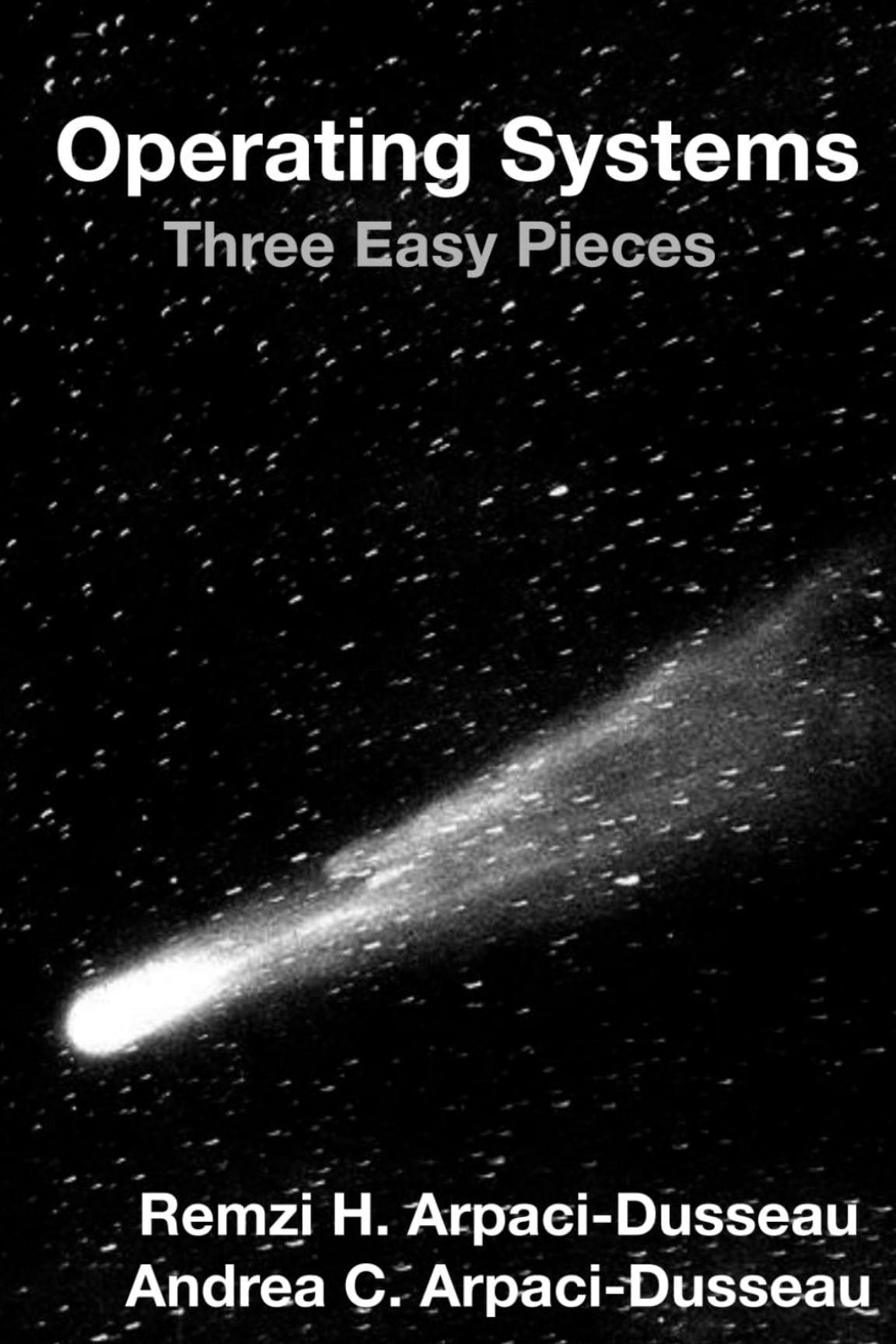


A black and white photograph of a comet streaking across a dark, starry sky. The comet's bright, glowing head is in the lower-left corner, and its long, diffuse tail extends diagonally towards the upper-right corner. The background is filled with numerous small, distant stars.

Operating Systems

Three Easy Pieces

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

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** (available at <http://www.ostep.org>), 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 for¹.

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

¹Hint: 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. We are also making code available at <https://github.com/remzi-arpacidusseau/ostep-code> for your viewing pleasure.

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.

We are slowly making project descriptions, and a testing framework, available. See <https://github.com/remzi-arpacidusseau/ostep-projects> for more information. If not part of a class, this will give you a chance to do these projects on your own, to better learn the material. Unfortunately, you don't have a TA to bug when you get stuck, but not everything in life can be free (but books can be!).

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. 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

August, 2018 (Version 1.00)

<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 20 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, and others have contributed their materials as well.

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 online². 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 winners³), 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.

²A 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.

³The 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: Aaron Gember (Colgate), Aashrith H Govindraj (USF), Abhinav Mehra, Abhirami Senthilkumaran*, Adam Drescher* (WUSTL), Adam Eggum, Aditya Venkataraman, Adriana Iamnitchi and class (USF), Ahmad Jarara, Ahmed Fikri*, Ajaykrishna Raghavan, Akiel Khan, Alex Curtis, Alex Wyler, Alex Zhao (U. Colorado at Colorado Springs), Ali Razeen (Duke), Alistair Martin, AmirBehzad Eslami, Anand Mundada, Andrew Mahler, Andrew Valencik (Saint Mary's), Angela Demke Brown (Toronto), Antonella Bernobich (UoPeople)*, Arek Bulski, B. Brahmananda Reddy (Minnesota), Bala Subrahmanyam Kambala, Bart Miller, Ben Kushigian (U. Mass), Benita Bose, Biswajit Mazumder (Clemson), Bobby Jack, Björn Lindberg, Brandon Harshe (U. Minn), Brennan Payne, Brian Gorman, Brian Kroth, Caleb Sumner (Southern Adventist), Cara Lauritzen, Charlotte Kissinger, Cheng Su, Chien-Chung Shen (Delaware)*, Christian Stober, Christoph Jaeger, C.J. Stanbridge (Memorial U. of Newfoundland), Cody Hanson, Constantinos Georgiades, Dakota Crane (U. Washington-Tacoma), Dan Soendergaard (U. Aarhus), Dan Tsafrir (Technion), Danilo Bruschi (Universita Degli Studi Di Milano), Darby Asher Noam Haller, David Hanle (Grinnell), David Hartman, Deepika Muthukumar, Demir Delic, Dennis Zhou, Dheeraj Shetty (North Carolina State), Dorian Arnold (New Mexico), Dustin Metzler, Dustin Passofaro, Eduardo Stelmasczyk, Emad Sadeghi, Emil Hessman, Emily Jacobson, Emmett Witchel (Texas), Eric Freudenthal (UTEP), Eric Johansson, Erik Turk, Ernst Biersack (France), Fangjun Kuang (U. Stuttgart), Feng Zhang (IBM), Finn Kuusisto*, Giovanni Lagorio (DIBRIS), Glenn Bruns (CSU Monterey Bay), Glen Granzow (College of Idaho), Guilherme Baptista, Hamid Reza Ghasemi, Hao Chen, Henry Abbey, Hilmar Gústafsson (Aalborg University), Hrishikesh Amur, Huanchen Zhang*, Huseyin Sular, Hugo Diaz, Ilya Oblomkov, Itai Hass (Toronto), Jackson "Jake" Haenchen (Texas), Jagannathan Eachambadi, Jake Gillberg, Jakob Olandt, James Earley, James Perry (U. Michigan-Dearborn)*, Jan Reineke (Universität des Saarlandes), Jason MacLafferty (Southern Adventist), Jason Waterman (Vassar), Jay Lim, Jerod Weinman (Grinnell), Jhih-Cheng Luo, Jiao Dong (Rutgers), Jia-Shen Boon, Jiawen Bao, Jingxin Li, Joe Jean (NYU), Joel Kuntz (Saint Mary's), Joel Sommers (Colgate), John Brady (Grinnell), John Komenda, Jonathan Perry (MIT), Joshua Carpenter (NCSU), Jun He, Karl Wallinger, Kartik Singhal, Katherine Dudenas, Katie Coyle (Georgia Tech), Kaushik Kannan, Kemal Bıçakçı, Kevin Liu*, Lanyue Lu, Laura Xu, Lei Tian (U. Nebraska-Lincoln), Leonardo Medici (U. Milan), Leslie Schultz, Liang Yin, Lihao Wang, Looserof, Manav Batra (IIIT-Delhi), Manu Awasthi (Samsung), Marcel van der Holst, Marco Guazzone (U. Piemonte Orientale), Mart Oskamp, Martha Ferris, Masashi Kishikawa (Sony), Matt Reichoff, Mattia Monga (U. Milan), Matty Williams, Meng Huang, Michael Machtel (Hochschule Konstanz), Michael Walfish (NYU), Michael Wu (UCLA), Mike Griepentrog, Ming Chen (Stonybrook), Mohammed Alali (Delaware), Mohamed Omran (GUST), Murugan Kandawamy, Nadeem Shaikh, Natasha Eilbert, Natasha Stopa, Nathan Dipiazza, Nathan Sullivan, Neeraj Badlani (N.C. State), Neil Perry, Nelson Gomez, Nghia Huynh (Texas), Nicholas Mandal, Nick Weinandt, Patel Pratyush Ashesh (BITS-Pilani), Patricio Jara, Pavle Kostovic, Perry Kivolowitz, Peter Peterson (Minnesota), Pieter Kockx, Radford Smith, Riccardo Mutschlechner, Ripudaman Singh, Robert Ordóñez and class (Southern Adventist), Roger Wattenhofer (ETH), Rohan Das (Toronto)*, Rohan Pasalkar (Minnesota), Rohan Puri, Ross Aiken, Ruslan Kiselev, Ryland Herrick, Sam Kelly, Sam Noh (UNIST), Samer Al-Kiswany, Sandeep Ummadi (Minnesota), Sankaralingam Panneerselvam, Satish Chebrolu (NetApp), Satyanarayana

Shanmugam*, Scott Catlin, Scott Lee (UCLA), Seth Pollen, Sharad Punuganti, Shreevatsa R., Simon Pratt (Waterloo), Sivaraman Sivaraman*, Song Jiang (Wayne State), Spencer Harston (Weber State), Srinivasan Thirunarayanan*, Stefan Dekanski, Stephen Bye, Suriyhaprakhas Balaram Sankari, Sy Jin Cheah, Teri Zhao (EMC), Thanumalayan S. Pillai, Thomas Griebel, Thomas Scrace, Tianxia Bai, Tong He, Tongxin Zheng, Tony Adkins, Torin Rudeen (Princeton), Tuo Wang, Tyler Couto, Varun Vats, Vikas Goel, Waciuma Wanjohi, William Royle (Grinnell), Xiang Peng, Xu Di, Yifan Hao, Yuanyuan Chen, Yubin Ruan, Yudong Sun, Yue Zhuo (Texas A&M), Yufui Ren, Zef RosnBrick, Zeyuan Hu (Texas), Zihan Zheng (USTC), 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 Meehean (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 incredibly detailed comments and tips, Michael Walfish (NYU) for his enthusiasm and detailed suggestions for improvement, Peter Peterson (UMD) for his many bits of useful feedback and commentary, Mark Kampe (Pomona) for detailed criticism (we only wish we could fix all suggestions!), and Youjip Won (Hanyang) for his translation work into Korean(!) and numerous insightful suggestions. All have helped these authors immeasurably 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 Wacławik. 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 and post-docs with whom we have published papers; an asterisk marks those who received a Ph.D. under our guidance: Abhishek Rajimwale, Aishwarya Ganesan, Andrew Krioukov, Ao Ma, Brian Forney, Chris Dragga, Deepak Ramamurthi, Dennis Zhou, Edward Oakes, Florentina Popovici*, Hariharan Gopalakrishnan, Haryadi S. Gunawi*, James Nugent, Joe Meehean*, John Bent*, Jun He, Kevin Houck, Lanyue Lu*, Lakshmi Bairava-sundaram*, Laxman Visampalli, Leo Arulraj*, Leon Yang, Meenali Rungta, Muthian

Sivathanu*, Nathan Burnett*, Nitin Agrawal*, Ram Alagappan, Samer Al-Kiswany, Scott Hendrickson, Sriram Subramanian*, Stephen Todd Jones*, Stephen Sturdevant, Sudarsun Kannan, 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*, Yuvraj Patel, Zev Weiss*.

Our graduate students have largely been funded by the National Science Foundation (NSF), the Department of Energy Office of Science (DOE), and by industry grants. We are especially grateful to the NSF for their support over many years, as our research has shaped the content of many chapters herein.

We thank Thomas Griebel, who demanded a better cover for the book. Although we didn't take his specific suggestion (a dinosaur, can you believe it?), the beautiful picture of Halley's comet would not be found on the cover without him.

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 time⁴. 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 Arpaci) 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, hopefully⁵

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

⁵If 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” by 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” by 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.) by 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” by 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” by 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.*

Contents

To Everyone	iii
To Educators	v
To Students	vi
Acknowledgments	vii
Final Words	x
References	xi
1 A Dialogue on the Book	1
2 Introduction to Operating Systems	3
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
Homework	20
I Virtualization	21
3 A Dialogue on Virtualization	23
4 The Abstraction: The Process	25
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 (Simulation) 35

5 Interlude: Process API 37

5.1 The `fork()` System Call 37

5.2 The `wait()` System Call 39

5.3 Finally, The `exec()` System Call 40

5.4 Why? Motivating The API 41

5.5 Process Control And Users 44

5.6 Useful Tools 45

5.7 Summary 45

References 47

Homework (Code) 48

6 Mechanism: Limited Direct Execution 49

6.1 Basic Technique: Limited Direct Execution 49

6.2 Problem #1: Restricted Operations 50

6.3 Problem #2: Switching Between Processes 55

6.4 Worried About Concurrency? 59

6.5 Summary 60

References 62

Homework (Measurement) 63

7 Scheduling: Introduction 65

7.1 Workload Assumptions 65

7.2 Scheduling Metrics 66

7.3 First In, First Out (FIFO) 66

7.4 Shortest Job First (SJF) 68

7.5 Shortest Time-to-Completion First (STCF) 69

7.6 A New Metric: Response Time 70

7.7 Round Robin 71

7.8 Incorporating I/O 73

7.9 No More Oracle 74

7.10 Summary 74

References 75

Homework (Simulation) 76

8 Scheduling: 77

The Multi-Level Feedback Queue 77

8.1 MLFQ: Basic Rules 78

8.2 Attempt #1: How To Change Priority 79

8.3 Attempt #2: The Priority Boost 83

8.4 Attempt #3: Better Accounting 84

8.5 Tuning MLFQ And Other Issues 84

8.6 MLFQ: Summary 86

References 87

Homework (Simulation) 88

9	Scheduling: Proportional Share	89
9.1	Basic Concept: Tickets Represent Your Share	89
9.2	Ticket Mechanisms	91
9.3	Implementation	92
9.4	An Example	93
9.5	How To Assign Tickets?	94
9.6	Why Not Deterministic?	94
9.7	The Linux Completely Fair Scheduler (CFS)	95
9.8	Summary	100
	References	101
	Homework (Simulation)	102
10	Multiprocessor Scheduling (Advanced)	103
10.1	Background: Multiprocessor Architecture	104
10.2	Don't Forget Synchronization	106
10.3	One Final Issue: Cache Affinity	107
10.4	Single-Queue Scheduling	107
10.5	Multi-Queue Scheduling	109
10.6	Linux Multiprocessor Schedulers	112
10.7	Summary	112
	References	113
	Homework (Simulation)	114
11	Summary Dialogue on CPU Virtualization	117
12	A Dialogue on Memory Virtualization	119
13	The Abstraction: Address Spaces	121
13.1	Early Systems	121
13.2	Multiprogramming and Time Sharing	122
13.3	The Address Space	123
13.4	Goals	125
13.5	Summary	127
	References	128
	Homework (Code)	129
14	Interlude: Memory API	131
14.1	Types of Memory	131
14.2	The <code>malloc()</code> Call	132
14.3	The <code>free()</code> Call	134
14.4	Common Errors	134
14.5	Underlying OS Support	137
14.6	Other Calls	138
14.7	Summary	138
	References	139
	Homework (Code)	140

15 Mechanism: Address Translation **141**

15.1 Assumptions 142

15.2 An Example 142

15.3 Dynamic (Hardware-based) Relocation 145

15.4 Hardware Support: A Summary 148

15.5 Operating System Issues 149

15.6 Summary 152

References 153

Homework (Simulation) 154

16 Segmentation **155**

16.1 Segmentation: Generalized Base/Bounds 155

16.2 Which Segment Are We Referring To? 158

16.3 What About The Stack? 159

16.4 Support for Sharing 160

16.5 Fine-grained vs. Coarse-grained Segmentation 161

16.6 OS Support 161

16.7 Summary 163

References 164

Homework (Simulation) 165

17 Free-Space Management **167**

17.1 Assumptions 168

17.2 Low-level Mechanisms 169

17.3 Basic Strategies 177

17.4 Other Approaches 179

17.5 Summary 181

References 182

Homework (Simulation) 183

18 Paging: Introduction **185**

18.1 A Simple Example And Overview 185

18.2 Where Are Page Tables Stored? 189

18.3 What’s Actually In The Page Table? 190

18.4 Paging: Also Too Slow 191

18.5 A Memory Trace 192

18.6 Summary 195

References 196

Homework (Simulation) 197

19 Paging: Faster Translations (TLBs) **199**

19.1 TLB Basic Algorithm 199

19.2 Example: Accessing An Array 201

19.3 Who Handles The TLB Miss? 203

19.4 TLB Contents: What’s In There? 205

19.5 TLB Issue: Context Switches 206

19.6 Issue: Replacement Policy 208

19.7	A Real TLB Entry	209
19.8	Summary	210
	References	211
	Homework (Measurement)	212
20	Paging: Smaller Tables	215
20.1	Simple Solution: Bigger Pages	215
20.2	Hybrid Approach: Paging and Segments	216
20.3	Multi-level Page Tables	219
20.4	Inverted Page Tables	226
20.5	Swapping the Page Tables to Disk	227
20.6	Summary	227
	References	228
	Homework (Simulation)	229
21	Beyond Physical Memory: Mechanisms	231
21.1	Swap Space	232
21.2	The Present Bit	233
21.3	The Page Fault	234
21.4	What If Memory Is Full?	235
21.5	Page Fault Control Flow	236
21.6	When Replacements Really Occur	237
21.7	Summary	238
	References	239
	Homework (Measurement)	240
22	Beyond Physical Memory: Policies	243
22.1	Cache Management	243
22.2	The Optimal Replacement Policy	244
22.3	A Simple Policy: FIFO	246
22.4	Another Simple Policy: Random	248
22.5	Using History: LRU	249
22.6	Workload Examples	250
22.7	Implementing Historical Algorithms	253
22.8	Approximating LRU	254
22.9	Considering Dirty Pages	255
22.10	Other VM Policies	256
22.11	Thrashing	256
22.12	Summary	257
	References	258
	Homework (Simulation)	259
23	Complete Virtual Memory Systems	261
23.1	VAX/VMS Virtual Memory	262
23.2	The Linux Virtual Memory System	268
23.3	Summary	277
	References	278

24 Summary Dialogue on Memory Virtualization 279

II Concurrency 283

25 A Dialogue on Concurrency 285

26 Concurrency: An Introduction 287

26.1 Why Use Threads? 288

26.2 An Example: Thread Creation 289

26.3 Why It Gets Worse: Shared Data 292

26.4 The Heart Of The Problem: Uncontrolled Scheduling . . . 294

26.5 The Wish For Atomicity 296

26.6 One More Problem: Waiting For Another 298

26.7 Summary: Why in OS Class? 298

References 300

Homework (Simulation) 301

27 Interlude: Thread API 303

27.1 Thread Creation 303

27.2 Thread Completion 304

27.3 Locks 307

27.4 Condition Variables 309

27.5 Compiling and Running 311

27.6 Summary 311

References 313

Homework (Code) 314

28 Locks 315

28.1 Locks: The Basic Idea 315

28.2 Pthread Locks 316

28.3 Building A Lock 317

28.4 Evaluating Locks 317

28.5 Controlling Interrupts 318

28.6 A Failed Attempt: Just Using Loads/Stores 319

28.7 Building Working Spin Locks with Test-And-Set 320

28.8 Evaluating Spin Locks 322

28.9 Compare-And-Swap 323

28.10 Load-Linked and Store-Conditional 324

28.11 Fetch-And-Add 326

28.12 Too Much Spinning: What Now? 327

28.13 A Simple Approach: Just Yield, Baby 328

28.14 Using Queues: Sleeping Instead Of Spinning 329

28.15 Different OS, Different Support 332

28.16 Two-Phase Locks 332

28.17 Summary 334

References 335

Homework (Simulation)	336
29 Lock-based Concurrent Data Structures	337
29.1 Concurrent Counters	337
29.2 Concurrent Linked Lists	342
29.3 Concurrent Queues	345
29.4 Concurrent Hash Table	346
29.5 Summary	348
References	349
Homework (Code)	350
30 Condition Variables	351
30.1 Definition and Routines	352
30.2 The Producer/Consumer (Bounded Buffer) Problem	355
30.3 Covering Conditions	363
30.4 Summary	364
References	365
Homework (Code)	366
31 Semaphores	367
31.1 Semaphores: A Definition	367
31.2 Binary Semaphores (Locks)	369
31.3 Semaphores For Ordering	370
31.4 The Producer/Consumer (Bounded Buffer) Problem	372
31.5 Reader-Writer Locks	376
31.6 The Dining Philosophers	378
31.7 How To Implement Semaphores	381
31.8 Summary	382
References	383
Homework (Code)	384
32 Common Concurrency Problems	385
32.1 What Types Of Bugs Exist?	385
32.2 Non-Deadlock Bugs	386
32.3 Deadlock Bugs	389
32.4 Summary	397
References	399
Homework (Code)	400
33 Event-based Concurrency (Advanced)	401
33.1 The Basic Idea: An Event Loop	401
33.2 An Important API: <code>select ()</code> (or <code>poll ()</code>)	402
33.3 Using <code>select ()</code>	403
33.4 Why Simpler? No Locks Needed	404
33.5 A Problem: Blocking System Calls	405
33.6 A Solution: Asynchronous I/O	405
33.7 Another Problem: State Management	408

33.8 What Is Still Difficult With Events 409

33.9 Summary 409

References 410

Homework (Code) 411

34 Summary Dialogue on Concurrency 413

III Persistence 415

35 A Dialogue on Persistence 417

36 I/O Devices 419

36.1 System Architecture 419

36.2 A Canonical Device 421

36.3 The Canonical Protocol 422

36.4 Lowering CPU Overhead With Interrupts 423

36.5 More Efficient Data Movement With DMA 424

36.6 Methods Of Device Interaction 425

36.7 Fitting Into The OS: The Device Driver 426

36.8 Case Study: A Simple IDE Disk Driver 427

36.9 Historical Notes 430

36.10 Summary 430

References 431

37 Hard Disk Drives 433

37.1 The Interface 433

37.2 Basic Geometry 434

37.3 A Simple Disk Drive 435

37.4 I/O Time: Doing The Math 438

37.5 Disk Scheduling 442

37.6 Summary 446

References 447

Homework (Simulation) 448

38 Redundant Arrays of Inexpensive Disks (RAIDs) 449

38.1 Interface And RAID Internals 450

38.2 Fault Model 451

38.3 How To Evaluate A RAID 451

38.4 RAID Level 0: Striping 452

38.5 RAID Level 1: Mirroring 455

38.6 RAID Level 4: Saving Space With Parity 458

38.7 RAID Level 5: Rotating Parity 462

38.8 RAID Comparison: A Summary 463

38.9 Other Interesting RAID Issues 464

38.10 Summary 464

References 465

Homework (Simulation)	466
39 Interlude: Files and Directories	467
39.1 Files And Directories	467
39.2 The File System Interface	469
39.3 Creating Files	469
39.4 Reading And Writing Files	470
39.5 Reading And Writing, But Not Sequentially	472
39.6 Shared File Table Entries: <code>fork()</code> And <code>dup()</code>	475
39.7 Writing Immediately With <code>fsync()</code>	477
39.8 Renaming Files	478
39.9 Getting Information About Files	479
39.10 Removing Files	480
39.11 Making Directories	480
39.12 Reading Directories	481
39.13 Deleting Directories	482
39.14 Hard Links	482
39.15 Symbolic Links	484
39.16 Permission Bits And Access Control Lists	485
39.17 Making And Mounting A File System	488
39.18 Summary	490
References	491
Homework (Code)	492
40 File System Implementation	493
40.1 The Way To Think	493
40.2 Overall Organization	494
40.3 File Organization: The Inode	496
40.4 Directory Organization	501
40.5 Free Space Management	501
40.6 Access Paths: Reading and Writing	502
40.7 Caching and Buffering	506
40.8 Summary	508
References	509
Homework (Simulation)	510
41 Locality and The Fast File System	511
41.1 The Problem: Poor Performance	511
41.2 FFS: Disk Awareness Is The Solution	513
41.3 Organizing Structure: The Cylinder Group	513
41.4 Policies: How To Allocate Files and Directories	515
41.5 Measuring File Locality	517
41.6 The Large-File Exception	518
41.7 A Few Other Things About FFS	520
41.8 Summary	522
References	523
Homework (Simulation)	524

42 Crash Consistency: FSK and Journaling **525**

42.1 A Detailed Example 526

42.2 Solution #1: The File System Checker 529

42.3 Solution #2: Journaling (or Write-Ahead Logging) 531

42.4 Solution #3: Other Approaches 541

42.5 Summary 542

References 543

Homework (Simulation) 545

43 Log-structured File Systems **547**

43.1 Writing To Disk Sequentially 548

43.2 Writing Sequentially And Effectively 549

43.3 How Much To Buffer? 550

43.4 Problem: Finding Inodes 551

43.5 Solution Through Indirection: The Inode Map 551

43.6 Completing The Solution: The Checkpoint Region 553

43.7 Reading A File From Disk: A Recap 553

43.8 What About Directories? 554

43.9 A New Problem: Garbage Collection 555

43.10 Determining Block Liveness 556

43.11 A Policy Question: Which Blocks To Clean, And When? 557

43.12 Crash Recovery And The Log 558

43.13 Summary 558

References 560

Homework (Simulation) 561

44 Flash-based SSDs **563**

44.1 Storing a Single Bit 563

44.2 From Bits to Banks/Planes 564

44.3 Basic Flash Operations 565

44.4 Flash Performance And Reliability 567

44.5 From Raw Flash to Flash-Based SSDs 568

44.6 FTL Organization: A Bad Approach 569

44.7 A Log-Structured FTL 570

44.8 Garbage Collection 572

44.9 Mapping Table Size 574

44.10 Wear Leveling 579

44.11 SSD Performance And Cost 579

44.12 Summary 581

References 583

Homework (Simulation) 585

45 Data Integrity and Protection **587**

45.1 Disk Failure Modes 587

45.2 Handling Latent Sector Errors 589

45.3 Detecting Corruption: The Checksum 590

45.4 Using Checksums 593