Chapter 8

Exceptional Control Flow

From the time you first apply power to a processor until the time you shut it off, the program counter assumes a sequence of values

$$a_0, a_1, \ldots, a_{n-1}$$

where each a_k is the address of some corresponding instruction I_k. Each transition from a_k to a_{k+1} is called a *control transfer*. A sequence of such control transfers is called the *flow of control*, or *control flow* of the processor.

The simplest kind of control flow is a "smooth" sequence where each I_k and I_{k+1} are adjacent in memory. Typically, abrupt changes to this smooth flow, where I_{k+1} is not adjacent to I_k, are caused by familiar program instructions such as jumps, calls, and returns. Such instructions are necessary mechanisms that allow programs to react to changes in internal program state represented by program variables.

But systems must also be able to react to changes in system state that are not captured by internal program variables and are not necessarily related to the execution of the program. For example, a hardware timer goes off at regular intervals and must be dealt with. Packets arrive at the network adapter and must be stored in memory. Programs request data from a disk and then sleep until they are notified that the data are ready. Parent processes that create child processes must be notified when their children terminate.

Modern systems react to these situations by making abrupt changes in the control flow. In general, we refer to these abrupt changes as *exceptional control flow* (ECF). Exceptional control flow occurs at all levels of a computer system. For example, at the hardware level, events detected by the hardware trigger abrupt control transfers to exception handlers. At the operating systems level, the kernel transfers control from one user process to another via context switches. At the application level, a process can send a *signal* to another process that abruptly transfers control to a signal handler in the recipient. An individual program can react to errors by sidestepping the usual stack discipline and making nonlocal jumps to arbitrary locations in other functions.

As programmers, there are a number of reasons why it is important for you to understand ECF:

- *Understanding ECF will help you understand important systems concepts.* ECF is the basic mechanism that operating systems use to implement I/O, processes, and virtual memory. Before you can really understand these important ideas, you need to understand ECF.

- *Understanding ECF will help you understand how applications interact with the operating system.* Applications request services from the operating system by using a form of ECF known as a *trap* or *system call*. For example, writing data to a disk, reading data from a network, creating a new process, and terminating the current process are all accomplished by application programs making system calls. Understanding the basic system call mechanism will help you understand how these services are provided to applications.

- *Understanding ECF will help you write interesting new application programs.* The operating system provides application programs with powerful ECF mechanisms for creating new processes, waiting for processes to terminate, notifying other processes of exceptional events in the system, and detecting and responding to these events. If you understand these ECF mechanisms, then you can use them to write interesting programs such as Unix shells and Web servers.

- *Understanding ECF will help you understand how software exceptions work.* Languages such as C++ and Java provide software exception mechanisms via `try`, `catch`, and `throw` statements. Software exceptions allow the program to make *nonlocal* jumps (i.e., jumps that violate the usual call/return stack discipline) in response to error conditions. Nonlocal jumps are a form of application-level ECF, and are provided in C via the `setjmp` and `longjmp` functions. Understanding these low level functions will help you understand how higher level software exceptions can be implemented.

Up to this point in your study of systems, you have learned how applications interact with the hardware. This chapter is pivotal in the sense that you will begin to learn how your applications interact with the operating system. Interestingly, these interactions all revolve around ECF. We describe the various forms of ECF that exist at all levels of a computer system. We start with exceptions, which lie at the intersection of the hardware and the operating system. We also discuss system calls, which are exceptions that provide applications with entry points into the operating system. We then move up a level of abstraction and describe processes and signals, which lie at the intersection of applications and the operating system. Finally, we discuss nonlocal jumps, which are an application-level form of ECF.

8.1 Exceptions

Exceptions are a form of exceptional control flow that are implemented partly by the hardware and partly by the operating system. Because they are partly implemented in hardware, the details vary from system to system. However, the basic ideas are the same for every system. Our aim in this section is to give you a general understanding of exceptions and exception handling, and to help demystify what is often a confusing aspect of modern computer systems.

An *exception* is an abrupt change in the control flow in response to some change in the processor's state. Figure 8.1 shows the basic idea. In the figure, the processor is executing some current instruction I_{curr} when a significant change in the processor's *state* occurs. The state is encoded in various bits and signals inside the processor. The change in state is known as an *event*. The event might be directly related to the execution of the current instruction. For example, a virtual memory page fault occurs, an arithmetic overflow occurs, or an instruction attempts a divide by zero. On the other hand, the event might be unrelated to the execution of the current instruction. For example, a system timer goes off or an I/O request completes.

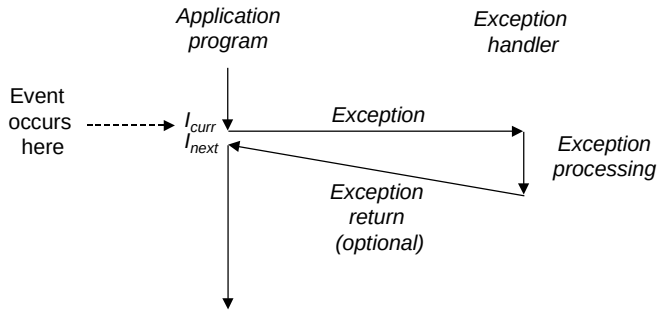


Figure 8.1: **Anatomy of an exception.** A change in the processor's state (event) triggers an abrupt control transfer (an exception) from the application program to an exception handler. After it finishes processing, the handler either returns control to the interrupted program or aborts.

In any case, when the processor detects that the event has occurred, it makes an indirect procedure call (the exception), through a jump table called an *exception table*, to an operating system subroutine (the *exception handler*) that is specifically designed to process this particular kind of event.

When the exception handler finishes processing, one of three things happens, depending on the type of event that caused the exception:

1. The handler returns control to the current instruction I_{curr}, the instruction that was executing when the event occurred.
2. The handler returns control to I_{next}, the instruction that would have executed next had the exception not occurred.
3. The handler aborts the interrupted program.

Section 8.1.2 says more about these possibilities.

Aside: Hardware vs. software exceptions.
C++ and Java programmers will have noticed that the term "exception" is also used to describe the application-level ECF mechanism provided by C++ and Java in the form of `catch`, `throw`, and `try` statements. If we wanted to be perfectly clear, we might distinguish between "hardware" and "software" exceptions, but this is usually unnecessary because the meaning is clear from the context. **End Aside.**

8.1.1 Exception Handling

Exceptions can be difficult to understand because handling them involves close cooperation between hardware and software. It is easy to get confused about which component performs which task. Let's look at the division of labor between hardware and software in more detail.

Each type of possible exception in a system is assigned a unique nonnegative integer *exception number*. Some of these numbers are assigned by the designers of the processor. Other numbers are assigned by the designers of the operating system *kernel* (the memory-resident part of the operating system). Examples of the former include divide by zero, page faults, memory access violations, breakpoints, and arithmetic overflows. Examples of the latter include system calls and signals from external I/O devices.

At system boot time (when the computer is reset or powered on) the operating system allocates and initializes a jump table called an *exception table*, so that entry k contains the address of the handler for exception k. Figure 8.2 shows the format of an exception table.

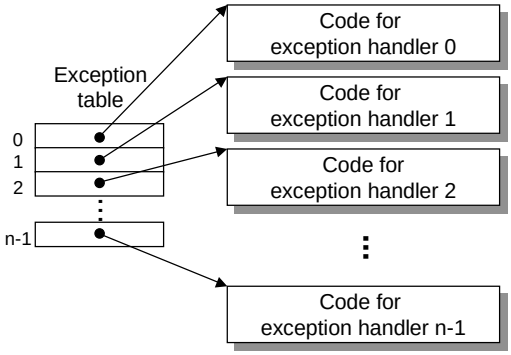


Figure 8.2: **Exception table.** The exception table is a jump table where entry k contains the address of the handler code for exception k.

At run time (when the system is executing some program), the processor detects that an event has occurred and determines the corresponding exception number k. The processor then triggers the exception by making an indirect procedure call, through entry k of the exception table, to the corresponding handler. Figure 8.3 shows how the processor uses the exception table to form the address of the appropriate exception handler. The exception number is an index into the exception table, whose starting address is contained in a special CPU register called the *exception table base register*.

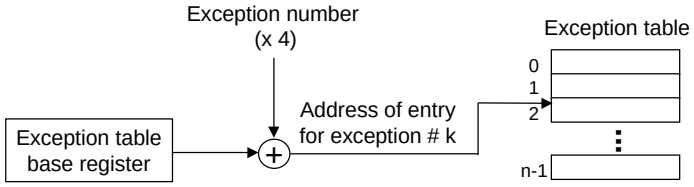


Figure 8.3: **Generating the address of an exception handler.** The exception number is an index into the exception table.

An exception is akin to a procedure call, but with some important differences.

- As with a procedure call, the processor pushes a return address on the stack before branching to the handler. However, depending on the class of exception, the return address is either the current instruction (the instruction that was executing when the event occurred) or the next instruction (the instruction that would have executed after the current instruction had the event not occurred).
- The processor also pushes some additional processor state onto the stack that will be necessary to restart the interrupted program when the handler returns. For example, an IA32 system pushes the EFLAGS register containing, among other things, the current condition codes, onto the stack.
- If control is being transferred from a user program to the kernel, all of these items are pushed onto the kernel's stack rather than onto the user's stack.

- Exception handlers run in *kernel mode* (Section 8.2.3, which means they have complete access to all system resources.

Once the hardware triggers the exception, the rest of the work is done in software by the exception handler. After the handler has processed the event, it optionally returns to the interrupted program by executing a special "return from interrupt" instruction, which pops the appropriate state back into the processor's control and data registers, restores the state to *user mode* (Section 8.2.3) if the exeption interrupted a user program, and then returns control to the interrupted program.

8.1.2 Classes of Exceptions

Exceptions can be divided into four classes: *interrupts*, *traps*, *faults*, and *aborts*. The table in Figure 8.4 summarizes the attributes of these classes.

Class	Cause	Async/Sync	Return behavior
Interrupt	Signal from I/O device	Async	Always returns to next instruction
Trap	Intentional exception	Sync	Always returns to next instruction
Fault	Potentially recoverable error	Sync	Might return to current instruction
Abort	Nonrecoverable error	Sync	Never returns

Figure 8.4: **Classes of exceptions.** Asynchronous exceptions occur as a result of events in I/O devices that are external to the processor. Synchronous exceptions occur as a direct result of executing an instruction.

Interrupts

Interrupts occur *asynchronously* as a result of signals from I/O devices that are external to the processor. Hardware interrupts are asynchronous in the sense that they are not caused by the execution of any particular instruction. Exception handlers for hardware interrupts are often called *interrupt handlers*.

Figure 8.5 summarizes the processing for an interrupt. I/O devices such as network adapters, disk controllers, and timer chips trigger interrupts by signalling a pin on the processor chip and placing the exception number on the system bus that identifies the device that caused the interrupt.

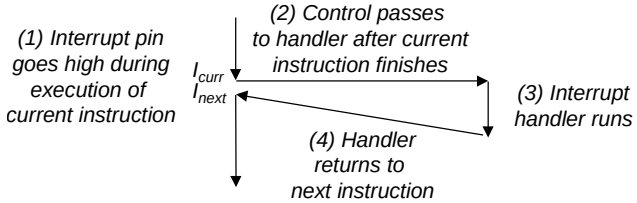


Figure 8.5: **Interrupt handling.** The interrupt handler returns control to the next instruction in the application program's control flow.

After the current instruction finishes executing, the processor notices that the interrupt pin has gone high, reads the exception number from the system bus, and then calls the appropriate interrupt handler. When

the handler returns, it returns control to the next instruction (i.e., the instruction that would have followed the current instruction in the control flow had the interrupt not occurred). The effect is that the program continues executing as though the interrupt had never happened.

The remaining classes of exceptions (traps, faults, and aborts) occur *synchronously* as a result of executing the current instruction. We refer to this instruction as the *faulting instruction*.

Traps

Traps are *intentional* exceptions that occur as a result of executing an instruction. Like interrupt handlers, trap handlers return control to the next instruction. The most important use of traps is to provide a procedure-like interface between user programs and the kernel known as a *system call*.

User programs often need to request services from the kernel such as reading a file (`read`), creating a new process (`fork`), loading a new program (`execve`), or terminating the current process (`exit`). To allow controlled access to such kernel services, processors provide a special "`syscall` n" instruction that user programs can execute when they want to request service n. Executing the `syscall` instruction causes a trap to an exception handler that decodes the argument and calls the appropriate kernel routine. Figure 8.6 summarizes the processing for a system call. From a programmer's perspective, a system call is identical

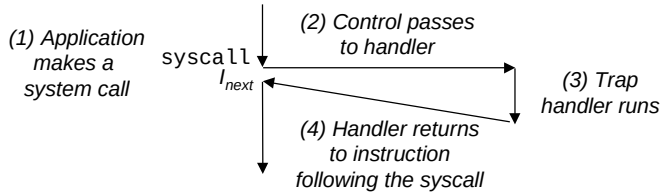


Figure 8.6: **Trap handling.** The trap handler returns control to the next instruction in the application program's control flow.

to a regular function call. However, their implementations are quite different. Regular functions run in *user mode*, which restricts the types of instructions they can execute, and they access the same stack as the calling function. A system call runs in *kernel mode*, which allows it to execute instructions, and accesses a stack defined in the kernel. Section 8.2.3 discusses user and kernel modes in more detail.

Faults

Faults result from error conditions that a handler might be able to correct. When a fault occurs, the processor transfers control to the fault handler. If the handler is able to correct the error condition, it returns control to the faulting instruction, thereby reexecuting it. Otherwise, the handler returns to an `abort` routine in the kernel that terminates the application program that caused the fault. Figure 8.7 summarizes the processing for a fault.

A classic example of a fault is the page fault exception, which occurs when an instruction references a virtual address whose corresponding physical page is not resident in memory and must therefore be retrieved from disk. As we will see in Chapter 10, a page is a contiguous block (typically 4 KB) of virtual memory. The

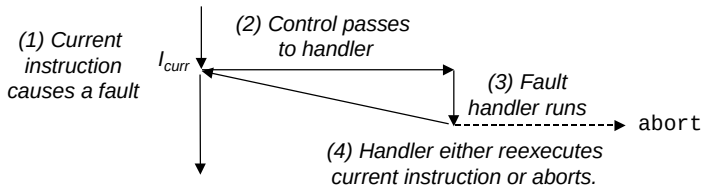


Figure 8.7: **Fault handling.** Depending on the whether the fault can be repaired or not, the fault handler either re-executes the faulting instruction or aborts.

page fault handler loads the appropriate page from disk and then returns control to the instruction that caused the fault. When the instruction executes again, the appropriate physical page is resident in memory and the instruction is able to run to completion without faulting.

Aborts

Aborts result from unrecoverable fatal errors, typically hardware errors such as parity errors that occur when DRAM or SRAM bits are corrupted. Abort handlers never return control to the application program. As shown in Figure 8.8, the handler returns control to an `abort` routine that terminates the application program.

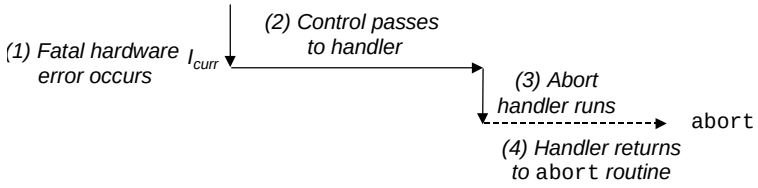


Figure 8.8: **Abort handling.** The abort handler passes control to a kernel `abort` routine that terminates the application program.

8.1.3 Exceptions in Intel Processors

To help make things more concrete, let's look at some of the exceptions defined for Intel systems. A Pentium system can have up to 256 different exception types. Numbers in the range from 0 to 31 correspond to exceptions that are defined by the Pentium architecture, and thus are identical for any Pentium-class system. Numbers in the range from 32 to 255 correspond to interrupts and traps that are defined by the operating system. Figure 8.9 shows a few examples.

A divide error (exception 0) occurs when an application attempts to divide by zero, or when the result of a divide instruction is too big for the destination operand. Unix does not attempt to recover from divide errors, opting instead to abort the program. Unix shells typically report divide errors as "Floating exceptions".

The infamous general protection fault (exception 13) occurs for many reasons, usually because a program references an undefined area of virtual memory, or because the program attempts to write to a read-only text

Exception Number	Description	Exception Class
0	Divide error	Fault
13	General protection fault	Fault
14	Page fault	Fault
18	Machine check	Abort
32–127	OS-defined exceptions	Interrupt or trap
128 (0x80)	System call	Trap
129–255	OS-defined exceptions	Iinterrupt or trap

Figure 8.9: **Examples of exceptions in Pentium systems.**

segment. Unix does not attempt to recover from this fault. Unix shells typically report general protection faults as "Segmentation faults".

A page fault (exception 14) is an example of an exception where the faulting instruction is restarted. The handler maps the appropriate page of physical memory on disk into a page of virtual memory, and then restarts the faulting instruction. We will see how this works in detail in Chapter 10.

A machine check (exception 18) occurs as a result of a fatal hardware error that is detected during the execution of the faulting instruction. Machine check handlers never return control to the application program.

System calls are provided on IA32 systems via a trapping instruction called `INT` n, where n can be the index of any of the 256 entries in the exception table. Historically, systems calls are provided through exception 128 (`0x80`).

Aside: A note on terminology.
The terminology for the various classes of exceptions varies from system to system. Processor macroarchitecture specifications often distinguish between asynchronous "interrupts" and synchronous "exceptions," yet provide no umbrella term to refer to these very similar concepts. To avoid having to constantly refer to "exceptions and interrupts" and "exceptions or interrupts," we use the word "exception" as the general term and distinguish between asynchronous exceptions (interrupts) and synchronous exceptions (traps, faults, and aborts) only when it is appropriate. As we have noted, the basic ideas are the same for every system, but you should be aware that some manufacturers' manuals use the word "exception" to refer only to those changes in control flow caused by synchronous events. **End Aside.**

8.2 Processes

Exceptions provide the basic building blocks that allow the operating system to provide the notion of a *process*, one of the most profound and successful ideas in computer science.

When we run a program on a modern system, we are presented with the illusion that our program is the only one currently running in the system. Our program appears to have exclusive use of both the processor and the memory. The processor appears to execute the instructions in our program, one after the other, without interruption. Finally, the code and data of our program appear to be the only objects in the system's memory. These illusions are provided to us by the notion of a process.

The classic definition of a process is *an instance of a program in execution*. Each program in the system runs in the *context* of some process. The context consists of the state that the program needs to run correctly. This

state includes the program's code and data stored in memory, its stack, the contents of its general-purpose registers, its program counter, environment variables, and the set of open file descriptors.

Each time a user runs a program by typing the name of an executable object file to the shell, the shell creates a new process and then runs the executable object file in the context of this new process. Application programs can also create new processes and run either their own code or other applications in the context of the new process.

A detailed discussion of how operating systems implement processes is beyond our scope. Instead, we will focus on the key abstractions that a process provides to the application:

- An independent *logical control flow* that provides the illusion that our program has exclusive use of the processor.
- A private address space that provides the illusion that our program has exclusive use of the memory system.

Let's look more closely at these abstractions.

8.2.1 Logical Control Flow

A process provides each program with the illusion that it has exclusive use of the processor, even though many other programs are typically running on the system. If we were to use a debugger to single step the execution of our program, we would observe a series of program counter (PC) values that corresponded exclusively to instructions contained in our program's executable object file or in shared objects linked into our program dynamically at run time. This sequence of PC values is known as a *logical control flow*.

Consider a system that runs three processes, as shown in Figure 8.10. The single physical control flow of the processor is partitioned into three *logical flows*, one for each process. Each vertical line represents a portion of the logical flow for a process. In the example, process A runs for a while, followed by B, which runs to completion. C then runs for awhile, followed by A, which runs to completion. Finally, C is able to run to completion.

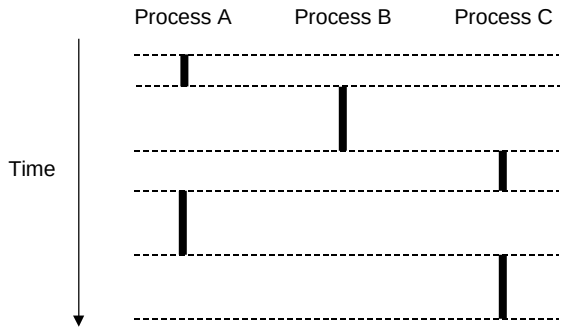


Figure 8.10: **Logical control flows.** Processes provide each program with the illusion that it has exclusive use of the processor. Each vertical bar represents a portion of the logical control flow for a process.

The key point in Figure 8.10 is that processes take turns using the processor. Each process executes a

portion of its flow and then is *preempted* (temporarily suspended) while other processes take their turns. To a program running in the context of one of these processes, it appears to have exclusive use of the processor. The only evidence to the contrary is that if we were to precisely measure the elapsed time of each instruction (see Chapter 9), we would notice that the CPU appears to periodically stall between the execution of some of the instructions in our program. However, each time the processor stalls, it subsequently resumes execution of our program without any change to the contents of the program's memory locations or registers.

In general, each logical flow is independent of any other flow in the sense that the logical flows associated with different processes do not affect the states of any other processes. The only exception to this rule occurs when processes use interprocess communication (IPC) mechanisms such as pipes, sockets, shared memory, and semaphores to explicitly interact with each other.

Any process whose logical flow overlaps in time with another flow is called a *concurrent process*, and the two processes are said to run *concurrently*. For example, in Figure 8.10, processes A and B run concurrently, as do A and C. On the other hand, B and C do not run concurrently because the last instruction of B executes before the first instruction of C.

The notion of processes taking turns with other processes is known as *multitasking*. Each time period that a process executes a portion of its flow is called a *time slice*. Thus, multitasking is also referred to as *time slicing*.

8.2.2 Private Address Space

A process also provides each program with the illusion that it has exclusive use of the system's address space. On a machine with n-bit addresses, the *address space* is the set of 2^n possible addresses, $0, 1, \ldots, 2^n - 1$. A process provides each program with its own *private address space*. This space is private in the sense that a byte of memory associated with a particular address in the space cannot in general be read or written by any other process.

Although the contents of the memory associated with each private address space is different in general, each such space has the same general organization. For example, Figure 8.11 shows the organization of the address space for a Linux process. The bottom three-fourths of the address space is reserved for the user program, with the usual text, data, heap, and stack segments. The top quarter of the address space is reserved for the kernel. This portion of the address space contains the code, data, and stack that the kernel uses when it executes instructions on behalf of the process (e.g., when the application program executes a system call).

8.2.3 User and Kernel Modes

In order for the operating system kernel to provide an airtight process abstraction, the processor must provide a mechanism that restricts the instructions that an application can execute, as well as the portions of the address space that it can access.

Processors typically provide this capability with a *mode bit* in some control register that characterizes the privileges that the process currently enjoys. When the mode bit is set, the process is running in *kernel mode* (sometimes called *supervisor mode*). A process running in kernel mode can execute any instruction in the

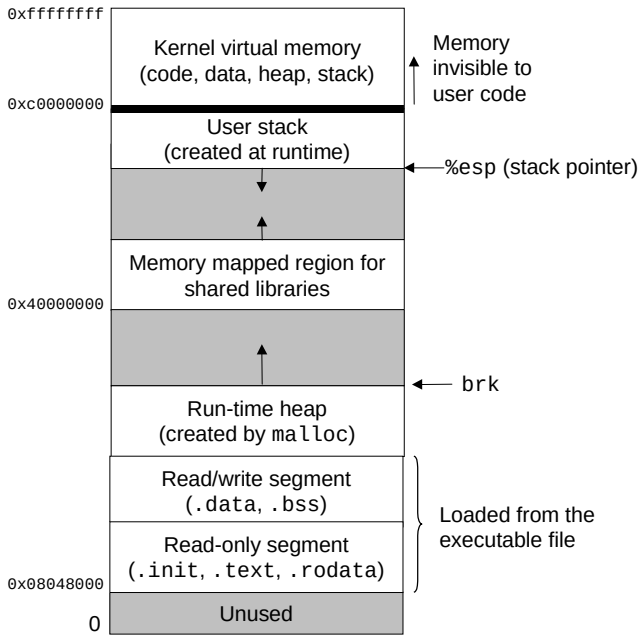


Figure 8.11: **Process address space.**

instruction set and access any memory location in the system.

When the mode bit is not set, the process is running in *user mode*. A process in user mode is not allowed to execute *privileged instructions* that do things such as halt the processor, change the mode bit, or initiate an I/O operation. Nor is it allowed to directly reference code or data in the kernel area of the address space. Any such attempt results in a fatal protection fault. User programs must instead access kernel code and data indirectly via the system call interface.

A process running application code is initially in user mode. The only way for the process to change from user mode to kernel mode is via an exception such as an interrupt, a fault, or a trapping system call. When the exception occurs, and control passes to the exception handler, the processor changes the mode from user mode to kernel mode. The handler runs in kernel mode. When it returns to the application code, the processor changes the mode from kernel mode back to user mode.

Linux and Solaris provides a clever mechanism, called the `/proc` filesystem, that allows user mode processes to access the contents of kernel data structures. The `/proc` filesystem exports the contents of many kernel data structures as a hierarchy of ASCII files that can read by user programs. For example, you can use the Linux `proc` filesystem to find out general system attributes such as CPU type (`/proc/cpuinfo`), or the memory segments used by a particular process (`/proc/<process id>/maps`).

8.2.4 Context Switches

The operating system kernel implements multitasking using a higher level form of exceptional control flow known as a *context switch*. The context switch mechanism is built on top of the lower level exception mechanism that we discussed in Section 8.1.

The kernel maintains a *context* for each process. The context is the state that the kernel needs to restart a preempted process. It consists of the values of objects such as the general purpose registers, the floating-point registers, the program counter, user's stack, status registers, kernel's stack, and various kernel data structures such as a *page table* that characterizes the address space, a *process table* that contains information about the current process, and a *file table* that contains information about the files that the process has opened.

At certain points during the execution of a process, the kernel can decide to preempt the current process and restart a previously preempted process. This decision is known as *scheduling* and is handled by code in the kernel called the *scheduler*. When the kernel selects a new process to run, we say that the kernel has *scheduled* that process. After the kernel has scheduled a new process to run, it preempts the current process and transfers control to the new process using a mechanism called a *context switch* that (1) saves the context of the current process, (2) restores the saved context of some previously preempted process, and (3) passes control to this newly restored process.

A context switch can occur while the kernel is executing a system call on behalf of the user. If the system call blocks because it is waiting for some event to occur, then the kernel can put the current process to sleep and switch to another process. For example, if a `read` system call requires a disk access, the kernel can opt to perform a context switch and run another process instead of waiting for the data to arrive from the disk. Another example is the `sleep` system call, which is an explicit request to put the calling process to sleep. In general, even if a system call does not block, the kernel can decide to perform a context switch rather than return control to the calling process.

A context switch can also occur as a result of an interrupt. For example, all systems have some mechanism for generating periodic timer interrupts, typically every 1 ms or 10 ms. Each time a timer interrupt occurs, the kernel can decide that the current process has run long enough and switch to a new process.

Figure 8.12 shows an example of context switching between a pair of processes A and B. In this example, initially process A is running in user mode until it traps to the kernel by executing a `read` system call. The trap handler in the kernel requests a DMA transfer from the disk controller and arranges for the disk to interrupt the processor after the disk controller has finished transferring the data from disk to memory.

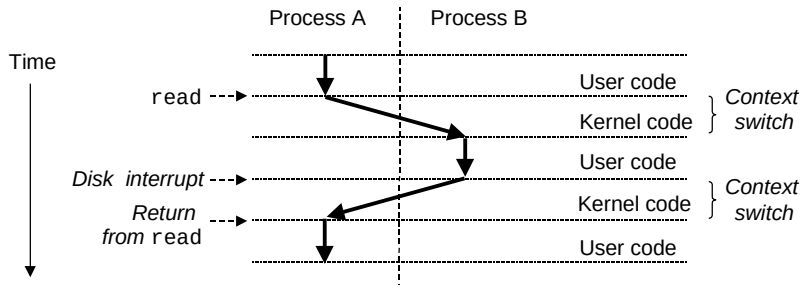


Figure 8.12: **Anatomy of a process context switch.**

The disk will take a relatively long time to fetch the data (on the order of tens of milliseconds), so instead of waiting and doing nothing in the interim, the kernel performs a context switch from process A to B. Note that before the switch, the kernel is executing instructions in user mode on behalf of process A. During the first part of the switch, the kernel is executing instructions in kernel mode on behalf of process A. Then

at some point it begins executing instructions (still in kernel mode) on behalf of process B. And after the switch, the kernel is executing instructions in user mode on behalf of process B.

Process B then runs for a while in user mode until the disk sends an interrupt to signal that data has been transferred from disk to memory. The kernel decides that process B has run long enough and performs a context switch from process B to A, returning control in process A to the instruction immediately following the `read` system call. Process A continues to run until the next exception occurs, and so on.

Aside: Cache pollution and exceptional control flow.
In general, hardware cache memories do not interact well with exceptional control flows such as interrupts and context switches. If the current process is interrupted briefly by an interrupt, then the cache is cold for the interrupt handler. If the handler accesses enough items from main memory, then the cache will also be cold for the interrupted process when it resumes. In this case we say that the handler has *polluted* the cache. A similar phenomenon occurs with context switches. When a process resumes after a context switch, the cache is cold for the application program and must be warmed up again. **End Aside.**

8.3 System Calls and Error Handling

Unix systems provide a variety of systems calls that application programs use when they want to request services from the kernel, such as reading a file or creating a new process. For example, Linux provides about 160 system calls. Typing "`man syscalls`" will give you the complete list.

C programs can invoke any system call directly by using the `_syscall` macro described in "`man 2 intro`". However, it is usually neither necessary nor desirable to invoke system calls directly. The standard C library provides a set of convenient wrapper functions for the most frequently used system calls. The wrapper functions package up the arguments, trap to the kernel with the appropriate system call, and then pass the return status of the system call back to the calling program. In our discussion in the following sections, we will refer to system calls and their associated wrapper functions interchangeably as *system-level functions*.

When Unix system-level functions encounter an error, they typically return -1 and set the global integer variable `errno` to indicate what went wrong. Programmers should *always* check for errors, but unfortunately, many skip error checking because it bloats the code and makes it harder to read. For example, here is how we might check for errors when we call the Unix `fork` function:

```
1     if ((pid = fork()) < 0) {
2         fprintf(stderr, "fork error: %s\n", strerror(errno));
3         exit(0);
4     }
```

The `strerror` function returns a text string that describes the error associated with a particular value of `errno`. We can simplify this code somewhat by defining the following *error-reporting function*:

```
1 void unix_error(char *msg) /* unix-style error */
2 {
3     fprintf(stderr, "%s: %s\n", msg, strerror(errno));
4     exit(0);
5 }
```

Given this function, our call to `fork` reduces from four lines to two lines:

```
1     if ((pid = fork()) < 0)
2         unix_error("fork error");
```

We can simplify our code even further by using *error-handling wrappers*. For a given base function `foo`, we define a wrapper function `Foo` with identical arguments, but with the first letter of the name capitalized. The wrapper calls the base function, checks for errors, and terminates if there are any problems. For example, here is the error-handling wrapper for the `fork` function:

```
1 pid_t Fork(void)
2 {
3     pid_t pid;
4
5     if ((pid = fork()) < 0)
6         unix_error("Fork error");
7     return pid;
8 }
```

Given this wrapper, our call to `fork` shrinks to a single compact line:

```
1     pid = Fork();
```

We will use error-handling wrappers throughout the remainder of this book. They allow us to keep our code examples concise, without giving you the mistaken impression that it is permissible to ignore error checking. Note that when we discuss system-level functions in the text, we will always refer to them by their lowercase base names, rather than by their uppercase wrapper names.

See Appendix B for a discussion of Unix error handling and the error-handling wrappers used throughout this book. The wrappers are defined in a file called `csapp.c`, and their prototypes are defined in a header file called `csapp.h`. For your reference, Appendix B provides the sources for these files.

8.4 Process Control

Unix provides a number of system calls for manipulating processes from C programs. This section describes the important functions and gives examples of how they are used.

8.4.1 Obtaining Process ID's

Each process has a unique positive (nonzero) *process ID (PID)*. The `getpid` function returns the PID of the calling process. The `getppid` function returns the PID of its *parent* (i.e., the process that created the calling process).

```
#include <unistd.h>
#include <sys/types.h>

pid_t getpid(void);
pid_t getppid(void);
```

Returns: PID of either the caller or the parent

The `getpid` and `getppid` routines return an integer value of type `pid_t`, which on Linux systems is defined in `types.h` as an `int`.

8.4.2 Creating and Terminating Processes

From a programmer's perspective, we can think of a process as being in one of three states:

- *Running.* The process is either executing on the CPU, or is waiting to be executed and will eventually be scheduled.
- *Stopped.* The execution of the process is *suspended* and will not be scheduled. A process stops as a result of receiving a SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal, and it remains stopped until it receives a SIGCONT signal, at which point is becomes running again. (A *signal* is a form of software interrupt that is described in detail in Section 8.5.)
- *Terminated.* The process is stopped permanently. A process becomes terminated for one of three reasons: (1) receiving a signal whose default action is to terminate the process; (2) returning from the main routine; or (3) calling the `exit` function:

```
#include <stdlib.h>

void exit(int status);
```

This function does not return

The `exit` function terminates the process with an *exit status* of `status`. (The other way to set the exit status is to return an integer value from the main routine.)

A *parent process* creates a new running *child process* by calling the `fork` function.

```
#include <unistd.h>
#include <sys/types.h>

pid_t fork(void);
```

Returns: 0 to child, PID of child to parent, -1 on error

The newly created child process is almost, but not quite, identical to the parent. The child gets an identical (but separate) copy of the parent's user-level virtual address space, including the text, data, and bss segments,

heap, and user stack. The child also gets identical copies of any of the parent's open file descriptors, which means the child can read and write any files that were open in the parent when it called `fork`. The most significant difference between the parent and the newly created child is that they have different PIDs.

The `fork` function is interesting (and often confusing) because it is called *once* but it returns *twice*: once in the calling process (the parent), and once in the newly created child process. In the parent, `fork` returns the PID of the child. In the child, `fork` returns a value of 0. Since the PID of the child is always nonzero, the return value provides an unambiguous way to tell whether the program is executing in the parent or the child.

Figure 8.13 shows a simple example of a parent process that uses `fork` to create a child process. When the `fork` call returns in line 8, `x` has a value of 1 in both the parent and child. The child increments and prints its copy of `x` in line 10. Similarly, the parent decrements and prints its copy of `x` in line 15.

code/ecf/fork.c

```
1  #include "csapp.h"
2
3  int main()
4  {
5      pid_t pid;
6      int x = 1;
7
8      pid = Fork();
9      if (pid == 0) {  /* child */
10         printf("child : x=%d\n", ++x);
11         exit(0);
12     }
13
14     /* parent */
15     printf("parent: x=%d\n", --x);
16     exit(0);
17 }
```

code/ecf/fork.c

Figure 8.13: **Using `fork` to create a new process.**

When we run the program on our Unix system, we get the following result:

```
unix> ./fork
parent: x=0
child : x=2
```

There are some subtle aspects to this simple example.

- *Call once, return twice.* The `fork` function is called once by the parent, but it returns twice: once to the parent and once to the newly created child. This is fairly straightforward for programs that create a single child. But programs with multiple instances of `fork` can be confusing and need to be reasoned about carefully.

- *Concurrent execution.* The parent and the child are separate processes that run concurrently. The instructions in their logical control flows can be interleaved by the kernel in an arbitrary way. When we run the program on our system, the parent process completes its `printf` statement first, followed by the child. However, on another system the reverse might be true. In general, as programmers we can never make assumptions about the interleaving of the instructions in different processes.

- *Duplicate but separate address spaces.* If we could halt both the parent and the child immediately after the `fork` function returned in each process, we would see that the address space of each process is identical. Each process has the same user stack, the same local variable values, the same heap, the same global variable values, and the same code. Thus, in our example program, local variable `x` has a value of 1 in both the parent and the child when the `fork` function returns in line 8. However, since the parent and the child are separate processes, they each have their own private address spaces. Any subsequent changes that a parent or child makes to `x` are private and are not reflected in the memory of the other process. This is why the variable `x` has different values in the parent and child when they call their respective `printf` statements.

- *Shared files.* When we run the example program, we notice that both parent and child print their output on the screen. The reason is that the child inherits all of the parent's open files. When the parent calls `fork`, the `stdout` file is open and directed to the screen. The child inherits this file and thus its output is also directed to the screen.

When you are first learning about the `fork` function, it is often helpful to sketch the *process graph*, where each horizontal arrow corresponds to a process that executes instructions from left to right, and each vertical arrow corresponds to the execution of a `fork` function.

For example, how many lines of output would the program in Figure 8.14(a) generate? Figure 8.14(b) shows the corresponding process graph. The parent creates a child when it executes the first (and only) `fork` in the program. Each of these calls `printf` once, so the program prints two output lines.

Now what if we were to call `fork` twice, as shown in Figure 8.14(c)? As we see from Figure 8.14(d), the parent calls `fork` to create a child, and then the parent and child each call `fork`, which results in two more processes. Thus, there are four processes, each of which calls `printf`, so the program generates four output lines.

Continuing this line of thought, what would happen if we were to call `fork` three times, as in Figure 8.14(e)? As we see from the process graph in Figure 8.14(f), there are a total of eight processes. Each process calls `printf`, so the program produces eight output lines.

Practice Problem 8.1:

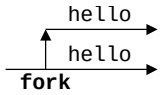
Consider the following program:

code/ecf/forkprob0.c

```
1 #include "csapp.h"
2
3 int main()
4 {
5     int x = 1;
```

```
1 #include "csapp.h"
2
3 int main()
4 {
5     Fork();
6     printf("hello!\n");
7     exit(0);
8 }
```

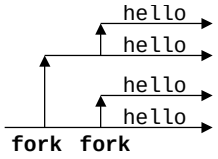
(a) Calls `fork` once.



(b) Prints two output lines.

```
1 #include "csapp.h"
2
3 int main()
4 {
5     Fork();
6     Fork();
7     printf("hello!\n");
8     exit(0);
9 }
```

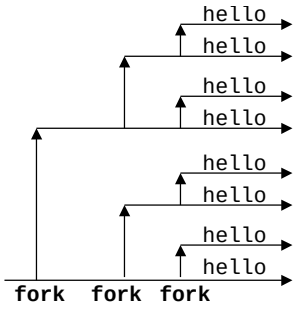
(c) Calls `fork` twice.



(d) Prints four output lines.

```
1 #include "csapp.h"
2
3 int main()
4 {
5     Fork();
6     Fork();
7     Fork();
8     printf("hello!\n");
9     exit(0);
10 }
```

(e) Calls `fork` three times.



(f) Prints eight output lines.

Figure 8.14: **Examples of `fork` programs.**

```
6
7       if (Fork() == 0)
8           printf("printf1: x=%d\n", ++x);
9       printf("printf2: x=%d\n", --x);
10      exit(0);
11  }
```

code/ecf/forkprob0.c

A. What is the output of the child process?

B. What is the output of the parent process?

Practice Problem 8.2:

How many "hello" output lines does this program print?

code/ecf/forkprob1.c

```
1  #include "csapp.h"
2
3  int main()
4  {
5      int i;
6
7      for (i = 0; i < 2; i++)
8          Fork();
9      printf("hello!\n");
10     exit(0);
11 }
```

code/ecf/forkprob1.c

Practice Problem 8.3:

How many "hello" output lines does this program print?

code/ecf/forkprob4.c

```
1  #include "csapp.h"
2
3  void doit()
4  {
5      Fork();
6      Fork();
7      printf("hello\n");
8      return;
9  }
10
11 int main()
12 {
```

```
13      doit();
14      printf("hello\n");
15      exit(0);
16  }
```

code/ecf/forkprob4.c

8.4.3 Reaping Child Processes

When a process terminates for any reason, the kernel does not remove it from the system immediately. Instead, the process is kept around in a terminated state until it is *reaped* by its parent. When the parent reaps the terminated child, the kernel passes the child's exit status to the parent, and then discards the terminated process, at which point it ceases to exist. A terminated process that has not yet been reaped is called a *zombie*.

> **Aside: Why are terminated children called zombies?**
> In folklore, a zombie is a living corpse, an entity that is half alive and half dead. A zombie process is similar in the sense that while it has already terminated, the kernel maintains some of its state until it can be reaped by the parent.
> **End Aside.**

If the parent process terminates without reaping its zombie children, the kernel arranges for the `init` process to reap them. The `init` process has a PID of 1 and is created by the kernel during system initialization. Long-running programs such as shells or servers should always reap their zombie children. Even though zombies are not running, they still consume system memory resources.

A process waits for its children to terminate or stop by calling the `waitpid` function.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
```

Returns: PID of child if OK, 0 (if WNOHANG) or -1 on error

The `waitpid` function is complicated. By default (when `options = 0`), `waitpid` suspends execution of the calling process until a child process in its *wait set* terminates. If a process in the wait set has already terminated at the time of the call, then `waitpid` returns immediately. In either case, `waitpid` returns the PID of the terminated child that caused `waitpid` to return, and the terminated child is removed from the system.

Determining the Members of the Wait Set

The members of the wait set are determined by the `pid` argument:

- If `pid > 0`, then the wait set is the singleton child process whose process ID is equal to `pid`.