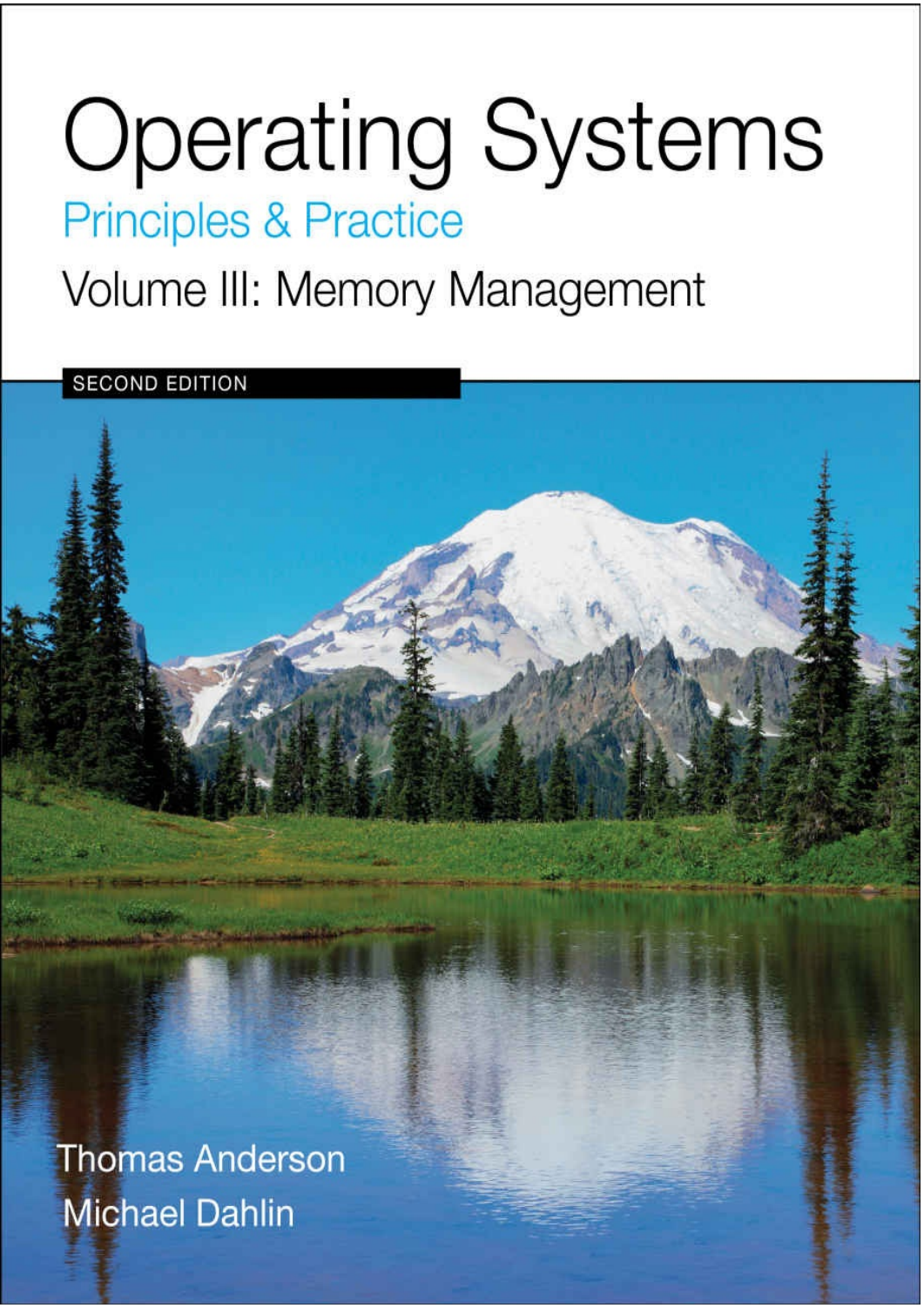

Operating Systems

Principles & Practice

Volume III: Memory Management

Second Edition

Thomas Anderson

University of Washington

Mike Dahlin

University of Texas and Google

Recursive Books

recursivebooks.com

Operating Systems: Principles and Practice (Second Edition) Volume III: Memory Management by Thomas Anderson and Michael Dahlin

ISBN 978-0-9856735-5-0
Publisher: Recursive Books, Ltd., http://recursivebooks.com/
Cover: Reflection Lake, Mt. Rainier
Cover design: Cameron Neat
Illustrations: Cameron Neat
Copy editors: Sandy Kaplan, Whitney Schmidt
Ebook design: Robin Briggs
Web design: Adam Anderson

SUGGESTIONS, COMMENTS, and ERRORS. We welcome suggestions, comments and error reports, by email to suggestions@recursivebooks.com

To Robin, Sandra, Katya, and Adam
Tom Anderson

To Marla, Kelly, and Keith
Mike Dahlin

Contents

IV: Persistent Storage

Preface

Preface to the eBook Edition

Operating Systems: Principles and Practice is a textbook for a first course in undergraduate operating systems. In use at over 50 colleges and universities worldwide, this textbook provides:

- A path for students to understand high level concepts all the way down to working code.
- Extensive worked examples integrated throughout the text provide students concrete guidance for completing homework assignments.
- A focus on up-to-date industry technologies and practice

The eBook edition is split into four volumes that together contain exactly the same material as the (2nd) print edition of Operating Systems: Principles and Practice, reformatted for various screen sizes. Each volume is self-contained and can be used as a standalone text, e.g., at schools that teach operating systems topics across multiple courses.

- **Volume 1: Kernels and Processes.** This volume contains Chapters 1-3 of the print edition. We describe the essential steps needed to isolate programs to prevent buggy applications and computer viruses from crashing or taking control of your system.
- **Volume 2: Concurrency.** This volume contains Chapters 4-7 of the print edition. We provide a concrete methodology for writing correct concurrent programs that is in widespread use in industry, and we explain the mechanisms for context switching and synchronization from fundamental concepts down to assembly code.
- **Volume 3: Memory Management.** This volume contains Chapters 8-10 of the print edition. We explain both the theory and mechanisms behind 64-bit address space translation, demand paging, and virtual machines.
- **Volume 4: Persistent Storage.** This volume contains Chapters 11-14 of the print edition. We explain the technologies underlying modern extent-based, journaling, and versioning file systems.

A more detailed description of each chapter is given in the preface to the print edition.

Preface to the Print Edition

Why We Wrote This Book

Many of our students tell us that operating systems was the best course they took as an undergraduate and also the most important for their careers. We are not alone — many of our colleagues report receiving similar feedback from their students.

Part of the excitement is that the core ideas in a modern operating system — protection, concurrency, virtualization, resource allocation, and reliable storage — have become

widely applied throughout computer science, not just operating system kernels. Whether you get a job at Facebook, Google, Microsoft, or any other leading-edge technology company, it is impossible to build resilient, secure, and flexible computer systems without the ability to apply operating systems concepts in a variety of settings. In a modern world, nearly everything a user does is distributed, nearly every computer is multi-core, security threats abound, and many applications such as web browsers have become mini-operating systems in their own right.

It should be no surprise that for many computer science students, an undergraduate operating systems class has become a *de facto* requirement: a ticket to an internship and eventually to a full-time position.

Unfortunately, many operating systems textbooks are still stuck in the past, failing to keep pace with rapid technological change. Several widely-used books were initially written in the mid-1980's, and they often act as if technology stopped at that point. Even when new topics are added, they are treated as an afterthought, without pruning material that has become less important. The result are textbooks that are very long, very expensive, and yet fail to provide students more than a superficial understanding of the material.

Our view is that operating systems have changed dramatically over the past twenty years, and that justifies a fresh look at both *how* the material is taught and *what* is taught. The pace of innovation in operating systems has, if anything, increased over the past few years, with the introduction of the iOS and Android operating systems for smartphones, the shift to multicore computers, and the advent of cloud computing.

To prepare students for this new world, we believe students need three things to succeed at understanding operating systems at a deep level:

- **Concepts and code.** We believe it is important to teach students both *principles* and *practice*, concepts and implementation, rather than either alone. This textbook takes concepts all the way down to the level of working code, e.g., how a context switch works in assembly code. In our experience, this is the only way students will really understand and master the material. All of the code in this book is available from the author's web site, ospp.washington.edu.
- **Extensive worked examples.** In our view, students need to be able to apply concepts in practice. To that end, we have integrated a large number of example exercises, along with solutions, throughout the text. We uses these exercises extensively in our own lectures, and we have found them essential to challenging students to go beyond a superficial understanding.
- **Industry practice.** To show students how to apply operating systems concepts in a variety of settings, we use detailed, concrete examples from Facebook, Google, Microsoft, Apple, and other leading-edge technology companies throughout the textbook. Because operating systems concepts are important in a wide range of computer systems, we take these examples not only from traditional operating systems like Linux, Windows, and OS X but also from other systems that need to solve problems of protection, concurrency, virtualization, resource allocation, and reliable storage like databases, web browsers, web servers, mobile applications, and search engines.

Taking a fresh perspective on what students need to know to apply operating systems concepts in practice has led us to innovate in every major topic covered in an undergraduate-level course:

- **Kernels and Processes.** The safe execution of untrusted code has become central to many types of computer systems, from web browsers to virtual machines to operating systems. Yet existing textbooks treat protection as a side effect of UNIX processes, as if they are synonyms. Instead, we start from first principles: what are the minimum requirements for process isolation, how can systems implement process isolation efficiently, and what do students need to know to implement functions correctly when the caller is potentially malicious?
- **Concurrency.** With the advent of multi-core architectures, most students today will spend much of their careers writing concurrent code. Existing textbooks provide a blizzard of concurrency alternatives, most of which were abandoned decades ago as impractical. Instead, we focus on providing students a *single* methodology based on Mesa monitors that will enable students to write correct concurrent programs — a methodology that is by far the dominant approach used in industry.
- **Memory Management.** Even as demand-paging has become less important, virtualization has become even more important to modern computer systems. We provide a deep treatment of address translation hardware, sparse address spaces, TLBs, and on-chip caches. We then use those concepts as a springboard for describing virtual machines and related concepts such as checkpointing and copy-on-write.
- **Persistent Storage.** Reliable storage in the presence of failures is central to the design of most computer systems. Existing textbooks survey the history of file systems, spending most of their time ad hoc approaches to failure recovery and de-fragmentation. Yet no modern file systems still use those ad hoc approaches. Instead, our focus is on how file systems use extents, journaling, copy-on-write, and RAID to achieve both high performance and high reliability.

Intended Audience

Operating Systems: Principles and Practice is a textbook for a first course in undergraduate operating systems. We believe operating systems should be taken as early as possible in an undergraduate's course of study; many students use the course as a springboard to an internship and a career. To that end, we have designed the textbook to assume minimal pre-requisites: specifically, students should have taken a data structures course and one on computer organization. The code examples are written in a combination of x86 assembly, C, and C++. In particular, we have designed the book to interface well with the Bryant and O'Halloran textbook. We review and cover in much more depth the material from the second half of that book.

We should note what this textbook is *not*: it is not intended to teach the API or internals of any specific operating system, such as Linux, Android, Windows 8, OS X, or iOS. We use many concrete examples from these systems, but our focus is on the shared problems these

systems face and the technologies these systems use to solve those problems.

A Guide to Instructors

One of our goals is enable instructors to choose an appropriate level of depth for each course topic. Each chapter begins at a conceptual level, with implementation details and the more advanced material towards the end. The more advanced material can be omitted without compromising the ability of students to follow later material. No single-quarter or single-semester course is likely to be able to cover every topic we have included, but we think it is a good thing for students to come away from an operating systems course with an appreciation that there is *always* more to learn.

For each topic, we attempt to convey it at three levels:

- **How to reason about systems.** We describe core systems concepts, such as protection, concurrency, resource scheduling, virtualization, and storage, and we provide practice applying these concepts in various situations. In our view, this provides the biggest long-term payoff to students, as they are likely to need to apply these concepts in their work throughout their career, almost regardless of what project they end up working on.
- **Power tools.** We introduce students to a number of abstractions that they can apply in their work in industry immediately after graduation, and that we expect will continue to be useful for decades such as sandboxing, protected procedure calls, threads, locks, condition variables, caching, checkpointing, and transactions.
- **Details of specific operating systems.** We include numerous examples of how different operating systems work in practice. However, this material changes rapidly, and there is an order of magnitude more material than can be covered in a single semester-length course. The purpose of these examples is to illustrate how to use the operating systems principles and power tools to solve concrete problems. We do not attempt to provide a comprehensive description of Linux, OS X, or any other particular operating system.

The book is divided into five parts: an introduction (Chapter 1), kernels and processes (Chapters 2-3), concurrency, synchronization, and scheduling (Chapters 4-7), memory management (Chapters 8-10), and persistent storage (Chapters 11-14).

- **Introduction.** The goal of Chapter 1 is to introduce the recurring themes found in the later chapters. We define some common terms, and we provide a bit of the history of the development of operating systems.
- **The Kernel Abstraction.** Chapter 2 covers kernel-based process protection — the concept and implementation of executing a user program with restricted privileges. Given the increasing importance of computer security issues, we believe protected execution and safe transfer across privilege levels are worth treating in depth. We have broken the description into sections, to allow instructors to choose either a quick introduction to the concepts (up through Section 2.3), or a full treatment of the kernel implementation details down to the level of interrupt handlers. Some instructors start with concurrency, and cover kernels and kernel protection afterwards. While our textbook can be used that way, we have found that students benefit from a basic understanding of the role of operating systems in executing user programs, before introducing concurrency.
- **The Programming Interface.** Chapter 3 is intended as an impedance match for students of differing backgrounds. Depending on student background, it can be skipped or covered in depth. The chapter covers the operating system from a programmer's perspective: process creation and management, device-independent input/output, interprocess communication, and network sockets. Our goal is that students should understand at a detailed level what happens when a user clicks a link in a web browser, as the request is transferred through operating system kernels and user space processes at the client, server, and back again. This chapter also covers the organization of the operating system itself: how device drivers and the hardware abstraction layer work in a modern operating system; the difference between a monolithic and a microkernel operating system; and how policy and mechanism are separated in modern operating systems.
- **Concurrency and Threads.** Chapter 4 motivates and explains the concept of threads. Because of the increasing importance of concurrent programming, and its integration with modern programming languages like Java, many students have been introduced to multi-threaded programming in an earlier class. This is a bit dangerous, as students at this stage are prone to writing programs with race conditions, problems that may or may not be discovered with testing. Thus, the goal of this chapter is to provide a solid conceptual framework for understanding the semantics of concurrency, as well as how concurrent threads are implemented in both the operating system kernel and in user-level libraries. Instructors needing to go more quickly can omit these implementation details.
- **Synchronization.** Chapter 5 discusses the synchronization of multi-threaded programs, a central part of all operating systems and increasingly important in many other contexts. Our approach is to describe one effective method for structuring concurrent programs (based on Mesa monitors), rather than to attempt to cover several different approaches. In our view, it is more important for students to master one methodology. Monitors are a particularly robust and simple one, capable of implementing most concurrent programs efficiently. The implementation of synchronization primitives should be included if there is time, so students see that there is no magic.
- **Multi-Object Synchronization.** Chapter 6 discusses advanced topics in concurrency — specifically, the twin challenges of multiprocessor lock contention and deadlock. This material is increasingly important for students working on multicore systems, but some courses may not have time to cover it in detail.
- **Scheduling.** This chapter covers the concepts of resource allocation in the specific context of processor scheduling. With the advent of data center computing and multicore architectures, the principles and practice of resource allocation have renewed importance. After a quick tour through the tradeoffs between response time and throughput for uniprocessor scheduling, the chapter covers a set of more

advanced topics in affinity and multiprocessor scheduling, power-aware and deadline scheduling, as well as basic queueing theory and overload management. We conclude these topics by walking students through a case study of server-side load management.

- **Address Translation.** Chapter 8 explains mechanisms for hardware and software address translation. The first part of the chapter covers how hardware and operating systems cooperate to provide flexible, sparse address spaces through multi-level segmentation and paging. We then describe how to make memory management efficient with translation lookaside buffers (TLBs) and virtually addressed caches. We consider how to keep TLBs consistent when the operating system makes changes to its page tables. We conclude with a discussion of modern software-based protection mechanisms such as those found in the Microsoft Common Language Runtime and Google's Native Client.
- **Caching and Virtual Memory.** Caches are central to many different types of computer systems. Most students will have seen the concept of a cache in an earlier class on machine structures. Thus, our goal is to cover the theory and implementation of caches: when they work and when they do not, as well as how they are implemented in hardware and software. We then show how these ideas are applied in the context of memory-mapped files and demand-paged virtual memory.
- **Advanced Memory Management.** Address translation is a powerful tool in system design, and we show how it can be used for zero copy I/O, virtual machines, process checkpointing, and recoverable virtual memory. As this is more advanced material, it can be skipped by those classes pressed for time.
- **File Systems: Introduction and Overview.** Chapter 11 frames the file system portion of the book, starting top down with the challenges of providing a useful file abstraction to users. We then discuss the UNIX file system interface, the major internal elements inside a file system, and how disk device drivers are structured.
- **Storage Devices.** Chapter 12 surveys block storage hardware, specifically magnetic disks and flash memory. The last two decades have seen rapid change in storage technology affecting both application programmers and operating systems designers; this chapter provides a snapshot for students, as a building block for the next two chapters. If students have previously seen this material, this chapter can be skipped.
- **Files and Directories.** Chapter 13 discusses file system layout on disk. Rather than survey all possible file layouts — something that changes rapidly over time — we use file systems as a concrete example of mapping complex data structures onto block storage devices.
- **Reliable Storage.** Chapter 14 explains the concept and implementation of reliable storage, using file systems as a concrete example. Starting with the ad hoc techniques used in early file systems, the chapter explains checkpointing and write ahead logging as alternate implementation strategies for building reliable storage, and it discusses how redundancy such as checksums and replication are used to improve reliability and availability.

We welcome and encourage suggestions for how to improve the presentation of the material; please send any comments to the publisher's website, suggestions@recursivebooks.com.

Acknowledgements

We have been incredibly fortunate to have the help of a large number of people in the conception, writing, editing, and production of this book.

We started on the journey of writing this book over dinner at the USENIX NSDI conference in 2010. At the time, we thought perhaps it would take us the summer to complete the first version and perhaps a year before we could declare ourselves done. We were very wrong! It is no exaggeration to say that it would have taken us a lot longer without the help we have received from the people we mention below.

Perhaps most important have been our early adopters, who have given us enormously useful feedback as we have put together this edition:

Carnegie-Mellon	David Eckhardt and Garth Gibson
Clarkson	Jeanna Matthews
Cornell	Gun Sirer
ETH Zurich	Mothy Roscoe
New York University	Laskshmi Subramanian
Princeton University	Kai Li
Saarland University	Peter Druschel
Stanford University	John Ousterhout
University of California Riverside	Harsha Madhyastha
University of California Santa Barbara	Ben Zhao
University of Maryland	Neil Spring
University of Michigan	Pete Chen
University of Southern California	Ramesh Govindan
University of Texas-Austin	Lorenzo Alvisi

Universtiy of Toronto	Ding Yuan
University of Washington	Gary Kimura and Ed Lazowska

In developing our approach to teaching operating systems, both before we started writing and afterwards as we tried to put our thoughts to paper, we made extensive use of lecture notes and slides developed by other faculty. Of particular help were the materials created by Pete Chen, Peter Druschel, Steve Gribble, Eddie Kohler, John Ousterhout, Mothy Roscoe, and Geoff Voelker. We thank them all.

Our illustrator for the second edition, Cameron Neat, has been a joy to work with. We would also like to thank Simon Peter for running the multiprocessor experiments introducing Chapter 6.

We are also grateful to Lorenzo Alvisi, Adam Anderson, Pete Chen, Steve Gribble, Sam Hopkins, Ed Lazowska, Harsha Madhyastha, John Ousterhout, Mark Rich, Mothy Roscoe, Will Scott, Gun Sirer, Ion Stoica, Lakshmi Subramanian, and John Zahorjan for their helpful comments and suggestions as to how to improve the book.

We thank Josh Berlin, Marla Dahlin, Rasit Eskicioglu, Sandy Kaplan, John Ousterhout, Whitney Schmidt, and Mike Walfish for helping us identify and correct grammatical or technical bugs in the text.

We thank Jeff Dean, Garth Gibson, Mark Oskin, Simon Peter, Dave Probert, Amin Vahdat, and Mark Zbikowski for their help in explaining the internal workings of some of the commercial systems mentioned in this book.

We would like to thank Dave Wetherall, Dan Weld, Mike Walfish, Dave Patterson, Olav Kvern, Dan Halperin, Armando Fox, Robin Briggs, Katya Anderson, Sandra Anderson, Lorenzo Alvisi, and William Adams for their help and advice on textbook economics and production.

The Helen Riaboff Whiteley Center as well as Don and Jeanne Dahlin were kind enough to lend us a place to escape when we needed to get chapters written.

Finally, we thank our families, our colleagues, and our students for supporting us in this larger-than-expected effort.

III
Memory Management

8. Address Translation

There is nothing wrong with your television set. Do not attempt to adjust the picture. We are controlling transmission. If we wish to make it louder, we will bring up the volume. If we wish to make it softer, we will tune it to a whisper. We will control the horizontal. We will control the vertical. We can roll the image, make it flutter. We can change the focus to a soft blur or sharpen it to crystal clarity. For the next hour, sit quietly and we will control all that you see and hear. We repeat: there is nothing wrong with your television set.
—Opening narration, The Outer Limits

The promise of virtual reality is compelling. Who wouldn't want the ability to travel anywhere without leaving the holodeck? Of course, the promise is far from becoming a reality. In theory, by adjusting the inputs to all of your senses in response to your actions, a virtual reality system could perfectly set the scene. However, your senses are not so easily controlled. We might soon be able to provide an immersive environment for vision, but balance, hearing, taste, and smell will take a lot longer. Touch, prioperception (the sense of being near something else), and g-forces are even farther off. Get a single one of these wrong and the illusion disappears.

Can we create a virtual reality environment for computer programs? We have already seen an example of this with the UNIX I/O interface, where the program does not need to know, and sometimes cannot tell, if its inputs and outputs are files, devices, or other processes.

In the next three chapters, we take this idea a giant step further. An amazing number of advanced system features are enabled by putting the operating system in control of *address translation*, the conversion from the memory address the program thinks it is referencing to the physical location of that memory cell. From the programmer's perspective, address translation occurs transparently — the program behaves correctly despite the fact that its memory is stored somewhere completely different from where it thinks it is stored.

You were probably taught in some early programming class that a memory address is just an address. A pointer in a linked list contains the actual memory address of what it is pointing to. A jump instruction contains the actual memory address of the next instruction to be executed. This is a useful fiction! The programmer is often better off not thinking about how each memory reference is converted into the data or instruction being referenced. In practice, there is quite a lot of activity happening beneath the covers.

Address translation is a simple concept, but it turns out to be incredibly powerful. What can an operating system do with address translation? This is only a partial list:

- **Process isolation.** As we discussed in Chapter 2, protecting the operating system kernel and other applications against buggy or malicious code requires the ability to limit memory references by applications. Likewise, address translation can be used by applications to construct safe execution sandboxes for third party extensions.
- **Interprocess communication.** Often processes need to coordinate with each other, and an efficient way to do that is to have the processes share a common memory region.
- **Shared code segments.** Instances of the same program can share the program's instructions, reducing their memory footprint and making the processor cache more efficient. Likewise, different programs can share common libraries.
- **Program initialization.** Using address translation, we can start a program running before all of its code is loaded into memory from disk.
- **Efficient dynamic memory allocation.** As a process grows its heap, or as a thread grows its stack, we can use address translation to trap to the kernel to allocate memory for those purposes only as needed.
- **Cache management.** As we will explain in the next chapter, the operating system can arrange how programs are positioned in physical memory to improve cache efficiency, through a system called page coloring.
- **Program debugging.** The operating system can use memory translation to prevent a buggy program from overwriting its own code region; by catching pointer errors earlier, it makes them much easier to debug. Debuggers also use address translation to install data breakpoints, to stop a program when it references a particular memory location.
- **Efficient I/O.** Server operating systems are often limited by the rate at which they can transfer data to and from the disk and the network. Address translation enables data to be safely transferred directly between user-mode applications and I/O devices.
- **Memory mapped files.** A convenient and efficient abstraction for many applications is to map files into the address space, so that the contents of the file can be directly referenced with program instructions.
- **Virtual memory.** The operating system can provide applications the abstraction of more memory than is physically present on a given computer.
- **Checkpointing and restart.** The state of a long-running program can be periodically checkpointed so that if the program or system crashes, it can be restarted from the saved state. The key challenge is to be able to perform an internally consistent checkpoint of the program's data while the program continues to run.
- **Persistent data structures.** The operating system can provide the abstraction of a persistent region of memory, where changes to the data structures in that region survive program and system crashes.
- **Process migration.** An executing program can be transparently moved from one server to another, for example, for load balancing.
- **Information flow control.** An extra layer of security is to verify that a program is not sending your private data to a third party; e.g., a smartphone application may need access to your phone list, but it shouldn't be allowed to transmit that data. Address translation can be the basis for managing the flow of information into and out of a system.

- **Distributed shared memory.** We can transparently turn a network of servers into a large-scale shared-memory parallel computer using address translation.

In this chapter, we focus on the mechanisms needed to implement address translation, as that is the foundation of all of these services. We discuss how the operating system and applications use the mechanisms to provide these services in the following two chapters.

For runtime efficiency, most systems have specialized hardware to do address translation; this hardware is managed by the operating system kernel. In some systems, however, the translation is provided by a trusted compiler, linker or byte-code interpreter. In other systems, the application does the pointer translation as a way of managing the state of its own data structures. In still other systems, a hybrid model is used where addresses are translated both in software and hardware. The choice is often an engineering tradeoff between performance, flexibility, and cost. However, the functionality provided is often the same regardless of the mechanism used to implement the translation. In this chapter, we will cover a range of hardware and software mechanisms.

Chapter roadmap:

- **Address Translation Concept.** We start by providing a conceptual framework for understanding both hardware and software address translation. (Section 8.1)
- **Flexible Address Translation.** We focus first on hardware address translation; we ask how can we design the hardware to provide maximum flexibility to the operating system kernel? (Section 8.2)
- **Efficient Address Translation.** The solutions we present will seem flexible but terribly slow. We next discuss mechanisms that make address translation much more efficient, without sacrificing flexibility. (Section 8.3)
- **Software Protection.** Increasingly, software compilers and runtime interpreters are using address translation techniques to implement operating system functionality. What changes when the translation is in software rather than in hardware? (Section 8.4)

8.1 Address Translation Concept

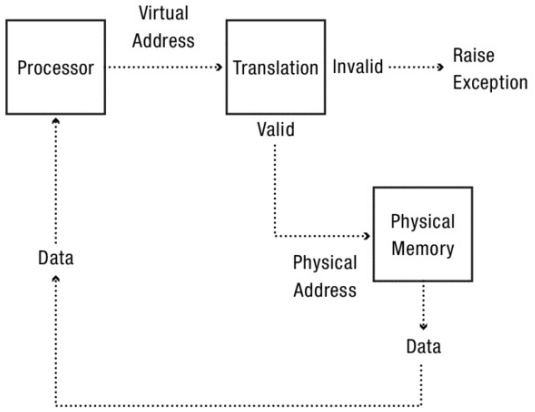

Figure 8.1: Address translation in the abstract. The translator converts (virtual) memory addresses generated by the program into physical memory addresses.

Considered as a black box, address translation is a simple function, illustrated in Figure 8.1. The translator takes each instruction and data memory reference generated by a process, checks whether the address is legal, and converts it to a physical memory address that can be used to fetch or store instructions or data. The data itself — whatever is stored in memory — is returned as is; it is not transformed in any way. The translation is usually implemented in hardware, and the operating system kernel configures the hardware to accomplish its aims.

The task of this chapter is to fill in the details about how that black box works. If we asked you right now how you might implement it, your first several guesses would probably be on the mark. If you said we could use an array, a tree, or a hash table, you would be right — all of those approaches have been taken by real systems.

Given that a number of different implementations are possible, how should we evaluate the alternatives? Here are some goals we might want out of a translation box; the design we end up with will depend on how we balance among these various goals.

- **Memory protection.** We need the ability to limit the access of a process to certain regions of memory, e.g., to prevent it from accessing memory not owned by the process. Often, however, we may want to limit access of a program to its own memory, e.g., to prevent a pointer error from overwriting the code region or to cause a trap to the debugger when the program references a specific data location.
- **Memory sharing.** We want to allow multiple processes to share selected regions of memory. These shared regions can be large (e.g., if we are sharing a program's code segment among multiple processes executing the same program) or relatively small

(e.g., if we are sharing a common library, a file, or a shared data structure).

- **Flexible memory placement.** We want to allow the operating system the flexibility to place a process (and each part of a process) anywhere in physical memory; this will allow us to pack physical memory more efficiently. As we will see in the next chapter, flexibility in assigning process data to physical memory locations will also enable us to make more effective use of on-chip caches.
- **Sparse addresses.** Many programs have multiple dynamic memory regions that can change in size over the course of the execution of the program: the heap for data objects, a stack for each thread, and memory mapped files. Modern processors have 64-bit address spaces, allowing each dynamic object ample room to grow as needed, but making the translation function more complex.
- **Runtime lookup efficiency.** Hardware address translation occurs on every instruction fetch and every data load and store. It would be impractical if a lookup took, on average, much longer to execute than the instruction itself. At first, many of the schemes we discuss will seem wildly impractical! We will discuss ways to make even the most convoluted translation systems efficient.
- **Compact translation tables.** We also want the space overhead of translation to be minimal; any data structures we need should be small compared to the amount of physical memory being managed.
- **Portability.** Different hardware architectures make different choices as to how they implement translation; if an operating system kernel is to be easily portable across multiple processor architectures, it needs to be able to map from its (hardware-independent) data structures to the specific capabilities of each architecture.

We will end up with a fairly complex address translation mechanism, and so our discussion will start with the simplest possible mechanisms and add functionality only as needed. It will be helpful during the discussion for you to keep in mind the two views of memory: the process sees its own memory, using its own addresses. We will call these *virtual addresses*, because they do not necessarily correspond to any physical reality. By contrast, to the memory system, there are only *physical addresses* — real locations in memory. From the memory system perspective, it is given physical addresses and it does lookups and stores values. The translation mechanism converts between the two views: from a virtual address to a physical memory address.

Address translation in linkers and loaders

Even without the kernel-user boundary, multiprogramming requires some form of address translation. On a multiprogramming system, when a program is compiled, the compiler does not know which regions of physical memory will be in use by other applications; it cannot control where in physical memory the program will land. The machine instructions for a program contains both relative and absolute addresses; relative addresses, such as to branch forward or backward a certain number of instructions, continue to work regardless of where in memory the program is located. However, some instructions contain absolute addresses, such as to load a global variable or to jump to the start of a procedure. These will stop working unless the program is loaded into memory exactly where the compiler expects it to go.

Before hardware translation became commonplace, early operating systems dealt with this issue by using a *relocating loader* for copying programs into memory. Once the operating system picked an empty region of physical memory for the program, the loader would modify any instructions in the program that used an absolute address. To simplify the implementation, there was a table at the beginning of the executable image that listed all of the absolute addresses used in the program. In modern systems, this is called a *symbol table*.

Today, we still have something similar. Complex programs often have multiple files, each of which can be compiled independently and then *linked* together to form the executable image. When the compiler generates the machine instructions for a single file, it cannot know where in the executable this particular file will go. Instead, the compiler generates a symbol table at the beginning of each compiled file, indicating which values will need to be modified when the individual files are assembled together.

Most commercial operating systems today support the option of dynamic linking, taking the notion of a relocating loader one step further. With a dynamically linked library (DLL), a library is linked into a running program on demand, when the program first calls into the library. We will explain in a bit how the code for a DLL can be shared between multiple different processes, but the linking procedure is straightforward. A table of valid entry points into the DLL is kept by the compiler; the calling program indirects through this table to reach the library routine.

8.2 Towards Flexible Address Translation

Our discussion of hardware address translation is divided into two steps. First, we put the issue of lookup efficiency aside, and instead consider how best to achieve the other goals listed above: flexible memory assignment, space efficiency, fine-grained protection and sharing, and so forth. Once we have the features we want, we will then add mechanisms to gain back lookup efficiency.

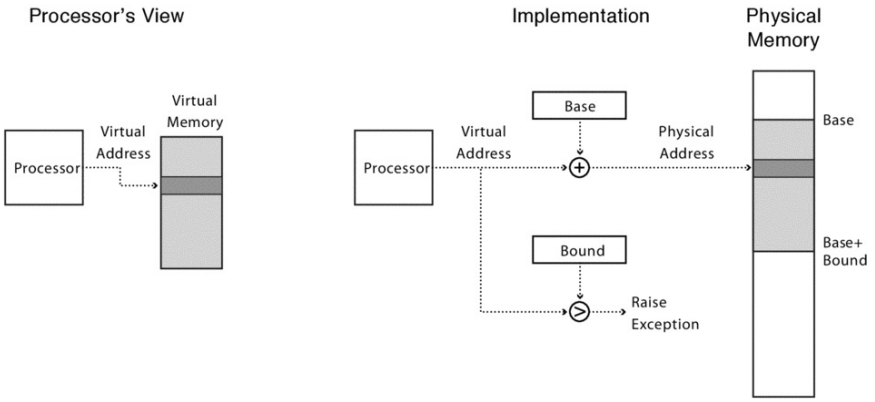

Figure 8.2: Address translation with base and bounds registers. The virtual address is added to the base to generate the physical address; the bound register is checked against the virtual address to prevent a process from reading or writing outside of its allocated memory region.

In Chapter 2, we illustrated the notion of hardware memory protection using the simplest hardware imaginable: base and bounds. The translation box consists of two extra registers per process. The *base* register specifies the start of the process's region of physical memory; the *bound* register specifies the extent of that region. If the base register is added to every address generated by the program, then we no longer need a relocating loader — the virtual addresses of the program start from 0 and go to bound, and the physical addresses start from base and go to base + bound. Figure 8.2 shows an example of base and bounds translation. Since physical memory can contain several processes, the kernel resets the contents of the base and bounds registers on each process context switch to the appropriate values for that process.

Base and bounds translation is both simple and fast, but it lacks many of the features needed to support modern programs. Base and bounds translation supports only coarse-grained protection at the level of the entire process; it is not possible to prevent a program from overwriting its own code, for example. It is also difficult to share regions of memory between two processes. Since the memory for a process needs to be contiguous, supporting dynamic memory regions, such as for heaps, thread stacks, or memory mapped files, becomes difficult to impossible.

8.2.1 Segmented Memory

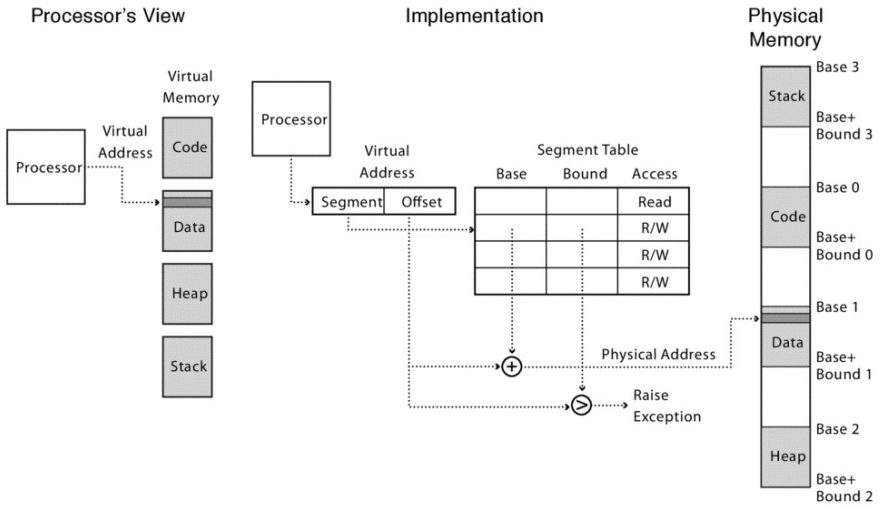

Figure 8.3: Address translation with a segment table. The virtual address has two components: a segment number and a segment offset. The segment number indexes into the segment table to locate the start of the segment in physical memory. The bound register is checked against the segment offset to prevent a process from reading or writing outside of its allocated memory region. Processes can have restricted rights to certain segments, e.g., to prevent writes to the code segment.

Many of the limitations of base and bounds translation can be remedied with a small change: instead of keeping only a single pair of base and bounds registers per process, the hardware can support an array of pairs of base and bounds registers, for each process. This is called *segmentation*. Each entry in the array controls a portion, or *segment*, of the virtual address space. The physical memory for each segment is stored contiguously, but different segments can be stored at different locations. Figure 8.3 shows segment translation in action. The high order bits of the virtual address are used to index into the array; the rest of the address is then treated as above — added to the base and checked against the bound stored at that index. In addition, the operating system can assign different segments different permissions, e.g., to allow execute-only access to code and read-write access to data. Although four segments are shown in the figure, in general the number of segments is determined by the number of bits for the segment number that are set aside in the virtual address.

It should seem odd to you that segmented memory has gaps; program memory is no longer a single contiguous region, but instead it is a set of regions. Each different segment starts at a new segment boundary. For example, code and data are not immediately adjacent to each other in either the virtual or physical address space.

What happens if a program branches into or tries to load data from one of these gaps? The hardware will generate an exception, trapping into the operating system kernel. On UNIX systems, this is still called a *segmentation fault*, that is, a reference outside of a legal segment of memory. How does a program keep from wandering into one of these gaps?

Correct programs will not generate references outside of valid memory. Put another way, trying to execute code or reading data that does not exist is probably an indication that the program has a bug in it.

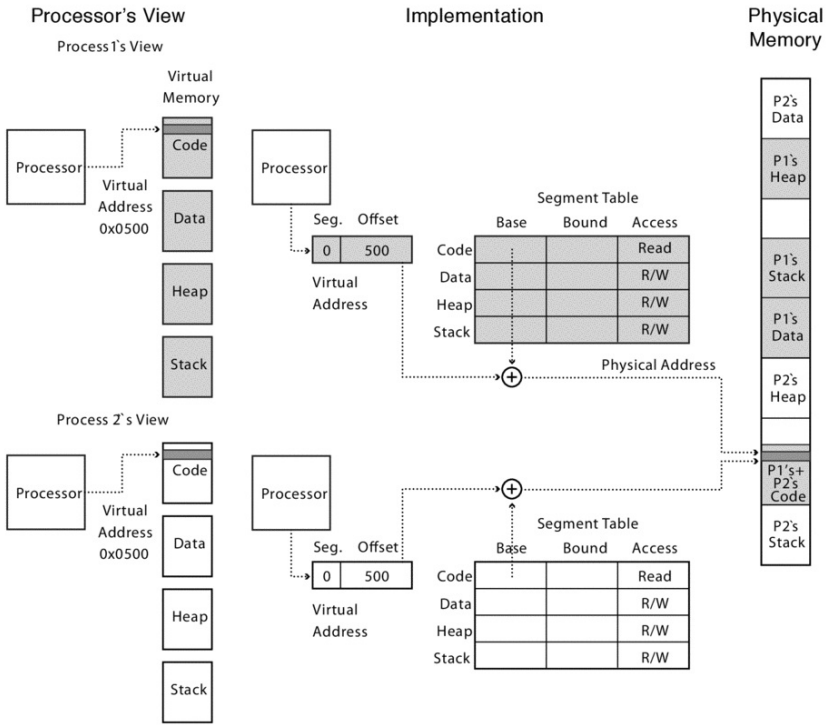

Figure 8.4: Two processes sharing a code segment, but with separate data and stack segments. In this case, each process uses the same virtual addresses, but these virtual addresses map to either the same region of physical memory (if code) or different regions of physical memory (if data).

Although simple to implement and manage, segmented memory is both remarkably powerful and widely used. For example, the x86 architecture is segmented (with some enhancements that we will describe later). With segments, the operating system can allow processes to share some regions of memory while keeping other regions protected. For example, two processes can share a code segment by setting up an entry in their segment tables to point to the same region of physical memory — to use the same base and bounds. The processes can share the same code while working off different data, by setting up the segment table to point to different regions of physical memory for the data segment. We illustrate this in Figure 8.4.

Likewise, shared library routines, such as a graphics library, can be placed into a segment and shared between processes. As before, the library data would be in a separate, non-shared segment. This is frequently done in modern operating systems with dynamically linked libraries. A practical issue is that different processes may load different numbers of libraries, and so may assign the same library a different segment number. Depending on the processor architecture, sharing can still work, if the library code uses *segment-local addresses*, addresses that are relative to the current segment.

UNIX fork and copy-on-write

In Chapter 3, we described the UNIX fork system call. UNIX creates a new process by making a complete copy of the parent process; the parent process and the child process are identical except for the return value from fork. The child process can then set up its I/O and eventually use the UNIX exec system call to run a new program. We promised at the time we would explain how this can be done efficiently.

With segments, this is now possible. To fork a process, we can simply make a copy of the parent's segment table; we do not need to copy *any* of its physical memory. Of course, we want the child to be a copy of the parent, and not just point to the same memory as the parent. If the child changes some data, it should change only its copy, and not its parent's data. On the other hand, most of the time, the child process in UNIX fork simply calls UNIX exec; the shared data is there as a programming convenience.

We can make this work efficiently by using an idea called *copy-on-write*. During the fork, all of the segments shared between the parent and child process are marked "read-only" in both segment tables. If either side modifies data in a segment, an exception is raised and a full memory copy of that segment is made at that time. In the common case, the child process modifies only its stack before calling UNIX exec, and if so, only the stack needs to be physically copied.

We can also use segments for interprocess communication, if processes are given read and write permission to the same segment. Multics, an operating system from the 1960's that contained many of the ideas we now find in Microsoft's Windows 7, Apple's Mac OS X, and Linux, made extensive use of segmented memory for interprocess sharing. In Multics, a segment was allocated for every data structure, allowing fine-grained protection and sharing between processes. Of course, this made the segment table pretty large! More modern systems tend to use segments only for coarser-grained memory regions, such as the code and data for an entire shared library, rather than for each of the data structures within the library.

As a final example of the power of segments, they enable the efficient management of dynamically allocated memory. When an operating system reuses memory or disk space that had previously been used, it must first zero out the contents of the memory or disk. Otherwise, private data from one application could inadvertently leak into another, potentially malicious, application. For example, you could enter a password into one web site, say for a bank, and then exit the browser. However, if the underlying physical memory used by the browser is then re-assigned to a new process, then the password could be leaked to a malicious web site.

Of course, we only want to pay the overhead of zeroing memory if it will be used. This is

particularly an issue for dynamically allocated memory on the heap and stack. It is not clear when the program starts how much memory it will use; the heap could be anywhere from a few kilobytes to several gigabytes, depending on the program. The operating system can address this using *zero-on-reference*. With zero-on-reference, the operating system allocates a memory region for the heap, but only zeroes the first few kilobytes. Instead, it sets the bound register in the segment table to limit the program to just the zeroed part of memory. If the program expands its heap, it will take an exception, and the operating system kernel can zero out additional memory before resuming execution.

Given all these advantages, why not stop here? The principal downside of segmentation is the overhead of managing a large number of variable size and dynamically growing memory segments. Over time, as processes are created and finish, physical memory will be divided into regions that are in use and regions that are not, that is, available to be allocated to a new process. These free regions will be of varying sizes. When we create a new segment, we will need to find a free spot for it. Should we put it in the smallest open region where it will fit? The largest open region?

However we choose to place new segments, as more memory becomes allocated, the operating system may reach a point where there is enough free space for a new segment, but the free space is not contiguous. This is called *external fragmentation*. The operating system is free to compact memory to make room without affecting applications, because virtual addresses are unchanged when we relocate a segment in physical memory. Even so, compaction can be costly in terms of processor overhead: a typical server configuration would take roughly a second to compact its memory.

All this becomes even more complex when memory segments can grow. How much memory should we set aside for a program's heap? If we put the heap segment in a part of physical memory with lots of room, then we will have wasted memory if that program turns out to need only a small heap. If we do the opposite — put the heap segment in a small chunk of physical memory — then we will need to copy it somewhere else if it changes size.

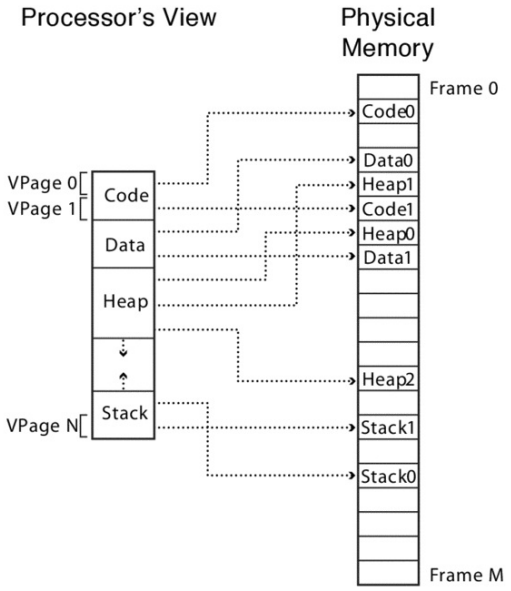

Figure 8.5: Logical view of page table address translation. Physical memory is split into page frames, with a page-size aligned block of virtual addresses assigned to each frame. Unused addresses are not assigned page frames in physical memory.

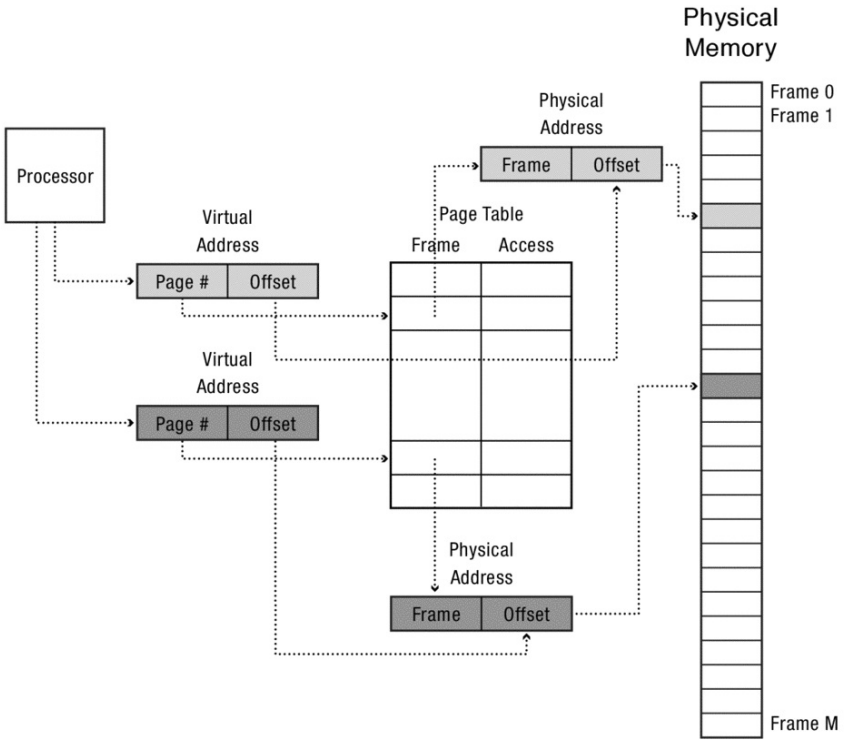

Figure 8.6: Address translation with a page table. The virtual address has two components: a virtual page number and an offset within the page. The virtual page number indexes into the page table to yield a page frame in physical memory. The physical address is the physical page frame from the page table, concatenated with the page offset from the virtual address. The operating system can restrict process access to certain pages, e.g., to prevent writes to pages containing instructions.

8.2.2 Paged Memory

An alternative to segmented memory is *paged memory*. With paging, memory is allocated in fixed-sized chunks called *page frames*. Address translation is similar to how it works with segmentation. Instead of a segment table whose entries contain pointers to variable-sized segments, there is a page table for each process whose entries contain pointers to page frames. Because page frames are fixed-sized and a power of two, the page table entries only need to provide the upper bits of the page frame address, so they are more compact. There is no need for a "bound" on the offset; the entire page in physical memory is allocated as a unit. Figure 8.6 illustrates address translation with paged memory.

What will seem odd, and perhaps cool, about paging is that while a program thinks of its memory as linear, in fact its memory can be, and usually is, scattered throughout physical memory in a kind of abstract mosaic. The processor will execute one instruction after another using virtual addresses; its virtual addresses are still linear. However, the instruction located at the end of a page will be located in a completely different region of physical memory from the next instruction at the start of the next page. Data structures will appear to be contiguous using virtual addresses, but a large matrix may be scattered across many physical page frames.

An apt analogy is what happens when you shuffle several decks of cards together. A single process in its virtual address space sees the cards of a single deck in order. A different process sees a completely different deck, but it will also be in order. In physical memory, however, the decks of all the processes currently running will be shuffled together, apparently at random. The page tables are the magician's assistant: able to instantly find the queen of hearts from among the shuffled decks.

Paging addresses the principal limitation of segmentation: free-space allocation is very straightforward. The operating system can represent physical memory as a bit map, with each bit representing a physical page frame that is either free or in use. Finding a free frame is just a matter of finding an empty bit.

Sharing memory between processes is also convenient: we need to set the page table entry for each process sharing a page to point to the same physical page frame. For a large shared region that spans multiple page frames, such as a shared library, this may require setting up a number of page table entries. Since we need to know when to release memory when a process finishes, shared memory requires some extra bookkeeping to keep track of whether the shared page is still in use. The data structure for this is called a *core map*; it records information about each physical page frame such as which page table entries point to it.

Many of the optimizations we discussed under segmentation can also be done with paging. For copy-on-write, we need to copy the page table entries and set them to read-only; on a store to one of these pages, we can make a real copy of the underlying page frame before resuming the process. Likewise, for zero-on-reference, we can set the page table entry at the top of the stack to be invalid, causing a trap into the kernel. This allows us to extend the stack only as needed.

Page tables allow other features to be added. For example, we can start a program running before all of its code and data are loaded into memory. Initially, the operating system marks all of the page table entries for a new process as invalid; as pages are brought in from disk, it marks those pages as read-only (for code pages) or read-write (for data pages). Once the first few pages are in memory, however, the operating system can start execution of the program in user-mode, while the kernel continues to transfer the rest of the program's code in the background. As the program starts up, if it happens to jump to a location that has not been loaded yet, the hardware will cause an exception, and the kernel can stall the program until that page is available. Further, the compiler can reorganize the program executable for more efficient startup, by coalescing the initialization pages into a few pages at the start of the program, thus overlapping initialization and loading the program from disk.

As another example, a *data breakpoint* is request to stop the execution of a program when

it references or modifies a particular memory location. It is helpful during debugging to know when a data structure has been changed, particularly when tracking down pointer errors. Data breakpoints are sometimes implemented with special hardware support, but they can also be implemented with page tables. For this, the page table entry containing the location is marked read-only. This causes the process to trap to the operating system on every change to the page; the operating system can then check if the instruction causing the exception affected the specific location or not.

A downside of paging is that while the management of physical memory becomes simpler, the management of the virtual address space becomes more challenging. Compilers typically expect the execution stack to be contiguous (in virtual addresses) and of arbitrary size; each new procedure call assumes the memory for the stack is available. Likewise, the runtime library for dynamic memory allocation typically expects a contiguous heap. In a single-threaded process, we can place the stack and heap at opposite ends of the virtual address space, and have them grow towards each other, as shown in Figure 8.5. However, with multiple threads per process, we need multiple thread stacks, each with room to grow.

This becomes even more of an issue with 64-bit virtual address spaces. The size of the page table is proportional to the size of the virtual address space, not to the size of physical memory. The more sparse the virtual address space, the more overhead is needed for the page table. Most of the entries will be invalid, representing parts of the virtual address space that are not in use, but physical memory is still needed for all of those page table entries.

We can reduce the space taken up by the page table by choosing a larger page frame. How big should a page frame be? A larger page frame can waste space if a process does not use all of the memory inside the frame. This is called *internal fragmentation*. Fixed-size chunks are easier to allocate, but waste space if the entire chunk is not used. Unfortunately, this means that with paging, either pages are very large (wasting space due to internal fragmentation), or the page table is very large (wasting space), or both. For example, with 16 KB pages and a 64 bit virtual address space, we might need 2^{50} page table entries!

8.2.3 Multi-Level Translation

If you were to design an efficient system for doing a lookup on a sparse keyspace, you probably would not pick a simple array. A tree or a hash table are more appropriate, and indeed, modern systems use both. We focus in this subsection on trees; we discuss hash tables afterwards.

Many systems use tree-based address translation, although the details vary from system to system, and the terminology can be a bit confusing. Despite the differences, the systems we are about to describe have similar properties. They support coarse and fine-grained memory protection and memory sharing, flexible memory placement, efficient memory allocation, and efficient lookup for sparse address spaces, even for 64-bit machines.

Almost all multi-level address translation systems use paging as the lowest level of the tree. The main differences between systems are in how they reach the page table at the leaf of the tree — whether using segments plus paging, or multiple levels of paging, or segments plus multiple levels of paging. There are several reasons for this:

- **Efficient memory allocation.** By allocating physical memory in fixed-size page frames, management of free space can use a simple bitmap.
- **Efficient disk transfers.** Hardware disks are partitioned into fixed-sized regions called sectors; disk sectors must be read or written in their entirety. By making the page size a multiple of the disk sector, we simplify transfers to and from memory, for loading programs into memory, reading and writing files, and in using the disk to simulate a larger memory than is physically present on the machine.
- **Efficient lookup.** We will describe in the next section how we can use a cache called a translation lookaside buffer to make lookups fast in the common case; the translation buffer caches lookups on a page by page basis. Paging also allows the lookup tables to be more compact, especially important at the lowest level of the tree.
- **Efficient reverse lookup.** Using fixed-sized page frames also makes it easy to implement the core map, to go from a physical page frame to the set of virtual addresses that share the same frame. This will be crucial for implementing the illusion of an infinite virtual memory in the next chapter.
- **Page-granularity protection and sharing.** Typically, every table entry at every level of the tree will have its own access permissions, enabling both coarse-grained and fine-grained sharing, down to the level of the individual page frame.

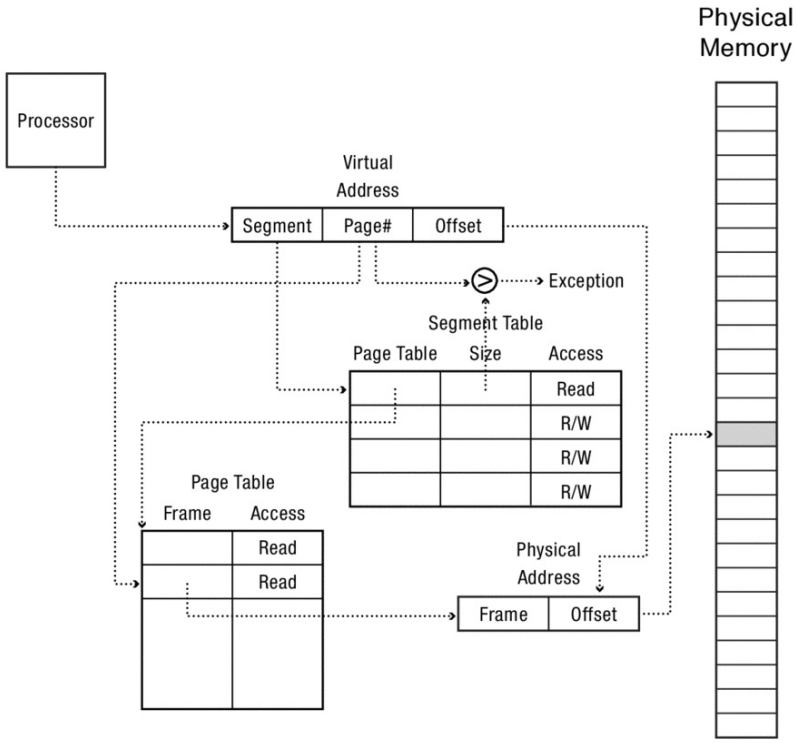

Figure 8.7: Address translation with paged segmentation. The virtual address has three components: a segment number, a virtual page number within the segment, and an offset within the page. The segment number indexes into a segment table that yields the page table for that segment. The page number from the virtual address indexes into the page table (from the segment table) to yield a page frame in physical memory. The physical address is the physical page frame from the page table, concatenated with the page offset from the virtual address. The operating system can restrict access to an entire segment, e.g., to prevent writes to the code segment, or to an individual page, e.g., to implement copy-on-write.

Paged Segmentation

Let us start a system with only two levels of a tree. With *paged segmentation*, memory is segmented, but instead of each segment table entry pointing directly to a contiguous region of physical memory, each segment table entry points to a page table, which in turn points to the memory backing that segment. The segment table entry "bound" describes the page table length, that is, the length of the segment in pages. Because paging is used at the lowest level, all segment lengths are some multiple of the page size. Figure 8.7 illustrates translation with paged segmentation.

Although segment tables are sometimes stored in special hardware registers, the page tables for each segment are quite a bit larger in aggregate, and so they are normally stored in physical memory. To keep the memory allocator simple, the maximum segment size is usually chosen to allow the page table for each segment to be a small multiple of the page size.

For example, with 32-bit virtual addresses and 4 KB pages, we might set aside the upper ten bits for the segment number, the next ten bits for the page number, and twelve bits for the page offset. In this case, if each page table entry is four bytes, the page table for each segment would exactly fit into one physical page frame.

Multi-Level Paging

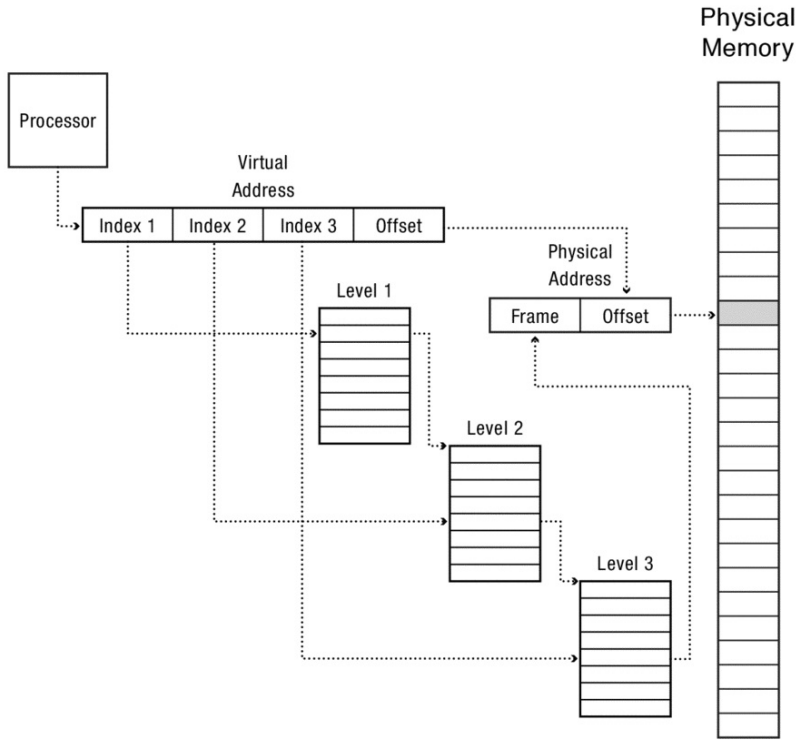

Figure 8.8: Address translation with three levels of page tables. The virtual address has four components: an index into each level of the page table and an offset within the physical page frame.

A nearly equivalent approach to paged segmentation is to use multiple levels of page tables. On the Sun Microsystems SPARC processor for example, there are three levels of page table. As shown in Figure 8.8, the top-level page table contains entries, each of which points to a second-level page table whose entries are pointers to page tables. On the SPARC, as with most other systems that use multiple levels of page tables, each level of

page table is designed to fit in a physical page frame. Only the top-level page table must be filled in; the lower levels of the tree are allocated only if those portions of the virtual address space are in use by a particular process. Access permissions can be specified at each level, and so sharing between processes is possible at each level.

Multi-Level Paged Segmentation

We can combine these two approaches by using a segmented memory where each segment is managed by a multi-level page table. This is the approach taken by the x86, for both its 32-bit and 64-bit addressing modes.

We describe the 32-bit case first. The x86 terminology differs slightly from what we have used here. The x86 has a per-process *Global Descriptor Table* (GDT), equivalent to a segment table. The GDT is stored in memory; each entry (descriptor) points to the (multi-level) page table for that segment along with the segment length and segment access permissions. To start a process, the operating system sets up the GDT and initializes a register, the *Global Descriptor Table Register (GDTR)*, that contains the address and length of the GDT.

Because of its history, the x86 uses separate processor registers to specify the segment number (that is, the index into the GDT) and the virtual address for use by each instruction. For example, on the "32-bit" x86, there is both a segment number and 32 bits of virtual address within each segment. On the 64-bit x86, the virtual address within each segment is extended to 64 bits. Most applications only use a few segments, however, so the per-process segment table is usually short. The operating system kernel has its own segment table; this is set up to enable the kernel to access, with virtual addresses, all of the per-process and shared segments on the system.

For encoding efficiency, the segment register is often implicit as part of the instruction. For example, the x86 stack instructions such as push and pop assume the stack segment (the index stored in the stack segment register), branch instructions assume the code segment (the index stored in the code segment register), and so forth. As an optimization, whenever the x86 initializes a code, stack, or data segment register it also reads the GDT entry (that is, the top-level page table pointer and access permissions) into the processor, so the processor can go directly to the page table on each reference.

Many instructions also have an option to specify the segment index explicitly. For example, the ljmp, or long jump, instruction changes the program counter to a new segment number and offset within that segment.

For the 32-bit x86, the virtual address space within a segment has a two-level page table. The first 10 bits of the virtual address index the top level page table, called the *page directory*, the next 10 bits index the second level page table, and the final 12 bits are the offset within a page. Each page table entry takes four bytes and the page size is 4 KB, so the top-level page table and each second-level page table fits in a single physical page. The number of second-level page tables needed depends on the length of the segment; they are not needed to map empty regions of virtual address space. Both the top-level and second-level page table entries have permissions, so fine-grained protection and sharing is possible within a segment.

Today, the amount of memory per computer is often well beyond what can 32 bits can address; for example, a high-end server could have two terabytes of physical memory. For the 64-bit x86, virtual addresses within a segment can be up to 64 bits. However, to simplify address translation, current processors only allow 48 bits of the virtual address to be used; this is sufficient to map 128 terabytes, using four levels of page tables. The lower levels of the page table tree are only filled in if that portion of the virtual address space is in use.

As an optimization, the 64-bit x86 has the option to eliminate one or two levels of the page table. Each physical page frame on the x86 is 4 KB. Each page of fourth level page table maps 2 MB of data, and each page of the third level page table maps 1 GB of data. If the operating system places data such that the entire 2 MB covered by the fourth level page table is allocated contiguously in physical memory, then the page table entry one layer up can be marked to point directly to this region instead of to a page table. Likewise, a page of third level page table can be omitted if the operating system allocates the process a 1 GB chunk of physical memory. In addition to saving space needed for page table mappings, this improves translation buffer efficiency, a point we will discuss in more detail in the next section.

8.2.4 Portability

The diversity of different translation mechanisms poses a challenge to the operating system designer. To be widely used, we want our operating system to be easily portable to a wide variety of different processor architectures. Even within a given processor family, such as an x86, there are a number of different variants that an operating system may need to support. Main memory density is increasing both the physical and virtual address space by almost a bit per year. In other words, for a multi-level page table to be able to map all of memory, an extra level of the page table is needed every decade just to keep up with the increasing size of main memory.

A further challenge is that the operating system often needs to keep two sets of books with respect to address translation. One set of books is the hardware view — the processor consults a set of segment and multi-level page tables to be able to correctly and securely execute instructions and load and store data. A different set of books is the operating system view of the virtual address space. To support features such as copy-on-write, zero-on-reference, and fill-on-reference, as well as other applications we will describe in later chapters, the operating system must keep track of additional information about each virtual page beyond what is stored in the hardware page table.

This software memory management data structures mirror, but are not identical to, the hardware structures, consisting of three parts:

- **List of memory objects.** Memory objects are logical segments. Whether or not the underlying hardware is segmented, the kernel memory manager needs to keep track of which memory regions represent which underlying data, such as program code, library code, shared data between two or more processes, a copy-on-write region, or a memory-mapped file. For example, when a process starts up, the kernel can check the object list to see if the code is already in memory; likewise, when a process opens a

library, it can check if it has already been linked by some other process. Similarly, the kernel can keep reference counts to determine which memory regions to reclaim on process exit.

- **Virtual to physical translation.** On an exception, and during system call parameter copying, the kernel needs to be able to translate from a process's virtual addresses to its physical locations. While the kernel could use the hardware page tables for this, the kernel also needs to keep track of whether an invalid page is truly invalid, or simply not loaded yet (in the case of fill-on-reference) or if a read-only page is truly read-only or just simulating a data breakpoint or a copy-on-write page.
- **Physical to virtual translation.** We referred to this above as the *core map*. The operating system needs to keep track of the processes that map to a specific physical memory location, to ensure that when the kernel updates a page's status, it can also updated every page table entry that refers to that physical page.

The most interesting of these are the data structures used for the virtual to physical translation. For the software page table, we have all of the same options as before with respect to segmentation and multiple levels of paging, as well as some others. The software page table need not use the same structure as the underlying hardware page table; indeed, if the operating system is to be easily portable, the software data structures may be quite different from the underlying hardware.

Linux models the operating system's internal address translation data structures after the x86 architecture of segments plus multi-level page tables. This has made porting Linux to new x86 architectures relatively easy, but porting Linux to other architectures somewhat more difficult.

A different approach, taken first in a research system called Mach and later in Apple OS X, is to use a hash table, rather than a tree, for the software translation data. For historical reasons, the use of a hash table for paged address translation is called an *inverted page table*. Particularly as we move to deeper multi-level page tables, using a hash table for translation can speed up translation.

With an inverted page table, the virtual page number is hashed into a table of size proportional to the number of physical page frames. Each entry in the hash table contains tuples of the form (in the figure, the physical page is implicit):

```
inverted page table entry = {
    process or memory object ID,
    virtual page number,
    physical page frame number,
    access permissions
}
```

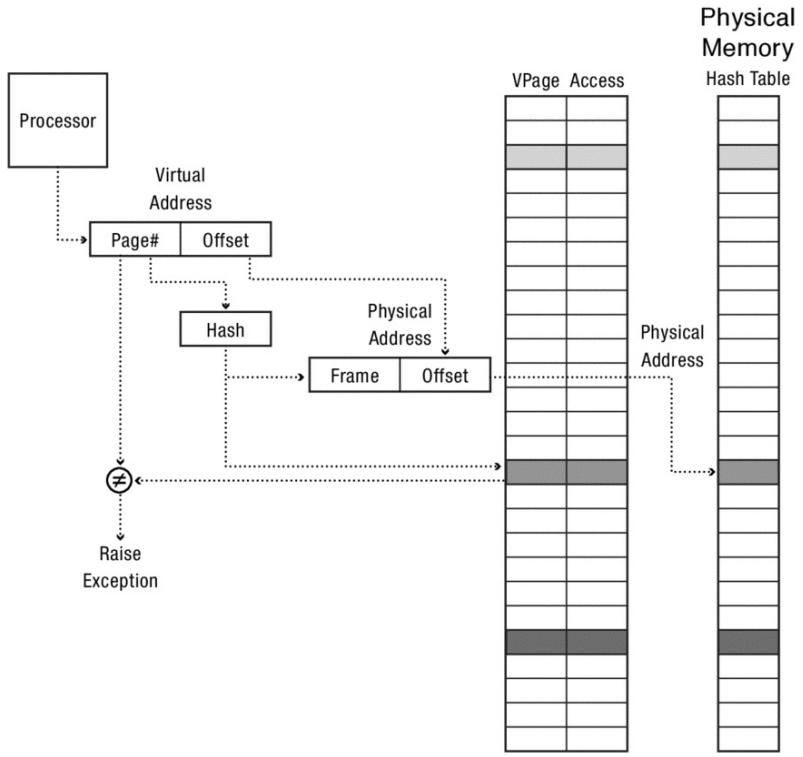

Figure 8.9: Address translation with a software hash table. The hardware page tables are omitted from the picture. The virtual page number is hashed; this yields a position in the hash table that indicates the physical page frame. The virtual page number must be checked against the contents of the hash entry to handle collisions and to check page access permissions.

As shown in Figure 8.9, if there is a match on both the virtual page number and the process ID, then the translation is valid. Some systems do a two stage lookup: they first map the virtual address to a memory object ID, and then do the hash table lookup on the relative virtual address within the memory object. If memory is mostly shared, this can save space in the hash table without unduly slowing the translation.

An inverted page table does need some way to handle hash collisions, when two virtual addresses map to the same hash table entry. Standard techniques — such as chaining or rehashing — can be used to handle collisions.

A particularly useful consequence of having a portability layer for memory management is that the contents of the hardware multi-level translation table can be treated as a *hint*. A hint is a result of some computation whose results may no longer be valid, but where using an invalid hint will trigger an exception.

With a portability layer, the software page table is the ground truth, while the hardware

page table is a hint. The hardware page table can be safely used, provided that the translations and permissions are a *subset* of the translations in the software page table.

Is an inverted page table enough?

The concept of an inverted page table raises an intriguing question: do we need to have a multi-level page table in hardware? Suppose, in hardware, we hash the virtual address. But instead of using the hash value to look up in a table where to find the physical page frame, suppose we just use the hash value *as* the physical page. For this to work, we need the hash table size to have exactly as many entries as physical memory page frames, so that there is a one-to-one correspondence between the hash table entry and the page frame.

We still need a table to store permissions and to indicate which virtual page is stored in each entry; if the process does not have permission to access the page, or if two virtual pages hash to the same physical page, we need to be able to detect this and trap to the operating system kernel to handle the problem. This is why a hash table for managing memory is often called called an *inverted page table*: the entries in the table are virtual page numbers, not physical page numbers. The physical page number is just the position of that virtual page in the table.

The drawback to this approach? Handling hash collisions becomes much harder. If two pages hash to the same table entry, only one can be stored in the physical page frame. The other has to be elsewhere — either in a secondary hash table entry or possibly stored on disk. Copying in the new page can take time, and if the program is unlucky enough to need to simultaneously access two virtual pages that both hash to the same physical page, the system will slow down even further. As a result, on modern systems, inverted page tables are typically used in software to improve portability, rather than in hardware, to eliminate the need for multi-level page tables.

8.3 Towards Efficient Address Translation

At this point, you should be getting a bit antsy. After all, most of the hardware mechanisms we have described involve at least two and possibly as many as four memory extra references, on each instruction, before we even reach the intended physical memory location! It should seem completely impractical for a processor to do several memory lookups on every instruction fetch, and even more that for every instruction that loads or stores data.

In this section, we will discuss how to improve address translation performance without changing its logical behavior. In other words, despite the optimization, every virtual address is translated to exactly the same physical memory location, and every permission exception causes a trap, exactly as would have occurred without the performance optimization.

For this, we will use a *cache*, a copy of some data that can be accessed more quickly than the original. This section concerns how we might use caches to improve translation performance. Caches are widely used in computer architecture, operating systems, distributed systems, and many other systems; in the next chapter, we discuss more generally when caches work and when they do not. For now, however, our focus is just on the use of caches for reducing the overhead of address translation. There is a reason for this: the very first hardware caches were used to improve translation performance.

8.3.1 Translation Lookaside Buffers

If you think about how a processor executes instructions with address translation, there are some obvious ways to improve performance. After all, the processor normally executes instructions in a sequence:

```
...
add r1, r2
mult r1, 2
...
```

The hardware will first translate the program counter for the add instruction, walking the multi-level translation table to find the physical memory where the add instruction is stored. When the program counter is incremented, the processor must walk the multiple levels again to find the physical memory where the mult instruction is stored. If the two instructions are on the same page in the virtual address space, then they will be on the same page in physical memory. The processor will just repeat the same work — the table walk will be exactly the same, and again for the next instruction, and the next after that.

A *translation lookaside buffer (TLB)* is a small hardware table containing the results of recent address translations. Each entry in the TLB maps a virtual page to a physical page:

```
TLB entry = {
    virtual page number,
    physical page frame number,
    access permissions
}
```

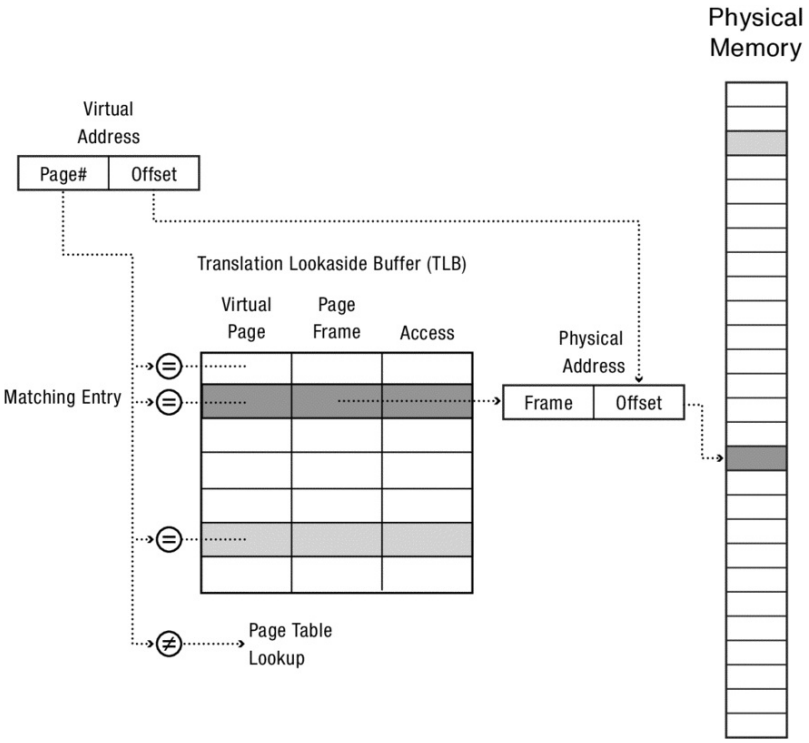

Figure 8.10: Operation of a translation lookaside buffer. In the diagram, each virtual page number is checked against all of the entries in the TLB at the same time; if there is a match, the matching table entry contains the physical page frame and permissions. If not, the hardware multi-level page table lookup is invoked; note the hardware page tables are omitted from the picture.

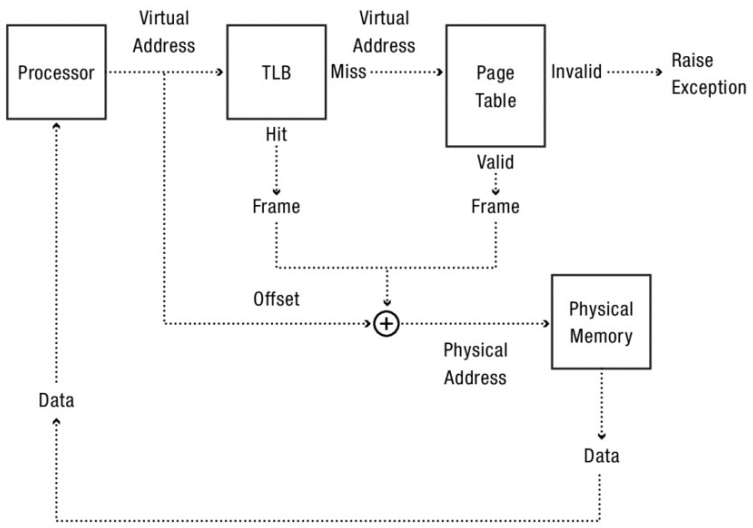

Figure 8.11: Combined operation of a translation lookaside buffer and hardware page tables.

Instead of finding the relevant entry by a multi-level lookup or by hashing, the TLB hardware (typically) checks all of the entries simultaneously against the virtual page. If there is a match, the processor uses that entry to form the physical address, skipping the rest of the steps of address translation. This is called a *TLB hit*. On a TLB hit, the hardware still needs to check permissions, in case, for example, the program attempts to write to a code-only page or the operating system needs to trap on a store instruction to a copy-on-write page.

A *TLB miss* occurs if none of the entries in the TLB match. In this case, the hardware does the full address translation in the way we described above. When the address translation completes, the physical page is used to form the physical address, and the translation is installed in an entry in the TLB, replacing one of the existing entries. Typically, the replaced entry will be one that has not been used recently.

The TLB lookup is illustrated in Figure 8.10, and Figure 8.11 shows how a TLB fits into the overall address translation system.

Although the hardware cost of a TLB might seem large, it is modest compared to the potential gain in processor performance. To be useful, the TLB lookup needs to be much more rapid than doing a full address translation; thus, the TLB table entries are implemented in very fast, on-chip static memory, situated near the processor. In fact, to keep lookups rapid, many systems now include multiple levels of TLB. In general, the smaller the memory, the faster the lookup. So, the first level TLB is small and close to the

processor (and often split for engineering reasons into one for instruction lookups and a separate one for data lookups). If the first level TLB does not contain the translation, a larger second level TLB is consulted, and the full translation is only invoked if the translation misses both levels. For simplicity, our discussion will assume a single-level TLB.

A TLB also requires an address comparator for each entry to check in parallel if there is a match. To reduce this cost, some TLBs are *set associative*. Compared to fully associative TLBs, set associative ones need fewer comparators, but they may have a higher miss rate. We will discuss set associativity, and its implications for operating system design, in the next chapter.

What is the cost of address translation with a TLB? There are two factors. We pay the cost of the TLB lookup regardless of whether the address is in the TLB or not; in the case of an unsuccessful TLB lookup, we also pay the cost of the full translation. If P(hit) is the likelihood that the TLB has the entry cached:

$$\text{Cost (address translation)} = \text{Cost (TLB lookup)} + \text{Cost (full translation)} \times (1 - P(\text{hit}))$$

In other words, the processor designer needs to include a sufficiently large TLB that most addresses generated by a program will hit in the TLB, so that doing the full translation is the rare event. Even so, TLB misses are a significant cost for many applications.

Software-loaded TLB

If the TLB is effective at amortizing the cost of doing a full address translation across many memory references, we can ask a radical question: do we need hardware multi-level page table lookup on a TLB miss? This is the concept behind a software-loaded TLB. A TLB hit works as before, as a fast path. On a TLB miss, instead of doing hardware address translation, the processor traps to the operating system kernel. In the trap handler, the kernel is responsible for doing the address lookup, loading the TLB with the new translation, and restarting the application.

This approach dramatically simplifies the design of the operating system, because it no longer needs to keep two sets of page tables, one for the hardware and one for itself. On a TLB miss, the operating system can consult its own portable data structures to determine what data should be loaded into the TLB.

Although convenient for the operating system, a software-loaded TLB is somewhat slower for executing applications, as the cost of trapping to the kernel is significantly more than the cost of doing hardware address translation. As we will see in the next chapter, the contents of page table entries can be stored in on-chip hardware caches; this means that even on a TLB miss, the hardware can often find every level of the multi-level page table already stored in an on-chip cache, but not in the TLB. For example, a TLB miss on a modern generation x86 can be completed in the best case in the equivalent of 17 instructions. By contrast, a trap to the operating system kernel will take several hundred to a few thousand instructions to process, even in the best case.

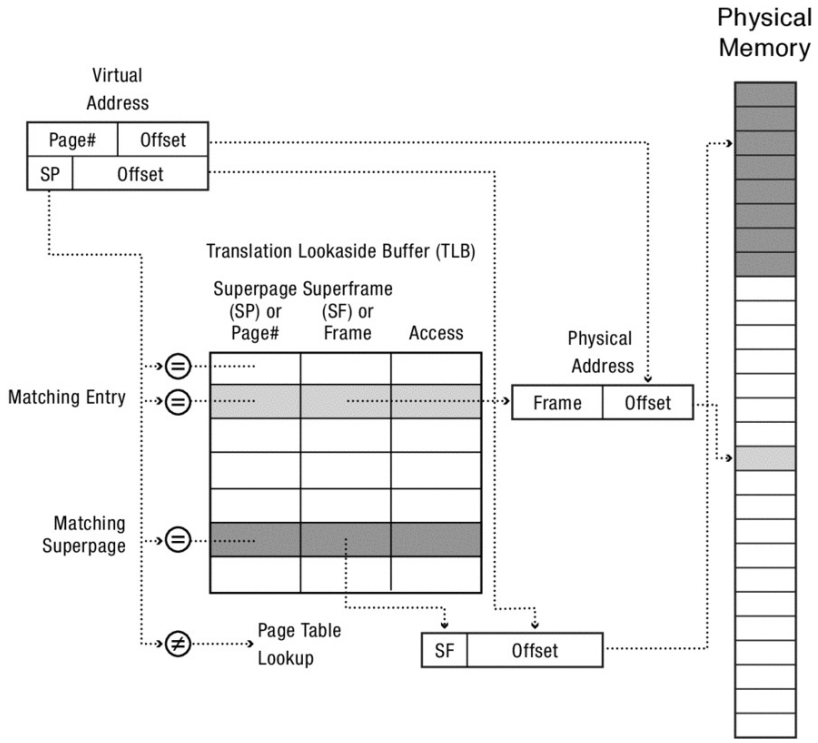

Figure 8.12: Operation of a translation lookaside buffer with superpages. In the diagram, some entries in the TLB can be superpages; these match if the virtual page is in the superpage. The superpage in the diagram covers an entire memory segment, but this need not always be the case.

8.3.2 Superpages

One way to improve the TLB hit rate is using a concept called superpages. A *superpage* is a set of contiguous pages in physical memory that map a contiguous region of virtual memory, where the pages are aligned so that they share the same high-order (superpage) address. For example, an 8 KB superpage would consist of two adjacent 4 KB pages that lie on an 8 KB boundary in both virtual and physical memory. Superpages are at the

discretion of the operating system — small programs or memory segments that benefit from a smaller page size can still operate with the standard, smaller page size.

Superpages complicate operating system memory allocation by requiring the system to allocate chunks of memory in different sizes. However, the upside is that a superpage can drastically reduce the number of TLB entries needed to map large, contiguous regions of memory. Each entry in the TLB has a flag, signifying whether the entry is a page or a superpage. For superpages, the TLB matches the superpage number — that is, it ignores the portion of the virtual address that is the page number within the superpage. This is illustrated in Figure 8.12.

To make this concrete, the x86 skips one or two levels of the page table when there is a 2 MB or 1 GB region of physical memory that is mapped as a unit. When the processor references one of these regions, only a single entry is loaded into the TLB. When looking for a match against a superpage, the TLB only considers the most significant bits of the address, ignoring the offset within the superpage. For a 2 MB superpage, the offset is the lowest 21 bits of the virtual address. For a 1 GB superpage it is the lowest 30 bits.

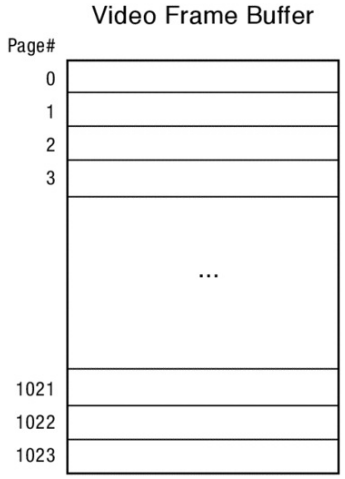

Figure 8.13: Layout of a high-resolution frame buffer in physical memory. Each line of the pixel display can take up an entire page, so that adjacent pixels in the vertical dimension lie on different pages.

A common use of superpages is to map the frame buffer for the computer display. When redrawing the screen, the processor may touch every pixel; with a high-resolution display, this can involve stepping through many megabytes of memory. If each TLB entry maps a 4 KB page, even a large on-chip TLB with 256 entries would only be able to contain mappings for 1 MB of the frame buffer at the same time. Thus, the TLB would need to repeatedly do page table lookups to pull in new TLB entries as it steps through memory. An even worse case occurs when drawing a vertical line. The frame buffer is a two-dimensional array in row-major order, so that each horizontal line of pixels is on a separate page. Thus, modifying each separate pixel in a vertical line would require loading a separate TLB entry! With superpages, the entire frame buffer can be mapped with a single TLB entry, leaving more room for the other pages needed by the application.

Similar issues occur with large matrices in scientific code.

8.3.3 TLB Consistency

Whenever we introduce a cache into a system, we need to consider how to ensure consistency of the cache with the original data when the entries are modified. A TLB is no exception. For secure and correct program execution, the operating system must ensure that the each program sees its memory and no one else's. Any inconsistency between the TLB, the hardware multi-level translation table, and the portable operating system layer is a potential correctness and security flaw.

There are three issues to consider:

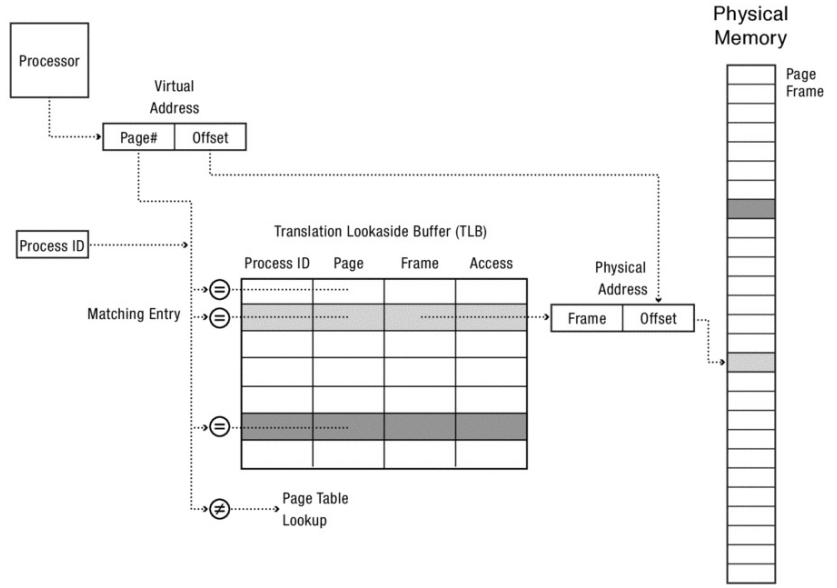

Figure 8.14: Operation of a translation lookaside buffer with process ID's. The TLB contains entries for multiple processes; only the entries for the current process are valid. The operating system kernel must change the current process ID when performing a context switch between processes.

- **Process context switch.** What happens on a process context switch? The virtual addresses of the old process are no longer valid, and should no longer be valid, for the new process. Otherwise, the new process will be able to read the old process's

data structures, either causing the new process to crash, or potentially allowing it to scavenge sensitive information such as passwords stored in memory.

On a context switch, we need to change the hardware page table register to point to the new process's page table. However, the TLB also contains copies of the old process's page translations and permissions. One approach is to *flush* the TLB — discard its contents — on every context switch. Since emptying the cache carries a performance penalty, modern processors have a *tagged TLB*, shown in Figure 8.14. Entries in a tagged TLB contain the process ID that produced each translation:

```
tagged TLB entry = {
    process ID,
    virtual page number,
    physical page frame number,
    access permissions
}
```

With a tagged TLB, the operating system stores the current process ID in a hardware register on each context switch. When performing a lookup, the hardware ignores TLB entries from other processes, but it can reuse any TLB entries that remain from the last time the current process executed.

- **Permission reduction.** What happens when the operating system modifies an entry in a page table? For the processor's regular data cache of main memory, special-purpose hardware keeps cached data consistent with the data stored in memory. However, hardware consistency is not usually provided for the TLB; keeping the TLB consistent with the page table is the responsibility of the operating system kernel.

  Software involvement is needed for several reasons. First, page table entries can be shared between processes, so a single modification can affect multiple TLB entries (e.g., one for each process sharing the page). Second, the TLB contains only the virtual to physical page mapping — it does not record the address where the mapping came from, so it cannot tell if a write to memory would affect a TLB entry. Even if it did track this information, most stores to memory do not affect the page table, so repeatedly checking each memory store to see if it affects any TLB entry would involve a large amount of overhead that would rarely be needed.

  Instead, whenever the operating system changes the page table, it ensures that the TLB does not contain an incorrect mapping.

  Nothing needs to be done when the operating system *adds permissions* to a portion of the virtual address space. For example, the operating system might dynamically extend the heap or the stack by allocating physical memory and changing invalid page table entries to point to the new memory, or the operating system might change a page from read-only to read-write. In these cases, the TLB can be left alone because any references that require the new permissions will either cause the hardware load the new entries or cause an exception, allowing the operating system to load the new entries.

  However, if the operating system needs to *reduce permissions* to a page, then the kernel needs to ensure the TLB does not have a copy of the old translation before resuming the process. If the page was shared, the kernel needs to ensure that the TLB does not have the copy for any of the process ID's that might have referenced the page. For example, to mark a region of memory as copy-on-write, the operating system must reduce permissions to the region to read-only, and it must remove any entries for that region from the TLB, since the old TLB entries would still be read-write.

  Early computers discarded the entire contents of the TLB whenever there was a change to a page table, but more modern architectures, including the x86 and the ARM, support the removal of individual TLB entries.

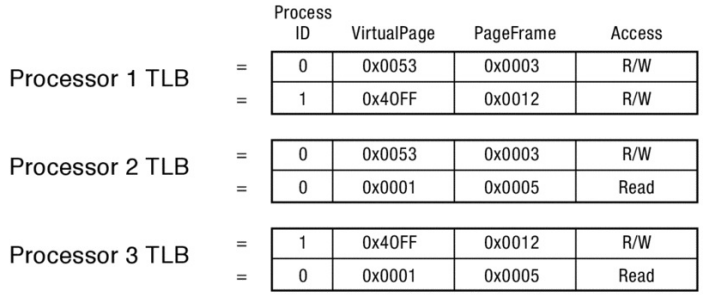

Figure 8.15: Illustration of the need for TLB shootdown to preserve correct translation behavior. In order for processor 1 to change the translation for page 0x53 in process 0 to read-only, it must remove the entry from its TLB, and it must ensure that no other processor has the old translation in its TLB. To do this, it sends an interprocessor interrupt to each processor, requesting it to remove the old translation. The operating system does not know if a particular TLB contains an entry (e.g., processor 3's TLB does not contain page 0x53), so it must remove it from all TLBs. The shootdown is complete only when all processors have verified that the old translation has been removed.

- **TLB shootdown.** On a multiprocessor, there is a further complication. Any processor in the system may have a cached copy of a translation in its TLB. Thus, to be safe and correct, whenever a page table entry is modified, the corresponding entry in *every* processor's TLB has to be discarded before the change will take effect. Typically, only the current processor can invalidate its own TLB, so removing the entry from all processors on the system requires that the operating system interrupt each processor and request that it remove the entry from its TLB.

  This heavyweight operation has its own name: it is a *TLB shootdown*, illustrated in Figure 8.15. The operating system first modifies the page table, then sends a TLB shootdown request to all of the other processors. Once another processor has ensured that its TLB has been cleaned of any old entries, that processor can resume. The original processor can continue only when *all* of the processors have acknowledged

removing the old entry from their TLB. Since the overhead of a TLB shootdown increases linearly with the number of processors on the system, many operating systems batch TLB shootdown requests, to reduce the frequency of interprocess interrupts at some increased cost in latency to complete the shootdown.

8.3.4 Virtually Addressed Caches

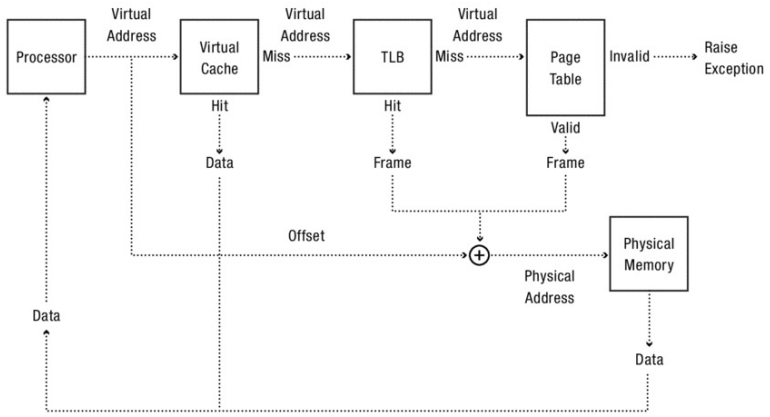

Figure 8.16: Combined operation of a virtually addressed cache, translation lookaside buffer, and hardware page table.

Another step to improving the performance of address translation is to include a virtually addressed cache *before* the TLB is consulted, as shown in Figure 8.16. A virtually addressed cache stores a copy of the contents of physical memory, indexed by the virtual address. When there is a match, the processor can use the data immediately, without waiting for a TLB lookup or page table translation to generate a physical address, and without waiting to retrieve the data from main memory. Almost all modern multicore chips include a small, virtually addressed on-chip cache near each processor core. Often, like the TLB, the virtually addressed cache will be split in half, one for instruction lookups and one for data.

The same consistency issues that apply to TLBs also apply to virtually addressed caches:

- **Process context switch.** Entries in the virtually addressed cache must either be either with the process ID or they must be invalidated on a context switch to prevent the new process from accessing the old process's data.
- **Permission reduction and shootdown.** When the operating system changes the permission for a page in the page table, the virtual cache will not reflect that change. Invalidating the affected cache entries would require either flushing the entire cache or finding all memory locations stored in the cache on the affected page, both relatively heavyweight operations.

  Instead, most systems with virtually addressed caches use them in tandem with the TLB. Each virtual address is looked up in both the cache and the TLB at the same time; the TLB specifies the permissions to use, while the cache provides the data if the access is permitted. This way, only the TLB's permissions need to be kept up to date. The TLB and virtual cache are co-designed to take the same amount of time to perform a lookup, so the processor does not stall waiting for the TLB.

A further issue is aliasing. Many operating systems allow processes sharing memory to use different virtual addresses to refer to the same memory location. This is called a *memory address alias*. Each process will have its own TLB entry for that memory, and the virtual cache may store a copy of the memory for each process. The problem occurs when one process modifies its copy; how does the system know to update the other copy?

The most common solution to this issue is to store the physical address along with the virtual address in the virtual cache. In parallel with the virtual cache lookup, the TLB is consulted to generate the physical address and page permissions. On a store instruction modifying data in the virtual cache, the system can do a reverse lookup to find all the entries that match the same physical address, to allow it to update those entries.

8.3.5 Physically Addressed Caches

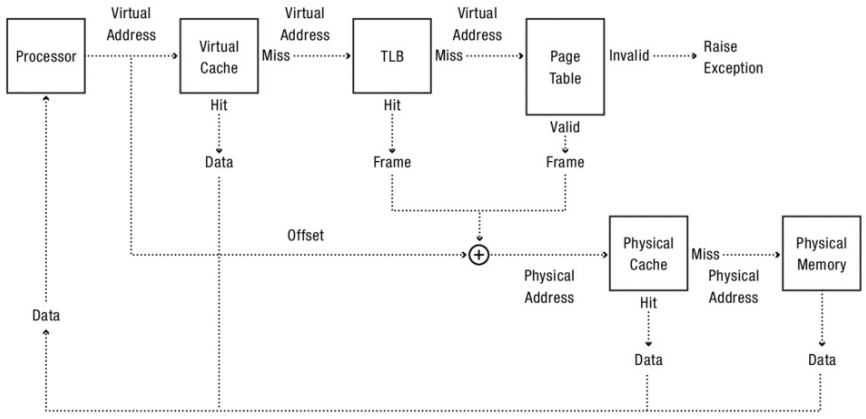

Figure 8.17: Combined operation of a virtually addressed cache, translation lookaside buffer, hardware page table, and physically addressed cache.

Many processor architectures include a physically addressed cache that is consulted as a second-level cache after the virtually addressed cache and TLB, but before main memory.

This is illustrated in Figure 8.17. Once the physical address of the memory location is formed from the TLB lookup, the second-level cache is consulted. If there is a match, the value stored at that location can be returned directly to the processor without the need to go to main memory.

With today's chip densities, an on-chip physically addressed cache can be quite large. In fact, many systems include both a second-level and a third-level physically addressed cache. Typically, the second-level cache is per-core and is optimized for latency; a typical size is 256 KB. The third-level cache is shared among all of the cores on the same chip and will be optimized for size; it can be as large as 2 MB on a modern chip. In other words, the entire UNIX operating system from the 70's, and all of its applications, would fit on a single modern chip, with no need to ever go to main memory.

Together, these physically addressed caches serve a dual purpose:

- **Faster memory references.** An on-chip physically addressed cache will have a lookup latency that is ten times (2nd level) or three times (3rd level) faster than main memory.
- **Faster TLB misses.** In the event of a TLB miss, the hardware will generate a sequence of lookups through its multiple levels of page tables. Because the page tables are stored in physical memory, they can be cached. Thus, even a TLB miss and page table lookup may be handled entirely on chip.

8.4 Software Protection

An increasing number of systems complement hardware-based address translation with software-based protection mechanisms. Obviously, software-only protection is possible. A machine code interpreter, implemented in software, can simulate the exact behavior of hardware protection. The interpreter could fetch each instruction, interpret it, look each address up in a page table to determine if the instruction is permitted, and if so, execute the instruction. Of course, that would be very slow!

In this section, we ask: are there practical software techniques to execute code within a restricted domain, without relying on hardware address translation? The focus of our discussion will be on using software for providing an efficient protection boundary, as a way of improving computer security. However, the techniques we describe can also be used to provide other operating system services, such as copy-on-write, stack extensibility, recoverable memory, and user-level virtual machines. Once you have the infrastructure to reinterpret references to code and data locations, whether in software or hardware, a number of services become possible.

Hardware protection is nearly universal on modern computers, so it is reasonable to ask, why do we need to implement protection in software?

- **Simplify hardware.** One goal is simple curiosity. Do we really need hardware address translation, or is it just an engineering tradeoff? If software can provide efficient protection, we could eliminate a large amount of hardware complexity and runtime overhead from computers, with a substantial increase in flexibility.
- **Application-level protection.** Even if we need hardware address translation to protect the operating system from misbehaving applications, we often want to run untrusted code within an application. An example is inside a web browser; web pages can contain code to configure the display for a web site, but the browser needs to protect itself against malicious or buggy code provided by web sites.
- **Protection inside the kernel.** We also sometimes need to run untrusted, or at least less trusted, code inside kernel. Examples include third-party device drivers and code to customize the behavior of the operating system on behalf of applications. Because the kernel runs with the full capability of the entire machine, any user code run inside the kernel must be protected in software rather than in hardware.
- **Portable security.** The proliferation of consumer devices poses a challenge to application portability. No single operating system runs on every embedded sensor, smartphone, tablet, netbook, laptop, desktop, and server machine. Applications that want to run across a wide range of devices need a common runtime environment that isolates the application from the specifics of the underlying operating system and hardware device. Providing protection as part of the runtime system means that users can download and run applications without concern that the application will corrupt the underlying operating system.

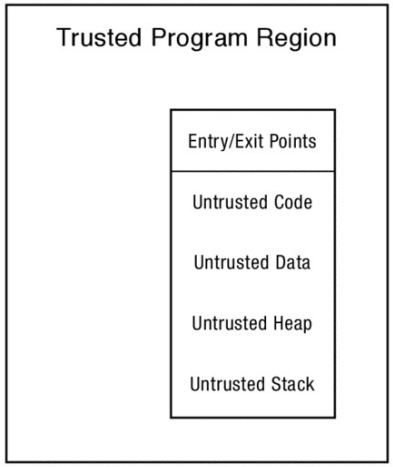

Figure 8.18: Execution of untrusted code inside a region of trusted code. The trusted region can be a process, such as a browser, executing untrusted JavaScript, or the trusted region can be the operating system kernel, executing untrusted packet filters or device drivers.

The need for software protection is widespread enough that it has its own term: how do we provide a software *sandbox* for executing untrusted code so that it can do its work without causing harm to the rest of the system?

8.4.1 Single Language Operating Systems

A very simple approach to software protection is to restrict all applications to be written in a single, carefully designed programming language. If the language and its environment permits only safe programs to be expressed, and the compiler and runtime system are trustworthy, then no hardware protection is needed.

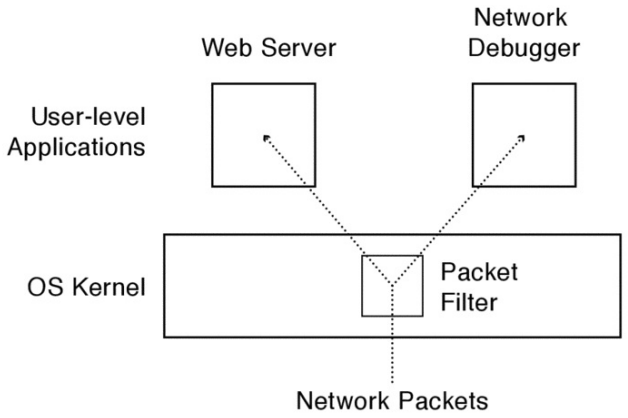

Figure 8.19: Execution of a packet filter inside the kernel. A packet filter can be installed by a network debugger to trace packets for a particular user or application. Packet headers matching the filter are copied to the debugger, while normal packet processing continues unaffected.

A practical example of this approach that is still in wide use is UNIX packet filters, shown in Figure 8.19. UNIX packet filters allow users to download code into the operating system kernel to customize kernel network processing. For example, a packet filter can be installed in the kernel to make a copy of packet headers arriving for a particular connection and to send those to a user-level debugger.

A UNIX packet filter is typically only a small amount of code, but because it needs to run in kernel-mode, the system cannot rely on hardware protection to prevent a misbehaving packet filter from causing havoc to unrelated applications. Instead, the system restricts the packet filter language to permit only safe packet filters. For example, filters may only branch on the contents of packets and no loops are allowed. Since the filters are typically short, the overhead of using an interpreted language is not prohibitive.

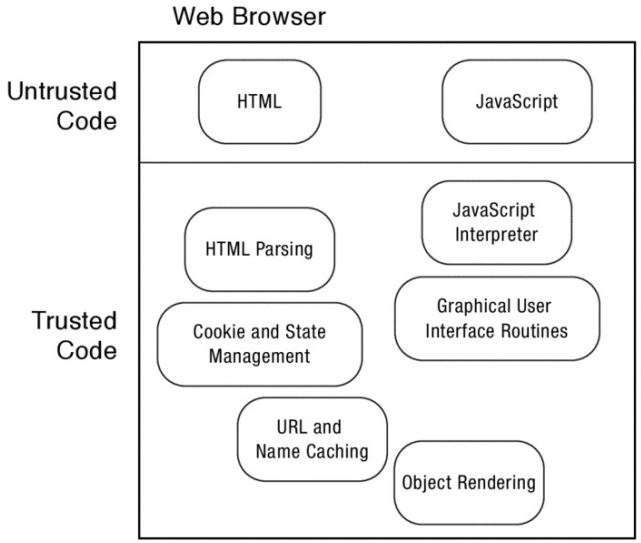

Figure 8.20: Execution of a JavaScript program inside a modern web browser. The JavaScript interpreter is responsible for containing effects of the JavaScript program to its specific page. JavaScript programs can call out to a broad set of routines in the browser, so these routines must also be protected against malicious JavaScript programs.

Another example of the same approach is the use of JavaScript in modern web browsers, illustrated in Figure 8.20. A JavaScript program customizes the user interface and presentation of a web site; it is provided by the web site, but it executes on the client machine inside the browser. As a result, the browser execution environment for JavaScript must prevent malicious JavaScript programs from taking control over the browser and possibly the rest of the client machine. Since JavaScript programs tend to be relatively short, they are often interpreted; JavaScript can also call into a predefined set of library routines. If a JavaScript program attempts to call a procedure that does not exist or reference arbitrary memory locations, the interpreter will cause a runtime exception and stop the program before any harm can be done.

Several early personal computers were single language systems with protection implemented in software rather than hardware. Most famously, the Xerox Alto research prototype used software and not hardware protection; the Alto inspired the Apple Macintosh, and the language it used, Mesa, was a forerunner of Java. Other systems included the Lisp Machine, a computer that executed only programs written in Lisp, and computers that executed only Smalltalk (a precursor to Python).

Language protection and garbage collection

JavaScript, Lisp, and Smalltalk all provide memory-compacting garbage collection for dynamically created data structures. One motivation for this is programmer convenience and to reduce avoidable programmer error. However, there is a close relationship between software protection and garbage collection. Garbage collection requires the runtime system to keep track of all valid pointers visible to the program, so that data structures can be relocated without affecting program behavior. Programs expressible in the language cannot point to or jump to arbitrary memory locations, as then the behavior of the program would be altered by the garbage collector. Every address generated by the program is necessarily within the region of the application's code, and every load and store instruction is to the program's data, and no one else's. In other words, this is exactly what is needed for software protection!

Unfortunately, language-based software protection has some practical limitations, so that on modern systems, it is often used in tandem with, rather than as a replacement for, hardware protection. Using an interpreted language seems like a safe option, but it requires trust in both the interpreter and its runtime libraries. An interpreter is a complex piece of software, and any flaw in the interpreter could provide a way for a malicious program to gain control over the process, that is, to escape its protection boundary. Such attacks are common for browsers running JavaScript, although over time JavaScript interpreters have become more robust to these types of attacks.

Worse, because running interpreted code is often slow, many interpreted systems put most of their functionality into system libraries that can be compiled into machine code and run directly on the processor. For example, commercial web browsers provide JavaScript programs a huge number of user interface objects, so that the interpreted code is just a small amount of glue. Unfortunately, this raises the attack surface — any library routine that does not completely protect itself against malicious use can be a vector for the program to escape its protection. For example, a JavaScript program could attempt to cause a library routine to overwrite the end of a buffer, and depending on what was stored in memory, that might provide a way for the JavaScript program to gain control of the system. These types of attacks against JavaScript runtime libraries are widespread.

This leads most systems to use both hardware and software protection. For example, Microsoft Windows runs its web browser in a special process with restricted permissions. This way, if a system administrator visits a web site containing a malicious JavaScript program, even if the program takes over the browser, it cannot store files or do other operations that would normally be available to the system administrator. We know a computer security expert who runs each new web page in a separate virtual machine; even if the web page contains a virus that takes over the browser, and the browser is able to take over the operating system, the original, uninfected, operating system can be automatically restored by resetting the virtual machine.

Cross-site scripting

Another JavaScript attack makes use of the storage interface provided to JavaScript programs. To allow JavaScript programs to communicate with each other, they can store data in cookies in the browser. For some web sites, these cookies can contain sensitive information such as the user's login authentication. A JavaScript program that can gain access to a user's cookies can potentially pretend to be the user, and therefore access the user's sensitive data stored at the server. If a web site is compromised, it can be modified to serve pages containing a JavaScript program that gathers and exploits the user's sensitive data. These are called *cross-site scripting attacks*, and they are widespread.

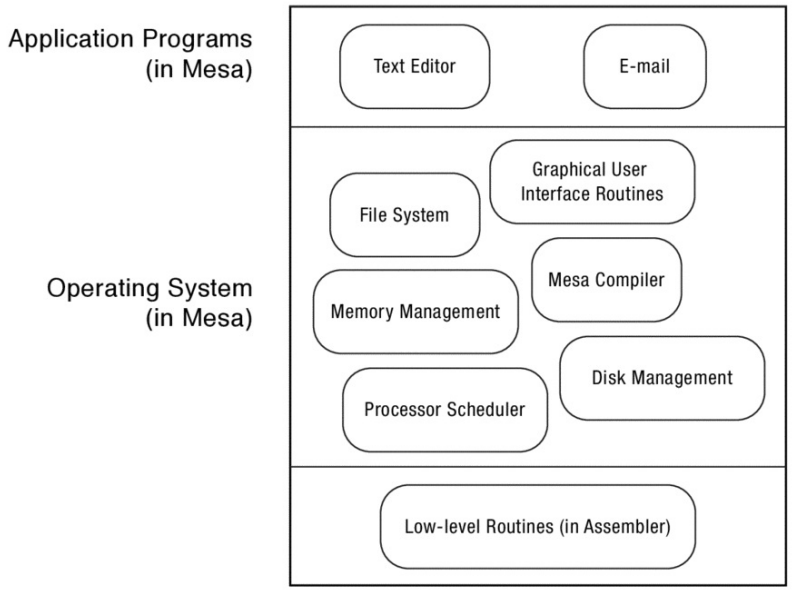

Figure 8.21: Design of the Xerox Alto operating system. Application programs and most of the operating system were implemented in a type-safe programming language called Mesa; Mesa isolated most errors to the module that caused the error.

A related approach is to write all the software on a system in a single, safe language, and then to compile the code into machine instructions that execute directly on the processor. Unlike interpreted languages, the libraries themselves can be written in the safe language. The Xerox Alto took this approach: both applications and the entire operating system were written in the same language, Mesa. Like Java, Mesa had support for thread synchronization built directly into the language. Even with this, however, there are practical issues. You still need to do defensive programming at the trust boundary — between untrusted application code (written in the safe language) and trusted operating system code (written in the safe language). You also need to be able to trust the compiler to generate correct code that enforces protection; any weakness in the compiler could allow a buggy program to crash the system. The designers of the Alto built a successor system, called the Digital Equipment Firefly, which used a successor language to Mesa, called Modula-2, for implementing both applications and the operating system. However,

for an extra level of protection, the Firefly also used hardware protection to isolate applications from the operating system kernel.

8.4.2 Language-Independent Software Fault Isolation

A limitation of trusting a language and its interpreter or compiler to provide safety is that many programmers value the flexibility to choose their own programming language. For example, some might use Ruby for configuring web servers, Matlab or Python for writing scientific code, or C++ for large software engineering efforts.

Since it would be impractical for the operating system to trust every compiler for every possible language, can we efficiently isolate application code, in software without hardware support, in a programming language independent fashion?

One reason for considering this is that there are many cases where systems need an extra level of protection within a process. We saw an example of this with web browsers needing to safely execute JavaScript programs, but there are many other examples. With software protection, we could give users the ability to customize the operating system by downloading code into the kernel, as with packet filters, but on a more widespread basis. Kernel device drivers have been shown to be the primary cause of operating system crashes; providing a way for the kernel to execute device drivers in a restricted environment could potentially cut down on the severity of these faults. Likewise, many complex software packages such as databases, spreadsheets, desktop publishing systems, and systems for computer-aided design, provide their users a way to download code into the system to customize and configure the system's behavior to meet the user's specific needs. If this downloaded code causes the system to crash, the user will not be able to tell who is really at fault and is likely to end up blaming the vendor.

Of course, one way to do this is to rely on the JavaScript interpreter. Tools exist to compile code written in one language, like C or C++, into JavaScript. This lets applications written in those languages to run on any browser that supports JavaScript. If executing JavaScript were safe and fast enough, then we could declare ourselves done.

In this section, we discuss an alternate approach: can we take any chunk of machine instructions and modify it to ensure that the code does not touch any memory outside of its own region of data? That way, the code could be written in any language, compiled by any compiler, and directly execute at the full speed of the processor.

Both Google and Microsoft have products that accomplish this: a sandbox that can run code written in any programming language, executed safely inside a process. Google's product is called Native Client; Microsoft's is called Application Domains. These implementations are efficient: Google reports that the runtime overhead of executing code safely inside a sandbox is less than 10%.

For simplicity of our discussion, we will assume that the memory region for the sandbox is contiguous, that is, the sandbox has a base and bound that needs to be enforced in software. Because we can disallow the execution of obviously malicious code, we can start by checking that the code in the sandbox does not use self-modifying instructions or privileged instructions.

We proceed in two steps. First, we insert machine instructions into the executable to do what hardware protection would have done, that is, to check that each address is legally within the region specified by the base and bounds, and to raise an exception if not. Second, we use control and data flow analysis to remove checks that are not strictly necessary for the sandbox to be correct. This mirrors what we did for hardware translation — first, we designed a general-purpose and flexible mechanism, and then we showed how to optimize it using TLBs so that the full translation mechanism was not needed on every instruction.

The added instructions for every load and store instruction are simple: just add a check that the address to be used by each load or store instruction is within the correct region of data. In the code, r1 is a machine register.

```
test r1, data.base
if less-than, branch to exception
test r1, data.bound
if greater-than, branch to exception
store data at r1
```

Note that the store instructions must be limited to just the data region of the sandbox; otherwise a store could modify the instruction sequence, e.g., to cause a jump out of the protected region.

We also need to check indirect branch instructions. We need to make sure the program cannot branch outside of the sandbox except for predefined entry and exit points. Relative branches and named procedure calls can be directly verified. Indirect branches and procedure returns jump to a location stored in a register or in memory; the address must be checked before use.

```
test r1, code.base
if less-than, branch to exception
test r1, code.bound
if greater-than, branch to exception
jump to r1
```

As a final detail, the above code verifies that indirect branch instructions stay within the code region. This turns out to be insufficient for protection, for two reasons. First, x86 code is byte addressable, and if you allow a jump to the middle of an instruction, you cannot be guaranteed as to what the code will do. In particular, an erroneous or malicious program might jump to the middle of an instruction, whose bytes would cause the processor to jump outside of the protected region. Although this may seem unlikely, remember that the attacker has the advantage; the attacker can try various code sequences to see if that causes an escape from the sandbox. A second issue is that an indirect branch might jump past the protection checks for a load or store instruction. We can prevent both of these by doing all indirect jumps through a table that only contains valid entry points

into the code; of course, the table must also be protected from being modified by the code in the sandbox.

Now that we have logical correctness, we can run control and data flow analysis to eliminate many of the extra inserted instructions, if it can be proven that they are not needed. Examples of possible optimizations include:

- **Loop invariants.** If a loop strides through memory, the sandbox may be able to prove with a simple test at the beginning of the loop that all memory accesses in the loop will be within the protected region.
- **Return values.** If static code analysis of a procedure can prove that the procedure does not modify the return program counter stored on the stack, the return can be made safely without further checks.
- **Cross-procedure checks.** If the code analysis can prove that a parameter is always checked before it is passed as an argument to a subroutine, it need not be checked when it is used inside the procedure.

Virtual machines without kernel support

Modifying machine code to transparently change the behavior of a program, while still enforcing protection, can be used for other purposes. One application is transparently executing a guest operating system inside a user-level process without kernel support.

Normally, when we run a guest operating system in a virtual machine, the hardware catches any privileged instructions executed by the guest kernel and traps into the host kernel. The host kernel emulates the instructions and returns control back to the guest kernel at the instruction immediately after the hardware exception. This allows the host kernel to emulate privilege levels, interrupts, exceptions, and kernel management of hardware page tables.

What happens if we are running on top of an operating system that does not support a virtual machine? We can still emulate a virtual machine by modifying the machine code of the guest operating system kernel. For example, we can convert instructions to enable and disable interrupts to a no op. We can convert an instruction to start executing a user program to take the contents of the application memory, re-write those contents into a user-level sandbox, and start it executing. From the perspective of the guest kernel, the application program execution looks normal; it is the sandbox that keeps the application program from corrupting kernel's data structures and passes control to the guest kernel when the application makes a system call.

Because of the widespread use of virtual machines, some hardware architectures have begun to add support for directly executing guest operating systems in user-mode without kernel support. We will return to this issue in a later chapter, as it is closely related to the topic of stackable virtual machines: how do we manipulate page tables to handle the case where the guest operating system is itself a virtual machine monitor running a virtual machine.

8.4.3 Sandboxes Via Intermediate Code

To improve portability, both Microsoft and Google can construct their sandboxes from intermediate code generated by the compiler. This makes it easier for the system to do the code modification and data flow analysis to enforce the sandbox. Instead of generating x86 or ARM code directly, the various compilers generate their code in the intermediate language, and the sandbox runtime converts that into sandboxed code on the specific processor architecture.

The intermediate representation can be thought of as a virtual machine, with a simpler instruction set. From the compiler perspective, it is as easy to generate code for the virtual machine as it would be to go directly to x86 or ARM instructions. From the sandbox perspective though, using a virtual machine as the intermediate representation is much simpler. The intermediate code can include annotations as to which pointers can be proven to be safe and which must be checked before use. For example, pointers in a C program would require runtime checks while the memory references in a Java program may be able to be statically proven as safe from the structure of the code.

Microsoft has compilers for virtually every commercially important programming language. To avoid trusting all of these compilers with the safety of the system, the runtime is responsible for validating any of the type information needed for efficient code generation for the sandbox. Typically, verifying the correctness of static analysis is much simpler than generating it in the first place.

The Java virtual machine (JVM) is also a kind of sandbox; Java code is translated into intermediate byte code instructions that can be verified at runtime as being safely contained in the sandbox. Several languages have been compiled into Java byte code, such as Python, Ruby, and JavaScript. Thus, a JVM can also be considered a language-independent sandbox. However, because of the structure of the intermediate representation in Java, it is more difficult to generate correct Java byte code for languages such as C or Fortran.

8.5 Summary and Future Directions

Address translation is a powerful abstraction enabling a wide variety of operating system services. It was originally designed to provide isolation between processes and to protect the operating system kernel from misbehaving applications, but it is more widely applicable. It is now used to simplify memory management, to speed interprocess communication, to provide for efficient shared libraries, to map files directly into memory, and a host of other uses.

A huge challenge to effective hardware address translation is the cumulative effect of decades of Moore's Law: both servers and desktop computers today contain vast amounts of memory. Processes are now able to map their code, data, heap, shared libraries, and files directly into memory. Each of these segments can be dynamic; they can be shared across processes or private to a single process. To handle these demands, hardware systems have converged on a two-tier structure: a multi-level segment and page table to

provide very flexible but space-efficient lookup, along with a TLB to provide time-efficient lookup for repeated translations of the same page.

Much of what we can do in hardware we can also do in software; a combination of hardware and software protection has proven attractive in a number of contexts. Modern web browsers execute code embedded in web pages in a software sandbox that prevents the code from infecting the browser; the operating system uses hardware protection to provide an extra level of defense in case the browser itself is compromised.

The future trends are clear:

- **Very large memory systems.** The cost of a gigabyte of memory is likely to continue to plummet, making ever larger memory systems practical. Over the past few decades, the amount of memory per system has almost doubled each year. We are likely to look back at today's computers and wonder how we could have gotten by with as little as a gigabyte of DRAM! These massive memories will require ever deeper multi-level page tables. Fortunately, the same trends that make it possible to build gigantic memories also make it possible to design very large TLBs to hide the increasing depth of the lookup trees.
- **Multiprocessors.** On the other hand, multiprocessors will mean that maintaining TLB consistency will become increasingly expensive. A key assumption for using page table protection hardware for implementing copy-on-write and fill-on-demand is that the cost of modifying page table entries is modest. One possibility is that hardware will be added to systems to make TLB shootdown a much cheaper operation, e.g., by making TLBs cache coherent. Another possibility is to follow the trend towards software sandboxes. If TLB shootdown remains expensive, we may start to see copy-on-write and other features implemented in software rather than hardware.
- **User-level sandboxes.** Applications like browsers that run untrusted code are becoming increasingly prevalent. Operating systems have only recently begun to recognize the need to support these types of applications. Software protection has become common, both at the language level with JavaScript, and in the runtime system with Native Client and Application Domains. As these technologies become more widely used, it seems likely we may direct hardware support for application-level protection — to allow each application to set up its own protected execution environment, but enforced in hardware. If so, we may come to think of many applications as having their own embedded operating system, and the underlying operating system kernel as mediating between these operating systems.

Exercises

1. True or false. A virtual memory system that uses paging is vulnerable to external fragmentation. Why or why not?
2. For systems that use paged segmentation, what translation state does the kernel need to change on a process context switch?
3. For the three-level SPARC page table, what translation state does the kernel need to change on a process context switch?
4. Describe the advantages of an architecture that incorporates segmentation and paging over ones that are either pure paging or pure segmentation. Present your answer as separate lists of advantages over each of the pure schemes.
5. For a computer architecture with multi-level paging, a page size of 4 KB, and 64-bit physical and virtual addresses:
    a. List the required and optional fields of its page table entry, along with the number of bits per field.
    b. Assuming a compact encoding, what is the smallest possible size for a page table entry in bytes, rounded up to an even number.
    c. Assuming a requirement that each page table fits into a single page, and given your answer above, how many levels of page tables would be required to completely map the 64-bit virtual address space?
6. Consider the following piece of code which multiplies two matrices:

```
float a[1024][1024], b[1024][1024], c[1024][1024];

multiply() {
   unsigned i, j, k;

   for (i = 0; i < 1024; i++)
       for (j = 0; j < 1024; j++)
           for (k = 0; k < 1024; k++)
               c[i][j] += a[i,k] * b[k,j];
}
```

   Assume that the binary for executing this function fits in one page and that the stack also fits in one page. Assume that storing a floating point number takes 4 bytes of memory. If the page size is 4 KB, the TLB has 8 entries, and the TLB always keeps the most recently used pages, compute the number of TLB misses assuming the TLB is initially empty.

7. Of the following items, which are stored in the thread control block, which are stored in the process control block, and which in neither?
    a. Page table pointer
    b. Page table
    c. Stack pointer
    d. Segment table
    e. Ready list
    f. CPU registers
    g. Program counter
8. Draw the segment and page table for the 32-bit Intel architecture.
9. Draw the segment and page table for the 64-bit Intel architecture.

10. For a computer architecture with multi-level paging, a page size of 4 KB, and 64-bit physical and virtual addresses:
    a. What is the smallest possible size for a page table entry, rounded up to a power of two?
    b. Using your result above, and assuming a requirement that each page table fits into a single page, how many levels of page tables would be required to completely map the 64-bit virtual address space?
11. Suppose you are designing a system with paged segmentation, and you anticipate the memory segment size will be uniformly distributed between 0 and 4 GB. The overhead of the design is the sum of the internal fragmentation and the space taken up by the page tables. If each page table entry uses four bytes per page, what page size minimizes overhead?
12. In an architecture with paged segmentation, the 32-bit virtual address is divided into fields as follows:

4 bit segment number	12 bit page number	16 bit offset

The segment and page tables are as follows (all values in hexadecimal):

Segment Table	Page Table A	Page Table B
0 Page Table A	0 CAFE	0 F000
1 Page Table B	1 DEAD	1 D8BF
x (rest invalid)	2 BEEF	2 3333
	3 BA11	x (rest invalid)
	x (rest invalid)	

Find the physical address corresponding to each of the following virtual addresses (answer “invalid virtual address” if the virtual address is invalid):

a. 00000000
b. 20022002
c. 10015555

13. Suppose a machine with 32-bit virtual addresses and 40-bit physical addresses is designed with a two-level page table, subdividing the virtual address into three pieces as follows:

10 bit page table number	10 bit page number	12 bit offset

The first 10 bits are the index into the top-level page table, the second 10 bits are the index into the second-level page table, and the last 12 bits are the offset into the page. There are 4 protection bits per page, so each page table entry takes 4 bytes.

    a. What is the page size in this system?
    b. How much memory is consumed by the first and second level page tables and wasted by internal fragmentation for a process that has 64K of memory starting at address 0?
    c. How much memory is consumed by the first and second level page tables and wasted by internal fragmentation for a process that has a code segment of 48K starting at address 0x1000000, a data segment of 600K starting at address 0x80000000 and a stack segment of 64K starting at address 0xf0000000 and growing upward (towards higher addresses)?
14. Write pseudo-code to convert a 32-bit virtual address to a 32-bit physical address for a two-level address translation scheme using segmentation at the first level of translation and paging at the second level. Explicitly define whatever constants and data structures you need (e.g., the format of the page table entry, the page size, and so forth).

9. Caching and Virtual Memory

Cash is king. —*Per Gyllenhammar*

Some may argue that we no longer need a chapter on caching and virtual memory in an operating systems textbook. After all, most students will have seen caches in an earlier machine structures class, and most desktops and laptops are configured so that they only very rarely, if ever, run out of memory. Maybe caching is no longer an operating systems topic?

We could not disagree more. Caches are central to the design of a huge number of hardware and software systems, including operating systems, Internet naming, web clients, and web servers. In particular, smartphone operating systems are often memory constrained and must manage memory carefully. Server operating systems make extensive use of remote memory and remote disk across the data center, using the local server memory as a cache. Even desktop operating systems use caching extensively in the implementation of the file system. Most importantly, understanding when caches work and when they do not is essential to every computer systems designer.

Consider a typical Facebook page. It contains information about you, your interests and privacy settings, your posts, and your photos, plus your list of friends, their interests and privacy settings, their posts, and their photos. In turn, your friends' pages contain an overlapping view of much of the same data, and in turn, their friends' pages are constructed the same way.

Now consider how Facebook organizes its data to make all of this work. How does Facebook assemble the data needed to display a page? One option would be to keep all of the data for a particular user's page in one place. However, the information that I need to draw my page overlaps with the information that my friends' friends need to draw their pages. My friends' friends' friends' friends include pretty much the entire planet. We can either store everyone's data in one place or spread the data around. Either way, performance will suffer! If we store all the data in California, Facebook will be slow for everyone from Europe, and vice versa. Equally, integrating data from many different locations is also likely to be slow, especially for Facebook's more cosmopolitan users.

To resolve this dilemma, Facebook makes heavy use of caches; it would not be practical without them. A *cache* is a copy of a computation or data that can be accessed more quickly than the original. While any object on my page might change from moment to moment, it seldom does. In the common case, Facebook relies on a local, cached copy of the data for my page; it only goes back to the original source if the data is not stored locally or becomes out of date.

Caches work because both users and programs are predictable. You (probably!) do not change your friend list every nanosecond; if you did, Facebook could still cache your friend list, but it would be out of date before it could be used again, and so it would not help. If everyone changed their friends every nanosecond, Facebook would be out of luck! In most cases, however, what users do now is predictive of what they are likely to do soon,

and what programs do now is predictive of what they will do next. This provides an opportunity for a cache to save work through reuse.

Facebook is not alone in making extensive use of caches. Almost all large computer systems rely on caches. In fact, it is hard to think of any widely used, complex hardware or software system that does *not* include a cache of some sort.

We saw three examples of hardware caches in the previous chapter:

- **TLBs.** Modern processors use a translation lookaside buffer, or TLB, to cache the recent results of multi-level page table address translation. Provided programs reference the same pages repeatedly, translating an address is as fast as a single table lookup in the common case. The full multi-level lookup is needed only in the case where the TLB does not contain the relevant address translation.
- **Virtually addressed caches.** Most modern processor designs take this idea a step farther by including a virtually addressed cache close to the processor. Each entry in the cache stores the memory value associated with a virtual address, allowing that value to be returned more quickly to the processor when needed. For example, the repeated instruction fetches inside a loop are well handled by a virtually addressed cache.
- **Physically addressed caches.** Most modern processors complement the virtually addressed cache with a second- (and sometimes third-) level physically addressed cache. Each entry in a physically addressed cache stores the memory value associated with a physical memory location. In the common case, this allows the memory value to be returned directly to the processor without the need to go to main memory.

There are many more examples of caches:

- **Internet naming.** Whenever you type in a web request or click on a link, the client computer needs to translate the name in the link (e.g., amazon.com) to an IP network address of where to send each packet. The client gets this information from a network service, called the Domain Name System (DNS), and then caches the translation so that the client can go directly to the web server in the common case.
- **Web content.** Web clients cache copies of HTML, images, JavaScript programs, and other data so that web pages can be refreshed more quickly, using less bandwidth. Web servers also keep copies of frequently requested pages in memory so that they can be transmitted more quickly.
- **Web search.** Both Google and Bing keep a cached copy of every web page they index. This allows them to provide the copy of the web page if the original is unavailable for some reason. The cached copy may be out of date — the search engines do not guarantee that the copy instantaneously reflects any change in the original web page.
- **Email clients.** Many email clients store a copy of mail messages on the client computer to improve the client performance and to allow disconnected operation. In the background, the client communicates with the server to keep the two copies in sync.
- **Incremental compilation.** If you have ever built a program from multiple source files, you have used caching. The build manager saves and reuses the individual object files instead of recompiling everything from scratch each time.
- **Just in time translation.** Some memory-constrained devices such as smartphones do not contain enough memory to store the entire executable image for some programs. Instead, systems such as the Google Android operating system and the ARM runtime store programs in a more compact intermediate representation, and convert parts of the program to machine code as needed. Repeated use of the same code is fast because of caching; if the system runs out of memory, less frequently used code may be converted each time it is needed.
- **Virtual memory.** Operating systems can run programs that do not fit in physical memory by using main memory as a cache for disk. Application pages that fit in memory have their page table entries set to valid; these pages can be accessed directly by the processor. Those pages that do not fit have their permissions set to invalid, triggering a trap to the operating system kernel. The kernel will then fetch the required page from disk and resume the application at the instruction that caused the trap.
- **File systems.** File systems also treat memory as a cache for disk. They store copies in memory of frequently used directories and files, reducing the need for disk accesses.
- **Conditional branch prediction.** Another use of caches is in predicting whether a conditional branch will be taken or not. In the common case of a correct prediction, the processor can start decoding the next instruction before the result of the branch is known for sure; if the prediction turns out to be wrong, the decoding is restarted with the correct next instruction.

In other words, caches are a central design technique to making computer systems faster. However, caches are not without their downsides. Caches can make understanding the performance of a system much harder. Something that seems like it should be fast — and even something that usually is fast — can end up being very slow if most of the data is not in the cache. Because the details of the cache are often hidden behind a level of abstraction, the user or the programmer may have little idea as to what is causing the poor performance. In other words, the abstraction of fast access to data can cause problems if the abstraction does not live up to its promise. One of our aims is to help you understand when caches do and do not work well.

In this chapter, we will focus on the caching of memory values, but the principles we discuss apply much more widely. Memory caching is common in both hardware (by the processor to improve memory latency) and in software (by the operating system to hide disk and network latency). Further, the structure and organization of processor caches requires special care by the operating system in setting up page tables; otherwise, much of the advantage of processor caches can evaporate.

Regardless of the context, all caches face three design challenges:

- **Locating the cached copy.** Because caches are designed to improve performance, a key question is often how to quickly determine whether the cache contains the needed data or not. Because the processor consults at least one hardware cache on every instruction, hardware caches in particular are organized for efficient lookup.
- **Replacement policy.** Most caches have physical limits on how many items they can store; when new data arrives in the cache, the system must decide which data is most valuable to keep in the cache and which can be replaced. Because of the high relative latency of fetching data from disk, operating systems and applications have focused more attention on the choice of replacement policy.
- **Coherence.** How do we detect, and repair, when a cached copy becomes out of date? This question, cache coherence, is central to the design of multiprocessor and distributed systems. Despite being very important, cache coherence beyond the scope of this version of the textbook. Instead, we focus on the first two of these issues.

Chapter roadmap:

- **Cache Concept.** What operations does a cache do and how can we evaluate its performance? (Section 9.1)
- **Memory Hierarchy.** What hardware building blocks do we have in constructing a cache in an application or operating system? (Section 9.2)
- **When Caches Work and When They Do Not.** Can we predict how effective a cache will be in a system we are designing? Can we know in advance when caching will *not* work? (Section 9.3)
- **Memory Cache Lookup.** What options do we have for locating whether an item is cached? How can we organize hardware caches to allow for rapid lookup, and what are the implications of cache organization for operating systems and applications? (Section 9.4)
- **Replacement Policies.** What options do we have for choosing which item to replace when there is no more room? (Section 9.5)
- **Case Study: Memory-Mapped Files.** How does the operating system provide the abstraction of file access without first reading the entire file into memory? (Section 9.6)
- **Case Study: Virtual Memory.** How does the operating system provide the illusion of a near-infinite memory that can be shared between applications? What happens if both applications and the operating system want to manage memory at the same time? (Section 9.7)

9.1 Cache Concept

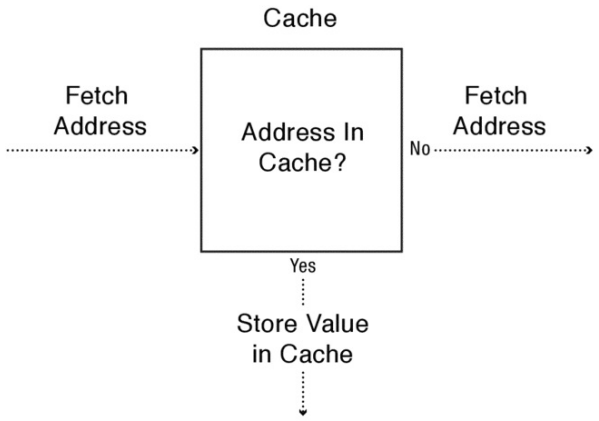

Figure 9.1: Abstract operation of a memory cache on a read request. Memory read requests are sent to the cache; the cache either returns the value stored at that memory location, or it forwards the request onward to the next level of cache.

We start by defining some terms. The simplest kind of a cache is a memory cache. It stores (address, value) pairs. As shown in Figure 9.1, when we need to read value of a certain memory location, we first consult the cache, and it either replies with the value (if the cache knows it) and otherwise it forwards the request onward. If the cache has the value, that is called a *cache hit*. If the cache does not, that is called a *cache miss*.

For a memory cache to be useful, two properties need to hold. First, the cost of retrieving data out of the cache must be significantly less than fetching the data from memory. In other words, the cost of a cache hit must be less than a cache miss, or we would just skip using the cache.

Second, the likelihood of a cache hit must be high enough to make it worth the effort. One source of predictability is *temporal locality*: programs tend to reference the same instructions and data that they had recently accessed. Examples include the instructions inside a loop, or a data structure that is repeatedly accessed. By caching these memory values, we can improve performance.

Another source of predictability is *spatial locality*. Programs tend to reference data near other data that has been recently referenced. For example, the next instruction to execute is usually near to the previous one, and different fields in the same data structure tend to be referenced at nearly the same time. To exploit this, caches are often designed to load a block of data at the same time, instead of only a single location. Hardware memory caches often store 4-64 memory words as a unit; file caches often store data in powers of two of the hardware page size.

A related design technique that also takes advantage of spatial locality is to *prefetch* data into the cache before it is needed. For example, if the file system observes the application

reading a sequence of blocks into memory, it will read the subsequent blocks ahead of time, without waiting to be asked.

Putting these together, the latency of a read request is as follows:

$$\text{Latency(read request)} = \text{Prob(cache hit)} \times \text{Latency(cache hit)} + \text{Prob(cache miss)} \times \text{Latency(cache miss)}$$

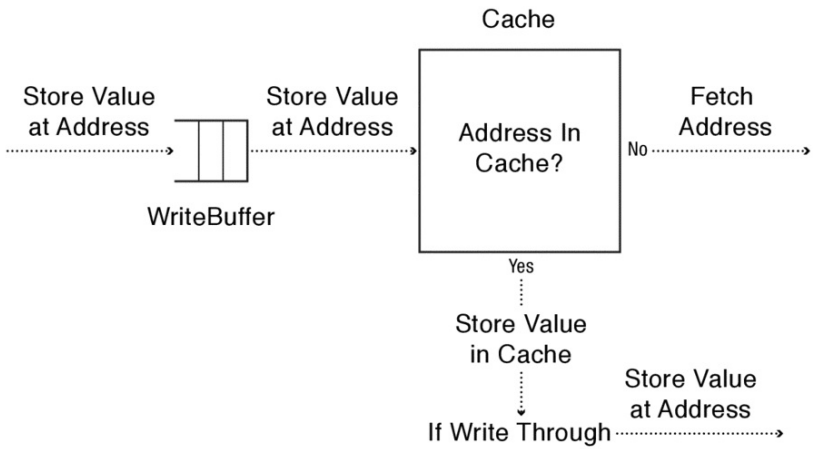

Figure 9.2: Abstract operation of a memory cache write. Memory requests are buffered and then sent to the cache in the background. Typically, the cache stores a block of data, so each write ensures that the rest of the block is in the cache before updating the cache. If the cache is write through, the data is then sent onward to the next level of cache or memory.

The behavior of a cache on a write operation is shown in Figure 9.2. The operation is a bit more complex, but the latency of a write operation is easier to understand. Most systems buffer writes. As long as there is room in the buffer, the computation can continue immediately while the data is transferred into the cache and to memory in the background. (There are certain restrictions on the use of write buffers in a multiprocessor system, so for this chapter, we are simplifying matters to some degree.) Subsequent read requests must check both the write buffer and the cache — returning data from the write buffer if it is the latest copy.

In the background, the system checks if the address is in the cache. If not, the rest of the cache block must be fetched from memory and then updated with the changed value. Finally, if the cache is *write-through*, all updates are sent immediately onward to memory. If the cache is *write-back*, updates can be stored in the cache, and only sent to memory when the cache runs out of space and needs to evict a block to make room for a new memory block.

Since write buffers allow write requests to appear to complete immediately, the rest of our discussion focuses on using caches to improve memory reads.

We first discuss the part of the equation that deals with the latency of a cache hit and a cache miss: how long does it take to access different types of memory? We caution, however, that the issues that affect the likelihood of a cache hit or miss are just as important to the overall memory latency. In particular, we will show that application characteristics are often the limiting factor to good cache performance.

9.2 Memory Hierarchy

When we are deciding whether to use a cache in the operating system or some new application, it is helpful to start with an understanding of the cost and performance of various levels of memory and disk storage.

Cache	Hit Cost	Size
1st level cache / 1st level TLB	1 ns	64 KB
2nd level cache / 2nd level TLB	4 ns	256 KB
3rd level cache	12 ns	2 MB
Memory (DRAM)	100 ns	10 GB
Data center memory (DRAM)	100 µs	100 TB
Local non-volatile memory	100 µs	100 GB
Local disk	10 ms	1 TB
Data center disk	10 ms	100 PB

Remote data center disk	200 ms	1 XB

Figure 9.3: Memory hierarchy, from on-chip processor caches to disk storage at a remote data center. On-chip cache size and latency is typical of a high-end processor. The entries for data center DRAM and disk latency assume the access is from one server to another in the same data center; remote data center disk latency if for access to a geographically distant data center.

From a hardware perspective, there is a fundamental tradeoff between the speed, size, and cost of storage. The smaller memory is, the faster it can be; the slower memory is, the cheaper it can be.

This motivates systems to have not just one cache, but a whole hierarchy of caches, from the nanosecond memory possible inside a chip to the multiple exabytes of worldwide data center storage. This hierarchy is illustrated by the table in Figure 9.3. We should caution that this list is just a snapshot; additional layers keep being added over time.

- **First-level cache.** Most modern processor architectures contain a small first-level, virtually addressed, cache very close to the processor, designed to keep the processor fed with instructions and data at the clock rate of the processor.
- **Second-level cache.** Because it is impossible to build a large cache as fast as a small one, the processor will often contain a second-level, physically addressed cache to handle cache misses from the first-level cache.
- **Third-level cache.** Likewise, many processors include an even larger, slower third-level cache to catch second-level cache misses. This cache is often shared across all of the on-chip processor cores.
- **First- and second-level TLB.** The translation lookaside buffer (TLB) will also be organized with multiple levels: a small, fast first-level TLB designed to keep up with the processor, backed up by a larger, slightly slower, second-level TLB to catch first-level TLB misses.
- **Main memory (DRAM).** From a hardware perspective, the first-, second-, and third-level caches provide faster access to main memory; from a software perspective, however, main memory itself can be viewed as a cache.
- **Data center memory (DRAM).** With a high-speed local area network such as a data center, the latency to fetch a page of data from the memory of a nearby computer is much faster than fetching it from disk. In aggregate, the memory of nearby nodes will often be larger than that of the local disk. Using the memory of nearby nodes to avoid the latency of going to disk is called *cooperative caching*, as it requires the cooperative management of the nodes in the data center. Many large scale data center services, such as Google and Facebook, make extensive use of cooperative caching.
- **Local disk or non-volatile memory.** For client machines, local disk or non-volatile flash memory can serve as backing store when the system runs out of memory. In turn, the local disk serves as a cache for remote disk storage. For example, web browsers store recently fetched web pages in the client file system to avoid the cost of transferring the data again the next time it is used; once cached, the browser only needs to validate with the server whether the page has changed before rendering the web page for the user.
- **Data center disk.** The aggregate disks inside a data center provide enormous storage capacity compared to a computer's local disk, and even relative to the aggregate memory of the data center.
- **Remote data center disk.** Geographically remote disks in a data center are much slower because of wide-area network latencies, but they provide access to even larger storage capacity in aggregate. Many data centers also store a copy of their data on a remote robotic tape system, but since these systems have very high latency (measured in the tens of seconds), they are typically accessed only in the event of a failure.

If caching always worked perfectly, we could provide the illusion of instantaneous access to all the world's data, with the latency (on average) of a first level cache and the size and the cost (on average) of disk storage.

However, there are reasons to be skeptical. Even with temporal and spatial locality, there are thirteen orders of magnitude difference in storage capacity from the first level cache to the stored data of a typical data center; this is the equivalent of the smallest visible dot on this page versus those dots scattered across the pages of a million textbooks just like this one. How can a cache be effective if it can store only a tiny amount of the data that could be stored?

The cost of a cache miss can also be high. There are eight orders of magnitude difference between the latency of the first-level cache and a remote data center disk; that is equivalent to the difference between the shortest latency a human can perceive — roughly one hundred milliseconds — versus one year. How can a cache be effective if the cost of a cache miss is enormous compared to a cache hit?

9.3 When Caches Work and When They Do Not

How do we know whether a cache will be effective for a given workload? Even the same program will have different cache behavior depending on how it is used.

Suppose you write a program that reads and writes items into a hash table. How well does that interact with caching? It depends on the size of the hash table. If the hash table fits in the first-level cache, once the table is loaded into the cache, each access will be very rapid. If on the other hand, the hash table is too large to store in memory, each lookup may require a disk access.

Thus, neither the cache size nor the program behavior alone governs the effectiveness of caching. Rather, the interaction between the two determines cache effectiveness.

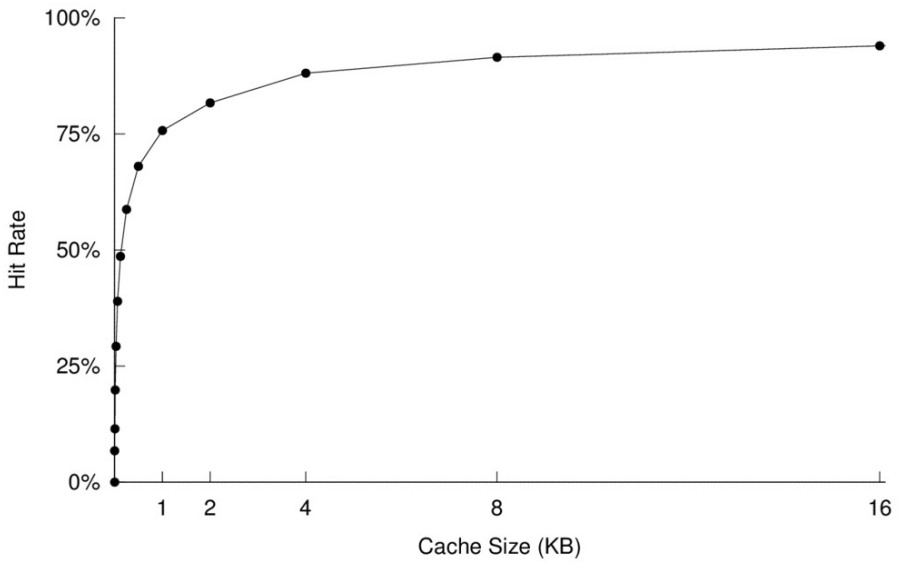

Figure 9.4: Cache hit rate as a function of cache size for a million instruction run of a C compiler. The hit rate vs. cache size graph has a similar shape for many programs. The knee of the curve is called the working set of the program.

9.3.1 Working Set Model

A useful graph to consider is the cache hit rate versus the size of the cache. We give an example in Figure 9.4; of course, the precise shape of the graph will vary from program to program.

Regardless of the program, a sufficiently large cache will have a high cache hit rate. In the limit, if the cache can fit all of the program's memory and data, the miss rate will be zero once the data is loaded into the cache. At the other extreme, a sufficiently small cache will have a very low cache hit rate. Anything other than a trivial program will have multiple procedures and multiple data structures; if the cache is sufficiently small, each new instruction and data reference will push out something from the cache that will be used in the near future. For the hash table example, if the size of the cache is much smaller than the size of the hash table, each time we do a lookup, the hash bucket we need will no longer be in the cache.

Most programs will have an inflection point, or knee of the curve, where a critical mass of program data can just barely fit in the cache. This critical mass is called the program's *working set*. As long as the working set can fit in the cache, most references will be a cache hit, and application performance will be good.

Thrashing

A closely related concept to the working set is thrashing. A program thrashes if the cache is too small to hold its working set, so that most references are cache misses. Each time there is a cache miss, we need to evict a cache block to make room for the new reference. However, the new cache block may in turn be evicted before it is reused.

The word "thrash" dates from the 1960's, when disk drives were as large as washing machines. If a program's working set did not fit in memory, the system would need to shuffle memory pages back and forth to disk. This burst of activity would literally make the disk drive shake violently, making it very obvious to everyone nearby why the system was not performing well.

The notion of a working set can also apply to user behavior. Consider what happens when you are developing code for a homework assignment. If the files you need fit in memory, compilation will be rapid; if not, compilation will be slow as each file is brought in from disk as it is used.

Different programs, and different users, will have working sets of different sizes. Even within the same program, different phases of the program may have different size working sets. For example, the parser for a compiler needs different data in cache than the code generator. In a text editor, the working set shifts when we switch from one page to the next. Users also change their focus from time to time, as when you shift from a programming assignment to a history assignment.

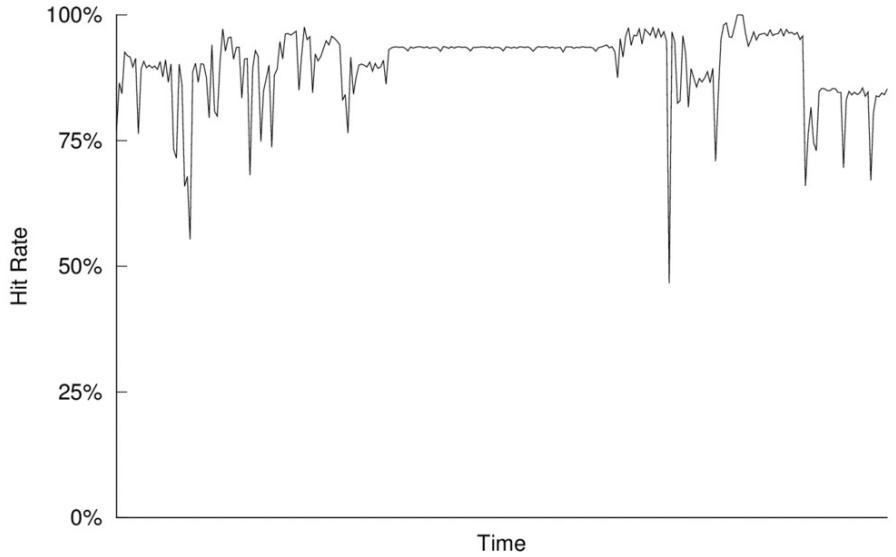

Figure 9.5: Example cache hit rate over time. At a phase change within a process, or due to a context switch between processes, there will be a spike of cache misses before the system settles into a new equilibrium.

The result of this *phase change behavior* is that caches will often have bursty miss rates:

periods of low cache misses interspersed with periods of high cache misses, as shown in Figure 9.5. Process context switches will also cause bursty cache misses, as the cache discards the working set from the old process and brings in the working set of the new process.

We can combine the graph in Figure 9.4 with the table in Figure 9.3 to see the impact of the size of the working set on computer system performance. A program whose working set fits in the first level cache will run four times faster than one whose working set fits in the second level cache. A program whose working set does not fit in main memory will run a thousand times slower than one who does, assuming it has access to data center memory. It will run a hundred thousand times slower if it needs to go to disk.

Because of the increasing depth and complexity of the memory hierarchy, an important area of work is the design of algorithms that adapt their working set to the memory hierarchy. One focus has been on algorithms that manage the gap between main memory and disk, but the same principles apply at other levels of the memory hierarchy.

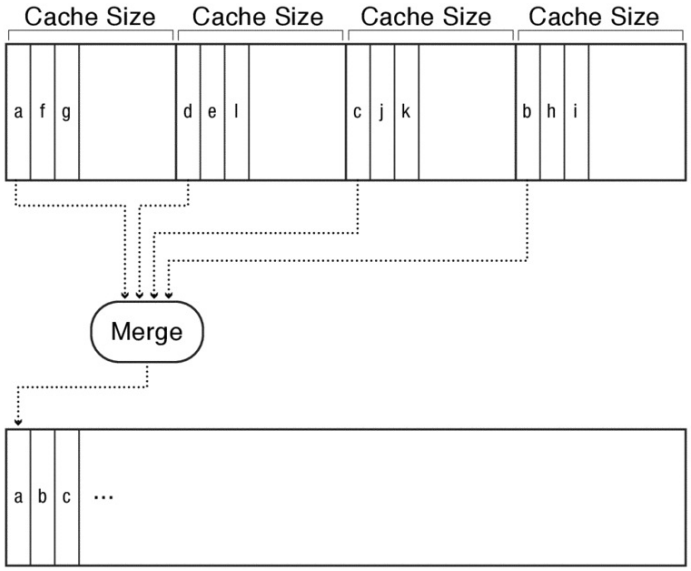

Figure 9.6: Algorithm to sort a large array that does not fit into main memory, by breaking the problem into pieces that do fit into memory.

A simple example is how to efficiently sort an array that does not fit in main memory. (Equivalently, we could consider how to sort an array that does not fit in the first level cache.) As shown in Figure 9.6, we can break the problem up into chunks each of which does fit in memory. Once we sort each chunk, we can merge the sorted chunks together efficiently. To sort a chunk that fits in main memory, we can in turn break the problem into sub-chunks that fit in the on-chip cache.

We will discuss later in this chapter what the operating system needs to do when managing memory between programs that in turn adapt their behavior to manage memory.

9.3.2 Zipf Model

Although the working set model often describes program and user behavior quite well, it is not always a good fit. For example, consider a *web proxy cache*. A web proxy cache stores frequently accessed web pages to speed web access and reduce network traffic. Web access patterns cause two challenges to a cache designer:

- **New data.** New pages are being added to the web at a rapid rate, and page contents also change. Every time a user accesses a page, the system needs to check whether the page has changed in the meantime.
- **No working set.** Although some web pages are much more popular than others, there is no small subset of web pages that, if cached, give you the bulk of the benefit. Unlike with a working set, even very small caches have some value. Conversely, increasing cache size yields diminishing returns: even very large caches tend to have only modest cache hit rates, as there are an enormous group of pages that are visited from time to time.

A useful model for understanding the cache behavior of web access is the *Zipf distribution*. Zipf developed the model to describe the frequency of individual words in a text, but it also applies in a number of other settings.

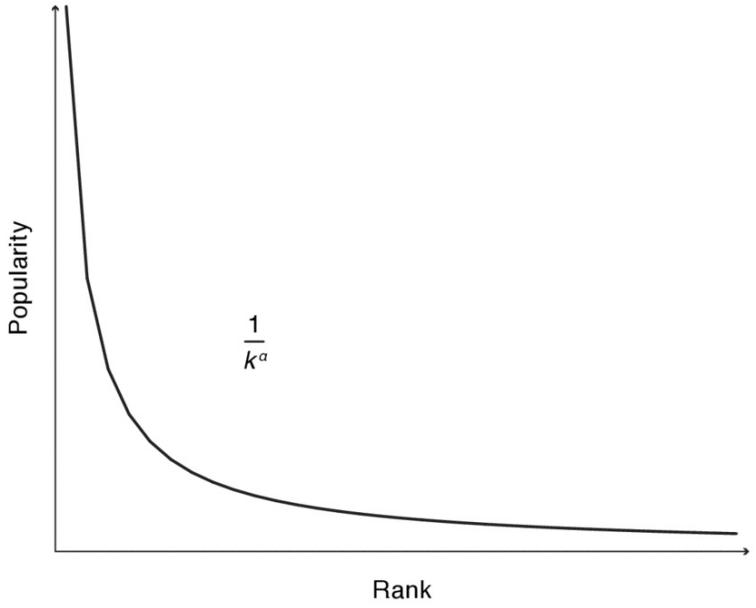

Figure 9.7: Zipf distribution

Suppose we have a set of web pages (or words), and we rank them in order of popularity. Then the frequency users visit a particular web page is (approximately) inversely proportional to its rank:

$$\text{Frequency of visits to the kth most popular page} \propto 1 / k^{\alpha}$$

where α is value between 1 and 2. A Zipf probability distribution is illustrated in Figure 9.7.

The Zipf distribution fits a surprising number of disparate phenomena: the popularity of library books, the population of cities, the distribution of salaries, the size of friend lists in social networks, and the distribution of references in scientific papers. The exact cause of the Zipf distribution in many of these cases is unknown, but they share a theme of popularity in human social networks.

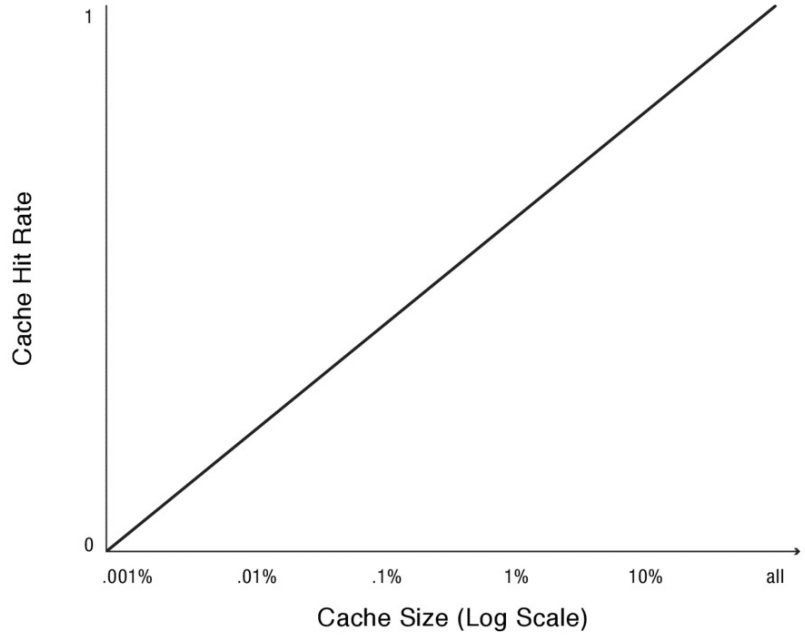

Figure 9.8: Cache hit rate as a function of the percentage of total items that can fit in the cache, on a log scale, for a Zipf distribution.

A characteristic of a Zipf curve is a *heavy-tailed distribution*. Although a significant number of references will be to the most popular items, a substantial portion of references will be to less popular ones. If we redraw Figure 9.4 of the relationship between cache hit rate and cache size, but for a Zipf distribution, we get Figure 9.8. Note that we have rescaled the x-axis to be log scale. Rather than a threshold as we see in the working set model, increasing the cache size continues to improve cache hit rates, but with diminishing returns.

9.4 Memory Cache Lookup

Now that we have outlined the available technologies for constructing caches, and the usage patterns that lend (or do not lend) themselves to effective caching, we turn to cache design. How do we find whether an item is in the cache, and what do we do when we run out of room in the cache? We answer the first question here, and we defer the second question to the next section.

A memory cache maps a sparse set of addresses to the data values stored at those addresses. You can think of a cache as a giant table with two columns: one for the address and one for the data stored at that address. To exploit spatial locality, each entry in the

table will store the values for a block of memory, not just the value for a single memory word. Modern Intel processors cache data in 64 byte chunks. For operating systems, the block size is typically the hardware page size, or 4 KB on an Intel processor.

We need to be able to rapidly convert an address to find the corresponding data, while minimizing storage overhead. The options we have for cache lookup are all of the same ones we explored in the previous chapter for address lookup: we can use a linked list, a multi-level tree, or a hash table. Operating systems use each of those techniques in different settings, depending on the size of the cache, its access pattern, and how important it is to have very rapid lookup.

For hardware caches, the design choices are more limited. The latency gap between cache levels is very small, so any added overhead in the lookup procedure can swamp the benefit of the cache. To make lookup faster, hardware caches often constrain where in the table we might find any specific address. This constraint means that there could be room in one part of the table, but not in another, raising the cache miss rate. There is a tradeoff here: a faster cache lookup needs to be balanced against the cost of increased cache misses.

Three common mechanisms for cache lookup are:

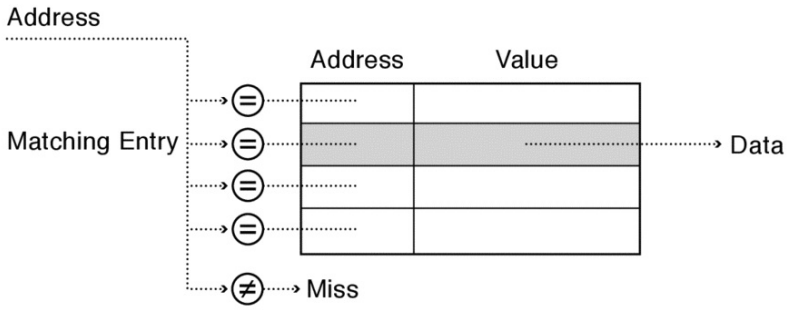

Figure 9.9: Fully associative cache lookup. The cache checks the address against every entry and returns the matching value, if any.

- **Fully associative.** With a fully associative cache, the address can be stored anywhere in the table, and so on a lookup, the system must check the address against all of the entries in the table as illustrated in Figure 9.9. There is a cache hit if any of the table entries match. Because any address can be stored anywhere, this provides the system maximal flexibility when it needs to choose an entry to discard when it runs out of space.

  We saw two examples of fully associative caches in the previous chapter. Until very recently, TLBs were often fully associative — the TLB would check the virtual page against every entry in the TLB in parallel. Likewise, physical memory is a fully associative cache. Any page frame can hold any virtual page, and we can find where each virtual page is stored using a multi-level tree lookup. The set of page tables defines whether there is a match.

  A problem with fully associative lookup is the cumulative impact of Moore's Law. As more memory can be packed on chip, caches become larger. We can use some of the added memory to make each table entry larger, but this has a limit depending on the amount of spatial locality in typical applications. Alternately, we can add more table entries, but this means more lookup hardware and comparators. As an example, a 2 MB on-chip cache with 64 byte blocks has 32K cache table entries! Checking each address against every table entry in parallel is not practical.

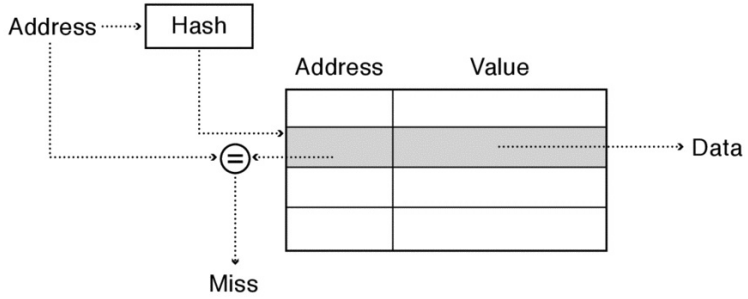

Figure 9.10: Direct mapped cache lookup. The cache hashes the address to determine which location in the table to check. The cache returns the value stored in the entry if it matches the address.

- **Direct mapped.** With a direct mapped cache, each address can only be stored in one location in the table. Lookup is easy: we hash the address to its entry, as shown in Figure 9.10. There is a cache hit if the address matches that entry and a cache miss otherwise.

  A direct mapped cache allows efficient lookup, but it loses much of that advantage in decreased flexibility. If a program happens to need two different addresses that both hash to the same entry, such as the program counter and the stack pointer, the system will thrash. We will first get the instruction; then, oops, we need the stack. Then, oops, we need the instruction again. Then oops, we need the stack again. The programmer will see the program running slowly, with no clue why, as it will depend on which addresses are assigned to which instructions and data. If the programmer inserts a print statement to try to figure out what is going wrong, that might shift the instructions to a different cache block, making the problem disappear!

- **Set associative.** A set associative cache melds the two approaches, allowing a tradeoff of slightly slower lookup than a direct mapped cache in exchange for most of the flexibility of a fully associative cache. With a set associative cache, we replicate the direct mapped table and lookup in each replica in parallel. A k set associative cache has k replicas; a particular address block can be in any of the k replicas. (This

is equivalent to a hash table with a bucket size of k.) There is a cache hit if the address matches any of the replicas.

A set associative cache avoids the problem of thrashing with a direct mapped cache, provided the working set for a given bucket is larger than k. Almost all hardware caches and TLBs today use set associative matching; an 8-way set associative cache structure is common.

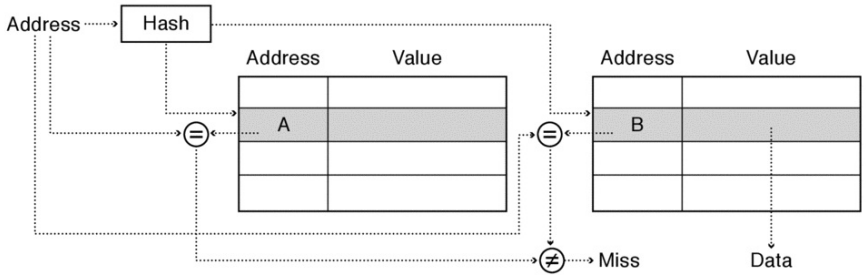

Figure 9.11: Set associative cache lookup. The cache hashes the address to determine which location to check. The cache checks the entry in each table in parallel. It returns the value if any of the entries match the address.

Direct mapped and set associative caches pose a design challenge for the operating system. These caches are much more efficient if the working set of the program is spread across the different buckets in the cache. This is easy with a TLB or a virtually addressed cache, as each successive virtual page or cache block will be assigned to a cache bucket. A data structure that straddles a page or cache block boundary will be automatically assigned to two different buckets.

However, the assignment of physical page frames is up to the operating system, and this choice can have a large impact on the performance of a physically addressed cache. To make this concrete, suppose we have a 2 MB physically addressed cache with 8-way set associativity and 4 KB pages; this is typical for a high performance processor. Now suppose the operating system happens to assign page frames in a somewhat odd way, so that an application is given physical page frames that are separated by exactly 256 KB. Perhaps those were the only page frames that were free. What happens?

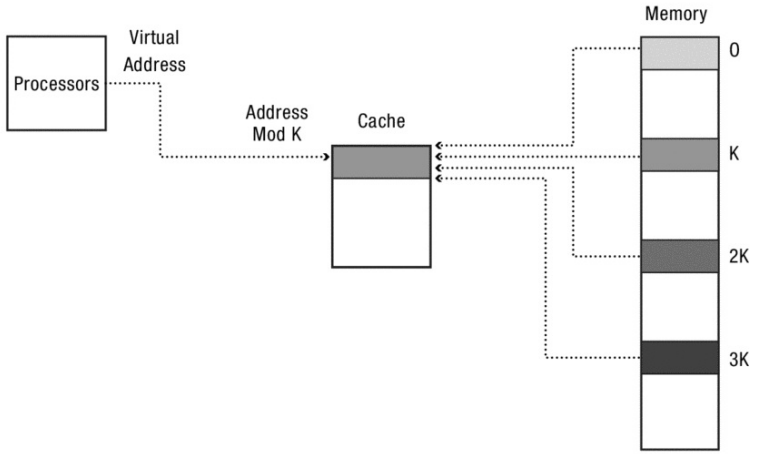

Figure 9.12: When caches are larger than the page size, multiple page frames can map to the same slice of the cache. A process assigned page frames that are separated by exactly the cache size will only use a small portion of the cache. This applies to both set associative and direct mapped caches; the figure assumes a direct mapped cache to simplify the illustration.

If the hardware uses the low order bits of the page frame to index the cache, then every page of the current process will map to the same buckets in the cache. We show this in Figure 9.12. Instead of the cache having 2 MB of useful space, the application will only be able to use 32 KB (4 KB pages times the 8-way set associativity). This makes it a lot more likely for the application to thrash.

Even worse, the application would have no way to know this had happened. If by random chance an application ended up with page frames that map to the same cache buckets, its performance will be poor. Then, when the user re-runs the application, the operating system might assign the application a completely different set of page frames, and performance returns to normal.

To make cache behavior more predictable and more effective, operating systems use a concept called *page coloring*. With page coloring, physical page frames are partitioned into sets based on which cache buckets they will use. For example, with a 2 MB 8-way set associative cache and 4 KB pages, there will be 64 separate sets, or colors. The operating system can then assign page frames to spread each application's data across the various colors.

9.5 Replacement Policies

Once we have looked up an address in the cache and found a cache miss, we have a new problem. Which memory block do we choose to replace? Assuming the reference pattern exhibits temporal locality, the new block is likely to be needed in the near future, so we need to choose some block of memory to evict from the cache to make room for the new

data. Of course, with a direct mapped cache we do not have a choice: there is only one block that can be replaced. In general, however, we will have a choice, and this choice can have a significant impact on the cache hit rate.

As with processor scheduling, there are a number of options for the replacement policy. We caution that there is no single right answer! Many replacement policies are optimal for some workloads and pessimal for others, in terms of the cache hit rate; policies that are good for a working set model will not be good for Zipf workloads.

Policies also vary depending on the setting: hardware caches use a different replacement policy than the operating system does in managing main memory as a cache for disk. A hardware cache will often have a limited number of replacement choices, constrained by the set associativity of the cache, and it must make its decisions very rapidly. In the operating system, there is often both more time to make a choice and a much larger number cached items to consider; e.g., with 4 GB of memory, a system will have a million separate 4 KB pages to choose from when deciding which to replace. Even within the operating system, the replacement policy for the file buffer cache is often different than the one used for demand paged virtual memory, depending on what information is easily available about the access pattern.

We first discuss several different replacement policies in the abstract, and then in the next two sections we consider how these concepts are applied to the setting of demand paging memory from disk.

9.5.1 Random

Although it may seem arbitrary, a practical replacement policy is to choose a random block to replace. Particularly for a first-level hardware cache, the system may not have the time to make a more complex decision, and the cost of making the wrong choice can be small if the item is in the next level cache. The bookkeeping cost for more complex policies can be non-trivial: keeping more information about each block requires space that may be better spent on increasing the cache size.

Random's biggest weakness is also its biggest strength. Whatever the access pattern is, Random will not be pessimal — it will not make the worst possible choice, at least, not on average. However, it is also unpredictable, and so it might foil an application that was designed to carefully manage its use of different levels of the cache.

9.5.2 First-In-First-Out (FIFO)

A less arbitrary policy is to evict the cache block or page that has been in memory the longest, that is, First In First Out, or FIFO. Particularly for using memory as a cache for disk, this can seem fair — each program's pages spend a roughly equal amount of time in memory before being evicted.

Unfortunately, FIFO can be the worst possible replacement policy for workloads that happen quite often in practice. Consider a program that cycles through a memory array repeatedly, but where the array is too large to fit in the cache. Many scientific applications do an operation on every element in an array, and then repeat that operation until the data reaches a fixed point. Google's PageRank algorithm for determining which search results to display uses a similar approach. PageRank iterates repeatedly through all pages, estimating the popularity of a page based on the popularity of the pages that refer to it as computed in the previous iteration.

FIFO

Ref.	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A				E				D				C		
2		B				A				E				D	
3			C				B				A				E
4				D				C				B			

Figure 9.13: Cache behavior for FIFO for a repeated scan through memory, where the scan is slightly larger than the cache size. Each row represents the contents of a page frame or cache block; each new reference triggers a cache miss.

On a repeated scan through memory, FIFO does exactly the wrong thing: it always evicts the block or page that will be needed next. Figure 9.13 illustrates this effect. Note that in this figure, and other similar figures in this chapter, we show only a small number of cache slots; note that these policies also apply to systems with a very large number of slots.

9.5.3 Optimal Cache Replacement (MIN)

If FIFO can be pessimal for some workloads, that raises the question: what replacement policy is optimal for minimizing cache misses? The optimal policy, called MIN, is to replace whichever block is used farthest in the future. Equivalently, the worst possible strategy is to replace the block that is used soonest.

Optimality of MIN

The proof that MIN is optimal is a bit involved. If MIN is not optimal, there must be some alternative optimal replacement policy, which we will call ALT, that has fewer cache misses than MIN on some specific sequence of references. There may be many such alternate policies, so let us focus on the one that differs from MIN at the latest possible point. Consider the first cache replacement where ALT differs from MIN — by definition, ALT must choose a block to replace that is used sooner than the block chosen by MIN.

We construct a new policy, ALT′, that is at least as good as ALT, but differs from MIN at a later point and so contradicts the assumption. We construct ALT′ to differ from ALT in only one respect: at the first point where ALT differs from MIN, ALT′ chooses to evict the block that MIN would have chosen. From that point, the contents of the cache differ between ALT and ALT′ only for that one block. ALT contains y, the block referenced farther in the future; ALT′ is the same, except it contains x, the block referenced sooner. On subsequent cache misses to other blocks, ALT′ mimics ALT, evicting exactly the same blocks that ALT would have evicted.

It is possible that ALT chooses to evict y before the next reference to x or y; in this case, if ALT′ chooses to evict x, the contents of the cache for ALT and ALT′ are identical. Further, ALT′ has the same number of cache misses as ALT, but it differs from MIN at a later point than ALT. This contradicts our assumption above, so we can exclude this case.

Eventually, the system will reference x, the block that ALT chose to evict; by construction, this occurs before the reference to y, the block that ALT′ chose to evict. Thus, ALT will have a cache miss, but ALT′ will not. ALT will evict some block, q, to make room for x; now ALT and ALT′ differ only in that ALT contains y and ALT′ contains q. (If ALT evicts y instead, then ALT and ALT′ have the same cache contents, but ALT′ has fewer misses than ALT, a contradiction.) Finally, when we reach the reference to y, ALT′ will take a cache miss. If ALT′ evicts q, then it will have the same number of cache misses as ALT, but it will differ from MIN at a point later than ALT, a contradiction.

As with Shortest Job First, MIN requires knowledge of the future, and so we cannot implement it directly. Rather, we can use it as a goal: we want to come up with mechanisms which are effective at predicting which blocks will be used in the near future, so that we can keep those in the cache.

If we were able to predict the future, we could do even better than MIN by prefetching blocks so that they arrive “just in time” — exactly when they are needed. In the best case, this can reduce the number of cache misses to zero. For example, if we observe a program scanning through a file, we can prefetch the blocks of the file into memory. Provided we can read the file into memory fast enough to keep up with the program, the program will always find its data in memory and never have a cache miss.

9.5.4 Least Recently Used (LRU)

One way to predict the future is to look at the past. If programs exhibit temporal locality, the locations they reference in the future are likely to be the same as the ones they have referenced in the recent past.

A replacement policy that captures this effect is to evict the block that has not been used for the longest period of time, or the least recently used (LRU) block. In software, LRU is simple to implement: on every cache hit, you move the block to the front of the list, and on a cache miss, you evict the block at the end of the list. In hardware, keeping a linked list of cached blocks is too complex to implement at high speed; instead, we need to approximate LRU, and we will discuss exactly how in a bit.

LRU

Ref.	A	B	A	C	B	D	A	D	E	D	A	E	B	A	C
1	A		+				+				+			+	
2		B			+								+		
3				C					E			+			
4						D		+		+					C

FIFO

1	A		+				+		E						
2		B			+						A			+	
3				C								+	B		
4						D		+		+					C

MIN

1	A		+				+				+			+	
2		B			+								+		C
3				C					E			+			
4						D		+		+					

Figure 9.14: Cache behavior for LRU (top), FIFO (middle), and MIN (bottom) for a reference pattern that exhibits temporal locality. Each row represents the contents of a page frame or cache block; + indicates a cache hit. On this reference pattern, LRU is the same as MIN up to the final reference, where MIN can choose to replace any block.

In some cases, LRU can be optimal, as in the example in Figure 9.14. The table illustrates a reference pattern that exhibits a high degree of temporal locality; when recent references are more likely to be referenced in the near future, LRU can outperform FIFO.

LRU

Ref.	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A				E				D				C		
2		B				A				E				D	
3			C				B				A				E
4				D				C				B			
	MIN														
1	A					+					+			+	
2		B					+					+	C		
3			C					+	D					+	
4				D	E					+					+

Figure 9.15: Cache behavior for LRU (top) and MIN (bottom) for a reference pattern that repeatedly scans through memory. Each row represents the contents of a page frame or cache block; + indicates a cache hit. On this reference pattern, LRU is the same as FIFO, with a cache miss on every reference; the optimal strategy is to replace the most recently used page, as that will be referenced farthest into the future.

On this particular sequence of references, LRU behaves similarly to the optimal strategy MIN, but that will not always be the case. In fact, LRU can sometimes be the worst possible cache replacement policy. This occurs whenever the least recently used block is the next one to be referenced. A common situation where LRU is pessimal is when the program makes repeated scans through memory, illustrated in Figure 9.15; we saw earlier that FIFO is also pessimal for this reference pattern. The best possible strategy is to replace the most recently referenced block, as this block will be used farthest into the future.

9.5.5 Least Frequently Used (LFU)

Consider again the case of a web proxy cache. Whenever a user accesses a page, it is more likely for that user to access other nearby pages (spatial locality); sometimes, as with a flash crowd, it can be more likely for other users to access the same page (temporal locality). On the surface, Least Recently Used seems like a good fit for this workload.

However, when a user visits a rarely used page, LRU will treat the page as important, even though it is probably just a one-off. When I do a Google search for a mountain hut for a stay in Western Iceland, the web pages I visit will not suddenly become more popular than the latest Facebook update from Katy Perry.

A better strategy for references that follow a Zipf distribution is Least Frequently Used (LFU). LFU discards the block that has been used least often; it therefore keeps popular pages, even when less popular pages have been touched more recently.

LRU and LFU both attempt to predict future behavior, and they have complementary strengths. Many systems meld the two approaches to gain the benefits of each. LRU is better at keeping the current working set in memory; once the working set is taken care of, however, LRU will yield diminishing returns. Instead, LFU may be better at predicting what files or memory blocks will be needed in the more distant future, e.g., after the next working set phase change.

Replacement policy and file size

Our discussion up to now has assumed that all cached items are equal, both in size and in cost to replace. When these assumptions do not hold, however, we may sometimes want to vary the policy from LFU or LFU, that is, to keep some items that are less frequently or less recently used ahead of others that are more frequently or more recently used.

For example, consider a web proxy that caches files to improve web responsiveness. These files may have vastly different sizes. When making room for a new file, we have a choice between evicting one very large web page object or a much larger number of smaller objects. Even if each small file is less frequently used than the large file, it may still make sense to keep the small files. In aggregate they may be more frequently used, and therefore they may have a larger benefit to overall system performance. Likewise, if a cached item is expensive to regenerate, it is more important to keep cached than one that is more easily replaced.

Parallel computing makes the calculus even more complex. The performance of a parallel program depends on its critical path — the minimum sequence of steps for the program to produce its result. Cache misses that occur on the critical path affect the response time while those that occur off the critical path do not. For example, a parallel MapReduce job forks a set of tasks onto processors; each task reads in a file and produces an output. Because MapReduce must wait until all tasks are complete before moving onto the next step, if any file is not cached it is as bad as if all of the needed files were not cached.

9.5.6 Belady's Anomaly

Intuitively, it seems like it should always help to add space to a memory cache; being able to store more blocks should always either improve the cache hit rate, or at least, not make the cache hit rate any worse. For many cache replacement strategies, this intuition is true. However, in some cases, adding space to a cache can actually hurt the cache hit rate. This is called *Belady's anomaly*, after the person that discovered it.

First, we note that many of the schemes we have defined can be proven to yield no worse

cache behavior with larger cache sizes. For example, with the optimal strategy MIN, if we have a cache of size k blocks, we will keep the next k blocks that will be referenced. If we have a cache of size k + 1 blocks, we will keep all of the same blocks as with a k sized cache, plus the additional block that will be the k + 1 next reference.

We can make a similar argument for LRU and LFU. For LRU, a cache of size k + 1 keeps all of the same blocks as a k sized cache, plus the block that is referenced farthest in the past. Even if LRU is a lousy replacement policy — if it rarely keeps the blocks that will be used in the near future — it will always do at least as well as a slightly smaller cache also using the same replacement policy. An equivalent argument can be used for LFU.

FIFO (3 slots)

Ref.	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					+
2		B			A			+		C		
3			C			B			+		D	

FIFO (4 slots)

1	A				+		E				D	
2		B				+		A				E
3			C						B			
4				D						C		

Figure 9.16: Cache behavior for FIFO with two different cache sizes, illustrating Belady's anomaly. For this sequence of references, the larger cache suffers ten cache misses, while the smaller cache has one fewer.

Some replacement policies, however, do not have this behavior. Instead, the contents of a cache with k + 1 blocks may be completely different than the contents of a cache with k blocks. As a result, there cache hit rates may diverge. Among the policies we have discussed, FIFO suffers from Belady's anomaly, and we illustrate that in Figure 9.16.

9.6 Case Study: Memory-Mapped Files

To illustrate the concepts presented in this chapter, we consider in detail how an operating system can implement *demand paging*. With demand paging, applications can access more memory than is physically present on the machine, by using memory pages as a cache for disk blocks. When the application accesses a missing memory page, it is transparently brought in from disk. We start with the simpler case of a demand paging for a single, memory-mapped file and then extend the discussion to managing multiple processes competing for space in main memory.

As we discussed in Chapter 3, most programs use explicit read/write system calls to perform file I/O. Read/write system calls allow the program to work on a *copy* of file data. The program opens a file and then invokes the system call read to copy chunks of file data into buffers in the program's address space. The program can then use and modify those chunks, without affecting the underlying file. For example, it can convert the file from the disk format into a more convenient in-memory format. To write changes back to the file, the program invokes the system call write to copy the data from the program buffers out to disk. Reading and writing files via system calls is simple to understand and reasonably efficient for small files.

An alternative model for file I/O is to map the file contents into the program's virtual address space. For a *memory-mapped file*, the operating system provides the illusion that the file is a program segment; like any memory segment, the program can directly issue instructions to load and store values to the memory. Unlike file read/write, the load and store instructions do not operate on a copy; they directly access and modify the contents of the file, treating memory as a write-back cache for disk.

We saw an example of a memory-mapped file in the previous chapter: the program executable image. To start a process, the operating system brings the executable image into memory, and creates page table entries to point to the page frames allocated to the executable. The operating system can start the program executing as soon as the first page frame is initialized, without waiting for the other pages to be brought in from disk. For this, the other page table entries are set to invalid — if the process accesses a page that has not reached memory yet, the hardware traps to the operating system and then waits until the page is available so it can continue to execute. From the program's perspective, there is no difference (except for performance) between whether the executable image is entirely in memory or still mostly on disk.

We can generalize this concept to any file stored on disk, allowing applications to treat any file as part of its virtual address space. File blocks are brought in by the operating system when they are referenced, and modified blocks are copied back to disk, with the bookkeeping done entirely by the operating system.

9.6.1 Advantages

Memory-mapped files offer a number of advantages:

- **Transparency.** The program can operate on the bytes in the file as if they are part of memory; specifically, the program can use a pointer into the file without needing to check if that portion of the file is in memory or not.
- **Zero copy I/O.** The operating system does not need to copy file data from kernel buffers into user memory and back; rather, it just changes the program's page table

entry to point to the physical page frame containing that portion of the file. The kernel is responsible for copying data back and forth to disk. We should note that it is possible to implement zero copy I/O for explicit read/write file system calls in certain restricted cases; we will explain how in the next chapter.

- **Pipelining.** The program can start operating on the data in the file as soon as the page tables have been set up; it does not need to wait for the entire file to be read into memory. With multiple threads, a program can use explicit read/write calls to pipeline disk I/O, but it needs to manage the pipeline itself.
- **Interprocess communication.** Two or more processes can share information instantaneously through a memory-mapped file without needing to shuffle data back and forth to the kernel or to disk. If the hardware architecture supports it, the page table for the shared segment can also be shared.
- **Large files.** As long as the page table for the file can fit in physical memory, the only limit on the size of a memory-mapped file is the size of the virtual address space. For example, an application may have a giant multi-level tree indexing data spread across a number of disks in a data center. With read/write system calls, the application needs to explicitly manage which parts of the tree are kept in memory and which are on disk; alternatively, with memory-mapped files, the application can leave that bookkeeping to the operating system.

9.6.2 Implementation

To implement memory-mapped files, the operating system provides a system call to map the file into a portion of the virtual address space. In the system call, the kernel initializes a set of page table entries for that region of the virtual address space, setting each entry to invalid. The kernel then returns to the user process.

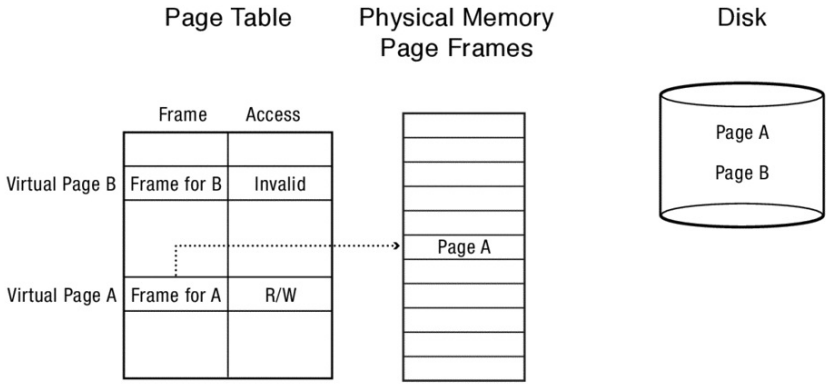

Figure 9.17: Before a page fault, the page table has an invalid entry for the referenced page and the data for the page is stored on disk.

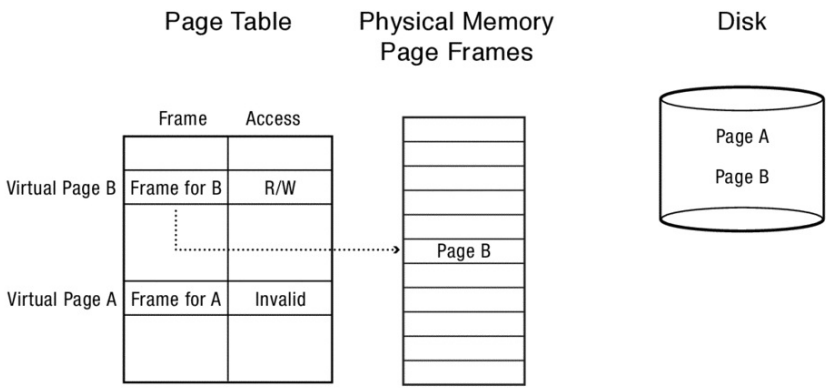

Figure 9.18: After the page fault, the page table has a valid entry for the referenced page with the page frame containing the data that had been stored on disk. The old contents of the page frame are stored on disk and the page table entry that previously pointed to the page frame is set to invalid.

When the process issues an instruction that touches an invalid mapped address, a sequence of events occurs, illustrated in Figures 9.17 and 9.18:

- **TLB miss.** The hardware looks the virtual page up in the TLB, and finds that there is not a valid entry. This triggers a full page table lookup in hardware.
- **Page table exception.** The hardware walks the multi-level page table and finds the page table entry is invalid. This causes a hardware *page fault* exception trap into the operating system kernel.
- **Convert virtual address to file offset.** In the exception handler, the kernel looks up in its segment table to find the file corresponding to the faulting virtual address and converts the address to a file offset.
- **Disk block read.** The kernel allocates an empty page frame and issues a disk operation to read the required file block into the allocated page frame. While the disk operation is in progress, the processor can be used for running other threads or processes.
- **Disk interrupt.** The disk interrupts the processor when the disk read finishes, and the scheduler resumes the kernel thread handling the page fault exception.
- **Page table update.** The kernel updates the page table entry to point to the page frame allocated for the block and sets the entry to valid.
- **Resume process.** The operating system resumes execution of the process at the instruction that caused the exception.
- **TLB miss.** The TLB still does not contain a valid entry for the page, triggering a full page table lookup.

- **Page table fetch.** The hardware walks the multi-level page table, finds the page table entry valid, and returns the page frame to the processor. The processor loads the TLB with the new translation, evicting a previous TLB entry, and then uses the translation to construct a physical address for the instruction.

To make this work, we need an empty page frame to hold the incoming page from disk. To create an empty page frame, the operating system must:

- **Select a page to evict.** Assuming there is not an empty page of memory already available, the operating system needs to select some page to be replaced. We discuss how to implement this selection in Section 9.6.3 below.
- **Find page table entries that point to the evicted page.** The operating system then locates the set of page table entries that point to the page to be replaced. It can do this with a *core map* — an array of information about each physical page frame, including which page table entries contain pointers to that particular page frame.
- **Set each page table entry to invalid.** The operating system needs to prevent anyone from using the evicted page while the new page is being brought into memory. Because the processor can continue to execute while the disk read is in progress, the page frame may temporarily contain a mixture of bytes from the old and the new page. Therefore, because the TLB may cache a copy of the old page table entry, a TLB shootdown is needed to evict the old translation from the TLB.
- **Copy back any changes to the evicted page.** If the evicted page was modified, the contents of the page must be copied back to disk before the new page can be brought into memory. Likewise, the contents of modified pages must also be copied back when the application closes the memory-mapped file.

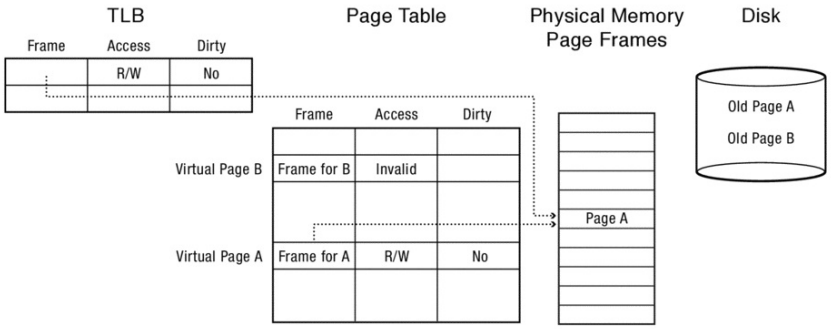

Figure 9.19: When a page is clean, its dirty bit is set to zero in both the TLB and the page table, and the data in memory is the same as the data stored on disk.

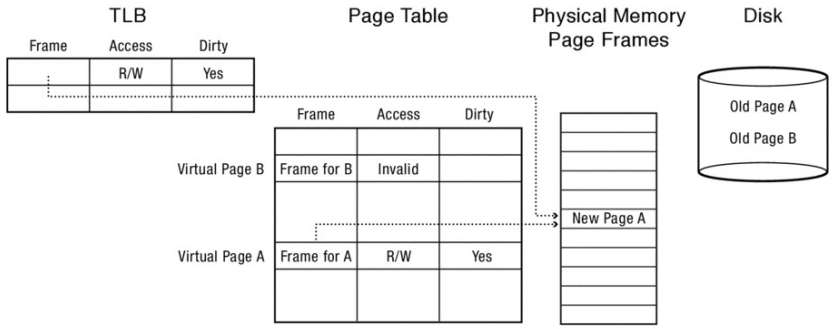

Figure 9.20: On the first store instruction to a clean page, the hardware sets the dirty bit for that page in the TLB and the page table. The contents of the page will differ from what is stored on disk.

How does the operating system know which pages have been modified? A correct, but inefficient, solution is to simply assume that every page in a memory-mapped file has been modified; if the data has not been changed, the operating system will have wasted some work, but the contents of the file will not be affected.

A more efficient solution is for the hardware to keep track of which pages have been modified. Most processor architectures reserve a bit in each page table entry to record whether the page has been modified. This is called a *dirty bit*. The operating system initializes the bit to zero, and the hardware sets the bit automatically when it executes a store instruction for that virtual page. Since the TLB can contain a copy of the page table entry, the TLB also needs a dirty bit per entry. The hardware can ignore the dirty bit if it is set in the TLB, but whenever it goes from zero to one, the hardware needs to copy the bit back to the corresponding page table entry. Figures 9.19 and 9.20 show the state of the TLB, page table, memory and disk before and after the first store instruction to a page.

If there are multiple page table entries pointing at the same physical page frame, the page is dirty (and must be copied back to disk) if *any* of the page tables have the dirty bit set. Normally, of course, a memory-mapped file will have a single page table shared between all of the processes mapping the file.

Because evicting a dirty page takes more time than evicting a clean page, the operating system can proactively clean pages in the background. A thread runs in the background, looking for pages that are likely candidates for being evicted if they were clean. If the hardware dirty bit is set in the page table entry, the kernel resets the bit in the page table entry and does a TLB shootdown to remove the entry from the TLB (with the old value of the dirty bit). It then copies the page to disk. Of course, the on-chip processor memory cache and write buffers can contain modifications to the page that have not reached main memory; the hardware ensures that the new data reaches main memory before those bytes are copied to the disk interface.

The kernel can then restart the application; it need not wait for the block to reach disk — if the process modifies the page again, the hardware will simply reset the dirty bit,

signaling that the block cannot be reclaimed without saving the new set of changes to disk.

Emulating a hardware dirty bit in software

Interestingly, hardware support for a dirty bit is not strictly required. The operating system can emulate a hardware dirty bit using page table access permissions. An unmodified page is set to allow only read-only access, even though the program is logically allowed to write the page. The program can then execute normally. On a store instruction to the page, the hardware will trigger a memory exception. The operating system can then record the fact that the page is dirty, upgrade the page protection to read-write, and restart the process.

To clean a page in the background, the kernel resets the page protection to read-only and does a TLB shootdown. The shootdown removes any translation that allows for read-write access to the page, forcing subsequent store instructions to cause another memory exception.

9.6.3 Approximating LRU

A further challenge to implementing demand paged memory-mapped files is that the hardware does not keep track of which pages are least recently or least frequently used. Doing so would require the hardware to keep a linked list of every page in memory, and to modify that list on every load and store instruction (and for memory-mapped executable images, every instruction fetch as well). This would be prohibitively expensive. Instead, the hardware maintains a minimal amount of access information per page to allow the operating system to approximate LRU or LFU if it wants to do so.

We should note that explicit read/write file system calls do not have this problem. Each time a process reads or writes a file block, the operating system can keep track of which blocks are used. The kernel can use this information to prioritize its cache of file blocks when the system needs to find space for a new block.

Most processor architectures keep a *use bit* in each page table entry, next to the hardware dirty bit we discussed above. The operating system clears the use bit when the page table entry is initialized; the bit is set in hardware whenever the page table entry is brought into the TLB. As with the dirty bit, a physical page is used if *any* of the page table entries have their use bit set.

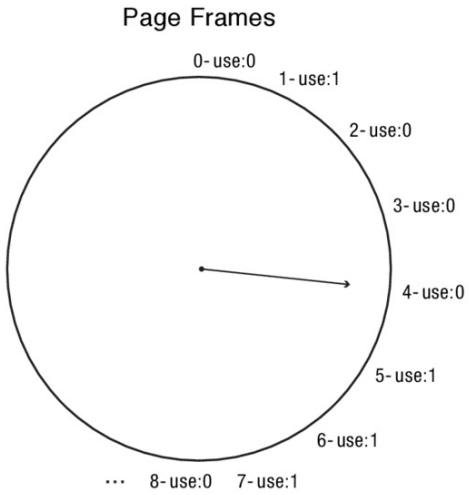

Figure 9.21: The clock algorithm sweeps through each page frame, collecting the current value of the use bit for that page and resetting the use bit to zero. The clock algorithm stops when it has reclaimed a sufficient number of unused page frames.

The operating system can leverage the use bit in various ways, but a commonly used approach is the *clock algorithm*, illustrated in Figure 9.21. Periodically, the operating system scans through the core map of physical memory pages. For each page frame, it records the value of the use bit in the page table entries that point to that frame, and then clears their use bits. Because the TLB can have a cached copy of the translation, the operating system also does a shootdown for any page table entry where the use bit is cleared. Note that if the use bit is already zero, the translation cannot be in the TLB. While it is scanning, the kernel can also look for dirty and recently unused pages and flush these out to disk.

Each sweep of the clock algorithm through memory collects one bit of information about page usage; by adjusting the frequency of the clock algorithm, we can collect increasingly fine-grained information about usage, at the cost of increased software overhead. On modern systems with hundreds of thousands and sometimes millions of physical page frames, the overhead of the clock algorithm can be substantial.

The policy for what to do with the usage information is up to the operating system kernel. A common policy is called *not recently used*, or k'th chance. If the operating system needs to evict a page, the kernel picks one that has not been used (has not had its use bit set) for the last k sweeps of the clock algorithm. The clock algorithm partitions pages based on how recently they have been used; among page frames in the same k'th chance partition, the operating system can evict pages in FIFO order.

Some systems trigger the clock algorithm only when a page is needed, rather than periodically in the background. Provided some pages have not been accessed since the last sweep, an on-demand clock algorithm will find a page to reclaim. If all pages have been accessed, e.g., if there is a storm of page faults due to phase change behavior, then the system will default to FIFO.

Emulating a hardware use bit in software

Hardware support for a use bit is also not strictly required. The operating system kernel can emulate a use bit with page table permissions, in the same way that the kernel can emulate a hardware dirty bit. To collect usage information about a page, the kernel sets the page table entry to be invalid even though the page is in memory and the application has permission to access the page. When the page is accessed, the hardware will trigger an exception and the operating system can record the use of the page. The kernel then changes the permission on the page to allow access, before restarting the process. To collect usage information over time, the operating system can periodically reset the page table entry to invalid and shootdown any cached translations in the TLB.

Many systems use a hybrid approach. In addition to active pages where the hardware collects the use bit, the operating system maintains a pool of unused, clean page frames that are unmapped in any virtual address space, but still contain their old data. When a new page frame is needed, pages in this pool can be used without any further work. However, if the old data is referenced before the page frame is reused, the page can be pulled out of the pool and mapped back into the virtual address space.

Systems with a software-managed TLB have an even simpler time. Each time there is a TLB miss with a software-managed TLB, there is a trap to the kernel to look up the translation. During the trap, the kernel can update its list of frequently used pages.

9.7 Case Study: Virtual Memory

We can generalize on the concept of memory-mapped files, by backing *every* memory segment with a file on disk. This is called *virtual memory*. Program executables, individual libraries, data, stack and heap segments can all be demand paged to disk. Unlike memory-mapped files, though, process memory is ephemeral: when the process exits, there is no need to write modified data back to disk, and we can reclaim the disk space.

The advantage of virtual memory is flexibility. The system can continue to function even though the user has started more processes than can fit in main memory at the same time. The operating system simply makes room for the new processes by paging the memory of idle applications to disk. Without virtual memory, the user has to do memory management by hand, closing some applications to make room for others.

All of the mechanisms we have described for memory-mapped files apply when we generalize to virtual memory, with one additional twist. We need to balance the allocation of physical page frames between processes. Unfortunately, this balancing is quite tricky. If we add a few extra page faults to a system, no one will notice. However, a modern disk can handle at most 100 page faults per second, while a modern multi-core processor can execute 10 billion instructions per second. Thus, if page faults are anything but extremely rare, performance will suffer.

9.7.1 Self-Paging

One consideration is that the behavior of one process can significantly hurt the performance of other programs running at the same time. For example, suppose we have two processes. One is a normal program, with a working set equal to say, a quarter of physical memory. The other program is greedy; while it can run fine with less memory, it will run faster if it is given more memory. We gave an example of this earlier with the sort program.

Can you design a program to take advantage of the clock algorithm to acquire more than its fair share of memory pages?

```
static char *workingSet;    // The memory this program wants to acquire.
static int soFar;           // How many pages the program has so far.
static sthread_t refreshThread;

// This thread touches the pages we have in memory, to keep them recently used.
void refresh () {
    int i;

    while (1) {
        // Keep every page in memory recently used.
        for (i = 0; i < soFar; i += PAGESIZE)
            workingSet[i] = 0;
    }
}

int main (int argc, char **argv) {
    // Allocate a giant array.
    workingSet = malloc(ARRAYSIZE);
    soFar = 0;

    // Create a thread to keep our pages in memory, once they get there.
    thread_create(&refreshThread, refresh, 0);

    // Touch every page to bring it into memory.
    for (; soFar < ARRAYSIZE; soFar += PAGESIZE)
        workingSet[soFar] = 0;

    // Now that everything is in memory, run computation.
    // ...
}
```

Figure 9.22: The “pig” program to greedily acquire memory pages. The implementation assumes we are running on a multicore computer. When the pig triggers a page fault by touching a new memory page (soFar), the operating system will find all of the pig’s pages up to soFar recently used. The operating system will keep these in memory and it will

choose to evict a page from some other application.

We give an example in Figure 9.22, which we will dub “pig” for obvious reasons. It allocates an array in virtual memory equal in size to physical memory; it then uses multiple threads to cycle through memory, causing each page to be brought in while the other pages remain very recently used.

A normal program sharing memory with the pig will eventually be frozen out of memory and stop making progress. When the pig touches a new page, it triggers a page fault, but all of its pages are recently used because of the background thread. Meanwhile, the normal program will have recently touched many of its pages but there will be some that are less recently used. The clock algorithm will choose those for replacement.

As time goes on, more and more of the pages will be allocated to the pig. As the number of pages assigned to the normal program drops, it starts experiencing page faults at an increasing frequency. Eventually, the number of pages drops below the working set, at which point the program stops making much progress. Its pages are even less frequently used, making them easier to evict.

Of course, a normal user would probably never run (or write!) a program like this, but a malicious attacker launching a computer virus might use this approach to freeze out the system administrator. Likewise, in a data center setting, a single server can be shared between multiple applications from different users, for example, running in different virtual machines. It is in the interest of any single application to acquire as many physical resources as possible, even if that hurts performance for other users.

A widely adopted solution is *self-paging*. With self-paging, each process or user is assigned its fair share of page frames, using the max-min scheduling algorithm we described in Chapter 7. If all of the active processes can fit in memory at the same time, the system does not need to page. As the system starts to page, it evicts the page from whichever process has the most allocated to it. Thus, the pig would only be able to allocate its fair share of page frames, and beyond that any page faults it triggers would evict its own pages.

Unfortunately, self-paging comes at a cost in reduced resource utilization. Suppose we have two processes, both of which allocate large amounts of virtual address space. However, the working sets of the two programs can fit in memory at the same time, for example, if one working set takes up 2/3rds of memory and the other takes up 1/3rd. If they cooperate, both can run efficiently because the system has room for both working sets. However, if we need to bulletproof the operating system against malicious programs by self-paging, then each will be assigned half of memory and the larger program will thrash.

9.7.2 Swapping

Another issue is what happens as we increase the workload for a system with virtual memory. If we are running a data center, for example, we can share physical machines among a much larger number of applications each running in a separate virtual machine. To reduce costs, the data center needs to support the maximum number of applications on each server, within some performance constraint.

If the working sets of the applications easily fit in memory, then as page faults occur, the clock algorithm will find lightly used pages — that is, those outside of the working set of any process — to evict to make room for new pages. So far so good. As we keep adding active processes, however, their working sets may no longer fit, even if each process is given their fair share of memory. In this case, the performance of the system will degrade dramatically.

This can be illustrated by considering how system throughput is affected by the number of processes. As we add work to the system, throughput increases as long as there is enough processing capacity and I/O bandwidth. When we reach the point where there are too many tasks to fit entirely in memory, the system starts demand paging. Throughput can continue to improve if there are enough lightly used pages to make room for new tasks, but eventually throughput levels off and then falls off a cliff. In the limit, every instruction will trigger a page fault, meaning that the processor executes at 100 instructions per second, rather than 10 billion instructions per second. Needless to say, the user will think the system is dead even if it is in fact inching forward very slowly.

As we explained in the Chapter 7 discussion on overload control, the only way to achieve good performance in this case is to prevent the overload condition from occurring. Both response time and throughput will be better if we prevent additional tasks from starting or if we remove some existing tasks. It is better to completely starve some tasks of their resources, if the alternative, assigning each task their fair share, will drag the system to a halt.

Evicting an entire process from memory is called *swapping*. When there is too much paging activity, the operating system can prevent a catastrophic degradation in performance by moving *all* of the page frames of a particular process to disk, preventing it from running at all. Although this may seem terribly unfair, the alternative is that *every* process, not just the swapped process, will run much more slowly. By distributing the swapped process’s pages to other processes, we can reduce the number of page faults, allowing system performance to recover. Eventually the other tasks will finish, and we can bring the swapped process back into memory.

9.8 Summary and Future Directions

Caching is central to many areas of computer science: caches are used in processor design, file systems, web browsers, web servers, compilers, and kernel memory management, to name a few. To understand these systems, it is important to understand how caches work, and even more importantly, when they fail.

The management of memory in operating systems is a particularly useful case study. Every major commercial operating system includes support for demand paging of memory, using memory as a cache for disk. Often, application memory pages and blocks in the file buffer are allocated from a common pool of memory, where the operating system attempts to keep blocks that are likely to be used in memory and evicting those blocks that are less likely to be used. However, on modern systems, the difference between

finding a block in memory and needing to bring it in from disk can be as much as a factor of 100,000. This makes virtual memory paging fragile, acceptable only when used in small doses.

Moving forward, several trends are in progress:

- **Low latency backing store.** Due to the weight and power drain of magnetic disks, many portable devices have moved to solid state persistent storage, such as non-volatile RAM. Current solid state storage devices have significantly lower latency than disk, and even faster devices are likely in the future. Similarly, the move towards data center computing has added a new option to memory management: using DRAM on other nodes in the data center as a low-latency, very high capacity backing store for local memory. Both of these trends reduce the cost of paging, making it relatively more attractive.
- **Variable page sizes.** Many systems use a standard 4 KB page size, but there is nothing fundamental about that choice — it is a tradeoff chosen to balance internal fragmentation, page table overhead, disk latency, the overhead of collecting dirty and usage bits, and application spatial locality. On modern disks, it only takes twice as long to transfer 256 contiguous pages as it does to transfer one, so internally, most operating systems arrange disk transfers to include many blocks at a time. With new technologies such as low latency solid state storage and cluster memory, this balance may shift back towards smaller effective page sizes.
- **Memory aware applications.** The increasing depth and complexity of the memory hierarchy is both a boon and a curse. For many applications, the memory hierarchy delivers reasonable performance without any special effort. However, the wide gulf in performance between the first level cache and main memory, and between main memory and disk, implies that there is a significant performance benefit to tuning applications to the available memory. The poses a particular challenge for operating systems to adapt to applications that are adapting to their physical resources.

Exercises

1. A computer system has a 1 KB page size and keeps the page table for each process in main memory. Because the page table entries are usually cached on chip, the average overhead for doing a full page table lookup is 40 ns. To reduce this overhead, the computer has a 32-entry TLB. A TLB lookup requires 1 ns. What TLB hit rate is required to ensure an average virtual address translation time of 2 ns?
2. Most modern computer systems choose a page size of 4 KB.
   a. Give a set of reasons why doubling the page size might increase performance.
   b. Give a set of reasons why doubling the page size might decrease performance.
3. For each of the following statements, indicate whether the statement is true or false, and explain why.
   a. A direct mapped cache can sometimes have a higher hit rate than a fully associative cache (on the same reference pattern).
   b. Adding a cache never hurts performance.
4. Suppose an application is assigned 4 pages of physical memory and the memory is initially empty. It then references pages in the following sequence:

   ACBDBAEFBFAGEFA

   a. Show how the system would fault pages into the four frames of physical memory, using the LRU replacement policy.
   b. Show how the system would fault pages into the four frames of physical memory, using the MIN replacement policy.
   c. Show how the system would fault pages into the four frames of physical memory, using the clock replacement policy.
5. Is least recently used a good cache replacement algorithm to use for a workload following a zipf distribution? Briefly explain why or why not.
6. Briefly explain how to simulate a modify bit per page for the page replacement algorithm if the hardware does not provide one.
7. Suppose we have four programs:
   a. One exhibits both spatial and temporal locality.
   b. One touches each page sequentially, and then repeats the scan in a loop.
   c. One references pages according to a Zipf distribution (e.g., it is a web server and its memory consists of cached web pages).
   d. One generates memory references completely at random using a uniform random number generator.

   All four programs use the same total amount of virtual memory — that is, they both touch N distinct virtual pages, amongst a much larger number of total references.

   For each program, sketch a graph showing the rate of progress (instructions per unit time) of each program as a function of the physical memory available to the program, from 0 to N, assuming the page replacement algorithm approximates least recently used.
8. Suppose a program repeatedly scans linearly through a large array in virtual memory. In other words, if the array is four pages long, its page reference pattern is ABCDABCDABCD…

   For each of the following page replacement algorithms, sketch a graph showing the rate of progress (instructions per unit time) of each program as a function of the physical memory available to the program, from 0 to N, where N is sufficient to hold the entire array.

   a. FIFO
   b. Least recently used
   c. Clock algorithm
   d. Nth chance algorithm
   e. MIN
9. Consider a computer system running a general-purpose workload with demand

paging. The system has two disks, one for demand paging and one for file system operations. Measured utilizations (in terms of time, not space) are given in Figure 9.23.

Processor utilization	20.0%
Paging Disk	99.7%
File Disk	10.0%
Network	5.0%

Figure 9.23: Measured utilizations for sample system under consideration.

For each of the following changes, say what its likely impact will be on processor utilization, and explain why. Is it likely to significantly increase, marginally increase, significantly decrease, marginally decrease, or have no effect on the processor utilization?

a. Get a faster CPU
b. Get a faster paging disk
c. Increase the degree of multiprogramming

10. An operating system with a physically addressed cache uses page coloring to more fully utilize the cache.
    a. How many page colors are needed to fully utilize a physically addressed cache, with 1 TB of main memory, an 8 MB cache with 4-way set associativity, and a 4 KB page size?
    b. Develop an algebraic formula to compute the number of page colors needed for an arbitrary configuration of cache size, set associativity, and page size.
11. The sequence of virtual pages referenced by a program has length p with n distinct page numbers occurring in it. Let m be the number of page frames that are allocated to the process (all the page frames are initially empty). Let n > m.
    a. What is the lower bound on the number of page faults?
    b. What is the upper bound on the number of page faults?

    The lower/upper bound should be for any page replacement policy.
12. You have decided to splurge on a low end netbook for doing your operating systems homework during lectures in your non-computer science classes. The netbook has a single-level TLB and a single-level, physically addressed cache. It also has two levels of page tables, and the operating system does demand paging to disk.

    The netbook comes in various configurations, and you want to make sure the configuration you purchase is fast enough to run your applications. To get a handle on this, you decide to measure its cache, TLB and paging performance running your applications in a virtual machine. Figure 9.24 lists what you discover for your workload.

Measurement	Value
$P_{CacheMiss}$ = probability of a cache miss	0.01
$P_{TLBmiss}$ = probability of a TLB miss	0.01
P_{fault} = probability of a page fault, given a TLB miss occurs	0.00002
T_{cache} = time to access cache	1 ns = 0.001 µs
T_{TLB} = time to access TLB	1 ns = 0.001 µs
T_{DRAM} = time to access main memory	100 ns = 0.1 µs
T_{disk} = time to transfer a page to/from disk	10^7 ns = 10 ms

Figure 9.24: Sample measurements of cache behavior on the low-end netbook described in the exercises.

The TLB is refilled automatically by the hardware on a TLB miss. The page tables are kept in physical memory and are not cached, so looking up a page table entry incurs two memory accesses (one for each level of the page table). You may assume the operating system keeps a pool of clean pages, so pages do not need to be written back to disk on a page fault.

a. What is the average memory access time (the time for an application program to do one memory reference) on the netbook? Express your answer algebraically and compute the result to two
significant digits.
b. The netbook has a few optional performance enhancements:

Item	Specs	Price
Faster disk drive	Transfers a page in 7 ms	$100
500 MB more DRAM	Makes probability of a page fault 0.00001	$100

Faster network card	Allows paging to remote memory.	$100

With the faster network card, the time to access remote memory is 500 ms, and the probability of a remote memory miss (need to go to disk), given there is a page fault is 0.5.

Suppose you have $200. What options should you buy to maximize the performance of the netbook for this workload?

13. On a computer with virtual memory, suppose a program repeatedly scans through a very large array. In other words, if the array is four pages long, its page reference pattern is ABCDABCDABCD…

    Sketch a graph showing the paging behavior, for each of the following page replacement algorithms. The y-axis of the graph is the number of page faults per referenced page, varying from 0 to 1; the x-axis is the size of the array being scanned, varying from smaller than physical memory to much larger than physical memory. Label any interesting points on the graph on both the x and y axes.

    a. FIFO
    b. LRU
    c. Clock
    d. MIN

14. Consider two programs, one that exhibits spatial and temporal locality, and the other that exhibits neither. To make the comparison fair, they both use the same total amount of virtual memory — that is, they both touch N distinct virtual pages, among a much larger number of total references.

    Sketch graphs showing the rate of progress (instructions per unit time) of each program as a function of the physical memory available to the program, from 0 to N, assuming the clock algorithm is used for page replacement.

    a. Program exhibiting locality, running by itself
    b. Program exhibiting no locality, running by itself
    c. Program exhibiting locality, running with the program exhibiting no locality (assume both have the same value for N).
    d. Program exhibiting no locality, running with the program exhibiting locality (assume both have the same N).

15. Suppose we are using the clock algorithm to decide page replacement, in its simplest form (“first-chance” replacement, where the clock is only advanced on a page fault and not in the background).

    A crucial issue in the clock algorithm is how many page frames must be considered in order to find a page to replace. Assuming we have a sequence of F page faults in a system with P page frames, let C(F,P) be the number of pages considered for replacement in handling the F page faults (if the clock hand sweeps by a page frame multiple times, it is counted each time).

    a. Give an algebraic formula for the minimum possible value of C(F,P).
    b. Give an algebraic formula for the maximum possible value of C(F,P).

10. Advanced Memory Management

All problems in computer science can be solved by another level of indirection. —*David Wheeler*

At an abstract level, an operating system provides an execution context for application processes, consisting of limits on privileged instructions, the process's memory regions, a set of system calls, and some way for the operating system to periodically regain control of the processor. By interposing on that interface — most commonly, by catching and transforming system calls or memory references — the operating system can transparently insert new functionality to improve system performance, reliability, and security.

Interposing on system calls is straightforward. The kernel uses a table lookup to determine which routine to call for each system call invoked by the application program. The kernel can redirect a system call to a new enhanced routine by simply changing the table entry.

A more interesting case is the memory system. Address translation hardware provides an efficient way for the operating system to monitor and gain control on every memory reference to a specific region of memory, while allowing other memory references to continue unaffected. (Equivalently, software-based fault isolation provides many of the same hooks, with different tradeoffs between interposition and execution speed.) This makes address translation a powerful tool for operating systems to introduce new, advanced services to applications. We have already shown how to use address translation for:

- **Protection.** Operating systems use address translation hardware, along with segment and page table permissions, to restrict access by applications to privileged memory locations such as those in the kernel.
- **Fill-on-demand/zero-on-demand.** By setting some page table permissions to invalid, the kernel can start executing a process before all of its code and data has been loaded into memory; the hardware will trap to the kernel if the process references data before it is ready. Similarly, the kernel can zero data and heap pages in the background, relying on page reference faults to catch the first time an application uses an empty page. The kernel can also allocate memory for kernel and user stacks only as needed. By marking unused stack pages as invalid, the kernel needs to allocate those pages only if the program executes a deep procedure call chain.
- **Copy-on-write.** Copy-on-write allows multiple processes to have logically separate copies of the same memory region, backed by a single physical copy in memory. Each page in the region is mapped read-only in each process; the operating system makes a physical copy only when (and if) a page is modified.
- **Memory-mapped files.** Disk files can be made part of a process's virtual address space, allowing the process to access the data in the file using normal processor instructions. When a page from a memory-mapped file is first accessed, a protection

fault traps to the operating system so that it can bring the page into memory from disk. The first write to a file block can also be caught, marking the block as needing to be written back to disk.

- **Demand paged virtual memory.** The operating system can run programs that use more memory than is physically present on the computer, by catching references to pages that are not physically present and filling them from disk or cluster memory.

In this chapter, we explore how to construct a number of other advanced operating system services by catching and re-interpreting memory references and system calls.

Chapter roadmap:

- **Zero-Copy I/O.** How do we improve the performance of transferring blocks of data between user-level programs and hardware devices? (Section 10.1)
- **Virtual Machines.** How do we execute an operating system on top of another operating system, and how can we use that abstraction to introduce new operating system services? (Section 10.2)
- **Fault Tolerance.** How can we make applications resilient to machine crashes? (Section 10.3)
- **Security.** How can we contain malicious applications that can exploit unknown faults inside the operating system? (Section 10.4)
- **User-Level Memory Management.** How do we give applications control over how their memory is managed? (Section 10.5)

10.1 Zero-Copy I/O

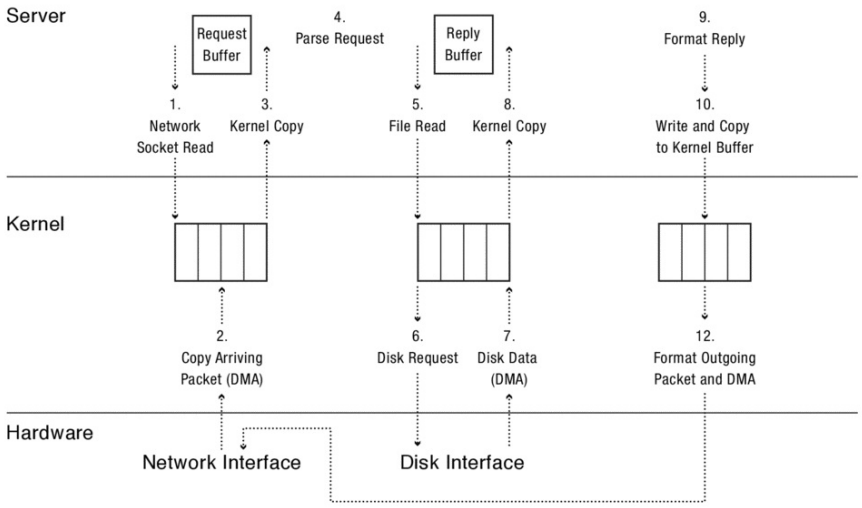

Figure 10.1: A web server gets a request from the network. The server first asks the kernel to copy the requested file from disk or its file buffer into the server's address space. The server then asks the kernel to copy the contents of the file back out to the network.

A common task for operating systems is to stream data between user-level programs and physical devices such as disks and network hardware. However, this streaming can be expensive in processing time if the data is copied as it moves across protection boundaries. A network packet needs to go from the network interface hardware, into kernel memory, and then to user-level; the response needs to go from user-level back into kernel memory and then from kernel memory to the network hardware.

Consider the operation of the web server, as pictured in Figure 10.1. Almost all web servers are implemented as user-level programs. This way, it is easy to reconfigure server behavior, and bugs in the server implementation do not necessarily compromise system security.

A number of steps need to happen for a web server to respond to a web request. For this example, assume that the connection between the client and server is already established, there is a server thread allocated to each client connection, and we use explicit read/write system calls rather than memory mapped files.

- **Server reads from network.** The server thread calls into the kernel to wait for an arriving request.
- **Packet arrival.** The web request arrives from the network; the network hardware uses DMA to copy the packet data into a kernel buffer.
- **Copy packet data to user-level.** The operating system parses the packet header to determine which user process is to receive the web request. The kernel copies the data into the user-level buffer provided by the server thread and returns to user-level.

- **Server reads file.** The server parses the data in the web request to determine which file is requested. It issues a file read system call back to the kernel, providing a user-level buffer to hold the file contents.
- **Data arrival.** The kernel issues the disk request, and the disk controller copies the data from the disk into a kernel buffer. If the file data is already in the file buffer cache, as will often be the case for popular web requests, this step is skipped.
- **Copy file data to user-level.** The kernel copies the data into the buffer provided by the user process and returns to user-level.
- **Server write to network.** The server turns around and hands the buffer containing the file data back to the kernel to send out to the network.
- **Copy data to kernel.** The kernel copies the data from the user-level buffer into a kernel buffer, formats the packet, and issues the request to the network hardware.
- **Data send.** The hardware uses DMA to copy the data from the kernel buffer out to the network.

Although we have illustrated this with a web server, a similar process occurs for any application that streams data in or out of a computer. Examples include a web client, online video or music service, BitTorrent, network file systems, and even a software download. For each of these, data is copied from hardware into the kernel and then into user-space, or vice versa.

We could eliminate the extra copy across the kernel-user boundary by moving each of these applications into the kernel. However, that would be impractical as it would require trusting the applications with the full power of the operating system. Alternately, we could modify the system call interface to allow applications to directly manipulate data stored in a kernel buffer, without first copying it to user memory. However, this is not a general-purpose solution; it would not work if the application needed to do any work on the buffer as opposed to only transferring it from one hardware device to another.

Instead, two solutions to *zero-copy I/O* are used in practice. Both eliminate the copy across the kernel-user boundary for large blocks of data; for small chunks of data, the extra copy does not hurt performance.

The more widely used approach manipulates the process page table to simulate a copy. For this to work, the application must first align its user-level buffer to a page boundary. The user-level buffer is provided to the kernel on a read or write system call, and its alignment and size is up to the application.

The key idea is that a page-to-page copy from user to kernel space or vice versa can be simulated by changing page table pointers instead of physically copying memory.

For a copy from user-space to the kernel (e.g., on a network or file system write), the kernel changes the permissions on the page table entry for the user-level buffer to prevent it from being modified. The kernel must also *pin* the page to prevent it from being evicted by the virtual memory manager. In the common case, this is enough — the page will not normally be modified while the I/O request is in progress. If the user program does try to modify the page, the program will trap to the kernel and the kernel can make an explicit copy at that point.

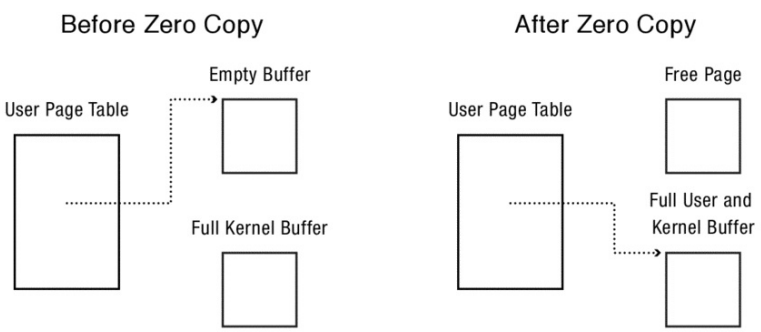

Figure 10.2: The contents of the page table before and after the kernel "copies" data to user-level by swapping the page table entry to point to the kernel buffer.

In the other direction, once the data is in the kernel buffer, the operating system can simulate a copy up to user-space by switching the pointer in the page table, as shown in Figure 10.2. The process page table originally pointed to the page frame containing the (empty) user buffer; now it points to the page frame containing the (full) kernel buffer. To the user program, the data appears exactly where it was expected! The kernel can reclaim any physical memory behind the empty buffer.

More recently, some hardware I/O devices have been designed to be able to transfer data to and from virtual addresses, rather than only to and from physical addresses. The kernel hands the virtual address of the user-level buffer to the hardware device. The hardware device, rather than the kernel, walks the multi-level page table to determine which physical page frame to use for the device transfer. When the transfer completes, the data is automatically where it belongs, with no extra work by the kernel. This procedure is a bit more complicated for incoming network packets, as the decision as to which process should receive which packet is determined by the contents of the packet header. The network interface hardware therefore has to parse the incoming packet to deliver the data to the appropriate process.

10.2 Virtual Machines

A virtual machine is a way for a host operating system to run a guest operating system as an application process. The host simulates the behavior of a physical machine so that the guest system behaves as if it was running on real hardware. Virtual machines are widely used on client machines to run applications that are not native to the current version of the operating system. They are also widely used in data centers to allow a single physical machine to be shared between multiple independent uses, each of which can be written as if it has system administrator control over the entire (virtual) machine. For example,

multiple web servers, representing different web sites, can be hosted on the same physical machine if they each run inside a separate virtual machine.

Address translation throws a wrinkle into the challenge of implementing a virtual machine, but it also opens up opportunities for efficiencies and new services.

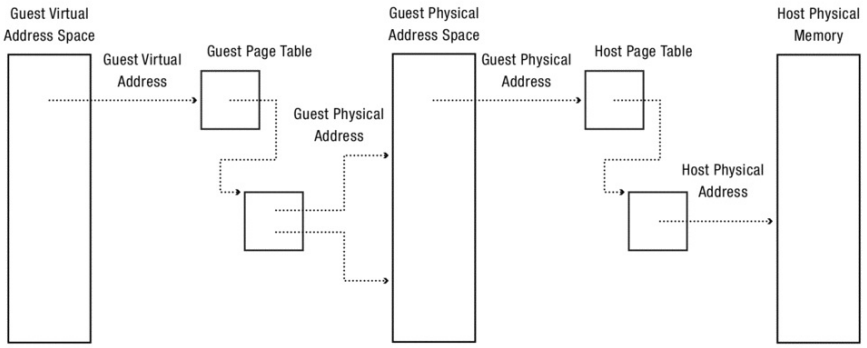

Figure 10.3: A virtual machine typically has two page tables: one to translate from guest process addresses to the guest physical memory, and one to translate from guest physical memory addresses to host physical memory addresses.

10.2.1 Virtual Machine Page Tables

With virtual machines, we have two sets of page tables, instead of one, as shown in Figure 10.3:

- **Guest physical memory to host physical memory.** The host operating system provides a set of page tables to constrain the execution of the guest operating system kernel. The guest kernel thinks it is running on real, physical memory, but in fact its addresses are virtual. The hardware page table translates each guest operating system memory reference into a physical memory location, after checking that the guest has permission to read or write each location. This way the host operating system can prevent bugs in the guest operating system from overwriting memory in the host, exactly as if the guest were a normal user-level process.
- **Guest user memory to guest physical memory.** In turn, the guest operating system manages page tables for its guest processes, exactly as if the guest kernel was running on real hardware. Since the guest kernel does not know anything about the physical page frames it has been assigned by the host kernel, these page tables translate from the guest process addresses to the guest operating system kernel addresses.

First, consider what happens when the host operating system transfers control to the guest kernel. Everything works as expected. The guest operating system can read and write its memory, and the hardware page tables provide the illusion that the guest kernel is running directly on physical memory.

Now consider what happens when the guest operating system transfers control to the guest process. The guest kernel is running at user-level, so its attempt to transfer of control is a privileged instruction. Thus, the hardware processor will first trap back to the host. The host kernel can then simulate the transfer instruction, handing control to the user process.

However, what page table should we use in this case? We cannot use the page table as set up by the guest operating system, as the guest operating system thinks it is running in physical memory, but it is actually using virtual addresses. Nor can we use the page table as set up by the host operating system, as that would provide permission to the guest process to access and modify the guest kernel data structures. If we grant access to the guest kernel memory to the guest process, then the behavior of the virtual machine will be compromised.

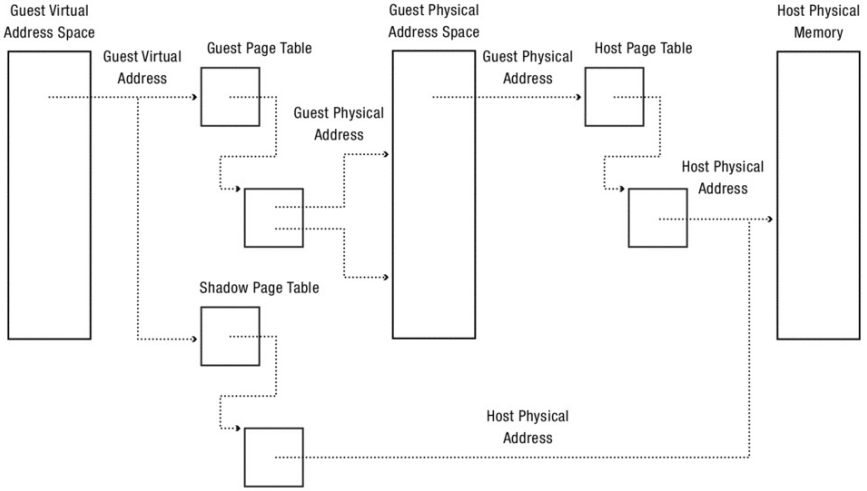

Figure 10.4: To run a guest process, the host operating system constructs a shadow page table consisting of the composition of the contents of the two page tables.

Instead, we need to construct a composite page table, called a *shadow page table*, that represents the composition of the guest page table and the host page table, as shown in Figure 10.4. When the guest kernel transfers control to a guest process, the host kernel gains control and changes the page table to the shadow version.

To keep the shadow page table up to date, the host operating system needs to keep track of changes that either the guest or the host operating systems make to their page tables. This is easy in the case of the host OS — it can check to see if any shadow page tables need to be updated before it changes a page table entry.

To keep track of changes that the guest operating system makes to its page tables, however, we need to do a bit more work. The host operating system sets the memory of the guest page tables as read-only. This ensures that the guest OS traps to the host every

time it attempts to change a page table entry. The host uses this trap to change the both the guest page table and the corresponding shadow page table, before resuming the guest operating system (with the page table still read-only).

Paravirtualization

One way to enable virtual machines to run faster is to assume that the guest operating system is *ported* to the virtual machine. The hardware dependent layer, specific to the underlying hardware, is replaced with code that understands about the virtual machine. This is called paravirtualization, because the resulting guest operating system is almost, but not precisely, the same as if it were running on real, physical hardware.

Paravirtualization is helpful in several ways. Perhaps the most important is handling the idle loop. What should happen when the guest operating system has no threads to run? If the guest believes it is running on physical hardware, then nothing — the guest spins waiting for more work to do, perhaps putting itself in low power mode. Eventually the hardware will cause a timer interrupt, transferring control to the host operating system. The host can then decide whether to resume the virtual machine or run some other thread (or even some other virtual machine).

With paravirtualization, however, the idle loop can be more efficient. The hardware dependent software implementing the idle loop can trap into the host kernel, yielding the processor immediately to some other use.

Likewise, with paravirtualization, the hardware dependent code inside the guest operating system can make explicit calls to the host kernel to change its page tables, removing the need for the host to simulate guest page table management.

The Intel architecture has recently added direct hardware support for the composition of page tables in virtual machines. Instead of a single page table, the hardware can be set up with two page tables, one for the host and one for the guest operating system. When running a guest process, on a TLB miss, the hardware translates the virtual address to a guest physical page frame using the guest page table, and the hardware then translates the guest physical page frame to the host physical page frame using the host page table. In other words, the TLB contains the composition of the two page tables, exactly as if the host maintained an explicit shadow page table. Of course, if the guest operating system itself hosts a virtual machine as a guest user process, then the guest kernel must construct a shadow page table.

Although this hardware support simplifies the construction of virtual machines, it is not clear if it improves performance. The handling of a TLB miss is slower since the host operating system must consult two page tables instead of one; changes to the guest page table are faster because the host does not need to maintain the shadow page table. It remains to be seen if this tradeoff is useful in practice.

10.2.2 Transparent Memory Compression

A theme running throughout this book is the difficulty of multiplexing multiplexors. With virtual machines, both the host operating system and the guest operating system are attempting to do the same task: to efficiently multiplex a set of tasks onto a limited amount of memory. Decisions the guest operating system takes to manage its memory may work at cross-purposes to the decisions that the host operating system takes to manage its memory.

Efficient use of memory can become especially important in data centers. Often, a single physical machine in a data center is configured to run many virtual machines at the same time. For example, one machine can host many different web sites, each of which is too small to merit a dedicated machine on its own.

To make this work, the system needs enough memory to be able to run many different operating systems at the same time. The host operating system can help by sharing memory between guest kernels, e.g., if it is running two guest kernels with the same executable kernel image. Likewise, the guest operating system can help by sharing memory between guest applications, e.g., if it is running two copies of the same program. However, if different guest kernels both run a copy of the same user process (e.g., both run the Apache web server), or use the same library, the host kernel has no (direct) way to share pages between those two instances.

Another example occurs when a guest process exits. The guest operating system places the page frames for the exiting process on the free list for reallocation to other processes. The contents of any data pages will never be used again; in fact, the guest kernel will need to zero those pages before they are reassigned. However, the host operating system has no (direct) way to know this. Eventually those pages will be evicted by the host, e.g., when they become least recently used. In the meantime, however, the host operating system might have evicted pages from the guest that are still active.

One solution is to more tightly coordinate the guest and host memory managers so that each knows what the other is doing. We discuss this in more detail later in this Chapter.

Commercial virtual machine implementations take a different approach, exploiting hardware address protection to manage the sharing of common pages between virtual machines. These systems run a scavenger in the background that looks for pages that can be shared across virtual machines. Once a common page is identified, the host kernel manipulates the page table pointers to provide the illusion that each guest has its own copy of the page, even though the physical representation is more compact.

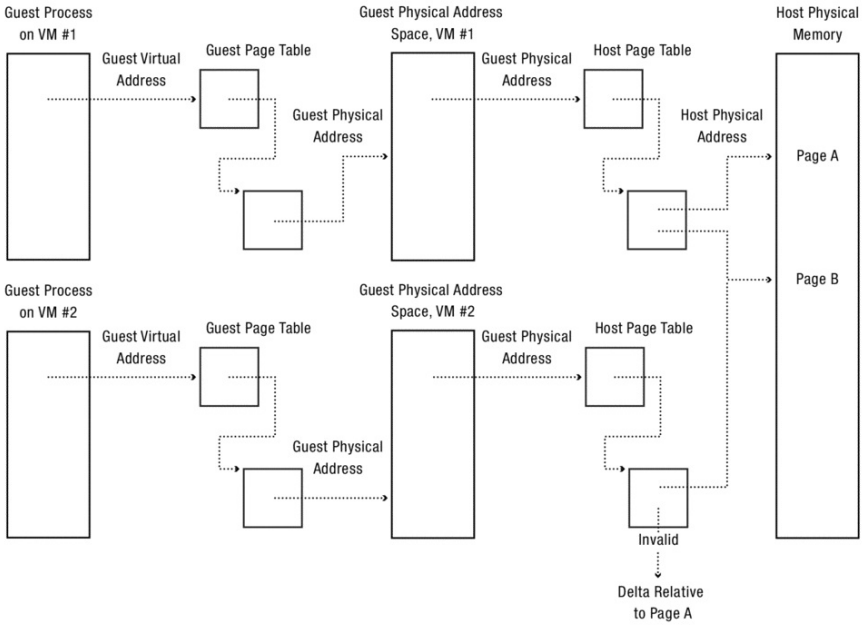

Figure 10.5: When a host kernel runs multiple virtual machines, it can save space by storing a delta to an existing page (page A) and by using the same physical page frame for multiple copies of the same page (page B).

There are two cases to consider, shown in Figure 10.5:

- **Multiple copies of the same page.** Two different virtual machines will often have pages with the same contents. An obvious case is zeroed pages: each kernel keeps a pool of pages that have been zeroed, ready to be allocated to a new process. If each guest operating system were running on its own machine, there would be little cost to keeping this pool at the ready; no one else but the kernel can use that memory. However, when the physical machine is shared between virtual machines, having each guest keep its own pool of zero pages is wasteful.

  Instead, the host can allocate a single zero page in physical memory for all of these instances. All pointers to the page will be set read-only, so that any attempt to modify the page will cause a trap to the host kernel; the kernel can then allocate a new (zeroed) physical page for that use, exactly as in copy-on-write. Of course, the guest kernels do not need to tell anyone when they create a zero page, so in the background, the host kernel runs a scavenger to look for zero pages in guest memory. When it finds one, it reclaims the physical page and changes the page table pointers to point at the shared zero page, with read-only permission.

  The scavenger can do the same for other shared page frames. The code and data segments for both applications and shared libraries will often be the same or quite similar, even across different operating systems. An application like the Apache web server will not be re-written from scratch for every separate operating system; rather, some OS-specific glue code will be added to match the portable portion of the application to its specific environment.

- **Compression of unused pages.** Even if a page is different, it may be close to some other page in a different virtual machine. For example, different versions of the operating system may differ in only some small respects. This provides an opportunity for the host kernel to introduce a new layer in the memory hierarchy to save space.

  Instead of evicting a relatively unused page, the operating system can compress it. If the page is a delta of an existing page, the compressed version may be quite small. The kernel manipulates page table permissions to maintain the illusion that the delta is a real page. The full copy of the page is marked read-only; the delta is marked invalid. If the delta is referenced, it can be re-constituted as a full page more quickly than if it was stored on disk. If the original page is modified, the delta can be re-compressed or evicted, as necessary.

10.3 Fault Tolerance

All systems break. Despite our best efforts, application code can have bugs that cause the process to exit abruptly. Operating system code can have bugs that cause the machine to halt and reboot. Power failures and hardware errors can also cause a system to stop without warning.

Most applications are structured to periodically save user data to disk for just these types of events. When the operating system or application restarts, the program can read the saved data off disk to allow the user to resume their work.

In this section, we take this a step further, to see if we can manage memory to recover application data structures after a failure, and not just user file data.

10.3.1 Checkpoint and Restart

One reason we might want to recover application data is when a program takes a long time to run. If a simulation of the future global climate takes a week to compute, we do not want to have to start again from scratch every time there is a power glitch. If enough machines are involved and the computation takes long enough, it is likely that at least one of the machines will encounter a failure sometime during the computation.

Of course, the program could be written to treat its internal data as precious — to periodically save its partial results to a file. To make sure the data is internally consistent, the program would need some natural stopping point; for example, the program can save the predicted climate for 2050 before it moves onto computing the climate in 2051.

A more general approach is to have the operating system use the virtual memory system to provide application recovery as a service. If we can save the state of a process, we can transparently restart it whenever the power fails, exactly where it left off, with the user

none the wiser.

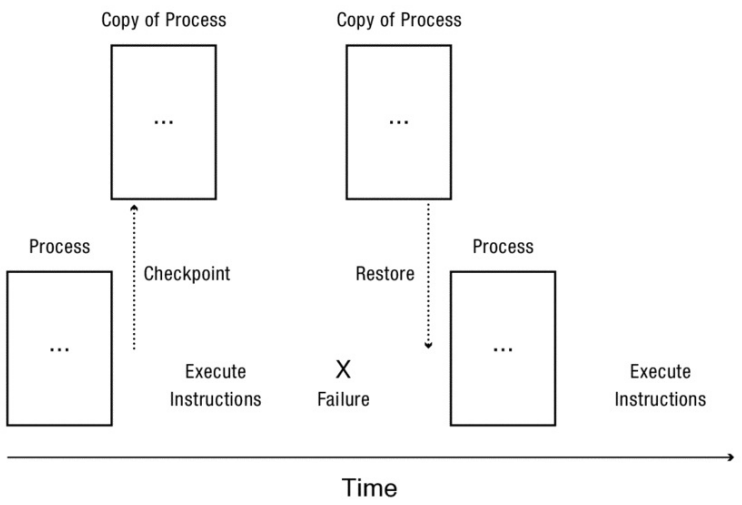

Figure 10.6: By checkpointing the state of a process, we can recover the saved state of the process after a failure by restoring the saved copy.

To make this work, we first need to suspend each thread executing in the process and save its state — the program counter, stack pointer, and registers to application memory. Once all threads are suspended, we can then store a copy of the contents of the application memory on disk. This is called a *checkpoint* or snapshot, illustrated in Figure 10.6. After a failure, we can resume the execution by restoring the contents of memory from the checkpoint and resuming each of the threads from from exactly the point we stopped them. This is called an application *restart*.

What would happen if we allow threads to continue to run while we are saving the contents of memory to disk? During the copy, we have a race condition: some pages could be saved before being modified by some thread, while others could be saved after being modified by that same thread. When we try to restart the application, its data structures could appear to be corrupted. The behavior of the program might be different from what would have happened if the failure had not occurred.

Fortunately, we can use address translation to minimize the amount of time we need to have the system stalled during a checkpoint. Instead of copying the contents of memory to disk, we can mark the application's pages as copy-on-write. At this point, we can restart the program's threads. As each page reaches disk, we can reset the protection on that page to read-write. When the program tries to modify a page before it reaches disk, the hardware will take an exception, and the kernel can make a copy of the page — one to be saved to disk and one to be used by the running program.

We can take checkpoints of the operating system itself in the same way. It is easiest to do this if the operating system is running in a virtual machine. The host can take a checkpoint by stopping the virtual machine, saving the processor state, and changing the page table protections (in the host page table) to read-only. The virtual machine is then safe to restart while the host writes the checkpoint to disk in the background.

Checkpoints and system calls

An implementation challenge for checkpoint/restart is to correctly handle any system calls that are in process. The state of a program is not only its user-level memory; it also includes the state of any threads that are executing in the kernel and any per-process state maintained by the kernel, such as its open file descriptors. While some operating systems have been designed to allow the kernel state of a process to be captured as part of the checkpoint, it is more common for checkpointing to be supported only at the virtual machine layer. A virtual machine has no state in the kernel except for the contents of its memory and processor registers. If we need to take a checkpoint while a trap handler is in progress, the handler can simply be restarted.

Process migration is the ability to take a running program on one system, stop its execution, and resume it on a different machine. Checkpoint and restart provide a basis for transparent process migration. For example, it is now common practice to checkpoint and migrate entire virtual machines inside a data center, as one way to balance load. If one system is hosting two web servers, each of which becomes heavily loaded, we can stop one and move it to a different machine so that each can get better performance.

10.3.2 Recoverable Virtual Memory

Taking a complete checkpoint of a process or a virtual machine is a heavyweight operation, and so it is only practical to do relatively rarely. We can use copy-on-write page protection to resume the process after starting the checkpoint, but completing the checkpoint will still take considerable time while we copy the contents of memory out to disk.

Can we provide an application the illusion of persistent memory, so that the contents of memory are restored to a point not long before the failure? The ability to do this is called *recoverable virtual memory*. An example where we might like recoverable virtual memory is in an email client; as you read, reply, and delete email, you do not want your work to be lost if the system crashes.

If we put efficiency aside, recoverable virtual memory is possible. First, we take a checkpoint so that some consistent version of the application's data is on disk. Next, we record an ordered sequence, or *log*, of every update that the application makes to memory. Once the log is written to disk we recover after a failure by reading the checkpoint and applying the changes from the log.

This is exactly how most text editors save their backups, to allow them to recover

uncommitted user edits after a machine or application failure. A text editor could repeatedly write an entire copy of the file to a backup, but this would be slow, particularly for a large file. Instead, a text editor will write a version of the file, and then it will append a sequence of every change the user makes to that version. To avoid having to separately write every typed character to disk, the editor will batch changes, e.g., all of the changes the user made in the past 100 milliseconds, and write those to disk as a unit. Even if the very latest batch has not been written to disk, the user can usually recover the state of the file at almost the instant immediately before the machine crash.

A downside of this algorithm for text editors is that it can cause information to be leaked without it being visible in the current version of the file. Text editors sometimes use this same method when the user hits "save" — just append any changes from the previous version, rather than writing a fresh copy of the entire file. This means that the old version of a file can potentially still be recovered from a file. So if you write a memo insulting your boss, and then edit it to tone it down, it is best to save a completely new version of your file before you send it off!

Will this method work for persistent memory? Keeping a log of every change to every memory location in the process would be too slow. We would need to trap on every store instruction and save the value to disk. In other words, we would run at the speed of the trap handler, rather than the speed of the processor.

However, we can come close. When we take a checkpoint, we mark all pages as read-only to ensure that the checkpoint includes a consistent snapshot of the state of the process's memory. Then we trap to the kernel on the first store instruction to each page, to allow the kernel to make a copy-on-write. The kernel resets the page to be read-write so that successive store instructions to the same page can go at full speed, but it can also record the page as having been modified.

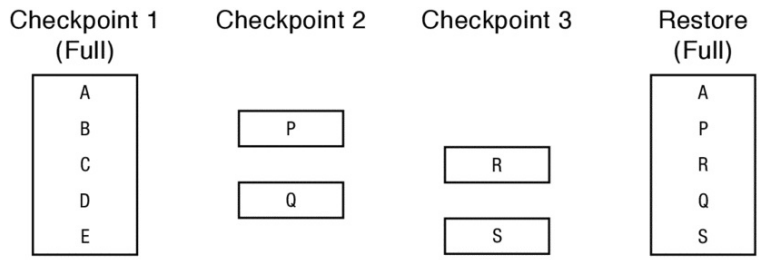

Figure 10.7: The operating system can recover the state of a memory segment after a crash by saving a sequence of incremental checkpoints.

We can take an *incremental checkpoint* by stopping the program and saving a copy of any pages that have been modified since the previous checkpoint. Once we change those pages back to read-only, we can restart the program, wait a bit, and take another incremental checkpoint. After a crash, we can recover the most recent memory by reading in the first checkpoint and then applying each of the incremental checkpoints in turn, as shown in Figure 10.7.

How much work we lose during a machine crash is a function of how quickly we can completely write an incremental checkpoint to disk. This is governed by the rate at which the application creates new data. To reduce the cost of an incremental checkpoint, applications needing recoverable virtual memory will designate a specific memory segment as persistent. After a crash, that memory will be restored to the latest incremental checkpoint, allowing the program to quickly resume its work.

10.3.3 Deterministic Debugging

A key to building reliable systems software is the ability to locate and fix problems when they do occur. Debugging a sequential program is comparatively easy: if you give it the same input, it will execute the same code in the same order, and produce the same output.

Debugging a concurrent program is much harder: the behavior of the program may change depending on the precise scheduling order chosen by the operating system. If the program is correct, the same output should be produced on the same input. If we are debugging a program, however, it is probably not correct. Instead, the precise behavior of the program may vary from run to run depending on which threads are scheduled first.

Debugging an operating system is even harder: not only does the operating system make widespread use of concurrency, but it is hard to tell sometimes what is its "input" and "output."

It turns out, however, that we can use a virtual machine abstraction to provide a repeatable debugging environment for an operating system, and we can in turn use that to provide a repeatable debugging environment for concurrent applications.

It is easiest to see this on a uniprocessor. The execution of an operating system running in a virtual machine can only be affected by three factors: its initial state, the input data provided by its I/O devices, and the precise timing of interrupts.

Because the host kernel mediates each of these for the virtual machine, it can record them and play them back during debugging. As long as the host exactly mimics what it did the first time, the behavior of the guest operating system will be the same and the behavior of all applications running on top of the guest operating system will be the same.

Replaying the input is easy, but how do we replay the precise timing of interrupts? Most modern computer architectures have a counter on the processor to measure the number of instructions executed. The host operating system can use this to measure how many instructions the guest operating system (or guest application) executed between the point where the host gave up control of the processor to the guest, and when control returned to the kernel due to an interrupt or trap.

To replay the precise timing of an asynchronous interrupt, the host kernel records the guest program counter and the instruction count at the point when the interrupt was delivered to the guest. On replay, the host kernel can set a trap on the page containing the program

counter where the next interrupt will be taken. Since the guest might visit the same program counter multiple times, the host kernel uses the instruction count to determine which visit corresponds to the one where the interrupt was delivered. (Some systems make this even easier, by allowing the kernel to request a trap whenever the instruction count reaches a certain value.)

Moreover, if we want to skip ahead to some known good intermediate point, we can take a checkpoint, and play forward the sequence of interrupts and input data from there. This is important as sometimes bugs in operating systems can take weeks to manifest themselves; if we needed to replay everything from boot the debugging process would be much more cumbersome.

Matters are more complex on a multicore system, as the precise behavior of both the guest operating system and the guest applications will depend on the precise ordering of instructions across the different processors. It is an ongoing area of research how best to provide deterministic execution in this setting. Provided that the program being debugged has no race conditions — that is, no access to shared memory outside of a critical section — then its behavior will be deterministic with one more piece of information. In addition to the initial state, inputs, and asynchronous interrupts, we also need to record which thread acquires each critical section in which order. If we replay the threads in that order and deliver interrupts precisely and provide the same device input, the behavior will be the same. Whether this is a practical solution is still an open question.

10.4 Security

Hardware or software address translation provides a basis for executing untrusted application code, to allow the operating system kernel to protect itself and other applications from malicious or buggy implementations.

A modern smartphone or tablet computer, however, has literally hundreds of thousands of applications that could be installed. Many or most are completely trustworthy, but others are specifically designed to steal or corrupt local data by exploiting weaknesses in the underlying operating system or the natural human tendency to trust technology. How is a user to know which is which? A similar situation exists for the web: even if most web sites are innocuous, some embed code that exploits known vulnerabilities in the browser defenses.

If we cannot limit our exposure to potentially malicious applications, what can we do? One important step is to keep your system software up to date. The malicious code authors recognize this: a recent survey showed that the most likely web sites to contain viruses are those targeted at the most novice users, e.g., screensavers and children's games.

In this section, we discuss whether there are additional ways to use virtual machines to limit the scope of malicious applications.

Suppose you want to download a new application, or visit a new web site. There is some chance it will work as advertised, and there is some chance it will contain a virus. Is there any way to limit the potential of the new software to exploit some unknown vulnerability in your operating system or browser?

One interesting approach is to clone your operating system into a new virtual machine, and run the application in the clone rather than on the native operating system. A virtual machine constructed for the purpose of executing suspect code is called a *virtual machine honeypot*. By using a virtual machine, if the code turns out to be malicious, we can delete the virtual machine and leave the underlying operating system as it was before we attempted to run the application.

Creating a virtual machine to execute a new application might seem extravagant. However, earlier in this chapter, we discussed various ways to make this more efficient: shadow page tables, memory compression, efficient checkpoint and restart, and copy-on-write. And of course, reinstalling your system after it has become infected with a virus is even slower!

Both researchers and vendors of commercial anti-virus software make extensive use of virtual machine honeypots to detect and understand viruses. For example, a frequent technique is to create an array of virtual machines, each with a different version of the operating system. By loading a potential virus into each one, and then simulating user behavior, we can more easily determine which versions of software are vulnerable and which are not.

A limitation is that we need to be able to tell if the browser or operating system running in the virtual machine honeypot has been corrupted. Often, viruses operate instantly, by attempting to install logging software or scanning the disk for sensitive information such as credit card numbers. There is nothing to keep the virus from lying in wait; this has become more common recently, particularly those designed for military or business espionage.

Another limitation is that the virus might be designed to infect both the guest operating system running in the clone and the host kernel implementing the virtual machine. (In the case of the web, the virus must infect the browser, the guest operating system, and the host.) As long as the system software is kept up to date, the system is vulnerable only if the virus is able to exploit some unknown weakness in the guest operating system and a separate unknown weakness in the host implementation of the virtual machine. This provides *defense in depth*, improving security through multiple layers of protection.

10.5 User-Level Memory Management

With the increasing sophistication of applications and their runtime systems, most widely used operating systems have introduced hooks for applications to manage their own memory. While the details of the interface differs from system to system, the hooks preserve the role of the kernel in allocating resources between processes and in preventing access to privileged memory. Once a page frame has been assigned to a process, however, the kernel can leave it up to the process to determine what to do with that resource.

Operating systems can provide applications the flexibility to decide:

- **Where to get missing pages.** As we noted in the previous chapter, a modern memory hierarchy is deep and complex: local disk, local non-volatile memory, remote memory inside a data center, or remote disk. By giving applications control, the

kernel can keep its own memory hierarchy simple and local, while still allowing sophisticated applications to take advantage of network resources when they are available, even when those resources are on machines running completely different operating systems.

- **Which pages can be accessed.** Many applications such as browsers and databases need to set up their own application-level sandboxes for executing untrusted code. Today this is done with a combination of hardware and software techniques, as we described in Chapter 8. Finer-grained control over page fault handling allows more sophisticated models for managing sharing between regions of untrusted code.
- **Which pages should be evicted.** Often, an application will have better information than the operating system over which pages it will reference in the near future.

Many applications can adapt the size of their working set to the resources provided by the kernel but they will have worse performance whenever there is a mismatch.

- **Garbage collected programs.** Consider a program that does its own garbage collection. When it starts up, it allocates a block of memory in its virtual address space to serve as the heap. Periodically, the program scans through the heap to compact its data structures, freeing up room for additional data structures. This causes all pages to appear to be recently used, confounding the kernel's memory manager. By contrast, the application knows that the best page to replace is one that was recently cleaned of application data.

  It is equally confounding to the application. How does the garbage collector know how much memory it should allocate for the heap? Ideally, the garbage collector should use exactly as much memory as the kernel is able to provide, and no more. If the runtime heap is too small, the program must garbage collect, even though more page frames available. If the heap is too large, the kernel will page parts of the heap to disk instead of asking the application to pay the lower overhead of compacting its memory.
- **Databases.** Databases and other data processing systems often manipulate huge data sets that must be streamed from disk into memory. As we noted in Chapter 9, algorithms for large data sets will be more efficient if they are customized to the amount of available physical memory. If the operating system evicts a page that the database expects to be in memory, these algorithms will run much more slowly.
- **Virtual machines.** A similar issue arises with virtual machines. The guest operating system running inside of a virtual machine thinks it has a set of physical page frames, which it can assign to the virtual pages of applications running in the virtual machine. In reality, however, the page frames in the guest operating system are virtual and can be paged to disk by the host operating system. If the host operating system could tell the guest operating system when it needed to steal a page frame (or donate a page frame), then the guest would know exactly how many page frames were available to be allocated to its applications.

In each of these cases, the performance of a resource manager can be compromised if it runs on top of a virtualized, rather than a physical, resource. What is needed is for the operating system kernel to communicate how much memory is assigned to a process or virtual machine so that the application to do its own memory management. As processes start and complete, the amount of available physical memory will change, and therefore the assignment to each application will change.

To handle these needs, most operating systems provide some level of application control over memory. Two models have emerged:

- **Pinned pages.** A simple and widely available model is to allow applications to *pin* virtual memory pages to physical page frames, preventing those pages from being evicted unless absolutely necessary. Once pinned, the application can manage its memory however it sees fit, for example, by explicitly shuffling data back and forth to disk.

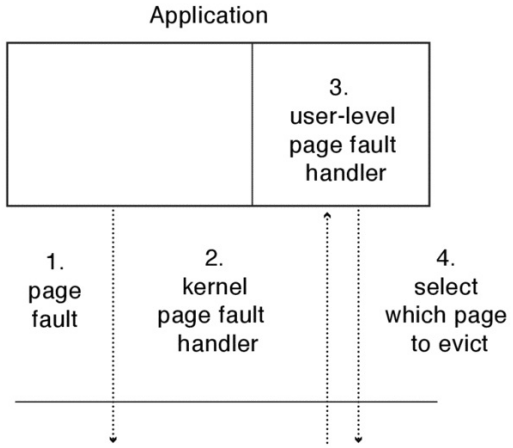

Figure 10.8: The operation of a user-level page handler. On a page fault, the hardware traps to the kernel; if the fault is for a segment with a user-level pager, the kernel passes the fault to the user-level handler to manage. The user-level handler is pinned in memory to avoid recursive faults.

- **User-level pagers.** A more general solution is for applications to specify a *user-level page handler* for a memory segment. On a page fault or protection violation, the kernel trap handler is invoked. Instead of handling the fault itself, the kernel passes control to user-level handler, as in a UNIX signal handler. The user-level handler can then decide how to manage the trap: where to fetch the missing page, what action to take if the application was sandbox, and which page to replace. To avoid infinite recursion, the user-level page handler must itself be stored in pinned memory.

10.6 Summary and Future Directions

In this chapter, we have argued that address translation provides a powerful tool for operating systems to provide a set of advanced services to applications to improve system performance, reliability, and security. Services such as checkpointing, recoverable memory, deterministic debugging, and honeypots are now widely supported at the virtual machine layer, and we believe that they will come to be standard in most operating systems as well.

Moving forward, it is clear that the demands on the memory management system for advanced services will increase. Not only are memory hierarchies becoming increasingly complex, but the diversity of services provided by the memory management system has added even more complexity.

Operating systems often go through cycles of gradually increasing complexity followed by rapid shifts back towards simplicity. The recent commercial interest in virtual machines may yield a shift back towards simpler memory management, by reducing the need for the kernel to provide every service that any application might need. Processor architectures now directly support user-level page tables. This potentially opens up an entire realm for more sophisticated runtime systems, for those applications that are themselves miniature operating systems, and a concurrent simplification of the kernel. With the right operating system support, applications will be able to set up and manage their own page tables directly, implement their own user-level process abstractions, and provide their own transparent checkpointing and recovery on memory segments.

Exercises

1. This question concerns the operation of shadow page tables for virtual machines, where a guest process is running on top of a guest operating system on top of a host operating system. The architecture uses paged segmentation, with a 32-bit virtual address divided into fields as follows:

   | 4 bit segment number | 12 bit page number | 16 bit offset |

   The guest operating system creates and manages segment and page tables to map the guest virtual addresses to guest physical memory. These tables are as follows (all values in hexadecimal):

Segment Table	Page Table A	Page Table B
0 Page Table A	0 0002	0 0001
1 Page Table B	1 0006	1 0004
x (rest invalid)	2 0000	2 0003
	3 0005	x (rest invalid)
	x (rest invalid)	

   The host operating system creates and manages segment and page tables to map the guest physical memory to host physical memory. These tables are as follows:

Segment Table	Page Table K
0 Page Table K	0 BEEF
x (rest invalid)	1 F000
	2 CAFE
	3 3333
	4 (invalid)
	5 BA11
	6 DEAD
	7 5555
	x (rest invalid)

   a. Find the host physical address corresponding to each of the following guest virtual addresses. Answer “invalid guest virtual address” if the guest virtual address is invalid; answer “invalid guest physical address if the guest virtual address maps to a valid guest physical page frame, but the guest physical page has an invalid virtual address.

      i. 00000000
      ii. 20021111
      iii. 10012222

iv. 00023333
v. 10024444

b. Using the information in the tables above, fill in the contents of the shadow segment and page tables for direct execution of the guest process.

c. Assuming that the guest physical memory is contiguous, list three reasons why the host page table might have an invalid entry for a guest physical page frame, with valid entries on either side.

2. Suppose we doing incremental checkpoints on a system with 4 KB pages and a disk capable of transferring data at 10 MB/s.

a. What is the maximum rate of updates to new pages if every modified page is sent in its entirety to disk on every checkpoint and we require that each checkpoint reach disk before we start the next checkpoint?

b. Suppose that most pages saved during an incremental checkpoint are only partially modified. Describe how you would design a system to save only the modified portions of each page as part of the checkpoint.