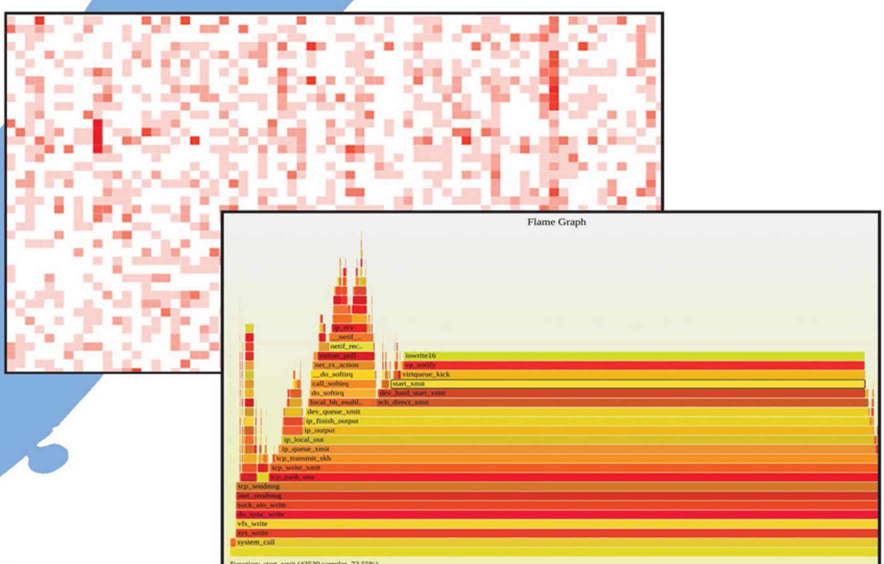
Systems Performance

Enterprise and the Cloud

Second Edition

Brendan Gregg



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Systems Performance

Second Edition

This page intentionally left blank

Systems Performance

Enterprise and the Cloud

Second Edition

Brendan Gregg

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2020944455

Copyright © 2021 Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions.

Cover images by Brendan Gregg
Page 9, Figure 1.5: Screenshot of System metrics GUI (Grafana) © 2020 Grafana Labs
Page 84, Figure 2.32: Screenshot of Firefox timeline chart © Netflix
Page 164, Figure 4.7: Screenshot of sar(1) sadf(1) SVG output © 2010 W3C
Page 560, Figure 10.12: Screenshot of Wireshark screenshot © Wireshark
Page 740, Figure 14.3: Screenshot of KernelShark © KernelShark

ISBN-13: 978-0-13-682015-4
ISBN-10: 0-13-682015-8

ScoutAutomatedPrintCode

Publisher
Mark L. Taub

Executive Editor
Greg Doench

Managing Producer
Sandra Schroeder

Sr. Content Producer
Julie B. Nahil

Project Manager
Rachel Paul

Copy Editor
Kim Wimpsett

Indexer
Ted Laux

Proofreader
Rachel Paul

Compositor
The CIP Group

For Deirdré Straughan,
an amazing person in technology,
and an amazing person—we did it.

This page intentionally left blank

Contents at a Glance

Contents ix
Preface xxix
Acknowledgments xxxv
About the Author xxxvii

1 **Introduction 1**

2 **Methodologies 21**

3 **Operating Systems 89**

4 **Observability Tools 129**

5 **Applications 171**

6 **CPUs 219**

7 **Memory 303**

8 **File Systems 359**

9 **Disks 423**

10 **Network 499**

11 **Cloud Computing 579**

12 **Benchmarking 641**

13 **perf 671**

14 **Ftrace 705**

15 **BPF 751**

16 **Case Study 783**

A **USE Method: Linux 795**

B **sar Summary 801**

C **bpftrace One-Liners 803**

D **Solutions to Selected Exercises 809**

E **Systems Performance Who's Who 811**

Glossary 815

Index 825

This page intentionally left blank

Contents

Preface xxix
Acknowledgments xxxv
About the Author xxxvii

1 Introduction 1
1.1 Systems Performance 1
1.2 Roles 2
1.3 Activities 3
1.4 Perspectives 4
1.5 Performance Is Challenging 5
1.5.1 Subjectivity 5
1.5.2 Complexity 5
1.5.3 Multiple Causes 6
1.5.4 Multiple Performance Issues 6
1.6 Latency 6
1.7 Observability 7
1.7.1 Counters, Statistics, and Metrics 8
1.7.2 Profiling 10
1.7.3 Tracing 11
1.8 Experimentation 13
1.9 Cloud Computing 14
1.10 Methodologies 15
1.10.1 Linux Perf Analysis in 60 Seconds 15
1.11 Case Studies 16
1.11.1 Slow Disks 16
1.11.2 Software Change 18
1.11.3 More Reading 19
1.12 References 19

2 Methodologies 21
2.1 Terminology 22
2.2 Models 23
2.2.1 System Under Test 23
2.2.2 Queueing System 23
2.3 Concepts 24
2.3.1 Latency 24
2.3.2 Time Scales 25

2.3.3 Trade-Offs 26
2.3.4 Tuning Efforts 27
2.3.5 Level of Appropriateness 28
2.3.6 When to Stop Analysis 29
2.3.7 Point-in-Time Recommendations 29
2.3.8 Load vs. Architecture 30
2.3.9 Scalability 31
2.3.10 Metrics 32
2.3.11 Utilization 33
2.3.12 Saturation 34
2.3.13 Profiling 35
2.3.14 Caching 35
2.3.15 Known-Unknowns 37
2.4 Perspectives 37
2.4.1 Resource Analysis 38
2.4.2 Workload Analysis 39
2.5 Methodology 40
2.5.1 Streetlight Anti-Method 42
2.5.2 Random Change Anti-Method 42
2.5.3 Blame-Someone-Else Anti-Method 43
2.5.4 Ad Hoc Checklist Method 43
2.5.5 Problem Statement 44
2.5.6 Scientific Method 44
2.5.7 Diagnosis Cycle 46
2.5.8 Tools Method 46
2.5.9 The USE Method 47
2.5.10 The RED Method 53
2.5.11 Workload Characterization 54
2.5.12 Drill-Down Analysis 55
2.5.13 Latency Analysis 56
2.5.14 Method R 57
2.5.15 Event Tracing 57
2.5.16 Baseline Statistics 59
2.5.17 Static Performance Tuning 59
2.5.18 Cache Tuning 60
2.5.19 Micro-Benchmarking 60
2.5.20 Performance Mantras 61

2.6 Modeling 62
2.6.1 Enterprise vs. Cloud 62
2.6.2 Visual Identification 62
2.6.3 Amdahl's Law of Scalability 64
2.6.4 Universal Scalability Law 65
2.6.5 Queueing Theory 66
2.7 Capacity Planning 69
2.7.1 Resource Limits 70
2.7.2 Factor Analysis 71
2.7.3 Scaling Solutions 72
2.8 Statistics 73
2.8.1 Quantifying Performance Gains 73
2.8.2 Averages 74
2.8.3 Standard Deviation, Percentiles, Median 75
2.8.4 Coefficient of Variation 76
2.8.5 Multimodal Distributions 76
2.8.6 Outliers 77
2.9 Monitoring 77
2.9.1 Time-Based Patterns 77
2.9.2 Monitoring Products 79
2.9.3 Summary-Since-Boot 79
2.10 Visualizations 79
2.10.1 Line Chart 80
2.10.2 Scatter Plots 81
2.10.3 Heat Maps 82
2.10.4 Timeline Charts 83
2.10.5 Surface Plot 84
2.10.6 Visualization Tools 85
2.11 Exercises 85
2.12 References 86

3 Operating Systems 89
3.1 Terminology 90
3.2 Background 91
3.2.1 Kernel 91
3.2.2 Kernel and User Modes 93
3.2.3 System Calls 94

3.2.4 Interrupts 96
3.2.5 Clock and Idle 99
3.2.6 Processes 99
3.2.7 Stacks 102
3.2.8 Virtual Memory 104
3.2.9 Schedulers 105
3.2.10 File Systems 106
3.2.11 Caching 108
3.2.12 Networking 109
3.2.13 Device Drivers 109
3.2.14 Multiprocessor 110
3.2.15 Preemption 110
3.2.16 Resource Management 110
3.2.17 Observability 111
3.3 Kernels 111
3.3.1 Unix 112
3.3.2 BSD 113
3.3.3 Solaris 114
3.4 Linux 114
3.4.1 Linux Kernel Developments 115
3.4.2 systemd 120
3.4.3 KPTI (Meltdown) 121
3.4.4 Extended BPF 121
3.5 Other Topics 122
3.5.1 PGO Kernels 122
3.5.2 Unikernels 123
3.5.3 Microkernels and Hybrid Kernels 123
3.5.4 Distributed Operating Systems 123
3.6 Kernel Comparisons 124
3.7 Exercises 124
3.8 References 125
3.8.1 Additional Reading 127

4 Observability Tools 129
4.1 Tool Coverage 130
4.1.1 Static Performance Tools 130
4.1.2 Crisis Tools 131

4.2 Tool Types 133
4.2.1 Fixed Counters 133
4.2.2 Profiling 135
4.2.3 Tracing 136
4.2.4 Monitoring 137
4.3 Observability Sources 138
4.3.1 /proc 140
4.3.2 /sys 143
4.3.3 Delay Accounting 145
4.3.4 netlink 145
4.3.5 Tracepoints 146
4.3.6 kprobes 151
4.3.7 uprobes 153
4.3.8 USDT 155
4.3.9 Hardware Counters (PMCs) 156
4.3.10 Other Observability Sources 159
4.4 sar 160
4.4.1 sar(1) Coverage 161
4.4.2 sar(1) Monitoring 161
4.4.3 sar(1) Live 165
4.4.4 sar(1) Documentation 165
4.5 Tracing Tools 166
4.6 Observing Observability 167
4.7 Exercises 168
4.8 References 168

5 Applications 171
5.1 Application Basics 172
5.1.1 Objectives 173
5.1.2 Optimize the Common Case 174
5.1.3 Observability 174
5.1.4 Big O Notation 175
5.2 Application Performance Techniques 176
5.2.1 Selecting an I/O Size 176
5.2.2 Caching 176
5.2.3 Buffering 177
5.2.4 Polling 177
5.2.5 Concurrency and Parallelism 177

5.2.6 Non-Blocking I/O 181
5.2.7 Processor Binding 181
5.2.8 Performance Mantras 182
5.3 Programming Languages 182
5.3.1 Compiled Languages 183
5.3.2 Interpreted Languages 184
5.3.3 Virtual Machines 185
5.3.4 Garbage Collection 185
5.4 Methodology 186
5.4.1 CPU Profiling 187
5.4.2 Off-CPU Analysis 189
5.4.3 Syscall Analysis 192
5.4.4 USE Method 193
5.4.5 Thread State Analysis 193
5.4.6 Lock Analysis 198
5.4.7 Static Performance Tuning 198
5.4.8 Distributed Tracing 199
5.5 Observability Tools 199
5.5.1 perf 200
5.5.2 profile 203
5.5.3 offcputime 204
5.5.4 strace 205
5.5.5 execsnoop 207
5.5.6 syscount 208
5.5.7 bpftrace 209
5.6 Gotchas 213
5.6.1 Missing Symbols 214
5.6.2 Missing Stacks 215
5.7 Exercises 216
5.8 References 217

6 CPUs 219
6.1 Terminology 220
6.2 Models 221
6.2.1 CPU Architecture 221
6.2.2 CPU Memory Caches 221
6.2.3 CPU Run Queues 222

6.3 Concepts 223
6.3.1 Clock Rate 223
6.3.2 Instructions 223
6.3.3 Instruction Pipeline 224
6.3.4 Instruction Width 224
6.3.5 Instruction Size 224
6.3.6 SMT 225
6.3.7 IPC, CPI 225
6.3.8 Utilization 226
6.3.9 User Time/Kernel Time 226
6.3.10 Saturation 226
6.3.11 Preemption 227
6.3.12 Priority Inversion 227
6.3.13 Multiprocess, Multithreading 227
6.3.14 Word Size 229
6.3.15 Compiler Optimization 229
6.4 Architecture 229
6.4.1 Hardware 230
6.4.2 Software 241
6.5 Methodology 244
6.5.1 Tools Method 245
6.5.2 USE Method 245
6.5.3 Workload Characterization 246
6.5.4 Profiling 247
6.5.5 Cycle Analysis 251
6.5.6 Performance Monitoring 251
6.5.7 Static Performance Tuning 252
6.5.8 Priority Tuning 252
6.5.9 Resource Controls 253
6.5.10 CPU Binding 253
6.5.11 Micro-Benchmarking 253
6.6 Observability Tools 254
6.6.1 uptime 255
6.6.2 vmstat 258
6.6.3 mpstat 259
6.6.4 sar 260
6.6.5 ps 260

6.6.6 top 261
6.6.7 pidstat 262
6.6.8 time, ptime 263
6.6.9 turbostat 264
6.6.10 showboost 265
6.6.11 pmcarch 265
6.6.12 tlbstat 266
6.6.13 perf 267
6.6.14 profile 277
6.6.15 cpudist 278
6.6.16 runqlat 279
6.6.17 runqlen 280
6.6.18 softirqs 281
6.6.19 hardirqs 282
6.6.20 bpftrace 282
6.6.21 Other Tools 285
6.7 Visualizations 288
6.7.1 Utilization Heat Map 288
6.7.2 Subsecond-Offset Heat Map 289
6.7.3 Flame Graphs 289
6.7.4 FlameScope 292
6.8 Experimentation 293
6.8.1 Ad Hoc 293
6.8.2 SysBench 294
6.9 Tuning 294
6.9.1 Compiler Options 295
6.9.2 Scheduling Priority and Class 295
6.9.3 Scheduler Options 295
6.9.4 Scaling Governors 297
6.9.5 Power States 297
6.9.6 CPU Binding 297
6.9.7 Exclusive CPU Sets 298
6.9.8 Resource Controls 298
6.9.9 Security Boot Options 298
6.9.10 Processor Options (BIOS Tuning) 299
6.10 Exercises 299
6.11 References 300

7 Memory 303
7.1 Terminology 304
7.2 Concepts 305
7.2.1 Virtual Memory 305
7.2.2 Paging 306
7.2.3 Demand Paging 307
7.2.4 Overcommit 308
7.2.5 Process Swapping 308
7.2.6 File System Cache Usage 309
7.2.7 Utilization and Saturation 309
7.2.8 Allocators 309
7.2.9 Shared Memory 310
7.2.10 Working Set Size 310
7.2.11 Word Size 310
7.3 Architecture 311
7.3.1 Hardware 311
7.3.2 Software 315
7.3.3 Process Virtual Address Space 319
7.4 Methodology 323
7.4.1 Tools Method 323
7.4.2 USE Method 324
7.4.3 Characterizing Usage 325
7.4.4 Cycle Analysis 326
7.4.5 Performance Monitoring 326
7.4.6 Leak Detection 326
7.4.7 Static Performance Tuning 327
7.4.8 Resource Controls 328
7.4.9 Micro-Benchmarking 328
7.4.10 Memory Shrinking 328
7.5 Observability Tools 328
7.5.1 vmstat 329
7.5.2 PSI 330
7.5.3 swapon 331
7.5.4 sar 331
7.5.5 slabtop 333
7.5.6 numastat 334
7.5.7 ps 335
7.5.8 top 336

7.5.9 pmap 337
7.5.10 perf 338
7.5.11 drsnoop 342
7.5.12 wss 342
7.5.13 bpftrace 343
7.5.14 Other Tools 347
7.6 Tuning 350
7.6.1 Tunable Parameters 350
7.6.2 Multiple Page Sizes 352
7.6.3 Allocators 353
7.6.4 NUMA Binding 353
7.6.5 Resource Controls 353
7.7 Exercises 354
7.8 References 355

8 File Systems 359
8.1 Terminology 360
8.2 Models 361
8.2.1 File System Interfaces 361
8.2.2 File System Cache 361
8.2.3 Second-Level Cache 362
8.3 Concepts 362
8.3.1 File System Latency 362
8.3.2 Caching 363
8.3.3 Random vs. Sequential I/O 363
8.3.4 Prefetch 364
8.3.5 Read-Ahead 365
8.3.6 Write-Back Caching 365
8.3.7 Synchronous Writes 366
8.3.8 Raw and Direct I/O 366
8.3.9 Non-Blocking I/O 366
8.3.10 Memory-Mapped Files 367
8.3.11 Metadata 367
8.3.12 Logical vs. Physical I/O 368
8.3.13 Operations Are Not Equal 370
8.3.14 Special File Systems 371
8.3.15 Access Timestamps 371
8.3.16 Capacity 371

8.4 Architecture 372
8.4.1 File System I/O Stack 372
8.4.2 VFS 373
8.4.3 File System Caches 373
8.4.4 File System Features 375
8.4.5 File System Types 377
8.4.6 Volumes and Pools 382
8.5 Methodology 383
8.5.1 Disk Analysis 384
8.5.2 Latency Analysis 384
8.5.3 Workload Characterization 386
8.5.4 Performance Monitoring 388
8.5.5 Static Performance Tuning 389
8.5.6 Cache Tuning 389
8.5.7 Workload Separation 389
8.5.8 Micro-Benchmarking 390
8.6 Observability Tools 391
8.6.1 mount 392
8.6.2 free 392
8.6.3 top 393
8.6.4 vmstat 393
8.6.5 sar 393
8.6.6 slabtop 394
8.6.7 strace 395
8.6.8 fatrace 395
8.6.9 LatencyTOP 396
8.6.10 opensnoop 397
8.6.11 filetop 398
8.6.12 cachestat 399
8.6.13 ext4dist (xfs, zfs, btrfs, nfs) 399
8.6.14 ext4slower (xfs, zfs, btrfs, nfs) 401
8.6.15 bpftrace 402
8.6.17 Other Tools 409
8.6.18 Visualizations 410
8.7 Experimentation 411
8.7.1 Ad Hoc 411
8.7.2 Micro-Benchmark Tools 412
8.7.3 Cache Flushing 414

8.8 Tuning 414
8.8.1 Application Calls 415
8.8.2 ext4 416
8.8.3 ZFS 418
8.9 Exercises 419
8.10 References 420

9 Disks 423
9.1 Terminology 424
9.2 Models 425
9.2.1 Simple Disk 425
9.2.2 Caching Disk 425
9.2.3 Controller 426
9.3 Concepts 427
9.3.1 Measuring Time 427
9.3.2 Time Scales 429
9.3.3 Caching 430
9.3.4 Random vs. Sequential I/O 430
9.3.5 Read/Write Ratio 431
9.3.6 I/O Size 432
9.3.7 IOPS Are Not Equal 432
9.3.8 Non-Data-Transfer Disk Commands 432
9.3.9 Utilization 433
9.3.10 Saturation 434
9.3.11 I/O Wait 434
9.3.12 Synchronous vs. Asynchronous 434
9.3.13 Disk vs. Application I/O 435
9.4 Architecture 435
9.4.1 Disk Types 435
9.4.2 Interfaces 442
9.4.3 Storage Types 443
9.4.4 Operating System Disk I/O Stack 446
9.5 Methodology 449
9.5.1 Tools Method 450
9.5.2 USE Method 450
9.5.3 Performance Monitoring 452
9.5.4 Workload Characterization 452
9.5.5 Latency Analysis 454

9.5.6 Static Performance Tuning 455
9.5.7 Cache Tuning 456
9.5.8 Resource Controls 456
9.5.9 Micro-Benchmarking 456
9.5.10 Scaling 457
9.6 Observability Tools 458
9.6.1 iostat 459
9.6.2 sar 463
9.6.3 PSI 464
9.6.4 pidstat 464
9.6.5 perf 465
9.6.6 biolatency 468
9.6.7 biosnoop 470
9.6.8 iotop, biotop 472
9.6.9 biostacks 474
9.6.10 blktrace 475
9.6.11 bpftrace 479
9.6.12 MegaCli 484
9.6.13 smartctl 484
9.6.14 SCSI Logging 486
9.6.15 Other Tools 487
9.7 Visualizations 487
9.7.1 Line Graphs 487
9.7.2 Latency Scatter Plots 488
9.7.3 Latency Heat Maps 488
9.7.4 Offset Heat Maps 489
9.7.5 Utilization Heat Maps 490
9.8 Experimentation 490
9.8.1 Ad Hoc 490
9.8.2 Custom Load Generators 491
9.8.3 Micro-Benchmark Tools 491
9.8.4 Random Read Example 491
9.8.5 ioping 492
9.8.6 fio 493
9.8.7 blkreplay 493
9.9 Tuning 493
9.9.1 Operating System Tunables 493

9.9.2 Disk Device Tunables 494
9.9.3 Disk Controller Tunables 494
9.10 Exercises 495
9.11 References 496

10 Network 499
10.1 Terminology 500
10.2 Models 501
10.2.1 Network Interface 501
10.2.2 Controller 501
10.2.3 Protocol Stack 502
10.3 Concepts 503
10.3.1 Networks and Routing 503
10.3.2 Protocols 504
10.3.3 Encapsulation 504
10.3.4 Packet Size 504
10.3.5 Latency 505
10.3.6 Buffering 507
10.3.7 Connection Backlog 507
10.3.8 Interface Negotiation 508
10.3.9 Congestion Avoidance 508
10.3.10 Utilization 508
10.3.11 Local Connections 509
10.4 Architecture 509
10.4.1 Protocols 509
10.4.2 Hardware 515
10.4.3 Software 517
10.5 Methodology 524
10.5.1 Tools Method 525
10.5.2 USE Method 526
10.5.3 Workload Characterization 527
10.5.4 Latency Analysis 528
10.5.5 Performance Monitoring 529
10.5.6 Packet Sniffing 530
10.5.7 TCP Analysis 531
10.5.8 Static Performance Tuning 531
10.5.9 Resource Controls 532
10.5.10 Micro-Benchmarking 533

10.6 Observability Tools 533
10.6.1 ss 534
10.6.2 ip 536
10.6.3 ifconfig 537
10.6.4 nstat 538
10.6.5 netstat 539
10.6.6 sar 543
10.6.7 nicstat 545
10.6.8 ethtool 546
10.6.9 tcplife 548
10.6.10 tcptop 549
10.6.11 tcpretrans 549
10.6.12 bpftrace 550
10.6.13 tcpdump 558
10.6.14 Wireshark 560
10.6.15 Other Tools 560
10.7 Experimentation 562
10.7.1 ping 562
10.7.2 traceroute 563
10.7.3 pathchar 564
10.7.4 iperf 564
10.7.5 netperf 565
10.7.6 tc 566
10.7.7 Other Tools 567
10.8 Tuning 567
10.8.1 System-Wide 567
10.8.2 Socket Options 573
10.8.3 Configuration 574
10.9 Exercises 574
10.10 References 575

11 Cloud Computing 579
11.1 Background 580
11.1.1 Instance Types 581
11.1.2 Scalable Architecture 581
11.1.3 Capacity Planning 582
11.1.4 Storage 584
11.1.5 Multitenancy 585
11.1.6 Orchestration (Kubernetes) 586

11.2 Hardware Virtualization 587
11.2.1 Implementation 588
11.2.2 Overhead 589
11.2.3 Resource Controls 595
11.2.4 Observability 597
11.3 OS Virtualization 605
11.3.1 Implementation 607
11.3.2 Overhead 610
11.3.3 Resource Controls 613
11.3.4 Observability 617
11.4 Lightweight Virtualization 630
11.4.1 Implementation 631
11.4.2 Overhead 632
11.4.3 Resource Controls 632
11.4.4 Observability 632
11.5 Other Types 634
11.6 Comparisons 634
11.7 Exercises 636
11.8 References 637

12 Benchmarking 641
12.1 Background 642
12.1.1 Reasons 642
12.1.2 Effective Benchmarking 643
12.1.3 Benchmarking Failures 645
12.2 Benchmarking Types 651
12.2.1 Micro-Benchmarking 651
12.2.2 Simulation 653
12.2.3 Replay 654
12.2.4 Industry Standards 654
12.3 Methodology 656
12.3.1 Passive Benchmarking 656
12.3.2 Active Benchmarking 657
12.3.3 CPU Profiling 660
12.3.4 USE Method 661
12.3.5 Workload Characterization 662
12.3.6 Custom Benchmarks 662
12.3.7 Ramping Load 662

12.3.8 Sanity Check 664
12.3.9 Statistical Analysis 665
12.3.10 Benchmarking Checklist 666
12.4 Benchmark Questions 667
12.5 Exercises 668
12.6 References 669

13 perf 671
13.1 Subcommands Overview 672
13.2 One-Liners 674
13.3 perf Events 679
13.4 Hardware Events 681
13.4.1 Frequency Sampling 682
13.5 Software Events 683
13.6 Tracepoint Events 684
13.7 Probe Events 685
13.7.1 kprobes 685
13.7.2 uprobes 687
13.7.3 USDT 690
13.8 perf stat 691
13.8.1 Options 692
13.8.2 Interval Statistics 693
13.8.3 Per-CPU Balance 693
13.8.4 Event Filters 693
13.8.5 Shadow Statistics 694
13.9 perf record 694
13.9.1 Options 695
13.9.2 CPU Profiling 695
13.9.3 Stack Walking 696
13.10 perf report 696
13.10.1 TUI 697
13.10.2 STDIO 697
13.11 perf script 698
13.11.1 Flame Graphs 700
13.11.2 Trace Scripts 700
13.12 perf trace 701
13.12.1 Kernel Versions 702
13.13 Other Commands 702

13.14 perf Documentation 703
13.15 References 703

14 Ftrace 705
14.1 Capabilities Overview 706
14.2 tracefs (/sys) 708
14.2.1 tracefs Contents 709
14.3 Ftrace Function Profiler 711
14.4 Ftrace Function Tracing 713
14.4.1 Using trace 713
14.4.2 Using trace_pipe 715
14.4.3 Options 716
14.5 Tracepoints 717
14.5.1 Filter 717
14.5.2 Trigger 718
14.6 kprobes 719
14.6.1 Event Tracing 719
14.6.2 Arguments 720
14.6.3 Return Values 721
14.6.4 Filters and Triggers 721
14.6.5 kprobe Profiling 722
14.7 uprobes 722
14.7.1 Event Tracing 722
14.7.2 Arguments and Return Values 723
14.7.3 Filters and Triggers 723
14.7.4 uprobe Profiling 723
14.8 Ftrace function_graph 724
14.8.1 Graph Tracing 724
14.8.2 Options 725
14.9 Ftrace hwlat 726
14.10 Ftrace Hist Triggers 727
14.10.1 Single Keys 727
14.10.2 Fields 728
14.10.3 Modifiers 729
14.10.4 PID Filters 729
14.10.5 Multiple Keys 730
14.10.6 Stack Trace Keys 730
14.10.7 Synthetic Events 731

14.11 trace-cmd 734
14.11.1 Subcommands Overview 734
14.11.2 trace-cmd One-Liners 736
14.11.3 trace-cmd vs. perf(1) 738
14.11.4 trace-cmd function_graph 739
14.11.5 KernelShark 739
14.11.6 trace-cmd Documentation 740
14.12 perf ftrace 741
14.13 perf-tools 741
14.13.1 Tool Coverage 742
14.13.2 Single-Purpose Tools 743
14.13.3 Multi-Purpose Tools 744
14.13.4 perf-tools One-Liners 745
14.13.5 Example 747
14.13.6 perf-tools vs. BCC/BPF 747
14.13.7 Documentation 748
14.14 Ftrace Documentation 748
14.15 References 749

15 BPF 751
15.1 BCC 753
15.1.1 Installation 754
15.1.2 Tool Coverage 754
15.1.3 Single-Purpose Tools 755
15.1.4 Multi-Purpose Tools 757
15.1.5 One-Liners 757
15.1.6 Multi-Tool Example 759
15.1.7 BCC vs. bpftrace 760
15.1.8 Documentation 760
15.2 bpftrace 761
15.2.1 Installation 762
15.2.2 Tools 762
15.2.3 One-Liners 763
15.2.4 Programming 766
15.2.5 Reference 774
15.2.6 Documentation 781
15.3 References 782

16 Case Study 783

16.1 An Unexplained Win 783

16.1.1 Problem Statement 783

16.1.2 Analysis Strategy 784

16.1.3 Statistics 784

16.1.4 Configuration 786

16.1.5 PMCs 788

16.1.6 Software Events 789

16.1.7 Tracing 790

16.1.8 Conclusion 792

16.2 Additional Information 792

16.3 References 793

A USE Method: Linux 795

B sar Summary 801

C bpftrace One-Liners 803

D Solutions to Selected Exercises 809

E Systems Performance Who's Who 811

Glossary 815

Index 825

Preface

> *"There are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns—there are things we do not know we don't know."*
>
> —U.S. Secretary of Defense Donald Rumsfeld, February 12, 2002

While the previous statement was met with chuckles from those attending the press briefing, it summarizes an important principle that is as relevant in complex technical systems as it is in geopolitics: performance issues can originate from anywhere, including areas of the system that you know nothing about and you are therefore not checking (the unknown unknowns). This book may reveal many of these areas, while providing methodologies and tools for their analysis.

About This Edition

I wrote the first edition eight years ago and designed it to have a long shelf life. Chapters are structured to first cover durable skills (models, architecture, and methodologies) and then faster-changing skills (tools and tuning) as example implementations. While the example tools and tuning will go out of date, the durable skills show you how to stay updated.

There has been a large addition to Linux in the past eight years: Extended BPF, a kernel technology that powers a new generation of performance analysis tools, which is used by companies including Netflix and Facebook. I have included a BPF chapter and BPF tools in this new edition, and I have also published a deeper reference on the topic [Gregg 19]. The Linux perf and Ftrace tools have also seen many developments, and I have added separate chapters for them as well. The Linux kernel has gained many performance features and technologies, also covered. The hypervisors that drive cloud computing virtual machines, and container technologies, have also changed considerably; that content has been updated.

The first edition covered both Linux and Solaris equally. Solaris market share has shrunk considerably in the meantime [ITJobsWatch 20], so the Solaris content has been largely removed from this edition, making room for more Linux content to be included. However, your understanding of an operating system or kernel can be enhanced by considering an alternative, for perspective. For that reason, some mentions of Solaris and other operating systems are included in this edition.

For the past six years I have been a senior performance engineer at Netflix, applying the field of systems performance to the Netflix microservices environment. I've worked on the performance of hypervisors, containers, runtimes, kernels, databases, and applications. I've developed new methodologies and tools as needed, and worked with experts in cloud performance and Linux kernel engineering. These experiences have contributed to improving this edition.

About This Book

Welcome to *Systems Performance: Enterprise and the Cloud*, 2nd Edition! This book is about the performance of operating systems and of applications from the operating system context, and it is written for both enterprise server and cloud computing environments. Much of the material in this book can also aid your analysis of client devices and desktop operating systems. My aim is to help you get the most out of your systems, whatever they are.

When working with application software that is under constant development, you may be tempted to think of operating system performance—where the kernel has been developed and tuned for decades—as a solved problem. It isn't! The operating system is a complex body of software, managing a variety of ever-changing physical devices with new and different application workloads. The kernels are also in constant development, with features being added to improve the performance of particular workloads, and newly encountered bottlenecks being removed as systems continue to scale. Kernel changes such as the mitigations for the Meltdown vulnerability that were introduced in 2018 can also hurt performance. Analyzing and working to improve the performance of the operating system is an ongoing task that should lead to continual performance improvements. Application performance can also be analyzed from the operating system context to find more clues that might be missed using application-specific tools alone; I'll cover that here as well.

Operating System Coverage

The main focus of this book is the study of systems performance, using Linux-based operating systems on Intel processors as the primary example. The content is structured to help you study other kernels and processors as well.

Unless otherwise noted, the specific Linux distribution is not important in the examples used. The examples are mostly from the Ubuntu distribution and, when necessary, notes are included to explain differences for other distributions. The examples are also taken from a variety of system types: bare metal and virtualized, production and test, servers and client devices.

Across my career I've worked with a variety of different operating systems and kernels, and this has deepened my understanding of their design. To deepen your understanding as well, this book includes some mentions of Unix, BSD, Solaris, and Windows.

Other Content

Example screenshots from performance tools are included, not just for the data shown, but also to illustrate the types of data available. The tools often present the data in intuitive and self-explanatory ways, many in the familiar style of earlier Unix tools. This means that screenshots can be a powerful way to convey the purpose of these tools, some requiring little additional description. (If a tool does require laborious explanation, that may be a failure of design!)

Where it provides useful insight to deepen your understanding, I touch upon the history of certain technologies. It is also useful to learn a bit about the key people in this industry: you're likely to come across them or their work in performance and other contexts. A "who's who" list has been provided in Appendix E.

A handful of topics in this book were also covered in my prior book, *BPF Performance Tools* [Gregg 19]: in particular, BPF, BCC, bpftrace, tracepoints, kprobes, uprobes, and various BPF-based tools. You can refer to that book for more information. The summaries of these topics in this book are often based on that earlier book, and sometimes use the same text and examples.

What Isn't Covered

This book focuses on performance. To undertake all the example tasks given will require, at times, some system administration activities, including the installation or compilation of software (which is not covered here).

The content also summarizes operating system internals, which are covered in more detail in separate dedicated texts. Advanced performance analysis topics are summarized so that you are aware of their existence and can study them as needed from additional sources. See the Supplemental Material section at the end of this Preface.

How This Book Is Structured

Chapter 1, Introduction, is an introduction to systems performance analysis, summarizing key concepts and providing examples of performance activities.

Chapter 2, Methodologies, provides the background for performance analysis and tuning, including terminology, concepts, models, methodologies for observation and experimentation, capacity planning, analysis, and statistics.

Chapter 3, Operating Systems, summarizes kernel internals for the performance analyst. This is necessary background for interpreting and understanding what the operating system is doing.

Chapter 4, Observability Tools, introduces the types of system observability tools available, and the interfaces and frameworks upon which they are built.

Chapter 5, Applications, discusses application performance topics and observing them from the operating system.

Chapter 6, CPUs, covers processors, cores, hardware threads, CPU caches, CPU interconnects, device interconnects, and kernel scheduling.

Chapter 7, Memory, is about virtual memory, paging, swapping, memory architectures, buses, address spaces, and allocators.

Chapter 8, File Systems, is about file system I/O performance, including the different caches involved.

Chapter 9, Disks, covers storage devices, disk I/O workloads, storage controllers, RAID, and the kernel I/O subsystem.

Chapter 10, Network, is about network protocols, sockets, interfaces, and physical connections.

Chapter 11, Cloud Computing, introduces operating system– and hardware-based virtualization methods in common use for cloud computing, along with their performance overhead, isolation, and observability characteristics. This chapter covers hypervisors and containers.

Chapter 12, Benchmarking, shows how to benchmark accurately, and how to interpret others' benchmark results. This is a surprisingly tricky topic, and this chapter shows how you can avoid common mistakes and try to make sense of it.

Chapter 13, perf, summarizes the standard Linux profiler, perf(1), and its many capabilities. This is a reference to support perf(1)'s use throughout the book.

Chapter 14, Ftrace, summarizes the standard Linux tracer, Ftrace, which is especially suited for exploring kernel code execution.

Chapter 15, BPF, summarizes the standard BPF front ends: BCC and bpftrace.

Chapter 16, Case Study, contains a systems performance case study from Netflix, showing how a production performance puzzle was analyzed from beginning to end.

Chapters 1 to 4 provide essential background. After reading them, you can reference the remainder of the book as needed, in particular Chapters 5 to 12, which cover specific targets for analysis. Chapters 13 to 15 cover advanced profiling and tracing, and are optional reading for those who wish to learn one or more tracers in more detail.

Chapter 16 uses a storytelling approach to paint a bigger picture of a performance engineer's work. If you're new to performance analysis, you might want to read this first as an example of performance analysis using a variety of different tools, and then return to it when you've read the other chapters.

As a Future Reference

This book has been written to provide value for many years, by focusing on background and methodologies for the systems performance analyst.

To support this, many chapters have been separated into two parts. The first part consists of terms, concepts, and methodologies (often with those headings), which should stay relevant many years from now. The second provides examples of how the first part is implemented: architecture, analysis tools, and tunables, which, while they will become out-of-date, will still be useful as examples.

Tracing Examples

We frequently need to explore the operating system in depth, which can be done using tracing tools.

Since the first edition of this book, extended BPF has been developed and merged into the Linux kernel, powering a new generation of tracing tools that use the BCC and bpftrace front ends. This book focuses on BCC and bpftrace, and also the Linux kernel's built-in Ftrace tracer. BPF, BCC, and bpftrace, are covered in more depth in my prior book [Gregg 19].

Linux perf is also included in this book and is another tool that can do tracing. However, perf is usually included in chapters for its sampling and PMC analysis capabilities, rather than for tracing.

You may need or wish to use different tracing tools, which is fine. The tracing tools in this book are used to show the questions that you can ask of the system. It is often these questions, and the methodologies that pose them, that are the most difficult to know.

Intended Audience

The intended audience for this book is primarily systems administrators and operators of enterprise and cloud computing environments. It is also a reference for developers, database administrators, and web server administrators who need to understand operating system and application performance.

As a performance engineer at a company with a large compute environment (Netflix), I frequently work with SREs (site reliability engineers) and developers who are under enormous time pressure to solve multiple simultaneous performance issues. I have also been on the Netflix CORE SRE on-call rotation and have experienced this pressure firsthand. For many people, performance is not their primary job, and they need to know just enough to solve the current issues. Knowing that your time may be limited has encouraged me to keep this book as short as possible, and structure it to facilitate jumping ahead to specific chapters.

Another intended audience is students: this book is also suitable as a supporting text for a systems performance course. I have taught these classes before and learned which types of material work best in leading students to solve performance problems; that has guided my choice of content for this book.

Whether or not you are a student, the chapter exercises give you an opportunity to review and apply the material. These include some optional advanced exercises, which you are not expected to solve. (They may be impossible; they should at least be thought-provoking.)

In terms of company size, this book should contain enough detail to satisfy environments from small to large, including those with dozens of dedicated performance staff. For many smaller companies, the book may serve as a reference when needed, with only some portions of it used day to day.

Typographic Conventions

The following typographical conventions are used throughout this book:

Example	Description
netif_receive_skb()	Function name
iostat(1)	A command referenced by chapter 1 of its man page
read(2)	A system call referenced by its man page
malloc(3)	A C library function call referenced by its man page
vmstat(8)	An administration command referenced by its man page
Documentation/...	Linux documentation in the Linux kernel source tree
kernel/...	Linux kernel source code
fs/...	Linux kernel source code, file systems
CONFIG_...	Linux kernel configuration option (Kconfig)
`r_await`	Command line input and output

Example	Description
`mpstat 1`	Highlighting of a typed command or key detail
`#`	Superuser (root) shell prompt
`$`	User (non-root) shell prompt
`^C`	A command was interrupted (Ctrl-C)
`[...]`	Truncation

Supplemental Material, References, and Bibliography

References are listed are at the end of each chapter rather than in a single bibliography, allowing you to browse references related to each chapter's topic. The following selected texts can also be referenced for further background on operating systems and performance analysis:

[Jain 91] Jain, R., *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, Wiley, 1991.

[Vahalia 96] Vahalia, U., *UNIX Internals: The New Frontiers*, Prentice Hall, 1996.

[Cockcroft 98] Cockcroft, A., and Pettit, R., *Sun Performance and Tuning: Java and the Internet*, Prentice Hall, 1998.

[Musumeci 02] Musumeci, G. D., and Loukides, M., *System Performance Tuning*, 2nd Edition, O'Reilly, 2002.

[Bovet 05] Bovet, D., and Cesati, M., *Understanding the Linux Kernel,* 3rd Edition, O'Reilly, 2005.

[McDougall 06a] McDougall, R., Mauro, J., and Gregg, B., *Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris*, Prentice Hall, 2006.

[Gove 07] Gove, D., *Solaris Application Programming*, Prentice Hall, 2007.

[Love 10] Love, R., *Linux Kernel Development*, 3rd Edition, Addison-Wesley, 2010.

[Gregg 11a] Gregg, B., and Mauro, J., *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*, Prentice Hall, 2011.

[Gregg 13a] Gregg, B., *Systems Performance: Enterprise and the Cloud*, Prentice Hall, 2013 (first edition).

[Gregg 19] Gregg, B., *BPF Performance Tools: Linux System and Application Observability*, Addison-Wesley, 2019.

[ITJobsWatch 20] ITJobsWatch, "Solaris Jobs," https://www.itjobswatch.co.uk/jobs/uk/solaris.do#demand_trend, accessed 2020.

Acknowledgments

Thanks to all those who bought the first edition, especially those who made it recommended or required reading at their companies. Your support for the first edition has led to the creation of this second edition. Thank you.

This is the latest book on systems performance, but not the first. I'd like to thank prior authors for their work, work that I have built upon and referenced in this text. In particular I'd like to thank Adrian Cockcroft, Jim Mauro, Richard McDougall, Mike Loukides, and Raj Jain. As they have helped me, I hope to help you.

I'm grateful for everyone who provided feedback on this edition:

Deirdré Straughan has again supported me in various ways throughout this book, including using her years of experience in technical copy editing to improve every page. The words you read are from both of us. We enjoy not just spending time together (we are married now), but also working together. Thank you.

Philipp Marek is an IT forensics specialist, IT architect, and performance engineer at the Austrian Federal Computing Center. He provided early technical feedback on every topic in this book (an amazing feat) and even spotted problems in the first edition text. Philipp started programming in 1983 on a 6502, and has been looking for additional CPU cycles ever since. Thanks, Philipp, for your expertise and relentless work.

Dale Hamel (Shopify) also reviewed every chapter, providing important insights for various cloud technologies, and another consistent point of view across the entire book. Thanks for taking this on, Dale—right after helping with the BPF book!

Daniel Borkmann (Isovalent) provided deep technical feedback for a number of chapters, especially the networking chapter, helping me to better understand the complexities and technologies involved. Daniel is a Linux kernel maintainer with years of experience working on the kernel network stack and extended BPF. Thank you, Daniel, for the expertise and rigor.

I'm especially thankful that perf maintainer Arnaldo Carvalho de Melo (Red Hat) helped with Chapter 13, perf; and Ftrace creator Steven Rostedt (VMware) helped with Chapter 14, Ftrace, two topics that I had not covered well enough in the first edition. Apart from their help with this book, I also appreciate their excellent work on these advanced performance tools, tools that I've used to solve countless production issues at Netflix.

It has been a pleasure to have Dominic Kay pick through several chapters and find even more ways to improve their readability and technical accuracy. Dominic also helped with the first edition (and before that, was my colleague at Sun Microsystems working on performance). Thank you, Dominic.

My current performance colleague at Netflix, Amer Ather, provided excellent feedback on several chapters. Amer is a go-to engineer for understanding complex technologies. Zachary Jones (Verizon) also provided feedback for complex topics, and shared his performance expertise to improve the book. Thank you, Amer and Zachary.

A number of reviewers took on multiple chapters and engaged in discussion on specific topics: Alejandro Proaño (Amazon), Bikash Sharma (Facebook), Cory Lueninghoener (Los Alamos

National Laboratory), Greg Dunn (Amazon), John Arrasjid (Ottometric), Justin Garrison (Amazon), Michael Hausenblas (Amazon), and Patrick Cable (Threat Stack). Thanks, all, for your technical help and enthusiasm for the book.

Also thanks to Aditya Sarwade (Facebook), Andrew Gallatin (Netflix), Bas Smit, George Neville-Neil (JUUL Labs), Jens Axboe (Facebook), Joel Fernandes (Google), Randall Stewart (Netflix), Stephane Eranian (Google), and Toke Høiland-Jørgensen (Red Hat), for answering questions and timely technical feedback.

The contributors to my earlier book, *BPF Performance Tools*, have indirectly helped, as some material in this edition is based on that earlier book. That material was improved thanks to Alastair Robertson (Yellowbrick Data), Alexei Starovoitov (Facebook), Daniel Borkmann, Jason Koch (Netflix), Mary Marchini (Netflix), Masami Hiramatsu (Linaro), Mathieu Desnoyers (EfficiOS), Yonghong Song (Facebook), and more. See that book for the full acknowledgments.

This second edition builds upon the work in the first edition. The acknowledgments from the first edition thanked the many people who supported and contributed to that work; in summary, across multiple chapters I had technical feedback from Adam Leventhal, Carlos Cardenas, Darryl Gove, Dominic Kay, Jerry Jelinek, Jim Mauro, Max Bruning, Richard Lowe, and Robert Mustacchi. I also had feedback and support from Adrian Cockcroft, Bryan Cantrill, Dan McDonald, David Pacheco, Keith Wesolowski, Marsell Kukuljevic-Pearce, and Paul Eggleton. Roch Bourbonnais and Richard McDougall helped indirectly as I learned so much from their prior performance engineering work, and Jason Hoffman helped behind the scenes to make the first edition possible.

The Linux kernel is complicated and ever-changing, and I appreciate the stellar work by Jonathan Corbet and Jake Edge of lwn.net for summarizing so many deep topics. Many of their articles are referenced in this book.

A special thanks to Greg Doench, executive editor at Pearson, for his help, encouragement, and flexibility in making this process more efficient than ever. Thanks to content producer Julie Nahil (Pearson) and project manager Rachel Paul, for their attention to detail and help in delivering a quality book. Thanks to copy editor Kim Wimpsett for the working through another one of my lengthy and deeply technical books, finding many ways to improve the text.

And thanks, Mitchell, for your patience and understanding.

Since the first edition, I've continued to work as a performance engineer, debugging issues everywhere in the stack, from applications to metal. I now have many new experiences with performance tuning hypervisors, analyzing runtimes including the JVM, using tracers including Ftrace and BPF in production, and coping with the fast pace of changes in the Netflix microservices environment and the Linux kernel. So much of this is not well documented, and it had been daunting to consider what I needed to do for this edition. But I like a challenge.

About the Author

Brendan Gregg is an industry expert in computing performance and cloud computing. He is a senior performance architect at Netflix, where he does performance design, evaluation, analysis, and tuning. The author of multiple technical books, including *BPF Performance Tools*, he received the USENIX LISA Award for Outstanding Achievement in System Administration. He has also been a kernel engineer, performance lead, and professional technical trainer, and was program co-chair for the USENIX LISA 2018 conference. He has created performance tools included in multiple operating systems, along with visualizations and methodologies for performance analysis, including flame graphs.

This page intentionally left blank

Chapter 3

Operating Systems

An understanding of the operating system and its kernel is essential for systems performance analysis. You will frequently need to develop and then test hypotheses about system behavior, such as how system calls are being performed, how the kernel schedules threads on CPUs, how limited memory could be affecting performance, or how a file system processes I/O. These activities will require you to apply your knowledge of the operating system and the kernel.

The learning objectives of this chapter are:

- Learn kernel terminology: context switches, swapping, paging, preemption, etc.
- Understand the role of the kernel and system calls.
- Gain a working knowledge of kernel internals, including: interrupts, schedulers, virtual memory, and the I/O stack.
- See how kernel performance features have been added from Unix to Linux.
- Develop a basic understanding of extended BPF.

This chapter provides an overview of operating systems and kernels and is assumed knowledge for the rest of the book. If you missed operating systems class, you can treat this as a crash course. Keep an eye out for any gaps in your knowledge, as there will be an exam at the end (I'm kidding; it's just a quiz). For more on kernel internals, see the references at the end of this chapter.

This chapter has three sections:

- **Terminology** lists essential terms.
- **Background** summarizes key operating system and kernel concepts.
- **Kernels** summarizes implementation specifics of Linux and other kernels.

Areas related to performance, including CPU scheduling, memory, disks, file systems, networking, and many specific performance tools, are covered in more detail in the chapters that follow.

3.1 Terminology

For reference, here is the core operating system terminology used in this book. Many of these are also concepts that are explained in more detail in this and later chapters.

- **Operating system**: This refers to the software and files that are installed on a system so that it can boot and execute programs. It includes the kernel, administration tools, and system libraries.
- **Kernel**: The kernel is the program that manages the system, including (depending on the kernel model) hardware devices, memory, and CPU scheduling. It runs in a privileged CPU mode that allows direct access to hardware, called *kernel mode*.
- **Process**: An OS abstraction and environment for executing a program. The program runs in *user mode*, with access to kernel mode (e.g., for performing device I/O) via system calls or traps into the kernel.
- **Thread**: An executable context that can be scheduled to run on a CPU. The kernel has multiple threads, and a process contains one or more.
- **Task**: A Linux runnable entity, which can refer to a process (with a single thread), a thread from a multithreaded process, or kernel threads.
- **BPF program**: A kernel-mode program running in the BPF[1] execution environment.
- **Main memory**: The physical memory of the system (e.g., RAM).
- **Virtual memory**: An abstraction of main memory that supports multitasking and over-subscription. It is, practically, an infinite resource.
- **Kernel space**: The virtual memory address space for the kernel.
- **User space**: The virtual memory address space for processes.
- **User land**: User-level programs and libraries (/usr/bin, /usr/lib...).
- **Context switch**: A switch from running one thread or process to another. This is a normal function of the kernel CPU scheduler, and involves switching the set of running CPU registers (the thread context) to a new set.
- **Mode switch**: A switch between kernel and user modes.
- **System call (syscall)**: A well-defined protocol for user programs to request the kernel to perform privileged operations, including device I/O.
- **Processor**: Not to be confused with *process*, a processor is a physical chip containing one or more CPUs.
- **Trap**: A signal sent to the kernel to request a system routine (privileged action). Trap types include system calls, processor exceptions, and interrupts.

---

[1] BPF originally stood for Berkeley Packet Filter, but the technology today has so little to do with Berkeley, packets, or filtering that BPF has become a name in itself rather than an acronym.

- **Hardware interrupt**: A signal sent by physical devices to the kernel, usually to request servicing of I/O. An interrupt is a type of trap.

The Glossary includes more terminology for reference if needed for this chapter, including *address space*, *buffer*, *CPU*, *file descriptor*, *POSIX*, and *registers*.

3.2 Background

The following sections describe generic operating system and kernel concepts, and will help you understand any operating system. To aid your comprehension, this section includes some Linux implementation details. The next sections, 3.3 Kernels, and 3.4 Linux, focus on Unix, BSD, and Linux kernel implementation specifics.

3.2.1 Kernel

The kernel is the core software of the operating system. What it does depends on the kernel model: Unix-like operating systems including Linux and BSD have a *monolithic* kernel that manages CPU scheduling, memory, file systems, network protocols, and system devices (disks, network interfaces, etc.). This kernel model is shown in Figure 3.1.

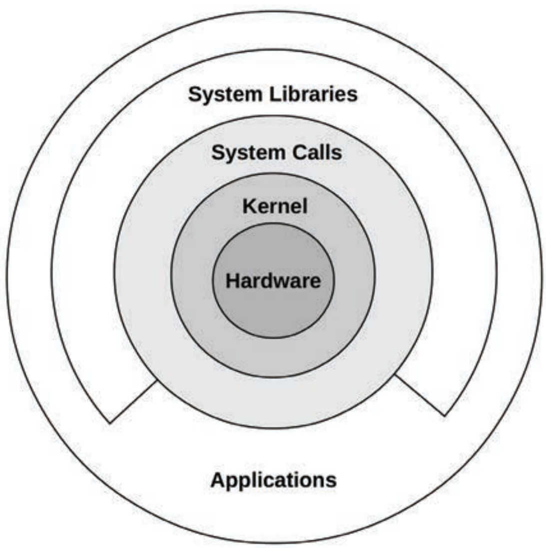


Figure 3.1 Role of a monolithic operating system kernel

Also shown are system libraries, which are often used to provide a richer and easier programming interface than the system calls alone. Applications include all running user-level software, including databases, web servers, administration tools, and operating system shells.

System libraries are pictured here as a broken ring to show that applications can call system calls (*syscalls*) directly.[2] For example, the Golang runtime has its own syscall layer that doesn't require the system library, libc. Traditionally, this diagram is drawn with complete rings, which reflect decreasing levels of privilege starting with the kernel at the center (a model that originated in Multics [Graham 68], the predecessor of Unix).

Other kernel models also exist: *microkernels* employ a small kernel with functionality moved to user-mode programs; and *unikernels* compile kernel and application code together as a single program. There are also *hybrid kernels*, such as the Windows NT kernel, which use approaches from both monolithic kernels and microkernels together. These are summarized in Section 3.5, Other Topics.

Linux has recently changed its model by allowing a new software type: Extended BPF, which enables secure kernel-mode applications along with its own kernel API: BPF helpers. This allows some applications and system functions to be rewritten in BPF, providing higher levels of security and performance. This is pictured in Figure 3.2.

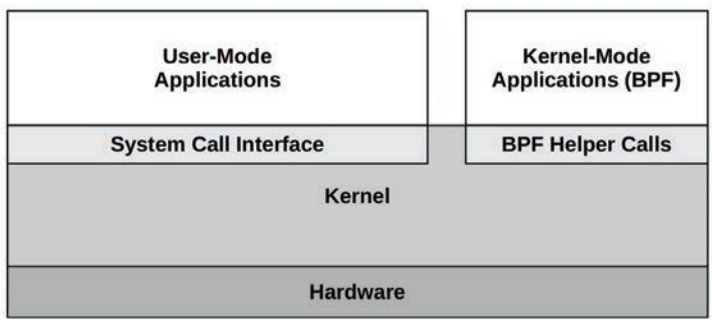


Figure 3.2 BPF applications

Extended BPF is summarized is Section 3.4.4, Extended BPF.

Kernel Execution

The kernel is a large program, typically millions of lines of code. It primarily executes on demand, when a user-level program makes a system call, or a device sends an interrupt. Some kernel threads operate asynchronously for housekeeping, which may include the kernel clock routine and memory management tasks, but these try to be lightweight and consume very little CPU resources.

[2] There are some exceptions to this model. Kernel bypass technologies, sometimes used for networking, allow user-level to access hardware directly (see Chapter 10, Network, Section 10.4.3, Software, heading Kernel Bypass). I/O to hardware may also be submitted without the expense of the syscall interface (although syscalls are required for initialization), for example, with memory-mapped I/O, major faults (see Chapter 7, Memory, Section 7.2.3, Demand Paging), sendfile(2), and Linux io_uring (see Chapter 5, Applications, Section 5.2.6, Non-Blocking I/O).

Workloads that perform frequent I/O, such as web servers, execute mostly in kernel context. Workloads that are compute-intensive usually run in user mode, uninterrupted by the kernel. It may be tempting to think that the kernel cannot affect the performance of these compute-intensive workloads, but there are many cases where it does. The most obvious is CPU contention, when other threads are competing for CPU resources and the kernel scheduler needs to decide which will run and which will wait. The kernel also chooses which CPU a thread will run on and can choose CPUs with warmer hardware caches or better memory locality for the process, to significantly improve performance.

3.2.2 Kernel and User Modes

The kernel runs in a special CPU mode called *kernel mode*, allowing full access to devices and the execution of privileged instructions. The kernel arbitrates device access to support multitasking, preventing processes and users from accessing each other's data unless explicitly allowed.

User programs (processes) run in *user mode*, where they request privileged operations from the kernel via system calls, such as for I/O.

Kernel and user mode are implemented on processors using *privilege rings* (or *protection rings*) following the model in Figure 3.1. For example, x86 processors support four privilege rings, numbered 0 to 3. Typically only two or three are used: for user mode, kernel mode, and the hypervisor if present. Privileged instructions for accessing devices are only allowed in kernel mode; executing them in user mode causes *exceptions*, which are then handled by the kernel (e.g., to generate a permission denied error).

In a traditional kernel, a system call is performed by switching to kernel mode and then executing the system call code. This is shown in Figure 3.3.

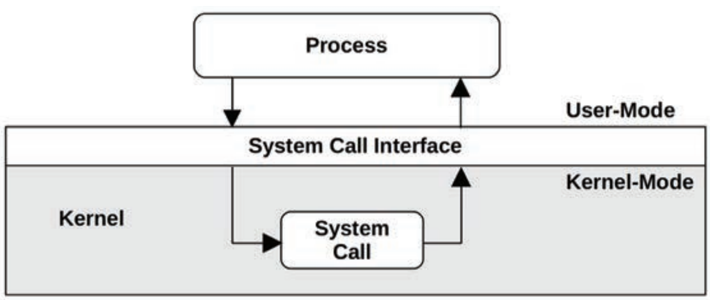


Figure 3.3 System call execution modes

Switching between user and kernel modes is a *mode switch*.

All system calls mode switch. Some system calls also *context switch*: those that are blocking, such as for disk and network I/O, will context switch so that another thread can run while the first is blocked.

Since mode and context switches cost a small amount of overhead (CPU cycles),[3] there are various optimizations to avoid them, including:

- **User-mode syscalls**: It is possible to implement some syscalls in a user-mode library alone. The Linux kernel does this by exporting a virtual dynamic shared object (vDSO) that is mapped into the process address space, which contains syscalls such as gettimeofday(2) and getcpu(2) [Drysdale 14].
- **Memory mappings**: Used for demand paging (see Chapter 7, Memory, Section 7.2.3, Demand Paging), it can also be used for data stores and other I/O, avoiding syscall overheads.
- **Kernel bypass**: This allows user-mode programs to access devices directly, bypassing syscalls and the typical kernel code path. For example, DPDK for networking: the Data Plane Development Kit.
- **Kernel-mode applications**: These include the TUX web server [Lever 00], implemented in-kernel, and more recently the extended BPF technology pictured in Figure 3.2.

Kernel and user mode have their own software execution contexts, including a stack and registers. Some processor architectures (e.g., SPARC) use a separate address space for the kernel, which means the mode switch must also change the virtual memory context.

3.2.3 System Calls

System calls request the kernel to perform privileged system routines. There are hundreds of system calls available, but some effort is made by kernel maintainers to keep that number as small as possible, to keep the kernel simple (Unix philosophy; [Thompson 78]). More sophisticated interfaces can be built upon them in user-land as system libraries, where they are easier to develop and maintain. Operating systems generally include a C standard library that provides easier-to-use interfaces for many common syscalls (e.g., the libc or glibc libraries).

Key system calls to remember are listed in Table 3.1.

Table 3.1 **Key system calls**

System Call	Description
read(2)	Read bytes
write(2)	Write bytes
open(2)	Open a file
close(2)	Close a file
fork(2)	Create a new process
clone(2)	Create a new process or thread
exec(2)	Execute a new program

[3] With the current mitigation for the Meltdown vulnerability, context switches are now more expensive. See Section 3.4.3 KPTI (Meltdown).

System Call	Description
connect(2)	Connect to a network host
accept(2)	Accept a network connection
stat(2)	Fetch file statistics
ioctl(2)	Set I/O properties, or other miscellaneous functions
mmap(2)	Map a file to the memory address space
brk(2)	Extend the heap pointer
futex(2)	Fast user-space mutex

System calls are well documented, each having a man page that is usually shipped with the operating system. They also have a generally simple and consistent interface and use error codes to describe errors when needed (e.g., ENOENT for "no such file or directory").[4]

Many of these system calls have an obvious purpose. Here are a few whose common usage may be less obvious:

- **ioctl(2)**: This is commonly used to request miscellaneous actions from the kernel, especially for system administration tools, where another (more obvious) system call isn't suitable. See the example that follows.
- **mmap(2)**: This is commonly used to map executables and libraries to the process address space, and for memory-mapped files. It is sometimes used to allocate the working memory of a process, instead of the brk(2)-based malloc(2), to reduce the syscall rate and improve performance (which doesn't always work due to the trade-off involved: memory-mapping management).
- **brk(2)**: This is used to extend the heap pointer, which defines the size of the working memory of the process. It is typically performed by a system memory allocation library, when a malloc(3) (memory allocate) call cannot be satisfied from the existing space in the heap. See Chapter 7, Memory.
- **futex(2)**: This syscall is used to handle part of a user space lock: the part that is likely to block.

If a system call is unfamiliar, you can learn more in its man page (these are in section 2 of the man pages: syscalls).

The ioctl(2) syscall may be the most difficult to learn, due to its ambiguous nature. As an example of its usage, the Linux perf(1) tool (introduced in Chapter 6, CPUs) performs privileged actions to coordinate performance instrumentation. Instead of system calls being added for each action, a single system call is added: perf_event_open(2), which returns a file descriptor for use with ioctl(2). This ioctl(2) can then be called using different arguments to perform the different desired actions. For example, ioctl(fd, PERF_EVENT_IOC_ENABLE) enables instrumentation. The arguments, in this example PERF_EVENT_IOC_ENABLE, can be more easily added and changed by the developer.

[4] glibc provides these errors in an errno (error number) integer variable.

3.2.4 Interrupts

An *interrupt* is a signal to the processor that some event has occurred that needs processing, and interrupts the current execution of the processor to handle it. It typically causes the processor to enter kernel mode if it isn't already, save the current thread state, and then run an *interrupt service routine* (ISR) to process the event.

There are asynchronous interrupts generated by external hardware and synchronous interrupts generated by software instructions. These are pictured in Figure 3.4.

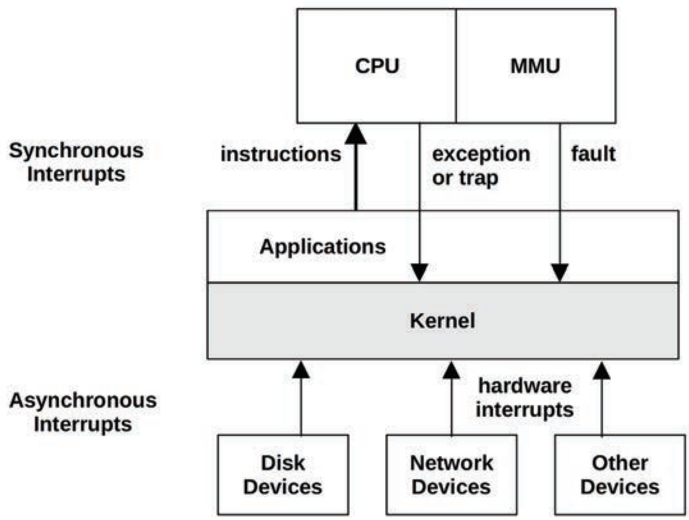


Figure 3.4 Interrupt types

For simplicity Figure 3.4 shows all interrupts sent to the kernel for processing; these are sent to the CPU first, which selects the ISR in the kernel to run the event.

Asynchronous Interrupts

Hardware devices can send *interrupt service requests* (IRQs) to the processor, which arrive asynchronously to the currently running software. Examples of hardware interrupts include:

- Disk devices signaling the completion of disk I/O
- Hardware indicating a failure condition
- Network interfaces signaling the arrival of a packet
- Input devices: keyboard and mouse input

To explain the concept of asynchronous interrupts, an example scenario is pictured in Figure 3.5 showing the passage of time as a database (MySQL) running on CPU 0 reads from a file system. The file system contents must be fetched from disk, so the scheduler context switches to another thread (a Java application) while the database is waiting. Sometime later, the disk I/O completes, but at this point the database is no longer running on CPU 0. The completion interrupt has occurred asynchronously to the database, showed by a dotted line in Figure 3.5.

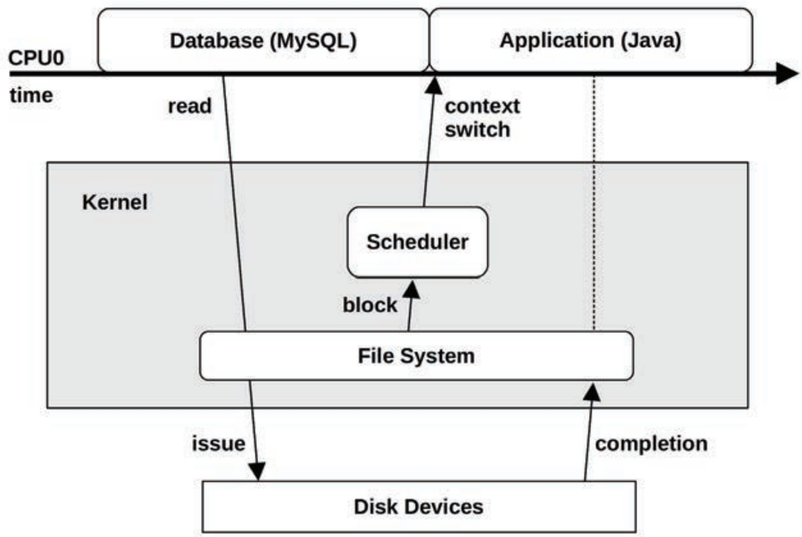


Figure 3.5 Asynchronous interrupt example

Synchronous Interrupts

Synchronous interrupts are generated by software instructions. The following describes different types of software interrupts using the terms *traps*, *exceptions*, and *faults*; however, these terms are often used interchangeably.

- **Traps**: A deliberate call into the kernel, such as by the int (interrupt) instruction. One implementation of syscalls involves calling the int instruction with a vector for a syscall handler (e.g., int 0x80 on Linux x86). int raises a software interrupt.
- **Exceptions**: A exceptional condition, such as by an instruction performing a divide by zero.
- **Faults**: A term often used for memory events, such as *page faults* triggered by accessing a memory location without an MMU mapping. See Chapter 7, Memory.

For these interrupts, the responsible software and instruction are still on CPU.

Interrupt Threads

Interrupt service routines (ISRs) are designed to operate as quickly as possible, to reduce the effects of interrupting active threads. If an interrupt needs to perform more than a little work, especially if it may block on locks, it can be processed by an interrupt thread that can be scheduled by the kernel. This is pictured in Figure 3.6.

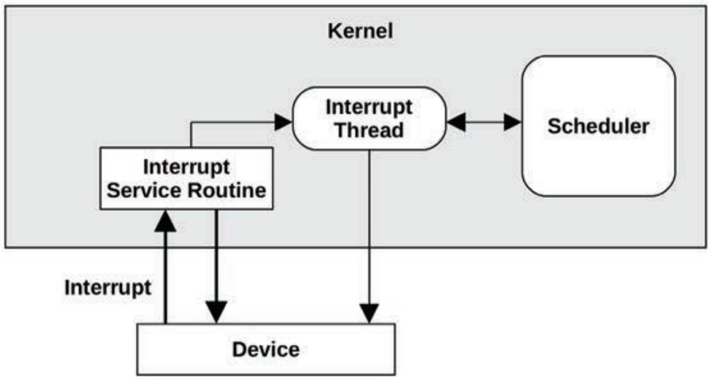


Figure 3.6 Interrupt processing

How this is implemented depends on the kernel version. On Linux, device drivers can be modeled as two halves, with the top half handling the interrupt quickly, and scheduling work to a bottom half to be processed later [Corbet 05]. Handling the interrupt quickly is important as the top half runs in *interrupt-disabled* mode to postpone the delivery of new interrupts, which can cause latency problems for other threads if it runs for too long. The bottom half can be either *tasklets* or *work queues*; the latter are threads that can be scheduled by the kernel and can sleep when necessary.

Linux network drivers, for example, have a top half to handle IRQs for inbound packets, which calls the bottom half to push the packet up the network stack. The bottom half is implemented as a *softirq* (software interrupt).

The time from an interrupt's arrival to when it is serviced is the *interrupt latency*, which is dependent on the hardware and implementation. This is a subject of study for real-time or low-latency systems.

Interrupt Masking

Some code paths in the kernel cannot be interrupted safely. An example is kernel code that acquires a spin lock during a system call, for a spin lock that might also be needed by an interrupt. Taking an interrupt with such a lock held could cause a deadlock. To prevent such a situation, the kernel can temporarily mask interrupts by setting the CPU's *interrupt mask* register. The interrupt disabled time should be as short as possible, as it can perturb the timely execution of applications that are woken up by other interrupts. This is an important factor for *real-time* systems—those that have strict response time requirements. Interrupt disabled time is also a target of performance analysis (such analysis is supported directly by the Ftrace irqsoff tracer, mentioned in Chapter 14, Ftrace).

Some high-priority events should not be ignored, and so are implemented as *non-maskable interrupts* (NMIs). For example, Linux can use an Intelligent Platform Management Interface (IPMI) watchdog timer that checks if the kernel appears to have locked up based on a lack of interrupts during a period of time. If so, the watchdog can issue an NMI interrupt to reboot the system.[5]

3.2.5 Clock and Idle

A core component of the original Unix kernel is the clock() routine, executed from a timer interrupt. It has historically been executed at 60, 100, or 1,000 times per second[6] (often expressed in Hertz: cycles per second), and each execution is called a *tick*.[7] Its functions have included updating the system time, expiring timers and time slices for thread scheduling, maintaining CPU statistics, and executing scheduled kernel routines.

There have been performance issues with the clock, improved in later kernels, including:

- **Tick latency**: For 100 Hertz clocks, up to 10 ms of additional latency may be encountered for a timer as it waits to be processed on the next tick. This has been fixed using high-resolution real-time interrupts so that execution occurs immediately.
- **Tick overhead**: Ticks consume CPU cycles and slightly perturb applications, and are one cause of what is known as *operating system jitter*. Modern processors also have dynamic power features, which can power down parts during idle periods. The clock routine interrupts this idle time, which can consume power needlessly.

Modern kernels have moved much functionality out of the clock routine to on-demand interrupts, in an effort to create a *tickless kernel*. This reduces overhead and improves power efficiency by allowing processors to remain in sleep states for longer.

The Linux clock routine is scheduler_tick(), and Linux has ways to omit calling the clock while there isn't any CPU load. The clock itself typically runs at 250 Hertz (configured by the CONFIG_HZ Kconfig option and variants), and its calls are reduced by the NO_HZ functionality (configured by CONFIG_NO_HZ and variants), which is now commonly enabled [Linux 20a].

Idle Thread

When there is no work for the CPUs to perform, the kernel schedules a placeholder thread that waits for work, called the *idle thread*. A simple implementation would check for the availability of new work in a loop. In modern Linux the *idle task* can call the hlt (halt) instruction to power down the CPU until the next interrupt is received, saving power.

3.2.6 Processes

A process is an environment for executing a user-level program. It consists of a memory address space, file descriptors, thread stacks, and registers. In some ways, a process is like a virtual early computer, where only one program is executing with its own registers and stacks.

---

[5] Linux also has a software NMI watchdog for detecting lockups [Linux 20d].

[6] Other rates include 250 for Linux 2.6.13, 256 for Ultrix, and 1,024 for OSF/1 [Mills 94].

[7] Linux also tracks *jiffies*, a unit of time similar to ticks.

Processes are multitasked by the kernel, which typically supports the execution of thousands of processes on a single system. They are individually identified by their *process ID* (PID), which is a unique numeric identifier.

A process contains one or more *threads*, which operate in the process address space and share the same file descriptors. A thread is an executable context consisting of a stack, registers, and an instruction pointer (also called a *program counter*). Multiple threads allow a single process to execute in parallel across multiple CPUs. On Linux, threads and processes are both *tasks*.

The first process launched by the kernel is called "init," from /sbin/init (by default), with PID 1, which launches user space services. In Unix this involved running start scripts from /etc, a method now referred to as SysV (after Unix System V). Linux distributions now commonly use the systemd software to start services and track their dependencies.

Process Creation

Processes are normally created using the fork(2) system call on Unix systems. On Linux, C libraries typically implement the fork function by wrapping around the versatile clone(2) syscall. These syscalls create a duplicate of the process, with its own process ID. The exec(2) system call (or a variant, such as execve(2)) can then be called to begin execution of a different program.

Figure 3.7 shows an example process creation for a bash shell (`bash`) executing the `ls` command.

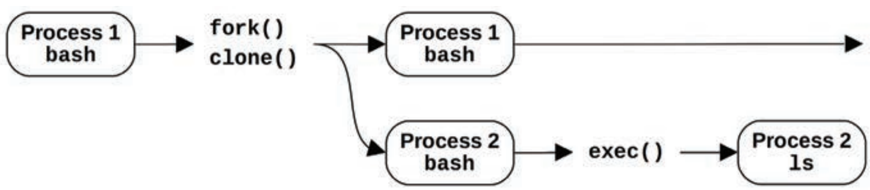


Figure 3.7 Process creation

The fork(2) or clone(2) syscall may use a copy-on-write (COW) strategy to improve performance. This adds references to the previous address space rather than copying all of the contents. Once either process modifies the multiple-referenced memory, a separate copy is then made for the modifications. This strategy either defers or eliminates the need to copy memory, reducing memory and CPU usage.

Process Life Cycle

The life cycle of a process is shown in Figure 3.8. This is a simplified diagram; for modern multithreaded operating systems it is the threads that are scheduled and run, and there are some additional implementation details regarding how these map to process states (see Figures 5.6 and 5.7 in Chapter 5 for more detailed diagrams).

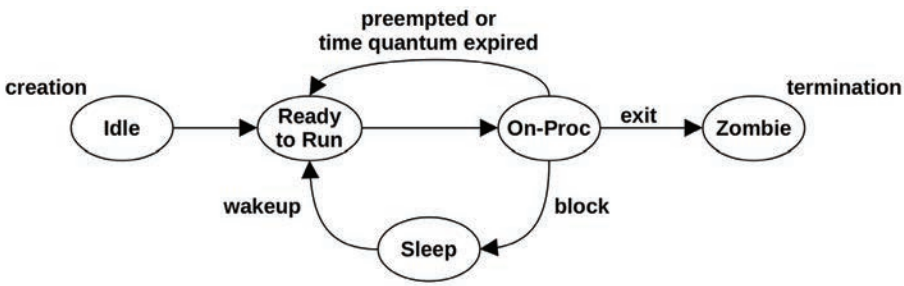


Figure 3.8 Process life cycle

The on-proc state is for running on a processor (CPU). The ready-to-run state is when the process is runnable but is waiting on a CPU run queue for its turn on a CPU. Most I/O will block, putting the process in the sleep state until the I/O completes and the process is woken up. The zombie state occurs during process termination, when the process waits until its process status has been reaped by the parent process or until it is removed by the kernel.

Process Environment

The process environment is shown in Figure 3.9; it consists of data in the address space of the process and metadata (context) in the kernel.

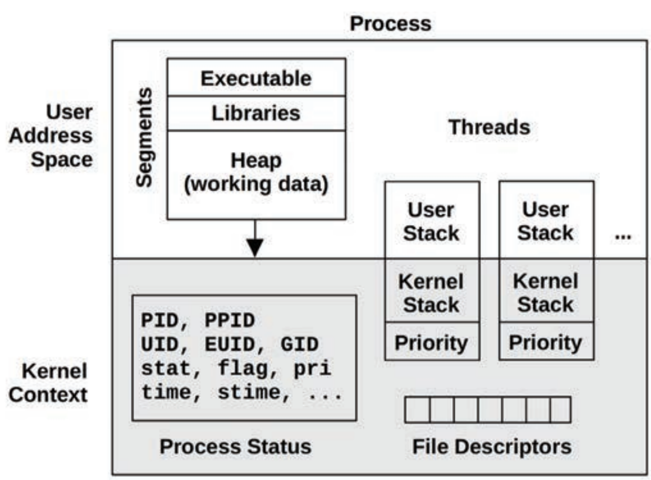


Figure 3.9 Process environment

The kernel context consists of various process properties and statistics: its process ID (PID), the owner's user ID (UID), and various times. These are commonly examined via the ps(1) and top(1) commands. It also has a set of file descriptors, which refer to open files and which are (usually) shared between threads.

This example pictures two threads, each containing some metadata, including a priority in kernel context[8] and user stack in the user address space. The diagram is not drawn to scale; the kernel context is very small compared to the process address space.

The user address space contains memory segments of the process: executable, libraries, and heap. For more details, see Chapter 7, Memory.

On Linux, each thread has its own user stack and a kernel exception stack[9] [Owens 20].

3.2.7 Stacks

A stack is a memory storage area for temporary data, organized as a last-in, first-out (LIFO) list. It is used to store less important data than that which fits in the CPU register set. When a function is called, the return address is saved to the stack. Some registers may be saved to the stack as well if their values are needed after the call.[10] When the called function has finished, it restores any required registers and, by fetching the return address from the stack, passes execution to the calling function. The stack can also be used for passing parameters to functions. The set of data on a stack related to a function's execution is called a *stack frame*.

The call path to the currently executing function can be seen by examining the saved return addresses across all the stack frames in the thread's stack (a process called *stack walking*).[11] This call path is referred to as a *stack back trace* or a *stack trace*. In performance engineering it is often called just a "stack" for short. These stacks can answer *why* something is executing, and are an invaluable tool for debugging and performance analysis.

How to Read a Stack

The following example kernel stack (from Linux) shows the path taken for TCP transmission, as printed by a tracing tool:

```
tcp_sendmsg+1
sock_sendmsg+62
SYSC_sendto+319
sys_sendto+14
do_syscall_64+115
entry_SYSCALL_64_after_hwframe+61
```

[8] The kernel context may be its own full address space (as with SPARC processors) or a restricted range that does not overlap with user addresses (as with x86 processors).

[9] There are also special-purpose kernel stacks per-CPU, including those used for interrupts.

[10] The calling convention from the processor ABI specifies which registers should retain their values after a function call (they are *non-volatile*) and are saved to the stack by the called function ("callee-saves"). Other registers are *volatile* and may be clobbered by the called function; if the caller wishes to retain their values, it must save them to the stack ("caller-saves").

[11] For more detail on stack walking and the different possible techniques (which include: frame-pointer based, debuginfo, last branch record, and ORC) see Chapter 2, Tech, Section 2.4, Stack Trace Walking, of *BPF Performance Tools* [Gregg 19].

Stacks are usually printed in leaf-to-root order, so the first line printed is the function currently executing, and beneath it is its parent, then its grandparent, and so on. In this example, the tcp_sendmsg() function was executing, called by sock_sendmsg(). In this stack example, to the right of the function name is the instruction offset, showing the location within a function. The first line shows tcp_sendmsg() offset 1 (which would be the second instruction), called by sock_sendmsg() offset 62. This offset is only useful if you desire a low-level understanding of the code path taken, down to the instruction level.

By reading down the stack, the full ancestry can be seen: function, parent, grandparent, and so on. Or, by reading bottom-up, you can follow the path of execution to the current function: how we got here.

Since stacks expose the internal path taken through source code, there is typically no documentation for these functions other than the code itself. For this example stack, this is the Linux kernel source code. An exception to this is where functions are part of an API and are documented.

User and Kernel Stacks

While executing a system call, a process thread has two stacks: a user-level stack and a kernel-level stack. Their scope is pictured in Figure 3.10.

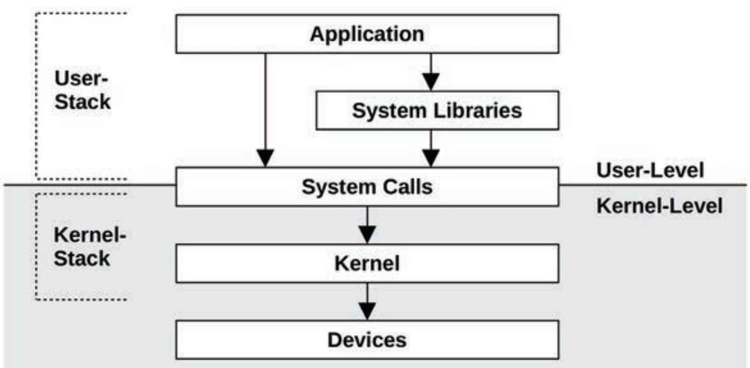


Figure 3.10 User and kernel stacks

The user-level stack of the blocked thread does not change for the duration of a system call, as the thread is using a separate kernel-level stack while executing in kernel context. (An exception to this may be signal handlers, which may borrow a user-level stack depending on their configuration.)

On Linux, there are multiple kernel stacks for different purposes. Syscalls use a kernel exception stack associated with each thread, and there are also stacks associated with soft and hard interrupts (IRQs) [Bovet 05].

3.2.8 Virtual Memory

Virtual memory is an abstraction of main memory, providing processes and the kernel with their own, almost infinite,[12] private view of main memory. It supports multitasking, allowing processes and the kernel to operate on their own private address spaces without worrying about contention. It also supports oversubscription of main memory, allowing the operating system to transparently map virtual memory between main memory and secondary storage (disks) as needed.

The role of virtual memory is shown in Figure 3.11. Primary memory is main memory (RAM), and secondary memory is the storage devices (disks).

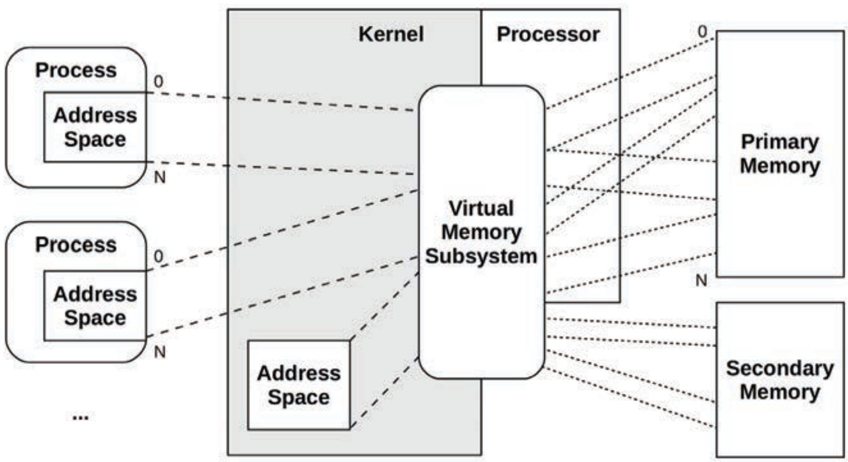


Figure 3.11 Virtual memory address spaces[13]

Virtual memory is made possible by support in both the processor and operating system. It is not real memory, and most operating systems map virtual memory to real memory only on demand, when the memory is first populated (written).

See Chapter 7, Memory, for more about virtual memory.

Memory Management

While virtual memory allows main memory to be extended using secondary storage, the kernel strives to keep the most active data in main memory. There are two kernel schemes for this:

- **Process swapping** moves entire processes between main memory and secondary storage.
- **Paging** moves small units of memory called *pages* (e.g., 4 Kbytes).

[12] On 64-bit processors, anyway. For 32-bit processors, virtual memory is limited to 4 Gbytes due to the limits of a 32-bit address (and the kernel may limit it to an even smaller amount).

[13] Process virtual memory is shown as starting from 0 as a simplification. Kernels today commonly begin a process's virtual address space at some offset such as 0x10000 or a random address. One benefit is that a common programming error of dereferencing a NULL (0) pointer will then cause the program to crash (SIGSEGV) as the 0 address is invalid. This is generally preferable to dereferencing data at address 0 by mistake, as the program would continue to run with corrupt data.

Process swapping is the original Unix method and can cause severe performance loss. Paging is more efficient and was added to BSD with the introduction of paged virtual memory. In both cases, least recently used (or not recently used) memory is moved to secondary storage and moved back to main memory only when needed again.

In Linux, the term *swapping* is used to refer to *paging*. The Linux kernel does not support the (older) Unix-style process swapping of entire threads and processes.

For more on paging and swapping, see Chapter 7, Memory.

3.2.9 Schedulers

Unix and its derivatives are time-sharing systems, allowing multiple processes to run at the same time by dividing execution time among them. The scheduling of processes on processors and individual CPUs is performed by the *scheduler*, a key component of the operating system kernel. The role of the scheduler is pictured in Figure 3.12, which shows that the scheduler operates on threads (in Linux, *tasks*), mapping them to CPUs.

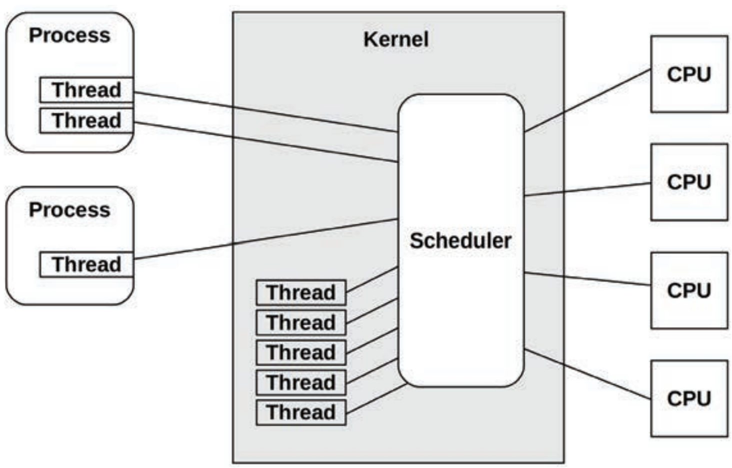


Figure 3.12 Kernel scheduler

The basic intent is to divide CPU time among the active processes and threads, and to maintain a notion of *priority* so that more important work can execute sooner. The scheduler keeps track of all threads in the ready-to-run state, traditionally on per-priority queues called *run queues* [Bach 86]. Modern kernels may implement these queues per CPU and may also use other data structures, apart from queues, to track the threads. When more threads want to run than there are available CPUs, the lower-priority threads wait their turn. Most kernel threads run with a higher priority than user-level processes.

Process priority can be modified dynamically by the scheduler to improve the performance of certain workloads. Workloads can be categorized as either:

- **CPU-bound**: Applications that perform heavy compute, for example, scientific and mathematical analysis, which are expected to have long runtimes (seconds, minutes, hours, days, or even longer). These become limited by CPU resources.
- **I/O-bound**: Applications that perform I/O, with little compute, for example, web servers, file servers, and interactive shells, where low-latency responses are desirable. When their load increases, they are limited by I/O to storage or network resources.

A commonly used scheduling policy dating back to UNIX identifies CPU-bound workloads and decreases their priority, allowing I/O-bound workloads—where low-latency responses are more desirable—to run sooner. This can be achieved by calculating the ratio of recent compute time (time executing on-CPU) to real time (elapsed time) and decreasing the priority of processes with a high (compute) ratio [Thompson 78]. This mechanism gives preference to shorter-running processes, which are usually those performing I/O, including human interactive processes.

Modern kernels support multiple *scheduling classes* or *scheduling policies* (Linux) that apply different algorithms for managing priority and runnable threads. These may include *real-time scheduling*, which uses a priority higher than all noncritical work, including kernel threads. Along with preemption support (described later), real-time scheduling provides predictable and low-latency scheduling for systems that require it.

See Chapter 6, CPUs, for more about the kernel scheduler and other scheduling algorithms.

3.2.10 File Systems

File systems are an organization of data as files and directories. They have a file-based interface for accessing them, usually based on the POSIX standard. Kernels support multiple file system types and instances. Providing a file system is one of the most important roles of the operating system, once described as *the* most important role [Ritchie 74].

The operating system provides a global file namespace, organized as a top-down tree topology starting with the root level ("/"). File systems join the tree by *mounting*, attaching their own tree to a directory (the *mount point*). This allows the end user to navigate the file namespace transparently, regardless of the underlying file system type.

A typical operating system may be organized as shown in Figure 3.13.

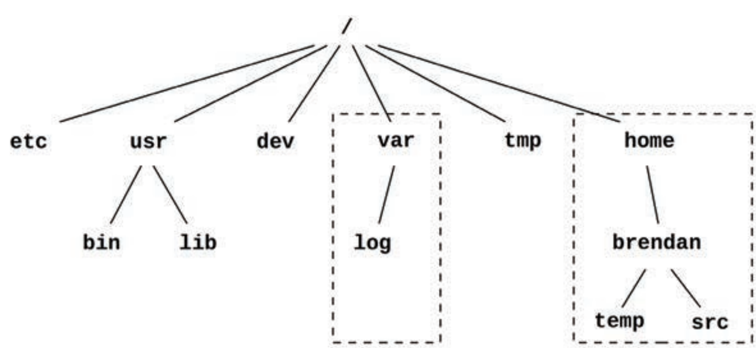


Figure 3.13 Operating system file hierarchy

The top-level directories include `etc` for system configuration files, `usr` for system-supplied user-level programs and libraries, `dev` for device nodes, `var` for varying files including system logs, `tmp` for temporary files, and `home` for user home directories. In the example pictured, `var` and `home` may reside on their own file system instances and separate storage devices; however, they can be accessed like any other component of the tree.

Most file system types use storage devices (disks) to store their contents. Some file system types are dynamically created by the kernel, such as `/proc` and `/dev`.

Kernels typically provide different ways to isolate processes to a portion of the file namespace, including chroot(8), and, on Linux, mount namespaces, commonly used for containers (see Chapter 11, Cloud Computing).

VFS

The virtual file system (VFS) is a kernel interface to abstract file system types, originally developed by Sun Microsystems so that the Unix file system (UFS) and the Network file system (NFS) could more easily coexist. Its role is pictured in Figure 3.14.

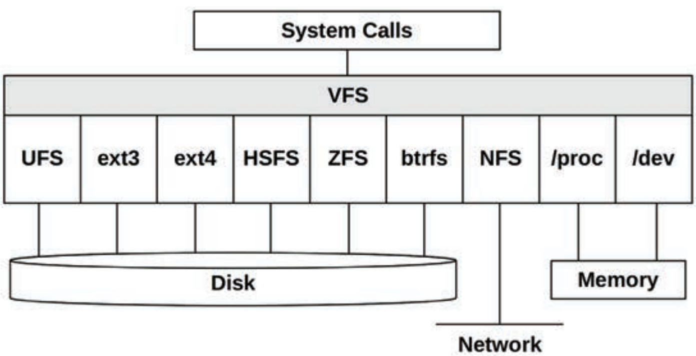


Figure 3.14 Virtual file system

The VFS interface makes it easier to add new file system types to the kernel. It also supports providing the global file namespace, pictured earlier, so that user programs and applications can access various file system types transparently.

I/O Stack

For storage-device-based file systems, the path from user-level software to the storage device is called the *I/O stack*. This is a subset of the entire software stack shown earlier. A generic I/O stack is shown in Figure 3.15.

Figure 3.15 shows a direct path to block devices on the left, bypassing the file system. This path is sometimes used by administrative tools and databases.

File systems and their performance are covered in detail in Chapter 8, File Systems, and the storage devices they are built upon are covered in Chapter 9, Disks.

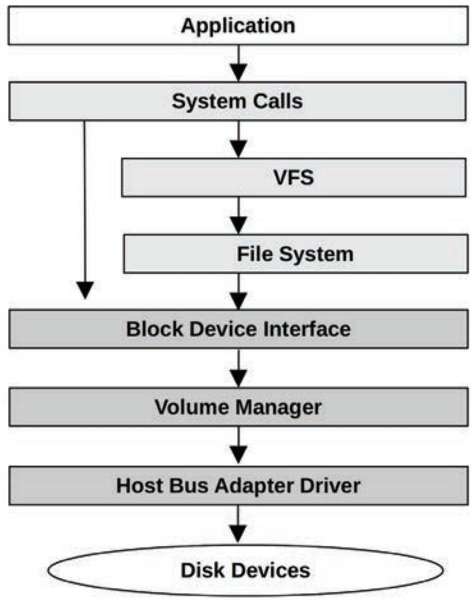


Figure 3.15 Generic I/O stack

3.2.11 Caching

Since disk I/O has historically had high latency, many layers of the software stack attempt to avoid it by caching reads and buffering writes. Caches may include those shown in Table 3.2 (in the order in which they are checked).

Table 3.2 **Example cache layers for disk I/O**

	Cache	Examples
1	Client cache	Web browser cache
2	Application cache	—
3	Web server cache	Apache cache
4	Caching server	memcached
5	Database cache	MySQL buffer cache
6	Directory cache	dcache
7	File metadata cache	inode cache
8	Operating system buffer cache	Buffer cache
9	File system primary cache	Page cache, ZFS ARC
10	File system secondary cache	ZFS L2ARC
11	Device cache	ZFS vdev

	Cache	Examples
12	Block cache	Buffer cache
13	Disk controller cache	RAID card cache
14	Storage array cache	—
15	On-disk cache	—

For example, the buffer cache is an area of main memory that stores recently used disk blocks. Disk reads may be served immediately from the cache if the requested block is present, avoiding the high latency of disk I/O.

The types of caches present will vary based on the system and environment.

3.2.12 Networking

Modern kernels provide a stack of built-in network protocols, allowing the system to communicate via the network and take part in distributed system environments. This is referred to as the *networking stack* or the *TCP/IP stack*, after the commonly used TCP and IP protocols. User-level applications access the network through programmable endpoints called *sockets*.

The physical device that connects to the network is the *network interface* and is usually provided on a *network interface card* (NIC). A historical duty of the system administrator was to associate an IP address with a network interface, so that it can communicate with the network; these mappings are now typically automated via the dynamic host configuration protocol (DHCP).

Network protocols do not change often, but there is a new transport protocol seeing growing adoption: QUIC (summarized in Chapter 10, Network). Protocol enhancements and options change more often, such as newer TCP options and TCP congestion control algorithms. Newer protocols and enhancements typically require kernel support (with the exception of user-space protocol implementations). Another change is support for different network interface cards, which require new device drivers for the kernel.

For more on networking and network performance, see Chapter 10, Network.

3.2.13 Device Drivers

A kernel must communicate with a wide variety of physical devices. Such communication is achieved using *device drivers*: kernel software for device management and I/O. Device drivers are often provided by the vendors who develop the hardware devices. Some kernels support *pluggable* device drivers, which can be loaded and unloaded without requiring a system restart.

Device drivers can provide *character* and/or *block* interfaces to their devices. Character devices, also called *raw devices*, provide unbuffered sequential access of any I/O size down to a single character, depending on the device. Such devices include keyboards and serial ports (and in original Unix, paper tape and line printer devices).

Block devices perform I/O in units of blocks, which have historically been 512 bytes each. These can be accessed randomly based on their block offset, which begins at 0 at the start of the block device. In original Unix, the block device interface also provided caching of block device buffers to improve performance, in an area of main memory called the *buffer cache*. In Linux, this buffer cache is now part of the page cache.

3.2.14 Multiprocessor

Multiprocessor support allows the operating system to use multiple CPU instances to execute work in parallel. It is usually implemented as *symmetric multiprocessing* (SMP) where all CPUs are treated equally. This was technically difficult to accomplish, posing problems for accessing and sharing memory and CPUs among threads running in parallel. On multiprocessor systems there may also be banks of main memory connected to different sockets (physical processors) in a *non-uniform memory access* (NUMA) architecture, which also pose performance challenges. See Chapter 6, CPUs, for details, including scheduling and thread synchronization, and Chapter 7, Memory, for details on memory access and architectures.

IPIs

For a multiprocessor system, there are times when CPUs need to coordinate, such as for cache coherency of memory translation entries (informing other CPUs that an entry, if cached, is now stale). A CPU can request other CPUs, or all CPUs, to immediately perform such work using an inter-processor interrupt (IPI) (also known as an *SMP call* or a *CPU cross call*). IPIs are processor interrupts designed to be executed quickly, to minimize interruption of other threads.

IPIs can also be used by preemption.

3.2.15 Preemption

Kernel preemption support allows high-priority user-level threads to interrupt the kernel and execute. This enables real-time systems that can execute work within a given time constraint, including systems in use by aircraft and medical devices. A kernel that supports preemption is said to be *fully preemptible*, although practically it will still have some small critical code paths that cannot be interrupted.

Another approach supported by Linux is *voluntary kernel preemption*, where logical stopping points in the kernel code can check and perform preemption. This avoids some of the complexity of supporting a fully preemptive kernel and provides low-latency preemption for common workloads. Voluntary kernel preemption is commonly enabled in Linux via the CONFIG_PREEMPT_VOLUNTARY Kconfig option; there is also CONFIG_PREEMPT to allow all kernel code (except critical sections) to be preemptible, and CONFIG_PREEMPT_NONE to disable preemption, improving throughput at the cost of higher latencies.

3.2.16 Resource Management

The operating system may provide various configurable controls for fine-tuning access to system resources, such as CPUs, memory, disk, and the network. These are *resource controls* and can be used to manage performance on systems that run different applications or host multiple tenants (cloud computing). Such controls may impose fixed limits per process (or groups of processes) for resource usage, or a more flexible approach—allowing spare usage to be shared among them.

Early versions of Unix and BSD had basic per-process resource controls, including CPU priorities with nice(1), and some resource limits with ulimit(1).

For Linux, control groups (cgroups) have been developed and integrated in Linux 2.6.24 (2008), and various additional controls have been added since then. These are documented in the kernel source under Documentation/cgroups. There is also an improved unified hierarchical scheme called *cgroup v2*, made available in Linux 4.5 (2016) and documented in Documentation/admin-guide/cgroup-v2.rst.

Specific resource controls are mentioned in later chapters as appropriate. An example use case is described in Chapter 11, Cloud Computing, for managing the performance of OS-virtualized tenants.

3.2.17 Observability

The operating system consists of the kernel, libraries, and programs. These programs include tools to observe system activity and analyze performance, typically installed in /usr/bin and /usr/sbin. Third-party tools may also be installed on the system to provide additional observability.

Observability tools, and the operating system components upon which they are built, are introduced in Chapter 4.

3.3 Kernels

The following sections discuss Unix-like kernel implementation details with a focus on performance. As background, the performance features of earlier kernels are discussed: Unix, BSD, and Solaris. The Linux kernel is discussed in more detail in Section 3.4, Linux.

Kernel differences can include the file systems they support (see Chapter 8, File Systems), the system call (syscall) interfaces, network stack architecture, real-time support, and scheduling algorithms for CPUs, disk I/O, and networking.

Table 3.3 shows Linux and other kernel versions for comparison, with syscall counts based on the number of entries in section 2 of the OS man pages. This is a crude comparison, but enough to see some differences.

Table 3.3 **Kernel versions with documented syscall counts**

Kernel Version	Syscalls
UNIX Version 7	48
SunOS (Solaris) 5.11	142
FreeBSD 12.0	222
Linux 2.6.32-21-server	408
Linux 2.6.32-220.el6.x86_64	427
Linux 3.2.6-3.fc16.x86_64	431
Linux 4.15.0-66-generic	480
Linux 5.3.0-1010-aws	493

These are just the syscalls with documentation; more are usually provided by the kernel for private use by operating system software.

> UNIX had twenty system calls at the very first, and today Linux—which is a direct descendant—has over a thousand . . . I just worry about the complexity and the size of things that grow.
>
> Ken Thompson, ACM Turing Centenary Celebration, 2012

Linux is growing in complexity and exposing this complexity to user-land by adding new system calls or through other kernel interfaces. Extra complexity makes learning, programming, and debugging more time-consuming.

3.3.1 Unix

Unix was developed by Ken Thompson, Dennis Ritchie, and others at AT&T Bell Labs during 1969 and the years that followed. Its exact origin was described in *The UNIX Time-Sharing System* [Ritchie 74]:

> The first version was written when one of us (Thompson), dissatisfied with the available computer facilities, discovered a little-used PDP-7 and set out to create a more hospitable environment.

The developers of UNIX had previously worked on the Multiplexed Information and Computer Services (Multics) operating system. UNIX was developed as a *lightweight* multitasked operating system and kernel, originally named UNiplexed Information and Computing Service (UNICS), as a pun on Multics. From *UNIX Implementation* [Thompson 78]:

> The kernel is the only UNIX code that cannot be substituted by a user to his own liking. For this reason, the kernel should make as few real decisions as possible. This does not mean to allow the user a million options to do the same thing. Rather, it means to allow only one way to do one thing, but have that way be the least-common divisor of all the options that might have been provided.

While the kernel was small, it did provide some features for high performance. Processes had scheduler priorities, reducing run-queue latency for higher-priority work. Disk I/O was performed in large (512-byte) blocks for efficiency and cached in an in-memory per-device buffer cache. Idle processes could be swapped out to storage, allowing busier processes to run in main memory. And the system was, of course, multitasking—allowing multiple processes to run concurrently, improving job throughput.

To support networking, multiple file systems, paging, and other features we now consider standard, the kernel had to grow. And with multiple derivatives, including BSD, SunOS (Solaris), and later Linux, kernel performance became competitive, which drove the addition of more features and code.

3.3.2 BSD

The Berkeley Software Distribution (BSD) OS began as enhancements to Unix 6th Edition at the University of California, Berkeley, and was first released in 1978. As the original Unix code required an AT&T software license, by the early 1990s this Unix code had been rewritten in BSD under a new BSD license, allowing free distributions including FreeBSD.

Major BSD kernel developments, especially performance-related, include:

- **Paged virtual memory**: BSD brought paged virtual memory to Unix: instead of swapping out entire processes to free main memory, smaller least-recently-used chunks of memory could be moved (paged). See Chapter 7, Memory, Section 7.2.2, Paging.
- **Demand paging**: This defers the mapping of physical memory to virtual memory to when it is first written, avoiding an early and sometimes unnecessary performance and memory cost for pages that may never be used. Demand paging was brought to Unix by BSD. See Chapter 7, Memory, Section 7.2.2, Paging.
- **FFS**: The Berkeley Fast File System (FFS) grouped disk allocation into cylinder groups, greatly reducing fragmentation and improving performance on rotational disks, as well as supporting larger disks and other enhancements. FFS became the basis for many other file systems, including UFS. See Chapter 8, File Systems, Section 8.4.5, File System Types.
- **TCP/IP network stack**: BSD developed the first high-performance TCP/IP network stack for Unix, included in 4.2BSD (1983). BSD is still known for its performant network stack.
- **Sockets**: Berkeley sockets are an API for connection endpoints. Included in 4.2BSD, they have become a standard for networking. See Chapter 10, Network.
- **Jails**: Lightweight OS-level virtualization, allowing multiple guests to share one kernel. Jails were first released in FreeBSD 4.0.
- **Kernel TLS**: As transport layer security (TLS) is now commonly used on the Internet, kernel TLS moves much of TLS processing to the kernel, improving performance[14] [Stewart 15].

While not as popular as Linux, BSD is used for some performance-critical environments, including for the Netflix content delivery network (CDN), as well as file servers from NetApp, Isilon, and others. Netflix summarized FreeBSD performance on its CDN in 2019 as [Looney 19]:

> "Using FreeBSD and commodity parts, we achieve 90 Gb/s serving TLS-encrypted connections with ~55% CPU on a 16-core 2.6-GHz CPU."

There is an excellent reference on the internals of FreeBSD, from the same publisher that brings you this book: *The Design and Implementation of the FreeBSD Operating System,* 2nd Edition [McKusick 15].

[14] Developed to improve the performance of the Netflix FreeBSD open connect appliances (OCAs) that are the Netflix CDN.

3.3.3 Solaris

Solaris is a Unix and BSD-derived kernel and OS created by Sun Microsystems in 1982. It was originally named SunOS and optimized for Sun workstations. By the late 1980s, AT&T developed a new Unix standard, Unix System V Release 4 (SVR4) based on technologies from SVR3, SunOS, BSD, and Xenix. Sun created a new kernel based on SVR4, and rebranded the OS under the name Solaris.

Major Solaris kernel developments, especially performance-related, include:

- **VFS**: The virtual file system (VFS) is an abstraction and interface that allows multiple file systems to easily coexist. Sun initially created it so that NFS and UFS could coexist. VFS is covered in Chapter 8, File Systems.
- **Fully preemptible kernel**: This provided low latency for high-priority work, including real-time work.
- **Multiprocessor support**: In the early 1990s, Sun invested heavily in multiprocessor operating system support, developing kernel support for both asymmetric and symmetric multiprocessing (ASMP and SMP) [Mauro 01].
- **Slab allocator**: Replacing the SVR4 buddy allocator, the kernel slab memory allocator provided better performance via per-CPU caches of preallocated buffers that could be quickly reused. This allocator type, and its derivatives, has become the standard for kernels including Linux.
- **DTrace**: A static and dynamic tracing framework and tool providing virtually unlimited observability of the entire software stack, in real time and in production. Linux has BPF and bpftrace for this type of observability.
- **Zones**: An OS-based virtualization technology for creating OS instances that share one kernel, similar to the earlier FreeBSD jails technology. OS virtualization is now in widespread use as Linux containers. See Chapter 11, Cloud Computing.
- **ZFS**: A file system with enterprise-level features and performance. It is now available for other OSes, including Linux. See Chapter 8, File Systems.

Oracle purchased Sun Microsystems in 2010, and Solaris is now called Oracle Solaris. Solaris is covered in more detail in the first edition of this book.

3.4 Linux

Linux was created in 1991 by Linus Torvalds as a free operating system for Intel personal computers. He announced the project in a Usenet post:

> I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since April, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

This refers to the MINIX operating system, which was being developed as a free and small (mini) version of Unix for small computers. BSD was also aiming to provide a free Unix version although at the time it had legal troubles.

The Linux kernel was developed taking general ideas from many ancestors, including:

- **Unix (and Multics)**: Operating system layers, system calls, multitasking, processes, process priorities, virtual memory, global file system, file system permissions, device nodes, buffer cache
- **BSD**: Paged virtual memory, demand paging, fast file system (FFS), TCP/IP network stack, sockets
- **Solaris**: VFS, NFS, page cache, unified page cache, slab allocator
- **Plan 9**: Resource forks (rfork), for creating different levels of sharing between processes and threads (*tasks*)

Linux now sees widespread use for servers, cloud instances, and embedded devices including mobile phones.

3.4.1 Linux Kernel Developments

Linux kernel developments, especially those related to performance, include the following (many of these descriptions include the Linux kernel version where they were first introduced):

- **CPU scheduling classes**: Various advanced CPU scheduling algorithms have been developed, including scheduling domains (2.6.7) to make better decisions regarding non-uniform memory access (NUMA). See Chapter 6, CPUs.
- **I/O scheduling classes**: Different block I/O scheduling algorithms have been developed, including deadline (2.5.39), anticipatory (2.5.75), and completely fair queueing (CFQ) (2.6.6). These are available in kernels up to Linux 5.0, which removed them to support only newer multi-queue I/O schedulers. See Chapter 9, Disks.
- **TCP congestion algorithms**: Linux allows different TCP congestion control algorithms to be configured, and supports Reno, Cubic, and more in later kernels mentioned in this list. See also Chapter 10, Network.
- **Overcommit**: Along with the out-of-memory (OOM) killer, this is a strategy for doing more with less main memory. See Chapter 7, Memory.
- **Futex** (2.5.7): Short for *fast user-space mutex*, this is used to provide high-performing user-level synchronization primitives.
- **Huge pages** (2.5.36): This provides support for preallocated large memory pages by the kernel and the memory management unit (MMU). See Chapter 7, Memory.
- **OProfile** (2.5.43): A system profiler for studying CPU usage and other events, for both the kernel and applications.
- **RCU** (2.5.43): The kernel provides a read-copy update synchronization mechanism that allows multiple reads to occur concurrently with updates, improving performance and scalability for data that is mostly read.
- **epoll** (2.5.46): A system call for efficiently waiting for I/O across many open file descriptors, which improves the performance of server applications.

- **Modular I/O scheduling** (2.6.10): Linux provides pluggable scheduling algorithms for scheduling block device I/O. See Chapter 9, Disks.
- **DebugFS** (2.6.11): A simple unstructured interface for the kernel to expose data to user level, which is used by some performance tools.
- **Cpusets** (2.6.12): exclusive CPU grouping for processes.
- **Voluntary kernel preemption** (2.6.13): This process provides low-latency scheduling without the complexity of full preemption.
- **inotify** (2.6.13): A framework for monitoring file system events.
- **blktrace** (2.6.17): A framework and tool for tracing block I/O events (later migrated into tracepoints).
- **splice** (2.6.17): A system call to move data quickly between file descriptors and pipes, without a trip through user-space. (The sendfile(2) syscall, which efficiently moves data between file descriptors, is now a wrapper to splice(2).)
- **Delay accounting** (2.6.18): Tracks per-task delay states. See Chapter 4, Observability Tools.
- **IO accounting** (2.6.20): Measures various storage I/O statistics per process.
- **DynTicks** (2.6.21): Dynamic ticks allow the kernel timer interrupt (clock) to not fire during idle, saving CPU resources and power.
- **SLUB** (2.6.22): A new and simplified version of the slab memory allocator.
- **CFS** (2.6.23): Completely fair scheduler. See Chapter 6, CPUs.
- **cgroups** (2.6.24): Control groups allow resource usage to be measured and limited for groups of processes.
- **TCP LRO** (2.6.24): TCP Large Receive Offload (LRO) allows network drivers and hardware to aggregate packets into larger sizes before sending them to the network stack. Linux also supports Large Send Offload (LSO) for the send path.
- **latencytop** (2.6.25): Instrumentation and a tool for observing sources of latency in the operating system.
- **Tracepoints** (2.6.28): Static kernel tracepoints (aka *static probes*) that instrument logical execution points in the kernel, for use by tracing tools (previously called *kernel markers*). Tracing tools are introduced in Chapter 4, Observability Tools.
- **perf** (2.6.31): Linux Performance Events (perf) is a set of tools for performance observability, including CPU performance counter profiling and static and dynamic tracing. See Chapter 6, CPUs, for an introduction.
- **No BKL** (2.6.37): Final removal of the big kernel lock (BKL) performance bottleneck.
- **Transparent huge pages** (2.6.38): This is a framework to allow easy use of huge (large) memory pages. See Chapter 7, Memory.
- **KVM**: The Kernel-based Virtual Machine (KVM) technology was developed for Linux by Qumranet, which was purchased by Red Hat in 2008. KVM allows virtual operating system instances to be created, running their own kernel. See Chapter 11, Cloud Computing.

- **BPF JIT** (3.0): A Just-In-Time (JIT) compiler for the Berkeley Packet Filter (BPF) to improve packet filtering performance by compiling BPF bytecode to native instructions.
- **CFS bandwidth control** (3.2): A CPU scheduling algorithm that supports CPU quotas and throttling.
- **TCP anti-bufferbloat** (3.3+): Various enhancements were made from Linux 3.3 onwards to combat the bufferbloat problem, including Byte Queue Limits (BQL) for the transmission of packet data (3.3), CoDel queue management (3.5), TCP small queues (3.6), and the Proportional Integral controller Enhanced (PIE) packet scheduler (3.14).
- **uprobes** (3.5): The infrastructure for dynamic tracing of user-level software, used by other tools (perf, SystemTap, etc.).
- **TCP early retransmit** (3.5): RFC 5827 for reducing duplicate acknowledgments required to trigger fast retransmit.
- **TFO** (3.6, 3.7, 3.13): TCP Fast Open (TFO) can reduce the TCP three-way handshake to a single SYN packet with a TFO cookie, improving performance. It was made the default in 3.13.
- **NUMA balancing** (3.8+): This added ways for the kernel to automatically balance memory locations on multi-NUMA systems, reducing CPU interconnect traffic and improving performance.
- **SO_REUSEPORT** (3.9): A socket option to allow multiple listener sockets to bind to the same port, improving multi-threaded scalability.
- **SSD cache devices** (3.9): Device mapper support for an SSD device to be used as a cache for a slower rotating disk.
- **bcache** (3.10): An SSD cache technology for the block interface.
- **TCP TLP** (3.10): TCP Tail Loss Probe (TLP) is a scheme to avoid costly timer-based retransmits by sending new data or the last unacknowledged segment after a shorter probe timeout, to trigger faster recovery.
- **NO_HZ_FULL** (3.10, 3.12): Also known as *timerless multitasking* or a *tickless kernel*, this allows non-idle threads to run without clock ticks, avoiding workload perturbations [Corbet 13a].
- **Multiqueue block I/O** (3.13): This provides per-CPU I/O submission queues rather than a single request queue, improving scalability especially for high IOPS SSD devices [Corbet 13b].
- **SCHED_DEADLINE** (3.14): An optional scheduling policy that implements earliest deadline first (EDF) scheduling [Linux 20b].
- **TCP autocorking** (3.14): This allows the kernel to coalesce small writes, reducing the sent packets. An automatic version of the TCP_CORK setsockopt(2).
- **MCS locks and qspinlocks** (3.15): Efficient kernel locks, using techniques such as per-CPU structures. MCS is named after the original lock inventors (Mellor-Crummey and Scott) [Mellor-Crummey 91][Corbet 14].

- **Extended BPF** (3.18+): An in-kernel execution environment for running secure kernel-mode programs. The bulk of extended BPF was added in the 4.x series. Support for attached to kprobes was added in 3.19, to tracepoints in 4.7, to software and hardware events in 4.9, and to cgroups in 4.10. Bounded loops were added in 5.3, which also increased the instruction limit to allow complex applications. See Section 3.4.4, Extended BPF.
- **Overlayfs** (3.18): A union mount file system included in Linux. It creates virtual file systems on top of others, which can also be modified without changing the first. Often used for containers.
- **DCTCP** (3.18): The Data Center TCP (DCTCP) congestion control algorithm, which aims to provide high burst tolerance, low latency, and high throughput [Borkmann 14a].
- **DAX** (4.0): Direct Access (DAX) allows user space to read from persistent-memory storage devices directly, without buffer overheads. ext4 can use DAX.
- **Queued spinlocks** (4.2): Offering better performance under contention, these became the default spinlock kernel implementation in 4.2.
- **TCP lockless listener** (4.4): The TCP listener fast path became lockless, improving performance.
- **cgroup v2** (4.5, 4.15): A unified hierarchy for cgroups was in earlier kernels, and considered stable and exposed in 4.5, named cgroup v2 [Heo 15]. The cgroup v2 CPU controller was added in 4.15.
- **epoll scalability** (4.5): For multithreaded scalability, epoll(7) avoids waking up all threads that are waiting on the same file descriptors for each event, which caused a thundering-herd performance issue [Corbet 15].
- **KCM** (4.6): The Kernel Connection Multiplexor (KCM) provides an efficient message-based interface over TCP.
- **TCP NV** (4.8): New Vegas (NV) is a new TCP congestion control algorithm suited for high-bandwidth networks (those that run at 10+ Gbps).
- **XDP** (4.8, 4.18): eXpress Data Path (XDP) is a BPF-based programmable fast path for high-performance networking [Herbert 16]. An AF_XDP socket address family that can bypass much of the network stack was added in 4.18.
- **TCP BBR** (4.9): Bottleneck Bandwidth and RTT (BBR) is a TCP congestion control algorithm that provides improved latency and throughput over networks suffering packet loss and bufferbloat [Cardwell 16].
- **Hardware latency tracer** (4.9): An Ftrace tracer that can detect system latency caused by hardware and firmware, including system management interrupts (SMIs).
- **perf c2c** (4.10): The cache-to-cache (c2c) perf subcommand can help identify CPU cache performance issues, including false sharing.
- **Intel CAT** (4.10): Support for Intel Cache Allocation Technology (CAT) allowing tasks to have dedicated CPU cache space. This can be used by containers to help with the noisy neighbor problem.

- **Multiqueue I/O schedulers: BPQ, Kyber** (4.12): The Budget Fair Queueing (BFQ) multi-queue I/O scheduler provides low latency I/O for interactive applications, especially for slower storage devices. BFQ was significantly improved in 5.2. The Kyber I/O scheduler is suited for fast multiqueue devices [Corbet 17].
- **Kernel TLS** (4.13, 4.17): Linux version of kernel TLS [Edge 15].
- **MSG_ZEROCOPY** (4.14): A send(2) flag to avoid extra copies of packet bytes between an application and the network interface [Linux 20c].
- **PCID** (4.14): Linux added support for process-context ID (PCID), a processor MMU feature to help avoid TLB flushes on context switches. This reduced the performance cost of the kernel page table isolation (KPTI) patches needed to mitigate the meltdown vulnerability. See Section 3.4.3, KPTI (Meltdown).
- **PSI** (4.20, 5.2): Pressure stall information (PSI) is a set of new metrics to show time spent stalled on CPU, memory, or I/O. PSI threshold notifications were added in 5.2 to support PSI monitoring.
- **TCP EDT** (4.20): The TCP stack switched to Early Departure Time (EDT): This uses a timing-wheel scheduler for sending packets, providing better CPU efficiency and smaller queues [Jacobson 18].
- **Multi-queue I/O** (5.0): Multi-queue block I/O schedulers became the default in 5.0, and classic schedulers were removed.
- **UDP GRO** (5.0): UDP Generic Receive Offload (GRO) improves performance by allowing packets to be aggregated by the driver and card and passed up stack.
- **io_uring** (5.1): A generic asynchronous interface for fast communication between applications and the kernel, making use of shared ring buffers. Primary uses include fast disk and network I/O.
- **MADV_COLD, MADV_PAGEOUT** (5.4): These madvise(2) flags are hints to the kernel that memory is needed but not anytime soon. MADV_PAGEOUT is also a hint that memory can be reclaimed immediately. These are especially useful for memory-constrained embedded Linux devices.
- **MultiPath TCP** (5.6): Multiple network links (e.g., 3G and WiFi) can be used to improve the performance and reliability of a single TCP connection.
- **Boot-time tracing** (5.6): Allows Ftrace to trace the early boot process. (systemd can provide timing information on the late boot process: see Section 3.4.2, systemd.)
- **Thermal pressure** (5.7): The scheduler accounts for thermal throttling to make better placement decisions.
- **perf flame graphs** (5.8): perf(1) support for the flame graph visualization.

Not listed here are the many small performance improvements for locking, drivers, VFS, file systems, asynchronous I/O, memory allocators, NUMA, new processor instruction support, GPUs, and the performance tools perf(1) and Ftrace. System boot time has also been improved by the adoption of systemd.

The following sections describe in more detail three Linux topics important to performance: systemd, KPTI, and extended BPF.

3.4.2 systemd

systemd is a commonly used service manager for Linux, developed as a replacement for the original UNIX init system. systemd has various features including dependency-aware service startup and service time statistics.

An occasional task in systems performance is to tune the system's boot time, and the systemd time statistics can show where to tune. The overall boot time can be reported using systemd-analyze(1):

```
# systemd-analyze
Startup finished in 1.657s (kernel) + 10.272s (userspace) = 11.930s
graphical.target reached after 9.663s in userspace
```

This output shows that the system booted (reached the graphical.target in this case) in 9.663 seconds. More information can be seen using the `critical-chain` subcommand:

```
# systemd-analyze critical-chain
The time when unit became active or started is printed after the "@" character.
The time the unit took to start is printed after the "+" character.

graphical.target @9.663s
└─multi-user.target @9.661s
  └─snapd.seeded.service @9.062s +62ms
    └─basic.target @6.336s
      └─sockets.target @6.334s
        └─snapd.socket @6.316s +16ms
          └─sysinit.target @6.281s
            └─cloud-init.service @5.361s +905ms
              └─systemd-networkd-wait-online.service @3.498s +1.860s
                └─systemd-networkd.service @3.254s +235ms
                  └─network-pre.target @3.251s
                    └─cloud-init-local.service @2.107s +1.141s
                      └─systemd-remount-fs.service @391ms +81ms
                        └─systemd-journald.socket @387ms
                          └─system.slice @366ms
                            └─-.slice @366ms
```

This output shows the *critical path*: the sequence of steps (in this case, services) that causes the latency. The slowest service was `systemd-networkd-wait-online.service`, taking 1.86 seconds to start.

There are other useful subcommands: `blame` shows the slowest initialization times, and `plot` produces an SVG diagram. See the man page for systemd-analyze(1) for more information.

3.4.3 KPTI (Meltdown)

The kernel page table isolation (KPTI) patches added to Linux 4.14 in 2018 are a mitigation for the Intel processor vulnerability called "meltdown." Older Linux kernel versions had KAISER patches for a similar purpose, and other kernels have employed mitigations as well. While these work around the security issue, they also reduce processor performance due to extra CPU cycles and additional TLB flushing on context switches and syscalls. Linux added process-context ID (PCID) support in the same release, which allows some TLB flushes to be avoided, provided the processor supports pcid.

I evaluated the performance impact of KPTI as between 0.1% and 6% for Netflix cloud production workloads, depending on the workload's syscall rate (higher costs more) [Gregg 18a]. Additional tuning will further reduce the cost: the use of huge pages so that a flushed TLB warms up faster, and using tracing tools to examine syscalls to identify ways to reduce their rate. A number of such tracing tools are implemented using extended BPF.

3.4.4 Extended BPF

BPF stands for Berkeley Packet Filter, an obscure technology first developed in 1992 that improved the performance of packet capture tools [McCanne 92]. In 2013, Alexei Starovoitov proposed a major rewrite of BPF [Starovoitov 13], which was further developed by himself and Daniel Borkmann and included in the Linux kernel in 2014 [Borkmann 14b]. This turned BPF into a general-purpose execution engine that can be used for a variety of things, including networking, observability, and security.

BPF itself is a flexible and efficient technology composed of an instruction set, storage objects (maps), and helper functions. It can be considered a virtual machine due to its virtual instruction set specification. BPF programs run in kernel mode (as pictured earlier in Figure 3.2) and are configured to run on events: socket events, tracepoints, USDT probes, kprobes, uprobes, and perf_events. These are shown in Figure 3.16.

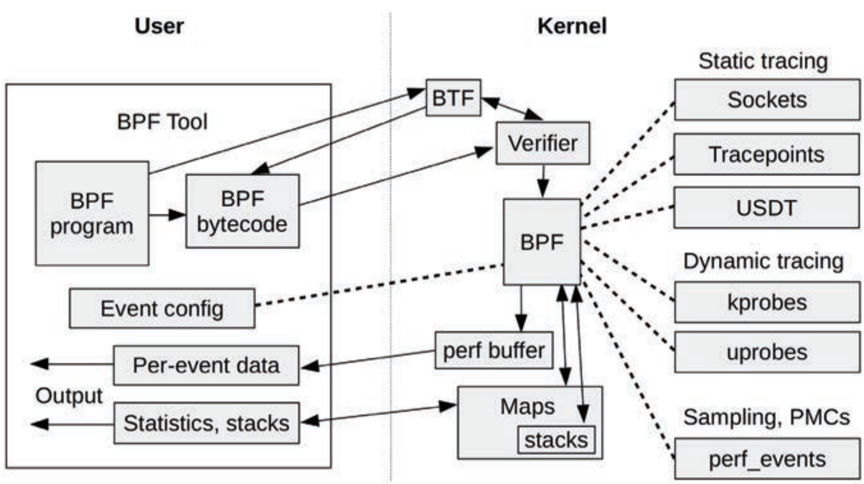


Figure 3.16 BPF components

BPF bytecode must first pass through a verifier that checks for safety, ensuring that the BPF program will not crash or corrupt the kernel. It may also use a BPF Type Format (BTF) system for understanding data types and structures. BPF programs can output data via a perf ring buffer, an efficient way to emit per-event data, or via maps, which are suited for statistics.

Because it is powering a new generation of efficient, safe, and advanced tracing tools, BPF is important for systems performance analysis. It provides programmability to existing kernel event sources: tracepoints, kprobes, uprobes, and perf_events. A BPF program can, for example, record a timestamp on the start and end of I/O to time its duration, and record this in a custom histogram. This book contains many BPF-based programs using the BCC and bpftrace front-ends. These front-ends are covered in Chapter 15.

3.5 Other Topics

Some additional kernel and operating system topics worth summarizing are PGO kernels, Unikernels, microkernels, hybrid kernels, and distributed operating systems.

3.5.1 PGO Kernels

Profile-guided optimization (PGO), also known as feedback-directed optimization (FDO), uses CPU profile information to improve compiler decisions [Yuan 14a]. This can be applied to kernel builds, where the procedure is:

1. While in production, take a CPU profile.
2. Recompile the kernel based on that CPU profile.
3. Deploy the new kernel in production.

This creates a kernel with improved performance for a specific workload. Runtimes such as the JVM do this automatically, recompiling Java methods based on their runtime performance, in conjunction with just-in-time (JIT) compilation. The process for creating a PGO kernel instead involves manual steps.

A related compile optimization is link-time optimization (LTO), where an entire binary is compiled at once to allow optimizations across the entire program. The Microsoft Windows kernel makes heavy use of both LTO and PGO, seeing 5 to 20% improvements from PGO [Bearman 20]. Google also use LTO and PGO kernels to improve performance [Tolvanen 20].

The gcc and clang compilers, and the Linux kernel, all have support for PGO. Kernel PGO typically involves running a specially instrumented kernel to collect profile data. Google has released an AutoFDO tool that bypasses the need for such a special kernel: AutoFDO allows a profile to be collected from a normal kernel using perf(1), which is then converted to the correct format for compilers to use [Google 20a].

The only recent documentation on building a Linux kernel with PGO or AutoFDO is two talks from Linux Plumber's Conference 2020 by Microsoft [Bearman 20] and Google [Tolvanen 20].[15]

---

[15] For a while the most recent documentation was from 2014 for Linux 3.13 [Yuan 14b], hindering adoption on newer kernels.

3.5.2 Unikernels

A unikernel is a single-application machine image that combines kernel, library, and application software together, and can typically run this in a single address space in either a hardware VM or on bare metal. This potentially has performance and security benefits: less instruction text means higher CPU cache hit ratios and fewer security vulnerabilities. This also creates a problem: there may be no SSH, shells, or performance tools available for you to log in and debug the system, nor any way to add them.

For unikernels to be performance tuned in production, new performance tooling and metrics must be built to support them. As a proof of concept, I built a rudimentary CPU profiler that ran from Xen dom0 to profile a domU unikernel guest and then built a CPU flame graph, just to show that it was possible [Gregg 16a].

Examples of unikernels include MirageOS [MirageOS 20].

3.5.3 Microkernels and Hybrid Kernels

Most of this chapter discusses Unix-like kernels, also described as *monolithic kernels*, where all the code that manages devices runs together as a single large kernel program. For the *microkernel* model, kernel software is kept to a minimum. A microkernel supports essentials such as memory management, thread management, and inter-process communication (IPC). File systems, the network stack, and drivers are implemented as user-mode software, which allows those user-mode components to be more easily modified and replaced. Imagine not only choosing which database or web server to install, but also choosing which network stack to install. The microkernel is also more fault-tolerant: a crash in a driver does not crash the entire kernel. Examples of microkernels include QNX and Minix 3.

A disadvantage with microkernels is that there are additional IPC steps for performing I/O and other functions, reducing performance. One solution for this is *hybrid kernels*, which combine the benefits of microkernels and monolithic kernels. Hybrid kernels move performance-critical services back into kernel space (with direct function calls instead of IPC) as they are with a monolithic kernel, but retains the modular design and fault tolerance of a micro kernel. Examples of hybrid kernels include the Windows NT kernel and the Plan 9 kernel.

3.5.4 Distributed Operating Systems

A distributed operating system runs a single operating system instance across a set of separate computer nodes, networked together. A microkernel is commonly used on each of the nodes. Examples of distributed operating systems include Plan 9 from Bell Labs, and the Inferno operating system.

While an innovative design, this model has not seen widespread use. Rob Pike, co-creator of Plan 9 and Inferno, has described various reasons for this, including [Pike 00]:

> “There was a claim in the late 1970s and early 1980s that Unix had killed operating systems research because no one would try anything else. At the time, I didn’t believe it. Today, I grudgingly accept that the claim may be true (Microsoft notwithstanding).”

On the cloud, today's common model for scaling compute nodes is to load-balance across a group of identical OS instances, which may scale in response to load (see Chapter 11, Cloud Computing, Section 11.1.3, Capacity Planning).

3.6 Kernel Comparisons

Which kernel is fastest? This will depend partly on the OS configuration and workload and how much the kernel is involved. In general, I expect that Linux will outperform other kernels due to its extensive work on performance improvements, application and driver support, and widespread use and the large community who discover and report performance issues. The top 500 supercomputers, as tracked by the TOP500 list since 1993, became 100% Linux in 2017 [TOP500 17]. There will be some exceptions; for example, Netflix uses Linux on the cloud and FreeBSD for its CDN.[16]

Kernel performance is commonly compared using micro-benchmarks, and this is error-prone. Such benchmarks may discover that one kernel is much faster at a particular syscall, but that syscall is not used in the production workload. (Or it is used, but with certain flags not tested by the microbenchmark, which greatly affect performance.) Comparing kernel performance accurately is a task for a senior performance engineer—a task that can take weeks. See Chapter 12, Benchmarking, Section 12.3.2, Active Benchmarking, as a methodology to follow.

In the first edition of this book, I concluded this section by noting that Linux did not have a mature dynamic tracer, without which you might miss out on large performance wins. Since that first edition, I have moved to a full-time Linux performance role, and I helped develop the dynamic tracers that Linux was missing: BCC and bpftrace, based on extended BPF. These are covered in Chapter 15 and in my previous book [Gregg 19].

Section 3.4.1, Linux Kernel Developments, lists many other Linux performance developments that have occurred in the time between the first edition and this edition, spanning kernel versions 3.1 and 5.8. A major development not listed earlier is that OpenZFS now supports Linux as its primary kernel, providing a high-performance and mature file system option on Linux.

With all this Linux development, however, comes complexity. There are so many performance features and tunables on Linux that it has become laborious to configure and tune them for each workload. I have seen many deployments running untuned. Bear this in mind when comparing kernel performance: has each kernel been tuned? Later chapters of this book, and their tuning sections, can help you remedy this.

3.7 Exercises

1. Answer the following questions about OS terminology:
   - What is the difference between a process, a thread, and a task?
   - What is a mode switch and a context switch?

[16] FreeBSD delivers higher performance for the Netflix CDN workload, especially due to kernel improvements made by the Netflix OCA team. This is routinely tested, most recently during 2019 with a production comparison of Linux 5.0 versus FreeBSD, which I helped analyze.

- What is the difference between paging and process swapping?
- What is the difference between I/O-bound and CPU-bound workloads?

2. Answer the following conceptual questions:
   - Describe the role of the kernel.
   - Describe the role of system calls.
   - Describe the role of VFS and its location in the I/O stack.
3. Answer the following deeper questions:
   - List the reasons why a thread would leave the current CPU.
   - Describe the advantages of virtual memory and demand paging.

3.8 References

[Graham 68] Graham, B., "Protection in an Information Processing Utility," *Communications of the ACM*, May 1968.

[Ritchie 74] Ritchie, D. M., and Thompson, K., "The UNIX Time-Sharing System," *Communications of the ACM* 17, no. 7, pp. 365–75, July 1974.

[Thompson 78] Thompson, K., *UNIX Implementation*, Bell Laboratories, 1978.

[Bach 86] Bach, M. J., *The Design of the UNIX Operating System*, Prentice Hall, 1986.

[Mellor-Crummey 91] Mellor-Crummey, J. M., and Scott, M., "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *ACM Transactions on Computing Systems*, Vol. 9, No. 1, https://www.cs.rochester.edu/u/scott/papers/1991_TOCS_synch.pdf, 1991.

[McCanne 92] McCanne, S., and Jacobson, V., "The BSD Packet Filter: A New Architecture for User-Level Packet Capture", *USENIX Winter Conference*, 1993.

[Mills 94] Mills, D., "RFC 1589: A Kernel Model for Precision Timekeeping," *Network Working Group*, 1994.

[Lever 00] Lever, C., Eriksen, M. A., and Molloy, S. P., "An Analysis of the TUX Web Server," *CITI Technical Report 00-8*, http://www.citi.umich.edu/techreports/reports/citi-tr-00-8.pdf, 2000.

[Pike 00] Pike, R., "Systems Software Research Is Irrelevant," http://doc.cat-v.org/bell_labs/utah2000/utah2000.pdf, 2000.

[Mauro 01] Mauro, J., and McDougall, R., *Solaris Internals: Core Kernel Architecture*, Prentice Hall, 2001.

[Bovet 05] Bovet, D., and Cesati, M., *Understanding the Linux Kernel*, 3rd Edition, O'Reilly, 2005.

[Corbet 05] Corbet, J., Rubini, A., and Kroah-Hartman, G., *Linux Device Drivers*, 3rd Edition, O'Reilly, 2005.

[Corbet 13a] Corbet, J., "Is the whole system idle?" *LWN.net*, https://lwn.net/Articles/558284, 2013.

[Corbet 13b] Corbet, J., "The multiqueue block layer," *LWN.net*, https://lwn.net/Articles/552904, 2013.

[Starovoitov 13] Starovoitov, A., "[PATCH net-next] extended BPF," *Linux kernel mailing list*, https://lkml.org/lkml/2013/9/30/627, 2013.

[Borkmann 14a] Borkmann, D., "net: tcp: add DCTCP congestion control algorithm," https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e3118e8359bb7c59555aca60c725106e6d78c5ce, 2014.

[Borkmann 14b] Borkmann, D., "[PATCH net-next 1/9] net: filter: add jited flag to indicate jit compiled filters," *netdev mailing list*, https://lore.kernel.org/netdev/1395404418-25376-1-git-send-email-dborkman@redhat.com/T, 2014.

[Corbet 14] Corbet, J., "MCS locks and qspinlocks," *LWN.net*, https://lwn.net/Articles/590243, 2014.

[Drysdale 14] Drysdale, D., "Anatomy of a system call, part 2," *LWN.net*, https://lwn.net/Articles/604515, 2014.

[Yuan 14a] Yuan, P., Guo, Y., and Chen, X., "Experiences in Profile-Guided Operating System Kernel Optimization," *APSys*, 2014.

[Yuan 14b] Yuan P., Guo, Y., and Chen, X., "Profile-Guided Operating System Kernel Optimization," http://coolypf.com, 2014.

[Corbet 15] Corbet, J., "Epoll evolving," LWN.*net*, https://lwn.net/Articles/633422, 2015.

[Edge 15] Edge, J., "TLS in the kernel," *LWN.net*, https://lwn.net/Articles/666509, 2015.

[Heo 15] Heo, T., "Control Group v2," *Linux documentation*, https://www.kernel.org/doc/Documentation/cgroup-v2.txt, 2015.

[McKusick 15] McKusick, M. K., Neville-Neil, G. V., and Watson, R. N. M., *The Design and Implementation of the FreeBSD Operating System*, 2nd Edition, Addison-Wesley, 2015.

[Stewart 15] Stewart, R., Gurney, J. M., and Long, S., "Optimizing TLS for High-Bandwidth Applicationsin FreeBSD," *AsiaBSDCon*, https://people.freebsd.org/~rrs/asiabsd_2015_tls.pdf, 2015.

[Cardwell 16] Cardwell, N., Cheng, Y., Stephen Gunn, C., Hassas Yeganeh, S., and Jacobson, V., "BBR: Congestion-Based Congestion Control," *ACM queue*, https://queue.acm.org/detail.cfm?id=3022184, 2016.

[Gregg 16a] Gregg, B., "Unikernel Profiling: Flame Graphs from dom0," http://www.brendangregg.com/blog/2016-01-27/unikernel-profiling-from-dom0.html, 2016.

[Herbert 16] Herbert, T., and Starovoitov, A., "eXpress Data Path (XDP): Programmable and High Performance Networking Data Path," https://github.com/iovisor/bpf-docs/raw/master/Express_Data_Path.pdf, 2016.

[Corbet 17] Corbet, J., "Two new block I/O schedulers for 4.12," *LWN.net*, https://lwn.net/Articles/720675, 2017.

[TOP500 17] TOP500, "List Statistics," https://www.top500.org/statistics/list, 2017.

[Gregg 18a] Gregg, B., "KPTI/KAISER Meltdown Initial Performance Regressions," http://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html, 2018.

[Jacobson 18] Jacobson, V., "Evolving from AFAP: Teaching NICs about Time," *netdev 0x12*, https://netdevconf.info/0x12/session.html?evolving-from-afap-teaching-nics-about-time, 2018.

[Gregg 19] Gregg, B., *BPF Performance Tools: Linux System and Application Observability*, Addison-Wesley, 2019.

[Looney 19] Looney, J., "Netflix and FreeBSD: Using Open Source to Deliver Streaming Video," *FOSDEM*, https://papers.freebsd.org/2019/fosdem/looney-netflix_and_freebsd, 2019.

[Bearman 20] Bearman, I., "Exploring Profile Guided Optimization of the Linux Kernel," *Linux Plumber's Conference*, https://linuxplumbersconf.org/event/7/contributions/771, 2020.

[Google 20a] Google, "AutoFDO," https://github.com/google/autofdo, accessed 2020.

[Linux 20a] "NO_HZ: Reducing Scheduling-Clock Ticks," *Linux documentation*, https://www.kernel.org/doc/html/latest/timers/no_hz.html, accessed 2020.

[Linux 20b] "Deadline Task Scheduling," *Linux documentation*, https://www.kernel.org/doc/Documentation/scheduler/sched-deadline.rst, accessed 2020.

[Linux 20c] "MSG_ZEROCOPY," *Linux documentation*, https://www.kernel.org/doc/html/latest/networking/msg_zerocopy.html, accessed 2020.

[Linux 20d] "Softlockup Detector and Hardlockup Detector (aka nmi_watchdog)," *Linux documentation*, https://www.kernel.org/doc/html/latest/admin-guide/lockup-watchdogs.html, accessed 2020.

[MirageOS 20] MirageOS, "Mirage OS," https://mirage.io, accessed 2020.

[Owens 20] Owens, K., et al., "4. Kernel Stacks," *Linux documentation*, https://www.kernel.org/doc/html/latest/x86/kernel-stacks.html, accessed 2020.

[Tolvanen 20] Tolvanen, S., Wendling, B., and Desaulniers, N., "LTO, PGO, and AutoFDO in the Kernel," *Linux Plumber's Conference*, https://linuxplumbersconf.org/event/7/contributions/798, 2020.

3.8.1 Additional Reading

Operating systems and their kernels is a fascinating and extensive topic. This chapter summarized only the essentials. In addition to the sources mentioned in this chapter, the following are also excellent references, applicable to Linux-based operating systems and others:

[Goodheart 94] Goodheart, B., and Cox J., *The Magic Garden Explained: The Internals of UNIX System V Release 4, an Open Systems Design*, Prentice Hall, 1994.

[Vahalia 96] Vahalia, U., *UNIX Internals: The New Frontiers*, Prentice Hall, 1996.

[Singh 06] Singh, A., *Mac OS X Internals: A Systems Approach*, Addison-Wesley, 2006.

[McDougall 06b] McDougall, R., and Mauro, J., *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*, Prentice Hall, 2006.

[Love 10] Love, R., *Linux Kernel Development*, 3rd Edition, Addison-Wesley, 2010.

[Tanenbaum 14] Tanenbaum, A., and Bos, H., *Modern Operating Systems*, 4th Edition, Pearson, 2014.

[Yosifovich 17] Yosifovich, P., Ionescu, A., Russinovich, M. E., and Solomon, D. A., *Windows Internals, Part 1 (Developer Reference)*, 7th Edition, Microsoft Press, 2017.

Index

A

Accelerated receive flow steering, 523
Accelerators in USE method, 49
`accept` system calls, 95
Access timestamps, 371
ACK detection in TCP, 512
Actions in `bpftrace`, 769
Active benchmarking, 657–660
Active listening in three-way handshakes, 511
Active pages in page caches, 318
Activities overview, 3–4
Ad hoc checklist method, 43–44
Adaptive mutex locks, 198
Adaptive Replacement Cache (ARC), 381
Address space, 304
  guests, 603
  kernel, 90
  memory, 304, 310
  processes, 95, 99–102, 319–322
  threads, 227–228
  virtual memory, 104, 305
Address space layout randomization (ASLR), 723
Advanced Format for magnetic rotational disks, 437
AF_NETLINK address family, 145–146
Agents
  monitoring software, 137–138
  product monitoring, 79
AKS (Azure Kubernetes Service), 586

Alerts, 8

Algorithms

caching, 36

congestion control, 115, 118, 513–514

big O notation, 175–176

Allocation groups in XFS, 380

Allocators

memory, 309

multithreaded applications, 353

process virtual address space, 320–321

Amazon EKS (Elastic Kubernetes Service), 586

Amdahl's Law of Scalability, 64–65

Analysis

benchmarking, 644–646, 665–666

capacity planning, 38, 71–72

drill-down, 55–56

I/O traces, 478–479

latency, 56–57, 384–386, 454–455

off-CPU, 188–192

resource, 38–39

thread state, 193–197

workload, 4–5, 39–40

Analysis step in scientific method, 44–45

Analysis strategy in case study, 784

`annotate` subcommand for `perf`, 673

Anonymous memory, 304

Anonymous paging, 305–307

Anti-methods

blame-someone-else, 43

random change, 42–43

streetlight, 42

Apdex (application performance index), 174

Application calls, tuning, 415–416

Application I/O, 369, 435

Application instrumentation in off-CPU analysis, 189

Application internals, 213

Application layer, file system latency in, 384

Application performance index (Apdex), 174

Applications, 171

basics, 172–173

big O notation, 175–176

`bpftrace` for, 765

common case optimization, 174

common problems, 213–215

exercises, 216–217

internals, 213

latency documentation, 385

methodology. *See* Applications methodology

missing stacks, 215–216

missing symbols, 214

objectives, 173–174

observability, 174

observability tools. *See* Applications observability tools

performance techniques. *See* Applications performance techniques

programming languages. *See* Applications programming languages

references, 217–218

Applications methodology

CPU profiling, 187–189

distributed tracing, 199

lock analysis, 198

off-CPU analysis, 189–192

overview, 186–187

static performance tuning, 198–199

syscall analysis, 192

thread state analysis, 193–197

USE method, 193

Applications observability tools

`bpftrace`, 209–213

`execsnoop`, 207–208

`offcputime`, 204–205

overview, 199–200

`perf`, 200–203

`profile`, 203–204

`strace`, 205–207

`syscount`, 208–209

Applications performance techniques
- buffers, 177
- caching, 176
- concurrency and parallelism, 177–181
- I/O size selection, 176
- non-blocking I/O, 181
- Performance Mantras, 182
- polling, 177
- processor binding, 181–182

Applications programming languages, 182–183
- compiled, 183–184
- garbage collection, 184–185
- interpreted, 184–185
- virtual machines, 185

Appropriateness level in methodologies, 28–29

ARC (Adaptive Replacement Cache), 381

Architecture
- CPUs. *See* CPUs architecture
- disks. *See* Disks architecture
- file systems. *See* File systems architecture
- vs. loads, 581–582
- memory. *See* Memory architecture
- networks. *See* Networks architecture
- scalable, 581–582

`archive` subcommand for `perf`, 673

`arcstat.pl` tool, 410

`arg` variables for `bpftrace`, 778

`argdist` tool, 757–759

Arguments
- kprobes, 152
- networks, 507
- tracepoints, 148–149
- uprobes, 154

Arithmetic mean, 74

Arrival process in queueing systems, 67

ASG (auto scaling group)
- capacity planning, 72
- cloud computing, 583–584

ASLR (address space layout randomization), 723

Associativity in caches, 234

Asynchronous disk I/O, 434–435

Asynchronous interrupts, 96–97

Asynchronous writes, 366

`atop` tool, 285

Auto scaling group (ASG)
- capacity planning, 72
- cloud computing, 583–584

available_filter_functions file, 710

Available swap, 309

available_tracers file, 710

Averages, 74–75

`avg` function, 780

`await` metric, 461

Axes
- flame graphs, 10, 187, 290
- heat maps, 289, 410, 488–489
- line charts, 59, 80
- scalability tests, 62
- scatter plots, 81–82, 488

Azure Kubernetes Service (AKS), 586

B

Back-ends in instruction pipeline, 224

Background color in flame graphs, 291

Backlogs in network connections, 507, 519–520, 556–557, 569

Bad paging, 305

Balloon drivers, 597

Bandwidth
- disks, 424
- interconnects, 237
- networks, 500, 508, 532–533
- OS virtualization, 614–615

Bare-metal hypervisors, 587

Baseline statistics, 59

BATCH scheduling policy, 243

BBR (Bottleneck Bandwidth and RTT) algorithm, 118, 513
bcache technology, 117
BCC (BPF Compiler Collection), 12
vs. `bpftrace`, 760
disks, 450
documentation, 760–761
installing, 754
multi-purpose tools, 757
multi-tool example, 759
networks, 526
one-liners, 757–759
overview, 753–754
vs. perf-tools, 747–748
single-purpose tools, 755–757
slow disks case study, 17
system-wide tracing, 136
tool overview, 754–755
bcc-tools tool package, 132
`BEGIN` probes in `bpftrace`, 774
`bench` subcommand for `perf`, 673
Benchmark paradox, 648–649
Benchmarketing, 642
Benchmarking, 641–642
analysis, 644–646
capacity planning, 70
CPUs, 254
effective, 643–644
exercises, 668
failures, 645–651
industry standards, 654–656
memory, 328
micro-benchmarking. *See* Micro-benchmarking
questions, 667–668
reasons, 642–643
references, 669–670
replay, 654
simulation, 653–654
specials, 650
SysBench system, 294
types, 13, 651–656
Benchmarking methodology
active, 657–660
checklist, 666–667
CPU profiling, 660–661
custom benchmarks, 662
overview, 656
passive, 656–657
ramping load, 662–664
sanity checks, 664–665
statistical analysis, 665–666
USE method, 661
workload characterization, 662
Berkeley Packet Filter (BPF), 751–752
BCC compiler. *See* BCC (BPF Compiler Collection)
`bpftrace`. *See* `bpftrace` tool
description, 12–13
extended. *See* Extended BPF
iterator, 562
JIT compiler, 117
kernels, 92
OS virtualization tracing, 620, 624–625, 629
vs. perf-tools, 747–748
program, 90
Berkeley Software Distribution (BSD), 113
BFQ (Budget Fair Queueing) I/O schedulers, 119, 449
Big kernel lock (BKL) performance bottleneck, 116
Big O notation, 175–176
Billing in cloud computing, 584
Bimodal performance, 76
Binary executable files, 183
Binary translations in hardware virtualization, 588, 590

Binding
CPU, 253, 297–298
NUMA, 353
processor, 181–182
bioerr tool, 487
biolatency tool
BCC, 753–755
disks, 450, 468–470
example, 753–754
biopattern tool, 487
BIOS, tuning, 299
biosnoop tool
BCC, 755
disks, 470–472
event tracing, 58
hardware virtualization, 604–605
outliers, 471–472
queued time, 472
system-wide tracing, 136
biostacks tool, 474–475
biotop tool
BCC, 755
disks, 450, 473–474
Bit width in CPUs, 229
bitesize tool
BCC, 755
perf-tools, 743
blame command, 120
Blame-someone-else anti-method, 43
Blanco, Brenden, 753
Blind faith benchmarking, 645
blk tracer, 708
blkio control group, 610, 617
blkreplay tool, 493
blktrace tool
action filtering, 478
action identifiers, 477
analysis, 478–479
default output, 476–477
description, 116
disks, 475–479
RWBS description, 477
visualizations, 479
Block-based file systems, 375–376
Block caches in disk I/O, 430
Block device interface, 109–110, 447
Block I/O state in delay accounting, 145
Block I/O times for disks, 427–428, 472
Block interleaving, 378
Block size
defined, 360
FFS, 378
Block stores in cloud computing, 584
Blue-green cloud computing deployments, 3–4
Bonnie and Bonnie++ benchmarking tools
active benchmarking, 657–660
file systems, 412–414
Boolean expressions in bpftrace, 775–776
Boot options, security, 298–299
Boot-time tracing, 119
Borkmann, Daniel, 121
Borrowed virtual time (BVT) schedulers, 595
Bottleneck Bandwidth and RTT (BBR) algorithm, 118, 513
Bottlenecks
capacity planning, 70–71
complexity, 6
defined, 22
USE method, 47–50, 245, 324, 450–451
BPF. *See* Berkeley Packet Filter (BPF)
bpftrace tool, 12–13
application internals, 213
vs. BCC, 752–753, 760
block I/O events, 625, 658–659
description, 282
disk I/O errors, 483

disk I/O latency, 482–483
disk I/O size, 480–481
event sources, 558
examples, 284, 761–762
file system internals, 408
hardware virtualization, 602
I/O profiling, 210–212
installing, 762
lock tracing, 212–213
`malloc()` bytes flame graph, 346
memory internals, 346–347
one-liners for CPUs, 283, 803–804
one-liners for disks, 479–480, 806–807
one-liners for file systems, 402–403, 805–806
one-liners for memory, 343–344, 804–805
one-liners for networks, 550–552, 807–808
one-liners overview, 763–765
package contents, 132
packet inspection, 526
page fault flame graphs, 346
programming. *See* `bpftrace` tool programming
references, 782
scheduling internals, 284–285
signal tracing, 209–210
socket tracing, 552–555
stacks viewing, 450
syscall tracing, 403–405
system-wide tracing, 136
TCP tracing, 555–557
tracepoints, 149
user allocation stacks, 345
VFS tracing, 405–408

`bpftrace` tool programming
actions, 769
comments, 767
documentation, 781
example, 766
filters, 769
flow control, 775–777
functions, 770–772, 778–781
Hello, World! program, 770
operators, 776–777
probe arguments, 775
probe format, 768
probe types, 774–775
probe wildcards, 768–769
program structure, 767
timing, 772–773
usage, 766–767
variables, 770–771, 777–778

BQL (Byte Queue Limits)
driver queues, 524
tuning, 571

Branch prediction in instruction pipeline, 224

Breakpoints in `perf`, 680

brk system calls, 95

`brkstack` tool, 348

Broadcast network messages, 503

BSD (Berkeley Software Distribution), 113

`btrace` tool, 476, 478

btrfs file system, 381–382, 399

`btrfsdist` tool, 755

`btrfsslower` tool, 755

`btt` tool, 478

Buckets
hash tables, 180
heat maps, 82–83

Buddy allocators, 317

Budget Fair Queueing (BFQ) I/O schedulers, 119, 449

`buf` function, 778

Buffer caches, 110, 374

Bufferbloat, 507

Buffers
applications, 177
block devices, 110, 374
networks, 507

ring, 522
TCP, 520, 569
`bufgrow` tool, 409
Bug database systems
applications, 172
case studies, 792–793
`buildid-cache` subcommand for `perf`, 673
Built-in `bpftrace` variables, 770, 777–778
Bursting in cloud computing, 584, 614–615
Buses, memory, 312–313
BVT (borrowed virtual time) schedulers, 595
Bypass, kernel, 94
Byte Queue Limits (BQL)
driver queues, 524
tuning, 571
Bytecode, 185

C

C, C++
compiled languages, 183
symbols, 214
stacks, 215
C-states in CPUs, 231
`c2c` subcommand for `perf`, 673, 702
Cache Allocation Technology (CAT), 118, 596
Cache miss rate, 36
Cache warmth, 222
`cachegrind` tool, 135
Caches and caching
applications, 176
associativity, 234
block devices, 110, 374
cache line size, 234
coherency, 234–235
CPUs, hardware virtualization, 596
CPUs, memory, 221–222, 314
CPUs, OS virtualization, 615–616
CPUs, processors, 230–235
CPUs, vs. GPUs, 240
defined, 23
dentry, 375
disks, I/O, 430
disks, on-disk, 437
disks, tuning, 456
file systems, flushing, 414
file systems, OS virtualization, 613
file systems, overview, 361–363
file systems, tuning, 389, 414–416
file systems, types, 373–375
file systems, usage, 309
inode, 375
methodologies, 35–37
micro-benchmarking test, 390
operating systems, 108–109
page, 315, 374
`perf` events, 680
RAID, 445
tuning, 60
write-back, 365
`cachestat` tool
file systems, 399, 658–659
memory, 348
perf-tools, 743
slow disks case study, 17
Caching disk model, 425–426
Canary testing, 3
Capacity-based utilization, 34
Capacity of file systems, 371
Capacity planning
benchmarking for, 642
cloud computing, 582–584
defined, 4
factor analysis, 71–72
micro-benchmarking, 70
overview, 69
resource analysis, 38
resource limits, 70–71
scaling solutions, 72–73

CAPI (Coherent Accelerator Processor Interface), 236

Carrier sense multiple access with collision detection (CSMA/CD) algorithm, 516

CAS (column address strobe) latency, 311

Cascading failures, 5

Case studies

- analysis strategy, 784
- bug database systems, 792–793
- conclusion, 792
- configuration, 786–788
- PMCs, 788–789
- problem statement, 783–784
- references, 793
- slow disks, 16–18
- software change, 18–19
- software events, 789–790
- statistics, 784–786
- tracing, 790–792

Casual benchmarking, 645

CAT (Cache Allocation Technology), 118, 596

`cat` function, 779

CAT (Intel Cache Allocation Technology), 118, 596

CFQ (completely fair queueing), 115, 449

CFS (completely fair scheduler), 116–117

- CPU scheduling, 241
- CPU shares, 614–615
- description, 243

`cgroup` file, 141

`cgroup` variable, 778

`cgroupid` function, 779

cgroups

- block I/O, 494
- description, 116, 118
- Linux kernel, 116
- memory, 317, 353
- OS virtualization, 606, 608–611, 613–620, 630
- resource management, 111, 298
- statistics, 139, 141, 620–622, 627–628

`cgtop` tool, 621

Character devices, 109–110

Characterizing memory usage, 325–326

Cheating in benchmarking, 650–651

Checklists

- ad hoc checklist method, 43–44
- benchmarking, 666
- CPUs, 247, 527
- disks, 453
- file systems, 387
- Linux 60-second analysis, 15
- memory, 325

Chip-level multiprocessing (CMP), 220

chrt command, 295

Cilium, 509, 586, 617

Circular buffers for applications, 177

CISCs (complex instruction set computers), 224

clang complier, 122

Classes, scheduling

- CPUs, 242–243
- I/O, 493
- kernel, 106, 115
- priority, 295

Clean memory, 306

`clear` function in `bpftrace`, 780

`clear` subcommand in `trace-cmd`, 735

clock routine, 99

Clocks

- CPUs, 223, 230
- CPUs vs. GPUs, 240
- operating systems, 99

clone system calls, 94, 100

Cloud APIs, 580

Cloud computing, 579–580

- background, 580–581
- capacity planning, 582–584

comparisons, 634–636
vs. enterprise, 62
exercises, 636–637
hardware virtualization. *See* Hardware virtualization
instance types, 581
lightweight virtualization, 630–633
multitenancy, 585–586
orchestration, 586
OS virtualization. *See* OS virtualization
overview, 14
PMCs, 158
proof-of-concept testing, 3
references, 637–639
scalable architecture, 581–582
storage, 584–585
types, 634
Cloud-native databases, 582
Clue-based approach in thread state analysis, 196
Clusters in cloud computing, 586
CMP (chip-level multiprocessing), 220
CNI (container network interface) software, 586
Co-routines in applications, 178
Coarse view in profiling, 35
Code changes in cloud computing, 583
Coefficient of variation (CoV), 76
Coherence
caches, 234–235
models, 63
Coherent Accelerator Processor Interface (CAPI), 236
Cold caches, 36
`collectd` agent, 138
Collisions
hash, 180
networks, 516
Colors in flame graphs, 291
Column address strobe (CAS) latency, 311
Column quantizations, 82–83
`comm` variable in `bpftrace`, 778
Comma-separated values (CSV) format for `sar`, 165
Comments in `bpftrace`, 767
Common case optimization in applications, 174
Communication in multiprocess vs. multithreading, 228
Community applications, 172–173
Comparing benchmarks, 648
Competition, benchmarking, 649
Compiled programming languages
optimizations, 183–184
overview, 183
Compilers
CPU optimization, 229
options, 295
Completely fair queueing (CFQ), 115, 449
Completely fair scheduler (CFS), 116–117
CPU scheduling, 241
CPU shares, 614–615
description, 243
Completion target in workload analysis, 39
Complex benchmark tools, 646
Complex instruction set computers (CISCs), 224
Complexity, 5
Comprehension in flame graphs, 249
Compression
btrfs, 382
disks, 369
ZFS, 381
Compute kernel, 240
Compute Unified Device Architecture (CUDA), 240
Concurrency
applications, 177–181
micro-benchmarking, 390, 456
CONFIG options, 295–296

CONFIG_TASK_DELAY_ACCT option, 145
Configuration
applications, 172
case study, 786–788
network options, 574
Congestion avoidance and control
Linux kernel, 115
networks, 508
TCP, 510, 513
tuning, 570
`connect` system calls, 95
Connections for networks, 509
backlogs, 507, 519–520, 556–557, 569
characteristics, 527–528
firewalls, 517
latency, 7, 24–25, 505–506, 528
life span, 507
local, 509
monitoring, 529
NICs, 109
QUIC, 515
TCP queues, 519–520
three-way handshakes, 511–512
UDP, 514
Container network interface (CNI) software, 586
Containers
lightweight virtualization, 631–632
orchestration, 586
observability, 617–630
OS virtualization, 605–630
resource controls, 52, 70, 613–617, 626
Contention
locks, 198
models, 63
Context switches
defined, 90
kernels, 93
Contributors to system performance technologies, 811–814
Control groups (cgroups). *See* cgroups
Control paths in hardware virtualization, 594
Control units in CPUs, 230
Controllers
caches, 430
disk, 426
mechanical disks, 439
micro-benchmarking, 457
network, 501–502, 516
solid-state drives, 440–441
tunable, 494–495
USE method, 49, 451
Controls, resource. *See* Resource controls
Cookies, TCP, 511, 520
Copy-on-write (COW) file systems, 376
btrfs, 382
ZFS, 380
Copy-on-write (COW) process strategy, 100
CoreLink Interconnects, 236
Cores
CPUs vs. GPUs, 240
defined, 220
Corrupted file system data, 365
`count` function in `bpftrace`, 780
Counters, 8–9
fixed, 133–135
hardware, 156–158
CoV (coefficient of variation), 76
COW (copy-on-write) file systems, 376
btrfs, 382
ZFS, 380
COW (copy-on-write) process strategy, 100
CPCs (CPU performance counters), 156
CPI (cycles per instruction), 225
CPU affinity, 222
CPU-bound applications, 106
cpu control group, 610
CPU mode for applications, 172
CPU performance counters (CPCs), 156
CPU registers, perf-tools for, 746–747

`cpu` variable in `bpftrace`, 777
cpuacct control group, 610
`cpudist` tool
BCC, 755
case study, 790–791
threads, 278–279
`cpufreq` tool, 285
`cpuinfo` tool, 142
`cpupower` tool, 286–287
CPUs, 219–220
architecture. *See* CPUs architecture
benchmark questions, 667–668
binding, 181–182
`bpftrace` for, 763, 803–804
clock rate, 223
compiler optimization, 229
cross calls, 110
exercises, 299–300
experiments, 293–294
feedback-directed optimization, 122
flame graphs. *See* Flame graphs
FlameScope tool, 292–293
garbage collection, 185
hardware virtualization, 589–592, 596–597
I/O wait, 434
instructions, defined, 220
instructions, IPC, 225
instructions, pipeline, 224
instructions, size, 224
instructions, steps, 223
instructions, width, 224
memory caches, 221–222
memory tradeoffs with, 27
methodology. *See* CPUs methodology
models, 221–222
multiprocess and multithreading, 227–229
observability tools. *See* CPUs observability tools
OS virtualization, 611, 614, 627, 630
preemption, 227
priority inversion, 227
profiling. *See* CPUs profiling
references, 300–302
run queues, 222
saturation, 226–227
scaling in networks, 522–523
schedulers, 105–106
scheduling classes, 115
simultaneous multithreading, 225
statistic accuracy, 142–143
subsecond-offset heat maps, 289
terminology, 220
thread pools, 178
tuning. *See* CPUs tuning
USE method, 49–51, 795–797
user time, 226
utilization, 226
utilization heat maps, 288–289
virtualization support, 588
visualizations, 288–293
volumes and pools, 383
word size, 229
CPUs architecture, 221, 229
accelerators, 240–242
associativity, 234
caches, 230–235
GPUs, 240–241
hardware, 230–241
idle threads, 244
interconnects, 235–237
latency, 233–234
memory management units, 235
NUMA grouping, 244
P-states and C-states, 231
PMCs, 237–239
processors, 230
schedulers, 241–242
scheduling classes, 242–243
software, 241–244

CPUs methodology
CPU binding, 253
cycle analysis, 251
micro-benchmarking, 253–254
overview, 244–245
performance monitoring, 251
priority tuning, 252–253
profiling, 247–250
resource controls, 253
sample processing, 247–248
static performance tuning, 252
tools method, 245
USE, 245–246
workload characterization, 246–247
CPUs observability tools, 254–255
`bpftrace`, 282–285
`cpudist`, 278–279
GPUs, 287
`hardirqs`, 282
miscellaneous, 285–286
`mpstat`, 259
`perf`, 267–276
`pidstat`, 262
`pmcarch`, 265–266
`profile`, 277–278
`ps`, 260–261
`ptime`, 263–264
`runqlat`, 279–280
`runqlen`, 280–281
`sar`, 260
`showboost`, 265
`softirqs`, 281–282
`time`, 263–264
`tlbstat`, 266–267
`top`, 261–262
`turbostat`, 264–265
`uptime`, 255–258
`vmstat`, 258
CPUs profiling
applications, 187–189
benchmarking, 660–661
`perf`, 200–201
`record`, 695–696
steps, 247–250
system-wide, 268–270
CPUs tuning
compiler options, 295
CPU binding, 297–298
exclusive CPU sets, 298
overview, 294–295
power states, 297
processor options, 299
resource controls, 298
scaling governors, 297
scheduler options, 295–296
scheduling priority and class, 295
security boot options, 298–299
Cpusets, 116
CPU binding, 253
exclusive, 298
cpusets control group, 610, 614, 627
`cpuunclaimed` tool, 755
Crash resilience, multiprocess vs. multithreading, 228
Credit-based schedulers, 595
Crisis tools, 131–133
`critical-chain` command, 120
Critical paths in systemd service manager, 120
`criticalstat` tool, 756
CSMA/CD (carrier sense multiple access with collision detection) algorithm, 516
CSV (comma-separated values) format for `sar`, 165
CUBIC algorithm for TCP congestion control, 513
CUDA (Compute Unified Device Architecture), 240
CUMASK values in MSRs, 238–239
current_tracer file, 710
`curtask` variable for `bpftrace`, 778

Custom benchmarks, 662
Custom load generators, 491
Cycle analysis
CPUs, 251
memory, 326
Cycles per instruction (CPI), 225
Cylinder groups in FFS, 378

D

Daily patterns, monitoring, 78
Data Center TCP (DCTCP) congestion control, 118, 513
Data deduplication in ZFS, 381
Data integrity in magnetic rotational disks, 438
Data paths in hardware virtualization, 594
Data Plane Development Kit (DPDK), 523
Data rate in throughput, 22
Databases
applications, 172
case studies, 792–793
cloud computing, 582
Datagrams
OSI model, 502
UDP, 514
DAX (Direct Access), 118
`dbslower` tool, 756
`dbstat` tool, 756
Dcache (dentry cache), 375
`dcsnoop` tool, 409
`dcstat` tool, 409
DCTCP (Data Center TCP) congestion control, 118, 513
`dd` command
disks, 490–491
file systems, 411–412
DDR SDRAM (double data rate synchronous dynamic random-access memory), 313
Deadline I/O schedulers, 243, 448
DEADLINE scheduling policy, 243
DebugFS interface, 116
Decayed average, 75
Deflated disk I/O, 369
Defragmentation in XFS, 380
Degradation in scalability, 31–32
Delay accounting
kernel, 116
off-CPU analysis, 197
overview, 145
Delayed ACKs algorithm, 513
Delayed allocation
ext4, 379
XFS, 380
`delete` function in `bpftrace`, 780
Demand paging
BSD kernel, 113
memory, 307–308
Dentry caches (dcaches), 375
Dependencies in perf-tools, 748
Development, benchmarking for, 642
Development attribute, multiprocess vs. multithreading, 228
Devices
backlog tuning, 569
disk I/O caches, 430
drivers, 109–110, 522
hardware virtualization, 588, 594, 597
devices control group, 610
`df` tool, 409
Dhrystone benchmark
CPUs, 254
simulations, 653
Diagnosis cycle, 46
`diff` subcommand for `perf`, 673
Differentiated Services Code Points (DSCPs), 509–510
Direct Access (DAX), 118
Direct buses, 313

Direct I/O, 366
Direct mapped caches, 234
Direct measurement approach in thread state analysis, 197
Direct-reclaim memory method, 318–319
Directories in file systems, 107
Directory indexes in ext3, 379
Directory name lookup cache (DNLC), 375
Dirty memory, 306
Disk commands, 424
Disk controllers
caches, 430
magnetic rotational disks, 439
tunable, 494–495
USE method, 451
Disk I/O state in thread state analysis, 194–197
Disk request time, 428
Disk response time, 428
Disk service time, 428–429
Disk wait time, 428
Disks, 423–424
architecture. *See* Disks architecture
exercises, 495–496
experiments, 490–493
I/O. *See* Disks I/O
IOPS, 432
latency analysis, 384–386
methodology. *See* Disks methodology
models. *See* Disks models
non-data-transfer disk commands, 432
observability tools. *See* Disks observability tools
read/write ratio, 431
references, 496–498
resource controls, 494
saturation, 434
terminology, 424
tunable, 494
tuning, 493–495
USE method, 451
utilization, 433
visualizations, 487–490
Disks architecture
interfaces, 442–443
magnetic rotational disks, 435–439
operating system disk I/O stack, 446–449
persistent memory, 441
solid-state drives, 439–441
storage types, 443–446
Disks I/O
vs. application I/O, 435
`bpftrace` for, 764, 806–807
caching, 430
errors, 483
heat maps, 488–490
latency, 428–430, 454–455, 467–472, 482–483
operating system stacks, 446–449
OS virtualization, 613, 616
OS virtualization strategy, 630
random vs. sequential, 430–431
scatter plots, 488
simple disk, 425
size, 432, 480–481
synchronous vs. asynchronous, 434–435
time measurements, 427–429
time scales, 429–430
wait, 434
Disks methodology
cache tuning, 456
latency analysis, 454–455
micro-benchmarking, 456–457
overview, 449–450
performance monitoring, 452
resource controls, 456
scaling, 457–458
static performance tuning, 455–456

tools method, 450
USE method, 450–451
workload characterization, 452–454
Disks models
caching disk, 425–426
controllers, 426
simple disk, 425
Disks observability tools, 484–486
`biolatency`, 468–470
`biosnoop`, 470–472
`biostacks`, 474–475
`biotop`, 473–474
`blktrace`, 475–479
`bpftrace`, 479–483
`iostat`, 459–463
`iotop`, 472–473
MegaCli, 484
miscellaneous, 487
overview, 458–459
`perf`, 465–468
`pidstat`, 464–465
PSI, 464
`sar`, 463–464
SCSI event logging, 486
`diskstats` tool, 142, 487
Dispatcher-queue latency, 222
Distributed operating systems, 123–124
Distributed tracing, 199
Distributions
multimodal, 76–77
normal, 75
`dmesg` tool
CPUs, 245
description, 15
memory, 348
OS virtualization, 619
`dmidecode` tool, 348–349
DNLC (directory name lookup cache), 375
DNS latency, 24–25
Docker 607, 620–622
Documentation
application latency, 385
BCC, 760–761
`bpftrace`, 781
Ftrace, 748–749
kprobes, 153
`perf`, 276, 703
perf-tools, 748
PMCs, 158
`sar`, 165–166
`trace-cmd`, 740
tracepoints, 150–151
uprobes, 155
USDT, 156
Domains
scheduling, 244
Xen, 589
Double data rate synchronous dynamic random-access memory (DDR SDRAM), 313
Double-pumped data transfer for CPUs, 237
DPDK (Data Plane Development Kit), 523
DRAM (dynamic random-access memory), 311
Drill-down analysis
overview, 55–56
slow disks case study, 17
Drivers
balloon, 597
device, 109–110, 522
parameterized, 593–595
`drsnoop` tool
BCC, 756
memory, 342
DSCPs (Differentiated Services Code Points), 509–510
DTrace tool
description, 12
Solaris kernel, 114
Duplex for networks, 508

Duplicate ACK detection, 512

Duration in RED method, 53

DWARF (debugging with attributed record formats) stack walking, 216, 267, 676, 696

Dynamic instrumentation
- kprobes, 151
- latency analysis, 385
- overview, 12

Dynamic priority in scheduling classes, 242–243

Dynamic random-access memory (DRAM), 311

Dynamic sizing in cloud computing, 583–584

Dynamic tracers, 12

Dynamic tracing
- DTrace, 114
- `perf`, 677–678
- tools, 12

Dynamic USDT, 156

DynTicks, 116

E

`e2fsck` tool, 418

Early Departure Time (EDT), 119, 524

eBPF. *See* Extended BPF

EBS (Elastic Block Store), 585

ECC (error-correcting code) for magnetic rotational disks, 438

ECN (Explicit Congestion Notification) field
- IP, 508–510
- TCP, 513
- tuning, 570

EDT (Early Departure Time), 119, 524

EFS (Elastic File System), 585

EKS (Elastic Kubernetes Service), 586

`elasped` variable in `bpftrace`, 777

Elastic Block Store (EBS), 585

Elastic File System (EFS), 585

Elastic Kubernetes Service (EKS), 586

Elevator seeking in magnetic rotational disks, 437–438

ELF (Executable and Linking Format) binaries
- description, 183
- missing symbols in, 214

Embedded caches, 232

eMLC (enterprise multi-level cell) flash memory, 440

Encapsulation for networks, 504

`END` probes in `bpftrace`, 774

End-to-end network arguments, 507

Enterprise models, 62

Enterprise multi-level cell (eMLC) flash memory, 440

Environment
- benchmarking, 647
- processes, 101–102

Ephemeral drives, 584

Ephemeral ports, 531

`epoll` system call, 115, 118

EPTs (extended page tables), 593

Erlang virtual machines, 185

Error-correcting code (ECC) for magnetic rotational disks, 438

Errors
- applications, 193
- benchmarking, 647
- CPUs, 245–246, 796, 798
- disk controllers, 451
- disk devices, 451
- I/O, 483, 798
- kernels, 798
- memory, 324–325, 796, 798
- networks, 526–527, 529, 796–797
- RED method, 53
- storage, 797
- task capacity, 799
- USE method overview, 47–48, 51–53
- user mutex, 799

Ethernet congestion avoidance, 508

ethtool tool, 132, 546–547
Event-based concurrency, 178
Event-based tools, 133
Event-select MSRs, 238
Event sources for Wireshark, 559
Event tracing
- disks, 454
- file systems, 388
- Ftrace, 707–708
- kprobes, 719–720
- methodologies, 57–58
- perf-tools for, 745–746
- trace-cmd for, 737
- uprobes, 722–723

Event worker threads, 178
Events
- case study, 789–790
- CPUs, 273–274
- frequency sampling, 682–683
- observability source, 159
- perf. *See* perf tool events
- SCSI logging, 486
- selecting, 274–275
- stat filters, 693–694
- synthetic, 731–733
- trace, 148

events directory in tracefs, 710
Eviction policies for caching, 36
evlist subcommand for perf, 673
Exceptions
- synchronous interrupts, 97
- user mode, 93

Exclusive CPU sets, 298
exec system calls
- kernel, 94
- processes, 100

execsnoop tool
- BCC, 756
- CPUs, 285
- perf-tools, 743
- process tracing, 207–208
- static instrumentation, 11–12
- tracing, 136

Executable and Linking Format (ELF) binaries
- description, 183
- missing symbols in, 214

Executable data in process virtual address space, 319
Executable text in process virtual address space, 319
Execution in kernels, 92–93
execve system call, 11
exit function in bpftrace, 770, 779
Experimentation-based performance gains, 73–74
Experiments
- CPUs, 293–294
- disks, 490–493
- file systems, 411–414
- networks, 562–567
- observability, 7
- overview, 13–14
- scientific method, 45–46

Experts for applications, 173
Explicit Congestion Notification (ECN) field
- IP, 508–510
- TCP, 513
- tuning, 570

Explicit logical metadata in file systems, 368
Exporters for monitoring, 55, 79, 137
Express Data Path (XDP) technology
- description, 118
- event sources, 558
- kernel bypass, 523

ext3 file system, 378–379
ext4 file system
- features, 379
- tuning, 416–418

ext4dist tool, 399–401, 756

`ext4slower` tool, 401–402, 756
Extended BPF, 12
BCC 751–761
bpftrace 752–753, 761–781, 803–808
description, 118
firewalls, 517
histograms, 744
kernel-mode applications, 92
overview, 121–122
tracing tools, 166
Extended page tables (EPTs), 593
Extent-based file systems, 375–376
Extents, 375–376
btrfs, 382
ext4, 380
External caches, 232

F

FaaS (functions as a service), 634
FACK (forward acknowledgments) in TCP, 514
Factor analysis in capacity planning, 71–72
Failures, benchmarking, 645–651
Fair-share schedulers, 595
False sharing for hash tables, 181
Families of instance types, 581
Fast File System (FFS)
description, 113
overview, 377–378
Fast open in TCP, 510
Fast recovery in TCP, 510
Fast retransmits in TCP, 510, 512
Fast user-space mutex (Futex), 115
Fastpath state in Mutex locks, 179
`fatrace` tool, 395–396
Faults
in synchronous interrupts, 97
page faults. *See* page faults
`faults` tool, 348
FC (Fibre Channel) interface, 442–443
`fd` tool, 141
Feedback-directed optimization (FDO), 122
`ffaults` tool, 348
FFS (Fast File System)
description, 113
overview, 377–378
Fiber threads, 178
Fibre Channel (FC) interface, 442–443
Field-programmable gate arrays (FPGAs), 240–241
FIFO scheduling policy, 243
File descriptor capacity in USE method, 52
File offset pattern, micro-benchmarking for, 390
File stores in cloud computing, 584
File system internals, `bpftrace` for, 408
File systems
access timestamps, 371
ad hoc tools, 411–412
architecture. *See* File systems architecture
`bpftrace` for, 764, 805–806
caches. *See* File systems caches
capacity, OS virtualization, 616
capacity, performance issues, 371
exercises, 419–420
experiments, 411–414
hardware virtualization, 597
I/O, logical vs. physical, 368–370
I/O, non-blocking, 366–367
I/O, random vs. sequential, 363–364
I/O, raw and direct, 366
I/O, stack, 107–108
interfaces, 361
latency, 362–363
memory-mapped files, 367
metadata, 367–368
methodology. *See* File systems methodology
micro-benchmark tools, 412–414

models, 361–362
observability tools. *See* File systems observability tools
operations, 370–371
OS virtualization, 611–612
overview, 106–107, 359–360
paging, 306
prefetch, 364–365
read-ahead, 365
reads, micro-benchmarking for, 61
record size tradeoffs, 27
references, 420–421
special, 371
synchronous writes, 366
terminology, 360
tuning, 414–419
types. *See* File systems types
visualizations, 410–411
volumes and pools, 382–383

File systems architecture
caches, 373–375
features, 375–377
I/O stacks, 107–108, 372
VFS, 107, 373

File systems caches, 361–363
defined, 360
flushing, 414
hit ratio, 17
OS virtualization, 616
OS virtualization strategy, 630
tuning, 389
usage, 309
write-back, 365

File systems methodology
cache tuning, 389
disk analysis, 384
latency analysis, 384–386
micro-benchmarking, 390–391
overview, 383–384
performance monitoring, 388
static performance tuning, 389
workload characterization, 386–388
workload separation, 389

File systems observability tools
`bpftrace`, 402–408
`cachestat`, 399
`ext4dist`, 399–401
`ext4slower`, 401–402
`fatrace`, 395–396
`filetop`, 398–399
`free`, 392–393
`LatencyTOP`, 396
miscellaneous, 409–410
`mount`, 392
`opensnoop`, 397
overview, 391–392
`sar`, 393–394
`slabtop`, 394–395
`strace`, 395
`top`, 393
`vmstat`, 393

File systems types
btrfs, 381–382
ext3, 378–379
ext4, 379
FFS, 377–378
XFS, 379–380
ZFS, 380–381

FileBench tool, 414
`filelife` tool, 409, 756
`fileslower` tool, 409
`filetop` tool, 398–399
`filetype` tool, 409

Filters
`bpftrace`, 769, 776
event, 693–694
kprobes, 721–722
PID, 729–730
tracepoints, 717–718
uprobes, 723

fio (Flexible IO Tester) tool
disks, 493
file systems, 413–414
Firecracker project, 631
Firewalls, 503
misconfigured, 505
overview, 517
tuning, 574
First-byte latency, 506, 528
Five Whys in drill-down analysis, 56
Fixed counters, 133–135
Flame graphs
automated, 201
characteristics, 290–291
colors, 291
CPU profiling, 10–11, 187–188, 278, 660–661
generating, 249, 270–272
interactivity, 291
interpretation, 291–292
`malloc()` bytes, 346
missing stacks, 215
off-CPU time, 190–191, 205
overview, 289–290
page faults, 340–342, 346
`perf`, 119
performance wins, 250
profiles, 278
sample processing, 249–250
scripts, 700
FlameScope tool, 292–293, 700
Flash-memory-based SSDs, 439–440
Flash translation layer (FTL) in solid-state drives, 440–441
Flent (FLExible Network Tester) tool, 567
Flexible IO Tester (fio) tool
disks, 493
file systems, 413–414
FLExible Network Tester (Flent) tool, 567
Floating point events in `perf`, 680
floating-point operations per second (FLOPS) in benchmarking, 655
Flow control in `bpftrace`, 775–777
Flusher threads, 374
Flushing caches, 365, 414
`fmapfault` tool, 409
Footprints, off-CPU, 188–189
fork system calls, 94, 100
`forks.bt` tool, 624–625
Format string for tracepoints, 148–149
Forward acknowledgments (FACK) in TCP, 514
4-wide processors, 224
FPGAs (field-programmable gate arrays), 240–241
Fragmentation
FFS, 377
file systems, 364
memory, 321
packets, 505
reducing, 380
Frames
defined, 500
networks, 515
OSI model, 502
Free memory lists, 315–318
`free` tool
description, 15
file systems, 392–393
memory, 348
OS virtualization, 619
FreeBSD
jails, 606
jemalloc, 322
kernel, 113
TSA analysis, 217
network stack, 514
performance vs. Linux, 124
TCP LRO, 523
Freeing memory, 315–318

Frequency sampling for hardware events, 682–683
Front-ends in instruction pipeline, 224
Front-side buses, 235–237
fsck time in ext4, 379
`fsrwstat` tool, 409
FTL (flash translation layer) in solid-state drives, 440–441
`ftrace` subcommand for `perf`, 673
Ftrace, 13, 705–706
- capabilities overview, 706–708
- description, 166
- documentation, 748–749
- function_graph, 724–725
- function profiler, 711–712
- function tracer, 713–716
- hist triggers, 727–733
- hwlat, 726
- kprobes, 719–722
- options, 716
- OS virtualization, 629
- `perf`, 741
- perf-tools, 741–748
- references, 749
- `trace-cmd`, 734–740
- trace file, 713–715
- trace_pipe file, 715
- tracefs, 708–711
- tracepoints, 717–718
- tracing, 136
- uprobes, 722–723

Full I/O distributions disk latency, 454
Full stack in systems performance, 1
Fully associative caches, 234
Fully-preemptible kernels, 110, 114
`func` variable in `bpftrace`, 778
`funccount` tool
- BCC, 756–758
- example, 747
- perf-tools, 744, 748

`funcgraph` tool
- Ftrace, 706–707
- perf-tools, 744, 748

`funclatency` tool, 757
`funcslower` tool
- BCC, 757
- perf-tools, 744

function_graph tracer
- description, 708
- graph tracing, 724–725
- options, 725
- `trace-cmd` for, 737, 739

function_profile_enabled file, 710
Function profiling
- Ftrace, 707, 711–712
- observability source, 159

Function tracer. *See* Ftrace tool
Function tracing
- profiling, 248
- `trace-cmd` for, 736–737

Functional block diagrams in USE method, 49–50
Functional units in CPUs, 223
Functions as a service (FaaS), 634
Functions in `bpftrace`, 770, 778–781
`functrace` tool, 744
Futex (fast user-space mutex), 115
`futex` system calls, 95

G

Garbage collection, 185–186
gcc compiler
- optimizations, 183–184
- PGO kernels, 122

gdb tool, 136
Generic segmentation offload (GSO) in networks, 520–521
Generic system performance methodologies, 40–41
Geometric mean, 74

`getdelays.c` tool, 286
`gethostlatency` tool, 561, 756
github.com tool package, 132
GKE (Google Kubernetes Engine), 586
`glibc` allocator, 322
Glossary of terms, 815–823
Golang
goroutines, 178
syscalls, 92
Good/fast/cheap trade-offs, 26–27
Google Kubernetes Engine (GKE), 586
Goroutines for applications, 178
`gprof` tool, 135
Grafana, 8-9, 138
Graph tracing, 724–725
Graphics processing units (GPUs)
vs. CPUs, 240
tools, 287
GRO (Generic Receive Offload), 119
Growth
big O notation, 175
heap, 320
memory, 185, 316, 327
GSO (generic segmentation offload) in networks, 520–521
Guests
hardware virtualization, 590–593, 596–605
lightweight virtualization, 632–633
OS virtualization, 617, 627–629
gVisor project, 631

H

Hard disk drives (HDDs), 435–439
Hard interrupts, 282
`hardirqs` tool, 282, 756
Hardware
memory, 311–315
networks, 515–517
threads, 220
tracing, 276
Hardware-assisted virtualization, 590
Hardware counters. *See* Performance monitoring counters (PMCs)
Hardware events
CPUs, 273–274
frequency sampling, 682–683
`perf`, 680–683
selecting, 274–275
Hardware instances in cloud computing, 580
Hardware interrupts, 91
Hardware latency detector (hwlat), 708, 726
Hardware latency tracer, 118
Hardware probes, 774
Hardware RAID, 444
Hardware resources in capacity planning, 70
Hardware virtualization
comparisons, 634–636
CPU support, 589–592
I/O, 593–595
implementation, 588–589
memory mapping, 592–593
multi-tenant contention, 595
observability, 597–605
overhead, 589–595
overview, 587–588
resource controls, 595–597
Harmonic mean, 74
Hash fields in hist triggers, 728
Hash tables in applications, 180–181
HBAs (host bus adapters), 426
HDDs (hard disk drives), 435–439
`hdparm` tool, 491–492
Head-based sampling in distributed tracing, 199
Heads in magnetic rotational disks, 436

Heap
anonymous paging, 306
description, 304
growth, 320
process virtual address space, 319
Heat maps
CPU utilization, 288–289
disk offset, 489–490
disk utilization, 490
file systems, 410–411
FlameScope, 292–293
I/O latency, 488–489
overview, 82–83
subsecond-offset, 289
Hello, World! program, 770
`hfaults` tool, 348
`hist` function in `bpftrace`, 780
Hist triggers
fields, 728–729
modifiers, 729
multiple keys, 730
perf-tools, 748
PID filters, 729–730
single keys, 727–728
stack trace keys, 730–731
synthetic events, 731–733
usage, 727
hist triggers profiler, 707
Histogram, 76–77
Hits, cache, 35–36, 361
Hold times for locks, 198
Holistic approach, 6
Horizontal pod autoscalers (HPAs), 73
Horizontal scaling and scalability
capacity planning, 72
cloud computing, 581–582
Host bus adapters (HBAs), 426
Hosts
applications, 172
cloud computing, 580
hardware virtualization, 597–603
lightweight virtualization, 632
OS virtualization, 617, 619–627
Hot caches, 37
Hot/cold flame graphs, 191
Hourly patterns, monitoring, 78
HPAs (horizontal pod autoscalers), 73
HT (HyperTransport) for CPUs, 236
`htop` tool, 621
HTTP/3 protocol, 515
Hubs in networks, 516
Hue in flame graphs, 291
Huge pages, 115–116, 314, 352–353
hugetlb control group, 610
hwlat (hardware latency detector), 708, 726
Hybrid clouds, 580
Hybrid kernels, 92, 123
Hyper-Threading Technology, 225
Hyper-V, 589
Hypercalls in paravirtualization, 588
Hyperthreading-aware scheduling classes, 243
HyperTransport (HT) for CPUs, 236
Hypervisors
cloud computing, 580
hardware virtualization, 587–588
kernels, 93
Hypothesis step in scientific method, 44–45

I

I/O. *See* Input/output (I/O)
IaaS (infrastructure as a service), 580
Icicle graphs, 250
`icstat` tool, 409
IDDs (isolated driver domains), 596
Identification in drill-down analysis, 55
Idle memory, 315
Idle scheduling class, 243
IDLE scheduling policy, 243

Idle state in thread state analysis, 194, 196–197
Idle threads, 99, 244
`ieee80211scan` tool, 561
If statements, 776
`ifconfig` tool, 537–538
`ifpps` tool, 561
`iftop` tool, 562
Implicit disk I/O, 369
Implicit logical metadata, 368
Inactive pages in page caches, 318
Incast problem in networks, 524
Index nodes (inodes)
- caches, 375
- defined, 360
- VFS, 373

Indirect disk I/O, 369
Individual synchronous writes, 366
Industry benchmarking, 60–61
Industry standards for benchmarking, 654–655
Inflated disk I/O, 369
Infrastructure as a service (IaaS), 580
init process, 100
Initial window in TCP, 514
`inject` subcommand for `perf`, 673
Inodes (index nodes)
- caches, 375
- defined, 360
- VFS, 373

inotify framework, 116
`inotify` tool, 409
Input
- event tracing, 58
- solid-state drive controllers, 440

Input/output (I/O)
- disks. *See* Disks I/O
- file systems, 360
- hardware virtualization, 593–595, 597
- I/O-bound applications, 106
- latency, 424
- logical vs. physical, 368–370
- merging, 448
- multiqueue schedulers, 119
- non-blocking, 181, 366–367
- OS virtualization, 611–612, 616–617
- random vs. sequential, 363–364
- raw and direct, 366
- request time, 427
- schedulers, 448
- scheduling, 115–116
- service time, 427
- size, applications, 176
- size, micro-benchmarking, 390
- stacks, 107–108, 372
- USE method, 798
- wait time, 427

Input/output operations per second. *See* IOPS (input/output operations per second)
Input/output profiling
- `bpftrace`, 210–212
- `perf`, 202–203
- syscall analysis, 192

Installing
- BCC, 754
- `bpftrace`, 762

instances directory in tracefs, 710
Instances in cloud computing
- description, 14
- types, 580

Instruction pointer for threads, 100
Instructions, CPU
- defined, 220
- IPC, 225
- pipeline, 224
- size, 224
- steps, 223
- text, 304
- width, 224

Instructions per cycle (IPC), 225, 251, 326
Integrated caches, 232
Intel Cache Allocation Technology (CAT), 118, 596
Intel Clear Containers, 631
Intel processor cache sizes, 230–231
Intel VTune Amplifier XE tool, 135
Intelligent Platform Management Interface (IPMI), 98–99
Intelligent prefetch in ZFS, 381
Inter-processor interrupts (IPIs), 110
Inter-stack latency in networks, 529
Interactivity in flame graphs, 291
Interconnects
- buses, 313
- CPUs, 235–237
- USE method, 49–51

Interfaces
- defined, 500
- device drivers, 109–110
- disks, 442–443
- file systems, 361
- kprobes, 153
- network, 109, 501
- network hardware, 515–516
- network IOPS, 527–529
- network negotiation, 508
- PMCs, 157–158
- scheduling in NAPI, 522
- tracepoints, 149–150
- uprobes, 154–155

Interleaving in FFS, 378
Internet Protocol (IP)
- congestion avoidance, 508
- overview, 509–510
- sockets, 509

Interpretation of flame graphs, 291–292
Interpreted programming languages, 184–185
Interrupt coalescing mode for networks, 522
Interrupt-disabled mode, 98
Interrupt service requests (IRQs), 96–97
Interrupt service routines (ISRs), 96
Interrupts
- asynchronous, 96–97
- defined, 91
- hardware, 282
- masking, 98–99
- network latency, 529
- overview, 96
- soft, 281–282
- synchronous, 97
- threads, 97–98

`interrupts` tool, 142
`interval` probes in `bpftrace`, 774
Interval statistics, `stat` for, 693
IO accounting, 116
`io_submit` command, 181
`io_uring_enter` command, 181
io_uring interface, 119
`ioctl` system calls, 95
`iolatency` tool, 743
`ionice` tool, 493–494
`ioping` tool, 492
`ioprofile` tool, 409
IOPS (input/output operations per second)
- defined, 22
- description, 7
- disks, 429, 431–432
- networks, 527–529
- performance metric, 32
- resource analysis, 38

`iosched` tool, 487
`iosnoop` tool, 743
`iostat` tool
- bonnie++ tool, 658
- default output, 459–460
- description, 15
- disks, 450, 459–463

extended output, 460–463
fixed counters, 134
memory, 348
options, 460
OS virtualization, 619, 627
percent busy metric, 33
slow disks case study, 17

iotop tool, 450, 472–473

IP (Internet Protocol)
congestion avoidance, 508
overview, 509–510
sockets, 509

ip tool, 525, 536–537

ipc control group, 608

IPC (instructions per cycle), 225, 251, 326

ipecn tool, 561

iperf tool
example, 13–14
network micro-benchmarking, 10
network throughput, 564–565

IPIs (inter-processor interrupts), 110

IPMI (Intelligent Platform Management Interface), 98–99

iproute2 tool package, 132

IRQs (interrupt service requests), 96–97

irqsoff tracer, 708

iscpu tool, 285

Isolated driver domains (IDDs), 596

Isolation in OS virtualization, 629

ISRs (interrupt service routines), 96

istopo tool, 286

J

Jails in BSD kernel, 113, 606

Java
analysis, 29
case study, 783–792
flame graphs, 201, 271
dynamic USDT, 156, 213
garbage colleciton, 185–186
Java Flight Recorder, 135
stack traces, 215
symbols, 214
uprobes, 213
USDT probes, 155, 213
virtual machines, 185

Java Flight Recorder (JFR), 135

JavaScript Object Notation (JSON) format, 163–164

JBOD (just a bunch of disks), 443

jemalloc allocator, 322

JFR (Java Flight Recorder), 135

JIT (just-in-time) compilation
Linux kernel, 117
PGO kernels, 122
runtime missing symbols, 214

Jitter in operating systems, 99

jmaps tool, 214

join function, 778

Journaling
btrfs, 382
ext3, 378–379
file systems, 376
XFS, 380

JSON (JavaScript Object Notation) format, 163–164

Jumbo frames
packets, 505
tuning, 574

Just a bunch of disks (JBOD), 443

Just-in-time (JIT) compilation
Linux kernel, 117
PGO kernels, 122
runtime missing symbols, 214

K

kaddr function, 779

Kata Containers, 631

KCM (Kernel Connection Multiplexor), 118

Keep-alive strategy in networks, 507

Kendall's notation for queueing systems, 67–68

Kernel-based Virtual Machine (KVM) technology

- CPU quotas, 595
- description, 589
- I/O path, 594
- Linux kernel, 116
- observability, 600–603

Kernel bypass for networks, 523

Kernel Connection Multiplexor (KCM), 118

Kernel mode, 93

Kernel page table isolation (KPTI) patches, 121

Kernel space, 90

Kernel state in thread state analysis, 194–197

Kernel statistics (Kstat) framework, 159–160

Kernel time

- CPUs, 226
- syscall analysis, 192

Kernels

- `bpftrace` for, 765
- BSD, 113
- comparisons, 124
- defined, 90
- developments, 115–120
- execution, 92–93
- file systems, 107
- filtering in OS virtualization, 629
- Linux, 114–122, 124
- microkernels, 123
- monolithic, 123
- overview, 91–92
- PGO, 122
- PMU events, 680
- preemption, 110
- schedulers, 105–106
- Solaris, 114
- stacks, 103
- system calls, 94–95
- time analysis, 202
- unikernels, 123
- Unix, 112
- USE method, 798
- user modes, 93–94
- versions, 111–112

KernelShark software, 83–84, 739–740

`kfunc` probes, 774

`killsnoop` tool

- BCC, 756
- perf-tools, 743

`klockstat` tool, 756

`kmem` subcommand for `perf`, 673, 702

Knee points

- models, 62–64
- scalability, 31

Known-knowns, 37

Known-unknowns, 37

kprobe_events file, 710

`kprobe` probes, 774

kprobe profiler, 707

`kprobe` tool, 744

kprobes, 685–686

- arguments, 686–687, 720–721
- event tracing, 719–720
- filters, 721–722
- overview, 151–153
- profiling, 722
- return values, 721
- triggers, 721–722

kprobes tracer, 708

KPTI (kernel page table isolation) patches, 121

`kretfunc` probes, 774

kretprobes, 152–153, 774

`kstack` function in `bpftrace`, 779

`kstack` variable in `bpftrace`, 778

Kstat (kernel statistics) framework, 159–160

`kswapd` tool, 318–319, 374

ksym function, 779
kubectl command, 621
Kubernetes
node, 608
orchestration, 586
OS virtualization, 620–621
KVM. *See* Kernel-based Virtual Machine (KVM) technology
kvm_entry tool, 602
kvm_exit tool, 602
kvm subcommand for perf, 673, 702
kvm_vcpu_halt command, 592
kvmexits.bt tool, 602–603
Kyber multi-queue schedulers, 449

L

L2ARC cache in ZFS, 381
Label selectors in cloud computing, 586
Language virtual machines, 185
Large Receive Offload (LRO), 116
Large segment offload for packet size, 505
Last-level caches (LLCs), 232
Latency
analysis methodologies, 56–57
applications, 173
biolatency, 468–470
CPUs, 233–234
defined, 22
disk I/O, 428–430, 454–455, 467–472, 482–483
distributions, 76–77
file systems, 362–363, 384–386, 388
graph tracing, 724–725
hardware, 118
hardware virtualization, 604
heat maps, 82–83, 488–489
I/O profiling, 210–211
interrupts, 98
line charts, 80–81
memory, 311, 441
methodologies, 24–25
networks, analysis, 528–529
networks, connections, 7, 24–25, 505–506, 528
networks, defined, 500
networks, types, 505–507
outliers, 58, 186, 424, 471–472
overview, 6–7
packets, 532–533
percentiles, 413–414
perf, 467–468
performance metric, 32
run-queue, 222
scatter plots, 81–82, 488
scheduler, 226, 272–273
solid-state drives, 441
ticks, 99
transaction costs analysis, 385–386
VFS, 406–408
workload analysis, 39–40
LatencyTOP tool for file systems, 396
latencytop tool for operating systems, 116
Lazy shootdowns, 367
LBR (last branch record), 216, 676, 696
Leak detection for memory, 326–327
Least frequently used (LFU) caching algorithm, 36
Least recently used (LRU) caching algorithm, 36
Level 1 caches
data, 232
instructions, 232
memory, 314
Level 2 ARC, 381
Level 2 caches
embedded, 232
memory, 314
Level 3 caches
LLC, 232
memory, 314

Level of appropriateness in methodologies, 28–29

LFU (least frequently used) caching algorithm, 36

`lhist` function, 780

libpcap library as observability source, 159

Life cycle for processes, 100–101

Life span

- network connections, 507
- solid-state drives, 441

Lightweight threads, 178

Lightweight virtualization

- comparisons, 634–636
- implementation, 631–632
- observability, 632–633
- overhead, 632
- overview, 630
- resource controls, 632

Limit investigations, benchmarking for, 642

Limitations of averages, 75

Limits for OS virtualization resources, 613

`limits` tool, 141

Line charts

- baseline statistics, 59
- disks, 487–488
- working with, 80–81

Linear scalability

- methodologies, 32
- models, 63

Link aggregation tuning, 574

Link-time optimization (LTO), 122

Linux 60-second analysis, 15–16

Linux operating system

- crisis tools, 131–133
- extended BPF, 121–122
- kernel developments, 115–120
- KPTI patches, 121
- network stacks, 518–519
- observability sources, 138–146
- observability tools, 130
- operating system disk I/O stack, 447–448
- overview, 114–115
- static performance tools, 130–131
- systemd service manager, 120
- thread state analysis, 195–197

linux-tools-common linux-tools tool package, 132

`list` subcommand

- `perf`, 673
- `trace-cmd`, 735

Listen backlogs in networks, 519

`listen` subcommand in `trace-cmd`, 735

Listing events

- `perf`, 674–675
- `trace-cmd` for, 736

Little's Law, 66

Live reporting in `sar`, 165

LLCs (last-level caches), 232

`llcstat` tool

- BCC, 756
- CPUs, 285

Load averages for `uptime`, 255–257

Load balancers

- capacity planning, 72
- schedulers, 241

Load generation

- capacity planning, 70
- custom load generators, 491
- micro-benchmarking, 61

Load vs. architecture in methodologies, 30–31

`loadavg` tool, 142

Local memory, 312

Local network connections, 509

Localhost network connections, 509

Lock state in thread state analysis, 194–197

`lock` subcommand for `perf`, 673, 702

Locks

- analysis, 198
- applications, 179–181
- tracing, 212–213

Logging
- applications, 172
- SCSI events, 486
- ZFS, 381

Logical CPUs
- defined, 220
- hardware threads, 221

Logical I/O
- defined, 360
- vs. physical, 368–370

Logical metadata in file systems, 368
Logical operations in file systems, 361
Longest-latency caches, 232
Loopbacks in networks, 509
Loops in `bpftrace`, 776–777
LRO (Large Receive Offload), 116
LRU (least recently used) caching algorithm, 36
`lsof` tool, 561
LTO (link-time optimization), 122
LTTng tool, 166

M

M/D/1 queueing systems, 68–69
M/G/1 queueing systems, 68
M/M/1 queueing systems, 68
M/M/c queueing systems, 68
Macro-benchmarks, 13, 653–654
MADV_COLD option, 119
MADV_PAGEOUT option, 119
`madvise` system call, 367, 415–416
Magnetic rotational disks, 435–439
Main memory
- caching, 37–39
- defined, 90, 304
- latency, 26
- managing, 104–105
- overview, 311–312

`malloc()` bytes flame graphs, 346
Map functions in `bpftrace`, 771–772, 780–781
Map variables in `bpftrace`, 771
Mapping memory. *See* Memory mappings
`maps` tool, 141
Marketing, benchmarking for, 642
Markov model, 654
Markovian arrivals in queueing systems, 68–69
Masking interrupts, 98–99
`max` function in `bpftrace`, 780
Maximum controller operation rate, 457
Maximum controller throughput, 457
Maximum disk operation rate, 457
Maximum disk random reads, 457
Maximum disk throughput
- magnetic rotational disks, 436–437
- micro-benchmarking, 457

Maximum transmission unit (MTU) size for packets, 504–505
MCS locks, 117
`mdflush` tool, 487
Mean, 74
"A Measure of Transaction Processing Power," 655
Measuring disk time, 427–429
Medians, 75
MegaCli tool, 484
Melo, Arnaldo Carvalho de, 671
Meltdown vulnerability, 121
`mem` subcommand for `perf`, 673
`meminfo` tool, 142
`memleak` tool
- BCC, 756
- memory, 348

Memory, 303–304
- allocators, 309, 353
- architecture. *See* Memory architecture
- benchmark questions, 667–668
- `bpftrace` for, 763–764, 804–805

BSD kernel, 113
CPU caches, 221–222
CPU tradeoffs with, 27
demand paging, 307–308
exercises, 354–355
file system cache usage, 309
garbage collection, 185
hardware virtualization, 596–597
internals, 346–347
mappings. *See* Memory mappings
methodology. *See* Memory methodology
multiple page sizes, 352–353
multiprocess vs. multithreading, 228
NUMA binding, 353
observability tools. *See* Memory observability tools
OS virtualization, 611, 613, 615–616
OS virtualization strategy, 630
overcommit, 308
overprovisioning in solid-state drives, 441
paging, 306–307
persistent, 441
process swapping, 308–309
references, 355–357
resource controls, 353–354
shared, 310
shrinking method, 328
terminology, 304
tuning, 350–354
USE method, 49–51, 796–798
utilization and saturation, 309
virtual, 90, 104–105, 304–305
word size, 310
working set size, 310

Memory architecture, 311
buses, 312–313
CPU caches, 314
freeing memory, 315–318
hardware, 311–315
latency, 311
main memory, 311–312
MMU, 314
process virtual address space, 319–322
software, 315–322
TLB, 314

memory control group, 610, 616

Memory locality, 222

Memory management units (MMUs), 235, 314

Memory mappings
displaying, 337–338
files, 367
hardware virtualization, 592–593
heap growth, 320
kernel, 94
micro-benchmarking, 390
OS virtualization, 611

Memory methodology
cycle analysis, 326
leak detection, 326–327
memory shrinking, 328
micro-benchmarking, 328
overview, 323
performance monitoring, 326
resource controls, 328
static performance tuning, 327–328
tools method, 323–324
usage characterization, 325–326
USE method, 324–325

Memory observability tools
`bpftrace`, 343–347
`drsnoop`, 342
miscellaneous, 347–350
`numastat`, 334–335
overview, 328–329
`perf`, 338–342
`pmap`, 337–338
`ps`, 335–336
PSI, 330–331
`sar`, 331–333

`slabtop`, 333–334
`swapon`, 331
`top`, 336–337
`vmstat`, 329–330
`wss`, 342–343

Memory reclaim state in delay accounting, 145

Metadata
ext3, 378
file systems, 367–368

Method R, 57

Methodologies, 21–22
ad hoc checklist method, 43–44
anti-methods, 42–43
applications. *See* Applications methodology
baseline statistics, 59
benchmarking. *See* Benchmarking methodology
cache tuning, 60
caching, 35–37
capacity planning, 69–73
CPUs. *See* CPUs methodology
diagnosis cycle, 46
disks. *See* Disks methodology
drill-down analysis, 55–56
event tracing, 57–58
exercises, 85–86
file systems. *See* File systems methodology
general, 40–41
known-unknowns, 37
latency analysis, 56–57
latency overview, 24–25
level of appropriateness, 28–29
Linux 60-second analysis checklist, 15–16
load vs. architecture, 30–31
memory. *See* Memory methodology
Method R, 57
metrics, 32–33
micro-benchmarking, 60–61
modeling. *See* Methodologies modeling
models, 23–24
monitoring, 77–79
networks. *See* Networks methodology
performance, 41–42
performance mantras, 61
perspectives, 37–40
point-in-time recommendations, 29–30
problem statement, 44
profiling, 35
RED method, 53
references, 86–87
resource analysis, 38–39
saturation, 34–35
scalability, 31–32
scientific method, 44–46
static performance tuning, 59–60
statistics, 73–77
stop indicators, 29
terminology, 22–23
time scales, 25–26
tools method, 46
trade-offs, 26–27
tuning efforts, 27–28
USE method, 47–53
utilization, 33–34
visualizations. *See* Methodologies visualizations
workload analysis, 39–40
workload characterization, 54

Methodologies modeling, 62
Amdahl's Law of Scalability, 64–65
enterprise vs. cloud, 62
queueing theory, 66–69
Universal Scalability Law, 65–66
visual identification, 62–64

Methodologies visualizations, 79
heat maps, 82–83
line charts, 80–81
scatter plots, 81–82

surface plots, 84–85
timeline charts, 83–84
tools, 85
Metrics, 8–9
applications, 172
fixed counters, 133–135
methodologies, 32–33
observability tools, 167–168
resource analysis, 38
USE method, 48–51
workload analysis, 40
MFU (most frequently used) caching algorithm, 36
Micro-benchmarking
capacity planning, 70
CPUs, 253–254
description, 13
design example, 652–653
disks, 456–457, 491–492
file systems, 390–391, 412–414
memory, 328
methodologies, 60–61
networks, 533
overview, 651–652
Micro-operations (uOps), 224
Microcode ROM in CPUs, 230
Microkernels, 92, 123
Microservices
cloud computing, 583–584
USE method, 53
Midpath state for Mutex locks, 179
Migration types for free lists, 317
`min` function in `bpftrace`, 780
MINIX operating system, 114
Minor faults, 307
MIPS (millions of instructions per second) in benchmarking, 655
Misleading benchmarks, 650
Missing stacks, 215–216
Missing symbols, 214
Mixed-mode CPU profiles, 187
Mixed-mode flame graphs, 187
MLC (multi-level cell) flash memory, 440
`mmap` sys call
description, 95
memory mapping, 320, 367
`mmapfiles` tool, 409
`mmapsnoop` tool, 348
mmiotrace tracer, 708
MMUs (memory management units), 235, 314
mnt control group, 609
Mode switches
defined, 90
kernels, 93
Model-specific registers (MSRs)
CPUs, 238
observability source, 159
Models
Amdahl's Law of Scalability, 64–65
CPUs, 221–222
disks, 425–426
enterprise vs. cloud, 62
file systems, 361–362
methodologies, 23–24
networks, 501–502
overview, 62
queueing theory, 66–69
Universal Scalability Law, 65–66
visual identification, 62–64
wireframe, 84–85
Modular I/O scheduling, 116
Monitoring, 77–79
CPUs, 251
disks, 452
drill-down analysis, 55
file systems, 388
memory, 326
networks, 529, 537
observability tools, 137–138

products, 79
sar, 161–162
summary-since-boot values, 79
time-based patterns, 77–78
Monolithic kernels, 91, 123
Most frequently used (MFU) caching algorithm, 36
Most recently used (MRU) caching algorithm, 36
Mount points in file systems, 106
mount tool
file systems, 392
options, 416–417
Mounting file systems, 106, 392
mountsnoop tool, 409
mpstat tool
case study, 785–786
CPUs, 245, 259
description, 15
fixed counters, 134
lightweight virtualization, 633
OS virtualization, 619
mq-deadline multi-queue schedulers, 449
MR-IOV (multiroot I/O virtualization), 593–594
MRU (most recently used) caching algorithm, 36
MSG_ZEROCOPY flag, 119
msr-tools tool package, 132
MSRs (model-specific registers)
CPUs, 238
observability source, 159
mtr tool, 567
Multi-level cell (MLC) flash memory, 440
Multi-queue schedulers
description, 119
operating system disk I/O stack, 449
Multiblock allocators in ext4, 379
Multicalls in paravirtualization, 588
Multicast network transmissions, 503
Multichannel memory buses, 313
Multics (Multiplexed Information and Computer Services) operating system, 112
Multimodal distributions, 76–77
MultiPath TCP, 119
Multiple causes as performance challenge, 6
Multiple page sizes, 352–353
Multiple performance issues, 6
Multiple prefetch streams in ZFS, 381
Multiple-zone disk recording, 437
Multiplexed Information and Computer Services (Multics) operating system, 112
Multiprocess CPUs, 227–229
Multiprocessors
applications, 177–181
overview, 110
Solaris kernel support, 114
Multiqueue block I/O, 117
Multiqueue I/O schedulers, 119
Multiroot I/O virtualization (MR-IOV), 593–594
Multitenancy in cloud computing, 580
contention in hardware virtualization, 595
contention in OS virtualization, 612–613
overview, 585–586
Multithreading
applications, 177–181
CPUs, 227–229
SMT, 225
Mutex (MUTually EXclusive) locks
applications, 179–180
contention, 198
tracing, 212–213
USE method, 52
MySQL database
bpftrace tracing, 212–213
CPU flame graph, 187–188

CPU profiling, 200, 203, 269–270, 277, 283–284, 697–700
disk I/O tracing, 466–467, 470–471, 488
file tracing, 397–398, 401–402
memory allocation, 345
memory mappings, 337–338
network tracing, 552–554
Off–CPU analysis, 204–205, 275–276
Off–CPU Time flame graphs, 190–192
page fault sampling, 339–341
query latency analysis, 56
scheduler latency, 272, 279–280
shards, 582
slow query log, 172
stack traces, 215
syscall tracing, 201–202
working set size, 342
`mysqld_qslower` tool, 756

N

NAGLE algorithm for TCP congestion control, 513
Name resolution latency, 505, 528
Namespaces in OS virtualization, 606–609, 620, 623–624
NAPI (New API) framework, 522
NAS (network-attached storage), 446
Native Command Queueing (NCQ), 437
Native hypervisors, 587
Negative caching in Dcache, 375
Nested page tables (NPTs), 593
net control group, 609
net_cls control group, 610
Net I/O state in thread state analysis, 194–197
net_prio control group, 610
`net` tool
description, 562
socket information, 142
Netfilter conntrack as observability source, 159
Netflix cloud performance team, 2–3
netlink observability tools, 145–146, 536
`netperf` tool, 565–566
`netsize` tool, 561
`netstat` tool, 525, 539–542
`nettxlat` tool, 561
Network-attached storage (NAS), 446
Network interface cards (NICs)
description, 501–502
network connections, 109
sent and received packets, 522
Networks, 499–500
architecture. *See* Networks architecture
benchmark questions, 668
`bpftrace` for, 764–765, 807–808
buffers, 27, 507
congestion avoidance, 508
connection backlogs, 507
controllers, 501–502
encapsulation, 504
exercises, 574–575
experiments, 562–567
hardware virtualization, 597
interface negotiation, 508
interfaces, 501
latency, 505–507
local connections, 509
methodology. *See* Networks methodology
micro-benchmarking for, 61
models, 501–502
observability tools. *See* Networks observability tools
on-chip interfaces, 230
operating systems, 109
OS virtualization, 611–613, 617, 630
packet size, 504–505
protocol stacks, 502
protocols, 504
references, 575–578
round-trip time, 507, 528

routing, 503
sniffing, 159
stacks, 518–519
terminology, 500
throughput, 527–529
tuning. *See* Networks tuning
USE method, 49–51, 796–797
utilization, 508–509

Networks architecture
hardware, 515–517
protocols, 509–515
software, 517–524

Networks methodology
latency analysis, 528–529
micro-benchmarking, 533
overview, 524–525
packet sniffing, 530–531
performance monitoring, 529
resource controls, 532–533
static performance tuning, 531–532
TCP analysis, 531
tools method, 525
USE method, 526–527
workload characterization, 527–528

Networks observability tools
`bpftrace`, 550–558
`ethtool`, 546–547
`ifconfig`, 537–538
`ip`, 536–537
miscellaneous, 560–562
`netstat`, 539–542
`nicstat`, 545–546
`nstat`, 538–539
overview, 533–534
`sar`, 543–545
`ss`, 534–536
`tcpdump`, 558–559
`tcplife`, 548
`tcpretrans`, 549–550
`tcptop`, 549
Wireshark, 560

Networks tuning, 567
configuration, 574
socket options, 573
system-wide, 567–572

New API (NAPI) framework, 522

New Vegas (NV) congestion control algorithm, 118

`nfsdist` tool
BCC, 756
file systems, 399

`nfsslower` tool, 756

`nfsstat` tool, 561

NFU (not frequently used) caching algorithm, 36

`nice` command
CPU priorities, 252
resource management, 111
scheduling priorities, 295

NICs (network interface cards)
description, 501–502
network connections, 109
sent and received packets, 522

`nicstat` tool, 132, 525, 545–546

"A Nine Year Study of File System and Storage Benchmarking," 643

Nitro hardware virtualization
description, 589
I/O path, 594–595

NMIs (non-maskable interrupts), 98

NO_HZ_FULL option, 117

Node taints in cloud computing, 586

Node.js
dynamic USDT, 156
event-based concurrency, 178
non-blocking I/O, 181
symbols, 214
USDT tracing, 677, 690–691

Nodes
cloud computing, 586
free lists, 317
main memory, 312
Noisy neighbors
multitenancy, 585
OS virtualization, 617
Non-blocking I/O
applications, 181
file systems, 366–367
Non-data-transfer disk commands, 432
Non-idle time, 34
Non-maskable interrupts (NMIs), 98
Non-regression testing
benchmarking for, 642
software change case study, 18
Non-uniform memory access (NUMA)
CPUs, 244
main memory, 312
memory balancing, 117
memory binding, 353
multiprocessors, 110
Non-uniform random distributions, 413
Non-Volatile Memory express (NVMe) interface, 443
Noop I/O schedulers, 448
nop tracer, 708
Normal distribution, 75
NORMAL scheduling policy, 243
Not frequently used (NFU) caching algorithm, 36
NPTs (nested page tables), 593
`nsecs` variable in `bpftrace`, 777
`nsenter` command, 624
`nstat` tool, 134, 525, 538–539
`ntop` function, 779
NUMA. *See* Non-uniform memory access (NUMA)
`numactl` command, 298, 353
`numactl` tool package, 132
`numastat` tool, 334–335
Number of service centers in queueing systems, 67
NV (New Vegas) congestion control algorithm, 118
`nvmelatency` tool, 487

O

O in Big O notation, 175–176
O(1) scheduling class, 243
Object stores in cloud computing, 584
Observability
allocators, 321
applications, 174
benchmarks, 643
counters, statistics, and metrics, 8–9
hardware virtualization, 597–605
operating systems, 111
OS virtualization. *See* OS virtualization observability
overview, 7–8
profiling, 10–11
RAID, 445
tracing, 11–12
volumes and pools, 383
Observability tools, 129
applications. *See* Applications observability tools
coverage, 130
CPUs. *See* CPUs observability tools
crisis, 131–133
disks. *See* Disks observability tools
evaluating results, 167–168
exercises, 168
file system. *See* File systems observability tools
fixed counters, 133–135
memory. *See* Memory observability tools
monitoring, 137–138
network. *See* Networks observability tools

profiling, 135
references, 168–169
sar, 160–166
static performance, 130–131
tracing, 136, 166
types, 133
Observability tools sources, 138–140
delay accounting, 145
hardware counters, 156–158
kprobes, 151–153
miscellaneous, 159–160
netlink, 145–146
/proc file system, 140–143
/sys file system, 143–144
tracepoints, 146–151
uprobes, 153–155
USDT, 155–156
Observation-based performance gains, 73
Observational tests in scientific method, 44–45
Observer effect in metrics, 33
off-CPU
analysis process, 189–192
footprints, 188–189
thread state analysis, 197
time flame graphs, 205
offcputime tool
BCC, 756
description, 285
networks, 561
scheduler tracing, 190
slow disks case study, 17
stack traces, 204–205
time flame graphs, 205
Offset heat maps, 289, 489–490
offwaketime tool, 756
On-chip caches, 231
On-die caches, 231
On-disk caches, 425–426, 430, 437
Online balancing, 382
Online defragmentation, 380
OOM killer (out-of-memory killer), 316–317, 324
OOM (out of memory), defined, 304
oomkill tool
BCC, 756
description, 348
open command
description, 94
non-blocking I/O, 181
Open Container Interface, 586
openat syscalls, 404
opensnoop tool
BCC, 756
file systems, 397
perf-tools, 743
Operating systems, 89
additional reading, 127–128
caching, 108–109
clocks and idle, 99
defined, 90
device drivers, 109–110
disk I/O stack, 446–449
distributed, 123–124
exercises, 124–125
file systems, 106–108
hybrid kernels, 123
interrupts, 96–99
jitter, 99
kernels, 91–95, 111–114, 124
Linux. *See* Linux operating system
microkernels, 123
multiprocessors, 110
networking, 109
observability, 111
PGO kernels, 122
preemption, 110
processes, 99–102

references, 125–127
resource management, 110–111
schedulers, 105–106
stacks, 102–103
system calls, 94–95
terminology, 90–91
tunables for disks, 493–494
unikernels, 123
virtual memory, 104–105
virtualization. *See* OS virtualization
Operation rate
defined, 22
file systems, 387–388
Operations
applications, 172
defined, 360
file systems, 370–371
micro-benchmarking, 390
Operators for `bpftrace`, 776–777
OProfile system profiler, 115
`oprofile` tool, 285
Optimistic spinning in Mutex locks, 179
Optimizations
applications, 174
compiler, 183–184, 229
feedback-directed, 122
networks, 524
Orchestration in cloud computing, 586
Ordered mode in ext3, 378
Orlov block allocator, 379
OS instances in cloud computing, 580
OS virtualization
comparisons, 634–636
control groups, 609–610
implementation, 607–610
namespaces, 606–609
overhead, 610–613
overview, 605–607
resource controls, 613–617
OS virtualization observability
BPF tracing, 624–625
containers, 620–621
guests, 627–629
hosts, 619–627
namespaces, 623–624
overview, 617–618
resource controls, 626–627
strategy, 629–630
tracing tools, 629
traditional tools, 618–619
OS X syscall tracing, 205
OS wait time for disks, 472
OSI model, 502
Out-of-memory killer (OOM killer), 316–317, 324
Out of memory (OOM), defined, 304
Out-of-order packets, 529
Outliers
heat maps, 82
latency, 186, 424, 471–472
normal distributions, 77
Output formats `in sar`, 163–165
Output with solid-state drive controllers, 440
Overcommit strategy, 115
Overcommitted main memory, 305, 308
Overflow sampling
hardware events, 683
PMCs, 157–158
Overhead
hardware virtualization, 589–595
kprobes, 153
lightweight virtualization, 632
metrics, 33
multiprocess vs. multithreading, 228
OS virtualization, 610–613
`strace`, 207
ticks, 99
tracepoints, 150

uprobes, 154–155
volumes and pools, 383
Overlayfs file system, 118
Overprovisioning cloud computing, 583
`override` function, 779
Oversize arenas, 322

P

P-caches in CPUs, 230
P-states in CPUs, 231
Pacing in networks, 524
Packages, CPUs vs. GPUs, 240
Packets
defined, 500
latency, 532–533
networks, 504
OSI model, 502
out-of-order, 529
size, 504–505
sniffing, 530–531
throttling, 522
Padding locks for hash tables, 181
Page caches
file systems, 374
memory, 315
Page faults
defined, 304
flame graphs, 340–342, 346
sampling, 339–340
Page-outs
daemons, 317
working with, 306
Page scanning, 318–319, 323, 374
Page tables, 235
Paged virtual memory, 113
Pages
defined, 304
kernel, 115
sizes, 352–353
Paging
anonymous, 305–307
demand, 307–308
file system, 306
memory, 104–105
overview, 306
PAPI (performance application programming interface), 158
Parallelism in applications, 177–181
Paravirtualization (PV), 588, 590
Paravirtualized I/O drivers, 593–595
Parity in RAID, 445
Partitions in Hyper-V, 589
Passive benchmarking, 656–657
Passive listening in three-way handshakes, 511
`pathchar` tool, 564
Pathologies in solid-state drives, 441
Patrol reads in RAID, 445
Pause frames in congestion avoidance, 508
`pchar` tool, 564
PCI pass-through in hardware virtualization, 593
PCP (Performance Co-Pilot), 138
PE (Portable Executable) format, 183
PEBS (precise event-based sampling), 158
Per-I/O latency values, 454
Per-interval I/O averages latency values, 454
Per-interval statistics with `stat`, 693
Per-process observability tools, 133
fixed counters, 134–135
`/proc` file system, 140–141
profiling, 135
tracing, 136
Percent busy metric, 33
Percentiles
description, 75
latency, 413–414
perf c2c command, 118

perf_event control group, 610
`perf-stat-hist` tool, 744
`perf` tool, 13
case study, 789–790
CPU flame graphs, 201
CPU one-liners, 267–268
CPU profiling, 200–201, 245, 268–270
description, 116
disk block devices, 465–467
disk I/O, 450, 467–468
documentation, 276
events. *See* `perf` tool events
flame graphs, 119, 270–272
hardware tracing, 276
hardware virtualization, 601–602, 604
I/O profiling, 202–203
kernel time analysis, 202
memory, 324
networks, 526, 562
one-liners for counting events, 675
one-liners for CPUs, 267–268
one-liners for disks, 467
one-liners for dynamic tracing, 677–678
one-liners for listing events, 674–675
one-liners for memory, 338–339
one-liners for profiling, 675–676
one-liners for reporting, 678–679
one-liners for static tracing, 676–677
OS virtualization, 619, 629
overview, 671–672
page fault flame graphs, 340–342
page fault sampling, 339–340
PMCs, 157, 273–274
process profiling, 271–272
profiling overview, 135
references, 703–704
scheduler latency, 272–273
software tracing, 275–276
subcommands. *See* `perf` tool subcommands
syscall tracing, 201–202
thread state analysis, 196
tools collection. *See* perf-tools collection
vs. `trace-cmd`, 738–739
tracepoint events, 684–685
tracepoints, 147, 149
tracing, 136, 166
`perf` tool events
hardware, 274–275, 680–683
kprobes, 685–687
overview, 679–681
software, 683–684
uprobes, 687–689
USDT probes, 690–691
`perf` tool subcommands
documentation, 703
`ftrace`, 741
miscellaneous, 702–703
overview, 672–674
`record`, 694–696
`report`, 696–698
`script`, 698–701
`stat`, 691–694
`trace`, 701–702
perf-tools collection
vs. BCC/BPF, 747–748
coverage, 742
documentation, 748
example, 747
multi-purpose tools, 744–745
one-liners, 745–747
overview, 741–742
single-purpose tools, 743–744
perf-tools-unstable tool package, 132
Performance and performance monitoring
applications, 172
challenges, 5–6
cloud computing, 14, 586
CPUs, 251
disks, 452

file systems, 388
memory, 326
networks, 529
OS virtualization, 620
resource analysis investments, 38
Performance application programming interface (PAPI), 158
Performance Co-Pilot (PCP), 138
Performance engineers, 2–3
Performance instrumentation counters (PICs), 156
Performance Mantras
applications, 182
list of, 61
Performance monitoring counters (PMCs), 156
case study, 788–789
challenges, 158
CPUs, 237–239, 273–274
cycle analysis, 251
documentation, 158
example, 156–157
interface, 157–158
memory, 326
Performance monitoring unit (PMU) events, 156, 680
perftrace tool, 136
Periods in OS virtualization, 615
Persistent memory, 441
Personalities in FileBench, 414
Perspectives
overview, 4–5
performance analysis, 37–38
resource analysis, 38–39
workload analysis, 39–40
Perturbations
benchmarks, 648
FlameScope, 292–293
system tests, 23
pfm-events, 681
PGO (profile-guided optimization) kernels, 122
Physical I/O
defined, 360
vs. logical, 368–370
Physical metadata in file systems, 368
Physical operations in file systems, 361
Physical resources in USE method, 795–798
PICs (performance instrumentation counters), 156
pid control group, 609
pid variable in **bpftrace**, 777
pids control group, 610
PIDs (process IDs)
filters, 729–730
process environment, 101
pidstat tool
CPUs, 245, 262
description, 15
disks, 464–465
OS virtualization, 619
thread state analysis, 196
Ping latency, 505–506, 528
ping tool, 562–563
Pipelines in ZFS, 381
pktgen tool, 567
Platters in magnetic rotational disks, 435–436
Plugins for monitoring software, 137
pmap tool, 135, 337–338
pmcarch tool
CPUs, 265–266
memory, 348
PMCs. *See* Performance monitoring counters (PMCs)
pmheld tool, 212–213
pmlock tool, 212
PMU (performance monitoring unit) events, 156, 680

Pods in cloud computing, 586
Point-in-time recommendations in methodologies, 29–30
Policies for scheduling classes, 106, 242–243
`poll` system call, 177
Polling applications, 177
Pooled storage
- btrfs, 382
- overview, 382–383
- ZFS, 380

Portability of benchmarks, 643
Portable Executable (PE) format, 183
Ports
- ephemeral, 531
- network, 501

`posix_fadvise` call, 415
Power states in processors, 297
Preallocation in ext4, 379
Precise event-based sampling (PEBS), 158
Prediction step in scientific method, 44–45
Preemption
- CPUs, 227
- Linux kernel, 116
- operating systems, 110
- schedulers, 241
- Solaris kernel, 114

preemptirsqoff tracer, 708
preemptoff tracer, 708
Prefetch caches, 230
Prefetch for file systems
- overview, 364–365
- ZFS, 381

Presentability of benchmarks, 643
Pressure stall information (PSI)
- CPUs, 257–258
- description, 119
- disks, 464
- memory, 323, 330–331

`pressure` tool, 142
Price/performance ratio
- applications, 173
- benchmarking for, 643

`print` function, 780
`printf` function, 770, 778
Priority
- CPUs, 227, 252–253
- OS virtualization resources, 613
- schedulers, 105–106
- scheduling classes, 242–243, 295

Priority inheritance scheme, 227
Priority inversion, 227
Priority pause frames in congestion avoidance, 508
Private clouds, 580
Privilege rings in kernels, 93
`probe` subcommand for `perf`, 673
`probe` variable in `bpftrace`, 778
Probes and probe events
- `bpftrace`, 767–768, 774–775
- kprobes, 685–687
- `perf`, 685
- uprobes, 687–689
- USDT, 690–691
- wildcards, 768–769

Problem statement
- case study, 16, 783–784
- determining, 44

`/proc` file system observability tools, 140–143
Process-context IDs (PCIDs), 119
Process IDs (PIDs)
- filters, 729–730
- process environment, 101

Processes
- accounting, 159
- creating, 100
- defined, 90
- environment, 101–102
- life cycle, 100–101

overview, 99–100
profiling, 271–272
schedulers, 105–106
swapping, 104–105, 308–309
syscall analysis, 192
tracing, 207–208
USE method, 52
virtual address space, 319–322

Processors
binding, 181–182
defined, 90, 220
power states, 297
tuning, 299

`procps` tool package, 131

Products, monitoring, 79

Profile-guided optimization (PGO) kernels, 122

`profile` probes, 774

`profile` tool
applications, 203–204
BCC, 756
CPUs, 245, 277–278
profiling, 135
`trace-cmd`, 735

Profilers
Ftrace, 707
perf-tools for, 745

Profiling
CPUs. *See* CPUs profiling
I/O, 203–204, 210–212
interpretation, 249–250
kprobes, 722
methodologies, 35
observability tools, 135
overview, 10–11
`perf`, 675–676
uprobes, 723

Program counter threads, 100

Programming languages
`bpftrace`. *See* `bpftrace` tool programming
compiled, 183–184
garbage collection, 185–186
interpreted, 184–185
overview, 182–183
virtual machines, 185

Prometheus monitoring software, 138

Proofs of concept
benchmarking for, 642
testing, 3

Proportional set size (PSS) in shared memory, 310

Protection rings in kernels, 93

Protocols
HTTP/3, 515
IP, 509–510
networks, 502, 504, 509–515
QUIC, 515
TCP, 510–514
UDP, 514

`ps` tool
CPUs, 260–261
fixed counters, 134
memory, 335–336
OS virtualization, 619

PSI. *See* Pressure stall information (PSI)

PSS (proportional set size) in shared memory, 310

Pterodactyl latency heat maps, 488–489

`ptime` tool, 263–264

`ptrace` tool, 159

Public clouds, 580

PV (paravirtualization), 588, 590

Q

`qdisc-fq` tool, 561

QEMU (Quick Emulator)
hardware virtualization, 589
lightweight virtualization, 631

`qemu-system-x86` process, 600

QLC (quad-level cell) flash memory, 440

QoS (quality of service) for networks, 532–533
QPI (Quick Path Interconnect), 236–237
Qspinlocks, 117–118
Quad-level cell (QLC) flash memory, 440
Quality of service (QoS) for networks, 532–533
Quantifying issues, 6
Quantifying performance gains, 73–74
Quarterly patterns, monitoring, 79
Question step in scientific method, 44–45
Queued spinlocks, 117–118
Queued time for disks, 472
Queueing disciplines
networks, 521
OS virtualization, 617
tuning, 571
Queues
I/O schedulers, 448–449
interrupts, 98
overview, 23–24
queueing theory, 66–69
run. *See* Run queues
TCP connections, 519–520
QUIC protocol, 515
Quick Emulator (QEMU)
hardware virtualization, 589
lightweight virtualization, 631
Quick Path Interconnect (QPI), 236–237
Quotas in OS virtualization, 615

R

RACK (recent acknowledgments) in TCP, 514
RAID (redundant array of independent disks) architecture, 444–445
Ramping load benchmarking, 662–664
Random-access pattern in micro-benchmarking, 390
Random change anti-method, 42–43
Random I/O
disk read example, 491–492
disks, 430–431, 436
latency profile, micro-benchmarking, 457
vs. sequential, 363–364
Rate transitions in networks, 517
Raw hardware event descriptors, 680
Raw I/O, 366, 447
Raw tracepoints, 150
RCU (read-copy update), 115
RCU-walk (read-copy-update-walk) algorithm, 375
rdma control group, 610
Re-exec method in heap growth, 320
Read-ahead in file systems, 365
Read-copy update (RCU), 115
Read-copy-update-walk (RCU-walk) algorithm, 375
Read latency profile in micro-benchmarking, 457
Read-modify-write operation in RAID, 445
`read` syscalls
description, 94
tracing, 404–405
Read/write ratio in disks, 431
`readahead` tool, 409
Reader/writer (RW) locks, 179
Real-time scheduling classes, 106, 253
Real-time systems, interrupt masking in, 98
Realism in benchmarks, 643
Reaping memory, 316, 318
Rebuilding volumes and pools, 383
Receive Flow Steering (RFS) in networks, 523
Receive Packet Steering (RPS) in networks, 523
Receive packets in NICs, 522
Receive Side Scaling (RSS) in networks, 522–523

Recent acknowledgments (RACK) in TCP, 514
Reclaimed pages, 317
Record size, defined, 360
`record` subcommand for `perf`
  CPU profiling, 695–696
  example, 672
  options, 695
  overview, 694–695
  software events, 683–684
  stack walking, 696
`record` subcommand for `trace-cmd`, 735
RED method, 53
Reduced instruction set computers (RISCs), 224
Redundant array of independent disks (RAID) architecture, 444–445
`reg` function, 779
Regression testing, 18
Remote memory, 312
Reno algorithm for TCP congestion control, 513
Repeatability of benchmarks, 643
Replay benchmarking, 654
`report` subcommand for `perf`
  example, 672
  overview, 696–697
  STDIO, 697–698
  TUI interface, 697
`report` subcommand for `trace-cmd`, 735
Reporting
  `perf`, 678–679
  `sar`, 163, 165
  `trace-cmd`, 737
Request latency, 7
Request rate in RED method, 53
Request time in I/O, 427
Requests in workload analysis, 39
Resident memory, defined, 304
Resident set size (RSS), 308
Resilvering volumes and pools, 383
Resource analysis perspectives, 4–5, 38–39
Resource controls
  cloud computing, 586
  CPUs, 253, 298
  disks, 456, 494
  hardware virtualization, 595–597
  lightweight virtualization, 632
  memory, 328, 353–354
  networks, 532–533
  operating systems, 110–111
  OS virtualization, 613–617, 626–627
  tuning, 571
  USE method, 52
Resource isolation in cloud computing, 586
Resource limits in capacity planning, 70–71
Resource lists in USE method, 49
Resource utilization in applications, 173
Resources in USE method, 47
Response time
  defined, 22
  disks, 452
  latency, 24
`restart` subcommand in `trace-cmd`, 735
Results in event tracing, 58
Retention policy for caching, 36
Retransmits
  latency, 528
  TCP, 510, 512, 529
  UDP, 514
Retrospectives, 4
Return values
  kprobes, 721
  kretprobes, 152
  ukretprobes, 154
  uprobes, 723
`retval` variable in `bpftrace`, 778
RFS (Receive Flow Steering) in networks, 523

Ring buffers
applications, 177
networks, 522
RISCs (reduced instruction set computers), 224
Robertson, Alastair 761
Roles, 2–3
Root level in file systems, 106
Rostedt, Steven, 705, 711, 734, 739–740
Rotation time in magnetic rotational disks, 436
Round-trip time (RTT) in networks, 507, 528
Route tables, 537
Routers, 516–517
Routing networks, 503
RPS (Receive Packet Steering) in networks, 523
RR scheduling policy, 243
RSS (Receive Side Scaling) in networks, 522–523
RSS (resident set size), 308
RT scheduling class, 242–243
RTT (round-trip time) in networks, 507, 528
Run queues
CPUs, 222
defined, 220
latency, 222
schedulers, 105, 241
Runnability of benchmarks, 643
Runnable state in thread state analysis, 194–197
`runqlat` tool
CPUs, 279–280
description, 756
`runqlen` tool
CPUs, 280–281
description, 756
`runqslower` tool
CPUs, 285
description, 756
RW (reader/writer) locks, 179

S

S3 (Simple Storage Service), 585
SaaS (software as a service), 634
SACK (selective acknowledgment) algorithm, 514
SACKs (selective acknowledgments), 510
Sampling
CPU profiling, 35, 135, 187, 200–201, 247–248
distributed tracing, 199
off-CPU analysis, 189–190
page faults, 339–340
PMCs, 157–158
run queues, 242–243
Sanity checks in benchmarking, 664–665
`sar` (system activity reporter)
configuration, 162
coverage, 161
CPUs, 260
description, 15
disks, 463–464
documentation, 165–166
file systems, 393–394
fixed counters, 134
live reporting, 165
memory, 331–333
monitoring, 137, 161–165
networks, 543–545
options, 801–802
OS virtualization, 619
output formats, 163–165
overview, 160
reporting, 163
thread state analysis, 196
SAS (Serial Attached SCSI) disk interface, 442
SATA (Serial ATA) disk interface, 442
Saturation
applications, 193
CPUs, 226–227, 245–246, 251, 795, 797

- defined, 22
- disk controllers, 451
- disk devices, 434, 451
- flame graphs, 291
- I/O, 798
- kernels, 798
- memory, 309, 324–326, 796–797
- methodologies, 34–35
- networks, 526–527, 796–797
- resource analysis, 38
- storage, 797
- task capacity, 799
- USE method, 47–48, 51–53
- user mutex, 799

Saturation points in scalability, 31

Scalability and scaling
- Amdahl's Law of Scalability, 64–65
- capacity planning, 72–73
- cloud computing, 581–584
- CPU, 522–523
- CPUs vs. GPUs, 240
- disks, 457–458
- methodologies, 31–32
- models, 63–64
- multithreading, 227
- Universal Scalability Law, 65–66

Scalability ceiling, 64

Scalable Vector Graphics (SVG) files, 164

Scaling governors, 297

Scanning pages, 318–319, 323, 374

Scatter plots
- disk I/O, 81–82
- I/O latency, 488

`sched` command, 141

SCHED_DEADLINE policy, 117

`sched` subcommand for `perf`, 272–273, 673, 702

`schedstat` tool, 141–142

Scheduler latency
- CPUs, 226, 272–273
- delay accounting, 145
- run queues, 222

Scheduler tracing off-CPU analysis, 189–190

Schedulers
- CPUs, 241–242
- defined, 220
- hardware virtualization, 596–597
- kernel, 105–106
- multiqueue I/O, 119
- options, 295–296
- OS disk I/O stack, 448–449
- scheduling internals, 284–285

Scheduling classes
- CPUs, 115, 242–243
- I/O, 115, 493
- kernel, 106
- priority, 295

Scheduling in Kubernetes, 586

Scientific method, 44–46

Scratch variables in `bpftrace`, 770–771

`scread` tool, 409

`script` subcommand
- flame graphs, 700
- overview, 698–700
- trace scripts, 700–701

`script` subcommand for `perf`, 673

Scrubbing file systems, 376

SCSI (Small Computer System Interface)
- disks, 442
- event logging, 486

`scsilatency` tool, 487

`scsiresult` tool, 487

SDT events, 681

Second-level caches in file systems, 362

Sectors in disks
- defined, 424
- size, 437
- zoning, 437

Security boot options, 298–299

SEDA (staged event-driven architecture), 178
SEDF (simple earliest deadline first) schedulers, 595
Seek time in magnetic rotational disks, 436
seeksize tool, 487
seekwatcher tool, 487
Segments
  defined, 304
  OSI model, 502
  process virtual address space, 319
  segmentation offload, 520–521
Selective acknowledgment (SACK) algorithm, 514
Selective acknowledgments (SACKs), 510
Self-Monitoring, Analysis and Reporting Technology (SMART) data, 485
self tool, 142
Semaphores for applications, 179
Send packets in NICs, 522
sendfile command, 181
Sequential I/O
  disks, 430–431, 436
  vs. random, 363–364
Serial ATA (SATA) disk interface, 442
Serial Attached SCSI (SAS) disk interface, 442
Server instances in cloud computing, 580
Service consoles in hardware virtualization, 589
Service thread pools for applications, 178
Service time
  defined, 22
  I/O, 427–429
  queueing systems, 67–69
Set associative caches, 234
set_ftrace_filter file, 710
Shadow page tables, 593
Shadow statistics, 694
Shards
  capacity planning, 73
  cloud computing, 582
Shared memory, 310
Shared system buses, 312
Shares in OS virtualization, 614–615, 626
Shell scripting, 184
Shingled Magnetic Recording (SMR) drives, 439
shmsnoop tool, 348
Short-lived processes, 12, 207–208
Short-stroking in magnetic rotational disks, 437
showboost tool, 245, 265
signal function, 779
Signal tracing, 209–210
Simple disk model, 425
Simple earliest deadline first (SEDF) schedulers, 595
Simple Network Management Protocol (SNMP), 55, 137
Simple Storage Service (S3), 585
Simulation benchmarking, 653–654
Simultaneous multithreading (SMT), 220, 225
Single-level cell (SLC) flash memory, 440
Single root I/O virtualization (SR-IOV), 593
Site reliability engineers (SREs), 4
Size
  blocks, 27, 360, 375, 378
  cloud computing, 583–584
  disk I/O, 432, 480–481
  disk sectors, 437
  free lists, 317
  I/O, 176, 390
  instruction, 224
  multiple page, 352–353
  packets, 504–505
  virtual memory, 308
  word, 229, 310
  working set. *See* Working set size (WSS)
sizeof function, 779
skbdrop tool, 561

skblife tool, 561
Slab
allocator, 114
process virtual address space, 321–322
slabinfo tool, 142
slabtop tool, 333–334, 394–395
SLC (single-level cell) flash memory, 440
Sleeping state in thread state analysis, 194–197
Sliding windows in TCP, 510
SLOG log in ZFS, 381
Sloth disks, 438
Slow disks case study, 16–18
Slow-start in TCP, 510
Slowpath state in Mutex locks, 179
SLUB allocator, 116, 322
Small Computer System Interface (SCSI)
disks, 442
event logging, 486
smaps tool, 141
SMART (Self-Monitoring, Analysis and Reporting Technology) data, 485
smartctl tool, 484–486
SMP (symmetric multiprocessing), 110
smpcalls tool, 285
SMR (Shingled Magnetic Recording) drives, 439
SMs (streaming multiprocessors), 240
SMT (simultaneous multithreading), 220, 225
Snapshots
btrfs, 382
ZFS, 381
Sniffing packets, 530–531
SNMP (Simple Network Management Protocol), 55, 137
SO_BUSY_POLL socket option, 522
SO_REUSEPORT socket option, 117
SO_TIMESTAMP socket option, 529
SO_TIMESTAMPING socket option, 529
so1stbyte tool, 561
soaccept tool, 561
socketio tool, 561
socketio.bt tool, 553–554
Sockets
BSD, 113
defined, 500
description, 109
local connections, 509
options, 573
statistics, 534–536
tracing, 552–555
tuning, 569
socksize tool, 561
sockstat tool, 561
soconnect tool, 561
soconnlat tool, 561
sofamily tool, 561
Soft interrupts, 281–282
softirqs tool, 281–282
Software
memory, 315–322
networks, 517–524
Software as a service (SaaS), 634
Software change case study, 18–19
Software events
case study, 789–790
observability source, 159
perf, 680, 683–684
recording and tracing, 275–276
software probes, 774
Software resources
capacity planning, 70
USE method, 52, 798–799
Solaris
kernel, 114
Kstat, 160
Slab allocator, 322, 652
syscall tracing, 205

top tool Solaris mode, 262
zones, 606, 620
Solid-state disks (SSDs)
cache devices, 117
overview, 439–441
`soprotocol` tool, 561
`sormem` tool, 561
Source code for applications, 172
SPEC (Standard Performance Evaluation Corporation) benchmarks, 655–656
Special file systems, 371
Speedup with latency, 7
Spin locks
applications, 179
contention, 198
queued, 118
`splice` call, 116
SPs (streaming processors), 240
SR-IOV (single root I/O virtualization), 593
SREs (site reliability engineers), 4
`ss` tool, 145–146, 525, 534–536
SSDs (solid-state disks)
cache devices, 117
overview, 439–441
Stack helpers, 214
Stack traces
description, 102
displaying, 204–205
keys, 730–731
Stack walking, 102, 696
`stackcount` tool, 757–758
Stacks
I/O, 107–108, 372
JIT symbols, 214
missing, 215–216
network, 109, 518–519
operating system disk I/O, 446–449
overview, 102
process virtual address space, 319
protocol, 502
reading, 102–103
user and kernel, 103
Staged event-driven architecture (SEDA), 178
Stall cycles in CPUs, 223
Standard deviation, 75
Standard Performance Evaluation Corporation (SPEC) benchmarks, 655–656
Starovoitov, Alexei, 121
`start` subcommand in `trace-cmd`, 735
Starvation in deadline I/O schedulers, 448
`stat` subcommand in `perf`
description, 635
event filters, 693–694
interval statistics, 693
options, 692–693
overview, 691–692
per-CPU balance, 693
shadow statistics, 694
`stat` subcommand in `trace-cmd`, 735
`stat` tool, 95, 141–142
Stateful workload simulation, 654
Stateless workload simulation, 653
Statelessness of UDP, 514
States
TCP, 511–512
thread state analysis, 193–197
Static instrumentation
overview, 11–12
`perf` events, 681
tracepoints, 146, 717
Static performance tuning
applications methodology, 198–199
CPUs, 252
disks, 455–456
file systems, 389
memory, 327–328
methodologies, 59–60
networks, 531–532
tools, 130–131

Static priority of threads, 242–243
Static probes, 116
Static tracing in `perf`, 676–677
Statistical analysis in benchmarking, 665–666
Statistics, 8–9
averages, 74–75
baseline, 59
case study, 784–786
coefficient of variation, 76
line charts, 80–81
multimodal distributions, 76–77
outliers, 77
quantifying performance gains, 73–74
standard deviation, percentiles, and median, 75
`statm` tool, 141
`stats` function, 780
`statsnoop` tool, 409
`status` tool, 141
STDIO report option, 697–698
`stop` subcommand in `trace-cmd`, 735
Storage
benchmark questions, 668
cloud computing, 584–585
disks. *See* Disks
sample processing, 248–249
USE method, 49–51, 796–797
Storage array caches, 430
Storage arrays, 446
`str` function, 770, 778
`strace` tool
bonnie++ tool, 660
file system latency, 395
format strings, 149–150
limitations, 202
networks, 561
overhead, 207
system call tracing, 205–207
tracing, 136
`stream` subcommand in `trace-cmd`, 735
Streaming multiprocessors (SMs), 240
Streaming processors (SPs), 240
Streaming workloads in disks, 430–431
Streetlight effect, 42
Stress testing in software change case study, 18
Stripe width of volumes and pools, 383
Striped allocation in XFS, 380
Stripes in RAID, 444–445
`strncmp` function, 778
Stub domains in hardware virtualization, 596
Subjectivity, 5
Subsecond-offset heat maps, 289
`sum` function in `bpftrace`, 780
Summary-since-boot values monitoring, 79
Super-serial model, 65–66
Superblocks in VFS, 373
`superping` tool, 561
Superscalar architectures for CPUs, 224
Surface plots, 84–85
SUT (system under test) models, 23
SVG (Scalable Vector Graphics) files, 164
Swap areas, defined, 304
Swap capacity in OS virtualization, 613, 616
`swapin` tool, 348
`swapon` tool
disks, 487
memory, 331
Swapping
defined, 304
memory, 316, 323
overview, 305–307
processes, 104–105, 308–309
Swapping state
delay accounting, 145
thread state analysis, 194–197
Switches in networks, 516–517
Symbol churn, 214

Symbols, missing, 214
Symmetric multiprocessing (SMP), 110
SYN backlogs, 519
SYN cookies, 511, 520
Synchronization primitives for applications, 179
Synchronous disk I/O, 434–435
Synchronous interrupts, 97
Synchronous writes, 366
`syncsnoop` tool
  BCC, 756
  file systems, 409
Synthetic events in hist triggers, 731–733
`/sys` file system, 143–144
`/sys`/fs options, 417–418
SysBench system benchmark, 294
`syscount` tool
  BCC, 756
  CPUs, 285
  file systems, 409
  perf-tools, 744
  system calls count, 208–209
`sysctl` tool
  congestion control, 570
  network tuning, 567–568
  schedulers, 296
  SCSI logging, 486
sysstat tool package, 131
System activity reporter. *See* `sar` (system activity reporter)
System calls
  analysis, 192
  connect latency, 528
  counting, 208–209
  defined, 90
  file system latency, 385
  kernel, 92, 94–95
  micro-benchmarking for, 61
  observability source, 159
  send/receive latency, 528
  tracing in `bpftrace`, 403–405
  tracing in `perf`, 201–202
  tracing in `strace`, 205–207
System design, benchmarking for, 642
`system` function in `bpftrace`, 770, 779
System statistics, monitoring, 138
System under test (SUT) models, 23
System-wide CPU profiling, 268–270
System-wide observability tools, 133
  fixed counters, 134
  `/proc` file system, 141–142
  profiling, 135
  tracing, 136
System-wide tunable parameters
  byte queue limits, 571
  device backlog, 569
  ECN, 570
  networks, 567–572
  production example, 568
  queueing disciplines, 571
  resource controls, 571
  sockets and TCP buffers, 569
  TCP backlog, 569
  TCP congestion control, 570
  Tuned Project, 572
`systemd-analyze` command, 120
systemd service manager, 120
Systems performance overview, 1–2
  activities, 3–4
  cascading failures, 5
  case studies, 16–19
  cloud computing, 14
  complexity, 5
  counters, statistics, and metrics, 8–9
  experiments, 13–14
  latency, 6–7
  methodologies, 15–16
  multiple performance issues, 6
  observability, 7–13
  performance challenges, 5–6

perspectives, 4–5
references, 19–20
roles, 2–3
SystemTap tool, 166

T

Tagged Command Queueing (TCQ), 437
Tahoe algorithm for TCP congestion control, 513
Tail-based sampling in distributed tracing, 199
Tail Loss Probe (TLP), 117, 512
Task capacity in USE method, 799
task tool, 141
Tasklets with interrupts, 98
Tasks
defined, 90
idle, 99
taskset command, 297
tc tool, 566
tcdump tool, 136
TCMalloc allocator, 322
TCP. *See* Transmission Control Protocol (TCP)
TCP Fast Open (TFO), 117
TCP/IP stack
BSD, 113
kernels, 109
protocol, 502
stack bypassing, 509
TCP segmentation offload (TSO), 521
TCP Small Queues (TSQ), 524
TCP Tail Loss Probe (TLP), 117
TCP TIME_WAIT latency, 528
tcpaccept tool, 561
tcpconnect tool, 561
tcpdump tool
BPF for, 12
description, 526
event tracing, 57–58
overview, 558–559
packet sniffing, 530–531
tcplife tool
BCC, 756
description, 525
overview, 548
tcpnagle tool, 561
tcpreplay tool, 567
tcpretrans tool
BCC, 756
overview, 549–550
perf-tools, 743
tcpsynbl.bt tool, 556–557
tcptop tool
BCC, 756
description, 526
top processes, 549
tcpwin tool, 561
TCQ (Tagged Command Queueing), 437
Temperature-aware scheduling classes, 243
Temperature sensors for CPUs, 230
Tenancy in cloud computing, 580
contention in hardware virtualization, 595
contention in OS virtualization, 612–613
overview, 585–586
Tensor processing units (TPUs), 241
Test errors in benchmarking, 646–647
Text step in scientific method, 44–45
Text user interface (TUI), 697
TFO (TCP Fast Open), 117
Theoretical maximum disk throughput, 436–437
Thermal pressure in Linux kernel, 119
THP (transparent huge pages)
Linux kernel, 116
memory, 353
Thread blocks in GPUs, 240
Thread pools in USE method, 52

Thread state analysis, 193–194
Linux, 195–197
software change case study, 19
states, 194–195
Threads
applications, 177–181
CPU time, 278–279
CPUs, 227–229
CPUs vs. GPUs, 240
defined, 90
flusher, 374
hardware, 221
idle, 99, 244
interrupts, 97–98
lightweight, 178
micro-benchmarking, 653
processes, 100
schedulers, 105–106
SMT, 225
static priority, 242–243
USE method, 52
3-wide processors, 224
3D NAND flash memory, 440
3D XPoint persistent memory, 441
Three-way handshakes in TCP, 511
Throttling
benchmarks, 661
hardware virtualization, 597
OS virtualization, 626
packets, 522
Throughput
applications, 173
defined, 22
disks, 424
file systems, 360
magnetic rotational disks, 436–437
networks, defined, 500
networks, measuring, 527–529
networks, monitoring, 529
performance metric, 32
resource analysis, 38
solid-state drives, 441
workload analysis, 40
Tickless kernels, 99, 117
Ticks, clock, 99
`tid` variable in `bpftrace`, 777
Time
averages over, 74
disk measurements, 427–429
event tracing, 58
kernel analysis, 202
Time-based patterns in monitoring, 77–78
Time-based utilization, 33–34
time control group, 609
`time` function in `bpftrace`, 778
Time scales
disks, 429–430
methodologies, 25–26
Time-series metrics, 8
Time sharing for schedulers, 241
Time slices for schedulers, 242
Time to first byte (TTFB) in networks, 506
`time` tool for CPUs, 263–264
TIME_WAIT latency, 528
TIME_WAIT state, 512
`timechart` subcommand for `perf`, 673
Timeline charts, 83–84
Timer-based profile sampling, 247–248
Timer-based retransmits, 512
Timerless multitasking, 117
Timers in TCP, 511–512
Timestamps
CPU counters, 230
file systems, 371
TCP, 511
`tiptop` tool, 348
`tiptop` tool package, 132

TLBs. ***See*** **Translation lookaside buffers (TLBs)**

`tlbstat` tool

CPUs, 266–267

memory, 348

TLC (tri-level cell) flash memory, 440

TLP (Tail Loss Probe), 117, 512

TLS (transport layer security), 113

Tools method

CPUs, 245

disks, 450

memory, 323–324

networks, 525

overview, 46

Top-level directories, 107

Top of file system layer, file system latency in, 385

`top` subcommand for `perf`, 673

`top` tool

CPUs, 245, 261–262

description, 15

file systems, 393

fixed counters, 135

hardware virtualization, 600

lightweight virtualization, 632–633

memory, 324, 336–337

OS virtualization, 619, 624

TPC (Transaction Processing Performance Council) benchmarks, 655

TPC-A benchmark, 650–651

`tpoint` tool, 744

TPUs (tensor processing units), 241

`trace-cmd` front end, 132

documentation, 740

function_graph, 739

KernelShark, 739–740

one-liners, 736–737

overview, 734

vs. `perf`, 738–739

subcommands overview, 734–736

trace file, 710, 713–715

trace_options file, 710

trace_pipe file, 710, 715

Trace scripts, 698, 700–701

trace_stat directory, 710

`trace` subcommand for `perf`, 673, 701–702

`trace` tool, 757–758

tracefs file system, 149–150

contents, 709–711

overview, 708–709

`tracepoint` probes, 774

Tracepoints

arguments and format string, 148–149

description, 11

documentation, 150–151

events in `perf`, 681, 684–685

example, 147–148

filters, 717–718

interface, 149–150

Linux kernel, 116

overhead, 150

overview, 146

triggers, 718

tracepoints tracer, 707

`traceroute` tool, 563–564

Tracing

BPF, 12–13

`bpftrace`. *See* `bpftrace` tool

case study, 790–792

distributed, 199

dynamic instrumentation, 12

events. *See* Event tracing

Ftrace. *See* Ftrace tool

locks, 212–213

observability tools, 136

OS virtualization, 620, 624–625, 629

`perf`, 676–678

perf-tools for, 745

schedulers, 189–190

sockets, 552–555

software, 275–276
static instrumentation, 11–12
`strace`, 136, 205–207
tools, 166
`trace-cmd`. *See* `trace-cmd` front end
virtual file system, 405–406
tracing_on file, 710
Trade-offs in methodologies, 26–27
Traffic control utility in networks, 566
Transaction costs of latency, 385–386
Transaction groups (TXGs) in ZFS, 381
Transaction Processing Performance Council (TPC) benchmarks, 655
Translation lookaside buffers (TLBs)
cache statistics, 266–267
CPUs, 232
flushing, 121
memory, 314–315
MMU, 235
shootdowns, 367
Translation storage buffers (TSBs), 235
Transmission Control Protocol (TCP)
analysis, 531
anti-bufferbloat, 117
autocorking, 117
backlog, tuning, 569
buffers, 520, 569
congestion algorithms, 115
congestion avoidance, 508
congestion control, 118, 513, 570
connection latency, 24, 506, 528
connection queues, 519–520
connection rate, 527–529
duplicate ACK detection, 512
features, 510–511
first-byte latency, 528
friends, 509
initial window, 514
Large Receive Offload, 116
lockless listener, 118
New Vegas, 118
offload in packet size, 505
out-of-order pack*ets*, 529
retransmits, 117, 512, 528–529
SACK, FACK, and RACK, 514
states and timers, 511–512
three-way handshakes, 511
tracing in `bpftrace`, 555–557
transfer time, 24–25
Transmit Packet Steering (XPS) in networks, 523
Transparent huge pages (THP)
Linux kernel, 116
memory, 353
Transport, defined, 424
Transport layer security (TLS), 113
Traps
defined, 90
synchronous interrupts, 97
Tri-level cell (TLC) flash memory, 440
Triggers
hist. *See* Hist triggers
kprobes, 721–722
tracepoints, 718
uprobes, 723
Troubleshooting, benchmarking for, 642
TSBs (translation storage buffers), 235
`tshark` tool, 559
TSO (TCP segmentation offload), 521
TSQ (TCP Small Queues), 524
TTFB (time to first byte) in networks, 506
TUI (text user interface), 697
Tunable parameters
disks, 494
memory, 350–351
micro-benchmarking, 390
networks, 567
operating systems, 493–495
point-in-time recommendations, 29–30
tradeoffs with, 27

tune2fs tool, 416–417
Tuned Project, 572
Tuning
  benchmarking for, 642
  caches, 60
  CPUs. *See* CPUs tuning
  disk caches, 456
  disks, 493–495
  file system caches, 389
  file systems, 414–419
  memory, 350–354
  methodologies, 27–28
  networks, 567–574
  static performance. *See* Static performance tuning
  targets, 27–28
turboboost tool, 245
turbostat tool, 264–265
TXGs (transaction groups) in ZFS, 381
Type 1 hypervisors, 587
Type 2 hypervisors, 587

U

uaddr function, 779
Ubuntu Linux distribution
  crisis tools, 131–132
  memory tunables, 350–351
  sar configuration, 162
  scheduler options, 295–296
UDP Generic Receive Offload (GRO), 119
UDP (User Datagram Protocol), 514
udpconnect tool, 561
UDS (Unix domain sockets), 509
uid variable in bpftrace, 777
UIDs (user IDs) for processes, 101
UIO (user space I/O) in kernel bypass, 523
ulimit command, 111
Ultra Path Interconnect (UPI), 236–237
UMA (uniform memory access) memory system, 311–312
UMA (universal memory allocator), 322
UMASK values in MSRs, 238–239
Unicast network transmissions, 503
UNICS (UNiplexed Information and Computing Service), 112
Unified buffer caches, 374
Uniform memory access (UMA) memory system, 311–312
Unikernels, 92, 123, 634
UNiplexed Information and Computing Service (UNICS), 112
Units of time for latency, 25
Universal memory allocator (UMA), 322
Universal Scalability Law (USL), 65–66
Unix domain sockets (UDS), 509
Unix kernels, 112
UnixBench benchmarks, 254
Unknown-unknowns, 37
Unrelated disk I/O, 368
unroll function, 776
UPI (Ultra Path Interconnect), 236–237
uprobe_events file, 710
uprobe profiler, 707
uprobe tool, 744
uprobes, 687–688
  arguments, 154, 688–689, 723
  bpftrace, 774
  documentation, 155
  event tracing, 722–723
  example, 154
  filters, 723
  Ftrace, 708
  interface and overload, 154–155
  Linux kernel, 117
  overview, 153
  profiling, 723
  return values, 723
  triggers, 723
uptime tool
  case study, 784–785
  CPUs, 245

description, 15
load averages, 255–257
OS virtualization, 619
PSI, 257–258
uretprobes, 154
usdt probes, 774
USDT (user-level static instrumentation events)
perf, 681
probes, 690–691
USDT (user-level statically defined tracing), 11, 155–156
USE method. *See* Utilization, saturation, and errors (USE) method
User address space in processes, 102
User allocation stacks, 345
user control group, 609
User Datagram Protocol (UDP), 514
User IDs (UIDs) for processes, 101
User land, 90
User-level static instrumentation events (USDT)
perf, 681
probes, 690–691
User-level statically defined tracing (USDT), 11, 155–156
User modes in kernels, 93–94
User mutex in USE method, 799
User space, defined, 90
User space I/O (UIO) in kernel bypass, 523
User stacks, 103
User state in thread state analysis, 194–197
User time in CPUs, 226
username variable in bpftrace, 777
USL (Universal Scalability Law), 65–66
ustack function in bpftrace, 779
ustack variable in bpftrace, 778
usym function, 779
util-linux tool package, 131
Utilization
applications, 173, 193
CPUs, 226, 245–246, 251, 795, 797
defined, 22
disk controllers, 451
disk devices, 451
disks, 433, 452
heat maps, 288–289, 490
I/O, 798
kernels, 798
memory, 309, 324–326, 796–797
methodologies, 33–34
networks, 508–509, 526–527, 796–797
performance metric, 32
resource analysis, 38
storage, 796–797
task capacity, 799
USE method, 47–48, 51–53
user mutex, 799
Utilization, saturation, and errors (USE) method
applications, 193
benchmarking, 661
CPUs, 245–246
disks, 450–451
functional block diagrams, 49–50
memory, 324–325
metrics, 48–51
microservices, 53
networks, 526–527
overview, 47
physical resources, 795–798
procedure, 47–48
references, 799
resource controls, 52
resource lists, 49
slow disks case study, 17
software resources, 52, 798–799
uts control group, 609

V

V-NAND (vertical NAND) flash memory, 440

`valgrind` tool

- CPUs, 286
- memory, 348

Variable block sizes in file systems, 375

Variables in `bpftrace`, 770–771, 777–778

Variance

- benchmarks, 647
- description, 75
- FlameScope, 292–293

Variation, coefficient of, 76

vCPUs (virtual CPUs), 595

Verification of observability tool results, 167–168

Versions

- applications, 172
- kernel, 111–112

Vertical NAND (V-NAND) flash memory, 440

Vertical scaling

- capacity planning, 72
- cloud computing, 581

VFIO (virtual function I/O) drivers, 523

VFS. *See* Virtual file system (VFS)

VFS layer, file system latency analysis in, 385

`vfs_read` function in `bpftrace`, 772–773

`vfs_read` tool in Ftrace, 706–707

`vfscount` tool, 409

`vfssize` tool, 409

`vfsstat` tool, 409

Vibration in magnetic rotational disks, 438

Virtual CPUs (vCPUs), 595

Virtual disks

- defined, 424
- utilization, 433

Virtual file system (VFS)

- defined, 360
- description, 107
- interface, 373
- latency, 406–408
- Solaris kernel, 114
- tracing, 405–406

Virtual function I/O (VFIO) drivers, 523

Virtual machine managers (VMMs)

- cloud computing, 580
- hardware virtualization, 587–605

Virtual machines (VMs)

- cloud computing, 580
- hardware virtualization, 587–605
- programming languages, 185

Virtual memory

- BSD kernel, 113
- defined, 90, 304
- managing, 104–105
- overview, 305
- size, 308

Virtual processors, 220

Virtual-to-guest physical translation, 593

Virtualization

- hardware. *See* Hardware virtualization
- OS. *See* OS virtualization

Visual identification of models, 62–64

Visualizations, 79

- `blktrace`, 479
- CPUs, 288–293
- disks, 487–490
- file systems, 410–411
- flame graphs. *See* Flame graphs
- heat maps. *See* Heat maps
- line charts, 80–81
- scatter plots, 81–82
- surface plots, 84–85
- timeline charts, 83–84
- tools, 85

VMMs (virtual machine managers)

- cloud computing, 580
- hardware virtualization, 587–588

VMs (virtual machines)
cloud computing, 580
hardware virtualization, 587–588
programming languages, 185
`vmscan` tool, 348
`vmstat` tool, 8
CPUs, 245, 258
description, 15
disks, 487
file systems, 393
fixed counters, 134
hardware virtualization, 604
memory, 323, 329–330
OS virtualization, 619
thread state analysis, 196
VMware ESX, 589
Volume managers, 360
Volumes
defined, 360
file systems, 382–383
Voluntary kernel preemption, 110, 116

W

W-caches in CPUs, 230
Wait time
disks, 434
I/O, 427
off-CPU analysis, 191–192
wakeup tracer, 708
wakeup_rt tracer, 708
`wakeuptime` tool, 756
Warm caches, 37
Warmth of caches, 37
`watchpoint` probes, 774
Waterfall charts, 83–84
Wear leveling in solid-state drives, 441
Weekly patterns, monitoring, 79
Whetstone benchmark, 254, 653
Whys in drill-down analysis, 56
Width
flame graphs, 290–291
instruction, 224
Wildcards for probes, 768–769
Windows
DiskMon, 493
fibers, 178
hybrid kernel, 92
Hyper-V, 589
LTO and PGO, 122
microkernel, 123
portable executable format, 183
ProcMon, 207
syscall tracing, 205
TIME_WAIT, 512
word size, 310
Wireframe models, 84–85
Wireshark tool, 560
Word size
CPUs, 229
memory, 310
Work queues with interrupts, 98
Working set size (WSS)
benchmarking, 664
memory, 310, 328, 342–343
micro-benchmarking, 390–391, 653
Workload analysis perspectives, 4–5, 39–40
Workload characterization
benchmarking, 662
CPUs, 246–247
disks, 452–454
file systems, 386–388
methodologies, 54
networks, 527–528
workload analysis, 39
Workload separation in file systems, 389
Workloads, defined, 22
Write amplification in solid-state drives, 440

Write-back caches
- file systems, 365
- on-disk, 425
- virtual disks, 433

`write` system calls, 94
Write-through caches, 425
Write type, micro-benchmarking for, 390
`writeback` tool, 409
Writes starving reads, 448
`writesync` tool, 409
`wss` tool, 342–343
WSS (working set size)
- benchmarking, 664
- memory, 310, 328, 342–343
- micro-benchmarking, 390–391, 653

X

XDP (Express Data Path) technology
- description, 118
- event sources, 558
- kernel bypass, 523

Xen hardware virtualization
- CPU usage, 595
- description, 589
- I/O path, 594
- network performance, 597
- observability, 599

`xentop` tool, 599
XFS file system, 379–380
`xfsdist` tool
- BCC, 756
- file systems, 399

`xfsslower` tool, 757
XPS (Transmit Packet Steering) in networks, 523

Y

Yearly patterns, monitoring, 79

Z

`zero` function, 780
ZFS file system
- features, 380–381
- options, 418–419
- pool statistics, 410
- Solaris kernel, 114

`zfsdist` tool
- BCC, 757
- file systems, 399

`zfsslower` tool, 757
ZIO pipeline in ZFS, 381
`zoneinfo` tool, 142
Zones
- free lists, 317
- magnetic rotational disks, 437
- OS virtualization, 606, 620
- Solaris kernel, 114

`zpool` tool, 410