Chapter 10. Signals

10.11. Signal Sets

We need a data type to represent multiple signals—a signal set. We'll use this with such functions as `sigprocmask` (in the next section) to tell the kernel not to allow any of the signals in the set to occur. As we mentioned earlier, the number of different signals can exceed the number of bits in an integer, so in general, we can't use an integer to represent the set with one bit per signal. POSIX.1 defines the data type `sigset_t` to contain a signal set and the following five functions to manipulate signal sets.

```
#include <signal.h>

int sigemptyset(sigset_t *set);

int sigfillset(sigset_t *set);

int sigaddset(sigset_t *set, int signo);

int sigdelset(sigset_t *set, int signo);
```

All four return: 0 if OK, –1 on error

```
int sigismember(const sigset_t *set, int signo);
```

Returns: 1 if true, 0 if false, –1 on error

The function `sigemptyset` initializes the signal set pointed to by set so that all signals are excluded. The function `sigfillset` initializes the signal set so that all signals are included. All applications have to call either `sigemptyset` or `sigfillset` once for each signal set, before using the signal set, because we cannot assume that the C initialization for external and static variables (0) corresponds to the implementation of signal sets on a given system.

Once we have initialized a signal set, we can add and delete specific signals in the set. The function `sigaddset` adds a single signal to an existing set, and `sigdelset` removes a single signal from a set. In all the functions that take a signal set as an argument, we always pass the address of the signal set as the argument.

Implementation

If the implementation has fewer signals than bits in an integer, a signal set can be implemented using one bit per signal. For the remainder of this section, assume that an implementation has 31 signals and 32-bit integers. The `sigemptyset` function zeros the integer, and the `sigfillset` function turns on all the bits in the integer. These two functions can be implemented as macros in the `<signal.h>` header:

```
#define sigemptyset(ptr)   (*(ptr) = 0)
#define sigfillset(ptr)    (*(ptr) = ~(sigset_t)0, 0)
```

Note that `sigfillset` must return 0, in addition to setting all the bits on in the signal set, so we use C's comma operator, which returns the value after the comma as the value of the expression.

Using this implementation, `sigaddset` turns on a single bit and `sigdelset` turns off a single bit; `sigismember` tests a certain bit. Since no signal is ever numbered 0, we subtract 1 from the signal number to obtain the bit to manipulate. Figure 10.12 shows implementations of these functions.

Figure 10.12. An implementation of `sigaddset`, `sigdelset`, and `sigismember`

```
#include     <signal.h>
#include     <errno.h>

/* <signal.h> usually defines NSIG to include signal number 0 */
#define SIGBAD(signo)   ((signo) <= 0 || (signo) >= NSIG)

int
sigaddset(sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

    *set |= 1 << (signo - 1);       /* turn bit on */
    return(0);
}

int
sigdelset(sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

    *set &= ~(1 << (signo - 1));    /* turn bit off */
    return(0);
}

int
sigismember(const sigset_t *set, int signo)
{
     if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

     return((*set & (1 << (signo - 1))) != 0);
}
```

We might be tempted to implement these three functions as one-line macros in the `<signal.h>` header, but POSIX.1 requires us to check the signal number argument for validity and to set `errno` if it is invalid. This is more difficult to do in a macro than in a function.

10.12. `sigprocmask` Function

Recall from Section 10.8 that the signal mask of a process is the set of signals currently blocked from delivery to that process. A process can examine its signal mask, change its signal mask, or perform both operations in one step by calling the following function.

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *restrict set,
                sigset_t *restrict oset);
```

Returns: 0 if OK, –1 on error

First, if oset is a non-null pointer, the current signal mask for the process is returned through oset.

Second, if set is a non-null pointer, the how argument indicates how the current signal mask is modified. Figure 10.13 describes the possible values for how. `SIG_BLOCK` is an inclusive-OR operation, whereas `SIG_SETMASK` is an assignment. Note that `SIGKILL` and `SIGSTOP` can't be blocked.

Figure 10.13. Ways to change current signal mask using `sigprocmask`

how	Description
`SIG_BLOCK`	The new signal mask for the process is the union of its current signal mask and the signal set pointed to by set. That is, set contains the additional signals that we want to block.
`SIG_UNBLOCK`	The new signal mask for the process is the intersection of its current signal mask and the complement of the signal set pointed to by set. That is, set contains the signals that we want to unblock.
`SIG_SETMASK`	The new signal mask for the process is replaced by the value of the signal set pointed to by set.

If set is a null pointer, the signal mask of the process is not changed, and how is ignored.

After calling `sigprocmask`, if any unblocked signals are pending, at least one of these signals is delivered to the process before `sigprocmask` returns.

The `sigprocmask` function is defined only for single-threaded processes. A separate function is provided to manipulate a thread's signal mask in a multithreaded process. We'll discuss this in Section 12.8.

Example

Figure 10.14 shows a function that prints the names of the signals in the signal mask of the calling process. We call this function from the programs shown in Figure 10.20 and Figure 10.22.

To save space, we don't test the signal mask for every signal that we listed in Figure 10.1. (See Exercise 10.9.)

Figure 10.14. Print the signal mask for the process

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

void
pr_mask(const char *str)
{
    sigset_t    sigset;
    int         errno_save;

    errno_save = errno;     /* we can be called by signal handlers */
    if (sigprocmask(0, NULL, &sigset) < 0)
        err_sys("sigprocmask error");

    printf("%s", str);
    if (sigismember(&sigset, SIGINT))   printf("SIGINT ");
    if (sigismember(&sigset, SIGQUIT))  printf("SIGQUIT ");
    if (sigismember(&sigset, SIGUSR1))  printf("SIGUSR1 ");
    if (sigismember(&sigset, SIGALRM))  printf("SIGALRM ");

    /* remaining signals can go here */

    printf("\n");
    errno = errno_save;
}
```

Chapter 8. Process Control

8.1. Introduction

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

8.2. Process Identifiers

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

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

There are some special processes, but the details differ from implementation to implementation. Process ID 0 is usually the scheduler process and is often known as the swapper. No program on disk corresponds to this process, which is part of the kernel and is known as a system process. Process ID 1 is usually the `init` process and is invoked by the kernel at the end of the bootstrap procedure. The program file for this process was `/etc/init` in older versions of the UNIX System and is `/sbin/init` in newer versions. This process is responsible for bringing up a UNIX system after the kernel has been bootstrapped. `init` usually reads the system-dependent initialization files—the `/etc/rc*` files or `/etc/inittab` and the files in `/etc/init.d`—and brings the system to a certain state, such as multiuser. The `init` process never dies. It is a normal user process, not a system process within the kernel, like the swapper, although it does run with superuser privileges. Later in this chapter, we'll see how `init` becomes the parent process of any orphaned child process.

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);
```

```
#include <unistd.h>

pid_t getpid(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 share the text segment (Section 7.6).

Current 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 2.4.22 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 5.2.1 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 9 provides two threads libraries: one for POSIX threads (pthreads) and one for Solaris threads. The behavior of `fork` differs between the two thread libraries. For POSIX threads, `fork` creates a process containing only the calling thread, but for Solaris threads, `fork` creates a process containing copies of all threads from the process of the calling thread. To provide similar semantics as POSIX threads, Solaris provides the `fork1` function, which can be used to create a process that duplicates only the calling thread, regardless of the thread library used. 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.

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. This depends on the scheduling algorithm used by the kernel. If it's required that the child and parent synchronize, 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 this 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 `printf` line, because the standard output buffer is flushed by the newline. But when we redirect standard output to a file, 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.

Figure 8.1. Example of `fork` function

```
#include "apue.h"

int     glob = 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;
```

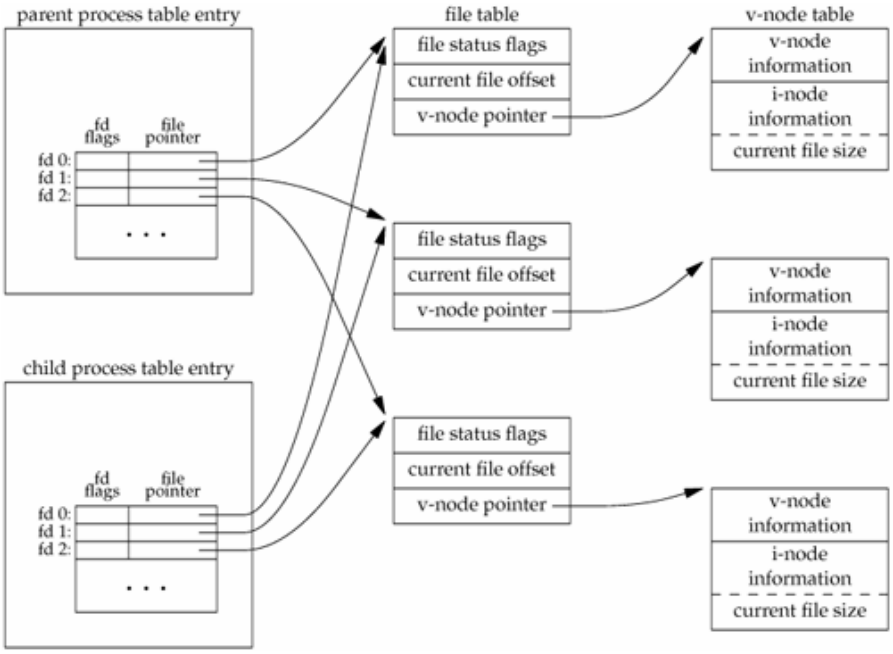
```
    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");
    printf("before fork\n");    /* we don't flush stdout */

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

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

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 3.8).

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.

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



It is important that the parent and the child share the same file offset. Consider a process that `fork`s a child, then `wait`s for the child to complete. Assume that both processes write to standard output as part of their normal processing. If the parent has its standard output redirected (by a shell, perhaps) it is essential that the parent's

file offset be updated by the child when the child writes to standard output. In this case, the child can write to standard output while the parent is `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.

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

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

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

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

- Real user ID, real group ID, effective user ID, 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 value from `fork`
- 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
- 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 2.10 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 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 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.

Chapter 15. Interprocess Communication

15.1. Introduction

In Chapter 8, we described the process control primitives and saw how to invoke 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 each other: IPC, or interprocess communication.

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 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.

Figure 15.1. Summary of UNIX System IPC

IPC type	SUS	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
half-duplex pipes	•	(full)	•	•	(full)
FIFOs	•	•	•	•	•
full-duplex pipes	allowed	•,UDS	opt, UDS	UDS	•, UDS
named full-duplex pipes	XSI option	UDS	opt, UDS	UDS	•, UDS
message queues	XSI	•	•		•
semaphores	XSI	•	•	•	•
shared memory	XSI	•	•	•	•
sockets	•	•	•	•	•
STREAMS	XSI option		opt		•

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 17.3), 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.

As we mentioned in Section 14.4, support for STREAMS is optional in the Single UNIX Specification. Named full-duplex pipes are provided as mounted STREAMS-based pipes and so are also optional in the Single UNIX Specification. On Linux, support for STREAMS is available in a separate, optional package called "LiS" (for Linux STREAMS). We show "opt" where the platform provides support for the feature through an optional package—one that is not usually installed by default.

The first seven 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 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 17.3) and named STREAMS-based pipes (Section 17.2.2) 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 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 filedes[2]);
```

Returns: 0 if OK, –1 on error

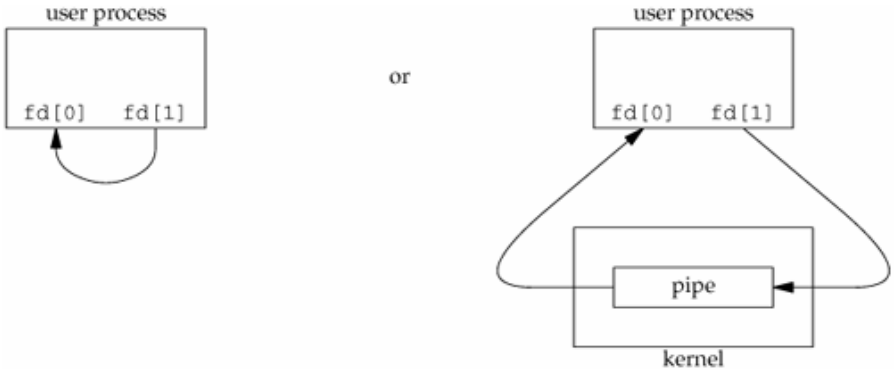
Two file descriptors are returned through the filedes argument: filedes[0] is open for reading, and filedes[1] is open for writing. The output of filedes[1] is the input for filedes[0].

Pipes are implemented using UNIX domain sockets in 4.3BSD, 4.4BSD, and Mac OS X 10.3. Even though UNIX domain sockets are full duplex by default, these operating systems hobble the sockets used with pipes so that they operate in half-duplex mode only.

POSIX.1 allows for an implementation to support full-duplex pipes. For these implementations, filedes[0] and filedes[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

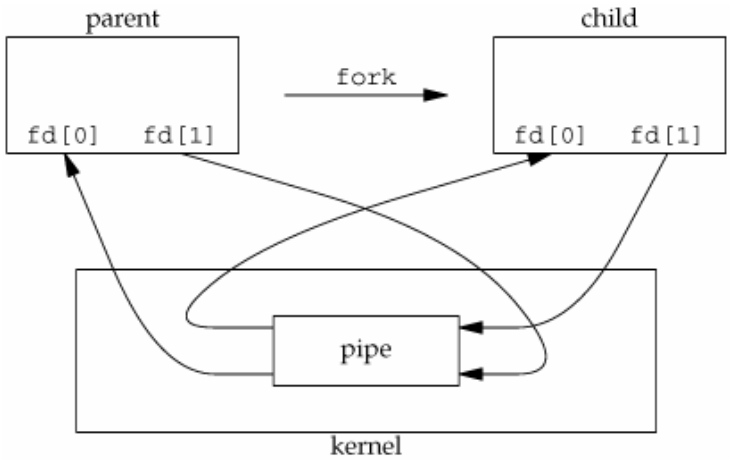


The `fstat` function (Section 4.2) returns a file type of FIFO for the file descriptor of either end of a pipe. We can test for a pipe with the `S_ISFIFO` macro.

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.

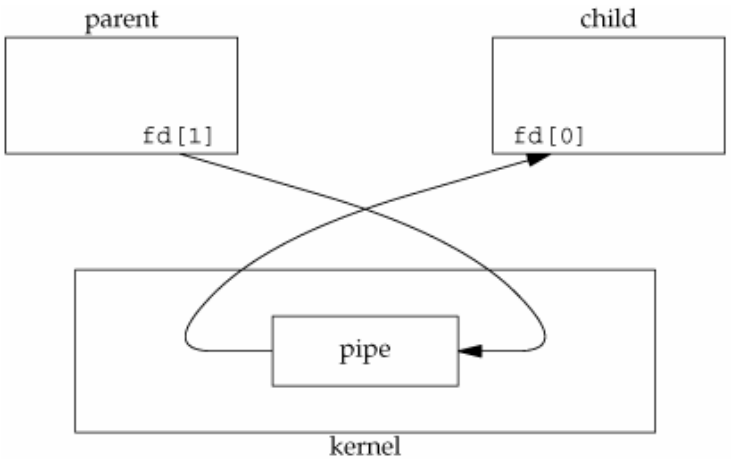
A pipe in a single process is next to useless. Normally, the process that calls `pipe` then calls `fork`, creating an IPC channel from the parent to the child or vice versa. Figure 15.3 shows this scenario.

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, the following 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 `write`s 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 2.11).

Example

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

Figure 15.5. Send data from parent to child over a 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);
}
```

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.)

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 a 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 a 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.

Figure 15.6. Copy file to pager program

```
#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);
}
```

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.

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.

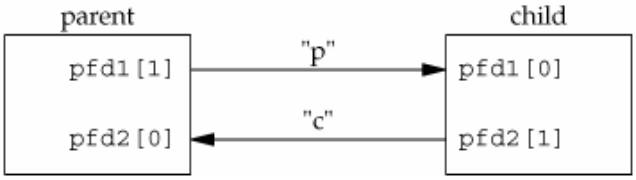
Note that each pipe has an extra reader, which doesn't matter. That is, in addition to the child reading from `pfd1[0]`, the parent also has this end of the top pipe open for reading. This doesn't affect us, since the parent doesn't try to read from this pipe.

Figure 15.7. Routines to let a parent and child synchronize

```
#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 15.8. Using two pipes for parent–child synchronization



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, `fork`ing 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.

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`.

Figure 15.11. Copy file to pager program using `popen`

```
#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);
}
```

Example—`popen` and `pclose` Functions

Figure 15.12 shows our version of `popen` and `pclose`.

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.

Calling pipe and fork and then duplicating the appropriate descriptors for each process 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.

Figure 15.12. The popen and pclose functions

```
#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 2.16.
 */
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;     /* required by POSIX */
        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 ((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 ((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 */
}
```

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 `popen`, we can interpose a program between the application and its input to transform the input. Figure 15.13 shows the arrangement of processes.

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.

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

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.

Figure 15.13. Transforming input using popen

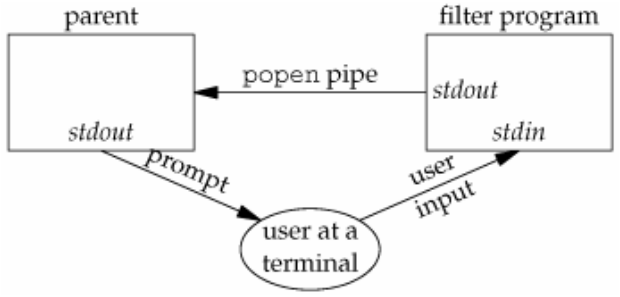


Figure 15.14. Filter to convert uppercase characters to lowercase

```
#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 15.15. Invoke uppercase/lowercase filter to read commands

```
#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);
```