8.2 Processes

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

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

The classic definition of a process is an instance of a program in execution. Each program in the system runs in the context of some process. The context consists of the state that the program needs to run correctly. This state includes the program’s code and data stored in memory, its stack, the contents of its general-purpose registers, its program counter, environment variables, and the set of open file descriptors.

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

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

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

Let’s look more closely at these abstractions.

8.2.1 Logical Control Flow

A process provides each program with the illusion that it has exclusive use of the processor, even though many other programs are typically running 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
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. 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 awhile, 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**

Consider three processes with the following starting and ending times:

<table>
<thead>
<tr>
<th>Process</th>
<th>Start time</th>
<th>End time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Process pair</th>
<th>Concurrent?</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td></td>
</tr>
<tr>
<td>AC</td>
<td></td>
</tr>
<tr>
<td>BC</td>
<td></td>
</tr>
</tbody>
</table>

**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 Linux process. The bottom portion of the address space is reserved for the user program, with the usual text, data, heap, and stack segments. Code segments begin at address $0x08048000$ for 32-bit processes, and at address $0x00400000$ for 64-bit processes. The top portion of the address space is reserved for the kernel. 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 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 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. During
the first part of the switch, the kernel is executing instructions in kernel mode on
behalf of process A. Then at some point it begins executing instructions (still in
kernel mode) on behalf of process B. And after the switch, the kernel is executing
instructions in user mode on behalf of process B.

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

**Aside**  Cache pollution and exceptional control flow

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

### 8.3 System Call Error Handling

When Unix system-level functions encounter an error, they typically return −1
and set the global integer variable \texttt{errno} to indicate what went wrong. Program-
mers 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 \texttt{fork} function:

```c
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:

```c
void unix_error(char *msg) /* Unix-style error */
{
    fprintf(stderr, "%s: %s\n", msg, strerror(errno));
    exit(0);
}
```

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

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

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

```c
pid_t Fork(void)
{
    pid_t pid;
    if ((pid = fork()) < 0)
        unix_error("Fork error");
    return pid;
}
```

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

```c
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).

```c
#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 is 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 can begin running again. (A signal is a form of software interrupt that is described in detail in Section 8.5.)
- **Terminated.** The process is stopped permanently. A process becomes terminated for one of three reasons: (1) receiving a signal whose default action is to terminate the process, (2) returning from the main routine, or (3) calling the exit function:

```c
#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.

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

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

Figure 8.15 shows a simple example of a parent process that uses fork to create a child process. When the fork call returns in line 8, x has a value of 1 in both the parent and child. The child increments and prints its copy of x in line 10. Similarly, the parent decrements and prints its copy of x in line 15.

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

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

There are some subtle aspects to this simple example.

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

- **Concurrent execution.** The parent and the child are separate processes that run concurrently. The instructions in their logical control flows can be interleaved by the kernel in an arbitrary way. When we run the program on our system, the parent process completes its printf statement first, followed by the child. However, on another system the reverse might be true. In general, as programmers we can never make assumptions about the interleaving of the instructions in different processes.
Duplicate but separate address spaces. If we could halt both the parent and the child immediately after the fork function returned in each process, we would see that the address space of each process is identical. Each process has the same user stack, the same local variable values, the same heap, the same global variable values, and the same code. Thus, in our example program, the parent and the child are separate processes, they each have their own private address spaces. Any subsequent changes that a parent or child makes to x are private and are not reflected in the memory of the other process. This is why the variable x has different values in the parent and child when they call their respective printf statements.

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

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

For example, how many lines of output would the program in Figure 8.16(a) generate? Figure 8.16(b) shows the corresponding process graph. The parent
creates a child when it executes the first (and only) \texttt{fork} in the program. Each of these calls \texttt{printf} once, so the program prints two output lines.

Now what if we were to call \texttt{fork} twice, as shown in Figure 8.16(c)? As we see from Figure 8.16(d), the parent calls \texttt{fork} to create a child, and then the parent and child each call \texttt{fork}, which results in two more processes. Thus, there are four processes, each of which calls \texttt{printf}, so the program generates four output lines.
Continuing this line of thought, what would happen if we were to call `fork` three times, as in Figure 8.16(e)? As we see from the process graph in Figure 8.16(f), there are a total of eight processes. Each process calls `printf`, so the program produces eight output lines.

**Practice Problem 8.2**

Consider the following program:

```c
#include "csapp.h"

int main()
{
    int x = 1;
    if (Fork() == 0)
        printf("printf1: x=%d\n", ++x);
    printf("printf2: x=%d\n", --x);
    exit(0);
}
```

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*.

**Aside** Why are terminated children called zombies?

In folklore, a zombie is a living corpse, an entity that is half alive and half dead. A zombie process is similar in the sense that while it has already terminated, the kernel maintains some of its state until it can be reaped by the parent.

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

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

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

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

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

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

### Determining the Members of the Wait Set

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

- If `pid > 0`, then the wait set is the singleton child process whose process ID is equal to `pid`.
- 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, that we will not discuss.

### Modifying the Default Behavior

The default behavior can be modified by setting `options` to various combinations of the `WNOHANG` and `WUNTRACED` 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.
- **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 status argument is non-NULL, then waitpid encodes status information about the child that caused the return in the status argument. 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. (Signals are explained in Section 8.5.)
- **WTERMSIG**(status): Returns the number of the signal that caused the child process to terminate. This status is only defined if WIFSIGNALED(status) 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(status) returned true.

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.

**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:

```c
/* 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:

```c
#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.
**Practice Problem 8.3**

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

```c
int main()
{
    if (Fork() == 0) {
        printf("a");
    }
    else {
        printf("b");
        waitpid(-1, NULL, 0);
    }
    printf("c");
    exit(0);
}
```

**The wait Function**

The wait function is a simpler version of waitpid:

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

pid_t wait(int *status);
```

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 8.17 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. 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.
```c
#include "csapp.h"

#define N 2

int main()
{
    int status, i;
    pid_t pid;

    /* Parent creates N children */
    for (i=0; i<N; i++)
        if ((pid = Fork()) == 0) /* Child */
            exit(100+i);

    /* Parent reaps N children in no particular order */
    while ((pid = waitpid(-1, &status, 0)) > 0) {
        if (WIFEXITED(status))
            printf("child %d terminated normally with exit status=%d\n", pid, WEXITSTATUS(status));
        else
            printf("child %d terminated abnormally\n", pid);
    }

    /* The only normal termination is if there are no more children */
    if (errno != ECHILD)
        unix_error("waitpid error");

    exit(0);
}
```

**Figure 8.17** Using the `waitpid` function to reap zombie children in no particular order.

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 Unix system, it produces the following output:

```
unix> ./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
#include "csapp.h"
#define N 2

int main()
{
    int status, i;
    pid_t pid[N], retpid;

    /* Parent creates N children */
    for (i=0; i<N; i++)
        if ((pid[i] = Fork()) == 0) /* Child */
            exit(100+i);

    /* Parent reaps N children in order */
    i = 0;
    while ((retpid = waitpid(pid[i++], &status, 0)) > 0) {
        if (WIFEXITED(status))
            printf("child %d terminated normally with exit status=%d\n", retpid, WEXITSTATUS(status));
        else
            printf("child %d terminated abnormally\n", retpid);
    }

    /* The only normal termination is if there are no more children */
    if (errno != ECHILD)
        unix_error("waitpid error");
    exit(0);
}

Figure 8.18 Using waitpid to reap zombie children in the order they were created.

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 8.18 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

Consider the following program:

```c
int main()
{
    int status;
    pid_t pid;

    printf("Hello\n");
    pid = Fork();
    printf("%d\n", !pid);
    if (pid != 0) {
        if (waitpid(-1, &status, 0) > 0) {
            if (WIFEXITED(status) != 0)
                printf("%d\n", WEXITSTATUS(status));
        }
    }
    printf("Bye\n");
    exit(2);
}
```

A. How many output lines does this program generate?
B. What is one possible ordering of these output lines?

8.4.4 Putting Processes to Sleep

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

```c
#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.

```c
#include <unistd.h>
int pause(void);
```

**Practice Problem 8.5**

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

```c
unsigned int snooze(unsigned int secs);
```

The `snooze` function behaves exactly as the `sleep` function, except that it prints a message describing how long the process actually slept:

```
Slept for 4 of 5 secs.
```

### 8.4.5 Loading and Running Programs

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

```c
#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 8.19. 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 8.20. 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”.

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

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

or equivalently,

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

When main begins executing in a 32-bit Linux process, the user stack has the organization shown in Figure 8.21. 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, which are stored contiguously on the stack, one after the other without any gaps. 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 immediately by the null-terminated `argv[]` array, with each element pointing to an argument string on the stack. At the top of the stack are the three arguments to the `main` routine: (1) `envp`, which points to the `envp[]` array, (2) `argv`, which points to the `argv[]` array, and (3) `argc`, which gives the number of non-null pointers in the `argv[]` array.

Unix provides several functions for manipulating the environment array:

```c
#include <stdlib.h>

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

#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
```

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

```c
#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.

**Aside** Programs vs. 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 modules 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.

### Practice Problem 8.6

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

```
unix> ./myecho arg1 arg2
Command line 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 (Chapter 11) 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 8.22 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 8.23 shows the code that evaluates the command line. Its first task is to call the `parseline` function (Figure 8.24), 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
Figure 8.22 The main routine for a simple shell program.

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).

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.
/* eval - Evaluate a command line */
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE]; /* Holds modified command line */
    int bg; /* Should the job run in bg or fg? */
    pid_t pid; /* Process id */

    strcpy(buf, cmdline);
    bg = parseLine(buf, argv);
    if (argv[0] == NULL)
        return; /* Ignore empty lines */

    if (!builtin_command(argv)) {
        if (((pid = Fork()) == 0) { /* Child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }
        else
            printf("%d %s", pid, cmdline);
    }
    return;
}

/* If first arg is a builtin command, run it and return true */
int builtin_command(char **argv)
{
    if (!strcmp(argv[0], "quit")) /* quit command */
        exit(0);
    if (!strcmp(argv[0], ".") /* Ignore singleton & */)  /* Not a builtin command */
        return 0;
}
/* parseline - Parse the command line and build the argv array */

int parseline(char *buf, char **argv)
{
    char *delim;  /* Points to first space delimiter */
    int argc;    /* Number of args */
    int bg;      /* Background job? */

    buf[strlen(buf)-1] = ' '; /* Replace trailing '\n' with space */
    while (*buf&& (*buf == ' ')) /* Ignore leading spaces */
        buf++;

    /* Build the argv list */
    argc = 0;
    while ((delim = strchr(buf, ' '))) {
        argv[argc++] = buf;
        *delim = '\0';
        buf = delim + 1;
        while (*buf&& (*buf == ' ')) /* Ignore spaces */
            buf++;
    }
    argv[argc] = NULL;

    if (argc == 0) /* Ignore blank line */
        return 1;

    /* Should the job run in the background? */
    if (((bg = (*argv[--argc] == '&')) != 0)
        argv[--argc] = NULL;
    return bg;
}

Figure 8.24 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 Unix signal, that allows processes and the kernel to interrupt other processes.
### Section 8.5 Signals

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

Figure 8.25  **Linux signals.** Notes: (1) 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. (2) This signal can neither be caught nor ignored.

A signal is a small message that notifies a process that an event of some type has occurred in the system. For example, Figure 8.25 shows the 30 different types of signals that are supported on Linux systems. Typing “man 7 signal” on the shell command line gives the list.

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 a `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 the foreground process. 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 8.26 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 delivered, but the resulting pending signal will not be received until the process unblocks the signal.

Figure 8.26

**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.
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. 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.

```c
#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:

```c
#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

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

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

#### Sending Signals with the `/bin/kill` Program

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

```
unix> /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

```
unix> /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

```
unix> 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 8.27 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.

Typing `ctrl-c` at the keyboard causes a SIGINT signal to be sent to the shell. The shell catches the signal (see Section 8.5.3) and then sends a SIGINT to every process in the foreground process group. In the default case, the result is
to terminate the foreground job. Similarly, typing `crtl-z` sends a SIGTSTP signal to the shell, which catches it and sends 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.

```c
#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 less than zero, then `kill` sends signal `sig` to every process in process group `abs(pid)`. Figure 8.28 shows an example of a parent that uses the `kill` function to send a SIGKILL signal to its child.

```c
#include "csapp.h"

int main()
{
    pid_t pid;

    /* Child sleeps until SIGKILL signal received, then dies */
    if ((pid = Fork()) == 0) {
        Pause(); /* Wait for a signal to arrive */
        printf("control should never reach here!\n");
        exit(0);
    }

    /* Parent sends a SIGKILL signal to a child */
    Kill(pid, SIGKILL);
    exit(0);
}
```

Figure 8.28 Using the `kill` function to send a signal to a child.