, 따라서 커널에서 Alpha 지원을 제거하기 위한 Paul E. McKenney 의 동기를 크게 줄여줍니다.

**Quick Quiz 15.41:**

DEC Alpha 는 가능한 메모리 순서 규칙 중 가장 약한 것을 가지니 중요하지 않나요?

**Answer:**

DEC Alpha 가 상당한 비난을 받기는 하나, 같은 CPU에서 같은 변수로의 읽기들에 대한 재배치는 막습니다. 또한 Java 와 C11 메모리 모델을 괴롭히는 out-of-thin-air 문제도 막습니다 [BD14, BMN⁺15, BS14, Boe20, Gol19, Jef14, MB20, MJST16, Š11, VBC⁺15].

**Quick Quiz 15.42:**

하드웨어가 반쪽 메모리 배리어를 가질 수 있다면, 락킹 기능은 왜 컴파일러가 메모리 참조 명령을 락 기반 크리티컬 섹션 안으로 옮길 수 있게 하지 않죠?

**Answer:**

사실, Section 15.5.3 에서 보았고 Section 15.5.6 에서 보게 될 것이지만, 하드웨어는 정말로 부분적 메모리 순서 인스트럭션을 구현하며 이것들이 실제로 락킹 기능을 구축하는데 사용되었습니다. 그러나, 이 락킹 기능들은 완전한 컴파일러 배리어를 사용하며, 따라서 컴파일러가 메모리 참조 명령을 연관된 크리티컬 섹션의 안팎으로 재배치하는 것을 막습니다.

왜 컴파일러가 하드웨어에 의해 허용된 재배치를 하지 못하게 했는지 알기 위해 Listing E.16 의 예제 코드를 보십시오. 이 코드는 userspace RCU update-side 코드 [DMS⁺12, Supplementary Materials Figure 5] 에 기반합니다.

컴파일러가 라인 27와 28를 라인 29에서 시작하는 크리티컬 섹션 내로 재배치했다고 해봅시다. 이제 두 업데이트 쓰레드가 거의 같은 시간에 synchronize_rcu() 를 시작한다고 해봅시다. 그럼 다음과 같은 이벤트들이 가능할 겁니다:

1. CPU 0 이 라인 29에서 락을 획득합니다.
2. 라인 27 가 CPU 0 이 온라인임을 알게 되고, 따라서 라인 28에서 자신의 카운터를 비웁니다. (라인 27 와 28 가 컴파일러에 의해 라인 29 뒤로 재배치 되었음을 기억하세요).
3. CPU 0 이 30에서 update_counter_and_wait() 를 호출합니다.
4. CPU 0 이 라인 16에서 스스로 rCU_gp_ongoing() 을 호출하고 라인 5 가 CPU 0 이 quiescent state에 있음을 봅니다. 따라서 수행은 update_counter_and_wait() 로 돌아가고 라인 15 는 CPU 1 로 넘어갑니다.

Listing E.16: Userspace RCU Code Reordering

```

1 static inline int rcu_gp_ongoing(unsigned long *ctr)
2 {
3     unsigned long v;
4
5     v = LOAD_SHARED(*ctr);
6     return v && (v != rcu_gp_ctr);
7 }
8
9 static void update_counter_and_wait(void)
10 {
11     struct rcu_reader *index;
12
13     STORE_SHARED(rcu_gp_ctr, rcu_gp_ctr + RCU_GP_CTR);
14     barrier();
15     list_for_each_entry(index, &registry, node)
16     {
17         while (rcu_gp_ongoing(&index->ctr))
18             msleep(10);
19     }
20
21 void synchronize_rcu(void)
22 {
23     unsigned long was_online;
24
25     was_online = rcu_reader.ctr;
26     smp_mb();
27     if (was_online)
28         STORE_SHARED(rcu_reader.ctr, 0);
29     mutex_lock(&rcu_gp_lock);
30     update_counter_and_wait();
31     mutex_unlock(&rcu_gp_lock);
32     if (was_online)
33         STORE_SHARED(rcu_reader.ctr, LOAD_SHARED(rcu_gp_ctr));
34     smp_mb();
35 }

```

5. CPU 1 이 synchronize_rcu() 를 수행하지만 CPU 0 이 이미 락을 잡고 있으므로, CPU 1 은 이 락이 사용 가능해지길 기다립니다. 컴파일러는 라인 27 와 28 를 29 뒤로 재배치했으므로 CPU 1 은 온라인이면서도 자신의 카운터를 지우지 않습니다.
6. CPU 0 이 라인 16 에서 CPU 1 위에서 rcu_gp_ongoing() 을 호출하고 라인 5 는 CPU 1 이 quiescent state 에 있지 않음을 보게 됩니다. 따라서 라인 16 의 while 반복문은 결코 종료되지 않습니다.

그리므로 컴파일러의 재배치가 deadlock 을 초래합니다. 반면, 하드웨어 재배치는 일시적이어서 CPU 1 은 라인 29 에서의 mutex 획득 시도를 라인 27 와 28 수행 전에 차수할 수 있지만, 결국은 라인 27 와 28 를 수행할 겁니다. 하드웨어 재배치는 짧은 지연을 초래할 뿐이므로, 견뎌질 수 있습니다. 다른 한편, 컴파일러 재배치는 deadlock 을 초래하므로 반드시 막아져야 합니다.

어떤 연구 노력은 컴파일러가 더 적극적인 재배치를 안전하게 할 수 있도록 하드웨어 transactional memory 를 사용했습니다만, 하드웨어 transaction 의 오버헤드는

아직까지는 그런 최적화를 충분히 매력적이진 못하게 했습니다.

Quick Quiz 15.43:

왜 store-to-load 가 아닌 load-to-store 와 store-to-store 링크에는 보다 무거운 순서 오퍼레이션을 사용해야만 하죠? 대체 무엇이 store-to-load 링크를 그렇게 특별하게 합니까???

**Answer:**

Load-to-store 와 store-to-store 링크는 Section 15.2.7.2 의 Figures 15.10 and 15.11 에서 설명된 것처럼 임시적이지 않을 수 있음을 기억하십시오. 이 load-to-store 와 store-to-store 링크의 반 임시성은 강력한 순서 규칙을 필요로 합니다.

대조적으로, store-to-load 링크는 Listings 15.12 and 15.13 에 보인 것처럼 임시적입니다. 이 store-to-load 링크의 임시성이 최소한의 순서 규칙만 지키는 것을 허용합니다.

**E.16 Ease of Use****Quick Quiz 16.1:**

원소의 삭제에도 비슷한 알고리즘을 사용할 수 있나요?

**Answer:**

그렇습니다. 그러나, 각 쓰레드는 가운데의 원소를 제거하기 위해 세개의 연속된 원소의 락을 잡아야 하므로, N 개 쓰레드가 있다면, 데드락을 막기 위해 그 리스트에는 $N + 1$ 개가 아니라 $2N + 1$ 개 원소가 있어야 합니다.

**Quick Quiz 16.2:**

와우! 도대체 무엇이 누군가로 하여금 이것 만큼 깎아낼 가치가 있는 알고리즘을 생각해내게 할까요???

**Answer:**

그건 Paul 일 겁니다.

그는 다섯명의 철학자가 참석하는 이상한 스파게티 저녁식사를 포함하는 식사하는 철학자들의 문제를 생각하고 있었습니다. 테이블에 다섯개의 쟁반에 다섯개의 포크만이 있다는 점으로 보아, 그리고 각 철학자는 식사를 위해 한번에 두개의 포크를 필요로 한다는 점으로 보아, 누군가는 데드락을 회피하는 포크 할당 알고리즘을 생각해낼 겁니다. Paul 의 답은 “윽! 그냥 포크 다섯개를 더 줘!”였습니다.

이것 자체로는 괜찮았습니다만, Paul 은 이 동일한 해결책을 순환 링크드 리스트에 적용했습니다.

이것 역시 크게 나쁘지 않았습니다만 그는 누군가에게 이를 이야기 해야만 했습니다!



Quick Quiz 16.3:

이 규칙에 대한 예외를 하나 주세요.



Answer:

이에 대한 예외 하나는 특정 상황에 동작하는 유일한 것으로 알려진 어렵고 복잡한 알고리즘을 겁니다. 또 다른 예외는 해당 상황에서 동작하는 것으로 알려진 것들 중 가장 간단한, 그러나 여전히 어렵고 복잡한 알고리즘을 겁니다. 그러나, 이 경우에도, 더 간단한 알고리즘을 찾기 위해 약간의 시간을 사용하는게 매우 가치있을 겁니다! 어쨌건, 어떤 작업을 위한 첫번째 알고리즘을 발명해 냈다면, 더 간단한 것을 발명하는게 그렇게 어렵지만은 않을 겁니다.



E.17 Conflicting Visions of the Future

Quick Quiz 17.1:

하지만 어떤 어플리케이션이 파일로 매핑된 메모리 지역에 있는 `pthread_mutex_lock()` 을 쥔 채로 종료된다면요?



Answer:

실제로, 이 경우 그 락은 지속될 것이며, 이 락을 획득하려 시도하는 다른 프로세스들은 더이상 존재하지 않는 프로세스에 의해 그 락이 이미 잡혀 있음에 놀라게 될 겁니다. 이는 파일에 매핑된 메모리 영역에 위치한 `pthread_mutex` 객체를 사용할 때 세심한 주의가 필요한 이유입니다.



Quick Quiz 17.2:

`mmap()` 메모리 영역의 데이터 구조로 표현되는 비지속적 기능은 어떻습니까? 크리티컬 섹션이나 그런 기능 내에서의 `exec()` 이 존재하면 어떻게 되죠?



Answer:

만약 그 `exec()` 된 프로그램이 메모리의 같은 영역을 매핑한다면, 이 프로그램은 이론적으로 그 락을 간단히

해제할 수 있습니다. 이 방법이 소프트웨어 엔지니어링 관점에서 말이 되는가에 대한 질문은 독자 여러분의 연습 문제로 남겨두겠습니다.



Quick Quiz 17.3:

MV-RLU는 상당히 좋아 보입니다! 이게 RCU를 무찌를 수 있을까요?



Answer:

그 초록을 빠르게 읽었다면 그런 인상을 받을 수 있겠습니다만, 더 주의 깊은 독자들은 마지막 문장의 “여러 워크로드에 대해” 부분을 볼 겁니다. 이 부분은 상당히 중요한 것으로 드러났습니다:

1. 그들의 RCU 성능 평가는 동기성 grace period 를 사용하는데, 이는 필요 없이 업데이트를 멈추게 하는데, 그들의 섹션 6.2.1에 설명되어 있습니다. 덕이 깊은 비동기 `call_rcu()` 기능의 사용이 RCU 를 많은 수의 업데이트 쓰레드와 상당히 잘 동작하고 확장되게 한다는 점을 보기 위해 이 책의 Figure 10.11를 읽어보시기 바랍니다. 더 나아가, 그들의 논문의 섹션 3.7에서, 저자들은 비동기 grace period 가 MV-RLU 확장성을 위해 중요하다고 인정합니다. 공정한 비교는 RCU 또한 비동기성의 이득을 얻게 해줘야 합니다.
2. 그들은 잘 튜닝되지 않은, 10,000 개 원소를 담은 1,000 개 버킷 해쉬 테이블을 사용합니다. 또한, 그들의 448 하드웨어 쓰레드는 그들이 그들의 벤치 마크에서 RCU 성능을 제한한다고 올바르게 명시하는 락 경쟁을 막기 위해 1,000 개보다 훨씬 많은 버킷이 필요합니다. 유용한 비교는 올바르게 튜닝된 해쉬 테이블을 사용할 겁니다.
3. 그들의 RCU 해쉬 테이블은 버킷별 락을 사용했는데, 그들은 이를 병목지점이라 했고, 이는 긴 해쉬 채인과 작은 버킷 대 쓰레드 비율을 볼 때 놀랍지 않습니다. 이와 경쟁하는 그들의 다른 여러 메커니즘들은 그대신 lockfree 기법을 사용하고, 따라서 버킷별 락 병목을 회피하며, 이는 냉소적인 사람들은 그러지 않고는 설명할 수 없는 그들의 제대로 튜닝되지 않은 해쉬 테이블의 사용에 빛을 비춰야한다고 할수도 있겠습니다. 해당 논문 Figure 4의 가운데 열의 첫번째 그래프는 인위적 병목에 붙잡히지 않는다면 무엇을 이를 수 있는지 보이며, 같은 열의 두번째 그래프의 첫번째 부분 또한 그립니다.
4. 그들의 링크드 리스트 오퍼레이션은 RLU 가 리스트의 다른 원소들에 동시적 수정을 가하는 걸 허용하지만, RCU 는 직렬화된 업데이트를 강제합니다.

다시 말하지만, RCU 는 lockless 업데이트 쓰레드와 결합되어서도 항상 잘 동작해 왔으며, 이 저자들이 인용한 [DMS⁺12] 학계의 논문에서도 설명된 사실입니다. 공정한 비교는 MV-RLU 에서와 같은 형태의 업데이트를 RCU 에 사용해야 할 겁니다.

5. 저자들은 리눅스 커널에서 읽기 쓰레드들에게 여러 포인터 업데이트 사이에 일관적인 모습을 보이기 위해 사용되는 RCU 와 시퀀스 락킹을 조합을 고려하지 못했습니다.
6. 저자들은 읽기 쓰레드에게 완화된 형태의 일관성이긴 하지만 다수 포인터 업데이트의 일관적 모습을 제공하는 Issaguah Challenge [McK16] 에의 RCU 기반 해결책을 고려하지 못했습니다.

이 논문의 익명 리뷰어들이 MV-RLU 와 RCU 의 apples-to-apples 비교를 요구하지 않았다는 건 놀랍습니다. 그러나, 저자들은 강력한 읽기 쪽 일관성과 결합된 훌륭한 확장성이라는 너무 희귀한 예를 선보이는 학술 논문을 만들어냈음에 축하받아야 합니다. 그들은 또한 학계의 비동기적 grace period 에 대한 전통적 편견을 이겨냈음에 축하받을 겁니다. 이는 그들의 확장성을 크게 개선시킬 겁니다.

흥미롭게도, RLU 와 RCU 는 Hagit Attiya 등 [AHM09]에 의해 설명된 STM 의 본성적 한계를 다른 방법으로 회피합니다. RCU 는 엄격한 serializability 를 제공하는 걸 회피하고 RLU 는 보이지 않는 읽기 전용 트랜잭션을 제공하는 걸 회피하는데, 따라서 둘 다 그 한계를 회피합니다.



Quick Quiz 17.4:

spin_trylock() 같은게 존재하는데 TM 이 실패의 컨셉을 가져왔다고 말하는게 말이 됩니까???



Answer:

락킹을 사용할 때, spin_trylock() 은 선택으로, 실패에서 자유로운 선택은 spin_lock() 이 되는데 이는 흔한 경우로 v5.11 리눅스 커널에서 spin_lock() 호출이 spin_trylock() 호출의 100배가 넘음으로 알 수 있습니다. TM 을 사용할 때에는 실패로부터 자유롭는 선택은 취소 불가 트랜잭션 뿐으로, 이는 흔한 경우에 사용되지 않습니다. 사실, 취소 불가 트랜잭션은 모든 TM 구현에서 사용 가능하지도 않습니다.



Quick Quiz 17.5:

무얼 배우나요? 이 웃긴 섹션에 나열된 웃긴 많은 특수 경우들 같은 드문 경우들을 위한 메모리 기반 데이터 구조와 락킹에 TM 을 그냥 사용하는게 어떤가요???



Answer:

2005년으로부터 방금 전화를 받았는데, TM 에 대한 큰 광고를 원한다고 합니다.

2021년, TM 은 곧 따라올 Section 17.3 에서 다룰 HTM 의 발전에도 불구하고 해야 할 것들이 많습니다.



Quick Quiz 17.6:

왜 락 변수와 보호되는 변수가 캐쉬 라인을 공유하는 일반적 경우가 중요하죠?



Answer:

그 락이 자신이 보호하는 변수들과 같은 캐쉬라인에 있다면, 한 CPU 에 의한 이 변수들로의 쓰기는 모든 다른 CPU 의 캐쉬 라인을 무효화 시킬 겁니다. 이 무효화는 큰 수의 충돌과 재시도를 발생시키는데, 이는 아마 락킹에 비해서도 성능과 확장성을 더 떨어뜨릴 수도 있습니다.



Quick Quiz 17.7:

HTM 의 성능과 확장성에 있어 상대적으로 적은 업데이트가 왜 중요한가요?



Answer:

업데이트가 클수록, 충돌의 가능성성이 커지며, 따라서 재시도의 가능성이 높아지는데, 이는 성능을 떨어뜨립니다.



Quick Quiz 17.8:

동기화 메커니즘의 선택에 관계 없이 red-black 트리가 어떻게 모든 원소를 효율적으로 접근할 수 있죠???



Answer:

많은 경우, 그 접근은 정확할 필요가 없습니다. 이 경우, 읽기 쓰레드를 보호하기 위해 어떤 삽입이나 삭제와도 낮은 충돌 확률을 제공하는 해저드 포인터나 RCU 를 사용할 수 있을 겁니다.



Quick Quiz 17.9:

하지만 디버거가 트랜잭션의 성공한 라인에 브레이크포인트를 설정하고 앞의 트랜잭션 수행의 단계를 다시 추적하길 재시도하는 방법으로 단계별 수행을 애뮬레이션하는 건 왜 안되죠?

**Answer:**

이 방법은 높은 확률로 동작할 수도 있습니다만, 많은 사용자들에게 무척 놀라운 방식으로 실패할 수 있습니다. 이를 자세히 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 라인 4에 걸리면 트랜잭션을 중단시키고 디버거에 진입할 브레이크포인트를 설정했다고 해봅시다. 브레이크포인트가 걸린 시각과 디버거가 모든 쓰레드를 중단시키는 시각 사이에 어떤 다른 쓰레드가 a의 값을 0으로 설정합니다. 불쌍한 사용자가 이 프로그램을 한단계 더 움직이려 하면, 놀랍게도! 이 프로그램은 이제 then-절이 아니라 else-절을 수행합니다.

이는 제가 사용하기 쉬운 디버거라 부르는 게 아닙니다.

**Quick Quiz 17.10:**

하지만 누가 빈 락 기반 크리티컬 섹션을 필요로 합니까???

**Answer:**

Section 7.2.1의 Quick Quiz 7.18의 답을 보시기 바랍니다.

그러나 진행 보장이 없이 강력한 원자성을 제공하는 HTM 구현 때문에 빈 크리티컬 섹션에 기반한 모든 메모리 기반 락킹 설계는 transactional lock elision의 존재에도 올바르게 동작할 거라는 주장이 있습니다. 저는 이 말의 증명을 보지 못했지만, 이 주장에는 단순한 이유가 있습니다. 요점은 강력하게 원자적인 HTM 구현에서, 특정 트랜잭션의 결과는 이 트랜잭션이 성공적으로 완료되기 전까지는 보이지 않는다는 겁니다. 따라서, 여러분이 어떤 트랜잭션의 시작된 걸 본다면, 이는 이미 완료되었음이 보장되어서 뒤따르는 빈 락 기반 크리티컬 섹션은 그걸 “기다린다”는 겁니다—어쨌건, 기다림이 필요치 않으니까요.

이 근거는 완화된 원자적 시스템에서는 (많은 STM 구현이 포함됩니다) 성립되지 않으며, 통신에 메모리 이외의 수단을 사용하는 락 기반 프로그램에도 적용되지 않습니다. 그런 방법들에는 시간의 흐름 (예를 들어, 하드 리얼타임 시스템)이나 우선순위의 흐름이 (예를 들어, 소프트 리얼타임 시스템) 있습니다.

우선순위 높이기에 기반하는 락킹 설계는 특히 흥미로울 겁니다.

**Quick Quiz 17.11:**

Transactional lock elision은 간단히 빈 락 기반 크리티컬 섹션을 제거하지 않음으로써 락킹의 시간 기반 메세징 의미를 다룰 수 있지 않을까요?

**Answer:**

그럴 수 있겠습니다만, 이는 불필요하고 불충분할 겁니다.

이 텅빈 크리티컬 섹션의 조건적 컴파일 때문이었다면 이는 불필요합니다. 이 락의 목표는 데이터 보호 뿐이었다면 이를 완전히 제거하는게 올바른 일일 겁니다. 실제로, 빈 락 기반 크리티컬 섹션을 놔두는 건 성능과 확장성을 떨어뜨릴 겁니다.

다른 한편, 비지 않은 락 기반 크리티컬 섹션의 락킹의 데이터 보호와 시간 기반 메세징 의미에 의존하는 게 가능합니다. 그런 경우에 transactional lock elision을 사용하는 것은 올바르지 않으며 버그를 초래할 겁니다.

**Quick Quiz 17.12:**

현대의 하드웨어에서 [MOZ09] 어떻게 시간에 기반한 병렬 소프트웨어가 동작할 거라 생각될 수 있습니까?

**Answer:**

짧게 답하자면 일반적인 공간의 일반적 하드웨어에서는 모든 세밀한 타이밍에 기반한 동기화 설계는 무모하며 모든 상황에서 올바르게 동작할 거라 예상될 수 없다는 겁니다.

그렇다고는 하나, 훨씬 결정론적인 하드 리얼타임에서의 사용을 위해 설계된 시스템이 있습니다. 그런 시스템을 사용하는 (매우 드물) 경우, 어떻게 시간 기반 동기화가 동작할 수 있는지 보이는 작은 예가 있습니다. 다시 말하지만, 일반적 마이크로프로세서에서는 상당히 비결정적인 성능 특성이 있으므로 이를 시도하지 마세요.

이 예는 하나의 제어 쓰레드와 여러 일꾼 쓰레드를 사용합니다. 각 일꾼 쓰레드는 바깥으로의 데이터 넘기기를 하고 각 일단위를 마친 후 쓰레드별 my_timestamp

변수에 현재 시간을 (예를 들어, `clock_gettime()` 시스템콜로부터 얻을 수 있겠습니다) 기록합니다. 이 예의 리얼타임 특성은 다음과 같은 제한을 초래합니다:

1. 특정 일꾼 쓰레드가 `MAX_LOOP_TIME` 이상의 시간 동안 이 시간 기록을 업데이트하지 못한다면 이는 치명적 오류입니다.
2. 락은 전역 상태를 접근하고 업데이트하기 위해 아껴서 사용됩니다.
3. 락은 주어진 쓰레드 우선순위 내에서 엄격한 FIFO 순서로 얻어집니다.

일꾼 쓰레드가 데이터 넘기기를 완료하면, 이들은 스스로를 이 어플리케이션의 나머지 부분으로부터 떼어내고 상태 값을 `-1`로 초기화 되는 쓰레드별 `my_status` 변수에 저장합니다. 쓰레드는 끝나지 않습니다; 대신 이들은 나중의 처리 요구를 위해 쓰레드 풀에 위치합니다. 제어 쓰레드는 필요에 따라 일꾼 쓰레드를 할당하고, 또한 쓰레드 상태들의 히스토그램을 유지합니다. 이 제어 쓰레드는 일꾼 쓰레드의 그것보다 더 높지 않은 리얼타임 우선순위로 수행됩니다.

일꾼 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            /* Bug if TLE! */
12        }
13        /* Repurpose threads as needed. */
14    }

```

라인 5는 쓰레드가 종료되었는지 추측하기 위해 시간의 흐름을 사용하며, 그렇다면 라인 6와 10를 수행합니다. 라인 7와 8의 빈 락 기반 크리티컬 섹션은 종료 중인 모든 쓰레드가 완료되었길 보장합니다(락은 FIFO 순서로 주어짐을 기억하세요!).

다시 말하지만, 일반 마이크로프로세서에서 이런 일을 하지 마세요. 어쨌건, 하드 리얼타임 사용처를 위해 특수 설계된 시스템에서도 이를 올바르게 만들기는 충분히 어렵습니다.



Quick Quiz 17.13:

하지만 Listing 17.1의 `boostee()` 함수는 그 대신 락을 반대 순서로 잡습니다! 이는 데드락을 초래할 수 있지 않나요?



Answer:

데드락은 일어나지 않습니다. 데드락에 도달하기 위해 선, 두개의 다른 쓰레드가 두 락을 반대 순서로 획득해야 하는데, 이 예에서는 일어나지 않는 일입니다. 그러나, `lockdep` [Cor06a] 같은 데드락 탐지기는 이를 false positive로 잡아낼 겁니다.



Quick Quiz 17.14:

그러니까 여러 사람이 락킹을 대체하려고 했고, 그들은 대부분 그저 락킹을 최적화하는데 그치는 건가요???



Answer:

그들은 최소한 무언가 유용한 것을 이루어냈습니다! 그리고 시간에 따라 추가적인 HTM 쪽 진보가 계속될 겁니다 [SNGK17, SBN⁺20, GGK18, PMDY20].



Quick Quiz 17.15:

SEL4 프로젝트에서 사용된 여러 검증 도구의 놀라운 본성을 놓고 보면, 왜 이 챕터는 그걸 더 다루지 않는지 궁금하군요?



Answer:

SEL4 프로젝트에서 사용된 검증 도구들이 상당히 쓸모있을 거라는 데에는 의심의 여지가 없습니다. 그러나, SEL4는 단일 CPU 프로젝트로 시작되었습니다. 그리고 SEL4가 멀티 프로세서 기능을 얻었지만, 현재로써는 리눅스 커널의 과거의 Big Kernel Lock (BKL)과 유사한 큰 규모의 락킹을 사용하고 있습니다. SEL4의 검증 도구를 별별 프로그래밍에 대한 책에 추가하는 게 말이 되는 날이 오길 기대합니다만, 아직은 그 날이 아닙니다.



Quick Quiz 17.16:

Listing 17.2 의 라인 27 에서는 왜 단순하게 그 조건을 `exists` 절에 추가하는 대신 별도의 `filter` 커맨드를 사용하나요? 그리고 `cmpxchg_acquire()` 대신 `xchg_acquire()`를 사용하는게 더 간단하지 않을까요?

■

Answer:

이 `filter` 절은 `herd` 도구가 `exists` 절이 그리는 것보다 이론 처리 단계에서 수행을 폐기하게 해줘서 상당한 속도향상을 가져옵니다.

Table E.5: Emulating Locking: Performance Comparison (s)

#	Lock	cmpxchg_acquire()		xchg_acquire()	
		filter	exists	filter	exists
2	0.004	0.022	0.039	0.027	0.058
3	0.041	0.743	1.653	0.968	3.203
4	0.374	59.565	151.962	74.818	500.96
5	4.905				

`xchg_acquire()`의 경우, 이 어토믹 오퍼레이션은 락 획득이 성공했는지와 관계없이 쓰기를 할건데, 이는 `xchg_acquire()`를 사용하는 모델은 락 획득 실패의 경우 쓰기를 하지 않을 `cmpxchg_acquire()`를 사용하는 것보다 더 많은 오퍼레이션을 갖게 됨을 의미합니다. 더 많은 쓰기는 폭발할 수 있는 더 많은 조합을 의미하는데, Table E.5 에 보인 것과 같습니다 (C-SB+l-o-o-u+l-o-o-*u.litmus, C-SB+l-o-o-u+l-o-o-u*-C.litmus, C-SB+l-o-o-u+l-o-o-u*-CE.litmus, C-SB+l-o-o-u+l-o-o-u*-X.litmus, 그리고 C-SB+l-o-o-u+l-o-o-u*-XE.litmus). 이 표는 `cmpxchg_acquire()` 가 `xchg_acquire()` 보다, 그리고 `filter`의 사용이 `exists` 절의 사용보다 성능이 나음을 분명하게 보입니다.

□

Quick Quiz 17.17:

알려진 버그의 MTBF 들이 아직 발견되지 않은 버그의 MTBF 를 예측하기 좋은 정보임을 어떻게 아나요?

■

Answer:

우린 모릅니다, 그렇지만 그건 중요치 않습니다.

이를 보기 위해, 7% 라는 숫자는 뒤따라서 발견된 버그의 주입에만 적용됨을 기억하십시오: 이는 발견되지 못한 버그의 주입은 완전히 무시해야 합니다. 따라서, 이 알려진 버그에 대한 MTBF 통계는 뒤따라서 발견된 주입된 버그에 대한 좋은 추정이 됩니다.

이 전체 섹션의 핵심 요점은, 결코 발견되지 못한 버그보다는 사용자를 불편하게 하는 버그를 더 주의해야 한다는 겁니다. 이는 물론 우리가 아직 사용자를 불편하게 하지 않은 버그를 완전히 무시해야 한다는 말이 아니라, 우리는 가장 중요하고 시급한 버그를 고치는데 있어 노력의 우선순위를 올바르게 잡아야 한다는 것입니다.

□

Quick Quiz 17.18:

하지만 정형적 검증 도구는 그 수정에 의해 만들어진 버그를 곧바로 찾아낼텐데 왜 이게 문제인가요?

■

Answer:

실제 세계의 정형적 검증 도구는 (더 목소리 높은 제안자와 정형 검증의 상상에만 존재하는 것과는 달리) 현명치 못하기 때문에, 그리고 따라서 특정 종류의 버그를 찾아내지 못하기 때문에 문제가 됩니다. 한가지만 예를 들어보자면, 정형적 검증 도구는 누락된 단정문에 연관된, 또는 명세의 발견되지 못한 부분에 연관된 버그를 찾지 못할 겁니다.

□

Quick Quiz 17.19:

하지만 많은 정형적 검증 도구는 한번에 하나의 버그만 찾을 수 있으므로, 이 도구가 다음 버그를 찾기 전에 각 버그가 고쳐져야만 합니다. 그런 도구가 주어졌을 때 어떻게 버그 수정 노력의 우선순위를 정할 수 있겠습니까?

■

Answer:

한가지 방법은 제품 환경에서는 적합하지 않을 수도 있으나 그 도구가 다음 버그를 찾는 것은 허용할 수 있는 간단한 수정을 제공하는 것입니다. 또 다른 방법은 설정이나 입력을 제한해서 지금까지 발견된 버그가 발생하지 못하게 하는 겁니다. 여러 비슷한 방법들이 있습니다만, 공통적인 주제는 이 도구의 관점에서 버그를 고치는 것은 제품 품질 수정을 만들고 검증하는 것보다 훨씬 쉽다는 것이며, 핵심은 제품 품질 수정을 만들고 검증하는데 필요한 더 거대한 노력을 우선순위화 조정하자는 겁니다.

□

Quick Quiz 17.20:

Table 17.5 에 보인 점수판에서 테스트는 어떤 모습을 갖게 될까요?

■

Answer:

그건 세번째 열 (오버헤드) 에서의 경우 테스트의 중요 버그를 찾는데 있어서의 어려움에 따라 달라질 수 있는 예외를 제외하고는 모두 파랑일 겁니다.

다른 한편으로는, 일어날 성 싫지 않은 버그는 종종 관계 없는 버그이므로, 경우에 따라 다를 겁니다.

여러분의 설치 기반의 크기에 많이 종속적일 겁니다. 여러분의 코드가 (말하자면) 10,000 개의 시스템에서만 평생 동작할 거라면, Murphy 는 정말 친절한 사람이 될 수 있습니다. 모든 것은 잘못될 수 있고, 그럴 겁니다. 결국은요. 아마도 지질학적 시간에서는 말이죠.

하지만 여러분의 코드가 2017년 말 기준으로 리눅스 커널이 그렇다고 이야기 되듯 200억개 시스템에서 돌 아간다면, Murphy 는 진짜 나쁜놈일 수 있습니다! 모든 것은 잘못될 것이고, 그럴 것이며, 정말 빨리 그렇게 될 수 있습니다!!!

□

Quick Quiz 17.21:

하지만 Table 17.5 에 보인 것보다 훨씬 많은 정형 검증 시스템이 존재하지 않나요?

■

Answer:

실제로 그렇습니다! 이 표는 Paul 이 사용해 온 것들에 집중해 있지만, 다른 것들도 유용한 것으로 드러났습니다. 정형 검증은 SEL4 프로젝트 [SM13] 에서 널리 사용되었으며, 그 도구들은 이제 약간의 동시성을 다룰 수 있습니다. 더 최근에는, Catalin Marinas 가 Lamport 의 TLA 도구를 [Lam02] 리눅스 커널의 queued spinlock 구현의 진행 보장 버그를 찾는데 사용했습니다. Will Deacon 은 이 버그를 고쳤고 [Dea18], Catalin 은 Will 의 수정을 검증했습니다 [Mar18].

더 가벼운 정형 검증 도구들은 제품 단계에서 널리 사용되어왔습니다 [LBD⁺04, BBC⁺10, Coo18, SAE⁺18, DFLO19].

□

E.18 Important Questions

Quick Quiz A.1:

이 예들에서 어떤 SMP 코딩 오류를 당신은 찾았나요? 전체 코드를 위해선 time.c 를 보세요.

■

Answer:

- 반복문에서의 barrier() 나 volatile 사용 누락
- 업데이트 쪽에서의 메모리 배리어 누락
- 생성 쓰레드와 소비 쓰레드 사이의 동기화 부재.

□

Quick Quiz A.2:

연속된 소비 쓰레드의 읽기 사이에 어떻게 그렇게 큰 차이가 발생하죠? 전체 코드를 위해선 timelocked.c 를 보시기 바랍니다.

■

Answer:

- 이 소비 쓰레드가 오랫동안 preemption 당했을 수도 있습니다.
- 오랫동안 수행되는 인터럽트가 이 소비 쓰레드를 지연시켰을 수도 있습니다.
- 캐쉬미스가 이 소비 쓰레드를 지연시켰을 수도 있습니다.
- 생성 쓰레드는 소비 쓰레드보다 빠른 CPU 에서 수행되었을 수도 있습니다 (예를 들어, CPU 들 중 하나는 열 처리나 전력 소비 제한을 위해 처리 속도를 낮췄을 수도 있습니다).

□

Quick Quiz A.3:

프로그램의 한 부분이 RCU read-side 기능을 유일한 동기화 메커니즘으로 사용한다고 해봅시다. 이는 병렬 성입니다 동기화입니다?

■

Answer:

그렇습니다.

□

Quick Quiz A.4:

두번째 (스캐줄러 기반) 관점의 어떤 부분에서 락 기반 CPU 별 싱글쓰레드 워크로드가 “동시적”으로 여겨질 수 있을까요?

■

Answer:

임의로 워크로드를 쪼개고 짜깁으려는 사람들에 의해요. 물론, 임의의 쪼개기는 관련된 락 해제로부터 획득을 분리시키는 결과를 낳을 수도 있을텐데, 이는 다른 쓰레드가 그 락을 획득하는 걸 방지할 겁니다. 그 락이 순수한 스피드락이라면, 이는 테드락을 초래할 수도 있습니다.

□

Quick Quiz A.5:

하지만 완전히 순서잡힌 구현이 성능도 확장성도 더 좋은 완화된 순서의 구현보다 더 강한 보장을 제공하지 못한다면, 완전한 순서를 사용할 이유가 뭡니까?

**Answer:**

강력하게 순서잡힌 구현은 가끔 주어진 데이터 구조를 접근하는 함수들의 호출 집합들 사이에 더 강한 일관성을 제공할 수 있기 때문입니다. 예를 들어, Listing 5.2의 어토믹 카운터를 Section 5.2의 통계적 마운터와 비교해 보세요. 한 쓰레드가 값 3을 더하고 또 다른 쓰레드가 값 5를 더하며, 다른 두 쓰레드가 이 카운터의 값을 동시에 읽는다고 해봅시다. 어토믹 카운터에서는 읽기 쓰레드 중 하나가 값 3을 얻고 다른 쓰레드는 값 5를 얻는게 불가능 합니다. 통계적 카운터에서, 그런 결과는 정말 벌어질 수 있습니다. 사실, 어떤 컴퓨팅 환경에서는 이 결과는 x86과 같이 상대적으로 강한 순서 규칙의 하드웨어에서조차 벌어질 수 있습니다.

따라서, 여러분의 사용자가 이런 비일상적인 수준의 일관성을 필요로 하게 된다면, 여러분은 완화된 순서 규칙의 통계적 카운터를 사용하지 말아야 합니다.



E.19 “Toy” RCU Implementations

Quick Quiz B.1:

왜 Listing B.1에서의 RCU 구현 내의 모든 데드락은 다른 RCU 구현에서도 데드락이지 않을까요?

**Answer:**

Listing E.17에서의 함수 `foo()` 와 `bar()` 가 다른 CPU에서 동시에 호출되었다고 해봅시다. 그럼 `foo()` 는 라인 3에서 `my_lock()` 을 획득하고, 라인 13에서 `bar()` 는 `rcu_gp_lock` 을 획득합니다.

`foo()` 가 라인 4로 나아가면, `bar()` 가 잡고 있는 `rcu_gp_lock` 을 획득하려 시도할 겁니다. 그러면 `bar()` 가 라인 14로 나아갈 때, `foo()` 가 잡고 있는 `my_lock` 을 획득하려 할 겁니다.

그럼 각 함수는 서로가 잡고 있는 락을 기다리므로, 고전적인 데드락이 됩니다.

다른 RCU 구현은 `rcu_read_lock()` 에서 `spin` 하지도 `block` 하지도 않으므로 데드락이 회피됩니다.

**Listing E.17: Deadlock in Lock-Based RCU Implementation**

```

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_something();
16     spin_unlock(&my_lock);
17     do_whatever();
18     rcu_read_unlock();
19 }

```

Quick Quiz B.2:

Listing B.1의 RCU 구현에서는 왜 RCU 읽기 쓰레드들이 병렬로 수행될 수 있게끔 단순히 reader-writer 락을 사용하지 않죠?

**Answer:**

실제로 그런 방식으로 reader-writer 락을 사용할 수 있을 겁니다. 그러나, 교재 상의 reader-writer 락은 메모리 경쟁으로 고통받으므로, 병렬 수행을 실제로 허용하기 위해선 RCU read-side 크리티컬 섹션이 상당히 길어져야 할 겁니다 [McK03].

다른 한편, `rcu_read_lock()` 에서 읽기 모드로 획득된 reader-writer 락의 사용은 앞서 언급된 데드락 조건을 막을 겁니다.

**Quick Quiz B.3:**

Listing B.2의 라인 15–18에서의 반복문에서는 모든 락을 획득한 후 한번에 해제하는게 더 깔끔하지 않을까요? 어쨌건, 이 변경으로 인해 어떤 읽기 쓰레드도 존재하지 않는 시점이 존재하게 되어서 모든 것을 훨씬 간단하게 만들 겁니다.

**Answer:**

이 변경을 가하는 것은 데드락을 다시 불러일으키며, 따라서 아니오, 그건 더 깔끔하지 않습니다.

**Quick Quiz B.4:**

Listing B.2에 보인 구현은 데드락으로부터 자유로운가요? 이유는 뭐죠?

**Answer:**

한가지 데드락 상황은 어떤 락이 `synchronize_rcu()`

에 걸쳐 잡혀져 있으며, 같은 락이 어떤 RCU read-side 크리티컬 섹션에 의해 획득될 때일 겁니다. 그러나, 이 상황은 모든 올바르게 설계된 RCU 구현을 데드락에 빠지게 할 겁니다. 어쨌건, `synchronize_rcu()` 기능은 모든 앞서 존재했던 RCU read-side 크리티컬 섹션이 완료되길 기다려야만 하지만, 이 크리티컬 섹션들 가운데 하나가 `synchronize_rcu()` 가 수행중인 쓰레드에 의해 잡힌 락을 기다린다면, RCU 의 정의에 내재한 데드락을 갖게 됩니다.

또 다른 데드락 상황은 RCU read-side 크리티컬 섹션을 중첩시키려 할 때 발생합니다. 이 데드락은 이 구현에만 있으며, 재귀 락을 사용하거나 `rcu_read_lock()`에 의해 읽기 모드로 획득되고 `synchronize_rcu()`에 의해 쓰기 모드로 획득되는 reader-writer 락의 사용으로 회피될 수도 있을 겁니다.

그러나, 앞의 두 경우를 배제한다면, 이 RCU 구현은 어떤 데드락 상황도 초래하지 않습니다. 이는 어떤 다른 쓰레드의 락이 획득된 시점은 `synchronize_rcu()` 가 수행중일 때 뿐이며, 락은 곧바로 해제될 것이어서 앞의 첫번째 경우인, `synchronize_rcu()`에 걸쳐 잡힌 락이 관여되는 데드락 사이클을 방지하기 때문입니다.

□

Quick Quiz B.5:

Listing B.2 에 보인 RCU 알고리즘의 한가지 장점은 예를 들면 POSIX pthreads 같은 곳에서 널리 사용 가능한 기능만을 사용한다는 점일까요?

■

Answer:

이는 실제로 장점입니다만, `rcu_dereference()` 와 `rcu_assign_pointer()` 는 여전히 필요한데, `rcu_dereference()` 에서의 `volatile` 조정과 `rcu_assign_pointer()` 에서의 메모리 배리어 사용을 의미합니다. 물론, 두 기능 모두 Alpha CPU 에서는 메모리 배리어가 필요할 겁니다.

□

Quick Quiz B.6:

하지만 여러분이 `synchronize_rcu()` 호출 전반에 걸쳐 락을 잡고 있고, 어떤 RCU read-side 크리티컬 섹션 내에서 같은 락을 잡으면 어떻게 될까요?

■

Answer:

실제로 이는 모든 합법적 RCU 구현에 데드락을 초래 할 수 있습니다. 하지만 `rcu_read_lock()` 은 정말로 이 데드락 사이클에 참여하고 있나요? 그렇다고 믿는다면, Appendix B.9 의 RCU 구현을 볼 때 같은 질문을 스스로에게 해보세요.

□

Quick Quiz B.7:

`synchronize_rcu()` 가 10-밀리세컨드 지연을 갖는데 어떻게 grace period 는 40 나노세컨드에 끝날 수 있죠?

■

Answer:

이 업데이트 쪽 테스트는 읽기 쓰레드의 부재 하에 수행되었으므로, `poll()` 시스템콜이 아예 수행되지 않았습니다. 또한, 실제 코드는 이 `poll()` 시스템콜이 주석처리되어 있어서, 이 업데이트 쪽 코드의 실제 오버헤드를 더 잘 평가할 수 있었습니다. 이 코드의 제품 환경에서의 사용은 `poll()` 시스템콜의 사용을 통해 나아질 수 있겠습니다만, 다시 말하지만 제품 환경에서의 사용의 경우 이 섹션의 뒤쪽에서 소개할 다른 구현을 사용하는 게 나을 겁니다.

□

Quick Quiz B.8:

Listing B.3 의 RCU 구현에서 `synchronize_rcu()` 가 너무 오래 기다리고 있었다면 `rcu_read_lock()` 이 기다리게 하는건 어떤가요? 이게 `synchronize_rcu()` 가 기아에 빠지는 걸 방지하지 않을까요?

■

Answer:

이게 실제로 기아를 제거하겠지만, 이는 또한 `rcu_read_lock()` 이 쓰기 쓰레드를 기다리느라 spin 또는 block 을 함을 의미하는데, 이는 결국 읽기 쓰레드를 기다리는 일입니다. 이 읽기 쓰레드들 중 하나가 이 spin/block 하는 `rcu_read_lock()` 이 잡고 있는 락을 잡으려 한다면 우린 또다시 데드락에 빠집니다.

짧게 말해서, 이 치료법은 질병보다 더 나쁩니다. 올바른 치료를 위해선 Appendix B.4 를 보십시오.

□

Quick Quiz B.9:

Listing B.6 의 `synchronize_rcu()` 에서의 라인 5 에서의 메모리 배리어는 바로 뒤에 스피너 획득이 있는데 왜 필요하죠?

■

Answer:

이 스피너 획득은 이 스피너의 크리티컬 섹션이 이 획득 앞으로 “새어나가지” 않게 보장할 뿐입니다. 스피너 획득을 앞서는 코드가 이 크리티컬 섹션 안으로 재배치 되는 것은 막지 않습니다. 그런 재배치는 RCU 로 보호되는 리스트로부터의 우너소 삭제가 `rcu_idx` 의 보상을 뒤따르게끔 재배치 하는 것을 허용해서 새로 시작된 RCU read-side 크리티컬 섹션이 최근 제거된 데이터 원소를 볼 수 있게 할 수 있습니다.

독자 여러분을 위한 연습: Listing B.6 의 메모리 배리어 가운데 무엇이 정말로 필요한지 판단하기 위해

Promela/spin 같은 도구를 사용해 보세요. 이 도구들을 사용하는 법을 알기 위해 Chapter 12 을 보세요. 첫번째 올바르고 완전한 답은 상을 받을 겁니다.



Quick Quiz B.10:

왜 이 카운터는 Listing B.6에서 두번 뒤집히나요? 한번 뒤집고 기다리기만으로도 충분하지 않나요?



Answer:

두번의 뒤집기가 분명 필요합니다. 이를 보기 위해, 다음 이벤트의 연속을 생각해 봅시다:

1. Listing B.5의 `rcu_read_lock()`의 라인 8에서 `rcu_idx`를 가져오고 그 값이 0임을 확인합니다.
2. Listing B.6의 `synchronize_rcu()`의 라인 8에서 `rcu_idx`의 값을 보상해서 그 값을 1로 만듭니다.
3. `synchronize_rcu()`의 라인 10-12에서 `rcu_refcnt[0]`의 값이 0임을 확인하고 따라서 리턴합니다. (질문은 라인 13-20이 제거되면 어떻게 되는가임을 상기합시다.)
4. `rcu_read_lock()`의 라인 9와 10 가이 쓰레드의 `rcu_read_idx` 인스턴스에 값 0을 저장하고 `rcu_refcnt[0]`의 값을 증가시킵니다. 수행은 이제 이 RCU read-side 크리티컬 섹션 내부로 이어집니다.
5. `synchronize_rcu()`의 또 다른 인스턴스가 다시 `rcu_idx`를 보상하는데, 이번에는 그 값을 0으로 만듭니다. `rcu_refcnt[1]`은 0 이므로, `synchronize_rcu()`는 곧바로 리턴합니다. (`rcu_read_lock()`은 `rcu_refcnt[1]`이 아닌 `rcu_refcnt[0]`을 값 증가시켰음을 기억하세요!)
6. Step 5에서 시작된 grace period는 이제 종료가 허가되었는데, 이전에 step 4에서 시작된 RCU read-side 크리티컬 섹션은 완료되지 않았으므로 불구하고 그렇습니다. 이는 RCU semantic의 위반이며, 업데이트가 RCU read-side 크리티컬 섹션에 여전히 참조하고 있는 데이터 원소를 메모리 해제할 수 있게 합니다.

독자 여러분을 위한 연습: `rcu_read_lock()`이 라인 8 후에 매우 긴 시간(수시간!) preemption 당하면 무슨 일이 벌어질까요? 이 구현은 그런 경우에 잘 동작할까요? 왜 그럴까요? 첫번째 올바르고 완전한 답변은 상을 받을 겁니다.



Quick Quiz B.11:

어토믹 값 증가와 감소가 그렇게 비싸면, Listing B.5의 라인 10에서 어토믹하지 않은 값 증가를 사용하고 라인 25에서 어토믹하지 않은 값 감소를 사용하는 건 어떻습니까?



Answer:

어토믹하지 않은 오퍼레이션을 사용하는 것은 값 증가와 감소를 손실되게 만들어서 결국 이 구현이 실패하게 합니다. `rcu_read_lock()`과 `rcu_read_unlock()`에서 비 어토믹 오퍼레이션을 사용하는 안전한 방법을 위해선 Appendix B.5를 참고하십시오.



Quick Quiz B.12:

나와요! `rcu_read_lock()`의 `atomic_read()`가 보인다구요!!! 왜 `rcu_read_lock()`이 어토믹 오퍼레이션을 가지고 있지 않은 척 하는 거죠???



Answer:

이 `atomic_read()` 기능은 실제로 어토믹 기계명령을 수행하지 않고 그저 평범한 `atomic_t`로부터의 로드를 합니다. 이것의 유일한 목적인 컴파일러의 타입 검사를 만족시키는 겁니다. 만약 리눅스 커널이 8-bit CPU에서 수행된다면 이는 또한 16-bit 포인터를 두개의 8-bit 액세스를 통해 저장해야 하기 때문에 일부 8-bit 시스템에서 발생하는 “store tearing”을 방지하기 위해 필요합니다. 그러나 감사하게도, 아무도 리눅스를 8-bit 시스템에서 수행하지 않는 듯 합니다.



Quick Quiz B.13:

훌륭하군요, 우리가 N 쓰레드를 갖는다면 우린 $2N$ 수십 밀리세컨드(`flip_counter_and_wait()` 호출당 한 세트씩, 쓰레드별로 한번씩만 기다린다 가정하더라도) 대기할 수 있군요. 우린 grace period가 훨씬 더 빨리 끝나길 필요로 하지 않나요?



Answer:

우린 쓰레드가 여전히 앞서서부터 존재한 RCU read-side 크리티컬 섹션 안에 있을 때에만 그 쓰레드를 기다리며, 하나의 베티는 쓰레드를 기다리는 것은 모든 다른 쓰레드에게 그들이 여전히 수행중일 수도 있는 기존부터 존재한 RCU read-side 크리티컬 섹션을 완료시킬 기회를 줍니다. 따라서 우리가 $2N$ 인터벌을 기다리게 되는 유일한 방법은 이 마지막 쓰레드가 앞의 모든 쓰레드에 대한 모든 기다림에도 불구하고 여전히 기존부터 존재한 RCU read-side 크리티컬 섹션에 남아있는 겁니다. 짧게 말하자면, 이 구현은 불필요하게 기다리지 않을 겁니다.

그러나, 여러분이 RCU 를 사용하는 코드를 스트레스 테스트 한다면, 여러분은 RCU read-side 크리티컬 섹션 밖에서 RCU 로 보호되는 데이터 원소를 잘못되게 참조하는 버그를 더 잘 발견하기 위해 이 `poll()` 문을 주석처리하고 싶을 수도 있습니다.

□

Quick Quiz B.14:

이 모든 toy RCU 구현은 `rcu_read_lock()` 과 `rcu_read_unlock()`, 또는 `synchronize_rcu()` 에서 어토믹 오퍼레이션을 가지고 있어서 쓰레드의 수에 따라 오버헤드가 선형적으로 증가합니다. 어떤 환경에서야 어떤 RCU 구현은 이 세개의 모든 기능이 결정적인 ($O(1)$) 오버헤드와 응답시간을 제공하는 가벼운 구현을 즐길 수 있을까요?

■

Answer:

RCU 의 특수 목적 유니프로세서 구현은 이 이상을 달성할 수 있습니다 [McK09a].

□

Quick Quiz B.15:

어떤 짹수 값이든 `synchronize_rcu()` 에게 그 태스크를 무시하라고 말하기 충분하다면, 왜 Listing B.14 의 라인 11 와 12 는 간단히 `rcu_reader_gp` 에 0 을 할당하지 않나요?

■

Answer:

0 (또는 어떤 짹수 값이든 짹수 상수) 를 할당하는 것도 실제로 동작할 것입니다만, `rcu_gp_ctrl` 에 값은 할당하는 것은 의미있는 디버깅에 도움이 될 것인데, 이는 개발자에게 언제 연관된 쓰레드가 마지막으로 RCU read-side 크리티컬 섹션을 빠져나왔는지 눈치챌 수 있게 해주기 때문입니다.

□

Quick Quiz B.16:

Listing B.14 의 라인 19 와 31 의 메모리 배리어는 왜 필요하죠? 라인 20 와 30 의 락킹 기능에 내재된 메모리 배리어로 충분하지 않나요?

■

Answer:

락킹 기능은 크리티컬 섹션을 국한시키는 것만을 보장하기 때문에 이 메모리 배리어들이 필요합니다. 락킹 기능들은 다른 코드가 크리티컬 섹션 내로 새어들어오는 것을 막을 책임은 전혀 없습니다. 따라서 이런 유의 코드 움직임이 컴파일러에 의해서든 CPU 에 의해서든 일어나는 걸 방지하기 위해 이 한쌍의 메모리 배리어가 필요합니다.

□

Quick Quiz B.17:

Appendix B.6 에 설명된 업데이트 쪽 batching 최적화는 Listing B.14 에 보인 구현에 적용될 수 없나요?

■

Answer:

약간의 수정과 함께 실제로 그럴 수 있습니다. 이 일은 독자 여러분의 연습을 위해 남겨둡니다.

□

Quick Quiz B.18:

Listing B.14 의 라인 3-4 에서 읽기 쓰레드가 preemption 될 수 있다는 가능성은 잔짜로 문제인가요? 달리 말하자면, 문제를 일으킬 수 있는 실제 이벤트 순서가 존재합니까? 아니라면 왜 그렇습니까? 그렇다면 그 이벤트 순서는 무엇이며 어떻게 그 문제가 처리될 수 있습니까?

■

Answer:

이는 진짜 문제이고, 문제를 일으키는 이벤트의 순서가 존재하며, 이를 해결하기 위한 여러 가능한 방법들이 있습니다. 더 자세한 부분을 위해선, Appendix B.8 끝부분의 Quick Quizz 를 보시기 바랍니다. 그 논의를 거기에는 이유는 (1) 여러분에게 생각할 시간을 더 드리기 위해, 그리고 (2) 그 섹션에서 더해지는 중첩에 대한 지원이 카운터를 오버플로우 시키는데 걸리는 시간을 크게 줄이기 때문입니다.

□

Quick Quiz B.19:

왜 이 복잡한 비트 조정 대신 앞의 섹션에서처럼 가난하게 별개의 쓰레드별 중첩 수준 변수를 유지하지 않나요?

■

Answer:

별개 쓰레드별 변수의 분명한 단순성은 사기입니다. 이 방법은 오퍼레이션의 주의 깊은 순서 잡기라는 형태로 아주 큰 복잡도를 일으키는데, 특히 시그널 핸들러가 RCU read-side 크리티컬 섹션을 포함할 수 있을 때 그렇습니다. 하지만 제말만 듣지 말고, 코드를 직접 짜보고 어떻게 되나 보세요!

□

Quick Quiz B.20:

Listing B.16 에 보인 알고리즘을 가지고 어떻게 전역 `rcu_gp_ctrl` 의 오버플로우에 걸리는 시간을 두배로 늘릴 수 있죠?

■

Answer:

한가지 방법은 라인 32 와 33 에서의 정도 비교를 쓰레드별 `rcu_reader_gp` 변수의 `rcu_gp_ctrl+RCU_GP_CTRL_BOTTOM_BIT`에 대한 동일성 검사로 대체하는 것입니다.



Quick Quiz B.21:

다시, Listing B.16 의 알고리즘에서 카운터 오버플로우는 치명적인가요? 왜 그렇죠? 그게 치명적이라면 이를 고치기 위해 뭘 할 수 있을까요?



Answer:

이는 실제로 치명적일 수 있습니다. 이를 보기 위해, 다음 이벤트 순서를 생각해 봅시다:

- 쓰레드 0 이 `rcu_read_lock()` 을 진입하고, 중첩 되지 않았음을 파악한 후, 따라서 전역 `rcu_gp_ctrl` 의 값을 가져옵니다. 쓰레드 0 은 이어서 굉장히 긴 시간 동안 `preemption` 됩니다 (자신의 쓰레드 별 `rcu_reader_gp` 변수에 값을 저장하기 전).
- 다른 쓰레드들이 반복적으로 `synchronize_rcu()` 를 호출하고, 따라서 전역 `rcu_gp_ctrl` 의 새 값은 쓰레드 0 이 읽었을 때보다 `RCU_GP_CTR_BOTTOM_BIT` 만큼 작은 값이 됩니다.
- 쓰레드 0 이 수행을 재개하고, 자신의 쓰레드 별 `rcu_reader_gp` 변수에 값을 저장합니다. 이 때 저장되는 값은 전역 `rcu_gp_ctrl` 보다 `RCU_GP_CTR_BOTTOM_BIT+1` 큰 값입니다.
- 쓰레드 0 이 RCU 로 보호되는 데이터 원소 A 로의 참조를 획득합니다.
- 이제 쓰레드 1 이 쓰레드 0 이 막 참조를 획득한 데이터 원소 A 를 제거합니다.
- 쓰레드 1 이 `synchronize_rcu()` 를 호출하는데, 이는 전역 `rcu_gp_ctrl` 을 `RCU_GP_CTR_BOTTOM_BIT` 만큼 증가시킵니다. 이어서 이 쓰레드는 모든 쓰레드 별 `rcu_reader_gp` 변수를 검사하지만, 쓰레드 0 의 값은 (잘못되게도) 자신이 쓰레드 1 의 `synchronize_rcu()` 호출 되에 시작했다고 이야기 하므로, 쓰레드 1 은 쓰레드 1 이 쓰레드 1 의 RCU read-side 크리티컬 섹션을 완료할 때까지 기다리지 않습니다.
- 이제 쓰레드 1 은 쓰레드 0 이 여전히 참조하고 있는 데이터 원소 A 를 메모리 해제합니다.

이 시나리오는 Appendix B.7 에 보인 구현에서도 발생 가능함을 알아 두십시오.

이 문제를 해결하는 한가지 방법은 이를 오버플로우시키는데 걸리는 시간이 컴퓨터 시스템의 유용한 수명을 넘기게끔 64-비트 카운터를 쓰는 겁니다. 골동품은 아닌 32-비트 x86 CPU 제품군의 멤버들은 `cmpxchg64b`

명령을 통해 64-비트 카운터를 어토믹하게 조정할 수 있음을 알아 두시기 바랍니다.

또 다른 방법은 비슷한 효과를 위해 grace period 가 허용되는 일어날 수 있는 비율에 한계를 두는 겁니다. 예를 들어, `synchronize_rcu()` 가 자신이 호출된 마지막 시간을 기록할 수 있다면, 이를 뒤따르는 호출은 이 시간을 검사하고 필요한 기간을 강제하기 위해 블록될 수 있습니다. 예를 들어, 이 카운터의 아래쪽 네개 비트가 중첩을 위해 예약되었다면, 그리고 grace period 가 초당 최대 10번 발생하는게 허용되어 있다면, 이 카운터가 오버플로우 되기 위해선 300일이 넘게 걸릴 겁니다. 그러나, 이 방법은 시스템이 CPU 위주의 높은 우선순위 리얼타임 쓰레드로 인한 부하를 300일 동안 걸려 있을 수 있다면 도움이 되지 않습니다. (희박한 확률이겠지만 미리 생각해 두는게 최선입니다.)

세번째 방법은 시스템에서 리얼타임 쓰레드를 없애 버리는 겁니다. 이 경우, `preemption` 당한 프로세스는 시간에 따라 우선순위를 높이고, 따라서 카운터가 오버플로우 되기 한참 전에 수행을 재개합니다. 물론, 이 방법은 리얼타임 어플리케이션에는 만족스럽지 못할 겁니다.

마지막 방법은 `rcu_read_lock()` 이 전역 `rcu_gp_ctrl` 의 값을 자신의 쓰레드 별 `rcu_reader_gp` 카운터에 저장한 후 다시 검사하고, 전역 `rcu_gp_ctrl` 의 새 값이 잘못되어 있다면 재시도 하는 겁니다. 이는 동작 하지만, 비결정적인 수행 시간을 `rcu_read_lock()` 에 가져옵니다. 다른 한편, 여러분의 어플리케이션이 카운터가 오버플로우 되기 충분할 정도로 오래 `preemption` 당한다면, 어쨌든 결정적 수행 시간을 희망할 순 없을 겁니다!



Quick Quiz B.22:

Listing B.18 의 라인 14 에 보인 추가적인 메모리 배리어는 `rcu_quiescent_state()` 의 오버헤드를 크게 늘리지 않나요?



Answer:

실제로 그렇습니다! 따라서 이 구현의 RCU 를 사용하는 어플리케이션은 `rcu_quiescent_state()` 를 절약하고 대부분의 경우 `rcu_read_lock()` 과 `rcu_read_unlock()` 을 사용해야 합니다.

그러나, 이 메모리 배리어는 다른 쓰레드가 그 호출자에 의해 뒤따르는 RCU read-side 크리티컬 섹션이 수행되기 전에 라인 12-13 에 저장된 값을 보게 하기 위해 절대적으로 필요합니다.



Quick Quiz B.23:

Listing B.18 의 라인 11 와 14 의 메모리 배리어 한쌍은 왜 필요한가요?

**Answer:**

라인 11 에서의 메모리 배리어는 `rcu_thread_offline()` 앞에 있을 수 있는 RCU read-side 크리티컬 섹션이 컴파일러나 CPU 에 의해 라인 12-13 에서의 값 할당의 뒤로 재배치 되는 걸 방지합니다. 라인 14 에서의 메모리 배리어는 엄밀히 말하면 불필요한데, `rcu_thread_offline()` 호출 뒤에 RCU read-side 크리티컬 섹션을 두는 건 불법이기 때문입니다.

**Quick Quiz B.24:**

분명히 해두자면, 2008년의 POWER 시스템의 클락 주파수는 상당히 높았지만, 5 GHz 클락 주파수조차도 반복문이 50 피코세컨드 내에 수행되게 하는데에는 불충분해요! 뭐가 어떻게 된거죠?

**Answer:**

이 측정 반복문은 한쌍의 텅 빈 함수를 가지고 있으므로, 컴파일러는 이를 최적화해 제거해 버립니다. 이 측정 반복문은 `rcu_quiescent_state()` 호출 사이에 1,000 번 돌고, 따라서 이 측정은 대략 `rcu_quiescent_state()` 호출 한번의 오버헤드를 1,000 으로 나눈 값입니다.

**Quick Quiz B.25:**

왜 그 코드가 라이브러리에 있다는 사실이 Listings B.18 and B.19 에 보인 RCU 구현의 사용 편의성에 차이를 끼칠 수 있죠?

**Answer:**

라이브러리 함수는 호출자에 대한 제어가 전혀 없으며, 따라서 호출자가 주기적으로 `rcu_quiescent_state()` 를 호출할 것을 강제할 수 없습니다. 다른 한편, 특정 RCU 로 보호되는 데이터 구조체로의 참조를 여럿 만든 라이브러리 함수는 진입 시에 `rcu_thread_online()` 을, 주기적으로 `rcu_quiescent_state()` 를, 그리고 종료 시에 `rcu_thread_offline()` 을 호출할 수도 있을 겁니다.

**Quick Quiz B.26:**

하지만 `synchronize_rcu()` 호출 동안 락을 잡고 있고, 똑같은 락을 어떤 RCU read-side 크리티컬 섹션 내에서 잡는다면 어떻게 될까요? 이는 데드락일 겁니다만, 어떤 코드도 만들어내지 않는 기능이 어떻게 데드락 사이클에 참여할 수 있을까요?

**Answer:**

이 RCU read-side 크리티컬 섹션은 이를 감싸는 `rcu_read_lock()` 과 `rcu_read_unlock()` 을 넘어서서 앞과 나중의 `rcu_quiescent_state()` 로 연장되는 효과를 가짐을 명심하십시오. 이 `rcu_quiescent_state()` 는 `rcu_read_lock()` 뒤에 곧바로 따라오는 `rcu_read_unlock()` 으로 생각될 수 있습니다.

그렇다고 해도, 실제 데드락 그 자체는 RCU read-side 크리티컬 섹션과 `synchronize_rcu()` 내에서의 락 획득에 연관되지, `rcu_quiescent_state()` 에 연관되는 않을 겁니다.

**Quick Quiz B.27:**

RCU read-side 크리티컬 섹션 내에서 grace period 가 금지된다면, 어떻게 RCU 로 보호되는 데이터 구조가 RCU read-side 크리티컬 섹션 내에서 업데이트 될 수 있죠?

**Answer:**

이 상황이 `call_rcu()` 같은 비동기적 grace-period 기능이 필요한 이유입니다. 이 기능은 RCU read-side 크리티컬 섹션 내에서 호출되고, 여기 명시된 RCU 콜백이 하나의 grace period 가 지난 후에 호출될 겁니다.

RCU read-side 크리티컬 섹션 내에서 RCU 업데이트를 수행할 수 있는 능력은 매우 편리할 수 있으며, reader-writer 락킹의 (실존하지 않는) 무조건적 read-to-write 업그레이드와 비슷합니다.



E.20 Why Memory Barriers?

Quick Quiz C.1:

Writeback 메세지는 어디서 생겨나서 어디로 향하게 되나요?

**Answer:**

Writeback 메세지는 특정 CPU 에서, 어떤 설계에서는 특정 CPU 의 캐쉬의 특정 레벨에서—또는 심지어 여러 CPU 들에 공유되어 있는 캐쉬에서도—생성됩니다. 핵심은 특정 캐쉬가 특정 데이터 항목을 위한 공간이

없으며, 따라서 공간을 만들기 위해 이 캐쉬에서 어떤 데이터 조각들이 제거되어야 한다는 겁니다. 메모리나 다른 캐쉬에 복사되어 있는 데이터 조각이 존재한다면 그 조각은 writeback 메세지의 필요 없이 단순히 폐기될 수 있을 겁니다.

다른 한편, 만약 제거될 수 있는 모든 데이터 조각이 modified 상태여서 유일한 최신 상태의 데이터 복사본은 이 캐쉬에만 있다면, 이 데이터 항목들 가운데 하나는 어딘가 다른곳에 복사되어야만 합니다. 이 복사 오퍼레이션은 “writeback message”를 사용해 취해집니다.

Writeback 메세지의 목적지는 이 새 값을 저장할 어딘가가 됩니다. 이는 메인 메모리일 수도 있으나, 다른 캐쉬일 수도 있습니다. 그게 캐쉬라면, 이는 보통 같은 CPU를 위한 더 높은 레벨의 캐쉬인데, 예를 들어 level-1 캐쉬는 level-2 캐쉬로 도로 쓰기를 할수도 있습니다. 그러나, 어떤 하드웨어 설계는 CPU 간 writeback 을 허용해서 CPU 0 의 캐쉬는 writeback 메세지를 CPU 1에게 보낼 수도 있습니다. 이는 일반적으로 예를 들면 최근에 read request 를 보냈다던지 하는 식으로 CPU 1이 어떻게든 이 데이터에의 흥미를 알게 된 경우 행해질 겁니다.

요약하자면, writeback 메세지는 공간이 부족한 시스템의 어느 부분에서 보내어지며, 이 데이터를 수용할 수 있는 시스템의 어떤 다른 부분에서 받아지게 됩니다.



Quick Quiz C.2:

두개의 CPU 가 동시에 같은 캐쉬 라인을 무효화 하려 하면 어떻게 되나요?



Answer:

해당 CPU 들 가운데 하나가 공유 버스에의 액세스를 먼저 얻고, 그 CPU 가 “승리” 합니다. 다른 CPU 는 자신의 해당 캐쉬 라인 복사본을 무효화하고 그 다른 CPU 에게 “invalidate acknowledge” 메세지를 보내야만 합니다.

물론, 진 CPU 는 곧바로 “read invalidate” 트랜잭션을 요청할 것으로 예상될 수 있으며, 따라서 승리한 CPU 의 승리는 짧은 것일 겁니다.



Quick Quiz C.3:

거대 멀티프로세서에서 “invalidate” 메세지가 나타날 때, 모든 CPU 는 “invalidate acknowledge” 응답을 보내야만 합니다. 이로 인한 “invalidate acknowledge” 응답의 “폭풍” 은 시스템 버스를 완전히 포화시키지 않을까요?



Answer:

이 거대 규모 멀티프로세서가 실제로 그런 방식으로 구현되어 있다면 그럴 수도 있습니다. 특히 NUMA 머신 같은 거대한 멀티프로세서들은 이것을 포함한 문제들을

막기 위해 “directory-based” 캐쉬 일관성 프로토콜이라 불리는 것을 사용합니다.



Quick Quiz C.4:

SMP 머신이 정말로 메세지 전달을 어쨌든 사용한다면, 왜 SMP 를 신경쓰죠?



Answer:

과거의 수십년간 이 주제에 대한 상당한 논란이 있었습니다. 한가지 답은 캐쉬 일관성 프로토콜은 상당히 간단하며, 따라서 하드웨어에 직접 구현될 수 있어서 소프트웨어 메세지 전달로는 얻어질 수 없는 대역폭과 응답시간 향상을 얻는다는 겁니다. 또 다른 답은 거대 SMP 머신과 작은 SMP 머신의 클러스터의 상대적 가격 때문에 경제로부터 진실을 찾을 수 있다는 겁니다. 세번째 답은 SMP 프로그래밍 모델은 분산 시스템의 그것보다 더 사용하기 쉽다는 것입니다만, 반박론자들은 HPC 클러스터와 MPI 의 등장을 이야기할 수도 있겠습 니다. 그러므로 토론은 계속됩니다.



Quick Quiz C.5:

하드웨어는 앞서 설명된 지연된 전환을 어떻게 처리하나요?



Answer:

일반적으로 상태를 더함으로써 처리합니다만, 이 추가적인 상태는 이 캐쉬 라인에 정말로 저장될 필요는 없는데, 한번에 몇개의 라인만이 전환되고 있을 것이라는 사실 덕입니다. 하지만 전환을 지연해야 하는 필요는 실제 세계의 캐쉬 일관성 프로토콜이 이 부록에 묘사된 과하게 단순화된 MESI 프로토콜보다 훨씬 더 복잡해지게 만드는 하나의 문제입니다. Hennessy 와 Patterson 의 고전적인 컴퓨터 구조 소개 [HP95] 는 이 문제 여력을 다룹니다.



Quick Quiz C.6:

어떤 순서의 오퍼레이션들이 이 CPU 의 캐쉬들을 “invalid” 상태로 되돌릴까요?



Answer:

CPU 의 명령 집합에 특수한 “내 캐쉬를 비워줘” 명령이 없는 그런 순서의 오퍼레이션 집합은 존재하지 않습니다. 대부분의 CPU 는 그런 명령을 갖습니다.



Quick Quiz C.7:

하지만 스토어 버퍼의 주요 목적이 멀티프로세서 캐쉬 일관성 프로토콜에서의 응답 지연을 감추기 위함이라면, 단일프로세서는 왜 스토어 버퍼를 갖죠?

**Answer:**

스토어 버퍼의 목적은 멀티프로세서 캐쉬 일관성 프로토콜에서의 응답 지연을 감추기 위한 것만이 아니라 일반적인 메모리 응답시간을 감추는 것이기 때문입니다. 메모리는 유니프로세서에서의 캐쉬보다 훨씬 느리므로, 유니프로세서에서의 스토어 버퍼는 write-miss 응답시간을 감추는데 도움이 됩니다.

**Quick Quiz C.8:**

앞의 step 1에서, CPU 0은 왜 단순한 “invalidate”가 아닌 “read invalidate”를 보내야 하죠?

**Answer:**

해당 캐쉬 라인은 변수 a 외에도 더 많은 것을 담고 있기 때문입니다.

**Quick Quiz C.9:**

Page 409의 Appendix C.3.3의 step 15 뒤에, 두 CPU가 “b”의 새 값을 담은 캐쉬라인을 버릴 수도 있을 겁니다. 이게 이 새로운 값을 잊어버리게 할 수 있을까요?

**Answer:**

그럴 수도 있고, 그것이 실제 하드웨어는 이 문제를 막기 위한 단계를 취하는 이유입니다. Vasilevsky Alexander에 의해 지적된 전통적은 접근법은 이 캐쉬 라인을 “shared”로 표시하기 전에 메인 메모리에 도로 쓰는 겁니다. 보다 효율적인 (더 복잡하지만) 방법은 이 캐쉬 라인이 “dirty” 한지를 가리키는 상태를 추가하여 write-back이 일어나게 해주는 겁니다. 2000년대의 시스템은 더 나아가서 반복적인 writeback을 막기 위해 훨씬 많은 상태를 사용합니다 [CS99, Figure 8.42]. 그로부터의 시간 사이에 이 복잡성이 줄어들지는 않았다고 가정하는게 합리적일 겁니다.

**Quick Quiz C.10:**

Appendix C.4.3의 첫번째 시나리오에서의 step 1에서, 왜 “read invalidate” 대신 “invalidate” 메세지가 보내지죠? CPU 0은 “a”와 캐쉬 라인을 공유하는 다른 변수의 값이 필요하지 않을까요?

**Answer:**

CPU 0은 “a”를 담는 캐쉬라인의 읽기 전용 복사본을

가지고 있으므로 이미 이 변수들의 값을 가지고 있습니다. 따라서, CPU 0이 해야 하는 것은 다른 CPU들이 이 캐쉬라인의 복사본을 제거하게 하는 겁니다. 따라서 “invalidate” 메세지로 충분합니다.

**Quick Quiz C.11:**

뭐라고요??? 이 CPU는 while 반복문이 완료되기 전까지는 assert()를 수행할 수 없는데 메모리 배리어가 왜 필요하죠?

**Answer:**

이 메모리 배리어가 누락되었다고 해봅시다.

CPU들은 뒤따르는 로드를 예측할 자유가 있는데, 이는 while 반복문이 완료되기 전에 이 단정문을 수행하는 효과를 낼 수 있음을 명심하십시오. 더 나아가서, 컴파일러는 현재 수행되는 쓰레드만이 이 변수를 업데이트한다고 가정하며, 이 가정은 컴파일러가 a의 로드를 이 반복문 앞에 이루어지게끔 만들 수 있습니다.

실제로, 일부 컴파일러는 이 반복문을 다음과 같이 무한 반복문을 감싸는 분기문으로 변환할 겁니다:

```

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    assert(a == 1);
14 }
```

이 최적화 때문에, 이 코드는 원래의 코드와는 완전히 다른 방식으로 동작할 겁니다. bar()가 “b == 0”를 판별하면, 이 무한 반복문 때문에 단정문은 닿지도 못할 겁니다. 그러나, bar()가 “foo()”가 저장한 값 “1”을 로드했다면, 이 CPU는 자신의 캐쉬에 “a”的 오래된 값 0을 가지고 있어서 단정문이 격발되게 할 겁니다. 여러분은 물론이 컴파일러가 여러분의 병렬 코드를 망가뜨리는 것을 방지하기 위해 volatile 캐스팅을 (예를 들면, C11의 relaxed atomic load 오퍼레이션에 의해 제공되는 volatile 캐스팅 같은 것) 사용해야 합니다. 그러나 volatile 캐스팅은 완화된 순서규칙의 CPU가 자신의 캐쉬에서 “a”的 오래된 값을 로드하는 것을 막지 않는데, 이는 이 코드가 “bar()” 내에 명시적인 메모리 배리어를 필요로 함을 의미합니다.

요약하자면, 컴파일러와 CPU 모두 코드 재배치 최적화를 가할 수 있으므로 여러분은 컴파일러 지시어와 메모리 배리어를 이용해 여러분의 제약을 분명히 전달해야 합니다.



Quick Quiz C.12:

각 CPU 각 자신의 메모리 액세스를 순서대로 본다는 보장은 각 사용자 수준 쓰레드가 자신의 메모리 액세스를 순서대로 본다는 것을 보장하나요? 그 이유는요?



Answer:

아니오. 쓰레드가 한 CPU에서 다른 CPU로 옮겨지며 그 목표 CPU는 원천 CPU의 최근의 메모리 오퍼레이션을 순서대로 보지 않는 경우를 생각해 봅시다. 사용자 모드의 정상성을 지키기 위해, 커널 해커들은 컨텍스트 스위치 경로에서 메모리 배리어를 사용해야만 합니다. 그러나, 컨텍스트 스위치를 안전하게 하기 위해 필요한 락킹은 사용자 수준 테스크가 자신의 액세스를 순서대로 보는데 필요한 메모리 배리어를 자동으로 제공합니다. 그렇다고는 하나, 여러분이 매우 최적화된 스케줄러를 설계한다면, 그것이 커널 수준이든 사용자 수준이든, 이 시나리오를 명심하십시오!



Quick Quiz C.13:

이 코드는 CPU 1의 “while”과 “c”에의 값 할당 사이에 메모리 배리어를 넣음으로써 고쳐질 수 있을까요? 왜죠?



Answer:

아니요. 그런 메모리 배리어는 CPU 1에 지역적인 순서만을 강제합니다. 이는 CPU 0과 CPU 1의 액세스에의 상대적인 순서에는 영향을 끼치지 않아서, 이 할당은 여전히 실패할 수 있습니다. 그러나, 모든 주류 컴퓨터 시스템은 “타동성 (transitivity)”를 제공하기 위한 메커니즘을 제공하는데, 이는 직관적인 인과 순서를 제공합니다: 만약 B가 A의 액세스의 효과를 봤다면, 그리고 C가 B의 액세스의 효과를 봤다면, C는 A의 액세스의 효과도 봐야만 합니다. 요약하자면, 하드웨어 설계자들은 소프트웨어 개발자들에게 약간의 동정은 가졌습니다.



Quick Quiz C.14:

Listing C.3의 CPU 1과 2에 의한 라인 3-5가 인터럽트 핸들러이고, CPU 2의 라인 9는 프로세스 수준에서 수행된다고 해봅시다. 달리 말하자면, 이 표의 세 열의 코드는 같은 CPU에서 수행되나, 첫번째 두 열은 인터럽트 핸들러에서, 그리고 세번째 열은 프로세스 수준에서 수행되므로, 세번째 열의 코드는 첫번째 열의 코드에

의해 인터럽트 당할 수 있습니다. 이 코드가 올바르게 동작하려면, 달리 말해 이 단정문이 격발되는 걸 막기 위해 무언가가 필요할까요?



Answer:

이 단정문은 “e”의 로드가 “a”의 것을 앞서는 것을 보장해야 합니다. 리눅스 커널에서, `barrier()` 기능은 앞의 예에서 메모리 배리어가 사용되었던 것과 상당히 같은 방식으로 이를 달성하기 위해 사용될 수 있습니다. 예를 들어, 이 단정문은 다음과 같이 수정될 수 있습니다:

```
r1 = e;
barrier();
assert(r1 == 0 || a == 1);
```

첫번째 두 열의 코드에는 어떤 변경도 필요치 않은데, 인터럽트 핸들러는 인터럽트 당한 코드의 관점에서는 어토믹하게 수행되기 때문입니다.



Quick Quiz C.15:

Listing C.3의 예에서 CPU 2가 `assert(e==0 || c==1)`를 수행했다면, 이 단정문은 격발될까요?



Answer:

결과는 그 CPU가 “transitivity”를 지원하느냐에 달렸습니다. 달리 말해서, CPU 0은 CPU 1의 “c” 스토어를 본 후에 “e”를 저장했으며, CPU 0의 “c” 로드와 “e” 스토어 사이에 메모리 배리어도 쳤습니다. 만약 다른 CPU가 CPU 0의 “e”로의 스토어를 봤다면, CPU 1의 스토어도 볼 것이 보장될까요?

제가 아는 모든 CPU는 `transitivity`를 제공한다고 주장합니다.



Glossary

Dictionaries are inherently circular in nature.

*“Self Reference in word definitions”,
David Levary et al.*

Associativity: The number of cache lines that can be held simultaneously in a given cache, when all of these cache lines hash identically in that cache. A cache that could hold four cache lines for each possible hash value would be termed a “four-way set-associative” cache, while a cache that could hold only one cache line for each possible hash value would be termed a “direct-mapped” cache. A cache whose associativity was equal to its capacity would be termed a “fully associative” cache. Fully associative caches have the advantage of eliminating associativity misses, but, due to hardware limitations, fully associative caches are normally quite limited in size. The associativity of the large caches found on modern microprocessors typically range from two-way to eight-way.

Associativity Miss: A cache miss incurred because the corresponding CPU has recently accessed more data hashing to a given set of the cache than will fit in that set. Fully associative caches are not subject to associativity misses (or, equivalently, in fully associative caches, associativity and capacity misses are identical).

Atomic: An operation is considered “atomic” if it is not possible to observe any intermediate state. For example, on most CPUs, a store to a properly aligned pointer is atomic, because other CPUs will see either the old value or the new value, but are guaranteed not to see some mixed value containing some pieces of the new and old values.

Atomic Read-Modify-Write Operation: An atomic operation that both reads and writes memory is considered an atomic read-modify-write operation, or atomic RMW operation for short. Although the value written usually depends on the value read, `atomic_xchg()` is the exception that proves this rule.

Bounded Wait Free: A forward-progress guarantee in which every thread makes progress within a specific

finite period of time, the specific time being the bound.

Cache: In modern computer systems, CPUs have caches in which to hold frequently used data. These caches can be thought of as hardware hash tables with very simple hash functions, but in which each hash bucket (termed a “set” by hardware types) can hold only a limited number of data items. The number of data items that can be held by each of a cache’s hash buckets is termed the cache’s “associativity”. These data items are normally called “cache lines”, which can be thought of a fixed-length blocks of data that circulate among the CPUs and memory.

Cache Coherence: A property of most modern SMP machines where all CPUs will observe a sequence of values for a given variable that is consistent with at least one global order of values for that variable. Cache coherence also guarantees that at the end of a group of stores to a given variable, all CPUs will agree on the final value for that variable. Note that cache coherence applies only to the series of values taken on by a single variable. In contrast, the memory consistency model for a given machine describes the order in which loads and stores to groups of variables will appear to occur. See Section 15.2.6 for more information.

Cache Coherence Protocol: A communications protocol, normally implemented in hardware, that enforces memory consistency and ordering, preventing different CPUs from seeing inconsistent views of data held in their caches.

Cache Geometry: The size and associativity of a cache is termed its geometry. Each cache may be thought of as a two-dimensional array, with rows of cache lines (“sets”) that have the same hash value, and columns of cache lines (“ways”) in which every cache line has a different hash value. The associativity of a given cache is its number of columns (hence the

name “way”—a two-way set-associative cache has two “ways”), and the size of the cache is its number of rows multiplied by its number of columns.

Cache Line: (1) The unit of data that circulates among the CPUs and memory, usually a moderate power of two in size. Typical cache-line sizes range from 16 to 256 bytes.

(2) A physical location in a CPU cache capable of holding one cache-line unit of data.

(3) A physical location in memory capable of holding one cache-line unit of data, but that it also aligned on a cache-line boundary. For example, the address of the first word of a cache line in memory will end in 0x00 on systems with 256-byte cache lines.

Cache Miss: A cache miss occurs when data needed by the CPU is not in that CPU’s cache. The data might be missing because of a number of reasons, including: (1) this CPU has never accessed the data before (“startup” or “warmup” miss), (2) this CPU has recently accessed more data than would fit in its cache, so that some of the older data had to be removed (“capacity” miss), (3) this CPU has recently accessed more data in a given set¹ than that set could hold (“associativity” miss), (4) some other CPU has written to the data (or some other data in the same cache line) since this CPU has accessed it (“communication miss”), or (5) this CPU attempted to write to a cache line that is currently read-only, possibly due to that line being replicated in other CPUs’ caches.

Capacity Miss: A cache miss incurred because the corresponding CPU has recently accessed more data than will fit into the cache.

Clash Free: A forward-progress guarantee in which, in the absence of contention, at least one thread makes progress within a finite period of time.

Code Locking: A simple locking design in which a “global lock” is used to protect a set of critical sections, so that access by a given thread to that set is granted or denied based only on the set of threads currently occupying the set of critical sections, not based on what data the thread intends to access. The scalability of a code-locked program is limited by

¹ In hardware-cache terminology, the word “set” is used in the same way that the word “bucket” is used when discussing software caches.

the code; increasing the size of the data set will normally not increase scalability (in fact, will typically *decrease* scalability by increasing “lock contention”). Contrast with “data locking”.

Communication Miss: A cache miss incurred because some other CPU has written to the cache line since the last time this CPU accessed it.

Concurrent: In this book, a synonym of parallel. Please see Appendix A.2 on page 385 for a discussion of the recent distinction between these two terms.

Critical Section: A section of code guarded by some synchronization mechanism, so that its execution constrained by that primitive. For example, if a set of critical sections are guarded by the same global lock, then only one of those critical sections may be executing at a given time. If a thread is executing in one such critical section, any other threads must wait until the first thread completes before executing any of the critical sections in the set.

Data Locking: A scalable locking design in which each instance of a given data structure has its own lock. If each thread is using a different instance of the data structure, then all of the threads may be executing in the set of critical sections simultaneously. Data locking has the advantage of automatically scaling to increasing numbers of CPUs as the number of instances of data grows. Contrast with “code locking”.

Data Race: A race condition in which several CPUs or threads access a variable concurrently, and in which at least one of those accesses is a store and at least one of those accesses is unmarked. It is important to note that while the presence of data races often indicates the presence of bugs, the absence of data races in no way implies the absence of bugs. (See “Marked access”.)

Deadlock Free: A forward-progress guarantee in which, in the absence of failures, at least one thread makes progress within a finite period of time.

Direct-Mapped Cache: A cache with only one way, so that it may hold only one cache line with a given hash value.

Efficiency: A measure of effectiveness normally expressed as a ratio of some metric actually achieved to some maximum value. The maximum value might

be a theoretical maximum, but in parallel programming is often based on the corresponding measured single-threaded metric.

Embarrassingly Parallel: A problem or algorithm where adding threads does not significantly increase the overall cost of the computation, resulting in linear speedups as threads are added (assuming sufficient CPUs are available).

Exclusive Lock: An exclusive lock is a mutual-exclusion mechanism that permits only one thread at a time into the set of critical sections guarded by that lock.

False Sharing: If two CPUs each frequently write to one of a pair of data items, but the pair of data items are located in the same cache line, this cache line will be repeatedly invalidated, “ping-ponging” back and forth between the two CPUs’ caches. This is a common cause of “cache thrashing”, also called “cacheline bouncing” (the latter most commonly in the Linux community). False sharing can dramatically reduce both performance and scalability.

Fragmentation: A memory pool that has a large amount of unused memory, but not laid out to permit satisfying a relatively small request is said to be fragmented. External fragmentation occurs when the space is divided up into small fragments lying between allocated blocks of memory, while internal fragmentation occurs when specific requests or types of requests have been allotted more memory than they actually requested.

Fully Associative Cache: A fully associative cache contains only one set, so that it can hold any subset of memory that fits within its capacity.

Grace Period: A grace period is any contiguous time interval such that any RCU read-side critical section that began before the start of that interval has completed before the end of that same interval. Many RCU implementations define a grace period to be a time interval during which each thread has passed through at least one quiescent state. Since RCU read-side critical sections by definition cannot contain quiescent states, these two definitions are almost always interchangeable.

Hazard Pointer: A scalable counterpart to a reference counter in which an object’s reference count is represented implicitly by a count of the number of special hazard pointers referencing that object.

Heisenbug: A timing-sensitive bug that disappears from sight when you add print statements or tracing in an attempt to track it down.

Hot Spot: Data structure that is very heavily used, resulting in high levels of contention on the corresponding lock. One example of this situation would be a hash table with a poorly chosen hash function.

Humiliatingly Parallel: A problem or algorithm where adding threads significantly *decreases* the overall cost of the computation, resulting in large superlinear speedups as threads are added (assuming sufficient CPUs are available).

Invalidation: When a CPU wishes to write to a data item, it must first ensure that this data item is not present in any other CPUs’ cache. If necessary, the item is removed from the other CPUs’ caches via “invalidation” messages from the writing CPUs to any CPUs having a copy in their caches.

IPI: Inter-processor interrupt, which is an interrupt sent from one CPU to another. IPIs are used heavily in the Linux kernel, for example, within the scheduler to alert CPUs that a high-priority process is now runnable.

IRQ: Interrupt request, often used as an abbreviation for “interrupt” within the Linux kernel community, as in “irq handler”.

Latency: The wall-clock time required for a given operation to complete.

Linearizable: A sequence of operations is “linearizable” if there is at least one global ordering of the sequence that is consistent with the observations of all CPUs and/or threads. Linearizability is much prized by many researchers, but less useful in practice than one might expect [HKLP12].

Lock: A software abstraction that can be used to guard critical sections, as such, an example of a “mutual exclusion mechanism”. An “exclusive lock” permits only one thread at a time into the set of critical sections guarded by that lock, while a “reader-writer lock” permits any number of reading threads, or but one writing thread, into the set of critical sections guarded by that lock. (Just to be clear, the presence of a writer thread in any of a given reader-writer lock’s critical sections will prevent any reader from

entering any of that lock's critical sections and vice versa.)

Lock Contention: A lock is said to be suffering contention when it is being used so heavily that there is often a CPU waiting on it. Reducing lock contention is often a concern when designing parallel algorithms and when implementing parallel programs.

Lock Free: A forward-progress guarantee in which at least one thread makes progress within a finite period of time.

Marked Access: A source-code memory access that uses a special function or macro, such as `READ_ONCE()`, `WRITE_ONCE()`, `atomic_inc()`, and so on, in order to protect that access from compiler and/or hardware optimizations. In contrast, an unmarked access simply mentions the name of the object being accessed, so that in the following, line 2 is the unmarked equivalent of line 1:

```

1  WRITE_ONCE(a, READ_ONCE(b) + READ_ONCE(c));
2  a = b + c;

```

Memory: From the viewpoint of memory models, the main memory, caches, and store buffers in which values might be stored. However, this term is often used to denote the main memory itself, excluding caches and store buffers.

Memory Consistency: A set of properties that impose constraints on the order in which accesses to groups of variables appear to occur. Memory consistency models range from sequential consistency, a very constraining model popular in academic circles, through process consistency, release consistency, and weak consistency.

MESI Protocol: The cache-coherence protocol featuring modified, exclusive, shared, and invalid (MESI) states, so that this protocol is named after the states that the cache lines in a given cache can take on. A modified line has been recently written to by this CPU, and is the sole representative of the current value of the corresponding memory location. An exclusive cache line has not been written to, but this CPU has the right to write to it at any time, as the line is guaranteed not to be replicated into any other CPU's cache (though the corresponding location in main memory is up to date). A shared cache line is (or might be) replicated in some other CPUs' cache,

meaning that this CPU must interact with those other CPUs before writing to this cache line. An invalid cache line contains no value, instead representing "empty space" in the cache into which data from memory might be loaded.

Mutual-Exclusion Mechanism: A software abstraction that regulates threads' access to "critical sections" and corresponding data.

NMI: Non-maskable interrupt. As the name indicates, this is an extremely high-priority interrupt that cannot be masked. These are used for hardware-specific purposes such as profiling. The advantage of using NMIs for profiling is that it allows you to profile code that runs with interrupts disabled.

NUCA: Non-uniform cache architecture, where groups of CPUs share caches and/or store buffers. CPUs in a group can therefore exchange cache lines with each other much more quickly than they can with CPUs in other groups. Systems comprised of CPUs with hardware threads will generally have a NUCA architecture.

NUMA: Non-uniform memory architecture, where memory is split into banks and each such bank is "close" to a group of CPUs, the group being termed a "NUMA node". An example NUMA machine is Sequent's NUMA-Q system, where each group of four CPUs had a bank of memory nearby. The CPUs in a given group can access their memory much more quickly than another group's memory.

NUMA Node: A group of closely placed CPUs and associated memory within a larger NUMA machines.

Obstruction Free: A forward-progress guarantee in which, in the absence of contention, every thread makes progress within a finite period of time.

Overhead: Operations that must be executed, but which do not contribute directly to the work that must be accomplished. For example, lock acquisition and release is normally considered to be overhead, and specifically to be synchronization overhead.

Parallel: In this book, a synonym of concurrent. Please see Appendix A.2 on page 385 for a discussion of the recent distinction between these two terms.

Performance: Rate at which work is done, expressed as work per unit time. If this work is fully serialized,

then the performance will be the reciprocal of the mean latency of the work items.

Pipelined CPU: A CPU with a pipeline, which is an internal flow of instructions internal to the CPU that is in some way similar to an assembly line, with many of the same advantages and disadvantages. In the 1960s through the early 1980s, pipelined CPUs were the province of supercomputers, but started appearing in microprocessors (such as the 80486) in the late 1980s.

Process Consistency: A memory-consistency model in which each CPU's stores appear to occur in program order, but in which different CPUs might see accesses from more than one CPU as occurring in different orders.

Program Order: The order in which a given thread's instructions would be executed by a now-mythical "in-order" CPU that completely executed each instruction before proceeding to the next instruction. (The reason such CPUs are now the stuff of ancient myths and legends is that they were extremely slow. These dinosaurs were one of the many victims of Moore's Law-driven increases in CPU clock frequency. Some claim that these beasts will roam the earth once again, others vehemently disagree.)

Quiescent State: In RCU, a point in the code where there can be no references held to RCU-protected data structures, which is normally any point outside of an RCU read-side critical section. Any interval of time during which all threads pass through at least one quiescent state each is termed a "grace period".

Read-Copy Update (RCU): A synchronization mechanism that can be thought of as a replacement for reader-writer locking or reference counting. RCU provides extremely low-overhead access for readers, while writers incur additional overhead maintaining old versions for the benefit of pre-existing readers. Readers neither block nor spin, and thus cannot participate in deadlocks, however, they also can see stale data and can run concurrently with updates. RCU is thus best-suited for read-mostly situations where stale data can either be tolerated (as in routing tables) or avoided (as in the Linux kernel's System V IPC implementation).

Read-Side Critical Section: A section of code guarded by read-acquisition of some reader-writer synchro-

nization mechanism. For example, if one set of critical sections are guarded by read-acquisition of a given global reader-writer lock, while a second set of critical section are guarded by write-acquisition of that same reader-writer lock, then the first set of critical sections will be the read-side critical sections for that lock. Any number of threads may concurrently execute the read-side critical sections, but only if no thread is executing one of the write-side critical sections.

Reader-Writer Lock: A reader-writer lock is a mutual-exclusion mechanism that permits any number of reading threads, or but one writing thread, into the set of critical sections guarded by that lock. Threads attempting to write must wait until all pre-existing reading threads release the lock, and, similarly, if there is a pre-existing writer, any threads attempting to write must wait for the writer to release the lock. A key concern for reader-writer locks is "fairness": can an unending stream of readers starve a writer or vice versa.

Real Time: A situation in which getting the correct result is not sufficient, but where this result must also be obtained within a given amount of time.

Scalability: A measure of how effectively a given system is able to utilize additional resources. For parallel computing, the additional resources are usually additional CPUs.

Sequence Lock: A reader-writer synchronization mechanism in which readers retry their operations if a writer was present.

Sequential Consistency: A memory-consistency model where all memory references appear to occur in an order consistent with a single global order, and where each CPU's memory references appear to all CPUs to occur in program order.

Starvation Free: A forward-progress guarantee in which, in the absence of failures, every thread makes progress within a finite period of time.

Store Buffer: A small set of internal registers used by a given CPU to record pending stores while the corresponding cache lines are making their way to that CPU. Also called "store queue".

Store Forwarding: An arrangement where a given CPU refers to its store buffer as well as its cache so as to

ensure that the software sees the memory operations performed by this CPU as if they were carried out in program order.

Superscalar CPU: A scalar (non-vector) CPU capable of executing multiple instructions concurrently. This is a step up from a pipelined CPU that executes multiple instructions in an assembly-line fashion—in a superscalar CPU, each stage of the pipeline would be capable of handling more than one instruction. For example, if the conditions were exactly right, the Intel Pentium Pro CPU from the mid-1990s could execute two (and sometimes three) instructions per clock cycle. Thus, a 200 MHz Pentium Pro CPU could “retire”, or complete the execution of, up to 400 million instructions per second.

Synchronization: Means for avoiding destructive interactions among CPUs or threads. Synchronization mechanisms include atomic RMW operations, memory barriers, locking, reference counting, hazard pointers, sequence locking, RCU, non-blocking synchronization, and transactional memory.

Teachable: A topic, concept, method, or mechanism that teachers believe that they understand completely and are therefore comfortable teaching.

Throughput: A performance metric featuring work items completed per unit time.

Transactional Lock Elision (TLE): The use of transactional memory to emulate locking. Synchronization is instead carried out by conflicting accesses to the data to be protected by the lock. In some cases, this can increase performance because TLE avoids contention on the lock word [PD11, Kle14, FIMR16, PMDY20].

Transactional Memory (TM): A synchronization mechanism that gathers groups of memory accesses so as to execute them atomically from the viewpoint of transactions on other CPUs or threads.

Unteachable: A topic, concept, method, or mechanism that the teacher does not understand well is therefore uncomfortable teaching.

Vector CPU: A CPU that can apply a single instruction to multiple items of data concurrently. In the 1960s through the 1980s, only supercomputers had vector capabilities, but the advent of MMX in x86 CPUs and VMX in PowerPC CPUs brought vector processing to the masses.

Wait Free: A forward-progress guarantee in which every thread makes progress within a finite period of time.

Write Miss: A cache miss incurred because the corresponding CPU attempted to write to a cache line that is read-only, most likely due to its being replicated in other CPUs’ caches.

Write-Side Critical Section: A section of code guarded by write-acquisition of some reader-writer synchronization mechanism. For example, if one set of critical sections are guarded by write-acquisition of a given global reader-writer lock, while a second set of critical section are guarded by read-acquisition of that same reader-writer lock, then the first set of critical sections will be the write-side critical sections for that lock. Only one thread may execute in the write-side critical section at a time, and even then only if there are no threads are executing concurrently in any of the corresponding read-side critical sections.

Bibliography

- [AA14] Maya Arbel and Hagit Attiya. Concurrent updates with RCU: Search tree as an example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, page 196–205, Paris, France, 2014. ACM.
- [AAKL06] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, and Charles E. Leiserson. Unbounded transactional memory. *IEEE Micro*, pages 59–69, January–February 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. Vandevoorde, 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.
- [ACA⁺18] A. Aljuhni, C. E. Chow, A. Aljaedi, S. Yusuf, and F. Torres-Reyes. Towards understanding application performance and system behavior with the full dynticks feature. In *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 394–401, 2018.
- [ACHS13] Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. Are lock-free concurrent algorithms practically wait-free?, December 2013. ArXiv:1311.3200v2.
- [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, San Antonio, Texas, USA, June 2003. USENIX Association.
- [Ada11] Andrew Adamatzky. Slime mould solves maze in one pass . . . assisted by gradient of chemo-attractants, August 2011. arXiv:1108.4956.
- [ADF⁺19] Jade Alglave, Will Deacon, Boqun Feng, David Howells, Daniel Lustig, Luc Maranget, Paul E. McKenney, Andrea Parri, Nicholas Pigglin, Alan Stern, Akira Yokosawa, and Peter Zijlstra. Who's afraid of a big bad optimizing compiler?, July 2019. Linux Weekly News.
- [Adv02] Advanced Micro Devices. *AMD x86-64 Architecture Programmer's Manual Volumes 1–5*, 2002.
- [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, Austin, TX, USA, 2011. ACM.
- [AGH⁺11b] Hagit Attiya, Rachid Guerraoui, Danny Helder, 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.
- [AHM09] Hagit Attiya, Eshcar Hillel, and Alessia Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures, SPAA ’09*, pages 69–78, Calgary, AB, Canada, 2009. ACM.
- [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. Rosenberg, M. Stumm, and J. Xenidis. Enabling autonomic behavior in systems software with hot swapping. *IBM Systems Journal*, 42(1):60–76, January 2003.
- [AKK⁺14] Dan Alistarh, Justin Kopinsky, Petr Kuznetsov, Srivatsan Ravi, and Nir Shavit. Inherent limitations of hybrid transactional memory. *CoRR*, abs/1405.5689, 2014.
- [AKNT13] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschinig. 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, Rome, Italy, 2013. Springer-Verlag.
- [AKT13] Jade Alglave, Daniel Kroening, and Michael Tautschinig. 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.
- [AM15] Maya Arbel and Adam Morrison. Predicate RCU: An RCU for scalable concurrent updates. *SIGPLAN Not.*, 50(8):21–30, January 2015.
- [Amd67] Gene Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings, AFIPS ’67 (Spring)*, pages 483–485, Atlantic City, New Jersey, 1967. Association for Computing Machinery.
- [AMD20] AMD. Professional compute products - GPUOpen, March 2020. <https://gpuopen.com/professional-compute/>.
- [AMM⁺17a] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. A formal kernel memory-ordering model (part 1), April 2017. <https://lwn.net/Articles/718628/>.
- [AMM⁺17b] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. A formal kernel memory-ordering model (part 2), April 2017. <https://lwn.net/Articles/720550/>.

- [AMM⁺18] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. Frightening small children and disconcerting grown-ups: Concurrency in the Linux kernel. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 405–418, Williamsburg, VA, USA, 2018. ACM.
- [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, June 2011. <https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/pldi105-sarkar.pdf>.
- [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, Edinburgh, United Kingdom, 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.
- [And19] Jim Anderson. Software transactional memory for real-time systems, August 2019. <https://www.cs.unc.edu/~anderson/projects/rtstm.html>.
- [ARM10] ARM Limited. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R Edition*, 2010.
- [ARM17] ARM Limited. *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)*, 2017.
- [Ash15] Mike Ash. Concurrent memory deallocation in the objective-c runtime, May 2015. mikeash.com: just this guy, you know?
- [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*, New Orleans, LA, USA, June 2011. ACM SIGPLAN.
- [ATS09] Ali-Reza Adl-Tabatabai and Tatiana Shpeisman. Draft specification of transactional language constructs for C++, August 2009. URL: <https://software.intel.com/sites/default/files/ee/47/21569> (may need to append .pdf to view after download).
- [Att10] Hagit Attiya. The inherent complexity of transactional memory and what to do about it. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 1–5, Zurich, Switzerland, 2010. ACM.
- [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.
- [Bah11a] Samy Al Bahra. ck_epoch: Support per-object destructors, October 2011. <https://github.com/concurrencykit/ck/commit/10ffb2e6f1737a30e2dcf3862d105ad45fcd60a4>.

- [Bah11b] Samy Al Bahra. `ck_hp.c`, February 2011. Hazard pointers: https://github.com/concurrencykit/ck/blob/master/src/ck_hp.c.
- [Bah11c] Samy Al Bahra. `ck_sequence.h`, February 2011. Sequence locking: https://github.com/concurrencykit/ck/blob/master/include/ck_sequence.h.
- [Bas18] JF Bastien. P1152R0: Deprecating volatile, October 2018. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1152r0.html>.
- [BBC⁺10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, February 2010.
- [BCR03] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. *SIGPLAN Not.*, 38(1):285–298, 2003.
- [BD13] Paolo Bonzini and Mike Day. RCU implementation for Qemu, August 2013. <https://lists.gnu.org/archive/html/qemu-devel/2013-08/msg02055.html>.
- [BD14] Hans-J. Boehm and Brian Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC ’14, pages 7:1–7:6, Edinburgh, United Kingdom, 2014. ACM.
- [Bec11] Pete Becker. Working draft, standard for programming language C++, February 2011. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf>.
- [BG87] D. Bertsekas and R. Gallager. *Data Networks*. Prentice-Hall, Inc., 1987.
- [BGHZ16] Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zabolotchi. Fast and robust memory reclamation for concurrent data structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA ’16, pages 349–359, Pacific Grove, California, USA, 2016. ACM.
- [BGOS18] Sam Blackshear, Nikos Gorogiannis, Peter W. O’Hearn, and Ilya Sergey. Racerd: Compositional static race detection. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.
- [BGV17] Hans-J. Boehm, Olivier Giroux, and Viktor Vafeiades. P0668r1: Revising the C++ memory model, July 2017. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0668r1.html>.
- [Bha14] Srivatsa S. Bhat. `percpu_rwlock`: Implement the core design of per-CPU reader-writer locks, February 2014. <https://patchwork.kernel.org/patch/2157401/>.
- [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.
- [Bir89] Andrew D. Birrell. *An Introduction to Programming with Threads*. Digital Systems Research Center, January 1989.

- [BJ12] Rex Black and Capers Jones. Economics of software quality: An interview with Capers Jones, part 1 of 2 (podcast transcript), January 2012. <https://www.informati.com/articles/article.aspx?p=1824791>.
- [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://acg.cis.upenn.edu/papers/wddd05_atomic_semantics.pdf [Viewed February 28, 2021].
- [BLM06] C. Blundell, E. C. Lewis, and M. Martin. Subtleties of transactional memory and atomicity semantics. *Computer Architecture Letters*, 5(2), 2006. Available: http://acg.cis.upenn.edu/papers/ca106_atomic_semantics.pdf [Viewed February 28, 2021].
- [BM18] JF Bastien and Paul E. McKenney. P0750r1: Consume, February 2018. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0750r1.html>.
- [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. US Patent 6,842,809, Assigned to International Business Machines Corporation, Washington, DC, January 2005.
- [BMN⁺15] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The problem of programming language concurrency semantics. In Jan Vitek, editor, *Programming Languages and Systems*, volume 9032 of *Lecture Notes in Computer Science*, pages 283–307. Springer Berlin Heidelberg, 2015.
- [BMP08] R. F. Berry, P. E. McKenney, and F. N. Parr. Responsive systems: An introduction. *IBM Systems Journal*, 47(2):197–206, April 2008.
- [Boe05] Hans-J. Boehm. Threads cannot be implemented as a library. *SIGPLAN Not.*, 40(6):261–268, June 2005.
- [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: https://www.usenix.org/event/hotpar09/tech/full_papers/boehm/boehm.pdf [Viewed May 24, 2009].
- [Boe20] Hans Boehm. “Undefined behavior” and the concurrency memory model, August 2020. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2215r0.pdf>.
- [Boh01] Kristoffer Bohmann. Response time still matters, July 2001. URL: http://www.bohmann.dk/articles/response_time_still_matters.html [broken, November 2016].
- [Bon13] Paolo Bonzini. seqlock: introduce read-write seqlock, September 2013. <https://git.qemu.org/?p=qemu.git;a=commit;h=ea753d81e8b085d679f13e4a6023e003e9854d51>.
- [Bon15] Paolo Bonzini. rcu: add rcu library, February 2015. <https://git.qemu.org/?p=qemu.git;a=commit;h=7911747bd46123ef8d8eef2ee49422bb8a4b274f>.

- [Bon21a] Paolo Bonzini. An introduction to lockless algorithms, February 2021. Available: <https://lwn.net/Articles/844224/> [Viewed February 19, 2021].
- [Bon21b] Paolo Bonzini. Lockless patterns: an introduction to compare-and-swap, March 2021. Available: <https://lwn.net/Articles/847973/> [Viewed March 13, 2021].
- [Bon21c] Paolo Bonzini. Lockless patterns: full memory barriers, March 2021. Available: <https://lwn.net/Articles/847481/> [Viewed March 8, 2021].
- [Bon21d] Paolo Bonzini. Lockless patterns: more read-modify-write operations, March 2021. Available: <https://lwn.net/Articles/849237/> [Viewed March 19, 2021].
- [Bon21e] Paolo Bonzini. Lockless patterns: relaxed access and partial memory barriers, February 2021. Available: <https://lwn.net/Articles/846700/> [Viewed February 27, 2021].
- [Bor06] Richard Bornat. Dividing the sheep from the goats, January 2006. Seminar at School of Computing, Univ. of Kent. Abstract is available at https://www.cs.kent.ac.uk/seminar_archive/2005_06/abs_2006_01_24.html. Retracted in July 2014: http://www.eis.mdx.ac.uk/staffpages/r_bornat/papers/camel_hump_retraction.pdf.
- [Bos10] Keith Bostic. Switch lockless programming style from epoch to hazard references, January 2010. <https://github.com/wiredtiger/wiredtiger/commit/dddcc21014fc494a956778360a14d96c762495e09>.
- [BPP⁺16] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. The IX operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Trans. Comput. Syst.*, 34(4):11:1–11:39, December 2016.
- [Bra07] Reg Braithwaite. Don't overthink fizzbuzz, January 2007. <http://weblog.raganwald.com/2007/01/dont-overthink-fizzbuzz.html>.
- [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>.
- [Bro15a] Neil Brown. Pathname lookup in Linux, June 2015. <https://lwn.net/Articles/649115/>.
- [Bro15b] Neil Brown. RCU-walk: faster pathname lookup in Linux, July 2015. <https://lwn.net/Articles/649729/>.
- [Bro15c] Neil Brown. A walk among the symlinks, July 2015. <https://lwn.net/Articles/650786/>.
- [BS75] Paul J. Brown and Ronald M. Smith. Shared data controlled by a plurality of users, May 1975. US Patent 3,886,525, filed June 29, 1973.
- [BS14] Mark Batty and Peter Sewell. The thin-air problem, February 2014. <http://www.cl.cam.ac.uk/~pes20/cpp/notes42.html>.
- [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>.
- [BWCM⁺10] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of Linux scalability to many cores. In *9th USENIX Symposium on Operating System Design and Implementation*, pages 1–16, Vancouver, BC, Canada, October 2010. USENIX.
- [CAK⁺96] Crispin Cowan, Tito Autrey, Charles Krasic, Calton Pu, and Jonathan Walpole. Fast concurrent dynamic linking for an adaptive operating system. In *International Conference on Configurable Distributed Systems (ICCD'S'96)*, pages 108–115, Annapolis, MD, May 1996.
- [CBF13] UPC Consortium, Dan Bonachea, and Gary Funck. UPC language and library specifications, version 1.3. Technical report, UPC Consortium, November 2013.
- [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.
- [CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
- [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, Farmington, Pennsylvania, 2013. ACM.
- [Cli09] Cliff Click. And now some hardware transactional memory comments..., February 2009. URL: <http://www.cliffc.org/blog/2009/02/25/and-now-some-hardware-transactional-memory-comments/>.
- [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.
- [CnRR18] Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Unifying concurrent objects and distributed tasks: Interval-linearizability. *J. ACM*, 65(6), November 2018.
- [Com01] Compaq Computer Corporation. Shared memory, threads, interprocess communication, August 2001. Zipped archive: [wiz_](http://www.csail.mit.edu/papers/sbw-phd-thesis.pdf)

- 2637.txt in https://www.digiater.nl/openvms/freeware/v70/ask_the_wizard/wizard.zip.
- [Coo18] Byron Cook. Formal reasoning about the security of amazon web services. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 38–47, Cham, 2018. Springer International Publishing.
- [Cor02] Compaq Computer Corporation. *Alpha Architecture Reference Manual*. Digital Press, fourth edition, 2002.
- [Cor03] Jonathan Corbet. Driver porting: mutual exclusion with seqlocks, February 2003. <https://lwn.net/Articles/22818/>.
- [Cor04a] Jonathan Corbet. Approaches to realtime Linux, October 2004. URL: <https://lwn.net/Articles/106010/>.
- [Cor04b] Jonathan Corbet. Finding kernel problems automatically, June 2004. <https://lwn.net/Articles/87538/>.
- [Cor04c] Jonathan Corbet. Realtime preemption, part 2, October 2004. URL: <https://lwn.net/Articles/107269/>.
- [Cor06a] Jonathan Corbet. The kernel lock validator, May 2006. Available: <https://lwn.net/Articles/185666/> [Viewed: March 26, 2010].
- [Cor06b] Jonathan Corbet. Priority inheritance in the kernel, April 2006. Available: <https://lwn.net/Articles/178253/> [Viewed June 29, 2009].
- [Cor10a] Jonathan Corbet. Dcache scalability and RCU-walk, December 2010. Available: <https://lwn.net/Articles/419811/> [Viewed May 29, 2017].
- [Cor10b] Jonathan Corbet. sys_membarrier(), January 2010. <https://lwn.net/Articles/369567/>.
- [Cor11] Jonathan Corbet. How to ruin linus’s vacation, July 2011. Available: <https://lwn.net/Articles/452117/> [Viewed May 29, 2017].
- [Cor12] Jonathan Corbet. ACCESS_ONCE(), August 2012. <https://lwn.net/Articles/508991/>.
- [Cor13] Jonathan Corbet. (Nearly) full tickless operation in 3.10, May 2013. <https://lwn.net/Articles/549580/>.
- [Cor14a] Jonathan Corbet. ACCESS_ONCE() and compiler bugs, December 2014. <https://lwn.net/Articles/624126/>.
- [Cor14b] Jonathan Corbet. MCS locks and qspinlocks, March 2014. <https://lwn.net/Articles/590243/>.
- [Cor14c] Jonathan Corbet. Relativistic hash tables, part 1: Algorithms, September 2014. <https://lwn.net/Articles/612021/>.
- [Cor14d] Jonathan Corbet. Relativistic hash tables, part 2: Implementation, September 2014. <https://lwn.net/Articles/612100/>.
- [Cor16a] Jonathan Corbet. Finding race conditions with KCSAN, June 2016. <https://lwn.net/Articles/691128/>.
- [Cor16b] Jonathan Corbet. Time to move to C11 atomics?, June 2016. <https://lwn.net/Articles/691128/>.
- [Cor18] Jonathan Corbet. membarrier(2), October 2018. <https://man7.org/linux/man-pages/man2/membarrier.2.html>.

- [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. URL: <https://lwn.net/Kernel/LDD3/>.
- [CSG99] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: a Hardware/Software Approach*. Morgan Kaufman, 1999.
- [cut17] crates.io user ticki. conc v0.5.0: Hazard-pointer-based concurrent memory reclamation, August 2017. <https://crates.io/crates/conc>.
- [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, page 39–52, Newport Beach, CA, USA, 2011. ACM.
- [Dea18] Will Deacon. [PATCH 00/10] kernel/locking: qspinlock improvements, April 2018. <https://lkml.kernel.org/r/1522947547-24081-1-git-send-email-will.deacon@arm.com>.
- [Dea19] Will Deacon. Re: [PATCH 1/1] Fix: trace sched switch start/stop racy updates, August 2019. <https://lore.kernel.org/lkml/20190821103200.kpufwtviqhpbuv2n@willie-the-truck/>.
- [Den15] Peter Denning. Perspectives on OS foundations. In *SOSP History Day 2015*, SOSP '15, pages 3:1–3:46, Monterey, California, 2015. ACM.
- [Dep06] Department of Computing and Information Systems, University of Melbourne. CSIRAC, 2006. <https://cis.unimelb.edu.au/about/csirac/>.
- [Des09a] Mathieu Desnoyers. *Low-Impact Operating System Tracing*. PhD thesis, Ecole Polytechnique de Montréal, December 2009. Available: <https://lttng.org/files/thesis/desnoyers-dissertation-2009-12-v27.pdf> [Viewed February 27, 2021].
- [Des09b] Mathieu Desnoyers. [RFC git tree] userspace RCU (urcu) for Linux, February 2009. <https://liburcu.org>.
- [DFGG11] Aleksandar Dragovic, 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.
- [DFLO19] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at facebook. *Commun. ACM*, 62(8):62–70, July 2019.
- [DHJ⁺07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter

- Vosshall, and Werner Vogels. Dynamo: Amazon’s highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.
- [DHK12] Vijay D’Silva, Leopold Haller, and Daniel Kroening. Satisfiability solvers are static analyzers. In *Static Analysis Symposium (SAS)*, volume 7460 of *LNCS*, pages 317–333. Springer, 2012.
- [DHL⁺08] Dave Dice, Maurice Herlihy, Doug Lea, Yossi Lev, Victor Luchangco, Wayne Mesard, Mark Moir, Kevin Moore, and Dan Nussbaum. Applications of the adaptive transactional memory test platform. In *3rd ACM SIGPLAN Workshop on Transactional Computing*, pages 1–10, Salt Lake City, UT, USA, February 2008.
- [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: <https://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF> [Viewed January 13, 2008].
- [DKS89] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queuing algorithm. *SIGCOMM ’89*, pages 1–12, 1989.
- [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, Thira, Santorini, Greece, 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.
- [DMD13] Mathieu Desnoyers, Paul E. McKenney, and Michel R. Dagenais. Multi-core systems modeling for formal verification of parallel algorithms. *SIGOPS Oper. Syst. Rev.*, 47(2):51–65, July 2013.
- [DMLP79] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, May 1979.
- [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.
- [dO18a] Daniel Bristot de Oliveira. Deadline scheduler part 2 – details and usage, January 2018. URL: <https://lwn.net/Articles/743946/>.
- [dO18b] Daniel Bristot de Oliveira. Deadline scheduling part 1 – overview and theory, January 2018. URL: <https://lwn.net/Articles/743740/>.
- [dOCdO19] Daniel Bristot de Oliveira, Tommaso Cucinotta, and Rômulo Silva de Oliveira. Modeling the behavior of threads in the PREEMPT_RT Linux kernel using automata. *SIGBED Rev.*, 16(3):63–68, November 2019.

- [Dov90] Ken F. Dove. A high capacity TCP/IP in parallel STREAMS. In *UKUUG Conference Proceedings*, London, June 1990.
- [Dow20] Travis Downs. Gathering intel on Intel AVX-512 transitions, January 2020. <https://travisdowns.github.io/blog/2020/01/17/avxfreq1.html>.
- [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.
- [Duf10a] Joe Duffy. A (brief) retrospective on transactional memory, January 2010. <http://joeduffyblog.com/2010/01/03/a-brief-retrospective-on-transactional-memory/>.
- [Duf10b] Joe Duffy. More thoughts on transactional memory, May 2010. <http://joeduffyblog.com/2010/05/16/more-thoughts-on-transactional-memory/>.
- [Dug10] Abhinav Duggal. Stopping data races using redflag. Master's thesis, Stony Brook University, 2010.
- [Eas71] William B. Easton. Process synchronization without long-term interlock. In *Proceedings of the Third ACM Symposium on Operating Systems Principles*, SOSP '71, pages 95–100, Palo Alto, California, USA, 1971. Association for Computing Machinery.
- [Edg13] Jake Edge. The future of realtime Linux, November 2013. URL: <https://lwn.net/Articles/572740/>.
- [Edg14] Jake Edge. The future of the realtime patch set, October 2014. URL: <https://lwn.net/Articles/617140/>.
- [EGCD03] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. UPC language specifications v1.1, May 2003. URL: <http://upc.gwu.edu> [broken, February 27, 2021].
- [EGMdB11] Stephane Eranian, Eric Gouriou, Tipp Moseley, and Willem de Bruijn. Linux kernel profiling with perf, June 2011. <https://perf.wiki.kernel.org/index.php/Tutorial>.
- [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, Portland, Oregon, USA, 2007. ACM.
- [EMV⁺20a] Marco Elver, Paul E. McKenney, Dmitry Vyukov, Andrey Konovalov, Alexander Potapenko, Kostya Serebryany, Alan Stern, Andrea Parri, Akira Yokosawa, Peter Zijlstra, Will Deacon, Daniel Lustig, Boqun Feng, Joel Fernandes, Jade Alglave, and Luc Maranget. Concurrency bugs should fear the big bad data-race detector (part 1), April 2020. Linux Weekly News.
- [EMV⁺20b] Marco Elver, Paul E. McKenney, Dmitry Vyukov, Andrey Konovalov, Alexander Potapenko, Kostya Serebryany, Alan Stern, Andrea Parri, Akira Yokosawa, Peter Zijlstra, Will Deacon, Daniel Lustig, Boqun Feng, Joel Fernandes,

- Jade Alglave, and Luc Maranget. Concurrency bugs should fear the big bad data-race detector (part 2), April 2020. *Linux Weekly News*.
- [Eng68] Douglas Engelbart. The demo, December 1968. URL: <http://thedemo.org/>.
- [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, Guelph, Ontario, Canada, 2005. IEEE Computer Society.
- [Eri08] Christer Ericson. Aiding pathfinding with cellular automata, June 2008. <http://realtimecollisiondetection.net/blog/?p=57>.
- [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, Guelph, Ontario, Canada, 2005. IEEE Computer Society.
- [ETH11] ETH Zurich. Parallel solver for a perfect maze, March 2011. URL: <http://nativesystems.inf.ethz.ch/pub/Main/WebHomeLecturesParallelProgrammingExercises/pp2011hw04.pdf> [broken, November 2016].
- [Eva11] Jason Evans. Scalable memory allocation using jemalloc, January 2011. <https://engineering.fb.com/2011/01/03/core-data/scalable-memory-allocation-using-jemalloc/>.
- [Fel50] W. Feller. *An Introduction to Probability Theory and its Applications*. John Wiley, 1950.
- [Fen73] J. Fennel. Instruction selection in a two-program counter instruction unit. Technical Report US Patent 3,728,692, Assigned to International Business Machines Corp, Washington, DC, April 1973.
- [Fen15] Boqun Feng. powerpc: Make value-returning atomics fully ordered, November 2015. Git commit: <https://git.kernel.org/linus/49e9cf3f0c04>.
- [FH07] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2):1–61, 2007.
- [FIMR16] Pascal Felber, Shady Issa, Alexander Matveev, and Paolo Romano. Hardware read-write lock elision. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, London, United Kingdom, 2016. Association for Computing Machinery.
- [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.
- [Fra04] Keir Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, February 2004.

- [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: <https://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf> [Viewed May 22, 2011].
- [FSP⁺17] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. Mixed-size concurrency: ARM, POWER, C/C++11, and SC. *SIGPLAN Not.*, 52(1):429–442, January 2017.
- [GAJM15] Alex Groce, Iftekhar Ahmed, Carlos Jensen, and Paul E. McKenney. How verified is my code? falsification-driven verification (t). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE ’15, pages 737–748, Washington, DC, USA, 2015. IEEE Computer Society.
- [Gar90] Arun Garg. Parallel STREAMS: a multi-processor implementation. In *USENIX Conference Proceedings*, pages 163–176, Berkeley CA, February 1990. USENIX Association. Available: <https://archive.org/details/1990-proceedings-winter-dc/page/163/mode/2up>.
- [Gar07] Bryan Gardiner. IDF: Gordon Moore predicts end of Moore’s law (again), September 2007. Available: <https://www.wired.com/2007/09/idf-gordon-mo-1/> [Viewed: February 27, 2021].
- [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 Embedded Architectures and Compilers*, HiPEAC’10, pages 81–95, Pisa, Italy, 2010. Springer-Verlag.
- [GG14] Vincent Gramoli and Rachid Guerraoui. Democratizing transactional programming. *Commun. ACM*, 57(1):86–93, January 2014.
- [GGK18] Christina Giannoula, Georgios Goumas, and Nectarios Koziris. Combining HTM with RCU to speed up graph coloring on multicore platforms. In Rio Yokota, Michèle Weiland, David Keyes, and Carsten Trinitis, editors, *High Performance Computing*, pages 350–369, Cham, 2018. Springer International Publishing.
- [GGL⁺19] Rachid Guerraoui, Hugo Guioux, Renaud Lachaize, Vivien Quéma, and Vasileios Trigonakis. Lock–unlock: Is that all? a pragmatic analysis of locking in software systems. *ACM Trans. Comput. Syst.*, 36(1):1:1–1:149, March 2019.
- [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: <https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-9.pdf> [Viewed: October 11, 2004].

- [GHH⁺14] Alex Groce, Klaus Havelund, Gerard J. Holzmann, Rajeev Joshi, and Ru-Gang Xu. Establishing flight software reliability: testing, model checking, constraint-solving, monitoring and learning. *Ann. Math. Artif. Intell.*, 70(4):315–349, 2014.
- [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, November 1995. Technical Report CSRI-339, Available: <ftp://ftp.cs.toronto.edu/pub/reports/csri/339/339.ps>.
- [Gle10] Thomas Gleixner. Realtime linux: academia v. reality, July 2010. URL: <https://lwn.net/Articles/397422/>.
- [Gle12] Thomas Gleixner. Linux -rt kvm guest demo, December 2012. Personal communication.
- [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.
- [Gol18] David Goldblatt. P1202: Asymmetric fences, October 2018. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1202r0.pdf>.
- [Gol19] David Goldblatt. There might not be an elegant OOTA fix, October 2019. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1916r0.pdf>.
- [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.
- [Gra91] Jim Gray. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann, 1991.
- [Gra02] Jim Gray. Super-servers: Commodity computer clusters pose a software challenge, April 2002. 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].
- [Gre19] Brendan Gregg. *BPF Performance Tools: Linux System and Application Observability*. Addison-Wesley Professional, 1st edition, 2019.
- [Gri00] Scott Griffen. Internet pioneers: Doug Englebart, May 2000. Available: <https://www.ibiblio.org/pioneers/englebart.html> [Viewed November 28, 2008].

- [Gro01] The Open Group. Single UNIX specification, July 2001. <http://www.opengroup.org/onlinepubs/007908799/index.html>.
- [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, Montreal, Quebec, Canada, October 2007. ACM. Available: https://homes.cs.washington.edu/~dkg/papers/analogy_oopsla07.pdf [Viewed February 27, 2021].
- [GRY12] Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. Verifying highly concurrent algorithms with grace (extended version), July 2012. <https://software.imdea.org/~gotsman/papers/recycling-esop13-ext.pdf>.
- [GRY13] Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. Verifying concurrent memory reclamation algorithms with grace. In *ESOP'13: European Symposium on Programming*, pages 249–269, Rome, Italy, 2013. Springer-Verlag.
- [GT90] Gary Graunke and Shreekant Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–69, June 1990.
- [Gui18] Hugo Guiroux. *Understanding the performance of mutual exclusion algorithms on modern multicore machines*. PhD thesis, Université Grenoble Alpes, 2018. <https://hugoguiroux.github.io/assets/thesis.pdf>.
- [Gwy15] David Gwynne. introduce srp, which according to the manpage i wrote is short for “shared reference pointers”, July 2015. https://github.com/openbsd/src/blob/HEAD/sys/kern/kern_srp.c.
- [GYW⁺19] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. Pisces: A scalable and efficient persistent transactional memory. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 913–928, Renton, WA, USA, 2019. USENIX Association.
- [Har16] "No Bugs" Hare. Infographics: Operation costs in CPU clock cycles, September 2016. <http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/>.
- [Hay20] Timothy Hayes. A shift to concurrency, October 2020. <https://community.arm.com/developer/research/b/articles/posts/arms-transactional-memory-extension-support->.
- [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, Seattle, WA, 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, Seattle, WA, USA, March 1990.

- [Her91] Maurice Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, January 1991.
- [Her93] Maurice Herlihy. A methodology for 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, Chicago, IL, USA, 2005. ACM Press.
- [Her11] Benjamin Herrenschmidt. powerpc: Fix atomic_xxx_return barrier semantics, November 2011. Git commit: <https://git.kernel.org/linus/b97021f85517>.
- [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*, Ischia, Italy, 2013. ACM.
- [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, Toulouse, France, 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. In *ISCA '93: Proceeding of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, San Diego, CA, USA, 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, March 2006. <https://www.kernel.org/doc/Documentation/memory-barriers.txt>.

- [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.
- [Hor18] Jann Horn. Reading privileged memory with a side-channel, January 2018. <https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html>.
- [HOS89] James P. Hennessy, Damian L. Osisek, and Joseph W. Seigh II. Passive serialization in a multitasking environment. Technical Report US Patent 4,809,168, Assigned to International Business Machines Corp, Washington, DC, February 1989.
- [How12] Phil Howard. *Extending Relativistic Programming to Multiple Writers*. PhD thesis, Portland State University, 2012.
- [HP95] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 1995.
- [HP11] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, Fifth Edition*. Morgan Kaufman, 2011.
- [HP17] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, Sixth Edition*. Morgan Kaufman, 2017.
- [Hra13] Adam Hraška. Read-copy-update for helenos. Master’s thesis, Charles University in Prague, Faculty of Mathematics and Physics, Department of Distributed and Dependable Systems, 2013.
- [HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, Burlington, MA, USA, 2008.
- [HSLS20] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The Art of Multiprocessor Programming, 2nd Edition*. Morgan Kaufmann, Burlington, MA, USA, 2020.
- [HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.
- [HW92] Wilson C. Hsieh and William E. Weihl. Scalable reader-writer locks for parallel systems. In *Proceedings of the 6th International Parallel Processing Symposium*, pages 216–230, Beverly Hills, CA, USA, March 1992.
- [HW11] Philip W. Howard and Jonathan Walpole. A relativistic enhancement to software transactional memory. In *Proceedings of the 3rd USENIX conference on Hot topics in parallelism*, HotPar’11, pages 1–6, Berkeley, CA, 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.
- [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, January 2007. URL: <http://www.b-eye-network.com/view/3365> [broken, February 2021].
- [Int92] International Standards Organization. *Information Technology - Database Language SQL*. ISO, 1992. Available (Second informal review draft of ISO/IEC 9075:1992): <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.
- [Int04b] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, 2004.
- [Int04c] International Business Machines Corporation. z/Architecture principles of operation, May 2004. Available: <http://publibz.boulder.ibm.com/epubs/pdf/dz9zr003.pdf> [Viewed: February 16, 2005].
- [Int07] Intel Corporation. *Intel 64 Architecture Memory Ordering White Paper*, 2007.
- [Int11] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*, 2011. Available: <http://www.intel.com/Assets/PDF/manual/253668.pdf> [Viewed: February 12, 2011].
- [Int16] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*, 2016.
- [Int20] Intel Corporation. *Intel Transactional Synchronization Extensions (Intel TSX) Programming Considerations*, 2021.1 edition, December 2020. In *Intel C++ Compiler Classic Developer Guide and Reference*, https://software.intel.com/content/dam/develop/external/us/en/documents/cpp_compiler_classic.pdf, page 1506.
- [Jac88] Van Jacobson. Congestion avoidance and control. In *SIGCOMM '88*, pages 314–329, August 1988.
- [Jac93] Van Jacobson. Avoid read-side locking via delayed free, September 1993. private communication.
- [Jac08] Daniel Jackson. MapReduce course, January 2008. Available: <https://sites.google.com/site/mriap2008/> [Viewed January 3, 2013].
- [Jef14] Alan Jeffrey. Jmm revision status, July 2014. <https://mail.openjdk.java.net/pipermail/jmm-dev/2014-July/000072.html>.
- [JLK16a] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization (KASLR) with Intel TSX, July 2016. Black Hat USA 2018 <https://www.blackhat.com/us-16/briefings.html#breaking-kernel-address-space-layout-randomization-kaslr-with-intel-tsx>.
- [JLK16b] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with Intel TSX. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 380–392, Vienna, Austria, 2016. ACM.

- [JMRR02] Benedict Joseph Jackson, Paul E. McKenney, Ramakrishnan Rajamony, and Ronald Lynn Rockhold. Scalable interruptible queue locks for shared-memory multiprocessor. US Patent 6,473,819, Assigned to International Business Machines Corporation, Washington, DC, October 2002.
- [Joh77] Stephen Johnson. Lint, a C program checker, December 1977. Computer Science Technical Report 65, Bell Laboratories.
- [Joh95] Aju John. Dynamic vnodes – design and implementation. In *USENIX Winter 1995*, pages 11–23, New Orleans, LA, January 1995. USENIX Association. Available: https://www.usenix.org/publications/library/proceedings/neworl/full_papers/john.a [Viewed October 1, 2010].
- [Jon11] Dave Jones. Trinity: A system call fuzzer. In *Proceedings of the 13th Ottawa Linux Symposium*, pages ???–???, Ottawa, Canada, June 2011. Project repository: <https://github.com/kernelslacker/trinity>.
- [JSG12] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional memory architecture and implementation for IBM System z, December 2012. The 45th Annual IEEE/ACM International Symposium on MicroArchitecture, URL: <https://www.microarch.org/micro45/talks-posters/3-jacobi-presentation.pdf>.
- [Kaa15] Frans Kaashoek. Parallel computing and the os. In *SOSP History Day*, October 2015.
- [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*, New York, New York, United States, 2006. ACM SIGPLAN.
- [KDI20] Alex Kogan, Dave Dice, and Shady Issa. Scalable range locks for scalable address spaces and beyond. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, Heraklion, Greece, 2020. Association for Computing Machinery.
- [Kel17] Michael J. Kelly. How might the manufacturability of the hardware at device level impact on exascale computing?, 2017. Keynote speech at Multicore World 2017, URL: <https://openparallel.com/multicore-world-2017/program-2017/abstracts2017/>.
- [Ken20] Chris Kennelly. TCMalloc overview, February 2020. <https://github.io/tcmalloc/overview.html>.
- [KFC11] KFC. Memristor processor solves mazes, March 2011. URL: <https://www.technologyreview.com/2011/03/03/196572/memristor-processor-solves-mazes/>.
- [Khi14] Maxim Khizhinsky. Memory management schemes, June 2014. <https://kukuruku.co/post/lock-free-data-structures-the-inside-memory-management-schemes/>.
- [Khi15] Max Khiszinsky. Lock-free data structures. the inside. RCU, February 2015. <https://kukuruku.co/post/lock-free-data-structures-the-inside-rcu/>.
- [Kis14] Jan Kiszka. Real-time virtualization - how crazy are we? In *Linux Plumbers Conference*, Duesseldorf, Germany, October 2014. URL: <https://blog.linuxplumbersconf.org/2014/ocw/proposals/1935>.

- [Kiv13] Avi Kivity. rcu: add basic read-copy-update implementation, August 2013. <https://github.com/cloudius-systems/osv/commit/94b69794fb9e6c99d78ca9a58ddae1c31256b43>.
- [Kiv14a] Avi Kivity. rcu hashtable, July 2014. <https://github.com/cloudius-systems/osv/commit/7fa2728e5d03b2174b4a39d94b21940d11926e90>.
- [Kiv14b] Avi Kivity. rcu: introduce an rcu list type, April 2014. <https://github.com/cloudius-systems/osv/commit/4e46586093aeaf339fef8e08d123a6f6b0abde5b>.
- [KL80] H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems*, 5(3):354–382, September 1980.
- [Kle14] Andi Kleen. Scaling existing lock-based applications with lock elision. *Commun. ACM*, 57(3):52–56, March 2014.
- [Kle17] Matt Klein. Envoy threading model, July 2017. <https://blog.envoyproxy.io/envoy-threading-model-a8d44b922310>.
- [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.
- [KM13] Konstantin Khlebnikov and Paul E. McKenney. RCU: non-atomic assignment to long/pointer variables in gcc, January 2013. <https://lore.kernel.org/lkml/50F52FC8.4000701@openvz.org/>.
- [KMK⁺19] Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhava Krishnan Ramanathan, and Changwoo Min. Mv-rlu: Scaling read-log-update with multi-versioning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’19, pages 779–792, Providence, RI, USA, 2019. ACM.
- [Kni86] Tom Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP ’86, pages 105–112, Cambridge, Massachusetts, USA, 1986. ACM.
- [Kni08] John U. Knickerbocker. 3D chip technology. *IBM Journal of Research and Development*, 52(6), November 2008. URL: <http://www.research.ibm.com/journal/rd52-6.html> [Link to each article is broken as of November 2016; Available via <https://ieeexplore.ieee.org/xpl/tocresult.jsp?isnumber=5388557>].
- [Knu73] Donald Knuth. *The Art of Computer Programming*. Addison-Wesley, 1973.
- [Kra17] Vlad Krasnov. On the dangers of Intel’s frequency scaling, November 2017. <https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/>.
- [KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.
- [KS17a] Michalis Kokologiannakis and Konstantinos Sagonas. Stateless model checking of the linux kernel’s hierarchical read-copy update (Tree RCU). Technical report, National Technical University of Athens, January 2017. <https://github.com/michalis-/rcu/blob/master/rcupaper.pdf>.

- [KS17b] Michalis Kokologiannakis and Konstantinos Sagonas. Stateless model checking of the Linux kernel’s hierarchical read-copy-update (Tree RCU). In *Proceedings of International SPIN Symposium on Model Checking of Software*, SPIN 2017, New York, NY, USA, July 2017. ACM.
- [KS19] Michalis Kokologiannakis and Konstantinos Sagonas. Stateless model checking of the linux kernel’s read—copy update (RCU). *Int. J. Softw. Tools Technol. Transf.*, 21(3):287–306, June 2019.
- [KWS97] Leonidas Kothothanassis, Robert W. Wisniewski, and Michael L. Scott. Scheduler-conscious synchronization. *ACM Transactions on Computer Systems*, 15(1):3–40, February 1997.
- [Lam74] Leslie Lamport. A new solution of Dijkstra’s concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.
- [Lam77] Leslie Lamport. Concurrent reading and writing. *Commun. ACM*, 20(11):806–811, November 1977.
- [Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [LBD⁺04] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fahndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting software. *IEEE Softw.*, 21(3):92–100, May 2004.
- [Lea97] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison Wesley Longman, Reading, MA, USA, 1997.
- [Lem18] Daniel Lemire. AVX-512: when and how to use these new instructions, September 2018. <https://lemire.me/blog/2018/09/07/avx-512-when-and-how-to-use-these-new-instructions/>.
- [LGW⁺15] H. Q. Le, G. L. Guthrie, D. E. Williams, M. M. Michael, B. G. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike. Transactional memory support in the ibm power8 processor. *IBM J. Res. Dev.*, 59(1):8:1–8:14, January 2015.
- [LHF05] Michael Lyons, Bill Hay, and Brad Frey. PowerPC storage model and AIX programming, November 2005. <http://www.ibm.com/developerworks/systems/articles/powerpc.html>.
- [Lis88] Barbara Liskov. Distributed programming in Argus. *Commun. ACM*, 31(3):300–312, 1988.
- [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, Calgary, AB, Canada, 2009. ACM.
- [LLS13] Yujie Liu, Victor Luchangco, and Michael Spear. Mindicators: A scalable approach to quiescence. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ICDCS ’13, pages 206–215, Washington, DC, USA, 2013. IEEE Computer Society.
- [LMKM16] Lihao Liang, Paul E. McKenney, Daniel Kroening, and Tom Melham. Verification of the tree-based hierarchical read-copy update in the Linux kernel. Technical report, Cornell University Library, October 2016. <https://arxiv.org/abs/1610.03052>.

- [LMKM18] Lihao Liang, Paul E. McKenney, Daniel Kroening, and Tom Melham. Verification of tree-based hierarchical Read-Copy Update in the Linux Kernel. In *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19–23, 2018*, 2018.
- [Loc02] Doug Locke. Priority inheritance: The real story, July 2002. URL: <http://www.linuxdevices.com/articles/AT5698775833.html> [broken, November 2016], page capture available at <https://www.math.unipd.it/%7Etullio/SCD/2007/Materiale/Locke.pdf>.
- [Lom77] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *SIGSOFT Softw. Eng. Notes*, 2(2):128–137, 1977. URL: <http://portal.acm.org/citation.cfm?id=808319#>.
- [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, Dallas, Texas, United States, 1986. IEEE Computer Society Press.
- [LS11] Yujie Liu and Michael Spear. Toxic transactions. In *TRANSACT 2011*, San Jose, CA, USA, June 2011. ACM SIGPLAN.
- [LSLK14] Carl Leonardsson, Kostis Sagonas, Truc Nguyen Lam, and Michalis Kokologiannakis. Nidhugg, July 2014. <https://github.com/nidhugg/nidhugg>.
- [LVK⁺17] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. *SIGPLAN Not.*, 52(6):618–632, June 2017.
- [LZC14] Ran Liu, Heng Zhang, and Haibo Chen. Scalable read-mostly synchronization using passive reader-writer locks. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 219–230, Philadelphia, PA, June 2014. USENIX Association.
- [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. URL: <https://www.kernel.org/doc/ols/2001/read-copy.pdf>, http://www.rdrop.com/users/paulmck/RCU_rclock_OLS.2001.05.01c.pdf.
- [Mar17] Luc Maraget. Aarch64 model vs. hardware, May 2017. <http://pauillac.inria.fr/~maranget/cats7/model-aarch64/specific.html>.
- [Mar18] Catalin Marinas. Queued spinlocks model, March 2018. <https://git.kernel.org/pub/scm/linux/kernel/git/cmarinas/kernel-tla.git>.
- [Mas92] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, New York, NY, 1992.
- [Mat17] Norm Matloff. *Programming on Parallel Machines*. University of California, Davis, Davis, CA, USA, 2017.
- [MB20] Paul E. McKenney and Hans Boehm. P2055R0: A relaxed guide to memory_order_relaxed, January 2020. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2055r0.pdf>.

- [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)*, Austin, Texas, United States, 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, September 2012. Technical report paulmck.2012.09.17, <http://rdrop.com/users/paulmck/techreports/survey.2012.09.17a.pdf>.
- [McK90] 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].
- [McK95] Paul E. McKenney. Differential profiling. In *MASCOTS 1995*, pages 237–241, Toronto, Canada, January 1995.
- [McK96a] Paul E. McKenney. *Pattern Languages of Program Design*, volume 2, chapter 31: Selecting Locking Designs for Parallel Programs, pages 501–531. Addison-Wesley, June 1996. Available: <http://www.rdrop.com/users/paulmck/scalability/paper/mutexdesignpat.pdf> [Viewed February 17, 2005].
- [McK96b] Paul E. McKenney. Selecting locking primitives for parallel programs. *Communications of the ACM*, 39(10):75–82, October 1996.
- [McK99] Paul E. McKenney. Differential profiling. *Software - Practice and Experience*, 29(3):219–234, 1999.
- [McK01] Paul E. McKenney. RFC: patch to allow lock-free traversal of lists with insertion, October 2001. Available: <https://lore.kernel.org/lkml/200110090155.f991tPt22329@eng4.beaverton.ibm.com/> [Viewed January 05, 2021].
- [McK03] Paul E. McKenney. Using RCU in the Linux 2.5 kernel. *Linux Journal*, 1(114):18–26, October 2003. Available: <https://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: <https://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: <https://www.linuxjournal.com/article/8212> <http://www.rdrop.com/users/paulmck/scalability/paper/ordering.2007.09.19a.pdf> [Viewed November 30, 2007].

- [McK05c] Paul E. McKenney. A realtime preemption overview, August 2005. URL: <https://lwn.net/Articles/146861/>.
- [McK06] Paul E. McKenney. Sleepable RCU, October 2006. Available: <https://lwn.net/Articles/202847/> Revised: <http://www.rdrop.com/users/paulmck/RCU/srcu.2007.01.14a.pdf> [Viewed August 21, 2006].
- [McK07a] Paul E. McKenney. The design of preemptible read-copy-update, October 2007. Available: <https://lwn.net/Articles/253651/> [Viewed October 25, 2007].
- [McK07b] Paul E. McKenney. Immunize `rcu_dereference()` against crazy compiler writers, October 2007. Git commit: <https://git.kernel.org/linus/97b430320ce7>.
- [McK07c] Paul E. McKenney. [PATCH] QRCU with lockless fastpath, February 2007. Available: <https://lkml.org/lkml/2007/2/25/18> [Viewed March 27, 2008].
- [McK07d] Paul E. McKenney. Priority-boosting RCU read-side critical sections, February 2007. <https://lwn.net/Articles/220677/>.
- [McK07e] Paul E. McKenney. RCU and unloadable modules, January 2007. Available: <https://lwn.net/Articles/217484/> [Viewed November 22, 2007].
- [McK07f] Paul E. McKenney. Using Promela and Spin to verify parallel algorithms, August 2007. Available: <https://lwn.net/Articles/243851/> [Viewed September 8, 2007].
- [McK08a] Paul E. McKenney. Hierarchical RCU, November 2008. <https://lwn.net/Articles/305782/>.
- [McK08b] Paul E. McKenney. `rcu: fix rcu_try_flip_waitack_needed()` to prevent grace-period stall, May 2008. Git commit: <https://git.kernel.org/linus/d7c0651390b6>.
- [McK08c] Paul E. McKenney. `rcu: fix misplaced mb() in rcu_enter/exit_nohz()`, March 2008. Git commit: <https://git.kernel.org/linus/ae66be9b71b1>.
- [McK08d] Paul E. McKenney. RCU part 3: the RCU API, January 2008. Available: <https://lwn.net/Articles/264090/> [Viewed January 10, 2008].
- [McK08e] Paul E. McKenney. "Tree RCU": scalable classic RCU implementation, December 2008. Git commit: <https://git.kernel.org/linus/64db4cff99c>.
- [McK08f] Paul E. McKenney. What is RCU? part 2: Usage, January 2008. Available: <https://lwn.net/Articles/263130/> [Viewed January 4, 2008].
- [McK09a] Paul E. McKenney. Re: [PATCH fyi] RCU: the bloatwatch edition, January 2009. Available: <https://lkml.org/lkml/2009/1/14/449> [Viewed January 15, 2009].
- [McK09b] Paul E. McKenney. Transactional memory everywhere?, September 2009. <https://paulmck.livejournal.com/13841.html>.
- [McK11a] Paul E. McKenney. 3.0 and RCU: what went wrong, July 2011. <https://lwn.net/Articles/453002/>.

- [McK11b] Paul E. McKenney. Concurrent code and expensive instructions, January 2011. Available: <https://lwn.net/Articles/423994> [Viewed January 28, 2011].
- [McK11c] Paul E. McKenney. Transactional memory everywhere: Htm and cache geometry, June 2011. <https://paulmck.livejournal.com/tag/transactional%20memory%20everywhere>.
- [McK11d] Paul E. McKenney. Validating memory barriers and atomic instructions, December 2011. <https://lwn.net/Articles/470681/>.
- [McK11e] Paul E. McKenney. Verifying parallel software: Can theory meet practice?, January 2011. <http://www.rdrop.com/users/paulmck/scalability/paper/VericoTheoryPractice.2011.01.28a.pdf>.
- [McK12a] Paul E. McKenney. Beyond expert-only parallel programming? In *Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability*, RACES '12, pages 25–32, Tucson, Arizona, USA, 2012. ACM.
- [McK12b] Paul E. McKenney. Making RCU safe for battery-powered devices, February 2012. Available: <http://www.rdrop.com/users/paulmck/RCU/RCUdynclocks.2012.02.15b.pdf> [Viewed March 1, 2012].
- [McK12c] Paul E. McKenney. Retrofitted parallelism considered grossly sub-optimal. In *4th USENIX Workshop on Hot Topics on Parallelism*, page 7, Berkeley, CA, USA, June 2012.
- [McK12d] Paul E. McKenney. Signed overflow optimization hazards in the kernel, August 2012. <https://lwn.net/Articles/511259/>.
- [McK12e] Paul E. McKenney. Transactional memory everywhere: Hardware transactional lock elision, May 2012. Available: <https://paulmck.livejournal.com/32267.html> [Viewed January 28, 2021].
- [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. N4037: Non-transactional implementation of atomic tree move, May 2014. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4037.pdf>.
- [McK14c] Paul E. McKenney. Proper care and feeding of return values from `rcu_dereference()`, February 2014. https://www.kernel.org/doc/Documentation/RCU/rcu_dereference.txt.
- [McK14d] Paul E. McKenney. The RCU API, 2014 edition, September 2014. <https://lwn.net/Articles/609904/>.
- [McK14e] Paul E. McKenney. Recent read-mostly research, November 2014. <https://lwn.net/Articles/619355/>.
- [McK15a] Paul E. McKenney. Formal verification and Linux-kernel concurrency. In *Compositional Verification Methods for Next-Generation Concurrency*, Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2015. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

- [McK15b] Paul E. McKenney. [PATCH tip/core/rcu 01/10] rcu: Make `rcu_nmi_enter()` handle nesting, January 2015. <https://lore.kernel.org/lkml/1420651257-553-1-git-send-email-paulmck@linux.vnet.ibm.com/>.
- [McK15c] Paul E. McKenney. Practical experience with formal verification tools. In *Verified Trustworthy Software Systems Specialist Meeting*. The Royal Society, April 2015. <http://www.rdrop.com/users/paulmck/scalability/paper/Validation.2016.04.06e.SpecMtg.pdf>.
- [McK15d] Paul E. McKenney. RCU requirements part 2 — parallelism and software engineering, August 2015. <https://lwn.net/Articles/652677/>.
- [McK15e] Paul E. McKenney. RCU requirements part 3, August 2015. <https://lwn.net/Articles/653326/>.
- [McK15f] Paul E. McKenney. Re: [patch tip/locking/core v4 1/6] powerpc: atomic: Make `*xchg` and `*cmpxchg` a full barrier, October 2015. Email thread: <https://lore.kernel.org/lkml/20151014201916.GB3910@linux.vnet.ibm.com/>.
- [McK15g] Paul E. McKenney. Requirements for RCU part 1: the fundamentals, July 2015. <https://lwn.net/Articles/652156/>.
- [McK16] Paul E. McKenney. Beyond the Issaquah challenge: High-performance scalable complex updates, September 2016. <http://www2.rdrop.com/users/paulmck/RCU/Updates.2016.09.19i.CPPCON.pdf>.
- [McK17] Paul E. McKenney. Verification challenge 6: Linux-kernel Tree RCU, June 2017. <https://paulmck.livejournal.com/46993.html>.
- [McK19a] Paul E. McKenney. A critical RCU safety property is... Ease of use! In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, pages 132–143, Haifa, Israel, 2019. ACM.
- [McK19b] Paul E. McKenney. The RCU API, 2019 edition, January 2019. <https://lwn.net/Articles/777036/>.
- [McK19c] Paul E. McKenney. RCU's first-ever CVE, and how i lived to tell the tale, January 2019. linux.conf.au Slides: <http://www.rdrop.com/users/paulmck/RCU/cve.2019.01.23e.pdf> Video: <https://www.youtube.com/watch?v=hZX1aokdNiY>.
- [MCM02] Paul E. McKenney, Kevin A. Clossen, and Raghupathi Malige. Lingering locks with fairness control for multi-node computer systems. US Patent 6,480,918, Assigned to International Business Machines Corporation, 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, November 2013. <https://lwn.net/Articles/573431/>.

- [MDJ13b] Paul E. McKenney, Mathieu Desnoyers, and Lai Jiangshan. URCU-protected queues and stacks, November 2013. <https://lwn.net/Articles/573433/>.
- [MDJ13c] Paul E. McKenney, Mathieu Desnoyers, and Lai Jiangshan. User-space RCU, November 2013. <https://lwn.net/Articles/573424/>.
- [MDR16] Paul E. McKenney, Will Deacon, and Luis R. Rodriguez. Semantics of MMIO mapping attributes across architectures, August 2016. <https://lwn.net/Articles/698014/>.
- [MDSS20] Hans Meuer, Jack Dongarra, Erich Strohmaier, and Horst Simon. Top 500: The list, November 2020. Available: <https://top500.org/lists/> [Viewed March 6, 2021].
- [Men16] Alexis Menard. Move OneWriterSeqLock and SharedMemorySeqLockBuffer from content/ to device/base/synchronization, September 2016. <https://source.chromium.org/chromium/chromium/src/+/b39a3082846d5877a15e8b7e18d66cb142abe8af>.
- [Mer11] Rick Merritt. IBM plants transactional memory in CPU, August 2011. EE Times <https://www.eetimes.com/ibm-plants-transactional-memory-in-cpu/>.
- [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. URL: <https://archives.pdx.edu/ds/psu/10386> [Viewed February 13, 2021].
- [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.
- [Mic02] Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 21–30, August 2002.
- [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.

- [Mic08] Microsoft. *FlushProcessWriteBuffers* function, 2008. <https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-flushprocesswritebuffers>.
- [Mic18] Maged Michael. Rewrite from experimental, use of deterministic schedule, improvements, June 2018. Hazard pointers: <https://github.com/facebook/folly/commit/d42832d2a529156275543c7fa7183e1321df605d>.
- [Mil06] David S. Miller. Re: [PATCH, RFC] RCU : OOM avoidance and lower latency, January 2006. Available: <https://lkml.org/lkml/2006/1/7/22> [Viewed February 29, 2012].
- [MJST16] Paul E. McKenney, Alan Jeffrey, Ali Sezgin, and Tony Tye. Out-of-thin-air execution is vacuous, July 2016. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0422r0.html>.
- [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.
- [MKM12] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 183–196, Bern, Switzerland, 2012. ACM.
- [ML82] Udi Manber and Richard E. Ladner. Concurrency control in a dynamic search structure. Technical Report 82-01-01, Department of Computer Science, University of Washington, Seattle, Washington, January 1982.
- [ML84] Udi Manber and Richard E. Ladner. Concurrency control in a dynamic search structure. *ACM Transactions on Database Systems*, 9(3):439–455, September 1984.
- [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, March 2000. URL: http://kernel.nic.funet.fi/pub/linux/kernel/v2.3/patch-html/patch-2.3.49/linux_include_linux_brlock.h.html.
- [MMM⁺20] Paul E. McKenney, Maged Michael, Jens Maurer, Peter Sewell, Martin Uecker, Hans Boehm, Hubert Tong, Niall Douglas, Thomas Rodgers, Will Deacon, Michael Wong, David Goldblatt, Kostya Serebryany, and Anthony Williams. P1726R4: Pointer lifetime-end zap, July 2020. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1726r4.pdf>.
- [MMS19] Paul E. McKenney, Maged Michael, and Peter Sewell. N2369: Pointer lifetime-end zap, April 2019. <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2369.pdf>.
- [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, Stevenson, Washington, USA, October 2007. ACM SIGOPS.
- [Mol05] Ingo Molnar. Index of /pub/linux/kernel/projects/rt, February 2005. URL: <https://www.kernel.org/pub/linux/kernel/projects/rt/>.
- [Mol06] Ingo Molnar. Lightweight robust futexes, March 2006. Available: <https://www.kernel.org/doc/Documentation/robust-futexes.txt> [Viewed February 14, 2021].
- [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.
- [Mor07] Richard Morris. Sir Tony Hoare: Geek of the week, August 2007. <https://www.red-gate.com/simple-talk/opinion/geek-of-the-week/sir-tony-hoare-geek-of-the-week/>.
- [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.
- [MP15a] Paul E. McKenney and Aravinda Prasad. Recent read-mostly research in 2015, December 2015. <https://lwn.net/Articles/667593/>.
- [MP15b] Paul E. McKenney and Aravinda Prasad. Some more details on read-log-update, December 2015. <https://lwn.net/Articles/667720/>.
- [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, July 2006. <https://en.wikipedia.org/wiki/Read-copy-update>.
- [MPI08] MPI Forum. Message passing interface forum, September 2008. Available: <http://www.mpi-forum.org/> [Viewed September 9, 2008].
- [MR08] Paul E. McKenney and Steven Rostedt. Integrating and validating dynticks and preemptable RCU, April 2008. Available: <https://lwn.net/Articles/279077/> [Viewed April 24, 2008].
- [MRP⁺17] Paul E. McKenney, Torvald Riegel, Jeff Preshing, Hans Boehm, Clark Nelson, Olivier Giroux, Lawrence Crowl, JF Bastian, and Michael Wong. Marking memory order consume dependency chains, February 2017. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0462r1.pdf>.
- [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, December 1995. Technical Report TR599.
- [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.

- [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.
- [MS01] Paul E. McKenney and Dipankar Sarma. Read-copy update mutual exclusion in Linux, February 2001. Available: http://lse.sourceforge.net/locking/rcu/rcupdate_doc.html [Viewed October 18, 2004].
- [MS08] MySQL AB and Sun Microsystems. MySQL Downloads, November 2008. Available: <http://dev.mysql.com/downloads/> [Viewed November 26, 2008].
- [MS09] Paul E. McKenney and Raul Silvera. Example POWER implementation for C/C++ memory model, February 2009. Available: <http://www.rdrop.com/users/paulmck/scalability/paper/N2745r.2009.02.27a.html> [Viewed: April 5, 2009].
- [MS12] Alexander Matveev and Nir Shavit. Towards a fully pessimistic STM model. In *TRANSACT 2012*, San Jose, CA, USA, February 2012. ACM SIGPLAN.
- [MS14] Paul E. McKenney and Alan Stern. Axiomatic validation of memory barriers and atomic instructions, August 2014. <https://lwn.net/Articles/608550/>.
- [MS18] Luc Maranget and Alan Stern. lock.cat, May 2018. <https://github.com/torvalds/linux/blob/master/tools/memory-model/lock.cat>.
- [MSA⁺02] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read-copy update. In *Ottawa Linux Symposium*, pages 338–367, June 2002. Available: <https://www.kernel.org/doc/ols/2002/ols2002-pages-338-367.pdf> [Viewed February 14, 2021].
- [MSFM15] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: A lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP ’15, pages 168–183, Monterey, California, 2015. ACM.
- [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, October 2012. <https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf>.
- [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: https://iacoma.cs.uiuc.edu/iacoma-papers/wmpi_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.
- [Mud01] Trevor Mudge. POWER: A first-class architectural design constraint. *IEEE Computer*, 34(4):52–58, April 2001.
- [Mus04] Museum Victoria Australia. CSIRAC: Australia’s first computer, 2004. URL: <http://museumvictoria.com.au/csirac/>.
- [MW05] Paul E. McKenney and Jonathan Walpole. RCU semantics: A first attempt, January 2005. Available: <http://www.rdrop.com/users/paulmck/RCU/rcu-semantics.2005.01.30a.pdf> [Viewed December 6, 2009].
- [MW07] Paul E. McKenney and Jonathan Walpole. What is RCU, fundamentally?, December 2007. Available: <https://1wn.net/Articles/262464/> [Viewed December 27, 2007].
- [MWB⁺17] Paul E. McKenney, Michael Wong, Hans Boehm, Jens Maurer, Jeffrey Yasskin, and JF Bastien. P0190R4: Proposal for new `memory_order_consume` definition, July 2017. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0190r4.pdf>.
- [MWPF18] Paul E. McKenney, Ulrich Weigand, Andrea Parri, and Boqun Feng. Linux-kernel memory model, September 2018. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0124r6.html>.
- [Mye79] Glenford J. Myers. *The Art of Software Testing*. Wiley, 1979.
- [NA18] Catherine E. Nemitz and James H. Anderson. Work-in-progress: Lock-based software transactional memory for real-time systems. In *2018 IEEE Real-Time Systems Symposium*, RTSS’18, pages 147–150, Nashville, TN, USA, 2018. IEEE.
- [Nag18] Honnappa Nagarahalli. rcu: add RCU library supporting QSBR mechanism, May 2018. https://git.dpdk.org/dpdk/tree/lib/librte_rcu.
- [Nes06a] Oleg Nesterov. Re: [patch] cpufreq: mark `cpufreq_tsc()` as `core_initcall_sync`, November 2006. Available: <https://lkml.org/lkml/2006/11/19/69> [Viewed May 28, 2007].
- [Nes06b] Oleg Nesterov. Re: [rfc, patch 1/2] qrcu: "quick" srcu implementation, November 2006. Available: <https://lkml.org/lkml/2006/11/29/330> [Viewed November 26, 2008].
- [NSHW20] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2020.
- [NVi17a] NVidia. Accelerated computing — training, January 2017. <https://developer.nvidia.com/accelerated-computing-training>.
- [NVi17b] NVidia. Existing university courses, January 2017. <https://developer.nvidia.com/educators/existing-courses>.
- [O’H19] Peter W. O’Hearn. Incorrectness logic. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

- [OHOC20] Robert O’Callahan, Kyle Huey, Devon O’Dell, and Terry Coatta. To catch a failure: The record-and-replay approach to debugging: A discussion with robert o’callahan, kyle huey, devon o’dell, and terry coatta. *Queue*, 18(1):61–79, February 2020.
- [ON07] Robert Olsson and Stefan Nilsson. TRASH: A dynamic LC-trie and hash data structure. In *Workshop on High Performance Switching and Routing (HPSR’07)*, May 2007.
- [ONH⁺96] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *ASPLOS VII*, Cambridge, MA, USA, October 1996.
- [Ope97] Open Group. The single UNIX specification, version 2: Threads, 1997. Available: <http://www.opengroup.org/onlinepubs/007908799/xsh/threads.html> [Viewed September 19, 2008].
- [PAB⁺95] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *15th ACM Symposium on Operating Systems Principles (SOSP’95)*, pages 314–321, Copper Mountain, CO, December 1995.
- [Pat10] David Patterson. The trouble with multicore. *IEEE Spectrum*, 2010:28–32, 52–53, July 2010.
- [PAT11] V Pankratius and A R Adl-Tabatabai. A study of transactional memory vs. locks in practice. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures (2011)*, SPAA ’11, pages 43–52, San Jose, CA, USA, 2011. ACM.
- [PBCE20] Elizabeth Patitsas, Jesse Berlin, Michelle Craig, and Steve Easterbrook. Evidence that computer science grades are not bimodal. *Commun. ACM*, 63(1):91–98, January 2020.
- [PD11] Martin Pohlack and Stephan Diestelhorst. From lightweight hardware transactional memory to lightweight lock elision. In *TRANSACT 2011*, San Jose, CA, USA, June 2011. ACM SIGPLAN.
- [Pen18] Roman Penyaev. [PATCH v2 01/26] introduce list_next_or_null_rr_rcu(), May 2018. <https://lkml.kernel.org/r/20180518130413.16997-2-roman.penzaev@profitbricks.com>.
- [Pet06] Jeremy Peters. From reuters, automatic trading linked to news events, December 2006. URL: <http://www.nytimes.com/2006/12/11/technology/11reuters.html?ei=5088&en=e5e9416415a9eeb2&ex=1323493200>.
...
- [Pig06] Nick Piggin. [patch 3/3] radix-tree: RCU lockless readside, June 2006. Available: <https://lkml.org/lkml/2006/6/20/238> [Viewed March 25, 2008].
- [Pik17] Fedor G. Pikus. Read, copy, update... Then what?, September 2017. <https://www.youtube.com/watch?v=rxQ5K9lo034>.
- [PMDY20] SeongJae Park, Paul E. McKenney, Laurent Dufour, and Heon Y. Yeom. An htm-based update-side synchronization for rcu on numa systems. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys ’20, Heraklion, Greece, 2020. Association for Computing Machinery.

- [Pod10] Andrej Podzimek. Read-copy-update for opensolaris. Master's thesis, Charles University in Prague, 2010.
- [Pok16] Michael Pokorny. The deadlock empire, February 2016. <https://deadlockempire.github.io/>.
- [Pos08] PostgreSQL Global Development Group. PostgreSQL, November 2008. Available: <https://www.postgresql.org/> [Viewed November 26, 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.
- [Pug00] William Pugh. Reordering on an Alpha processor, 2000. Available: <https://www.cs.umd.edu/~pugh/java/memoryModel/AlphaReordering.html> [Viewed: June 23, 2004].
- [Pul00] Geoffrey K. Pullum. How Dr. Seuss would prove the halting problem undecidable. *Mathematics Magazine*, 73(4):319–320, 2000. <http://www.1el.ed.ac.uk/~gpullum/loopsnoop.html>.
- [PW07] Donald E. Porter and Emmett Witchel. Lessons from large transactional systems, December 2007. Personal communication <20071214220521.GA5721@olive-green.cs.utexas.edu>.
- [Ras14] Mindaugas Rasiukevicius. NPF—progress and perspective. In *AsiaBSDCon*, Tokyo, Japan, March 2014.
- [Ras16] Mindaugas Rasiukevicius. Quiescent-state and epoch based reclamation, July 2016. <https://github.com/rmind/libqsbr>.
- [Ray99] Eric S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly, 1999.
- [RC15] Pedro Ramalhete and Andreia Correia. Poor man's URCU, August 2015. <https://github.com/pramalhe/ConcurrencyFreaks/blob/master/papers/poormanurcu-2015.pdf>.
- [RD12] Ravi Rajwar and Martin Dixon. Intel transactional synchronization extensions, September 2012. Intel Developer Forum (IDF) 2012 ARCS004.
- [Reg10] John Regehr. A guide to undefined behavior in C and C++, part 1, July 2010. <https://blog.regehr.org/archives/213>.
- [Rei07] James Reinders. *Intel Threading Building Blocks*. O'Reilly, Sebastopol, CA, USA, 2007.
- [RG01] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 294–305, Austin, TX, December 2001. The Institute of Electrical and Electronics Engineers, Inc.
- [RG02] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, Austin, TX, October 2002.

- [RH02] Zoran Radović and Erik Hagersten. Efficient synchronization for nonuniform communication architectures. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 1–13, Baltimore, Maryland, USA, November 2002. The Institute of Electrical and Electronics Engineers, Inc.
- [RH03] Zoran Radović and Erik Hagersten. Hierarchical backoff locks for nonuniform communication architectures. In *Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, pages 241–252, Anaheim, California, USA, February 2003.
- [RH18] Geoff Romer and Andrew Hunter. An RAII interface for deferred reclamation, March 2018. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0561r4.html>.
- [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*, Stevenson, WA, USA, October 2007. ACM SIGOPS. Available: <http://www.sosp2007.org/papers/sosp056-rossbach.pdf> [Viewed October 21, 2007].
- [Rin13] Martin Rinard. Parallel synchronization-free approximate data structure construction. In *Proceedings of the 5th USENIX Conference on Hot Topics in Parallelism*, HotPar’13, page 6, San Jose, CA, 2013. USENIX Association.
- [RKM⁺10] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. Speculative parallelization using software multi-threaded transactions. *SIGARCH Comput. Archit. News*, 38(1):65–76, 2010.
- [RLPB18] Yuxin Ren, Guyue Liu, Gabriel Parmer, and Björn Brandenburg. Scalable memory reclamation for multi-core, real-time systems. In *Proceedings of the 2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, page 12, Porto, Portugal, April 2018. IEEE.
- [RMF19] Federico Reghennzani, Giuseppe Massari, and William Fornaciari. The real-time Linux kernel: A survey on PREEMPT_RT. *ACM Comput. Surv.*, 52(1):18:1–18:36, February 2019.
- [Ros06] Steven Rostedt. Lightweight PI-futexes, June 2006. Available: <https://www.kernel.org/doc/html/latest/locking/pi-futex.html> [Viewed February 14, 2021].
- [Ros10a] Steven Rostedt. tracing: Harry Potter and the Deathly Macros, December 2010. Available: <https://lwn.net/Articles/418710/> [Viewed: August 28, 2011].
- [Ros10b] Steven Rostedt. Using the TRACE_EVENT() macro (part 1), March 2010. Available: <https://lwn.net/Articles/379903/> [Viewed: August 28, 2011].
- [Ros10c] Steven Rostedt. Using the TRACE_EVENT() macro (part 2), March 2010. Available: <https://lwn.net/Articles/381064/> [Viewed: August 28, 2011].
- [Ros10d] Steven Rostedt. Using the TRACE_EVENT() macro (part 3), April 2010. Available: <https://lwn.net/Articles/383362/> [Viewed: August 28, 2011].

- [Ros11] Steven Rostedt. lockdep: How to read its cryptic output, September 2011. <http://www.linuxplumbersconf.org/2011/ocw/sessions/153>.
- [Roy17] Lance Roy. rcutorture: Add CBMC-based formal verification for SRCU, January 2017. URL: <https://www.spinics.net/lists/kernel/msg2421833.html>.
- [RR20] Sergio Rajsbaum and Michel Raynal. Mastering concurrent computing through sequential thinking. *Commun. ACM*, 63(1):78–87, January 2020.
- [RSB⁺97] Rajeev Rastogi, S. Seshadri, Philip Bohannon, Dennis W. Leinbaugh, Abraham Silberschatz, and S. Sudarshan. Logical and physical versioning in main memory databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases, VLDB '97*, pages 86–95, San Francisco, CA, USA, August 1997. Morgan Kaufmann Publishers Inc.
- [RTY⁺87] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, Palo Alto, CA, October 1987. Association for Computing Machinery.
- [Rus00a] Rusty Russell. Re: modular net drivers, June 2000. URL: <http://oss.sgi.com/projects/netdev/archive/2000-06/msg00250.html> [broken, February 15, 2021].
- [Rus00b] Rusty Russell. Re: modular net drivers, June 2000. URL: <http://oss.sgi.com/projects/netdev/archive/2000-06/msg00254.html> [broken, February 15, 2021].
- [Rus03] Rusty Russell. Hanging out with smart people: or... things I learned being a kernel monkey, July 2003. 2003 Ottawa Linux Symposium Keynote <https://ozlabs.org/~rusty/ols-2003-keynote/ols-keynote-2003.html>.
- [Rut17] Mark Rutland. compiler.h: Remove ACCESS_ONCE(), November 2017. Git commit: <https://git.kernel.org/linus/b899a850431e>.
- [SAE⁺18] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Commun. ACM*, 61(4):58–66, March 2018.
- [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 Rosengrub, and Jimi Xenidis. System support for online reconfiguration. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 141–154, San Antonio, Texas, USA, June 2003. USENIX Association.
- [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, Calgary, AB, Canada, 2009. ACM.
- [SBN⁺20] Dimitrios Siakavaras, Panagiotis Billis, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Efficient concurrent range queries in b+-trees using rcu-hm. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '20*, page 571–573, Virtual Event, USA, 2020. Association for Computing Machinery.

- [SBV10] Martin Schoeberl, Florian Brandner, and Jan Vitek. RTTM: Real-time transactional memory. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 326–333, 01 2010.
- [Sch35] E. Schrödinger. Die gegenwärtige Situation in der Quantenmechanik. *Naturwissenschaften*, 23:807–812; 823–828; 844–849, November 1935.
- [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.
- [Sco15] Michael Scott. *Programming Language Pragmatics, 4th Edition*. Morgan Kaufmann, Burlington, MA, USA, 2015.
- [Seq88] Sequent Computer Systems, Inc. *Guide to Parallel Programming*, 1988.
- [Sew] Peter Sewell. Relaxed-memory concurrency. Available: <https://www.cl.cam.ac.uk/~pes20/weakmemory/> [Viewed: February 15, 2021].
- [Sey12] Justin Seyster. *Runtime Verification of Kernel-Level Concurrency Using Compiler-Based Instrumentation*. PhD thesis, Stony Brook University, 2012.
- [SF95] Janice M. Stone and Robert P. Fitzgerald. Storage in the PowerPC. *IEEE Micro*, 15(2):50–58, April 1995.
- [Sha11] Nir Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, March 2011.
- [She06] Gautham R. Shenoy. [patch 4/5] lock_cpu_hotplug: Redesign - lightweight implementation of lock_cpu_hotplug, October 2006. Available: <https://lkml.org/lkml/2006/10/26/73> [Viewed January 26, 2009].
- [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.
- [Slo10] Lubos Slovak. First steps for utilizing userspace RCU library, July 2010. <https://gitlab.labs.nic.cz/knot/knot-dns/commit/f67acc0178ee9a781d7a63fb041b5d09eb5fb4a2>.
- [SM95] John D. Slingwine and Paul E. McKenney. Apparatus and method for achieving reduced overhead mutual exclusion and maintaining coherency in a multiprocessor system utilizing execution history and thread monitoring. Technical Report US Patent 5,442,758, Assigned to International Business Machines Corp, Washington, DC, August 1995.
- [SM97] John D. Slingwine and Paul E. McKenney. Method for maintaining data coherency using thread activity summaries in a multiprocessor system. Technical Report US Patent 5,608,893, Assigned to International Business Machines Corp, Washington, DC, March 1997.
- [SM98] John D. Slingwine and Paul E. McKenney. Apparatus and method for achieving reduced overhead mutual exclusion and maintaining coherency in a multiprocessor system utilizing execution history and thread monitoring. Technical Report US Patent 5,727,209, Assigned to International Business Machines Corp, Washington, DC, March 1998.

- [SM04] Dipankar Sarma and Paul E. McKenney. Issues with selected scalability features of the 2.6 kernel. In *Ottawa Linux Symposium*, page 16, July 2004. <https://www.kernel.org/doc/ols/2004/ols2004v2-pages-195-208.pdf>.
- [SM13] Thomas Sewell and Toby Murray. Above and beyond: seL4 noninterference and binary verification, May 2013. <https://cips-vo.org/node/7706>.
- [Smi19] Richard Smith. Working draft, standard for programming language C++, January 2019. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/n4800.pdf>.
- [SMS08] Michael Spear, Maged Michael, and Michael Scott. Inevitability mechanisms for software transactional memory. In *3rd ACM SIGPLAN Workshop on Transactional Computing*, Salt Lake City, Utah, February 2008. ACM. Available: http://www.cs.rochester.edu/u/scott/papers/2008_TRANSACT_inevitability.pdf [Viewed January 10, 2009].
- [SNGK17] Dimitrios Siakavaras, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Combining HTM and RCU to implement highly efficient balanced binary search trees. In *12th ACM SIGPLAN Workshop on Transactional Computing*, Austin, TX, USA, February 2017.
- [SPA94] SPARC International. *The SPARC Architecture Manual*, 1994. Available: <https://sparc.org/wp-content/uploads/2014/01/SPARCV9.pdf.gz>.
- [Spi77] Keith R. Spitz. Tell which is which and you'll be rich, 1977. Inscription on wall of dungeon.
- [Spr01] Manfred Spraul. Re: RFC: patch to allow lock-free traversal of lists with insertion, October 2001. URL: <http://lkml.iu.edu/hypermail/linux/kernel/0110.1/0410.html>.
- [Spr08] Manfred Spraul. [RFC, PATCH] state machine based rcu, August 2008. Available: <https://lkml.org/lkml/2008/8/21/336> [Viewed December 8, 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.
- [SRK⁺11] Justin Seyster, Prabakar Radhakrishnan, Samriti Katoh, Abhinav Duggal, Scott D. Stoller, and Erez Zadok. Redflag: a framework for analysis of kernel-level concurrency. In *Proceedings of the 11th international conference on Algorithms and architectures for parallel processing - Volume Part I*, ICA3PP'11, pages 66–79, Melbourne, Australia, 2011. Springer-Verlag.
- [SRL90] 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, Monte Verita, Ascona, Switzerland, 1994.
- [SS06] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, May 2006.

- [SSA⁺11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. POWER and ARM litmus tests, 2011. <https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test6.pdf>.
- [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.
- [Ste13] W. Richard Stevens. *Advanced Programming in the UNIX Environment, 3rd Edition*. Addison Wesley, 2013.
- [Sut08] Herb Sutter. Effective concurrency, 2008. Series in Dr. Dobbs Journal.
- [Sut13] Adrian Sutton. Concurrent programming with the Disruptor, January 2013. Presentation at Linux.conf.au 2013, URL: <https://www.youtube.com/watch?v=ItptVmRHyl>.
- [SW95] Richard L. Sites and Richard T. Witek. *Alpha AXP Architecture*. Digital Press, second edition, 1995.
- [SWS16] Harshal Sheth, Aashish Welling, and Nihar Sheth. Read-copy update in a garbage collected environment, 2016. MIT PRIMES program: <https://math.mit.edu/research/highschool/primes/materials/2016/conf/10-1%20Sheth-Welling-Sheth.pdf>.
- [SZJ12] KC Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. Eliminating read barriers through procrastination and cleanliness. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM ’12, pages 49–60, Beijing, China, 2012. ACM.
- [Tal07] Nassim Nicholas Taleb. *The Black Swan*. Random House, 2007.
- [TDV15] Joseph Tassarotti, Derek Dreyer, and Victor Vafeiadis. Verifying read-copy-update in a logic for weak memory. In *Proceedings of the 2015 Proceedings of the 36th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI ’15, pages 110–120, New York, NY, USA, June 2015. ACM.
- [The08] The Open MPI Project. Open MPI, November 2008. Available: <http://www.open-mpi.org/software/> [Viewed November 26, 2008].

- [The11] The Valgrind Developers. Valgrind, November 2011. <http://www.valgrind.org/>.
- [The12a] The NetBSD Foundation. pserialize(9), October 2012. <http://netbsd.gw.com/cgi-bin/man-cgi?pserialize+9+NetBSD-current>.
- [The12b] The OProfile Developers. Oprofile, April 2012. <http://oprofile.sourceforge.net>.
- [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, October 2001. URL: <https://lkml.org/lkml/2001/10/13/105>, <https://lkml.org/lkml/2001/10/13/82>.
- [Tor02] Linus Torvalds. Linux 2.5.43, October 2002. Available: <https://lkml.org/lkml/2002/10/15/425> [Viewed March 30, 2008].
- [Tor03] Linus Torvalds. Linux 2.6, August 2003. Available: <https://kernel.org/pub/linux/kernel/v2.6> [Viewed February 16, 2021].
- [Tor08] Linus Torvalds. Move ACCESS_ONCE() to <linux/compiler.h>, May 2008. Git commit: <https://git.kernel.org/linus/9c3cdc1f83a6>.
- [Tor19] Linus Torvalds. rcu: locking and unlocking need to always be at least barriers, June 2019. Git commit: <https://git.kernel.org/linus/66be4e66a7f4>.
- [Tra01] Transaction Processing Performance Council. TPC, 2001. Available: <http://www.tpc.org/> [Viewed December 7, 2008].
- [Tre86] R. K. Treiber. Systems programming: Coping with parallelism, April 1986. RJ 5118.
- [Tri12] Josh Triplett. *Relativistic Causal Ordering: A Memory Model for Scalable Concurrent Data Structures*. PhD thesis, Portland State University, 2012.
- [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 160–167, Tokyo, Japan, 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.
- [TZK⁺13] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, Farmington, Pennsylvania, 2013. ACM.
- [Ung11] David Ungar. Everything you know (about parallel programming) is wrong!: A wild screed about the future. In *Dynamic Languages Symposium 2011*, Portland, OR, USA, October 2011. Invited talk presentation.

- [Uni08a] University of California, Berkeley. BOINC: compute for science, October 2008. Available: <http://boinc.berkeley.edu/> [Viewed January 31, 2008].
- [Uni08b] University of California, Berkeley. SETI@HOME, December 2008. Available: <http://setiathome.berkeley.edu/> [Viewed January 31, 2008].
- [Uni10] University of Maryland. Parallel maze solving, November 2010. URL: <http://www.cs.umd.edu/class/fall2010/cmsc433/p3/> [broken, February 2021].
- [Val95] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 214–222, Ottawa, Ontario, Canada, 1995. ACM.
- [Van18] Michal Vaner. ArcSwap, April 2018. <https://crates.io/crates/arc-swap>.
- [VBC⁺15] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the c11 memory model and what we can do about it. *SIGPLAN Not.*, 50(1):209–220, January 2015.
- [VGS08] Haris Volos, Neelam Goyal, and Michael M. Swift. Pathological interaction of locks with transactional memory. In *3rd ACM SIGPLAN Workshop on Transactional Computing*, Salt Lake City, Utah, 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.
- [Š11] Jaroslav Ševčík. Safe optimisations for shared-memory concurrent programs. *SIGPLAN Not.*, 46(6):306–316, June 2011.
- [Wav16] Wave Computing, Inc. *MIPS®Architecture For Programmers Volume II-A: The MIPS64®Instruction Set Reference Manual*, 2016. URL: <https://www.mips.com/downloads/the-mips64-instruction-set-v6-06/>.
- [Wei63] J. Weizenbaum. Symmetric list processor. *Commun. ACM*, 6(9):524–536, September 1963.
- [Wei12] Frédéric Weisbecker. Interruption timer périodique, 2012. http://www.dailymotion.com/video/xtxtew_interruption-timer-periodique-frederic-weisbecker-kernel-recipes-12_tech.
- [Wei13] Stewart Weiss. Unix lecture notes, May 2013. Available: http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/unix_lecture_notes/ [Viewed April 8, 2014].
- [Wik08] Wikipedia. Zilog Z80, 2008. Available: <https://en.wikipedia.org/wiki/Z80> [Viewed: December 7, 2008].
- [Wik12] Wikipedia. Labyrinth, January 2012. <https://en.wikipedia.org/wiki/Labyrinth>.
- [Wil12] Anthony Williams. *C++ Concurrency in Action: Practical Multithreading*. Manning, Shelter Island, NY, USA, 2012.
- [Wil19] Anthony Williams. *C++ Concurrency in Action, 2nd Edition*. Manning, Shelter Island, NY, USA, 2019.

- [WKS94] Robert W. Wisniewski, Leonidas Kontothanassis, 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.
- [Won19] William G. Wong. VHS or Betamax ... CCIX or CXL ... so many choices, March 2019. <https://www.electronicdesign.com/industrial-automation/article/21807721/vhs-or-betamaxccix-or-cxlso-many-choices>.
- [WTS96] Cai-Dong Wang, Hiroaki Takada, and Ken Sakamura. Priority inheritance spin locks for multiprocessor real-time systems. In *Proceedings of the 2nd International Symposium on Parallel Architectures, Algorithms, and Networks*, ISPAN '96, pages 70–76, Beijing, China, 1996. IEEE Computer Society.
- [xen14] xenomai.org. Xenomai, December 2014. URL: <http://xenomai.org/>.
- [Xu10] Herbert Xu. bridge: Add core IGMP snooping support, February 2010. Available: <https://marc.info/?t=126719855400006&r=1&w=2> [Viewed March 20, 2011].
- [YHLR13] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of Intel® Transactional Synchronization Extensions for high-performance computing. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, Denver, Colorado, 2013. ACM.
- [Yod04a] Victor Yodaiken. Against priority inheritance, September 2004. Available: <https://www.yodaiken.com/papers/inherit.pdf> [Viewed May 26, 2007].
- [Yod04b] Victor Yodaiken. Temporal inventory and real-time synchronization in RTLinuxPro, September 2004. URL: <https://www.yodaiken.com/papers/sync.pdf>.
- [Zel11] Cyril Zeller. CUDA C/C++ basics: Supercomputing 2011 tutorial, November 2011. <https://www.nvidia.com/docs/I0/116711/sc11-cuda-c-basics.pdf>.
- [Zha89] Lixia Zhang. *A New Architecture for Packet Switching Network Protocols*. PhD thesis, Massachusetts Institute of Technology, July 1989.
- [Zij14] Peter Zijlstra. Another go at speculative page faults, October 2014. <https://lkml.org/lkml/2014/10/20/620>.

Credits

L^AT_EX Advisor

Akira Yokosawa is this book's L^AT_EX advisor, which perhaps most notably includes the care and feeding of the style guide laid out in Appendix D. This work includes table layout, listings, fonts, rendering of math, acronyms, bibliography formatting, epigraphs, hyperlinks, paper size. Akira also perfected the cross-referencing of quick quizzes, allowing easy and exact navigation between quick quizzes and their answers. He also added build options that permit quick quizzes to be hidden and to be gathered at the end of each chapter, textbook style.

This role also includes the build system, which Akira has optimized and made much more user-friendly. His enhancements have included automating response to bibliography changes, automatically determining which source files are present, and automatically generating listings (with automatically generated hyperlinked line-number references) from the source files.

Reviewers

- Alan Stern (Chapter 15).
- Andy Whitcroft (Section 9.5.2, Section 9.5.3).
- Artem Bityutskiy (Chapter 15, Appendix C).
- Dave Keck (Appendix C).
- David S. Horner (Section 12.1.5).
- Gautham Shenoy (Section 9.5.2, Section 9.5.3).
- “jarkao2”, AKA LWN guest #41960 (Section 9.5.3).
- Jonathan Walpole (Section 9.5.3).
- Josh Triplett (Chapter 12).
- Michael Factor (Section 17.2).
- Mike Fulton (Section 9.5.2).

If I have seen further it is by standing on the shoulders of giants.

Isaac Newton, modernized

- Peter Zijlstra (Section 9.5.4).
- Richard Woodruff (Appendix C).
- 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.

Machine Owners

Readers might have noticed some graphs showing scalability data out to several hundred CPUs, courtesy of my current employer, with special thanks to Paul Saab, Yashar Bayani, Joe Boyd, and Kyle McMartin.

From back in my time at IBM, 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 Breedts.

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 4.3.4.1 (“Shared-Variable Shenanigans”) on page 40 originally appeared in Linux Weekly News [ADF⁺19].

3. Section 6.5 (“Retrofitted Parallelism Considered Grossly Sub-Optimal”) on page 92 originally appeared in 4th USENIX Workshop on Hot Topics on Parallelism [McK12c].
4. Section 9.5.2 (“RCU Fundamentals”) on page 141 originally appeared in Linux Weekly News [MW07].
5. Section 9.5.3 (“RCU Linux-Kernel API”) on page 147 originally appeared in Linux Weekly News [McK08d].
6. Section 9.5.4 (“RCU Usage”) on page 156 originally appeared in Linux Weekly News [McK08f].
7. Section 9.5.5 (“RCU Related Work”) on page 167 originally appeared in Linux Weekly News [McK14e].
8. Section 9.5.5 (“RCU Related Work”) on page 167 originally appeared in Linux Weekly News [MP15a].
9. Chapter 12 (“Formal Verification”) on page 219 originally appeared in Linux Weekly News [McK07f, MR08, McK11d].
10. Section 12.3 (“Axiomatic Approaches”) on page 250 originally appeared in Linux Weekly News [MS14].
11. Section 13.5.4 (“Correlated Fields”) on page 268 originally appeared in Oregon Graduate Institute [McK04].
12. Chapter 15 (“Advanced Synchronization: Memory Ordering”) on page 297 originally appeared in the Linux kernel [HMDZ06].
13. Chapter 15 (“Advanced Synchronization: Memory Ordering”) on page 297 originally appeared in Linux Weekly News [AMM⁺17a, AMM⁺17b].
14. Chapter 15 (“Advanced Synchronization: Memory Ordering”) on page 297 originally appeared in ASPLOS ’18 [AMM⁺18].
15. Section 15.3.2 (“Address- and Data-Dependency Difficulties”) on page 319 originally appeared in the Linux kernel [McK14c].
16. Section 15.5 (“Memory-Barrier Instructions For Specific CPUs”) on page 331 originally appeared in Linux Journal [McK05a, McK05b].

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 3.5 (p 19) by Melissa Broussard.
5. Figure 3.6 (p 20) by Melissa Broussard.
6. Figure 3.7 (p 20) by Melissa Broussard.
7. Figure 3.8 (p 21) by Melissa Broussard.
8. Figure 3.9 (p 21) by Melissa Broussard.
9. Figure 3.11 (p 25) by Melissa Broussard.
10. Figure 5.3 (p 51) by Melissa Broussard.
11. Figure 6.1 (p 73) by Korniliос Kourtis.
12. Figure 6.2 (p 75) by Melissa Broussard.
13. Figure 6.3 (p 75) by Korniliос Kourtis.
14. Figure 6.4 (p 75) by Korniliос Kourtis.
15. Figure 6.13 (p 84) by Melissa Broussard.
16. Figure 6.14 (p 85) by Melissa Broussard.
17. Figure 6.15 (p 85) by Melissa Broussard.
18. Figure 7.1 (p 99) by Melissa Broussard.
19. Figure 7.2 (p 100) by Melissa Broussard.
20. Figure 10.13 (p 183) by Melissa Broussard.
21. Figure 10.14 (p 184) by Melissa Broussard.
22. Figure 11.1 (p 199) by Melissa Broussard.
23. Figure 11.2 (p 200) by Melissa Broussard.
24. Figure 11.3 (p 206) by Melissa Broussard.
25. Figure 11.6 (p 217) by Melissa Broussard.
26. Figure 14.1 (p 277) by Melissa Broussard.
27. Figure 14.2 (p 277) by Melissa Broussard.
28. Figure 14.3 (p 278) by Melissa Broussard.
29. Figure 14.10 (p 286) by Melissa Broussard.

30. Figure 14.11 (p 286) by Melissa Broussard.
31. Figure 14.14 (p 288) by Melissa Broussard.
32. Figure 14.15 (p 295) by Sarah McKenney.
33. Figure 14.15 (p 295) by Sarah McKenney.
34. Figure 14.16 (p 295) by Sarah McKenney.
35. Figure 14.16 (p 295) by Sarah McKenney.
36. Figure 15.2 (p 299) by Melissa Broussard.
37. Figure 15.5 (p 305) by Akira Yokosawa.
38. Figure 15.18 (p 335) by Melissa Brossard.
39. Figure 16.2 (p 343) by Melissa Broussard.
40. Figure 17.1 (p 345) by Melissa Broussard.
41. Figure 17.2 (p 346) by Melissa Broussard.
42. Figure 17.3 (p 346) by Melissa Broussard.
43. Figure 17.4 (p 347) by Melissa Broussard.
44. Figure 17.5 (p 347) by Melissa Broussard, remixed.
45. Figure 17.9 (p 359) by Melissa Broussard.
46. Figure 17.10 (p 359) by Melissa Broussard.
47. Figure 17.11 (p 360) by Melissa Broussard.
48. Figure 17.12 (p 360) by Melissa Broussard.
49. Figure 18.1 (p 381) by Melissa Broussard.
50. Figure 18.1 (p 381) by Melissa Broussard.
51. Figure A.2 (p 386) by Melissa Broussard.
52. Figure E.2 (p 460) by Korniliос Kourtis.

Figure 9.30 was adapted from Fedor Pikus's "When to use RCU" slide [Pik17]. The discussion of mechanical reference counters in Section 9.2 stemmed from a private conversation with Dave Regan.

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!

The bibtex-generation service of the Association for Computing Machinery has saved us a huge amount of time and effort compiling the bibliography, for which we are grateful. Thanks are also due to Stamatis Karnouskos, who convinced me to drag my antique bibliography database kicking and screaming into the 21st century. Any technical work of this sort owes thanks to the many individuals and organizations that keep Internet and the World Wide Web up and running, and this one is no exception.

Portions of this material are based upon work supported by the National Science Foundation under Grant No. CNS-0719851.