8

Process Control

8.1 Introduction

We now turn to the process control provided by the UNIX System. This includes the creation of new processes, program execution, and process termination. We also look at the various IDs that are the property of the process—real, effective, and saved; user and group IDs—and how they're affected by the process control primitives. Interpreter files and the `system` function are also covered. We conclude the chapter by looking at the process accounting provided by most UNIX systems. This lets us look at the process control functions from a different perspective.

8.2 Process Identifiers

Every process has a unique process ID, a non-negative integer. Because the process ID is the only well-known identifier of a process that is always unique, it is often used as a piece of other identifiers, to guarantee uniqueness. For example, applications sometimes include the process ID as part of a filename in an attempt to generate unique filenames.

Although unique, process IDs are reused. As processes terminate, their IDs become candidates for reuse. Most UNIX systems implement algorithms to delay reuse, however, so that newly created processes are assigned IDs different from those used by processes that terminated recently. This prevents a new process from being mistaken for the previous process to have used the same ID.

There are some special processes, but the details differ from implementation to implementation. Process ID 0 is usually the scheduler process and is often known as the *swapper*. No program on disk corresponds to this process, which is part of the

kernel and is known as a system process. Process ID 1 is usually the `init` process and is invoked by the kernel at the end of the bootstrap procedure. The program file for this process was `/etc/init` in older versions of the UNIX System and is `/sbin/init` in newer versions. This process is responsible for bringing up a UNIX system after the kernel has been bootstrapped. `init` usually reads the system-dependent initialization files—the `/etc/rc*` files or `/etc/inittab` and the files in `/etc/init.d`—and brings the system to a certain state, such as multiuser. The `init` process never dies. It is a normal user process, not a system process within the kernel, like the swapper, although it does run with superuser privileges. Later in this chapter, we'll see how `init` becomes the parent process of any orphaned child process.

> In Mac OS X 10.4, the `init` process was replaced with the `launchd` process, which performs the same set of tasks as `init`, but has expanded functionality. See Section 5.10 in Singh [2006] for a discussion of how `launchd` operates.

Each UNIX System implementation has its own set of kernel processes that provide operating system services. For example, on some virtual memory implementations of the UNIX System, process ID 2 is the *pagedaemon*. This process is responsible for supporting the paging of the virtual memory system.

In addition to the process ID, there are other identifiers for every process. The following functions return these identifiers.

```
#include <unistd.h>

pid_t getpid(void);
                                        Returns: process ID of calling process

pid_t getppid(void);
                                    Returns: parent process ID of calling process

uid_t getuid(void);
                                        Returns: real user ID of calling process

uid_t geteuid(void);
                                    Returns: effective user ID of calling process

gid_t getgid(void);
                                        Returns: real group ID of calling process

gid_t getegid(void);
                                   Returns: effective group ID of calling process
```

Note that none of these functions has an error return. We'll return to the parent process ID in the next section when we discuss the `fork` function. The real and effective user and group IDs were discussed in Section 4.4.

8.3 fork Function

An existing process can create a new one by calling the fork function.

```
#include <unistd.h>

pid_t fork(void);
```

Returns: 0 in child, process ID of child in parent, –1 on error

The new process created by fork is called the *child process*. This function is called once but returns twice. The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child. The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children. The reason fork returns 0 to the child is that a process can have only a single parent, and the child can always call getppid to obtain the process ID of its parent. (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)

Both the child and the parent continue executing with the instruction that follows the call to fork. The child is a copy of the parent. For example, the child gets a copy of the parent's data space, heap, and stack. Note that this is a copy for the child; the parent and the child do not share these portions of memory. The parent and the child do share the text segment, however (Section 7.6).

Modern implementations don't perform a complete copy of the parent's data, stack, and heap, since a fork is often followed by an exec. Instead, a technique called *copy-on-write* (COW) is used. These regions are shared by the parent and the child and have their protection changed by the kernel to read-only. If either process tries to modify these regions, the kernel then makes a copy of that piece of memory only, typically a "page" in a virtual memory system. Section 9.2 of Bach [1986] and Sections 5.6 and 5.7 of McKusick et al. [1996] provide more detail on this feature.

> Variations of the fork function are provided by some platforms. All four platforms discussed in this book support the vfork(2) variant discussed in the next section.
>
> Linux 3.2.0 also provides new process creation through the clone(2) system call. This is a generalized form of fork that allows the caller to control what is shared between parent and child.
>
> FreeBSD 8.0 provides the rfork(2) system call, which is similar to the Linux clone system call. The rfork call is derived from the Plan 9 operating system (Pike et al. [1995]).
>
> Solaris 10 provides two threads libraries: one for POSIX threads (pthreads) and one for Solaris threads. In previous releases, the behavior of fork differed between the two thread libraries. For POSIX threads, fork created a process containing only the calling thread, but for Solaris threads, fork created a process containing copies of all threads from the process of the calling thread. In Solaris 10, this behavior has changed; fork creates a child containing a copy of the calling thread only, regardless of which thread library is used. Solaris also provides the fork1 function, which can be used to create a process that duplicates only the calling thread, and the forkall function, which can be used to create a process that duplicates all the threads in the process. Threads are discussed in detail in Chapters 11 and 12.

Example

The program in Figure 8.1 demonstrates the `fork` function, showing how changes to variables in a child process do not affect the value of the variables in the parent process.

```
#include "apue.h"

int     globvar = 6;        /* external variable in initialized data */
char    buf[] = "a write to stdout\n";

int
main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");
    printf("before fork\n");    /* we don't flush stdout */

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {      /* child */
        globvar++;              /* modify variables */
        var++;
    } else {
        sleep(2);               /* parent */
    }

    printf("pid = %ld, glob = %d, var = %d\n", (long)getpid(), globvar,
      var);
    exit(0);
}
```

Figure 8.1 Example of `fork` function

If we execute this program, we get

```
$ ./a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89      child's variables were changed
pid = 429, glob = 6, var = 88      parent's copy was not changed
$ ./a.out > temp.out
$ cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88
```

In general, we never know whether the child starts executing before the parent, or vice versa. The order depends on the scheduling algorithm used by the kernel. If it's required that the child and parent synchronize their actions, some form of interprocess

communication is required. In the program shown in Figure 8.1, we simply have the parent put itself to sleep for 2 seconds, to let the child execute. There is no guarantee that the length of this delay is adequate, and we talk about this and other types of synchronization in Section 8.9 when we discuss race conditions. In Section 10.16, we show how to use signals to synchronize a parent and a child after a `fork`.

When we write to standard output, we subtract 1 from the size of `buf` to avoid writing the terminating null byte. Although `strlen` will calculate the length of a string not including the terminating null byte, `sizeof` calculates the size of the buffer, which does include the terminating null byte. Another difference is that using `strlen` requires a function call, whereas `sizeof` calculates the buffer length at compile time, as the buffer is initialized with a known string and its size is fixed.

Note the interaction of `fork` with the I/O functions in the program in Figure 8.1. Recall from Chapter 3 that the `write` function is not buffered. Because `write` is called before the `fork`, its data is written once to standard output. The standard I/O library, however, is buffered. Recall from Section 5.12 that standard output is line buffered if it's connected to a terminal device; otherwise, it's fully buffered. When we run the program interactively, we get only a single copy of the first `printf` line, because the standard output buffer is flushed by the newline. When we redirect standard output to a file, however, we get two copies of the `printf` line. In this second case, the `printf` before the `fork` is called once, but the line remains in the buffer when `fork` is called. This buffer is then copied into the child when the parent's data space is copied to the child. Both the parent and the child now have a standard I/O buffer with this line in it. The second `printf`, right before the `exit`, just appends its data to the existing buffer. When each process terminates, its copy of the buffer is finally flushed. □

File Sharing

When we redirect the standard output of the parent from the program in Figure 8.1, the child's standard output is also redirected. Indeed, one characteristic of `fork` is that all file descriptors that are open in the parent are duplicated in the child. We say "duplicated" because it's as if the `dup` function had been called for each descriptor. The parent and the child share a file table entry for every open descriptor (recall Figure 3.9).

Consider a process that has three different files opened for standard input, standard output, and standard error. On return from `fork`, we have the arrangement shown in Figure 8.2.

It is important that the parent and the child share the same file offset. Consider a process that `fork`s a child, then `wait`s for the child to complete. Assume that both processes write to standard output as part of their normal processing. If the parent has its standard output redirected (by a shell, perhaps), it is essential that the parent's file offset be updated by the child when the child writes to standard output. In this case, the child can write to standard output while the parent is `wait`ing for it; on completion of the child, the parent can continue writing to standard output, knowing that its output will be appended to whatever the child wrote. If the parent and the child did not share the same file offset, this type of interaction would be more difficult to accomplish and would require explicit actions by the parent.

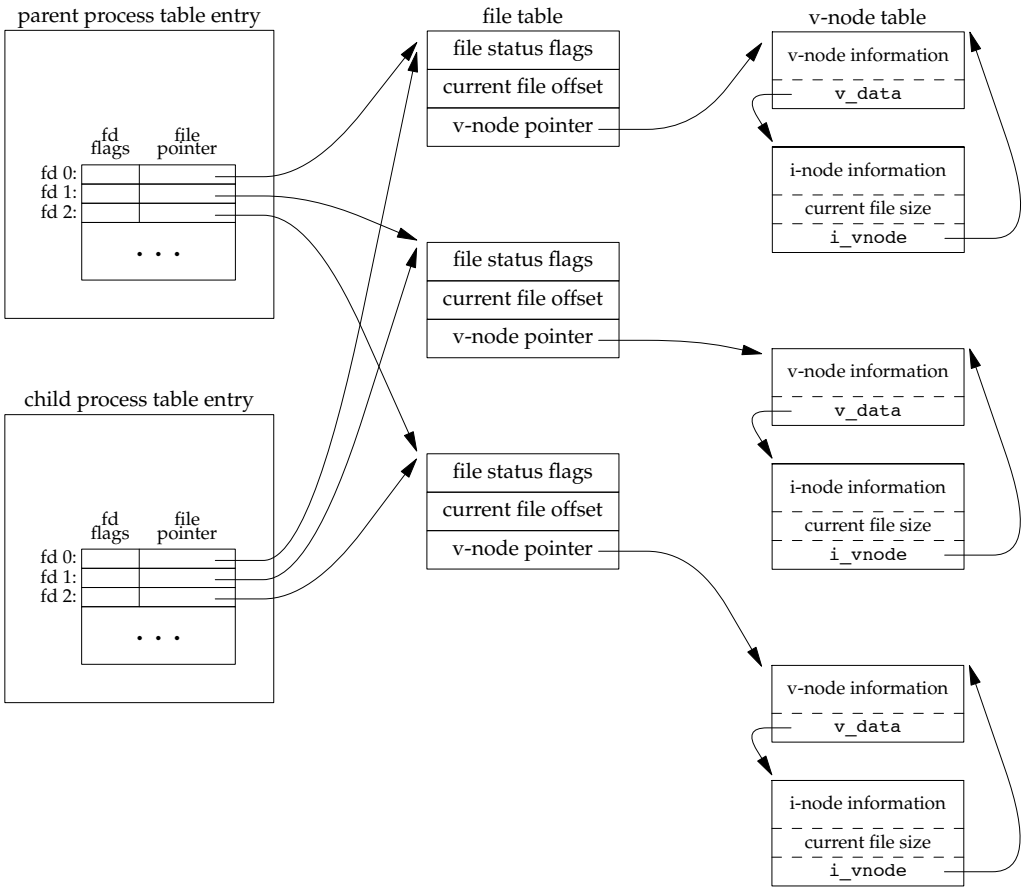

Figure 8.2 Sharing of open files between parent and child after `fork`

If both parent and child write to the same descriptor, without any form of synchronization, such as having the parent `wait` for the child, their output will be intermixed (assuming it's a descriptor that was open before the `fork`). Although this is possible—we saw it in Figure 8.2—it's not the normal mode of operation.

There are two normal cases for handling the descriptors after a `fork`.

1. The parent waits for the child to complete. In this case, the parent does not need to do anything with its descriptors. When the child terminates, any of the shared descriptors that the child read from or wrote to will have their file offsets updated accordingly.
2. Both the parent and the child go their own ways. Here, after the `fork`, the parent closes the descriptors that it doesn't need, and the child does the same thing. This way, neither interferes with the other's open descriptors. This scenario is often found with network servers.

Besides the open files, numerous other properties of the parent are inherited by the child:

- Real user ID, real group ID, effective user ID, and effective group ID
- Supplementary group IDs
- Process group ID
- Session ID
- Controlling terminal
- The set-user-ID and set-group-ID flags
- Current working directory
- Root directory
- File mode creation mask
- Signal mask and dispositions
- The close-on-exec flag for any open file descriptors
- Environment
- Attached shared memory segments
- Memory mappings
- Resource limits

The differences between the parent and child are

- The return values from `fork` are different.
- The process IDs are different.
- The two processes have different parent process IDs: the parent process ID of the child is the parent; the parent process ID of the parent doesn't change.
- The child's `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime` values are set to 0 (these times are discussed in Section 8.17).
- File locks set by the parent are not inherited by the child.
- Pending alarms are cleared for the child.
- The set of pending signals for the child is set to the empty set.

Many of these features haven't been discussed yet—we'll cover them in later chapters.

The two main reasons for `fork` to fail are (a) if too many processes are already in the system, which usually means that something else is wrong, or (b) if the total number of processes for this real user ID exceeds the system's limit. Recall from Figure 2.11 that `CHILD_MAX` specifies the maximum number of simultaneous processes per real user ID.

There are two uses for `fork`:

1. When a process wants to duplicate itself so that the parent and the child can each execute different sections of code at the same time. This is common for network servers—the parent waits for a service request from a client. When the request arrives, the parent calls `fork` and lets the child handle the request. The parent goes back to waiting for the next service request to arrive.
2. When a process wants to execute a different program. This is common for shells. In this case, the child does an `exec` (which we describe in Section 8.10) right after it returns from the `fork`.

Some operating systems combine the operations from step 2—a `fork` followed by an `exec`—into a single operation called a *spawn*. The UNIX System separates the two, as there are numerous cases where it is useful to `fork` without doing an `exec`. Also, separating the two operations allows the child to change the per-process attributes between the `fork` and the `exec`, such as I/O redirection, user ID, signal disposition, and so on. We'll see numerous examples of this in Chapter 15.

> The Single UNIX Specification does include `spawn` interfaces in the advanced real-time option group. These interfaces are not intended to be replacements for `fork` and `exec`, however. They are intended to support systems that have difficulty implementing `fork` efficiently, especially systems without hardware support for memory management.

8.4 `vfork` Function

The function `vfork` has the same calling sequence and same return values as `fork`, but the semantics of the two functions differ.

> The `vfork` function originated with 2.9BSD. Some consider the function a blemish, but all the platforms covered in this book support it. In fact, the BSD developers removed it from the 4.4BSD release, but all the open source BSD distributions that derive from 4.4BSD added support for it back into their own releases. The `vfork` function was marked as an obsolescent interface in Version 3 of the Single UNIX Specification and was removed entirely in Version 4. We include it here for historical reasons only. Portable applications should not use it.

The `vfork` function was intended to create a new process for the purpose of executing a new program (step 2 at the end of the previous section), similar to the method used by the bare-bones shell from Figure 1.7. The `vfork` function creates the new process, just like `fork`, without copying the address space of the parent into the child, as the child won't reference that address space; the child simply calls `exec` (or `exit`) right after the `vfork`. Instead, the child runs in the address space of the parent until it calls either `exec` or `exit`. This optimization is more efficient on some implementations of the UNIX System, but leads to undefined results if the child modifies any data (except the variable used to hold the return value from `vfork`), makes function calls, or returns without calling `exec` or `exit`. (As we mentioned in the previous section, implementations use copy-on-write to improve the efficiency of a `fork` followed by an `exec`, but no copying is still faster than some copying.)

Another difference between the two functions is that `vfork` guarantees that the child runs first, until the child calls `exec` or `exit`. When the child calls either of these functions, the parent resumes. (This can lead to deadlock if the child depends on further actions of the parent before calling either of these two functions.)

Example

The program in Figure 8.3 is a modified version of the program from Figure 8.1. We've replaced the call to `fork` with `vfork` and removed the `write` to standard output. Also, we don't need to have the parent call `sleep`, as we're guaranteed that it is put to sleep by the kernel until the child calls either `exec` or `exit`.

```
#include "apue.h"

int     globvar = 6;        /* external variable in initialized data */

int
main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    printf("before vfork\n");   /* we don't flush stdio */
    if ((pid = vfork()) < 0) {
        err_sys("vfork error");
    } else if (pid == 0) {      /* child */
        globvar++;              /* modify parent's variables */
        var++;
        _exit(0);               /* child terminates */
    }

    /* parent continues here */
    printf("pid = %ld, glob = %d, var = %d\n", (long)getpid(), globvar,
      var);
    exit(0);
}
```

Figure 8.3 Example of `vfork` function

Running this program gives us

```
$ ./a.out
before vfork
pid = 29039, glob = 7, var = 89
```

Here, the incrementing of the variables done by the child changes the values in the parent. Because the child runs in the address space of the parent, this doesn't surprise us. This behavior, however, differs from the behavior of `fork`.

Note in Figure 8.3 that we call `_exit` instead of `exit`. As we described in Section 7.3, `_exit` does not perform any flushing of standard I/O buffers. If we call `exit` instead, the results are indeterminate. Depending on the implementation of the standard I/O library, we might see no difference in the output, or we might find that the output from the first `printf` in the parent has disappeared.

If the child calls `exit`, the implementation flushes the standard I/O streams. If this is the only action taken by the library, then we will see no difference from the output generated if the child called `_exit`. If the implementation also closes the standard I/O streams, however, the memory representing the `FILE` object for the standard output will be cleared out. Because the child is borrowing the parent's address space, when the parent resumes and calls `printf`, no output will appear and `printf` will return –1. Note that the parent's `STDOUT_FILENO` is still valid, as the child gets a copy of the parent's file descriptor array (refer back to Figure 8.2).

Most modern implementations of `exit` do not bother to close the streams. Because the process is about to exit, the kernel will close all the file descriptors open in the process. Closing them in the library simply adds overhead without any benefit.

□

Section 5.6 of McKusick et al. [1996] contains additional information on the implementation issues of `fork` and `vfork`. Exercises 8.1 and 8.2 continue the discussion of `vfork`.

8.5 `exit` Functions

As we described in Section 7.3, a process can terminate normally in five ways:

1. Executing a `return` from the `main` function. As we saw in Section 7.3, this is equivalent to calling `exit`.
2. Calling the `exit` function. This function is defined by ISO C and includes the calling of all exit handlers that have been registered by calling `atexit` and closing all standard I/O streams. Because ISO C does not deal with file descriptors, multiple processes (parents and children), and job control, the definition of this function is incomplete for a UNIX system.
3. Calling the `_exit` or `_Exit` function. ISO C defines `_Exit` to provide a way for a process to terminate without running exit handlers or signal handlers. Whether standard I/O streams are flushed depends on the implementation. On UNIX systems, `_Exit` and `_exit` are synonymous and do not flush standard I/O streams. The `_exit` function is called by `exit` and handles the UNIX system-specific details; `_exit` is specified by POSIX.1.

   In most UNIX system implementations, `exit`(3) is a function in the standard C library, whereas `_exit`(2) is a system call.

4. Executing a `return` from the start routine of the last thread in the process. The return value of the thread is not used as the return value of the process, however. When the last thread returns from its start routine, the process exits with a termination status of 0.
5. Calling the `pthread_exit` function from the last thread in the process. As with the previous case, the exit status of the process in this situation is always 0, regardless of the argument passed to `pthread_exit`. We'll say more about `pthread_exit` in Section 11.5.

The three forms of abnormal termination are as follows:

1. Calling `abort`. This is a special case of the next item, as it generates the `SIGABRT` signal.
2. When the process receives certain signals. (We describe signals in more detail in Chapter 10.) The signal can be generated by the process itself (e.g., by calling the `abort` function), by some other process, or by the kernel. Examples of

signals generated by the kernel include the process referencing a memory location not within its address space or trying to divide by 0.

3. The last thread responds to a cancellation request. By default, cancellation occurs in a deferred manner: one thread requests that another be canceled, and sometime later the target thread terminates. We discuss cancellation requests in detail in Sections 11.5 and 12.7.

Regardless of how a process terminates, the same code in the kernel is eventually executed. This kernel code closes all the open descriptors for the process, releases the memory that it was using, and so on.

For any of the preceding cases, we want the terminating process to be able to notify its parent how it terminated. For the three exit functions (`exit`, `_exit`, and `_Exit`), this is done by passing an exit status as the argument to the function. In the case of an abnormal termination, however, the kernel—not the process—generates a termination status to indicate the reason for the abnormal termination. In any case, the parent of the process can obtain the termination status from either the `wait` or the `waitpid` function (described in the next section).

Note that we differentiate between the exit status, which is the argument to one of the three exit functions or the return value from `main`, and the termination status. The exit status is converted into a termination status by the kernel when `_exit` is finally called (recall Figure 7.2). Figure 8.4 describes the various ways the parent can examine the termination status of a child. If the child terminated normally, the parent can obtain the exit status of the child.

When we described the `fork` function, it was obvious that the child has a parent process after the call to `fork`. Now we're talking about returning a termination status to the parent. But what happens if the parent terminates before the child? The answer is that the `init` process becomes the parent process of any process whose parent terminates. In such a case, we say that the process has been inherited by `init`. What normally happens is that whenever a process terminates, the kernel goes through all active processes to see whether the terminating process is the parent of any process that still exists. If so, the parent process ID of the surviving process is changed to be 1 (the process ID of `init`). This way, we're guaranteed that every process has a parent.

Another condition we have to worry about is when a child terminates before its parent. If the child completely disappeared, the parent wouldn't be able to fetch its termination status when and if the parent was finally ready to check if the child had terminated. The kernel keeps a small amount of information for every terminating process, so that the information is available when the parent of the terminating process calls `wait` or `waitpid`. Minimally, this information consists of the process ID, the termination status of the process, and the amount of CPU time taken by the process. The kernel can discard all the memory used by the process and close its open files. In UNIX System terminology, a process that has terminated, but whose parent has not yet waited for it, is called a *zombie*. The `ps`(1) command prints the state of a zombie process as *Z*. If we write a long-running program that `fork`s many child processes, they become zombies unless we wait for them and fetch their termination status.

Some systems provide ways to prevent the creation of zombies, as we describe in Section 10.7.

The final condition to consider is this: What happens when a process that has been inherited by `init` terminates? Does it become a zombie? The answer is "no," because `init` is written so that whenever one of its children terminates, `init` calls one of the `wait` functions to fetch the termination status. By doing this, `init` prevents the system from being clogged by zombies. When we say "one of `init`'s children," we mean either a process that `init` generates directly (such as `getty`, which we describe in Section 9.2) or a process whose parent has terminated and has been subsequently inherited by `init`.

8.6 `wait` and `waitpid` Functions

When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the `SIGCHLD` signal to the parent. Because the termination of a child is an asynchronous event—it can happen at any time while the parent is running—this signal is the asynchronous notification from the kernel to the parent. The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs: a signal handler. The default action for this signal is to be ignored. We describe these options in Chapter 10. For now, we need to be aware that a process that calls `wait` or `waitpid` can

- Block, if all of its children are still running
- Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched
- Return immediately with an error, if it doesn't have any child processes

If the process is calling `wait` because it received the `SIGCHLD` signal, we expect `wait` to return immediately. But if we call it at any random point in time, it can block.

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

pid_t wait(int *statloc);

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

Both return: process ID if OK, 0 (see later), or –1 on error

The differences between these two functions are as follows:

- The `wait` function can block the caller until a child process terminates, whereas `waitpid` has an option that prevents it from blocking.
- The `waitpid` function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.

If a child has already terminated and is a zombie, `wait` returns immediately with that child's status. Otherwise, it blocks the caller until a child terminates. If the caller blocks and has multiple children, `wait` returns when one terminates. We can always tell which child terminated, because the process ID is returned by the function.

For both functions, the argument *statloc* is a pointer to an integer. If this argument is not a null pointer, the termination status of the terminated process is stored in the location pointed to by the argument. If we don't care about the termination status, we simply pass a null pointer as this argument.

Traditionally, the integer status that these two functions return has been defined by the implementation, with certain bits indicating the exit status (for a normal return), other bits indicating the signal number (for an abnormal return), one bit indicating whether a core file was generated, and so on. POSIX.1 specifies that the termination status is to be looked at using various macros that are defined in `<sys/wait.h>`. Four mutually exclusive macros tell us how the process terminated, and they all begin with `WIF`. Based on which of these four macros is true, other macros are used to obtain the exit status, signal number, and the like. The four mutually exclusive macros are shown in Figure 8.4.

Macro	Description
`WIFEXITED(`*status*`)`	True if status was returned for a child that terminated normally. In this case, we can execute `WEXITSTATUS(`*status*`)` to fetch the low-order 8 bits of the argument that the child passed to `exit`, `_exit`, or `_Exit`.

`WIFSIGNALED(`*status*`)`	True if status was returned for a child that terminated abnormally, by receipt of a signal that it didn't catch. In this case, we can execute
	`WTERMSIG(`*status*`)`
	to fetch the signal number that caused the termination.
	Additionally, some implementations (but not the Single UNIX Specification) define the macro
	`WCOREDUMP(`*status*`)`
	that returns true if a core file of the terminated process was generated.

`WIFSTOPPED(`*status*`)`	True if status was returned for a child that is currently stopped. In this case, we can execute `WSTOPSIG(`*status*`)` to fetch the signal number that caused the child to stop.
`WIFCONTINUED(`*status*`)`	True if status was returned for a child that has been continued after a job control stop (XSI option; `waitpid` only).

Figure 8.4 Macros to examine the termination status returned by `wait` and `waitpid`

We'll discuss how a process can be stopped in Section 9.8 when we discuss job control.

Example

The function `pr_exit` in Figure 8.5 uses the macros from Figure 8.4 to print a description of the termination status. We'll call this function from numerous programs in the text. Note that this function handles the `WCOREDUMP` macro, if it is defined.

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

void
pr_exit(int status)
{
    if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n",
                WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number = %d%s\n",
                WTERMSIG(status),
#ifdef  WCOREDUMP
                WCOREDUMP(status) ? " (core file generated)" : "");
#else
                "");
#endif
    else if (WIFSTOPPED(status))
        printf("child stopped, signal number = %d\n",
                WSTOPSIG(status));
}
```

Figure 8.5 Print a description of the `exit` status

FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10 all support the `WCOREDUMP` macro. However, some platforms hide its definition if the `_POSIX_C_SOURCE` constant is defined (recall Section 2.7).

The program shown in Figure 8.6 calls the `pr_exit` function, demonstrating the various values for the termination status. If we run the program in Figure 8.6, we get

```
$ ./a.out
normal termination, exit status = 7
abnormal termination, signal number = 6 (core file generated)
abnormal termination, signal number = 8 (core file generated)
```

For now, we print the signal number from `WTERMSIG`. We can look at the `<signal.h>` header to verify that `SIGABRT` has a value of 6 and that `SIGFPE` has a value of 8. We'll see a portable way to map a signal number to a descriptive name in Section 10.22. □

As we mentioned, if we have more than one child, `wait` returns on termination of any of the children. But what if we want to wait for a specific process to terminate (assuming we know which process ID we want to wait for)? In older versions of the UNIX System, we would have to call `wait` and compare the returned process ID with the one we're interested in. If the terminated process wasn't the one we wanted, we would have to save the process ID and termination status and call `wait` again. We would need to continue doing this until the desired process terminated. The next time we wanted to wait for a specific process, we would go through the list of already terminated processes to see whether we had already waited for it, and if not, call `wait`

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

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

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        exit(7);

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        abort();                    /* generates SIGABRT */

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        status /= 0;                /* divide by 0 generates SIGFPE */

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    exit(0);
}
```

Figure 8.6 Demonstrate various `exit` statuses

again. What we need is a function that waits for a specific process. This functionality (and more) is provided by the POSIX.1 `waitpid` function.

The interpretation of the *pid* argument for `waitpid` depends on its value:

pid == –1	Waits for any child process. In this respect, `waitpid` is equivalent to `wait`.
pid > 0	Waits for the child whose process ID equals *pid*.
pid == 0	Waits for any child whose process group ID equals that of the calling process. (We discuss process groups in Section 9.4.)
pid < –1	Waits for any child whose process group ID equals the absolute value of *pid*.

The `waitpid` function returns the process ID of the child that terminated and stores the child's termination status in the memory location pointed to by *statloc*. With `wait`, the only real error is if the calling process has no children. (Another error return is possible, in case the function call is interrupted by a signal. We'll discuss this in Chapter 10.) With `waitpid`, however, it's also possible to get an error if the specified process or process group does not exist or is not a child of the calling process.

The *options* argument lets us further control the operation of `waitpid`. This argument either is 0 or is constructed from the bitwise OR of the constants in Figure 8.7.

> FreeBSD 8.0 and Solaris 10 support one additional, but nonstandard, *option* constant. `WNOWAIT` has the system keep the process whose termination status is returned by `waitpid` in a wait state, so that it may be waited for again.

Constant	Description
`WCONTINUED`	If the implementation supports job control, the status of any child specified by *pid* that has been continued after being stopped, but whose status has not yet been reported, is returned (XSI option).
`WNOHANG`	The `waitpid` function will not block if a child specified by *pid* is not immediately available. In this case, the return value is 0.
`WUNTRACED`	If the implementation supports job control, the status of any child specified by *pid* that has stopped, and whose status has not been reported since it has stopped, is returned. The `WIFSTOPPED` macro determines whether the return value corresponds to a stopped child process.

Figure 8.7 The *options* constants for `waitpid`

The `waitpid` function provides three features that aren't provided by the `wait` function.

1. The `waitpid` function lets us wait for one particular process, whereas the `wait` function returns the status of any terminated child. We'll return to this feature when we discuss the `popen` function.
2. The `waitpid` function provides a nonblocking version of `wait`. There are times when we want to fetch a child's status, but we don't want to block.
3. The `waitpid` function provides support for job control with the `WUNTRACED` and `WCONTINUED` options.

Example

Recall our discussion in Section 8.5 about zombie processes. If we want to write a process so that it `fork`s a child but we don't want to wait for the child to complete and we don't want the child to become a zombie until we terminate, the trick is to call `fork` twice. The program in Figure 8.8 does this.

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

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {      /* first child */
        if ((pid = fork()) < 0)
            err_sys("fork error");
        else if (pid > 0)
            exit(0);    /* parent from second fork == first child */

        /*
         * We're the second child; our parent becomes init as soon
         * as our real parent calls exit() in the statement above.
         * Here's where we'd continue executing, knowing that when
         * we're done, init will reap our status.
         */
        sleep(2);
        printf("second child, parent pid = %ld\n", (long)getppid());
        exit(0);
    }

    if (waitpid(pid, NULL, 0) != pid)   /* wait for first child */
        err_sys("waitpid error");

    /*
     * We're the parent (the original process); we continue executing,
     * knowing that we're not the parent of the second child.
     */
    exit(0);
}
```

Figure 8.8 Avoid zombie processes by calling `fork` twice

We call `sleep` in the second child to ensure that the first child terminates before printing the parent process ID. After a `fork`, either the parent or the child can continue executing; we never know which will resume execution first. If we didn't put the second child to sleep, and if it resumed execution after the `fork` before its parent, the parent process ID that it printed would be that of its parent, not process ID 1.

Executing the program in Figure 8.8 gives us

```
$ ./a.out
$ second child, parent pid = 1
```

Note that the shell prints its prompt when the original process terminates, which is before the second child prints its parent process ID. □

8.7 waitid Function

The Single UNIX Specification includes an additional function to retrieve the exit status of a process. The `waitid` function is similar to `waitpid`, but provides extra flexibility.

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

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

Returns: 0 if OK, –1 on error

Like `waitpid`, `waitid` allows a process to specify which children to wait for. Instead of encoding this information in a single argument combined with the process ID or process group ID, two separate arguments are used. The *id* parameter is interpreted based on the value of *idtype*. The types supported are summarized in Figure 8.9.

Constant	Description
`P_PID`	Wait for a particular process: *id* contains the process ID of the child to wait for.
`P_PGID`	Wait for any child process in a particular process group: *id* contains the process group ID of the children to wait for.
`P_ALL`	Wait for any child process: *id* is ignored.

Figure 8.9 The *idtype* constants for `waitid`

The *options* argument is a bitwise OR of the flags shown in Figure 8.10. These flags indicate which state changes the caller is interested in.

Constant	Description
`WCONTINUED`	Wait for a process that has previously stopped and has been continued, and whose status has not yet been reported.
`WEXITED`	Wait for processes that have exited.
`WNOHANG`	Return immediately instead of blocking if there is no child exit status available.
`WNOWAIT`	Don't destroy the child exit status. The child's exit status can be retrieved by a subsequent call to `wait`, `waitid`, or `waitpid`.
`WSTOPPED`	Wait for a process that has stopped and whose status has not yet been reported.

Figure 8.10 The *options* constants for `waitid`

At least one of `WCONTINUED`, `WEXITED`, or `WSTOPPED` must be specified in the *options* argument.

The *infop* argument is a pointer to a `siginfo` structure. This structure contains detailed information about the signal generated that caused the state change in the child process. The `siginfo` structure is discussed further in Section 10.14.

> Of the four platforms covered in this book, only Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10 provide support for `waitid`. Note, however, that Mac OS X 10.6.8 doesn't set all the information we expect in the `siginfo` structure.

8.8 wait3 and wait4 Functions

Most UNIX system implementations provide two additional functions: wait3 and wait4. Historically, these two variants descend from the BSD branch of the UNIX System. The only feature provided by these two functions that isn't provided by the wait, waitid, and waitpid functions is an additional argument that allows the kernel to return a summary of the resources used by the terminated process and all its child processes.

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

pid_t wait3(int *statloc, int options, struct rusage *rusage);

pid_t wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
```

Both return: process ID if OK, 0, or –1 on error

The resource information includes such statistics as the amount of user CPU time, amount of system CPU time, number of page faults, number of signals received, and the like. Refer to the getrusage(2) manual page for additional details. (This resource information differs from the resource limits we described in Section 7.11.) Figure 8.11 details the various arguments supported by the wait functions.

Function	*pid*	*options*	*rusage*	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
wait				•	•	•	•	•
waitid	•	•		•		•	•	•
waitpid	•	•		•	•	•	•	•
wait3		•	•		•	•	•	•
wait4	•	•	•		•	•	•	•

Figure 8.11 Arguments supported by wait functions on various systems

The wait3 function was included in earlier versions of the Single UNIX Specification. In Version 2, wait3 was moved to the legacy category; wait3 was removed from the specification in Version 3.

8.9 Race Conditions

For our purposes, a race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run. The fork function is a lively breeding ground for race conditions, if any of the logic after the fork either explicitly or implicitly depends on whether the parent or child runs first after the fork. In general, we cannot predict which process runs first. Even if we knew which process would run first, what happens after that process starts running depends on the system load and the kernel's scheduling algorithm.

We saw a potential race condition in the program in Figure 8.8 when the second child printed its parent process ID. If the second child runs before the first child, then its parent process will be the first child. But if the first child runs first and has enough time to `exit`, then the parent process of the second child is `init`. Even calling `sleep`, as we did, guarantees nothing. If the system was heavily loaded, the second child could resume after `sleep` returns, before the first child has a chance to run. Problems of this form can be difficult to debug because they tend to work "most of the time."

A process that wants to wait for a child to terminate must call one of the `wait` functions. If a process wants to wait for its parent to terminate, as in the program from Figure 8.8, a loop of the following form could be used:

```
while (getppid() != 1)
    sleep(1);
```

The problem with this type of loop, called *polling*, is that it wastes CPU time, as the caller is awakened every second to test the condition.

To avoid race conditions and to avoid polling, some form of signaling is required between multiple processes. Signals can be used for this purpose, and we describe one way to do this in Section 10.16. Various forms of interprocess communication (IPC) can also be used. We'll discuss some of these options in Chapters 15 and 17.

For a parent and child relationship, we often have the following scenario. After the `fork`, both the parent and the child have something to do. For example, the parent could update a record in a log file with the child's process ID, and the child might have to create a file for the parent. In this example, we require that each process tell the other when it has finished its initial set of operations, and that each wait for the other to complete, before heading off on its own. The following code illustrates this scenario:

```
#include  "apue.h"

TELL_WAIT();    /* set things up for TELL_xxx & WAIT_xxx */

if ((pid = fork()) < 0) {
    err_sys("fork error");
} else if (pid == 0) {            /* child */

    /* child does whatever is necessary ... */

    TELL_PARENT(getppid());     /* tell parent we're done */
    WAIT_PARENT();              /* and wait for parent */

    /* and the child continues on its way ... */

    exit(0);
}

/* parent does whatever is necessary ... */

TELL_CHILD(pid);            /* tell child we're done */
WAIT_CHILD();               /* and wait for child */

/* and the parent continues on its way ... */

exit(0);
```

We assume that the header `apue.h` defines whatever variables are required. The five routines `TELL_WAIT`, `TELL_PARENT`, `TELL_CHILD`, `WAIT_PARENT`, and `WAIT_CHILD` can be either macros or functions.

We'll show various ways to implement these `TELL` and `WAIT` routines in later chapters: Section 10.16 shows an implementation using signals; Figure 15.7 shows an implementation using pipes. Let's look at an example that uses these five routines.

Example

The program in Figure 8.12 outputs two strings: one from the child and one from the parent. The program contains a race condition because the output depends on the order in which the processes are run by the kernel and the length of time for which each process runs.

```
#include "apue.h"

static void charatatime(char *);

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
    }
    exit(0);
}

static void
charatatime(char *str)
{
    char    *ptr;
    int     c;

    setbuf(stdout, NULL);           /* set unbuffered */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

Figure 8.12 Program with a race condition

We set the standard output unbuffered, so every character output generates a `write`. The goal in this example is to allow the kernel to switch between the two processes as often as possible to demonstrate the race condition. (If we didn't do this, we might never see the type of output that follows. Not seeing the erroneous output doesn't

mean that the race condition doesn't exist; it simply means that we can't see it on this particular system.) The following actual output shows how the results can vary:

```
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
output from child
output from parent
```

We need to change the program in Figure 8.12 to use the `TELL` and `WAIT` functions. The program in Figure 8.13 does this. The lines preceded by a plus sign are new lines.

```
  #include "apue.h"

  static void charatatime(char *);

  int
  main(void)
  {
      pid_t   pid;

+     TELL_WAIT();
+
      if ((pid = fork()) < 0) {
          err_sys("fork error");
      } else if (pid == 0) {
+         WAIT_PARENT();      /* parent goes first */
          charatatime("output from child\n");
      } else {
          charatatime("output from parent\n");
+         TELL_CHILD(pid);
      }
      exit(0);
  }

  static void
  charatatime(char *str)
  {
      char    *ptr;
      int     c;

      setbuf(stdout, NULL);           /* set unbuffered */
      for (ptr = str; (c = *ptr++) != 0; )
          putc(c, stdout);
  }
```

Figure 8.13 Modification of Figure 8.12 to avoid race condition

When we run this program, the output is as we expect; there is no intermixing of output from the two processes.

In the program shown in Figure 8.13, the parent goes first. The child goes first if we change the lines following the fork to be

```
} else if (pid == 0) {
    charatatime("output from child\n");
    TELL_PARENT(getppid());
} else {
    WAIT_CHILD();        /* child goes first */
    charatatime("output from parent\n");
}
```

Exercise 8.4 continues this example. □

8.10 exec Functions

We mentioned in Section 8.3 that one use of the fork function is to create a new process (the child) that then causes another program to be executed by calling one of the exec functions. When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function. The process ID does not change across an exec, because a new process is not created; exec merely replaces the current process—its text, data, heap, and stack segments—with a brand-new program from disk.

There are seven different exec functions, but we'll often simply refer to "the exec function," which means that we could use any of the seven functions. These seven functions round out the UNIX System process control primitives. With fork, we can create new processes; and with the exec functions, we can initiate new programs. The exit function and the wait functions handle termination and waiting for termination. These are the only process control primitives we need. We'll use these primitives in later sections to build additional functions, such as popen and system.

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, ...
           /* (char *)0, char *const envp[] */ );

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

int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );

int execvp(const char *filename, char *const argv[]);

int fexecve(int fd, char *const argv[], char *const envp[]);
```

All seven return: –1 on error, no return on success

The first difference in these functions is that the first four take a pathname argument, the next two take a filename argument, and the last one takes a file descriptor argument. When a *filename* argument is specified,

- If *filename* contains a slash, it is taken as a pathname.
- Otherwise, the executable file is searched for in the directories specified by the `PATH` environment variable.

The `PATH` variable contains a list of directories, called path prefixes, that are separated by colons. For example, the *name=value* environment string

```
PATH=/bin:/usr/bin:/usr/local/bin/:.
```

specifies four directories to search. The last path prefix specifies the current directory. (A zero-length prefix also means the current directory. It can be specified as a colon at the beginning of the *value*, two colons in a row, or a colon at the end of the *value*.)

> There are security reasons for *never* including the current directory in the search path. See Garfinkel et al. [2003].

If either `execlp` or `execvp` finds an executable file using one of the path prefixes, but the file isn't a machine executable that was generated by the link editor, the function assumes that the file is a shell script and tries to invoke `/bin/sh` with the *filename* as input to the shell.

With `fexecve`, we avoid the issue of finding the correct executable file altogether and rely on the caller to do this. By using a file descriptor, the caller can verify the file is in fact the intended file and execute it without a race. Otherwise, a malicious user with appropriate privileges could replace the executable file (or a portion of the path to the executable file) after it has been located and verified, but before the caller can execute it (recall the discussion of TOCTTOU errors in Section 3.3).

The next difference concerns the passing of the argument list (`l` stands for list and `v` stands for vector). The functions `execl`, `execlp`, and `execle` require each of the command-line arguments to the new program to be specified as separate arguments. We mark the end of the arguments with a null pointer. For the other four functions (`execv`, `execvp`, `execve`, and `fexecve`), we have to build an array of pointers to the arguments, and the address of this array is the argument to these three functions.

Before using ISO C prototypes, the normal way to show the command-line arguments for the three functions `execl`, `execle`, and `execlp` was

```
char *arg0, char *arg1, ..., char *argn, (char *)0
```

This syntax explicitly shows that the final command-line argument is followed by a null pointer. If this null pointer is specified by the constant `0`, we must cast it to a pointer; if we don't, it's interpreted as an integer argument. If the size of an integer is different from the size of a `char *`, the actual arguments to the `exec` function will be wrong.

The final difference is the passing of the environment list to the new program. The three functions whose names end in an `e` (`execle`, `execve`, and `fexecve`) allow us to pass a pointer to an array of pointers to the environment strings. The other four

functions, however, use the environ variable in the calling process to copy the existing environment for the new program. (Recall our discussion of the environment strings in Section 7.9 and Figure 7.8. We mentioned that if the system supported such functions as setenv and putenv, we could change the current environment and the environment of any subsequent child processes, but we couldn't affect the environment of the parent process.) Normally, a process allows its environment to be propagated to its children, but in some cases, a process wants to specify a certain environment for a child. One example of the latter is the login program when a new login shell is initiated. Normally, login creates a specific environment with only a few variables defined and lets us, through the shell start-up file, add variables to the environment when we log in.

Before using ISO C prototypes, the arguments to execle were shown as

```
char *pathname, char *arg0, ..., char *argn, (char *)0, char *envp[]
```

This syntax specifically shows that the final argument is the address of the array of character pointers to the environment strings. The ISO C prototype doesn't show this, as all the command-line arguments, the null pointer, and the *envp* pointer are shown with the ellipsis notation (...).

The arguments for these seven exec functions are difficult to remember. The letters in the function names help somewhat. The letter p means that the function takes a *filename* argument and uses the PATH environment variable to find the executable file. The letter l means that the function takes a list of arguments and is mutually exclusive with the letter v, which means that it takes an *argv*[] vector. Finally, the letter e means that the function takes an *envp*[] array instead of using the current environment. Figure 8.14 shows the differences among these seven functions.

Function	*pathname*	*filename*	*fd*	Arg list	*argv*[]	environ	*envp*[]
execl	•			•		•	
execlp		•		•		•	
execle	•			•			•
execv	•				•	•	
execvp		•			•	•	
execve	•				•		•
fexecve			•		•		•
(letter in name)		p	f	l	v		e

Figure 8.14 Differences among the seven exec functions

Every system has a limit on the total size of the argument list and the environment list. From Section 2.5.2 and Figure 2.8, this limit is given by ARG_MAX. This value must be at least 4,096 bytes on a POSIX.1 system. We sometimes encounter this limit when using the shell's filename expansion feature to generate a list of filenames. On some systems, for example, the command

```
grep getrlimit /usr/share/man/*/*
```

can generate a shell error of the form

```
Argument list too long
```

> Historically, the limit in older System V implementations was 5,120 bytes. Older BSD systems had a limit of 20,480 bytes. The limit in current systems is much higher. (See the output from the program in Figure 2.14, which is summarized in Figure 2.15.)

To get around the limitation in argument list size, we can use the `xargs`(1) command to break up long argument lists. To look for all the occurrences of `getrlimit` in the man pages on our system, we could use

```
find /usr/share/man -type f -print | xargs grep getrlimit
```

If the man pages on our system are compressed, however, we could try

```
find /usr/share/man -type f -print | xargs bzgrep getrlimit
```

We use the `type -f` option to the `find` command to restrict the list so that it contains only regular files, because the `grep` commands can't search for patterns in directories, and we want to avoid unnecessary error messages.

We've mentioned that the process ID does not change after an `exec`, but the new program inherits additional properties from the calling process:

- Process ID and parent process ID
- Real user ID and real group ID
- Supplementary group IDs
- Process group ID
- Session ID
- Controlling terminal
- Time left until alarm clock
- Current working directory
- Root directory
- File mode creation mask
- File locks
- Process signal mask
- Pending signals
- Resource limits
- Nice value (on XSI-conformant systems; see Section 8.16)
- Values for `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime`

The handling of open files depends on the value of the close-on-exec flag for each descriptor. Recall from Figure 3.7 and our mention of the `FD_CLOEXEC` flag in Section 3.14 that every open descriptor in a process has a close-on-exec flag. If this flag is set, the descriptor is closed across an `exec`. Otherwise, the descriptor is left open across the `exec`. The default is to leave the descriptor open across the `exec` unless we specifically set the close-on-exec flag using `fcntl`.

POSIX.1 specifically requires that open directory streams (recall the `opendir`

function from Section 4.22) be closed across an exec. This is normally done by the opendir function calling fcntl to set the close-on-exec flag for the descriptor corresponding to the open directory stream.

Note that the real user ID and the real group ID remain the same across the exec, but the effective IDs can change, depending on the status of the set-user-ID and the set-group-ID bits for the program file that is executed. If the set-user-ID bit is set for the new program, the effective user ID becomes the owner ID of the program file. Otherwise, the effective user ID is not changed (it's not set to the real user ID). The group ID is handled in the same way.

In many UNIX system implementations, only one of these seven functions, execve, is a system call within the kernel. The other six are just library functions that eventually invoke this system call. We can illustrate the relationship among these seven functions as shown in Figure 8.15.

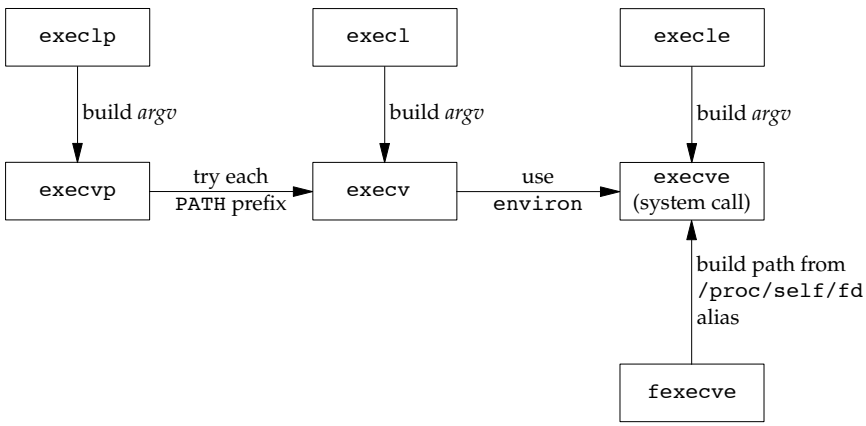

Figure 8.15 Relationship of the seven exec functions

In this arrangement, the library functions execlp and execvp process the PATH environment variable, looking for the first path prefix that contains an executable file named *filename*. The fexecve library function uses /proc to convert the file descriptor argument into a pathname that can be used by execve to execute the program.

> This describes how fexecve is implemented in FreeBSD 8.0 and Linux 3.2.0. Other systems might take a different approach. For example, a system without /proc or /dev/fd could implement fexecve as a system call veneer that translates the file descriptor argument into an i-node pointer, implement execve as a system call veneer that translates the pathname argument into an i-node pointer, and place all the rest of the exec code common to both execve and fexecve in a separate function to be called with an i-node pointer for the file to be executed.

Example

The program in Figure 8.16 demonstrates the exec functions.

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

char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* specify pathname, specify environment */
        if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
                "MY ARG2", (char *)0, env_init) < 0)
            err_sys("execle error");
    }

    if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* specify filename, inherit environment */
        if (execlp("echoall", "echoall", "only 1 arg", (char *)0) < 0)
            err_sys("execlp error");
    }

    exit(0);
}
```

Figure 8.16 Example of exec functions

We first call `execle`, which requires a pathname and a specific environment. The next call is to `execlp`, which uses a filename and passes the caller's environment to the new program. The only reason the call to `execlp` works is that the directory `/home/sar/bin` is one of the current path prefixes. Note also that we set the first argument, `argv[0]` in the new program, to be the filename component of the pathname. Some shells set this argument to be the complete pathname. This is a convention only; we can set `argv[0]` to any string we like. The `login` command does this when it executes the shell. Before executing the shell, `login` adds a dash as a prefix to `argv[0]` to indicate to the shell that it is being invoked as a login shell. A login shell will execute the start-up profile commands, whereas a nonlogin shell will not.

The program `echoall` that is executed twice in the program in Figure 8.16 is shown in Figure 8.17. It is a trivial program that echoes all its command-line arguments and its entire environment list.

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int         i;
    char        **ptr;
    extern char **environ;

    for (i = 0; i < argc; i++)      /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);

    for (ptr = environ; *ptr != 0; ptr++)   /* and all env strings */
        printf("%s\n", *ptr);

    exit(0);
}
```

Figure 8.17 Echo all command-line arguments and all environment strings

When we execute the program from Figure 8.16, we get

```
$ ./a.out
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
$ argv[0]: echoall
argv[1]: only 1 arg
USER=sar
LOGNAME=sar
SHELL=/bin/bash
                                    47 more lines that aren't shown
HOME=/home/sar
```

Note that the shell prompt appeared before the printing of `argv[0]` from the second `exec`. This occurred because the parent did not `wait` for this child process to finish. □

8.11 Changing User IDs and Group IDs

In the UNIX System, privileges, such as being able to change the system's notion of the current date, and access control, such as being able to read or write a particular file, are based on user and group IDs. When our programs need additional privileges or need to gain access to resources that they currently aren't allowed to access, they need to change their user or group ID to an ID that has the appropriate privilege or access. Similarly, when our programs need to lower their privileges or prevent access to certain resources, they do so by changing either their user ID or group ID to an ID without the privilege or ability access to the resource.

In general, we try to use the *least-privilege* model when we design our applications. According to this model, our programs should use the least privilege necessary to accomplish any given task. This reduces the risk that security might be compromised by a malicious user trying to trick our programs into using their privileges in unintended ways.

We can set the real user ID and effective user ID with the `setuid` function. Similarly, we can set the real group ID and the effective group ID with the `setgid` function.

```
#include <unistd.h>

int setuid(uid_t uid);

int setgid(gid_t gid);
```

Both return: 0 if OK, –1 on error

There are rules for who can change the IDs. Let's consider only the user ID for now. (Everything we describe for the user ID also applies to the group ID.)

1. If the process has superuser privileges, the `setuid` function sets the real user ID, effective user ID, and saved set-user-ID to *uid*.
2. If the process does not have superuser privileges, but *uid* equals either the real user ID or the saved set-user-ID, `setuid` sets only the effective user ID to *uid*. The real user ID and the saved set-user-ID are not changed.
3. If neither of these two conditions is true, `errno` is set to `EPERM` and –1 is returned.

Here, we are assuming that `_POSIX_SAVED_IDS` is true. If this feature isn't provided, then delete all preceding references to the saved set-user-ID.

> The saved IDs are a mandatory feature in the 2001 version of POSIX.1. They were optional in older versions of POSIX. To see whether an implementation supports this feature, an application can test for the constant `_POSIX_SAVED_IDS` at compile time or call `sysconf` with the `_SC_SAVED_IDS` argument at runtime.

We can make a few statements about the three user IDs that the kernel maintains.

1. Only a superuser process can change the real user ID. Normally, the real user ID is set by the `login`(1) program when we log in and never changes. Because `login` is a superuser process, it sets all three user IDs when it calls `setuid`.
2. The effective user ID is set by the `exec` functions only if the set-user-ID bit is set for the program file. If the set-user-ID bit is not set, the `exec` functions leave the effective user ID as its current value. We can call `setuid` at any time to set the effective user ID to either the real user ID or the saved set-user-ID. Naturally, we can't set the effective user ID to any random value.
3. The saved set-user-ID is copied from the effective user ID by `exec`. If the file's set-user-ID bit is set, this copy is saved after `exec` stores the effective user ID from the file's user ID.

Figure 8.18 summarizes the various ways these three user IDs can be changed.

ID	exec		setuid(*uid*)	
	set-user-ID bit off	set-user-ID bit on	superuser	unprivileged user
real user ID	unchanged	unchanged	set to *uid*	unchanged
effective user ID	unchanged	set from user ID of program file	set to *uid*	set to *uid*
saved set-user ID	copied from effective user ID	copied from effective user ID	set to *uid*	unchanged

Figure 8.18 Ways to change the three user IDs

Note that we can obtain only the current value of the real user ID and the effective user ID with the functions `getuid` and `geteuid` from Section 8.2. We have no portable way to obtain the current value of the saved set-user-ID.

> FreeBSD 8.0 and LINUX 3.2.0 provide the `getresuid` and `getresgid` functions, which can be used to get the saved set-user-ID and saved set-group-ID, respectively.

`setreuid` and `setregid` Functions

Historically, BSD supported the swapping of the real user ID and the effective user ID with the `setreuid` function.

```
#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);

int setregid(gid_t rgid, gid_t egid);
```

Both return: 0 if OK, –1 on error

We can supply a value of –1 for any of the arguments to indicate that the corresponding ID should remain unchanged.

The rule is simple: an unprivileged user can always swap between the real user ID and the effective user ID. This allows a set-user-ID program to swap to the user's normal permissions and swap back again later for set-user-ID operations. When the saved set-user-ID feature was introduced with POSIX.1, the rule was enhanced to also allow an unprivileged user to set its effective user ID to its saved set-user-ID.

> Both `setreuid` and `setregid` are included in the XSI option in POSIX.1. As such, all UNIX System implementations are expected to provide support for them.
>
> 4.3BSD didn't have the saved set-user-ID feature described earlier; it used `setreuid` and `setregid` instead. This allowed an unprivileged user to swap back and forth between the two values. Be aware, however, that when programs that used this feature spawned a shell, they had to set the real user ID to the normal user ID before the `exec`. If they didn't do this, the real user ID could be privileged (from the swap done by `setreuid`) and the shell process could call `setreuid` to swap the two and assume the permissions of the more privileged user. As a defensive programming measure to solve this problem, programs set both the real user ID and the effective user ID to the normal user ID before the call to `exec` in the child.

`seteuid` and `setegid` Functions

POSIX.1 includes the two functions `seteuid` and `setegid`. These functions are similar to `setuid` and `setgid`, but only the effective user ID or effective group ID is changed.

```
#include <unistd.h>
int seteuid(uid_t uid);
int setegid(gid_t gid);
                                        Both return: 0 if OK, –1 on error
```

An unprivileged user can set its effective user ID to either its real user ID or its saved set-user-ID. For a privileged user, only the effective user ID is set to *uid*. (This behavior differs from that of the `setuid` function, which changes all three user IDs.)

Figure 8.19 summarizes all the functions that we've described here that modify the three user IDs.

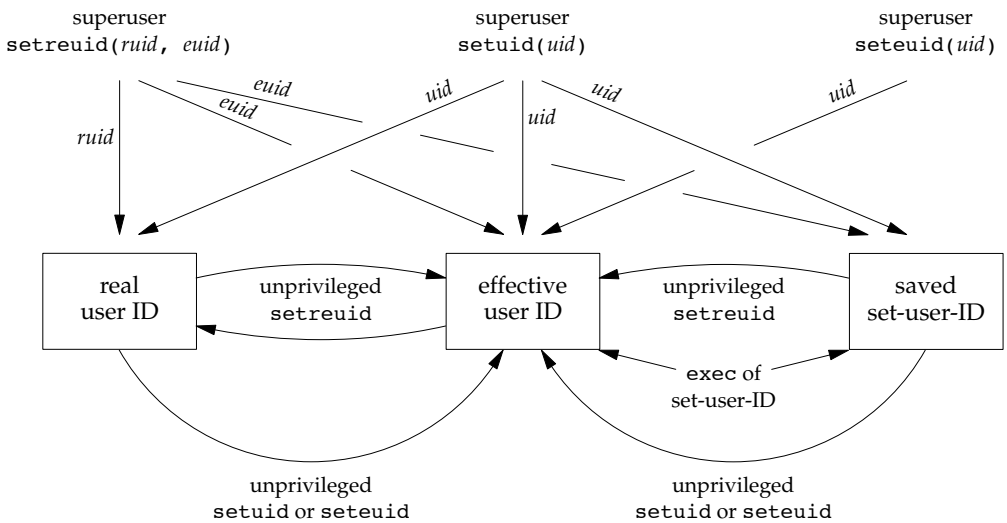

Figure 8.19 Summary of all the functions that set the various user IDs

Group IDs

Everything that we've said so far in this section also applies in a similar fashion to group IDs. The supplementary group IDs are not affected by `setgid`, `setregid`, or `setegid`.

Example

To see the utility of the saved set-user-ID feature, let's examine the operation of a program that uses it. We'll look at the `at`(1) program, which we can use to schedule commands to be run at some time in the future.

> On Linux 3.2.0, the `at` program is installed set-user-ID to user `daemon`. On FreeBSD 8.0, Mac OS X 10.6.8, and Solaris 10, the `at` program is installed set-user-ID to user `root`. This allows the `at` command to write privileged files owned by the daemon that will run the commands on behalf of the user running the `at` command. On Linux 3.2.0, the programs are run by the `atd`(8) daemon. On FreeBSD 8.0 and Solaris 10, the programs are run by the `cron`(1M) daemon. On Mac OS X 10.6.8, the programs are run by the `launchd`(8) daemon.

To prevent being tricked into running commands that we aren't allowed to run, or reading or writing files that we aren't allowed to access, the `at` command and the daemon that ultimately runs the commands on our behalf have to switch between sets of privileges: ours and those of the daemon. The following steps take place.

1. Assuming that the `at` program file is owned by `root` and has its set-user-ID bit set, when we run it, we have

   real user ID = our user ID (unchanged)
   effective user ID = `root`
   saved set-user-ID = `root`

2. The first thing the `at` command does is reduce its privileges so that it runs with our privileges. It calls the `seteuid` function to set the effective user ID to our real user ID. After this, we have

   real user ID = our user ID (unchanged)
   effective user ID = our user ID
   saved set-user-ID = `root` (unchanged)

3. The `at` program runs with our privileges until it needs to access the configuration files that control which commands are to be run and the time at which they need to run. These files are owned by the daemon that will run the commands for us. The `at` command calls `seteuid` to set the effective user ID to `root`. This call is allowed because the argument to `seteuid` equals the saved set-user-ID. (This is why we need the saved set-user-ID.) After this, we have

   real user ID = our user ID (unchanged)
   effective user ID = `root`
   saved set-user-ID = `root` (unchanged)

   Because the effective user ID is `root`, file access is allowed.

4. After the files are modified to record the commands to be run and the time at which they are to be run, the `at` command lowers its privileges by calling

`seteuid` to set its effective user ID to our user ID. This prevents any accidental misuse of privilege. At this point, we have

```
real user ID = our user ID (unchanged)
effective user ID = our user ID
saved set-user-ID = root (unchanged)
```

5. The daemon starts out running with `root` privileges. To run commands on our behalf, the daemon calls `fork` and the child calls `setuid` to change its user ID to our user ID. Because the child is running with `root` privileges, this changes all of the IDs. We have

```
real user ID = our user ID
effective user ID = our user ID
saved set-user-ID = our user ID
```

   Now the daemon can safely execute commands on our behalf, because it can access only the files to which we normally have access. We have no additional permissions.

By using the saved set-user-ID in this fashion, we can use the extra privileges granted to us by the set-user-ID of the program file only when we need elevated privileges. Any other time, however, the process runs with our normal permissions. If we weren't able to switch back to the saved set-user-ID at the end, we might be tempted to retain the extra permissions the whole time we were running (which is asking for trouble). □

8.12 Interpreter Files

All contemporary UNIX systems support interpreter files. These files are text files that begin with a line of the form

```
#! pathname [ optional-argument ]
```

The space between the exclamation point and the *pathname* is optional. The most common of these interpreter files begin with the line

```
#!/bin/sh
```

The *pathname* is normally an absolute pathname, since no special operations are performed on it (i.e., `PATH` is not used). The recognition of these files is done within the kernel as part of processing the `exec` system call. The actual file that gets executed by the kernel is not the interpreter file, but rather the file specified by the *pathname* on the first line of the interpreter file. Be sure to differentiate between the interpreter file—a text file that begins with `#!`—and the interpreter, which is specified by the *pathname* on the first line of the interpreter file.

Be aware that systems place a size limit on the first line of an interpreter file. This limit includes the `#!`, the *pathname*, the optional argument, the terminating newline, and any spaces.

> On FreeBSD 8.0, this limit is 4,097 bytes. On Linux 3.2.0, the limit is 128 bytes. Mac OS X 10.6.8 supports a limit of 513 bytes, whereas Solaris 10 places the limit at 1,024 bytes.

Example

Let's look at an example to see what the kernel does with the arguments to the exec function when the file being executed is an interpreter file and the optional argument on the first line of the interpreter file. The program in Figure 8.20 execs an interpreter file.

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

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {          /* child */
        if (execl("/home/sar/bin/testinterp",
                  "testinterp", "myarg1", "MY ARG2", (char *)0) < 0)
            err_sys("execl error");
    }
    if (waitpid(pid, NULL, 0) < 0)  /* parent */
        err_sys("waitpid error");
    exit(0);
}
```

Figure 8.20 A program that execs an interpreter file

The following shows the contents of the one-line interpreter file that is executed and the result from running the program in Figure 8.20:

```
$ cat /home/sar/bin/testinterp
#!/home/sar/bin/echoarg foo
$ ./a.out
argv[0]: /home/sar/bin/echoarg
argv[1]: foo
argv[2]: /home/sar/bin/testinterp
argv[3]: myarg1
argv[4]: MY ARG2
```

The program echoarg (the interpreter) just echoes each of its command-line arguments. (This is the program from Figure 7.4.) Note that when the kernel execs the interpreter (/home/sar/bin/echoarg), argv[0] is the *pathname* of the interpreter, argv[1] is the optional argument from the interpreter file, and the remaining arguments are the *pathname* (/home/sar/bin/testinterp) and the second and third arguments from the call to execl in the program shown in Figure 8.20 (myarg1 and MY ARG2). Both argv[1] and argv[2] from the call to execl have been shifted right two positions. Note that the kernel takes the *pathname* from the execl call instead of the first argument (testinterp), on the assumption that the *pathname* might contain more information than the first argument. □

Example

A common use for the optional argument following the interpreter *pathname* is to specify the `-f` option for programs that support this option. For example, an `awk`(1) program can be executed as

```
awk -f myfile
```

which tells `awk` to read the `awk` program from the file `myfile`.

> Systems derived from UNIX System V often include two versions of the `awk` language. On these systems, `awk` is often called "old `awk`" and corresponds to the original version distributed with Version 7. In contrast, `nawk` (new `awk`) contains numerous enhancements and corresponds to the language described in Aho, Kernighan, and Weinberger [1988]. This newer version provides access to the command-line arguments, which we need for the example that follows. Solaris 10 provides both versions.
>
> The `awk` program is one of the utilities included by POSIX in its 1003.2 standard, which is now part of the base POSIX.1 specification in the Single UNIX Specification. This utility is also based on the language described in Aho, Kernighan, and Weinberger [1988].
>
> The version of `awk` in Mac OS X 10.6.8 is based on the Bell Laboratories version, which has been placed in the public domain. FreeBSD 8.0 and some Linux distributions ship with GNU `awk`, called `gawk`, which is linked to the name `awk`. `gawk` conforms to the POSIX standard, but also includes other extensions. Because they are more up-to-date, `gawk` and the version of `awk` from Bell Laboratories are preferred to either `nawk` or old `awk`. (The Bell Labs version of `awk` is available at `http://cm.bell-labs.com/cm/cs/awkbook/index.html`.)

Using the `-f` option with an interpreter file lets us write

```
#!/bin/awk -f
(awk program follows in the interpreter file)
```

For example, Figure 8.21 shows `/usr/local/bin/awkexample` (an interpreter file).

```
#!/usr/bin/awk -f
# Note: on Solaris, use nawk instead
BEGIN {
    for (i = 0; i < ARGC; i++)
        printf "ARGV[%d] = %s\n", i, ARGV[i]
    exit
}
```

Figure 8.21 An awk program as an interpreter file

If one of the path prefixes is `/usr/local/bin`, we can execute the program in Figure 8.21 (assuming that we've turned on the execute bit for the file) as

```
$ awkexample file1 FILENAME2 f3
ARGV[0] = awk
ARGV[1] = file1
ARGV[2] = FILENAME2
ARGV[3] = f3
```

When `/bin/awk` is executed, its command-line arguments are

```
/bin/awk -f /usr/local/bin/awkexample file1 FILENAME2 f3
```

The pathname of the interpreter file (`/usr/local/bin/awkexample`) is passed to the interpreter. The filename portion of this pathname (what we typed to the shell) isn't adequate, because the interpreter (`/bin/awk` in this example) can't be expected to use the `PATH` variable to locate files. When it reads the interpreter file, `awk` ignores the first line, since the pound sign is `awk`'s comment character.

We can verify these command-line arguments with the following commands:

```
$ /bin/su                                        become superuser
Password:                                        enter superuser password
# mv /usr/bin/awk /usr/bin/awk.save              save the original program
# cp /home/sar/bin/echoarg /usr/bin/awk          and replace it temporarily
# suspend                                        suspend the superuser shell
[1] + Stopped          /bin/su                   using job control
$ awkexample file1 FILENAME2 f3
argv[0]: /bin/awk
argv[1]: -f
argv[2]: /usr/local/bin/awkexample
argv[3]: file1
argv[4]: FILENAME2
argv[5]: f3
$ fg                                             resume superuser shell using job control
/bin/su
# mv /usr/bin/awk.save /usr/bin/awk              restore the original program
# exit                                           and exit the superuser shell
```

In this example, the `-f` option for the interpreter is required. As we said, this tells `awk` where to look for the `awk` program. If we remove the `-f` option from the interpreter file, an error message usually results when we try to run it. The exact text of the message varies, depending on where the interpreter file is stored and whether the remaining arguments represent existing files. This is because the command-line arguments in this case are

```
/bin/awk /usr/local/bin/awkexample file1 FILENAME2 f3
```

and `awk` is trying to interpret the string `/usr/local/bin/awkexample` as an `awk` program. If we couldn't pass at least a single optional argument to the interpreter (`-f` in this case), these interpreter files would be usable only with the shells. □

Are interpreter files required? Not really. They provide an efficiency gain for the user at some expense in the kernel (since it's the kernel that recognizes these files). Interpreter files are useful for the following reasons.

1. They hide that certain programs are scripts in some other language. For example, to execute the program in Figure 8.21, we just say

   `awkexample` *optional-arguments*

instead of needing to know that the program is really an `awk` script that we would otherwise have to execute as

```
awk -f awkexample optional-arguments
```

2. Interpreter scripts provide an efficiency gain. Consider the previous example again. We could still hide that the program is an `awk` script, by wrapping it in a shell script:

```
awk 'BEGIN {
    for (i = 0; i < ARGC; i++)
        printf "ARGV[%d] = %s\n", i, ARGV[i]
    exit
}' $*
```

The problem with this solution is that more work is required. First, the shell reads the command and tries to `execlp` the filename. Because the shell script is an executable file but isn't a machine executable, an error is returned and `execlp` assumes that the file is a shell script (which it is). Then `/bin/sh` is executed with the pathname of the shell script as its argument. The shell correctly runs our script, but to run the `awk` program, the shell does a `fork`, `exec`, and `wait`. Thus there is more overhead involved in replacing an interpreter script with a shell script.

3. Interpreter scripts let us write shell scripts using shells other than `/bin/sh`. When it finds an executable file that isn't a machine executable, `execlp` has to choose a shell to invoke, and it always uses `/bin/sh`. Using an interpreter script, however, we can simply write

```
#!/bin/csh
(C shell script follows in the interpreter file)
```

Again, we could wrap all of this in a `/bin/sh` script (that invokes the C shell), as we described earlier, but more overhead is required.

None of this would work as we've shown here if the three shells and `awk` didn't use the pound sign as their comment character.

8.13 `system` Function

It is convenient to execute a command string from within a program. For example, assume that we want to put a time-and-date stamp into a certain file. We could use the functions described in Section 6.10 to do this: call `time` to get the current calendar time, then call `localtime` to convert it to a broken-down time, then call `strftime` to format the result, and finally write the result to the file. It is much easier, however, to say

```
system("date > file");
```

ISO C defines the `system` function, but its operation is strongly system dependent. POSIX.1 includes the `system` interface, expanding on the ISO C definition to describe its behavior in a POSIX environment.

```
#include <stdlib.h>

int system(const char *cmdstring);

                                                    Returns: (see below)
```

If *cmdstring* is a null pointer, `system` returns nonzero only if a command processor is available. This feature determines whether the `system` function is supported on a given operating system. Under the UNIX System, `system` is always available.

Because `system` is implemented by calling `fork`, `exec`, and `waitpid`, there are three types of return values.

1. If either the `fork` fails or `waitpid` returns an error other than `EINTR`, `system` returns –1 with `errno` set to indicate the error.
2. If the `exec` fails, implying that the shell can't be executed, the return value is as if the shell had executed `exit(127)`.
3. Otherwise, all three functions—`fork`, `exec`, and `waitpid`—succeed, and the return value from `system` is the termination status of the shell, in the format specified for `waitpid`.

> Some older implementations of `system` returned an error (`EINTR`) if `waitpid` was interrupted by a caught signal. Because there is no strategy that an application can use to recover from this type of error (the process ID of the child is hidden from the caller), POSIX later added the requirement that `system` not return an error in this case. (We discuss interrupted system calls in Section 10.5.)

Figure 8.22 shows an implementation of the `system` function. The one feature that it doesn't handle is signals. We'll update this function with signal handling in Section 10.18.

The shell's `-c` option tells it to take the next command-line argument—*cmdstring*, in this case—as its command input instead of reading from standard input or from a given file. The shell parses this null-terminated C string and breaks it up into separate command-line arguments for the command. The actual command string that is passed to the shell can contain any valid shell commands. For example, input and output redirection using < and > can be used.

If we didn't use the shell to execute the command, but tried to execute the command ourself, it would be more difficult. First, we would want to call `execlp`, instead of `execl`, to use the `PATH` variable, like the shell. We would also have to break up the null-terminated C string into separate command-line arguments for the call to `execlp`. Finally, we wouldn't be able to use any of the shell metacharacters.

Note that we call `_exit` instead of `exit`. We do this to prevent any standard I/O buffers, which would have been copied from the parent to the child across the `fork`, from being flushed in the child.

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

int
system(const char *cmdstring)   /* version without signal handling */
{
    pid_t   pid;
    int     status;

    if (cmdstring == NULL)
        return(1);      /* always a command processor with UNIX */

    if ((pid = fork()) < 0) {
        status = -1;    /* probably out of processes */
    } else if (pid == 0) {              /* child */
        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);     /* execl error */
    } else {                            /* parent */
        while (waitpid(pid, &status, 0) < 0) {
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
        }
    }

    return(status);
}
```

Figure 8.22 The `system` function, without signal handling

We can test this version of `system` with the program shown in Figure 8.23. (The `pr_exit` function was defined in Figure 8.5.) Running the program in Figure 8.23 gives us

```
$ ./a.out
Sat Feb 25 19:36:59 EST 2012
normal termination, exit status = 0     for date
sh: nosuchcommand: command not found
normal termination, exit status = 127   for nosuchcommand
sar      console  Jan  1 14:59
sar      ttys000  Feb  7 19:08
sar      ttys001  Jan 15 15:28
sar      ttys002  Jan 15 21:50
sar      ttys003  Jan 21 16:02
normal termination, exit status = 44    for exit
```

The advantage in using `system`, instead of using `fork` and `exec` directly, is that `system` does all the required error handling and (in our next version of this function in Section 10.18) all the required signal handling.

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

int
main(void)
{
    int     status;

    if ((status = system("date")) < 0)
        err_sys("system() error");

    pr_exit(status);

    if ((status = system("nosuchcommand")) < 0)
        err_sys("system() error");

    pr_exit(status);

    if ((status = system("who; exit 44")) < 0)
        err_sys("system() error");

    pr_exit(status);

    exit(0);
}
```

Figure 8.23 Calling the system function

Earlier systems, including SVR3.2 and 4.3BSD, didn't have the waitpid function available. Instead, the parent waited for the child, using a statement such as

```
while ((lastpid = wait(&status)) != pid && lastpid != -1)
    ;
```

A problem occurs if the process that calls system has spawned its own children before calling system. Because the while statement above keeps looping until the child that was generated by system terminates, if any children of the process terminate before the process identified by pid, then the process ID and termination status of these other children are discarded by the while statement. Indeed, this inability to wait for a specific child is one of the reasons given in the POSIX.1 Rationale for including the waitpid function. We'll see in Section 15.3 that the same problem occurs with the popen and pclose functions if the system doesn't provide a waitpid function.

Set-User-ID Programs

What happens if we call system from a set-user-ID program? Doing so creates a security hole and should never be attempted. Figure 8.24 shows a simple program that just calls system for its command-line argument.

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int     status;

    if (argc < 2)
        err_quit("command-line argument required");

    if ((status = system(argv[1])) < 0)
        err_sys("system() error");

    pr_exit(status);

    exit(0);
}
```

Figure 8.24 Execute the command-line argument using `system`

We'll compile this program into the executable file `tsys`.

Figure 8.25 shows another simple program that prints its real and effective user IDs.

```
#include "apue.h"

int
main(void)
{
    printf("real uid = %d, effective uid = %d\n", getuid(), geteuid());
    exit(0);
}
```

Figure 8.25 Print real and effective user IDs

We'll compile this program into the executable file `printuids`. Running both programs gives us the following:

```
$ tsys printuids                        normal execution, no special privileges
real uid = 205, effective uid = 205
normal termination, exit status = 0
$ su                                    become superuser
Password:                               enter superuser password
# chown root tsys                       change owner
# chmod u+s tsys                        make set-user-ID
# ls -l tsys                            verify file's permissions and owner
-rwsrwxr-x  1 root    7888 Feb 25 22:13 tsys
# exit                                  leave superuser shell
$ tsys printuids
real uid = 205, effective uid = 0       oops, this is a security hole
normal termination, exit status = 0
```

The superuser permissions that we gave the `tsys` program are retained across the `fork` and `exec` that are done by `system`.

> Some implementations have closed this security hole by changing `/bin/sh` to reset the effective user ID to the real user ID when they don't match. On these systems, the previous example doesn't work as shown. Instead, the same effective user ID will be printed regardless of the status of the set-user-ID bit on the program calling `system`.

If it is running with special permissions—either set-user-ID or set-group-ID—and wants to spawn another process, a process should use `fork` and `exec` directly, being certain to change back to normal permissions after the `fork`, before calling `exec`. The `system` function should *never* be used from a set-user-ID or a set-group-ID program.

> One reason for this admonition is that `system` invokes the shell to parse the command string, and the shell uses its `IFS` variable as the input field separator. Older versions of the shell didn't reset this variable to a normal set of characters when invoked. As a result, a malicious user could set `IFS` before `system` was called, causing `system` to execute a different program.

8.14 Process Accounting

Most UNIX systems provide an option to do process accounting. When enabled, the kernel writes an accounting record each time a process terminates. These accounting records typically contain a small amount of binary data with the name of the command, the amount of CPU time used, the user ID and group ID, the starting time, and so on. We'll take a closer look at these accounting records in this section, as it gives us a chance to look at processes again and to use the `fread` function from Section 5.9.

> Process accounting is not specified by any of the standards. Thus all the implementations have annoying differences. For example, the I/O counts maintained on Solaris 10 are in units of bytes, whereas FreeBSD 8.0 and Mac OS X 10.6.8 maintain units of blocks, although there is no distinction between different block sizes, making the counter effectively useless. Linux 3.2.0, on the other hand, doesn't try to maintain I/O statistics at all.
>
> Each implementation also has its own set of administrative commands to process raw accounting data. For example, Solaris provides `runacct`(1m) and `acctcom`(1), whereas FreeBSD provides the `sa`(8) command to process and summarize the raw accounting data.

A function we haven't described (`acct`) enables and disables process accounting. The only use of this function is from the `accton`(8) command (which happens to be one of the few similarities among platforms). A superuser executes `accton` with a pathname argument to enable accounting. The accounting records are written to the specified file, which is usually `/var/account/acct` on FreeBSD and Mac OS X, `/var/log/account/pacct` on Linux, and `/var/adm/pacct` on Solaris. Accounting is turned off by executing `accton` without any arguments.

The structure of the accounting records is defined in the header `<sys/acct.h>`. Although the implementation of each system differs, the accounting records look something like

```
typedef  u_short comp_t;  /* 3-bit base 8 exponent; 13-bit fraction */

struct  acct
{
  char   ac_flag;    /* flag (see Figure 8.26) */
  char   ac_stat;    /* termination status (signal & core flag only) */
                     /* (Solaris only) */
  uid_t  ac_uid;     /* real user ID */
  gid_t  ac_gid;     /* real group ID */
  dev_t  ac_tty;     /* controlling terminal */
  time_t ac_btime;   /* starting calendar time */
  comp_t ac_utime;   /* user CPU time */
  comp_t ac_stime;   /* system CPU time */
  comp_t ac_etime;   /* elapsed time */
  comp_t ac_mem;     /* average memory usage */
  comp_t ac_io;      /* bytes transferred (by read and write) */
                     /* "blocks" on BSD systems */
  comp_t ac_rw;      /* blocks read or written */
                     /* (not present on BSD systems) */
  char   ac_comm[8]; /* command name: [8] for Solaris, */
                     /* [10] for Mac OS X, [16] for FreeBSD, and */
                     /* [17] for Linux */
};
```

Times are recorded in units of clock ticks on most platforms, but FreeBSD stores microseconds instead. The `ac_flag` member records certain events during the execution of the process. These events are described in Figure 8.26.

The data required for the accounting record, such as CPU times and number of characters transferred, is kept by the kernel in the process table and initialized whenever a new process is created, as in the child after a `fork`. Each accounting record is written when the process terminates. This has two consequences.

First, we don't get accounting records for processes that never terminate. Processes like `init` that run for the lifetime of the system don't generate accounting records. This also applies to kernel daemons, which normally don't exit.

Second, the order of the records in the accounting file corresponds to the termination order of the processes, not the order in which they were started. To know the starting order, we would have to go through the accounting file and sort by the starting calendar time. But this isn't perfect, since calendar times are in units of seconds (Section 1.10), and it's possible for many processes to be started in any given second. Alternatively, the elapsed time is given in clock ticks, which are usually between 60 and 128 ticks per second. But we don't know the ending time of a process; all we know is its starting time and ending order. Thus, even though the elapsed time is more accurate than the starting time, we still can't reconstruct the exact starting order of various processes, given the data in the accounting file.

The accounting records correspond to processes, not programs. A new record is initialized by the kernel for the child after a `fork`, not when a new program is executed. Although `exec` doesn't create a new accounting record, the command name changes, and the `AFORK` flag is cleared. This means that if we have a chain of three programs—A

ac_flag	Description	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
AFORK	process is the result of fork, but never called exec	•	•	•	•
ASU	process used superuser privileges		•	•	•
ACORE	process dumped core	•	•	•	
AXSIG	process was killed by a signal	•	•	•	
AEXPND	expanded accounting entry				•
ANVER	new record format	•			

Figure 8.26 Values for ac_flag from accounting record

execs B, then B execs C, and C exits—only a single accounting record is written. The command name in the record corresponds to program C, but the CPU times, for example, are the sum for programs A, B, and C.

Example

To have some accounting data to examine, we'll create a test program to implement the diagram shown in Figure 8.27.

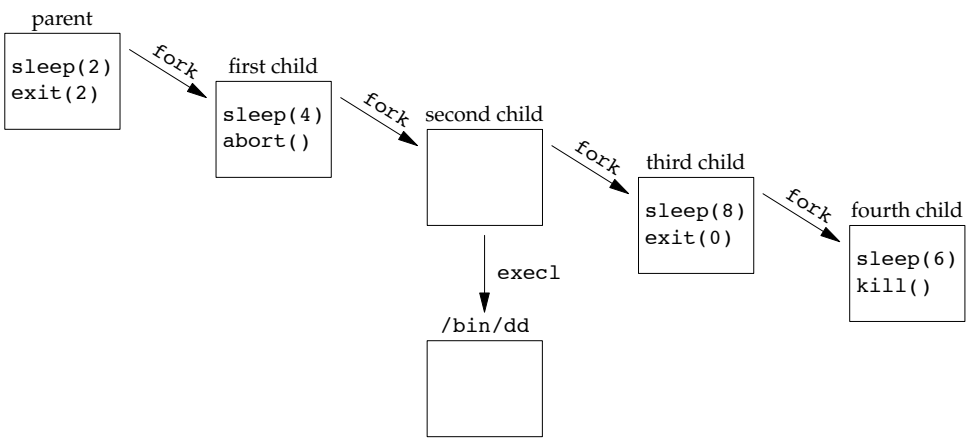

Figure 8.27 Process structure for accounting example

The source for the test program is shown in Figure 8.28. It calls fork four times. Each child does something different and then terminates.

```
#include "apue.h"

int
main(void)
{
    pid_t   pid;

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

```
    else if (pid != 0) {        /* parent */
        sleep(2);
        exit(2);                /* terminate with exit status 2 */
    }

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {        /* first child */
        sleep(4);
        abort();                /* terminate with core dump */
    }

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {        /* second child */
        execl("/bin/dd", "dd", "if=/etc/passwd", "of=/dev/null", NULL);
        exit(7);                /* shouldn't get here */
    }

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {        /* third child */
        sleep(8);
        exit(0);                /* normal exit */
    }

    sleep(6);                   /* fourth child */
    kill(getpid(), SIGKILL);    /* terminate w/signal, no core dump */
    exit(6);                    /* shouldn't get here */
}
```

Figure 8.28 Program to generate accounting data

We'll run the test program on Solaris and then use the program in Figure 8.29 to print out selected fields from the accounting records.

```
#include "apue.h"
#include <sys/acct.h>

#if defined(BSD)    /* different structure in FreeBSD */
#define acct acctv2
#define ac_flag ac_trailer.ac_flag
#define FMT "%-*.*s  e = %.0f, chars = %.0f, %c %c %c %c\n"
#elif defined(HAS_AC_STAT)
#define FMT "%-*.*s  e = %6ld, chars = %7ld, stat = %3u: %c %c %c %c\n"
#else
#define FMT "%-*.*s  e = %6ld, chars = %7ld, %c %c %c %c\n"
#endif
#if defined(LINUX)
#define acct acct_v3    /* different structure in Linux */
#endif

#if !defined(HAS_ACORE)
#define ACORE 0
```

```
#endif
#if !defined(HAS_AXSIG)
#define AXSIG 0
#endif

#if !defined(BSD)
static unsigned long
compt2ulong(comp_t comptime)    /* convert comp_t to unsigned long */
{
    unsigned long   val;
    int             exp;

    val = comptime & 0x1fff;    /* 13-bit fraction */
    exp = (comptime >> 13) & 7; /* 3-bit exponent (0-7) */
    while (exp-- > 0)
        val *= 8;
    return(val);
}
#endif

int
main(int argc, char *argv[])
{
    struct acct     acdata;
    FILE            *fp;

    if (argc != 2)
        err_quit("usage: pracct filename");
    if ((fp = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);
    while (fread(&acdata, sizeof(acdata), 1, fp) == 1) {
        printf(FMT, (int)sizeof(acdata.ac_comm),
            (int)sizeof(acdata.ac_comm), acdata.ac_comm,
#if defined(BSD)
            acdata.ac_etime, acdata.ac_io,
#else
            compt2ulong(acdata.ac_etime), compt2ulong(acdata.ac_io),
#endif
#if defined(HAS_AC_STAT)
            (unsigned char) acdata.ac_stat,
#endif
            acdata.ac_flag & ACORE ? 'D' : ' ',
            acdata.ac_flag & AXSIG ? 'X' : ' ',
            acdata.ac_flag & AFORK ? 'F' : ' ',
            acdata.ac_flag & ASU   ? 'S' : ' ');
    }
    if (ferror(fp))
        err_sys("read error");
    exit(0);
}
```

Figure 8.29 Print selected fields from system's accounting file

BSD-derived platforms don't support the `ac_stat` member, so we define the `HAS_AC_STAT` constant on the platforms that do support this member. Basing the defined symbol on the feature instead of on the platform makes the code read better and allows us to modify the program simply by adding the new definition to our compilation command. The alternative would be to use

```
#if !defined(BSD) && !defined(MACOS)
```

which becomes unwieldy as we port our application to additional platforms.

We define similar constants to determine whether the platform supports the `ACORE` and `AXSIG` accounting flags. We can't use the flag symbols themselves, because on Linux, they are defined as `enum` values, which we can't use in a `#ifdef` expression.

To perform our test, we do the following:

1. Become superuser and enable accounting, with the `accton` command. Note that when this command terminates, accounting should be on; therefore, the first record in the accounting file should be from this command.
2. Exit the superuser shell and run the program in Figure 8.28. This should append six records to the accounting file: one for the superuser shell, one for the test parent, and one for each of the four test children.

   A new process is not created by the `execl` in the second child. There is only a single accounting record for the second child.
3. Become superuser and turn accounting off. Since accounting is off when this `accton` command terminates, it should not appear in the accounting file.
4. Run the program in Figure 8.29 to print the selected fields from the accounting file.

The output from step 4 follows. We have appended the description of the process in italics to selected lines, for the discussion later.

```
accton     e =      1, chars =     336, stat =   0:       S
sh         e =   1550, chars =   20168, stat =   0:       S
dd         e =      2, chars =    1585, stat =   0:           second child
a.out      e =    202, chars =       0, stat =   0:           parent
a.out      e =    420, chars =       0, stat = 134:     F     first child
a.out      e =    600, chars =       0, stat =   9:     F     fourth child
a.out      e =    801, chars =       0, stat =   0:     F     third child
```

For this system, the elapsed time values are measured in units of clock ticks. Figure 2.15 shows that this system generates 100 clock ticks per second. For example, the `sleep(2)` in the parent corresponds to the elapsed time of 202 clock ticks. For the first child, the `sleep(4)` becomes 420 clock ticks. Note that the amount of time a process sleeps is not exact. (We'll return to the `sleep` function in Chapter 10.) Also, the calls to `fork` and `exit` take some amount of time.

Note that the `ac_stat` member is not the true termination status of the process, but rather corresponds to a portion of the termination status that we discussed in

Section 8.6. The only information in this byte is a core-flag bit (usually the high-order bit) and the signal number (usually the seven low-order bits), if the process terminated abnormally. If the process terminated normally, we are not able to obtain the `exit` status from the accounting file. For the first child, this value is 128 + 6. The 128 is the core flag bit, and 6 happens to be the value on this system for `SIGABRT`, which is generated by the call to `abort`. The value 9 for the fourth child corresponds to the value of `SIGKILL`. We can't tell from the accounting data that the parent's argument to `exit` was 2 and that the third child's argument to `exit` was 0.

The size of the file `/etc/passwd` that the `dd` process copies in the second child is 777 bytes. The number of characters of I/O is just over twice this value. It is twice the value, as 777 bytes are read in, then 777 bytes are written out. Even though the output goes to the null device, the bytes are still accounted for. The 31 additional bytes come from the `dd` command reporting the summary of bytes read and written, which it prints to `stdout`.

The `ac_flag` values are are what we would expect. The `F` flag is set for all the child processes except the second child, which does the `execl`. The `F` flag is not set for the parent, because the interactive shell that executed the parent did a `fork` and then an `exec` of the `a.out` file. The first child process calls `abort`, which generates a `SIGABRT` signal to generate the core dump. Note that neither the `X` flag nor the `D` flag is on, as they are not supported on Solaris; the information they represent can be derived from the `ac_stat` field. The fourth child also terminates because of a signal, but the `SIGKILL` signal does not generate a core dump; it just terminates the process.

As a final note, the first child has a 0 count for the number of characters of I/O, yet this process generated a `core` file. It appears that the I/O required to write the `core` file is not charged to the process. □

8.15 User Identification

Any process can find out its real and effective user ID and group ID. Sometimes, however, we want to find out the login name of the user who's running the program. We could call `getpwuid(getuid())`, but what if a single user has multiple login names, each with the same user ID? (A person might have multiple entries in the password file with the same user ID to have a different login shell for each entry.) The system normally keeps track of the name we log in under (Section 6.8), and the `getlogin` function provides a way to fetch that login name.

```
#include <unistd.h>

char *getlogin(void);
```

Returns: pointer to string giving login name if OK, `NULL` on error

This function can fail if the process is not attached to a terminal that a user logged in to. We normally call these processes *daemons*. We discuss them in Chapter 13.

Given the login name, we can then use it to look up the user in the password file—to determine the login shell, for example—using `getpwnam`.

> To find the login name, UNIX systems have historically called the `ttyname` function (Section 18.9) and then tried to find a matching entry in the `utmp` file (Section 6.8). FreeBSD and Mac OS X store the login name in the session structure associated with the process table entry and provide system calls to fetch and store this name.
>
> System V provided the `cuserid` function to return the login name. This function called `getlogin` and, if that failed, did a `getpwuid(getuid())`. The IEEE Standard 1003.1-1988 specified `cuserid`, but it called for the effective user ID to be used, instead of the real user ID. The 1990 version of POSIX.1 dropped the `cuserid` function.
>
> The environment variable `LOGNAME` is usually initialized with the user's login name by `login`(1) and inherited by the login shell. Realize, however, that a user can modify an environment variable, so we shouldn't use `LOGNAME` to validate the user in any way. Instead, we should use `getlogin`.

8.16 Process Scheduling

Historically, the UNIX System provided processes with only coarse control over their scheduling priority. The scheduling policy and priority were determined by the kernel. A process could choose to run with lower priority by adjusting its *nice value* (thus a process could be "nice" and reduce its share of the CPU by adjusting its nice value). Only a privileged process was allowed to increase its scheduling priority.

The real-time extensions in POSIX added interfaces to select among multiple scheduling classes and fine-tune their behavior. We discuss only the interfaces used to adjust the nice value here; they are part of the XSI option in POSIX.1. Refer to Gallmeister [1995] for more information on the real-time scheduling extensions.

In the Single UNIX Specification, nice values range from `0` to `(2*NZERO)-1`, although some implementations support a range from `0` to `2*NZERO`. Lower nice values have higher scheduling priority. Although this might seem backward, it actually makes sense: the more nice you are, the lower your scheduling priority is. `NZERO` is the default nice value of the system.

> Be aware that the header file defining `NZERO` differs among systems. In addition to the header file, Linux 3.2.0 makes the value of `NZERO` accessible through a nonstandard `sysconf` argument (`_SC_NZERO`).

A process can retrieve and change its nice value with the `nice` function. With this function, a process can affect only its own nice value; it can't affect the nice value of any other process.

```
#include <unistd.h>

int nice(int incr);
```

Returns: new nice value – `NZERO` if OK, –1 on error

The *incr* argument is added to the nice value of the calling process. If *incr* is too large, the system silently reduces it to the maximum legal value. Similarly, if *incr* is too small, the system silently increases it to the minimum legal value. Because –1 is a legal successful return value, we need to clear `errno` before calling `nice` and check its value if `nice` returns –1. If the call to `nice` succeeds and the return value is –1, then `errno` will still be zero. If `errno` is nonzero, it means that the call to `nice` failed.

The `getpriority` function can be used to get the nice value for a process, just like the `nice` function. However, `getpriority` can also get the nice value for a group of related processes.

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

int getpriority(int which, id_t who);
```

Returns: nice value between –NZERO and NZERO–1 if OK, –1 on error

The *which* argument can take on one of three values: `PRIO_PROCESS` to indicate a process, `PRIO_PGRP` to indicate a process group, and `PRIO_USER` to indicate a user ID. The *which* argument controls how the *who* argument is interpreted and the *who* argument selects the process or processes of interest. If the *who* argument is 0, then it indicates the calling process, process group, or user (depending on the value of the *which* argument). When *which* is set to `PRIO_USER` and *who* is 0, the real user ID of the calling process is used. When the *which* argument applies to more than one process, the highest priority (lowest value) of all the applicable processes is returned.

The `setpriority` function can be used to set the priority of a process, a process group, or all the processes belonging to a particular user ID.

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

int setpriority(int which, id_t who, int value);
```

Returns: 0 if OK, –1 on error

The *which* and *who* arguments are the same as in the `getpriority` function. The *value* is added to `NZERO` and this becomes the new nice value.

> The `nice` system call originated with an early PDP-11 version of the Research UNIX System. The `getpriority` and `setpriority` functions originated with 4.2BSD.

The Single UNIX Specification leaves it up to the implementation whether the nice value is inherited by a child process after a `fork`. However, XSI-compliant systems are required to preserve the nice value across a call to `exec`.

> A child process inherits the nice value from its parent process in FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10.

Example

The program in Figure 8.30 measures the effect of adjusting the nice value of a process. Two processes run in parallel, each incrementing its own counter. The parent runs with the default nice value, and the child runs with an adjusted nice value as specified by the

optional command argument. After running for 10 seconds, both processes print the value of their counter and exit. By comparing the counter values for different nice values, we can get an idea how the nice value affects process scheduling.

```
#include "apue.h"
#include <errno.h>
#include <sys/time.h>

#if defined(MACOS)
#include <sys/syslimits.h>
#elif defined(SOLARIS)
#include <limits.h>
#elif defined(BSD)
#include <sys/param.h>
#endif

unsigned long long count;
struct timeval end;

void
checktime(char *str)
{
    struct timeval  tv;

    gettimeofday(&tv, NULL);
    if (tv.tv_sec >= end.tv_sec && tv.tv_usec >= end.tv_usec) {
        printf("%s count = %lld\n", str, count);
        exit(0);
    }
}

int
main(int argc, char *argv[])
{
    pid_t   pid;
    char    *s;
    int     nzero, ret;
    int     adj = 0;

    setbuf(stdout, NULL);
#if defined(NZERO)
    nzero = NZERO;
#elif defined(_SC_NZERO)
    nzero = sysconf(_SC_NZERO);
#else
#error NZERO undefined
#endif
    printf("NZERO = %d\n", nzero);
    if (argc == 2)
        adj = strtol(argv[1], NULL, 10);
    gettimeofday(&end, NULL);
    end.tv_sec += 10;   /* run for 10 seconds */

    if ((pid = fork()) < 0) {
```

```
        err_sys("fork failed");
    } else if (pid == 0) {  /* child */
        s = "child";
        printf("current nice value in child is %d, adjusting by %d\n",
          nice(0)+nzero, adj);
        errno = 0;
        if ((ret = nice(adj)) == -1 && errno != 0)
            err_sys("child set scheduling priority");
        printf("now child nice value is %d\n", ret+nzero);
    } else {        /* parent */
        s = "parent";
        printf("current nice value in parent is %d\n", nice(0)+nzero);
    }
    for(;;) {
        if (++count == 0)
            err_quit("%s counter wrap", s);
        checktime(s);
    }
}
```

Figure 8.30 Evaluate the effect of changing the nice value

We run the program twice: once with the default nice value, and once with the highest valid nice value (the lowest scheduling priority). We run this on a uniprocessor Linux system to show how the scheduler shares the CPU among processes with different nice values. With an otherwise idle system, a multiprocessor system (or a multicore CPU) would allow both processes to run without the need to share a CPU, and we wouldn't see much difference between two processes with different nice values.

```
$ ./a.out
NZERO = 20
current nice value in parent is 20
current nice value in child is 20, adjusting by 0
now child nice value is 20
child count = 1859362
parent count = 1845338
$ ./a.out 20
NZERO = 20
current nice value in parent is 20
current nice value in child is 20, adjusting by 20
now child nice value is 39
parent count = 3595709
child count = 52111
```

When both processes have the same nice value, the parent process gets 50.2% of the CPU and the child gets 49.8% of the CPU. Note that the two processes are effectively treated equally. The percentages aren't exactly equal, because process scheduling isn't exact, and because the child and parent perform different amounts of processing between the time that the end time is calculated and the time that the processing loop begins.

In contrast, when the child has the highest possible nice value (the lowest priority), we see that the parent gets 98.5% of the CPU, while the child gets only 1.5% of the CPU. These values will vary based on how the process scheduler uses the nice value, so a different UNIX system will produce different ratios. □

8.17 Process Times

In Section 1.10, we described three times that we can measure: wall clock time, user CPU time, and system CPU time. Any process can call the `times` function to obtain these values for itself and any terminated children.

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

clock_t times(struct tms *buf);
```

Returns: elapsed wall clock time in clock ticks if OK, –1 on error

This function fills in the `tms` structure pointed to by *buf*:

```
struct tms {
  clock_t  tms_utime;  /* user CPU time */
  clock_t  tms_stime;  /* system CPU time */
  clock_t  tms_cutime; /* user CPU time, terminated children */
  clock_t  tms_cstime; /* system CPU time, terminated children */
};
```

Note that the structure does not contain any measurement for the wall clock time. Instead, the function returns the wall clock time as the value of the function, each time it's called. This value is measured from some arbitrary point in the past, so we can't use its absolute value; instead, we use its relative value. For example, we call `times` and save the return value. At some later time, we call `times` again and subtract the earlier return value from the new return value. The difference is the wall clock time. (It is possible, though unlikely, for a long-running process to overflow the wall clock time; see Exercise 1.5.)

The two structure fields for child processes contain values only for children that we have waited for with one of the `wait` functions discussed earlier in this chapter.

All the `clock_t` values returned by this function are converted to seconds using the number of clock ticks per second—the `_SC_CLK_TCK` value returned by `sysconf` (Section 2.5.4).

> Most implementations provide the `getrusage`(2) function. This function returns the CPU times and 14 other values indicating resource usage. Historically, this function originated with the BSD operating system, so BSD-derived implementations generally support more of the fields than do other implementations.

Example

The program in Figure 8.31 executes each command-line argument as a shell command string, timing the command and printing the values from the `tms` structure.

```
#include "apue.h"
#include <sys/times.h>

static void pr_times(clock_t, struct tms *, struct tms *);
static void do_cmd(char *);

int
main(int argc, char *argv[])
{
    int     i;

    setbuf(stdout, NULL);
    for (i = 1; i < argc; i++)
        do_cmd(argv[i]);    /* once for each command-line arg */
    exit(0);
}

static void
do_cmd(char *cmd)       /* execute and time the "cmd" */
{
    struct tms  tmsstart, tmsend;
    clock_t     start, end;
    int         status;

    printf("\ncommand: %s\n", cmd);

    if ((start = times(&tmsstart)) == -1)   /* starting values */
        err_sys("times error");

    if ((status = system(cmd)) < 0)         /* execute command */
        err_sys("system() error");

    if ((end = times(&tmsend)) == -1)       /* ending values */
        err_sys("times error");

    pr_times(end-start, &tmsstart, &tmsend);
    pr_exit(status);
}

static void
pr_times(clock_t real, struct tms *tmsstart, struct tms *tmsend)
{
    static long     clktck = 0;

    if (clktck == 0)    /* fetch clock ticks per second first time */
        if ((clktck = sysconf(_SC_CLK_TCK)) < 0)
            err_sys("sysconf error");

    printf("  real:  %7.2f\n", real / (double) clktck);
    printf("  user:  %7.2f\n",
      (tmsend->tms_utime - tmsstart->tms_utime) / (double) clktck);
    printf("  sys:   %7.2f\n",
      (tmsend->tms_stime - tmsstart->tms_stime) / (double) clktck);
    printf("  child user:  %7.2f\n",
```

```
      (tmsend->tms_cutime - tmsstart->tms_cutime) / (double) clktck);
    printf("  child sys:   %7.2f\n",
      (tmsend->tms_cstime - tmsstart->tms_cstime) / (double) clktck);
}
```

Figure 8.31 Time and execute all command-line arguments

If we run this program, we get

```
$ ./a.out "sleep 5" "date" "man bash >/dev/null"

command: sleep 5
  real:     5.01
  user:     0.00
  sys:      0.00
  child user:     0.00
  child sys:      0.00
normal termination, exit status = 0

command: date
Sun Feb 26 18:39:23 EST 2012
  real:     0.00
  user:     0.00
  sys:      0.00
  child user:     0.00
  child sys:      0.00
normal termination, exit status = 0

command: man bash >/dev/null
  real:     1.46
  user:     0.00
  sys:      0.00
  child user:     1.32
  child sys:      0.07
normal termination, exit status = 0
```

In the first two commands, execution is fast enough to avoid registering any CPU time at the reported resolution. In the third command, however, we run a command that takes enough processing time to note that all the CPU time appears in the child process, which is where the shell and the command execute. □

8.18 Summary

A thorough understanding of the UNIX System's process control is essential for advanced programming. There are only a few functions to master: `fork`, the `exec` family, `_exit`, `wait`, and `waitpid`. These primitives are used in many applications. The `fork` function also gave us an opportunity to look at race conditions.

Our examination of the `system` function and process accounting gave us another look at all these process control functions. We also looked at another variation of the

exec functions: interpreter files and how they operate. An understanding of the various user IDs and group IDs that are provided—real, effective, and saved—is critical to writing safe set-user-ID programs.

Given an understanding of a single process and its children, in the next chapter we examine the relationship of a process to other processes—sessions and job control. We then complete our discussion of processes in Chapter 10 when we describe signals.

Exercises

8.1 In Figure 8.3, we said that replacing the call to _exit with a call to exit might cause the standard output to be closed and printf to return –1. Modify the program to check whether your implementation behaves this way. If it does not, how can you simulate this behavior?

8.2 Recall the typical arrangement of memory in Figure 7.6. Because the stack frames corresponding to each function call are usually stored in the stack, and because after a vfork the child runs in the address space of the parent, what happens if the call to vfork is from a function other than main and the child does a return from this function after the vfork? Write a test program to verify this, and draw a picture of what's happening.

8.3 Rewrite the program in Figure 8.6 to use waitid instead of wait. Instead of calling pr_exit, determine the equivalent information from the siginfo structure.

8.4 When we execute the program in Figure 8.13 one time, as in

```
$ ./a.out
```

the output is correct. But if we execute the program multiple times, one right after the other, as in

```
$ ./a.out ; ./a.out ; ./a.out
output from parent
ooutput from parent
ouotuptut from child
put from parent
output from child
utput from child
```

the output is not correct. What's happening? How can we correct this? Can this problem happen if we let the child write its output first?

8.5 In the program shown in Figure 8.20, we call execl, specifying the *pathname* of the interpreter file. If we called execlp instead, specifying a *filename* of testinterp, and if the directory /home/sar/bin was a path prefix, what would be printed as argv[2] when the program is run?

8.6 Write a program that creates a zombie, and then call system to execute the ps(1) command to verify that the process is a zombie.

8.7 We mentioned in Section 8.10 that POSIX.1 requires open directory streams to be closed across an exec. Verify this as follows: call opendir for the root directory, peek at your system's implementation of the DIR structure, and print the close-on-exec flag. Then open the same directory for reading, and print the close-on-exec flag.