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 for[1].

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

[1]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. 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 online[2]. 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[3]), 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.

[2]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.

[3]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: 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 Stelmaszczyk, 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 Bıçakcı, 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 Kandaswamy, 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 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 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 Bairavasundaram*, 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*, Yiying 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 time[4]. 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, hopefully[5]

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

[5]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" 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

1

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*[1]*.*

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!*

[1]According to `http://www.barrypopik.com` (on, December 19, 2012, entitled "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!

2

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 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[1].

Thus, we have just described the basics of the **Von Neumann** model of computing[2]. 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

[1]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.

[2]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**)[3], 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.

[3]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.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
#include "common.h"

int
main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: cpu <string>\n");
        exit(1);
    }
    char *str = argv[1];
    while (1) {
        Spin(1);
        printf("%s\n", str);
    }
    return 0;
}
```

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; by pressing "Control-c" (which on UNIX-based systems will terminate the program running in the foreground) we can halt the program.

```
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
...
```

Figure 2.2: **Running Many Programs At Once**

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.

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?[4]

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 a 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**.

[4]Note how we ran four processes at the same time, by using the `&` symbol. Doing so runs a job in the background in the `zsh` shell, which means that the user is able to immediately issue their next command, which in this case is another program to run. If you're using a different shell (e.g., `tcsh`), it works slightly differently; read documentation online for details.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int
main(int argc, char *argv[])
{
    int *p = malloc(sizeof(int));                      // a1
    assert(p != NULL);
    printf("(%d) address pointed to by p: %p\n",
           getpid(), p);                               // a2
    *p = 0;                                            // a3
    while (1) {
        Spin(1);
        *p = *p + 1;
        printf("(%d) p: %d\n", getpid(), *p);          // a4
    }
    return 0;
}
```

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) address pointed to by p: 0x200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by p: 0x200000
(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**

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.

Again, this first result is not too interesting. The newly allocated memory is at address `0x200000`. 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 (`0x200000`), and yet each seems to be updating the value at `0x200000` independently! It is as if each running program has its own private memory, instead of sharing the same physical memory with other running programs[5].

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**.

[5]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...

2.3 Concurrency

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"
#include "common_threads.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads <value>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);

    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```

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

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.

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()`[6]. 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` 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,

[6]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?

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.

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[7]) 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!

[7]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.

```
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <fcntl.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    int fd = open("/tmp/file", O_WRONLY|O_CREAT|O_TRUNC,
                  S_IRWXU);
    assert(fd > -1);
    int rc = write(fd, "hello world\n", 13);
    assert(rc == 13);
    close(fd);
    return 0;
}
```

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

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".

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**[8] 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

[8]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.

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?

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.

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[9] 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.

[9]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!

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 challenge — 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[10].

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,

[10]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...

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 mem-

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, took advantage of the sophisticated GNU tools that already existed [G85], 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). 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.

ory. 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/macOS 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 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.

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" by L. Bairavasundaram, S. Sundararaman, A. Arpaci-Dusseau, R. 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" by 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" by D. Bobrow, J. Burchfiel, D. Murphy, R. 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" by F. Brooks. Addison-Wesley, 1975. *A classic text on software engineering; well worth the read.*

[BOH10] "Computer Systems: A Programmer's Perspective" by R. Bryant and D. 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!*

[G85] "The GNU Manifesto" by R. Stallman. 1985. `www.gnu.org/gnu/manifesto.html`. *A huge part of Linux's success was no doubt the presence of an excellent compiler,* `gcc`*, and other relevant pieces of open software, thanks to the GNU effort headed by Stallman. Stallman is a visionary when it comes to open source, and this manifesto lays out his thoughts as to why.*

[K+61] "One-Level Storage System" by 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" by S. H. Lavington. Communications of the ACM, Volume 21:1, January 1978. *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" by Elliott Organick. MIT Press, 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. 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" by Yale N. Patt, 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" by Dennis M. Ritchie, Ken Thompson. CACM, Volume 17: 7, July 1974. *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" by Scientific Data Systems. TECHNICAL MANUAL, SDS 90 11168, August 1968. *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" by S. Sundararaman, S. Subramanian, A. Rajimwale, A. Arpaci-Dusseau, R. Arpaci-Dusseau, 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.*

Homework

Most (and eventually, all) chapters of this book have homework sections at the end. Doing these homeworks is important, as each lets you, the reader, gain more experience with the concepts presented within the chapter.

There are two types of homeworks. The first is based on **simulation**. A simulation of a computer system is just a simple program that pretends to do some of the interesting parts of what a real system does, and then report some output metrics to show how the system behaves. For example, a hard drive simulator might take a series of requests, simulate how long they would take to get serviced by a hard drive with certain performance characteristics, and then report the average latency of the requests.

The cool thing about simulations is they let you easily explore how systems behave without the difficulty of running a real system. Indeed, they even let you create systems that cannot exist in the real world (for example, a hard drive with unimaginably fast performance), and thus see the potential impact of future technologies.

Of course, simulations are not without their downsides. By their very nature, simulations are just approximations of how a real system behaves. If an important aspect of real-world behavior is omitted, the simulation will report bad results. Thus, results from a simulation should always be treated with some suspicion. In the end, how a system behaves in the real world is what matters.

The second type of homework requires interaction with **real-world code**. Some of these homeworks are measurement focused, whereas others just require some small-scale development and experimentation. Both are just small forays into the larger world you should be getting into, which is how to write systems code in C on UNIX-based systems. Indeed, larger-scale projects, which go beyond these homeworks, are needed to push you in this direction; thus, beyond just doing homeworks, we strongly recommend you do projects to solidify your systems skills. See this page (`https://github.com/remzi-arpacidusseau/ostep-projects`) for some projects.

To do these homeworks, you likely have to be on a UNIX-based machine, running either Linux, macOS, or some similar system. It should also have a C compiler installed (e.g., **gcc**) as well as Python. You should also know how to edit code in a real code editor of some kind.