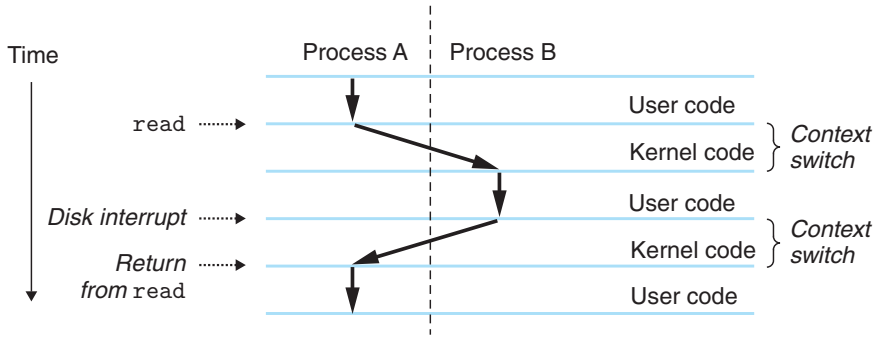


Figure 8.14
Anatomy of a process context switch.

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 `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*. 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 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:

```
#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 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.

code/ecf/fork.c

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

code/ecf/fork.c

Figure 8.15 **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 8. However, since the parent and the child are separate processes, they each have their own private address spaces. Any subsequent changes that a parent or child makes to `x` are private and are not reflected in the memory of the other process. This is why the variable `x` has different values in the parent and child when they call their respective `printf` statements.
- *Shared files.* When we run the example program, we notice that both parent and child print their output on the screen. The reason is that the child inherits all of the parent's open files. When the parent calls `fork`, the `stdout` file is open and directed to the screen. The child inherits this file and thus its output is also directed to the screen.

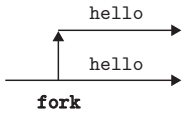
When you are first learning about the `fork` function, it is often helpful to sketch the *process graph*, where each horizontal arrow corresponds to a process that executes instructions from left to right, and each vertical arrow corresponds to the execution of a `fork` function.

For example, how many lines of output would the program in Figure 8.16(a) generate? Figure 8.16(b) shows the corresponding process graph. The parent

(a) Calls `fork` once

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

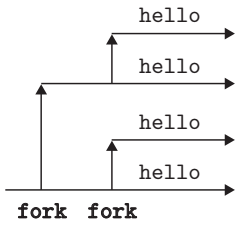
(b) Prints two output lines



(c) Calls `fork` twice

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

(d) Prints four output lines



(e) Calls `fork` three times

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

(f) Prints eight output lines

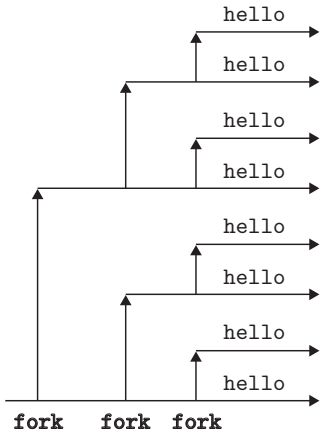


Figure 8.16 **Examples of `fork` programs and their process graphs.**

creates a child when it executes the first (and only) `fork` in the program. Each of these calls `printf` once, so the program prints two output lines.

Now what if we were to call `fork` twice, as shown in Figure 8.16(c)? As we see from Figure 8.16(d), the parent calls `fork` to create a child, and then the parent and child each call `fork`, which results in two more processes. Thus, there are four processes, each of which calls `printf`, so the program generates four output lines.

Continuing this line of thought, what would happen if we were to call `fork` three times, as in Figure 8.16(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:

code/ecf/forkprob0.c

```
1   #include "csapp.h"
2
3   int main()
4   {
5       int x = 1;
6
7       if (Fork() == 0)
8           printf("printf1: x=%d\n", ++x);
9       printf("printf2: x=%d\n", --x);
10      exit(0);
11  }
```

code/ecf/forkprob0.c

A. What is the output of the child process?

B. What is the output of the parent process?

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.

```
#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:

```
/* 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.

Practice Problem 8.3

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

code/ecf/waitprob0.c

```
1   int main()
2   {
3       if (Fork() == 0) {
4           printf("a");
5       }
6       else {
7           printf("b");
8           waitpid(-1, NULL, 0);
9       }
10      printf("c");
11      exit(0);
12  }
```

code/ecf/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 *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`.

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 8.17 **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

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 8.18 **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:

code/ecf/waitprob1.c

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

code/ecf/waitprob1.c

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.

```
#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

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

```
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.

```
#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

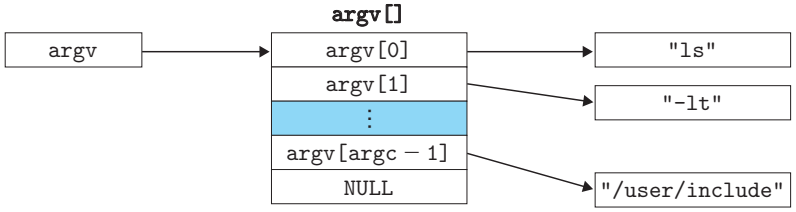
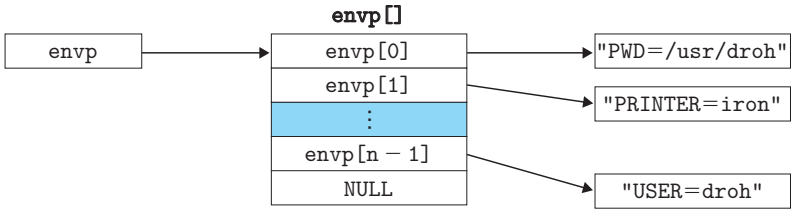


Figure 8.19
Organization of an argument list.

Figure 8.20
Organization of an environment variable list.



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

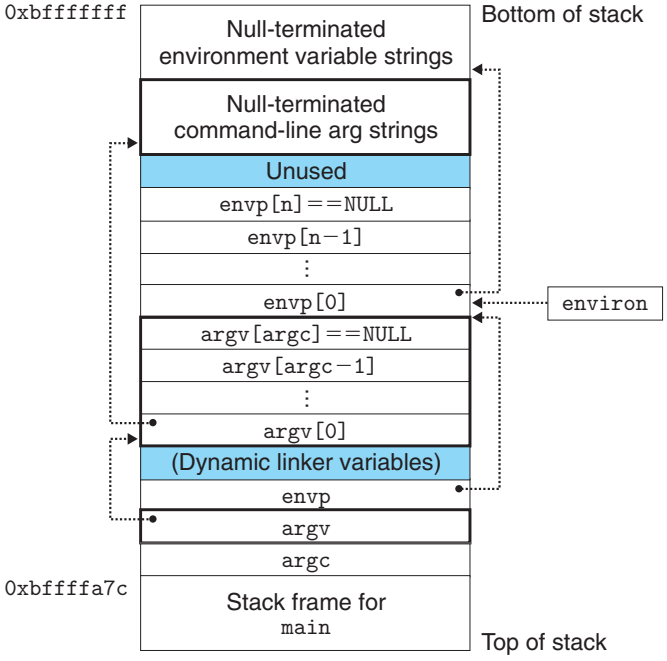
```
int main(int argc, char **argv, char **envp);
```

or equivalently,

```
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

Figure 8.21
Typical organization of the user stack when a new program starts.



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:

```
#include <stdlib.h>

char *getenv(const char *name);
```

Returns: ptr to `name` if exists, NULL if no match

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.

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

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 8.22 **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.

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 8.23 `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 8.24 `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.

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 (1)	Trace trap
6	SIGABRT	Terminate and dump core (1)	Abort signal from `abort` function
7	SIGBUS	Terminate	Bus error
8	SIGFPE	Terminate and dump core (1)	Floating point exception
9	SIGKILL	Terminate (2)	Kill program
10	SIGUSR1	Terminate	User-defined signal 1
11	SIGSEGV	Terminate and dump core (1)	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 (2)	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 8.25 **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.

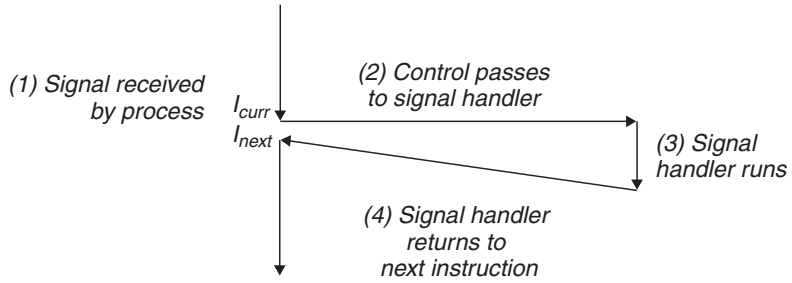


Figure 8.26
Signal handling. Receipt of a signal triggers a control transfer to a signal handler. After it finishes processing, the handler returns control to the interrupted program.

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.

```
#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.

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
```

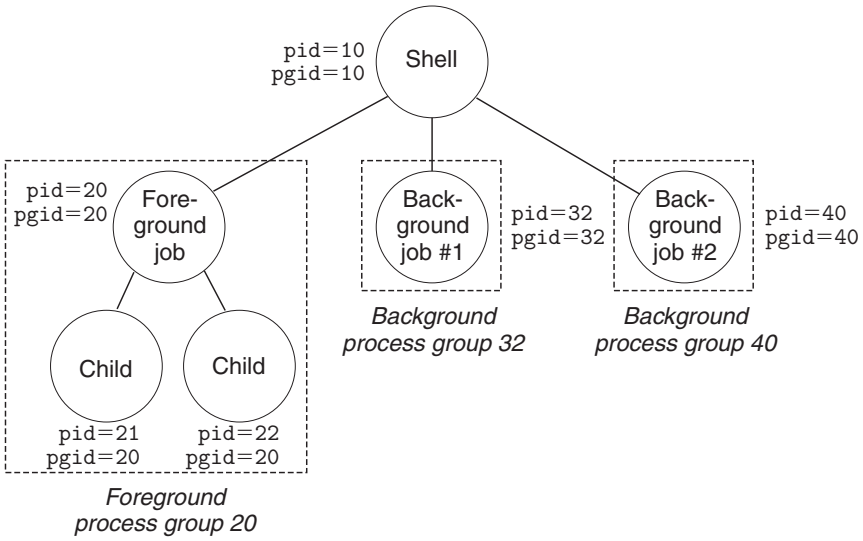


Figure 8.27
Foreground and background process groups.

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.

```
#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.

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 8.28 **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 secs 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 zero, 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.

Figure 8.29 shows a program called `alarm` that arranges to be interrupted by a SIGALRM signal every second for five seconds. When the sixth SIGALRM is delivered it terminates. When we run the program in Figure 8.29, we get the following output: a "BEEP" every second for five seconds, followed by a "BOOM" when the program terminates.

```
unix> ./alarm
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
```

Notice that the program in Figure 8.29 uses the `signal` function to install a *signal handler* function (`handler`) that is called asynchronously, interrupting the infinite `while` loop in `main`, whenever the process receives a SIGALRM signal. When the `handler` function returns, control passes back to `main`, which picks up where it was interrupted by the arrival of the signal. Installing and using signal handlers can be quite subtle, and is the topic of the next few sections.

8.5.3 Receiving Signals

When the kernel is returning from an exception handler and is ready to pass control to process p, it checks the set of unblocked pending signals (`pending & ~blocked`) for process 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.

code/ecf/alarm.c

```
1   #include "csapp.h"
2
3   void handler(int sig)
4   {
5       static int beeps = 0;
6
7       printf("BEEP\n");
8       if (++beeps < 5)
9           Alarm(1); /* Next SIGALRM will be delivered in 1 second */
10      else {
11          printf("BOOM!\n");
12          exit(0);
13      }
14  }
15
16  int main()
17  {
18      Signal(SIGALRM, handler); /* Install SIGALRM handler */
19      Alarm(1); /* Next SIGALRM will be delivered in 1s */
20
21      while (1) {
22          ;  /* Signal handler returns control here each time */
23      }
24      exit(0);
25  }
```

code/ecf/alarm.c

Figure 8.29 **Using the `alarm` function to schedule periodic events.**

- The process stops until restarted by a SIGCONT signal.
- The process ignores the signal.

Figure 8.25 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: ptr 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 sent by the shell whenever the user types `ctrl-c` at the keyboard. The default action for SIGINT 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.

The handler function is defined in lines 3–7. The main routine installs the handler in lines 12– 13, and then goes to sleep until a signal is received (line 15). When the SIGINT signal is received, the handler runs, prints a message (line 5), and then terminates the process (line 6).

Signal handlers are yet another example of concurrency in a computer system. The execution of the signal handler interrupts the execution of the main C routine, akin to the way that a low-level exception handler interrupts the control flow of the current application program. Since the logical control flow of the signal handler overlaps the logical control flow of the main routine, the signal handler and the main routine run concurrently.

Practice Problem 8.7

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:

```
unix> ./snooze 5
Slept for 3 of 5 secs.     User hits crtl-c after 3 seconds
unix>
```

code/ecf/sigint1.c

```
1  #include "csapp.h"
2
3  void 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, handler) == SIG_ERR)
13         unix_error("signal error");
14
15     pause(); /* Wait for the receipt of a signal */
16
17     exit(0);
18 }
```

code/ecf/sigint1.c

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

8.5.4 Signal Handling Issues

Signal handling is straightforward for programs that catch a single signal and then terminate. However, subtle issues arise when a program catches multiple signals.

- *Pending signals are blocked*. Unix signal handlers typically block pending signals of the type currently being processed by the handler. For example, suppose a process has caught a SIGINT signal and is currently running its SIGINT handler. If another SIGINT signal is sent to the process, then the SIGINT will become pending, but will not be received until after the handler returns.
- *Pending signals are not queued*. 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.
- *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 systems, 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.

Let's look more closely at the subtleties of signal handling, using 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 8.31 shows our first attempt. The parent installs a SIGCHLD handler, and then creates three children, each of which runs for 1 second and then terminates. 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, does some additional cleanup work (modeled by the `sleep(2)` statement), and then returns.

The `signal1` program in Figure 8.31 seems fairly straightforward. When we run it on our Linux system, however, we get the following output:

```
linux> ./signal1
Hello from child 10320
Hello from child 10321
Hello from child 10322
Handler reaped child 10320
Handler reaped child 10322
<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 10321 was never reaped and remains a zombie (indicated by the string "`defunct`" in the output of the `ps` command):

```
<ctrl-z>
Suspended
linux> ps
PID TTY STAT TIME COMMAND
...
10319  p5 T    0:03 signal1
10321  p5 Z    0:00 signal1 <defunct>
10323  p5 R    0:00 ps
```

What went wrong? The problem is that our code failed to account for the facts that signals can block and 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,

code/ecf/signal1.c

```
1   #include "csapp.h"
2
3   void handler1(int sig)
4   {
5       pid_t pid;
6
7       if ((pid = waitpid(-1, NULL, 0)) < 0)
8           unix_error("waitpid error");
9       printf("Handler reaped child %d\n", (int)pid);
10      Sleep(2);
11      return;
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              Sleep(1);
27              exit(0);
28          }
29      }
30
31      /* Parent waits for terminal input and then processes it */
32      if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
33          unix_error("read");
34
35      printf("Parent processing input\n");
36      while (1)
37          ;
38
39      exit(0);
40  }
```

code/ecf/signal1.c

Figure 8.31 `signal1`: This program is flawed because it fails to deal with the facts that signals can block, signals are not queued, and system calls can be interrupted.

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 as many zombie children as possible each time it is invoked. Figure 8.32 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 10378
Hello from child 10379
Hello from child 10380
Handler reaped child 10379
Handler reaped child 10378
Handler reaped child 10380
<cr>
Parent processing input
```

However, we are not finished yet. If we run the `signal2` program on an older version of the Solaris operating system, it correctly reaps all of the zombie children. However, now the blocked `read` system call returns prematurely with an error, before we are able to type in our input on the keyboard:

```
solaris> ./signal2
Hello from child 18906
Hello from child 18907
Hello from child 18908
Handler reaped child 18906
Handler reaped child 18908
Handler reaped child 18907
read: Interrupted system call
```

What went wrong? The problem arises because on this particular Solaris system, slow system calls such as `read` are not restarted automatically after they are interrupted by the delivery of a signal. Instead, they return prematurely to the calling application with an error condition, unlike Linux systems, which restart interrupted system calls automatically.

In order to write portable signal handling code, we must allow for the possibility that system calls will return prematurely and then restart them manually

code/ecf/signal2.c

```
1  #include "csapp.h"
2
3  void handler2(int sig)
4  {
5      pid_t pid;
6
7      while ((pid = waitpid(-1, NULL, 0)) > 0)
8          printf("Handler reaped child %d\n", (int)pid);
9      if (errno != ECHILD)
10         unix_error("waitpid error");
11     Sleep(2);
12     return;
13 }
14
15 int main()
16 {
17     int i, n;
18     char buf[MAXBUF];
19
20     if (signal(SIGCHLD, handler2) == SIG_ERR)
21         unix_error("signal error");
22
23     /* Parent creates children */
24     for (i = 0; i < 3; i++) {
25         if (Fork() == 0) {
26             printf("Hello from child %d\n", (int)getpid());
27             Sleep(1);
28             exit(0);
29         }
30     }
31
32     /* Parent waits for terminal input and then processes it */
33     if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
34         unix_error("read error");
35
36     printf("Parent processing input\n");
37     while (1)
38         ;
39
40     exit(0);
41 }
```

code/ecf/signal2.c

Figure 8.32 `signal2`: An improved version of Figure 8.31 that correctly accounts for the facts that signals can block and are not queued. However, it does not allow for the possibility that system calls can be interrupted.

when this occurs. Figure 8.33 shows the modification to `signal2` that manually restarts aborted `read` calls. The EINTR return code in `errno` indicates that the `read` system call returned prematurely after it was interrupted.

When we run our new `signal3` program on a Solaris system, the program runs correctly:

```
solaris> ./signal3
Hello from child 19571
Hello from child 19572
Hello from child 19573
Handler reaped child 19571
Handler reaped child 19572
Handler reaped child 19573
<cr>
Parent processing input
```

Practice Problem 8.8

What is the output of the following program?

code/ecf/signalprob0.c

```
1   pid_t pid;
2   int counter = 2;
3
4   void handler1(int sig) {
5       counter = counter - 1;
6       printf("%d", counter);
7       fflush(stdout);
8       exit(0);
9   }
10
11  int main() {
12      signal(SIGUSR1, handler1);
13
14      printf("%d", counter);
15      fflush(stdout);
16
17      if ((pid = fork()) == 0) {
18          while(1) {};
19      }
20      kill(pid, SIGUSR1);
21      waitpid(-1, NULL, 0);
22      counter = counter + 1;
23      printf("%d", counter);
24      exit(0);
25  }
```

code/ecf/signalprob0.c

code/ecf/signal3.c

```
1   #include "csapp.h"
2
3   void handler2(int sig)
4   {
5       pid_t pid;
6
7       while ((pid = waitpid(-1, NULL, 0)) > 0)
8           printf("Handler reaped child %d\n", (int)pid);
9       if (errno != ECHILD)
10          unix_error("waitpid error");
11      Sleep(2);
12      return;
13  }
14
15  int main() {
16      int i, n;
17      char buf[MAXBUF];
18      pid_t pid;
19
20      if (signal(SIGCHLD, handler2) == SIG_ERR)
21          unix_error("signal error");
22
23      /* Parent creates children */
24      for (i = 0; i < 3; i++) {
25          pid = Fork();
26          if (pid == 0) {
27              printf("Hello from child %d\n", (int)getpid());
28              Sleep(1);
29              exit(0);
30          }
31      }
32
33      /* Manually restart the read call if it is interrupted */
34      while ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
35          if (errno != EINTR)
36              unix_error("read error");
37
38      printf("Parent processing input\n");
39      while (1)
40          ;
41
42      exit(0);
43  }
```

code/ecf/signal3.c

Figure 8.33 `signal3`: An improved version of Figure 8.32 that correctly accounts for the fact that system calls can be interrupted.

8.5.5 Portable Signal Handling

The differences in signal handling semantics from system to system—such as whether or not an interrupted slow system call is restarted or aborted prematurely—is an ugly aspect of Unix signal handling. To deal with this problem, the Posix standard defines the `sigaction` function, which allows users on Posix-compliant systems such as Linux and Solaris to clearly specify the signal handling semantics they want.

```
#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 structure. A cleaner approach, originally proposed by W. Richard Stevens [109], is to define a wrapper function, called `Signal`, that calls `sigaction` for us. Figure 8.34 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.

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

- Once the signal handler is installed, it remains installed until `Signal` is called with a `handler` argument of either SIG_IGN or SIG_DFL. (Some older Unix systems restore the signal action to its default action after a signal has been processed by a handler.)

Figure 8.35 shows a version of the `signal2` program from Figure 8.32 that uses our `Signal` wrapper to get predictable signal handling semantics on different computer systems. The only difference is that we have installed the handler with a call to `Signal` rather than a call to `signal`. The program now runs correctly on both our Solaris and Linux systems, and we no longer need to manually restart interrupted `read` system calls.

8.5.6 Explicitly Blocking and Unblocking Signals

Applications can explicitly block and unblock selected signals using the `sigprocmask` function:

```
#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.

code/ecf/signal4.c

```
1  #include "csapp.h"
2
3  void handler2(int sig)
4  {
5      pid_t pid;
6
7      while ((pid = waitpid(-1, NULL, 0)) > 0)
8          printf("Handler reaped child %d\n", (int)pid);
9      if (errno != ECHILD)
10         unix_error("waitpid error");
11     Sleep(2);
12     return;
13 }
14
15 int main()
16 {
17     int i, n;
18     char buf[MAXBUF];
19     pid_t pid;
20
21     Signal(SIGCHLD, handler2); /* sigaction error-handling wrapper */
22
23     /* Parent creates children */
24     for (i = 0; i < 3; i++) {
25         pid = Fork();
26         if (pid == 0) {
27             printf("Hello from child %d\n", (int)getpid());
28             Sleep(1);
29             exit(0);
30         }
31     }
32
33     /* Parent waits for terminal input and then processes it */
34     if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
35         unix_error("read error");
36
37     printf("Parent processing input\n");
38     while (1)
39         ;
40     exit(0);
41 }
```

code/ecf/signal4.c

Figure 8.35 `signal4`: A version of Figure 8.32 that uses our `Signal` wrapper to get portable signal handling semantics.

8.5.7 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 8.36, which captures the structure of a typical Unix shell. The parent keeps track of its current children using entries in a job list, with one entry per job. The `addjob` and `deletejob` functions add and remove entries from the job list, respectively.

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 `deletejob` in the handler. If `addjob` wins the race, then the answer is correct. If

code/ecf/procmask1.c

```
1  void handler(int sig)
2  {
3      pid_t pid;
4      while ((pid = waitpid(-1, NULL, 0)) > 0) /* Reap a zombie child */
5          deletejob(pid); /* Delete the child from the job list */
6      if (errno != ECHILD)
7          unix_error("waitpid error");
8  }
9
10 int main(int argc, char **argv)
11 {
12     int pid;
13
14     Signal(SIGCHLD, handler);
15     initjobs(); /* Initialize the job list */
16
17     while (1) {
18         /* Child process */
19         if ((pid = Fork()) == 0) {
20             Execve("/bin/date", argv, NULL);
21         }
22
23         /* Parent process */
24         addjob(pid);  /* Add the child to the job list */
25     }
26     exit(0);
27 }
```

code/ecf/procmask1.c

Figure 8.36 **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.

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

Figure 8.37 shows one way to eliminate the race in Figure 8.36. 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`.

code/ecf/procmask2.c

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

code/ecf/procmask2.c

Figure 8.37 **Using `sigprocmask` to synchronize processes.** In this example, the parent ensures that `addjob` executes before the corresponding `deletejob`.

code/ecf/rfork.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4   #include <sys/time.h>
5   #include <sys/types.h>
6
7   /* Sleep for a random period between [0, MAX_SLEEP] us. */
8   #define MAX_SLEEP 100000
9
10  /* Macro that maps val into the range [0, RAND_MAX] */
11  #define CONVERT(val) (((double)val)/(double)RAND_MAX)
12
13  pid_t Fork(void)
14  {
15      static struct timeval time;
16      unsigned bool, secs;
17      pid_t pid;
18
19      /* Generate a different seed each time the function is called */
20      gettimeofday(&time, NULL);
21      srand(time.tv_usec);
22
23      /* Determine whether to sleep in parent of child and for how long */
24      bool = (unsigned)(CONVERT(rand()) + 0.5);
25      secs = (unsigned)(CONVERT(rand()) * MAX_SLEEP);
26
27      /* Call the real fork function */
28      if ((pid = fork()) < 0)
29          return pid;
30
31      /* Randomly decide to sleep in the parent or the child */
32      if (pid == 0) { /* Child */
33          if(bool) {
34              usleep(secs);
35          }
36      }
37      else { /* Parent */
38          if (!bool) {
39              usleep(secs);
40          }
41      }
42
43      /* Return the PID like a normal fork call */
44      return pid;
45  }
```

code/ecf/rfork.c

Figure 8.38 **A wrapper for `fork` that randomly determines the order in which the parent and child execute.** The parent and child flip a coin to determine which will sleep, thus giving the other process a chance to be scheduled.

Aside A handy trick for exposing races in your code

Races such as those in Figure 8.36 are difficult to detect because they depend on kernel-specific scheduling decisions. After a call to `fork`, some kernels schedule the child to run first, while other kernels schedule the parent to run first. If you were to run the code in Figure 8.36 on one of the latter systems, it would never fail, no matter how many times you tested it. But as soon as you ran it on one of the former systems, then the race would be exposed and the code would fail. Figure 8.38 shows a wrapper function that can help expose such hidden assumptions about the execution ordering of parent and child processes. The basic idea is that after each call to `fork`, the parent and child flip a coin to determine which of them will sleep for a bit, thus giving the other process the opportunity to run first. If we were to run the code multiple times, then with high probability we would exercise both orderings of child and parent executions, regardless of the particular kernel's scheduling policy.

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.

```
#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 a 0. The calling environment includes the program counter, stack pointer, and general purpose registers.

```
#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 8.39 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.

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 8.40 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 shell 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 ran the program on our system, we got the following output:

```
unix> ./restart
starting
processing...
processing...
restarting          User hits ctrl-c
processing...
restarting          User hits ctrl-c
processing...
```

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.

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      int rc;
13
14      rc = setjmp(buf);
15      if (rc == 0)
16          foo();
17      else if (rc == 1)
18          printf("Detected an error1 condition in foo\n");
19      else if (rc == 2)
20          printf("Detected an error2 condition in foo\n");
21      else
22          printf("Unknown error condition in foo\n");
23      exit(0);
24  }
25
26  /* Deeply nested function foo */
27  void foo(void)
28  {
29      if (error1)
30          longjmp(buf, 1);
31      bar();
32  }
33
34  void bar(void)
35  {
36      if (error2)
37          longjmp(buf, 2);
38  }
```

code/ecf/setjmp.c

Figure 8.39 **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      Signal(SIGINT, handler);
13
14      if (!sigsetjmp(buf, 1))
15          printf("starting\n");
16      else
17          printf("restarting\n");
18
19      while(1) {
20          Sleep(1);
21          printf("processing...\n");
22      }
23      exit(0);
24  }
```

code/ecf/restart.c

Figure 8.40 **A program that uses nonlocal jumps to restart itself when the user types** `ctrl-c`.

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. 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

The Intel macroarchitecture specification contains a detailed discussion of exceptions and interrupts on Intel processors [27]. Operating systems texts [98, 104, 112] contain additional information on exceptions, processes, and signals. The classic work by W. Richard Stevens [110] 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. Blum [9] is an excellent reference for x86 assembly language, and describes in detail the x86 syscall interface.