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.

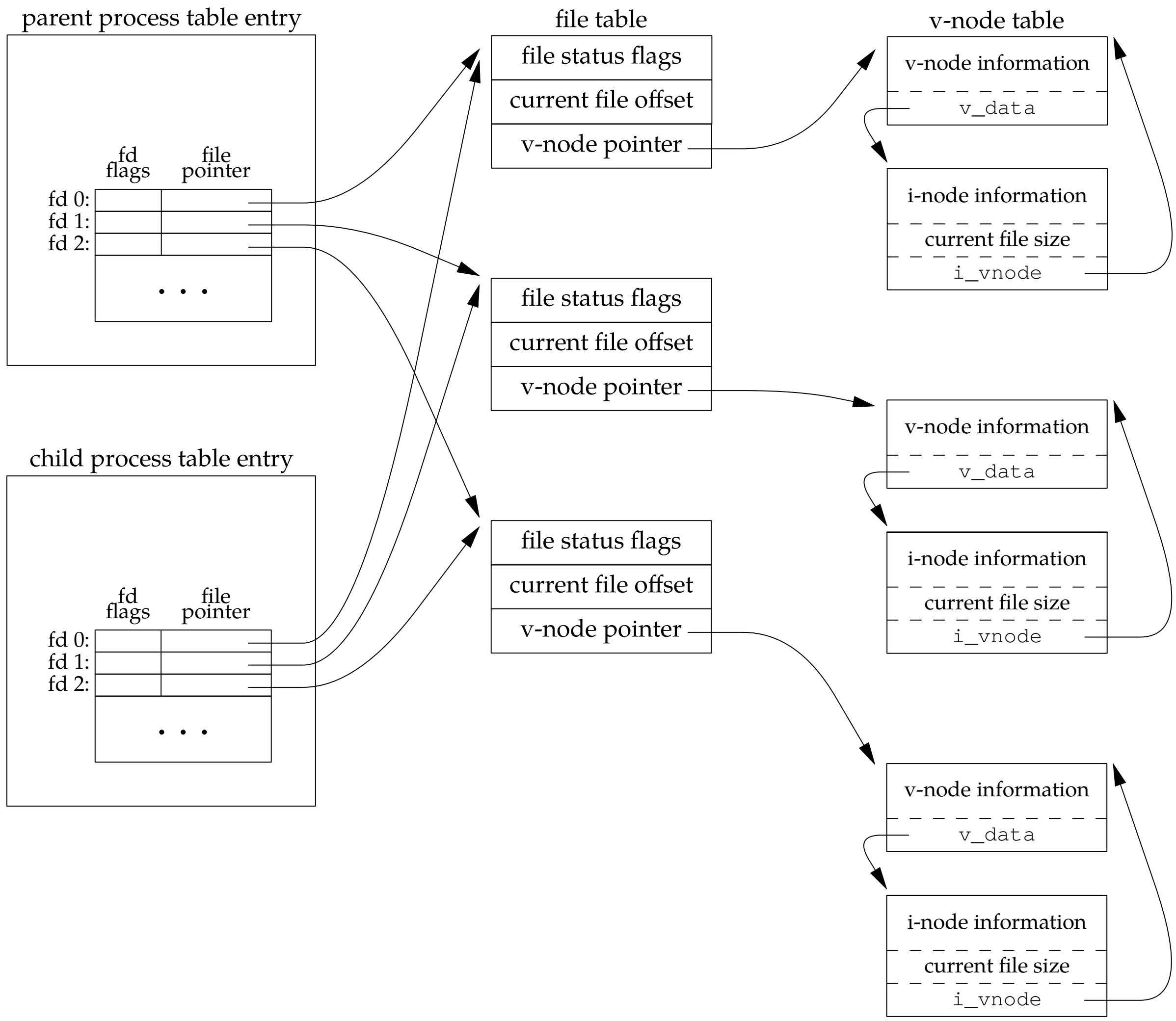

Figure 8.2 Sharing of open files between parent and child after `fork`

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`.

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*)	Tr ue 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*)	Tr ue 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*)	Tr ue 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*)	Tr ue 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 `SIGPFE` 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. □

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.

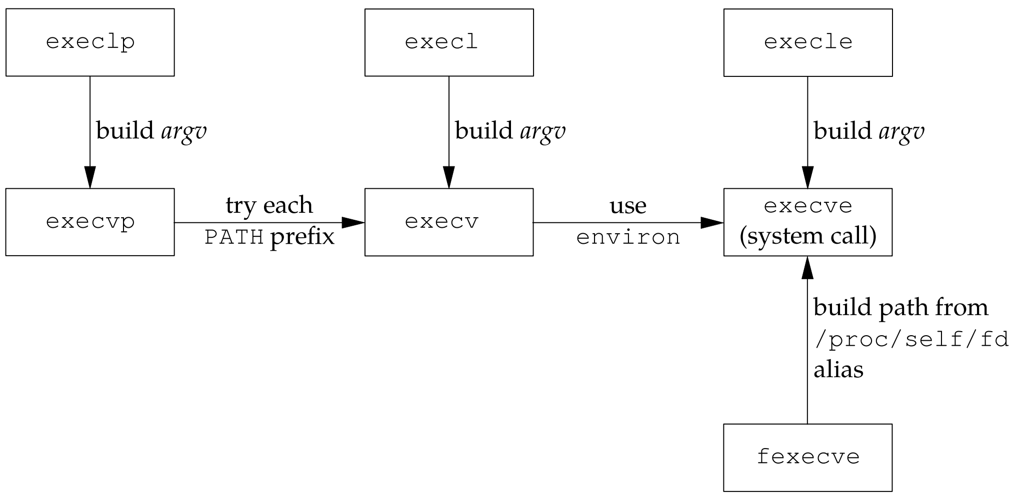

Figure 8.15 Relationship of the seven `exec` functions

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.