

# OPERATING SYSTEMS

THREE EASY PIECES

Remzi H. Arpaci-Dusseau  
Andrea C. Arpaci-Dusseau

**To Vedat S. Arpacı, a lifelong inspiration**

---

## Contents

To Everyone . . . . .	iii
To Educators . . . . .	v
To Students . . . . .	vi
Acknowledgments . . . . .	vii
Final Words . . . . .	ix
References . . . . .	x
<b>1 A Dialogue on the Book</b>	<b>1</b>
<b>2 Introduction to Operating Systems</b>	<b>3</b>
2.1 Virtualizing the CPU . . . . .	5
2.2 Virtualizing Memory . . . . .	7
2.3 Concurrency . . . . .	8
2.4 Persistence . . . . .	11
2.5 Design Goals . . . . .	13
2.6 Some History . . . . .	14
2.7 Summary . . . . .	18
References . . . . .	19
<b>I Virtualization</b>	<b>21</b>
<b>3 A Dialogue on Virtualization</b>	<b>23</b>
<b>4 The Abstraction: The Process</b>	<b>25</b>
4.1 The Abstraction: A Process . . . . .	26
4.2 Process API . . . . .	27
4.3 Process Creation: A Little More Detail . . . . .	28
4.4 Process States . . . . .	29
4.5 Data Structures . . . . .	31
4.6 Summary . . . . .	33
References . . . . .	34
Homework . . . . .	35

<b>5</b>	<b>Interlude: Process API</b>	<b>37</b>
5.1	The <code>fork()</code> System Call	37
5.2	The <code>wait()</code> System Call	39
5.3	Finally, The <code>exec()</code> System Call	40
5.4	Why? Motivating The API	41
5.5	Other Parts Of The API	44
5.6	Summary	44
	References	45
	Homework (Code)	46
<b>6</b>	<b>Mechanism: Limited Direct Execution</b>	<b>49</b>
6.1	Basic Technique: Limited Direct Execution	49
6.2	Problem #1: Restricted Operations	50
6.3	Problem #2: Switching Between Processes	54
6.4	Worried About Concurrency?	58
6.5	Summary	59
	References	61
	Homework (Measurement)	62
<b>7</b>	<b>Scheduling: Introduction</b>	<b>63</b>
7.1	Workload Assumptions	63
7.2	Scheduling Metrics	64
7.3	First In, First Out (FIFO)	64
7.4	Shortest Job First (SJF)	66
7.5	Shortest Time-to-Completion First (STCF)	67
7.6	A New Metric: Response Time	68
7.7	Round Robin	69
7.8	Incorporating I/O	71
7.9	No More Oracle	72
7.10	Summary	72
	References	73
	Homework	74
<b>8</b>	<b>Scheduling:</b>	
	<b>The Multi-Level Feedback Queue</b>	<b>75</b>
8.1	MLFQ: Basic Rules	76
8.2	Attempt #1: How To Change Priority	77
8.3	Attempt #2: The Priority Boost	80
8.4	Attempt #3: Better Accounting	81
8.5	Tuning MLFQ And Other Issues	82
8.6	MLFQ: Summary	83
	References	85
	Homework	86
<b>9</b>	<b>Scheduling: Proportional Share</b>	<b>87</b>
9.1	Basic Concept: Tickets Represent Your Share	87
9.2	Ticket Mechanisms	89

9.3	Implementation . . . . .	90
9.4	An Example . . . . .	91
9.5	How To Assign Tickets? . . . . .	92
9.6	Why Not Deterministic? . . . . .	92
9.7	Summary . . . . .	93
	References . . . . .	95
	Homework . . . . .	96
<b>10</b>	<b>Multiprocessor Scheduling (Advanced)</b>	<b>97</b>
10.1	Background: Multiprocessor Architecture . . . . .	98
10.2	Don't Forget Synchronization . . . . .	100
10.3	One Final Issue: Cache Affinity . . . . .	101
10.4	Single-Queue Scheduling . . . . .	101
10.5	Multi-Queue Scheduling . . . . .	103
10.6	Linux Multiprocessor Schedulers . . . . .	106
10.7	Summary . . . . .	106
	References . . . . .	107
<b>11</b>	<b>Summary Dialogue on CPU Virtualization</b>	<b>109</b>
<b>12</b>	<b>A Dialogue on Memory Virtualization</b>	<b>111</b>
<b>13</b>	<b>The Abstraction: Address Spaces</b>	<b>113</b>
13.1	Early Systems . . . . .	113
13.2	Multiprogramming and Time Sharing . . . . .	114
13.3	The Address Space . . . . .	115
13.4	Goals . . . . .	117
13.5	Summary . . . . .	119
	References . . . . .	120
<b>14</b>	<b>Interlude: Memory API</b>	<b>123</b>
14.1	Types of Memory . . . . .	123
14.2	The <code>malloc()</code> Call . . . . .	124
14.3	The <code>free()</code> Call . . . . .	126
14.4	Common Errors . . . . .	126
14.5	Underlying OS Support . . . . .	129
14.6	Other Calls . . . . .	130
14.7	Summary . . . . .	130
	References . . . . .	131
	Homework (Code) . . . . .	132
<b>15</b>	<b>Mechanism: Address Translation</b>	<b>135</b>
15.1	Assumptions . . . . .	136
15.2	An Example . . . . .	136
15.3	Dynamic (Hardware-based) Relocation . . . . .	139
15.4	Hardware Support: A Summary . . . . .	142
15.5	Operating System Issues . . . . .	143

15.6 Summary . . . . .	146
References . . . . .	147
Homework . . . . .	148
<b>16 Segmentation . . . . .</b>	<b>149</b>
16.1 Segmentation: Generalized Base/Bounds . . . . .	149
16.2 Which Segment Are We Referring To? . . . . .	152
16.3 What About The Stack? . . . . .	153
16.4 Support for Sharing . . . . .	154
16.5 Fine-grained vs. Coarse-grained Segmentation . . . . .	155
16.6 OS Support . . . . .	155
16.7 Summary . . . . .	157
References . . . . .	158
Homework . . . . .	160
<b>17 Free-Space Management . . . . .</b>	<b>161</b>
17.1 Assumptions . . . . .	162
17.2 Low-level Mechanisms . . . . .	163
17.3 Basic Strategies . . . . .	171
17.4 Other Approaches . . . . .	173
17.5 Summary . . . . .	175
References . . . . .	176
Homework . . . . .	177
<b>18 Paging: Introduction . . . . .</b>	<b>179</b>
18.1 A Simple Example And Overview . . . . .	179
18.2 Where Are Page Tables Stored? . . . . .	183
18.3 What's Actually In The Page Table? . . . . .	184
18.4 Paging: Also Too Slow . . . . .	185
18.5 A Memory Trace . . . . .	186
18.6 Summary . . . . .	189
References . . . . .	190
Homework . . . . .	191
<b>19 Paging: Faster Translations (TLBs) . . . . .</b>	<b>193</b>
19.1 TLB Basic Algorithm . . . . .	193
19.2 Example: Accessing An Array . . . . .	195
19.3 Who Handles The TLB Miss? . . . . .	197
19.4 TLB Contents: What's In There? . . . . .	199
19.5 TLB Issue: Context Switches . . . . .	200
19.6 Issue: Replacement Policy . . . . .	202
19.7 A Real TLB Entry . . . . .	203
19.8 Summary . . . . .	204
References . . . . .	205
Homework (Measurement) . . . . .	207
<b>20 Paging: Smaller Tables . . . . .</b>	<b>211</b>

20.1	Simple Solution: Bigger Pages . . . . .	211
20.2	Hybrid Approach: Paging and Segments . . . . .	212
20.3	Multi-level Page Tables . . . . .	215
20.4	Inverted Page Tables . . . . .	222
20.5	Swapping the Page Tables to Disk . . . . .	223
20.6	Summary . . . . .	223
	References . . . . .	224
	Homework . . . . .	225
<b>21</b>	<b>Beyond Physical Memory: Mechanisms</b>	<b>227</b>
21.1	Swap Space . . . . .	228
21.2	The Present Bit . . . . .	229
21.3	The Page Fault . . . . .	230
21.4	What If Memory Is Full? . . . . .	231
21.5	Page Fault Control Flow . . . . .	232
21.6	When Replacements Really Occur . . . . .	233
21.7	Summary . . . . .	234
	References . . . . .	235
<b>22</b>	<b>Beyond Physical Memory: Policies</b>	<b>237</b>
22.1	Cache Management . . . . .	237
22.2	The Optimal Replacement Policy . . . . .	238
22.3	A Simple Policy: FIFO . . . . .	240
22.4	Another Simple Policy: Random . . . . .	242
22.5	Using History: LRU . . . . .	243
22.6	Workload Examples . . . . .	244
22.7	Implementing Historical Algorithms . . . . .	247
22.8	Approximating LRU . . . . .	248
22.9	Considering Dirty Pages . . . . .	249
22.10	Other VM Policies . . . . .	250
22.11	Thrashing . . . . .	250
22.12	Summary . . . . .	251
	References . . . . .	252
	Homework . . . . .	254
<b>23</b>	<b>The VAX/VMS Virtual Memory System</b>	<b>255</b>
23.1	Background . . . . .	255
23.2	Memory Management Hardware . . . . .	256
23.3	A Real Address Space . . . . .	257
23.4	Page Replacement . . . . .	259
23.5	Other Neat VM Tricks . . . . .	260
23.6	Summary . . . . .	262
	References . . . . .	263
<b>24</b>	<b>Summary Dialogue on Memory Virtualization</b>	<b>265</b>

<b>II Concurrency</b>	<b>269</b>
<b>25 A Dialogue on Concurrency</b>	<b>271</b>
<b>26 Concurrency: An Introduction</b>	<b>273</b>
26.1 An Example: Thread Creation . . . . .	274
26.2 Why It Gets Worse: Shared Data . . . . .	277
26.3 The Heart Of The Problem: Uncontrolled Scheduling . . .	279
26.4 The Wish For Atomicity . . . . .	281
26.5 One More Problem: Waiting For Another . . . . .	283
26.6 Summary: Why in OS Class? . . . . .	283
References . . . . .	285
Homework . . . . .	286
<b>27 Interlude: Thread API</b>	<b>289</b>
27.1 Thread Creation . . . . .	289
27.2 Thread Completion . . . . .	290
27.3 Locks . . . . .	293
27.4 Condition Variables . . . . .	295
27.5 Compiling and Running . . . . .	297
27.6 Summary . . . . .	297
References . . . . .	299
<b>28 Locks</b>	<b>301</b>
28.1 Locks: The Basic Idea . . . . .	301
28.2 Pthread Locks . . . . .	302
28.3 Building A Lock . . . . .	303
28.4 Evaluating Locks . . . . .	303
28.5 Controlling Interrupts . . . . .	304
28.6 Test And Set (Atomic Exchange) . . . . .	306
28.7 Building A Working Spin Lock . . . . .	307
28.8 Evaluating Spin Locks . . . . .	309
28.9 Compare-And-Swap . . . . .	309
28.10 Load-Linked and Store-Conditional . . . . .	311
28.11 Fetch-And-Add . . . . .	312
28.12 Too Much Spinning: What Now? . . . . .	313
28.13 A Simple Approach: Just Yield, Baby . . . . .	314
28.14 Using Queues: Sleeping Instead Of Spinning . . . . .	315
28.15 Different OS, Different Support . . . . .	317
28.16 Two-Phase Locks . . . . .	318
28.17 Summary . . . . .	319
References . . . . .	320
Homework . . . . .	322
<b>29 Lock-based Concurrent Data Structures</b>	<b>325</b>
29.1 Concurrent Counters . . . . .	325
29.2 Concurrent Linked Lists . . . . .	330



29.3	Concurrent Queues . . . . .	333
29.4	Concurrent Hash Table . . . . .	334
29.5	Summary . . . . .	336
	References . . . . .	337
<b>30</b>	<b>Condition Variables</b>	<b>339</b>
30.1	Definition and Routines . . . . .	340
30.2	The Producer/Consumer (Bounded Buffer) Problem . . . .	343
30.3	Covering Conditions . . . . .	351
30.4	Summary . . . . .	352
	References . . . . .	353
<b>31</b>	<b>Semaphores</b>	<b>355</b>
31.1	Semaphores: A Definition . . . . .	355
31.2	Binary Semaphores (Locks) . . . . .	357
31.3	Semaphores As Condition Variables . . . . .	358
31.4	The Producer/Consumer (Bounded Buffer) Problem . . . .	360
31.5	Reader-Writer Locks . . . . .	364
31.6	The Dining Philosophers . . . . .	366
31.7	How To Implement Semaphores . . . . .	369
31.8	Summary . . . . .	370
	References . . . . .	371
<b>32</b>	<b>Common Concurrency Problems</b>	<b>373</b>
32.1	What Types Of Bugs Exist? . . . . .	373
32.2	Non-Deadlock Bugs . . . . .	374
32.3	Deadlock Bugs . . . . .	377
32.4	Summary . . . . .	385
	References . . . . .	386
<b>33</b>	<b>Event-based Concurrency (Advanced)</b>	<b>389</b>
33.1	The Basic Idea: An Event Loop . . . . .	389
33.2	An Important API: <code>select()</code> (or <code>poll()</code> ) . . . . .	390
33.3	Using <code>select()</code> . . . . .	391
33.4	Why Simpler? No Locks Needed . . . . .	392
33.5	A Problem: Blocking System Calls . . . . .	393
33.6	A Solution: Asynchronous I/O . . . . .	393
33.7	Another Problem: State Management . . . . .	396
33.8	What Is Still Difficult With Events . . . . .	397
33.9	Summary . . . . .	397
	References . . . . .	398
<b>34</b>	<b>Summary Dialogue on Concurrency</b>	<b>399</b>

<b>III Persistence</b>	<b>401</b>
<b>35 A Dialogue on Persistence</b>	<b>403</b>
<b>36 I/O Devices</b>	<b>405</b>
36.1 System Architecture . . . . .	405
36.2 A Canonical Device . . . . .	406
36.3 The Canonical Protocol . . . . .	407
36.4 Lowering CPU Overhead With Interrupts . . . . .	408
36.5 More Efficient Data Movement With DMA . . . . .	409
36.6 Methods Of Device Interaction . . . . .	410
36.7 Fitting Into The OS: The Device Driver . . . . .	411
36.8 Case Study: A Simple IDE Disk Driver . . . . .	412
36.9 Historical Notes . . . . .	415
36.10 Summary . . . . .	415
References . . . . .	416
<b>37 Hard Disk Drives</b>	<b>419</b>
37.1 The Interface . . . . .	419
37.2 Basic Geometry . . . . .	420
37.3 A Simple Disk Drive . . . . .	421
37.4 I/O Time: Doing The Math . . . . .	424
37.5 Disk Scheduling . . . . .	428
37.6 Summary . . . . .	432
References . . . . .	433
Homework . . . . .	434
<b>38 Redundant Arrays of Inexpensive Disks (RAIDs)</b>	<b>437</b>
38.1 Interface And RAID Internals . . . . .	438
38.2 Fault Model . . . . .	439
38.3 How To Evaluate A RAID . . . . .	439
38.4 RAID Level 0: Striping . . . . .	440
38.5 RAID Level 1: Mirroring . . . . .	443
38.6 RAID Level 4: Saving Space With Parity . . . . .	446
38.7 RAID Level 5: Rotating Parity . . . . .	450
38.8 RAID Comparison: A Summary . . . . .	451
38.9 Other Interesting RAID Issues . . . . .	452
38.10 Summary . . . . .	452
References . . . . .	453
Homework . . . . .	455
<b>39 Interlude: File and Directories</b>	<b>457</b>
39.1 Files and Directories . . . . .	457
39.2 The File System Interface . . . . .	459
39.3 Creating Files . . . . .	459
39.4 Reading and Writing Files . . . . .	460
39.5 Reading And Writing, But Not Sequentially . . . . .	462

39.6	Writing Immediately with <code>fsync()</code> . . . . .	463
39.7	Renaming Files . . . . .	464
39.8	Getting Information About Files . . . . .	465
39.9	Removing Files . . . . .	466
39.10	Making Directories . . . . .	466
39.11	Reading Directories . . . . .	467
39.12	Deleting Directories . . . . .	468
39.13	Hard Links . . . . .	468
39.14	Symbolic Links . . . . .	470
39.15	Making and Mounting a File System . . . . .	472
39.16	Summary . . . . .	473
	References . . . . .	474
	Homework . . . . .	475
<b>40</b>	<b>File System Implementation</b> . . . . .	<b>477</b>
40.1	The Way To Think . . . . .	477
40.2	Overall Organization . . . . .	478
40.3	File Organization: The Inode . . . . .	480
40.4	Directory Organization . . . . .	485
40.5	Free Space Management . . . . .	485
40.6	Access Paths: Reading and Writing . . . . .	486
40.7	Caching and Buffering . . . . .	490
40.8	Summary . . . . .	492
	References . . . . .	493
	Homework . . . . .	494
<b>41</b>	<b>Locality and The Fast File System</b> . . . . .	<b>495</b>
41.1	The Problem: Poor Performance . . . . .	495
41.2	FFS: Disk Awareness Is The Solution . . . . .	497
41.3	Organizing Structure: The Cylinder Group . . . . .	497
41.4	Policies: How To Allocate Files and Directories . . . . .	498
41.5	Measuring File Locality . . . . .	499
41.6	The Large-File Exception . . . . .	500
41.7	A Few Other Things About FFS . . . . .	502
41.8	Summary . . . . .	504
	References . . . . .	505
<b>42</b>	<b>Crash Consistency: FSCK and Journaling</b> . . . . .	<b>507</b>
42.1	A Detailed Example . . . . .	508
42.2	Solution #1: The File System Checker . . . . .	511
42.3	Solution #2: Journaling (or Write-Ahead Logging) . . . . .	513
42.4	Solution #3: Other Approaches . . . . .	523
42.5	Summary . . . . .	524
	References . . . . .	525
<b>43</b>	<b>Log-structured File Systems</b> . . . . .	<b>527</b>
43.1	Writing To Disk Sequentially . . . . .	528

43.2	Writing Sequentially And Effectively . . . . .	529
43.3	How Much To Buffer? . . . . .	530
43.4	Problem: Finding Inodes . . . . .	531
43.5	Solution Through Indirection: The Inode Map . . . . .	531
43.6	The Checkpoint Region . . . . .	532
43.7	Reading A File From Disk: A Recap . . . . .	533
43.8	What About Directories? . . . . .	533
43.9	A New Problem: Garbage Collection . . . . .	534
43.10	Determining Block Liveness . . . . .	536
43.11	A Policy Question: Which Blocks To Clean, And When? . . . . .	537
43.12	Crash Recovery And The Log . . . . .	537
43.13	Summary . . . . .	538
	References . . . . .	540
<b>44</b>	<b>Data Integrity and Protection</b>	<b>543</b>
44.1	Disk Failure Modes . . . . .	543
44.2	Handling Latent Sector Errors . . . . .	545
44.3	Detecting Corruption: The Checksum . . . . .	546
44.4	Using Checksums . . . . .	549
44.5	A New Problem: Misdirected Writes . . . . .	550
44.6	One Last Problem: Lost Writes . . . . .	551
44.7	Scrubbing . . . . .	551
44.8	Overheads Of Checksumming . . . . .	552
44.9	Summary . . . . .	552
	References . . . . .	553
<b>45</b>	<b>Summary Dialogue on Persistence</b>	<b>555</b>
<b>46</b>	<b>A Dialogue on Distribution</b>	<b>557</b>
<b>47</b>	<b>Distributed Systems</b>	<b>559</b>
47.1	Communication Basics . . . . .	560
47.2	Unreliable Communication Layers . . . . .	561
47.3	Reliable Communication Layers . . . . .	563
47.4	Communication Abstractions . . . . .	565
47.5	Remote Procedure Call (RPC) . . . . .	567
47.6	Summary . . . . .	572
	References . . . . .	573
<b>48</b>	<b>Sun's Network File System (NFS)</b>	<b>575</b>
48.1	A Basic Distributed File System . . . . .	576
48.2	On To NFS . . . . .	577
48.3	Focus: Simple and Fast Server Crash Recovery . . . . .	577
48.4	Key To Fast Crash Recovery: Statelessness . . . . .	578
48.5	The NFSv2 Protocol . . . . .	579
48.6	From Protocol to Distributed File System . . . . .	581
48.7	Handling Server Failure with Idempotent Operations . . . . .	583

48.8	Improving Performance: Client-side Caching . . . . .	585
48.9	The Cache Consistency Problem . . . . .	585
48.10	Assessing NFS Cache Consistency . . . . .	587
48.11	Implications on Server-Side Write Buffering . . . . .	587
48.12	Summary . . . . .	589
	References . . . . .	590
<b>49</b>	<b>The Andrew File System (AFS)</b>	<b>591</b>
49.1	AFS Version 1 . . . . .	591
49.2	Problems with Version 1 . . . . .	592
49.3	Improving the Protocol . . . . .	594
49.4	AFS Version 2 . . . . .	594
49.5	Cache Consistency . . . . .	596
49.6	Crash Recovery . . . . .	598
49.7	Scale And Performance Of AFSv2 . . . . .	598
49.8	AFS: Other Improvements . . . . .	600
49.9	Summary . . . . .	601
	References . . . . .	603
	Homework . . . . .	604
<b>50</b>	<b>Summary Dialogue on Distribution</b>	<b>605</b>
	<b>General Index</b>	<b>607</b>
	<b>Asides</b>	<b>617</b>
	<b>Tips</b>	<b>619</b>
	<b>Cruces</b>	<b>621</b>

---

## Preface

### To Everyone

Welcome to this book! We hope you'll enjoy reading it as much as we enjoyed writing it. The book is called **Operating Systems: Three Easy Pieces**, and the title is obviously an homage to one of the greatest sets of lecture notes ever created, by one Richard Feynman on the topic of Physics [F96]. While this book will undoubtedly fall short of the high standard set by that famous physicist, perhaps it will be good enough for you in your quest to understand what operating systems (and more generally, systems) are all about.

The three easy pieces refer to the three major thematic elements the book is organized around: **virtualization**, **concurrency**, and **persistence**. In discussing these concepts, we'll end up discussing most of the important things an operating system does; hopefully, you'll also have some fun along the way. Learning new things is fun, right? At least, it should be.

Each major concept is divided into a set of chapters, most of which present a particular problem and then show how to solve it. The chapters are short, and try (as best as possible) to reference the source material where the ideas really came from. One of our goals in writing this book is to make the paths of history as clear as possible, as we think that helps a student understand what is, what was, and what will be more clearly. In this case, seeing how the sausage was made is nearly as important as understanding what the sausage is good for<sup>1</sup>.

There are a couple devices we use throughout the book which are probably worth introducing here. The first is the **crux** of the problem. Anytime we are trying to solve a problem, we first try to state what the most important issue is; such a **crux of the problem** is explicitly called out in the text, and hopefully solved via the techniques, algorithms, and ideas presented in the rest of the text.

In many places, we'll explain how a system works by showing its behavior over time. These **timelines** are at the essence of understanding; if you know what happens, for example, when a process page faults, you are on your way to truly understanding how virtual memory operates. If you comprehend what takes place when a journaling file system writes a block to disk, you have taken the first steps towards mastery of storage systems.

There are also numerous **asides** and **tips** throughout the text, adding a little color to the mainline presentation. Asides tend to discuss something relevant (but perhaps not essential) to the main text; tips tend to be general lessons that can be

---

<sup>1</sup>Hint: eating! Or if you're a vegetarian, running away from.

applied to systems you build. An index at the end of the book lists all of these tips and asides (as well as cruces, the odd plural of crux) for your convenience.

We use one of the oldest didactic methods, the **dialogue**, throughout the book, as a way of presenting some of the material in a different light. These are used to introduce the major thematic concepts (in a peachy way, as we will see), as well as to review material every now and then. They are also a chance to write in a more humorous style. Whether you find them useful, or humorous, well, that's another matter entirely.

At the beginning of each major section, we'll first present an **abstraction** that an operating system provides, and then work in subsequent chapters on the mechanisms, policies, and other support needed to provide the abstraction. Abstractions are fundamental to all aspects of Computer Science, so it is perhaps no surprise that they are also essential in operating systems.

Throughout the chapters, we try to use **real code** (not **pseudocode**) where possible, so for virtually all examples, you should be able to type them up yourself and run them. Running real code on real systems is the best way to learn about operating systems, so we encourage you to do so when you can.

In various parts of the text, we have sprinkled in a few **homeworks** to ensure that you are understanding what is going on. Many of these homeworks are little **simulations** of pieces of the operating system; you should download the homeworks, and run them to quiz yourself. The homework simulators have the following feature: by giving them a different random seed, you can generate a virtually infinite set of problems; the simulators can also be told to solve the problems for you. Thus, you can test and re-test yourself until you have achieved a good level of understanding.

The most important addendum to this book is a set of **projects** in which you learn about how real systems work by designing, implementing, and testing your own code. All projects (as well as the code examples, mentioned above) are in the **C programming language** [KR88]; C is a simple and powerful language that underlies most operating systems, and thus worth adding to your tool-chest of languages. Two types of projects are available (see the online appendix for ideas). The first are **systems programming** projects; these projects are great for those who are new to C and UNIX and want to learn how to do low-level C programming. The second type are based on a real operating system kernel developed at MIT called xv6 [CK+08]; these projects are great for students that already have some C and want to get their hands dirty inside the OS. At Wisconsin, we've run the course in three different ways: either all systems programming, all xv6 programming, or a mix of both.

## To Educators

If you are an instructor or professor who wishes to use this book, please feel free to do so. As you may have noticed, they are free and available on-line from the following web page:

<http://www.ostep.org>

You can also purchase a printed copy from [lulu.com](http://lulu.com). Look for it on the web page above.

The (current) proper citation for the book is as follows:

**Operating Systems: Three Easy Pieces**

Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau

Arpaci-Dusseau Books

March, 2015 (Version 0.90)

<http://www.ostep.org>

The course divides fairly well across a 15-week semester, in which you can cover most of the topics within at a reasonable level of depth. Cramming the course into a 10-week quarter probably requires dropping some detail from each of the pieces. There are also a few chapters on virtual machine monitors, which we usually squeeze in sometime during the semester, either right at end of the large section on virtualization, or near the end as an aside.

One slightly unusual aspect of the book is that concurrency, a topic at the front of many OS books, is pushed off herein until the student has built an understanding of virtualization of the CPU and of memory. In our experience in teaching this course for nearly 15 years, students have a hard time understanding how the concurrency problem arises, or why they are trying to solve it, if they don't yet understand what an address space is, what a process is, or why context switches can occur at arbitrary points in time. Once they do understand these concepts, however, introducing the notion of threads and the problems that arise due to them becomes rather easy, or at least, easier.

As much as is possible, we use a chalkboard (or whiteboard) to deliver a lecture. On these more conceptual days, we come to class with a few major ideas and examples in mind and use the board to present them. Handouts are useful to give the students concrete problems to solve based on the material. On more practical days, we simply plug a laptop into the projector and show real code; this style works particularly well for concurrency lectures as well as for any discussion sections where you show students code that is relevant for their projects. We don't generally use slides to present material, but have now made a set available for those who prefer that style of presentation.

If you'd like a copy of any of these materials, please drop us an email. We have already shared them with many others around the world.

One last request: if you use the free online chapters, please just [link](#) to them, instead of making a local copy. This helps us track usage (over 1 million chapters downloaded in the past few years!) and also ensures students get the latest and greatest version.



## To Students

If you are a student reading this book, thank you! It is an honor for us to provide some material to help you in your pursuit of knowledge about operating systems. We both think back fondly towards some textbooks of our undergraduate days (e.g., Hennessy and Patterson [HP90], the classic book on computer architecture) and hope this book will become one of those positive memories for you.

You may have noticed this book is free and available online<sup>2</sup>. There is one major reason for this: textbooks are generally too expensive. This book, we hope, is the first of a new wave of free materials to help those in pursuit of their education, regardless of which part of the world they come from or how much they are willing to spend for a book. Failing that, it is one free book, which is better than none.

We also hope, where possible, to point you to the original sources of much of the material in the book: the great papers and persons who have shaped the field of operating systems over the years. Ideas are not pulled out of the air; they come from smart and hard-working people (including numerous Turing-award winners<sup>3</sup>), and thus we should strive to celebrate those ideas and people where possible. In doing so, we hopefully can better understand the revolutions that have taken place, instead of writing texts as if those thoughts have always been present [K62]. Further, perhaps such references will encourage you to dig deeper on your own; reading the famous papers of our field is certainly one of the best ways to learn.

---

<sup>2</sup>A digression here: “free” in the way we use it here does not mean open source, and it does not mean the book is not copyrighted with the usual protections – it is! What it means is that you can download the chapters and use them to learn about operating systems. Why not an open-source book, just like Linux is an open-source kernel? Well, we believe it is important for a book to have a single voice throughout, and have worked hard to provide such a voice. When you’re reading it, the book should kind of feel like a dialogue with the person explaining something to you. Hence, our approach.

<sup>3</sup>The Turing Award is the highest award in Computer Science; it is like the Nobel Prize, except that you have never heard of it.

## Acknowledgments

This section will contain thanks to those who helped us put the book together. The important thing for now: **your name could go here!** But, you have to help. So send us some feedback and help debug this book. And you could be famous! Or, at least, have your name in some book.

The people who have helped so far include: Abhirami Senthilkumaran\*, Adam Drescher\* (WUSTL), Adam Eggum, Aditya Venkataraman, Adriana Iamnitshi and class (USF), Ahmed Fikri\*, Ajaykrishna Raghavan, Akiel Khan, Alex Wyler, Ali Razeen (Duke), AmirBehzad Eslami, Anand Mundada, Andrew Valencik (Saint Mary's), Angela Demke Brown (Toronto), B. Brahmananda Reddy (Minnesota), Bala Subrahmanyam Kambala, Benita Bose, Biswajit Mazumder (Clemson), Bobby Jack, Björn Lindberg, Brennan Payne, Brian Gorman, Brian Kroth, Caleb Sumner (Southern Adventist), Cara Lauritzen, Charlotte Kissinger, Chien-Chung Shen (Delaware)\*, Christoph Jaeger, Cody Hanson, Dan Soendergaard (U. Aarhus), David Hanle (Grinnell), David Hartman, Deepika Muthukumar, Dheeraj Shetty (North Carolina State), Dorian Arnold (New Mexico), Dustin Metzler, Dustin Passofaro, Eduardo Stelmaszczyk, Emad Sadeghi, Emily Jacobson, Emmett Witchel (Texas), Erik Turk, Ernst Biersack (France), Finn Kuusisto\*, Glen Granzow (College of Idaho), Guilherme Baptista, Hamid Reza Ghasemi, Hao Chen, Henry Abbey, Hrishikesh Amur, Huanchen Zhang\*, Huseyin Sular, Hugo Diaz, Itai Hass (Toronto), Jake Gillberg, Jakob Olandt, James Perry (U. Michigan-Dearborn)\*, Jan Reineke (Universität des Saarlandes), Jay Lim, Jerod Weinman (Grinnell), Jiao Dong (Rutgers), Jingxin Li, Joe Jean (NYU), Joel Kuntz (Saint Mary's), Joel Sommers (Colgate), John Brady (Grinnell), Jonathan Perry (MIT), Jun He, Karl Wallinger, Kartik Singhal, Kaushik Kannan, Kevin Liu\*, Lei Tian (U. Nebraska-Lincoln), Leslie Schultz, Liang Yin, Lihao Wang, Martha Ferris, Masashi Kishikawa (Sony), Matt Reichoff, Matty Williams, Meng Huang, Michael Walfish (NYU), Mike Griepentrog, Ming Chen (Stonybrook), Mohammed Alali (Delaware), Murugan Kandaswamy, Natasha Eilbert, Nathan Dipiazza, Nathan Sullivan, Neeraj Badlani (N.C. State), Nelson Gomez, Nghia Huynh (Texas), Nick Weinandt, Patricio Jara, Perry Kivolowitz, Radford Smith, Riccardo Mutschlechner, Ripudaman Singh, Robert Ordóñez and class (Southern Adventist), Rohan Das (Toronto)\*, Rohan Pasalkar (Minnesota), Ross Aiken, Ruslan Kiselev, Ryland Herrick, Samer Al-Kiswani, Sandeep Ummadi (Minnesota), Satish Chebrolu (NetApp), Satyanarayana Shanmugam\*, Seth Pollen, Sharad Punuganti, Shreevatsa R., Sivaraman Sivaraman\*, Srinivasan Thirunarayanan\*, Suriyaparakhas Balaram Sankari, Sy Jin Cheah, Teri Zhao (EMC), Thomas Griebel, Tongxin Zheng, Tony Adkins, Torin Rudeen (Princeton), Tuo Wang, Varun Vats, William Royle (Grinnell), Xiang Peng, Xu Di, Yudong Sun, Yue Zhuo (Texas A&M), Yufui Ren, Zef RosnBrick, Zuyu Zhang. Special thanks to those marked with an asterisk above, who have gone above and beyond in their suggestions for improvement.

In addition, a hearty thanks to Professor Joe Meehan (Lynchburg) for his detailed notes on each chapter, to Professor Jerod Weinman (Grinnell) and his entire class for their incredible booklets, to Professor Chien-Chung Shen (Delaware) for his invaluable and detailed reading and comments, to Adam Drescher (WUSTL) for his careful reading and suggestions, to Glen Granzow (College of Idaho) for his detailed comments and tips, and Michael Walfish (NYU) for his enthusiasm and detailed suggestions for improvement. All have helped these authors immeasur-

ably in the refinement of the materials herein.

Also, many thanks to the hundreds of students who have taken 537 over the years. In particular, the Fall '08 class who encouraged the first written form of these notes (they were sick of not having any kind of textbook to read — pushy students!), and then praised them enough for us to keep going (including one hilarious “ZOMG! You should totally write a textbook!” comment in our course evaluations that year).

A great debt of thanks is also owed to the brave few who took the xv6 project lab course, much of which is now incorporated into the main 537 course. From Spring '09: Justin Cherniak, Patrick Deline, Matt Czech, Tony Gregerson, Michael Griepentrog, Tyler Harter, Ryan Kroiss, Eric Radzikowski, Wesley Reardan, Rajiv Vaidyanathan, and Christopher Waclawik. From Fall '09: Nick Bearson, Aaron Brown, Alex Bird, David Capel, Keith Gould, Tom Grim, Jeffrey Hugo, Brandon Johnson, John Kjell, Boyan Li, James Loethen, Will McCardell, Ryan Szaroletta, Simon Tso, and Ben Yule. From Spring '10: Patrick Blesi, Aidan Dennis-Oehling, Paras Doshi, Jake Friedman, Benjamin Frisch, Evan Hanson, Pikkili Hemanth, Michael Jeung, Alex Langenfeld, Scott Rick, Mike Treffert, Garret Staus, Brennan Wall, Hans Werner, Soo-Young Yang, and Carlos Griffin (almost).

Although they do not directly help with the book, our graduate students have taught us much of what we know about systems. We talk with them regularly while they are at Wisconsin, but they do all the real work — and by telling us about what they are doing, we learn new things every week. This list includes the following collection of current and former students with whom we have published papers; an asterisk marks those who received a Ph.D. under our guidance: Abhishek Rajimwale, Andrew Krioukov, Ao Ma, Brian Forney, Chris Dragga, Deepak Ramamurthi, Florentina Popovici\*, Haryadi S. Gunawi\*, James Nugent, John Bent\*, Jun He, Lanyue Lu, Lakshmi Bairavasundaram\*, Laxman Visampalli, Leo Arulraj, Meenali Rungta, Muthian Sivathanu\*, Nathan Burnett\*, Nitin Agrawal\*, Ram Alagappan, Sriram Subramanian\*, Stephen Todd Jones\*, Suli Yang, Swaminathan Sundararaman\*, Swetha Krishnan, Thanh Do\*, Thanumalayan S. Pillai, Timothy Denehy\*, Tyler Harter, Venkat Venkataramani, Vijay Chidambaram, Vijayan Prabhakaran\*, Yiyang Zhang\*, Yupu Zhang\*, Zev Weiss.

A final debt of gratitude is also owed to Aaron Brown, who first took this course many years ago (Spring '09), then took the xv6 lab course (Fall '09), and finally was a graduate teaching assistant for the course for two years or so (Fall '10 through Spring '12). His tireless work has vastly improved the state of the projects (particularly those in xv6 land) and thus has helped better the learning experience for countless undergraduates and graduates here at Wisconsin. As Aaron would say (in his usual succinct manner): “Thx.”

## Final Words

Yeats famously said “Education is not the filling of a pail but the lighting of a fire.” He was right but wrong at the same time<sup>4</sup>. You do have to “fill the pail” a bit, and these notes are certainly here to help with that part of your education; after all, when you go to interview at Google, and they ask you a trick question about how to use semaphores, it might be good to actually know what a semaphore is, right?

But Yeats’s larger point is obviously on the mark: the real point of education is to get you interested in something, to learn something more about the subject matter on your own and not just what you have to digest to get a good grade in some class. As one of our fathers (Remzi’s dad, Vedat Arpacı) used to say, “Learn beyond the classroom”.

We created these notes to spark your interest in operating systems, to read more about the topic on your own, to talk to your professor about all the exciting research that is going on in the field, and even to get involved with that research. It is a great field(!), full of exciting and wonderful ideas that have shaped computing history in profound and important ways. And while we understand this fire won’t light for all of you, we hope it does for many, or even a few. Because once that fire is lit, well, that is when you truly become capable of doing something great. And thus the real point of the educational process: to go forth, to study many new and fascinating topics, to learn, to mature, and most importantly, to find something that lights a fire for you.

*Andrea and Remzi*

*Married couple*

*Professors of Computer Science at the University of Wisconsin*

*Chief Lighters of Fires, hopefully<sup>5</sup>*

---

<sup>4</sup>If he actually said this; as with many famous quotes, the history of this gem is murky.

<sup>5</sup>If this sounds like we are admitting some past history as arsonists, you are probably missing the point. Probably. If this sounds cheesy, well, that’s because it is, but you’ll just have to forgive us for that.

## References

[CK+08] “The xv6 Operating System”

Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich

From: <http://pdos.csail.mit.edu/6.828/2008/index.html>

*xv6 was developed as a port of the original UNIX version 6 and represents a beautiful, clean, and simple way to understand a modern operating system.*

[F96] “Six Easy Pieces: Essentials Of Physics Explained By Its Most Brilliant Teacher”

Richard P. Feynman

Basic Books, 1996

*This book reprints the six easiest chapters of Feynman’s Lectures on Physics, from 1963. If you like Physics, it is a fantastic read.*

[HP90] “Computer Architecture a Quantitative Approach” (1st ed.)

David A. Patterson and John L. Hennessy

Morgan-Kaufman, 1990

*A book that encouraged each of us at our undergraduate institutions to pursue graduate studies; we later both had the pleasure of working with Patterson, who greatly shaped the foundations of our research careers.*

[KR88] “The C Programming Language”

Brian Kernighan and Dennis Ritchie

Prentice-Hall, April 1988

*The C programming reference that everyone should have, by the people who invented the language.*

[K62] “The Structure of Scientific Revolutions”

Thomas S. Kuhn

University of Chicago Press, 1962

*A great and famous read about the fundamentals of the scientific process. Mop-up work, anomaly, crisis, and revolution. We are mostly destined to do mop-up work, alas.*

## A Dialogue on the Book

**Professor:** *Welcome to this book! It's called **Operating Systems in Three Easy Pieces**, and I am here to teach you the things you need to know about operating systems. I am called "Professor"; who are you?*

**Student:** *Hi Professor! I am called "Student", as you might have guessed. And I am here and ready to learn!*

**Professor:** *Sounds good. Any questions?*

**Student:** *Sure! Why is it called "Three Easy Pieces"?*

**Professor:** *That's an easy one. Well, you see, there are these great lectures on Physics by Richard Feynman...*

**Student:** *Oh! The guy who wrote "Surely You're Joking, Mr. Feynman", right? Great book! Is this going to be hilarious like that book was?*

**Professor:** *Um... well, no. That book was great, and I'm glad you've read it. Hopefully this book is more like his notes on Physics. Some of the basics were summed up in a book called "Six Easy Pieces". He was talking about Physics; we're going to do Three Easy Pieces on the fine topic of Operating Systems. This is appropriate, as Operating Systems are about half as hard as Physics.*

**Student:** *Well, I liked physics, so that is probably good. What are those pieces?*

**Professor:** *They are the three key ideas we're going to learn about: **virtualization**, **concurrency**, and **persistence**. In learning about these ideas, we'll learn all about how an operating system works, including how it decides what program to run next on a CPU, how it handles memory overload in a virtual memory system, how virtual machine monitors work, how to manage information on disks, and even a little about how to build a distributed system that works when parts have failed. That sort of stuff.*

**Student:** *I have no idea what you're talking about, really.*

**Professor:** *Good! That means you are in the right class.*

**Student:** *I have another question: what's the best way to learn this stuff?*

**Professor:** *Excellent query! Well, each person needs to figure this out on their*

*own, of course, but here is what I would do: go to class, to hear the professor introduce the material. Then, at the end of every week, read these notes, to help the ideas sink into your head a bit better. Of course, some time later (hint: before the exam!), read the notes again to firm up your knowledge. Of course, your professor will no doubt assign some homeworks and projects, so you should do those; in particular, doing projects where you write real code to solve real problems is the best way to put the ideas within these notes into action. As Confucius said...*

**Student:** *Oh, I know! 'I hear and I forget. I see and I remember. I do and I understand.' Or something like that.*

**Professor:** *(surprised) How did you know what I was going to say?!*

**Student:** *It seemed to follow. Also, I am a big fan of Confucius, and an even bigger fan of Xunzi, who actually is a better source for this quote<sup>1</sup>.*

**Professor:** *(stunned) Well, I think we are going to get along just fine! Just fine indeed.*

**Student:** *Professor – just one more question, if I may. What are these dialogues for? I mean, isn't this just supposed to be a book? Why not present the material directly?*

**Professor:** *Ah, good question, good question! Well, I think it is sometimes useful to pull yourself outside of a narrative and think a bit; these dialogues are those times. So you and I are going to work together to make sense of all of these pretty complex ideas. Are you up for it?*

**Student:** *So we have to think? Well, I'm up for that. I mean, what else do I have to do anyhow? It's not like I have much of a life outside of this book.*

**Professor:** *Me neither, sadly. So let's get to work!*

---

<sup>1</sup> According to this website ([http://www.barrypopik.com/index.php/new\\_york\\_city/entry/tell\\_me\\_and\\_i\\_forget\\_teach\\_me\\_and\\_i\\_may\\_remember\\_involve\\_me\\_and\\_i\\_will\\_learn/](http://www.barrypopik.com/index.php/new_york_city/entry/tell_me_and_i_forget_teach_me_and_i_may_remember_involve_me_and_i_will_learn/)), Confucian philosopher Xunzi said "Not having heard something is not as good as having heard it; having heard it is not as good as having seen it; having seen it is not as good as knowing it; knowing it is not as good as putting it into practice." Later on, the wisdom got attached to Confucius for some reason. Thanks to Jiao Dong (Rutgers) for telling us!

## Introduction to Operating Systems

If you are taking an undergraduate operating systems course, you should already have some idea of what a computer program does when it runs. If not, this book (and the corresponding course) is going to be difficult — so you should probably stop reading this book, or run to the nearest bookstore and quickly consume the necessary background material before continuing (both Patt/Patel [PP03] and particularly Bryant/O'Hallaron [BOH10] are pretty great books).

So what happens when a program runs?

Well, a running program does one very simple thing: it executes instructions. Many millions (and these days, even billions) of times every second, the processor **fetches** an instruction from memory, **decodes** it (i.e., figures out which instruction this is), and **executes** it (i.e., it does the thing that it is supposed to do, like add two numbers together, access memory, check a condition, jump to a function, and so forth). After it is done with this instruction, the processor moves on to the next instruction, and so on, and so on, until the program finally completes<sup>1</sup>.

Thus, we have just described the basics of the **Von Neumann** model of computing<sup>2</sup>. Sounds simple, right? But in this class, we will be learning that while a program runs, a lot of other wild things are going on with the primary goal of making the system **easy to use**.

There is a body of software, in fact, that is responsible for making it easy to run programs (even allowing you to seemingly run many at the same time), allowing programs to share memory, enabling programs to interact with devices, and other fun stuff like that. That body of software

---

<sup>1</sup>Of course, modern processors do many bizarre and frightening things underneath the hood to make programs run faster, e.g., executing multiple instructions at once, and even issuing and completing them out of order! But that is not our concern here; we are just concerned with the simple model most programs assume: that instructions seemingly execute one at a time, in an orderly and sequential fashion.

<sup>2</sup>Von Neumann was one of the early pioneers of computing systems. He also did pioneering work on game theory and atomic bombs, and played in the NBA for six years. OK, one of those things isn't true.



THE CRUX OF THE PROBLEM:  
HOW TO VIRTUALIZE RESOURCES

One central question we will answer in this book is quite simple: how does the operating system virtualize resources? This is the crux of our problem. *Why* the OS does this is not the main question, as the answer should be obvious: it makes the system easier to use. Thus, we focus on the *how*: what mechanisms and policies are implemented by the OS to attain virtualization? How does the OS do so efficiently? What hardware support is needed?

We will use the “crux of the problem”, in shaded boxes such as this one, as a way to call out specific problems we are trying to solve in building an operating system. Thus, within a note on a particular topic, you may find one or more *cruces* (yes, this is the proper plural) which highlight the problem. The details within the chapter, of course, present the solution, or at least the basic parameters of a solution.

is called the **operating system (OS)**<sup>3</sup>, as it is in charge of making sure the system operates correctly and efficiently in an easy-to-use manner.

The primary way the OS does this is through a general technique that we call **virtualization**. That is, the OS takes a **physical** resource (such as the processor, or memory, or a disk) and transforms it into a more general, powerful, and easy-to-use **virtual** form of itself. Thus, we sometimes refer to the operating system as a **virtual machine**.

Of course, in order to allow users to tell the OS what to do and thus make use of the features of the virtual machine (such as running a program, or allocating memory, or accessing a file), the OS also provides some interfaces (APIs) that you can call. A typical OS, in fact, exports a few hundred **system calls** that are available to applications. Because the OS provides these calls to run programs, access memory and devices, and other related actions, we also sometimes say that the OS provides a **standard library** to applications.

Finally, because virtualization allows many programs to run (thus sharing the CPU), and many programs to concurrently access their own instructions and data (thus sharing memory), and many programs to access devices (thus sharing disks and so forth), the OS is sometimes known as a **resource manager**. Each of the CPU, memory, and disk is a **resource** of the system; it is thus the operating system’s role to **manage** those resources, doing so efficiently or fairly or indeed with many other possible goals in mind. To understand the role of the OS a little bit better, let’s take a look at some examples.

<sup>3</sup> Another early name for the OS was the **supervisor** or even the **master control program**. Apparently, the latter sounded a little overzealous (see the movie *Tron* for details) and thus, thankfully, “operating system” caught on instead.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/time.h>
4  #include <assert.h>
5  #include "common.h"
6
7  int
8  main(int argc, char *argv[])
9  {
10     if (argc != 2) {
11         fprintf(stderr, "usage: cpu <string>\n");
12         exit(1);
13     }
14     char *str = argv[1];
15     while (1) {
16         Spin(1);
17         printf("%s\n", str);
18     }
19     return 0;
20 }

```

Figure 2.1: Simple Example: Code That Loops and Prints (`cpu.c`)

## 2.1 Virtualizing the CPU

Figure 2.1 depicts our first program. It doesn't do much. In fact, all it does is call `Spin()`, a function that repeatedly checks the time and returns once it has run for a second. Then, it prints out the string that the user passed in on the command line, and repeats, forever.

Let's say we save this file as `cpu.c` and decide to compile and run it on a system with a single processor (or **CPU** as we will sometimes call it). Here is what we will see:

```

prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
A
^C
prompt>

```

Not too interesting of a run — the system begins running the program, which repeatedly checks the time until a second has elapsed. Once a second has passed, the code prints the input string passed in by the user (in this example, the letter "A"), and continues. Note the program will run forever; only by pressing "Control-c" (which on UNIX-based systems will terminate the program running in the foreground) can we halt the program.

Now, let's do the same thing, but this time, let's run many different instances of this same program. Figure 2.2 shows the results of this slightly more complicated example.

```

prompt> ./cpu A & ; ./cpu B & ; ./cpu C & ; ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
C
B
D
...
```

Figure 2.2: Running Many Programs At Once

Well, now things are getting a little more interesting. Even though we have only one processor, somehow all four of these programs seem to be running at the same time! How does this magic happen?<sup>4</sup>

It turns out that the operating system, with some help from the hardware, is in charge of this **illusion**, i.e., the illusion that the system has a very large number of virtual CPUs. Turning a single CPU (or small set of them) into a seemingly infinite number of CPUs and thus allowing many programs to seemingly run at once is what we call **virtualizing the CPU**, the focus of the first major part of this book.

Of course, to run programs, and stop them, and otherwise tell the OS which programs to run, there need to be some interfaces (APIs) that you can use to communicate your desires to the OS. We'll talk about these APIs throughout this book; indeed, they are the major way in which most users interact with operating systems.

You might also notice that the ability to run multiple programs at once raises all sorts of new questions. For example, if two programs want to run at a particular time, which *should* run? This question is answered by a **policy** of the OS; policies are used in many different places within an OS to answer these types of questions, and thus we will study them as we learn about the basic **mechanisms** that operating systems implement (such as the ability to run multiple programs at once). Hence the role of the OS as a **resource manager**.

---

<sup>4</sup>Note how we ran four processes at the same time, by using the `&` symbol. Doing so runs a job in the background in the `tcsh` shell, which means that the user is able to immediately issue their next command, which in this case is another program to run. The semi-colon between commands allows us to run multiple programs at the same time in `tcsh`. If you're using a different shell (e.g., `bash`), it works slightly differently; read documentation online for details.

```

1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "common.h"
5
6  int
7  main(int argc, char *argv[])
8  {
9      int *p = malloc(sizeof(int));          // a1
10     assert(p != NULL);
11     printf("(d) memory address of p: %08x\n",
12            getpid(), (unsigned) p);         // a2
13     *p = 0;                                // a3
14     while (1) {
15         Spin(1);
16         *p = *p + 1;
17         printf("(d) p: %d\n", getpid(), *p); // a4
18     }
19     return 0;
20 }

```

Figure 2.3: A Program that Accesses Memory (`mem.c`)

## 2.2 Virtualizing Memory

Now let's consider memory. The model of **physical memory** presented by modern machines is very simple. Memory is just an array of bytes; to **read** memory, one must specify an **address** to be able to access the data stored there; to **write** (or **update**) memory, one must also specify the data to be written to the given address.

Memory is accessed all the time when a program is running. A program keeps all of its data structures in memory, and accesses them through various instructions, like loads and stores or other explicit instructions that access memory in doing their work. Don't forget that each instruction of the program is in memory too; thus memory is accessed on each instruction fetch.

Let's take a look at a program (in Figure 2.3) that allocates some memory by calling `malloc()`. The output of this program can be found here:

```

prompt> ./mem
(2134) memory address of p: 00200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C

```

The program does a couple of things. First, it allocates some memory (line a1). Then, it prints out the address of the memory (a2), and then puts the number zero into the first slot of the newly allocated memory (a3). Finally, it loops, delaying for a second and incrementing the value stored at the address held in `p`. With every print statement, it also prints out what is called the process identifier (the PID) of the running program. This PID is unique per running process.

```

prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) memory address of p: 00200000
(24114) memory address of p: 00200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...

```

Figure 2.4: Running The Memory Program Multiple Times

Again, this first result is not too interesting. The newly allocated memory is at address 00200000. As the program runs, it slowly updates the value and prints out the result.

Now, we again run multiple instances of this same program to see what happens (Figure 2.4). We see from the example that each running program has allocated memory at the same address (00200000), and yet each seems to be updating the value at 00200000 independently! It is as if each running program has its own private memory, instead of sharing the same physical memory with other running programs<sup>5</sup>.

Indeed, that is exactly what is happening here as the OS is **virtualizing memory**. Each process accesses its own private **virtual address space** (sometimes just called its **address space**), which the OS somehow maps onto the physical memory of the machine. A memory reference within one running program does not affect the address space of other processes (or the OS itself); as far as the running program is concerned, it has physical memory all to itself. The reality, however, is that physical memory is a shared resource, managed by the operating system. Exactly how all of this is accomplished is also the subject of the first part of this book, on the topic of **virtualization**.

## 2.3 Concurrency

Another main theme of this book is **concurrency**. We use this conceptual term to refer to a host of problems that arise, and must be addressed, when working on many things at once (i.e., concurrently) in the same program. The problems of concurrency arose first within the operating system itself; as you can see in the examples above on virtualization, the OS is juggling many things at once, first running one process, then another, and so forth. As it turns out, doing so leads to some deep and interesting problems.

<sup>5</sup>For this example to work, you need to make sure address-space randomization is disabled; randomization, as it turns out, can be a good defense against certain kinds of security flaws. Read more about it on your own, especially if you want to learn how to break into computer systems via stack-smashing attacks. Not that we would recommend such a thing...

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "common.h"
4
5  volatile int counter = 0;
6  int loops;
7
8  void *worker(void *arg) {
9      int i;
10     for (i = 0; i < loops; i++) {
11         counter++;
12     }
13     return NULL;
14 }
15
16 int
17 main(int argc, char *argv[])
18 {
19     if (argc != 2) {
20         fprintf(stderr, "usage: threads <value>\n");
21         exit(1);
22     }
23     loops = atoi(argv[1]);
24     pthread_t p1, p2;
25     printf("Initial value : %d\n", counter);
26
27     Pthread_create(&p1, NULL, worker, NULL);
28     Pthread_create(&p2, NULL, worker, NULL);
29     Pthread_join(p1, NULL);
30     Pthread_join(p2, NULL);
31     printf("Final value   : %d\n", counter);
32     return 0;
33 }

```

Figure 2.5: A Multi-threaded Program (`threads.c`)

Unfortunately, the problems of concurrency are no longer limited just to the OS itself. Indeed, modern **multi-threaded** programs exhibit the same problems. Let us demonstrate with an example of a **multi-threaded** program (Figure 2.5).

Although you might not understand this example fully at the moment (and we'll learn a lot more about it in later chapters, in the section of the book on concurrency), the basic idea is simple. The main program creates two **threads** using `Pthread_create()`<sup>6</sup>. You can think of a thread as a function running within the same memory space as other functions, with more than one of them active at a time. In this example, each thread starts running in a routine called `worker()`, in which it simply increments a counter in a loop for `loops` number of times.

Below is a transcript of what happens when we run this program with the input value for the variable `loops` set to 1000. The value of `loops`

<sup>6</sup>The actual call should be to lower-case `pthread_create()`; the upper-case version is our own wrapper that calls `pthread_create()` and makes sure that the return code indicates that the call succeeded. See the code for details.

## THE CRUX OF THE PROBLEM:

## HOW TO BUILD CORRECT CONCURRENT PROGRAMS

When there are many concurrently executing threads within the same memory space, how can we build a correctly working program? What primitives are needed from the OS? What mechanisms should be provided by the hardware? How can we use them to solve the problems of concurrency?

determines how many times each of the two workers will increment the shared counter in a loop. When the program is run with the value of `loops` set to 1000, what do you expect the final value of `counter` to be?

```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value   : 2000
```

As you probably guessed, when the two threads are finished, the final value of the counter is 2000, as each thread incremented the counter 1000 times. Indeed, when the input value of `loops` is set to  $N$ , we would expect the final output of the program to be  $2N$ . But life is not so simple, as it turns out. Let's run the same program, but with higher values for `loops`, and see what happens:

```
prompt> ./thread 100000
Initial value : 0
Final value   : 143012    // huh??
prompt> ./thread 100000
Initial value : 0
Final value   : 137298    // what the??
```

In this run, when we gave an input value of 100,000, instead of getting a final value of 200,000, we instead first get 143,012. Then, when we run the program a second time, we not only again get the *wrong* value, but also a *different* value than the last time. In fact, if you run the program over and over with high values of `loops`, you may find that sometimes you even get the right answer! So why is this happening?

As it turns out, the reason for these odd and unusual outcomes relate to how instructions are executed, which is one at a time. Unfortunately, a key part of the program above, where the shared counter is incremented, takes three instructions: one to load the value of the counter from memory into a register, one to increment it, and one to store it back into memory. Because these three instructions do not execute **atomically** (all at once), strange things can happen. It is this problem of **concurrency** that we will address in great detail in the second part of this book.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <assert.h>
4  #include <fcntl.h>
5  #include <sys/types.h>
6
7  int
8  main(int argc, char *argv[])
9  {
10     int fd = open("/tmp/file", O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU);
11     assert(fd > -1);
12     int rc = write(fd, "hello world\n", 13);
13     assert(rc == 13);
14     close(fd);
15     return 0;
16 }
```

Figure 2.6: A Program That Does I/O (`io.c`)

## 2.4 Persistence

The third major theme of the course is **persistence**. In system memory, data can be easily lost, as devices such as DRAM store values in a **volatile** manner; when power goes away or the system crashes, any data in memory is lost. Thus, we need hardware and software to be able to store data **persistently**; such storage is thus critical to any system as users care a great deal about their data.

The hardware comes in the form of some kind of **input/output** or **I/O** device; in modern systems, a **hard drive** is a common repository for long-lived information, although **solid-state drives (SSDs)** are making headway in this arena as well.

The software in the operating system that usually manages the disk is called the **file system**; it is thus responsible for storing any **files** the user creates in a reliable and efficient manner on the disks of the system.

Unlike the abstractions provided by the OS for the CPU and memory, the OS does not create a private, virtualized disk for each application. Rather, it is assumed that often times, users will want to **share** information that is in files. For example, when writing a C program, you might first use an editor (e.g., Emacs<sup>7</sup>) to create and edit the C file (`emacs -nw main.c`). Once done, you might use the compiler to turn the source code into an executable (e.g., `gcc -o main main.c`). When you're finished, you might run the new executable (e.g., `./main`). Thus, you can see how files are shared across different processes. First, Emacs creates a file that serves as input to the compiler; the compiler uses that input file to create a new executable file (in many steps — take a compiler course for details); finally, the new executable is then run. And thus a new program is born!

To understand this better, let's look at some code. Figure 2.6 presents code to create a file (`/tmp/file`) that contains the string "hello world".

---

<sup>7</sup>You should be using Emacs. If you are using vi, there is probably something wrong with you. If you are using something that is not a real code editor, that is even worse.



THE CRUX OF THE PROBLEM:  
HOW TO STORE DATA PERSISTENTLY

The file system is the part of the OS in charge of managing persistent data. What techniques are needed to do so correctly? What mechanisms and policies are required to do so with high performance? How is reliability achieved, in the face of failures in hardware and software?

To accomplish this task, the program makes three calls into the operating system. The first, a call to `open()`, opens the file and creates it; the second, `write()`, writes some data to the file; the third, `close()`, simply closes the file thus indicating the program won't be writing any more data to it. These **system calls** are routed to the part of the operating system called the **file system**, which then handles the requests and returns some kind of error code to the user.

You might be wondering what the OS does in order to actually write to disk. We would show you but you'd have to promise to close your eyes first; it is that unpleasant. The file system has to do a fair bit of work: first figuring out where on disk this new data will reside, and then keeping track of it in various structures the file system maintains. Doing so requires issuing I/O requests to the underlying storage device, to either read existing structures or update (write) them. As anyone who has written a **device driver**<sup>8</sup> knows, getting a device to do something on your behalf is an intricate and detailed process. It requires a deep knowledge of the low-level device interface and its exact semantics. Fortunately, the OS provides a standard and simple way to access devices through its system calls. Thus, the OS is sometimes seen as a **standard library**.

Of course, there are many more details in how devices are accessed, and how file systems manage data persistently atop said devices. For performance reasons, most file systems first delay such writes for a while, hoping to batch them into larger groups. To handle the problems of system crashes during writes, most file systems incorporate some kind of intricate write protocol, such as **journaling** or **copy-on-write**, carefully ordering writes to disk to ensure that if a failure occurs during the write sequence, the system can recover to reasonable state afterwards. To make different common operations efficient, file systems employ many different data structures and access methods, from simple lists to complex b-trees. If all of this doesn't make sense yet, good! We'll be talking about all of this quite a bit more in the third part of this book on **persistence**, where we'll discuss devices and I/O in general, and then disks, RAIDs, and file systems in great detail.

---

<sup>8</sup>A device driver is some code in the operating system that knows how to deal with a specific device. We will talk more about devices and device drivers later.

## 2.5 Design Goals

So now you have some idea of what an OS actually does: it takes physical **resources**, such as a CPU, memory, or disk, and **virtualizes** them. It handles tough and tricky issues related to **concurrency**. And it stores files **persistently**, thus making them safe over the long-term. Given that we want to build such a system, we want to have some goals in mind to help focus our design and implementation and make trade-offs as necessary; finding the right set of trade-offs is a key to building systems.

One of the most basic goals is to build up some **abstractions** in order to make the system convenient and easy to use. Abstractions are fundamental to everything we do in computer science. Abstraction makes it possible to write a large program by dividing it into small and understandable pieces, to write such a program in a high-level language like C<sup>9</sup> without thinking about assembly, to write code in assembly without thinking about logic gates, and to build a processor out of gates without thinking too much about transistors. Abstraction is so fundamental that sometimes we forget its importance, but we won't here; thus, in each section, we'll discuss some of the major abstractions that have developed over time, giving you a way to think about pieces of the OS.

One goal in designing and implementing an operating system is to provide high **performance**; another way to say this is our goal is to **minimize the overheads** of the OS. Virtualization and making the system easy to use are well worth it, but not at any cost; thus, we must strive to provide virtualization and other OS features without excessive overheads. These overheads arise in a number of forms: extra time (more instructions) and extra space (in memory or on disk). We'll seek solutions that minimize one or the other or both, if possible. Perfection, however, is not always attainable, something we will learn to notice and (where appropriate) tolerate.

Another goal will be to provide **protection** between applications, as well as between the OS and applications. Because we wish to allow many programs to run at the same time, we want to make sure that the malicious or accidental bad behavior of one does not harm others; we certainly don't want an application to be able to harm the OS itself (as that would affect *all* programs running on the system). Protection is at the heart of one of the main principles underlying an operating system, which is that of **isolation**; isolating processes from one another is the key to protection and thus underlies much of what an OS must do.

The operating system must also run non-stop; when it fails, *all* applications running on the system fail as well. Because of this dependence, operating systems often strive to provide a high degree of **reliability**. As operating systems grow evermore complex (sometimes containing millions of lines of code), building a reliable operating system is quite a chal-

---

<sup>9</sup>Some of you might object to calling C a high-level language. Remember this is an OS course, though, where we're simply happy not to have to code in assembly all the time!

lenge — and indeed, much of the on-going research in the field (including some of our own work [BS+09, SS+10]) focuses on this exact problem.

Other goals make sense: **energy-efficiency** is important in our increasingly green world; **security** (an extension of protection, really) against malicious applications is critical, especially in these highly-networked times; **mobility** is increasingly important as OSes are run on smaller and smaller devices. Depending on how the system is used, the OS will have different goals and thus likely be implemented in at least slightly different ways. However, as we will see, many of the principles we will present on how to build an OS are useful on a range of different devices.

## 2.6 Some History

Before closing this introduction, let us present a brief history of how operating systems developed. Like any system built by humans, good ideas accumulated in operating systems over time, as engineers learned what was important in their design. Here, we discuss a few major developments. For a richer treatment, see Brinch Hansen’s excellent history of operating systems [BH00].

### Early Operating Systems: Just Libraries

In the beginning, the operating system didn’t do too much. Basically, it was just a set of libraries of commonly-used functions; for example, instead of having each programmer of the system write low-level I/O handling code, the “OS” would provide such APIs, and thus make life easier for the developer.

Usually, on these old mainframe systems, one program ran at a time, as controlled by a human operator. Much of what you think a modern OS would do (e.g., deciding what order to run jobs in) was performed by this operator. If you were a smart developer, you would be nice to this operator, so that they might move your job to the front of the queue.

This mode of computing was known as **batch** processing, as a number of jobs were set up and then run in a “batch” by the operator. Computers, as of that point, were not used in an interactive manner, because of cost: it was simply too expensive to let a user sit in front of the computer and use it, as most of the time it would just sit idle then, costing the facility hundreds of thousands of dollars per hour [BH00].

### Beyond Libraries: Protection

In moving beyond being a simple library of commonly-used services, operating systems took on a more central role in managing machines. One important aspect of this was the realization that code run on behalf of the OS was special; it had control of devices and thus should be treated differently than normal application code. Why is this? Well, imagine if you

allowed any application to read from anywhere on the disk; the notion of privacy goes out the window, as any program could read any file. Thus, implementing a **file system** (to manage your files) as a library makes little sense. Instead, something else was needed.

Thus, the idea of a **system call** was invented, pioneered by the Atlas computing system [K+61,L78]. Instead of providing OS routines as a library (where you just make a **procedure call** to access them), the idea here was to add a special pair of hardware instructions and hardware state to make the transition into the OS a more formal, controlled process.

The key difference between a system call and a procedure call is that a system call transfers control (i.e., jumps) into the OS while simultaneously raising the **hardware privilege level**. User applications run in what is referred to as **user mode** which means the hardware restricts what applications can do; for example, an application running in user mode can't typically initiate an I/O request to the disk, access any physical memory page, or send a packet on the network. When a system call is initiated (usually through a special hardware instruction called a **trap**), the hardware transfers control to a pre-specified **trap handler** (that the OS set up previously) and simultaneously raises the privilege level to **kernel mode**. In kernel mode, the OS has full access to the hardware of the system and thus can do things like initiate an I/O request or make more memory available to a program. When the OS is done servicing the request, it passes control back to the user via a special **return-from-trap** instruction, which reverts to user mode while simultaneously passing control back to where the application left off.

## The Era of Multiprogramming

Where operating systems really took off was in the era of computing beyond the mainframe, that of the **minicomputer**. Classic machines like the PDP family from Digital Equipment made computers hugely more affordable; thus, instead of having one mainframe per large organization, now a smaller collection of people within an organization could likely have their own computer. Not surprisingly, one of the major impacts of this drop in cost was an increase in developer activity; more smart people got their hands on computers and thus made computer systems do more interesting and beautiful things.

In particular, **multiprogramming** became commonplace due to the desire to make better use of machine resources. Instead of just running one job at a time, the OS would load a number of jobs into memory and switch rapidly between them, thus improving CPU utilization. This switching was particularly important because I/O devices were slow; having a program wait on the CPU while its I/O was being serviced was a waste of CPU time. Instead, why not switch to another job and run it for a while?

The desire to support multiprogramming and overlap in the presence of I/O and interrupts forced innovation in the conceptual development of operating systems along a number of directions. Issues such as **memory**

**protection** became important; we wouldn't want one program to be able to access the memory of another program. Understanding how to deal with the **concurrency** issues introduced by multiprogramming was also critical; making sure the OS was behaving correctly despite the presence of interrupts is a great challenge. We will study these issues and related topics later in the book.

One of the major practical advances of the time was the introduction of the UNIX operating system, primarily thanks to Ken Thompson (and Dennis Ritchie) at Bell Labs (yes, the phone company). UNIX took many good ideas from different operating systems (particularly from Multics [O72], and some from systems like TENEX [B+72] and the Berkeley Time-Sharing System [S+68]), but made them simpler and easier to use. Soon this team was shipping tapes containing UNIX source code to people around the world, many of whom then got involved and added to the system themselves; see the **Aside** (next page) for more detail<sup>10</sup>.

## The Modern Era

Beyond the minicomputer came a new type of machine, cheaper, faster, and for the masses: the **personal computer**, or **PC** as we call it today. Led by Apple's early machines (e.g., the Apple II) and the IBM PC, this new breed of machine would soon become the dominant force in computing, as their low-cost enabled one machine per desktop instead of a shared minicomputer per workgroup.

Unfortunately, for operating systems, the PC at first represented a great leap backwards, as early systems forgot (or never knew of) the lessons learned in the era of minicomputers. For example, early operating systems such as **DOS** (the **Disk Operating System**, from **Microsoft**) didn't think memory protection was important; thus, a malicious (or perhaps just a poorly-programmed) application could scribble all over memory. The first generations of the **Mac OS** (v9 and earlier) took a cooperative approach to job scheduling; thus, a thread that accidentally got stuck in an infinite loop could take over the entire system, forcing a reboot. The painful list of OS features missing in this generation of systems is long, too long for a full discussion here.

Fortunately, after some years of suffering, the old features of minicomputer operating systems started to find their way onto the desktop. For example, Mac OS X has UNIX at its core, including all of the features one would expect from such a mature system. Windows has similarly adopted many of the great ideas in computing history, starting in particular with Windows NT, a great leap forward in Microsoft OS technology. Even today's cell phones run operating systems (such as Linux) that are much more like what a minicomputer ran in the 1970s than what a PC

---

<sup>10</sup>We'll use asides and other related text boxes to call attention to various items that don't quite fit the main flow of the text. Sometimes, we'll even use them just to make a joke, because why not have a little fun along the way? Yes, many of the jokes are bad.

**ASIDE: THE IMPORTANCE OF UNIX**

It is difficult to overstate the importance of UNIX in the history of operating systems. Influenced by earlier systems (in particular, the famous **Multics** system from MIT), UNIX brought together many great ideas and made a system that was both simple and powerful.

Underlying the original “Bell Labs” UNIX was the unifying principle of building small powerful programs that could be connected together to form larger workflows. The **shell**, where you type commands, provided primitives such as **pipes** to enable such meta-level programming, and thus it became easy to string together programs to accomplish a bigger task. For example, to find lines of a text file that have the word “foo” in them, and then to count how many such lines exist, you would type: `grep foo file.txt|wc -l`, thus using the `grep` and `wc` (word count) programs to achieve your task.

The UNIX environment was friendly for programmers and developers alike, also providing a compiler for the new **C programming language**. Making it easy for programmers to write their own programs, as well as share them, made UNIX enormously popular. And it probably helped a lot that the authors gave out copies for free to anyone who asked, an early form of **open-source software**.

Also of critical importance was the accessibility and readability of the code. Having a beautiful, small kernel written in C invited others to play with the kernel, adding new and cool features. For example, an enterprising group at Berkeley, led by **Bill Joy**, made a wonderful distribution (the **Berkeley Systems Distribution**, or **BSD**) which had some advanced virtual memory, file system, and networking subsystems. Joy later co-founded **Sun Microsystems**.

Unfortunately, the spread of UNIX was slowed a bit as companies tried to assert ownership and profit from it, an unfortunate (but common) result of lawyers getting involved. Many companies had their own variants: **SunOS** from Sun Microsystems, **AIX** from IBM, **HPUX** (a.k.a. “H-Pucks”) from HP, and **IRIX** from SGI. The legal wrangling among AT&T/Bell Labs and these other players cast a dark cloud over UNIX, and many wondered if it would survive, especially as Windows was introduced and took over much of the PC market...

ran in the 1980s (thank goodness); it is good to see that the good ideas developed in the heyday of OS development have found their way into the modern world. Even better is that these ideas continue to develop, providing more features and making modern systems even better for users and applications.

#### ASIDE: AND THEN CAME LINUX

Fortunately for UNIX, a young Finnish hacker named **Linus Torvalds** decided to write his own version of UNIX which borrowed heavily on the principles and ideas behind the original system, but not from the code base, thus avoiding issues of legality. He enlisted help from many others around the world, and soon **Linux** was born (as well as the modern open-source software movement).

As the internet era came into place, most companies (such as Google, Amazon, Facebook, and others) chose to run Linux, as it was free and could be readily modified to suit their needs; indeed, it is hard to imagine the success of these new companies had such a system not existed. As smart phones became a dominant user-facing platform, Linux found a stronghold there too (via Android), for many of the same reasons. And Steve Jobs took his UNIX-based **NeXTStep** operating environment with him to Apple, thus making UNIX popular on desktops (though many users of Apple technology are probably not even aware of this fact). And thus UNIX lives on, more important today than ever before. The computing gods, if you believe in them, should be thanked for this wonderful outcome.

## 2.7 Summary

Thus, we have an introduction to the OS. Today's operating systems make systems relatively easy to use, and virtually all operating systems you use today have been influenced by the developments we will discuss throughout the book.

Unfortunately, due to time constraints, there are a number of parts of the OS we won't cover in the book. For example, there is a lot of **networking** code in the operating system; we leave it to you to take the networking class to learn more about that. Similarly, **graphics** devices are particularly important; take the graphics course to expand your knowledge in that direction. Finally, some operating system books talk a great deal about **security**; we will do so in the sense that the OS must provide protection between running programs and give users the ability to protect their files, but we won't delve into deeper security issues that one might find in a security course.

However, there are many important topics that we will cover, including the basics of virtualization of the CPU and memory, concurrency, and persistence via devices and file systems. Don't worry! While there is a lot of ground to cover, most of it is quite cool, and at the end of the road, you'll have a new appreciation for how computer systems really work. Now get to work!

## References

- [BS+09] “Tolerating File-System Mistakes with EnvyFS”  
Lakshmi N. Bairavasundaram, Swaminathan Sundararaman, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau  
USENIX ’09, San Diego, CA, June 2009  
*A fun paper about using multiple file systems at once to tolerate a mistake in any one of them.*
- [BH00] “The Evolution of Operating Systems”  
P. Brinch Hansen  
In *Classic Operating Systems: From Batch Processing to Distributed Systems*  
Springer-Verlag, New York, 2000  
*This essay provides an intro to a wonderful collection of papers about historically significant systems.*
- [B+72] “TENEX, A Paged Time Sharing System for the PDP-10”  
Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, Raymond S. Tomlinson  
CACM, Volume 15, Number 3, March 1972  
*TENEX has much of the machinery found in modern operating systems; read more about it to see how much innovation was already in place in the early 1970’s.*
- [B75] “The Mythical Man-Month”  
Fred Brooks  
Addison-Wesley, 1975  
*A classic text on software engineering; well worth the read.*
- [BOH10] “Computer Systems: A Programmer’s Perspective”  
Randal E. Bryant and David R. O’Hallaron  
Addison-Wesley, 2010  
*Another great intro to how computer systems work. Has a little bit of overlap with this book — so if you’d like, you can skip the last few chapters of that book, or simply read them to get a different perspective on some of the same material. After all, one good way to build up your own knowledge is to hear as many other perspectives as possible, and then develop your own opinion and thoughts on the matter. You know, by thinking!*
- [K+61] “One-Level Storage System”  
T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner  
IRE Transactions on Electronic Computers, April 1962  
*The Atlas pioneered much of what you see in modern systems. However, this paper is not the best read. If you were to only read one, you might try the historical perspective below [L78].*
- [L78] “The Manchester Mark I and Atlas: A Historical Perspective”  
S. H. Lavington  
Communications of the ACM archive  
Volume 21, Issue 1 (January 1978), pages 4-12  
*A nice piece of history on the early development of computer systems and the pioneering efforts of the Atlas. Of course, one could go back and read the Atlas papers themselves, but this paper provides a great overview and adds some historical perspective.*
- [O72] “The Multics System: An Examination of its Structure”  
Elliott Organick, 1972  
*A great overview of Multics. So many good ideas, and yet it was an over-designed system, shooting for too much, and thus never really worked as expected. A classic example of what Fred Brooks would call the “second-system effect” [B75].*



[PP03] “Introduction to Computing Systems:  
From Bits and Gates to C and Beyond”

Yale N. Patt and Sanjay J. Patel

McGraw-Hill, 2003

*One of our favorite intro to computing systems books. Starts at transistors and gets you all the way up to C; the early material is particularly great.*

[RT74] “The UNIX Time-Sharing System”

Dennis M. Ritchie and Ken Thompson

CACM, Volume 17, Number 7, July 1974, pages 365-375

*A great summary of UNIX written as it was taking over the world of computing, by the people who wrote it.*

[S68] “SDS 940 Time-Sharing System”

Scientific Data Systems Inc.

TECHNICAL MANUAL, SDS 90 11168 August 1968

Available: <http://goo.gl/EN0Zrm>

*Yes, a technical manual was the best we could find. But it is fascinating to read these old system documents, and see how much was already in place in the late 1960's. One of the minds behind the Berkeley Time-Sharing System (which eventually became the SDS system) was Butler Lampson, who later won a Turing award for his contributions in systems.*

[SS+10] “Membrane: Operating System Support for Restartable File Systems”

Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale,

Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Michael M. Swift

FAST '10, San Jose, CA, February 2010

*The great thing about writing your own class notes: you can advertise your own research. But this paper is actually pretty neat — when a file system hits a bug and crashes, Membrane auto-magically restarts it, all without applications or the rest of the system being affected.*

**Part I**

**Virtualization**



## A Dialogue on Virtualization

**Professor:** *And thus we reach the first of our three pieces on operating systems: virtualization.*

**Student:** *But what is virtualization, oh noble professor?*

**Professor:** *Imagine we have a peach.*

**Student:** *A peach? (incredulous)*

**Professor:** *Yes, a peach. Let us call that the **physical** peach. But we have many eaters who would like to eat this peach. What we would like to present to each eater is their own peach, so that they can be happy. We call the peach we give eaters **virtual** peaches; we somehow create many of these virtual peaches out of the one physical peach. And the important thing: in this illusion, it looks to each eater like they have a physical peach, but in reality they don't.*

**Student:** *So you are sharing the peach, but you don't even know it?*

**Professor:** *Right! Exactly.*

**Student:** *But there's only one peach.*

**Professor:** *Yes. And...?*

**Student:** *Well, if I was sharing a peach with somebody else, I think I would notice.*

**Professor:** *Ah yes! Good point. But that is the thing with many eaters; most of the time they are napping or doing something else, and thus, you can snatch that peach away and give it to someone else for a while. And thus we create the illusion of many virtual peaches, one peach for each person!*

**Student:** *Sounds like a bad campaign slogan. You are talking about computers, right Professor?*

**Professor:** *Ah, young grasshopper, you wish to have a more concrete example. Good idea! Let us take the most basic of resources, the CPU. Assume there is one physical CPU in a system (though now there are often two or four or more). What virtualization does is take that single CPU and make it look like many virtual CPUs to the applications running on the system. Thus, while each application*

*thinks it has its own CPU to use, there is really only one. And thus the OS has created a beautiful illusion: it has virtualized the CPU.*

**Student:** *Wow! That sounds like magic. Tell me more! How does that work?*

**Professor:** *In time, young student, in good time. Sounds like you are ready to begin.*

**Student:** *I am! Well, sort of. I must admit, I'm a little worried you are going to start talking about peaches again.*

**Professor:** *Don't worry too much; I don't even like peaches. And thus we begin...*

## The Abstraction: The Process

In this chapter, we discuss one of the most fundamental abstractions that the OS provides to users: the **process**. The definition of a process, informally, is quite simple: it is a **running program** [V+65,BH70]. The program itself is a lifeless thing: it just sits there on the disk, a bunch of instructions (and maybe some static data), waiting to spring into action. It is the operating system that takes these bytes and gets them running, transforming the program into something useful.

It turns out that one often wants to run more than one program at once; for example, consider your desktop or laptop where you might like to run a web browser, mail program, a game, a music player, and so forth. In fact, a typical system may be seemingly running tens or even hundreds of processes at the same time. Doing so makes the system easy to use, as one never need be concerned with whether a CPU is available; one simply runs programs. Hence our challenge:

### THE CRUX OF THE PROBLEM:

#### HOW TO PROVIDE THE ILLUSION OF MANY CPUS?

Although there are only a few physical CPUs available, how can the OS provide the illusion of a nearly-endless supply of said CPUs?

The OS creates this illusion by **virtualizing** the CPU. By running one process, then stopping it and running another, and so forth, the OS can promote the illusion that many virtual CPUs exist when in fact there is only one physical CPU (or a few). This basic technique, known as **time sharing** of the CPU, allows users to run as many concurrent processes as they would like; the potential cost is performance, as each will run more slowly if the CPU(s) must be shared.

To implement virtualization of the CPU, and to implement it well, the OS will need both some low-level machinery as well as some high-level intelligence. We call the low-level machinery **mechanisms**; mechanisms are low-level methods or protocols that implement a needed piece

TIP: USE TIME SHARING (AND SPACE SHARING)

**Time sharing** is one of the most basic techniques used by an OS to share a resource. By allowing the resource to be used for a little while by one entity, and then a little while by another, and so forth, the resource in question (e.g., the CPU, or a network link) can be shared by many. The natural counterpart of time sharing is **space sharing**, where a resource is divided (in space) among those who wish to use it. For example, disk space is naturally a space-shared resource, as once a block is assigned to a file, it is not likely to be assigned to another file until the user deletes it.

of functionality. For example, we'll learn later how to implement a **context switch**, which gives the OS the ability to stop running one program and start running another on a given CPU; this **time-sharing** mechanism is employed by all modern OSes.

On top of these mechanisms resides some of the intelligence in the OS, in the form of **policies**. Policies are algorithms for making some kind of decision within the OS. For example, given a number of possible programs to run on a CPU, which program should the OS run? A **scheduling policy** in the OS will make this decision, likely using historical information (e.g., which program has run more over the last minute?), workload knowledge (e.g., what types of programs are run), and performance metrics (e.g., is the system optimizing for interactive performance, or throughput?) to make its decision.

## 4.1 The Abstraction: A Process

The abstraction provided by the OS of a running program is something we will call a **process**. As we said above, a process is simply a running program; at any instant in time, we can summarize a process by taking an inventory of the different pieces of the system it accesses or affects during the course of its execution.

To understand what constitutes a process, we thus have to understand its **machine state**: what a program can read or update when it is running. At any given time, what parts of the machine are important to the execution of this program?

One obvious component of machine state that comprises a process is its *memory*. Instructions lie in memory; the data that the running program reads and writes sits in memory as well. Thus the memory that the process can address (called its **address space**) is part of the process.

Also part of the process's machine state are *registers*; many instructions explicitly read or update registers and thus clearly they are important to the execution of the process.

Note that there are some particularly special registers that form part of this machine state. For example, the **program counter (PC)** (sometimes called the **instruction pointer** or **IP**) tells us which instruction of the pro-

**TIP: SEPARATE POLICY AND MECHANISM**

In many operating systems, a common design paradigm is to separate high-level policies from their low-level mechanisms [L+75]. You can think of the mechanism as providing the answer to a *how* question about a system; for example, *how* does an operating system perform a context switch? The policy provides the answer to a *which* question; for example, *which* process should the operating system run right now? Separating the two allows one easily to change policies without having to rethink the mechanism and is thus a form of **modularity**, a general software design principle.

gram is currently being executed; similarly a **stack pointer** and associated **frame pointer** are used to manage the stack for function parameters, local variables, and return addresses.

Finally, programs often access persistent storage devices too. Such *I/O information* might include a list of the files the process currently has open.

## 4.2 Process API

Though we defer discussion of a real process API until a subsequent chapter, here we first give some idea of what must be included in any interface of an operating system. These APIs, in some form, are available on any modern operating system.

- **Create:** An operating system must include some method to create new processes. When you type a command into the shell, or double-click on an application icon, the OS is invoked to create a new process to run the program you have indicated.
- **Destroy:** As there is an interface for process creation, systems also provide an interface to destroy processes forcefully. Of course, many processes will run and just exit by themselves when complete; when they don't, however, the user may wish to kill them, and thus an interface to halt a runaway process is quite useful.
- **Wait:** Sometimes it is useful to wait for a process to stop running; thus some kind of waiting interface is often provided.
- **Miscellaneous Control:** Other than killing or waiting for a process, there are sometimes other controls that are possible. For example, most operating systems provide some kind of method to suspend a process (stop it from running for a while) and then resume it (continue it running).
- **Status:** There are usually interfaces to get some status information about a process as well, such as how long it has run for, or what state it is in.



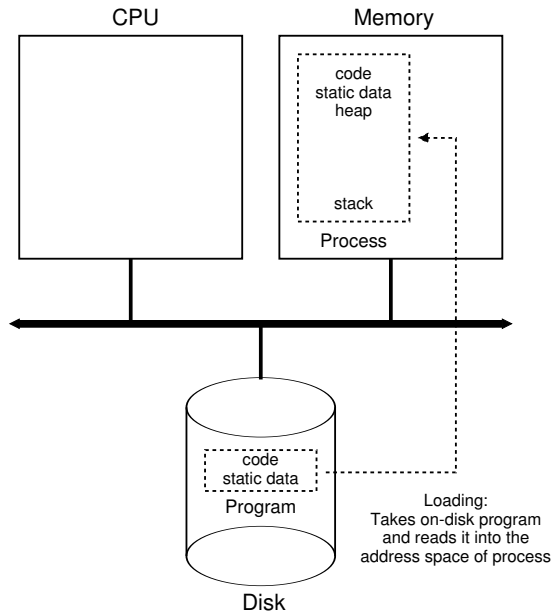


Figure 4.1: **Loading: From Program To Process**

### 4.3 Process Creation: A Little More Detail

One mystery that we should unmask a bit is how programs are transformed into processes. Specifically, how does the OS get a program up and running? How does process creation actually work?

The first thing that the OS must do to run a program is to **load** its code and any static data (e.g., initialized variables) into memory, into the address space of the process. Programs initially reside on **disk** (or, in some modern systems, **flash-based SSDs**) in some kind of **executable format**; thus, the process of loading a program and static data into memory requires the OS to read those bytes from disk and place them in memory somewhere (as shown in Figure 4.1).

In early (or simple) operating systems, the loading process is done **eagerly**, i.e., all at once before running the program; modern OSes perform the process **lazily**, i.e., by loading pieces of code or data only as they are needed during program execution. To truly understand how lazy loading of pieces of code and data works, you'll have to understand more about the machinery of **paging** and **swapping**, topics we'll cover in the future when we discuss the virtualization of memory. For now, just remember that before running anything, the OS clearly must do some work to get the important program bits from disk into memory.

Once the code and static data are loaded into memory, there are a few other things the OS needs to do before running the process. Some memory must be allocated for the program's **run-time stack** (or just **stack**). As you should likely already know, C programs use the stack for local variables, function parameters, and return addresses; the OS allocates this memory and gives it to the process. The OS will also likely initialize the stack with arguments; specifically, it will fill in the parameters to the `main()` function, i.e., `argc` and the `argv` array.

The OS may also allocate some memory for the program's **heap**. In C programs, the heap is used for explicitly requested dynamically-allocated data; programs request such space by calling `malloc()` and free it explicitly by calling `free()`. The heap is needed for data structures such as linked lists, hash tables, trees, and other interesting data structures. The heap will be small at first; as the program runs, and requests more memory via the `malloc()` library API, the OS may get involved and allocate more memory to the process to help satisfy such calls.

The OS will also do some other initialization tasks, particularly as related to input/output (I/O). For example, in UNIX systems, each process by default has three open **file descriptors**, for standard input, output, and error; these descriptors let programs easily read input from the terminal as well as print output to the screen. We'll learn more about I/O, file descriptors, and the like in the third part of the book on **persistence**.

By loading the code and static data into memory, by creating and initializing a stack, and by doing other work as related to I/O setup, the OS has now (finally) set the stage for program execution. It thus has one last task: to start the program running at the entry point, namely `main()`. By jumping to the `main()` routine (through a specialized mechanism that we will discuss next chapter), the OS transfers control of the CPU to the newly-created process, and thus the program begins its execution.

## 4.4 Process States

Now that we have some idea of what a process is (though we will continue to refine this notion), and (roughly) how it is created, let us talk about the different **states** a process can be in at a given time. The notion that a process can be in one of these states arose in early computer systems [DV66,V+65]. In a simplified view, a process can be in one of three states:

- **Running:** In the running state, a process is running on a processor. This means it is executing instructions.
- **Ready:** In the ready state, a process is ready to run but for some reason the OS has chosen not to run it at this given moment.
- **Blocked:** In the blocked state, a process has performed some kind of operation that makes it not ready to run until some other event takes place. A common example: when a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.

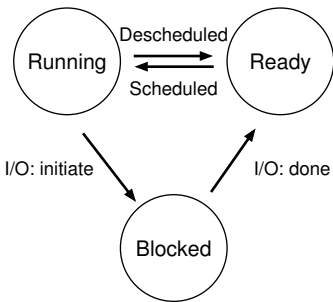


Figure 4.2: Process: State Transitions

If we were to map these states to a graph, we would arrive at the diagram in Figure 4.2. As you can see in the diagram, a process can be moved between the ready and running states at the discretion of the OS. Being moved from ready to running means the process has been **scheduled**; being moved from running to ready means the process has been **descheduled**. Once a process has become blocked (e.g., by initiating an I/O operation), the OS will keep it as such until some event occurs (e.g., I/O completion); at that point, the process moves to the ready state again (and potentially immediately to running again, if the OS so decides).

Let’s look at an example of how two processes might transition through some of these states. First, imagine two processes running, each of which only use the CPU (they do no I/O). In this case, a trace of the state of each process might look like this (Figure 4.3).

Time	Process <sub>0</sub>	Process <sub>1</sub>	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process <sub>0</sub> now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	Process <sub>1</sub> now done

Figure 4.3: Tracing Process State: CPU Only

In this next example, the first process issues an I/O after running for some time. At that point, the process is blocked, giving the other process a chance to run. Figure 4.4 shows a trace of this scenario.

More specifically, Process<sub>0</sub> initiates an I/O and becomes blocked waiting for it to complete; processes become blocked, for example, when read-

Time	Process <sub>0</sub>	Process <sub>1</sub>	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process <sub>0</sub> initiates I/O
4	Blocked	Running	Process <sub>0</sub> is blocked,
5	Blocked	Running	so Process <sub>1</sub> runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process <sub>1</sub> now done
9	Running	–	
10	Running	–	Process <sub>0</sub> now done

Figure 4.4: Tracing Process State: CPU and I/O

ing from a disk or waiting for a packet from a network. The OS recognizes Process<sub>0</sub> is not using the CPU and starts running Process<sub>1</sub>. While Process<sub>1</sub> is running, the I/O completes, moving Process<sub>0</sub> back to ready. Finally, Process<sub>1</sub> finishes, and Process<sub>0</sub> runs and then is done.

Note that there are many decisions the OS must make, even in this simple example. First, the system had to decide to run Process<sub>1</sub> while Process<sub>0</sub> issued an I/O; doing so improves resource utilization by keeping the CPU busy. Second, the system decided not to switch back to Process<sub>0</sub> when its I/O completed; it is not clear if this is a good decision or not. What do you think? These types of decisions are made by the OS **scheduler**, a topic we will discuss a few chapters in the future.

## 4.5 Data Structures

The OS is a program, and like any program, it has some key data structures that track various relevant pieces of information. To track the state of each process, for example, the OS likely will keep some kind of **process list** for all processes that are ready, as well as some additional information to track which process is currently running. The OS must also track, in some way, blocked processes; when an I/O event completes, the OS should make sure to wake the correct process and ready it to run again.

Figure 4.5 shows what type of information an OS needs to track about each process in the xv6 kernel [CK+08]. Similar process structures exist in “real” operating systems such as Linux, Mac OS X, or Windows; look them up and see how much more complex they are.

From the figure, you can see a couple of important pieces of information the OS tracks about a process. The **register context** will hold, for a stopped process, the contents of its registers. When a process is stopped, its registers will be saved to this memory location; by restoring these registers (i.e., placing their values back into the actual physical registers), the OS can resume running the process. We’ll learn more about this technique known as a **context switch** in future chapters.

```

// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
    int eip;
    int esp;
    int ebx;
    int ecx;
    int edx;
    int esi;
    int edi;
    int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem;                // Start of process memory
    uint sz;                  // Size of process memory
    char *kstack;             // Bottom of kernel stack
                                // for this process
    enum proc_state state;    // Process state
    int pid;                  // Process ID
    struct proc *parent;      // Parent process
    void *chan;               // If non-zero, sleeping on chan
    int killed;               // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;         // Current directory
    struct context context;    // Switch here to run process
    struct trapframe *tf;     // Trap frame for the
                                // current interrupt
};

```

Figure 4.5: The xv6 Proc Structure

You can also see from the figure that there are some other states a process can be in, beyond running, ready, and blocked. Sometimes a system will have an **initial** state that the process is in when it is being created. Also, a process could be placed in a **final** state where it has exited but has not yet been cleaned up (in UNIX-based systems, this is called the **zombie** state<sup>1</sup>). This final state can be useful as it allows other processes (usually the **parent** that created the process) to examine the return code of the process and see if the just-finished process executed successfully (usually, programs return zero in UNIX-based systems when they have accomplished a task successfully, and non-zero otherwise). When finished, the parent will make one final call (e.g., `wait()`) to wait for the completion of the child, and to also indicate to the OS that it can clean up any relevant data structures that referred to the now-extinct process.

---

<sup>1</sup>Yes, the zombie state. Just like real zombies, these zombies are relatively easy to kill. However, different techniques are usually recommended.

**ASIDE: DATA STRUCTURE — THE PROCESS LIST**

Operating systems are replete with various important **data structures** that we will discuss in these notes. The **process list** is the first such structure. It is one of the simpler ones, but certainly any OS that has the ability to run multiple programs at once will have something akin to this structure in order to keep track of all the running programs in the system. Sometimes people refer to the individual structure that stores information about a process as a **Process Control Block (PCB)**, a fancy way of talking about a C structure that contains information about each process.

## 4.6 Summary

We have introduced the most basic abstraction of the OS: the process. It is quite simply viewed as a running program. With this conceptual view in mind, we will now move on to the nitty-gritty: the low-level mechanisms needed to implement processes, and the higher-level policies required to schedule them in an intelligent way. By combining mechanisms and policies, we will build up our understanding of how an operating system virtualizes the CPU.

## References

[BH70] “The Nucleus of a Multiprogramming System”

Per Brinch Hansen

Communications of the ACM, Volume 13, Number 4, April 1970

*This paper introduces one of the first **microkernels** in operating systems history, called Nucleus. The idea of smaller, more minimal systems is a theme that rears its head repeatedly in OS history; it all began with Brinch Hansen’s work described herein.*

[CK+08] “The xv6 Operating System”

Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich

From: <http://pdos.csail.mit.edu/6.828/2008/index.html>

*The coolest real and little OS in the world. Download and play with it to learn more about the details of how operating systems actually work.*

[DV66] “Programming Semantics for Multiprogrammed Computations”

Jack B. Dennis and Earl C. Van Horn

Communications of the ACM, Volume 9, Number 3, March 1966

*This paper defined many of the early terms and concepts around building multiprogrammed systems.*

[L+75] “Policy/mechanism separation in Hydra”

R. Levin, E. Cohen, W. Corwin, F. Pollack, W. Wulf

SOSP 1975

*An early paper about how to structure operating systems in a research OS known as Hydra. While Hydra never became a mainstream OS, some of its ideas influenced OS designers.*

[V+65] “Structure of the Multics Supervisor”

V.A. Vyssotsky, F. J. Corbato, R. M. Graham

Fall Joint Computer Conference, 1965

*An early paper on Multics, which described many of the basic ideas and terms that we find in modern systems. Some of the vision behind computing as a utility are finally being realized in modern cloud systems.*

## Homework

### ASIDE: SIMULATION HOMEWORKS

Simulation homeworks come in the form of simulators you run to make sure you understand some piece of the material. The simulators are generally python programs that enable you both to *generate* different problems (using different random seeds) as well as to have the program solve the problem for you (with the `-c` flag) so that you can check your answers. Running any simulator with a `-h` or `--help` flag will provide with more information as to all the options the simulator gives you.

The README provided with each simulator gives more detail as to how to run it. Each flag is described in some detail therein.

This program, `process-run.py`, allows you to see how process states change as programs run and either use the CPU (e.g., perform an add instruction) or do I/O (e.g., send a request to a disk and wait for it to complete). See the README for details.

## Questions

1. Run the program with the following flags: `./process-run.py -l 5:100,5:100`. What should the CPU utilization be (e.g., the percent of time the CPU is in use?) Why do you know this? Use the `-c` and `-p` flags to see if you were right.
2. Now run with these flags: `./process-run.py -l 4:100,1:0`. These flags specify one process with 4 instructions (all to use the CPU), and one that simply issues an I/O and waits for it to be done. How long does it take to complete both processes? Use `-c` and `-p` to find out if you were right.
3. Now switch the order of the processes: `./process-run.py -l 1:0,4:100`. What happens now? Does switching the order matter? Why? (As always, use `-c` and `-p` to see if you were right)
4. We'll now explore some of the other flags. One important flag is `-s`, which determines how the system reacts when a process issues an I/O. With the flag set to `SWITCH_ON_END`, the system will NOT switch to another process while one is doing I/O, instead waiting until the process is completely finished. What happens when you run the following two processes, one doing I/O and the other doing CPU work? (`-l 1:0,4:100 -c -s SWITCH_ON_END`)
5. Now, run the same processes, but with the switching behavior set to switch to another process whenever one is WAITING for I/O (`-l 1:0,4:100 -c -s SWITCH_ON_IO`). What happens now? Use `-c` and `-p` to confirm that you are right.
6. One other important behavior is what to do when an I/O completes. With `-I IO_RUN_LATER`, when an I/O completes, the pro-



cess that issued it is not necessarily run right away; rather, whatever was running at the time keeps running. What happens when you run this combination of processes? (`./process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH_ON_IO -I IO_RUN_LATER -c -p`) Are system resources being effectively utilized?

7. Now run the same processes, but with `-I IO_RUN_IMMEDIATE` set, which immediately runs the process that issued the I/O. How does this behavior differ? Why might running a process that just completed an I/O again be a good idea?
8. Now run with some randomly generated processes, e.g., `-s 1 -l 3:50,3:50, -s 2 -l 3:50,3:50, -s 3 -l 3:50,3:50`. See if you can predict how the trace will turn out. What happens when you use `-I IO_RUN_IMMEDIATE` vs. `-I IO_RUN_LATER`? What happens when you use `-S SWITCH_ON_IO` vs. `-S SWITCH_ON_END`?

## Interlude: Process API

### ASIDE: INTERLUDES

Interludes will cover more practical aspects of systems, including a particular focus on operating system APIs and how to use them. If you don't like practical things, you could skip these interludes. But you should like practical things, because, well, they are generally useful in real life; companies, for example, don't usually hire you for your non-practical skills.

In this interlude, we discuss process creation in UNIX systems. UNIX presents one of the most intriguing ways to create a new process with a pair of system calls: `fork()` and `exec()`. A third routine, `wait()`, can be used by a process wishing to wait for a process it has created to complete. We now present these interfaces in more detail, with a few simple examples to motivate us. And thus, our problem:

### CRUX: HOW TO CREATE AND CONTROL PROCESSES

What interfaces should the OS present for process creation and control? How should these interfaces be designed to enable ease of use as well as utility?

## 5.1 The `fork()` System Call

The `fork()` system call is used to create a new process [C63]. However, be forewarned: it is certainly the strangest routine you will ever call<sup>1</sup>. More specifically, you have a running program whose code looks like what you see in Figure 5.1; examine the code, or better yet, type it in and run it yourself!

---

<sup>1</sup>Well, OK, we admit that we don't know that for sure; who knows what routines you call when no one is looking? But `fork()` is pretty odd, no matter how unusual your routine-calling patterns are.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int
6  main(int argc, char *argv[])
7  {
8      printf("hello world (pid:%d)\n", (int) getpid());
9      int rc = fork();
10     if (rc < 0) {                // fork failed; exit
11         fprintf(stderr, "fork failed\n");
12         exit(1);
13     } else if (rc == 0) { // child (new process)
14         printf("hello, I am child (pid:%d)\n", (int) getpid());
15     } else {                  // parent goes down this path (main)
16         printf("hello, I am parent of %d (pid:%d)\n",
17                rc, (int) getpid());
18     }
19     return 0;
20 }

```

Figure 5.1: Calling `fork()` (`p1.c`)

When you run this program (called `p1.c`), you'll see the following:

```

prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>

```

Let us understand what happened in more detail in `p1.c`. When it first started running, the process prints out a hello world message; included in that message is its **process identifier**, also known as a **PID**. The process has a PID of 29146; in UNIX systems, the PID is used to name the process if one wants to do something with the process, such as (for example) stop it from running. So far, so good.

Now the interesting part begins. The process calls the `fork()` system call, which the OS provides as a way to create a new process. The odd part: the process that is created is an (almost) *exact copy of the calling process*. That means that to the OS, it now looks like there are two copies of the program `p1` running, and both are about to return from the `fork()` system call. The newly-created process (called the **child**, in contrast to the creating **parent**) doesn't start running at `main()`, like you might expect (note, the "hello, world" message only got printed out once); rather, it just comes into life as if it had called `fork()` itself.

You might have noticed: the child isn't an *exact* copy. Specifically, although it now has its own copy of the address space (i.e., its own private memory), its own registers, its own PC, and so forth, the value it returns to the caller of `fork()` is different. Specifically, while the parent receives the PID of the newly-created child, the child is simply returned a 0. This differentiation is useful, because it is simple then to write the code that handles the two different cases (as above).

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int
7  main(int argc, char *argv[])
8  {
9      printf("hello world (pid:%d)\n", (int) getpid());
10     int rc = fork();
11     if (rc < 0) {          // fork failed; exit
12         fprintf(stderr, "fork failed\n");
13         exit(1);
14     } else if (rc == 0) { // child (new process)
15         printf("hello, I am child (pid:%d)\n", (int) getpid());
16     } else {              // parent goes down this path (main)
17         int wc = wait(NULL);
18         printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
19               rc, wc, (int) getpid());
20     }
21     return 0;
22 }

```

Figure 5.2: Calling `fork()` And `wait()` (`p2.c`)

You might also have noticed: the output (of `p1.c`) is not **deterministic**. When the child process is created, there are now two active processes in the system that we care about: the parent and the child. Assuming we are running on a system with a single CPU (for simplicity), then either the child or the parent might run at that point. In our example (above), the parent did and thus printed out its message first. In other cases, the opposite might happen, as we show in this output trace:

```

prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>

```

The CPU **scheduler**, a topic we'll discuss in great detail soon, determines which process runs at a given moment in time; because the scheduler is complex, we cannot usually make strong assumptions about what it will choose to do, and hence which process will run first. This **non-determinism**, as it turns out, leads to some interesting problems, particularly in **multi-threaded programs**; hence, we'll see a lot more non-determinism when we study **concurrency** in the second part of the book.

## 5.2 The `wait()` System Call

So far, we haven't done much: just created a child that prints out a message and exits. Sometimes, as it turns out, it is quite useful for a parent to wait for a child process to finish what it has been doing. This task is accomplished with the `wait()` system call (or its more complete sibling `waitpid()`); see Figure 5.2 for details.

In this example (`p2.c`), the parent process calls `wait()` to delay its execution until the child finishes executing. When the child is done, `wait()` returns to the parent.

Adding a `wait()` call to the code above makes the output deterministic. Can you see why? Go ahead, think about it.

*(waiting for you to think .... and done)*

Now that you have thought a bit, here is the output:

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```

With this code, we now know that the child will always print first. Why do we know that? Well, it might simply run first, as before, and thus print before the parent. However, if the parent does happen to run first, it will immediately call `wait()`; this system call won't return until the child has run and exited<sup>2</sup>. Thus, even when the parent runs first, it politely waits for the child to finish running, then `wait()` returns, and then the parent prints its message.

### 5.3 Finally, The `exec()` System Call

A final and important piece of the process creation API is the `exec()` system call<sup>3</sup>. This system call is useful when you want to run a program that is different from the calling program. For example, calling `fork()` in `p2.c` is only useful if you want to keep running copies of the same program. However, often you want to run a *different* program; `exec()` does just that (Figure 5.3, page 5).

In this example, the child process calls `execvp()` in order to run the program `wc`, which is the word counting program. In fact, it runs `wc` on the source file `p3.c`, thus telling us how many lines, words, and bytes are found in the file:

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
    29      107      1030 p3.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
prompt>
```

<sup>2</sup>There are a few cases where `wait()` returns before the child exits; read the man page for more details, as always. And beware of any absolute and unqualified statements this book makes, such as “the child will always print first” or “UNIX is the best thing in the world, even better than ice cream.”

<sup>3</sup>Actually, there are six variants of `exec()`: `execl()`, `execle()`, `execlp()`, `execv()`, and `execvp()`. Read the man pages to learn more.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <sys/wait.h>
6
7  int
8  main(int argc, char *argv[])
9  {
10     printf("hello world (pid:%d)\n", (int) getpid());
11     int rc = fork();
12     if (rc < 0) {          // fork failed; exit
13         fprintf(stderr, "fork failed\n");
14         exit(1);
15     } else if (rc == 0) { // child (new process)
16         printf("hello, I am child (pid:%d)\n", (int) getpid());
17         char *myargs[3];
18         myargs[0] = strdup("wc"); // program: "wc" (word count)
19         myargs[1] = strdup("p3.c"); // argument: file to count
20         myargs[2] = NULL;          // marks end of array
21         execvp(myargs[0], myargs); // runs word count
22         printf("this shouldn't print out");
23     } else {                // parent goes down this path (main)
24         int wc = wait(NULL);
25         printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
26               rc, wc, (int) getpid());
27     }
28     return 0;
29 }

```

Figure 5.3: Calling `fork()`, `wait()`, And `exec()` (`p3.c`)

The `fork()` system call is strange; its partner in crime, `exec()`, is not so normal either. What it does: given the name of an executable (e.g., `wc`), and some arguments (e.g., `p3.c`), it **loads** code (and static data) from that executable and overwrites its current code segment (and current static data) with it; the heap and stack and other parts of the memory space of the program are re-initialized. Then the OS simply runs that program, passing in any arguments as the `argv` of that process. Thus, it does *not* create a new process; rather, it transforms the currently running program (formerly `p3`) into a different running program (`wc`). After the `exec()` in the child, it is almost as if `p3.c` never ran; a successful call to `exec()` never returns.

## 5.4 Why? Motivating The API

Of course, one big question you might have: why would we build such an odd interface to what should be the simple act of creating a new process? Well, as it turns out, the separation of `fork()` and `exec()` is essential in building a UNIX shell, because it lets the shell run code *after* the call to `fork()` but *before* the call to `exec()`; this code can alter the environment of the about-to-be-run program, and thus enables a variety of interesting features to be readily built.

#### TIP: GETTING IT RIGHT (LAMPSON'S LAW)

As Lampson states in his well-regarded “Hints for Computer Systems Design” [L83], “**Get it right.** Neither abstraction nor simplicity is a substitute for getting it right.” Sometimes, you just have to do the right thing, and when you do, it is way better than the alternatives. There are lots of ways to design APIs for process creation; however, the combination of `fork()` and `exec()` are simple and immensely powerful. Here, the UNIX designers simply got it right. And because Lampson so often “got it right”, we name the law in his honor.

The shell is just a user program<sup>4</sup>. It shows you a **prompt** and then waits for you to type something into it. You then type a command (i.e., the name of an executable program, plus any arguments) into it; in most cases, the shell then figures out where in the file system the executable resides, calls `fork()` to create a new child process to run the command, calls some variant of `exec()` to run the command, and then waits for the command to complete by calling `wait()`. When the child completes, the shell returns from `wait()` and prints out a prompt again, ready for your next command.

The separation of `fork()` and `exec()` allows the shell to do a whole bunch of useful things rather easily. For example:

```
prompt> wc p3.c > newfile.txt
```

In the example above, the output of the program `wc` is **redirected** into the output file `newfile.txt` (the greater-than sign is how said redirection is indicated). The way the shell accomplishes this task is quite simple: when the child is created, before calling `exec()`, the shell closes **standard output** and opens the file `newfile.txt`. By doing so, any output from the soon-to-be-running program `wc` are sent to the file instead of the screen.

Figure 5.4 shows a program that does exactly this. The reason this redirection works is due to an assumption about how the operating system manages file descriptors. Specifically, UNIX systems start looking for free file descriptors at zero. In this case, `STDOUT_FILENO` will be the first available one and thus get assigned when `open()` is called. Subsequent writes by the child process to the standard output file descriptor, for example by routines such as `printf()`, will then be routed transparently to the newly-opened file instead of the screen.

Here is the output of running the `p4.c` program:

```
prompt> ./p4
prompt> cat p4.output
32      109      846 p4.c
prompt>
```

---

<sup>4</sup>And there are lots of shells; `tcsh`, `bash`, and `zsh` to name a few. You should pick one, read its man pages, and learn more about it; all UNIX experts do.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <fcntl.h>
6  #include <sys/wait.h>
7
8  int
9  main(int argc, char *argv[])
10 {
11     int rc = fork();
12     if (rc < 0) { // fork failed; exit
13         fprintf(stderr, "fork failed\n");
14         exit(1);
15     } else if (rc == 0) { // child: redirect standard output to a file
16         close(STDOUT_FILENO);
17         open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
18
19         // now exec "wc"...
20         char *myargs[3];
21         myargs[0] = strdup("wc"); // program: "wc" (word count)
22         myargs[1] = strdup("p4.c"); // argument: file to count
23         myargs[2] = NULL; // marks end of array
24         execvp(myargs[0], myargs); // runs word count
25     } else { // parent goes down this path (main)
26         int wc = wait(NULL);
27     }
28     return 0;
29 }

```

Figure 5.4: All Of The Above With Redirection (**p4.c**)

You'll notice (at least) two interesting tidbits about this output. First, when `p4` is run, it looks as if nothing has happened; the shell just prints the command prompt and is immediately ready for your next command. However, that is not the case; the program `p4` did indeed call `fork()` to create a new child, and then run the `wc` program via a call to `execvp()`. You don't see any output printed to the screen because it has been redirected to the file `p4.output`. Second, you can see that when we `cat` the output file, all the expected output from running `wc` is found. Cool, right?

UNIX pipes are implemented in a similar way, but with the `pipe()` system call. In this case, the output of one process is connected to an in-kernel **pipe** (i.e., queue), and the input of another process is connected to that same pipe; thus, the output of one process seamlessly is used as input to the next, and long and useful chains of commands can be strung together. As a simple example, consider looking for a word in a file, and then counting how many times said word occurs; with pipes and the utilities `grep` and `wc`, it is easy — just type `grep -o foo file | wc -l` into the command prompt and marvel at the result.

Finally, while we just have sketched out the process API at a high level, there is a lot more detail about these calls out there to be learned and digested; we'll learn more, for example, about file descriptors when we talk about file systems in the third part of the book. For now, suffice it to say that the `fork()/exec()` combination is a powerful way to create and manipulate processes.



#### ASIDE: RTFM — READ THE MAN PAGES

Many times in this book, when referring to a particular system call or library call, we'll tell you to read the **manual pages**, or **man pages** for short. Man pages are the original form of documentation that exist on UNIX systems; realize that they were created before the thing called **the web** existed.

Spending some time reading man pages is a key step in the growth of a systems programmer; there are tons of useful tidbits hidden in those pages. Some particularly useful pages to read are the man pages for whichever shell you are using (e.g., **tcsh**, or **bash**), and certainly for any system calls your program makes (in order to see what return values and error conditions exist).

Finally, reading the man pages can save you some embarrassment. When you ask colleagues about some intricacy of `fork()`, they may simply reply: "RTFM." This is your colleagues' way of gently urging you to Read The Man pages. The F in RTFM just adds a little color to the phrase...

## 5.5 Other Parts Of The API

Beyond `fork()`, `exec()`, and `wait()`, there are a lot of other interfaces for interacting with processes in UNIX systems. For example, the `kill()` system call is used to send **signals** to a process, including directives to go to sleep, die, and other useful imperatives. In fact, the entire signals subsystem provides a rich infrastructure to deliver external events to processes, including ways to receive and process those signals.

There are many command-line tools that are useful as well. For example, using the `ps` command allows you to see which processes are running; read the **man pages** for some useful flags to pass to `ps`. The tool `top` is also quite helpful, as it displays the processes of the system and how much CPU and other resources they are eating up. Humorously, many times when you run it, `top` claims it is the top resource hog; perhaps it is a bit of an egomaniac. Finally, there are many different kinds of CPU meters you can use to get a quick glance understanding of the load on your system; for example, we always keep **MenuMeters** (from Raging Menace software) running on our Macintosh toolbars, so we can see how much CPU is being utilized at any moment in time. In general, the more information about what is going on, the better.

## 5.6 Summary

We have introduced some of the APIs dealing with UNIX process creation: `fork()`, `exec()`, and `wait()`. However, we have just skimmed the surface. For more detail, read Stevens and Rago [SR05], of course, particularly the chapters on Process Control, Process Relationships, and Signals. There is much to extract from the wisdom therein.

## References

[C63] “A Multiprocessor System Design”

Melvin E. Conway

AFIPS ’63 Fall Joint Computer Conference

New York, USA 1963

*An early paper on how to design multiprocessing systems; may be the first place the term `fork()` was used in the discussion of spawning new processes.*

[DV66] “Programming Semantics for Multiprogrammed Computations”

Jack B. Dennis and Earl C. Van Horn

Communications of the ACM, Volume 9, Number 3, March 1966

*A classic paper that outlines the basics of multiprogrammed computer systems. Undoubtedly had great influence on Project MAC, Multics, and eventually UNIX.*

[L83] “Hints for Computer Systems Design”

Butler Lampson

ACM Operating Systems Review, 15:5, October 1983

*Lampson’s famous hints on how to design computer systems. You should read it at some point in your life, and probably at many points in your life.*

[SR05] “Advanced Programming in the UNIX Environment”

W. Richard Stevens and Stephen A. Rago

Addison-Wesley, 2005

*All nuances and subtleties of using UNIX APIs are found herein. Buy this book! Read it! And most importantly, live it.*

**ASIDE: CODING HOMEWORKS**

Coding homeworks are small exercises where you write code to run on a real machine to get some experience with some of the basic APIs that modern operating systems have to offer. After all, you are (probably) a computer scientist, and therefore should like to code, right? Of course, to truly become an expert, you have to spend more than a little time hacking away at the machine; indeed, find every excuse you can to write some code and see how it works. Spend the time, and become the wise master you know you can be.

**Homework (Code)**

In this homework, you are to gain some familiarity with the process management APIs about which you just read. Don't worry – it's even more fun than it sounds! You'll in general be much better off if you find as much time as you can to write some code<sup>5</sup>, so why not start now?

**Questions**

1. Write a program that calls `fork()`. Before calling `fork()`, have the main process access a variable (e.g., `x`) and set its value to something (e.g., 100). What value is the variable in the child process? What happens to the variable when both the child and parent change the value of `x`?
2. Write a program that opens a file (with the `open()` system call) and then calls `fork()` to create a new process. Can both the child and parent access the file descriptor returned by `open()`? What happens when they are writing to the file concurrently, i.e., at the same time?
3. Write another program using `fork()`. The child process should print "hello"; the parent process should print "goodbye". You should try to ensure that the child process always prints first; can you do this *without* calling `wait()` in the parent?
4. Write a program that calls `fork()` and then calls some form of `exec()` to run the program `/bin/ls`. See if you can try all of the variants of `exec()`, including `execl()`, `execle()`, `execlp()`, `execv()`, `execvp()`, and `execvpP()`. Why do you think there are so many variants of the same basic call?
5. Now write a program that uses `wait()` to wait for the child process to finish in the parent. What does `wait()` return? What happens if you use `wait()` in the child?

---

<sup>5</sup>If you don't like to code, but want to become a computer scientist, this means you need to either (a) become really good at the theory of computer science, or (b) perhaps rethink this whole "computer science" thing you've been telling everyone about.

6. Write a slight modification of the previous program, this time using `waitpid()` instead of `wait()`. When would `waitpid()` be useful?
7. Write a program that creates a child process, and then in the child closes standard output (`STDOUT_FILENO`). What happens if the child calls `printf()` to print some output after closing the descriptor?
8. Write a program that creates two children, and connects the standard output of one to the standard input of the other, using the `pipe()` system call.

## Mechanism: Limited Direct Execution

In order to virtualize the CPU, the operating system needs to somehow share the physical CPU among many jobs running seemingly at the same time. The basic idea is simple: run one process for a little while, then run another one, and so forth. By **time sharing** the CPU in this manner, virtualization is achieved.

There are a few challenges, however, in building such virtualization machinery. The first is *performance*: how can we implement virtualization without adding excessive overhead to the system? The second is *control*: how can we run processes efficiently while retaining control over the CPU? Control is particularly important to the OS, as it is in charge of resources; without control, a process could simply run forever and take over the machine, or access information that it should not be allowed to access. Obtaining high performance while maintaining control is thus one of the central challenges in building an operating system.

### THE CRUX:

#### HOW TO EFFICIENTLY VIRTUALIZE THE CPU WITH CONTROL

The OS must virtualize the CPU in an efficient manner while retaining control over the system. To do so, both hardware and operating-system support will be required. The OS will often use a judicious bit of hardware support in order to accomplish its work effectively.

### 6.1 Basic Technique: Limited Direct Execution

To make a program run as fast as one might expect, not surprisingly OS developers came up with a technique, which we call **limited direct execution**. The “direct execution” part of the idea is simple: just run the program directly on the CPU. Thus, when the OS wishes to start a program running, it creates a process entry for it in a process list, allocates some memory for it, loads the program code into memory (from disk), locates its entry point (i.e., the `main()` routine or something similar), jumps

OS	Program
Create entry for process list	
Allocate memory for program	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Execute <b>call</b> main()	
	Run main()
	Execute <b>return</b> from main
Free memory of process	
Remove from process list	

Figure 6.1: **Direct Execution Protocol (Without Limits)**

to it, and starts running the user’s code. Figure 6.1 shows this basic direct execution protocol (without any limits, yet), using a normal call and return to jump to the program’s `main()` and later to get back into the kernel.

Sounds simple, no? But this approach gives rise to a few problems in our quest to virtualize the CPU. The first is simple: if we just run a program, how can the OS make sure the program doesn’t do anything that we don’t want it to do, while still running it efficiently? The second: when we are running a process, how does the operating system stop it from running and switch to another process, thus implementing the **time sharing** we require to virtualize the CPU?

In answering these questions below, we’ll get a much better sense of what is needed to virtualize the CPU. In developing these techniques, we’ll also see where the “limited” part of the name arises from; without limits on running programs, the OS wouldn’t be in control of anything and thus would be “just a library” — a very sad state of affairs for an aspiring operating system!

6.2 Problem #1: Restricted Operations

Direct execution has the obvious advantage of being fast; the program runs natively on the hardware CPU and thus executes as quickly as one would expect. But running on the CPU introduces a problem: what if the process wishes to perform some kind of restricted operation, such as issuing an I/O request to a disk, or gaining access to more system resources such as CPU or memory?

THE CRUX: HOW TO PERFORM RESTRICTED OPERATIONS

A process must be able to perform I/O and some other restricted operations, but without giving the process complete control over the system. How can the OS and hardware work together to do so?

**TIP: USE PROTECTED CONTROL TRANSFER**

The hardware assists the OS by providing different modes of execution. In **user mode**, applications do not have full access to hardware resources. In **kernel mode**, the OS has access to the full resources of the machine. Special instructions to **trap** into the kernel and **return-from-trap** back to user-mode programs are also provided, as well instructions that allow the OS to tell the hardware where the **trap table** resides in memory.

One approach would simply be to let any process do whatever it wants in terms of I/O and other related operations. However, doing so would prevent the construction of many kinds of systems that are desirable. For example, if we wish to build a file system that checks permissions before granting access to a file, we can't simply let any user process issue I/Os to the disk; if we did, a process could simply read or write the entire disk and thus all protections would be lost.

Thus, the approach we take is to introduce a new processor mode, known as **user mode**; code that runs in user mode is restricted in what it can do. For example, when running in user mode, a process can't issue I/O requests; doing so would result in the processor raising an exception; the OS would then likely kill the process.

In contrast to user mode is **kernel mode**, which the operating system (or kernel) runs in. In this mode, code that runs can do what it likes, including privileged operations such as issuing I/O requests and executing all types of restricted instructions.

We are still left with a challenge, however: what should a user process do when it wishes to perform some kind of privileged operation, such as reading from disk? To enable this, virtually all modern hardware provides the ability for user programs to perform a **system call**. Pioneered on ancient machines such as the Atlas [K+61,L78], system calls allow the kernel to carefully expose certain key pieces of functionality to user programs, such as accessing the file system, creating and destroying processes, communicating with other processes, and allocating more memory. Most operating systems provide a few hundred calls (see the POSIX standard for details [P10]); early Unix systems exposed a more concise subset of around twenty calls.

To execute a system call, a program must execute a special **trap** instruction. This instruction simultaneously jumps into the kernel and raises the privilege level to kernel mode; once in the kernel, the system can now perform whatever privileged operations are needed (if allowed), and thus do the required work for the calling process. When finished, the OS calls a special **return-from-trap** instruction, which, as you might expect, returns into the calling user program while simultaneously reducing the privilege level back to user mode.

The hardware needs to be a bit careful when executing a trap, in that it must make sure to save enough of the caller's registers in order to be able

#### ASIDE: WHY SYSTEM CALLS LOOK LIKE PROCEDURE CALLS

You may wonder why a call to a system call, such as `open()` or `read()`, looks exactly like a typical procedure call in C; that is, if it looks just like a procedure call, how does the system know it's a system call, and do all the right stuff? The simple reason: it *is* a procedure call, but hidden inside that procedure call is the famous trap instruction. More specifically, when you call `open()` (for example), you are executing a procedure call into the C library. Therein, whether for `open()` or any of the other system calls provided, the library uses an agreed-upon calling convention with the kernel to put the arguments to `open` in well-known locations (e.g., on the stack, or in specific registers), puts the system-call number into a well-known location as well (again, onto the stack or a register), and then executes the aforementioned trap instruction. The code in the library after the trap unpacks return values and returns control to the program that issued the system call. Thus, the parts of the C library that make system calls are hand-coded in assembly, as they need to carefully follow convention in order to process arguments and return values correctly, as well as execute the hardware-specific trap instruction. And now you know why you personally don't have to write assembly code to trap into an OS; somebody has already written that assembly for you.

to return correctly when the OS issues the return-from-trap instruction. On x86, for example, the processor will push the program counter, flags, and a few other registers onto a per-process **kernel stack**; the return-from-trap will pop these values off the stack and resume execution of the user-mode program (see the Intel systems manuals [I11] for details). Other hardware systems use different conventions, but the basic concepts are similar across platforms.

There is one important detail left out of this discussion: how does the trap know which code to run inside the OS? Clearly, the calling process can't specify an address to jump to (as you would when making a procedure call); doing so would allow programs to jump anywhere into the kernel which clearly is a bad idea (imagine jumping into code to access a file, but just after a permission check; in fact, it is likely such an ability would enable a wily programmer to get the kernel to run arbitrary code sequences [S07]). Thus the kernel must carefully control what code executes upon a trap.

The kernel does so by setting up a **trap table** at boot time. When the machine boots up, it does so in privileged (kernel) mode, and thus is free to configure machine hardware as need be. One of the first things the OS thus does is to tell the hardware what code to run when certain exceptional events occur. For example, what code should run when a hard-disk interrupt takes place, when a keyboard interrupt occurs, or when a program makes a system call? The OS informs the hardware of the locations of these **trap handlers**, usually with some kind of special in-



OS @ boot (kernel mode)	Hardware	
initialize trap table	remember address of... syscall handler	
OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup user stack with argv Fill kernel stack with reg/PC return-from-trap	restore regs from kernel stack move to user mode jump to main	Run main() ... Call system call trap into OS
Handle trap Do work of syscall return-from-trap	save regs to kernel stack move to kernel mode jump to trap handler	
	restore regs from kernel stack move to user mode jump to PC after trap	... return from main trap (via exit ())
Free memory of process Remove from process list		

Figure 6.2: Limited Direct Execution Protocol

struction. Once the hardware is informed, it remembers the location of these handlers until the machine is next rebooted, and thus the hardware knows what to do (i.e., what code to jump to) when system calls and other exceptional events take place.

One last aside: being able to execute the instruction to tell the hardware where the trap tables are is a very powerful capability. Thus, as you might have guessed, it is also a **privileged** operation. If you try to execute this instruction in user mode, the hardware won't let you, and you can probably guess what will happen (hint: adios, offending program). Point to ponder: what horrible things could you do to a system if you could install your own trap table? Could you take over the machine?

The timeline (with time increasing downward, in Figure 6.2) summarizes the protocol. We assume each process has a kernel stack where registers (including general purpose registers and the program counter) are saved to and restored from (by the hardware) when transitioning into and out of the kernel.

There are two phases in the LDE protocol. In the first (at boot time), the kernel initializes the trap table, and the CPU remembers its location for subsequent use. The kernel does so via a privileged instruction (all privileged instructions are highlighted in bold).

In the second (when running a process), the kernel sets up a few things (e.g., allocating a node on the process list, allocating memory) before using a return-from-trap instruction to start the execution of the process; this switches the CPU to user mode and begins running the process. When the process wishes to issue a system call, it traps back into the OS, which handles it and once again returns control via a return-from-trap to the process. The process then completes its work, and returns from `main()`; this usually will return into some stub code which will properly exit the program (say, by calling the `exit()` system call, which traps into the OS). At this point, the OS cleans up and we are done.

### 6.3 Problem #2: Switching Between Processes

The next problem with direct execution is achieving a switch between processes. Switching between processes should be simple, right? The OS should just decide to stop one process and start another. What's the big deal? But it actually is a little bit tricky: specifically, if a process is running on the CPU, this by definition means the OS is *not* running. If the OS is not running, how can it do anything at all? (hint: it can't) While this sounds almost philosophical, it is a real problem: there is clearly no way for the OS to take an action if it is not running on the CPU. Thus we arrive at the crux of the problem.

#### THE CRUX: HOW TO REGAIN CONTROL OF THE CPU

How can the operating system **regain control** of the CPU so that it can switch between processes?

### A Cooperative Approach: Wait For System Calls

One approach that some systems have taken in the past (for example, early versions of the Macintosh operating system [M11], or the old Xerox Alto system [A79]) is known as the **cooperative** approach. In this style, the OS *trusts* the processes of the system to behave reasonably. Processes that run for too long are assumed to periodically give up the CPU so that the OS can decide to run some other task.

Thus, you might ask, how does a friendly process give up the CPU in this utopian world? Most processes, as it turns out, transfer control of the CPU to the OS quite frequently by making **system calls**, for example, to open a file and subsequently read it, or to send a message to another machine, or to create a new process. Systems like this often include an

**TIP: DEALING WITH APPLICATION MISBEHAVIOR**

Operating systems often have to deal with misbehaving processes, those that either through design (maliciousness) or accident (bugs) attempt to do something that they shouldn't. In modern systems, the way the OS tries to handle such malfeasance is to simply terminate the offender. One strike and you're out! Perhaps brutal, but what else should the OS do when you try to access memory illegally or execute an illegal instruction?

explicit **yield** system call, which does nothing except to transfer control to the OS so it can run other processes.

Applications also transfer control to the OS when they do something illegal. For example, if an application divides by zero, or tries to access memory that it shouldn't be able to access, it will generate a **trap** to the OS. The OS will then have control of the CPU again (and likely terminate the offending process).

Thus, in a cooperative scheduling system, the OS regains control of the CPU by waiting for a system call or an illegal operation of some kind to take place. You might also be thinking: isn't this passive approach less than ideal? What happens, for example, if a process (whether malicious, or just full of bugs) ends up in an infinite loop, and never makes a system call? What can the OS do then?

**A Non-Cooperative Approach: The OS Takes Control**

Without some additional help from the hardware, it turns out the OS can't do much at all when a process refuses to make system calls (or mistakes) and thus return control to the OS. In fact, in the cooperative approach, your only recourse when a process gets stuck in an infinite loop is to resort to the age-old solution to all problems in computer systems: **reboot the machine**. Thus, we again arrive at a subproblem of our general quest to gain control of the CPU.

**THE CRUX: HOW TO GAIN CONTROL WITHOUT COOPERATION**

How can the OS gain control of the CPU even if processes are not being cooperative? What can the OS do to ensure a rogue process does not take over the machine?

The answer turns out to be simple and was discovered by a number of people building computer systems many years ago: a **timer interrupt** [M+63]. A timer device can be programmed to raise an interrupt every so many milliseconds; when the interrupt is raised, the currently running process is halted, and a pre-configured **interrupt handler** in the OS runs. At this point, the OS has regained control of the CPU, and thus can do what it pleases: stop the current process, and start a different one.

**TIP: USE THE TIMER INTERRUPT TO REGAIN CONTROL**

The addition of a **timer interrupt** gives the OS the ability to run again on a CPU even if processes act in a non-cooperative fashion. Thus, this hardware feature is essential in helping the OS maintain control of the machine.

As we discussed before with system calls, the OS must inform the hardware of which code to run when the timer interrupt occurs; thus, at boot time, the OS does exactly that. Second, also during the boot sequence, the OS must start the timer, which is of course a privileged operation. Once the timer has begun, the OS can thus feel safe in that control will eventually be returned to it, and thus the OS is free to run user programs. The timer can also be turned off (also a privileged operation), something we will discuss later when we understand concurrency in more detail.

Note that the hardware has some responsibility when an interrupt occurs, in particular to save enough of the state of the program that was running when the interrupt occurred such that a subsequent return-from-trap instruction will be able to resume the running program correctly. This set of actions is quite similar to the behavior of the hardware during an explicit system-call trap into the kernel, with various registers thus getting saved (e.g., onto a kernel stack) and thus easily restored by the return-from-trap instruction.

## Saving and Restoring Context

Now that the OS has regained control, whether cooperatively via a system call, or more forcefully via a timer interrupt, a decision has to be made: whether to continue running the currently-running process, or switch to a different one. This decision is made by a part of the operating system known as the **scheduler**; we will discuss scheduling policies in great detail in the next few chapters.

If the decision is made to switch, the OS then executes a low-level piece of code which we refer to as a **context switch**. A context switch is conceptually simple: all the OS has to do is save a few register values for the currently-executing process (onto its kernel stack, for example) and restore a few for the soon-to-be-executing process (from its kernel stack). By doing so, the OS thus ensures that when the return-from-trap instruction is finally executed, instead of returning to the process that was running, the system resumes execution of another process.

To save the context of the currently-running process, the OS will execute some low-level assembly code to save the general purpose registers, PC, as well as the kernel stack pointer of the currently-running process, and then restore said registers, PC, and switch to the kernel stack for the soon-to-be-executing process. By switching stacks, the kernel enters the

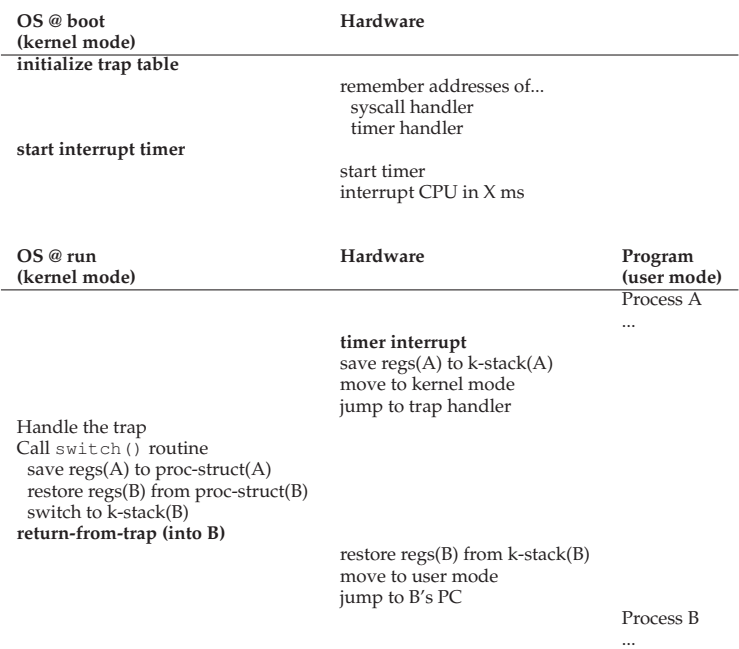


Figure 6.3: Limited Direct Execution Protocol (Timer Interrupt)

call to the switch code in the context of one process (the one that was interrupted) and returns in the context of another (the soon-to-be-executing one). When the OS then finally executes a return-from-trap instruction, the soon-to-be-executing process becomes the currently-running process. And thus the context switch is complete.

A timeline of the entire process is shown in Figure 6.3. In this example, Process A is running and then is interrupted by the timer interrupt. The hardware saves its registers (onto its kernel stack) and enters the kernel (switching to kernel mode). In the timer interrupt handler, the OS decides to switch from running Process A to Process B. At that point, it calls the `switch()` routine, which carefully saves current register values (into the process structure of A), restores the registers of Process B (from its process structure entry), and then **switches contexts**, specifically by changing the stack pointer to use B's kernel stack (and not A's). Finally, the OS returns-from-trap, which restores B's registers and starts running it.

Note that there are two types of register saves/restores that happen during this protocol. The first is when the timer interrupt occurs; in this case, the *user registers* of the running process are implicitly saved by the *hardware*, using the kernel stack of that process. The second is when the OS decides to switch from A to B; in this case, the *kernel registers* are ex-

```

1  # void switch(struct context **old, struct context *new);
2  #
3  # Save current register context in old
4  # and then load register context from new.
5  .globl switch
6  switch:
7      # Save old registers
8      movl 4(%esp), %eax # put old ptr into eax
9      popl 0(%eax)      # save the old IP
10     movl %esp, 4(%eax) # and stack
11     movl %ebx, 8(%eax) # and other registers
12     movl %ecx, 12(%eax)
13     movl %edx, 16(%eax)
14     movl %esi, 20(%eax)
15     movl %edi, 24(%eax)
16     movl %ebp, 28(%eax)
17
18     # Load new registers
19     movl 4(%esp), %eax # put new ptr into eax
20     movl 28(%eax), %ebp # restore other registers
21     movl 24(%eax), %edi
22     movl 20(%eax), %esi
23     movl 16(%eax), %edx
24     movl 12(%eax), %ecx
25     movl 8(%eax), %ebx
26     movl 4(%eax), %esp # stack is switched here
27     pushl 0(%eax)      # return addr put in place
28     ret               # finally return into new ctxt

```

Figure 6.4: The xv6 Context Switch Code

licitly saved by the *software* (i.e., the OS), but this time into memory in the process structure of the process. The latter action moves the system from running as if it just trapped into the kernel from A to as if it just trapped into the kernel from B.

To give you a better sense of how such a switch is enacted, Figure 6.4 shows the context switch code for xv6. See if you can make sense of it (you'll have to know a bit of x86, as well as some xv6, to do so). The context structures *old* and *new* are found in the old and new process's process structures, respectively.

## 6.4 Worried About Concurrency?

Some of you, as attentive and thoughtful readers, may be now thinking: "Hmm... what happens when, during a system call, a timer interrupt occurs?" or "What happens when you're handling one interrupt and another one happens? Doesn't that get hard to handle in the kernel?" Good questions — we really have some hope for you yet!

The answer is yes, the OS does indeed need to be concerned as to what happens if, during interrupt or trap handling, another interrupt occurs. This, in fact, is the exact topic of the entire second piece of this book, on **concurrency**; we'll defer a detailed discussion until then.

**ASIDE: HOW LONG CONTEXT SWITCHES TAKE**

A natural question you might have is: how long does something like a context switch take? Or even a system call? For those of you that are curious, there is a tool called **lmbench** [MS96] that measures exactly those things, as well as a few other performance measures that might be relevant.

Results have improved quite a bit over time, roughly tracking processor performance. For example, in 1996 running Linux 1.3.37 on a 200-MHz P6 CPU, system calls took roughly 4 microseconds, and a context switch roughly 6 microseconds [MS96]. Modern systems perform almost an order of magnitude better, with sub-microsecond results on systems with 2- or 3-GHz processors.

It should be noted that not all operating-system actions track CPU performance. As Ousterhout observed, many OS operations are memory intensive, and memory bandwidth has not improved as dramatically as processor speed over time [O90]. Thus, depending on your workload, buying the latest and greatest processor may not speed up your OS as much as you might hope.

To whet your appetite, we'll just sketch some basics of how the OS handles these tricky situations. One simple thing an OS might do is **disable interrupts** during interrupt processing; doing so ensures that when one interrupt is being handled, no other one will be delivered to the CPU. Of course, the OS has to be careful in doing so; disabling interrupts for too long could lead to lost interrupts, which is (in technical terms) bad.

Operating systems also have developed a number of sophisticated **locking** schemes to protect concurrent access to internal data structures. This enables multiple activities to be on-going within the kernel at the same time, particularly useful on multiprocessors. As we'll see in the next piece of this book on concurrency, though, such locking can be complicated and lead to a variety of interesting and hard-to-find bugs.

## 6.5 Summary

We have described some key low-level mechanisms to implement CPU virtualization, a set of techniques which we collectively refer to as **limited direct execution**. The basic idea is straightforward: just run the program you want to run on the CPU, but first make sure to set up the hardware so as to limit what the process can do without OS assistance.

This general approach is taken in real life as well. For example, those of you who have children, or, at least, have heard of children, may be familiar with the concept of **baby proofing** a room: locking cabinets containing dangerous stuff and covering electrical sockets. When the room is thus readied, you can let your baby roam freely, secure in the knowledge that the most dangerous aspects of the room have been restricted.

**TIP: REBOOT IS USEFUL**

Earlier on, we noted that the only solution to infinite loops (and similar behaviors) under cooperative preemption is to **reboot** the machine. While you may scoff at this hack, researchers have shown that reboot (or in general, starting over some piece of software) can be a hugely useful tool in building robust systems [C+04].

Specifically, reboot is useful because it moves software back to a known and likely more tested state. Reboots also reclaim stale or leaked resources (e.g., memory) which may otherwise be hard to handle. Finally, reboots are easy to automate. For all of these reasons, it is not uncommon in large-scale cluster Internet services for system management software to periodically reboot sets of machines in order to reset them and thus obtain the advantages listed above.

Thus, next time you reboot, you are not just enacting some ugly hack. Rather, you are using a time-tested approach to improving the behavior of a computer system. Well done!

In an analogous manner, the OS “baby proofs” the CPU, by first (during boot time) setting up the trap handlers and starting an interrupt timer, and then by only running processes in a restricted mode. By doing so, the OS can feel quite assured that processes can run efficiently, only requiring OS intervention to perform privileged operations or when they have monopolized the CPU for too long and thus need to be switched out.

We thus have the basic mechanisms for virtualizing the CPU in place. But a major question is left unanswered: which process should we run at a given time? It is this question that the scheduler must answer, and thus the next topic of our study.



## References

- [A79] "Alto User's Handbook"  
Xerox Palo Alto Research Center, September 1979  
Available: <http://history-computer.com/Library/AltoUsersHandbook.pdf>  
*An amazing system, way ahead of its time. Became famous because Steve Jobs visited, took notes, and built Lisa and eventually Mac.*
- [C+04] "Microreboot — A Technique for Cheap Recovery"  
George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, Armando Fox  
OSDI '04, San Francisco, CA, December 2004  
*An excellent paper pointing out how far one can go with reboot in building more robust systems.*
- [I11] "Intel 64 and IA-32 Architectures Software Developer's Manual"  
Volume 3A and 3B: System Programming Guide  
Intel Corporation, January 2011
- [K+61] "One-Level Storage System"  
T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner  
IRE Transactions on Electronic Computers, April 1962  
*The Atlas pioneered much of what you see in modern systems. However, this paper is not the best one to read. If you were to only read one, you might try the historical perspective below [L78].*
- [L78] "The Manchester Mark I and Atlas: A Historical Perspective"  
S. H. Lavington  
Communications of the ACM, 21:1, January 1978  
*A history of the early development of computers and the pioneering efforts of Atlas.*
- [M+63] "A Time-Sharing Debugging System for a Small Computer"  
J. McCarthy, S. Boilen, E. Fredkin, J. C. R. Licklider  
AFIPS '63 (Spring), May, 1963, New York, USA  
*An early paper about time-sharing that refers to using a timer interrupt; the quote that discusses it: "The basic task of the channel 17 clock routine is to decide whether to remove the current user from core and if so to decide which user program to swap in as he goes out."*
- [MS96] "Imbench: Portable tools for performance analysis"  
Larry McVoy and Carl Staelin  
USENIX Annual Technical Conference, January 1996  
*A fun paper about how to measure a number of different things about your OS and its performance. Download Imbench and give it a try.*
- [M11] "Mac OS 9"  
January 2011  
Available: [http://en.wikipedia.org/wiki/Mac\\_OS\\_9](http://en.wikipedia.org/wiki/Mac_OS_9)
- [O90] "Why Aren't Operating Systems Getting Faster as Fast as Hardware?"  
J. Ousterhout  
USENIX Summer Conference, June 1990  
*A classic paper on the nature of operating system performance.*
- [P10] "The Single UNIX Specification, Version 3"  
The Open Group, May 2010  
Available: <http://www.unix.org/version3/>  
*This is hard and painful to read, so probably avoid it if you can.*
- [S07] "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)"  
Hovav Shacham  
CCS '07, October 2007  
*One of those awesome, mind-blowing ideas that you'll see in research from time to time. The author shows that if you can jump into code arbitrarily, you can essentially stitch together any code sequence you like (given a large code base); read the paper for the details. The technique makes it even harder to defend against malicious attacks, alas.*

## Homework (Measurement)

### ASIDE: MEASUREMENT HOMEWORKS

Measurement homeworks are small exercises where you write code to run on a real machine, in order to measure some aspect of OS or hardware performance. The idea behind such homeworks is to give you a little bit of hands-on experience with a real operating system.

In this homework, you'll measure the costs of a system call and context switch. Measuring the cost of a system call is relatively easy. For example, you could repeatedly call a simple system call (e.g., performing a 0-byte read), and time how long it takes; dividing the time by the number of iterations gives you an estimate of the cost of a system call.

One thing you'll have to take into account is the precision and accuracy of your timer. A typical timer that you can use is `gettimeofday()`; read the man page for details. What you'll see there is that `gettimeofday()` returns the time in microseconds since 1970; however, this does not mean that the timer is precise to the microsecond. Measure back-to-back calls to `gettimeofday()` to learn something about how precise the timer really is; this will tell you how many iterations of your null system-call test you'll have to run in order to get a good measurement result. If `gettimeofday()` is not precise enough for you, you might look into using the `rdtsc` instruction available on x86 machines.

Measuring the cost of a context switch is a little trickier. The `lmbench` benchmark does so by running two processes on a single CPU, and setting up two UNIX pipes between them; a pipe is just one of many ways processes in a UNIX system can communicate with one another. The first process then issues a write to the first pipe, and waits for a read on the second; upon seeing the first process waiting for something to read from the second pipe, the OS puts the first process in the blocked state, and switches to the other process, which reads from the first pipe and then writes to the second. When the second process tries to read from the first pipe again, it blocks, and thus the back-and-forth cycle of communication continues. By measuring the cost of communicating like this repeatedly, `lmbench` can make a good estimate of the cost of a context switch. You can try to re-create something similar here, using pipes, or perhaps some other communication mechanism such as UNIX sockets.

One difficulty in measuring context-switch cost arises in systems with more than one CPU; what you need to do on such a system is ensure that your context-switching processes are located on the same processor. Fortunately, most operating systems have calls to bind a process to a particular processor; on Linux, for example, the `sched.setaffinity()` call is what you're looking for. By ensuring both processes are on the same processor, you are making sure to measure the cost of the OS stopping one process and restoring another on the same CPU.

## Scheduling: Introduction

By now low-level **mechanisms** of running processes (e.g., context switching) should be clear; if they are not, go back a chapter or two, and read the description of how that stuff works again. However, we have yet to understand the high-level **policies** that an OS scheduler employs. We will now do just that, presenting a series of **scheduling policies** (sometimes called **disciplines**) that various smart and hard-working people have developed over the years.

The origins of scheduling, in fact, predate computer systems; early approaches were taken from the field of operations management and applied to computers. This reality should be no surprise: assembly lines and many other human endeavors also require scheduling, and many of the same concerns exist therein, including a laser-like desire for efficiency. And thus, our problem:

### THE CRUX: HOW TO DEVELOP SCHEDULING POLICY

How should we develop a basic framework for thinking about scheduling policies? What are the key assumptions? What metrics are important? What basic approaches have been used in the earliest of computer systems?

## 7.1 Workload Assumptions

Before getting into the range of possible policies, let us first make a number of simplifying assumptions about the processes running in the system, sometimes collectively called the **workload**. Determining the workload is a critical part of building policies, and the more you know about workload, the more fine-tuned your policy can be.

The workload assumptions we make here are mostly unrealistic, but that is alright (for now), because we will relax them as we go, and eventually develop what we will refer to as ... (*dramatic pause*) ...

a **fully-operational scheduling discipline**<sup>1</sup>.

We will make the following assumptions about the processes, sometimes called **jobs**, that are running in the system:

1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU (i.e., they perform no I/O)
5. The run-time of each job is known.

We said many of these assumptions were unrealistic, but just as some animals are more equal than others in Orwell's *Animal Farm* [O45], some assumptions are more unrealistic than others in this chapter. In particular, it might bother you that the run-time of each job is known: this would make the scheduler omniscient, which, although it would be great (probably), is not likely to happen anytime soon.

## 7.2 Scheduling Metrics

Beyond making workload assumptions, we also need one more thing to enable us to compare different scheduling policies: a **scheduling metric**. A metric is just something that we use to *measure* something, and there are a number of different metrics that make sense in scheduling.

For now, however, let us also simplify our life by simply having a single metric: **turnaround time**. The turnaround time of a job is defined as the time at which the job completes minus the time at which the job arrived in the system. More formally, the turnaround time  $T_{\text{turnaround}}$  is:

$$T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}} \quad (7.1)$$

Because we have assumed that all jobs arrive at the same time, for now  $T_{\text{arrival}} = 0$  and hence  $T_{\text{turnaround}} = T_{\text{completion}}$ . This fact will change as we relax the aforementioned assumptions.

You should note that turnaround time is a **performance** metric, which will be our primary focus this chapter. Another metric of interest is **fairness**, as measured (for example) by **Jain's Fairness Index** [J91]. Performance and fairness are often at odds in scheduling; a scheduler, for example, may optimize performance but at the cost of preventing a few jobs from running, thus decreasing fairness. This conundrum shows us that life isn't always perfect.

## 7.3 First In, First Out (FIFO)

The most basic algorithm we can implement is known as **First In, First Out (FIFO)** scheduling or sometimes **First Come, First Served (FCFS)**.

---

<sup>1</sup>Said in the same way you would say "A fully-operational Death Star."

FIFO has a number of positive properties: it is clearly simple and thus easy to implement. And, given our assumptions, it works pretty well.

Let's do a quick example together. Imagine three jobs arrive in the system, A, B, and C, at roughly the same time ( $T_{arrival} = 0$ ). Because FIFO has to put some job first, let's assume that while they all arrived simultaneously, A arrived just a hair before B which arrived just a hair before C. Assume also that each job runs for 10 seconds. What will the **average turnaround time** be for these jobs?

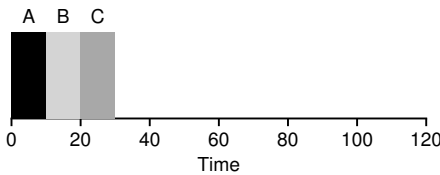


Figure 7.1: FIFO Simple Example

From Figure 7.1, you can see that A finished at 10, B at 20, and C at 30. Thus, the average turnaround time for the three jobs is simply  $\frac{10+20+30}{3} = 20$ . Computing turnaround time is as easy as that.

Now let's relax one of our assumptions. In particular, let's relax assumption 1, and thus no longer assume that each job runs for the same amount of time. How does FIFO perform now? What kind of workload could you construct to make FIFO perform poorly?

*(think about this before reading on ... keep thinking ... got it?!)*

Presumably you've figured this out by now, but just in case, let's do an example to show how jobs of different lengths can lead to trouble for FIFO scheduling. In particular, let's again assume three jobs (A, B, and C), but this time A runs for 100 seconds while B and C run for 10 each.

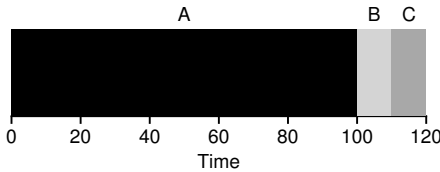


Figure 7.2: Why FIFO Is Not That Great

As you can see in Figure 7.2, Job A runs first for the full 100 seconds before B or C even get a chance to run. Thus, the average turnaround time for the system is high: a painful 110 seconds ( $\frac{100+110+120}{3} = 110$ ).

This problem is generally referred to as the **convoy effect** [B+79], where a number of relatively-short potential consumers of a resource get queued behind a heavyweight resource consumer. This scheduling scenario might remind you of a single line at a grocery store and what you feel like when

TIP: THE PRINCIPLE OF SJF

Shortest Job First represents a general scheduling principle that can be applied to any system where the perceived turnaround time per customer (or, in our case, a job) matters. Think of any line you have waited in: if the establishment in question cares about customer satisfaction, it is likely they have taken SJF into account. For example, grocery stores commonly have a “ten-items-or-less” line to ensure that shoppers with only a few things to purchase don’t get stuck behind the family preparing for some upcoming nuclear winter.

you see the person in front of you with three carts full of provisions and a checkbook out; it’s going to be a while<sup>2</sup>.

So what should we do? How can we develop a better algorithm to deal with our new reality of jobs that run for different amounts of time? Think about it first; then read on.

## 7.4 Shortest Job First (SJF)

It turns out that a very simple approach solves this problem; in fact it is an idea stolen from operations research [C54,PV56] and applied to scheduling of jobs in computer systems. This new scheduling discipline is known as **Shortest Job First (SJF)**, and the name should be easy to remember because it describes the policy quite completely: it runs the shortest job first, then the next shortest, and so on.

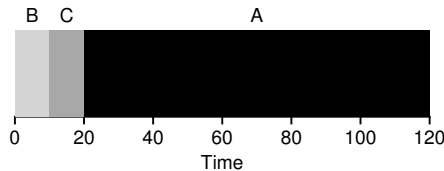


Figure 7.3: SJF Simple Example

Let’s take our example above but with SJF as our scheduling policy. Figure 7.3 shows the results of running A, B, and C. Hopefully the diagram makes it clear why SJF performs much better with regards to average turnaround time. Simply by running B and C before A, SJF reduces average turnaround from 110 seconds to 50 ( $\frac{10+20+120}{3} = 50$ ), more than a factor of two improvement.

In fact, given our assumptions about jobs all arriving at the same time, we could prove that SJF is indeed an **optimal** scheduling algorithm. How-

<sup>2</sup>Recommended action in this case: either quickly switch to a different line, or take a long, deep, and relaxing breath. That’s right, breathe in, breathe out. It will be OK, don’t worry.

### ASIDE: PREEMPTIVE SCHEDULERS

In the old days of batch computing, a number of **non-preemptive** schedulers were developed; such systems would run each job to completion before considering whether to run a new job. Virtually all modern schedulers are **preemptive**, and quite willing to stop one process from running in order to run another. This implies that the scheduler employs the mechanisms we learned about previously; in particular, the scheduler can perform a **context switch**, stopping one running process temporarily and resuming (or starting) another.

ever, you are in a systems class, not theory or operations research; no proofs are allowed.

Thus we arrive upon a good approach to scheduling with SJF, but our assumptions are still fairly unrealistic. Let's relax another. In particular, we can target assumption 2, and now assume that jobs can arrive at any time instead of all at once. What problems does this lead to?

*(Another pause to think ... are you thinking? Come on, you can do it)*

Here we can illustrate the problem again with an example. This time, assume A arrives at  $t = 0$  and needs to run for 100 seconds, whereas B and C arrive at  $t = 10$  and each need to run for 10 seconds. With pure SJF, we'd get the schedule seen in Figure 7.4.

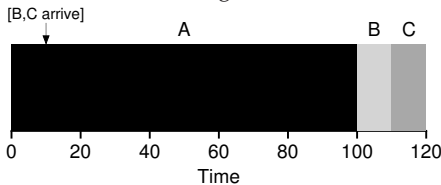


Figure 7.4: SJF With Late Arrivals From B and C

As you can see from the figure, even though B and C arrived shortly after A, they still are forced to wait until A has completed, and thus suffer the same convoy problem. Average turnaround time for these three jobs is 103.33 seconds ( $\frac{100 + (110 - 10) + (120 - 10)}{3}$ ). What can a scheduler do?

## 7.5 Shortest Time-to-Completion First (STCF)

To address this concern, we need to relax assumption 3 (that jobs must run to completion), so let's do that. We also need some machinery within the scheduler itself. As you might have guessed, given our previous discussion about timer interrupts and context switching, the scheduler can certainly do something else when B and C arrive: it can **preempt** job A and decide to run another job, perhaps continuing A later. SJF by our definition is a **non-preemptive** scheduler, and thus suffers from the problems described above.

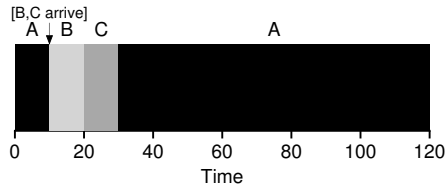


Figure 7.5: STCF Simple Example

Fortunately, there is a scheduler which does exactly that: add preempt to SJF, known as the **Shortest Time-to-Completion First (STCF)** or **Preemptive Shortest Job First (PSJF)** scheduler [CK68]. Any time a new job enters the system, the STCF scheduler determines which of the remaining jobs (including the new job) has the least time left, and schedules that one. Thus, in our example, STCF would preempt A and run B and C to completion; only when they are finished would A’s remaining time be scheduled. Figure 7.5 shows an example.

The result is a much-improved average turnaround time: 50 seconds  $(\frac{(120-0)+(20-10)+(30-10)}{3})$ . And as before, given our new assumptions, STCF is provably optimal; given that SJF is optimal if all jobs arrive at the same time, you should probably be able to see the intuition behind the optimality of STCF.

## 7.6 A New Metric: Response Time

Thus, if we knew job lengths, and that jobs only used the CPU, and our only metric was turnaround time, STCF would be a great policy. In fact, for a number of early batch computing systems, these types of scheduling algorithms made some sense. However, the introduction of time-shared machines changed all that. Now users would sit at a terminal and demand interactive performance from the system as well. And thus, a new metric was born: **response time**.

Response time is defined as the time from when the job arrives in a system to the first time it is scheduled. More formally:

$$T_{response} = T_{firstrun} - T_{arrival} \tag{7.2}$$

For example, if we had the schedule above (with A arriving at time 0, and B and C at time 10), the response time of each job is as follows: 0 for job A, 0 for B, and 10 for C (average: 3.33).

As you might be thinking, STCF and related disciplines are not particularly good for response time. If three jobs arrive at the same time, for example, the third job has to wait for the previous two jobs to run *in their entirety* before being scheduled just once. While great for turnaround time, this approach is quite bad for response time and interactivity. Indeed, imagine sitting at a terminal, typing, and having to wait 10 seconds



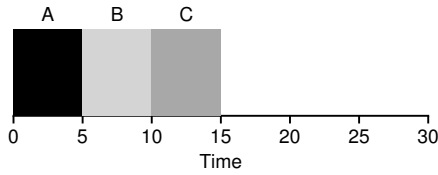


Figure 7.6: SJF Again (Bad for Response Time)

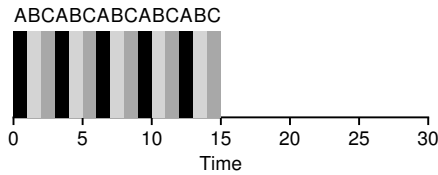


Figure 7.7: Round Robin (Good for Response Time)

to see a response from the system just because some other job got scheduled in front of yours: not too pleasant.

Thus, we are left with another problem: how can we build a scheduler that is sensitive to response time?

### 7.7 Round Robin

To solve this problem, we will introduce a new scheduling algorithm, classically referred to as **Round-Robin (RR)** scheduling [K64]. The basic idea is simple: instead of running jobs to completion, RR runs a job for a **time slice** (sometimes called a **scheduling quantum**) and then switches to the next job in the run queue. It repeatedly does so until the jobs are finished. For this reason, RR is sometimes called **time-slicing**. Note that the length of a time slice must be a multiple of the timer-interrupt period; thus if the timer interrupts every 10 milliseconds, the time slice could be 10, 20, or any other multiple of 10 ms.

To understand RR in more detail, let's look at an example. Assume three jobs A, B, and C arrive at the same time in the system, and that they each wish to run for 5 seconds. An SJF scheduler runs each job to completion before running another (Figure 7.6). In contrast, RR with a time-slice of 1 second would cycle through the jobs quickly (Figure 7.7).

The average response time of RR is:  $\frac{0+1+2}{3} = 1$ ; for SJF, average response time is:  $\frac{0+5+10}{3} = 5$ .

As you can see, the length of the time slice is critical for RR. The shorter it is, the better the performance of RR under the response-time metric. However, making the time slice too short is problematic: suddenly the cost of context switching will dominate overall performance. Thus, deciding on the length of the time slice presents a trade-off to a system designer, making it long enough to **amortize** the cost of switching without making it so long that the system is no longer responsive.

**TIP: AMORTIZATION CAN REDUCE COSTS**

The general technique of **amortization** is commonly used in systems when there is a fixed cost to some operation. By incurring that cost less often (i.e., by performing the operation fewer times), the total cost to the system is reduced. For example, if the time slice is set to 10 ms, and the context-switch cost is 1 ms, roughly 10% of time is spent context switching and is thus wasted. If we want to *amortize* this cost, we can increase the time slice, e.g., to 100 ms. In this case, less than 1% of time is spent context switching, and thus the cost of time-slicing has been amortized.

Note that the cost of context switching does not arise solely from the OS actions of saving and restoring a few registers. When programs run, they build up a great deal of state in CPU caches, TLBs, branch predictors, and other on-chip hardware. Switching to another job causes this state to be flushed and new state relevant to the currently-running job to be brought in, which may exact a noticeable performance cost [MB91].

RR, with a reasonable time slice, is thus an excellent scheduler if response time is our only metric. But what about our old friend turnaround time? Let's look at our example above again. A, B, and C, each with running times of 5 seconds, arrive at the same time, and RR is the scheduler with a (long) 1-second time slice. We can see from the picture above that A finishes at 13, B at 14, and C at 15, for an average of 14. Pretty awful!

It is not surprising, then, that RR is indeed one of the *worst* policies if turnaround time is our metric. Intuitively, this should make sense: what RR is doing is stretching out each job as long as it can, by only running each job for a short bit before moving to the next. Because turnaround time only cares about when jobs finish, RR is nearly pessimal, even worse than simple FIFO in many cases.

More generally, any policy (such as RR) that is **fair**, i.e., that evenly divides the CPU among active processes on a small time scale, will perform poorly on metrics such as turnaround time. Indeed, this is an inherent trade-off: if you are willing to be unfair, you can run shorter jobs to completion, but at the cost of response time; if you instead value fairness, response time is lowered, but at the cost of turnaround time. This type of **trade-off** is common in systems; you can't have your cake and eat it too<sup>3</sup>.

We have developed two types of schedulers. The first type (SJF, STCF) optimizes turnaround time, but is bad for response time. The second type (RR) optimizes response time but is bad for turnaround. And we still have two assumptions which need to be relaxed: assumption 4 (that jobs do no I/O), and assumption 5 (that the run-time of each job is known). Let's tackle those assumptions next.

<sup>3</sup>A saying that confuses people, because it should be "You can't *keep* your cake and eat it too" (which is kind of obvious, no?). Amazingly, there is a wikipedia page about this saying; even more amazingly, it is kind of fun to read [W15]. As they say in Italian, you can't *Avere la botte piena e la moglie ubriaca*.

**TIP: OVERLAP ENABLES HIGHER UTILIZATION**

When possible, **overlap** operations to maximize the utilization of systems. Overlap is useful in many different domains, including when performing disk I/O or sending messages to remote machines; in either case, starting the operation and then switching to other work is a good idea, and improves the overall utilization and efficiency of the system.

## 7.8 Incorporating I/O

First we will relax assumption 4 — of course all programs perform I/O. Imagine a program that didn't take any input: it would produce the same output each time. Imagine one without output: it is the proverbial tree falling in the forest, with no one to see it; it doesn't matter that it ran.

A scheduler clearly has a decision to make when a job initiates an I/O request, because the currently-running job won't be using the CPU during the I/O; it is **blocked** waiting for I/O completion. If the I/O is sent to a hard disk drive, the process might be blocked for a few milliseconds or longer, depending on the current I/O load of the drive. Thus, the scheduler should probably schedule another job on the CPU at that time.

The scheduler also has to make a decision when the I/O completes. When that occurs, an interrupt is raised, and the OS runs and moves the process that issued the I/O from blocked back to the ready state. Of course, it could even decide to run the job at that point. How should the OS treat each job?

To understand this issue better, let us assume we have two jobs, A and B, which each need 50 ms of CPU time. However, there is one obvious difference: A runs for 10 ms and then issues an I/O request (assume here that I/Os each take 10 ms), whereas B simply uses the CPU for 50 ms and performs no I/O. The scheduler runs A first, then B after (Figure 7.8).

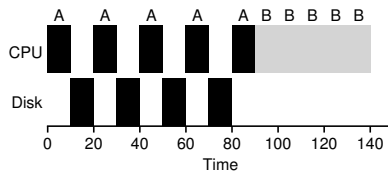


Figure 7.8: **Poor Use of Resources**

Assume we are trying to build a STCF scheduler. How should such a scheduler account for the fact that A is broken up into 5 10-ms sub-jobs, whereas B is just a single 50-ms CPU demand? Clearly, just running one job and then the other without considering how to take I/O into account makes little sense.

A common approach is to treat each 10-ms sub-job of A as an independent job. Thus, when the system starts, its choice is whether to schedule

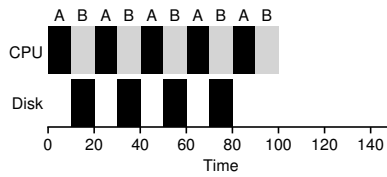


Figure 7.9: **Overlap Allows Better Use of Resources**

a 10-ms A or a 50-ms B. With STCF, the choice is clear: choose the shorter one, in this case A. Then, when the first sub-job of A has completed, only B is left, and it begins running. Then a new sub-job of A is submitted, and it preempts B and runs for 10 ms. Doing so allows for **overlap**, with the CPU being used by one process while waiting for the I/O of another process to complete; the system is thus better utilized (see Figure 7.9).

And thus we see how a scheduler might incorporate I/O. By treating each CPU burst as a job, the scheduler makes sure processes that are “interactive” get run frequently. While those interactive jobs are performing I/O, other CPU-intensive jobs run, thus better utilizing the processor.

## 7.9 No More Oracle

With a basic approach to I/O in place, we come to our final assumption: that the scheduler knows the length of each job. As we said before, this is likely the worst assumption we could make. In fact, in a general-purpose OS (like the ones we care about), the OS usually knows very little about the length of each job. Thus, how can we build an approach that behaves like SJF/STCF without such *a priori* knowledge? Further, how can we incorporate some of the ideas we have seen with the RR scheduler so that response time is also quite good?

## 7.10 Summary

We have introduced the basic ideas behind scheduling and developed two families of approaches. The first runs the shortest job remaining and thus optimizes turnaround time; the second alternates between all jobs and thus optimizes response time. Both are bad where the other is good, alas, an inherent trade-off common in systems. We have also seen how we might incorporate I/O into the picture, but have still not solved the problem of the fundamental inability of the OS to see into the future. Shortly, we will see how to overcome this problem, by building a scheduler that uses the recent past to predict the future. This scheduler is known as the **multi-level feedback queue**, and it is the topic of the next chapter.

## References

- [B+79] “The Convoy Phenomenon”  
M. Blasgen, J. Gray, M. Mitoma, T. Price  
ACM Operating Systems Review, 13:2, April 1979  
*Perhaps the first reference to convoys, which occurs in databases as well as the OS.*
- [C54] “Priority Assignment in Waiting Line Problems”  
A. Cobham  
Journal of Operations Research, 2:70, pages 70–76, 1954  
*The pioneering paper on using an SJF approach in scheduling the repair of machines.*
- [K64] “Analysis of a Time-Shared Processor”  
Leonard Kleinrock  
Naval Research Logistics Quarterly, 11:1, pages 59–73, March 1964  
*May be the first reference to the round-robin scheduling algorithm; certainly one of the first analyses of said approach to scheduling a time-shared system.*
- [CK68] “Computer Scheduling Methods and their Countermeasures”  
Edward G. Coffman and Leonard Kleinrock  
AFIPS ’68 (Spring), April 1968  
*An excellent early introduction to and analysis of a number of basic scheduling disciplines.*
- [J91] “The Art of Computer Systems Performance Analysis:  
Techniques for Experimental Design, Measurement, Simulation, and Modeling”  
R. Jain  
Interscience, New York, April 1991  
*The standard text on computer systems measurement. A great reference for your library, for sure.*
- [O45] “Animal Farm”  
George Orwell  
Secker and Warburg (London), 1945  
*A great but depressing allegorical book about power and its corruptions. Some say it is a critique of Stalin and the pre-WWII Stalin era in the U.S.S.R; we say it's a critique of pigs.*
- [PV56] “Machine Repair as a Priority Waiting-Line Problem”  
Thomas E. Phipps Jr. and W. R. Van Voorhis  
Operations Research, 4:1, pages 76–86, February 1956  
*Follow-on work that generalizes the SJF approach to machine repair from Cobham's original work; also postulates the utility of an STCF approach in such an environment. Specifically, “There are certain types of repair work, ... involving much dismantling and covering the floor with nuts and bolts, which certainly should not be interrupted once undertaken; in other cases it would be inadvisable to continue work on a long job if one or more short ones became available (p.81).”*
- [MB91] “The effect of context switches on cache performance”  
Jeffrey C. Mogul and Anita Borg  
ASPLOS, 1991  
*A nice study on how cache performance can be affected by context switching; less of an issue in today's systems where processors issue billions of instructions per second but context-switches still happen in the millisecond time range.*
- [W15] “You can't have your cake and eat it”  
[http://en.wikipedia.org/wiki/You\\_can't\\_have\\_your\\_cake\\_and\\_eat\\_it](http://en.wikipedia.org/wiki/You_can't_have_your_cake_and_eat_it)  
Wikipedia, as of December 2015  
*The best part of this page is reading all the similar idioms from other languages. In Tamil, you can't “have both the moustache and drink the soup.”*

## Homework

This program, `scheduler.py`, allows you to see how different schedulers perform under scheduling metrics such as response time, turnaround time, and total wait time. See the README for details.

## Questions

1. Compute the response time and turnaround time when running three jobs of length 200 with the SJF and FIFO schedulers.
2. Now do the same but with jobs of different lengths: 100, 200, and 300.
3. Now do the same, but also with the RR scheduler and a time-slice of 1.
4. For what types of workloads does SJF deliver the same turnaround times as FIFO?
5. For what types of workloads and quantum lengths does SJF deliver the same response times as RR?
6. What happens to response time with SJF as job lengths increase? Can you use the simulator to demonstrate the trend?
7. What happens to response time with RR as quantum lengths increase? Can you write an equation that gives the worst-case response time, given  $N$  jobs?

## Scheduling: The Multi-Level Feedback Queue

In this chapter, we'll tackle the problem of developing one of the most well-known approaches to scheduling, known as the **Multi-level Feedback Queue (MLFQ)**. The Multi-level Feedback Queue (MLFQ) scheduler was first described by Corbato et al. in 1962 [C+62] in a system known as the Compatible Time-Sharing System (CTSS), and this work, along with later work on Multics, led the ACM to award Corbato its highest honor, the **Turing Award**. The scheduler has subsequently been refined throughout the years to the implementations you will encounter in some modern systems.

The fundamental problem MLFQ tries to address is two-fold. First, it would like to optimize *turnaround time*, which, as we saw in the previous note, is done by running shorter jobs first; unfortunately, the OS doesn't generally know how long a job will run for, exactly the knowledge that algorithms like SJF (or STCF) require. Second, MLFQ would like to make a system feel responsive to interactive users (i.e., users sitting and staring at the screen, waiting for a process to finish), and thus minimize *response time*; unfortunately, algorithms like Round Robin reduce response time but are terrible for turnaround time. Thus, our problem: given that we in general do not know anything about a process, how can we build a scheduler to achieve these goals? How can the scheduler learn, as the system runs, the characteristics of the jobs it is running, and thus make better scheduling decisions?

### THE CRUX:

#### HOW TO SCHEDULE WITHOUT PERFECT KNOWLEDGE?

How can we design a scheduler that both minimizes response time for interactive jobs while also minimizing turnaround time without *a priori* knowledge of job length?

## TIP: LEARN FROM HISTORY

The multi-level feedback queue is an excellent example of a system that learns from the past to predict the future. Such approaches are common in operating systems (and many other places in Computer Science, including hardware branch predictors and caching algorithms). Such approaches work when jobs have phases of behavior and are thus predictable; of course, one must be careful with such techniques, as they can easily be wrong and drive a system to make worse decisions than they would have with no knowledge at all.

## 8.1 MLFQ: Basic Rules

To build such a scheduler, in this chapter we will describe the basic algorithms behind a multi-level feedback queue; although the specifics of many implemented MLFQs differ [E95], most approaches are similar.

In our treatment, the MLFQ has a number of distinct **queues**, each assigned a different **priority level**. At any given time, a job that is ready to run is on a single queue. MLFQ uses priorities to decide which job should run at a given time: a job with higher priority (i.e., a job on a higher queue) is chosen to run.

Of course, more than one job may be on a given queue, and thus have the *same* priority. In this case, we will just use round-robin scheduling among those jobs.

Thus, the key to MLFQ scheduling lies in how the scheduler sets priorities. Rather than giving a fixed priority to each job, MLFQ *varies* the priority of a job based on its *observed behavior*. If, for example, a job repeatedly relinquishes the CPU while waiting for input from the keyboard, MLFQ will keep its priority high, as this is how an interactive process might behave. If, instead, a job uses the CPU intensively for long periods of time, MLFQ will reduce its priority. In this way, MLFQ will try to *learn* about processes as they run, and thus use the *history* of the job to predict its *future* behavior.

Thus, we arrive at the first two basic rules for MLFQ:

- **Rule 1:** If  $\text{Priority}(A) > \text{Priority}(B)$ , A runs (B doesn't).
- **Rule 2:** If  $\text{Priority}(A) = \text{Priority}(B)$ , A & B run in RR.

If we were to put forth a picture of what the queues might look like at a given instant, we might see something like the following (Figure 8.1). In the figure, two jobs (A and B) are at the highest priority level, while job C is in the middle and Job D is at the lowest priority. Given our current knowledge of how MLFQ works, the scheduler would just alternate time slices between A and B because they are the highest priority jobs in the system; poor jobs C and D would never even get to run — an outrage!

Of course, just showing a static snapshot of some queues does not really give you an idea of how MLFQ works. What we need is to under-



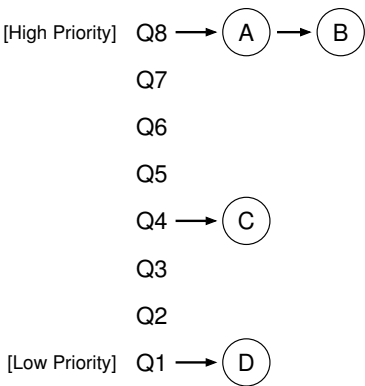


Figure 8.1: MLFQ Example

stand how job priority *changes* over time. And that, in a surprise only to those who are reading a chapter from this book for the first time, is exactly what we will do next.

## 8.2 Attempt #1: How To Change Priority

We now must decide how MLFQ is going to change the priority level of a job (and thus which queue it is on) over the lifetime of a job. To do this, we must keep in mind our workload: a mix of interactive jobs that are short-running (and may frequently relinquish the CPU), and some longer-running “CPU-bound” jobs that need a lot of CPU time but where response time isn’t important. Here is our first attempt at a priority-adjustment algorithm:

- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).
- **Rule 4a:** If a job uses up an entire time slice while running, its priority is *reduced* (i.e., it moves down one queue).
- **Rule 4b:** If a job gives up the CPU before the time slice is up, it stays at the *same* priority level.

### Example 1: A Single Long-Running Job

Let’s look at some examples. First, we’ll look at what happens when there has been a long running job in the system. Figure 8.2 shows what happens to this job over time in a three-queue scheduler.

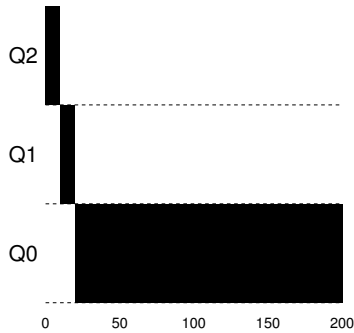


Figure 8.2: Long-running Job Over Time

As you can see in the example, the job enters at the highest priority (Q2). After a single time-slice of 10 ms, the scheduler reduces the job’s priority by one, and thus the job is on Q1. After running at Q1 for a time slice, the job is finally lowered to the lowest priority in the system (Q0), where it remains. Pretty simple, no?

**Example 2: Along Came A Short Job**

Now let’s look at a more complicated example, and hopefully see how MLFQ tries to approximate SJF. In this example, there are two jobs: A, which is a long-running CPU-intensive job, and B, which is a short-running interactive job. Assume A has been running for some time, and then B arrives. What will happen? Will MLFQ approximate SJF for B?

Figure 8.3 plots the results of this scenario. A (shown in black) is running along in the lowest-priority queue (as would any long-running CPU-intensive jobs); B (shown in gray) arrives at time  $T = 100$ , and thus is

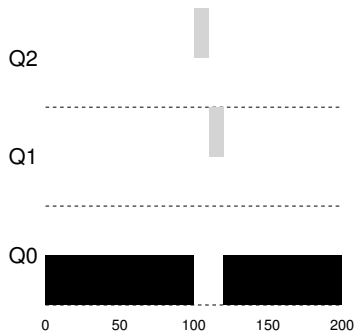


Figure 8.3: Along Came An Interactive Job

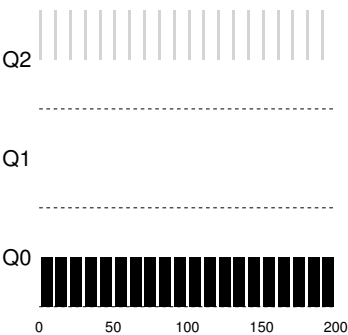


Figure 8.4: A Mixed I/O-intensive and CPU-intensive Workload

inserted into the highest queue; as its run-time is short (only 20 ms), B completes before reaching the bottom queue, in two time slices; then A resumes running (at low priority).

From this example, you can hopefully understand one of the major goals of the algorithm: because it doesn't *know* whether a job will be a short job or a long-running job, it first *assumes* it might be a short job, thus giving the job high priority. If it actually is a short job, it will run quickly and complete; if it is not a short job, it will slowly move down the queues, and thus soon prove itself to be a long-running more batch-like process. In this manner, MLFQ approximates SJF.

### Example 3: What About I/O?

Let's now look at an example with some I/O. As Rule 4b states above, if a process gives up the processor before using up its time slice, we keep it at the same priority level. The intent of this rule is simple: if an interactive job, for example, is doing a lot of I/O (say by waiting for user input from the keyboard or mouse), it will relinquish the CPU before its time slice is complete; in such case, we don't wish to penalize the job and thus simply keep it at the same level.

Figure 8.4 shows an example of how this works, with an interactive job B (shown in gray) that needs the CPU only for 1 ms before performing an I/O competing for the CPU with a long-running batch job A (shown in black). The MLFQ approach keeps B at the highest priority because B keeps releasing the CPU; if B is an interactive job, MLFQ further achieves its goal of running interactive jobs quickly.

### Problems With Our Current MLFQ

We thus have a basic MLFQ. It seems to do a fairly good job, sharing the CPU fairly between long-running jobs, and letting short or I/O-intensive interactive jobs run quickly. Unfortunately, the approach we have developed thus far contains serious flaws. Can you think of any?

*(This is where you pause and think as deviously as you can)*

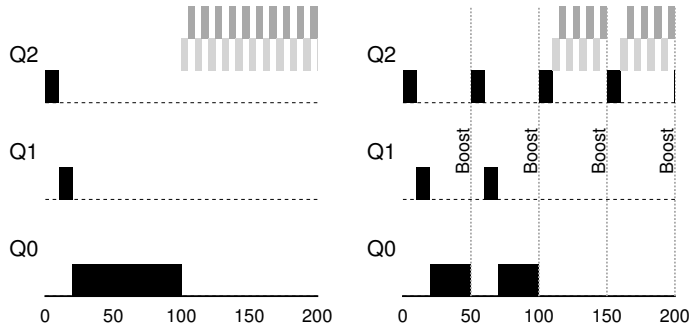


Figure 8.5: Without (Left) and With (Right) Priority Boost

First, there is the problem of **starvation**: if there are “too many” interactive jobs in the system, they will combine to consume *all* CPU time, and thus long-running jobs will *never* receive any CPU time (they **starve**). We’d like to make some progress on these jobs even in this scenario.

Second, a smart user could rewrite their program to **game the scheduler**. Gaming the scheduler generally refers to the idea of doing something sneaky to trick the scheduler into giving you more than your fair share of the resource. The algorithm we have described is susceptible to the following attack: before the time slice is over, issue an I/O operation (to some file you don’t care about) and thus relinquish the CPU; doing so allows you to remain in the same queue, and thus gain a higher percentage of CPU time. When done right (e.g., by running for 99% of a time slice before relinquishing the CPU), a job could nearly monopolize the CPU.

Finally, a program may *change its behavior* over time; what was CPU-bound may transition to a phase of interactivity. With our current approach, such a job would be out of luck and not be treated like the other interactive jobs in the system.

### 8.3 Attempt #2: The Priority Boost

Let’s try to change the rules and see if we can avoid the problem of starvation. What could we do in order to guarantee that CPU-bound jobs will make some progress (even if it is not much?).

The simple idea here is to periodically **boost** the priority of all the jobs in system. There are many ways to achieve this, but let’s just do something simple: throw them all in the topmost queue; hence, a new rule:

- **Rule 5:** After some time period  $S$ , move all the jobs in the system to the topmost queue.

Our new rule solves two problems at once. First, processes are guaranteed not to starve: by sitting in the top queue, a job will share the CPU

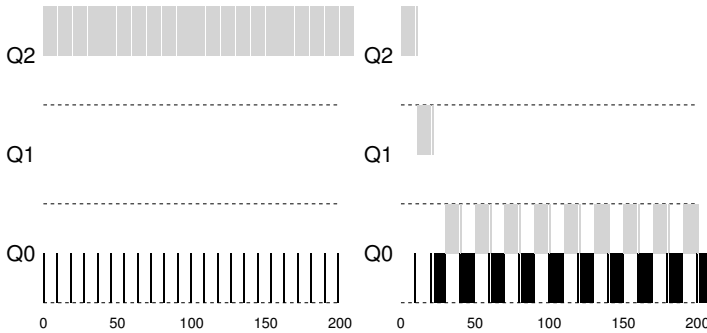


Figure 8.6: Without (Left) and With (Right) Gaming Tolerance

with other high-priority jobs in a round-robin fashion, and thus eventually receive service. Second, if a CPU-bound job has become interactive, the scheduler treats it properly once it has received the priority boost.

Let's see an example. In this scenario, we just show the behavior of a long-running job when competing for the CPU with two short-running interactive jobs. Two graphs are shown in Figure 8.5 (page 6). On the left, there is no priority boost, and thus the long-running job gets starved once the two short jobs arrive; on the right, there is a priority boost every 50 ms (which is likely too small of a value, but used here for the example), and thus we at least guarantee that the long-running job will make some progress, getting boosted to the highest priority every 50 ms and thus getting to run periodically.

Of course, the addition of the time period  $S$  leads to the obvious question: what should  $S$  be set to? John Ousterhout, a well-regarded systems researcher [O11], used to call such values in systems **voo-doo constants**, because they seemed to require some form of black magic to set them correctly. Unfortunately,  $S$  has that flavor. If it is set too high, long-running jobs could starve; too low, and interactive jobs may not get a proper share of the CPU.

## 8.4 Attempt #3: Better Accounting

We now have one more problem to solve: how to prevent gaming of our scheduler? The real culprit here, as you might have guessed, are Rules 4a and 4b, which let a job retain its priority by relinquishing the CPU before the time slice expires. So what should we do?

The solution here is to perform better **accounting** of CPU time at each level of the MLFQ. Instead of forgetting how much of a time slice a process used at a given level, the scheduler should keep track; once a process has used its allotment, it is demoted to the next priority queue. Whether

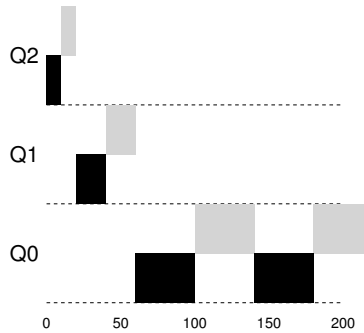


Figure 8.7: Lower Priority, Longer Quanta

it uses the time slice in one long burst or many small ones does not matter. We thus rewrite Rules 4a and 4b to the following single rule:

- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).

Let's look at an example. Figure 8.6 (page 7) shows what happens when a workload tries to game the scheduler with the old Rules 4a and 4b (on the left) as well the new anti-gaming Rule 4. Without any protection from gaming, a process can issue an I/O just before a time slice ends and thus dominate CPU time. With such protections in place, regardless of the I/O behavior of the process, it slowly moves down the queues, and thus cannot gain an unfair share of the CPU.

## 8.5 Tuning MLFQ And Other Issues

A few other issues arise with MLFQ scheduling. One big question is how to **parameterize** such a scheduler. For example, how many queues should there be? How big should the time slice be per queue? How often should priority be boosted in order to avoid starvation and account for changes in behavior? There are no easy answers to these questions, and thus only some experience with workloads and subsequent tuning of the scheduler will lead to a satisfactory balance.

For example, most MLFQ variants allow for varying time-slice length across different queues. The high-priority queues are usually given short time slices; they are comprised of interactive jobs, after all, and thus quickly alternating between them makes sense (e.g., 10 or fewer milliseconds). The low-priority queues, in contrast, contain long-running jobs that are CPU-bound; hence, longer time slices work well (e.g., 100s of ms). Figure 8.7 shows an example in which two long-running jobs run for 10 ms at the highest queue, 20 in the middle, and 40 at the lowest.

**TIP: AVOID VOO-DOO CONSTANTS (OUSTERHOUT'S LAW)**

Avoiding voo-doo constants is a good idea whenever possible. Unfortunately, as in the example above, it is often difficult. One could try to make the system learn a good value, but that too is not straightforward. The frequent result: a configuration file filled with default parameter values that a seasoned administrator can tweak when something isn't quite working correctly. As you can imagine, these are often left unmodified, and thus we are left to hope that the defaults work well in the field. This tip brought to you by our old OS professor, John Ousterhout, and hence we call it **Ousterhout's Law**.

The Solaris MLFQ implementation — the Time-Sharing scheduling class, or TS — is particularly easy to configure; it provides a set of tables that determine exactly how the priority of a process is altered throughout its lifetime, how long each time slice is, and how often to boost the priority of a job [AD00]; an administrator can muck with this table in order to make the scheduler behave in different ways. Default values for the table are 60 queues, with slowly increasing time-slice lengths from 20 milliseconds (highest priority) to a few hundred milliseconds (lowest), and priorities boosted around every 1 second or so.

Other MLFQ schedulers don't use a table or the exact rules described in this chapter; rather they adjust priorities using mathematical formulae. For example, the FreeBSD scheduler (version 4.3) uses a formula to calculate the current priority level of a job, basing it on how much CPU the process has used [LM+89]; in addition, usage is decayed over time, providing the desired priority boost in a different manner than described herein. See Epema's paper for an excellent overview of such **decay-usage** algorithms and their properties [E95].

Finally, many schedulers have a few other features that you might encounter. For example, some schedulers reserve the highest priority levels for operating system work; thus typical user jobs can never obtain the highest levels of priority in the system. Some systems also allow some user **advice** to help set priorities; for example, by using the command-line utility `nice` you can increase or decrease the priority of a job (somewhat) and thus increase or decrease its chances of running at any given time. See the man page for more.

## 8.6 MLFQ: Summary

We have described a scheduling approach known as the Multi-Level Feedback Queue (MLFQ). Hopefully you can now see why it is called that: it has *multiple levels* of queues, and uses *feedback* to determine the priority of a given job. History is its guide: pay attention to how jobs behave over time and treat them accordingly.

## TIP: USE ADVICE WHERE POSSIBLE

As the operating system rarely knows what is best for each and every process of the system, it is often useful to provide interfaces to allow users or administrators to provide some **hints** to the OS. We often call such hints **advice**, as the OS need not necessarily pay attention to it, but rather might take the advice into account in order to make a better decision. Such hints are useful in many parts of the OS, including the scheduler (e.g., with `nice`), memory manager (e.g., `madvise`), and file system (e.g., informed prefetching and caching [P+95]).

The refined set of MLFQ rules, spread throughout the chapter, are reproduced here for your viewing pleasure:

- **Rule 1:** If  $\text{Priority}(A) > \text{Priority}(B)$ , A runs (B doesn't).
- **Rule 2:** If  $\text{Priority}(A) = \text{Priority}(B)$ , A & B run in RR.
- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).
- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- **Rule 5:** After some time period  $S$ , move all the jobs in the system to the topmost queue.

MLFQ is interesting for the following reason: instead of demanding *a priori* knowledge of the nature of a job, it observes the execution of a job and prioritizes it accordingly. In this way, it manages to achieve the best of both worlds: it can deliver excellent overall performance (similar to SJF/STCF) for short-running interactive jobs, and is fair and makes progress for long-running CPU-intensive workloads. For this reason, many systems, including BSD UNIX derivatives [LM+89, B86], Solaris [M06], and Windows NT and subsequent Windows operating systems [CS97] use a form of MLFQ as their base scheduler.



## References

- [AD00] "Multilevel Feedback Queue Scheduling in Solaris"  
Andrea Arpaci-Dusseau  
Available: <http://www.cs.wisc.edu/~remzi/solaris-notes.pdf>  
*A great short set of notes by one of the authors on the details of the Solaris scheduler. OK, we are probably biased in this description, but the notes are pretty darn good.*
- [B86] "The Design of the UNIX Operating System"  
M.J. Bach  
Prentice-Hall, 1986  
*One of the classic old books on how a real UNIX operating system is built; a definite must-read for kernel hackers.*
- [C+62] "An Experimental Time-Sharing System"  
F. J. Corbato, M. M. Daggett, R. C. Daley  
IFIPS 1962  
*A bit hard to read, but the source of many of the first ideas in multi-level feedback scheduling. Much of this later went into Multics, which one could argue was the most influential operating system of all time.*
- [CS97] "Inside Windows NT"  
Helen Custer and David A. Solomon  
Microsoft Press, 1997  
*The NT book, if you want to learn about something other than UNIX. Of course, why would you? OK, we're kidding; you might actually work for Microsoft some day you know.*
- [E95] "An Analysis of Decay-Usage Scheduling in Multiprocessors"  
D.H.J. Epema  
SIGMETRICS '95  
*A nice paper on the state of the art of scheduling back in the mid 1990s, including a good overview of the basic approach behind decay-usage schedulers.*
- [LM+89] "The Design and Implementation of the 4.3BSD UNIX Operating System"  
S.J. Leffler, M.K. McKusick, M.J. Karels, J.S. Quarterman  
Addison-Wesley, 1989  
*Another OS classic, written by four of the main people behind BSD. The later versions of this book, while more up to date, don't quite match the beauty of this one.*
- [M06] "Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture"  
Richard McDougall  
Prentice-Hall, 2006  
*A good book about Solaris and how it works.*
- [O11] "John Ousterhout's Home Page"  
John Ousterhout  
Available: <http://www.stanford.edu/~ouster/>  
*The home page of the famous Professor Ousterhout. The two co-authors of this book had the pleasure of taking graduate operating systems from Ousterhout while in graduate school; indeed, this is where the two co-authors got to know each other, eventually leading to marriage, kids, and even this book. Thus, you really can blame Ousterhout for this entire mess you're in.*
- [P+95] "Informed Prefetching and Caching"  
R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, J. Zelenka  
SOSP '95  
*A fun paper about some very cool ideas in file systems, including how applications can give the OS advice about what files it is accessing and how it plans to access them.*

## Homework

This program, `mlfq.py`, allows you to see how the MLFQ scheduler presented in this chapter behaves. See the README for details.

## Questions

1. Run a few randomly-generated problems with just two jobs and two queues; compute the MLFQ execution trace for each. Make your life easier by limiting the length of each job and turning off I/Os.
2. How would you run the scheduler to reproduce each of the examples in the chapter?
3. How would you configure the scheduler parameters to behave just like a round-robin scheduler?
4. Craft a workload with two jobs and scheduler parameters so that one job takes advantage of the older Rules 4a and 4b (turned on with the `-S` flag) to game the scheduler and obtain 99% of the CPU over a particular time interval.
5. Given a system with a quantum length of 10 ms in its highest queue, how often would you have to boost jobs back to the highest priority level (with the `-B` flag) in order to guarantee that a single long-running (and potentially-starving) job gets at least 5% of the CPU?
6. One question that arises in scheduling is which end of a queue to add a job that just finished I/O; the `-I` flag changes this behavior for this scheduling simulator. Play around with some workloads and see if you can see the effect of this flag.

## Scheduling: Proportional Share

In this chapter, we'll examine a different type of scheduler known as a **proportional-share** scheduler, also sometimes referred to as a **fair-share** scheduler. Proportional-share is based around a simple concept: instead of optimizing for turnaround or response time, a scheduler might instead try to guarantee that each job obtain a certain percentage of CPU time.

An excellent modern example of proportional-share scheduling is found in research by Waldspurger and Weihl [WW94], and is known as **lottery scheduling**; however, the idea is certainly much older [KL88]. The basic idea is quite simple: every so often, hold a lottery to determine which process should get to run next; processes that should run more often should be given more chances to win the lottery. Easy, no? Now, onto the details! But not before our crux:

### CRUX: HOW TO SHARE THE CPU PROPORTIONALLY

How can we design a scheduler to share the CPU in a proportional manner? What are the key mechanisms for doing so? How effective are they?

## 9.1 Basic Concept: Tickets Represent Your Share

Underlying lottery scheduling is one very basic concept: **tickets**, which are used to represent the share of a resource that a process (or user or whatever) should receive. The percent of tickets that a process has represents its share of the system resource in question.

Let's look at an example. Imagine two processes, A and B, and further that A has 75 tickets while B has only 25. Thus, what we would like is for A to receive 75% of the CPU and B the remaining 25%.

Lottery scheduling achieves this probabilistically (but not deterministically) by holding a lottery every so often (say, every time slice). Holding a lottery is straightforward: the scheduler must know how many total tickets there are (in our example, there are 100). The scheduler then picks

TIP: USE RANDOMNESS

One of the most beautiful aspects of lottery scheduling is its use of **randomness**. When you have to make a decision, using such a randomized approach is often a robust and simple way of doing so.

Random approaches has at least three advantages over more traditional decisions. First, random often avoids strange corner-case behaviors that a more traditional algorithm may have trouble handling. For example, consider the LRU replacement policy (studied in more detail in a future chapter on virtual memory); while often a good replacement algorithm, LRU performs pessimally for some cyclic-sequential workloads. Random, on the other hand, has no such worst case.

Second, random also is lightweight, requiring little state to track alternatives. In a traditional fair-share scheduling algorithm, tracking how much CPU each process has received requires per-process accounting, which must be updated after running each process. Doing so randomly necessitates only the most minimal of per-process state (e.g., the number of tickets each has).

Finally, random can be quite fast. As long as generating a random number is quick, making the decision is also, and thus random can be used in a number of places where speed is required. Of course, the faster the need, the more random tends towards pseudo-random.

a winning ticket, which is a number from 0 to 99<sup>1</sup>. Assuming A holds tickets 0 through 74 and B 75 through 99, the winning ticket simply determines whether A or B runs. The scheduler then loads the state of that winning process and runs it.

Here is an example output of a lottery scheduler’s winning tickets:

63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43 0 49 49

Here is the resulting schedule:

A        A   A        A   A   A   A   A        A        A   A   A   A   A   A  
      B                B                                B        B

As you can see from the example, the use of randomness in lottery scheduling leads to a probabilistic correctness in meeting the desired proportion, but no guarantee. In our example above, B only gets to run 4 out of 20 time slices (20%), instead of the desired 25% allocation. However, the longer these two jobs compete, the more likely they are to achieve the desired percentages.

<sup>1</sup>Computer Scientists always start counting at 0. It is so odd to non-computer-types that famous people have felt obliged to write about why we do it this way [D82].

## TIP: USE TICKETS TO REPRESENT SHARES

One of the most powerful (and basic) mechanisms in the design of lottery (and stride) scheduling is that of the **ticket**. The ticket is used to represent a process's share of the CPU in these examples, but can be applied much more broadly. For example, in more recent work on virtual memory management for hypervisors, Waldspurger shows how tickets can be used to represent a guest operating system's share of memory [W02]. Thus, if you are ever in need of a mechanism to represent a proportion of ownership, this concept just might be ... (wait for it) ... the ticket.

## 9.2 Ticket Mechanisms

Lottery scheduling also provides a number of mechanisms to manipulate tickets in different and sometimes useful ways. One way is with the concept of **ticket currency**. Currency allows a user with a set of tickets to allocate tickets among their own jobs in whatever currency they would like; the system then automatically converts said currency into the correct global value.

For example, assume users A and B have each been given 100 tickets. User A is running two jobs, A1 and A2, and gives them each 500 tickets (out of 1000 total) in User A's own currency. User B is running only 1 job and gives it 10 tickets (out of 10 total). The system will convert A1's and A2's allocation from 500 each in A's currency to 50 each in the global currency; similarly, B1's 10 tickets will be converted to 100 tickets. The lottery will then be held over the global ticket currency (200 total) to determine which job runs.

```
User A -> 500 (A's currency) to A1 -> 50 (global currency)
        -> 500 (A's currency) to A2 -> 50 (global currency)
User B -> 10 (B's currency) to B1 -> 100 (global currency)
```

Another useful mechanism is **ticket transfer**. With transfers, a process can temporarily hand off its tickets to another process. This ability is especially useful in a client/server setting, where a client process sends a message to a server asking it to do some work on the client's behalf. To speed up the work, the client can pass the tickets to the server and thus try to maximize the performance of the server while the server is handling the client's request. When finished, the server then transfers the tickets back to the client and all is as before.

Finally, **ticket inflation** can sometimes be a useful technique. With inflation, a process can temporarily raise or lower the number of tickets it owns. Of course, in a competitive scenario with processes that do not trust one another, this makes little sense; one greedy process could give itself a vast number of tickets and take over the machine. Rather, inflation can be applied in an environment where a group of processes trust one another; in such a case, if any one process knows it needs more CPU time, it can boost its ticket value as a way to reflect that need to the system, all without communicating with any other processes.

```

1 // counter: used to track if we've found the winner yet
2 int counter = 0;
3
4 // winner: use some call to a random number generator to
5 //         get a value, between 0 and the total # of tickets
6 int winner = getRandom(0, totaltickets);
7
8 // current: use this to walk through the list of jobs
9 node_t *current = head;
10
11 // loop until the sum of ticket values is > the winner
12 while (current) {
13     counter = counter + current->tickets;
14     if (counter > winner)
15         break; // found the winner
16     current = current->next;
17 }
18 // 'current' is the winner: schedule it...

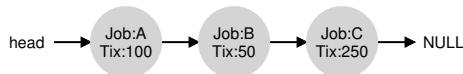
```

Figure 9.1: Lottery Scheduling Decision Code

### 9.3 Implementation

Probably the most amazing thing about lottery scheduling is the simplicity of its implementation. All you need is a good random number generator to pick the winning ticket, a data structure to track the processes of the system (e.g., a list), and the total number of tickets.

Let's assume we keep the processes in a list. Here is an example comprised of three processes, A, B, and C, each with some number of tickets.



To make a scheduling decision, we first have to pick a random number (the winner) from the total number of tickets (400)<sup>2</sup>. Let's say we pick the number 300. Then, we simply traverse the list, with a simple counter used to help us find the winner (Figure 9.1).

The code walks the list of processes, adding each ticket value to `counter` until the value exceeds `winner`. Once that is the case, the current list element is the winner. With our example of the winning ticket being 300, the following takes place. First, `counter` is incremented to 100 to account for A's tickets; because 100 is less than 300, the loop continues. Then `counter` would be updated to 150 (B's tickets), still less than 300 and thus again we continue. Finally, `counter` is updated to 400 (clearly greater than 300), and thus we break out of the loop with `current` pointing at C (the winner).

To make this process most efficient, it might generally be best to organize the list in sorted order, from the highest number of tickets to the

<sup>2</sup>Surprisingly, as pointed out by Björn Lindberg, this can be challenging to do correctly; for more details, see <http://stackoverflow.com/questions/2509679/how-to-generate-a-random-number-from-within-a-range>.

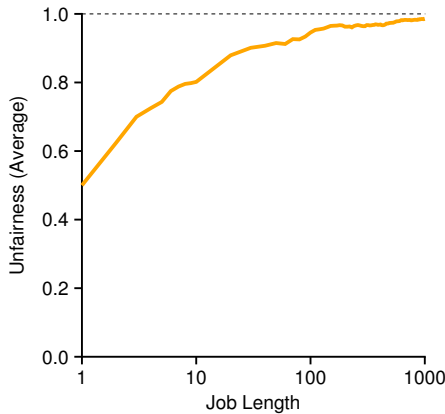


Figure 9.2: Lottery Fairness Study

lowest. The ordering does not affect the correctness of the algorithm; however, it does ensure in general that the fewest number of list iterations are taken, especially if there are a few processes that possess most of the tickets.

## 9.4 An Example

To make the dynamics of lottery scheduling more understandable, we now perform a brief study of the completion time of two jobs competing against one another, each with the same number of tickets (100) and same run time ( $R$ , which we will vary).

In this scenario, we'd like for each job to finish at roughly the same time, but due to the randomness of lottery scheduling, sometimes one job finishes before the other. To quantify this difference, we define a simple **unfairness metric**,  $U$  which is simply the time the first job completes divided by the time that the second job completes. For example, if  $R = 10$ , and the first job finishes at time 10 (and the second job at 20),  $U = \frac{10}{20} = 0.5$ . When both jobs finish at nearly the same time,  $U$  will be quite close to 1. In this scenario, that is our goal: a perfectly fair scheduler would achieve  $U = 1$ .

Figure 9.2 plots the average unfairness as the length of the two jobs ( $R$ ) is varied from 1 to 1000 over thirty trials (results are generated via the simulator provided at the end of the chapter). As you can see from the graph, when the job length is not very long, average unfairness can be quite severe. Only as the jobs run for a significant number of time slices does the lottery scheduler approach the desired outcome.

## 9.5 How To Assign Tickets?

One problem we have not addressed with lottery scheduling is: how to assign tickets to jobs? This problem is a tough one, because of course how the system behaves is strongly dependent on how tickets are allocated. One approach is to assume that the users know best; in such a case, each user is handed some number of tickets, and a user can allocate tickets to any jobs they run as desired. However, this solution is a non-solution: it really doesn't tell you what to do. Thus, given a set of jobs, the "ticket-assignment problem" remains open.

## 9.6 Why Not Deterministic?

You might also be wondering: why use randomness at all? As we saw above, while randomness gets us a simple (and approximately correct) scheduler, it occasionally will not deliver the exact right proportions, especially over short time scales. For this reason, Waldspurger invented **stride scheduling**, a deterministic fair-share scheduler [W95].

Stride scheduling is also straightforward. Each job in the system has a stride, which is inverse in proportion to the number of tickets it has. In our example above, with jobs A, B, and C, with 100, 50, and 250 tickets, respectively, we can compute the stride of each by dividing some large number by the number of tickets each process has been assigned. For example, if we divide 10,000 by each of those ticket values, we obtain the following stride values for A, B, and C: 100, 200, and 40. We call this value the **stride** of each process; every time a process runs, we will increment a counter for it (called its **pass** value) by its stride to track its global progress.

The scheduler then uses the stride and pass to determine which process should run next. The basic idea is simple: at any given time, pick the process to run that has the lowest pass value so far; when you run a process, increment its pass counter by its stride. A pseudocode implementation is provided by Waldspurger [W95]:

```
current = remove_min(queue);           // pick client with minimum pass
schedule(current);                     // use resource for quantum
current->pass += current->stride;        // compute next pass using stride
insert(queue, current);                // put back into the queue
```

In our example, we start with three processes (A, B, and C), with stride values of 100, 200, and 40, and all with pass values initially at 0. Thus, at first, any of the processes might run, as their pass values are equally low. Assume we pick A (arbitrarily; any of the processes with equal low pass values can be chosen). A runs; when finished with the time slice, we update its pass value to 100. Then we run B, whose pass value is then set to 200. Finally, we run C, whose pass value is incremented to 40. At this point, the algorithm will pick the lowest pass value, which is C's, and run it, updating its pass to 80 (C's stride is 40, as you recall). Then C will



Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	...

Figure 9.3: Stride Scheduling: A Trace

run again (still the lowest pass value), raising its pass to 120. A will run now, updating its pass to 200 (now equal to B’s). Then C will run twice more, updating its pass to 160 then 200. At this point, all pass values are equal again, and the process will repeat, ad infinitum. Figure 9.3 traces the behavior of the scheduler over time.

As we can see from the figure, C ran five times, A twice, and B just once, exactly in proportion to their ticket values of 250, 100, and 50. Lottery scheduling achieves the proportions probabilistically over time; stride scheduling gets them exactly right at the end of each scheduling cycle.

So you might be wondering: given the precision of stride scheduling, why use lottery scheduling at all? Well, lottery scheduling has one nice property that stride scheduling does not: no global state. Imagine a new job enters in the middle of our stride scheduling example above; what should its pass value be? Should it be set to 0? If so, it will monopolize the CPU. With lottery scheduling, there is no global state per process; we simply add a new process with whatever tickets it has, update the single global variable to track how many total tickets we have, and go from there. In this way, lottery makes it much easier to incorporate new processes in a sensible manner.

9.7 Summary

We have introduced the concept of proportional-share scheduling and briefly discussed two implementations: lottery and stride scheduling. Lottery uses randomness in a clever way to achieve proportional share; stride does so deterministically. Although both are conceptually interesting, they have not achieved wide-spread adoption as CPU schedulers for a variety of reasons. One is that such approaches do not particularly mesh well with I/O [AC97]; another is that they leave open the hard problem of ticket assignment, i.e., how do you know how many tickets your browser should be allocated? General-purpose schedulers (such as the MLFQ we discussed previously, and other similar Linux schedulers) do so more gracefully and thus are more widely deployed.

As a result, proportional-share schedulers are more useful in domains where some of these problems (such as assignment of shares) are relatively easy to solve. For example, in a **virtualized** data center, where you might like to assign one-quarter of your CPU cycles to the Windows VM and the rest to your base Linux installation, proportional sharing can be simple and effective. See Waldspurger [W02] for further details on how such a scheme is used to proportionally share memory in VMWare's ESX Server.

## References

[AC97] “Extending Proportional-Share Scheduling to a Network of Workstations”

Andrea C. Arpaci-Dusseau and David E. Culler

PDPTA’97, June 1997

*A paper by one of the authors on how to extend proportional-share scheduling to work better in a clustered environment.*

[D82] “Why Numbering Should Start At Zero”

Edsger Dijkstra, August 1982

<http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD831.PDF>

*A short note from E. Dijkstra, one of the pioneers of computer science. We’ll be hearing much more on this guy in the section on Concurrency. In the meanwhile, enjoy this note, which includes this motivating quote: “One of my colleagues — not a computing scientist — accused a number of younger computing scientists of ‘pedantry’ because they started numbering at zero.” The note explains why doing so is logical.*

[KL88] “A Fair Share Scheduler”

J. Kay and P. Lauder

CACM, Volume 31 Issue 1, January 1988

*An early reference to a fair-share scheduler.*

[WW94] “Lottery Scheduling: Flexible Proportional-Share Resource Management”

Carl A. Waldspurger and William E. Weihl

OSDI ’94, November 1994

*The landmark paper on lottery scheduling that got the systems community re-energized about scheduling, fair sharing, and the power of simple randomized algorithms.*

[W95] “Lottery and Stride Scheduling: Flexible

Proportional-Share Resource Management”

Carl A. Waldspurger

Ph.D. Thesis, MIT, 1995

*The award-winning thesis of Waldspurger’s that outlines lottery and stride scheduling. If you’re thinking of writing a Ph.D. dissertation at some point, you should always have a good example around, to give you something to strive for: this is such a good one.*

[W02] “Memory Resource Management in VMware ESX Server”

Carl A. Waldspurger

OSDI ’02, Boston, Massachusetts

*The paper to read about memory management in VMMs (a.k.a., hypervisors). In addition to being relatively easy to read, the paper contains numerous cool ideas about this new type of VMM-level memory management.*

## Homework

This program, `lottery.py`, allows you to see how a lottery scheduler works. See the README for details.

## Questions

1. Compute the solutions for simulations with 3 jobs and random seeds of 1, 2, and 3.
2. Now run with two specific jobs: each of length 10, but one (job 0) with just 1 ticket and the other (job 1) with 100 (e.g., `-1 10:1, 10:100`). What happens when the number of tickets is so imbalanced? Will job 0 ever run before job 1 completes? How often? In general, what does such a ticket imbalance do to the behavior of lottery scheduling?
3. When running with two jobs of length 100 and equal ticket allocations of 100 (`-1 100:100, 100:100`), how unfair is the scheduler? Run with some different random seeds to determine the (probabilistic) answer; let unfairness be determined by how much earlier one job finishes than the other.
4. How does your answer to the previous question change as the quantum size (`-q`) gets larger?
5. Can you make a version of the graph that is found in the chapter? What else would be worth exploring? How would the graph look with a stride scheduler?