Operating Systems

Principles & Practice

SECOND EDITION

Thomas Anderson
Michael Dahlin

Part I

Kernels and Processes

Good fences make good neighbors.

17th century proverb

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

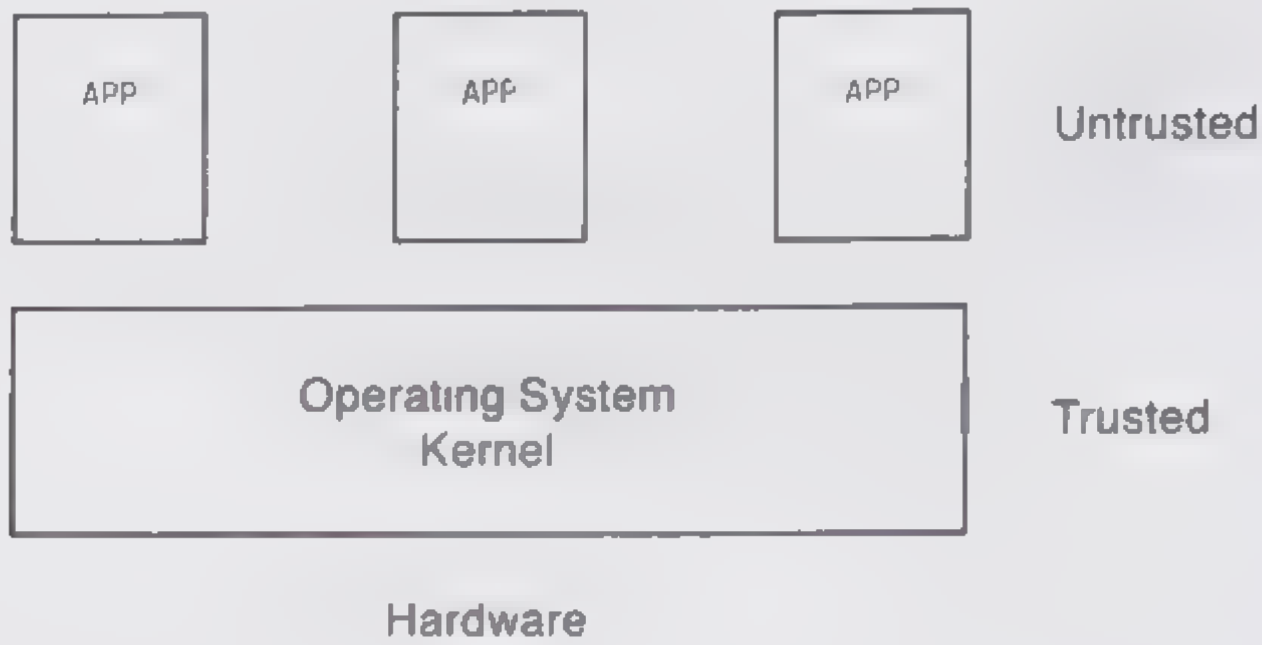

Figure 2.1: User-level and kernel-level operation. The operating system kernel is trusted to arbitrate between untrusted applications and users.

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. 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)

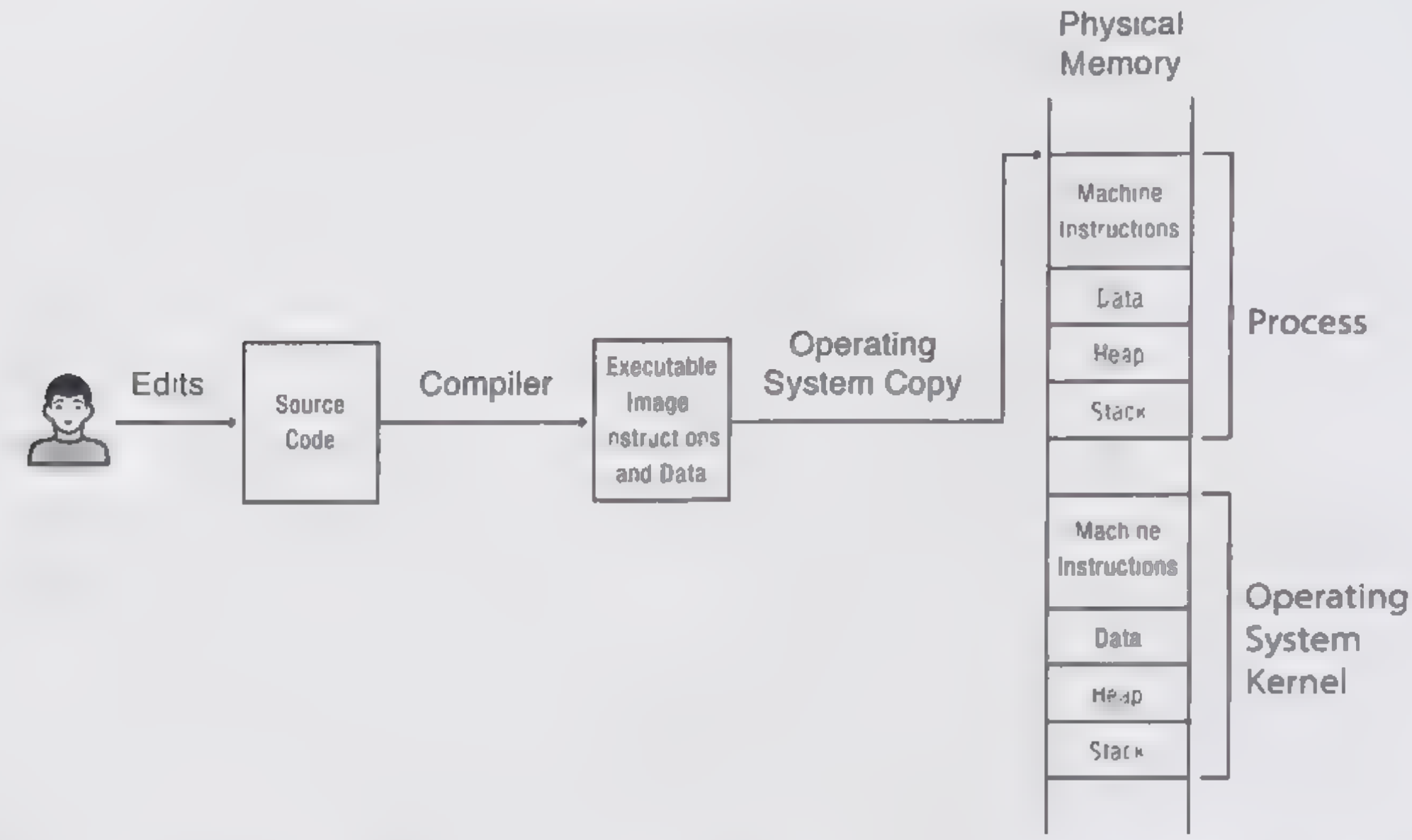

Figure 2.2: A user edits, compiles, and runs a user program. Other programs can also be stored in physical memory, including the operating system itself.

- **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,

executable image

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.

To run the program, the operating system copies the instructions and data from the executable image into physical memory. The operating system sets

execution stack

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,

heap

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.

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.

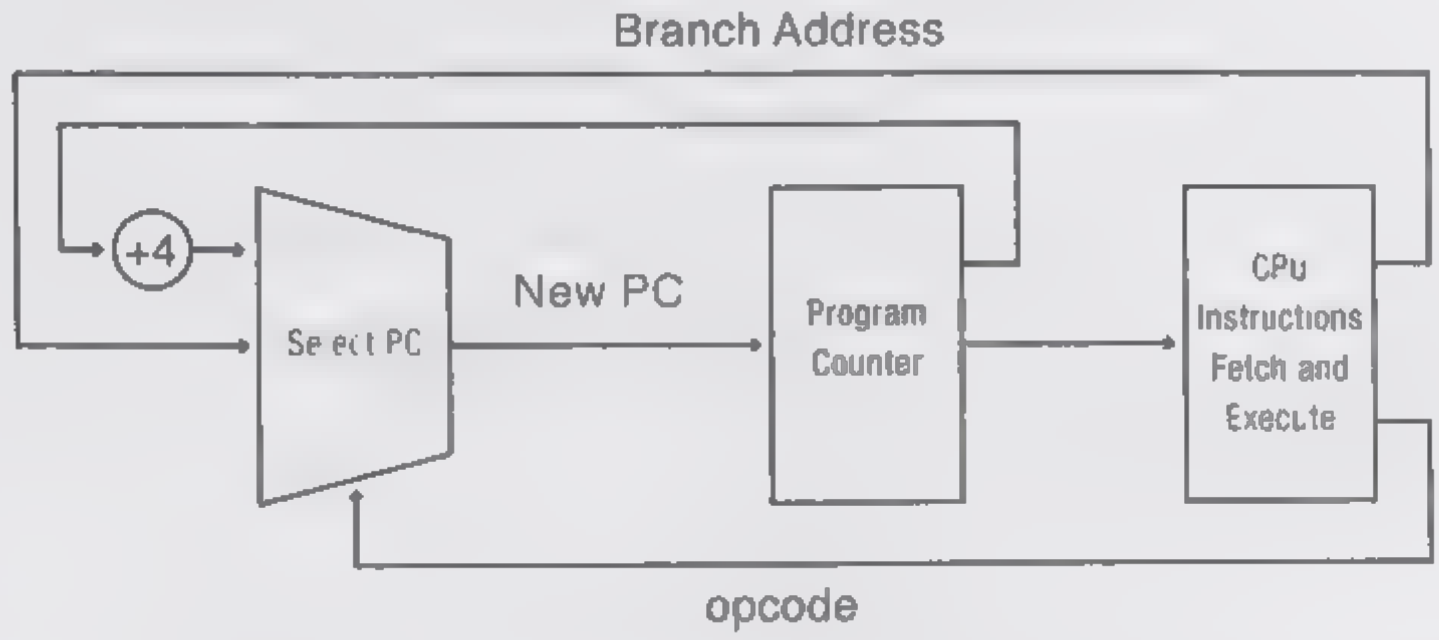

Figure 2.3: The basic operation of a CPU. Opcode, short for operation code, is the decoded instruction to be executed, e.g., branch, memory load, or arithmetic operation

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:

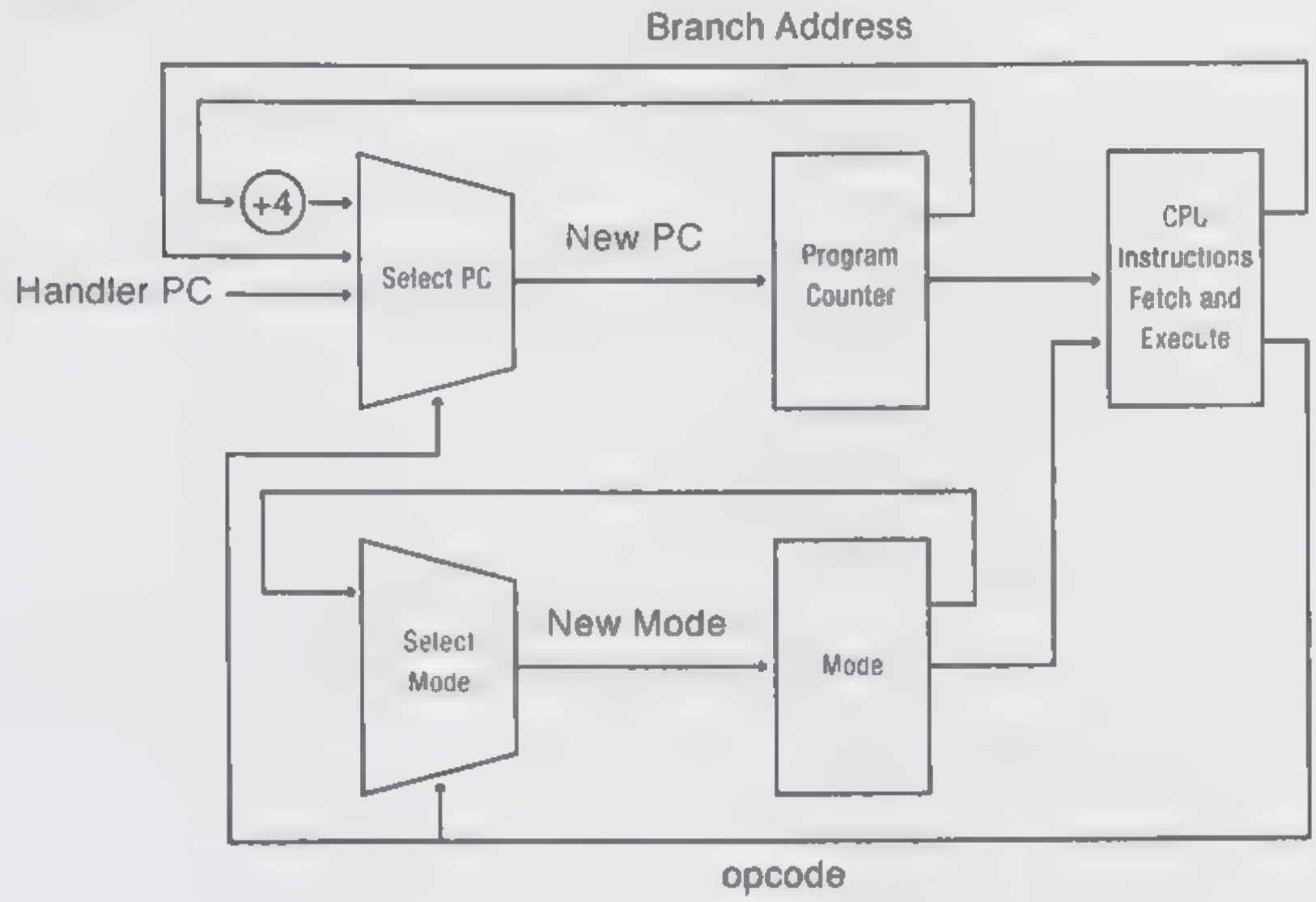

Figure 2.4: The operation of a CPU with kernel and user modes.

- **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 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-

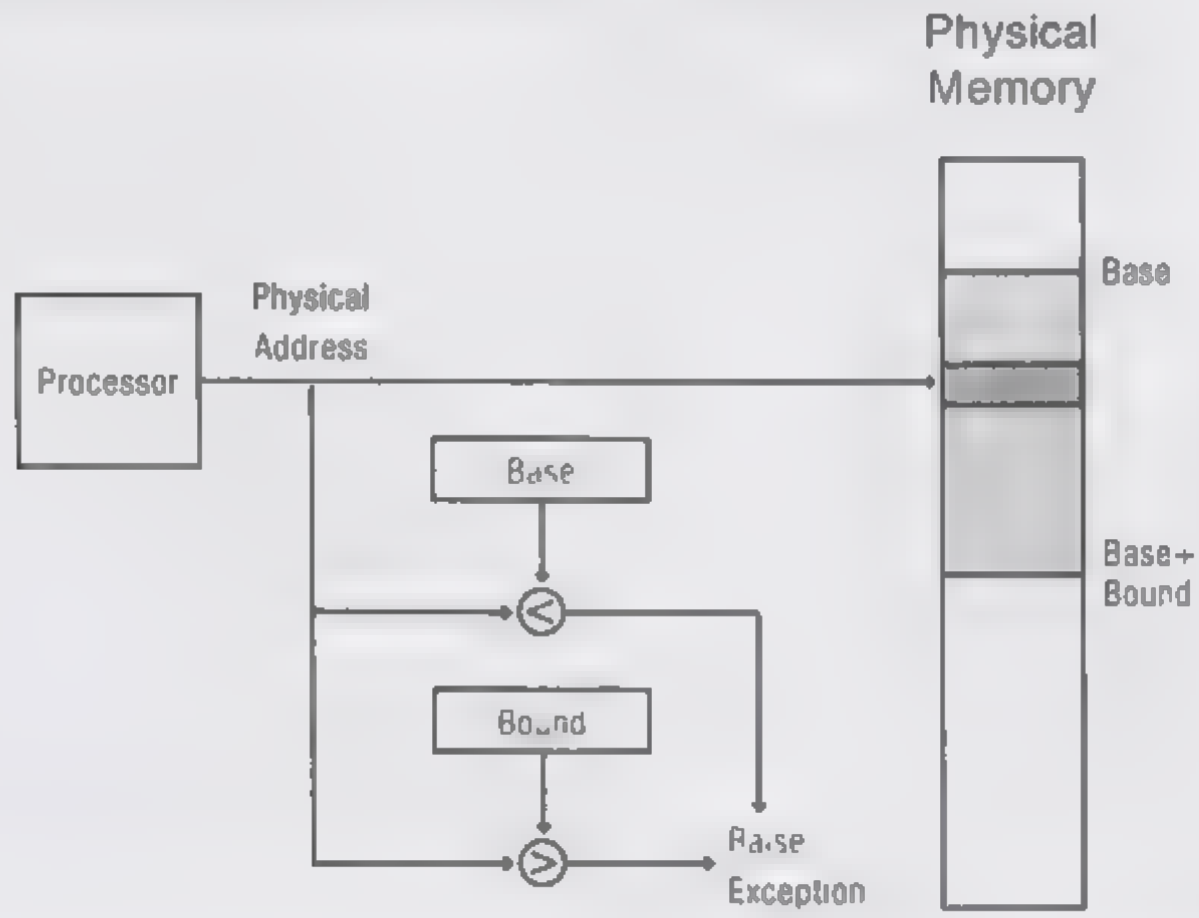

Figure 2.5: Base and bound memory protection using physical addresses. Every code and data address generated by the program is first checked to verify that its address lies within the memory region of the process.

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; thus 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

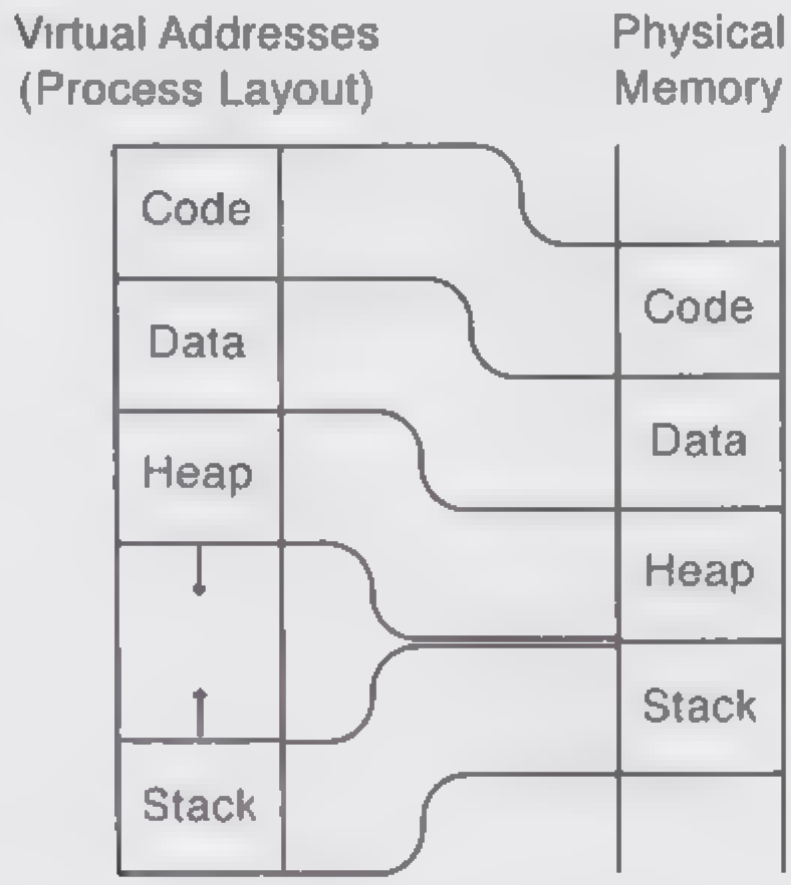

Figure 2.6: Virtual addresses allow the stack and heap regions of a process to grow independently. To grow the heap, the operating system can move the heap in physical memory without changing the heap's virtual address.

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 "undoing" 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.

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.

trap

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

arrived. If so, it completes or stalls any instructions that are in progress. Instead of fetching the next instruction, the processor hardware saves the current execution state and starts executing at a specially designated interrupt handler in the kernel. On a multiprocessor, an interrupt is taken on only one of the processors; the others continue to execute as if nothing happened.

Each different type of interrupt requires its own handler. For timer interrupts, the handler checks if the current process is being responsive to user input to detect if the process has gone into an infinite loop. The timer handler can also switch execution to a different process to ensure that each process gets a turn. If no change is needed, the timer handler resumes execution at the interrupted instruction, transparently to the user process.

Interrupts are also used to inform the kernel of the completion of I/O requests. For example, mouse device hardware triggers an interrupt every time the user moves or clicks on the mouse. The kernel, in turn, notifies the appropriate user process — the one the user was "mousing" across. Virtually every I/O device — the Ethernet, WiFi, hard disk, thumb drive, keyboard, mouse — generates an interrupt whenever some input arrives for the processor and whenever a request completes.

polling

An alternative to interrupts is *polling*: the kernel loops, checking each input/output device to see if an event has occurred that requires handling. Needless to say, if the kernel is polling, it is not available to run user-level code.

Interprocessor interrupts are another source of interrupts. A processor can send an interrupt to any other processor. The kernel uses these interrupts to coordinate actions across the multiprocessor, for example, when a parallel program exits, the kernel sends interrupts to stop the program from continuing to run on any other processor.

processor exception

What happens when a program executes an illegal instruction?

- **Processor exceptions.** A *processor exception* is a hardware event caused by user program behavior that causes a transfer of control to the kernel. As with an interrupt, the hardware finishes all previous instructions, saves the current execution state, and starts running at a specially designated exception handler in the kernel. For example, a processor exception occurs whenever a process attempts to perform a privileged instruction or accesses memory outside of its own memory region. Other processor exceptions occur when a process divides an integer by zero, accesses a word of memory with a non-aligned address, attempts to write to read-only memory, and so forth. In these cases, the operating system simply halts the process and returns an error code to the user. On a multiprocessor, the exception only stops execution on the processor triggering the exception, the kernel then needs to send interprocessor interrupts to stop execution of the parallel program on other processors.

Buffer descriptors and high-performance I/O

In early computer systems, the key to good performance was to keep the processor busy; particularly for servers, the key to good performance today is keeping I/O devices, such as the network and disk device, busy Neither Internet nor disk bandwidth has kept pace with the rapid improvement in processor performance over the past four decades, leaving them relatively more important than the CPU to system performance.

A simple, but inefficient, approach to designing the operating system software to manage an I/O device is to allow only one I/O operation to the device at any one time In this case, interrupt handling can be a limiting factor to performance When the device completes a request, it raises an interrupt, causing the device interrupt handler to run. The handler can then issue the next pending request to the hardware. In the meantime, while the processor is handling the interrupt, the device is idle.

For higher performance, the operating system sets up a circular queue of requests for each device to handle (A network interface will have two queues: one for incoming packets and one for outgoing packets.) Each entry in the queue, called a *buffer descriptor*, specifies one I/O operation: the requested operation (e g., disk read or write) and the location of the buffer to contain the data. The device hardware reads the buffer descriptor to determine what operations to perform. Provided the queue of buffer descriptors is full, the device can start working on the next operation while the operating system handles with the previous one.

Buffer descriptors are stored in memory, accessed by the device using DMA (direct memory access) An implication is that each logical I/O operation can involve several DMA requests: one to download the buffer descriptor from memory into the device, then to copy the data in or out, and then to store the success/failure of the operation back into buffer descriptor.

Processor exceptions are also caused by more benign program events. For example, to set a breakpoint in a program, the kernel replaces the machine instruction in memory with a special instruction that invokes a trap. When the program reaches that point in its execution, the hardware switches into the kernel. The kernel restores the old instruction and transfers control to the debugger. The debugger can then examine the program's variables, set a new breakpoint, and resume the program at the instruction causing the exception.

- **System calls.** User processes can also transition into the operating system kernel voluntarily to request that the kernel perform an operation on the user's behalf. A *system call* is any procedure provided by the kernel that can be called from user level. Most processors implement system calls with a special trap or syscall instruction. However, a special instruction is

system call

How does a user program ask the kernel to do something?

Processor exceptions and virtualization

Processor exceptions are a particularly powerful tool for virtualization the emulation of hardware that does not actually exist. As one example, it is common for different versions of a processor architecture family to support some parts of the instruction set and not others, such as when an inexpensive, low power processor does not support floating point operations. At some cost in performance, the operating system can use processor exceptions to make the difference completely transparent to the user process. When the program issues a floating point instruction, an exception is raised, trapping into the operating system kernel. Instead of halting the process, the operating system can *emulate* the missing instruction, and, on completion, return to the user process at the instruction immediately after the one that caused the exception. In this way, the same program binary can run on different versions of the processor.

More generally, processor exceptions are used to transparently emulate a virtual machine. When a guest operating system is running as a user-level process on top of an operating system, it will attempt to execute privileged instructions as if it were running on physical hardware. These instructions will cause processor exceptions, trapping into the host operating system kernel. To maintain the illusion of physical hardware, the host kernel then performs the requested instruction of behalf of the user-level virtual machine and restarts the guest operating system at the instruction immediately following the one that caused the exception.

As a final example, processor exceptions are a key building block for memory management. With most types of virtual addressing, the processor can be set up to take an exception whenever it reads or writes inside a particular virtual address range. This allows the kernel to treat memory as *virtual* — a portion of the program memory may be stored on disk instead on in physical memory. When the program touches a missing address, the operating system exception handler fills in the data from disk before resuming the program. In this way, the operating system can execute programs that require more memory than can fit on the machine at the same time.

---

not strictly required; on some systems, a process triggers a system call by executing an instruction with a specific invalid opcode.

As with an interrupt or a processor exception, the trap instruction changes the processor mode from user to kernel and starts executing in the kernel at a pre-defined handler. To protect the kernel from misbehaving user programs, it is essential that the hardware transfers control on a system call to a pre-defined address — user processes *cannot* be allowed to jump to arbitrary places in the kernel.

Operating systems can provide any number of system calls. Examples

include system calls to establish a connection to a web server, to send or receive packets over the network, to create or delete files, to read or write data into files, and to create a new user process. To the user program, these are called like normal procedures, with parameters and return values. The caller needs to be concerned only with the interface; it does not need to know that the routine is actually being implemented by the kernel. The kernel handles the details of checking and copying arguments, performing the operation, and copying return values back into the process's memory. When the kernel completes the system call, it resumes user-level execution at the instruction immediately after the trap.

2.3.2 Kernel to User Mode

What causes execution to switch back to user mode?

Just as there are several different types of transitions from user to kernel mode, there are several types of transitions from kernel to user mode:

- **New process.** To start a new process, the kernel copies the program into memory, sets the program counter to the first instruction of the process, sets the stack pointer to the base of the user stack, and switches to user mode.
- **Resume after an interrupt, processor exception, or system call.** When the kernel finishes handling the request, it resumes execution of the interrupted process by restoring its program counter (in the case of a system call, the instruction after the trap), restoring its registers, and changing the mode back to user level.
- **Switch to a different process.** In some cases, such as on a timer interrupt, the kernel switches to a different process than the one that had been running before the interrupt. Since the kernel will eventually resume the old process, the kernel needs to save the process state—its program counter, registers, and so forth—in the process's control block. The kernel can then resume a different process by loading its state—its program counter, registers, and so forth—from the process's control block into the processor and then switching to user mode.
- **User-level upcall.** Many operating systems provide user programs with the ability to receive asynchronous notification of events. The mechanism, which we describe in Section 2.8, is similar to kernel interrupt handling, except at user level.

2.4 Implementing Safe Mode Transfer

How do we safely switch between user and kernel modes?

Whether transitioning from user to kernel mode or in the opposite direction, care must be taken to ensure that a buggy or malicious user program cannot

corrupt the kernel. Although the basic idea is simple, the low level implementation can be a bit complex: the processor must save its state and switch what it is doing, *while* executing instructions that might alter the state that it is in the process of saving. This is akin to rebuilding a car's transmission while it barrels down the road at 60 mph.

The context switch code must be carefully crafted, and it relies on hardware support. To avoid confusion and reduce the possibility of error, most operating systems have a common sequence of instructions both for entering the kernel whether due to interrupts, processor exceptions or system calls and for returning to user level, again regardless of the cause.

At a minimum, this common sequence must provide:

- **Limited entry into the kernel.** To transfer control to the operating system kernel, the hardware must ensure that the entry point into the kernel is one set up by the kernel. User programs cannot be allowed to jump to arbitrary locations in the kernel. For example, the kernel code for handling the read file system call first checks whether the user program has permission to do so. If not, the kernel should return an error. Without limited entry points into the kernel, a malicious program could jump immediately after the code to perform the check, allowing the program to access to anyone's file.

- **Atomic changes to processor state.** In user mode, the program counter and stack point to memory locations in the user process' memory. Memory protection prevents the user process from accessing any memory outside of its region. In kernel mode, the program counter and stack point to memory locations in the kernel, memory protection is changed to allow the kernel to access both its own data and that of the user process. Transitioning between the two is atomic—the mode, program counter, stack, and memory protection are all changed at the same time.

- **Transparent, restartable execution.** An event may interrupt a user-level process at any point, between any instruction and the next one. For example, the processor could have calculated a memory address, loaded it into a register, and be about to store a value to that address. The operating system must be able to restore the state of the user program exactly as it was before the interrupt occurred. To the user process, an interrupt is invisible, except that the program temporarily slows down. A "hello world" program is not written to understand interrupts, but an interrupt might still occur while the program is running.

  On an interrupt, the processor saves its current state to memory, temporarily defers further events, changes to kernel mode, and then jumps to the interrupt or exception handler. When the handler finishes, the steps are reversed: the processor state is restored from its saved location, with the interrupted program none the wiser.

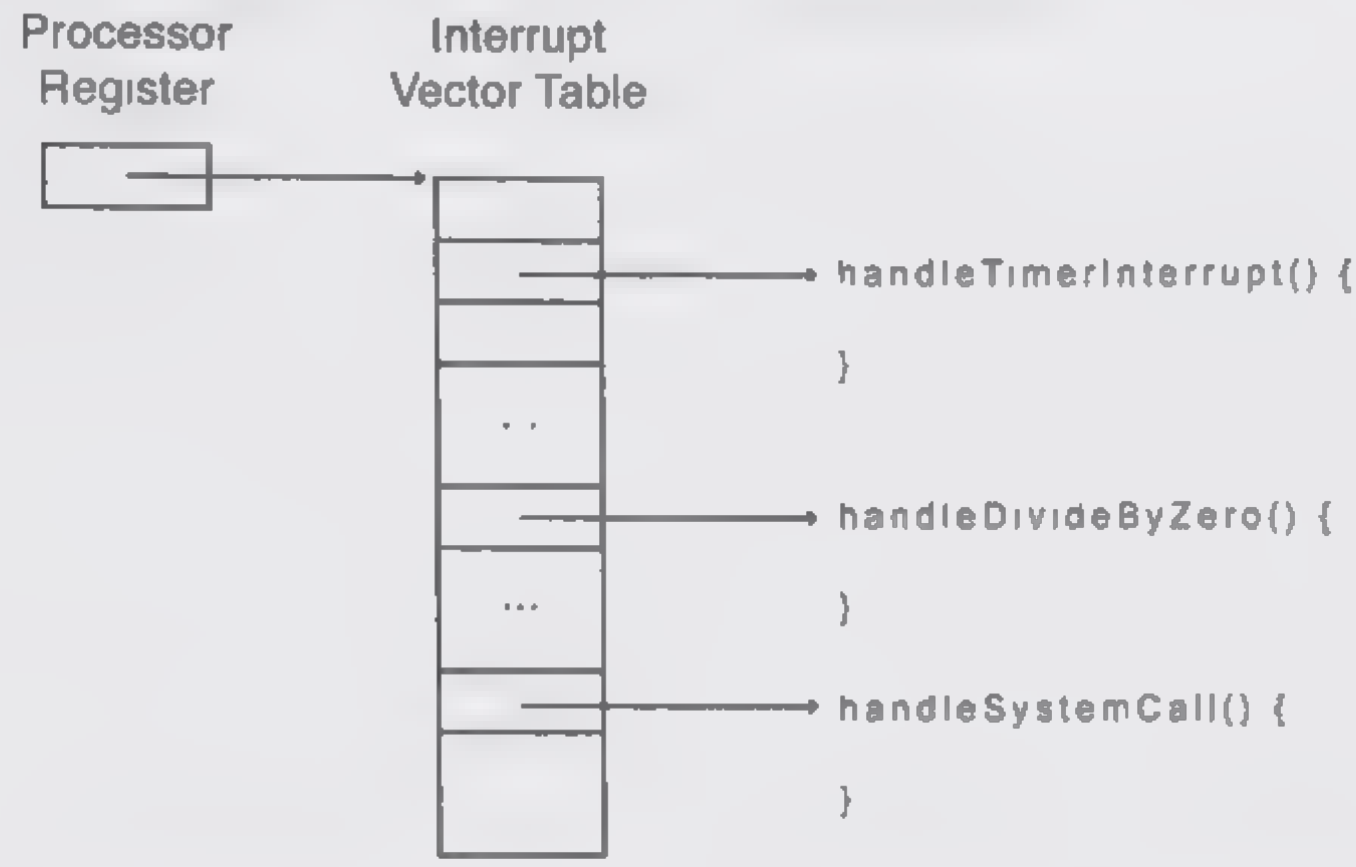

Figure 2.8: An interrupt vector table lists the kernel routines to handle various hardware interrupts, processor exceptions, and system calls.

With that context, we now describe the hardware and software mechanism for handling an interrupt, processor exception, or system call. Later, we reuse this same basic mechanism as a building block for implementing user-level signals.

2.4.1 Interrupt Vector Table

How does the hardware know which location to jump to on a trap?

When an interrupt, processor exception or system call trap occurs, the operating system must take different actions depending on whether the event is a divide-by-zero exception, a file read system call, or a timer interrupt. How does the processor know what code to run?

As Figure 2.8 illustrates, the processor has a special register that points to an area of kernel memory called the *interrupt vector table*. The interrupt vector table is an array of pointers, with each entry pointing to the first instruction of a different handler procedure in the kernel. An *interrupt handler* is the term used for the procedure called by the kernel on an interrupt.

interrupt vector table

interrupt handler

The format of the interrupt vector table is processor-specific. On the x86, for example, interrupt vector table entries 0 - 31 are for different types of processor exceptions (such as divide by-zero), entries 32 - 255 are for different types of interrupts (timer, keyboard, and so forth), and, by convention, entry 64 points to the system call trap handler. The hardware determines which hardware device caused the interrupt, whether the trap instruction was executed, or what exception condition occurred. Thus, the hardware can select the right entry from the interrupt vector table and invoke the appropriate handler.

Some other processors have a smaller number of entry points, instead putting a code indicating the cause of the interrupt into a special hardware

Multiprocessors and interrupt routing

On a multiprocessor, which of the various processors should take an interrupt? Some early multiprocessors dedicated a single processor ("processor 0") to handle all external interrupts. If an event required a change to what one of the other processors was doing, processor 0 could send an interprocessor interrupt to trigger that processor to switch to a new process.

For systems needing to do a large amount of input and output, such as a web server, directing all I/O through a single processor can become a bottleneck. In modern systems, interrupt routing is increasingly programmable, under control of the kernel. Each processor usually has its own hardware timer. Likewise, disk I/O events can be sent directly to the processor that requested the I/O operation rather than to a random processor. Modern processors can run substantially faster if their data is already loaded into the processor cache, versus if their code and data are in some other processor's cache.

Efficient delivery of network I/O packets is even more challenging. A high performance server might send and receive tens of thousands of packets per second, representing thousands of different connections. From a processing perspective, it is best to deliver incoming packets to the processor responsible for handling that connection, this requires the network interface hardware to re-direct the incoming packet based on the contents of its header (e.g., the IP address and port number of the client). Recent network controllers accomplish this by supporting multiple buffer descriptor rings for the same device, choosing which ring to use, and therefore which processor to interrupt, based on the header of the arriving packet.

---

register. In that case, the operating system software uses the code to index into the interrupt vector table.

EXAMPLE Why is the interrupt vector table stored in kernel rather than user memory?

ANSWER If the interrupt vector table could be modified by application code, the application could potentially hijack the network by directing all network interrupts to its own code. Similarly, the hardware register that points to the interrupt vector table must be a protected register that can be set only when in kernel mode. □

2.4.2 Interrupt Stack

Where should the interrupted process's state be saved, and what stack should the kernel's code use?

On most processors, a special, privileged hardware register points to a region of kernel memory called the *interrupt stack*. When an interrupt, pro-

interrupt stack

cessor exception, or system call trap causes a context switch into the kernel, the hardware changes the stack pointer to point to the base of the kernel's interrupt stack. The hardware automatically saves some of the interrupted process's registers by pushing them onto the interrupt stack before calling the kernel's handler.

When the kernel handler runs, it pushes any remaining registers onto the stack before performing its work. When returning from the interrupt, processor exception or system call trap, the reverse occurs: first, the handler pops the saved registers, and then, the hardware restores the registers it saved, returning to the point where the process was interrupted. When returning from a system call, the value of the saved program counter must be incremented so that the hardware returns to the instruction immediately *after* the one that caused the trap.

Why do interrupt handlers need to run on a kernel stack?

You might think you could use the process's user-level stack to store its state. However, a separate, kernel-level interrupt stack is needed for two reasons.

- **Reliability.** The process's user-level stack pointer might not be a valid memory address (e.g., if the program has a bug), but the kernel handler must continue to work properly.
- **Security.** On a multiprocessor, other threads running in the same process can modify user memory during the system call. If the kernel handler stores its local variables on the user-level stack, the user program might be able to modify the kernel's return address, potentially causing the kernel to jump to arbitrary code.

On a multiprocessor, each processor needs to have its own interrupt stack so that, for example, the kernel can handle simultaneous system calls and exceptions across multiple processors. For each processor, the kernel allocates a separate region of memory as that processor's interrupt stack.

2.4.3 Two Stacks per Process

Most operating system kernels go one step farther and allocate a kernel interrupt stack for every user-level process (and as we discuss in Chapter 4, every thread that executes user code). When a user-level process is running, the hardware interrupt stack points to that process's kernel stack. Note that when a process is running at user level, it is not running in the kernel so its kernel stack is empty.

Allocating a kernel stack per process makes it easier to switch to a new process inside an interrupt or system call handler. For example, a timer interrupt handler might decide to give the processor to a different process. Likewise, a system call might need to wait for an I/O operation to complete, in the meantime, some other process should run. With per process stacks, to suspend a process, we store a pointer to its kernel stack in the process control block, and

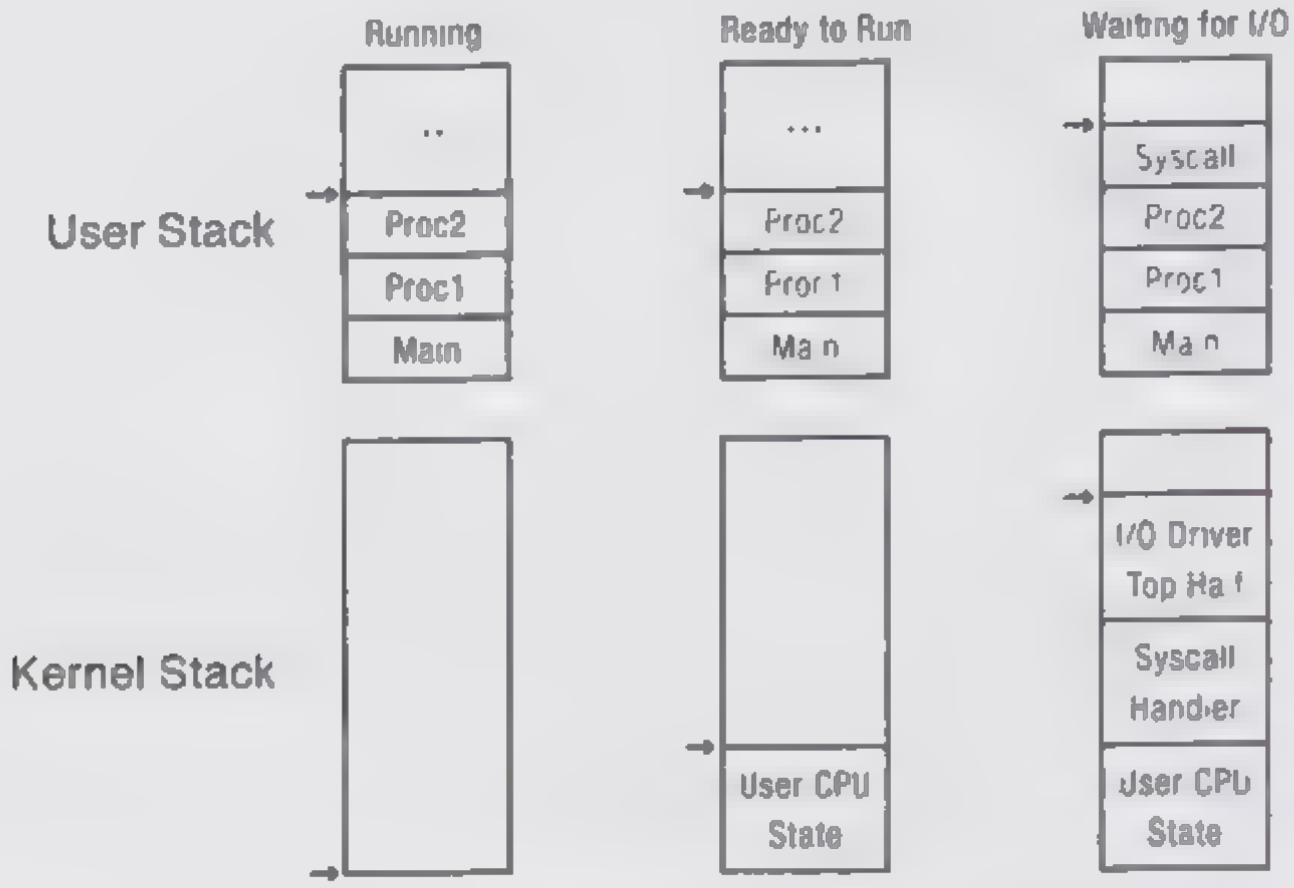

Figure 2.9: In most operating systems, a process has two stacks: one for executing user code and one for kernel code. The figure shows the kernel and user stacks for various states of a process. When a process is running in user mode, its kernel stack is empty. When a process has been preempted (ready but not running), its kernel stack will contain the user-level processor state at the point when the user process was interrupted. When a process is inside a system call waiting for I/O, the kernel stack contains the context to be resumed when the I/O completes, and the user stack contains the context to be resumed when the system call returns.

switch to the stack of the new process. We describe this mechanism in more detail in Chapter 4.

Figure 2.9 summarizes the various states of a process's user and kernel stacks:

- If the process is running on the processor in user mode, its kernel stack is empty, ready to be used for an interrupt, processor exception, or system call.
- If the process is running on the processor in kernel mode—due to an interrupt, processor exception or system call— its kernel stack is in use, containing the saved registers from the suspended user-level computation as well as the current state of the kernel handler.
- If the process is available to run but is waiting for its turn on the processor, its kernel stack contains the registers and state to be restored when the process is resumed.
- If the process is waiting for an I/O event to complete, its kernel stack contains the suspended computation to be resumed when the I/O finishes.

UNIX and kernel stacks

In the original implementation of UNIX, kernel memory was at a premium, main memory was roughly one million times more expensive per byte than it is today. The initial system could run with only 50KB of main memory. Instead of allocating an entire interrupt stack per process, UNIX allocated just enough memory in the process control block to store the user-level registers saved on a mode switch. In this way, UNIX could suspend a user level process with the minimal amount of memory. UNIX still needed a few kernel stacks: one to run the interrupt handler and one for every system call waiting for an I/O event to complete, but that is much less than one for every process.

Of course, now that memory is much cheaper, most systems keep things simple and allocate a kernel stack per process or thread.

2.4.4 Interrupt Masking

Can an interrupt occur while the kernel is in the middle of handling an interrupt?

Interrupts arrive asynchronously, the processor could be executing either user or kernel code when an interrupt arrives. In certain regions of the kernel — such as inside interrupt handlers themselves, or inside the CPU scheduler — taking an interrupt could cause confusion. If an interrupt handler is interrupted, we cannot set the stack pointer to point to the base of the kernel's interrupt stack — doing so would obliterate the state of the first handler.

interrupt disable

interrupt enable

To simplify the kernel design, the hardware provides a privileged instruction to temporarily defer delivery of an interrupt until it is safe to do so. On the x86 and several other processors, this instruction is called *disable interrupts*. However, this is a misnomer: the interrupt is only deferred (masked), and not ignored. Once a corresponding *enable interrupt* instruction is executed, any pending interrupts are delivered to the processor. The instructions to mask and unmask interrupts must be privileged, otherwise, user code could inadvertently or maliciously disable the hardware timer, allowing the machine to freeze.

If multiple interrupts arrive while interrupts are disabled, the hardware delivers them in turn when interrupts are re-enabled. However, since the hardware has limited buffering for pending interrupts, some interrupts may be lost if interrupts are disabled for too long a period of time. Generally, the hardware will buffer one interrupt of each type, the interrupt handler is responsible for checking the device hardware to see if multiple pending I/O events need to be processed.

If the processor takes an interrupt in kernel mode with interrupts enabled, it is safe to use the current stack pointer rather than resetting it to the base of the interrupt stack. This approach can recursively push a series of handlers' states onto the stack, then, as each one completes, its state is popped from the stack, and the earlier handler is resumed where it left off.

Interrupt handlers: top and bottom halves

When a machine invokes an interrupt handler because some hardware event occurred (e.g., a timer expired, a key was pressed, a network packet arrived, or a disk I/O completed), the processor hardware typically masks interrupts while the interrupt handler executes. While interrupts are disabled, another hardware event will not trigger another invocation of the interrupt handler until the interrupt is re-enabled.

Some interrupts can trigger a large amount of processing, and it is undesirable to leave interrupts masked for too long. Hardware I/O devices have a limited amount of buffering, which can lead to dropped events if interrupts are not processed in a timely fashion. For example, keyboard hardware can drop keystrokes if the keyboard buffer is full. Interrupt handlers are therefore divided into *a top half* and a *bottom half*. Unfortunately, this terminology can differ a bit from system to system; in Linux, the sense of top and bottom are reversed. In this book, we adopt the more common (non-Linux) usage.

The interrupt handler's bottom half is invoked by the hardware and executes with interrupts masked. It is designed to complete quickly. The bottom half typically saves the state of the hardware device, resets it so that it can receive a new event, and notifies the scheduler that the top half needs to run. At this point, the bottom half is done, and it can re-enable interrupts and return to the interrupted task or (if the event is high priority) switch to the top half but with interrupts enabled. When the top half runs, it can do more general kernel tasks, such as parsing the arriving packet, delivering it to the correct user-level process, sending an acknowledgment, and so forth. The top half can also do operations that require the kernel to wait for exclusive access to shared kernel data structures, the topic of Chapter 5.

2.4.5 Hardware Support for Saving and Restoring Registers

How do we save and restore the current state of the user process?

An interrupted process's registers must be saved so that the process can be restarted exactly where it left off. Because the handler might change the values in those registers as it executes, the state must be saved *before* the handler runs. Because most instructions modify the contents of registers, the hardware typically provides special instructions to make it easier to save and restore user state.

To make this concrete, consider the x86 architecture. Rather than relying on handler software to do all the work, when an interrupt or trap occurs:

- If the processor is in user mode, the x86 pushes the interrupted process's stack pointer onto the kernel's interrupt stack and switches to the kernel stack.
- The x86 pushes the interrupted process's instruction pointer.

- The x86 pushes the x86 *processor status word*. The processor status word includes control bits, such as whether the most recent arithmetic operation in the interrupted code resulted in a positive, negative, or zero value. This needs to be saved and restored for the correct behavior of any subsequent conditional branch instruction.

The hardware saves the values for the stack pointer, program counter, and processor status word *before* jumping through the interrupt vector table to the interrupt handler. Once the handler starts running, these values will be those of the handler, not those of the interrupted process.

Once the handler starts running, it can use the pushad ("push all double") instruction to save the remaining registers onto the stack. This instruction saves all 32 bit x86 integer registers. On a 16-bit x86, pusha is used instead. Because the kernel does not typically perform floating point operations, those do not need to be saved unless the kernel switches to a different process.

The x86 architecture has complementary features for restoring state: a popad instruction to pop an array of integer register values off the stack into the registers and an iret (return from interrupt) instruction that loads a stack pointer, instruction pointer, and processor status word off of the stack into the appropriate processor registers.

2.5 Putting It All Together: x86 Mode Transfer

The high level steps needed to handle an interrupt, processor exception, or system call are simple, but the details require some care.

To give a concrete example of how such "carefully crafted" code works, we now describe one way to implement an interrupt-triggered mode switch on the x86 architecture. Different operating systems on the x86 follow this basic approach, though details differ. Similarly, different architectures handle the same types of issues, but they may do so with different hardware support.

First, we provide some background on the x86 architecture. The x86 is segmented, so pointers come in two parts: (i) a segment, a region of memory such as code, data, or stack, and (ii) an offset within that segment. The current user-level instruction is a combination of the code segment (cs register) plus the instruction pointer (eip register). Likewise, the current stack position is the combination of the stack segment (ss) and the stack pointer within the stack segment (esp). The current privilege level is stored as the low-order bits of the cs register rather than in the processor status word (eflags register). The eflags register has condition codes that are modified as a by-product of executing instructions, the eflags register also has other flags that control the processor's behavior, such as whether interrupts are masked or not.

When a user-level process is running, the current state of the processor, stack, kernel interrupt vector table, and kernel stack is illustrated in Figure 2.10. When a processor exception or system call trap occurs, the hardware carefully

Architectural support for fast mode switches

Some processor architectures are able to execute user- and kernel-mode switches very efficiently, while other architectures are much slower at performing these switches.

The SPARC architecture is in the first camp. SPARC defines a set of *register windows* that operate like a hardware stack. Each register window includes a full set of the registers defined by the SPARC instruction set. When the processor performs a procedure call, it shifts to a new window, so the compiler never needs to save and restore registers across procedure calls, making them quite fast. (At a deep enough level of recursion, the SPARC will run out of its register windows; it then takes an exception that saves half the windows and resumes execution. Another exception occurs when the processor pops its last window, allowing the kernel to reload the saved windows.)

Mode switches can be quite fast on the SPARC. On a mode switch, the processor switches to a different register window. The kernel handler can then run, using the registers from the new window and not disturbing the values stored in the interrupted process's copy of its registers. Unfortunately, this comes at a cost: switching between different processes is quite expensive on the SPARC, as the kernel needs to save and restore the entire register set of every active window.

The Motorola 88000 was in the second camp. The 88000 was an early pipelined architecture; now, almost all modern computers are pipelined. For improved performance, pipelined architectures execute multiple instructions at the same time. For example, one instruction is being fetched while another is being decoded, a third is completing a floating point operation, and a fourth is finishing a store to memory. When an interrupt or processor exception occurred on the 88000, the pipeline operation was suspended, and the operating system kernel was required to save and restore the entire state of the pipeline to preserve transparency to user code.

Most modern processors with deep execution pipelines, such as the x86, instead provide *precise interrupts*: the hardware first completes all instructions that occur, in program order, before the interrupted instruction. The hardware annuls any instruction that occurs, in program order, after the interrupt or trap, even if the instruction is in progress when the processor detects the interrupt

---

saves a small amount of the interrupted thread state, leaving the system as shown in Figure 2.11:

1. **Mask interrupts.** The hardware starts by preventing any interrupts from occurring while the processor is in the middle of switching from user mode to kernel mode.

2. **Save three key values.** The hardware saves the values of the stack pointer (the x86 esp and ss registers), the execution flags (the x86 eflags register),

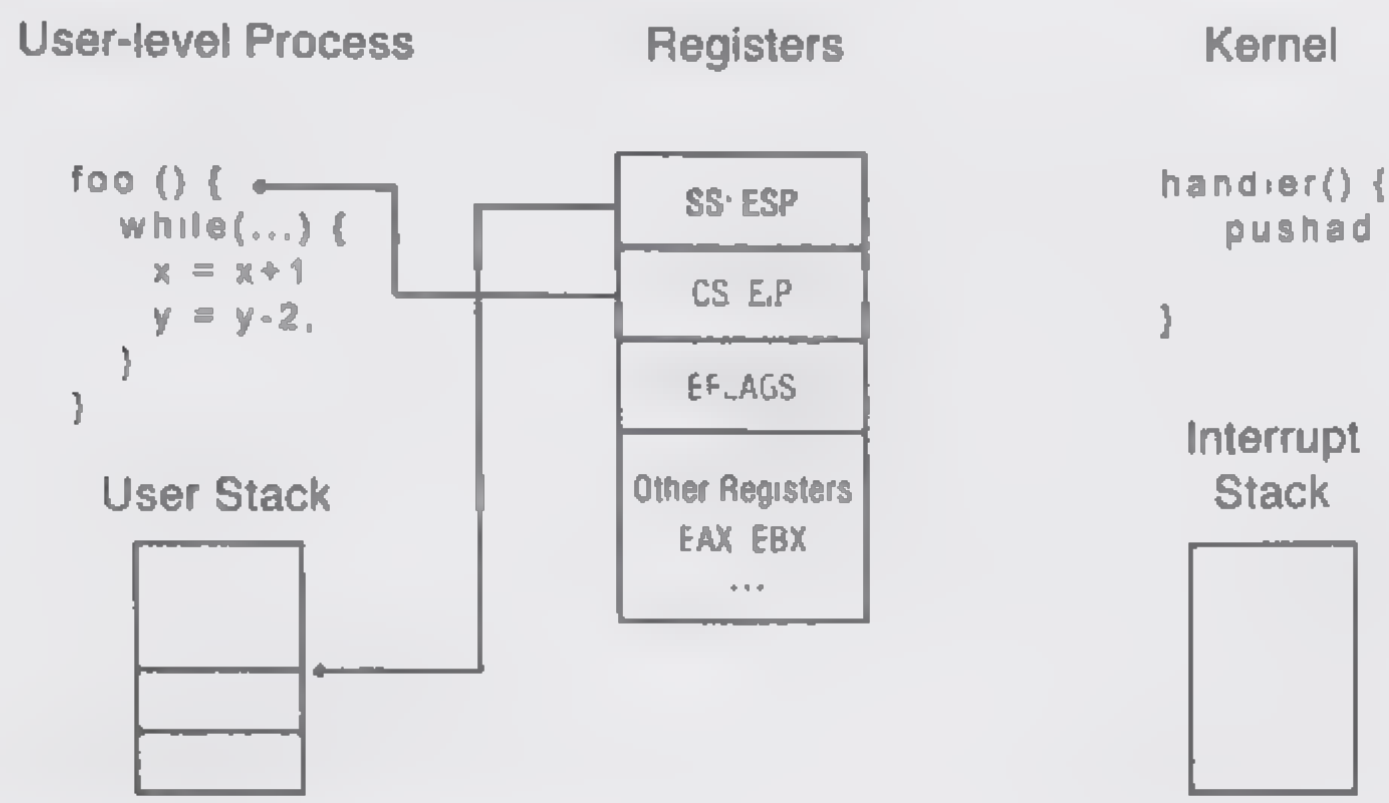

Figure 2.10: State of the system before an interrupt handler is invoked on the x86 architecture. SS is the stack segment, ESP is the stack pointer, CS is the code segment, and EIP is the program counter. The program counter and stack pointer refer to locations in the user process, and the interrupt stack is empty.

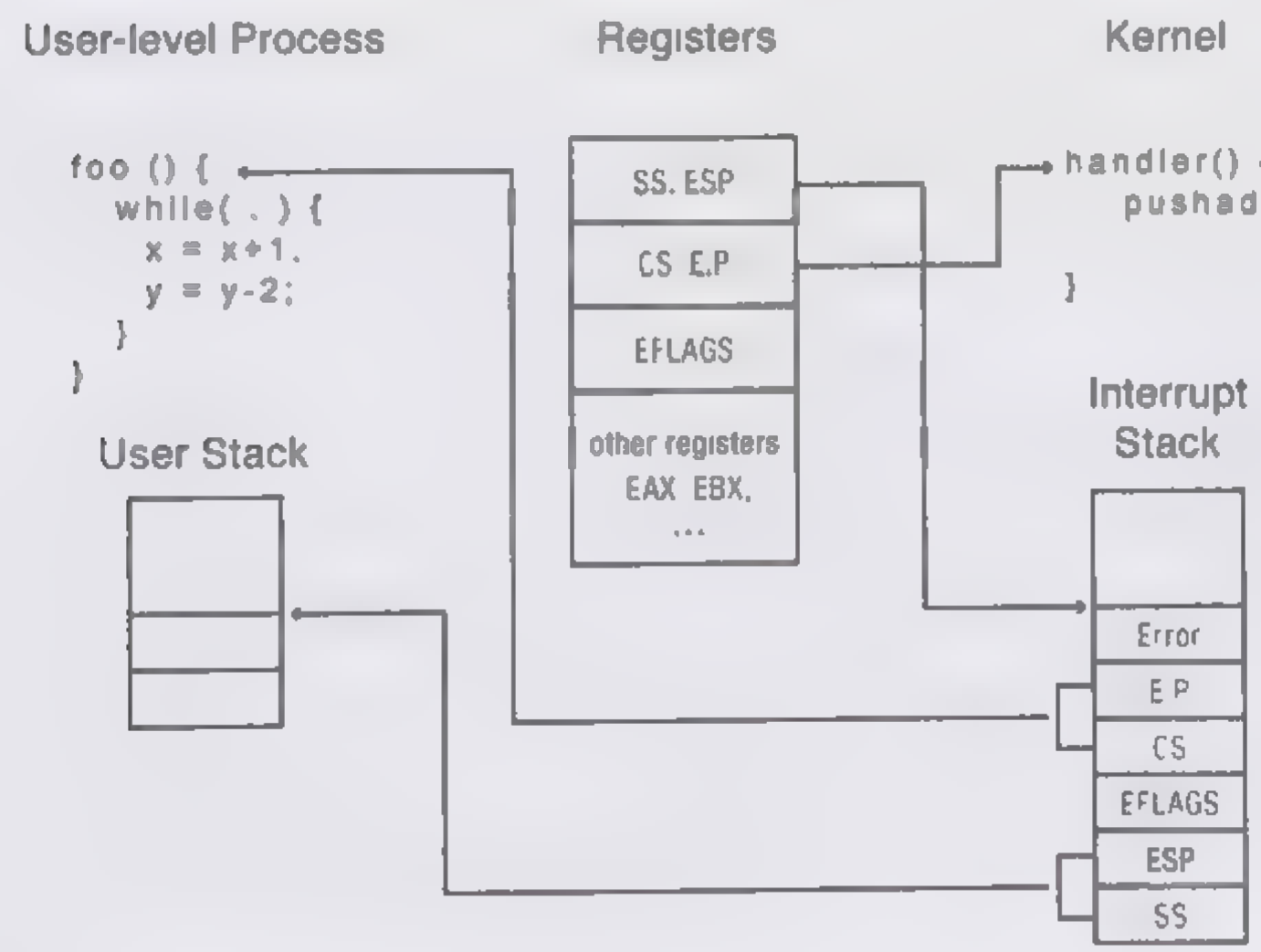

Figure 2.11: State of the system after the x86 hardware has jumped to the interrupt handler. The hardware saves the user context on the kernel interrupt stack and changes the program counter/stack to locations in kernel memory.

and the instruction pointer (the x86 eip and cs registers) to internal, temporary hardware registers.

3. **Switch onto the kernel interrupt stack.** The hardware then switches the stack segment/stack pointer to the base of the kernel interrupt stack, as specified in a special hardware register.

4. **Push the three key values onto the new stack.** Next, the hardware stores the internally saved values onto the stack.

5. **Optionally save an error code.** Certain types of exceptions, such as page faults, generate an error code to provide more information about the event, for these exceptions, the hardware pushes this code, making it the top item on the stack. For other types of events, the software interrupt handler pushes a dummy value onto the stack so that the stack format is identical in both cases.

6. **Invoke the interrupt handler.** Finally, the hardware changes the code segment/program counter to the address of the interrupt handler procedure. A special register in the processor contains the location of the interrupt vector table in kernel memory. This register can only be modified by the kernel. The type of interrupt is mapped to an index in this array, and the code segment/program counter is set to the value at this index. This starts the handler software.

The handler must first save the rest of the interrupted process's state — it needs to save the other registers before it changes them! The handler pushes the rest of the registers, including the current stack pointer, onto the stack using the x86 pushad instruction.

As Figure 2.12 shows, at this point the kernel's interrupt stack holds (1) the stack pointer, execution flags, and program counter saved by the hardware, (2) an error code or dummy value, and (3) a copy of all of the general registers (including the stack pointer but not the instruction pointer or eflags register).

Once the handler has saved the interrupted thread's state to the stack, it can use the registers as it pleases, and it can push additional items onto the stack. So, the handler can now do whatever work it needs to do.

When the handler completes, it can resume the interrupted process. To do this, the handler pops the registers it saved on the stack. This restores all registers except the execution flags, program counter, and stack pointer. For the x86 instruction set, the popad instruction is commonly used. The handler also pops the error value off the stack.

Finally, the handler executes the x86 iret instruction to restore the code segment, program counter, execution flags, stack segment, and stack pointer from the kernel's interrupt stack.

This restores the process state to exactly what it was before the interrupt. The process continues execution as if nothing happened.

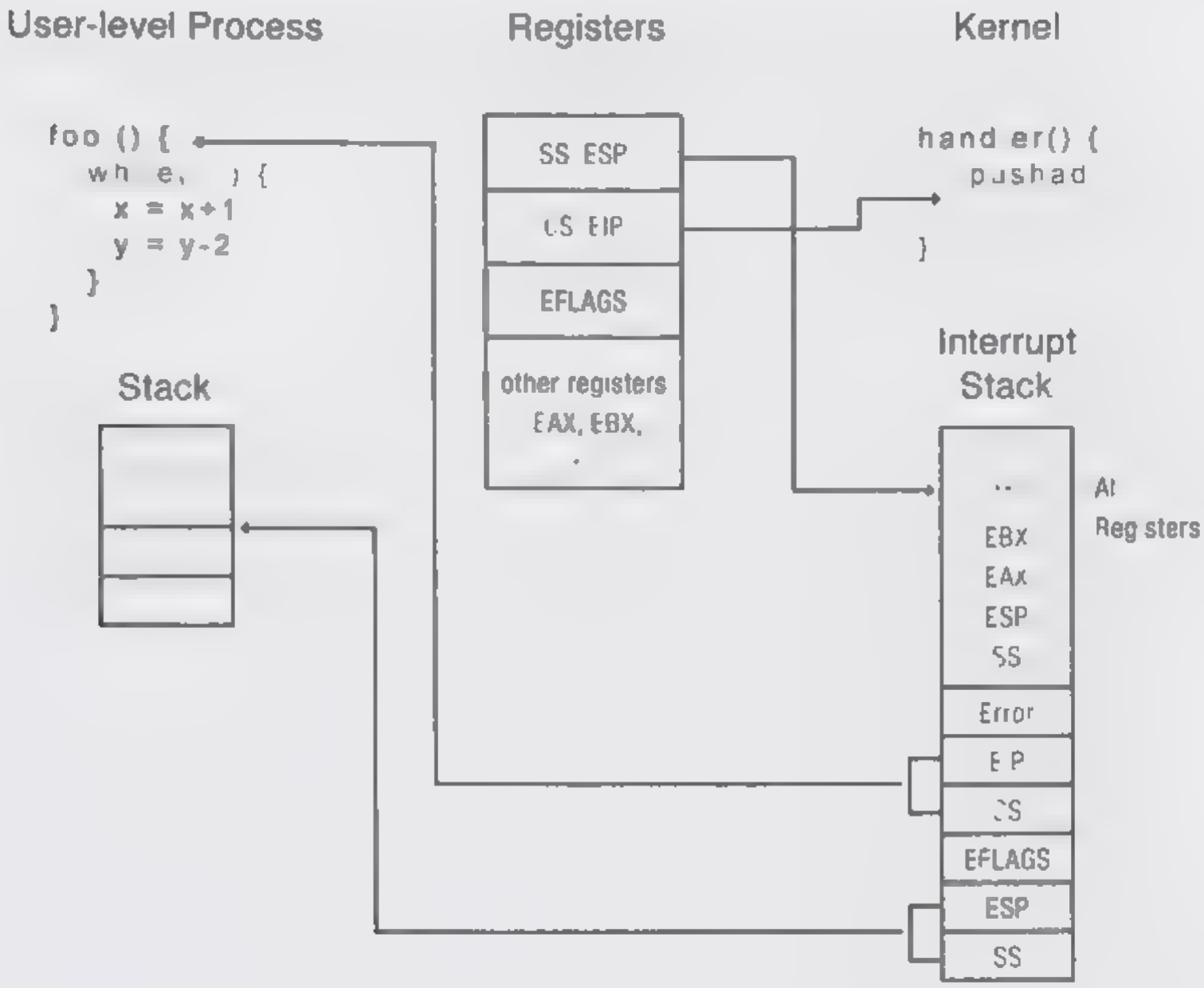

Figure 2.12: State of the system after the interrupt handler has started executing on the x86 architecture. The handler first saves the current state of the processor registers, since it may overwrite them. Note that this saves the stack pointer twice: first, the user stack pointer then the kernel stack pointer.

A small but important detail occurs when the hardware takes an exception to emulate an instruction in the kernel, e.g., for missing floating point hardware. If the handler returns back to the instruction that caused the exception, another exception would instantly recur! To prevent an infinite loop, the exception handler modifies the program counter stored at the base on the stack to point to the instruction immediately after the one causing the mode switch. The iret instruction can then return to the user process at the correct location.

For a system call trap, the Intel x86 hardware does the increment when it saves the user-level state. The program counter for the instruction after the trap is saved on the kernel's interrupt stack.

EXAMPLE A trapframe is the data stored by the hardware and interrupt handler at the base of the interrupt stack, describing the state of the user-level execution context. Typically, a pointer to the trapframe is passed as an argument to the handler, e.g., to allow system calls to access arguments passed in registers.

How large is the 32-bit x86 trapframe in the example given above?

ANSWER The hardware saves six registers; the interrupt handler saves another eight general-purpose registers. In all, **56 bytes** are saved in the trapframe. □

2.6 Implementing Secure System Calls

How do library code and the kernel work together to implement protected procedure calls from the application into the kernel?

The operating system kernel constructs a restricted environment for process execution to limit the impact of erroneous and malicious programs on system reliability. Any time a process needs to perform an action outside of its protection domain — to create a new process, read from the keyboard, or write a disk block — it must ask the operating system to perform the action on its behalf, **via a system call.**

System calls provide the illusion that the operating system kernel is simply a set of library routines available to user programs. To the user program, the kernel provides a set of system call procedures, each with its own arguments and return values, that can be called like any other routine. The user program need not concern itself with how the kernel implements these calls.

Implementing system calls requires the operating system to define a *calling convention* — how to name system calls, pass arguments, and receive return values across the user/kernel boundary. Typically, the operating system uses the same convention as the compiler uses for normal procedures — some combination of passing arguments in registers and on the execution stack.

Once the arguments are in the correct format, the user-level program can issue a system call by executing the trap instruction to transfer control to the kernel. System calls, like interrupts and processor exceptions, share the same mechanism for switching between user and kernel mode. In fact, the x86 instruction to trap into the kernel on a system call is called `int`, for "software interrupt."

Inside the kernel, a procedure implements each system call. This procedure behaves exactly as if the call was made from within the kernel but with one notable difference: the kernel must implement its system calls in a way that protects itself from all errors and attacks that might be launched by the misuse of the interface. Of course, most applications will use the interface correctly! But errors in an application program must not crash the kernel, and a computer virus must not be able to use the system call interface to take control of the kernel. One can think of this as an extreme version of defensive programming: the kernel should always assume that the parameters passed to a system call are intentionally designed to be as malicious as possible.

pair of stubs

We bridge these two views — the user program calling the system call, and the kernel implementing the system call — with a pair of stubs. A *pair of stubs* is a pair of procedures that mediate between two environments, in this case between the user program and the kernel. Stubs also mediate procedure calls between computers in a distributed system.

Figure 2.13 illustrates the sequence of steps involved in a system call:

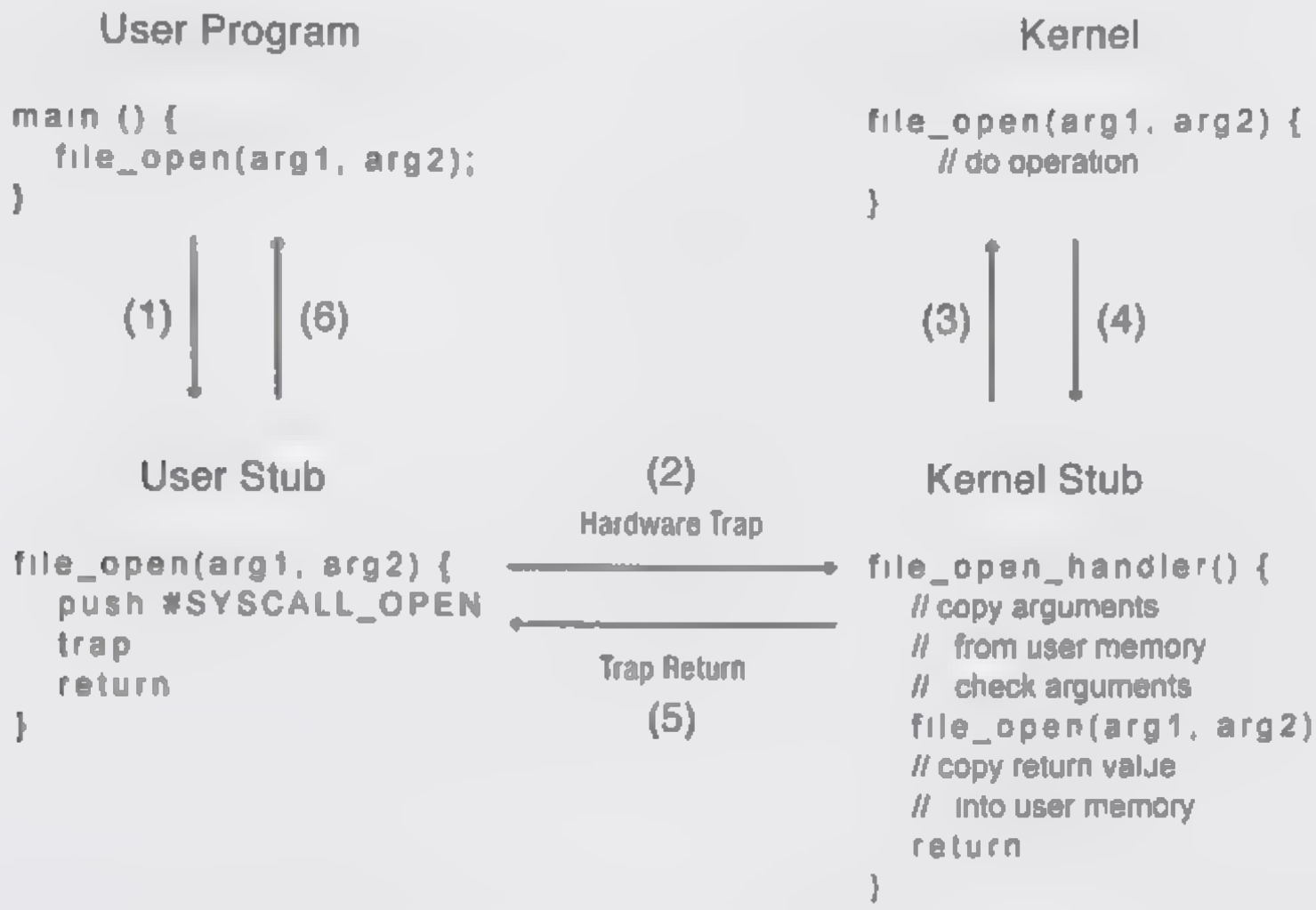

Figure 2.13: A pair of stubs mediates between the user-level caller and the kernel's implementation of system calls. The code is for the file_open system call; other calls have their own stubs. (1) The user process makes a normal procedure call to a stub linked with the process. (2) The stub executes the trap instruction. This transfers control to the kernel trap handler. The trap handler copies and checks its arguments and then (3) calls a routine to do the operation. Once the operation completes, (4) the code returns to the trap handler, which copies the return value into user memory and (5) resumes the user stub immediately after the trap. (6) The user stub returns to the user-level caller.

1. The user program calls the user stub in the normal way, oblivious to the fact the implementation of the procedure is in fact in the kernel.

2. The user stub fills in the code for the system call and executes the trap instruction.

3. The hardware transfers control to the kernel, vectoring to the system call handler. The handler acts as a stub on the kernel side, copying and checking arguments and then calling the kernel implementation of system call.

4. After the system call completes, it returns to the handler.

5. The handler returns to user level at the next instruction in the stub.

6. The stub returns to the caller.

We next describe these steps in more detail. Figure 2.14 illustrates the behavior of the user level stub for the x86. The operating system provides a

```
// We assume that the caller put the filename onto the stack,
// using the standard calling convention for the x86.

open:
// Put the code for the system call we want into %eax.
    movl #SysCallOpen, %eax

// Trap into the kernel.
    int #TrapCode

// Return to the caller; the kernel puts the return value in %eax.
    ret
```

Figure 2.14: User-level library stub for the file system open system call for the x86 processor. SysCallOpen is the code for the specific system call to run. TrapCode is the index into the x86 interrupt vector table for the system call handler.

library routine for each system call that takes its arguments, reformats them according to the calling convention, and executes a trap instruction. When the kernel returns, the stub returns the result provided by the kernel. Of course, the user program need not use the library routine—it is free to trap directly to the kernel. In turn, the kernel must protect itself from misbehaving programs that do not format arguments correctly.

The system call calling convention is arbitrary. In Figure 2.14, the code passes its arguments on the user stack, storing the system call code in the register %eax. The return value comes back in %eax, so there is no work to do on the return.

The int instruction saves the program counter, stack pointer, and eflags on the kernel stack before jumping to the system call handler through the interrupt vector table. The kernel handler saves any additional registers that must be preserved across function calls. It then examines the system call integer code in %eax, verifies that it is a legal opcode, and calls the correct stub for that system call.

The kernel stub has four tasks:

- **Locate system call arguments.** Unlike a regular kernel procedure, the arguments to a system call are stored in user memory, typically on the user stack. Of course, the user stack pointer may be corrupted! Even if it is valid, it is a virtual, not a physical, address. If the system call has a pointer argument (e.g., a file name or buffer), the stub must check the address to verify it is a legal address within the user domain. If so, the stub converts it to a physical address that the kernel can safely use it. In Figure 2.14, the pointer to the string representing the file name is stored on the stack; therefore, the stub must check and translate both the stack address and the string pointer.

- **Validate parameters.** The kernel must also protect itself against malicious or accidental errors in the format or content of its arguments. A file name is typically be a zero-terminated string, but the kernel cannot trust the

user code to always work correctly. The file name may be corrupted; it may point to memory outside the application's region, it may start inside the application's memory region but extend beyond it, the application may not have permission to access the file; the file may not exist; and so forth. If an error is detected, the kernel returns it to the user program; otherwise, the kernel performs the operation on the application's behalf

- **Copy before check.** In most cases, the kernel copies system call parameters into kernel memory before performing the necessary checks. The reason for this is to prevent the application from modifying the parameter *after* the stub checks the value, but *before* the parameter is used in the actual implementation of the routine. This is called a *time of check vs. time of use* (TOCTOU) attack. For example, the application could call open with a valid file name but, after the check, change the contents of the string to be a different name, such as a file containing another user's private data.

  time of check vs. time of use attack

  TOCTOU is not a new attack—the first occurrence dates from the mid-1960's. While it might seem that a process necessarily stops whenever it does a system call, this is not always the case. For example, if one process shares a memory region with another process, then the two processes working together can launch a TOCTOU attack. Similarly, a parallel program running on two processors can launch a TOCTOU attack, where one processor traps into the kernel while the other modifies the string at precisely the right (or wrong) time. Note that the kernel needs to be correct in every case, while the attacker can try any number of times before succeeding.

- **Copy back any results.** For the user program to access the results of the system call, the stub must copy the result from the kernel into user memory. Again, the kernel must first check the user address and convert it to a kernel address before performing the copy.

Putting this together, Figure 2.15 shows the kernel stub for the system call open. In this case, the return value fits in a register so the stub can return directly; in other cases, such as a file read, the stub would need to copy data back into a user-level buffer.

After the system call finishes, the handler pops any saved registers (except %eax) and uses the iret instruction to return to the user stub immediately after the trap, allowing the user stub to return to the user program.

2.7 Starting a New Process

How does the operating system kernel start a new process?

Thus far, we have described how to transfer control from a user-level process to the kernel on an interrupt, processor exception, or system call and how the kernel resumes execution at user level when done.

```
int KernelStub_Open() {
    char *localCopy[MaxFileNameSize + 1];

// Check that the stack pointer is valid and that the arguments are stored at
   valid addresses.

    if (!ValidUserAddressRange(userStackPointer, userStackPointer + sizeof arguments,
        return error_code;

// Fetch pointer to file name from user stack and convert it to a kernel pointer
    filename = VirtualToKernel(userStackPointer);

   Make a local copy of the filename.  This prevents the application
   from changing the name surreptitiously.

   The string copy needs to check each address in the string before use to make sure
   it is valid.

   The string copy terminates after it copies MaxFileNameSize to ensure we
   do not overwrite our internal buffer.

    if (!VirtualToKernelStringCopy(filename, localCopy, MaxFileNameSize))
        return error_code;

// Make sure the local copy of the file name is null terminated

    localCopy[MaxFileNameSize] = 0;

// Check if the user is permitted to access this file

    if (!UserFileAccessPermitted(localCopy, current_process)
        return error_code;

// Finally, call the actual routine to open the file.  This returns a file
   handle on success, or an error code on failure.

    return Kernel_Open(localCopy);
}
```

Figure 2.15: Stub routine for the open system call inside the kernel. The kernel must validate all parameters to a system call before it uses them.

We now examine how to start running at user level in the first place. The kernel must:

- Allocate and initialize the process control block.
- Allocate memory for the process.
- Copy the program from disk into the newly allocated memory.
- Allocate a user-level stack for user-level execution.
- Allocate a kernel level stack for handling system calls, interrupts and processor exceptions.

To start running the program, the kernel must also:

- **Copy arguments into user memory.** When starting a program, the user may give it arguments, much like calling a procedure. For example, when you click on a file icon in MacOS or Windows, the window manager asks the kernel to start the application associated with the file, passing it the file name to open. The kernel copies the file name from the memory of the window manager process to a special region of memory in the new process. By convention, arguments to a process are copied to the base of the user-level stack, and the user's stack pointer is incremented so those addresses are not overwritten when the program starts running.

- **Transfer control to user mode.** When a new process starts, there is no saved state to restore. While it would be possible to write special code for this case, most operating systems re-use the same code to exit the kernel for starting a new process and for returning from a system call. When we create the new process, we allocate a kernel stack to it, and we reserve room at the bottom of the kernel stack for the initial values of its user-space registers, program counter, stack pointer, and processor status word. To start the new program, we can then switch to the new stack and jump to the end of the interrupt handler. When the handler executes popad and iret, the processor "returns" to the start of the user program.

Finally, although you can think of a user program as starting with a call to main, in fact the compiler inserts one level of indirection. It puts a stub at the location in the process's memory where the kernel will jump when the process starts. The stub's job is to call main and then, if main returns, to call exit, the system call to terminate the process. Without the stub, a user program that returned from main would try to pop the return program counter, and since there is no such address on the stack, the processor would start executing random code.

```
start(arg1, arg2) {
    main(arg1, arg2);    // Call program main
    exit();              // If main returns, call exit.
}
```

2.8 Implementing Upcalls

How does the kernel deliver an asynchronous event to a user process?

We can use system calls for most of the communication between applications and the operating system kernel. When a program requests a protected operation, it can trap to ask the kernel to perform the operation on its behalf. Likewise, if the application needs data inside the kernel, a system call can retrieve it.

To allow applications to implement operating system-like functionality, we need something more. For many of the reasons that kernels need interrupt-based event delivery, applications can also benefit from being told when events occur that need their immediate attention. Throughout this book, we will see this pattern repeatedly: the need to *virtualize* some part of the kernel so that applications can behave more like operating systems. We call virtualized interrupts and exceptions *upcalls*. In UNIX, they are called *signals*; in Windows, they are *asynchronous events*.

upcall

There are several uses for immediate event delivery with upcalls:

- **Preemptive user-level threads.** Just as the operating system kernel runs multiple processes on a single processor, an application may run multiple tasks, or threads, in a process. A user-level thread package can use a periodic timer upcall as a trigger to switch tasks, to share the processor more evenly among user-level tasks or to stop a runaway task, e.g., if a web browser needs to terminate an embedded third party script.

- **Asynchronous I/O notification.** Most system calls wait until the requested operation completes and then return. What if the process has other work to do in the meantime? One approach is *asynchronous I/O*: a system call starts the request and returns immediately. Later, the application can poll the kernel for I/O completion, or a separate notification can be sent via an upcall to the application when the I/O completes.

asynchronous I/O

- **Interprocess communication.** Most interprocess communication can be handled with system calls—one process writes data, while the other reads it sometime later. A kernel upcall is needed if a process generates an event that needs the instant attention of another process. As an example, UNIX sends an upcall to notify a process when the debugger wants to suspend or resume the process. Another use is for logout—to notify applications that they should save file data and cleanly terminate.

- **User-level exception handling.** Earlier, we described a mechanism where processor exceptions, such as divide-by-zero errors, are handled by the kernel. However, many applications have their own exception handling routines, e.g., to ensure that files are saved before the application shuts down. For this, the operating system needs to inform the application when it receives a processor exception so the application runtime, rather than the kernel, handles the event.

- **User-level resource allocation.** Operating systems allocate resources deciding which users and processes should get how much CPU time, how much memory, and so forth. In turn, many applications are resource adaptive—able to optimize their behavior to differing amounts of CPU time or memory. An example is Java garbage collection. Within limits, a Java process can adapt to different amounts of available memory by changing the frequency with which it runs its garbage collector. The more memory, the less time Java needs to run its collector, speeding

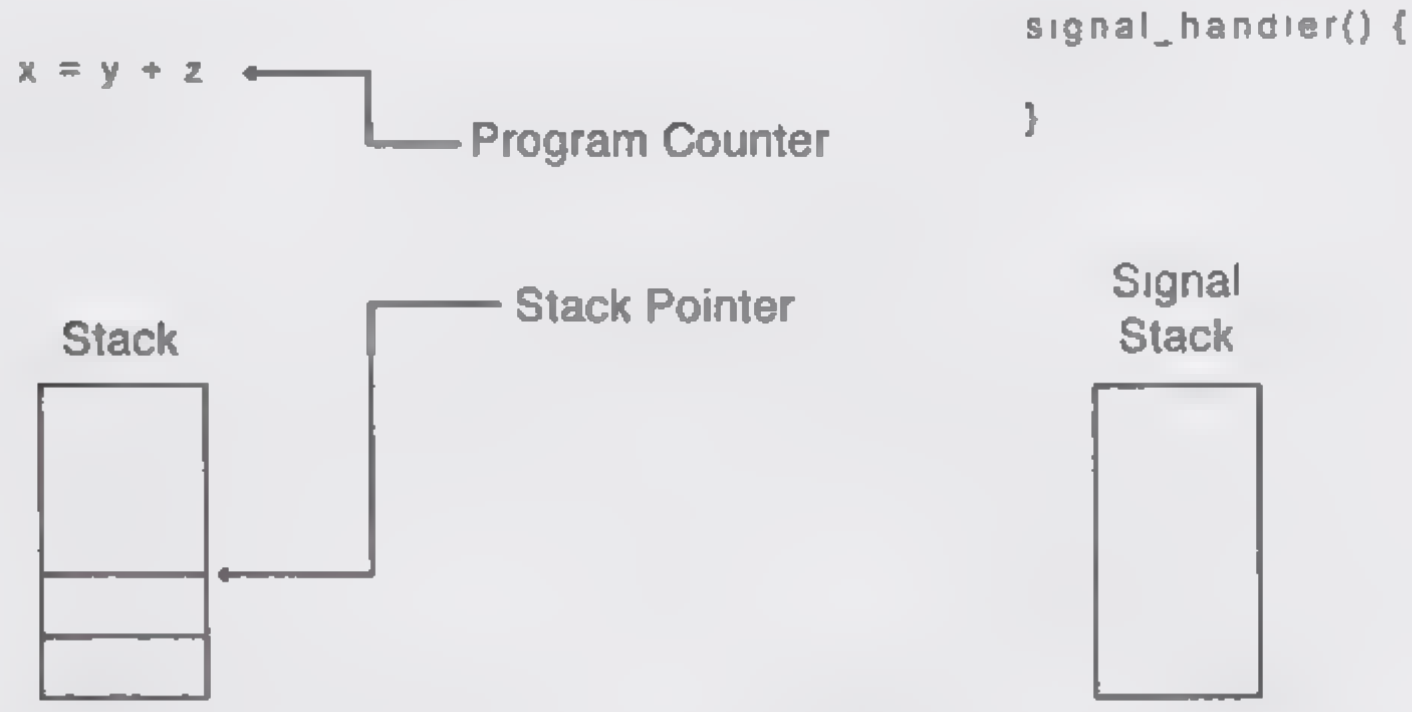

Figure 2.16: The state of the user program and signal handler before a UNIX signal UNIX signals behave analogously to processor exceptions, but at user level.

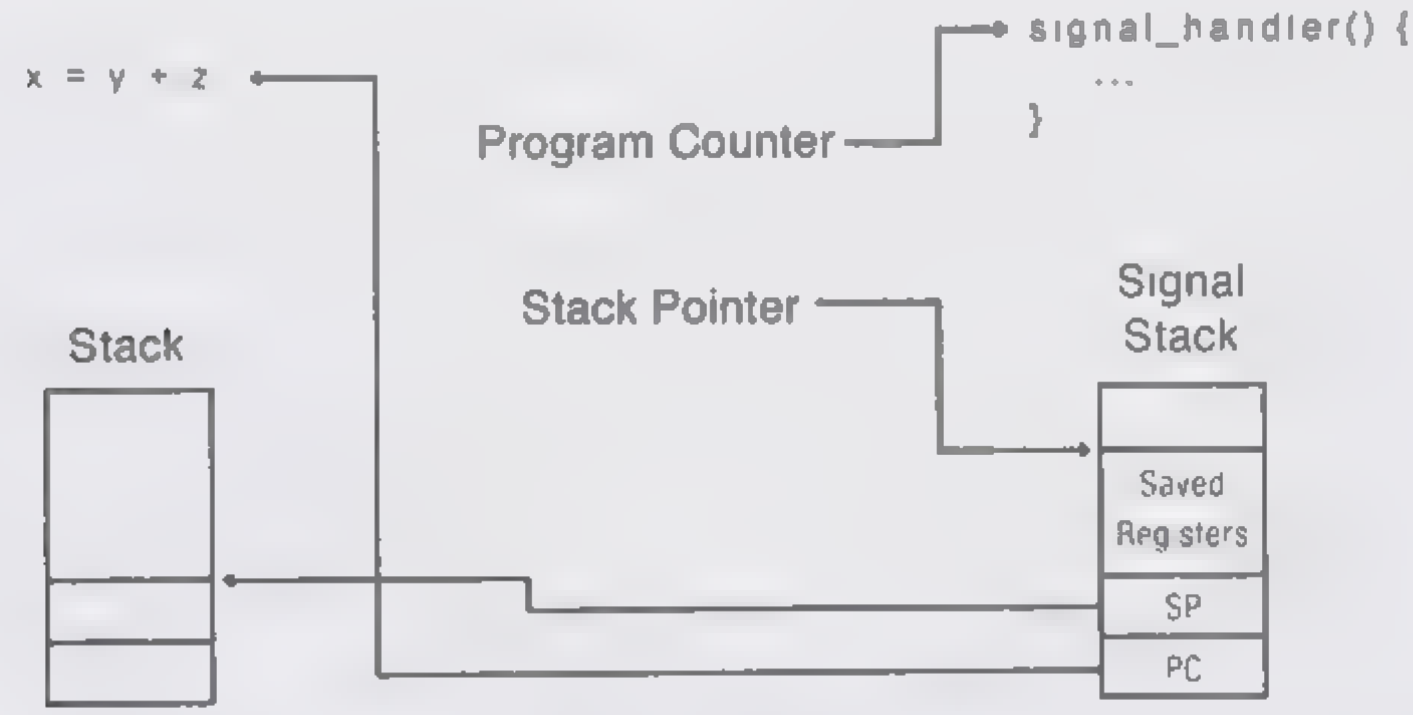

Figure 2.17: The state of the user program and signal handler during a UNIX signal The signal stack stores the state of the hardware registers at the point where the process was interrupted, with room for the signal handler to execute on the signal stack.

execution. For this, the operating system must inform the process when its allocation changes, e.g., because some other process needs more or less memory.

Upcalls from kernels to user processes are not always needed. Many applications are more simply structured around an event loop that polls for events and then processes each event in turn. In this model, the kernel can pass data to the process by sending it events that do not need to be handled immediately. In fact, until recently, Windows lacked support for the immediate delivery of upcalls to user-level programs.

We next describe UNIX signals as a concrete example of kernel support for upcalls. As shown in Figures 2.16 and 2.17, UNIX signals share many

similarities with hardware interrupts:

- **Types of signals.** In place of hardware-defined interrupts and processor exceptions, the kernel defines a limited number of signal types that a process can receive.

- **Handlers.** Each process defines its own handlers for each signal type, much as the kernel defines its own interrupt vector table. If a process does not define a handler for a specific signal, then the kernel calls a default handler instead.

- **Signal stack.** Applications have the option to run UNIX signal handlers on the process's normal execution stack or on a special signal stack allocated by the user process in user memory. Running signal handlers on the normal stack makes it more difficult for the signal handler to manipulate the stack, e.g., if the runtime needs to raise a language level exception.

- **Signal masking.** UNIX defers signals for events that occur while the signal handler for those types of events is in progress. Instead, the signal is delivered once the handler returns to the kernel. UNIX also provides a system call for applications to mask signals as needed.

- **Processor state.** The kernel copies onto the signal stack the saved state of the program counter, stack pointer, and general-purpose registers at the point when the program stopped. Normally, when the signal handler returns, the kernel reloads the saved state into the processor to resume program execution. The signal handler can also modify the saved state, e.g., so that the kernel resumes a different user-level task when the handler returns.

The mechanism for delivering UNIX signals to user processes requires only a small modification to the techniques already described for transferring control across the kernel user boundary. For example, on a timer interrupt, the hardware and the kernel interrupt handler save the state of the user-level computation. To deliver the timer interrupt to user level, the kernel copies that saved state to the bottom of the signal stack, resets the saved state to point to the signal handler and signal stack, and then exits the kernel handler. The `rett` instruction then resumes user-level execution at the signal handler. When the signal handler returns, these steps are unwound: the processor state is copied back from the signal handler into kernel memory, and the `rett` returns to the original computation.

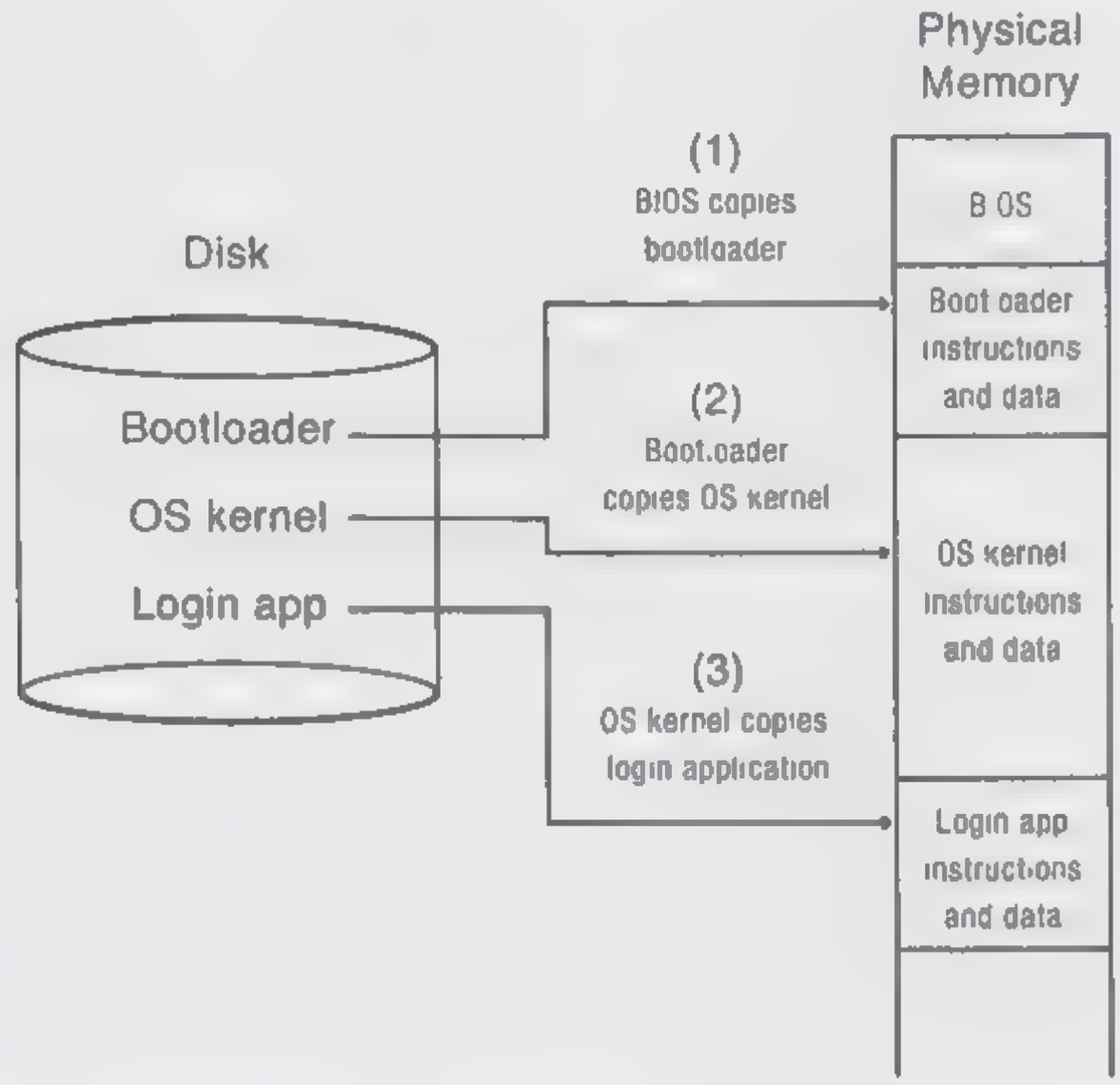

Figure 2.18: The boot ROM copies the bootloader image from disk into memory, and the bootloader copies the operating system kernel image from disk into memory

2.9 Case Study: Booting an Operating System Kernel

What steps are needed to start running an operating system kernel?

When a computer boots, it sets the machine's program counter to start executing at a pre determined position in memory. Since the computer is not yet running, the initial machine instructions must be fetched and executed immediately after the power is turned on before the system has had a chance to initialize its DRAM. Instead, systems typically use a special read-only hardware memory (*Boot ROM*) to store their boot instructions. On most x86 personal computers, the boot program is called the BIOS, for "Basic Input/Output System".

Boot ROM

What's not part of the kernel in ROM?

There are several drawbacks to trying to store the entire kernel in ROM. The most significant problem is that the operating system would be hard to update. ROM instructions are fixed when the computer is manufactured and (except in rare cases) are never changed. If an error occurs while the BIOS is being updated, the machine can be left in a permanently unusable state — unable to boot and unable to complete the update of the BIOS.

By contrast, operating systems need frequent updates, as bugs and security vulnerabilities are discovered and fixed. This, and the fact that ROM storage is relatively slow and expensive, argues for putting only a small amount of code in the BIOS.

Instead, the BIOS provides a level of indirection, as illustrated in Figure 2.18.

bootloader

The BIOS reads a fixed-size block of bytes from a fixed position on disk (or flash RAM) into memory. This block of bytes is called the *bootloader*. Once the BIOS has copied the bootloader into memory, it jumps to the first instruction in the block. On some newer machines, the BIOS also checks that the bootloader has not been corrupted by a computer virus. (If a virus could change the bootloader and get the BIOS to jump to it, the virus would then be in control of the machine.) As a check, the bootloader is stored with a *cryptographic signature*, a specially designed function of the bytes in a file and a private cryptographic key that allows someone with the corresponding public key to verify that an authorized entity produced the file. It is computationally intractable for an attacker without the private key to create a different file with a valid signature. The BIOS checks that the bootloader code matches the signature, verifying its authenticity.

cryptographic signature

The bootloader in turn loads the kernel into memory and jumps to it. Again, the bootloader can check the cryptographic signature of the operating system to verify that it has not been corrupted by a virus. The kernel's executable image is usually stored in the file system. Thus, to find the bootloader, the BIOS needs to read a block of raw bytes from disk, the bootloader, in turn, needs to know how to read from the file system to find and read the operating system image.

When the kernel starts running, it can initialize its data structures, including setting up the interrupt vector table to point to the various interrupt, processor exception, and system call handlers. The kernel then starts the first process, typically the user login page. To run this process, the operating system reads the code for the login program from its disk location, and jumps to the first instruction in the program, using the start process procedure described above. The login process in turn can trap into the kernel using a system call whenever it needs the kernel's services, e.g., to render the login prompt on the screen. We discuss the system calls needed for processes to do useful work in Chapter 3.

2.10 Case Study: Virtual Machines

Can an operating system run inside a process?

Some operating system kernels provide the abstraction of an entire virtual machine at user level. How do interrupts, processor exceptions, and system calls work in this context? To avoid confusion when discussing virtual machines, we need to recap some terminology introduced in Chapter 1. The operating system providing the virtual machine abstraction is called the *host operating system*. The operating system running inside the virtual machine is called the *guest operating system*.

host operating system

guest operating system

The host operating system provides the illusion that the guest kernel is running on real hardware. For example, to provide a guest disk, the host kernel simulates a virtual disk as a file on the physical disk. To provide network access to the guest kernel, the host kernel simulates a virtual network using physical network packets. Likewise, the host kernel must manage memory to provide

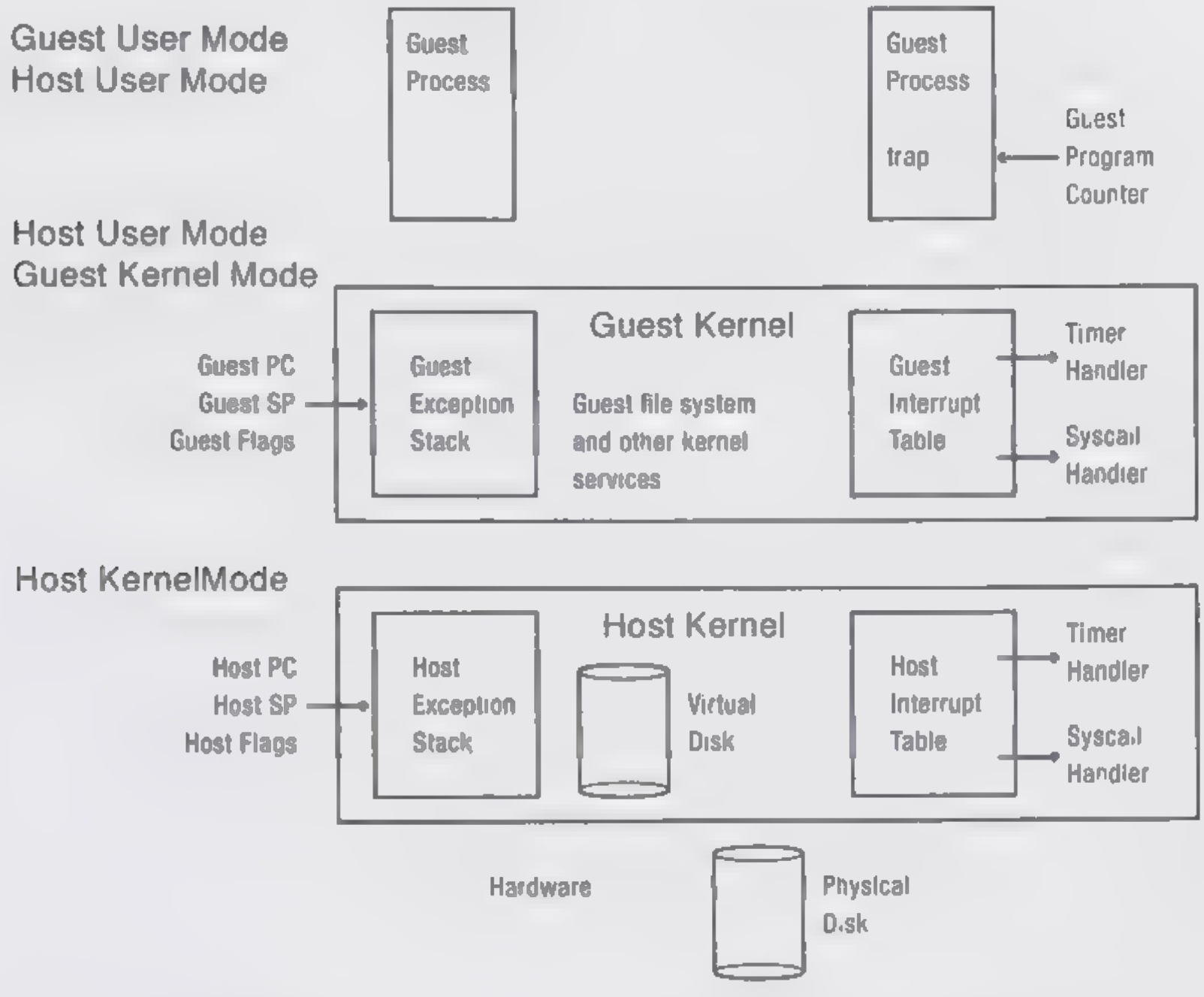

Figure 2.19: Emulation of user- and kernel-mode transfer for processes running inside a virtual machine. Both the guest kernel and the host kernel have their own copies of an interrupt vector table and interrupt stack. The guest vector table points to interrupt handlers in the guest kernel, the host vector table points to interrupt handlers in the host kernel

the illusion that the guest kernel is managing its own memory protection even though it is running with virtual addresses. We discuss address translation for virtual machines in more detail in Chapter 10.

How does the host kernel manage mode transfer between guest processes and the guest kernel? During boot, the host kernel initializes its interrupt vector table to point to its own interrupt handlers in host kernel memory. When the host kernel starts the virtual machine, the guest kernel starts running as if it is being booted:

1. The host loads the guest bootloader from the virtual disk and starts it running.
2. The guest bootloader loads the guest kernel from the virtual disk into memory and starts it running.
3. The guest kernel then initializes its interrupt vector table to point to the guest interrupt handlers.

4 The guest kernel loads a process from the virtual disk into guest memory.

5. To start a process, the guest kernel issues instructions to resume execution at user level, e g., using reti on the x86. Since changing the privilege level is a privileged operation, this instruction traps into the host kernel.

6. The host kernel simulates the requested mode transfer as if the processor had directly executed it. It restores the program counter, stack pointer, and processor status word exactly as the guest operating system had intended. Note that the host kernel must protect itself from bugs in the guest operating system, and so it also must check the validity of the mode transfer — to ensure that the guest kernel is not surreptitiously attempting to get the host kernel to "switch" to an arbitrary point in the kernel code.

Next, consider what happens when the guest user process does a system call, illustrated in Figure 2.19. To the hardware, there is only one kernel, the host operating system. Thus, the trap instruction traps into the host kernel's system call handler. Of course, the system call was not intended for the host! Rather, the host kernel simulates what would have happened had the system call instruction occurred on real hardware running the guest operating system:

1. The host kernel saves the instruction counter, processor status register, and user stack pointer on the interrupt stack of the guest operating system.

2. The host kernel transfers control to the guest kernel at the beginning of the interrupt handler, but with the guest kernel running with user mode privilege.

3. The guest kernel performs the system call — saving user state and checking arguments.

4. When the guest kernel attempts to return from the system call back to user level, this causes a processor exception, dropping back into the host kernel.

5 The host kernel can then restore the state of the user process, running at user level, as if the guest OS had been able to return there directly.

The host kernel handles processor exceptions similarly, with one caveat. Some exceptions generated by the virtual machine are due to the user process, the host kernel forwards these to the guest kernel for handling. Other exceptions are generated by the guest kernel (e g., when it tries to execute privileged instructions), the host kernel simulates these itself. Thus, the host kernel must track whether the virtual machine is executing in virtual user mode or virtual kernel mode.

The hardware vectors interrupts to the host kernel. Timer interrupts need special handling, as time can elapse in the host without elapsing in the guest.

When a timer interrupt occurs, enough virtual time may have passed that the guest kernel is due a timer interrupt. If so, the host kernel returns from the interrupt to the interrupt handler for the guest kernel. The guest kernel may in turn switch guest processes; its reti will cause a processor exception, returning to the host kernel, which can then resume the correct guest process.

Handling I/O interrupts is simpler: the simulation of the virtual device does not need to be anything like a real device. When the guest kernel makes a request to a virtual disk, the kernel writes instructions to the buffer descriptor ring for the disk device, the host kernel translates these instructions into operations on the virtual disk. The host kernel can simulate the disk request however it likes—e.g., through regular file reads and writes, copied into the guest kernel memory as if there was true DMA hardware. The guest kernel expects to receive an interrupt when the virtual disk completes its work; this can be triggered by the timer interrupt, but vectored to the guest disk interrupt handler instead of the guest timer interrupt handler.

2.11 Summary and Future Directions

The process concept—the ability to execute arbitrary user programs with restricted rights—has been remarkably successful. With the exception of devices that run only a single application at a time (such as embedded systems and game consoles), every commercially successful operating system provides process isolation.

The reason for this success is obvious. Without process isolation, computer systems would be much more fragile and less secure. As recently as a decade ago, it was common for personal computers to crash on a daily basis. Today, laptops can remain working for weeks at a time without rebooting. This has occurred even though the operating system and application software on these systems have become more complex. While some of the improvement is due to better hardware reliability and automated bug tracking, process isolation has been a key technology in constructing more reliable system software.

Process isolation is also essential to building more secure computer systems. Without isolation, computer users would be forced to trust every application loaded onto the computer, not just the operating system code. In practice, however, complete process isolation remains more an aspiration than a reality. Most operating systems are vulnerable to malicious applications because the attacker can exploit any vulnerability in the kernel implementation. Even a single bug in the kernel can leave the system vulnerable. Keeping your system current with the latest patches provides some level of defense, but it is still inadvisable to download and install untrusted software off the web.

In the future, we are likely to see three complementary trends:

- **Operating system support for fine-grained protection.** Process isolation is becoming more flexible and fine grained in order to reflect different levels of trust in different applications. Today, it is typical for a user

Hardware support for operating systems

We have described a number of hardware mechanisms that support operating systems:

- **Privilege levels**, user and kernel.
- **Privileged instructions**, instructions available only in kernel mode.
- **Memory translation** prevents user programs from accessing kernel data structures and aids in memory management.
- **Processor exceptions** trap to the kernel on a privilege violation or other unexpected event.
- **Timer interrupts** return control to the kernel on time expiration.
- **Device interrupts** return control to the kernel to signal I/O completion
- **Interprocessor interrupts** cause another processor to return control to the kernel.
- **Interrupt masking** prevents interrupts from being delivered at inopportune times.
- **System calls** trap to the kernel to perform a privileged action on behalf of a user program.
- **Return from interrupt**, switch from kernel mode to user mode, to a specific location in a user process.
- **Boot ROM** code that loads startup routines from disk into memory

To support threads, we will need one additional mechanism, described in Chapter 5:

- **Atomic read-modify-write instructions** used to implement synchronization in multi-threaded programs.

---

application to have the permissions of that user. This allows a virus masquerading as a screen saver to steal or corrupt that user's data without needing to compromise the operating system first. Smartphone operating systems have started to add better controls to prevent certain applications without a "need to know" from accessing sensitive information, such as the smartphone's location or the list of frequently called telephone numbers.

- **Application-layer sandboxing.** Increasingly, many applications are be-

coming mini-operating systems in their own right, capable of safely executing third-party software to extend and improve the user experience. Sophisticated scripts customize web site behavior; web browsers must efficiently and completely isolate these scripts so that they cannot steal the user's data or corrupt the browser. Other applications—such as databases and desktop publishing systems—are also moving in the direction of needing application-layer sandboxing. Google's Native Client and Microsoft's Application Domains are two example systems that provide general-purpose, safe execution of third-party code at the user level.

- **Hardware support for virtualization.** Virtual machines provide an extra layer of protection beneath the operating system. Even if a malicious process run by a guest operating system on a virtual machine were able to corrupt the kernel, its impact would be limited to just that virtual machine. Below the virtual machine interface, the host operating system needs to provide isolation between different virtual machines; this is much easier in practice because the virtual machine interface is much simpler than the kernel's system call interface. For example, in a data center, virtual machines provide users with the flexibility to run any application without compromising data center operation.

  Computer manufacturers are re-designing processor architectures to reduce the cost of running a virtual machine. For example, on some new processors, guest operating systems can directly handle their own interrupts, processor exceptions, and system calls without those events needing to be mediated by the host operating system. Likewise, I/O device manufacturers are re-designing their interfaces to do direct transfers to and from the guest operating system without the need to go through the host kernel.

Exercises

1. When a user process is interrupted or causes a processor exception, the x86 hardware switches the stack pointer to a kernel stack, before saving the current process state. Explain why.

2. For the "Hello world" program, we mentioned that the kernel must copy the string from the user program to screen memory. Why must the screen's buffer memory be protected? Explain what might happen if a malicious application could alter any pixel on the screen, not just those within its own window.

3. For each of the three mechanisms that supports dual-mode operation—privileged instructions, memory protection, and timer interrupts—explain what might go wrong without that mechanism, assuming the system still had the other two.

4. Suppose you are tasked with designing the security system for a new web browser that supports rendering web pages with embedded web page scripts. What checks would you need to implement to ensure that executing buggy or malicious scripts could not corrupt or crash the browser?

5. Define three types of user-mode to kernel-mode transfers.

6. Define four types of kernel-mode to user-mode transfers.

7. Most hardware architectures provide an instruction to return from an interrupt, such as iret. This instruction switches the mode of operation from kernel-mode to user-mode.

   a) Explain where in the operating system this instruction would be used.

   b) Explain what happens if an application program executes this instruction.

8. A hardware designer argues that there is now enough on-chip transistors to provide 1024 integer registers and 512 floating point registers. As a result, the compiler should almost never need to store anything on the stack. As an operating system guru, give your opinion of this design.

   a) What is the effect on the operating system of having a large number of registers?

   b) What hardware features would you recommend adding to the design?

   c) What happens if the hardware designer also wants to add a 16-stage pipeline into the CPU, with precise exceptions. How would that affect the user-kernel switching overhead?

9. With virtual machines, the host kernel runs in privileged mode to create a virtual machine that runs in user mode. The virtual machine provides the illusion that the guest kernel runs on its own machine in privileged mode, even though it is actually running in user mode.

   Early versions of the x86 architecture (pre-2006) were not *completely virtualizable*—these systems could not guarantee to run unmodified guest operating systems properly. One problem was the popf "pop flags" instruction that restores the processor status word. When popf was run in privileged mode, it changed both the ALU flags (e g., the condition codes) and the systems flags (e g , the interrupt mask) When popf was run in unprivileged mode, it changed just the ALU flags.

   a) Why do instructions like popf prevent transparent virtualization of the (old) x86 architecture?

   b) How would you change the (old) x86 hardware to fix this problem?

10 Which of the following components is responsible for loading the initial value in the program counter for an application program before it starts running: the compiler, the linker, the kernel, or the boot ROM?

11 We described how the operating system kernel mediates access to I/O devices for safety. Some newer I/O devices are *virtualizable* — they permit safe access from user-level programs, such as a guest operating system running in a virtual machine. Explain how you might design the hardware and software to get this to work (Hint: The device needs much of the same hardware support as the operating system kernel.)

12. System calls vs. procedure calls: How much more expensive is a system call than a procedure call? Write a simple test program to compare the cost of a simple procedure call to a simple system call (getpid() is a good candidate on UNIX, see the man page). To prevent the optimizing compiler from "optimizing out' your procedure calls, do not compile with optimization on You should use a system call such as the UNIX gettimeofday() for time measurements. Design your code so the measurement overhead is negligible Also, be aware that timer values in some systems have limited resolution (e.g., millisecond resolution).

    Explain the difference (if any) between the time required by your simple procedure call and simple system call by discussing what work each call must do.

13. Suppose you have to implement an operating system on hardware that supports interrupts and exceptions but does not have a trap instruction. Can you devise a satisfactory substitute for traps using interrupts and/or exceptions? If so, explain how. If not, explain why.

14. Suppose you have to implement an operating system on hardware that supports exceptions and traps but does not have interrupts. Can you devise a satisfactory substitute for interrupts using exceptions and/or traps? If so, explain how. If not, explain why.

15. Explain the steps that an operating system goes through when the CPU receives an interrupt.

16. When an operating system receives a system call from a program, a switch to operating system code occurs with the help of the hardware. The hardware sets the mode of operation to kernel mode, calls the operating system trap handler at a location specified by the operating system, and lets the operating system return to user mode after it finishes its trap handling.

    Consider the stack on which the operating system must run when it receives the system call. Should this stack be different from the one the application uses, or could it use the same stack as the application program? Assume that the application program is blocked while the system call runs.

17. Write a program to verify that the operating system on your computer correctly protects itself from rogue system calls. For a single system call—such as file system open—try all possible illegal calls: e.g., an invalid system call number, an invalid stack pointer, an invalid pointer stored on the stack, etc. What happens?