```
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:

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

We can simplify our code even further by using *error-handling wrappers*, as pioneered by Stevens in [110]. For a given base function `foo`, we define a wrapper function `Foo` with identical arguments but with the first letter of the name capitalized. The wrapper calls the base function, checks for errors, and terminates if there are any problems. For example, here is the error-handling wrapper for the `fork` function:

```
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:

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

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

See Appendix A for a discussion of Unix error handling and the error-handling wrappers used throughout this book. The wrappers are defined in a file called `csapp.c`, and their prototypes are defined in a header file called `csapp.h`. These are available online from the CS:APP Web site.

## 8.4 Process Control

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

### 8.4.1 Obtaining Process IDs

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

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

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

Returns: PID of either the caller or the parent

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

### 8.4.2 Creating and Terminating Processes

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

*Running.* The process is either executing on the CPU or waiting to be executed and will eventually be scheduled by the kernel.

*Stopped.* The execution of the process is *suspended* and will not be scheduled. A process stops as a result of receiving a SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal, and it remains stopped until it receives a SIGCONT signal, at which point it becomes running again. (A *signal* is a form of software interrupt that we will describe in detail in Section 8.5.)

*Terminated.* The process is stopped permanently. A process becomes terminated for one of three reasons: (1) receiving a signal whose default action is to terminate the process, (2) returning from the main routine, or (3) calling the `exit` function.

```
#include <stdlib.h>

void exit(int status);
```

This function does not return

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

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

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

pid_t fork(void);
```

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

The newly created child process is almost, but not quite, identical to the parent. The child gets an identical (but separate) copy of the parent's user-level virtual address space, including the code and data segments, heap, shared libraries, and user stack. The child also gets identical copies of any of the parent's open file descriptors, which means the child can read and write any files that were open in the parent when it called `fork`. The most significant difference between the parent and the newly created child is that they have different PIDs.

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

Figure 8.15 shows a simple example of a parent process that uses `fork` to create a child process. When the `fork` call returns in line 6, `x` has a value of 1 in both the parent and child. The child increments and prints its copy of `x` in line 8. Similarly, the parent decrements and prints its copy of `x` in line 13.

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

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

There are some subtle aspects to this simple example.

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

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

*code/ecf/fork.c*

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

*code/ecf/fork.c*

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

*Duplicate but separate address spaces.* If we could halt both the parent and the child immediately after the `fork` function returned in each process, we would see that the address space of each process is identical. Each process has the same user stack, the same local variable values, the same heap, the same global variable values, and the same code. Thus, in our example program, local variable `x` has a value of 1 in both the parent and the child when the `fork` function returns in line 6. However, since the parent and the child are separate processes, they each have their own private address spaces. Any subsequent changes that a parent or child makes to `x` are private and are not reflected in the memory of the other process. This is why the variable `x` has different values in the parent and child when they call their respective `printf` statements.

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

When you are first learning about the `fork` function, it is often helpful to sketch the *process graph*, which is a simple kind of precedence graph that captures the partial ordering of program statements. Each vertex $a$ corresponds to the execution of a program statement. A directed edge $a \rightarrow b$ denotes that statement $a$ "happens before" statement $b$. Edges can be labeled with information such as the current value of a variable. Vertices corresponding to `printf` statements can be labeled with the output of the `printf`. Each graph begins with a vertex that

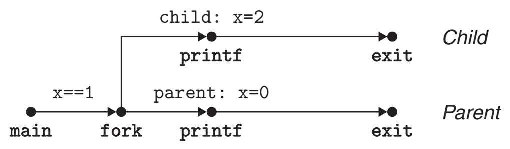

**Figure 8.16** Process graph for the example program in Figure 8.15.

```
int main()
{
    Fork();
    Fork();
    printf("hello\n");
    exit(0);
}
```

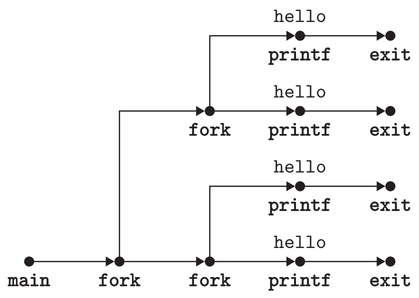

**Figure 8.17** **Process graph for a nested** `fork`.

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

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

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

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

**Practice Problem 8.2** (solution page 831)

Consider the following program:

*code/ecf/global-forkprob0.c*

```
int main()
{
    int a = 9;

    if (Fork() == 0)
        printf("p1: a=%d\n", a--);
    printf("p2: a=%d\n", a++);
    exit(0);
}
```

*code/ecf/global-forkprob0.c*

A. What is the output of the child process?

B. What is the output of the parent process?

### 8.4.3 Reaping Child Processes

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

When a parent process terminates, the kernel arranges for the `init` process to become the adopted parent of any orphaned children. The `init` process, which has a PID of 1, is created by the kernel during system start-up, never terminates, and is the ancestor of every process. If a parent process terminates without reaping its zombie children, then the kernel arranges for the `init` process to reap them. However, long-running programs such as shells or servers should always reap their zombie children. Even though zombies are not running, they still consume system memory resources.

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

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

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

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

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

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

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

### Determining the Members of the Wait Set

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

- If `pid > 0`, then the wait set is the singleton child process whose process ID is equal to `pid`.
- If `pid = -1`, then the wait set consists of all of the parent's child processes.

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

### Modifying the Default Behavior

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

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

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

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

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

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

### Checking the Exit Status of a Reaped Child

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

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

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

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

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

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

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

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

### Error Conditions

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

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

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

*code/ecf/global-waitprob0.c*

```c
int main()
{
    if (Fork() == 0) {
        printf("9"); fflush(stdout);
    }
    else {
            printf("0"); fflush(stdout);
            waitpid(-1, NULL, 0);
        }
        printf("3"); fflush(stdout);
        printf("6"); exit(0);
    }
```

*code/ecf/global-waitprob0.c*

### The `wait` Function

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

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

pid_t wait(int *statusp);
```

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

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

### Examples of Using `waitpid`

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

**Aside** Constants associated with Unix functions

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

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

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

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

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

*code/ecf/waitpid1.c*

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

*code/ecf/waitpid1.c*

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

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

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

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

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

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

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

## Practice Problem 8.4 (solution page 833)

Consider the following program:

*code/ecf/global-waitprob1.c*

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

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

*code/ecf/global-waitprob1.c*

A. How many output lines does this program generate?

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

*code/ecf/waitpid2.c*

```
#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);
}
```

*code/ecf/waitpid2.c*

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

### 8.4.4 Putting Processes to Sleep

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

```
#include <unistd.h>

unsigned int sleep(unsigned int secs);
```

Returns: seconds left to sleep

`Sleep` returns zero if the requested amount of time has elapsed, and the number of seconds still left to sleep otherwise. The latter case is possible if the `sleep` function returns prematurely because it was interrupted by a *signal*. We will discuss signals in detail in Section 8.5.

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

```
#include <unistd.h>

int pause(void);
```

Always returns −1

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

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

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

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

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

### 8.4.5 Loading and Running Programs

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

```
#include <unistd.h>

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

Does not return if OK; returns −1 on error

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

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

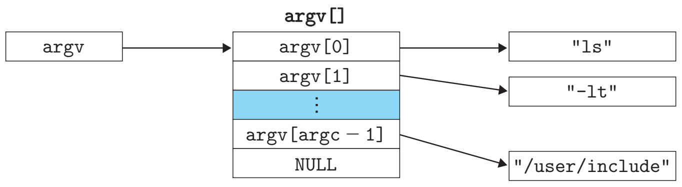

**Figure 8.20** Organization of an argument list.

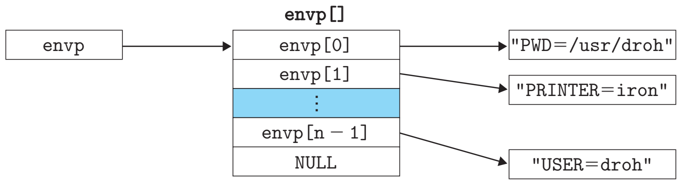

**Figure 8.21** Organization of an environment variable list.

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

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

or equivalently,

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

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

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

Linux provides several functions for manipulating the environment array:

```
#include <stdlib.h>

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

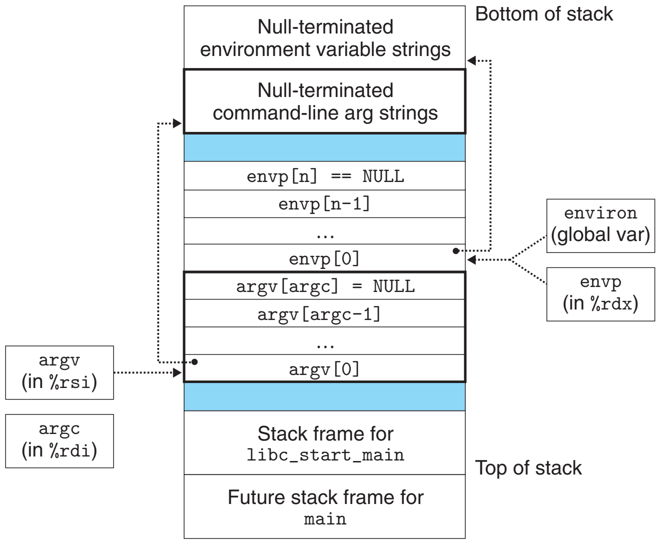

**Figure 8.22**
**Typical organization of the user stack when a new program starts.**

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

```
#include <stdlib.h>

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

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

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

## Practice Problem 8.6 (solution page 833)

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

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

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

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

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

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

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

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

**Aside** Programs versus processes

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

*code/ecf/shellex.c*

```c
#include "csapp.h"
#define MAXARGS   128

/* Function prototypes */
void eval(char *cmdline);
int parseline(char *buf, char **argv);
int builtin_command(char **argv);

int main()
{
    char cmdline[MAXLINE]; /* Command line */

    while (1) {
        /* Read */
        printf("> ");
        Fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);

        /* Evaluate */
        eval(cmdline);
    }
}
```

*code/ecf/shellex.c*

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

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

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

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

*code/ecf/shellex.c*

```
/* 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);
            }
        }

        /* Parent waits for foreground job to terminate */
        if (!bg) {
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        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 & */
        return 1;
    return 0;                     /* Not a builtin command */
}
```

*code/ecf/shellex.c*

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

*code/ecf/shellex.c*

```
/* 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-1] == '&')) != 0)
        argv[--argc] = NULL;

    return bg;
}
```

*code/ecf/shellex.c*

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

## 8.5 Signals

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