NAVODAYA INSTITUTE OF TECHNOLOGY, RAICHUR

DEPARMENT OF COMPUTER SCIENCE & ENGINEERING

Module - 4

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 forks a child, then waits 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 waiting 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.

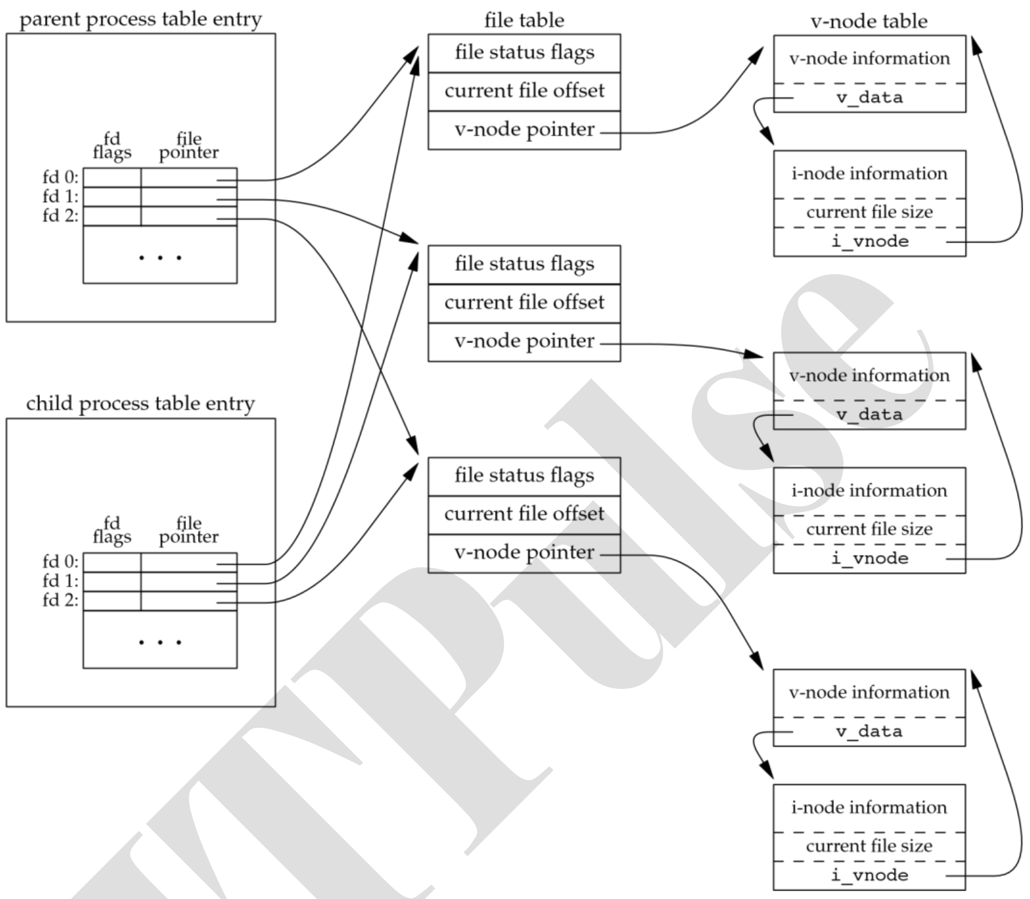


Figure 8.2 Sharing of open files between parent and child after fork

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 needto 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;   pid_t   /* automatic variable on the stack */
    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 inChapter 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, cancellationoccurs 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 forks 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*)	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 `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); if ((pid       /* and print its status */
    = 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); if ((pid       /* and print its status */
    = 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 forks 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 setgroup-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.

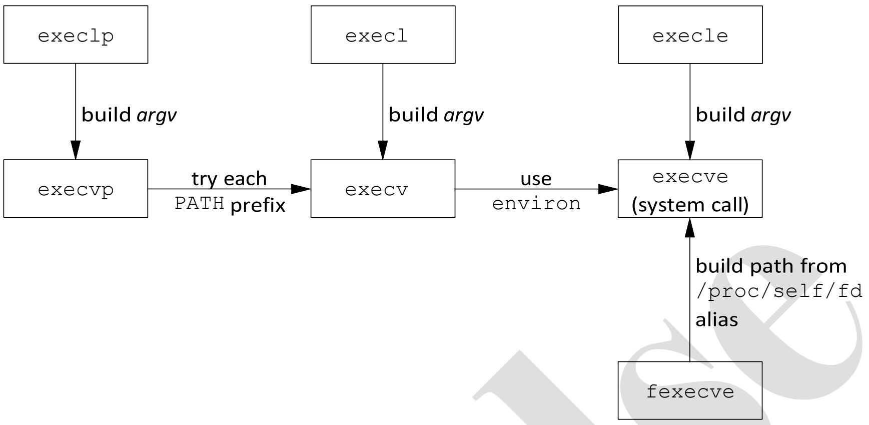


Figure 8.15 Relationship of the seven `exec` functions

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.

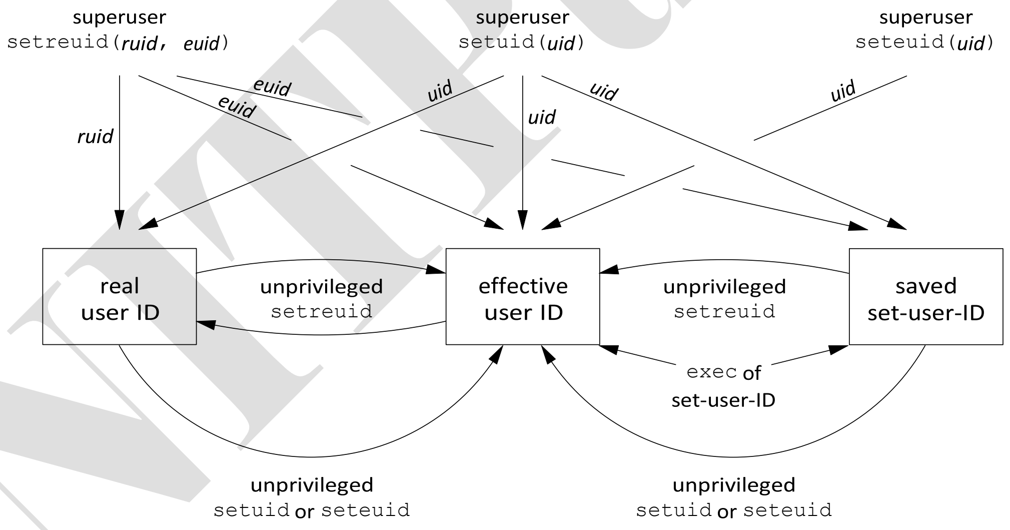


Figure 8.19 Summary of all the functions that set the various user IDs

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 atwhich 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 `exec`s 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 `exec`s 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 `exec`s 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 exampleagain. 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 tsysverify 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.

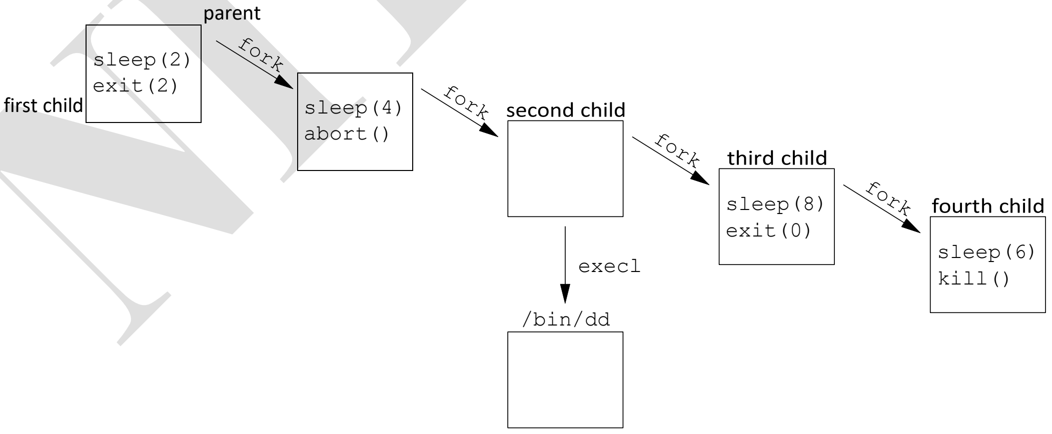


Figure 8.27 Process structure for accounting example

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 shouldappend 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 accountingfile.

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

15

Inter process Communica tion

15.1 Introduction

In Chapter 8, we described the process control primitives and saw how to work with multiple processes. But the only way for these processes to exchange information is by passing open files across a `fork` or an `exec` or through the file system. We'll now describe other techniques for processes to communicate with one another: interprocess communication (IPC).

In the past, UNIX System IPC was a hodgepodge of various approaches, few of which were portable across all UNIX system implementations. Through the POSIX and The Open Group (formerly X/Open) standardization efforts, the situation has since improved, but differences still exist. Figure 15.1 summarizes the various forms of IPC that are supported by the four implementations discussed in this text.

Note that the Single UNIX Specification (the ''SUS'' column) allows an implementation to support full-duplex pipes, but requires only half-duplex pipes. An implementation that supports full-duplex pipes will still work with correctly written applications that assume that the underlying operating system supports only half-duplex pipes. We use ''(full)'' instead of a bullet to show implementations that support half-duplex pipes by using full-duplex pipes.

In Figure 15.1, we show a bullet where basic functionality is supported. For full-duplex pipes, if the feature can be provided through UNIX domain sockets (Section 17.2), we show ''UDS'' in the column. Some implementations support the feature with pipes and UNIX domain sockets, so these entries have both ''UDS'' and a bullet.

The IPC interfaces introduced as part of the real-time extensions to POSIX.1 were included as options in the Single UNIX Specification. In SUSv4, the semaphore interfaces were moved from an option to the base specification.

IPC type	SUS	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
half-duplex pipes	•	(full)	•	•	(full)
FIFOs	•	•	•	•	•
full-duplex pipes	allowed	•, UDS	UDS	UDS	•, UDS
named full-duplex pipes	obsolescent	UDS	UDS	UDS	•, UDS
XSI message queues	XSI	•	•	•	•
XSI semaphores	XSI	•	•	•	•
XSI shared memory	XSI	•	•	•	•
message queues (real-time)	MSG option	•	•		•
semaphores	•	•	•	•	•
shared memory (real-time)	SHM option	•	•	•	•
sockets	•	•	•	•	•
STREAMS	obsolescent				•

Figure 15.1 Summary of UNIX System IPC

Named full-duplex pipes are provided as mounted STREAMS-based pipes, but are marked obsolescent in the Single UNIX Specification.

Although support for STREAMS on Linux is available in the ‘‘Linux Fast-STREAMS’’ package from the OpenSS7 project, the package hasn’t been updated recently. The latest release of the package from 2008 claims to work with kernels up to Linux 2.6.26.

The first ten forms of IPC in Figure 15.1 are usually restricted to IPC between processes on the same host. The final two rows — sockets and STREAMS—are the only two forms that are generally supported for IPC between processes on different hosts.

We have divided the discussion of IPC into three chapters. In this chapter, we examine classical IPC: pipes, FIFOs, message queues, semaphores, and shared memory. In the next chapter, we take a look at network IPC using the sockets mechanism. In Chapter 17, we take a look at some advanced features of IPC.

15.2 Pipes

Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. Pipes have two limitations.

1. Historically, they have been half duplex (i.e., data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.
2. Pipes can be used only between processes that have a common ancestor.Normally, a pipe is created by a process, that process calls `fork`, and the pipe is used between the parent and the child.

We'll see that FIFOs (Section 15.5) get around the second limitation, and that UNIX domain sockets (Section 17.2) get around both limitations.

Despite these limitations, half-duplex pipes are still the most commonly used form of IPC. Every time you type a sequence of commands in a pipeline for the shell to execute, the shell creates a separate process for each command and links the standard output of one process to the standard input of the next using a pipe. A pipe is created by calling the `pipe` function.

```
#include <unistd.h>

int pipe(int fd[2]);

                                        Returns: 0 if OK, −1 on error
```

Two file descriptors are returned through the *fd* argument: *fd[0]* is open for reading, and *fd[1]* is open for writing. The output of *fd[1]* is the input for *fd[0]*.

Originally in 4.3BSD and 4.4BSD, pipes were implemented using UNIX domain sockets. Even though UNIX domain sockets are full duplex by default, these operating systems hobbled the sockets used with pipes so that they operated in half-duplex mode only.

POSIX.1 allows for implementations to support full-duplex pipes. For these implementations, *fd[0]* and *fd[1]* are open for both reading and writing.

Two ways to picture a half-duplex pipe are shown in Figure 15.2. The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.

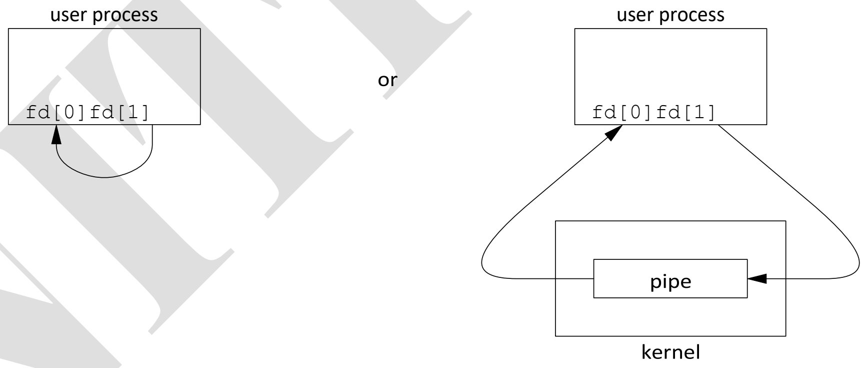


Figure 15.2 Two ways to view a half-duplex pipe

The `fstat` function (Section 4.2) returns a file type of FIFO for the file descriptor of either end of a pipe. We can test for a pipe with the `S_ISFIFO` macro.

POSIX.1 states that the `st_size` member of the `stat` structure is undefined for pipes. But when the `fstat` function is applied to the file descriptor for the read end of the pipe, many systems store in `st_size` the number of bytes available for reading in the pipe. This is, however, nonportable.

A pipe in a single process is next to useless. Normally, the process that calls `pipe` then calls `fork`, creating an IPC channel from the parent to the child, or vice versa.

Figure 15.3 shows this scenario.

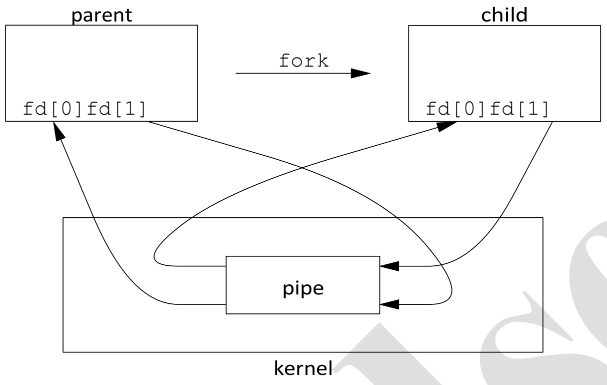


Figure 15.3 Half-duplex pipe after a `fork`

What happens after the `fork` depends on which direction of data flow we want. For a pipe from the parent to the child, the parent closes the read end of the pipe (`fd[0]`), and the child closes the write end (`fd[1]`). Figure 15.4 shows the resulting arrangement of descriptors.

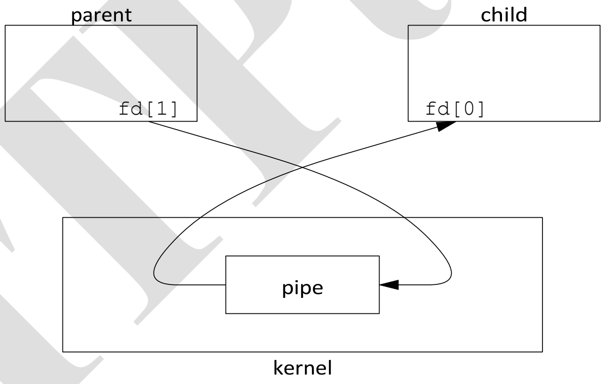


Figure 15.4 Pipe from parent to child

For a pipe from the child to the parent, the parent closes `fd[1]`, and the child closes `fd[0]`.

When one end of a pipe is closed, two rules apply.

1. If we `read` from a pipe whose write end has been closed, `read` returns 0 to indicate an end of file after all the data has been read. (Technically, we should say that this end of file is not generated until there are no more writers for the pipe. It's possible to duplicate a pipe descriptor so that multiple processes have the pipe open for writing. Normally, however, there is a single reader and a single writer for a pipe. When we get to FIFOs in the next section, we'll see that often there are multiple writers for a single FIFO.)

2. If we write to a pipe whose read end has been closed, the signal SIGPIPE is generated. If we either ignore the signal or catch it and return from the signal handler, write returns −1 with errno set to EPIPE.

When we're writing to a pipe (or FIFO), the constant PIPE_BUF specifies the kernel's pipe buffer size. A write of PIPE_BUF bytes or less will not be interleaved with the writes from other processes to the same pipe (or FIFO). But if multiple processes are writing to a pipe (or FIFO), and if we write more than PIPE_BUF bytes, the data might be interleaved with the data from the other writers. We can determine the value of PIPE_BUF by using pathconf or fpathconf (recall Figure 2.12).

Example

Figure 15.5 shows the code to create a pipe between a parent and its child and to send data down the pipe.

```
#include "apue.h"
int main(void)
{
    int       n; int
       fd[2]; pid_t pid; char
    line[MAXLINE]; if
    (pipe(fd) < 0)
    err_sys("pipe error"); if
    ((pid = fork()) < 0) {
    err_sys("fork error");
    } else if (pid > 0) {  /* parent */ close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {  /* child */ close(fd[1]); n =
        read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

Figure 15.5 Send data from parent to child over a pipe

Note that the pipe direction here matches the orientation shown in Figure 15.4. □

In the previous example, we called read and write directly on the pipe descriptors. What is more interesting is to duplicate the pipe descriptors onto standard input or standard output. Often, the child then runs some other program, and that program can either read from its standard input (the pipe that we created) or write to its standard output (the pipe).

Example

Consider a program that displays some output that it has created, one page at a time. Rather than reinvent the pagination done by several UNIX system utilities, we want to invoke the user's favorite pager. To avoid writing all the data to a temporary file and calling `system` to display that file, we want to pipe the output directly to the pager. To do this, we create a pipe, `fork` a child process, set up the child's standard input to be the read end of the pipe, and `exec` the user's pager program. Figure 15.6 shows how to do this. (This example takes a command-line argument to specify the name of a file to display. Often, a program of this type would already have the data to display to the terminal in memory.)

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

#define DEF_PAGER "/bin/more"     /* default pager program */ int

main(int argc, char *argv[])
{
    int n; int fd[2]; pid_t
    pid;    char    *pager,
    *argv0;             char
    line[MAXLINE];      FILE
    *fp;

    if (argc != 2) err_quit("usage: a.out
    <pathname>"); if ((fp = fopen(argv[1],
    "r")) == NULL) err_sys("can't open %s",
    argv[1]); if (pipe(fd) < 0)
    err_sys("pipe error"); if ((pid =
    fork()) < 0) { err_sys("fork error");
    } else if (pid > 0) {  /* parent */ close(fd[0]);        /* close
        read end */

        /* parent copies argv[1] to pipe */ while
        (fgets(line, MAXLINE, fp) != NULL) { n =
        strlen(line);
            if (write(fd[1], line, n) != n)
                err_sys("write error to pipe");
        }
        if (ferror(fp))
            err_sys("fgets error");

        close(fd[1]); /* close write end of pipe for reader */

        if (waitpid(pid, NULL, 0) < 0) err_sys("waitpid

        error");

        exit(0);
    } else {  /* child */ close(fd[1]); /* close write end */
        if (fd[0] != STDIN_FILENO) { if (dup2(fd[0],
        STDIN_FILENO) != STDIN_FILENO) err_sys("dup2 error to
        stdin");
            close(fd[0]); /* don’t need this after dup2 */
        }
        /* get arguments for execl() */ if
        ((pager = getenv("PAGER")) == NULL)
        pager = DEF_PAGER;
        if ((argv0 = strrchr(pager, ’/’)) != NULL)
            argv0++;        /* step past rightmost slash */
        else argv0 = pager; /* no slash in pager
        */ if (execl(pager, argv0, (char *)0) < 0)
        err_sys("execl error for %s", pager);
    }
    exit(0);
}
```

Figure 15.6 Copy file to pager program

Before calling fork, we create a pipe. After the fork, the parent closes its read end, and the child closes its write end. The child then calls dup2 to have its standard input be the read end of the pipe. When the pager program is executed, its standard input will be the read end of the pipe.

When we duplicate one descriptor onto another (fd[0] onto standard input in the child), we have to be careful that the descriptor doesn’t already have the desired value. If the descriptor already had the desired value and we called dup2 and close, the single copy of the descriptor would be closed. (Recall the operation of dup2 when its two arguments are equal, discussed in Section 3.12.) In this program, if standard input had not been opened by the shell, the fopen at the beginning of the program should have used descriptor 0, the lowest unused descriptor, so fd[0] should never equal standard input. Nevertheless, whenever we call dup2 and close to duplicate one descriptor onto another, we’ll always compare the descriptors first, as a defensive programming measure.

Note how we try to use the environment variable PAGER to obtain the name of the user ’s pager program. If this doesn’t work, we use a default. This is a common usage of environment variables. □

Example

Recall the five functions TELL_WAIT, TELL_PARENT, TELL_CHILD, WAIT_PARENT, and WAIT_CHILD from Section 8.9. In Figure 10.24, we showed an implementation using signals. Figure 15.7 shows an implementation using pipes.

```
#include "apue.h" static int

pfd1[2], pfd2[2];

void
TELL_WAIT(void)
{ if (pipe(pfd1) < 0 || pipe(pfd2) < 0)
    err_sys("pipe error");
}

void
TELL_PARENT(pid_t pid)
{ if (write(pfd2[1], "c", 1) != 1)
    err_sys("write error");
}

void
WAIT_PARENT(void)
{
    char c;

    if (read(pfd1[0], &c, 1) != 1)
        err_sys("read error");

    if (c != 'p') err_quit("WAIT_PARENT:
        incorrect data");
}

void
TELL_CHILD(pid_t pid)
{ if (write(pfd1[1], "p", 1) != 1)
    err_sys("write error");
}

void
WAIT_CHILD(void)
{
    char c;

    if (read(pfd2[0], &c, 1) != 1)
        err_sys("read error");

    if (c != 'c') err_quit("WAIT_CHILD:
        incorrect data");
}
```

Figure 15.7 Routines to let a parent and child synchronize

We create two pipes before the fork, as shown in Figure 15.8. The parent writes the character ‘‘p’’ across the top pipe when TELL_CHILD is called, and the child writes the character ‘‘c’’ across the bottom pipe when TELL_PARENT is called. The corresponding WAIT_xxx functions do a blocking read for the single character.

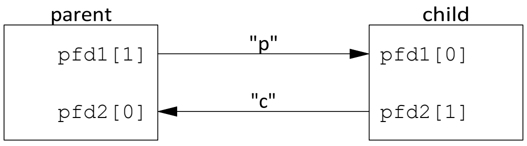


Figure 15.8 Using two pipes for parent–child synchronization

Note that each pipe has an extra reader, which doesn’t matter. That is, in addition to the child reading from pfd1[0], the parent has this end of the top pipe open for reading. This doesn’t affect us, since the parent doesn’t try to read from this pipe. □

15.3 popen and pclose Functions

Since a common operation is to create a pipe to another process to either read its output or send it input, the standard I/O library has historically provided the popen and pclose functions. These two functions handle all the dirty work that we’ve been doing ourselves: creating a pipe, forking a child, closing the unused ends of the pipe, executing a shell to run the command, and waiting for the command to terminate.

```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);

                                   Returns: file pointer if OK, NULL on error

int pclose(FILE *fp);

                     Returns: termination status of cmdstring, or −1 on error
```

The function popen does a fork and exec to execute the *cmdstring* and returns a standard I/O file pointer. If *type* is "r", the file pointer is connected to the standard output of *cmdstring* (Figure 15.9).

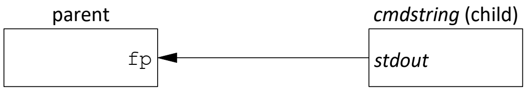


Figure 15.9 Result of fp = popen(*cmdstring*, "r")

If *type* is `"w"`, the file pointer is connected to the standard input of *cmdstring*, as shown in Figure 15.10.

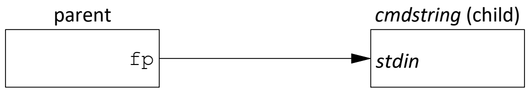


Figure 15.10 Result of `fp = popen(`*cmdstring*`, "w")`

One way to remember the final argument to `popen` is to remember that, like `fopen`, the returned file pointer is readable if *type* is `"r"` or writable if *type* is `"w"`.

The `pclose` function closes the standard I/O stream, waits for the command to terminate, and returns the termination status of the shell. (We described the termination status in Section 8.6. The `system` function, described in Section 8.13, also returns the termination status.) If the shell cannot be executed, the termination status returned by `pclose` is as if the shell had executed `exit(127)`. The *cmdstring* is executed by the Bourne shell, as in `sh -c` *cmdstring*

This means that the shell expands any of its special characters in *cmdstring*. This allows us to say, for example, `fp = popen("ls *.c", "r");`

or `fp = popen("cmd 2>&1", "r");`

Example

Let’s redo the program from Figure 15.6, using `popen`. This is shown in Figure 15.11.

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

#define PAGER "${PAGER:-more}" /* environment variable, or default */

int
main(int argc, char *argv[])
{ char line[MAXLINE]; FILE
    *fpin, *fpout;

    if (argc != 2) err_quit("usage: a.out
    <pathname>"); if ((fpin = fopen(argv[1],
    "r")) == NULL) err_sys("can’t open %s",
    argv[1]);

    if ((fpout = popen(PAGER, "w")) == NULL)
        err_sys("popen error");

    /* copy argv[1] to pager */
    while (fgets(line, MAXLINE, fpin) != NULL) {
        if (fputs(line, fpout) == EOF)
        err_sys("fputs error to pipe");
    }
    if (ferror(fpin))
    err_sys("fgets error"); if
    (pclose(fpout) == -1)
    err_sys("pclose error");

    exit(0);
}
```

Figure 15.11 Copy file to pager program using popen

Using popen reduces the amount of code we have to write.

The shell command ${PAGER:−more} says to use the value of the shell variable PAGER if it is defined and non-null; otherwise, use the string more.□

Example —popen and pclose Functions

Figure 15.12 shows our version of popen and pclose.

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

/*
 * Pointer to array allocated at run-time.
 */ static pid_t     *childpid =
NULL;

/*
 * From our open_max(), Figure 2.17.
 */
static int      maxfd;

FILE *
popen(const char *cmdstring, const char *type)
{
    int     i;
    int     pfd[2];
    pid_t pid; FILE
    *fp;
    /* only allow "r" or "w" */
    if ((type[0] != 'r' && type[0] != 'w') || type[1] != 0)
        { errno = EINVAL; return(NULL);
    }
    if (childpid == NULL) {       /* first time through */
        /* allocate zeroed out array for child pids */
        maxfd = open_max();
        if ((childpid = calloc(maxfd, sizeof(pid_t))) == NULL)
            return(NULL);
    }

    if (pipe(pfd) < 0) return(NULL); /* errno
    set by pipe() */ if (pfd[0] >= maxfd ||
    pfd[1] >= maxfd) { close(pfd[0]);
    close(pfd[1]); errno = EMFILE; return(NULL);
    }

    if ((pid = fork()) < 0) { return(NULL); /*
        errno set by fork() */
    } else if (pid == 0) { /* child */ if (*type == 'r') {
        close(pfd[0]); if (pfd[1] != STDOUT_FILENO) {
        dup2(pfd[1], STDOUT_FILENO);
                close(pfd[1]);
            }
        } else { close(pfd[1]); if (pfd[0]
            != STDIN_FILENO) {
            dup2(pfd[0], STDIN_FILENO);
                close(pfd[0]);
            }
        }

        /* close all descriptors in childpid[] */
        for (i = 0; i < maxfd; i++)
            if (childpid[i] > 0)
            close(i);

        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);
    }

    /* parent continues... */
    if (*type == 'r') {
    close(pfd[1]);
        if ((fp = fdopen(pfd[0], type)) == NULL)
            return(NULL);
    } else {
        close(pfd[0]);
        if ((fp = fdopen(pfd[1], type)) == NULL)
            return(NULL);
    }

    childpid[fileno(fp)] = pid; /* remember child pid for this fd */
    return(fp);
}

int
pclose(FILE *fp)
{
    int      fd, stat;
    pid_t pid;

    if (childpid == NULL) {
        errno = EINVAL;
        return(-1);      /* popen() has never been called */
    }

    fd = fileno(fp);
    if (fd >= maxfd) {
    errno = EINVAL;
        return(-1);      /* invalid file descriptor */
    }
    if ((pid = childpid[fd]) == 0) {
        errno = EINVAL;
        return(-1);      /* fp wasn't opened by popen() */
    }

    childpid[fd] = 0; if
    (fclose(fp) == EOF)
    return(-1);

    while (waitpid(pid, &stat, 0) < 0) if (errno != EINTR)
        return(-1); /* error other than EINTR from waitpid() */

    return(stat); /* return child's termination status */
}
```

Figure 15.12 The popen and pclose functions

Although the core of popen is similar to the code we've used earlier in this chapter, there are many details that we need to take care of. First, each time popen is called, we have to remember the process ID of the child that we create and either its file descriptor or FILE pointer. We choose to save the child's process ID in the array childpid, which we index by the file descriptor. This way, when pclose is called with the FILE pointer as its argument, we call the standard I/O function fileno to get the file descriptor and then have the child process ID for the call to waitpid. Since it's possible for a given process to call popen more than once, we dynamically allocate the childpid array (the first time popen is called), with room for as many children as there are file descriptors.

Note that our open_max function from Figure 2.17 can return a guess of the maximum number of open files if this value is indeterminate for the system. We need to be careful not to use a pipe file descriptor whose value is larger than (or equal to) what the open_max function returns. In popen, if the value returned by open_max happens to be too small, we close the pipe file descriptors, set errno to EMFILE to indicate too many file descriptors are open, and return −1. In pclose, if the file descriptor corresponding to the file pointer argument is larger than expected, we set errno to EINVAL and return −1.

Calling pipe and fork and then duplicating the appropriate descriptors for each process in the popen function is similar to what we did earlier in this chapter.

POSIX.1 requires that popen close any streams that are still open in the child from previous calls to popen. To do this, we go through the childpid array in the child, closing any descriptors that are still open.

What happens if the caller of pclose has established a signal handler for SIGCHLD? The call to waitpid from pclose would return an error of EINTR. Since the caller is allowed to catch this signal (or any other signal that might interrupt the call to waitpid), we simply call waitpid again if it is interrupted by a caught signal.

Note that if the application calls waitpid and obtains the exit status of the child created by popen, we will call waitpid when the application calls pclose, find that the child no longer exists, and return −1 with errno set to ECHILD. This is the behavior required by POSIX.1 in this situation.

Some early versions of pclose returned an error of EINTR if a signal interrupted the wait. Also, some early versions of pclose blocked or ignored the signals SIGINT, SIGQUIT, and SIGHUP during the wait. This is not allowed by POSIX.1.

□

Note that popen should never be called by a set-user-ID or set-group-ID program. When it executes the command, popen does the equivalent of execl("/bin/sh", "sh", "-c", *command*, NULL);

which executes the shell and *command* with the environment inherited by the caller. A malicious user can manipulate the environment so that the shell executes commands other than those intended, with the elevated permissions granted by the set-ID file mode.

One thing that popen is especially well suited for is executing simple filters to transform the input or output of the running command. Such is the case when a command wants to build its own pipeline.

Example

Consider an application that writes a prompt to standard output and reads a line from standard input. With the `popen` function, we can interpose a program between the application and its input to transform the input. Figure 15.13 shows the arrangement of processes in this situation.

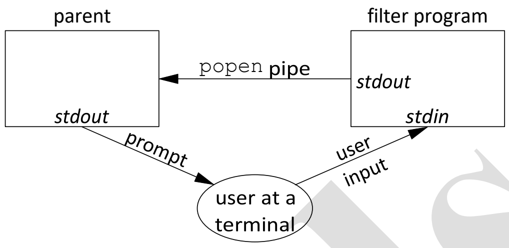


Figure 15.13 Transforming input using `popen`

The transformation could be pathname expansion, for example, or providing a history mechanism (remembering previously entered commands).

Figure 15.14 shows a simple filter to demonstrate this operation. The filter copies standard input to standard output, converting any uppercase character to lowercase. The reason we're careful to `fflush` standard output after writing a newline is discussed in the next section when we talk about coprocesses.

```
#include "apue.h"
#include <ctype.h>

int
main(void)
{
    int     c;

    while ((c = getchar()) != EOF)
        { if (isupper(c)) c =
        tolower(c); if (putchar(c)
        == EOF) err_sys("output
        error");
        if (c == '\n')
            fflush(stdout);
    }
    exit(0);
}
```

Figure 15.14 Filter to convert uppercase characters to lowercase

We compile this filter into the executable file `myuclc`, which we then invoke from the program in Figure 15.15 using `popen`.

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

int
main(void)
{ char line[MAXLINE];
    FILE *fpin;
    if ((fpin = popen("myuclc", "r")) ==
    NULL) err_sys("popen error"); for ( ; ; )
    { fputs("prompt> ", stdout);
        fflush(stdout);
        if (fgets(line, MAXLINE, fpin) == NULL) /* read from pipe
        */ break; if (fputs(line, stdout) == EOF) err_sys("fputs
        error to pipe");
    }
    if (pclose(fpin) == -1)
    err_sys("pclose error");
    putchar('\n'); exit(0);
}
```

Figure 15.15 Invoke uppercase/lowercase filter to read commands

We need to call `fflush` after writing the prompt, because the standard output is normally line buffered, and the prompt does not contain a newline. □

15.4 Coprocesses

A UNIX system filter is a program that reads from standard input and writes to standard output. Filters are normally connected linearly in shell pipelines. A filter becomes a *coprocess* when the same program generates the filter's input and reads the filter 's output.

The Korn shell provides coprocesses [Bolsky and Korn 1995]. The Bourne shell, the Bourne-again shell, and the C shell don't provide a way to connect processes together as coprocesses. A coprocess normally runs in the background from a shell, and its standard input and standard output are connected to another program using a pipe. Although the shell syntax required to initiate a coprocess and connect its input and output to other processes is quite contorted (see pp. 62–63 of Bolsky and Korn [1995] for all the details), coprocesses are also useful from a C program.

Whereas `popen` gives us a one-way pipe to the standard input or from the standard output of another process, with a coprocess we have two one-way pipes to the other process: one to its standard input and one from its standard output. We want to write to its standard input, let it operate on the data, and then read from its standard output.

Example

Let’s look at coprocesses with an example. The process creates two pipes: one is the standard input of the coprocess and the other is the standard output of the coprocess.
Figure 15.16 shows this arrangement.

–

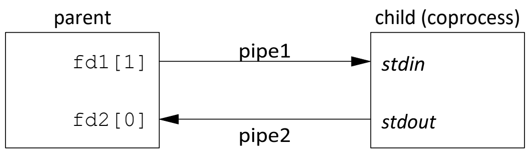


Figure 15.16 Driving a coprocess by writing its standard input and reading its standard output

The program in Figure 15.17 is a simple coprocess that reads two numbers from its standard input, computes their sum, and writes the sum to its standard output. (Coprocesses usually do more interesting work than we illustrate here. This example is admittedly contrived so that we can study the plumbing needed to connect the processes.)

```
#include "apue.h"

int
main(void)
{
    int     n, int1, int2;
    char line[MAXLINE];

    while ((n = read(STDIN_FILENO, line, MAXLINE)) > 0) {
        line[n] = 0;       /* null terminate */
        if (sscanf(line, "%d%d", &int1, &int2) == 2)
            { sprintf(line, "%d\n", int1 + int2); n =
            strlen(line);
            if (write(STDOUT_FILENO, line, n) != n)
                err_sys("write error");
        } else { if (write(STDOUT_FILENO, "invalid args\n", 13)
            != 13) err_sys("write error");
        }
    }
    exit(0);
}
```

Figure 15.17 Simple filter to add two numbers

We compile this program and leave the executable in the file `add2`.

The program in Figure 15.18 invokes the `add2` coprocess after reading two numbers from its standard input. The value from the coprocess is written to its standard output.

```
#include "apue.h" static void sig_pipe(int);   /* our

signal handler */ int

main(void)
{ int  n, fd1[2], fd2[2];
    pid_t pid; char
    line[MAXLINE];

    if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
        err_sys("signal error");

    if (pipe(fd1) < 0 || pipe(fd2) < 0)
        err_sys("pipe error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {  /* parent */ close(fd1[0]);
        close(fd2[1]);

        while (fgets(line, MAXLINE, stdin) != NULL) {
            n = strlen(line); if (write(fd1[1], line,
            n) != n) err_sys("write error to pipe");
            if ((n = read(fd2[0], line, MAXLINE)) <
            0) err_sys("read error from pipe"); if (n
            == 0) { err_msg("child closed pipe");
                break;
            } line[n] = 0; /* null terminate */
            if (fputs(line, stdout) == EOF)
            err_sys("fputs error");
        }

        if (ferror(stdin)) err_sys("fgets
            error on stdin");
        exit(0);
    } else {  /* child */ close(fd1[1]); close(fd2[0]); if
        (fd1[0] != STDIN_FILENO) { if (dup2(fd1[0],
        STDIN_FILENO) != STDIN_FILENO) err_sys("dup2 error
        to stdin");
            close(fd1[0]);
        }

        if (fd2[1] != STDOUT_FILENO) { if (dup2(fd2[1],
            STDOUT_FILENO) != STDOUT_FILENO) err_sys("dup2
            error to stdout");
            close(fd2[1]);
        }
        if (execl("./add2", "add2", (char *)0) < 0)
            err_sys("execl error");
```

_

```
     }
     exit(0);
}

static void
sig_pipe(int signo)
{ printf("SIGPIPE caught\n");
     exit(1);
}
```

Figure 15.18 Program to drive the add2 filter

Here, we create two pipes, with the parent and the child closing the ends they don't need. We have to use two pipes: one for the standard input of the coprocess and one for its standard output. The child then calls dup2 to move the pipe descriptors onto its standard input and standard output, before calling execl.

If we compile and run the program in Figure 15.18, it works as expected. Furthermore, if we kill the add2 coprocess while the program in Figure 15.18 is waiting for our input and then enter two numbers, the signal handler is invoked when the program writes to the pipe that has no reader. (See Exercise 15.4.) □

Example

In the coprocess add2 (Figure 15.17), we purposely used low-level I/O (UNIX system calls): read and write. What happens if we rewrite this coprocess to use standard I/O? Figure 15.19 shows the new version.

```
#include "apue.h"

int
main(void)
{
     int int1, int2; char
     line[MAXLINE];

     while (fgets(line, MAXLINE, stdin) != NULL) { if
          (sscanf(line, "%d%d", &int1, &int2) == 2) {
          if (printf("%d\n", int1 + int2) == EOF)
          err_sys("printf error");
          } else { if (printf("invalid args\n")
               == EOF) err_sys("printf error");
          }
```

```
    }
    exit(0);
}
```

Figure 15.19 Filter to add two numbers, using standard I/O

If we invoke this new coprocess from the program in Figure 15.18, it no longer works. The problem is the default standard I/O buffering. When the program in Figure 15.19 is invoked, the first `fgets` on the standard input causes the standard I/O library to allocate a buffer and choose the type of buffering. Since the standard input is a pipe, the standard I/O library defaults to fully buffered. The same thing happens with the standard output. While `add2` is blocked reading from its standard input, the program in Figure 15.18 is blocked reading from the pipe. We have a deadlock.

Here, we have control over the coprocess that's being run. We can change the program in Figure 15.19 by adding the following four lines before the `while` loop:

```
if (setvbuf(stdin, NULL, _IOLBF, 0) != 0)
err_sys("setvbuf error"); if
(setvbuf(stdout, NULL, _IOLBF, 0) != 0)
err_sys("setvbuf error");
```

These lines cause `fgets` to return when a line is available and cause `printf` to do an `fflush` when a newline is output (refer to Section 5.4 for the details on standard I/O buffering). Making these explicit calls to `setvbuf` fixes the program in Figure 15.19.

If we aren't able to modify the program that we're piping the output into, other techniques are required. For example, if we use `awk`(1) as a coprocess from our program (instead of the `add2` program), the following won't work:

```
#! /bin/awk -f
{ print $1 + $2 }
```

The reason this won't work is again the standard I/O buffering. But in this case, we cannot change the way `awk` works (unless we have the source code for it). We are unable to modify the executable of `awk` in any way to change the way the standard I/O buffering is handled.

The solution for this general problem is to make the coprocess being invoked (`awk` in this case) think that its standard input and standard output are connected to a terminal. That causes the standard I/O routines in the coprocess to line buffer these two I/O streams, similar to what we did with the explicit calls to `setvbuf` previously. We use pseudo terminals to do this in Chapter 19. □

15.5 FIFOs

FIFOs are sometimes called named pipes. Unnamed pipes can be used only between related processes when a common ancestor has created the pipe. With FIFOs, however, unrelated processes can exchange data.

We saw in Chapter 4 that a FIFO is a type of file. One of the encodings of the `st_mode` member of the `stat` structure (Section 4.2) indicates that a file is a FIFO. We can test for this with the `S_ISFIFO` macro.

Creating a FIFO is similar to creating a file. Indeed, the *pathname* for a FIFO exists in the file system.

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

int mkfifo(const char *path, mode_t mode); int

mkfifoat(int fd, const char *path, mode_t mode);

                                    Both return: 0 if OK, −1 on error
```

The specification of the *mode* argument is the same as for the `open` function (Section 3.3). The rules for the user and group ownership of the new FIFO are the same as we described in Section 4.6.

The `mkfifoat` function is similar to the `mkfifo` function, except that it can be used to create a FIFO in a location relative to the directory represented by the *fd* file descriptor argument. Like the other `*at` functions, there are three cases:

1. If the *path* parameter specifies an absolute pathname, then the *fd* parameter is ignored and the `mkfifoat` function behaves like the `mkfifo` function.
2. If the *path* parameter specifies a relative pathname and the *fd* parameter is a valid file descriptor for an open directory, the pathname is evaluated relative to this directory.
3. If the *path* parameter specifies a relative pathname and the *fd* parameter has the special value `AT_FDCWD`, the pathname is evaluated starting in the current working directory, and `mkfifoat` behaves like `mkfifo`.

Once we have used `mkfifo` or `mkfifoat` to create a FIFO, we open it using `open`. Indeed, the normal file I/O functions (e.g., `close`, `read`, `write`, `unlink`) all work with FIFOs.

Applications can create FIFOs with the `mknod` and `mknodat` functions. Because POSIX.1 originally didn't include `mknod`, the `mkfifo` function was invented specifically for POSIX.1. The `mknod` and `mknodat` functions are included in the XSI option in POSIX.1.

POSIX.1 also includes support for the `mkfifo`(1) command. All four platforms discussed in this text provide this command. As a result, we can create a FIFO using a shell command and then access it with the normal shell I/O redirection.

When we open a FIFO, the nonblocking flag (O_NONBLOCK) affects what happens.

- In the normal case (without O_NONBLOCK), an open for read-only blocks until some other process opens the FIFO for writing. Similarly, an open for writeonly blocks until some other process opens the FIFO for reading.
- If O_NONBLOCK is specified, an open for read-only returns immediately. But an open for write-only returns −1 with errno set to ENXIO if no process has the FIFO open for reading.

As with a pipe, if we write to a FIFO that no process has open for reading, the signal SIGPIPE is generated. When the last writer for a FIFO closes the FIFO, an end of file is generated for the reader of the FIFO.

It is common to have multiple writers for a given FIFO. This means that we have to worry about atomic writes if we don't want the writes from multiple processes to be interleaved. As with pipes, the constant PIPE_BUF specifies the maximum amount of data that can be written atomically to a FIFO. There are two uses for FIFOs.

1. FIFOs are used by shell commands to pass data from one shell pipeline toanother without creating intermediate temporary files.
2. FIFOs are used as rendezvous points in client–server applications to pass databetween the clients and the servers.

We discuss each of these uses with an example.

Example — Using FIFOs to Duplicate Output Streams

FIFOs can be used to duplicate an output stream in a series of shell commands. This prevents writing the data to an intermediate disk file (similar to using pipes to avoid intermediate disk files). But whereas pipes can be used only for linear connections between processes, a FIFO has a name, so it can be used for nonlinear connections.

Consider a procedure that needs to process a filtered input stream twice. Figure 15.20 shows this arrangement.

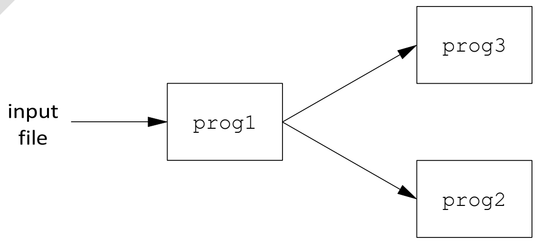


Figure 15.20 Procedure that processes a filtered input stream twice

With a FIFO and the UNIX program tee(1), we can accomplish this procedure without using a temporary file. (The tee program copies its standard input to both its standard output and the file named on its command line.)

```
mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2
```

We create the FIFO and then start prog3 in the background, reading from the FIFO. We then start prog1 and use tee to send its input to both the FIFO and prog2. Figure 15.21 shows the process arrangement. □

Example — Client–Server Communication Using a FIFO

Another use for FIFOs is to send data between a client and a server. If we have a server that is contacted by numerous clients, each client can write its request to a well-known

_

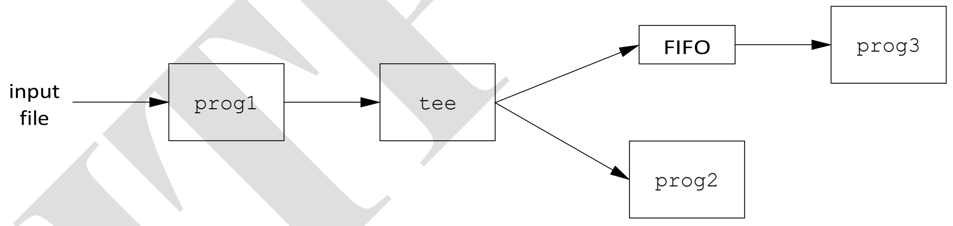


Figure 15.21 Using a FIFO and tee to send a stream to two different processes

FIFO that the server creates. (By ‘‘well-known,’’ we mean that the pathname of the FIFO is known to all the clients that need to contact the server.) Figure 15.22 shows this arrangement.

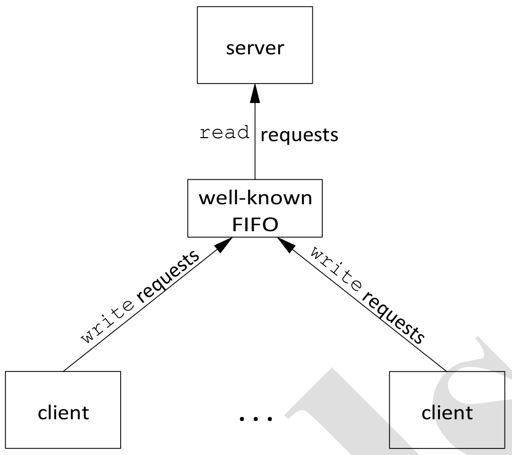


Figure 15.22 Clients sending requests to a server using a FIFO

Since there are multiple writers for the FIFO, the requests sent by the clients to the server need to be less than `PIPE_BUF` bytes in size. This prevents any interleaving of the client `write`s.

The problem in using FIFOs for this type of client–server communication is how to send replies back from the server to each client. A single FIFO can't be used, as the clients would never know when to read their response versus responses for other clients. One solution is for each client to send its process ID with the request. The server then creates a unique FIFO for each client, using a pathname based on the client's process ID. For example, the server can create a FIFO with the name `/tmp/serv1.XXXXX`, where `XXXXX` is replaced with the client's process ID. This arrangement is shown in Figure 15.23.

This arrangement works, although it is impossible for the server to tell whether a client crashes. A client crash leaves the client-specific FIFO in the file system. The

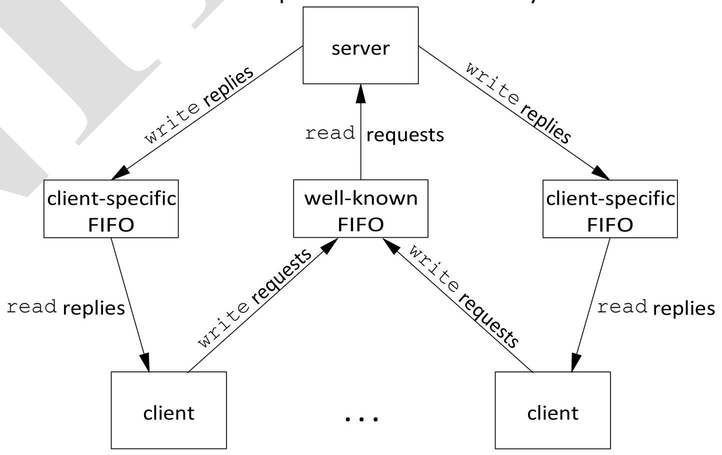


Figure 15.23 Client–server communication using FIFOs

server also must catch SIGPIPE, since it's possible for a client to send a request and terminate before reading the response, leaving the client-specific FIFO with one writer (the server) and no reader.

With the arrangement shown in Figure 15.23, if the server opens its well-known FIFO read-only (since it only reads from it) each time the number of clients goes from 1 to 0, the server will read an end of file on the FIFO. To prevent the server from having to handle this case, a common trick is just to have the server open its well-known FIFO for read–write. (See Exercise 15.10.) □

15.6 XSI IPC

The three types of IPC that we call XSI IPC—message queues, semaphores, and shared memory — have many similarities. In this section, we cover these similar features; in the following sections, we look at the specific functions for each of the three IPC types.

The XSI IPC functions are based closely on the System V IPC functions. These three types of IPC originated in the 1970s in an internal AT&T version of the UNIX System called ''Columbus UNIX.'' These IPC features were later added to System V. They are often criticized for inventing their own namespace instead of using the file system.

15.6.1 Identifiers and Keys

Each *IPC structure* (message queue, semaphore, or shared memory segment) in the kernel is referred to by a non-negative integer *identifier*. To send a message to or fetch a message from a message queue, for example, all we need know is the identifier for the queue. Unlike file descriptors, IPC identifiers are not small integers. Indeed, when a

_

given IPC structure is created and then removed, the identifier associated with that structure continually increases until it reaches the maximum positive value for an integer, and then wraps around to 0.

The identifier is an internal name for an IPC object. Cooperating processes need an external naming scheme to be able to rendezvous using the same IPC object. For this purpose, an IPC object is associated with a *key* that acts as an external name.

Whenever an IPC structure is being created (by calling msgget, semget, or shmget), a key must be specified. The data type of this key is the primitive system data type key_t, which is often defined as a long integer in the header <sys/types.h>. This key is converted into an identifier by the kernel.

There are various ways for a client and a server to rendezvous at the same IPC structure.

1. The server can create a new IPC structure by specifying a key of IPC_PRIVATE and store the returned identifier somewhere (such as a file) for the client to obtain. The key IPC_PRIVATE guarantees that the server creates a new IPC structure. The disadvantage of this technique is that file system operations are required for the server to write the integer identifier to a file, and then for the clients to retrieve this identifier later.

   The IPC_PRIVATE key is also used in a parent–child relationship. The parent creates a new IPC structure specifying IPC_PRIVATE, and the resulting identifier is then available to the child after the fork. The child can pass the identifier to a new program as an argument to one of the exec functions.
2. The client and the server can agree on a key by defining the key in a commonheader, for example. The server then creates a new IPC structure specifying this key. The problem with this approach is that it's possible for the key to already be associated with an IPC structure, in which case the get function (msgget, semget, or shmget) returns an error. The server must handle this error, deleting the existing IPC structure, and try to create it again.
3. The client and the server can agree on a pathname and project ID (the project IDis a character value between 0 and 255) and call the function ftok to convert these two values into a key. This key is then used in step 2. The only service provided by ftok is a way of generating a key from a pathname and project ID.

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

key_t ftok(const char *path, int id);

                                   Returns: key if OK, (key_t) −1 on error
```

The *path* argument must refer to an existing file. Only the lower 8 bits of *id* are used when generating the key.

The key created by ftok is usually formed by taking parts of the st_dev and st_ino fields in the stat structure (Section 4.2) corresponding to the given pathname and combining them with the project ID. If two pathnames refer to two different files,
then ftok usually returns two different keys for the two pathnames. However, because both i-node numbers and keys are often stored in long integers, information loss can occur when creating a key. This means that two different pathnames to different files can generate the same key if the same project ID is used.

The three get functions (msgget, semget, and shmget) all have two similar arguments: a *key* and an integer *flag*. A new IPC structure is created (normally by a server) if either *key* is IPC_PRIVATE or *key* is not currently associated with an IPC structure of the particular type and the IPC_CREAT bit of *flag* is specified. To reference an existing queue (normally done by a client), *key* must equal the key that was specified when the queue was created, and IPC_CREAT must not be specified.

Note that it's never possible to specify IPC_PRIVATE to reference an existing queue, since this special *key* value always creates a new queue. To reference an existing queue that was created with a *key* of IPC_PRIVATE, we must know the associated identifier and then use that identifier in the other IPC calls (such as msgsnd and msgrcv), bypassing the get function.

If we want to create a new IPC structure, making sure that we don't reference an existing one with the same identifier, we must specify a *flag* with both the IPC_CREAT and IPC_EXCL bits set. Doing this causes an error return of EEXIST if the IPC structure already exists. (This is similar to an open that specifies the O_CREAT and O_EXCL
flags.)

15.6.2 Permission Structure

XSI IPC associates an ipc_perm structure with each IPC structure. This structure defines the permissions and owner and includes at least the following members:

```
struct ipc_perm {
  uid_t uid; /* owner's effective user ID */ gid_t
  gid; /* owner's effective group ID */ uid_t
  cuid; /* creator's effective user ID */ gid_t
  cgid; /* creator's effective group ID */ mode_t
  mode; /* access modes */

  ...

};
```

Each implementation includes additional members. See <sys/ipc.h> on your system for the complete definition.

All the fields are initialized when the IPC structure is created. At a later time, we can modify the uid, gid, and mode fields by calling msgctl, semctl, or shmctl. To change these values, the calling process must be either the creator of the IPC structure or the superuser. Changing these fields is similar to calling chown or chmod for a file.

The values in the mode field are similar to the values we saw in Figure 4.6, but there is nothing corresponding to execute permission for any of the IPC structures. Also, message queues and shared memory use the terms *read* and *write*, but semaphores use the terms *read* and *alter*. Figure 15.24 shows the six permissions for each form of IPC.

Permission	Bit
user-read	0400
user-write (alter)	0200
group-read	0040
group-write (alter)	0020
other-read	0004
other-write (alter)	0002

Figure 15.24 XSI IPC permissions

Some implementations define symbolic constants to represent each permission, but these constants are not standardized by the Single UNIX Specification.

15.6.3 Configuration Limits

All three forms of XSI IPC have built-in limits that we may encounter. Most of these limits can be changed by reconfiguring the kernel. We describe the limits when we describe each of the three forms of IPC.

Each platform provides its own way to report and modify a particular limit. FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8 provide the `sysctl` command to view and modify kernel configuration parameters. On Solaris 10, changes to kernel IPC limits are made with the `prctl` command.

On Linux, you can display the IPC-related limits by running `ipcs -l`. On FreeBSD and Mac OS X, the equivalent command is `ipcs -T`. On Solaris, you can discover the tunable parameters by running `sysdef -i`.

15.6.4 Advantages and Disadvantages

A fundamental problem with XSI IPC is that the IPC structures are systemwide and do not have a reference count. For example, if we create a message queue, place some messages on the queue, and then terminate, the message queue and its contents are not deleted. They remain in the system until specifically read or deleted by some process calling `msgrcv` or `msgctl`, by someone executing the `ipcrm`(1) command, or by the system being rebooted. Compare this with a pipe, which is completely removed when the last process to reference it terminates. With a FIFO, although the name stays in the file system until explicitly removed, any data left in a FIFO is removed when the last process to reference the FIFO terminates.

Another problem with XSI IPC is that these IPC structures are not known by names in the file system. We can't access them and modify their properties with the functions we described in Chapters 3 and 4. Almost a dozen new system calls (`msgget`, `semop`, `shmat`, and so on) were added to the kernel to support these IPC objects. We can't see the IPC objects with an `ls` command, we can't remove them with the `rm` command, and we can't change their permissions with the `chmod` command. Instead, two new commands —`ipcs`(1) and `ipcrm`(1)—were added.

Since these forms of IPC don't use file descriptors, we can't use the multiplexed I/O functions (`select` and `poll`) with them. This makes it harder to use more than one of these IPC structures at a time or to use any of these IPC structures with file or device I/O. For example, we can't have a server wait for a message to be placed on one of two message queues without some form of busy–wait loop.

An overview of a transaction processing system built using System V IPC is given in Andrade, Carges, and Kovach [1989]. They claim that the namespace used by System V IPC (the identifiers) is an advantage, not a problem as we said earlier, because using identifiers allows a process to send a message to a message queue with a single function call (`msgsnd`), whereas other forms of IPC normally require an `open`, `write`, and `close`. This argument is false. Clients still have to obtain the identifier for the server 's queue somehow, to avoid using a key and calling `msgget`. The identifier assigned to a particular queue depends on how many other message queues exist when the queue is created and how many times the table in the kernel assigned to the new queue has been used since the kernel was bootstrapped. This is a dynamic value that can't be guessed or stored in a header. As we mentioned in Section 15.6.1, minimally a server has to write the assigned queue identifier to a file for its clients to read.

Other advantages listed by these authors for message queues are that they're reliable, flow controlled, and record oriented, and that they can be processed in other than first-in, first-out order. Figure 15.25 compares some of the features of these various forms of IPC.

IPC type	Connectionless?	Reliable?	Flow control?	Records?	Message types or priorities?
message queues	no	yes	yes	yes	yes
STREAMS	no	yes	yes	yes	yes
UNIX domain stream socket	no	yes	yes	no	no
UNIX domain datagram socket	yes	yes	no	yes	no
FIFOs (non-STREAMS)	no	yes	yes	no	no

Figure 15.25 Comparison of features of various forms of IPC

(We describe stream and datagram sockets in Chapter 16. We describe UNIX domain sockets in Section 17.2.) By ‘‘connectionless,’’ we mean the ability to send a message without having to call some form of an open function first. As described previously, we don't consider message queues connectionless, since some technique is required to obtain the identifier for a queue. Since all these forms of IPC are restricted to a single host, all are reliable. When the messages are sent across a network, the possibility of messages being lost becomes a concern. ‘‘Flow control’’ means that the sender is put to sleep if there is a shortage of system resources (buffers) or if the receiver can't accept any more messages. When the flow control condition subsides (i.e., when there is room in the queue), the sender should automatically be awakened.

One feature that we don't show in Figure 15.25 is whether the IPC facility can automatically create a unique connection to a server for each client. We'll see in Chapter 17 that UNIX stream sockets provide this capability. The next three sections describe each of the three forms of XSI IPC in detail.

15.7 Message Queues

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. We'll call the message queue just a *queue* and its identifier a *queue ID*.

The Single UNIX Specification message-passing option includes an alternative IPC message queue interface derived from the POSIX real-time extensions. We do not discuss it in this text.

A new queue is created or an existing queue opened by msgget. New messages are added to the end of a queue by msgsnd. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd when the message is added to a queue. Messages are fetched from a queue by msgrcv. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field. Each queue has the following msqid_ds structure associated with it:

```
struct msqid_ds {
   struct ipc_perm msg_perm;      /* see Section 15.6.2 */
  msgqnum_t        msg_qnum;       /* # of messages on queue */
  msglen_t         msg_qbytes; /* max # of bytes on queue */
  pid_t            msg_lspid;    /* pid of last msgsnd() */
  pid_t            msg_lrpid;    /* pid of last msgrcv() */
  time_t           msg_stime;    /* last-msgsnd() time */
  time_t           msg_rtime;    /* last-msgrcv() time */
  time_t           msg_ctime;    /* last-change time */

  ...

};
```

This structure defines the current status of the queue. The members shown are the ones defined by the Single UNIX Specification. Implementations include additional fields not covered by the standard.

Description	Typical values			
	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
size in bytes of largest message we can send	16,384	8,192	16,384	derived
maximum size in bytes of a particular queue (i.e., the sum of all the sizes of messages on the queue)	2,048	16,384	2,048	65,536
maximum number of messages queues, systemwide	40	derived	40	128
maximum number of messages, systemwide	40	derived	40	8,192

Figure 15.26 System limits that affect message queues

Figure 15.26 lists the system limits that affect message queues. We show ''derived'' where a limit is derived from other limits. On Linux, for example, the maximum number of messages is based on the maximum number of queues and the maximum amount of data allowed on the queues. The maximum number of queues, in turn, is based on the amount of RAM installed in the system. Note that the queue maximum byte size limit further limits the maximum size of a message to be placed on a queue.

The first function normally called is msgget to either open an existing queue or create a new queue.

```
#include <sys/msg.h> int

msgget(key_t key, int flag);

                                    Returns: message queue ID if OK, −1 on error
```

In Section 15.6.1, we described the rules for converting the *key* into an identifier and discussed whether a new queue is created or an existing queue is referenced. When a new queue is created, the following members of the `msqid_ds` structure are initialized.

- The `ipc_perm` structure is initialized as described in Section 15.6.2. The `mode` member of this structure is set to the corresponding permission bits of *flag*. These permissions are specified with the values from Figure 15.24.
- `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime`, and `msg_rtime` are all set to 0.
- `msg_ctime` is set to the current time.
- `msg_qbytes` is set to the system limit.

On success, `msgget` returns the non-negative queue ID. This value is then used with the other three message queue functions.

The `msgctl` function performs various operations on a queue. This function and the related functions for semaphores and shared memory (`semctl` and `shmctl`) are the `ioctl`-like functions for XSI IPC (i.e., the garbage-can functions).

```
#include <sys/msg.h> int msgctl(int msqid, int cmd, struct

msqid_ds *buf);

                                    Returns: 0 if OK, −1 on error
```

The *cmd* argument specifies the command to be performed on the queue specified by *msqid*.

`IPC_STAT` Fetch the `msqid_ds` structure for this queue, storing it in the structure pointed to by *buf*.

`IPC_SET` Copy the following fields from the structure pointed to by *buf* to the `msqid_ds` structure associated with this queue: `msg_perm.uid`, `msg_perm.gid`, `msg_perm.mode`, and `msg_qbytes`. This command can be executed only by a process whose effective user ID equals `msg_perm.cuid` or `msg_perm.uid` or by a process with superuser privileges. Only the superuser can increase the value of `msg_qbytes`.

`IPC_RMID` Remove the message queue from the system and any data still on the queue. This removal is immediate. Any other process still using the message queue will get an error of `EIDRM` on its next attempted operation on the queue. This command can be executed only by a process whose effective user ID equals `msg_perm.cuid` or `msg_perm.uid` or by a process with superuser privileges.

We'll see that these three commands (IPC_STAT, IPC_SET, and IPC_RMID) are also provided for semaphores and shared memory.

Data is placed onto a message queue by calling msgsnd.

```
#include <sys/msg.h> int msgsnd(int msqid, const void *ptr, size_t

nbytes, int flag);

                                              Returns: 0 if OK, −1 on error
```

As we mentioned earlier, each message is composed of a positive long integer type field, a non-negative length (*nbytes*), and the actual data bytes (corresponding to the length). Messages are always placed at the end of the queue.

The *ptr* argument points to a long integer that contains the positive integer message type, and it is immediately followed by the message data. (There is no message data if *nbytes* is 0.) If the largest message we send is 512 bytes, we can define the following structure:

```
struct mymesg {
  long mtype;    /* positive message type */ char
mtext[512]; /* message data, of length nbytes */ };
```

The *ptr* argument is then a pointer to a mymesg structure. The message type can be used by the receiver to fetch messages in an order other than first in, first out.

Some platforms support both 32-bit and 64-bit environments. This affects the size of long integers and pointers. For example, on 64-bit SPARC systems, Solaris allows both 32-bit and 64-bit applications to coexist. If a 32-bit application were to exchange this structure over a pipe or a socket with a 64-bit application, problems would arise, because the size of a long integer is 4 bytes in a 32-bit application, but 8 bytes in a 64-bit application. This means that a 32-bit application will expect that the mtext field will start 4 bytes after the start of the structure, whereas a 64-bit application will expect the mtext field to start 8 bytes after the start of the structure. In this situation, part of the 64-bit application's mtype field will appear as part of the mtext field to the 32-bit application, and the first 4 bytes in the 32-bit application's mtext field will be interpreted as a part of the mtype field by the 64-bit application.

This problem doesn't happen with XSI message queues, however. Solaris implements the 32-bit version of the IPC system calls with different entry points than the 64-bit version of the IPC system calls. The system calls know how to deal with a 32-bit application communicating with a 64-bit application, and treat the type field specially to avoid it interfering with the data portion of the message. The only potential problem is a loss of information when a 64-bit application sends a message with a value in the 8-byte type field that is larger than will fit in a 32-bit application's 4-byte type field. In this case, the 32-bit application will see a truncated type value.

A *flag* value of IPC_NOWAIT can be specified. This is similar to the nonblocking I/O flag for file I/O (Section 14.2). If the message queue is full (either the total number of messages on the queue equals the system limit, or the total number of bytes on the queue equals the system limit), specifying IPC_NOWAIT causes msgsnd to return immediately with an error of EAGAIN. If IPC_NOWAIT is not specified, we are blocked until there is room for the message, the queue is removed from the system, or a signal is caught and the signal handler returns. In the second case, an error of `EIDRM` is returned (‘‘identifier removed’’); in the last case, the error returned is `EINTR`.

Note how ungracefully the removal of a message queue is handled. Since a reference count is not maintained with each message queue (as there is for open files), the removal of a queue simply generates errors on the next queue operation by processes still using the queue. Semaphores handle this removal in the same fashion. In contrast, when a file is removed, the file’s contents are not deleted until the last open descriptor for the file is closed.

When `msgsnd` returns successfully, the `msqid_ds` structure associated with the message queue is updated to indicate the process ID that made the call (`msg_lspid`), the time that the call was made (`msg_stime`), and that one more message is on the queue (`msg_qnum`).

Messages are retrieved from a queue by `msgrcv`.

```
#include <sys/msg.h> ssize_t msgrcv(int msqid, void *ptr, size_t nbytes,

long type, int flag);

                        Returns: size of data portion of message if OK, −1 on error
```

As with `msgsnd`, the *ptr* argument points to a long integer (where the message type of the returned message is stored) followed by a data buffer for the actual message data. *nbytes* specifies the size of the data buffer. If the returned message is larger than *nbytes* and the `MSG_NOERROR` bit in *flag* is set, the message is truncated. (In this case, no notification is given to us that the message was truncated, and the remainder of the message is discarded.) If the message is too big and this *flag* value is not specified, an error of `E2BIG` is returned instead (and the message stays on the queue). The *type* argument lets us specify which message we want.

type == 0	The first message on the queue is returned.
type > 0	The first message on the queue whose message type equals *type* is returned.
type < 0	The first message on the queue whose message type is the lowest value less than or equal to the absolute value of *type* is returned.

A nonzero *type* is used to read the messages in an order other than first in, first out. For example, the *type* could be a priority value if the application assigns priorities to the messages. Another use of this field is to contain the process ID of the client if a single message queue is being used by multiple clients and a single server (as long as a process ID fits in a long integer).

We can specify a *flag* value of `IPC_NOWAIT` to make the operation nonblocking, causing `msgrcv` to return −1 with `errno` set to `ENOMSG` if a message of the specified type is not available. If `IPC_NOWAIT` is not specified, the operation blocks until a message of the specified type is available, the queue is removed from the system (−1 is returned with `errno` set to `EIDRM`), or a signal is caught and the signal handler returns (causing `msgrcv` to return −1 with `errno` set to `EINTR`).

When `msgrcv` succeeds, the kernel updates the `msqid_ds` structure associated with the message queue to indicate the caller’s process ID (`msg_lrpid`), the time of the call (`msg_rtime`), and that one less message is on the queue (`msg_qnum`).

Example — Timing Comparison of Message Queues and Full-Duplex Pipes

If we need a bidirectional flow of data between a client and a server, we can use either message queues or full-duplex pipes. (Recall from Figure 15.1 that full-duplex pipes are available through the UNIX domain sockets mechanism [Section 17.2], although some platforms provide a full-duplex pipe mechanism through the `pipe` function.)

Figure 15.27 shows a timing comparison of three of these techniques on Solaris: message queues, full-duplex (STREAMS) pipes, and UNIX domain sockets. The tests consisted of a program that created the IPC channel, called `fork`, and then sent about 200 megabytes of data from the parent to the child. The data was sent using 100,000 calls to `msgsnd`, with a message length of 2,000 bytes for the message queue, and 100,000 calls to `write`, with a length of 2,000 bytes for the full-duplex pipe and UNIX domain socket. The times are all in seconds.

Operation	User	System	Clock
message queue	0.58	4.16	5.09
full-duplex pipe	0.61	4.30	5.24
UNIX domain socket	0.59	5.58	7.49

Figure 15.27 Timing comparison of IPC alternatives on Solaris

These numbers show us that message queues, originally implemented to provide higher-than-normal-speed IPC, are no longer that much faster than other forms of IPC. (When message queues were implemented, the only other form of IPC available was half-duplex pipes.) When we consider the problems in using message queues (Section 15.6.4), we come to the conclusion that we shouldn't use them for new applications.□

15.8 Semaphores

A semaphore isn't a form of IPC similar to the others that we've described (pipes, FIFOs, and message queues). A semaphore is a counter used to provide access to a shared data object for multiple processes.

The Single UNIX Specification includes an alternative set of semaphore interfaces that were originally part of its real-time extensions. We discuss these interfaces in Section 15.10.

To obtain a shared resource, a process needs to do the following:

1. Test the semaphore that controls the resource.
2. If the value of the semaphore is positive, the process can use the resource. Inthis case, the process decrements the semaphore value by 1, indicating that it has used one unit of the resource.
3. Otherwise, if the value of the semaphore is 0, the process goes to sleep until thesemaphore value is greater than 0. When the process wakes up, it returns to step 1.

When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1. If any other processes are asleep, waiting for the semaphore, they are awakened.

To implement semaphores correctly, the test of a semaphore's value and the decrementing of this value must be an atomic operation. For this reason, semaphores are normally implemented inside the kernel.

A common form of semaphore is called a *binary semaphore*. It controls a single resource, and its value is initialized to 1. In general, however, a semaphore can be initialized to any positive value, with the value indicating how many units of the shared resource are available for sharing.

XSI semaphores are, unfortunately, more complicated than this. Three features contribute to this unnecessary complication.

1. A semaphore is not simply a single non-negative value. Instead, we have todefine a semaphore as a set of one or more semaphore values. When we create a semaphore, we specify the number of values in the set.
2. The creation of a semaphore (`semget`) is independent of its initialization (`semctl`). This is a fatal flaw, since we cannot atomically create a new semaphore set and initialize all the values in the set.
3. Since all forms of XSI IPC remain in existence even when no process is usingthem, we have to worry about a program that terminates without releasing the semaphores it has been allocated. The *undo* feature that we describe later is supposed to handle this.

The kernel maintains a `semid_ds` structure for each semaphore set:

```
struct semid_ds {
  struct ipc_perm sem_perm; /* see Section 15.6.2 */
  unsigned short sem_nsems; /* # of semaphores in set */
  time_t  sem_otime; /* last-semop() time */ time_t
   sem_ctime; /* last-change time */

  ...

};
```

The Single UNIX Specification defines the fields shown, but implementations can define additional members in the `semid_ds` structure.

Each semaphore is represented by an anonymous structure containing at least the following members:

```
struct {
  unsigned short semval; /* semaphore value, always >= 0 */ pid_t
   sempid; /* pid for last operation */
  unsigned short semncnt; /* # processes awaiting semval>curval */
  unsigned short semzcnt; /* # processes awaiting semval==0 */

  ...

};
```

Figure 15.28 lists the system limits that affect semaphore sets.

Description	Typical values			
	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
maximum value of any semaphore	32,767	32,767	32,767	65,535
maximum value of any semaphore's adjust-on-exit value	16,384	32,767	16,384	32,767
maximum number of semaphore sets, systemwide	10	128	87,381	128
maximum number of semaphores, systemwide	60	32,000	87,381	derived
maximum number of semaphores per semaphore set	60	250	87,381	512
maximum number of undo structures, systemwide	30	32,000	87,381	derived
maximum number of undo entries per undo structures	10	unlimited	10	derived
maximum number of operations per `semop` call	100	32	5	512

Figure 15.28 System limits that affect semaphores

When we want to use XSI semaphores, we first need to obtain a semaphore ID by calling the `semget` function.

```
#include <sys/sem.h> int semget(key_t key, int nsems, int
flag);
```
Returns: semaphore ID if OK, −1 on error

In Section 15.6.1, we described the rules for converting the *key* into an identifier and discussed whether a new set is created or an existing set is referenced. When a new set is created, the following members of the `semid_ds` structure are initialized.

- The `ipc_perm` structure is initialized as described in Section 15.6.2. The `mode` member of this structure is set to the corresponding permission bits of *flag*. These permissions are specified with the values from Figure 15.24.
- `sem_otime` is set to 0.
- `sem_ctime` is set to the current time.
- `sem_nsems` is set to *nsems*.

The number of semaphores in the set is *nsems*. If a new set is being created (typically by the server), we must specify *nsems*. If we are referencing an existing set (a client), we can specify *nsems* as 0.

The `semctl` function is the catchall for various semaphore operations.

```
#include <sys/sem.h> int semctl(int semid, int semnum, int cmd, ... /*
union semun arg */ );
```
Returns: (see following)

The fourth argument is optional, depending on the command requested, and if present, is of type `semun`, a union of various command-specific arguments:

```
    union semun {
```

```
    int                val; /* for SETVAL */
    struct semid_ds *buf; /* for IPC_STAT and IPC_SET */
  unsigned short *array; /* for GETALL and SETALL */ };
```

Note that the optional argument is the actual union, not a pointer to the union.

Usually our application must define the `semun` union. However, on FreeBSD 8.0, this is defined for us in `<sys/sem.h>`.

The *cmd* argument specifies one of the following ten commands to be performed on the set specified by *semid*. The five commands that refer to one particular semaphore value use *semnum* to specify one member of the set. The value of *semnum* is between 0 and *nsems−1*, inclusive.

`IPC_STAT`	Fetch the `semid_ds` structure for this set, storing it in the structure pointed to by *arg.buf*.
`IPC_SET`	Set the `sem_perm.uid`, `sem_perm.gid`, and `sem_perm.mode` fields from the structure pointed to by *arg.buf* in the `semid_ds` structure associated with this set. This command can be executed only by a process whose effective user ID equals `sem_perm.cuid` or `sem_perm.uid` or by a process with superuser privileges.
`IPC_RMID`	Remove the semaphore set from the system. This removal is immediate. Any other process still using the semaphore will get an error of `EIDRM` on its next attempted operation on the semaphore. This command can be executed only by a process whose effective user ID equals `sem_perm.cuid` or `sem_perm.uid` or by a process with superuser privileges.
`GETVAL`	Return the value of `semval` for the member *semnum*.
`SETVAL`	Set the value of `semval` for the member *semnum*. The value is specified by *arg.val*.
`GETPID`	Return the value of `sempid` for the member *semnum*.
`GETNCNT`	Return the value of `semncnt` for the member *semnum*.
`GETZCNT`	Return the value of `semzcnt` for the member *semnum*.
`GETALL`	Fetch all the semaphore values in the set. These values are stored in the array pointed to by *arg.array*.
`SETALL`	Set all the semaphore values in the set to the values pointed to by *arg.array*.

For all the `GET` commands other than `GETALL`, the function returns the corresponding value. For the remaining commands, the return value is 0 if the call succeeds. On error, the `semctl` function sets `errno` and returns −1.

The function `semop` atomically performs an array of operations on a semaphore set.

```
#include <sys/sem.h> int semop(int semid, struct sembuf semoparray[],

size_t nops);
```

Returns: 0 if OK, −1 on error

The *semoparray* argument is a pointer to an array of semaphore operations, represented by `sembuf` structures:

```
 struct sembuf {
   unsigned short sem_num; /* member # in set (0, 1, ..., nsems-1) */
   short     sem_op; /* operation (negative, 0, or positive) */ short
       sem_flg; /* IPC_NOWAIT, SEM_UNDO */
 };
```

The *nops* argument specifies the number of operations (elements) in the array.

The operation on each member of the set is specified by the corresponding `sem_op` value. This value can be negative, 0, or positive. (In the following discussion, we refer to the ‘‘undo’’ flag for a semaphore. This flag corresponds to the `SEM_UNDO` bit in the corresponding `sem_flg` member.)

1. The easiest case is when `sem_op` is positive. This case corresponds to the returning of resources by the process. The value of `sem_op` is added to the semaphore's value. If the undo flag is specified, `sem_op` is also subtracted from the semaphore's adjustment value for this process.
2. If `sem_op` is negative, we want to obtain resources that the semaphore controls.

   If the semaphore's value is greater than or equal to the absolute value of `sem_op` (the resources are available), the absolute value of `sem_op` is subtracted from the semaphore's value. This guarantees the resulting semaphore value is greater than or equal to 0. If the undo flag is specified, the absolute value of `sem_op` is also added to the semaphore's adjustment value for this process.

   If the semaphore's value is less than the absolute value of `sem_op` (the resources are not available), the following conditions apply.

a. If `IPC_NOWAIT` is specified, `semop` returns with an error of `EAGAIN`.

b. If `IPC_NOWAIT` is not specified, the `semncnt` value for this semaphore is incremented (since the caller is about to go to sleep), and the calling process is suspended until one of the following occurs.

   i. The semaphore's value becomes greater than or equal to the absolute value of `sem_op` (i.e., some other process has released some resources). The value of `semncnt` for this semaphore is decremented (since the calling process is done waiting), and the absolute value of `sem_op` is subtracted from the semaphore's value. If the undo flag is specified, the absolute value of `sem_op` is also added to the semaphore's adjustment value for this process. ii. The semaphore is removed from the system. In this case, the function returns an error of `EIDRM`.

   iii. A signal is caught by the process, and the signal handler returns. In this case, the value of `semncnt` for this semaphore is decremented (since the calling process is no longer waiting), and the function returns an error of `EINTR`.

3. If `sem_op` is 0, this means that the calling process wants to wait until the semaphore's value becomes 0.
   If the semaphore's value is currently 0, the function returns immediately.

   If the semaphore's value is nonzero, the following conditions apply.

   a. If `IPC_NOWAIT` is specified, return is made with an error of `EAGAIN`.

   b. If `IPC_NOWAIT` is not specified, the `semzcnt` value for this semaphore is incremented (since the caller is about to go to sleep), and the calling process is suspended until one of the following occurs.

      i. The semaphore's value becomes 0. The value of `semzcnt` for this semaphore is decremented (since the calling process is done waiting).

      ii. The semaphore is removed from the system. In this case, the function returns an error of `EIDRM`.

      iii. A signal is caught by the process, and the signal handler returns. In thiscase, the value of `semzcnt` for this semaphore is decremented (since the calling process is no longer waiting), and the function returns an error of `EINTR`.

The `semop` function operates atomically; it does either all the operations in the array or none of them.

Semaphore Adjustment on `exit`

As we mentioned earlier, it is a problem if a process terminates while it has resources allocated through a semaphore. Whenever we specify the `SEM_UNDO` flag for a semaphore operation and we allocate resources (a `sem_op` value less than 0), the kernel remembers how many resources we allocated from that particular semaphore (the absolute value of `sem_op`). When the process terminates, either voluntarily or involuntarily, the kernel checks whether the process has any outstanding semaphore adjustments and, if so, applies the adjustment to the corresponding semaphore.

If we set the value of a semaphore using `semctl`, with either the `SETVAL` or `SETALL` commands, the adjustment value for that semaphore in all processes is set to 0.

Example — Timing Comparison of Semaphores, Record Locking, and Mutexes

If we are sharing a single resource among multiple processes, we can use one of three techniques to coordinate access. We can use a a semaphore, record locking, or a mutex that is mapped into the address spaces of both processes. It's interesting to compare the timing differences between the three techniques.

With a semaphore, we create a semaphore set consisting of a single member and initialize the semaphore's value to 1. To allocate the resource, we call `semop` with a `sem_op` of −1; to release the resource, we perform a `sem_op` of +1. We also specify `SEM_UNDO` with each operation, to handle the case of a process that terminates without releasing its resource.

With record locking, we create an empty file and use the first byte of the file (which need not exist) as the lock byte. To allocate the resource, we obtain a write lock on the byte; to release it, we unlock the byte. The record locking properties guarantee that if a process terminates while holding a lock, the kernel automatically releases the lock.

To use a mutex, we need both processes to map the same file into their address spaces and initialize a mutex at the same offset in the file using the `PTHREAD_PROCESS_SHARED` mutex attribute. To allocate the resource, we lock the mutex; to release the resource, we unlock the mutex. If a process terminates without releasing the mutex, recovery is difficult unless we use a robust mutex (recall the `pthread_mutex_consistent` function discussed in Section 12.4.1).

Figure 15.29 shows the time required to perform these three locking techniques on Linux. In each case, the resource was allocated and then released 1,000,000 times. This was done simultaneously by three different processes. The times in Figure 15.29 are the totals in seconds for all three processes.

Operation	User	System	Clock
semaphores with undo	0.50	6.08	7.55
advisory record locking	0.51	9.06	4.38
mutex in shared memory	0.21	0.40	0.25

Figure 15.29 Timing comparison of locking alternatives on Linux

On Linux, record locking is faster than semaphores, but mutexes in shared memory outperform both semaphores and record locking. If we're locking a single resource and don't need all the fancy features of XSI semaphores, record locking is preferred over semaphores. The reasons are that it is much simpler to use, it is faster (on this platform), and the system takes care of any lingering locks when a process terminates. Even though using a mutex in shared memory is the fastest option on this platform, we still prefer to use record locking, unless performance is the primary concern. There are two reasons for this. First, recovery from process termination is more difficult using a mutex in memory shared among multiple processes. Second, the *process-shared* mutex attribute isn't universally supported yet. In older versions of the Single UNIX Specification, it was optional. Although it is still optional in SUSv4, it is now required by all XSI-conforming implementations.

Of the four platforms covered in this text, only Linux 3.2.0 and Solaris 10 currently support the *process-shared* mutex attribute.

□

15.9 Shared Memor y

Shared memory allows two or more processes to share a given region of memory. This is the fastest form of IPC, because the data does not need to be copied between the client and the server. The only trick in using shared memory is synchronizing access to a given region among multiple processes. If the server is placing data into a shared memory region, the client shouldn't try to access the data until the server is done. Often, semaphores are used to synchronize shared memory access. (But as we saw at the end of the previous section, record locking or mutexes can also be used.)

The Single UNIX Specification shared memory objects option includes alternative interfaces, originally real-time extensions, to access shared memory. We don't discuss them in this text.

We've already seen one form of shared memory when multiple processes map the same file into their address spaces. The XSI shared memory differs from memorymapped files in that there is no associated file. The XSI shared memory segments are anonymous segments of memory.

The kernel maintains a structure with at least the following members for each shared memory segment:

```
struct shmid_ds {
  struct ipc_perm shm_perm; /* see Section 15.6.2 */
  size_t          shm_segsz; /* size of segment in bytes */
  pid_t           shm_lpid;   /* pid of last shmop() */
  pid_t           shm_cpid;   /* pid of creator */
  shmatt_t        shm_nattch; /* number of current attaches */
  time_t          shm_atime; /* last-attach time */
  time_t          shm_dtime; /* last-detach time */
  time_t          shm_ctime; /* last-change time */

  ...

};
```

(Implementations add other structure members to support shared memory segments.)

The type `shmatt_t` is defined to be an unsigned integer at least as large as an `unsigned short`. Figure 15.30 lists the system limits that affect shared memory.

Description	Typical values			
	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
maximum size in bytes of a shared memory segment	33,554,432	32,768	4,194,304	derived
minimum size in bytes of a shared memory segment	1	1	1	1
maximum number of shared memory segments, systemwide	192	4,096	32	128
maximum number of shared memory segments, per process	128	4,096	8	128

Figure 15.30 System limits that affect shared memory

The first function called is usually `shmget`, to obtain a shared memory identifier.

```
#include <sys/shm.h> int shmget(key_t key, size_t size, int

flag);
```

Returns: shared memory ID if OK, −1 on error

In Section 15.6.1, we described the rules for converting the *key* into an identifier and whether a new segment is created or an existing segment is referenced. When a new segment is created, the following members of the `shmid_ds` structure are initialized.

- The ipc_perm structure is initialized as described in Section 15.6.2. The mode member of this structure is set to the corresponding permission bits of *flag*. These permissions are specified with the values from Figure 15.24.
- shm_lpid, shm_nattch, shm_atime, and shm_dtime are all set to 0.
- shm_ctime is set to the current time.
- shm_segsz is set to the *size* requested.

The *size* parameter is the size of the shared memory segment in bytes. Implementations will usually round up this size to a multiple of the system's page size, but if an application specifies *size* as a value other than an integral multiple of the system's page size, the remainder of the last page will be unavailable for use. If a new segment is being created (typically by the server), we must specify its *size*. If we are referencing an existing segment (a client), we can specify *size* as 0. When a new segment is created, the contents of the segment are initialized with zeros.

The shmctl function is the catchall for various shared memory operations.

```
#include <sys/shm.h> int shmctl(int shmid, int cmd, struct

shmid_ds *buf);

                                          Returns: 0 if OK, −1 on error
```

The *cmd* argument specifies one of the following five commands to be performed, on the segment specified by *shmid*.

IPC_STAT	Fetch the shmid_ds structure for this segment, storing it in the structure pointed to by *buf*.
IPC_SET	Set the following three fields from the structure pointed to by *buf* in the shmid_ds structure associated with this shared memory segment: shm_perm.uid, shm_perm.gid, and shm_perm.mode. This command can be executed only by a process whose effective user ID equals shm_perm.cuid or shm_perm.uid or by a process with superuser privileges.
IPC_RMID	Remove the shared memory segment set from the system. Since an attachment count is maintained for shared memory segments (the shm_nattch field in the shmid_ds structure), the segment is not removed until the last process using the segment terminates or detaches it. Regardless of whether the segment is still in use, the segment's identifier is immediately removed so that shmat can no longer attach the segment. This command can be executed only by a process whose effective user ID equals shm_perm.cuid or shm_perm.uid or by a process with superuser privileges.

Two additional commands are provided by Linux and Solaris, but are not part of the Single UNIX Specification.

`SHM_LOCK` Lock the shared memory segment in memory. This command can be executed only by the superuser.

`SHM_UNLOCK` Unlock the shared memory segment. This command can be executed only by the superuser.

Once a shared memory segment has been created, a process attaches it to its address space by calling `shmat`.

```
#include <sys/shm.h> void *shmat(int shmid, const void *addr,
int flag);
                    Returns: pointer to shared memory segment if OK, −1 on error
```

The address in the calling process at which the segment is attached depends on the *addr* argument and whether the `SHM_RND` bit is specified in *flag*.

- If *addr* is 0, the segment is attached at the first available address selected by the kernel. This is the recommended technique.
- If *addr* is nonzero and `SHM_RND` is not specified, the segment is attached at the address given by *addr*.
- If *addr* is nonzero and `SHM_RND` is specified, the segment is attached at the address given by (*addr* − (*addr* modulus `SHMLBA`)). The `SHM_RND` command stands for ''round.'' `SHMLBA` stands for ''low boundary address multiple'' and is always a power of 2. What the arithmetic does is round the address down to the next multiple of `SHMLBA`.

Unless we plan to run the application on only a single type of hardware (which is highly unlikely today), we should not specify the address where the segment is to be attached. Instead, we should specify an *addr* of 0 and let the system choose the address. If the `SHM_RDONLY` bit is specified in *flag*, the segment is attached as read-only. Otherwise, the segment is attached as read–write.

The value returned by `shmat` is the address at which the segment is attached, or −1 if an error occurred. If `shmat` succeeds, the kernel will increment the `shm_nattch` counter in the `shmid_ds` structure associated with the shared memory segment.

When we're done with a shared memory segment, we call `shmdt` to detach it. Note that this does not remove the identifier and its associated data structure from the system. The identifier remains in existence until some process (often a server) specifically removes it by calling `shmctl` with a command of `IPC_RMID`.

```
    #include <sys/shm.h> int
    shmdt(const void *addr);
                                             Returns: 0 if OK, −1 on error
```

The *addr* argument is the value that was returned by a previous call to `shmat`. If successful, `shmdt` will decrement the `shm_nattch` counter in the associated `shmid_ds` structure.

Example

Where a kernel places shared memory segments that are attached with an address of 0 is highly system dependent. Figure 15.31 shows a program that prints some information on where one particular system places various types of data.

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

#define ARRAY_SIZE 40000
#define MALLOC_SIZE 100000
#define SHM_SIZE    100000
#define SHM_MODE    0600 /* user read/write */ char

array[ARRAY_SIZE]; /* uninitialized data = bss */

int
main(void)
{
    int shmid; char *ptr,
    *shmptr;

    printf("array[] from %p to %p\n", (void *)&array[0],
      (void *)&array[ARRAY_SIZE]);
    printf("stack around %p\n", (void
    *)&shmid);

    if ((ptr = malloc(MALLOC_SIZE)) == NULL)
    err_sys("malloc error"); printf("malloced from
    %p to %p\n", (void *)ptr,
      (void *)ptr+MALLOC_SIZE);

    if ((shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE)) < 0)
    err_sys("shmget error"); if ((shmptr = shmat(shmid, 0, 0)) ==
    (void *)-1) err_sys("shmat error"); printf("shared memory
    attached from %p to %p\n", (void *)shmptr,
      (void *)shmptr+SHM_SIZE);

    if (shmctl(shmid, IPC_RMID, 0) < 0)
        err_sys("shmctl error");

    exit(0);
}
```

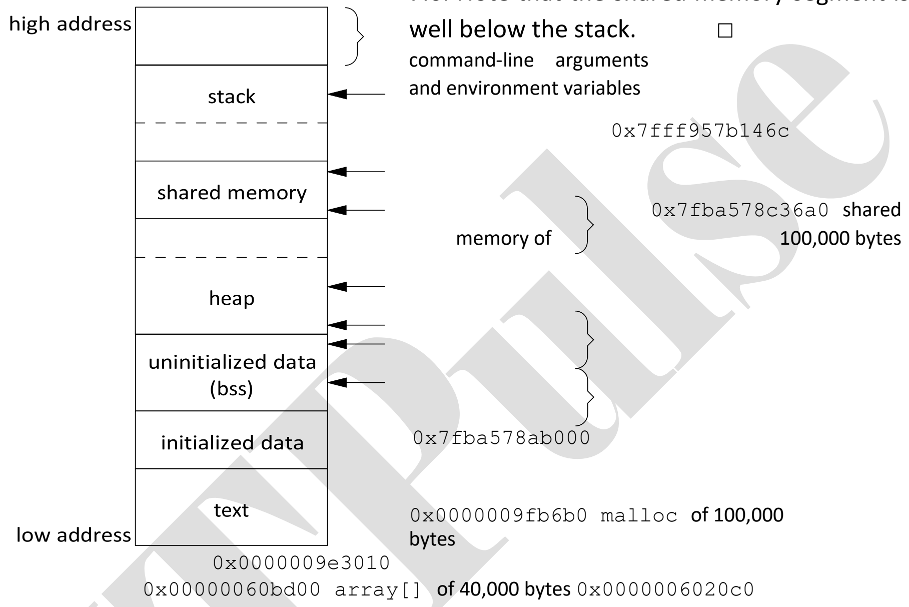
Figure 15.31 Print where various types of data are stored

Running this program on a 64-bit Intel-based Linux system gives us the following output:

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

```
array[] from 0x6020c0 to 0x60bd00
stack around 0x7fff957b146c
malloced from 0x9e3010 to 0x9fb6b0
shared memory attached from 0x7fba578ab000 to 0x7fba578c36a0
```

Figure 15.32 shows a picture of this, similar to what we said was a typical memory layout in Figure 7.6. Note that the shared memory segment is placed well below the stack. □



high address
stack
shared memory
heap
uninitialized data (bss)
initialized data
text
low address
command-line arguments and environment variables
0x7fff957b146c
0x7fba578c36a0 shared memory of 100,000 bytes
0x7fba578ab000
0x0000009fb6b0 malloc of 100,000 bytes
0x0000009e3010
0x00000060bd00 array[] of 40,000 bytes 0x0000006020c0

Figure 15.32 Memory layout on an Intel-based Linux system

Recall that the `mmap` function (Section 14.8) can be used to map portions of a file into the address space of a process. This is conceptually similar to attaching a shared memory segment using the `shmat` XSI IPC function. The main difference is that the memory segment mapped with `mmap` is backed by a file, whereas no file is associated with an XSI shared memory segment.

Example — Memory Mapping of `/dev/zero`

Shared memory can be used between unrelated processes. But if the processes are related, some implementations provide a different technique.

The following technique works on FreeBSD 8.0, Linux 3.2.0, and Solaris 10. Mac OS X 10.6.8 currently doesn't support the mapping of character devices into the address space of a process.

The device /dev/zero is an infinite source of 0 bytes when read. This device also accepts any data that is written to it, ignoring the data. Our interest in this device for IPC arises from its special properties when it is memory mapped.

- An unnamed memory region is created whose size is the second argument to mmap, rounded up to the nearest page size on the system.
- The memory region is initialized to 0.
- Multiple processes can share this region if a common ancestor specifies the MAP_SHARED flag to mmap.

The program in Figure 15.33 is an example that uses this special device.

```
#include "apue.h"
#include <fcntl.h>
#include <sys/mman.h>

#define NLOOPS      1000
#define SIZE  sizeof(long) /* size of shared memory area */ static
int update(long *ptr)
{ return((*ptr)++); /* return value before increment */
} int
main(void)
{
    int       fd, i,
    counter; pid_t pid;
    void *area;

    if ((fd = open("/dev/zero", O_RDWR)) < 0) err_sys("open
    error"); if ((area = mmap(0, SIZE, PROT_READ | PROT_WRITE,
    MAP_SHARED,
      fd, 0)) == MAP_FAILED)
        err_sys("mmap error");
    close(fd);      /* can close /dev/zero now that it's mapped */
    TELL_WAIT(); if ((pid = fork()) < 0) { err_sys("fork error");
    } else if (pid > 0) {  /* parent */ for (i = 0; i < NLOOPS; i +=
        2) { if ((counter = update((long *)area)) != i)
        err_quit("parent: expected %d, got %d", i, counter);

            TELL_CHILD(pid);
            WAIT_CHILD();
        }
    } else {  /* child */ for (i = 1; i < NLOOPS +
        1; i += 2) {
            WAIT_PARENT();

            if ((counter = update((long *)area)) != i)
                err_quit("child: expected %d, got %d", i, counter);
            TELL_PARENT(getppid());
        }
    }

    exit(0);
}
```

Figure 15.33 IPC between parent and child using memory mapped I/O of `/dev/zero`

The program opens the `/dev/zero` device and calls `mmap`, specifying a size of a long integer. Note that once the region is mapped, we can `close` the device. The process then creates a child. Since `MAP_SHARED` was specified in the call to `mmap`, writes to the memory-mapped region by one process are seen by the other process. (If we had specified `MAP_PRIVATE` instead, this example wouldn't work.)

The parent and the child then alternate running, incrementing a long integer in the shared memory-mapped region, using the synchronization functions from Section 8.9. The memory-mapped region is initialized to 0 by `mmap`. The parent increments it to 1, then the child increments it to 2, then the parent increments it to 3, and so on. Note that we have to use parentheses when we increment the value of the long integer in the `update` function, since we are incrementing the value and not the pointer.

The advantage of using `/dev/zero` in the manner that we've shown is that an actual file need not exist before we call `mmap` to create the mapped region. Mapping `/dev/zero` automatically creates a mapped region of the specified size. The disadvantage of this technique is that it works only between related processes. With related processes, however, it is probably simpler and more efficient to use threads (Chapters 11 and 12). Note that no matter which technique is used, we still need to synchronize access to the shared data. □

Example — Anonymous Memory Mapping

Many implementations provide anonymous memory mapping, a facility similar to the `/dev/zero` feature. To use this facility, we specify the `MAP_ANON` flag to `mmap` and specify the file descriptor as −1. The resulting region is anonymous (since it's not associated with a pathname through a file descriptor) and creates a memory region that can be shared with descendant processes.

The anonymous memory-mapping facility is supported by all four platforms discussed in this text. Note, however, that Linux defines the `MAP_ANONYMOUS` flag for this facility, but defines the `MAP_ANON` flag to be the same value for improved application portability.

To modify the program in Figure 15.33 to use this facility, we make three changes: (a) remove the `open` of `/dev/zero`, (b) remove the `close` of `fd`, and (c) change the call to `mmap` to the following:

```
    if ((area = mmap(0, SIZE, PROT_READ | PROT_WRITE,
                     MAP_ANON | MAP_SHARED, -1, 0)) == MAP_FAILED)
```

In this call, we specify the `MAP_ANON` flag and set the file descriptor to −1. The rest of the program from Figure 15.33 is unchanged. □

The last two examples illustrate sharing memory among multiple related processes. If shared memory is required between unrelated processes, there are two alternatives. Applications can use the XSI shared memory functions, or they can use `mmap` to map the same file into their address spaces using the `MAP_SHARED` flag.

15.10 POSIX Semaphores

The POSIX semaphore mechanism is one of three IPC mechanisms that originated with the real-time extensions to POSIX.1. The Single UNIX Specification placed the three mechanisms (message queues, semaphores, and shared memory) in option classes. Prior to SUSv4, the POSIX semaphore interfaces were included in the semaphores option. In SUSv4, these interfaces were moved to the base specification, but the message queue and shared memory interfaces remained optional.

The POSIX semaphore interfaces were meant to address several deficiencies with the XSI semaphore interfaces:

- The POSIX semaphore interfaces allow for higher-performance implementations compared to XSI semaphores.
- The POSIX semaphore interfaces are simpler to use: there are no semaphore sets, and several of the interfaces are patterned after familiar file system operations. Although there is no requirement that they be implemented in the file system, some systems do take this approach.
- The POSIX semaphores behave more gracefully when removed. Recall that when an XSI semaphore is removed, operations using the same semaphore identifier fail with `errno` set to `EIDRM`. With POSIX semaphores, operations continue to work normally until the last reference to the semaphore is released.

POSIX semaphores are available in two flavors: named and unnamed. They differ in how they are created and destroyed, but otherwise work the same. Unnamed semaphores exist in memory only and require that processes have access to the memory to be able to use the semaphores. This means they can be used only by threads in the same process or threads in different processes that have mapped the same memory extent into their address spaces. Named semaphores, in contrast, are accessed by name and can be used by threads in any processes that know their names.

To create a new named semaphore or use an existing one, we call the `sem_open` function.

```
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag, ... /* mode_t mode,
                unsigned int value */ );
```

Returns: Pointer to semaphore if OK, `SEM_FAILED` on error

When using an existing named semaphore, we specify only two arguments: the name of the semaphore and a zero value for the *oflag* argument. When the *oflag* argument has the `O_CREAT` flag set, we create a new named semaphore if it does not yet exist. If it already exists, it is opened for use, but no additional initialization takes place.

When we specify the `O_CREAT` flag, we need to provide two additional arguments. The *mode* argument specifies who can access the semaphore. It can take on the same values as the permission bits for opening a file: user-read, user-write, user-execute, group-read, group-write,

group-execute, other-read, other-write, and other-execute. The resulting permissions assigned to the semaphore are modified by the caller's file
creation mask (Sections 4.5 and 4.8). Note, however, that only read and write access matter, but the interfaces don't allow us to specify the mode when we open an existing semaphore. Implementations usually open semaphores for both reading and writing.

The *value* argument is used to specify the initial value for the semaphore when we create it. It can take on any value from 0 to `SEM_VALUE_MAX` (Figure 2.9).

If we want to ensure that we are creating the semaphore, we can set the *oflag* argument to `O_CREAT|O_EXCL`. This will cause `sem_open` to fail if the semaphore already exists.

To promote portability, we must follow certain conventions when selecting a semaphore name.

- The first character in the name should be a slash (/). Although there is no requirement that an implementation of POSIX semaphores uses the file system, if the file system *is* used, we want to remove any ambiguity as to the starting point from which the name is interpreted.
- The name should contain no other slashes to avoid implementation-defined behavior. For example, if the file system is used, the names `/mysem` and `//mysem` would evaluate to the same filename, but if the implementation doesn't use the file system, the two names could be treated as different (consider what would happen if the implementation hashed the name to an integer value used to identify the semaphore).
- The maximum length of the semaphore name is implementation defined. The name should be no longer than `_POSIX_NAME_MAX` (Figure 2.8) characters, because this is the minimum acceptable limit to the maximum name length if the implementation does use the file system.

The `sem_open` function returns a semaphore pointer that we can pass to other semaphore functions when we want to operate on the semaphore. When we are done with the semaphore, we can call the `sem_close` function to release any resources associated with the semaphore.

```
#include <semaphore.h>

int sem_close(sem_t

*sem);
```

Returns: 0 if OK, −1 on error

If our process exits without having first called `sem_close`, the kernel will close any open semaphores automatically. Note that this doesn't affect the state of the semaphore value — if we have incremented its value, this doesn't change just because we exit. Similarly, if we call `sem_close`, the semaphore value is unaffected. There is no mechanism equivalent to the `SEM_UNDO` flag found with XSI semaphores.

To destroy a named semaphore, we can use the `sem_unlink` function.

```
#include <semaphore.h> int sem_unlink(const char *name);
```

Returns: 0 if OK, −1 on error

The `sem_unlink` function removes the name of the semaphore. If there are no open references to the semaphore, then it is destroyed. Otherwise, destruction is deferred until the last open reference is closed.

Unlike with XSI semaphores, we can only adjust the value of a POSIX semaphore by one with a single function call. Decrementing the count is analogous to locking a binary semaphore or acquiring a resource associated with a counting semaphore.

> Note that there is no distinction of semaphore type with POSIX semaphores. Whether we use a binary semaphore or a counting semaphore depends on how we initialize and use the semaphore. If a semaphore only ever has a value of 0 or 1, then it is a binary semaphore. When a binary semaphore has a value of 1, we say that it is ''unlocked;'' when it has a value of 0, we say that it is ''locked.''

To decrement the value of a semaphore, we can use either the `sem_wait` or `sem_trywait` function.

```
#include <semaphore.h> int

sem_trywait(sem_t *sem);

int sem_wait(sem_t *sem);
```

Both return: 0 if OK, −1 on error

With the `sem_wait` function, we will block if the semaphore count is 0. We won't return until we have successfully decremented the semaphore count or are interrupted by a signal. We can use the `sem_trywait` function to avoid blocking. If the semaphore count is zero when we call `sem_trywait`, it will return −1 with `errno` set to `EAGAIN` instead of blocking.

A third alternative is to block for a bounded amount of time. We can use the `sem_timedwait` function for this purpose.

```
#include <semaphore.h>
#include <time.h>

int sem_timedwait(sem_t *restrict sem,
                  const struct timespec *restrict tsptr);
```

Returns: 0 if OK, −1 on error

The *tsptr* argument specifies the absolute time when we want to give up waiting for the semaphore. The timeout is based on the `CLOCK_REALTIME` clock (recall Figure 6.8). If the semaphore can be decremented immediately, then the value of the timeout doesn't matter — even though it might specify some time in the past, the attempt to decrement the semaphore will still succeed. If the timeout expires without being able to decrement the semaphore count, then `sem_timedwait` will return −1 with `errno` set to `ETIMEDOUT`.

To increment the value of a semaphore, we call the `sem_post` function. This is analogous to unlocking a binary semaphore or releasing a resource associated with a counting semaphore.

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

Returns: 0 if OK, −1 on error

If a process is blocked in a call to `sem_wait` (or `sem_timedwait`) when we call `sem_post`, the process is awakened and the semaphore count that was just incremented by `sem_post` is decremented by `sem_wait` (or `sem_timedwait`).

When we want to use POSIX semaphores within a single process, it is easier to use unnamed semaphores. This only changes the way we create and destroy the semaphore. To create an unnamed semaphore, we call the `sem_init` function.

```
#include <semaphore.h> int sem_init(sem_t *sem, int pshared, unsigned

int value);

                                                  Returns: 0 if OK, −1 on error
```

The *pshared* argument indicates if we plan to use the semaphore with multiple processes. If so, we set it to a nonzero value. The *value* argument specifies the initial value of the semaphore.

Instead of returning a pointer to the semaphore like `sem_open` does, we need to declare a variable of type `sem_t` and pass its address to `sem_init` for initialization. If we plan to use the semaphore between two processes, we need to ensure that the *sem* argument points into the memory extent that is shared between the processes.

When we are done using the unnamed semaphore, we can discard it by calling the `sem_destroy` function.

```
  #include <semaphore.h> int

  sem_destroy(sem_t *sem);

                                                  Returns: 0 if OK, −1 on error
```

After calling `sem_destroy`, we can't use any of the semaphore functions with *sem* unless we reinitialize it by calling `sem_init` again.

One other function is available to allow us to retrieve the value of a semaphore. We call the `sem_getvalue` function for this purpose.

```
#include <semaphore.h> int sem_getvalue(sem_t *restrict sem, int

*restrict valp);

                                                  Returns: 0 if OK, −1 on error
```

On success, the integer pointed to by the *valp* argument will contain the value of the semaphore. Be aware, however, that the value of the semaphore can change by the time that we try to use the value we just read. Unless we use additional synchronization mechanisms to avoid this race, the `sem_getvalue` function is useful only for debugging.

The `sem_getvalue` function is not supported by Mac OS X 10.6.8.

Example

One of the motivations for introducing the POSIX semaphore interfaces was that they can be made to perform significantly better than the existing XSI semaphore interfaces. It is instructive to see if this goal was reached in existing systems, even though these systems were not designed to support real-time applications.

In Figure 15.34, we compare the performance of using XSI semaphores (without `SEM_UNDO`) and POSIX semaphores when 3 processes compete to allocate and release the semaphore 1,000,000 times on two platforms (Linux 3.2.0 and Solaris 10).

Operation	Solaris 10			Linux 3.2.0		
	User	System	Clock	User	System	Clock
XSI semaphores	11.85	15.85	27.91	0.33	5.93	7.33
POSIX semaphores	13.72	10.52	24.44	0.26	0.75	0.41

Figure 15.34 Timing comparison of semaphore implementations

In Figure 15.34, we can see that POSIX semaphores provide only a 12% improvement over XSI semaphores on Solaris, but on Linux the improvement is 94% (almost 18 times faster)! If we trace the programs, we find that the Linux implementation of POSIX semaphores maps the file into the process address space and performs individual semaphore operations without using system calls. □

Example

Recall from Figure 12.5 that the Single UNIX Specification doesn't define what happens when one thread locks a normal mutex and a different thread tries to unlock it, but that error-checking mutexes and recursive mutexes generate errors in this case. Because a binary semaphore can be used like a mutex, we can use a semaphore to create our own locking primitive to provide mutual exclusion.

Assuming we were to create our own lock that could be locked by one thread and unlocked by another, our lock structure might look like

```
struct slock {
  sem_t *semp;
  char name[_POSIX_NAME_MAX];
};
```

The program in Figure 15.35 shows an implementation of a semaphore-based mutual exclusion primitive.

```
#include "slock.h"
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

struct slock *

s_alloc()
{ struct slock *sp; static
    int cnt;

    if ((sp = malloc(sizeof(struct slock))) == NULL) return(NULL); do {
    snprintf(sp->name, sizeof(sp->name), "/%ld.%d", (long)getpid(),
          cnt++);
```

```
        sp->semp = sem_open(sp->name, O_CREAT|O_EXCL, S_IRWXU, 1); }
    while ((sp->semp == SEM_FAILED) && (errno == EEXIST)); if (sp->semp
    == SEM_FAILED) { free(sp); return(NULL);
    }
    sem_unlink(sp->name);
    return(sp);
} void

s_free(struct slock *sp)
{ sem_close(sp->semp);
    free(sp);
} int

s_lock(struct slock *sp)
{ return(sem_wait(sp->semp));
} int

s_trylock(struct slock *sp)
{ return(sem_trywait(sp->semp));
} int

s_unlock(struct slock *sp)
{ return(sem_post(sp->semp));
}
```

Figure 15.35 Mutual exclusion using a POSIX semaphore

We create a name based on the process ID and a counter. We don't bother to protect the counter with a mutex, because if two racing threads call `s_alloc` at the same time and end up with the same name, using the `O_EXCL` flag in the call to `sem_open` will cause one to succeed and one to fail with `errno` set to `EEXIST`, so we just retry if this happens. Note that we unlink the semaphore after opening it. This destroys the name so that no other process can access it and simplifies cleanup when the process ends. □

_

15.11 Client–Server Proper ties

Let’s detail some of the properties of clients and servers that are affected by the various types of IPC used between them. The simplest type of relationship is to have the client `fork` and `exec` the desired server. Two half-duplex pipes can be created before the `fork` to allow data to be transferred in both directions. Figure 15.16 is an example of this arrangement. The server that is executed can be a set-user-ID program, giving it special privileges. Also, the server can determine the real identity of the client by looking at its real user ID. (Recall from Section 8.10 that the real user ID and real group
ID don’t change across an `exec`.)

With this arrangement, we can build an *open server*. (We show an implementation of this client–server mechanism in Section 17.5.) It opens files for the client instead of the client calling the `open` function. This way, additional permission checking can be added, above and beyond the normal UNIX system user/group/other permissions. We assume that the server is a set-user-ID program, giving it additional permissions (root permission, perhaps). The server uses the real user ID of the client to determine whether to give it access to the requested file. This way, we can build a server that allows certain users permissions that they don’t normally have.

In this example, since the server is a child of the parent, all the server can do is pass back the contents of the file to the parent. Although this works fine for regular files, it can’t be used for special device files, for example. We would like to be able to have the server open the requested file and pass back the file descriptor. Whereas a parent can pass a child an open descriptor, a child cannot pass a descriptor back to the parent (unless special programming techniques are used, which we cover in Chapter 17).

We showed the next type of server in Figure 15.23. The server is a daemon process that is contacted using some form of IPC by all clients. We can’t use pipes for this type of client–server arrangement. A form of named IPC is required, such as FIFOs or message queues. With FIFOs, we saw that an individual per-client FIFO is also required if the server is to send data back to the client. If the client–server application sends data only from the client to the server, a single well-known FIFO suffices. (The System V line printer spooler used this form of client–server arrangement. The client was the `lp`(1) command, and the server was the `lpsched` daemon process. A single FIFO was used, since the flow of data was only from the client to the server. Nothing was sent back to the client.)

Multiple possibilities exist with message queues.

1. A single queue can be used between the server and all the clients, using the typefield of each message to indicate the message recipient. For example, the clients can send their requests with a type field of 1. Included in the request must be the client’s process ID. The server then sends the response with the type field set to the client’s process ID. The server receives only the messages with a type field of 1 (the fourth argument for

msgrcv), and the clients receive only the messages with a type field equal to their process IDs.

2. Alternatively, an individual message queue can be used for each client. Beforesending the first request to a server, each client creates its own message queue with a key of IPC_PRIVATE. The server also has its own queue, with a key or identifier known to all clients. The client sends its first request to the server's well-known queue, and this request must contain the message queue ID of the client's queue. The server sends its first response to the client's queue, and all future requests and responses are exchanged on this queue.

One problem with this technique is that each client-specific queue usually has only a single message on it: a request for the server or a response for a client. This seems wasteful of a limited systemwide resource (a message queue), and a FIFO can be used instead. Another problem is that the server has to read messages from multiple queues. Neither select nor poll works with message queues.

Either of these two techniques using message queues can be implemented using shared memory segments and a synchronization method (a semaphore or record locking).

The problem with this type of client–server relationship (the client and the server being unrelated processes) is for the server to identify the client accurately. Unless the server is performing a nonprivileged operation, it is essential that the server know who the client is. This is required, for example, if the server is a set-user-ID program. Although all these forms of IPC go through the kernel, there is no facility provided by them to have the kernel identify the sender.

With message queues, if a single queue is used between the client and the server (so that only a single message is on the queue at a time, for example), the msg_lspid of the queue contains the process ID of the other process. But when writing the server, we want the effective user ID of the client, not its process ID. There is no portable way to obtain the effective user ID, given the process ID. (Naturally, the kernel maintains both values in the process table entry, but other than rummaging around through the kernel's memory, we can't obtain one, given the other.)

We'll use the following technique in Section 17.2 to allow the server to identify the client. The same technique can be used with FIFOs, message queues, semaphores, and shared memory. For the following description, assume that FIFOs are being used, as in Figure 15.23. The client must create its own FIFO and set the file access permissions of the FIFO so that only user-read and user-write are on. We assume that the server has superuser privileges (or else it probably wouldn't care about the client's true identity), so the server can still read and write to this FIFO. When the server receives the client's first request on the server's well-known FIFO (which must contain the identity of the client-specific FIFO), the server calls either stat or fstat on the client-specific FIFO. The server assumes that the effective user ID of the client is the owner of the FIFO (the st_uid field of the stat structure). The server verifies that only the user-read and user-write permissions are enabled. As another check, the server should look at the three times associated with the FIFO (the st_atime, st_mtime, and st_ctime fields of the stat structure) to verify that they are recent (no older than 15 or 30 seconds, for example). If a malicious client can create a FIFO with someone else as the owner and set the file's permission bits to user-read and user-write only, then the system has other fundamental security problems.

To use this technique with XSI IPC, recall that the ipc_perm structure associated with each message queue, semaphore, and shared memory segment identifies the creator of the IPC structure (the cuid and cgid fields). As with the example using FIFOs, the server should require the client to create the IPC structure and have the client set the access permissions to user-read and user-write only. The times associated with the IPC structure should also be verified by the server to be recent (since these IPC structures hang around until explicitly deleted).

We'll see in Section 17.3 that a far better way of doing this authentication is for the kernel to provide the effective user ID and effective group ID of the client. This is done by the socket subsystem when file descriptors are passed between processes.

15.12 Summary

We've detailed numerous forms of interprocess communication: pipes, named pipes (FIFOs), the three forms of IPC commonly called XSI IPC (message queues, semaphores, and shared memory), and an alternative semaphore mechanism provided by POSIX. Semaphores are really a synchronization primitive, not true IPC, and are often used to synchronize access to a shared resource, such as a shared memory segment. With pipes, we looked at the implementation of the popen function, at coprocesses, and at the pitfalls that can be encountered with the standard I/O library's buffering.

After comparing the timing of message queues versus full-duplex pipes, and semaphores versus record locking, we can make the following recommendations: learn pipes and FIFOs, since these two basic techniques can still be used effectively in numerous applications. Avoid using message queues and semaphores in any new applications. Full-duplex pipes and record locking should be considered instead, as they are far simpler. Shared memory still has its use, although the same functionality can be provided through the use of the mmap function (Section 14.8).

In the next chapter, we will look at network IPC, which can allow processes to communicate across machine boundaries.

Exercises

15.1 In the program shown in Figure 15.6, remove the close right before the waitpid at the end of the parent code. Explain what happens.

15.2 In the program in Figure 15.6, remove the waitpid at the end of the parent code. Explain what happens.

15.3 What happens if the argument to popen is a nonexistent command? Write a small program to test this.

15.4 In the program shown in Figure 15.18, remove the signal handler, execute the program, and then terminate the child. After entering a line of input, how can you tell that the parent was terminated by `SIGPIPE`?

15.5 In the program in Figure 15.18, use the standard I/O library for reading and writing the pipes instead of `read` and `write`.

17

Advanced IPC

17.1 Introduction

In the previous two chapters, we discussed various forms of IPC, including pipes and sockets. In this chapter, we look at an advanced form of IPC—the UNIX domain socket mechanism — and see what we can do with it. With this form of IPC, we can pass open file descriptors between processes running on the same computer system, server processes can associate names with their file descriptors, and client processes running on the same system can use these names to rendezvous with the servers. We'll also see how the operating system provides a unique IPC channel per client.

17.2 UNIX Domain Sockets

UNIX domain sockets are used to communicate with processes running on the same machine. Although Internet domain sockets can be used for this same purpose, UNIX domain sockets are more efficient. UNIX domain sockets only copy data; they have no protocol processing to perform, no network headers to add or remove, no checksums to calculate, no sequence numbers to generate, and no acknowledgements to send.

UNIX domain sockets provide both stream and datagram interfaces. The UNIX domain datagram service is reliable, however. Messages are neither lost nor delivered out of order. UNIX domain sockets are like a cross between sockets and pipes. You can use the network-oriented socket interfaces with them, or you can use the `socketpair` function to create a pair of unnamed, connected, UNIX domain sockets.

```
#include <sys/socket.h> int socketpair(int domain, int type, int
protocol, int sockfd[2]);
```

Returns: 0 if OK, −1 on error

Although the interface is sufficiently general to allow `socketpair` to be used with other domains, operating systems typically provide support only for the UNIX domain.

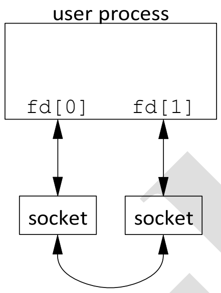


Figure 17.1 A socket pair

A pair of connected UNIX domain sockets acts like a full-duplex pipe: both ends are open for reading and writing (see Figure 17.1). We'll refer to these as ''fd-pipes'' to distinguish them from normal, half-duplex pipes.

Example —`fd_pipe` Function

Figure 17.2 shows the `fd_pipe` function, which uses the `socketpair` function to create a pair of connected UNIX domain stream sockets.

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

/*
 * Returns a full-duplex pipe (a UNIX domain socket) with
 * the two file descriptors returned in fd[0] and fd[1].
 */
int
fd_pipe(int fd[2])
{ return(socketpair(AF_UNIX, SOCK_STREAM, 0, fd));
}
```

Figure 17.2 Creating a full-duplex pipe

Some BSD-based systems use UNIX domain sockets to implement pipes. But when `pipe` is called, the write end of the first descriptor and the read end of the second descriptor are both closed. To get a full-duplex pipe, we must call `socketpair` directly.

Example — Polling XSI Message Queues with the Help of UNIX Domain Sockets

In Section 15.6.4, we said one of the problems with using XSI message queues is that we can't use `poll` or `select` with them, because they aren't associated with file descriptors. However, sockets *are* associated with file descriptors, and we can use them to notify us when messages arrive. We'll use one thread per message queue. Each thread will block in a call to `msgrcv`. When a message arrives, the thread will write it down one end of a UNIX domain socket. Our application will use the other end of the socket to receive the message when `poll` indicates data can be read from the socket.

The program in Figure 17.3 illustrates this technique. The `main` function creates the message queues and UNIX domain sockets and starts one thread to service each queue. Then it uses an infinite loop to poll one end of the sockets. When a socket is readable, it reads from the socket and writes the message on the standard output.

```
#include "apue.h"
#include <poll.h>
#include <pthread.h>
#include <sys/msg.h>
#include <sys/socket.h>

#define NQ      3       /* number of queues */
#define MAXMSZ 512  /* maximum message size */ #define
KEY    0x123 /* key for first message queue */ struct
threadinfo { int qid; int fd;
};

struct mymesg { long
    mtype; char
    mtext[MAXMSZ];
};

void *

helper(void *arg)
{
    int       n; struct mymesg
       m;
    struct threadinfo *tip = arg;
    for(;;) { memset(&m, 0,
    sizeof(m));
        if ((n = msgrcv(tip->qid, &m, MAXMSZ, 0, MSG_NOERROR)) <
        0) err_sys("msgrcv error"); if (write(tip->fd, m.mtext, n)
        < 0) err_sys("write error");
```

```
          }
       } int
       main()
{
    int                  i, n, err;
    int                  fd[2];
    int                  qid[NQ];
    struct pollfd        pfd[NQ];
    struct threadinfo ti[NQ];
    pthread_t       tid[NQ]; char
       buf[MAXMSZ];

    for (i = 0; i < NQ; i++) { if ((qid[i] = msgget((KEY+i),
        IPC_CREAT|0666)) < 0) err_sys("msgget error");
        printf("queue ID %d is %d\n", i, qid[i]); if
        (socketpair(AF_UNIX, SOCK_DGRAM, 0, fd) < 0)
        err_sys("socketpair error"); pfd[i].fd = fd[0];
        pfd[i].events = POLLIN; ti[i].qid = qid[i]; ti[i].fd =
        fd[1];
        if ((err = pthread_create(&tid[i], NULL, helper, &ti[i])) != 0)
            err_exit(err, "pthread_create error");
    } for (;;) { if (poll(pfd, NQ, -1) < 0) err_sys("poll error"); for
    (i = 0; i < NQ; i++) { if (pfd[i].revents & POLLIN) { if ((n =
    read(pfd[i].fd, buf, sizeof(buf))) < 0) err_sys("read error");
    buf[n] = 0;
                printf("queue id %d, message %s\n", qid[i], buf);
            }
        }
    } exit(0);
}
```

Figure 17.3 Poll for XSI messages using UNIX domain sockets

Note that we use datagram (SOCK_DGRAM) sockets instead of stream sockets. This allows us to retain message boundaries so when we read from the socket, we read only one message at a time.

This technique allows us to use either poll or select (indirectly) with message queues. As long as the costs of one thread per queue and copying each message two extra times (once to write it to the socket and once to read it from the socket) are acceptable, this technique will make it easier to use XSI message queues.

Well use the program shown in Figure 17.4 to send messages to our test program from Figure 17.3.

```
#include "apue.h"
#include <sys/msg.h>
#define MAXMSZ 512
struct mymesg { long
mtype; char
mtext[MAXMSZ];
};
int
main(int argc, char *argv[])
{ key_t key; long qid; size_t nbytes; struct mymesg m;
    if (argc != 3) { fprintf(stderr, "usage: sendmsg KEY
    message\n");
        exit(1);
    }
    key = strtol(argv[1], NULL, 0); if ((qid =
    msgget(key, 0)) < 0) err_sys("can't open queue
    key %s", argv[1]); memset(&m, 0, sizeof(m));
    strncpy(m.mtext, argv[2], MAXMSZ-1);
    nbytes = strlen(m.mtext);
    m.mtype = 1;
    if (msgsnd(qid, &m, nbytes, 0) < 0)
        err_sys("can't send message");
    exit(0);
}
```

Figure 17.4 Post a message to an XSI message queue

This program takes two arguments: the key associated with the queue and a string to be sent as the body of the message. When we send messages to the server, it prints them as shown below.

```
$ ./pollmsg &                              run the server in the background
[1] 12814
$ queue ID 0 is 196608
queue ID 1 is 196609
queue ID 2 is 196610
$ ./sendmsg 0x123 "hello, world"           send a message to the first queue
queue id 196608, message hello,
world
$ ./sendmsg 0x124 "just a test"            send a message to the second queue
queue id 196609, message just a
test
$ ./sendmsg 0x125 "bye"                    send a message to the third queue
queue id 196610, message bye
```

□

17.2.1 Naming UNIX Domain Sockets

Although the `socketpair` function creates sockets that are connected to each other, the individual sockets don't have names. This means that they can't be addressed by unrelated processes.

In Section 16.3.4, we learned how to bind an address to an Internet domain socket. Just as with Internet domain sockets, UNIX domain sockets can be named and used to advertise services. The address format used with UNIX domain sockets differs from that used with Internet domain sockets, however.

Recall from Section 16.3 that socket address formats differ from one implementation to the next. An address for a UNIX domain socket is represented by a `sockaddr_un` structure. On Linux 3.2.0 and Solaris 10, the `sockaddr_un` structure is defined in the header `<sys/un.h>` as

```
struct sockaddr_un { sa_family_t sun_family;
     /* AF_UNIX */
     char          sun_path[108]; /* pathname */
};
```

On FreeBSD 8.0 and Mac OS X 10.6.8, however, the `sockaddr_un` structure is defined as

```
struct sockaddr_un { unsigned char sun_len; /* sockaddr
     length */ sa_family_t sun_family; /* AF_UNIX */
     char            sun_path[104]; /* pathname */
};
```

The `sun_path` member of the `sockaddr_un` structure contains a pathname. When we bind an address to a UNIX domain socket, the system creates a file of type `S_IFSOCK` with the same name.

This file exists only as a means of advertising the socket name to clients. The file can't be opened or otherwise used for communication by applications.

If the file already exists when we try to bind the same address, the `bind` request will fail. When we close the socket, this file is not automatically removed, so we need to make sure that we unlink it before our application exits.

Example

The program in Figure 17.5 shows an example of binding an address to a UNIX domain socket.

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>

int
main(void)
{ int fd, size;
```

```
     struct sockaddr_un un;

     un.sun_family = AF_UNIX;
     strcpy(un.sun_path, "foo.socket");
     if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) err_sys("socket
     failed"); size = offsetof(struct sockaddr_un, sun_path) +
     strlen(un.sun_path);
     if (bind(fd, (struct sockaddr *)&un, size) < 0)
     err_sys("bind failed"); printf("UNIX domain
     socket bound\n");
     exit(0);
}
```

Figure 17.5 Binding an address to a UNIX domain socket

When we run this program, the `bind` request succeeds. If we run the program a second time, however, we get an error, because the file already exists. The program won't succeed again until we remove the file.

```
$ ./a.out                                          run the program
UNIX domain socket bound
$ ls -l foo.socket       look at the socket file srwxr-xr-x 1 sar   0 May 18
00:44 foo.socket
$ ./a.out try to run the program again bind failed: Address already in use
$ rm foo.socket                                    remove the socket file
$ ./a.out                                            run the program a third time
UNIX domain socket bound                           now it succeeds
```

The way we determine the size of the address to bind is to calculate the offset of the `sun_path` member in the `sockaddr_un` structure and add to it the length of the pathname, not including the terminating null byte. Since implementations vary in which members precede `sun_path` in the `sockaddr_un` structure, we use the `offsetof` macro from `<stddef.h>` (included by `apue.h`) to calculate the offset of the `sun_path` member from the start of the structure. If you look in `<stddef.h>`, you'll see a definition similar to the following:

```
#define offsetof(TYPE, MEMBER) ((int)&((TYPE *)0)->MEMBER)
```

The expression evaluates to an integer, which is the starting address of the member, assuming that the structure begins at address 0. □

17.3 Unique Connections

A server can arrange for unique UNIX domain connections to clients using the standard `bind`, `listen`, and `accept` functions. Clients use `connect` to contact the server; after the connect request is accepted by the server, a unique connection exists between the client and the server. This style of operation is the same that we illustrated with Internet domain sockets in Figures 16.16 and 16.17.

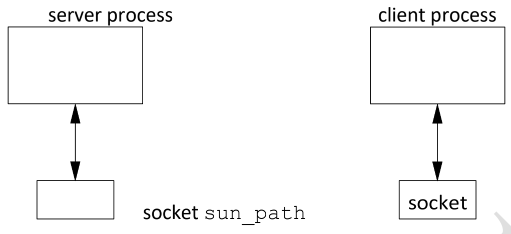


Figure 17.6 Client and server sockets before a `connect`

Figure 17.6 shows a client process and a server process before a connection exists between the two. The server has bound its socket to a `sockaddr_un` address and is listening for connection requests. Figure 17.7 shows the unique connection between the client and server after the server has accepted the client's connection request.

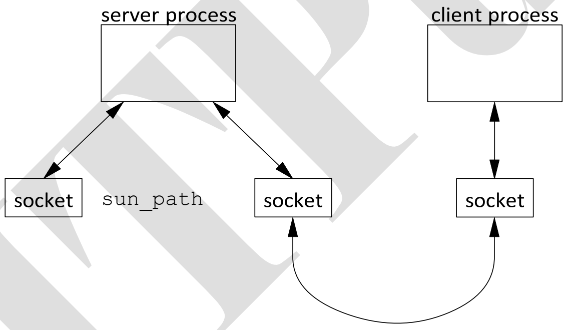


Figure 17.7 Client and server sockets after a `connect`

We will now develop three functions that can be used to create unique connections between unrelated processes running on the same machine. These functions mimic the connection-oriented socket functions discussed in Section 16.4. We use UNIX domain sockets for the underlying communication mechanism here.

```
#include "apue.h" int

serv_listen(const char *name);
```
Returns: file descriptor to listen on if OK, negative value on error

```
int serv_accept(int listenfd, uid_t *uidptr);
```
Returns: new file descriptor if OK, negative value on error

```
int cli_conn(const char *name);
```
Returns: file descriptor if OK, negative value on error

The `serv_listen` function (Figure 17.8) can be used by a server to announce its willingness to listen for client connect requests on a well-known name (some pathname in the file system). Clients will use this name when they want to connect to the server. The return value is the server's UNIX domain socket used to receive client connection requests.

The `serv_accept` function (Figure 17.9) is used by a server to wait for a client's connect request to arrive. When one arrives, the system automatically creates a new UNIX domain socket, connects it to the client's socket, and returns the new socket to the server. Additionally, the effective user ID of the client is stored in the memory to which *uidptr* points.

A client calls `cli_conn` (Figure 17.10) to connect to a server. The *name* argument specified by the client must be the same name that was advertised by the server's call to `serv_listen`. On return, the client gets a file descriptor connected to the server.

Figure 17.8 shows the `serv_listen` function.

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

       #define QLEN    10

/*
 * Create a server endpoint of a connection.
 * Returns fd if all OK, <0 on error.
 */ int
serv_listen(const char *name)
{
    int fd, len, err, rval; struct sockaddr_un un;

    if (strlen(name) >= sizeof(un.sun_path)) { errno =
        ENAMETOOLONG;
        return(-1);
    }

    /* create a UNIX domain stream socket */ if ((fd =
    socket(AF_UNIX, SOCK_STREAM, 0)) < 0) return(-2);
    unlink(name); /* in case it already exists */
    /* fill in socket address structure */
    memset(&un, 0, sizeof(un)); un.sun_family =
    AF_UNIX; strcpy(un.sun_path, name);
    len = offsetof(struct sockaddr_un, sun_path) + strlen(name);

    /* bind the name to the descriptor */ if (bind(fd,
    (struct sockaddr *)&un, len) < 0) { rval = -3;
        goto errout;
    }

    if (listen(fd, QLEN) < 0) { /* tell kernel we're a server */ rval
        = -4; goto errout;
    } return(fd);

errout:
    err = errno;
    close(fd); errno =
    err; return(rval);
}
```

Figure 17.8 The `serv_listen` function

First, we create a single UNIX domain socket by calling `socket`. We then fill in a `sockaddr_un` structure with the well-known pathname to be assigned to the socket. This structure is the argument to `bind`. Note that we don't need to set the `sun_len` field present on some platforms, because the operating system sets this for us, deriving it from the address length we pass to the `bind` function.

Finally, we call `listen` (Section 16.4) to tell the kernel that the process will be acting as a server awaiting connections from clients. When a connect request from a client arrives, the server calls the `serv_accept` function (Figure 17.9).

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

#define STALE 30 /* client's name can't be older than this (sec) */

/*
 * Wait for a client connection to arrive, and accept it.* We also
   obtain the client's user ID from the pathname * that it must bind
   before calling us.
 * Returns new fd if all OK, <0 on error
 */ int
serv_accept(int listenfd, uid_t *uidptr)
{
    int       clifd, err, rval; socklen_t
       len; time_t   staletime;
    struct sockaddr_un un;
    struct stat     statbuf; char
       *name;

    /* allocate enough space for longest name plus terminating null */
    if ((name = malloc(sizeof(un.sun_path + 1))) == NULL)
    return(-1); len = sizeof(un);
    if ((clifd = accept(listenfd, (struct sockaddr *)&un, &len)) < 0) {
        free(name);
               return(-2);     /* often errno=EINTR, if signal caught */
    }

    /* obtain the client's uid from its calling address */
    len -= offsetof(struct sockaddr_un, sun_path); /* len of pathname */
    memcpy(name, un.sun_path, len);
          name[len] = 0;           /* null terminate */
    if (stat(name, &statbuf) < 0) { rval
        = -3; goto errout;
    }

#ifdef S_ISSOCK     /* not defined for SVR4 */ if
    (S_ISSOCK(statbuf.st_mode) == 0) { rval = -4;
       /* not a socket */ goto errout;
    }
#endif if ((statbuf.st_mode & (S_IRWXG | S_IRWXO)) ||

        (statbuf.st_mode & S_IRWXU) != S_IRWXU) { rval = -
          5;  /* is not rwx------ */
          goto errout;
    }

    staletime = time(NULL) - STALE; if
    (statbuf.st_atime < staletime ||
    statbuf.st_ctime < staletime ||
    statbuf.st_mtime < staletime) {
                rval = -6;    /* i-node is too old */
          goto errout;
    }

    if (uidptr != NULL)
        *uidptr = statbuf.st_uid; /* return uid of caller */
unlink(name);       /* we're done with pathname now */ free(name);
return(clifd); errout:
    err = errno;
    close(clifd);
    free(name); errno =
    err; return(rval);
}
```

Figure 17.9 The serv_accept function

The server blocks in the call to accept, waiting for a client to call cli_conn. When accept returns, its return value is a brand-new descriptor that is connected to the client. Additionally, the pathname that the client assigned to its socket (the name that contained the client's process ID) is returned by accept, through the second argument (the pointer to the sockaddr_un structure). We copy this pathname and ensure that it is null terminated (if the pathname takes up all available space in the sun_path member of the sockaddr_un structure, there won't be room for the terminating null byte). Then we call stat to verify that the pathname is indeed a socket and that the permissions allow only user-read, user-write, and user-execute. We also verify that the three times associated with the socket are no older than 30 seconds. (Recall from Section 6.10 that the time function returns the current time and date in seconds past the Epoch.)

If all these checks are OK, we assume that the identity of the client (its effective user ID) is the owner of the socket. Although this check isn't perfect, it's the best we can do with current systems. (It would be better if the kernel returned the effective user ID to us through a parameter to accept.)

The client initiates the connection to the server by calling the cli_conn function (Figure 17.10).

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

       #define CLI_PATH     "/var/tmp/"
       #define CLI_PERM     S_IRWXU          /* rwx for user only */

/*
 * Create a client endpoint and connect to a server.
 * Returns fd if all OK, <0 on error.
 */ int
cli_conn(const char *name)
{
    int       fd, len, err, rval; struct sockaddr_un un,
    sun;
           int                  do_unlink = 0;

    if (strlen(name) >= sizeof(un.sun_path)) { errno =
        ENAMETOOLONG;
        return(-1);
    }

    /* create a UNIX domain stream socket */ if ((fd =
    socket(AF_UNIX, SOCK_STREAM, 0)) < 0) return(-1);

    /* fill socket address structure with our address */
    memset(&un, 0, sizeof(un)); un.sun_family =
    AF_UNIX;
    sprintf(un.sun_path, "%s%05ld", CLI_PATH, (long)getpid());
    len = offsetof(struct sockaddr_un, sun_path) + strlen(un.sun_path);

    unlink(un.sun_path);   /* in case it already exists */

    if (bind(fd, (struct sockaddr *)&un, len) < 0) { rval
        = -2; goto errout;
    }
    if (chmod(un.sun_path, CLI_PERM) < 0) { rval
        = -3; do_unlink = 1; goto errout;
    }

    /* fill socket address structure with server's address */
    memset(&sun, 0, sizeof(sun)); sun.sun_family =
    AF_UNIX; strcpy(sun.sun_path, name);
    len = offsetof(struct sockaddr_un, sun_path) + strlen(name); if
    (connect(fd, (struct sockaddr *)&sun, len) < 0) { rval = -4;
    do_unlink = 1; goto errout;
    } return(fd);

errout:
    err = errno; close(fd); if
    (do_unlink)
    unlink(un.sun_path); errno =
    err; return(rval);
}
```

Figure 17.10 The `cli_conn` function

We call `socket` to create the client's end of a UNIX domain socket. We then fill in a `sockaddr_un` structure with a client-specific name.

We don't let the system choose a default address for us, because the server would be unable to distinguish one client from another (if we don't explicitly bind a name to a UNIX domain socket, the kernel implicitly binds an address to it on our behalf and no file is created in the file system to represent the socket). Instead, we bind our own address — a step we usually don't take when developing a client program that uses sockets.

The last five characters of the pathname we bind are made from the process ID of the client. We call `unlink`, just in case the pathname already exists. We then call `bind` to assign a name to the client's socket. This creates a socket file in the file system with the same name as the bound pathname. We call `chmod` to turn off all permissions other than user-read, user-write, and user-execute. In `serv_accept`, the server checks these permissions and the user ID of the socket to verify the client's identity.

We then have to fill in another `sockaddr_un` structure, this time with the well-known pathname of the server. Finally, we call the `connect` function to initiate the connection with the server.

17.4 Passing File Descriptors

Passing an open file descriptor between processes is a powerful technique. It can lead to different ways of designing client–server applications. It allows one process (typically a server) to do everything that is required to open a file (involving such details as translating a network name to a network address, dialing a modem, and negotiating locks for the file) and simply pass back to the calling process a descriptor that can be used with all the I/O functions. All the details involved in opening the file or device are hidden from the client.

We must be more specific about what we mean by ‘‘passing an open file descriptor’’ from one process to another. Recall Figure 3.8, which showed two processes that have opened the same file. Although they share the same v-node, each process has its own file table entry.

When we pass an open file descriptor from one process to another, we want the passing process and the receiving process to share the same file table entry. Figure 17.11 shows the desired arrangement.

Technically, we are passing a pointer to an open file table entry from one process to another. This pointer is assigned the first available descriptor in the receiving process. (Saying that we are passing an open descriptor mistakenly gives the impression that the descriptor number in the receiving process is the same as in the sending process, which usually isn’t true.) Having two processes share an open file table is exactly what happens after a `fork` (recall Figure 8.2).

What normally happens when a descriptor is passed from one process to another is that the sending process, after passing the descriptor, then closes the descriptor. Closing the descriptor by the sender doesn’t really close the file or device, since the descriptor is still considered open by the receiving process (even if the receiver hasn’t specifically received the descriptor yet).

We define the following three functions that we use in this chapter to send and receive file descriptors. Later in this section, we’ll show the code for these three functions.

```
#include "apue.h" int send_fd(int fd, int

fd_to_send); int send_err(int fd, int status, const

char *errmsg);
```

Both return: 0 if OK, −1 on error

```
int recv_fd(int fd, ssize_t (*userfunc)(int, const void *, size_t));
```

Returns: file descriptor if OK, negative value on error

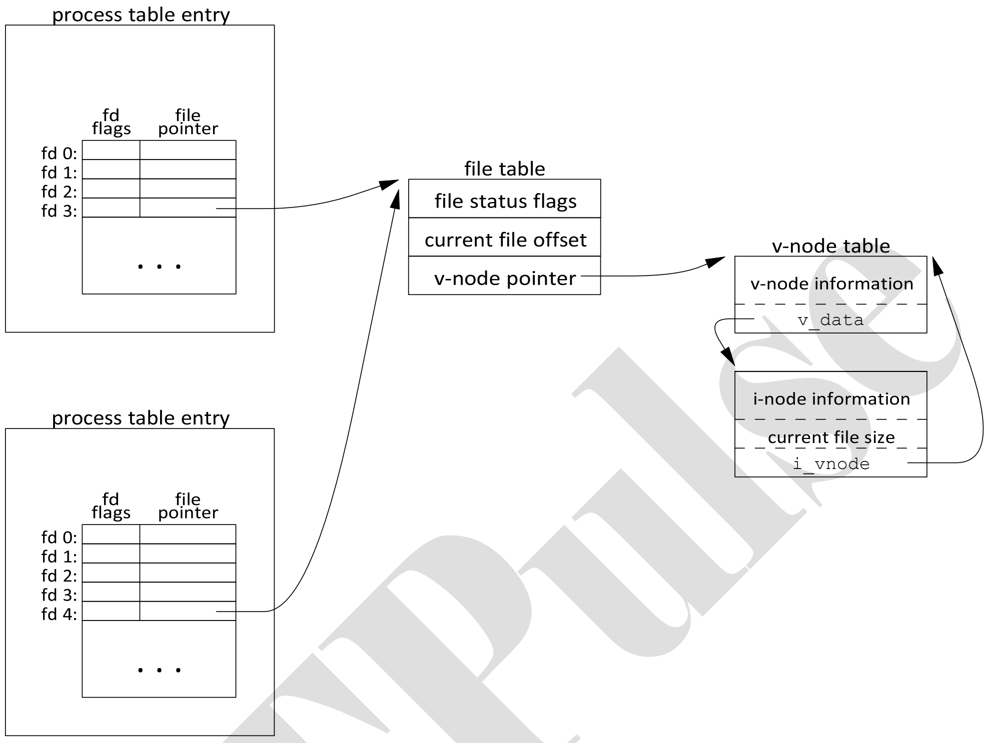


Figure 17.11 Passing an open file from the top process to the bottom process

A process (normally a server) that wants to pass a descriptor to another process calls either `send_fd` or `send_err`. The process waiting to receive the descriptor (the client) calls `recv_fd`.

The `send_fd` function sends the descriptor *fd_to_send* across using the UNIX domain socket represented by *fd*. The `send_err` function sends the *errmsg* using *fd*, followed by the *status* byte. The value of *status* must be in the range −1 through −255.

Clients call `recv_fd` to receive a descriptor. If all is OK (the sender called `send_fd`), the non-negative descriptor is returned as the value of the function. Otherwise, the value returned is the *status* that was sent by `send_err` (a negative value in the range −1 through −255). Additionally, if an error message was sent by the server, the client's *userfunc* is called to process the message. The first argument to *userfunc* is the constant `STDERR_FILENO`, followed by a pointer to the error message and its length. The return value from *userfunc* is the number of bytes written or a negative number on error. Often, the client specifies the normal `write` function as the *userfunc*.

We implement our own protocol that is used by these three functions. To send a descriptor, `send_fd` sends two bytes of 0, followed by the actual descriptor. To send an error, `send_err`

sends the *errmsg*, followed by a byte of 0, followed by the absolute value of the *status* byte (1 through 255). The recv_fd function reads everything on the
socket until it encounters a null byte. Any characters read up to this point are passed to the caller's *userfunc*. The next byte read by recv_fd is the status byte. If the status byte is 0, a descriptor was passed; otherwise, there is no descriptor to receive.

The function send_err calls the send_fd function after writing the error message to the socket. This is shown in Figure 17.12.

```
#include "apue.h"

/*
 * Used when we had planned to send an fd using send_fd(), *
 but encountered an error instead. We send the error back *
 using the send_fd()/recv_fd() protocol.
 */
int
send_err(int fd, int errcode, const char *msg)
{
    int     n;

    if ((n = strlen(msg)) > 0) if (writen(fd, msg, n) != n) /* send
        the error message */ return(-1);

    if (errcode >= 0) errcode = -1; /* must be
    negative */ if (send_fd(fd, errcode) < 0)
    return(-1);

    return(0);
}
```

Figure 17.12 The send_err function

To exchange file descriptors using UNIX domain sockets, we call the sendmsg(2) and recvmsg(2) functions (Section 16.5). Both functions take a pointer to a msghdr structure that contains all the information on what to send or receive. The structure on your system might look similar to the following:

```
    struct msghdr {

        void          *msg_name;        /* optional address */
        socklen_t      msg_namelen;     /* address size in bytes */
        struct iovec  *msg_iov;         /* array of I/O buffers */
        int            msg_iovlen;      /* number of elements in array */
        void          *msg_control;     /* ancillary data */
       socklen_t     msg_controllen; /* number of ancillary bytes */ int
       msg_flags;    /* flags for received message */
    };
```

The first two elements are normally used for sending datagrams on a network connection, where the destination address can be specified with each datagram. The next two elements allow us to specify an array of buffers (scatter read or gather write), as we described for the readv and writev functions (Section 14.6). The msg_flags field contains flags describing the message received, as summarized in Figure 16.15.

Two elements deal with the passing or receiving of control information. The msg_control field points to a cmsghdr (control message header) structure, and the msg_controllen field contains the number of bytes of control information.

```
struct cmsghdr { socklen_t cmsg_len; /* data byte count, including
    header */
   int    cmsg_level; /* originating protocol */ int
   cmsg_type; /* protocol-specific type */
   /* followed by the actual control message data */
};
```

To send a file descriptor, we set cmsg_len to the size of the cmsghdr structure, plus the size of an integer (the descriptor). The cmsg_level field is set to SOL_SOCKET, and cmsg_type is set to SCM_RIGHTS, to indicate that we are passing access rights. (SCM stands for *socket-level control message*.) Access rights can be passed only across a UNIX domain socket. The descriptor is stored right after the cmsg_type field, using the macro CMSG_DATA to obtain the pointer to this integer.

Three macros are used to access the control data, and one macro is used to help calculate the value to be used for cmsg_len.

```
#include <sys/socket.h> unsigned char

*CMSG_DATA(struct cmsghdr *cp);
```

Returns: pointer to data associated with cmsghdr structure

```
struct cmsghdr *CMSG_FIRSTHDR(struct msghdr *mp);
```

Returns: pointer to first cmsghdr structure associated with the msghdr structure, or NULL if none exists

```
struct cmsghdr *CMSG_NXTHDR(struct msghdr *mp,
                            struct cmsghdr *cp);
```

Returns: pointer to next cmsghdr structure associated with the msghdr structure given the current cmsghdr structure, or NULL if we're at the last one

```
unsigned int CMSG_LEN(unsigned int nbytes);
```

Returns: size to allocate for data object *nbytes* large

The Single UNIX Specification defines the first three macros, but omits CMSG_LEN.

The CMSG_LEN macro returns the number of bytes needed to store a data object of size *nbytes*, after adding the size of the cmsghdr structure, adjusting for any alignment constraints required by the processor architecture, and rounding up.

The program in Figure 17.13 is the send_fd function, which passes a file descriptor over a UNIX domain socket. In the sendmsg call, we send both the protocol data (the null and the status byte) and the descriptor.

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

/* size of control buffer to send/recv one file descriptor */
#define CONTROLLEN CMSG_LEN(sizeof(int)) static struct cmsghdr

*cmptr = NULL; /* malloc'ed first time */

/*
 * Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status.
 */
int
send_fd(int fd, int fd_to_send)
{   struct   iovec   iov[1];
    struct msghdr msg;
    char            buf[2]; /* send_fd()/recv_fd() 2-byte protocol */

    iov[0].iov_base = buf;
    iov[0].iov_len = 2; msg.msg_iov
       = iov; msg.msg_iovlen = 1;
    msg.msg_name = NULL;
    msg.msg_namelen = 0;

    if (fd_to_send < 0) {
        msg.msg_control = NULL;
        msg.msg_controllen = 0;
        buf[1] = -fd_to_send; /* nonzero status means error */
        if (buf[1] == 0) buf[1] = 1; /* -256, etc. would screw
        up protocol */
    } else { if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) ==
        NULL) return(-1);
        cmptr->cmsg_level = SOL_SOCKET; cmptr-
        >cmsg_type = SCM_RIGHTS; cmptr->cmsg_len =
        CONTROLLEN; msg.msg_control = cmptr;
        msg.msg_controllen = CONTROLLEN;
        *(int *)CMSG_DATA(cmptr) = fd_to_send; /* the fd to pass */
        buf[1] = 0; /* zero status means OK */
    }

    buf[0] = 0;          /* null byte flag to recv_fd() */
    if (sendmsg(fd, &msg, 0) != 2) return(-
        1);
    return(0);
}
```

Figure 17.13 Sending a file descriptor over a UNIX domain socket

To receive a descriptor (Figure 17.14), we allocate enough room for a `cmsghdr` structure and a descriptor, set `msg_control` to point to the allocated area, and call `recvmsg`. We use the `CMSG_LEN` macro to calculate the amount of space needed.

We read from the socket until we read the null byte that precedes the final status byte. Everything up to this null byte is an error message from the sender.

```
#include "apue.h"
#include <sys/socket.h>     /* struct msghdr */

/* size of control buffer to send/recv one file descriptor */
#define CONTROLLEN CMSG_LEN(sizeof(int))

static struct cmsghdr *cmptr = NULL;        /* malloc'ed first time */

/*
 * Receive a file descriptor from a server process. Also, any data
 * received is passed to (*userfunc)(STDERR_FILENO, buf, nbytes).
 * We have a 2-byte protocol for receiving the fd from send_fd().
 */
int
recv_fd(int fd, ssize_t (*userfunc)(int, const void *, size_t))
{
    int         newfd, nr, status; char
       *ptr;
    char            buf[MAXLINE];
    struct iovec    iov[1]; struct
    msghdr msg; status = -1; for ( ;
    ; ) { iov[0].iov_base = buf;
    iov[0].iov_len = sizeof(buf);
    msg.msg_iov     = iov;
    msg.msg_iovlen = 1; msg.msg_name
    = NULL; msg.msg_namelen = 0;
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control = cmptr;
        msg.msg_controllen = CONTROLLEN; if
        ((nr = recvmsg(fd, &msg, 0)) < 0) {
        err_ret("recvmsg error");
            return(-1);
        } else if (nr == 0) { err_ret("connection
            closed by server");
            return(-1);
        }

        /*
         * See if this is the final data with null & status. Null
         * is next to last byte of buffer; status byte is last byte.
         * Zero status means there is a file descriptor to receive.
         */
        for (ptr = buf; ptr < &buf[nr]; ) { if
            (*ptr++ == 0) { if (ptr != &buf[nr-1])
            err_dump("message format error");
                status = *ptr & 0xFF; /* prevent sign extension */
                if (status == 0) { if (msg.msg_controllen !=
                CONTROLLEN) err_dump("status = 0 but no fd");
                    newfd = *(int *)CMSG_DATA(cmptr);
                } else { newfd = -
                    status;
                } nr -=
                2;
            }
        }
        if (nr > 0 && (*userfunc)(STDERR_FILENO, buf, nr) != nr)
            return(-1);
        if (status >= 0)   /* final data has arrived */
    return(newfd); /* descriptor, or -status */ }
}
```

Figure 17.14 Receiving a file descriptor over a UNIX domain socket

Note that we are always prepared to receive a descriptor (we set `msg_control` and `msg_controllen` before each call to `recvmsg`), but only if `msg_controllen` is nonzero on return did we actually receive a descriptor.

Recall the hoops we needed to jump through to determine the identity of the caller in the `serv_accept` function (Figure 17.9). It would have been better for the kernel to pass us the credentials of the caller on return from the call to `accept`. Some UNIX domain socket implementations provide similar functionality when exchanging messages, but their interfaces differ.

FreeBSD 8.0 and Linux 3.2.0 provide support for sending credentials over UNIX domain sockets, but they do it differently. Mac OS X 10.6.8 is derived in part from FreeBSD, but has credential passing disabled. Solaris 10 doesn't support sending credentials over UNIX domain sockets. However, it supports the ability to obtain the credentials of a process passing a file descriptor over a STREAMS pipe, although we do not discuss the details here.

With FreeBSD, credentials are transmitted as a `cmsgcred` structure:

```
#define CMGROUP_MAX 16
```

```
struct cmsgcred {

    pid_t cmcred_pid;                    /* sender's process ID */
    uid_t cmcred_uid;                    /* sender's real UID */
    uid_t cmcred_euid;                   /* sender's effective UID */
    gid_t cmcred_gid;                    /* sender's real GID */
    short cmcred_ngroups;                /* number of groups */
    gid_t cmcred_groups[CMGROUP_MAX]; /* groups */ };
```

When we transmit credentials, we need to reserve space only for the `cmsgcred` structure. The kernel will fill in this structure for us to prevent an application from pretending to have a different identity.

On Linux, credentials are transmitted as a `ucred` structure:

```
struct ucred { pid_t pid; /* sender's
process ID */ uid_t uid; /* sender's user
ID */ gid_t gid; /* sender's group ID */
};
```

Unlike FreeBSD, Linux requires that we initialize this structure before transmission. The kernel will ensure that applications either use values that correspond to the caller or have the appropriate privilege to use other values.

Figure 17.15 shows the `send_fd` function updated to include the credentials of the sending process.

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

#if defined(SCM_CREDS)          /* BSD interface */
#define CREDSTRUCT      cmsgcred
#define SCM_CREDTYPE    SCM_CREDS
#elif defined(SCM_CREDENTIALS) /* Linux interface */
#define CREDSTRUCT      ucred
#define SCM_CREDTYPE    SCM_CREDENTIALS
#else #error passing credentials is
unsupported!
#endif

/* size of control buffer to send/recv one file descriptor */
#define RIGHTSLEN CMSG_LEN(sizeof(int))
#define CREDSLEN    CMSG_LEN(sizeof(struct CREDSTRUCT))
#define CONTROLLEN (RIGHTSLEN + CREDSLEN)

static struct cmsghdr *cmptr = NULL; /* malloc'ed first time */

/*
 * Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status.
 */
int
send_fd(int fd, int fd_to_send)
```

```
{ struct CREDSTRUCT *credp; struct cmsghdr *cmp; struct iovec iov[1];
    struct msghdr msg; char buf[2]; /* send_fd/recv_ufd 2-byte protocol
    */ iov[0].iov_base = buf; iov[0].iov_len = 2; msg.msg_iov = iov;
    msg.msg_iovlen = 1;
    msg.msg_name = NULL;
    msg.msg_namelen = 0;
    msg.msg_flags = 0;
    if (fd_to_send < 0) {
        msg.msg_control = NULL;
        msg.msg_controllen = 0;
        buf[1] = -fd_to_send; /* nonzero status means error */
        if (buf[1] == 0) buf[1] = 1; /* -256, etc. would screw
        up protocol */
    } else { if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) ==
        NULL) return(-1);
        msg.msg_control = cmptr; msg.msg_controllen =
        CONTROLLEN;
        cmp = cmptr; cmp->cmsg_level =
        SOL_SOCKET; cmp->cmsg_type =
        SCM_RIGHTS; cmp->cmsg_len =
        RIGHTSLEN;
        *(int *)CMSG_DATA(cmp) = fd_to_send;   /* the fd to pass */
        cmp = CMSG_NXTHDR(&msg, cmp); cmp->cmsg_level = SOL_SOCKET;
        cmp->cmsg_type = SCM_CREDTYPE; cmp->cmsg_len = CREDSLEN;
        credp = (struct CREDSTRUCT *)CMSG_DATA(cmp);
#if defined(SCM_CREDENTIALS) credp-
        >uid = geteuid(); credp->gid =
        getegid(); credp->pid =
        getpid();
#endif buf[1] = 0;  /* zero status means OK */
    }
    buf[0] = 0;         /* null byte flag to recv_ufd() */
    if (sendmsg(fd, &msg, 0) != 2) return(-
        1);
    return(0);
}
```

Figure 17.15 Sending credentials over UNIX domain sockets

Note that we need to initialize the credentials structure only on Linux.

The function in Figure 17.16 is a modified version of `recv_fd`, called `recv_ufd`, that returns the user ID of the sender through a reference parameter.

```
#include "apue.h"
#include <sys/socket.h>    /* struct msghdr */ #include
<sys/un.h>
```

```
#if defined(SCM_CREDS)          /* BSD interface */
#define CREDSTRUCT      cmsgcred
#define CR_UID          cmcred_uid
#define SCM_CREDTYPE    SCM_CREDS
#elif defined(SCM_CREDENTIALS) /* Linux interface */
#define CREDSTRUCT      ucred
#define CR_UID          uid
#define CREDOPT         SO_PASSCRED
#define SCM_CREDTYPE    SCM_CREDENTIALS
#else #error passing credentials is
unsupported!
#endif

/* size of control buffer to send/recv one file descriptor */
#define RIGHTSLEN CMSG_LEN(sizeof(int))
#define CREDSLEN    CMSG_LEN(sizeof(struct CREDSTRUCT))
#define CONTROLLEN (RIGHTSLEN + CREDSLEN)

static struct cmsghdr *cmptr = NULL;        /* malloc'ed first time */

/*
 * Receive a file descriptor from a server process. Also, any data *
 received is passed to (*userfunc)(STDERR_FILENO, buf, nbytes). * We
 have a 2-byte protocol for receiving the fd from send_fd().
 */ int
recv_ufd(int fd, uid_t *uidptr, ssize_t (*userfunc)(int,
         const void *, size_t))
{
    struct cmsghdr  *cmp; struct
    CREDSTRUCT *credp;
    char                 *ptr;
    char                 buf[MAXLINE];
    struct iovec         iov[1];
    struct msghdr        msg;
    int                  nr;
    int                  newfd = -1;
    int                  status = -1;
#if defined(CREDOPT)
    const int            on = 1;
    if (setsockopt(fd, SOL_SOCKET, CREDOPT, &on, sizeof(int)) < 0) {
        err_ret("setsockopt error");
        return(-1);
    }
#endif for ( ; ; ) { iov[0].iov_base =
    buf; iov[0].iov_len  =  sizeof(buf);
    msg.msg_iov = iov; msg.msg_iovlen =
    1;     msg.msg_name     =     NULL;
    msg.msg_namelen = 0;
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control = cmptr;
        msg.msg_controllen = CONTROLLEN; if ((nr
        = recvmsg(fd, &msg, 0)) < 0) {
        err_ret("recvmsg error");
            return(-1);
        } else if (nr == 0) { err_ret("connection
            closed by server");
            return(-1);
        }
        /*
         * See if this is the final data with null & status. Null
         * is next to last byte of buffer; status byte is last byte.
         * Zero status means there is a file descriptor to receive.
         */ for (ptr = buf; ptr < &buf[nr]; ) { if
        (*ptr++ == 0) { if (ptr != &buf[nr-1])
        err_dump("message format error");
                status = *ptr & 0xFF; /* prevent sign extension */
                if (status == 0) { if (msg.msg_controllen !=
                CONTROLLEN) err_dump("status = 0 but no fd");
                    /* process the control data */
                    for (cmp = CMSG_FIRSTHDR(&msg);
                      cmp != NULL; cmp = CMSG_NXTHDR(&msg, cmp)) {
                        if (cmp->cmsg_level != SOL_SOCKET)
                        continue; switch (cmp->cmsg_type)
                        { case SCM_RIGHTS: newfd = *(int
                        *)CMSG_DATA(cmp);
                            break; case
                        SCM_CREDTYPE:
                            credp = (struct CREDSTRUCT *)CMSG_DATA(cmp);
                            *uidptr = credp->CR_UID;
                        }
                    }
                } else { newfd = -
                    status;
                } nr -=
                2;
            }
        }
        if (nr > 0 && (*userfunc)(STDERR_FILENO, buf, nr) != nr)
            return(-1);
        if (status >= 0)   /* final data has arrived */
    return(newfd); /* descriptor, or -status */ }
}
```

Figure 17.16 Receiving credentials over UNIX domain sockets

On FreeBSD, we specify `SCM_CREDS` to transmit credentials; on Linux, we use `SCM_CREDENTIALS`.

17.5 An Open Server, Version 1

Using file descriptor passing, we now develop an open server—a program that is executed by a process to open one or more files. Instead of sending the contents of the file back to the calling process, however, this server sends back an open file descriptor. As a result, the open server can work with any type of file (such as a device or a socket) and not simply regular files. The client and server exchange a minimum amount of information using IPC: the filename and open mode sent by the client, and the descriptor returned by the server. The contents of the file are not exchanged using IPC.

There are several advantages in designing the server to be a separate executable program (either one that is executed by the client, as we develop in this section, or a daemon server, which we develop in the next section).

- The server can easily be contacted by any client, similar to the client calling a library function. We are not hard-coding a particular service into the application, but designing a general facility that others can reuse.
- If we need to change the server, only a single program is affected. Conversely, updating a library function can require that all programs that call the function be updated (i.e., relinked with the link editor). Shared libraries can simplify this updating (Section 7.7).
- The server can be a set-user-ID program, providing it with additional permissions that the client does not have. Note that a library function (or shared library function) can't provide this capability.

The client process creates an fd-pipe and then calls `fork` and `exec` to invoke the server. The client sends requests across the fd-pipe using one end, and the server sends back responses over the fd-pipe using the other end.

We define the following application protocol between the client and the server.

1. The client sends a request of the form ''`open` *<pathname>* *<openmode>*\0'' across the fd-pipe to the server. The *<openmode>* is the numeric value, in ASCII decimal, of the second argument to the `open` function. This request string is terminated by a null byte.
2. The server sends back an open descriptor or an error by calling either `send_fd` or `send_err`.

This is an example of a process sending an open descriptor to its parent. In Section 17.6, we'll modify this example to use a single daemon server, where the server sends a descriptor to a completely unrelated process.

We first have the header, `open.h` (Figure 17.17), which includes the standard headers and defines the function prototypes.

```
#include "apue.h"
#include <errno.h>

#define CL_OPEN "open"     /* client's request for server */ int

csopen(char *, int);
```

Figure 17.17 The `open.h` header

The `main` function (Figure 17.18) is a loop that reads a pathname from standard input and copies the file to standard output. The function calls `csopen` to contact the open server and return an open descriptor.

```
#include "open.h"
#include <fcntl.h> #define

BUFFSIZE     8192 int

main(int argc, char *argv[])
{
           int     n, fd;
    char buf[BUFFSIZE]; char
    line[MAXLINE];

    /* read filename to cat from stdin */ while (fgets(line, MAXLINE, stdin)
    != NULL) { if (line[strlen(line) - 1] == ’\n’) line[strlen(line) - 1] =
    0; /* replace newline with null */

        /* open the file */
        if ((fd = csopen(line, O_RDONLY)) < 0) continue; /* csopen()
            prints error from server */

        /* and cat to stdout */
        while ((n = read(fd, buf, BUFFSIZE)) > 0) if
            (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");
        if (n < 0) err_sys("read
            error");
        close(fd);
    } exit(0);

}
```

Figure 17.18 The client `main` function, version 1

The function `csopen` (Figure 17.19) does the `fork` and `exec` of the server, after creating the fd-pipe.

```
#include "open.h"
        #include <sys/uio.h>       /* struct iovec */

/*
 * Open the file by sending the "name" and "oflag" to the *
 connection server and reading a file descriptor back.
 */ int
csopen(char *name, int oflag)
{
    pid_t           pid;
    int             len;
           char            buf[10];
    struct iovec    iov[3]; static int  fd[2]
    = { -1, -1 };

    if (fd[0] < 0) {       /* fork/exec our open server first time */ if
        (fd_pipe(fd) < 0) { err_ret("fd_pipe error");
            return(-1);
        }
        if ((pid = fork()) < 0) { err_ret("fork
            error");
            return(-1);
        } else if (pid == 0) {    /* child */ close(fd[0]);
            if (fd[1] != STDIN_FILENO &&
              dup2(fd[1], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin"); if
            (fd[1] != STDOUT_FILENO &&
              dup2(fd[1], STDOUT_FILENO) != STDOUT_FILENO)
            err_sys("dup2 error to stdout"); if
            (execl("./opend", "opend", (char *)0) < 0)
            err_sys("execl error");
        }
               close(fd[1]);               /* parent */
    }
    sprintf(buf, " %d", oflag); /* oflag to ascii */ iov[0].iov_base =
    CL_OPEN  " ";  /*  string  concatenation  */  iov[0].iov_len  =
    strlen(CL_OPEN)  +  1;  iov[1].iov_base  =  name;  iov[1].iov_len  =
    strlen(name); iov[2].iov_base = buf;
    iov[2].iov_len = strlen(buf) + 1; /* +1 for null at end of buf */
    len = iov[0].iov_len + iov[1].iov_len + iov[2].iov_len;
    if (writev(fd[0], &iov[0], 3) != len) { err_ret("writev
        error");
        return(-1);
    }

    /* read descriptor, returned errors handled by write() */
    return(recv_fd(fd[0], write));
}
```

Figure 17.19 The `csopen` function, version 1

The child closes one end of the fd-pipe, and the parent closes the other. For the server that it executes, the child also duplicates its end of the fd-pipe onto its standard input and standard output.

(Another option would have been to pass the ASCII representation of the descriptor `fd[1]` as an argument to the server.)

The parent sends to the server the request containing the pathname and open mode. Finally, the parent calls `recv_fd` to return either the descriptor or an error. If an error is returned by the server, `write` is called to output the message to standard error.

Now let's look at the open server. It is the program `opend` that is executed by the client in Figure 17.19. First, we have the `opend.h` header (Figure 17.20), which includes the standard headers and declares the global variables and function prototypes.

```
#include "apue.h"
#include <errno.h>

       #define CL_OPEN "open"           /* client's request for server */

extern char errmsg[]; /* error message string to return to client */
extern int oflag;    /* open() flag: O_xxx ... */ extern char
*pathname; /* of file to open() for client */

int    cli_args(int, char **); void
       handle_request(char *, int, int);
```

Figure 17.20 The `opend.h` header, version 1

The `main` function (Figure 17.21) reads the requests from the client on the fd-pipe (its standard input) and calls the function `handle_request`.

```
#include "opend.h"

char errmsg[MAXLINE]; int oflag;
char *pathname;

int main(void)
{
    int nread; char
    buf[MAXLINE];

    for ( ; ; ) { /* read arg buffer from client, process request */ if
        ((nread = read(STDIN_FILENO, buf, MAXLINE)) < 0) err_sys("read error
        on stream pipe");
        else if (nread == 0) break;     /* client has closed the
            stream pipe */
        handle_request(buf, nread, STDOUT_FILENO);
    } exit(0);
}
```

Figure 17.21 The server `main` function, version 1

The function `handle_request` in Figure 17.22 does all the work. It calls the function `buf_args` to break up the client's request into a standard `argv`-style argument list and calls the function `cli_args` to process the client's arguments. If all is OK, `open` is called to open the file, and then `send_fd` sends the descriptor back to the client across the fd-pipe (its standard output). If an error is encountered, `send_err` is called to send back an error message, using the client–server protocol that we described earlier.

```
#include "opend.h" #include
<fcntl.h>

void
handle_request(char *buf, int nread, int fd)
{
          int     newfd;

    if (buf[nread-1] != 0) { snprintf(errmsg,
        MAXLINE-1,
          "request not null terminated: %*.*s\n", nread, nread, buf);
        send_err(fd, -1, errmsg);
        return;
    }
    if (buf_args(buf, cli_args) < 0) { /* parse args & set options */
        send_err(fd, -1, errmsg);
        return;
    }
    if ((newfd = open(pathname, oflag)) < 0) { snprintf(errmsg, MAXLINE-1,
        "can't open %s: %s\n", pathname,
          strerror(errno));
        send_err(fd, -1, errmsg);
        return;
    }
    if (send_fd(fd, newfd) < 0)   /* send the descriptor */ err_sys("send_fd
        error");
          close(newfd);       /* we're done with descriptor */
}
```

Figure 17.22 The `handle_request` function, version 1

The client's request is a null-terminated string of white-space-separated arguments. The function `buf_args` in Figure 17.23 breaks this string into a standard `argv`-style argument list and calls a user function to process the arguments. We use the ISO C function `strtok` to tokenize the string into separate arguments.

```
#include "apue.h"

       #define MAXARGC      50 /* max number of arguments in buf */
#define WHITE " \t\n" /* white space for tokenizing arguments */

/*
 * buf[] contains white-space-separated arguments. We convert it to an
 * argv-style array of pointers, and call the user's function (optfunc)
 * to process the array. We return -1 if there's a problem parsing buf,* else
   we return whatever optfunc() returns. Note that user's buf[] * array is
   modified (nulls placed after each token).
 */ int
buf_args(char *buf, int (*optfunc)(int, char **))
{
    char *ptr, *argv[MAXARGC]; int argc;

    if (strtok(buf, WHITE) == NULL)     /* an argv[0] is required */
    return(-1); argv[argc = 0] = buf;
    while ((ptr = strtok(NULL, WHITE)) != NULL) { if (++argc >= MAXARGC-1)
             /* -1 for room for NULL at end */ return(-1);
        argv[argc] = ptr;
    }
    argv[++argc] = NULL;

    /*
   *   Since argv[] pointers point into the user's buf[],* user's
   function can just copy the pointers, even * though argv[]
   array will disappear on return.
     */ return((*optfunc)(argc, argv));
}
```

Figure 17.23 The `buf_args` function

The server's function that is called by `buf_args` is `cli_args` (Figure 17.24). It verifies that the client sent the right number of arguments and stores the pathname and open mode in global variables.

```
#include "opend.h"

/*
   *   This function is called by buf_args(), which is called by*
   handle_request(). buf_args() has broken up the client's *
   buffer into an argv[]-style array, which we now process.
 */ int
cli_args(int argc, char **argv)
{ if (argc != 3 || strcmp(argv[0], CL_OPEN) != 0) {
    strcpy(errmsg, "usage: <pathname> <oflag>\n");
        return(-1);
    }
    pathname = argv[1];    /* save ptr to pathname to open */ oflag =
    atoi(argv[2]);
    return(0);
}
```

Figure 17.24 The `cli_args` function

This completes the open server that is invoked by a `fork` and `exec` from the client. A single fd-pipe is created before the `fork` and is used to communicate between the client and the server. With this arrangement, we have one server per client.

17.6 An Open Server, Version 2

In the previous section, we developed an open server that was invoked by a fork and exec by the client, demonstrating how we can pass file descriptors from a child to a parent. In this section, we develop an open server as a daemon process. One server handles all clients. We expect this design to be more efficient, since a fork and an exec are avoided. We use a UNIX domain socket connection between the client and the server and demonstrate passing file descriptors between unrelated processes. We'll use the three functions serv_listen, serv_accept, and cli_conn introduced in Section 17.3. This server also demonstrates how a single server can handle multiple clients, using both the select and poll functions from Section 14.4.

This version of the client is similar to the client from Section 17.5. Indeed, the file main.c is identical (Figure 17.18). We add the following line to the open.h header (Figure 17.17):

```
#define CS_OPEN "/tmp/opend.socket" /* server's well-known name */
```

The file open.c does change from Figure 17.19, since we now call cli_conn instead of doing the fork and exec. This is shown in Figure 17.25.

```
#include "open.h"
  #include <sys/uio.h>       /* struct iovec */

/*
 * Open the file by sending the "name" and "oflag" to the
 * connection server and reading a file descriptor back.
 */
int
csopen(char *name, int oflag)
{
     int             len;
     char            buf[12];
     struct iovec    iov[3];
     static int      csfd = -1;
     if (csfd < 0) { /* open connection to conn server */ if
          ((csfd = cli_conn(CS_OPEN)) < 0) { err_ret("cli_conn
          error");
               return(-1);
          }
     }

     sprintf(buf,  "  %d",  oflag);  /*  oflag  to  ascii  */
     iov[0].iov_base = CL_OPEN " "; /* string concatenation */
     iov[0].iov_len  =  strlen(CL_OPEN)  +  1;  iov[1].iov_base  =
     name; iov[1].iov_len = strlen(name); iov[2].iov_base = buf;
     iov[2].iov_len  =  strlen(buf)  +  1;  /*  null  always  sent  */
     len = iov[0].iov_len + iov[1].iov_len + iov[2].iov_len;
```

```
    if (writev(csfd, &iov[0], 3) != len) { err_ret("writev
        error");
        return(-1);
    }

    /* read back descriptor; returned errors handled by write() */
    return(recv_fd(csfd, write));
}
```

Figure 17.25 The `csopen` function, version 2

The protocol from the client to the server remains the same.

Next, we'll look at the server. The header `opend.h` (Figure 17.26) includes the standard headers and declares the global variables and the function prototypes.

```
#include "apue.h"
#include <errno.h>

#define CS_OPEN "/tmp/opend.socket" /* well-known name */
       #define CL_OPEN "open"               /* client's request for server */

extern int debug;   /* nonzero if interactive (not daemon) */ extern char
errmsg[]; /* error message string to return to client */
extern int oflag;   /* open flag: O_xxx ... */ extern char
*pathname; /* of file to open for client */

       typedef struct {     /* one Client struct per connected client */
  int fd;     /* fd, or -1 if available */ uid_t uid; }
Client;

extern Client *client;     /* ptr to malloc'ed array */ extern int
       client_size; /* # entries in client[] array */
int       cli_args(int, char **);
int       client_add(int, uid_t);
void      client_del(int);
void      loop(void);
void      handle_request(char *, int, int, uid_t);
```

Figure 17.26 The `opend.h` header, version 2

Since this server handles all clients, it must maintain the state of each client connection. This is done with the `client` array declared in the `opend.h` header. Figure 17.27 defines three functions that manipulate this array.

```
#include "opend.h"
```

```
        #define NALLOC 10       /* # client structs to alloc/realloc for */

static void
       client_alloc(void)      /* alloc more entries in the client[] array */
{
            int     i;

    if (client == NULL) client = malloc(NALLOC * sizeof(Client)); else client
    = realloc(client, (client_size+NALLOC)*sizeof(Client)); if (client ==
    NULL) err_sys("can't alloc for client array");

    /* initialize the new entries */
    for (i = client_size; i < client_size + NALLOC; i++) client[i].fd = -1;
        /* fd of -1 means entry available */

    client_size += NALLOC;
}

/*
 * Called by loop() when connection request from a new client arrives.
 */ int
client_add(int fd, uid_t uid)
{
            int     i;

    if (client == NULL)    /* first time we're called */
client_alloc(); again:
    for (i = 0; i < client_size; i++) { if (client[i].fd == -1) {  /* find
        an available entry */ client[i].fd = fd; client[i].uid = uid;
            return(i); /* return index in client[] array */ }
    }

    /* client array full, time to realloc for more */
    client_alloc();
            goto again;     /* and search again (will work this time) */
}

/*
 * Called by loop() when we're done with a client.
 */ void
client_del(int fd)
{
            int     i;

    for (i = 0; i < client_size; i++) { if
        (client[i].fd == fd) { client[i].fd = -1;
            return;
        }
    }
    log_quit("can't find client entry for fd %d", fd);
}
```

Figure 17.27 Functions to manipulate `client` array

The first time `client_add` is called, it calls `client_alloc`, which in turn calls `malloc` to allocate space for ten entries in the array. After these ten entries are all in use, a later call to `client_add` causes `realloc` to allocate additional space. By dynamically allocating space this way, we have not limited the size of the `client` array at compile time to some value that we guessed and put into a header. These functions call the `log_` functions (Appendix B) if an error occurs, since we assume that the server is a daemon.

Normally the server will run as a daemon, but we want to provide an option that allows it to be run in the foreground, with diagnostic messages sent to the standard error. This should make the server easier to test and debug, especially if we don't have permission to read the log file where the diagnostic messages are normally written. We'll use a command-line option to control whether the server runs in the foreground or as a daemon in the background.

It is important that all commands on a system follow the same conventions, because this makes them easier to use. If someone is familiar with the way command-line options are formed with one command, it would create more chances for mistakes if another command followed different conventions.

This problem is sometimes visible when dealing with white space on the command line. Some commands require that an option be separated from its argument by white space, but other commands require the argument to follow immediately after its option, without any intervening spaces. Without a consistent set of rules to follow, users either have to memorize the syntax of all commands or resort to a trial-and-error process when invoking them.

The Single UNIX Specification includes a set of conventions and guidelines that promote consistent command-line syntax. They include such suggestions as ‘‘Restrict each command-line option to a single alphanumeric character’’ and ‘‘All options should be preceded by a − character.’’

Luckily, the `getopt` function exists to help command developers process command-line options in a consistent manner.

```
#include <unistd.h>

int getopt(int argc, char * const argv[], const char
*options); extern int optind, opterr, optopt; extern char
*optarg;
```

Returns: the next option character, or −1 when all options have been processed

The *argc* and *argv* arguments are the same ones passed to the `main` function of the program. The *options* argument is a string containing the option characters supported by the command. If an option character is followed by a colon, then the option takes an argument. Otherwise, the option exists by itself. For example, if the usage statement for a command was `command [-i] [-u username] [-z] filename`

we would pass `"iu:z"` as the *options* string to `getopt`.

The getopt function is normally used in a loop that terminates when getopt returns −1. During each iteration of the loop, getopt will return the next option processed. It is up to the application to sort out any conflict in options, however; getopt simply parses the options and enforces a standard format.

When it encounters an invalid option, getopt returns a question mark instead of the character. If an option's argument is missing, getopt will also return a question mark, but if the first character in the options string is a colon, getopt returns a colon instead. The special pattern -- will cause getopt to stop processing options and return −1. This allows users to provide command arguments that start with a minus sign but aren't options. For example, if you have a file named −bar, you can't remove it by typing rm -bar

because rm will try to interpret −bar as options. The way to remove the file is to type rm -- -bar

The getopt function supports four external variables.

optarg	If an option takes an argument, getopt sets optarg to point to the option's argument string when an option is processed.
opterr	If an option error is encountered, getopt will print an error message by default. To disable this behavior, applications can set opterr to 0.
optind	The index in the argv array of the next string to be processed. It starts at 1 and is incremented for each argument processed by getopt.
optopt	If an error is encountered during options processing, getopt will set optopt to point to the option string that caused the error.

The open server's main function (Figure 17.28) defines the global variables, processes the command-line options, and calls the function loop. If we invoke the server with the −d option, the server runs interactively instead of as a daemon. This option is used when testing the server.

```
#include "opend.h" #include
<syslog.h>

int   debug,   oflag,   client_size,   log_to_stderr;   char
errmsg[MAXLINE]; char *pathname;
Client *client = NULL;

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

     log_open("open.serv", LOG_PID, LOG_USER);

            opterr = 0;     /* don't want getopt() writing to stderr */
     while ((c = getopt(argc, argv, "d")) != EOF) { switch (c)
          {
                 case 'd':        /* debug */
              debug = log_to_stderr = 1; break;
```

```
        case '?':
            err_quit("unrecognized option: -%c", optopt);
        }
    }

    if (debug == 0) daemonize("opend");

          loop();     /* never returns */
}
```

Figure 17.28 The server `main` function, version 2

The function `loop` is the server's infinite loop. We'll show two versions of this function. Figure 17.29 shows one version that uses `select`; Figure 17.30 shows another version that uses `poll`.

```
#include "opend.h" #include
<sys/select.h>

void loop(void)
{ int  i, n, maxfd, maxi, listenfd, clifd, nread; char
    buf[MAXLINE]; uid_t uid; fd_set rset, allset;

    FD_ZERO(&allset);

    /* obtain fd to listen for client requests on */ if
    ((listenfd = serv_listen(CS_OPEN)) < 0)
    log_sys("serv_listen error");
    FD_SET(listenfd, &allset); maxfd
    = listenfd; maxi = -1;
    for ( ; ; ) { rset = allset; /* rset gets modified each time around
        */ if ((n = select(maxfd + 1, &rset, NULL, NULL, NULL)) < 0)
        log_sys("select error");

        if (FD_ISSET(listenfd, &rset)) {
            /* accept new client request */
            if ((clifd = serv_accept(listenfd, &uid)) < 0)
                log_sys("serv_accept error: %d", clifd);
            i = client_add(clifd, uid); FD_SET(clifd, &allset);
            if (clifd > maxfd) maxfd = clifd; /* max fd for
            select() */
            if (i > maxi) maxi = i; /* max index in client[] array */
            log_msg("new connection: uid %d, fd %d", uid, clifd);
            continue;
        }

        for (i = 0; i <= maxi; i++) { /* go through client[] array */ if
            ((clifd = client[i].fd) < 0) continue;
            if (FD_ISSET(clifd, &rset)) {
                /* read argument buffer from client */ if ((nread =
                read(clifd, buf, MAXLINE)) < 0) { log_sys("read
                error on fd %d", clifd);
                } else if (nread == 0) { log_msg("closed: uid
                    %d, fd %d",
                         client[i].uid, clifd);
                    client_del(clifd); /* client has closed cxn */
                    FD_CLR(clifd, &allset);
                    close(clifd);
                } else {   /* process client's request */ handle_request(buf,
                    nread, clifd, client[i].uid);
                }
            }
        }
    }
}
```

Figure 17.29 The `loop` function using `select`

This function calls `serv_listen` (Figure 17.8) to create the server's endpoint for the client connections. The remainder of the function is a loop that starts with a call to `select`. Two conditions can be true after `select` returns.

1. The descriptor `listenfd` can be ready for reading, which means that a new client has called `cli_conn`. To handle this, we call `serv_accept` (Figure 17.9) and then update the `client` array and associated bookkeeping information for the new client. (We keep track of the highest descriptor number for the first
   argument to `select`. We also keep track of the highest index in use in the `client` array.)
2. An existing client's connection can be ready for reading. This means that theclient has either terminated or sent a new request. We find out about a client termination by `read` returning 0 (end of file). If `read` returns a value greater than 0, there is a new request to process, which we handle by calling `handle_request`.

We keep track of which descriptors are currently in use in the `allset` descriptor set. As new clients connect to the server, the appropriate bit is turned on in this descriptor set. The appropriate bit is turned off when the client terminates.

We always know when a client terminates, whether the termination is voluntary or not, since all the client's descriptors (including the connection to the server) are automatically closed by the kernel. This differs from the XSI IPC mechanisms.

The `loop` function that uses `poll` is shown in Figure 17.30.

```
#include "opend.h"
#include <poll.h>
```

```
#define NALLOC 10 /* # pollfd structs to alloc/realloc */

static struct pollfd *
grow_pollfd(struct pollfd *pfd, int *maxfd)
{
           int              i;
    int      oldmax = *maxfd; int       newmax =
    oldmax + NALLOC;

    if ((pfd = realloc(pfd, newmax * sizeof(struct pollfd))) == NULL)
    err_sys("realloc error"); for (i = oldmax; i < newmax; i++) { pfd[i].fd
    = -1; pfd[i].events = POLLIN; pfd[i].revents = 0;
    }
    *maxfd = newmax; return(pfd);
}

void loop(void)
{
           int              i, listenfd, clifd, nread;
    char buf[MAXLINE]; uid_t uid;
    struct pollfd *pollfd;
    int      numfd = 1; int      maxfd =
    NALLOC;

    if ((pollfd = malloc(NALLOC * sizeof(struct pollfd))) == NULL)
        err_sys("malloc error");
    for (i = 0; i < NALLOC; i++) { pollfd[i].fd = -
        1; pollfd[i].events = POLLIN;
        pollfd[i].revents = 0;
    }

    /* obtain fd to listen for client requests on */ if
    ((listenfd = serv_listen(CS_OPEN)) < 0)
    log_sys("serv_listen error");
    client_add(listenfd, 0);     /* we use [0] for listenfd */ pollfd[0].fd =
    listenfd;

    for ( ; ; ) { if (poll(pollfd, numfd, -1) <
        0) log_sys("poll error");

        if (pollfd[0].revents & POLLIN) {
            /* accept new client request */
            if ((clifd = serv_accept(listenfd, &uid)) < 0)
            log_sys("serv_accept error: %d", clifd); client_add(clifd, uid);

            /* possibly increase the size of the pollfd array */
            if (numfd == maxfd) pollfd = grow_pollfd(pollfd,
            &maxfd); pollfd[numfd].fd = clifd;
            pollfd[numfd].events = POLLIN;
            pollfd[numfd].revents = 0;
            numfd++;
            log_msg("new connection: uid %d, fd %d", uid, clifd);
        }

        for (i = 1; i < numfd; i++) { if
            (pollfd[i].revents & POLLHUP) { goto
            hungup;
            } else if (pollfd[i].revents & POLLIN) { /* read
                argument buffer from client */
           if ((nread = read(pollfd[i].fd, buf, MAXLINE)) < 0) { log_sys("read
                             error on fd %d", pollfd[i].fd);
                } else if (nread == 0) {
hungup:
                    /* the client closed the connection */ log_msg("closed:
                    uid %d, fd %d",
                           client[i].uid, pollfd[i].fd);
                    client_del(pollfd[i].fd);
                    close(pollfd[i].fd); if (i <
                    (numfd-1)) {
                        /* pack the array */ pollfd[i].fd = pollfd[numfd-
                        1].fd; pollfd[i].events = pollfd[numfd-1].events;
                        pollfd[i].revents = pollfd[numfd-1].revents; i--; /*
                        recheck this entry */
                    }
                    numfd--;
                } else {   /* process client's request */ handle_request(buf,
                    nread, pollfd[i].fd,
                      client[i].uid);
                }
            }
        }
    }
}
```

Figure 17.30 The `loop` function using `poll`

To allow for as many clients as there are possible open descriptors, we dynamically allocate space for the array of `pollfd` structures using the same strategy as used in the `client_alloc` function for the `client` array (see Figure 17.27).

We use the first entry (index 0) of the `pollfd` array for the `listenfd` descriptor. The arrival of a new client connection is indicated by a `POLLIN` on the `listenfd` descriptor. As before, we call `serv_accept` to accept the connection.

For an existing client, we have to handle two different events from `poll`: a client termination is indicated by `POLLHUP`, and a new request from an existing client is indicated by `POLLIN`. The client can close its end of the connection while there is still data to be read from the server's end of the connection.

Even though the endpoint is marked as hung up, the server can read all the data queued on its end. But with this server, when we receive the hangup from the client, we can `close` the connection to the client, effectively throwing away any queued data. There is no reason to process any requests still remaining, since we can’t send any responses back.

As with the `select` version of this function, new requests from a client are handled by calling the `handle_request` function (Figure 17.31). This function is similar to the earlier version (Figure 17.22). It calls the same function, `buf_args` (Figure 17.23), that calls `cli_args` (Figure 17.24), but since it runs from a daemon process, it logs error messages instead of printing them on the standard error.

```
#include "opend.h" #include
<fcntl.h>

void
handle_request(char *buf, int nread, int clifd, uid_t uid)
{
          int     newfd;

    if (buf[nread-1] != 0) { snprintf(errmsg,
        MAXLINE-1,
             "request from uid %d not null terminated: %*.*s\n",
          uid, nread, nread, buf);
        send_err(clifd, -1, errmsg);
        return;
    }
    log_msg("request: %s, from uid %d", buf, uid);
    /* parse the arguments, set options */
    if (buf_args(buf, cli_args) < 0) {
    send_err(clifd, -1, errmsg);
        log_msg(errmsg);
        return;
    }

    if ((newfd = open(pathname, oflag)) < 0) {
        snprintf(errmsg, MAXLINE-1, "can’t open %s: %s\n",
          pathname, strerror(errno));
        send_err(clifd, -1, errmsg);
        log_msg(errmsg);
        return;
    }

    /* send the descriptor */ if
    (send_fd(clifd, newfd) < 0)
    log_sys("send_fd error");
    log_msg("sent fd %d over fd %d for %s", newfd, clifd, pathname);
    close(newfd);   /* we’re done with descriptor */
}
```

Figure 17.31 The `request` function, version 2

This completes the second version of the open server, which uses a single daemon to handle all the client requests.

17.7 Summary

The key points in this chapter are the ability to pass file descriptors between processes and the ability of a server to accept unique connections from clients. Although all platforms provide support for UNIX domain sockets (refer back to Figure 15.1), we’ve seen that there are differences in each implementation, which makes it more difficult for us to develop portable applications.

We used UNIX domain sockets throughout this chapter. We saw how to use them to implement a full-duplex pipe and how they can be used to adapt the I/O multiplexing functions from Section 14.4 to work indirectly with XSI message queues.

We presented two versions of an open server. One version was invoked directly by the client, using `fork` and `exec`. The second was a daemon server that handled all client requests. Both versions used the file descriptor passing and receiving functions.

We also saw how to use the `getopt` function to enforce consistent command-line processing for our programs. The final version of the open server used the `getopt` function, the client–server connection functions introduced in Section 17.3, and the I/O multiplexing functions from Section 14.4.