2 The Kernel Abstraction

protection

A central role of operating systems is *protection* — the isolation of potentially misbehaving applications and users so that they do not corrupt other applications or the operating system itself. Protection is essential to achieving several of the operating systems goals noted in the previous chapter:

Why does an operating system have a kernel?

- **Reliability.** Protection prevents bugs in one program from causing crashes in other programs or in the operating system. To the user, a system crash appears to be the operating system's fault, even if the root cause of the problem is some unexpected behavior by an application or user. Thus, for high system reliability, an operating system must bullet proof itself to operate correctly regardless of what an application or user might do.

- **Security.** Some users or applications on a system may be less than completely trustworthy; therefore, the operating system must limit the scope of what they can do. Without protection, a malicious user might surreptitiously change application files or even the operating system itself, leaving the user none the wiser. For example, if a malicious application can write directly to the disk, it could modify the file containing the operating system's code; the next time the system starts, the modified operating system would boot instead, installing spyware and disabling virus protection. For security, an operating system must prevent untrusted code from modifying system state.

- **Privacy.** On a multi-user system, each user must be limited to only the data that she is permitted to access. Without protection provided by the operating system, any user or application running on a system could access anyone's data, without the knowledge or approval of the data's

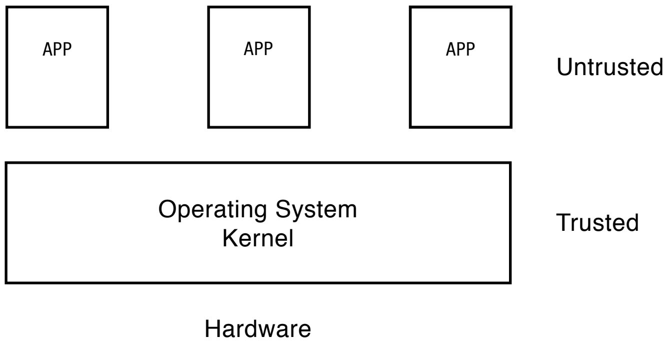

Figure 2.1: User-level and kernel-level operation. The operating system kernel is trusted to arbitrate between untrusted applications and users.

owner. For example, hackers often use popular applications — such as games or screen savers — as a way to gain access to personal email, telephone numbers, and credit card data stored on a smartphone or laptop. For privacy, an operating system must prevent untrusted code from accessing unauthorized data.

- **Fair resource allocation.** Protection is also needed for effective resource allocation. Without protection, an application could gather any amount of processing time, memory, or disk space that it wants. On a single-user system, a buggy application could prevent other applications from running or make them run so slowly that they appear to stall. On a multi-user system, one user could grab all of the system's resources. Thus, for efficiency and fairness, an operating system must be able to limit the amount of resources assigned to each application or user.

operating system kernel

Implementing protection is the job of the *operating system kernel*. The kernel, the lowest level of software running on the system, has full access to all of the machine hardware. The kernel is necessarily *trusted* to do anything with the hardware. Everything else — that is, the untrusted software running on the system — is run in a restricted environment with less than complete access to the full power of the hardware. Figure 2.1 illustrates this difference between kernel-level and user-level execution.

Do applications need to implement protection?

In turn, applications themselves often need to safely execute untrusted third party code. An example is a web browser executing embedded Javascript to draw a web page. Without protection, a script with an embedded virus can take control of the browser, making users think they are interacting directly with the web when in fact their web passwords are being forwarded to an attacker.

This design pattern — extensible applications running third-party scripts — occurs in many different domains. Applications become more powerful and widely used if third party developers and users can customize them, but doing so raises the issue of how to protect the application itself from rogue extensions. This chapter focuses on how the operating system protects the kernel from untrusted applications, but the principles also apply at the application level.

process

A *process* is the execution of an application program with restricted rights; the process is the abstraction for protected execution provided by the operating system kernel. A process needs permission from the operating system kernel before accessing the memory of any other process, before reading or writing to the disk, before changing hardware settings, and so forth. In other words, the operating system kernel mediates and checks each process's access to hardware. This chapter explains the process concept and how the kernel implements process isolation.

Does protection compromise performance?

A key consideration is the need to provide protection while still running application code at high speed. The operating system kernel runs directly on the processor with unlimited rights. The kernel can perform any operation available on the hardware. What about applications? They need to run on the processor with all potentially dangerous operations disabled. To make this work, hardware needs to provide a bit of assistance, which we will describe shortly. Throughout the book, there are similar examples of how small amounts of carefully designed hardware can help make it much easier for the operating system to provide what users want.

Of course, both the operating system kernel and application processes running with restricted rights are in fact sharing the same machine — the same processor, the same memory, and the same disk. When reading this chapter, keep these two perspectives in mind: when we are running the operating system kernel, it can do anything; when we are running an application process on behalf of a user, the process's behavior is restricted.

Thus, a processor running an operating system is somewhat akin to someone with a split personality. When running the operating system kernel, the processor is like a warden in charge on an insane asylum with complete access to everything. At other times, the processor runs application code in a process — the processor becomes an inmate, wearing a straightjacket locked in a padded cell by the warden, protected from harming anyone else. Of course, it is the same processor in both cases, sometimes completely trustworthy and at other times completely untrusted.

Chapter roadmap: Protection raises several important questions that we will answer in the rest of the chapter:

- **The Process Abstraction.** What is a process and how does it differ from a program? (Section 2.1)
- **Dual-Mode Operation.** What hardware enables the operating system to efficiently implement the process abstraction? (Section 2.2)
- **Types of Mode Transfer.** What causes the processor to switch control from a user-level program to the kernel? (Section 2.3)

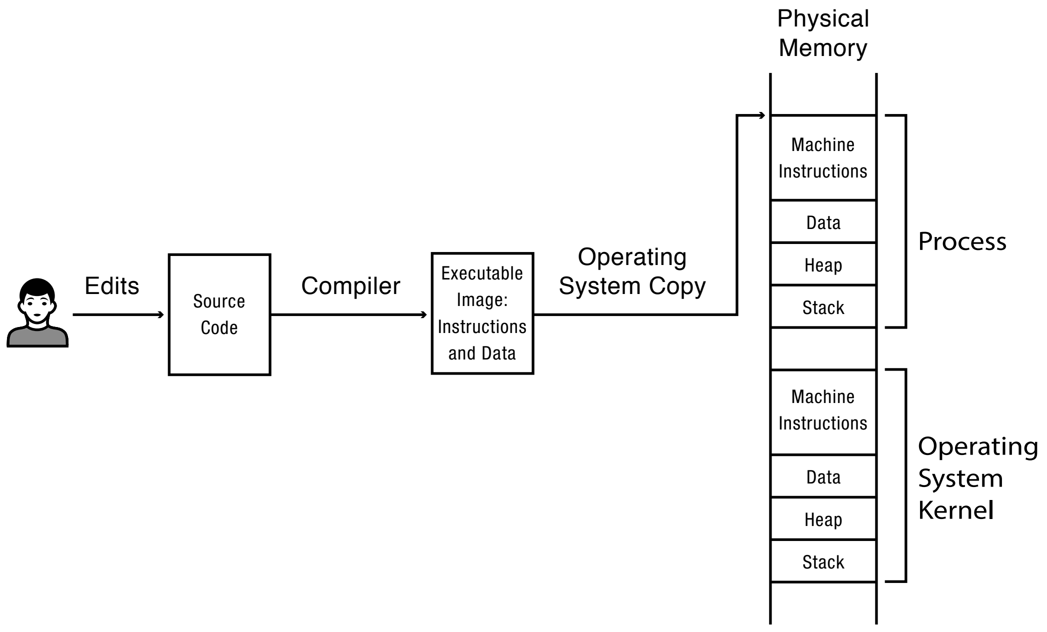

Figure 2.2: A user edits, compiles, and runs a user program. Other programs can also be stored in physical memory, including the operating system itself.

- **Implementing Safe Mode Transfer.** How do we safely switch between user level and the kernel? (Section 2.4)
- **Putting It All Together: x86 Mode Transfer.** What happens on an x86 mode switch? (Section 2.5)
- **Implementing Secure System Calls.** How do library code and the kernel work together to implement protected procedure calls from the application into the kernel? (Section 2.6)
- **Starting a New Process.** How does the operating system kernel start a new process? (Section 2.7)
- **Implementing Upcalls.** How does the operating system kernel deliver an asynchronous event to a user process? (Section 2.8)
- **Case Study: Booting an OS Kernel.** What steps are needed to start running an operating system kernel, to the point where it can create a process? (Section 2.9)
- **Case Study: Virtual Machines.** Can an operating system run inside a process? (Section 2.10)

2.1 The Process Abstraction

What is a process?

In the model you are likely familiar with, illustrated in Figure 2.2, a programmer types code in some high-level language. A compiler converts that code into a sequence of machine instructions and stores those instructions in a file, called the program's *executable image*. The compiler also defines any static data the program needs, along with its initial values, and includes them in the executable image.

executable image

To run the program, the operating system copies the instructions and data from the executable image into physical memory. The operating system sets aside a memory region, the *execution stack*, to hold the state of local variables during procedure calls. The operating system also sets aside a memory region, called the *heap*, for any dynamically allocated data structures the program might need. Of course, to copy the program into memory, the operating system itself must already be loaded into memory, with its own stack and heap.

execution stack

heap

Ignoring protection, once a program is loaded into memory, the operating system can start it running by setting the stack pointer and jumping to the program's first instruction. The compiler itself is just another program: the operating system starts the compiler by copying its executable image into memory and jumping to its first instruction.

To run multiple copies of the same program, the operating system can make multiple copies of the program's instructions, static data, heap, and stack in memory. As we describe in Chapter 8, most operating systems reuse memory wherever possible: they store only a single copy of a program's instructions when multiple copies of the program are executed at the same time. Even so, a separate copy of the program's data, heap, and stack are needed. For now, we will keep things simple and assume the operating system makes a separate copy of the entire program for each process.

What is the difference between a process and a program?

Thus, a process is an *instance* of a program, in much the same way that an object is an instance of a class in object-oriented programming. Each program can have zero, one or more processes executing it. For each instance of a program, there is a process with its own copy of the program in memory.

process control block

The operating system keeps track of the various processes on the computer using a data structure called the *process control block*, or PCB. The PCB stores all the information the operating system needs about a particular process: where it is stored in memory, where its executable image resides on disk, which user asked it to execute, what privileges the process has, and so forth.

A process combines execution and protection.

Earlier, we defined a process as an instance of a program executing with restricted rights. Each of these roles — execution and protection — is important enough to merit several chapters.

This chapter focuses on protection, and so we limit our discussion to simple processes, each with one program counter, code, data, heap, and stack.

Some programs consist of multiple concurrent activities, or threads. A web browser, for example, might need to receive user input at the same time it is drawing the screen or receiving network input. Each of these separate

Processes, lightweight processes, and threads

The word “process", like many terms in computer science, has evolved over time. The evolution of words can sometimes trip up the unwary — systems built at different times will use the same word in significantly different ways.

A “process" was originally coined to mean what is now called a “thread" — a logical sequence of instructions that executes either operating system or application code. The concept of a process was developed as a way of simplifying the correct construction of early operating systems that provided no protection between application programs.

Organizing the operating system as a cooperating set of processes proved immensely successful, and soon almost every new operating system was built this way, including systems that also provided protection against malicious or buggy user programs. At the time, almost all user programs were simple, single-threaded programs with only one program counter and one stack, so there was no confusion. A process was needed to run a program, that is, a single sequential execution stream with a protection boundary.

As parallel computers became more popular, though, we once again needed a word for a logical sequence of instructions. A multiprocessor program can have multiple instruction sequences running in parallel, each with its own program counter, but all cooperating within a single protection boundary. For a time, these were called “lightweight processes" (each a sequence of instructions cooperating inside a protection boundary), but eventually the word “thread” became more widely used.

This leads to the current naming convention used in almost all modern operating systems: a process executes a program, consisting of one or more threads running inside a protection boundary.

activities has its own program counter and stack but operates on the same code and data as the other threads. The operating system runs multiple threads in a process, in much the same way that it runs multiple processes in physical memory. We generalize on the process abstraction to allow multiple activities in the same protection domain in Chapter 4.

2.2 Dual-Mode Operation

What hardware enables the OS to efficiently implement the process abstraction?

Once a program is loaded into memory and the operating system starts the process, the processor fetches each instruction in turn, then decodes and executes it. Some instructions compute values, say, by multiplying two registers and putting the result into another register. Some instructions read or write locations in memory. Still other instructions, like branches or procedure calls, change the program counter and thus determine the next instruction to execute.

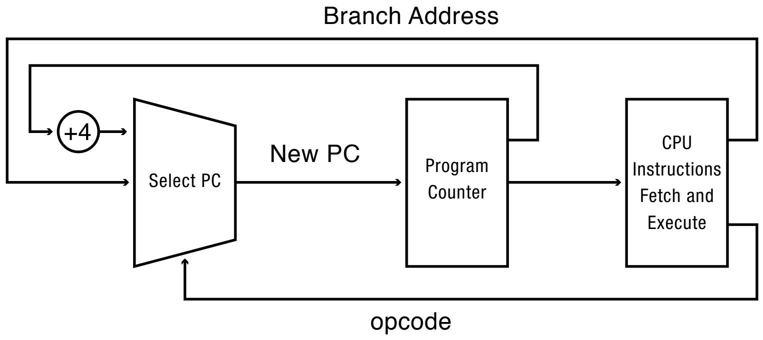

Figure 2.3: The basic operation of a CPU. Opcode, short for operation code, is the decoded instruction to be executed, e.g., branch, memory load, or arithmetic operation.

Figure 2.3 illustrates the basic operation of a processor.

How does the operating system kernel prevent a process from harming other processes or the operating system itself? After all, when multiple programs are loaded into memory at the same time, what prevents a process from overwriting another process's data structures, or even overwriting the operating system image stored on disk?

If we step back from any consideration of performance, a very simple, safe, and entirely hypothetical approach would be to have the operating system kernel simulate, step by step, every instruction in every user process. Instead of the processor directly executing instructions, a software interpreter would fetch, decode, and execute each user program instruction in turn. Before executing each instruction, the interpreter could check if the process had permission to do the operation in question: is it referencing part of its own memory, or someone else's? Is it trying to branch into someone else's code? Is it directly accessing the disk, or is it using the correct routines in the operating system to do so? The interpreter could allow all legal operations while halting any application that overstepped its bounds.

Now suppose we want to speed up our hypothetical simulator. Most instructions are perfectly safe, such as adding two registers together and storing the result in a third register. Can we modify the processor in some way to allow safe instructions to execute directly on the hardware?

dual-mode operation

user mode

kernel mode

To accomplish this, we implement the same checks as in our hypothetical interpreter, but in hardware rather than software. This is called *dual-mode operation*, represented by a single bit in the processor status register that signifies the current mode of the processor. In *user mode*, the processor checks each instruction before executing it to verify that it is permitted to be performed by that process. (We describe the specific checks next.) In *kernel mode*, the operating system executes with protection checks turned off.

Figure 2.4 shows the operation of a dual-mode processor; the program counter and the mode bit together control the processor's operation. In turn, the mode bit is modified by some instructions, just as the program counter is

The kernel vs. the rest of the operating system

The operating system kernel is a crucial piece of an operating system, but it is only a portion of the overall operating system. In most modern operating systems, a portion of the operating system runs in user mode as a library linked into each application. An example is library code that manages an application's menu buttons. To encourage a common user interface across applications, most operating systems provide a library of user interface widgets. Applications can write their own user interface routines, but most developers choose to reuse the routines provided by the operating system. This code could run in the kernel but does not need to do so. If the application crashes, it will not matter if that application's menu buttons stop working. The library code (but not the operating system kernel) *shares fate* with the rest of the application: a problem with one has the same effect as a problem with the other.

Likewise, parts of the operating system can run in their own user-level processes. A window manager is one example. The window manager directs mouse actions and keyboard input that occurs inside a window to the correct application, and the manager also ensures that each application modifies only that application's portion of the screen, and not the operating system's menu bar or any other application's window. Without this restriction, a malicious application could potentially take control of the machine. For example, a virus could present a login prompt that looked identical to the system login, potentially inducing users to disclose their passwords to the attacker.

Why not include the entire operating system — the library code and any user-level processes — in the kernel itself? While that might seem more logical, one reason is that it is often easier to debug user-level code than kernel code. The kernel can use low-level hardware to implement debugging support for breakpoints and for single stepping through application code; to single step the kernel requires an even lower-level debugger running underneath the kernel. The difficulty of debugging operating system kernels was the original motivation behind the development of virtual machines.

More importantly, the kernel must be trusted, as it has full control over the hardware. Any error in the kernel can corrupt the disk, the memory of some unrelated application, or simply crash the system. By separating out code that does not need to be in the kernel, the operating system can become more reliable — a bug in the window system is bad enough, but it would be even worse if it could corrupt the disk. This illustrates the *principle of least privilege*, that security and reliability are enhanced if each part of the system has exactly the privileges it needs to do its job, and no more.

modified by some instructions.

What hardware is needed to let the operating system kernel protect applications and users from one another, yet also let user code run directly on the processor? At a minimum, the hardware must support three things:

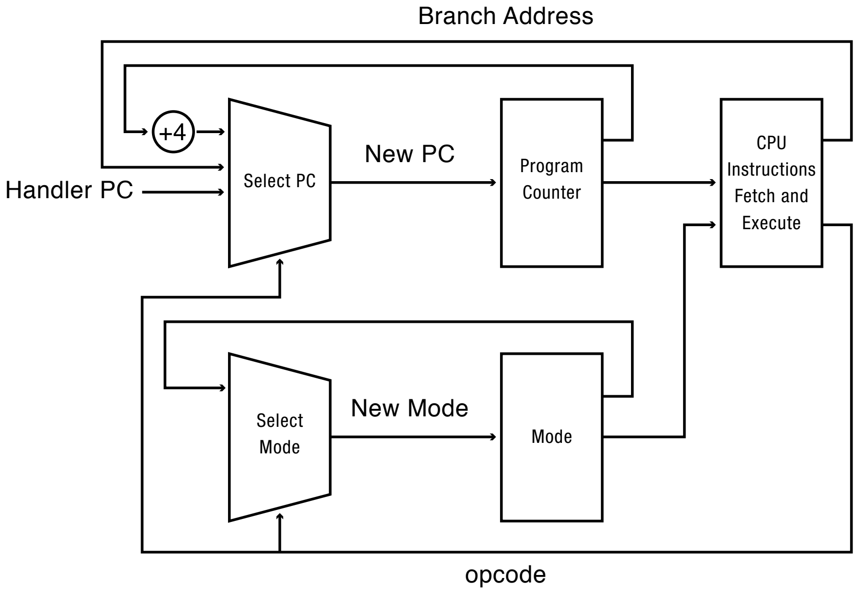

Figure 2.4: The operation of a CPU with kernel and user modes.

- **Privileged Instructions.** All potentially unsafe instructions are prohibited when executing in user mode. (Section 2.2.1)
- **Memory Protection.** All memory accesses outside of a process's valid memory region are prohibited when executing in user mode. (Section 2.2.2)
- **Timer Interrupts.** Regardless of what the process does, the kernel must have a way to periodically regain control from the current process. (Section 2.2.3)

In addition, the hardware must also provide a way to safely transfer control from user mode to kernel mode and back. As the mechanisms to do this are relatively involved, we defer the discussion of that topic to Sections 2.3 and 2.4.

2.2.1 Privileged Instructions

What instructions can't a process execute?

Process isolation is possible only if there is a way to limit programs running in user mode from directly changing their privilege level. We discuss in Section 2.3 that processes can indirectly change their privilege level by executing a special instruction, called a *system call*, to transfer control into the kernel at a fixed location defined by the operating system. Other than transferring control into the operating system kernel (that is, in effect, becoming the kernel) at these fixed locations, an application process cannot change its privilege level.

The processor status register and privilege levels

Conceptually, the kernel/user mode is a one-bit register. When set to 1, the processor is in kernel mode and can do anything. When set to 0, the processor is in user mode and is restricted. On most processors, the kernel/user mode is stored in the *processor status register*. This register contains flags that control the processor's operation and is typically not directly accessible to application code. Rather, flags are set or reset as a by-product of executing instructions. For example, the hardware automatically saves the status register to memory when an interrupt occurs because otherwise the interrupt handler code would inadvertently overwrite its contents.

The kernel/user mode bit is one flag in the processor status register, set whenever the kernel is entered and reset whenever the kernel switches back to user mode. Other flags include *condition codes*, set as a side effect of arithmetic operations, to allow a more compact encoding of conditional branch instructions. Still other flags can specify whether the processor is executing with 16-bit, 32-bit, or 64-bit addresses. The specific contents of the processor status register are processor architecture dependent.

Some processor architectures, including the Intel x86, support more than two privilege levels in the processor status register (the x86 supports four privilege levels). The original reason for this was to allow the operating system kernel to be separated into two layers: (i) a core with unlimited access to the machine, and (ii) an outer layer restricted from certain operations, but with more power than completely unprivileged application code. This way, bugs in one part of the operating system kernel might not crash the entire system. However, to our knowledge, neither MacOS, Windows, nor Linux make use of this feature.

A potential future use for multiple privilege levels would be to simplify running an operating system as an application, or virtual machine, on top of another operating system. Applications running on top of the virtual machine operating system would run at user level; the virtual machine would run at some intermediate level; and the true kernel would run in kernel mode. Of course, with only four levels, this does not work for a virtual machine running on a virtual machine running on a virtual machine. For our discussion, we assume the simpler and more universal case of two levels of hardware protection.

Other instructions are also limited to use by kernel code. The application cannot be allowed to change the set of memory locations it can access; we discuss in Section 2.2.2 how limiting an application to accessing only its own memory is essential to preventing it from either intentionally, or accidentally, corrupting or misusing the data or code from other applications or the operating system. Further, applications cannot disable processor interrupts, as we will explain in Section 2.2.3.

Instructions available in kernel mode, but not in user mode, are called

privileged instruction

privileged instructions. The operating system kernel must be able to execute these instructions to do its work — it needs to change privilege levels, adjust memory access, and disable and enable interrupts. If these instructions were available to applications, then a rogue application would in effect have the power of the operating system kernel.

Thus, while application programs can use only a subset of the full instruction set, the operating system executes in kernel mode with the full power of the hardware.

What happens if an application attempts to access restricted memory or attempts to change its privilege level? Such actions cause a *processor exception*. Unlike taking an exception in a programming language where the language runtime and user code handles the exception, a processor exception causes the processor to transfer control to an exception handler in the operating system kernel. Usually, the kernel simply halts the process after a privilege violation.

EXAMPLE What could happen if applications were allowed to jump into kernel mode at any location in the kernel?

ANSWER Although it might seem that the worst that could happen would be that the operating system would crash (bad enough!), this might also allow a malicious application to gain access to privileged data or possibly control over the machine. The operating system kernel implements a set of privileged services on behalf of applications. Typically, one of the first steps in a kernel routine is to verify whether the user has permission to perform the operation; for example, the file system checks if the user has permission to read a file before returning the data. If an application can jump past the permission check, it could potentially evade the kernel's security limits. □

2.2.2 Memory Protection

How does the hardware limit a program to only accessing its own memory?

To run an application process, both the operating system and the application must be resident in memory at the same time. The application must be in memory in order to execute, while the operating system must be there to start the program and to handle any interrupts, processor exceptions, or system calls that happen while the program runs. Further, other application processes may also be stored in memory; for example, you may read email, download songs, Skype, instant message, and browse the web at the same time.

To make memory sharing safe, the operating system must be able to configure the hardware so that each application process can read and write only its own memory, not the memory of the operating system or any other application. Otherwise, an application could modify the operating system kernel's code or data to gain control over the system. For example, the application could change the login program to give the attacker full system administrator privileges. While it might seem that read-only access to memory is harmless, recall that operating systems need to provide both security and privacy. Kernel data structures — such as the file system buffer — may contain private user

MS/DOS and memory protection

As an illustration of the power of memory protection, MS/DOS was an early Microsoft operating system that did not provide it. Instead, user programs could read and modify any memory location in the system, including operating system data structures. While this was seen as acceptable for a personal computer that was only used by a single person at a time, there were a number of downsides. One obvious problem was system reliability: application bugs frequently crashed the operating system or corrupted other applications. The lack of memory protection also made the system more vulnerable to computer viruses.

Over time, some applications took advantage of the ability to change operating system data structures, for example, to change certain control parameters or to directly manipulate the frame buffer for controlling the display. As a result, changing the operating system became quite difficult; either the new version could not run the old applications, limiting its appeal, or it needed to leave these data structures in precisely the same place as they were in the old version. In other words, memory protection is not only useful for reliability and security; it also helps to enforce a well-defined interface between applications and the operating system kernel to aid future evolvability and portability.

data. Likewise, user passwords may be stored in kernel memory while they are being verified.

How does the operating system prevent a user program from accessing parts of physical memory? We discuss a wide variety of different approaches in Chapter 8, but early computers pioneered a simple mechanism to provide protection. We describe it now to illustrate the general principle.

base and bound memory protection

With this approach, a processor has two extra registers, called *base and bound*. The base specifies the start of the process's memory region in physical memory, while the bound gives its endpoint (Figure 2.5). These registers can be changed only by privileged instructions, that is, by the operating system executing in kernel mode. User-level code cannot change their values.

Every time the processor fetches an instruction, it checks the address of the program counter to see if it is between the base and the bound registers. If so, the instruction fetch is allowed to proceed; otherwise, the hardware raises an exception, suspending the program and transferring control back to the operating system kernel. Although it might seem extravagant to perform two extra comparisons for each instruction, memory protection is worth the cost. In fact, we will discuss much more sophisticated and "extravagant" memory protection schemes in Chapter 8.

Likewise, for instructions that read or write data to memory, the processor checks each memory reference against the base and bound registers, generat-

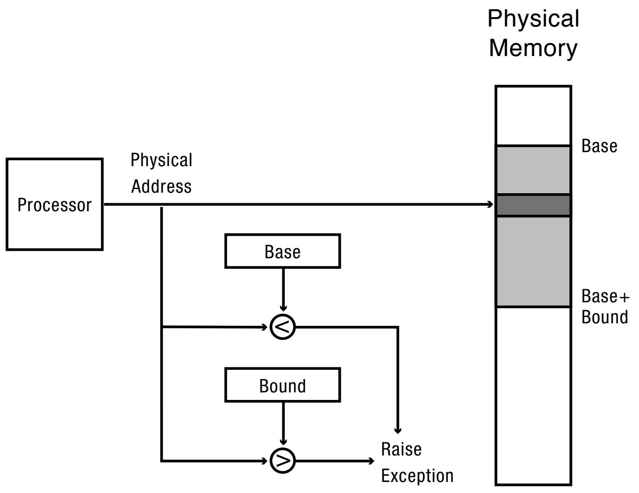

Figure 2.5: Base and bound memory protection using physical addresses. Every code and data address generated by the program is first checked to verify that its address lies within the memory region of the process.

ing a processor exception if the boundaries are violated. Complex instructions, such as a block copy instruction, must check every location touched by the instruction, to ensure that the application does not inadvertently or maliciously read or write to a buffer that starts in its own region but that extends into the kernel's region. Otherwise, applications could read or overwrite key parts of the operating system code or data and thereby gain control of the system.

The operating system kernel executes without the base and bound registers, allowing it to access any memory on the system — the kernel's memory or the memory of any application process running on the system. Because applications touch only their own memory, the kernel must explicitly copy any input or output into or out of the application's memory region. For example, a simple program might print "hello world". The kernel must copy the string out of the application's memory region into the screen buffer.

Memory allocation with base and bound registers is simple, analogous to heap memory allocation. When a program starts up, the kernel finds a free block of contiguous physical memory with enough room to store the entire program, its data, heap and execution stack. If the free block is larger than needed, the kernel returns the remainder to the heap for allocation to some other process.

Using physically addressed base and bound registers can provide protection, but this does not provide some important features:

- **Expandable heap and stack.** With a single pair of base and bound registers per process, the amount of memory allocated to a program is fixed when the program starts. Although the operating system can change the bound, most programs have two (or more) memory regions that need

Memory-mapped devices

On most computers, the operating system controls input/output devices — such as the disk, network, or keyboard — by reading and writing to special memory locations. Each device monitors the memory bus for the address assigned to it, and when it sees its address, the device triggers the desired I/O operation.

The operating system can use memory protection to prevent user-level processes from accessing these special memory locations. Thus, memory protection has the added advantage of limiting direct access to input/output devices by user code. By limiting each process to just its own memory locations, the kernel prevents processes from directly reading or writing to the disk controller or other devices. In this way, a buggy or malicious application cannot modify the operating system's image stored on disk, and a user cannot gain access to another user's files without first going through the operating system to check file permissions.

to independently expand depending on program behavior. The execution stack holds procedure local variables and grows with the depth of the procedure call graph; the heap holds dynamically allocated objects. Most systems today grow the heap and the stack from opposite sides of program memory; this is difficult to accommodate with a pair of base and bound registers.

- **Memory sharing.** Base and bound registers do not allow memory to be shared between different processes, as would be useful for sharing code between multiple processes running the same program or using the same library.

- **Physical memory addresses.** When a program is compiled and linked, the addresses of its procedures and global variables are set relative to the beginning of the executable file, that is, starting at zero. With the mechanism we have just described using base and bound registers, each program is loaded into physical memory at runtime and must use those physical memory addresses. Since a program may be loaded at different locations depending on what other programs are running at the same time, the kernel must change every instruction and data location that refers to a global address, each time the program is loaded into memory.

- **Memory fragmentation.** Once a program starts, it is nearly impossible to relocate it. The program might store pointers in registers or on the execution stack (for example, the program counter to use when returning from a procedure), and these pointers need to be changed to move the program to a different region of physical memory. Over time, as

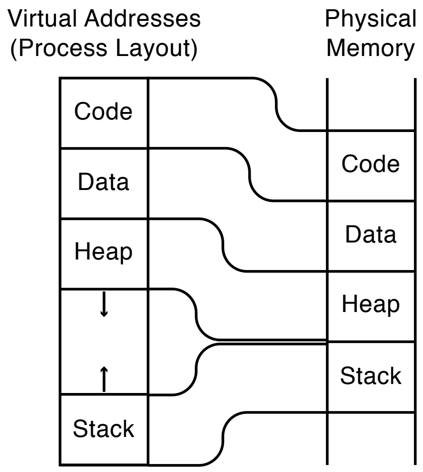

Figure 2.6: Virtual addresses allow the stack and heap regions of a process to grow independently. To grow the heap, the operating system can move the heap in physical memory without changing the heap's virtual address.

applications start and finish at irregular times, memory will become increasingly fragmented. Potentially, memory fragmentation may reach a point where there is not enough contiguous space to start a new process, despite sufficient free memory in aggregate.

virtual address

For these reasons, most modern processors introduce a level of indirection, called *virtual addresses*. With virtual addresses, every process's memory starts at the same place, e.g., zero. Each process thinks that it has the entire machine to itself, although obviously that is not the case in reality. The hardware translates these virtual addresses to physical memory locations. A simple algorithm would be to add the base register to every virtual address so that the process can use virtual addresses starting from zero.

In practice, modern systems use much more complex algorithms to translate between virtual and physical addresses. The layer of indirection provided by virtual addresses gives operating systems enormous flexibility to efficiently manage physical memory. For example, many systems with virtual addresses allocate physical memory in fixed-sized, rather than variable-sized, chunks to reduce fragmentation.

Virtual addresses can also let the heap and the stack start at separate ends of the virtual address space so they can grow according to program need (Figure 2.6). If either the stack or heap grows beyond its initially allocated region, the operating system can move it to a different larger region in physical memory but leave it at the same virtual address. The expansion is completely transparent to the user process. We discuss virtual addresses in more depth in Chapter 8.

```
int staticVar = 0;      // a static variable
main() {
    staticVar += 1;

    // sleep causes the program to wait for x seconds
    sleep(10);
    printf ("Address: %x; Value: %d\n", &staticVar, staticVar);
}

Produces:
    Address: 5328; Value: 1
```

Figure 2.7: A simple C program whose output illustrates the difference between execution in physical memory versus virtual memory. When multiple copies of this program run simultaneously, the output does not change.

How can we tell if a machine uses virtual addresses?

Figure 2.7 lists a simple test program to verify that a computer supports virtual addresses. The program has a single static variable; it updates the value of the variable, waits for a few seconds, and then prints the location of the variable and its value.

With virtual addresses, if multiple copies of this program run simultaneously, each copy of the program will print exactly the same result. This would be impossible if each copy were directly addressing physical memory locations. In other words, each instance of the program appears to run in its own complete copy of memory: when it stores a value to a memory location, it alone sees its changes to that location. Other processes change their own copies of the memory location. In this way, a process cannot alter any other process's memory, because it has no way to reference the other process's memory; only the kernel can read or write the memory of a process other than itself.

This is very much akin to a set of television shows, each occupying their own universe, even though they all appear on the same television. Events in one show do not (normally) affect the plot lines of other shows. Sitcom characters are blissfully unaware that Jack Bauer has just saved the world from nuclear Armageddon. Of course, just as television shows can from time to time share characters, processes can also communicate if the kernel allows it. We will discuss how this happens in Chapter 3.

EXAMPLE Suppose we have a "perfect" object-oriented language and compiler in which only an object's methods can access the data inside the object. If the operating system runs only programs written in that language, would it still need hardware memory address protection?

ANSWER In theory, no, but in practice, yes. The compiler would be responsible for ensuring that no application program read or modified data outside of its own objects. This requires, for example, the language runtime to do garbage collection: once an object is released back to the heap (and possibly reused by some other application), the application cannot continue to hold a pointer to the object.

Address randomization

Computer viruses often work by attacking hidden vulnerabilities in operating system and server code. For example, if the operating system developer forgets to check the length of a user string before copying it into a buffer, the copy can overwrite the data stored immediately after the buffer. If the buffer is stored on the stack, this might allow a malicious user to overwrite the return program counter from the procedure; the attacker can then cause the server to jump to an arbitrary point (for example, into code embedded in the string). These attacks are easier to mount when a program uses the same locations for the same variables each time it runs.

Most operating systems, such as Linux, MacOS, and Windows, combat viruses by randomizing (within a small range) the virtual addresses that a program uses each time it runs. This is called *address space layout randomization*. A common technique is to pick a slightly different start address for the heap and stack for each execution. Thus, in Figure 2.7, if instead we printed the address of a procedure local variable, the address might change from run to run, even though the value of the variable would still be 1.

Some systems have begun to randomize procedure and static variable locations each, as well as the offset between adjacent procedure records on the stack to make it harder to force the system to jump to the attacker's code. Nevertheless, each process appears to have its own copy of memory, disjoint from all other processes.

In practice, this approach means that system security depends on the correct operation of the compiler in addition to the operating system kernel. Any bug in the compiler or language runtime becomes a possible way for an attacker to gain control of the machine. Many languages have extensive runtime libraries to simplify the task of writing programs in that language; often these libraries are written for performance in a language closer to the hardware, such as C. Any bug in a library routine also becomes a possible means for an attacker to gain control.

Although it may seem redundant, many systems use both language-level protection and process-level protection. For example, Google's Chrome web browser creates a separate process (e.g., one per browser tab) to interpret the HTML, Javascript, or Java on a web page. This way, a malicious attacker must compromise both the language runtime as well as the operating system process boundary to gain control of the client machine. □

MacOS and preemptive scheduling

Until 2002, Apple's MacOS lacked the ability to force a process to yield the processor back to the kernel. Instead, all application programmers were told to design their systems to periodically call into the operating system to check if there was other work to be done. The operating system would then save the state of the original process, switch control to another application, and return only when it again became the original process's turn. This had a drawback: if a process failed to yield, e.g., because it had a bug and entered an infinite loop, the operating system kernel had no recourse. The user needed to reboot the machine to return control to the operating system. This happened frequently enough that it was given its own name: the "spinning cursor of death."

2.2.3 Timer Interrupts

How does the kernel regain control from a runaway process?

Process isolation also requires hardware to provide a way for the operating system kernel to periodically regain control of the processor. When the operating system starts a user-level program, the process is free to execute any user-level (non-privileged) instructions it chooses, call any function in the process's memory region, load or store any value to its memory, and so forth. To the user program, it appears to have complete control of the hardware within the limits of its memory region.

However, this too is only an illusion. If the application enters an infinite loop, or if the user simply becomes impatient and wants the system to stop the application, then the operating system must be able to regain control. Of course, the operating system needs to execute instructions to decide if it should stop the application, but if the application controls the processor, the operating system by definition is not running on that processor.

The operating system also needs to regain control of the processor in normal operation. Suppose you are listening to music on your computer, downloading a file, and typing at the same time. To smoothly play the music, and to respond in a timely way to user input, the operating system must be able to regain control to switch to a new task.

hardware timer

Almost all computer systems include a device called a *hardware timer*, which can be set to interrupt the processor after a specified delay (either in time or after some number of instructions have been executed). Each timer interrupts only one processor, so a multiprocessor will usually have a separate timer for each CPU. The operating system might set each timer to expire every few milliseconds; human reaction time is a few hundred of milliseconds. Resetting the timer is a privileged operation, accessible only within the kernel, so that the user-level process cannot inadvertently or maliciously disable the timer.

When the timer interrupt occurs, the hardware transfers control from the user process to the kernel running in kernel mode. Other hardware interrupts, such as to signal the processor that an I/O device has completed its work, likewise transfer control from the user process to the kernel. A timer or other interrupt does not imply that the program has an error; in most cases, after resetting the timer, the operating system resumes execution of the process, setting the mode, program counter and registers back to the values they had immediately before the interrupt occurred. We discuss the hardware and kernel mechanisms for implementing interrupts in Section 2.4.

EXAMPLE How does the kernel know if an application is in an infinite loop?

ANSWER It doesn't. Typically, the operating system will terminate a process only when requested by the user or system administrator, e.g., because the application has become non-responsive to user input. The operating system needs to be able to regain control to be able to ask the user if she wants to shut down a particular process. □

2.3 Types of Mode Transfer

Once the kernel has placed a user process in a carefully constructed sandbox, the next question is how to safely transition from executing a user process to executing the kernel, and vice versa. These transitions are not rare events. A high-performance web server, for example, might switch between user mode and kernel mode thousands of times per second. Thus, the mechanism must be both fast and safe, leaving no room for malicious or buggy programs to corrupt the kernel, either intentionally or inadvertently.

2.3.1 User to Kernel Mode

What causes execution to switch into the kernel?

We first focus on transitions from user mode to kernel mode; as we will see, transitioning in the other direction works by "undo"-ing the transition from the user process into the kernel.

There are three reasons for the kernel to take control from a user process: interrupts, processor exceptions, and system calls. Interrupts occur asynchronously — that is, they are triggered by an external event and can cause a transfer to kernel mode after any user-mode instruction.

trap

Processor exceptions and system calls are synchronous events triggered by process execution. We use the term *trap* to refer to any synchronous transfer of control from user mode to the kernel; some systems use the term more generically for any transfer of control from a less privileged to a more privileged level.

interrupt

How does an I/O device get the processor's attention?

- **Interrupts.** An *interrupt* is an asynchronous signal to the processor that some external event has occurred that may require its attention. As the processor executes instructions, it checks for whether an interrupt has