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)*. ECF 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 invoking 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 concurrency.* ECF is a basic mechanism for implementing concurrency in computer systems. The following are all examples of concurrency in action: an exception handler that interrupts the execution of an application program; processes and threads whose execution overlap in time; and a signal handler that interrupts the execution of an application program. Understanding ECF is a first step to understanding concurrency. We will return to study it in more detail in Chapter 12.
- *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*.

Aside Hardware versus 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.

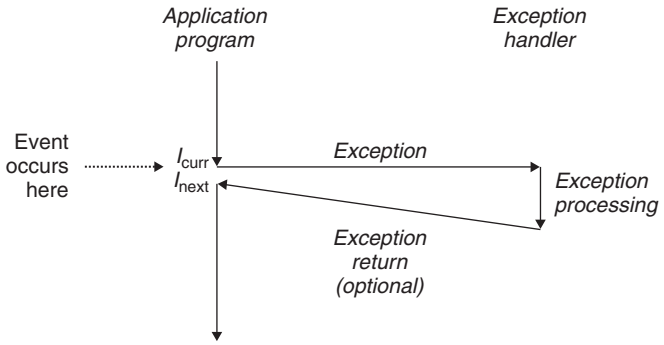


Figure 8.1
Anatomy of an exception. A change in the processor's state (an event) triggers an abrupt control transfer (an exception) from the application program to an exception handler. After it finishes processing, the handler either returns control to the interrupted program or aborts.

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.

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.

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

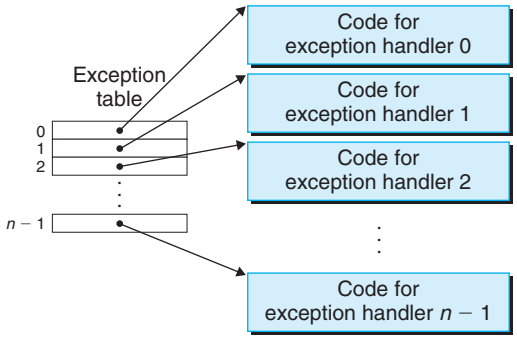


Figure 8.2
Exception table. The exception table is a jump table where entry k contains the address of the handler code for exception k.

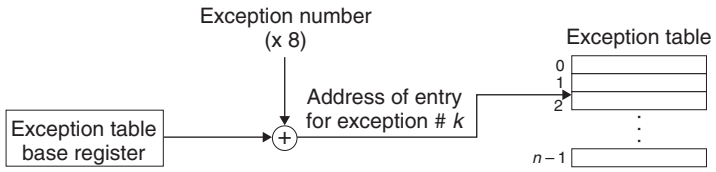


Figure 8.3
Generating the address of an exception handler. The exception number is an index into the exception table.

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.

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*.

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 x86-64 system pushes the EFLAGS register containing the current condition codes, among other things, onto the stack.
- When 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 8.2.4), 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 8.2.4) if the exception 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.

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 signaling a pin on the processor chip and placing onto the system bus the exception number that identifies the device that caused the interrupt.

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.

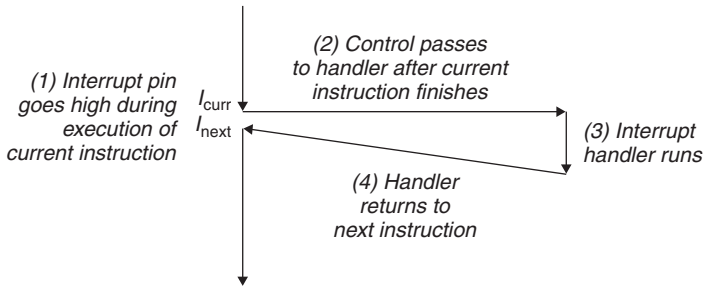


Figure 8.5
Interrupt handling. The interrupt handler returns control to the next instruction in the application program's control flow.

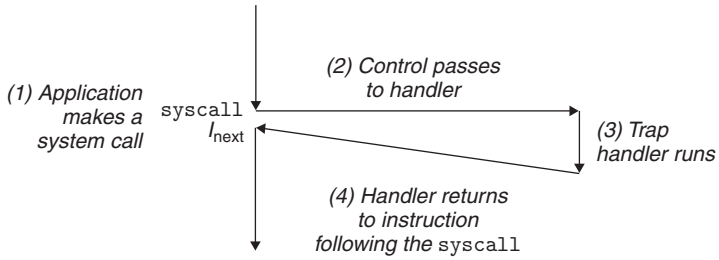


Figure 8.6
Trap handling. The trap handler returns control to the next instruction in the application program's control flow.

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 and System Calls

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`), and 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 to a regular function call. However, their implementations are quite different. Regular functions

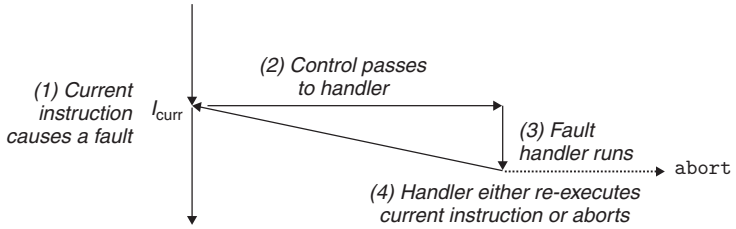


Figure 8.7
Fault handling. Depending on whether the fault can be repaired or not, the fault handler either re-executes the faulting instruction or aborts.

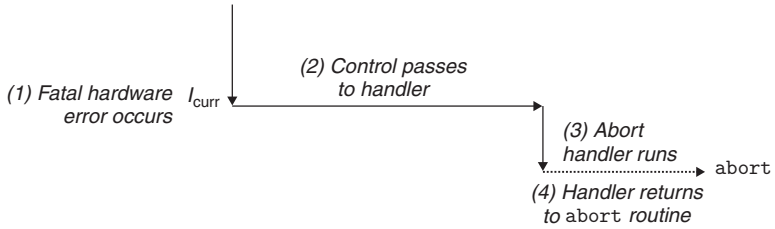


Figure 8.8
Abort handling. The abort handler passes control to a kernel `abort` routine that terminates the application program.

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 privileged instructions and access a stack defined in the kernel. Section 8.2.4 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 re-executing 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 page is not resident in memory and must therefore be retrieved from disk. As we will see in Chapter 9, a page is a contiguous block (typically 4 KB) of virtual memory. The 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 page is now 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.

Exception number	Description	Exception class
0	Divide error	Fault
13	General protection fault	Fault
14	Page fault	Fault
18	Machine check	Abort
32–255	OS-defined exceptions	Interrupt or trap

Figure 8.9 **Examples of exceptions in x86-64 systems.**

8.1.3 Exceptions in Linux/x86-64 Systems

To help make things more concrete, let's look at some of the exceptions defined for x86-64 systems. There are up to 256 different exception types [50]. Numbers in the range from 0 to 31 correspond to exceptions that are defined by the Intel architects and thus are identical for any x86-64 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.

Linux/x86-64 Faults and Aborts

Divide error. 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. Linux shells typically report divide errors as "Floating exceptions."

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

Page fault. A page fault (exception 14) is an example of an exception where the faulting instruction is restarted. The handler maps the appropriate page of virtual memory on disk into a page of physical memory and then restarts the faulting instruction. We will see how page faults work in detail in Chapter 9.

Machine check. 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.

Linux/x86-64 System Calls

Linux provides hundreds of system calls that application programs use when they want to request services from the kernel, such as reading a file, writing a file, and

Number	Name	Description	Number	Name	Description
0	`read`	Read file	33	`pause`	Suspend process until signal arrives
1	`write`	Write file	37	`alarm`	Schedule delivery of alarm signal
2	`open`	Open file	39	`getpid`	Get process ID
3	`close`	Close file	57	`fork`	Create process
4	`stat`	Get info about file	59	`execve`	Execute a program
9	`mmap`	Map memory page to file	60	`_exit`	Terminate process
12	`brk`	Reset the top of the heap	61	`wait4`	Wait for a process to terminate
32	`dup2`	Copy file descriptor	62	`kill`	Send signal to a process

Figure 8.10 **Examples of popular system calls in Linux x86-64 systems.**

creating a new process. Figure 8.10 lists some popular Linux system calls. Each system call has a unique integer number that corresponds to an offset in a jump table in the kernel. (Notice that this jump table is not the same as the exception table.)

C programs can invoke any system call directly by using the `syscall` function. However, this is rarely necessary in practice. The C standard library provides a set of convenient wrapper functions for most system calls. The wrapper functions package up the arguments, trap to the kernel with the appropriate system call instruction, and then pass the return status of the system call back to the calling program. Throughout this text, we will refer to system calls and their associated wrapper functions interchangeably as *system-level functions*.

System calls are provided on x86-64 systems via a trapping instruction called `syscall`. It is quite interesting to study how programs can use this instruction to invoke Linux system calls directly. All arguments to Linux system calls are passed through general-purpose registers rather than the stack. By convention, register `%rax` contains the syscall number, with up to six arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, and `%r9`. The first argument is in `%rdi`, the second in `%rsi`, and so on. On return from the system call, registers `%rcx` and `%r11` are destroyed, and `%rax` contains the return value. A negative return value between $-4{,}095$ and -1 indicates an error corresponding to negative `errno`.

For example, consider the following version of the familiar `hello` program, written using the `write` system-level function (Section 10.4) instead of `printf`:

```
1  int main()
2  {
3      write(1, "hello, world\n", 13);
4      _exit(0);
5  }
```

The first argument to `write` sends the output to `stdout`. The second argument is the sequence of bytes to write, and the third argument gives the number of bytes to write.

Aside A note on terminology

The terminology for the various classes of exceptions varies from system to system. Processor ISA 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.

code/ecf/hello-asm64.sa

```
1   .section .data
2   string:
3     .ascii "hello, world\n"
4   string_end:
5     .equ len, string_end - string
6   .section .text
7   .globl main
8   main:
      First, call write(1, "hello, world\n", 13)
9     movq $1, %rax          write is system call 1
10    movq $1, %rdi          Arg1: stdout has descriptor 1
11    movq $string, %rsi     Arg2: hello world string
12    movq $len, %rdx        Arg3: string length
13    syscall                Make the system call

      Next, call _exit(0)
14    movq $60, %rax         _exit is system call 60
15    movq $0, %rdi          Arg1: exit status is 0
16    syscall                Make the system call
```

code/ecf/hello-asm64.sa

Figure 8.11 **Implementing the `hello` program directly with Linux system calls.**

Figure 8.11 shows an assembly-language version of `hello` that uses the `syscall` instruction to invoke the `write` and `exit` system calls directly. Lines 9–13 invoke the `write` function. First, line 9 stores the number of the `write` system call in `%rax`, and lines 10–12 set up the argument list. Then, line 13 uses the `syscall` instruction to invoke the system call. Similarly, lines 14–16 invoke the `_exit` system call.

8.2 Processes

Exceptions are the basic building blocks that allow the operating system kernel 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 concurrently 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*, or simply *logical flow*.

Consider a system that runs three processes, as shown in Figure 8.12. 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

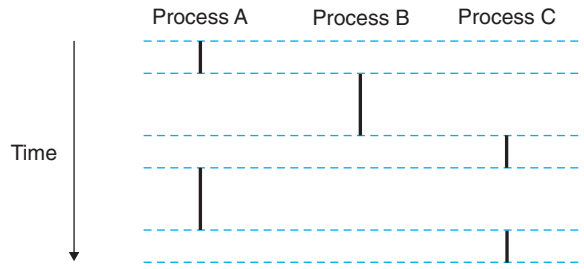


Figure 8.12
Logical control flows. Processes provide each program with the illusion that it has exclusive use of the processor. Each vertical bar represents a portion of the logical control flow for a process.

a process. In the example, the execution of the three logical flows is interleaved. Process A runs for a while, followed by B, which runs to completion. Process C then runs for a while, followed by A, which runs to completion. Finally, C is able to run to completion.

The key point in Figure 8.12 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, 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.

8.2.2 Concurrent Flows

Logical flows take many different forms in computer systems. Exception handlers, processes, signal handlers, threads, and Java processes are all examples of logical flows.

A logical flow whose execution overlaps in time with another flow is called a *concurrent flow*, and the two flows are said to *run concurrently*. More precisely, flows X and Y are concurrent with respect to each other if and only if X begins after Y begins and before Y finishes, or Y begins after X begins and before X finishes. For example, in Figure 8.12, 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 general phenomenon of multiple flows executing concurrently is known as *concurrency*. The notion of a process taking turns with other processes is also 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*. For example, in Figure 8.12, the flow for process A consists of two time slices.

Notice that the idea of concurrent flows is independent of the number of processor cores or computers that the flows are running on. If two flows overlap in time, then they are concurrent, even if they are running on the same processor. However, we will sometimes find it useful to identify a proper subset of concurrent

flows known as *parallel flows*. If two flows are running concurrently on different processor cores or computers, then we say that they are *parallel flows*, that they are *running in parallel*, and have *parallel execution*.

Practice Problem 8.1 **(solution page 831)**

Consider three processes with the following starting and ending times:

Process	Start time	End time
A	1	3
B	2	5
C	4	6

For each pair of processes, indicate whether they run concurrently (Y) or not (N):

Process pair	Concurrent?
AB	_____
AC	_____
BC	_____

8.2.3 Private Address Space

A process 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.

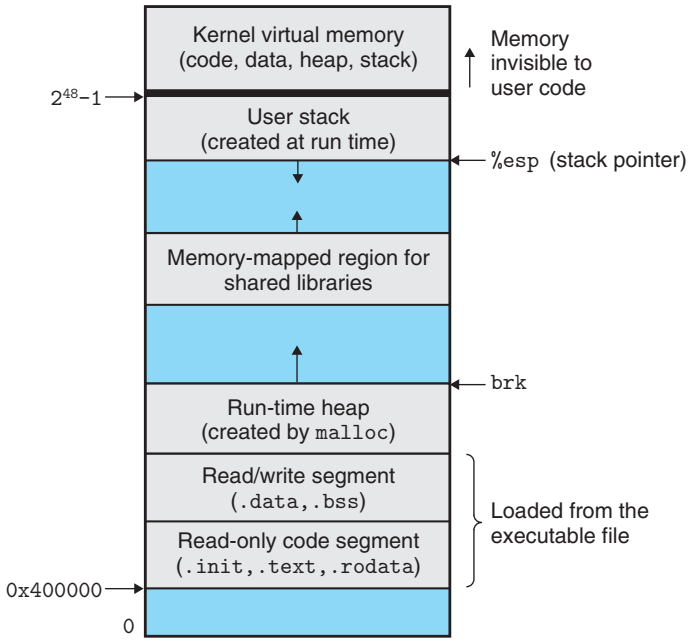
Although the contents of the memory associated with each private address space is different in general, each such space has the same general organization. For example, Figure 8.13 shows the organization of the address space for an x86-64 Linux process.

The bottom portion of the address space is reserved for the user program, with the usual code, data, heap, and stack segments. The code segment always begins at address `0x400000`. The top portion of the address space is reserved for the kernel (the memory-resident part of the operating system). This part 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).

8.2.4 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

Figure 8.13
Process address space.



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 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 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 text

files that can be read by user programs. For example, you can use the `/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`). The 2.6 version of the Linux kernel introduced a `/sys` filesystem, which exports additional low-level information about system buses and devices.

8.2.5 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 8.14 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

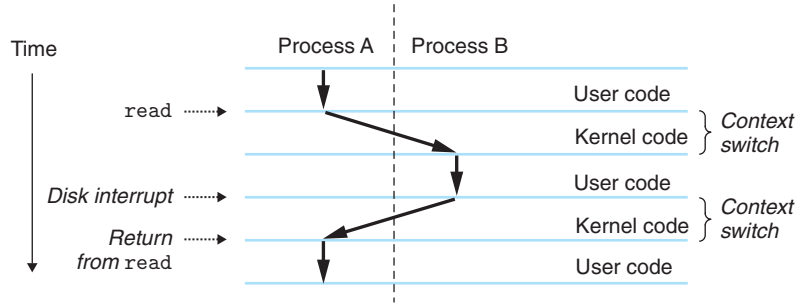


Figure 8.14
Anatomy of a process context switch.

processor after the disk controller has finished transferring the data from disk to memory.

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 (i.e., there is no separate kernel process). 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 have 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.

8.3 System Call Error Handling

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 Linux `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*, as pioneered by Stevens in [110]. 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 A 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`. These are available online from the CS:APP Web site.

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 IDs

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 <sys/types.h>
#include <unistd.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 waiting to be executed and will eventually be scheduled by the kernel.

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 it becomes running again. (A *signal* is a form of software interrupt that we will describe 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 <sys/types.h>
#include <unistd.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 code and data segments, heap, shared libraries, 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 8.15 shows a simple example of a parent process that uses `fork` to create a child process. When the `fork` call returns in line 6, `x` has a value of 1 in both the parent and child. The child increments and prints its copy of `x` in line 8. Similarly, the parent decrements and prints its copy of `x` in line 13.

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

```
linux> ./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.

code/ecf/fork.c

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

code/ecf/fork.c

Figure 8.15 **Using `fork` to create a new process.**

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 6. 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*, which is a simple kind of precedence graph that captures the partial ordering of program statements. Each vertex a corresponds to the execution of a program statement. A directed edge $a \rightarrow b$ denotes that statement a "happens before" statement b. Edges can be labeled with information such as the current value of a variable. Vertices corresponding to `printf` statements can be labeled with the output of the `printf`. Each graph begins with a vertex that

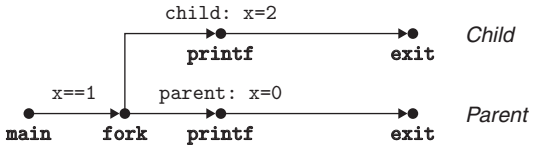


Figure 8.16
Process graph for the example program in Figure 8.15.

```
1   int main()
2   {
3       Fork();
4       Fork();
5       printf("hello\n");
6       exit(0);
7   }
```

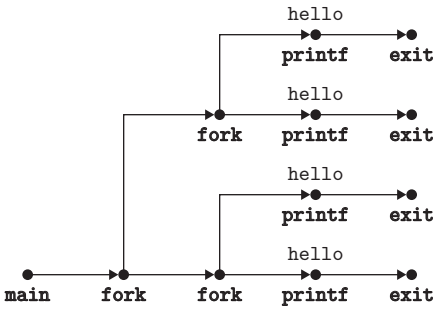


Figure 8.17 **Process graph for a nested `fork`.**

corresponds to the parent process calling `main`. This vertex has no inedges and exactly one outedge. The sequence of vertices for each process ends with a vertex corresponding to a call to `exit`. This vertex has one inedge and no outedges.

For example, Figure 8.16 shows the process graph for the example program in Figure 8.15. Initially, the parent sets variable `x` to 1. The parent calls `fork`, which creates a child process that runs concurrently with the parent in its own private address space.

For a program running on a single processor, any *topological sort* of the vertices in the corresponding process graph represents a feasible total ordering of the statements in the program. Here's a simple way to understand the idea of a topological sort: Given some permutation of the vertices in the process graph, draw the sequence of vertices in a line from left to right, and then draw each of the directed edges. The permutation is a topological sort if and only if each edge in the drawing goes from left to right. Thus, in our example program in Figure 8.15, the `printf` statements in the parent and child can occur in either order because each of the orderings corresponds to some topological sort of the graph vertices.

The process graph can be especially helpful in understanding programs with nested `fork` calls. For example, Figure 8.17 shows a program with two calls to `fork` in the source code. The corresponding process graph helps us see that this program runs four processes, each of which makes a call to `printf` and which can execute in any order.

Practice Problem 8.2 (solution page 831)

Consider the following program:

code/ecf/global-forkprob0.c

```
1   int main()
2   {
3       int a = 9;
4
5       if (Fork() == 0)
6           printf("p1: a=%d\n", a--);
7       printf("p2: a=%d\n", a++);
8       exit(0);
9   }
```

code/ecf/global-forkprob0.c

A. What is the output of the child process?

B. What is the output of the parent process?

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*.

When a parent process terminates, the kernel arranges for the `init` process to become the adopted parent of any orphaned children. The `init` process, which has a PID of 1, is created by the kernel during system start-up, never terminates, and is the ancestor of every process. If a parent process terminates without reaping its zombie children, then the kernel arranges for the `init` process to reap them. However, 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 *statusp, int options);
                    Returns: PID of child if OK, 0 (if WNOHANG), or −1 on error
```

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 although it has already terminated, the kernel maintains some of its state until it can be reaped by the parent.

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. At this point, the terminated child has been reaped and the kernel removes all traces of it 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`.
- If `pid = -1`, then the wait set consists of all of the parent's child processes.

The `waitpid` function also supports other kinds of wait sets, involving Unix process groups, which we will not discuss.

Modifying the Default Behavior

The default behavior can be modified by setting `options` to various combinations of the WNOHANG, WUNTRACED, and WCONTINUED constants:

WNOHANG. Return immediately (with a return value of 0) if none of the child processes in the wait set has terminated yet. The default behavior suspends the calling process until a child terminates; this option is useful in those cases where you want to continue doing useful work while waiting for a child to terminate.

WUNTRACED. Suspend execution of the calling process until a process in the wait set becomes either terminated or stopped. Return the PID of the terminated or stopped child that caused the return. The default behavior returns only for terminated children; this option is useful when you want to check for both terminated *and* stopped children.

WCONTINUED. Suspend execution of the calling process until a running process in the wait set is terminated or until a stopped process in the wait set has been resumed by the receipt of a SIGCONT signal. (Signals are explained in Section 8.5.)

You can combine options by ORing them together. For example:

- WNOHANG | WUNTRACED: Return immediately, with a return value of 0, if none of the children in the wait set has stopped or terminated, or with a return value equal to the PID of one of the stopped or terminated children.

Checking the Exit Status of a Reaped Child

If the `statusp` argument is non-NULL, then `waitpid` encodes status information about the child that caused the return in `status`, which is the value pointed to by `statusp`. The `wait.h` include file defines several macros for interpreting the `status` argument:

WIFEXITED(`status`). Returns true if the child terminated normally, via a call to `exit` or a return.

WEXITSTATUS(`status`). Returns the exit status of a normally terminated child. This status is only defined if WIFEXITED() returned true.

WIFSIGNALED(`status`). Returns true if the child process terminated because of a signal that was not caught.

WTERMSIG(`status`). Returns the number of the signal that caused the child process to terminate. This status is only defined if WIFSIGNALED() returned true.

WIFSTOPPED(`status`). Returns true if the child that caused the return is currently stopped.

WSTOPSIG(`status`). Returns the number of the signal that caused the child to stop. This status is only defined if WIFSTOPPED() returned true.

WIFCONTINUED(`status`). Returns true if the child process was restarted by receipt of a SIGCONT signal.

Error Conditions

If the calling process has no children, then `waitpid` returns -1 and sets `errno` to ECHILD. If the `waitpid` function was interrupted by a signal, then it returns -1 and sets `errno` to EINTR.

Practice Problem 8.3 **(solution page 833)**

List all of the possible output sequences for the following program:

code/ecf/global-waitprob0.c

```
1   int main()
2   {
3       if (Fork() == 0) {
4           printf("9"); fflush(stdout);
5       }
6       else {
```

```
7           printf("0"); fflush(stdout);
8           waitpid(-1, NULL, 0);
9       }
10      printf("3"); fflush(stdout);
11      printf("6"); exit(0);
12  }
```

code/ecf/global-waitprob0.c

The `wait` Function

The `wait` function is a simpler version of `waitpid`.

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

pid_t wait(int *statusp);
```

Returns: PID of child if OK or −1 on error

Calling `wait(&status)` is equivalent to calling `waitpid(-1, &status, 0)`.

Examples of Using `waitpid`

Because the `waitpid` function is somewhat complicated, it is helpful to look at a few examples. Figure 8.18 shows a program that uses `waitpid` to wait, in no particular order, for all of its N children to terminate. In line 11, the parent creates each of the N children, and in line 12, each child exits with a unique exit status.

Aside Constants associated with Unix functions

Constants such as WNOHANG and WUNTRACED are defined by system header files. For example, WNOHANG and WUNTRACED are defined (indirectly) by the `wait.h` header file:

```
/* Bits in the third argument to 'waitpid'. */
#define WNOHANG    1   /* Don't block waiting. */
#define WUNTRACED  2   /* Report status of stopped children. */
```

In order to use these constants, you must include the `wait.h` header file in your code:

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

The `man` page for each Unix function lists the header files to include whenever you use that function in your code. Also, in order to check return codes such as ECHILD and EINTR, you must include `errno.h`. To simplify our code examples, we include a single header file called `csapp.h` that includes the header files for all of the functions used in the book. The `csapp.h` header file is available online from the CS:APP Web site.

code/ecf/waitpid1.c

```
1   #include "csapp.h"
2   #define N 2
3
4   int main()
5   {
6       int status, i;
7       pid_t pid;
8
9       /* Parent creates N children */
10      for (i = 0; i < N; i++)
11          if ((pid = Fork()) == 0)  /* Child */
12              exit(100+i);
13
14      /* Parent reaps N children in no particular order */
15      while ((pid = waitpid(-1, &status, 0)) > 0) {
16          if (WIFEXITED(status))
17              printf("child %d terminated normally with exit status=%d\n",
18                     pid, WEXITSTATUS(status));
19          else
20              printf("child %d terminated abnormally\n", pid);
21      }
22
23      /* The only normal termination is if there are no more children */
24      if (errno != ECHILD)
25          unix_error("waitpid error");
26
27      exit(0);
28  }
```

code/ecf/waitpid1.c

Figure 8.18 **Using the `waitpid` function to reap zombie children in no particular order.**

Before moving on, make sure you understand why line 12 is executed by each of the children, but not the parent.

In line 15, the parent waits for all of its children to terminate by using `waitpid` as the test condition of a `while` loop. Because the first argument is -1, the call to `waitpid` blocks until an arbitrary child has terminated. As each child terminates, the call to `waitpid` returns with the nonzero PID of that child. Line 16 checks the exit status of the child. If the child terminated normally—in this case, by calling the `exit` function—then the parent extracts the exit status and prints it on `stdout`.

When all of the children have been reaped, the next call to `waitpid` returns -1 and sets `errno` to ECHILD. Line 24 checks that the `waitpid` function terminated normally, and prints an error message otherwise. When we run the program on our Linux system, it produces the following output:

```
linux> ./waitpid1
child 22966 terminated normally with exit status=100
child 22967 terminated normally with exit status=101
```

Notice that the program reaps its children in no particular order. The order that they were reaped is a property of this specific computer system. On another system, or even another execution on the same system, the two children might have been reaped in the opposite order. This is an example of the *nondeterministic* behavior that can make reasoning about concurrency so difficult. Either of the two possible outcomes is equally correct, and as a programmer you may *never* assume that one outcome will always occur, no matter how unlikely the other outcome appears to be. The only correct assumption is that each possible outcome is equally likely.

Figure 8.19 shows a simple change that eliminates this nondeterminism in the output order by reaping the children in the same order that they were created by the parent. In line 11, the parent stores the PIDs of its children in order and then waits for each child in this same order by calling `waitpid` with the appropriate PID in the first argument.

Practice Problem 8.4 (solution page 833)

Consider the following program:

code/ecf/global-waitprob1.c

```
1   int main()
2   {
3       int status;
4       pid_t pid;
5
6       printf("Start\n");
7       pid = Fork();
8       printf("%d\n", !pid);
9       if (pid == 0) {
10          printf("Child\n");
11      }
12      else if ((waitpid(-1, &status, 0) > 0) &&
                    (WIFEXITED(status) != 0)) {
13          printf("%d\n", WEXITSTATUS(status));
14      }
15      printf("Stop\n");
16      exit(2);
17  }
```

code/ecf/global-waitprob1.c

A. How many output lines does this program generate?

B. What is one possible ordering of these output lines?

code/ecf/waitpid2.c

```
1   #include "csapp.h"
2   #define N 2
3
4   int main()
5   {
6       int status, i;
7       pid_t pid[N], retpid;
8
9       /* Parent creates N children */
10      for (i = 0; i < N; i++)
11          if ((pid[i] = Fork()) == 0)  /* Child */
12              exit(100+i);
13
14      /* Parent reaps N children in order */
15      i = 0;
16      while ((retpid = waitpid(pid[i++], &status, 0)) > 0) {
17          if (WIFEXITED(status))
18              printf("child %d terminated normally with exit status=%d\n",
19                     retpid, WEXITSTATUS(status));
20          else
21              printf("child %d terminated abnormally\n", retpid);
22      }
23
24      /* The only normal termination is if there are no more children */
25      if (errno != ECHILD)
26          unix_error("waitpid error");
27
28      exit(0);
29  }
```

code/ecf/waitpid2.c

Figure 8.19 **Using `waitpid` to reap zombie children in the order they were created.**

8.4.4 Putting Processes to Sleep

The `sleep` function suspends a process for a specified period of time.

```
#include <unistd.h>

unsigned int sleep(unsigned int secs);
                                                    Returns: seconds left to sleep
```

`Sleep` returns zero if the requested amount of time has elapsed, and the number of seconds still left to sleep otherwise. The latter case is possible if the `sleep` function

returns prematurely because it was interrupted by a *signal*. We will discuss signals in detail in Section 8.5.

Another function that we will find useful is the `pause` function, which puts the calling function to sleep until a signal is received by the process.

```
#include <unistd.h>

int pause(void);
```

Always returns −1

Practice Problem 8.5 **(solution page 833)**

Write a wrapper function for `sleep`, called `wakeup`, with the following interface:

```
unsigned int wakeup(unsigned int secs);
```

The `wakeup` function behaves exactly as the `sleep` function, except that it prints a message describing when the process actually woke up:

```
Woke up at 4 secs.
```

8.4.5 Loading and Running Programs

The `execve` function loads and runs a new program in the context of the current process.

```
#include <unistd.h>

int execve(const char *filename, const char *argv[],
           const char *envp[]);
```

Does not return if OK; returns −1 on error

The `execve` function loads and runs the executable object file `filename` with the argument list `argv` and the environment variable list `envp`. `Execve` returns to the calling program only if there is an error, such as not being able to find `filename`. So unlike `fork`, which is called once but returns twice, `execve` is called once and never returns.

The argument list is represented by the data structure shown in Figure 8.20. The `argv` variable points to a null-terminated array of pointers, each of which points to an argument string. By convention, `argv[0]` is the name of the executable object file. The list of environment variables is represented by a similar data structure, shown in Figure 8.21. The `envp` variable points to a null-terminated array of pointers to environment variable strings, each of which is a name-value pair of the form *name=value*.

Figure 8.20
Organization of an argument list.

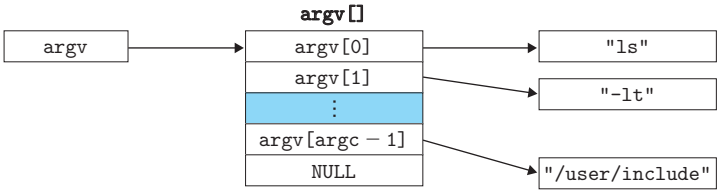
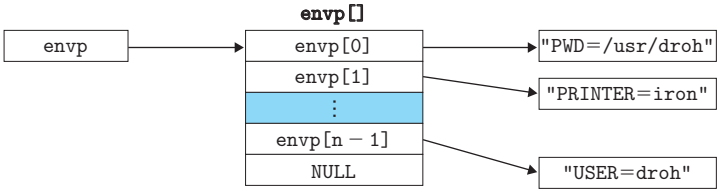


Figure 8.21
Organization of an environment variable list.



After `execve` loads `filename`, it calls the start-up code described in Section 7.9. The start-up code sets up the stack and passes control to the main routine of the new program, which has a prototype of the form

```
int main(int argc, char **argv, char **envp);
```

or equivalently,

```
int main(int argc, char *argv[], char *envp[]);
```

When `main` begins executing, the user stack has the organization shown in Figure 8.22. Let's work our way from the bottom of the stack (the highest address) to the top (the lowest address). First are the argument and environment strings. These are followed further up the stack by a null-terminated array of pointers, each of which points to an environment variable string on the stack. The global variable `environ` points to the first of these pointers, `envp[0]`. The environment array is followed by the null-terminated `argv[]` array, with each element pointing to an argument string on the stack. At the top of the stack is the stack frame for the system start-up function, `libc_start_main` (Section 7.9).

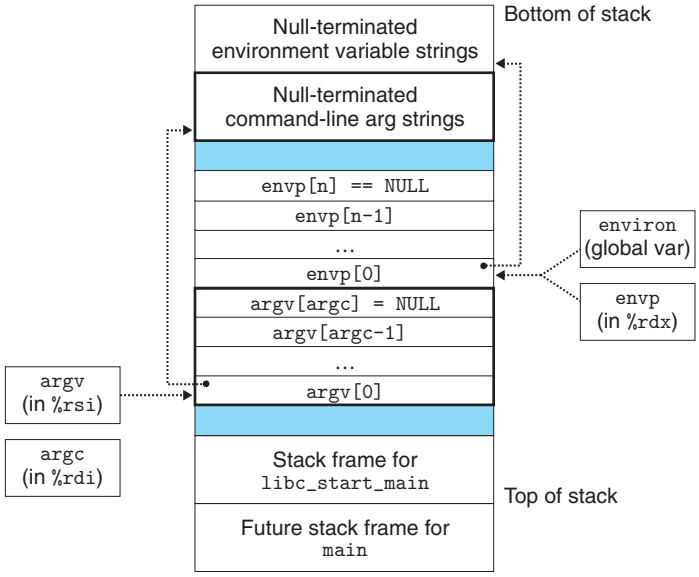
There are three arguments to function `main`, each stored in a register according to the x86-64 stack discipline: (1) `argc`, which gives the number of non-null pointers in the `argv[]` array; (2) `argv`, which points to the first entry in the `argv[]` array; and (3) `envp`, which points to the first entry in the `envp[]` array.

Linux provides several functions for manipulating the environment array:

```
#include <stdlib.h>

char *getenv(const char *name);
                                Returns: pointer to name if it exists, NULL if no match
```

Figure 8.22
Typical organization of the user stack when a new program starts.



The `getenv` function searches the environment array for a string `name=`*value*. If found, it returns a pointer to *value*; otherwise, it returns `NULL`.

```
#include <stdlib.h>

int setenv(const char *name, const char *newvalue, int overwrite);
                                                    Returns: 0 on success, −1 on error

void unsetenv(const char *name);
                                                    Returns: nothing
```

If the environment array contains a string of the form `name=`*oldvalue*, then `unsetenv` deletes it and `setenv` replaces *oldvalue* with `newvalue`, but only if `overwrite` is nonzero. If `name` does not exist, then `setenv` adds `name=newvalue` to the array.

Practice Problem 8.6 (solution page 833)

Write a program called `myecho` that prints its command-line arguments and environment variables. For example:

```
linux> ./myecho arg1 arg2
Command-ine arguments:
    argv[ 0]: myecho
    argv[ 1]: arg1
    argv[ 2]: arg2
```

```
Environment variables:
    envp[ 0]: PWD=/usr0/droh/ics/code/ecf
    envp[ 1]: TERM=emacs
    .
    .
    .
    envp[25]: USER=droh
    envp[26]: SHELL=/usr/local/bin/tcsh
    envp[27]: HOME=/usr0/droh
```

8.4.6 Using `fork` and `execve` to Run Programs

Programs such as Unix shells and Web servers make heavy use of the `fork` and `execve` functions. A *shell* is an interactive application-level program that runs other programs on behalf of the user. The original shell was the `sh` program, which was followed by variants such as `csh`, `tcsh`, `ksh`, and `bash`. A shell performs a sequence of *read/evaluate* steps and then terminates. The read step reads a command line from the user. The evaluate step parses the command line and runs programs on behalf of the user.

Figure 8.23 shows the main routine of a simple shell. The shell prints a command-line prompt, waits for the user to type a command line on `stdin`, and then evaluates the command line.

Figure 8.24 shows the code that evaluates the command line. Its first task is to call the `parseline` function (Figure 8.25), which parses the space-separated command-line arguments and builds the `argv` vector that will eventually be passed to `execve`. The first argument is assumed to be either the name of a built-in shell command that is interpreted immediately, or an executable object file that will be loaded and run in the context of a new child process.

If the last argument is an '&' character, then `parseline` returns 1, indicating that the program should be executed in the *background* (the shell does not wait for it to complete). Otherwise, it returns 0, indicating that the program should be run in the *foreground* (the shell waits for it to complete).

Aside Programs versus processes

This is a good place to pause and make sure you understand the distinction between a program and a process. A program is a collection of code and data; programs can exist as object files on disk or as segments in an address space. A process is a specific instance of a program in execution; a program always runs in the context of some process. Understanding this distinction is important if you want to understand the `fork` and `execve` functions. The `fork` function runs the same program in a new child process that is a duplicate of the parent. The `execve` function loads and runs a new program in the context of the current process. While it overwrites the address space of the current process, it does *not* create a new process. The new program still has the same PID, and it inherits all of the file descriptors that were open at the time of the call to the `execve` function.

code/ecf/shellex.c

```
1   #include "csapp.h"
2   #define MAXARGS   128
3
4   /* Function prototypes */
5   void eval(char *cmdline);
6   int parseline(char *buf, char **argv);
7   int builtin_command(char **argv);
8
9   int main()
10  {
11      char cmdline[MAXLINE]; /* Command line */
12
13      while (1) {
14          /* Read */
15          printf("> ");
16          Fgets(cmdline, MAXLINE, stdin);
17          if (feof(stdin))
18              exit(0);
19
20          /* Evaluate */
21          eval(cmdline);
22      }
23  }
```

code/ecf/shellex.c

Figure 8.23 **The main routine for a simple shell program.**

After parsing the command line, the `eval` function calls the `builtin_command` function, which checks whether the first command-line argument is a built-in shell command. If so, it interprets the command immediately and returns 1. Otherwise, it returns 0. Our simple shell has just one built-in command, the `quit` command, which terminates the shell. Real shells have numerous commands, such as `pwd`, `jobs`, and `fg`.

If `builtin_command` returns 0, then the shell creates a child process and executes the requested program inside the child. If the user has asked for the program to run in the background, then the shell returns to the top of the loop and waits for the next command line. Otherwise the shell uses the `waitpid` function to wait for the job to terminate. When the job terminates, the shell goes on to the next iteration.

Notice that this simple shell is flawed because it does not reap any of its background children. Correcting this flaw requires the use of signals, which we describe in the next section.

code/ecf/shellex.c

```
1   /* eval - Evaluate a command line */
2   void eval(char *cmdline)
3   {
4       char *argv[MAXARGS]; /* Argument list execve() */
5       char buf[MAXLINE];   /* Holds modified command line */
6       int bg;              /* Should the job run in bg or fg? */
7       pid_t pid;           /* Process id */
8
9       strcpy(buf, cmdline);
10      bg = parseline(buf, argv);
11      if (argv[0] == NULL)
12          return;   /* Ignore empty lines */
13
14      if (!builtin_command(argv)) {
15          if ((pid = Fork()) == 0) {   /* Child runs user job */
16              if (execve(argv[0], argv, environ) < 0) {
17                  printf("%s: Command not found.\n", argv[0]);
18                  exit(0);
19              }
20          }
21
22          /* Parent waits for foreground job to terminate */
23          if (!bg) {
24              int status;
25              if (waitpid(pid, &status, 0) < 0)
26                  unix_error("waitfg: waitpid error");
27          }
28          else
29              printf("%d %s", pid, cmdline);
30      }
31      return;
32  }
33
34  /* If first arg is a builtin command, run it and return true */
35  int builtin_command(char **argv)
36  {
37      if (!strcmp(argv[0], "quit")) /* quit command */
38          exit(0);
39      if (!strcmp(argv[0], "&"))    /* Ignore singleton & */
40          return 1;
41      return 0;                     /* Not a builtin command */
42  }
```

code/ecf/shellex.c

Figure 8.24 `eval` **evaluates the shell command line.**

code/ecf/shellex.c

```
1   /* parseline - Parse the command line and build the argv array */
2   int parseline(char *buf, char **argv)
3   {
4       char *delim;         /* Points to first space delimiter */
5       int argc;            /* Number of args */
6       int bg;              /* Background job? */
7
8       buf[strlen(buf)-1] = ' ';  /* Replace trailing '\n' with space */
9       while (*buf && (*buf == ' ')) /* Ignore leading spaces */
10          buf++;
11
12      /* Build the argv list */
13      argc = 0;
14      while ((delim = strchr(buf, ' '))) {
15          argv[argc++] = buf;
16          *delim = '\0';
17          buf = delim + 1;
18          while (*buf && (*buf == ' ')) /* Ignore spaces */
19                 buf++;
20      }
21      argv[argc] = NULL;
22
23      if (argc == 0)  /* Ignore blank line */
24          return 1;
25
26      /* Should the job run in the background? */
27      if ((bg = (*argv[argc-1] == '&')) != 0)
28          argv[--argc] = NULL;
29
30      return bg;
31  }
```

code/ecf/shellex.c

Figure 8.25 `parseline` **parses a line of input for the shell.**

8.5 Signals

To this point in our study of exceptional control flow, we have seen how hardware and software cooperate to provide the fundamental low-level exception mechanism. We have also seen how the operating system uses exceptions to support a form of exceptional control flow known as the process context switch. In this section, we will study a higher-level software form of exceptional control flow, known as a Linux signal, that allows processes and the kernel to interrupt other processes.

Number	Name	Default action	Corresponding event
1	SIGHUP	Terminate	Terminal line hangup
2	SIGINT	Terminate	Interrupt from keyboard
3	SIGQUIT	Terminate	Quit from keyboard
4	SIGILL	Terminate	Illegal instruction
5	SIGTRAP	Terminate and dump core [a]	Trace trap
6	SIGABRT	Terminate and dump core [a]	Abort signal from `abort` function
7	SIGBUS	Terminate	Bus error
8	SIGFPE	Terminate and dump core [a]	Floating-point exception
9	SIGKILL	Terminate [b]	Kill program
10	SIGUSR1	Terminate	User-defined signal 1
11	SIGSEGV	Terminate and dump core [a]	Invalid memory reference (seg fault)
12	SIGUSR2	Terminate	User-defined signal 2
13	SIGPIPE	Terminate	Wrote to a pipe with no reader
14	SIGALRM	Terminate	Timer signal from `alarm` function
15	SIGTERM	Terminate	Software termination signal
16	SIGSTKFLT	Terminate	Stack fault on coprocessor
17	SIGCHLD	Ignore	A child process has stopped or terminated
18	SIGCONT	Ignore	Continue process if stopped
19	SIGSTOP	Stop until next SIGCONT [b]	Stop signal not from terminal
20	SIGTSTP	Stop until next SIGCONT	Stop signal from terminal
21	SIGTTIN	Stop until next SIGCONT	Background process read from terminal
22	SIGTTOU	Stop until next SIGCONT	Background process wrote to terminal
23	SIGURG	Ignore	Urgent condition on socket
24	SIGXCPU	Terminate	CPU time limit exceeded
25	SIGXFSZ	Terminate	File size limit exceeded
26	SIGVTALRM	Terminate	Virtual timer expired
27	SIGPROF	Terminate	Profiling timer expired
28	SIGWINCH	Ignore	Window size changed
29	SIGIO	Terminate	I/O now possible on a descriptor
30	SIGPWR	Terminate	Power failure

Figure 8.26 **Linux signals.** *Notes:* (a) Years ago, main memory was implemented with a technology known as *core memory*. "Dumping core" is a historical term that means writing an image of the code and data memory segments to disk. (b) This signal can be neither caught nor ignored. (Source: `man 7 signal`. Data from the Linux Foundation.)

A *signal* is a small message that notifies a process that an event of some type has occurred in the system. Figure 8.26 shows the 30 different types of signals that are supported on Linux systems.

Each signal type corresponds to some kind of system event. Low-level hardware exceptions are processed by the kernel's exception handlers and would not normally be visible to user processes. Signals provide a mechanism for exposing

the occurrence of such exceptions to user processes. For example, if a process attempts to divide by zero, then the kernel sends it a SIGFPE signal (number 8). If a process executes an illegal instruction, the kernel sends it a SIGILL signal (number 4). If a process makes an illegal memory reference, the kernel sends it a SIGSEGV signal (number 11). Other signals correspond to higher-level software events in the kernel or in other user processes. For example, if you type Ctrl+C (i.e., press the Ctrl key and the 'c' key at the same time) while a process is running in the foreground, then the kernel sends a SIGINT (number 2) to each process in the foreground process group. A process can forcibly terminate another process by sending it a SIGKILL signal (number 9). When a child process terminates or stops, the kernel sends a SIGCHLD signal (number 17) to the parent.

8.5.1 Signal Terminology

The transfer of a signal to a destination process occurs in two distinct steps:

Sending a signal. The kernel *sends* (*delivers*) a signal to a destination process by updating some state in the context of the destination process. The signal is delivered for one of two reasons: (1) The kernel has detected a system event such as a divide-by-zero error or the termination of a child process. (2) A process has invoked the `kill` function (discussed in the next section) to explicitly request the kernel to send a signal to the destination process. A process can send a signal to itself.

Receiving a signal. A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal. The process can either ignore the signal, terminate, or *catch* the signal by executing a user-level function called a *signal handler*. Figure 8.27 shows the basic idea of a handler catching a signal.

A signal that has been sent but not yet received is called a *pending signal*. At any point in time, there can be at most one pending signal of a particular type. If a process has a pending signal of type k, then any subsequent signals of type k sent to that process are *not* queued; they are simply discarded. A process can selectively *block* the receipt of certain signals. When a signal is blocked, it can be

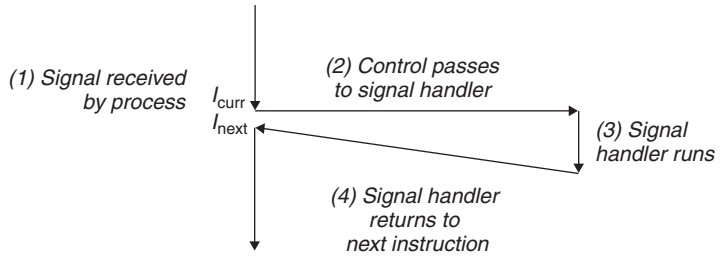


Figure 8.27
Signal handling. Receipt of a signal triggers a control transfer to a signal handler. After it finishes processing, the handler returns control to the interrupted program.

delivered, but the resulting pending signal will not be received until the process unblocks the signal.

A pending signal is received at most once. For each process, the kernel maintains the set of pending signals in the `pending` bit vector, and the set of blocked signals in the `blocked` bit vector.[1] The kernel sets bit k in `pending` whenever a signal of type k is delivered and clears bit k in `pending` whenever a signal of type k is received.

8.5.2 Sending Signals

Unix systems provide a number of mechanisms for sending signals to processes. All of the mechanisms rely on the notion of a *process group*.

Process Groups

Every process belongs to exactly one *process group*, which is identified by a positive integer *process group ID*. The `getpgrp` function returns the process group ID of the current process.

```
#include <unistd.h>

pid_t getpgrp(void);
```
Returns: process group ID of calling process

By default, a child process belongs to the same process group as its parent. A process can change the process group of itself or another process by using the `setpgid` function:

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```
Returns: 0 on success, −1 on error

The `setpgid` function changes the process group of process `pid` to `pgid`. If `pid` is zero, the PID of the current process is used. If `pgid` is zero, the PID of the process specified by `pid` is used for the process group ID. For example, if process 15213 is the calling process, then

```
setpgid(0, 0);
```

creates a new process group whose process group ID is 15213, and adds process 15213 to this new group.

1. Also known as the *signal mask*.

Sending Signals with the `/bin/kill` Program

The `/bin/kill` program sends an arbitrary signal to another process. For example, the command

```
linux> /bin/kill -9 15213
```

sends signal 9 (SIGKILL) to process 15213. A negative PID causes the signal to be sent to every process in process group PID. For example, the command

```
linux> /bin/kill -9 -15213
```

sends a SIGKILL signal to every process in process group 15213. Note that we use the complete path `/bin/kill` here because some Unix shells have their own built-in `kill` command.

Sending Signals from the Keyboard

Unix shells use the abstraction of a *job* to represent the processes that are created as a result of evaluating a single command line. At any point in time, there is at most one foreground job and zero or more background jobs. For example, typing

```
linux> ls | sort
```

creates a foreground job consisting of two processes connected by a Unix pipe: one running the `ls` program, the other running the `sort` program. The shell creates a separate process group for each job. Typically, the process group ID is taken from one of the parent processes in the job. For example, Figure 8.28 shows a shell with one foreground job and two background jobs. The parent process in the foreground job has a PID of 20 and a process group ID of 20. The parent process has created two children, each of which are also members of process group 20.

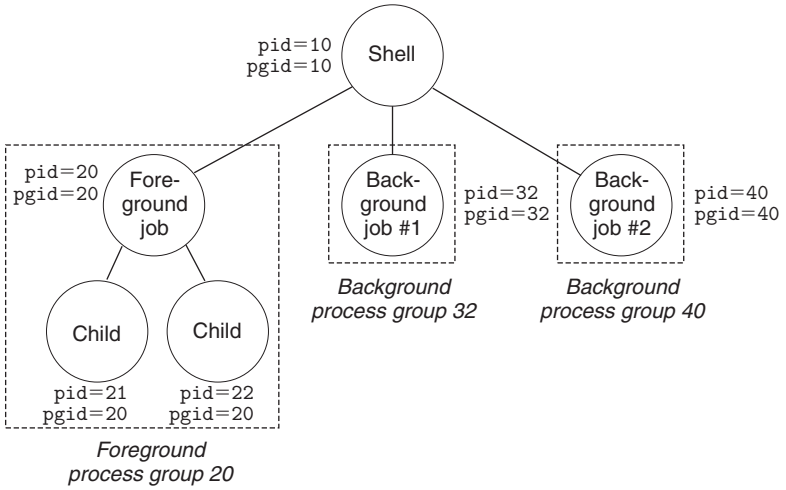


Figure 8.28
Foreground and background process groups.

Typing Ctrl+C at the keyboard causes the kernel to send a SIGINT signal to every process in the foreground process group. In the default case, the result is to terminate the foreground job. Similarly, typing Ctrl+Z causes the kernel to send a SIGTSTP signal to every process in the foreground process group. In the default case, the result is to stop (suspend) the foreground job.

Sending Signals with the `kill` Function

Processes send signals to other processes (including themselves) by calling the `kill` function.

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

int kill(pid_t pid, int sig);
```

Returns: 0 if OK, −1 on error

If `pid` is greater than zero, then the `kill` function sends signal number `sig` to process `pid`. If `pid` is equal to zero, then `kill` sends signal `sig` to every process in the process group of the calling process, including the calling process itself. If `pid` is less than zero, then `kill` sends signal `sig` to every process in process group |`pid`| (the absolute value of `pid`). Figure 8.29 shows an example of a parent that uses the `kill` function to send a SIGKILL signal to its child.

code/ecf/kill.c

```
1   #include "csapp.h"
2
3   int main()
4   {
5       pid_t pid;
6
7       /* Child sleeps until SIGKILL signal received, then dies */
8       if ((pid = Fork()) == 0) {
9           Pause();  /* Wait for a signal to arrive */
10          printf("control should never reach here!\n");
11          exit(0);
12      }
13
14      /* Parent sends a SIGKILL signal to a child */
15      Kill(pid, SIGKILL);
16      exit(0);
17  }
```

code/ecf/kill.c

Figure 8.29 **Using the `kill` function to send a signal to a child.**

Sending Signals with the `alarm` Function

A process can send SIGALRM signals to itself by calling the `alarm` function.

```
#include <unistd.h>

unsigned int alarm(unsigned int secs);
                Returns: remaining seconds of previous alarm, or 0 if no previous alarm
```

The `alarm` function arranges for the kernel to send a SIGALRM signal to the calling process in `secs` seconds. If `secs` is 0, then no new alarm is scheduled. In any event, the call to `alarm` cancels any pending alarms and returns the number of seconds remaining until any pending alarm was due to be delivered (had not this call to `alarm` canceled it), or 0 if there were no pending alarms.

8.5.3 Receiving Signals

When the kernel switches a process p from kernel mode to user mode (e.g., returning from a system call or completing a context switch), it checks the set of unblocked pending signals (`pending & ~blocked`) for p. If this set is empty (the usual case), then the kernel passes control to the next instruction (I_{next}) in the logical control flow of p. However, if the set is nonempty, then the kernel chooses some signal k in the set (typically the smallest k) and forces p to *receive* signal k. The receipt of the signal triggers some *action* by the process. Once the process completes the action, then control passes back to the next instruction (I_{next}) in the logical control flow of p. Each signal type has a predefined *default action*, which is one of the following:

- The process terminates.
- The process terminates and dumps core.
- The process stops (suspends) until restarted by a SIGCONT signal.
- The process ignores the signal.

Figure 8.26 shows the default actions associated with each type of signal. For example, the default action for the receipt of a SIGKILL is to terminate the receiving process. On the other hand, the default action for the receipt of a SIGCHLD is to ignore the signal. A process can modify the default action associated with a signal by using the `signal` function. The only exceptions are SIGSTOP and SIGKILL, whose default actions cannot be changed.

```
#include <signal.h>
typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
     Returns: pointer to previous handler if OK, SIG_ERR on error (does not set errno)
```

The `signal` function can change the action associated with a signal `signum` in one of three ways:

- If `handler` is SIG_IGN, then signals of type `signum` are ignored.
- If `handler` is SIG_DFL, then the action for signals of type `signum` reverts to the default action.
- Otherwise, `handler` is the address of a user-defined function, called a *signal handler*, that will be called whenever the process receives a signal of type `signum`. Changing the default action by passing the address of a handler to the `signal` function is known as *installing the handler*. The invocation of the handler is called *catching the signal*. The execution of the handler is referred to as *handling the signal*.

When a process catches a signal of type k, the handler installed for signal k is invoked with a single integer argument set to k. This argument allows the same handler function to catch different types of signals.

When the handler executes its `return` statement, control (usually) passes back to the instruction in the control flow where the process was interrupted by the receipt of the signal. We say "usually" because in some systems, interrupted system calls return immediately with an error.

Figure 8.30 shows a program that catches the SIGINT signal that is sent whenever the user types Ctrl+C at the keyboard. The default action for SIGINT

code/ecf/sigint.c

```
1   #include "csapp.h"
2
3   void sigint_handler(int sig) /* SIGINT handler */
4   {
5       printf("Caught SIGINT!\n");
6       exit(0);
7   }
8
9   int main()
10  {
11      /* Install the SIGINT handler */
12      if (signal(SIGINT, sigint_handler) == SIG_ERR)
13          unix_error("signal error");
14
15      pause(); /* Wait for the receipt of a signal */
16
17      return 0;
18  }
```

code/ecf/sigint.c

Figure 8.30 **A program that uses a signal handler to catch a SIGINT signal.**

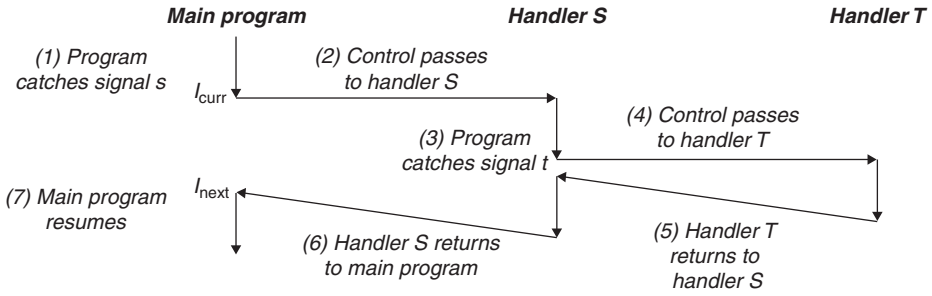


Figure 8.31 **Handlers can be interrupted by other handlers.**

is to immediately terminate the process. In this example, we modify the default behavior to catch the signal, print a message, and then terminate the process.

Signal handlers can be interrupted by other handlers, as shown in Figure 8.31. In this example, the main program catches signal s, which interrupts the main program and transfers control to handler S. While S is running, the program catches signal $t \neq s$, which interrupts S and transfers control to handler T. When T returns, S resumes where it was interrupted. Eventually, S returns, transferring control back to the main program, which resumes where it left off.

Practice Problem 8.7 (solution page 834)

Write a program called `snooze` that takes a single command-line argument, calls the `snooze` function from Problem 8.5 with this argument, and then terminates. Write your program so that the user can interrupt the `snooze` function by typing Ctrl+C at the keyboard. For example:

```
linux> ./snooze 5
CTRL+C                          User hits Crtl+C after 3 seconds
Slept for 3 of 5 secs.
linux>
```

8.5.4 Blocking and Unblocking Signals

Linux provides implicit and explicit mechanisms for blocking signals:

Implicit blocking mechanism. By default, the kernel blocks any pending signals of the type currently being processed by a handler. For example, in Figure 8.31, suppose the program has caught signal s and is currently running handler S. If another signal s is sent to the process, then s will become pending but will not be received until after handler S returns.

Explicit blocking mechanism. Applications can explicitly block and unblock selected signals using the `sigprocmask` function and its helpers.

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
```

Returns: 0 if OK, −1 on error

```
int sigismember(const sigset_t *set, int signum);
```

Returns: 1 if member, 0 if not, −1 on error

The `sigprocmask` function changes the set of currently blocked signals (the `blocked` bit vector described in Section 8.5.1). The specific behavior depends on the value of `how`:

SIG_BLOCK. Add the signals in `set` to `blocked` (`blocked = blocked | set`).

SIG_UNBLOCK. Remove the signals in `set` from `blocked` (`blocked = blocked & ~set`).

SIG_SETMASK. `blocked = set`.

If `oldset` is non-NULL, the previous value of the `blocked` bit vector is stored in `oldset`.

Signal sets such as `set` are manipulated using the following functions: The `sigemptyset` initializes `set` to the empty set. The `sigfillset` function adds every signal to `set`. The `sigaddset` function adds `signum` to `set`, `sigdelset` deletes `signum` from `set`, and `sigismember` returns 1 if `signum` is a member of `set`, and 0 if not.

For example, Figure 8.32 shows how you would use `sigprocmask` to temporarily block the receipt of SIGINT signals.

```
1       sigset_t mask, prev_mask;
2
3       Sigemptyset(&mask);
4       Sigaddset(&mask, SIGINT);
5
6       /* Block SIGINT and save previous blocked set */
7       Sigprocmask(SIG_BLOCK, &mask, &prev_mask);
8       . . . // Code region that will not be interrupted by SIGINT
9       /* Restore previous blocked set, unblocking SIGINT */
10      Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
11
```

Figure 8.32 **Temporarily blocking a signal from being received.**

8.5.5 Writing Signal Handlers

Signal handling is one of the thornier aspects of Linux system-level programming. Handlers have several attributes that make them difficult to reason about: (1) Handlers run concurrently with the main program and share the same global variables, and thus can interfere with the main program and with other handlers. (2) The rules for how and when signals are received is often counterintuitive. (3) Different systems can have different signal-handling semantics.

In this section, we address these issues and give you some basic guidelines for writing safe, correct, and portable signal handlers.

Safe Signal Handling

Signal handlers are tricky because they can run concurrently with the main program and with each other, as we saw in Figure 8.31. If a handler and the main program access the same global data structure concurrently, then the results can be unpredictable and often fatal.

We will explore concurrent programming in detail in Chapter 12. Our aim here is to give you some conservative guidelines for writing handlers that are safe to run concurrently. If you ignore these guidelines, you run the risk of introducing subtle concurrency errors. With such errors, your program works correctly most of the time. However, when it fails, it fails in unpredictable and unrepeatable ways that are horrendously difficult to debug. Forewarned is forearmed!

G0. Keep handlers as simple as possible. The best way to avoid trouble is to keep your handlers as small and simple as possible. For example, the handler might simply set a global flag and return immediately; all processing associated with the receipt of the signal is performed by the main program, which periodically checks (and resets) the flag.

G1. Call only async-signal-safe functions in your handlers. A function that is *async-signal-safe*, or simply *safe*, has the property that it can be safely called from a signal handler, either because it is *reentrant* (e.g., accesses only local variables; see Section 12.7.2), or because it cannot be interrupted by a signal handler. Figure 8.33 lists the system-level functions that Linux guarantees to be safe. Notice that many popular functions, such as `printf`, `sprintf`, `malloc`, and `exit`, are *not* on this list.

The only safe way to generate output from a signal handler is to use the `write` function (see Section 10.1). In particular, calling `printf` or `sprintf` is unsafe. To work around this unfortunate restriction, we have developed some safe functions, called the Sio (Safe I/O) package, that you can use to print simple messages from signal handlers.

_Exit	fexecve	poll	sigqueue
_exit	fork	posix_trace_event	sigset
abort	fstat	pselect	sigsuspend
accept	fstatat	raise	sleep
access	fsync	read	sockatmark
aio_error	ftruncate	readlink	socket
aio_return	futimens	readlinkat	socketpair
aio_suspend	getegid	recv	stat
alarm	geteuid	recvfrom	symlink
bind	getgid	recvmsg	symlinkat
cfgetispeed	getgroups	rename	tcdrain
cfgetospeed	getpeername	renameat	tcflow
cfsetispeed	getpgrp	rmdir	tcflush
cfsetospeed	getpid	select	tcgetattr
chdir	getppid	sem_post	tcgetpgrp
chmod	getsockname	send	tcsendbreak
chown	getsockopt	sendmsg	tcsetattr
clock_gettime	getuid	sendto	tcsetpgrp
close	kill	setgid	time
connect	link	setpgid	timer_getoverrun
creat	linkat	setsid	timer_gettime
dup	listen	setsockopt	timer_settime
dup2	lseek	setuid	times
execl	lstat	shutdown	umask
execle	mkdir	sigaction	uname
execv	mkdirat	sigaddset	unlink
execve	mkfifo	sigdelset	unlinkat
faccessat	mkfifoat	sigemptyset	utime
fchmod	mknod	sigfillset	utimensat
fchmodat	mknodat	sigismember	utimes
fchown	open	signal	wait
fchownat	openat	sigpause	waitpid
fcntl	pause	sigpending	write
fdatasync	pipe	sigprocmask	

Figure 8.33 **Async-signal-safe functions.** (Source: `man 7 signal`. Data from the Linux Foundation.)

```
#include "csapp.h"

ssize_t sio_putl(long v);
ssize_t sio_puts(char s[]);
```
Returns: number of bytes transferred if OK, −1 on error

```
void sio_error(char s[]);
```
Returns: nothing

The `sio_putl` and `sio_puts` functions emit a `long` and a string, respectively, to standard output. The `sio_error` function prints an error message and terminates.

Figure 8.34 shows the implementation of the Sio package, which uses two private reentrant functions from `csapp.c`. The `sio_strlen` function in line 3 returns the length of string `s`. The `sio_ltoa` function in line 10, which is based on the `itoa` function from [61], converts `v` to its base `b` string representation in `s`. The `_exit` function in line 17 is an async-signal-safe variant of `exit`.

Figure 8.35 shows a safe version of the SIGINT handler from Figure 8.30.

G2. Save and restore `errno`. Many of the Linux async-signal-safe functions set `errno` when they return with an error. Calling such functions inside a handler might interfere with other parts of the program that rely on `errno`.

code/src/csapp.c

```
1   ssize_t sio_puts(char s[]) /* Put string */
2   {
3       return write(STDOUT_FILENO, s, sio_strlen(s));
4   }
5
6   ssize_t sio_putl(long v) /* Put long */
7   {
8       char s[128];
9
10      sio_ltoa(v, s, 10); /* Based on K&R itoa() */
11      return sio_puts(s);
12  }
13
14  void sio_error(char s[]) /* Put error message and exit */
15  {
16      sio_puts(s);
17      _exit(1);
18  }
```

code/src/csapp.c

Figure 8.34 The Sio (Safe I/O) package for signal handlers.

code/ecf/sigintsafe.c

```
1  #include "csapp.h"
2
3  void sigint_handler(int sig) /* Safe SIGINT handler */
4  {
5      Sio_puts("Caught SIGINT!\n"); /* Safe output */
6      _exit(0);                     /* Safe exit */
7  }
```

code/ecf/sigintsafe.c

Figure 8.35 **A safe version of the SIGINT handler from Figure 8.30.**

The workaround is to save `errno` to a local variable on entry to the handler and restore it before the handler returns. Note that this is only necessary if the handler returns. It is not necessary if the handler terminates the process by calling `_exit`.

G3. Protect accesses to shared global data structures by blocking all signals. If a handler shares a global data structure with the main program or with other handlers, then your handlers and main program should temporarily block all signals while accessing (reading or writing) that data structure. The reason for this rule is that accessing a data structure d from the main program typically requires a sequence of instructions. If this instruction sequence is interrupted by a handler that accesses d, then the handler might find d in an inconsistent state, with unpredictable results. Temporarily blocking signals while you access d guarantees that a handler will not interrupt the instruction sequence.

G4. Declare global variables with `volatile`. Consider a handler and `main` routine that share a global variable g. The handler updates g, and `main` periodically reads g. To an optimizing compiler, it would appear that the value of g never changes in `main`, and thus it would be safe to use a copy of g that is cached in a register to satisfy every reference to g. In this case, the `main` function would never see the updated values from the handler.

You can tell the compiler not to cache a variable by declaring it with the `volatile` type qualifier. For example:

```
volatile int g;
```

The `volatile` qualifier forces the compiler to read the value of g from memory each time it is referenced in the code. In general, as with any shared data structure, each access to a global variable should be protected by temporarily blocking signals.

G5. Declare flags with `sig_atomic_t`. In one common handler design, the handler records the receipt of the signal by writing to a global *flag*. The main program periodically reads the flag, responds to the signal, and

clears the flag. For flags that are shared in this way, C provides an integer data type, `sig_atomic_t`, for which reads and writes are guaranteed to be *atomic* (uninterruptible) because they can be implemented with a single instruction:

```
volatile sig_atomic_t flag;
```

Since they can't be interrupted, you can safely read from and write to `sig_atomic_t` variables without temporarily blocking signals. Note that the guarantee of atomicity only applies to individual reads and writes. It does not apply to updates such as `flag++` or `flag = flag + 10`, which might require multiple instructions.

Keep in mind that the guidelines we have presented are conservative, in the sense that they are not always strictly necessary. For example, if you know that a handler can never modify `errno`, then you don't need to save and restore `errno`. Or if you can prove that no instance of `printf` can ever be interrupted by a handler, then it is safe to call `printf` from the handler. The same holds for accesses to shared global data structures. However, it is very difficult to prove such assertions in general. So we recommend that you take the conservative approach and follow the guidelines by keeping your handlers as simple as possible, calling safe functions, saving and restoring `errno`, protecting accesses to shared data structures, and using `volatile` and `sig_atomic_t`.

Correct Signal Handling

One of the nonintuitive aspects of signals is that pending signals are not queued. Because the `pending` bit vector contains exactly one bit for each type of signal, there can be at most one pending signal of any particular type. Thus, if two signals of type k are sent to a destination process while signal k is blocked because the destination process is currently executing a handler for signal k, then the second signal is simply discarded; it is not queued. The key idea is that the existence of a pending signal merely indicates that *at least* one signal has arrived.

To see how this affects correctness, let's look at a simple application that is similar in nature to real programs such as shells and Web servers. The basic structure is that a parent process creates some children that run independently for a while and then terminate. The parent must reap the children to avoid leaving zombies in the system. But we also want the parent to be free to do other work while the children are running. So we decide to reap the children with a SIGCHLD handler, instead of explicitly waiting for the children to terminate. (Recall that the kernel sends a SIGCHLD signal to the parent whenever one of its children terminates or stops.)

Figure 8.36 shows our first attempt. The parent installs a SIGCHLD handler and then creates three children. In the meantime, the parent waits for a line of input from the terminal and then processes it. This processing is modeled by an infinite loop. When each child terminates, the kernel notifies the parent by sending it a SIGCHLD signal. The parent catches the SIGCHLD, reaps one child,

code/ecf/signal1.c

```
1   /* WARNING: This code is buggy! */
2
3   void handler1(int sig)
4   {
5       int olderrno = errno;
6
7       if ((waitpid(-1, NULL, 0)) < 0)
8           sio_error("waitpid error");
9       Sio_puts("Handler reaped child\n");
10      Sleep(1);
11      errno = olderrno;
12  }
13
14  int main()
15  {
16      int i, n;
17      char buf[MAXBUF];
18
19      if (signal(SIGCHLD, handler1) == SIG_ERR)
20          unix_error("signal error");
21
22      /* Parent creates children */
23      for (i = 0; i < 3; i++) {
24          if (Fork() == 0) {
25              printf("Hello from child %d\n", (int)getpid());
26              exit(0);
27          }
28      }
29
30      /* Parent waits for terminal input and then processes it */
31      if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
32          unix_error("read");
33
34      printf("Parent processing input\n");
35      while (1)
36          ;
37
38      exit(0);
39  }
```

code/ecf/signal1.c

Figure 8.36 `signal1`. This program is flawed because it assumes that signals are queued.

does some additional cleanup work (modeled by the `sleep` statement), and then returns.

The `signal1` program in Figure 8.36 seems fairly straightforward. When we run it on our Linux system, however, we get the following output:

```
linux> ./signal1
Hello from child 14073
Hello from child 14074
Hello from child 14075
Handler reaped child
Handler reaped child
CR
Parent processing input
```

From the output, we note that although three SIGCHLD signals were sent to the parent, only two of these signals were received, and thus the parent only reaped two children. If we suspend the parent process, we see that, indeed, child process 14075 was never reaped and remains a zombie (indicated by the string `<defunct>` in the output of the `ps` command):

```
Ctrl+Z
Suspended
linux> ps t
  PID TTY      STAT   TIME COMMAND
  .
  .
  .
14072 pts/3    T      0:02 ./signal1
14075 pts/3    Z      0:00 [signal1] <defunct>
14076 pts/3    R+     0:00 ps t
```

What went wrong? The problem is that our code failed to account for the fact that signals are not queued. Here's what happened: The first signal is received and caught by the parent. While the handler is still processing the first signal, the second signal is delivered and added to the set of pending signals. However, since SIGCHLD signals are blocked by the SIGCHLD handler, the second signal is not received. Shortly thereafter, while the handler is still processing the first signal, the third signal arrives. Since there is already a pending SIGCHLD, this third SIGCHLD signal is discarded. Sometime later, after the handler has returned, the kernel notices that there is a pending SIGCHLD signal and forces the parent to receive the signal. The parent catches the signal and executes the handler a second time. After the handler finishes processing the second signal, there are no more pending SIGCHLD signals, and there never will be, because all knowledge of the third SIGCHLD has been lost. *The crucial lesson is that signals cannot be used to count the occurrence of events in other processes.*

To fix the problem, we must recall that the existence of a pending signal only implies that at least one signal has been delivered since the last time the process received a signal of that type. So we must modify the SIGCHLD handler to reap

code/ecf/signal2.c

```
1   void handler2(int sig)
2   {
3       int olderrno = errno;
4
5       while (waitpid(-1, NULL, 0) > 0) {
6           Sio_puts("Handler reaped child\n");
7       }
8       if (errno != ECHILD)
9           Sio_error("waitpid error");
10      Sleep(1);
11      errno = olderrno;
12  }
```

code/ecf/signal2.c

Figure 8.37 `signal2`. An improved version of Figure 8.36 that correctly accounts for the fact that signals are not queued.

as many zombie children as possible each time it is invoked. Figure 8.37 shows the modified SIGCHLD handler.

When we run `signal2` on our Linux system, it now correctly reaps all of the zombie children:

```
linux> ./signal2
Hello from child 15237
Hello from child 15238
Hello from child 15239
Handler reaped child
Handler reaped child
Handler reaped child
CR
Parent processing input
```

Practice Problem 8.8 (solution page 835)

What is the output of the following program?

code/ecf/signalprob0.c

```
1   volatile long counter = 2;
2
3   void handler1(int sig)
4   {
5       sigset_t mask, prev_mask;
6
7       Sigfillset(&mask);
8       Sigprocmask(SIG_BLOCK, &mask, &prev_mask);  /* Block sigs */
```

```
9       Sio_putl(--counter);
10      Sigprocmask(SIG_SETMASK, &prev_mask, NULL); /* Restore sigs */
11
12      _exit(0);
13  }
14
15  int main()
16  {
17      pid_t pid;
18      sigset_t mask, prev_mask;
19
20      printf("%ld", counter);
21      fflush(stdout);
22
23      signal(SIGUSR1, handler1);
24      if ((pid = Fork()) == 0) {
25          while(1) {};
26      }
27      Kill(pid, SIGUSR1);
28      Waitpid(-1, NULL, 0);
29
30      Sigfillset(&mask);
31      Sigprocmask(SIG_BLOCK, &mask, &prev_mask);  /* Block sigs */
32      printf("%ld", ++counter);
33      Sigprocmask(SIG_SETMASK, &prev_mask, NULL); /* Restore sigs */
34
35      exit(0);
36  }
```

code/ecf/signalprob0.c

Portable Signal Handling

Another ugly aspect of Unix signal handling is that different systems have different signal-handling semantics. For example:

- *The semantics of the* `signal` *function varies.* Some older Unix systems restore the action for signal k to its default after signal k has been caught by a handler. On these systems, the handler must explicitly reinstall itself, by calling `signal`, each time it runs.
- *System calls can be interrupted.* System calls such as `read`, `wait`, and `accept` that can potentially block the process for a long period of time are called *slow system calls*. On some older versions of Unix, slow system calls that are interrupted when a handler catches a signal do not resume when the signal handler returns but instead return immediately to the user with an error condition and `errno` set to EINTR. On these systems, programmers must include code that manually restarts interrupted system calls.

code/src/csapp.c

```
1   handler_t *Signal(int signum, handler_t *handler)
2   {
3       struct sigaction action, old_action;
4
5       action.sa_handler = handler;
6       sigemptyset(&action.sa_mask); /* Block sigs of type being handled */
7       action.sa_flags = SA_RESTART; /* Restart syscalls if possible */
8
9       if (sigaction(signum, &action, &old_action) < 0)
10          unix_error("Signal error");
11      return (old_action.sa_handler);
12  }
```

code/src/csapp.c

Figure 8.38 `Signal`. A wrapper for `sigaction` that provides portable signal handling on Posix-compliant systems.

To deal with these issues, the Posix standard defines the `sigaction` function, which allows users to clearly specify the signal-handling semantics they want when they install a handler.

```
#include <signal.h>

int sigaction(int signum,  struct sigaction *act,
              struct sigaction *oldact);
```

Returns: 0 if OK, −1 on error

The `sigaction` function is unwieldy because it requires the user to set the entries of a complicated structure. A cleaner approach, originally proposed by W. Richard Stevens [110], is to define a wrapper function, called `Signal`, that calls `sigaction` for us. Figure 8.38 shows the definition of `Signal`, which is invoked in the same way as the `signal` function.

The `Signal` wrapper installs a signal handler with the following signal-handling semantics:

- Only signals of the type currently being processed by the handler are blocked.
- As with all signal implementations, signals are not queued.
- Interrupted system calls are automatically restarted whenever possible.
- Once the signal handler is installed, it remains installed until `Signal` is called with a `handler` argument of either SIG_IGN or SIG_DFL.

We will use the `Signal` wrapper in all of our code.

8.5.6 Synchronizing Flows to Avoid Nasty Concurrency Bugs

The problem of how to program concurrent flows that read and write the same storage locations has challenged generations of computer scientists. In general, the number of potential interleavings of the flows is exponential in the number of instructions. Some of those interleavings will produce correct answers, and others will not. The fundamental problem is to somehow *synchronize* the concurrent flows so as to allow the largest set of feasible interleavings such that each of the feasible interleavings produces a correct answer.

Concurrent programming is a deep and important problem that we will discuss in more detail in Chapter 12. However, we can use what you've learned about exceptional control flow in this chapter to give you a sense of the interesting intellectual challenges associated with concurrency. For example, consider the program in Figure 8.39, which captures the structure of a typical Unix shell. The parent keeps track of its current children using entries in a global job list, with one entry per job. The `addjob` and `deletejob` functions add and remove entries from the job list.

After the parent creates a new child process, it adds the child to the job list. When the parent reaps a terminated (zombie) child in the SIGCHLD signal handler, it deletes the child from the job list.

At first glance, this code appears to be correct. Unfortunately, the following sequence of events is possible:

1. The parent executes the `fork` function and the kernel schedules the newly created child to run instead of the parent.
2. Before the parent is able to run again, the child terminates and becomes a zombie, causing the kernel to deliver a SIGCHLD signal to the parent.
3. Later, when the parent becomes runnable again but before it is executed, the kernel notices the pending SIGCHLD and causes it to be received by running the signal handler in the parent.
4. The signal handler reaps the terminated child and calls `deletejob`, which does nothing because the parent has not added the child to the list yet.
5. After the handler completes, the kernel then runs the parent, which returns from `fork` and incorrectly adds the (nonexistent) child to the job list by calling `addjob`.

Thus, for some interleavings of the parent's main routine and signal-handling flows, it is possible for `deletejob` to be called before `addjob`. This results in an incorrect entry on the job list, for a job that no longer exists and that will never be removed. On the other hand, there are also interleavings where events occur in the correct order. For example, if the kernel happens to schedule the parent to run when the `fork` call returns instead of the child, then the parent will correctly add the child to the job list before the child terminates and the signal handler removes the job from the list.

This is an example of a classic synchronization error known as a *race*. In this case, the race is between the call to `addjob` in the main routine and the call to

code/ecf/procmask1.c

```
1   /* WARNING: This code is buggy! */
2   void handler(int sig)
3   {
4       int olderrno = errno;
5       sigset_t mask_all, prev_all;
6       pid_t pid;
7
8       Sigfillset(&mask_all);
9       while ((pid = waitpid(-1, NULL, 0)) > 0) { /* Reap a zombie child */
10          Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
11          deletejob(pid); /* Delete the child from the job list */
12          Sigprocmask(SIG_SETMASK, &prev_all, NULL);
13      }
14      if (errno != ECHILD)
15          Sio_error("waitpid error");
16      errno = olderrno;
17  }
18
19  int main(int argc, char **argv)
20  {
21      int pid;
22      sigset_t mask_all, prev_all;
23
24      Sigfillset(&mask_all);
25      Signal(SIGCHLD, handler);
26      initjobs(); /* Initialize the job list */
27
28      while (1) {
29          if ((pid = Fork()) == 0) { /* Child process */
30              Execve("/bin/date", argv, NULL);
31          }
32          Sigprocmask(SIG_BLOCK, &mask_all, &prev_all); /* Parent process */
33          addjob(pid);  /* Add the child to the job list */
34          Sigprocmask(SIG_SETMASK, &prev_all, NULL);
35      }
36      exit(0);
37  }
```

code/ecf/procmask1.c

Figure 8.39 A shell program with a subtle synchronization error. If the child terminates before the parent is able to run, then `addjob` and `deletejob` will be called in the wrong order.

deletejob in the handler. If addjob wins the race, then the answer is correct. If not, the answer is incorrect. Such errors are enormously difficult to debug because it is often impossible to test every interleaving. You might run the code a billion times without a problem, but then the next test results in an interleaving that triggers the race.

Figure 8.40 shows one way to eliminate the race in Figure 8.39. By blocking SIGCHLD signals before the call to fork and then unblocking them only after we have called addjob, we guarantee that the child will be reaped *after* it is added to the job list. Notice that children inherit the blocked set of their parents, so we must be careful to unblock the SIGCHLD signal in the child before calling execve.

8.5.7 Explicitly Waiting for Signals

Sometimes a main program needs to explicitly wait for a certain signal handler to run. For example, when a Linux shell creates a foreground job, it must wait for the job to terminate and be reaped by the SIGCHLD handler before accepting the next user command.

Figure 8.41 shows the basic idea. The parent installs handlers for SIGINT and SIGCHLD and then enters an infinite loop. It blocks SIGCHLD to avoid the race between parent and child that we discussed in Section 8.5.6. After creating the child, it resets pid to zero, unblocks SIGCHLD, and then waits in a spin loop for pid to become nonzero. After the child terminates, the handler reaps it and assigns its nonzero PID to the global pid variable. This terminates the spin loop, and the parent continues with additional work before starting the next iteration.

While this code is correct, the spin loop is wasteful of processor resources. We might be tempted to fix this by inserting a pause in the body of the spin loop:

```
while (!pid)  /* Race! */
    pause();
```

Notice that we still need a loop because pause might be interrupted by the receipt of one or more SIGINT signals. However, this code has a serious race condition: if the SIGCHLD is received after the while test but before the pause, the pause will sleep forever.

Another option is to replace the pause with sleep:

```
while (!pid) /* Too slow! */
    sleep(1);
```

While correct, this code is too slow. If the signal is received after the while and before the sleep, the program must wait a (relatively) long time before it can check the loop termination condition again. Using a higher-resolution sleep function such as nanosleep isn't acceptable, either, because there is no good rule for determining the sleep interval. Make it too small and the loop is too wasteful. Make it too high and the program is too slow.

code/ecf/procmask2.c

```
1   void handler(int sig)
2   {
3       int olderrno = errno;
4       sigset_t mask_all, prev_all;
5       pid_t pid;
6
7       Sigfillset(&mask_all);
8       while ((pid = waitpid(-1, NULL, 0)) > 0) { /* Reap a zombie child */
9           Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
10          deletejob(pid); /* Delete the child from the job list */
11          Sigprocmask(SIG_SETMASK, &prev_all, NULL);
12      }
13      if (errno != ECHILD)
14          Sio_error("waitpid error");
15      errno = olderrno;
16  }
17
18  int main(int argc, char **argv)
19  {
20      int pid;
21      sigset_t mask_all, mask_one, prev_one;
22
23      Sigfillset(&mask_all);
24      Sigemptyset(&mask_one);
25      Sigaddset(&mask_one, SIGCHLD);
26      Signal(SIGCHLD, handler);
27      initjobs(); /* Initialize the job list */
28
29      while (1) {
30          Sigprocmask(SIG_BLOCK, &mask_one, &prev_one); /* Block SIGCHLD */
31          if ((pid = Fork()) == 0) { /* Child process */
32              Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
33              Execve("/bin/date", argv, NULL);
34          }
35          Sigprocmask(SIG_BLOCK, &mask_all, NULL); /* Parent process */
36          addjob(pid);  /* Add the child to the job list */
37          Sigprocmask(SIG_SETMASK, &prev_one, NULL);  /* Unblock SIGCHLD */
38      }
39      exit(0);
40  }
```

code/ecf/procmask2.c

Figure 8.40 **Using `sigprocmask` to synchronize processes.** In this example, the parent ensures that `addjob` executes before the corresponding `deletejob`.

code/ecf/waitforsignal.c

```
1   #include "csapp.h"
2
3   volatile sig_atomic_t pid;
4
5   void sigchld_handler(int s)
6   {
7       int olderrno = errno;
8       pid = waitpid(-1, NULL, 0);
9       errno = olderrno;
10  }
11
12  void sigint_handler(int s)
13  {
14  }
15
16  int main(int argc, char **argv)
17  {
18      sigset_t mask, prev;
19
20      Signal(SIGCHLD, sigchld_handler);
21      Signal(SIGINT, sigint_handler);
22      Sigemptyset(&mask);
23      Sigaddset(&mask, SIGCHLD);
24
25      while (1) {
26          Sigprocmask(SIG_BLOCK, &mask, &prev); /* Block SIGCHLD */
27          if (Fork() == 0) /* Child */
28              exit(0);
29
30          /* Parent */
31          pid = 0;
32          Sigprocmask(SIG_SETMASK, &prev, NULL); /* Unblock SIGCHLD */
33
34          /* Wait for SIGCHLD to be received (wasteful) */
35          while (!pid)
36              ;
37
38          /* Do some work after receiving SIGCHLD */
39          printf(".");
40      }
41      exit(0);
42  }
```

code/ecf/waitforsignal.c

Figure 8.41 Waiting for a signal with a spin loop. This code is correct, but the spin loop is wasteful.

The proper solution is to use `sigsuspend`.

```
#include <signal.h>

int sigsuspend(const sigset_t *mask);
                                                        Returns: −1
```

The `sigsuspend` function temporarily replaces the current blocked set with `mask` and then suspends the process until the receipt of a signal whose action is either to run a handler or to terminate the process. If the action is to terminate, then the process terminates without returning from `sigsuspend`. If the action is to run a handler, then `sigsuspend` returns after the handler returns, restoring the blocked set to its state when `sigsuspend` was called.

The `sigsuspend` function is equivalent to an *atomic* (uninterruptible) version of the following:

```
1       sigprocmask(SIG_BLOCK, &mask, &prev);
2       pause();
3       sigprocmask(SIG_SETMASK, &prev, NULL);
```

The atomic property guarantees that the calls to `sigprocmask` (line 1) and `pause` (line 2) occur together, without being interrupted. This eliminates the potential race where a signal is received after the call to `sigprocmask` and before the call to `pause`.

Figure 8.42 shows how we would use `sigsuspend` to replace the spin loop in Figure 8.41. Before each call to `sigsuspend`, SIGCHLD is blocked. The `sigsuspend` temporarily unblocks SIGCHLD, and then sleeps until the parent catches a signal. Before returning, it restores the original blocked set, which blocks SIGCHLD again. If the parent caught a SIGINT, then the loop test succeeds and the next iteration calls `sigsuspend` again. If the parent caught a SIGCHLD, then the loop test fails and we exit the loop. At this point, SIGCHLD is blocked, and so we can optionally unblock SIGCHLD. This might be useful in a real shell with background jobs that need to be reaped.

The `sigsuspend` version is less wasteful than the original spin loop, avoids the race introduced by `pause`, and is more efficient than `sleep`.

8.6 Nonlocal Jumps

C provides a form of user-level exceptional control flow, called a *nonlocal jump*, that transfers control directly from one function to another currently executing function without having to go through the normal call-and-return sequence. Nonlocal jumps are provided by the `setjmp` and `longjmp` functions.

code/ecf/sigsuspend.c

```
1   #include "csapp.h"
2
3   volatile sig_atomic_t pid;
4
5   void sigchld_handler(int s)
6   {
7       int olderrno = errno;
8       pid = Waitpid(-1, NULL, 0);
9       errno = olderrno;
10  }
11
12  void sigint_handler(int s)
13  {
14  }
15
16  int main(int argc, char **argv)
17  {
18      sigset_t mask, prev;
19
20      Signal(SIGCHLD, sigchld_handler);
21      Signal(SIGINT, sigint_handler);
22      Sigemptyset(&mask);
23      Sigaddset(&mask, SIGCHLD);
24
25      while (1) {
26          Sigprocmask(SIG_BLOCK, &mask, &prev); /* Block SIGCHLD */
27          if (Fork() == 0) /* Child */
28              exit(0);
29
30          /* Wait for SIGCHLD to be received */
31          pid = 0;
32          while (!pid)
33              sigsuspend(&prev);
34
35          /* Optionally unblock SIGCHLD */
36          Sigprocmask(SIG_SETMASK, &prev, NULL);
37
38          /* Do some work after receiving SIGCHLD */
39          printf(".");
40      }
41      exit(0);
42  }
```

code/ecf/sigsuspend.c

Figure 8.42 **Waiting for a signal with** `sigsuspend`.

```
#include <setjmp.h>

int setjmp(jmp_buf env);
int sigsetjmp(sigjmp_buf env, int savesigs);
                                    Returns: 0 from setjmp, nonzero from longjmps
```

The `setjmp` function saves the current *calling environment* in the `env` buffer, for later use by `longjmp`, and returns 0. The calling environment includes the program counter, stack pointer, and general-purpose registers. For subtle reasons beyond our scope, the value that `setjmp` returns should not be assigned to a variable:

```
rc = setjmp(env);  /* Wrong! */
```

However, it can be safely used as a test in a `switch` or conditional statement [62].

```
#include <setjmp.h>

void longjmp(jmp_buf env, int retval);
void siglongjmp(sigjmp_buf env, int retval);
                                                                    Never returns
```

The `longjmp` function restores the calling environment from the `env` buffer and then triggers a return from the most recent `setjmp` call that initialized `env`. The `setjmp` then returns with the nonzero return value `retval`.

The interactions between `setjmp` and `longjmp` can be confusing at first glance. The `setjmp` function is called once but returns *multiple times:* once when the `setjmp` is first called and the calling environment is stored in the `env` buffer, and once for each corresponding `longjmp` call. On the other hand, the `longjmp` function is called once but never returns.

An important application of nonlocal jumps is to permit an immediate return from a deeply nested function call, usually as a result of detecting some error condition. If an error condition is detected deep in a nested function call, we can use a nonlocal jump to return directly to a common localized error handler instead of laboriously unwinding the call stack.

Figure 8.43 shows an example of how this might work. The `main` routine first calls `setjmp` to save the current calling environment, and then calls function `foo`, which in turn calls function `bar`. If `foo` or `bar` encounter an error, they return immediately from the `setjmp` via a `longjmp` call. The nonzero return value of the `setjmp` indicates the error type, which can then be decoded and handled in one place in the code.

The feature of `longjmp` that allows it to skip up through all intermediate calls can have unintended consequences. For example, if some data structures were allocated in the intermediate function calls with the intention to deallocate them at the end of the function, the deallocation code gets skipped, thus creating a memory leak.

code/ecf/setjmp.c

```
1   #include "csapp.h"
2
3   jmp_buf buf;
4
5   int error1 = 0;
6   int error2 = 1;
7
8   void foo(void), bar(void);
9
10  int main()
11  {
12      switch(setjmp(buf)) {
13      case 0:
14          foo();
15          break;
16      case 1:
17          printf("Detected an error1 condition in foo\n");
18          break;
19      case 2:
20          printf("Detected an error2 condition in foo\n");
21          break;
22      default:
23          printf("Unknown error condition in foo\n");
24      }
25      exit(0);
26  }
27
28  /* Deeply nested function foo */
29  void foo(void)
30  {
31      if (error1)
32          longjmp(buf, 1);
33      bar();
34  }
35
36  void bar(void)
37  {
38      if (error2)
39          longjmp(buf, 2);
40  }
```

code/ecf/setjmp.c

Figure 8.43 Nonlocal jump example. This example shows the framework for using nonlocal jumps to recover from error conditions in deeply nested functions without having to unwind the entire stack.

code/ecf/restart.c

```
1   #include "csapp.h"
2
3   sigjmp_buf buf;
4
5   void handler(int sig)
6   {
7       siglongjmp(buf, 1);
8   }
9
10  int main()
11  {
12      if (!sigsetjmp(buf, 1)) {
13          Signal(SIGINT, handler);
14          Sio_puts("starting\n");
15      }
16      else
17          Sio_puts("restarting\n");
18
19      while(1) {
20          Sleep(1);
21          Sio_puts("processing...\n");
22      }
23      exit(0); /* Control never reaches here */
24  }
```

code/ecf/restart.c

Figure 8.44 A program that uses nonlocal jumps to restart itself when the user types Ctrl+C.

Another important application of nonlocal jumps is to branch out of a signal handler to a specific code location, rather than returning to the instruction that was interrupted by the arrival of the signal. Figure 8.44 shows a simple program that illustrates this basic technique. The program uses signals and nonlocal jumps to do a soft restart whenever the user types Ctrl+C at the keyboard. The `sigsetjmp` and `siglongjmp` functions are versions of `setjmp` and `longjmp` that can be used by signal handlers.

The initial call to the `sigsetjmp` function saves the calling environment and signal context (including the pending and blocked signal vectors) when the program first starts. The main routine then enters an infinite processing loop. When the user types Ctrl+C, the kernel sends a SIGINT signal to the process, which catches it. Instead of returning from the signal handler, which would pass control back to the interrupted processing loop, the handler performs a nonlocal jump back to the beginning of the `main` program. When we run the program on our system, we get the following output:

Aside Software exceptions in C++ and Java

The exception mechanisms provided by C++ and Java are higher-level, more structured versions of the C `setjmp` and `longjmp` functions. You can think of a `catch` clause inside a `try` statement as being akin to a `setjmp` function. Similarly, a `throw` statement is similar to a `longjmp` function.

```
linux> ./restart
starting
processing...
processing...
Ctrl+C
restarting
processing...
Ctrl+C
restarting
processing...
```

There a couple of interesting things about this program. First, To avoid a race, we must install the handler *after* we call `sigsetjmp`. If not, we would run the risk of the handler running before the initial call to `sigsetjmp` sets up the calling environment for `siglongjmp`. Second, you might have noticed that the `sigsetjmp` and `siglongjmp` functions are not on the list of async-signal-safe functions in Figure 8.33. The reason is that in general `siglongjmp` can jump into arbitrary code, so we must be careful to call only safe functions in any code reachable from a `siglongjmp`. In our example, we call the safe `sio_puts` and `sleep` functions. The unsafe `exit` function is unreachable.

8.7 Tools for Manipulating Processes

Linux systems provide a number of useful tools for monitoring and manipulating processes:

STRACE. Prints a trace of each system call invoked by a running program and its children. It is a fascinating tool for the curious student. Compile your program with `-static` to get a cleaner trace without a lot of output related to shared libraries.

PS. Lists processes (including zombies) currently in the system.

TOP. Prints information about the resource usage of current processes.

PMAP. Displays the memory map of a process.

`/proc`. A virtual filesystem that exports the contents of numerous kernel data structures in an ASCII text form that can be read by user programs. For example, type `cat /proc/loadavg` to see the current load average on your Linux system.

8.8 Summary

Exceptional control flow (ECF) occurs at all levels of a computer system and is a basic mechanism for providing concurrency in a computer system.

At the hardware level, exceptions are abrupt changes in the control flow that are triggered by events in the processor. The control flow passes to a software handler, which does some processing and then returns control to the interrupted control flow.

There are four different types of exceptions: interrupts, faults, aborts, and traps. Interrupts occur asynchronously (with respect to any instructions) when an external I/O device such as a timer chip or a disk controller sets the interrupt pin on the processor chip. Control returns to the instruction following the faulting instruction. Faults and aborts occur synchronously as the result of the execution of an instruction. Fault handlers restart the faulting instruction, while abort handlers never return control to the interrupted flow. Finally, traps are like function calls that are used to implement the system calls that provide applications with controlled entry points into the operating system code.

At the operating system level, the kernel uses ECF to provide the fundamental notion of a process. A process provides applications with two important abstractions: (1) logical control flows that give each program the illusion that it has exclusive use of the processor, and (2) private address spaces that provide the illusion that each program has exclusive use of the main memory.

At the interface between the operating system and applications, applications can create child processes, wait for their child processes to stop or terminate, run new programs, and catch signals from other processes. The semantics of signal handling is subtle and can vary from system to system. However, mechanisms exist on Posix-compliant systems that allow programs to clearly specify the expected signal-handling semantics.

Finally, at the application level, C programs can use nonlocal jumps to bypass the normal call/return stack discipline and branch directly from one function to another.

Bibliographic Notes

Kerrisk is the essential reference for all aspects of programming in the Linux environment [62]. The Intel ISA specification contains a detailed discussion of exceptions and interrupts on Intel processors [50]. Operating systems texts [102, 106, 113] contain additional information on exceptions, processes, and signals. The classic work by W. Richard Stevens [111] is a valuable and highly readable description of how to work with processes and signals from application programs. Bovet and Cesati [11] give a wonderfully clear description of the Linux kernel, including details of the process and signal implementations.

Homework Problems

8.9 ◆

Consider four processes with the following starting and ending times:

Process	Start time	End time
A	6	8
B	3	5
C	4	7
D	2	9

For each pair of processes, indicate whether they run concurrently (Y) or not (N):

Process pair	Concurrent?
AB	_____
AC	_____
AD	_____
BC	_____
BD	_____
CD	_____

8.10 ◆

In this chapter, we have introduced some functions with unusual call and return behaviors: `getenv`, `setenv`, `unsetenv`, and `execve`. Match each function with one of the following behaviors:

A. Called once, returns only if there is an error

B. Called once, returns nothing

C. Called once, returns either a pointer or `NULL`

8.11 ◆

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

code/ecf/global-forkprob1.c

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

code/ecf/global-forkprob1.c