Any (or all) of the middle three arguments to `select` (the pointers to the descriptor sets) can be null pointers if we're not interested in that condition. If all three pointers are `NULL`, then we have a higher precision timer than provided by `sleep`. (Recall from Section 10.19 that `sleep` waits for an integral number of seconds. With `select`, we can wait for intervals less than 1 second; the actual resolution depends on the system's clock.) Exercise 14.6 shows such a function.

The first argument to `select`, *maxfdp1*, stands for "maximum file descriptor plus 1." We calculate the highest descriptor that we're interested in, considering all three of the descriptor sets, add 1, and that's the first argument. We could just set the first argument to `FD_SETSIZE`, a constant in `<sys/select.h>` that specifies the maximum number of descriptors (often 1,024), but this value is too large for most applications. Indeed, most applications probably use between 3 and 10 descriptors. (Some applications need many more descriptors, but these UNIX programs are atypical.) By specifying the highest descriptor that we're interested in, we can prevent the kernel from going through hundreds of unused bits in the three descriptor sets, looking for bits that are turned on.

As an example, Figure 14.24 shows what two descriptor sets look like if we write

```
fd_set  readset, writeset;

FD_ZERO(&readset);
FD_ZERO(&writeset);
FD_SET(0, &readset);
FD_SET(3, &readset);
FD_SET(1, &writeset);
FD_SET(2, &writeset);
select(4, &readset, &writeset, NULL, NULL);
```

The reason we have to add 1 to the maximum descriptor number is that descriptors start at 0, and the first argument is really a count of the number of descriptors to check (starting with descriptor 0).

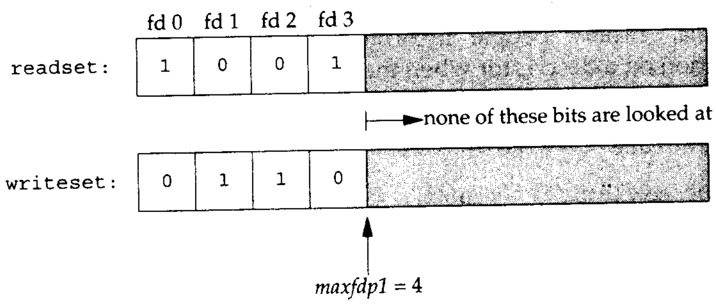

Figure 14.24 Example descriptor sets for `select`

There are three possible return values from `select`.

1. A return value of –1 means that an error occurred. This can happen, for example, if a signal is caught before any of the specified descriptors are ready. In this case, none of the descriptor sets will be modified.

2. A return value of 0 means that no descriptors are ready. This happens if the time limit expires before any of the descriptors are ready. When this happens, all the descriptor sets will be zeroed out.
3. A positive return value specifies the number of descriptors that are ready. This value is the sum of the descriptors ready in all three sets, so if the same descriptor is ready to be read *and* written, it will be counted twice in the return value. The only bits left on in the three descriptor sets are the bits corresponding to the descriptors that are ready.

We now need to be more specific about what "ready" means.

- A descriptor in the read set (*readfds*) is considered ready if a `read` from that descriptor won't block.
- A descriptor in the write set (*writefds*) is considered ready if a `write` to that descriptor won't block.
- A descriptor in the exception set (*exceptfds*) is considered ready if an exception condition is pending on that descriptor. Currently, an exception condition corresponds to either the arrival of out-of-band data on a network connection or certain conditions occurring on a pseudo terminal that has been placed into packet mode. (Section 15.10 of Stevens [1990] describes this latter condition.)
- File descriptors for regular files always return ready for reading, writing, and exception conditions.

It is important to realize that whether a descriptor is blocking or not doesn't affect whether `select` blocks. That is, if we have a nonblocking descriptor that we want to read from and we call `select` with a timeout value of 5 seconds, `select` will block for up to 5 seconds. Similarly, if we specify an infinite timeout, `select` blocks until data is ready for the descriptor or until a signal is caught.

If we encounter the end of file on a descriptor, that descriptor is considered readable by `select`. We then call `read` and it returns 0, the way to signify end of file on UNIX systems. (Many people incorrectly assume that `select` indicates an exception condition on a descriptor when the end of file is reached.)

POSIX.1 also defines a variant of `select` called `pselect`.

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

int pselect(int maxfdp1, fd_set *restrict readfds,
            fd_set *restrict writefds, fd_set *restrict exceptfds,
            const struct timespec *restrict tsptr,
            const sigset_t *restrict sigmask);
```

Returns: count of ready descriptors, 0 on timeout, –1 on error

The `pselect` function is identical to `select`, with the following exceptions.

- The timeout value for `select` is specified by a `timeval` structure, but for `pselect`, a `timespec` structure is used. (Recall the definition of the `timespec` structure in Section 11.6.) Instead of seconds and microseconds, the `timespec` structure represents the timeout value in seconds and nanoseconds. This provides a higher-resolution timeout if the platform supports that fine a level of granularity.

- The timeout value for `pselect` is declared `const`, and we are guaranteed that its value will not change as a result of calling `pselect`.
- An optional signal mask argument is available with `pselect`. If *sigmask* is null, `pselect` behaves as `select` does with respect to signals. Otherwise, *sigmask* points to a signal mask that is atomically installed when `pselect` is called. On return, the previous signal mask is restored.

14.5.2 `poll` Function

The `poll` function is similar to `select`, but the programmer interface is different. As we'll see, `poll` is tied to the STREAMS system, since it originated with System V, although we are able to use it with any type of file descriptor.

```
#include <poll.h>

int poll(struct pollfd fdarray[], nfds_t nfds, int timeout);

                    Returns: count of ready descriptors, 0 on timeout, –1 on error
```

With `poll`, instead of building a set of descriptors for each condition (readability, writability, and exception condition), as we did with `select`, we build an array of `pollfd` structures, with each array element specifying a descriptor number and the conditions that we're interested in for that descriptor:

```
struct pollfd {
  int    fd;        /* file descriptor to check, or <0 to ignore */
  short  events;    /* events of interest on fd */
  short  revents;   /* events that occurred on fd */
};
```

The number of elements in the *fdarray* array is specified by *nfds*.

Historically, there have been differences in how the *nfds* parameter was declared. SVR3 specified the number of elements in the array as an `unsigned long`, which seems excessive. In the SVR4 manual [AT&T 1990d], the prototype for `poll` showed the data type of the second argument as `size_t`. (Recall the primitive system data types, Figure 2.20.) But the actual prototype in the `<poll.h>` header still showed the second argument as an `unsigned long`. The Single UNIX Specification defines the new type `nfds_t` to allow the implementation to select the appropriate type and hide the details from applications. Note that this type has to be large enough to hold an integer, since the return value represents the number of entries in the array with satisfied events.

The SVID corresponding to SVR4 [AT&T 1989] showed the first argument to `poll` as `struct pollfd` *fdarray*`[]`, whereas the SVR4 manual page [AT&T 1990d] showed this argument as `struct pollfd` **fdarray*. In the C language, both declarations are equivalent. We use the first declaration to reiterate that `fdarray` points to an array of structures and not a pointer to a single structure.

To tell the kernel what events we're interested in for each descriptor, we have to set the `events` member of each array element to one or more of the values in Figure 14.25. On return, the `revents` member is set by the kernel, specifying which events have occurred for each descriptor. (Note that `poll` doesn't change the `events` member. This differs from `select`, which modifies its arguments to indicate what is ready.)

Name	Input to `events`?	Result from `revents`?	Description
POLLIN	•	•	Data other than high priority can be read without blocking (equivalent to POLLRDNORM | POLLRDBAND).
POLLRDNORM	•	•	Normal data (priority band 0) can be read without blocking.
POLLRDBAND	•	•	Data from a nonzero priority band can be read without blocking.
POLLPRI	•	•	High-priority data can be read without blocking.
POLLOUT	•	•	Normal data can be written without blocking.
POLLWRNORM	•	•	Same as POLLOUT.
POLLWRBAND	•	•	Data for a nonzero priority band can be written without blocking.
POLLERR		•	An error has occurred.
POLLHUP		•	A hangup has occurred.
POLLNVAL		•	The descriptor does not reference an open file.

Figure 14.25 The `events` and `revents` flags for `poll`

The first four rows of Figure 14.25 test for readability, the next three test for writability, and the final three are for exception conditions. The last three rows in Figure 14.25 are set by the kernel on return. These three values are returned in `revents` when the condition occurs, even if they weren't specified in the `events` field.

When a descriptor is hung up (POLLHUP), we can no longer write to the descriptor. There may, however, still be data to be read from the descriptor.

The final argument to `poll` specifies how long we want to wait. As with `select`, there are three cases.

timeout == −1

Wait forever. (Some systems define the constant `INFTIM` in `<stropts.h>` as −1.) We return when one of the specified descriptors is ready or when a signal is caught. If a signal is caught, `poll` returns −1 with `errno` set to `EINTR`.

timeout == 0

Don't wait. All the specified descriptors are tested, and we return immediately. This is a way to poll the system to find out the status of multiple descriptors, without blocking in the call to `poll`.

timeout > 0

Wait *timeout* milliseconds. We return when one of the specified descriptors is ready or when the *timeout* expires. If the *timeout* expires before any of the descriptors is ready, the return value is 0. (If your system doesn't provide millisecond resolution, *timeout* is rounded up to the nearest supported value.)

It is important to realize the difference between an end of file and a hangup. If we're entering data from the terminal and type the end-of-file character, `POLLIN` is turned on so we can read the end-of-file indication (`read` returns 0). `POLLHUP` is not turned on in `revents`. If we're reading from a modem and the telephone line is hung up, we'll receive the `POLLHUP` notification.

As with `select`, whether a descriptor is blocking or not doesn't affect whether `poll` blocks.

Interruptibility of `select` and `poll`

When the automatic restarting of interrupted system calls was introduced with 4.2BSD (Section 10.5), the `select` function was never restarted. This characteristic continues with most systems even if the `SA_RESTART` option is specified. But under SVR4, if `SA_RESTART` was specified, even `select` and `poll` were automatically restarted. To prevent this from catching us when we port software to systems derived from SVR4, we'll always use the `signal_intr` function (Figure 10.19) if the signal could interrupt a call to `select` or `poll`.

None of the implementations described in this book restart `poll` or `select` when a signal is received, even if the `SA_RESTART` flag is used.

14.6 Asynchronous I/O

Using `select` and `poll`, as described in the previous section, is a synchronous form of notification. The system doesn't tell us anything until we ask (by calling either `select` or `poll`). As we saw in Chapter 10, signals provide an asynchronous form of notification that something has happened. All systems derived from BSD and System V provide some form of asynchronous I/O, using a signal (`SIGPOLL` in System V; `SIGIO` in BSD) to notify the process that something of interest has happened on a descriptor.

We saw that `select` and `poll` work with any descriptors. But with asynchronous I/O, we now encounter restrictions. On systems derived from System V, asynchronous I/O works only with STREAMS devices and STREAMS pipes. On systems derived from BSD, asynchronous I/O works only with terminals and networks.

One limitation of asynchronous I/O is that there is only one signal per process. If we enable more than one descriptor for asynchronous I/O, we cannot tell which descriptor the signal corresponds to when the signal is delivered.

The Single UNIX Specification includes an optional generic asynchronous I/O mechanism, adopted from the real-time draft standard. It is unrelated to the mechanisms we describe here. This mechanism solves a lot of the limitations that exist with these older asynchronous I/O mechanisms, but we will not discuss it further.

14.6.1 System V Asynchronous I/O

In System V, asynchronous I/O is part of the STREAMS system and works only with STREAMS devices and STREAMS pipes. The System V asynchronous I/O signal is `SIGPOLL`.

To enable asynchronous I/O for a STREAMS device, we have to call `ioctl` with a second argument (*request*) of `I_SETSIG`. The third argument is an integer value formed from one or more of the constants in Figure 14.26. These constants are defined in `<stropts.h>`.

Constant	Description
`S_INPUT`	A message other than a high-priority message has arrived.
`S_RDNORM`	An ordinary message has arrived.
`S_RDBAND`	A message with a nonzero priority band has arrived.
`S_BANDURG`	If this constant is specified with `S_RDBAND`, the `SIGURG` signal is generated instead of `SIGPOLL` when a nonzero priority band message has arrived.
`S_HIPRI`	A high-priority message has arrived.
`S_OUTPUT`	The write queue is no longer full.
`S_WRNORM`	Same as `S_OUTPUT`.
`S_WRBAND`	We can send a nonzero priority band message.
`S_MSG`	A STREAMS signal message that contains the `SIGPOLL` signal has arrived.
`S_ERROR`	An `M_ERROR` message has arrived.
`S_HANGUP`	An `M_HANGUP` message has arrived.

Figure 14.26 Conditions for generating `SIGPOLL` signal

In Figure 14.26, whenever we say "has arrived," we mean "has arrived at the stream head's read queue."

In addition to calling `ioctl` to specify the conditions that should generate the `SIGPOLL` signal, we also have to establish a signal handler for this signal. Recall from Figure 10.1 that the default action for `SIGPOLL` is to terminate the process, so we should establish the signal handler before calling `ioctl`.

14.6.2 BSD Asynchronous I/O

Asynchronous I/O in BSD-derived systems is a combination of two signals: `SIGIO` and `SIGURG`. The former is the general asynchronous I/O signal, and the latter is used only to notify the process that out-of-band data has arrived on a network connection.

To receive the `SIGIO` signal, we need to perform three steps.

1. Establish a signal handler for `SIGIO`, by calling either `signal` or `sigaction`.
2. Set the process ID or process group ID to receive the signal for the descriptor, by calling `fcntl` with a command of `F_SETOWN` (Section 3.14).
3. Enable asynchronous I/O on the descriptor by calling `fcntl` with a command of `F_SETFL` to set the `O_ASYNC` file status flag (Figure 3.9).

Step 3 can be performed only on descriptors that refer to terminals or networks, which is a fundamental limitation of the BSD asynchronous I/O facility.

For the `SIGURG` signal, we need perform only steps 1 and 2. `SIGURG` is generated only for descriptors that refer to network connections that support out-of-band data.

14.7 `readv` and `writev` Functions

The `readv` and `writev` functions let us read into and write from multiple noncontiguous buffers in a single function call. These operations are called *scatter read* and *gather write*.

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

ssize_t readv(int filedes, const struct iovec *iov, int iovcnt);

ssize_t writev(int filedes, const struct iovec *iov, int iovcnt);
```

Both return: number of bytes read or written, –1 on error

The second argument to both functions is a pointer to an array of `iovec` structures:

```
struct iovec {
  void    *iov_base;   /* starting address of buffer */
  size_t  iov_len;     /* size of buffer */
};
```

The number of elements in the *iov* array is specified by *iovcnt*. It is limited to `IOV_MAX` (Recall Figure 2.10). Figure 14.27 shows a picture relating the arguments to these two functions and the `iovec` structure.

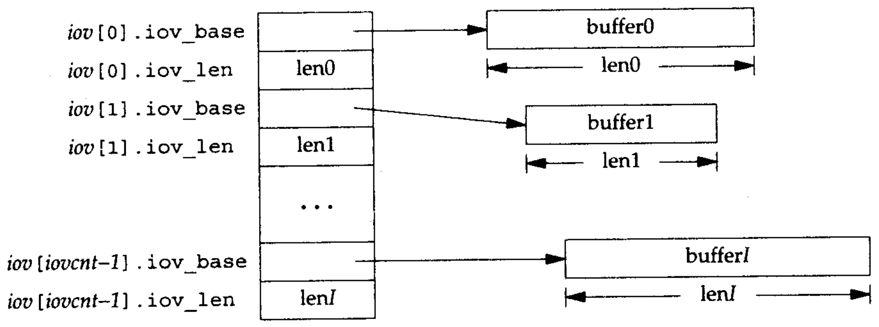

Figure 14.27 The `iovec` structure for `readv` and `writev`

The `writev` function gathers the output data from the buffers in order: *iov[0]*, *iov[1]*, through *iov[iovcnt–1]*; `writev` returns the total number of bytes output, which should normally equal the sum of all the buffer lengths.

The `readv` function scatters the data into the buffers in order, always filling one buffer before proceeding to the next. `readv` returns the total number of bytes that were read. A count of 0 is returned if there is no more data and the end of file is encountered.

These two functions originated in 4.2BSD and were later added to SVR4. These two functions are included in the XSI extension of the Single UNIX Specification.

Although the Single UNIX Specification defines the buffer address to be a `void *`, many implementations that predate the standard still use a `char *` instead.

Example

In Section 20.8, in the function `_db_writeidx`, we need to write two buffers consecutively to a file. The second buffer to output is an argument passed by the caller, and the first buffer is one we create, containing the length of the second buffer and a file offset of other information in the file. There are three ways we can do this.

1. Call `write` twice, once for each buffer.
2. Allocate a buffer of our own that is large enough to contain both buffers, and copy both into the new buffer. We then call `write` once for this new buffer.
3. Call `writev` to output both buffers.

The solution we use in Section 20.8 is to use `writev`, but it's instructive to compare it to the other two solutions.

Figure 14.28 shows the results from the three methods just described.

	Linux (Intel x86)			Mac OS X (PowerPC)		
Operation	User	System	Clock	User	System	Clock
two `writes`	1.29	3.15	7.39	1.60	17.40	19.84
buffer copy, then one `write`	1.03	1.98	6.47	1.10	11.09	12.54
one `writev`	0.70	2.72	6.41	0.86	13.58	14.72

Figure 14.28 Timing results comparing `writev` and other techniques

The test program that we measured output a 100-byte header followed by 200 bytes of data. This was done 1,048,576 times, generating a 300-megabyte file. The test program has three separate cases—one for each of the techniques measured in Figure 14.28. We used `times` (Section 8.16) to obtain the user CPU time, system CPU time, and wall clock time before and after the writes. All three times are shown in seconds.

As we expect, the system time increases when we call `write` twice, compared to calling either `write` or `writev` once. This correlates with the results in Figure 3.5.

Next, note that the sum of the CPU times (user plus system) is less when we do a buffer copy followed by a single `write` compared to a single call to `writev`. With the single `write`, we copy the buffers to a staging buffer at user level, and then the kernel will copy the data to its internal buffers when we call `write`. With `writev`, we should do less copying, because the kernel only needs to copy the data directly into its staging buffers. The fixed cost of using `writev` for such small amounts of data, however, is greater than the benefit. As the amount of data we need to copy increases, the more expensive it will be to copy the buffers in our program, and the `writev` alternative will be more attractive.

Be careful not to infer too much about the relative performance of Linux to Mac OS X from the numbers shown in Figure 14.28. The two computers were very different: they had different processor architectures, different amounts of RAM, and disks with different speeds. To do an apples-to-apples comparison of one operating system to another, we need to use the same hardware for each operating system.

□

In summary, we should always try to use the fewest number of system calls necessary to get the job done. If we are writing small amounts of data, we will find it less expensive to copy the data ourselves and use a single `write` instead of using `writev`. We might find, however, that the performance benefits aren't worth the extra complexity cost needed to manage our own staging buffers.

14.8 `readn` and `writen` Functions

Pipes, FIFOs, and some devices, notably terminals, networks, and STREAMS devices, have the following two properties.

1. A `read` operation may return less than asked for, even though we have not encountered the end of file. This is not an error, and we should simply continue reading from the device.
2. A `write` operation can also return less than we specified. This may be caused by flow control constraints by downstream modules, for example. Again, it's not an error, and we should continue writing the remainder of the data. (Normally, this short return from a `write` occurs only with a nonblocking descriptor or if a signal is caught.)

We'll never see this happen when reading or writing a disk file, except when the file system runs out of space or we hit our quota limit and we can't write all that we requested.

Generally, when we read from or write to a pipe, network device, or terminal, we need to take these characteristics into consideration. We can use the following two functions to read or write *N* bytes of data, letting these functions handle a possible return value that's less than requested. These two functions simply call `read` or `write` as many times as required to read or write the entire *N* bytes of data.

```
#include "apue.h"

ssize_t readn(int filedes, void *buf, size_t nbytes);

ssize_t writen(int filedes, void *buf, size_t nbytes);
```

Both return: number of bytes read or written, −1 on error

We define these functions as a convenience for later examples, similar to the error-handling routines used in many of the examples in this text. The `readn` and `writen` functions are not part of any standard.

We call `writen` whenever we're writing to one of the file types that we mentioned, but we call `readn` only when we know ahead of time that we will be receiving a certain number of bytes. Figure 14.29 shows implementations of `readn` and `writen` that we will use in later examples.

Note that if we encounter an error and have previously read or written any data, we return the amount of data transferred instead of the error. Similarly, if we reach end of file while reading, we return the number of bytes copied to the caller's buffer if we already read some data successfully and have not yet satisfied the amount requested.

```
#include "apue.h"

ssize_t             /* Read "n" bytes from a descriptor  */
readn(int fd, void *ptr, size_t n)
{
    size_t      nleft;
    ssize_t     nread;

    nleft = n;
    while (nleft > 0) {
        if ((nread = read(fd, ptr, nleft)) < 0) {
            if (nleft == n)
                return(-1); /* error, return -1 */
            else
                break;      /* error, return amount read so far */
        } else if (nread == 0) {
            break;          /* EOF */
        }
        nleft -= nread;
        ptr   += nread;
    }
    return(n - nleft);      /* return >= 0 */
}

ssize_t             /* Write "n" bytes to a descriptor  */
writen(int fd, const void *ptr, size_t n)
{
    size_t      nleft;
    ssize_t     nwritten;

    nleft = n;
    while (nleft > 0) {
        if ((nwritten = write(fd, ptr, nleft)) < 0) {
            if (nleft == n)
                return(-1); /* error, return -1 */
            else
                break;      /* error, return amount written so far */
        } else if (nwritten == 0) {
            break;
        }
        nleft -= nwritten;
        ptr   += nwritten;
    }
    return(n - nleft);      /* return >= 0 */
}
```

Figure 14.29 The readn and writen functions

14.9 Memory-Mapped I/O

Memory-mapped I/O lets us map a file on disk into a buffer in memory so that, when we fetch bytes from the buffer, the corresponding bytes of the file are read. Similarly, when we store data in the buffer, the corresponding bytes are automatically written to the file. This lets us perform I/O without using `read` or `write`.

> Memory-mapped I/O has been in use with virtual memory systems for many years. In 1981, 4.1BSD provided a different form of memory-mapped I/O with its `vread` and `vwrite` functions. These two functions were then removed in 4.2BSD and were intended to be replaced with the `mmap` function. The `mmap` function, however, was not included with 4.2BSD (for reasons described in Section 2.5 of McKusick et al. [1996]). Gingell, Moran, and Shannon [1987] describe one implementation of `mmap`. The `mmap` function is included in the memory-mapped files option in the Single UNIX Specification and is required on all XSI-conforming systems; most UNIX systems support it.

To use this feature, we have to tell the kernel to map a given file to a region in memory. This is done by the `mmap` function.

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

void *mmap(void *addr, size_t len, int prot, int flag, int filedes,
           off_t off);

      Returns: starting address of mapped region if OK, MAP_FAILED on error
```

The *addr* argument lets us specify the address of where we want the mapped region to start. We normally set this to 0 to allow the system to choose the starting address. The return value of this function is the starting address of the mapped area.

The *filedes* argument is the file descriptor specifying the file that is to be mapped. We have to open this file before we can map it into the address space. The *len* argument is the number of bytes to map, and *off* is the starting offset in the file of the bytes to map. (Some restrictions on the value of *off* are described later.)

The *prot* argument specifies the protection of the mapped region.

prot	Description
`PROT_READ`	Region can be read.
`PROT_WRITE`	Region can be written.
`PROT_EXEC`	Region can be executed.
`PROT_NONE`	Region cannot be accessed.

Figure 14.30 Protection of memory-mapped region

We can specify the protection as either `PROT_NONE` or the bitwise OR of any combination of `PROT_READ`, `PROT_WRITE`, and `PROT_EXEC`. The protection specified for a region can't allow more access than the open mode of the file. For example, we can't specify `PROT_WRITE` if the file was opened read-only.

Before looking at the *flag* argument, let's see what's going on here. Figure 14.31 shows a memory-mapped file. (Recall the memory layout of a typical process,

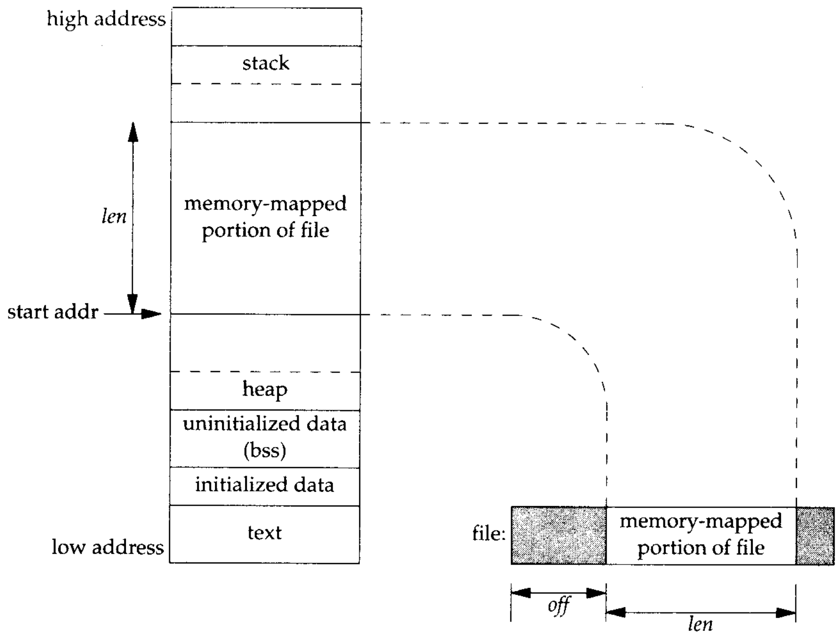

Figure 14.31 Example of a memory-mapped file

Figure 7.6.) In this figure, "start addr" is the return value from mmap. We have shown the mapped memory being somewhere between the heap and the stack: this is an implementation detail and may differ from one implementation to the next.

The *flag* argument affects various attributes of the mapped region.

MAP_FIXED	The return value must equal *addr*. Use of this flag is discouraged, as it hinders portability. If this flag is not specified and if *addr* is nonzero, then the kernel uses *addr* as a hint of where to place the mapped region, but there is no guarantee that the requested address will be used. Maximum portability is obtained by specifying *addr* as 0. Support for the MAP_FIXED flag is optional on POSIX-conforming systems, but required on XSI-conforming systems.
MAP_SHARED	This flag describes the disposition of store operations into the mapped region by this process. This flag specifies that store operations modify the mapped file—that is, a store operation is equivalent to a write to the file. Either this flag or the next (MAP_PRIVATE), but not both, must be specified.
MAP_PRIVATE	This flag says that store operations into the mapped region cause a private copy of the mapped file to be created. All successive

references to the mapped region then reference the copy. (One use of this flag is for a debugger that maps the text portion of a program file but allows the user to modify the instructions. Any modifications affect the copy, not the original program file.)

Each implementation has additional MAP_xxx flag values, which are specific to that implementation. Check the mmap(2) manual page on your system for details.

The value of *off* and the value of *addr* (if MAP_FIXED is specified) are required to be multiples of the system's virtual memory page size. This value can be obtained from the sysconf function (Section 2.5.4) with an argument of _SC_PAGESIZE or _SC_PAGE_SIZE. Since *off* and *addr* are often specified as 0, this requirement is not a big deal.

Since the starting offset of the mapped file is tied to the system's virtual memory page size, what happens if the length of the mapped region isn't a multiple of the page size? Assume that the file size is 12 bytes and that the system's page size is 512 bytes. In this case, the system normally provides a mapped region of 512 bytes, and the final 500 bytes of this region are set to 0. We can modify the final 500 bytes, but any changes we make to them are not reflected in the file. Thus, we cannot append to a file with mmap. We must first grow the file, as we will see in Figure 14.32.

Two signals are normally used with mapped regions. SIGSEGV is the signal normally used to indicate that we have tried to access memory that is not available to us. This signal can also be generated if we try to store into a mapped region that we specified to mmap as read-only. The SIGBUS signal can be generated if we access a portion of the mapped region that does not make sense at the time of the access. For example, assume that we map a file using the file's size, but before we reference the mapped region, the file's size is truncated by some other process. If we then try to access the memory-mapped region corresponding to the end portion of the file that was truncated, we'll receive SIGBUS.

A memory-mapped region is inherited by a child across a fork (since it's part of the parent's address space), but for the same reason, is not inherited by the new program across an exec.

We can change the permissions on an existing mapping by calling mprotect.

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

int mprotect(void *addr, size_t len, int prot);

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

The legal values for *prot* are the same as those for mmap (Figure 14.30). The address argument must be an integral multiple of the system's page size.

The mprotect function is included as part of the memory protection option in the Single UNIX Specification, but all XSI-conforming systems are required to support it.

If the pages in a shared mapping have been modified, we can call msync to flush the changes to the file that backs the mapping. The msync function is similar to fsync (Section 3.13), but works on memory-mapped regions.

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

int msync(void *addr, size_t len, int flags);

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

If the mapping is private, the file mapped is not modified. As with the other memory-mapped functions, the address must be aligned on a page boundary.

The *flags* argument allows us some control over how the memory is flushed. We can specify the MS_ASYNC flag to simply schedule the pages to be written. If we want to wait for the writes to complete before returning, we can use the MS_SYNC flag. Either MS_ASYNC or MS_SYNC must be specified.

An optional flag, MS_INVALIDATE, lets us tell the operating system to discard any pages that are out of sync with the underlying storage. Some implementations will discard all pages in the specified range when we use this flag, but this behavior is not required.

A memory-mapped region is automatically unmapped when the process terminates or by calling munmap directly. Closing the file descriptor *filedes* does not unmap the region.

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

int munmap(caddr_t addr, size_t len);

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

munmap does not affect the object that was mapped—that is, the call to munmap does not cause the contents of the mapped region to be written to the disk file. The updating of the disk file for a MAP_SHARED region happens automatically by the kernel's virtual memory algorithm as we store into the memory-mapped region. Modifications to memory in a MAP_PRIVATE region are discarded when the region is unmapped.

Example

The program in Figure 14.32 copies a file (similar to the cp(1) command) using memory-mapped I/O.

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

int
main(int argc, char *argv[])
{
    int         fdin, fdout;
    void        *src, *dst;
    struct stat statbuf;

    if (argc != 3)
```

```
        err_quit("usage: %s <fromfile> <tofile>", argv[0]);

    if ((fdin = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s for reading", argv[1]);

    if ((fdout = open(argv[2], O_RDWR | O_CREAT | O_TRUNC,
      FILE_MODE)) < 0)
        err_sys("can't creat %s for writing", argv[2]);

    if (fstat(fdin, &statbuf) < 0)   /* need size of input file */
        err_sys("fstat error");

    /* set size of output file */
    if (lseek(fdout, statbuf.st_size - 1, SEEK_SET) == -1)
        err_sys("lseek error");
    if (write(fdout, "", 1) != 1)
        err_sys("write error");

    if ((src = mmap(0, statbuf.st_size, PROT_READ, MAP_SHARED,
      fdin, 0)) == MAP_FAILED)
        err_sys("mmap error for input");

    if ((dst = mmap(0, statbuf.st_size, PROT_READ | PROT_WRITE,
      MAP_SHARED, fdout, 0)) == MAP_FAILED)
        err_sys("mmap error for output");

    memcpy(dst, src, statbuf.st_size);  /* does the file copy */
    exit(0);
}
```

Figure 14.32 Copy a file using memory-mapped I/O

We first open both files and then call `fstat` to obtain the size of the input file. We need this size for the call to mmap for the input file, and we also need to set the size of the output file. We call `lseek` and then `write` one byte to set the size of the output file. If we don't set the output file's size, the call to mmap for the output file is OK, but the first reference to the associated memory region generates SIGBUS. We might be tempted to use `ftruncate` to set the size of the output file, but not all systems extend the size of a file with this function. (See Section 4.13.)

Extending a file with `ftruncate` works on the four platforms discussed in this text.

We then call mmap for each file, to map the file into memory, and finally call memcpy to copy from the input buffer to the output buffer. As the bytes of data are fetched from the input buffer (`src`), the input file is automatically read by the kernel; as the data is stored in the output buffer (`dst`), the data is automatically written to the output file.

Exactly when the data is written to the file is dependent on the system's page management algorithms. Some systems have daemons that write dirty pages to disk slowly over time. If we want to ensure that the data is safely written to the file, we need to call msync with the MS_SYNC flag before exiting.

Let's compare this memory-mapped file copy to a copy that is done by calling `read` and `write` (with a buffer size of 8,192). Figure 14.33 shows the results. The times are given in seconds, and the size of the file being copied was 300 megabytes.

Operation	Linux 2.4.22 (Intel x86)			Solaris 9 (SPARC)		
	User	System	Clock	User	System	Clock
`read/write`	0.04	1.02	39.76	0.18	9.70	41.66
`mmap/memcpy`	0.64	1.31	24.26	1.68	7.94	28.53

Figure 14.33 Timing results comparing `read/write` versus `mmap/memcpy`

For Solaris 9, the total CPU time (user + system) is almost the same for both types of copies: 9.88 seconds versus 9.62 seconds. For Linux 2.4.22, the total CPU time is almost doubled when we use `mmap` and `memcpy` (1.06 seconds versus 1.95 seconds). The difference is probably because the two systems implement process time accounting differently.

As far as elapsed time is concerned, the version with `mmap` and `memcpy` is faster than the version with `read` and `write`. This makes sense, because we're doing less work with `mmap` and `memcpy`. With `read` and `write`, we copy the data from the kernel's buffer to the application's buffer (`read`), and then copy the data from the application's buffer to the kernel's buffer (`write`). With `mmap` and `memcpy`, we copy the data directly from one kernel buffer mapped into our address space into another kernel buffer mapped into our address space. □

Memory-mapped I/O is faster when copying one regular file to another. There are limitations. We can't use it to copy between certain devices (such as a network device or a terminal device), and we have to be careful if the size of the underlying file could change after we map it. Nevertheless, some applications can benefit from memory-mapped I/O, as it can often simplify the algorithms, since we manipulate memory instead of reading and writing a file. One example that can benefit from memory-mapped I/O is the manipulation of a frame buffer device that references a bit-mapped display.

Krieger, Stumm, and Unrau [1992] describe an alternative to the standard I/O library (Chapter 5) that uses memory-mapped I/O.

We return to memory-mapped I/O in Section 15.9, showing an example of how it can be used to provide shared memory between related processes.

14.10 Summary

In this chapter, we've described numerous advanced I/O functions, most of which are used in the examples in later chapters:

- Nonblocking I/O—issuing an I/O operation without letting it block
- Record locking (which we'll look at in more detail through an example, the database library in Chapter 20)

- System V STREAMS (which we'll need in Chapter 17 to understand STREAMS-based pipes, passing file descriptors, and System V client–server connections)
- I/O multiplexing—the `select` and `poll` functions (we'll use these in many of the later examples)
- The `readv` and `writev` functions (also used in many of the later examples)
- Memory-mapped I/O (`mmap`)

Exercises

14.1 Write a test program that illustrates your system's behavior when a process is blocked trying to write-lock a range of a file and additional read-lock requests are made. Is the process requesting a write lock starved by the processes read-locking the file?

14.2 Take a look at your system's headers and examine the implementation of `select` and the four `FD_` macros.

14.3 The system headers usually have a built-in limit on the maximum number of descriptors that the `fd_set` data type can handle. Assume that we need to increase this limit to handle up to 2,048 descriptors. How can we do this?

14.4 Compare the functions provided for signal sets (Section 10.11) and the `fd_set` descriptor sets. Also compare the implementation of the two on your system.

14.5 How many types of information does `getmsg` return?

14.6 Implement the function `sleep_us`, which is similar to `sleep`, but waits for a specified number of microseconds. Use either `select` or `poll`. Compare this function to the BSD `usleep` function.

14.7 Can you implement the functions `TELL_WAIT`, `TELL_PARENT`, `TELL_CHILD`, `WAIT_PARENT`, and `WAIT_CHILD` from Figure 10.24 using advisory record locking instead of signals? If so, code and test your implementation.

14.8 Determine the capacity of a pipe using nonblocking writes. Compare this value with the value of `PIPE_BUF` from Chapter 2.

14.9 Recall Figure 14.28. Determine the break-even point on your system where using `writev` is faster than copying the data yourself and using a single `write`.

14.10 Run the program in Figure 14.32 to copy a file and determine whether the last-access time for the input file is updated.

14.11 In the program from Figure 14.32, `close` the input file after calling `mmap` to verify that closing the descriptor does not invalidate the memory-mapped I/O.

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.

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

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		•

Figure 15.1 Summary of UNIX System IPC

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.

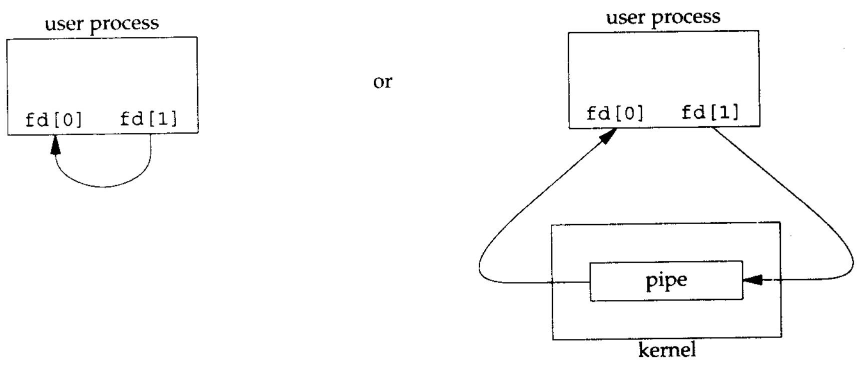

Figure 15.2 Two ways to view a half-duplex pipe

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.

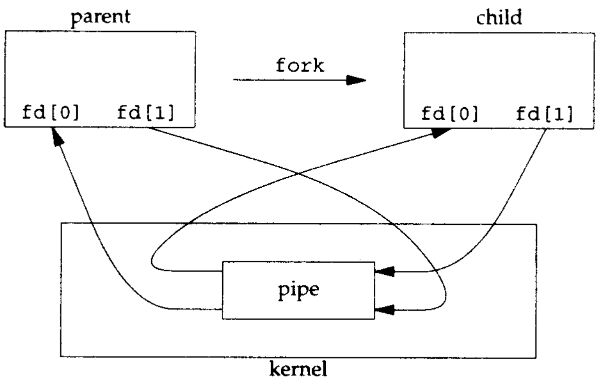

Figure 15.3 Half-duplex pipe after a `fork`

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.

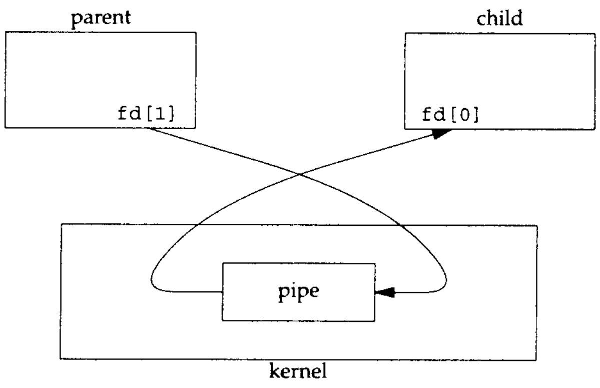

Figure 15.4 Pipe from parent to child

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 generated. If we either ignore the signal or catch it and return from 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 2.11).

Example

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

```
#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 □

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);                                        /* child */
    } else {
        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 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. □

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.

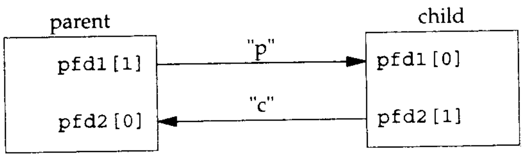

Figure 15.8 Using two pipes for parent–child synchronization

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 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. □

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).

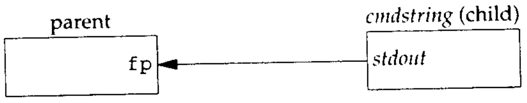

Figure 15.9 Result of `fp = popen(cmdstring, "r")`

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

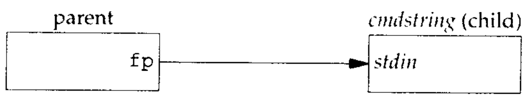

Figure 15.10 Result of `fp = popen(`*cmdstring*`, "w")`

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.

```
#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 2.16.
 */
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 */
}
```

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.

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.

□

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.

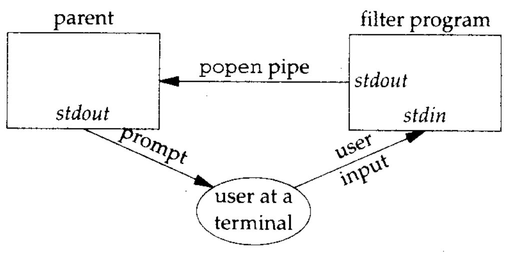

Figure 15.13 Transforming input using `popen`

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.

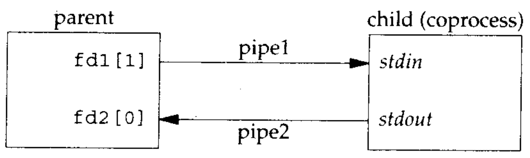

Figure 15.16 Driving a coprocess by writing its standard input and reading its standard output

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.)

Recall from Figure 15.1 that not all systems provide full-duplex pipes using the `pipe` function. In Figure 17.4, we provide another version of this example using a single full-duplex pipe instead of two half-duplex pipes, for those systems that support full-duplex pipes. □

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 back 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. Pipes can be used only between related processes when a common ancestor has created the pipe. (An exception to this is mounted STREAMS-based pipes, which we discuss in Section 17.2.2.) 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 *pathname, mode_t mode);

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

The specification of the *mode* argument for the `mkfifo` function 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.

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

Applications can create FIFOs with the mknod function. Because POSIX.1 originally didn't include mknod, the mkfifo function was invented specifically for POSIX.1. The mknod function is now included as an XSI extension. On most systems, the mkfifo function calls mknod to create the FIFO.

POSIX.1 also includes support for the mkfifo(1) command. All four platforms discussed in this text provide this command. This allows a FIFO to be created using a shell command and then accessed with the normal shell I/O redirection.

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

- In the normal case (O_NONBLOCK not specified), an open for read-only blocks until some other process opens the FIFO for writing. Similarly, an open for write-only 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. (We'll see a way around this problem in Section 17.2.2.) 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 to another without creating intermediate temporary files.
2. FIFOs are used as rendezvous points in client–server applications to pass data between 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.

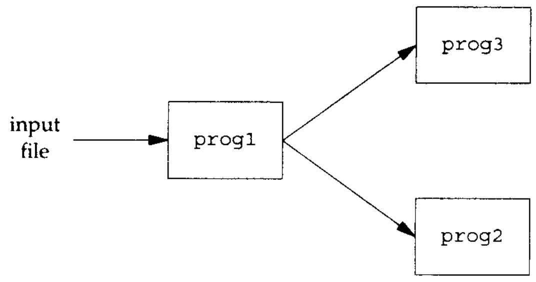

Figure 15.20 Procedure that processes a filtered input stream twice

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 to 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.

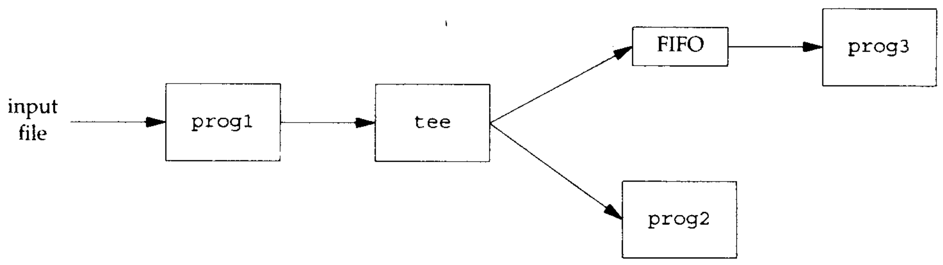

Figure 15.21 Using a FIFO and tee to send a stream to two different processes

□

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 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. 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 writes.

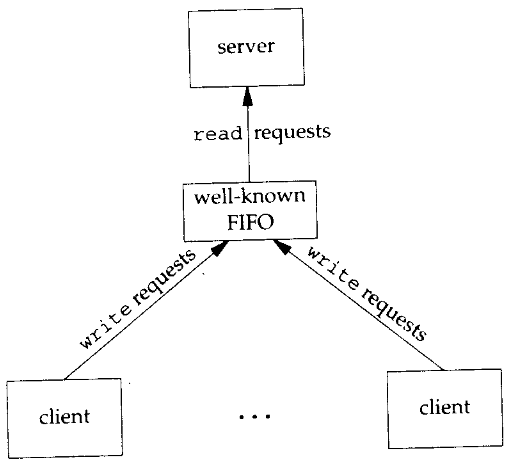

Figure 15.22 Clients sending requests to a server using a FIFO

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. Figure 15.23 shows this arrangement.

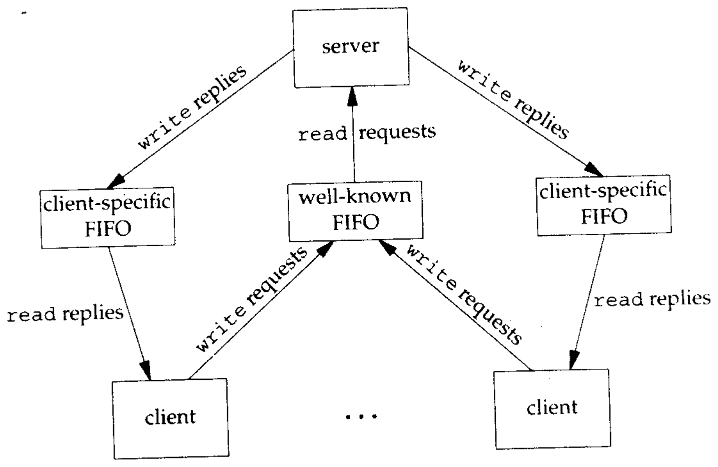

Figure 15.23 Client–server communication using FIFOs

This arrangement works, although it is impossible for the server to tell whether a client crashes. This causes the client-specific FIFOs to be left in the file system. The 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. We'll see a more elegant approach to this problem when we discuss mounted STREAMS-based pipes and connld in Section 17.2.2.

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.

Recall from Figure 15.1 that message queues, semaphores, and shared memory are defined as XSI extensions in the Single UNIX Specification.

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 or fetch a message to or 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_
   and store the returned identifier somewhere (such as a file) for the
   obtain. The key IPC_PRIVATE guarantees that the server creates a new IPC structure. The disadvantage to 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 common header, 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 ID is 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, there can be information loss 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, however, 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 5.2.1, Linux 2.4.22, and Mac OS X 10.3 provide the `sysctl` command to view and modify kernel configuration parameters. On Solaris 9, changes to kernel configuration parameters are made by modifying the file `/etc/system` and rebooting.

On Linux, you can display the IPC-related limits by running `ipcs -l`. On FreeBSD, 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, record oriented, and can be processed in other than first-in, first-out order. As we saw in Section 14.4, the STREAMS mechanism also possesses all these properties, although an `open` is required before sending data to a stream, and a `close` is required when we're finished. 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 17.3.) 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, 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 STREAMS and 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 includes an alternate IPC message queue implementation in the message-passing option of its real-time extensions. We do not cover the real-time extensions 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.

Figure 15.26 lists the system limits that affect message queues. We show "notsup" where the platform doesn't support the feature. We show "derived" whenever a limit is derived from other limits. For example, the maximum number of messages in a Linux system is based on the maximum number of queues and the maximum amount of data allowed on the queues. If the minimum message size is 1 byte, that would limit the number of messages systemwide to *maximum # queues * maximum size of a queue*. Given the limits in Figure 15.26, Linux has an upper bound of 262,144 messages with the default configuration. (Even though a message can contain zero bytes of data, Linux treats it as if it contained 1 byte, to limit the number of messages queued.)

Description	Typical values			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
Size in bytes of largest message we can send	16,384	8,192	notsup	2,048
The maximum size in bytes of a particular queue (i.e., the sum of all the messages on the queue)	2,048	16,384	notsup	4,096
The maximum number of messages queues, systemwide	40	16	notsup	50
The maximum number of messages, systemwide	40	derived	notsup	40

Figure 15.26 System limits that affect message queues

Recall from Figure 15.1 that Mac OS X 10.3 doesn't support XSI message queues. Since Mac OS X is based in part on FreeBSD, and FreeBSD supports message queues, it is possible for Mac OS X to support them, too. Indeed, a good Internet search engine will provide pointers to a third-party port of XSI message queues for Mac OS X.

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 versus Stream 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 17.3), 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, STREAMS-based 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 STREAMS-based pipe and UNIX domain socket. The times are all in seconds.

Operation	User	System	Clock
message queue	0.57	3.63	4.22
STREAMS pipe	0.50	3.21	3.71
UNIX domain socket	0.43	4.45	5.59

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 (in fact, STREAMS-based pipes are faster than message queues). (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 alternate set of semaphore interfaces in the semaphore option of its real-time extensions. We do not discuss these interfaces in this text.

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. In this 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 the semaphore 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 to define 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 using them, 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 (Section 15.6.3) that affect semaphore sets.

Description	Typical values			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
The maximum value of any semaphore	32,767	32,767	32,767	32,767
The maximum value of any semaphore's adjust-on-exit value	16,384	32,767	16,384	16,384
The maximum number of semaphore sets, systemwide	10	128	87,381	10
The maximum number of semaphores, systemwide	60	32,000	87,381	60
The maximum number of semaphores per semaphore set	60	250	87,381	25
The maximum number of undo structures, systemwide	30	32,000	87,381	30
The maximum number of undo entries per undo structures	10	32	10	10
The maximum number of operations per semop call	100	32	100	10

Figure 15.28 System limits that affect semaphores

The first function to call is `semget` to obtain a semaphore ID.

```
#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 in 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.

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.

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 that the resulting value for the semaphore 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 this case, 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 versus Record Locking

If we are sharing a single resource among multiple processes, we can use either a semaphore or record locking. It's interesting to compare the timing differences between the two 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 properties of record locking guarantee that if a process terminates while holding a lock, then the lock is automatically released by the kernel.

Figure 15.29 shows the time required to perform these two locking techniques on Linux. In each case, the resource was allocated and then released 100,000 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	0.38	0.48	0.86
advisory record locking	0.41	0.95	1.36

Figure 15.29 Timing comparison of locking alternatives on Linux

On Linux, there is about a 60 percent penalty in the elapsed time for record locking compared to semaphore locking.

Even though record locking is slower than semaphore locking, if we're locking a single resource (such as a shared memory segment) and don't need all the fancy features of XSI semaphores, record locking is preferred. The reasons are that it is much simpler to use, and the system takes care of any lingering locks when a process terminates. □

15.9 Shared Memory

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 can also be used.)

The Single UNIX Specification includes an alternate set of interfaces to access shared memory in the shared memory objects option of its real-time extensions. We do not cover the real-time extensions in this text.

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 */
  .
  .
  .
};
```

(Each implementation adds other structure members as needed 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 (Section 15.6.3) that affect shared memory.

Description	Typical values			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
The maximum size in bytes of a shared memory segment	33,554,432	33,554,432	4,194,304	8,388,608
The minimum size in bytes of a shared memory segment	1	1	1	1
The maximum number of shared memory segments, systemwide	192	4,096	32	100
The maximum number of shared memory segments, per process	128	4,096	8	6

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_nattach`, `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 the 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 in 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 read-only. Otherwise, the segment is attached 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(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 %lx to %lx\n", (unsigned long)&array[0],
      (unsigned long)&array[ARRAY_SIZE]);
    printf("stack around %lx\n", (unsigned long)&shmid);

    if ((ptr = malloc(MALLOC_SIZE)) == NULL)
        err_sys("malloc error");
    printf("malloced from %lx to %lx\n", (unsigned long)ptr,
      (unsigned long)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 %lx to %lx\n",
      (unsigned long)shmptr, (unsigned long)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 an Intel-based Linux system gives us the following output:

```
$ ./a.out
array[] from 804a080 to 8053cc0
stack around bffff9e4
malloced from 8053cc8 to 806c368
shared memory attached from 40162000 to 4017a6a0
```

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. □

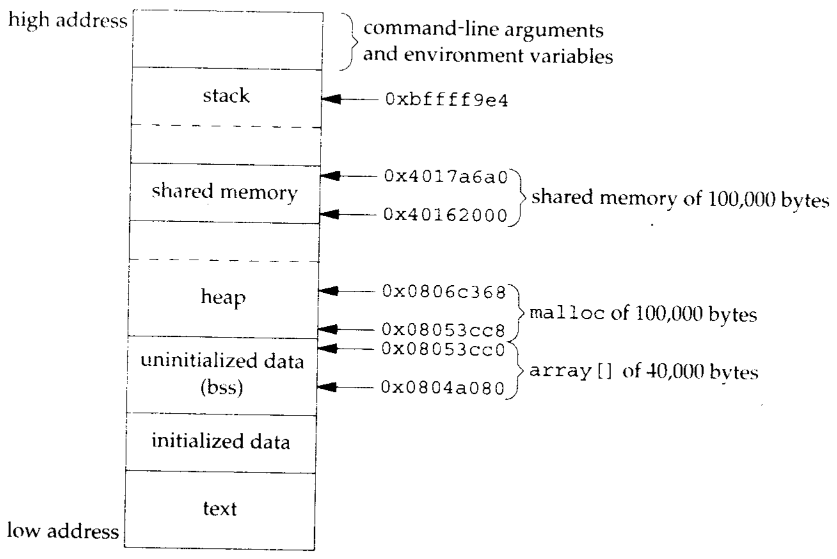

Figure 15.32 Memory layout on an Intel-based Linux system

Recall that the `mmap` function (Section 14.9) 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 5.2.1, Linux 2.4.22, and Solaris 9. Mac OS X 10.3 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 regardless of 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 Client–Server Properties

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. 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 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. 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 type field 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. Before sending 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 17.3 to allow the server to identify the client. The same technique can be used with FIFOs, message queues, semaphores, or 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 also 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 17.2.2 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 STREAMS subsystem when file descriptors are passed between processes.

15.11 Summary

We've detailed numerous forms of interprocess communication: pipes, named pipes (FIFOs), and the three forms of IPC commonly called XSI IPC (message queues, semaphores, and shared memory). 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 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 14.9).

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`.

15.6 The Rationale for POSIX.1 gives as one of the reasons for adding the `waitpid` function that most pre-POSIX.1 systems can't handle the following:

```
if ((fp = popen("/bin/true", "r")) == NULL)
    ...
if ((rc = system("sleep 100")) == -1)
    ...
if (pclose(fp) == -1)
    ...
```

What happens in this code if `waitpid` isn't available and `wait` is used instead?

15.7 Explain how `select` and `poll` handle an input descriptor that is a pipe, when the pipe is closed by the writer. To determine the answer, write two small test programs: one using `select` and one using `poll`.

Redo this exercise, looking at an output descriptor that is a pipe, when the read end is closed.

15.8 What happens if the *cmdstring* executed by popen with a *type* of `"r"` writes to its standard error?

15.9 Since popen invokes a shell to execute its *cmdstring* argument, what happens when `cmdstring` terminates? (Hint: draw all the processes involved.)

15.10 POSIX.1 specifically states that opening a FIFO for read–write is undefined. Although most UNIX systems allow this, show another method for opening a FIFO for both reading and writing, without blocking.

15.11 Unless a file contains sensitive or confidential data, allowing other users to read the file causes no harm. (It is usually considered antisocial, however, to go snooping around in other people's files.) But what happens if a malicious process reads a message from a message queue that is being used by a server and several clients? What information does the malicious process need to know to read the message queue?

15.12 Write a program that does the following. Execute a loop five times: create a message queue, print the queue identifier, delete the message queue. Then execute the next loop five times: create a message queue with a key of `IPC_PRIVATE`, and place a message on the queue. After the program terminates, look at the message queues using `ipcs`(1). Explain what is happening with the queue identifiers.

15.13 Describe how to build a linked list of data objects in a shared memory segment. What would you store as the list pointers?

15.14 Draw a time line of the program in Figure 15.33 showing the value of the variable `i` in both the parent and child, the value of the long integer in the shared memory region, and the value returned by the `update` function. Assume that the child runs first after the `fork`.

15.15 Redo the program in Figure 15.33 using the XSI shared memory functions from Section 15.9 instead of the shared memory-mapped region.

15.16 Redo the program in Figure 15.33 using the XSI semaphore functions from Section 15.8 to alternate between the parent and the child.

15.17 Redo the program in Figure 15.33 using advisory record locking to alternate between the parent and the child.

16

Network IPC: Sockets

16.1 Introduction

In the previous chapter, we looked at pipes, FIFOs, message queues, semaphores, and shared memory: the classical methods of IPC provided by various UNIX systems. These mechanisms allow processes running on the same computer to communicate with one another. In this chapter, we look at the mechanisms that allow processes running on different computers (connected to a common network) to communicate with one another: network IPC.

In this chapter, we describe the socket network IPC interface, which can be used by processes to communicate with other processes, regardless of where they are running: on the same machine or on different machines. Indeed, this was one of the design goals of the socket interface. The same interfaces can be used for both intermachine communication and intramachine communication. Although the socket interface can be used to communicate using many different network protocols, we will restrict our discussion to the TCP/IP protocol suite in this chapter, since it is the de facto standard for communicating over the Internet.

The socket API as specified by POSIX.1 is based on the 4.4BSD socket interface. Although minor changes have been made over the years, the current socket interface closely resembles the interface when it was originally introduced in 4.2BSD in the early 1980s.

This chapter is only an overview of the socket API. Stevens, Fenner, and Rudoff [2004] discuss the socket interface in detail in the definitive text on network programming in the UNIX System.

16.2 Socket Descriptors

A socket is an abstraction of a communication endpoint. Just as they would use file descriptors to access a file, applications use socket descriptors to access sockets. Socket descriptors are implemented as file descriptors in the UNIX System. Indeed, many of the functions that deal with file descriptors, such as `read` and `write`, will work with a socket descriptor.

To create a socket, we call the `socket` function.

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

int socket(int domain, int type, int protocol);
```

Returns: file (socket) descriptor if OK, –1 on error

The *domain* argument determines the nature of the communication, including the address format (described in more detail in the next section). Figure 16.1 summarizes the domains specified by POSIX.1. The constants start with `AF_` (for *address family*) because each domain has its own format for representing an address.

Domain	Description
`AF_INET`	IPv4 Internet domain
`AF_INET6`	IPv6 Internet domain
`AF_UNIX`	UNIX domain
`AF_UNSPEC`	unspecified

Figure 16.1 Socket communication domains

We discuss the UNIX domain in Section 17.3. Most systems define the `AF_LOCAL` domain also, which is an alias for `AF_UNIX`. The `AF_UNSPEC` domain is a wildcard that represents "any" domain. Historically, some platforms provide support for additional network protocols, such as `AF_IPX` for the NetWare protocol family, but domain constants for these protocols are not defined by the POSIX.1 standard.

The *type* argument determines the type of the socket, which further determines the communication characteristics. The socket types defined by POSIX.1 are summarized in Figure 16.2, but implementations are free to add support for additional types.

Type	Description
`SOCK_DGRAM`	fixed-length, connectionless, unreliable messages
`SOCK_RAW`	datagram interface to IP (optional in POSIX.1)
`SOCK_SEQPACKET`	fixed-length, sequenced, reliable, connection-oriented messages
`SOCK_STREAM`	sequenced, reliable, bidirectional, connection-oriented byte streams

Figure 16.2 Socket types

The *protocol* argument is usually zero, to select the default protocol for the given domain and socket type. When multiple protocols are supported for the same domain and socket type, we can use the *protocol* argument to select a particular protocol. The default protocol for a SOCK_STREAM socket in the AF_INET communication domain is TCP (Transmission Control Protocol). The default protocol for a SOCK_DGRAM socket in the AF_INET communication domain is UDP (User Datagram Protocol).

With a datagram (SOCK_DGRAM) interface, no logical connection needs to exist between peers for them to communicate. All you need to do is send a message addressed to the socket being used by the peer process.

A datagram, therefore, provides a connectionless service. A byte stream (SOCK_STREAM), on the other hand, requires that, before you can exchange data, you set up a logical connection between your socket and the socket belonging to the peer you want to communicate with.

A datagram is a self-contained message. Sending a datagram is analogous to mailing someone a letter. You can mail many letters, but you can't guarantee the order of delivery, and some might get lost along the way. Each letter contains the address of the recipient, making the letter independent from all the others. Each letter can even go to different recipients.

In contrast, using a connection-oriented protocol for communicating with a peer is like making a phone call. First, you need to establish a connection by placing a phone call, but after the connection is in place, you can communicate bidirectionally with each other. The connection is a peer-to-peer communication channel over which you talk. Your words contain no addressing information, as a point-to-point virtual connection exists between both ends of the call, and the connection itself implies a particular source and destination.

With a SOCK_STREAM socket, applications are unaware of message boundaries, since the socket provides a byte stream service. This means that when we read data from a socket, it might not return the same number of bytes written by the process sending us data. We will eventually get everything sent to us, but it might take several function calls.

A SOCK_SEQPACKET socket is just like a SOCK_STREAM socket except that we get a message-based service instead of a byte-stream service. This means that the amount of data received from a SOCK_SEQPACKET socket is the same amount as was written. The Stream Control Transmission Protocol (SCTP) provides a sequential packet service in the Internet domain.

A SOCK_RAW socket provides a datagram interface directly to the underlying network layer (which means IP in the Internet domain). Applications are responsible for building their own protocol headers when using this interface, because the transport protocols (TCP and UDP, for example) are bypassed. Superuser privileges are required to create a raw socket to prevent malicious applications from creating packets that might bypass established security mechanisms.

Calling socket is similar to calling open. In both cases, you get a file descriptor that can be used for I/O. When you are done using the file descriptor, you call close to relinquish access to the file or socket and free up the file descriptor for reuse.

Although a socket descriptor is actually a file descriptor, you can't use a socket descriptor with every function that accepts a file descriptor argument. Figure 16.3 summarizes most of the functions we've described so far that are used with file descriptors and describes how they behave when used with a socket descriptor. Unspecified and implementation-defined behavior usually means that the function doesn't work with socket descriptors. For example, `lseek` doesn't work with sockets, since sockets don't support the concept of a file offset.

Function	Behavior with socket
`close` (Section 3.3)	deallocates the socket
`dup`, `dup2` (Section 3.12)	duplicates the file descriptor as normal
`fchdir` (Section 4.22)	fails with `errno` set to `ENOTDIR`
`fchmod` (Section 4.9)	unspecified
`fchown` (Section 4.11)	implementation defined
`fcntl` (Section 3.14)	some commands supported, including `F_DUPFD`, `F_GETFD`, `F_GETFL`, `F_GETOWN`, `F_SETFD`, `F_SETFL`, and `F_SETOWN`
`fdatasync`, `fsync` (Section 3.13)	implementation defined
`fstat` (Section 4.2)	some `stat` structure members supported, but how left up to the implementation
`ftruncate` (Section 4.13)	unspecified
`getmsg`, `getpmsg` (Section 14.4)	works if sockets are implemented with STREAMS (i.e., on Solaris)
`ioctl` (Section 3.15)	some commands work, depending on underlying device driver
`lseek` (Section 3.6)	implementation defined (usually fails with `errno` set to `ESPIPE`)
`mmap` (Section 14.9)	unspecified
`poll` (Section 14.5.2)	works as expected
`putmsg`, `putpmsg` (Section 14.4)	works if sockets are implemented with STREAMS (i.e., on Solaris)
`read` (Section 3.7) and `readv` (Section 14.7)	equivalent to `recv` (Section 16.5) without any flags
`select` (Section 14.5.1)	works as expected
`write` (Section 3.8) and `writev` (Section 14.7)	equivalent to `send` (Section 16.5) without any flags

Figure 16.3 How file descriptor functions act with sockets

Communication on a socket is bidirectional. We can disable I/O on a socket with the `shutdown` function.

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

int shutdown(int sockfd, int how);

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

If *how* is `SHUT_RD`, then reading from the socket is disabled. If *how* is `SHUT_WR`, then we can't use the socket for transmitting data. We can use `SHUT_RDWR` to disable both data transmission and reception.

Given that we can `close` a socket, why is `shutdown` needed? There are several reasons. First, `close` will deallocate the network endpoint only when the last active reference is closed. This means that if we duplicate the socket (with `dup`, for example), the socket won't be deallocated until we close the last file descriptor referring to it. The `shutdown` function allows us to deactivate a socket independently of the number of active file descriptors referencing it. Second, it is sometimes convenient to shut a socket down in one direction only. For example, we can shut a socket down for writing if we want the process we are communicating with to be able to determine when we are done transmitting data, while still allowing us to use the socket to receive data sent to us by the process.

16.3 Addressing

In the previous section, we learned how to create and destroy a socket. Before we learn to do something useful with a socket, we need to learn how to identify the process that we want to communicate with. Identifying the process has two components. The machine's network address helps us identify the computer on the network we wish to contact, and the service helps us identify the particular process on the computer.

16.3.1 Byte Ordering

When communicating with processes running on the same computer, we generally don't have to worry about byte ordering. The byte order is a characteristic of the processor architecture, dictating how bytes are ordered within larger data types, such as integers. Figure 16.4 shows how the bytes within a 32-bit integer are numbered.

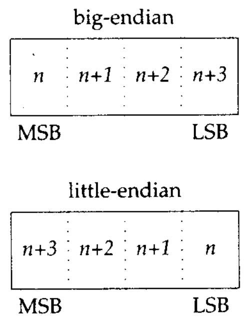

Figure 16.4 Byte order in a 32-bit integer

If the processor architecture supports *big-endian* byte order, then the highest byte address occurs in the least significant byte (LSB). *Little-endian* byte order is the opposite: the least significant byte contains the lowest byte address. Note that regardless of the byte ordering, the most significant byte (MSB) is always on the left, and the least significant byte is always on the right. Thus, if we were to assign a 32-bit integer the value `0x04030201`, the most significant byte would contain 4, and the least significant byte would contain 1, regardless of the byte ordering. If we were then to cast a character pointer (cp) to the address of the integer, we would see a difference from the byte ordering. On a little-endian processor, cp[0] would refer to the least significant byte and contain 1; cp[3] would refer to the most significant byte and contain 4. Compare that to a big-endian processor, where cp[0] would contain 4, referring to the most significant byte, and cp[3] would contain 1, referring to the least significant byte. Figure 16.5 summarizes the byte ordering for the four platforms discussed in this text.

Operating system	Processor architecture	Byte order
FreeBSD 5.2.1	Intel Pentium	little-endian
Linux 2.4.22	Intel Pentium	little-endian
Mac OS X 10.3	PowerPC	big-endian
Solaris 9	Sun SPARC	big-endian

Figure 16.5 Byte order for test platforms

To confuse matters further, some processors can be configured for either little-endian or big-endian operation.

Network protocols specify a byte ordering so that heterogeneous computer systems can exchange protocol information without confusing the byte ordering. The TCP/IP protocol suite uses big-endian byte order. The byte ordering becomes visible to applications when they exchange formatted data. With TCP/IP, addresses are presented in network byte order, so applications sometimes need to translate them between the processor's byte order and the network byte order. This is common when printing an address in a human-readable form, for example.

Four common functions are provided to convert between the processor byte order and the network byte order for TCP/IP applications.

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostint32);

                                   Returns: 32-bit integer in network byte order

uint16_t htons(uint16_t hostint16);

                                   Returns: 16-bit integer in network byte order

uint32_t ntohl(uint32_t netint32);

                                      Returns: 32-bit integer in host byte order

uint16_t ntohs(uint16_t netint16);

                                      Returns: 16-bit integer in host byte order
```

The h is for "host" byte order, and the n is for "network" byte order. The l is for "long" (i.e., 4-byte) integer, and the s is for "short" (i.e., 2-byte) integer. These four functions are defined in <arpa/inet.h>, although some older systems define them in <netinet/in.h>.

16.3.2 Address Formats

An address identifies a socket endpoint in a particular communication domain. The address format is specific to the particular domain. So that addresses with different formats can be passed to the socket functions, the addresses are cast to a generic sockaddr address structure:

```
struct sockaddr {
  sa_family_t   sa_family;  /* address family */
  char          sa_data[];  /* variable-length address */
  .
  .
  .
};
```

Implementations are free to add additional members and define a size for the sa_data member. For example, on Linux, the structure is defined as

```
struct sockaddr {
  sa_family_t  sa_family;     /* address family */
  char         sa_data[14];   /* variable-length address */
};
```

But on FreeBSD, the structure is defined as

```
struct sockaddr {
  unsigned char  sa_len;         /* total length */
  sa_family_t    sa_family;      /* address family */
  char           sa_data[14];    /* variable-length address */
};
```

Internet addresses are defined in <netinet/in.h>. In the IPv4 Internet domain (AF_INET), a socket address is represented by a sockaddr_in structure:

```
struct in_addr {
  in_addr_t       s_addr;          /* IPv4 address */
};

struct sockaddr_in {
  sa_family_t    sin_family;      /* address family */
  in_port_t      sin_port;        /* port number */
  struct in_addr sin_addr;        /* IPv4 address */
};
```

The in_port_t data type is defined to be a uint16_t. The in_addr_t data type is defined to be a uint32_t. These integer data types specify the number of bits in the data type and are defined in <stdint.h>.

In contrast to the AF_INET domain, the IPv6 Internet domain (AF_INET6) socket address is represented by a sockaddr_in6 structure:

```
struct in6_addr {
  uint8_t        s6_addr[16];      /* IPv6 address */
};
```

```
struct sockaddr_in6 {
  sa_family_t     sin6_family;   /* address family */
  in_port_t       sin6_port;     /* port number */
  uint32_t        sin6_flowinfo; /* traffic class and flow info */
  struct in6_addr sin6_addr;     /* IPv6 address */
  uint32_t        sin6_scope_id; /* set of interfaces for scope */
};
```

These are the definitions required by the Single UNIX Specification. Individual implementations are free to add additional fields. For example, on Linux, the sockaddr_in structure is defined as

```
struct sockaddr_in {
  sa_family_t     sin_family;   /* address family */
  in_port_t       sin_port;     /* port number */
  struct in_addr  sin_addr;     /* IPv4 address */
  unsigned char   sin_zero[8];  /* filler */
};
```

where the sin_zero member is a filler field that should be set to all-zero values.

Note that although the sockaddr_in and sockaddr_in6 structures are quite different, they are both passed to the socket routines cast to a sockaddr structure. In Section 17.3, we will see that the structure of a UNIX domain socket address is different from both of the Internet domain socket address formats.

It is sometimes necessary to print an address in a format that is understandable by a person instead of a computer. The BSD networking software included the inet_addr and inet_ntoa functions to convert between the binary address format and a string in dotted-decimal notation (a.b.c.d). These functions, however, work only with IPv4 addresses. Two new functions—inet_ntop and inet_pton—support similar functionality and work with both IPv4 and IPv6 addresses.

```
#include <arpa/inet.h>

const char *inet_ntop(int domain, const void *restrict addr,
                      char *restrict str, socklen_t size);

                  Returns: pointer to address string on success, NULL on error

int inet_pton(int domain, const char *restrict str,
              void *restrict addr);

                  Returns: 1 on success, 0 if the format is invalid, or –1 on error
```

The inet_ntop function converts a binary address in network byte order into a text string; inet_pton converts a text string into a binary address in network byte order. Only two *domain* values are supported: AF_INET and AF_INET6.

For inet_ntop, the *size* parameter specifies the size of the buffer (*str*) to hold the text string. Two constants are defined to make our job easier: INET_ADDRSTRLEN is large enough to hold a text string representing an IPv4 address, and INET6_ADDRSTRLEN is large enough to hold a text string representing an IPv6 address. For inet_pton, the *addr* buffer needs to be large enough to hold a 32-bit address if *domain* is AF_INET or large enough to hold a 128-bit address if *domain* is AF_INET6.

16.3.3 Address Lookup

Ideally, an application won't have to be aware of the internal structure of a socket address. If an application simply passes socket addresses around as sockaddr structures and doesn't rely on any protocol-specific features, then the application will work with many different protocols that provide the same type of service.

Historically, the BSD networking software has provided interfaces to access the various network configuration information. In Section 6.7, we briefly discussed the networking data files and the functions used to access them. In this section, we discuss them in a little more detail and introduce the newer functions used to look up addressing information.

The network configuration information returned by these functions can be kept in a number of places. They can be kept in static files (/etc/hosts, /etc/services, etc.), or they can be managed by a name service, such as DNS (Domain Name System) or NIS (Network Information Service). Regardless of where the information is kept, the same functions can be used to access it.

The hosts known by a given computer system are found by calling gethostent.

```
#include <netdb.h>

struct hostent *gethostent(void);

                                        Returns: pointer if OK, NULL on error

void sethostent(int stayopen);

void endhostent(void);
```

If the host database file isn't already open, gethostent will open it. The gethostent function returns the next entry in the file. The sethostent function will open the file or rewind it if it is already open. The endhostent function will close the file.

When gethostent returns, we get a pointer to a hostent structure which might point to a static data buffer that is overwritten each time we call gethostent. The hostent structure is defined to have at least the following members:

```
struct hostent {
  char   *h_name;        /* name of host */
  char  **h_aliases;     /* pointer to alternate host name array */
  int     h_addrtype;    /* address type */
  int     h_length;      /* length in bytes of address */
  char  **h_addr_list;   /* pointer to array of network addresses */
   .
   .
   .
};
```

The addresses returned are in network byte order.

Two additional functions—gethostbyname and gethostbyaddr—originally were included with the hostent functions, but are now considered to be obsolete. We'll see replacements for them shortly.

We can get network names and numbers with a similar set of interfaces.

```
#include <netdb.h>

struct netent *getnetbyaddr(uint32_t net, int type);

struct netent *getnetbyname(const char *name);

struct netent *getnetent(void);

                              All return: pointer if OK, NULL on error

void setnetent(int stayopen);

void endnetent(void);
```

The netent structure contains at least the following fields:

```
struct netent {
  char     *n_name;      /* network name */
  char    **n_aliases;   /* alternate network name array pointer */
  int       n_addrtype;  /* address type */
  uint32_t  n_net;       /* network number */
  .
  .
  .
};
```

The network number is returned in network byte order. The address type is one of the address family constants (AF_INET, for example).

We can map between protocol names and numbers with the following functions.

```
#include <netdb.h>

struct protoent *getprotobyname(const char *name);

struct protoent *getprotobynumber(int proto);

struct protoent *getprotoent(void);

                              All return: pointer if OK, NULL on error

void setprotoent(int stayopen);

void endprotoent(void);
```

The protoent structure as defined by POSIX.1 has at least the following members:

```
struct protoent {
  char   *p_name;     /* protocol name */
  char  **p_aliases;  /* pointer to alternate protocol name array */
  int     p_proto;    /* protocol number */
  .
  .
  .
};
```

Services are represented by the port number portion of the address. Each service is offered on a unique, well-known port number. We can map a service name to a port number with getservbyname, map a port number to a service name with getservbyport, or scan the services database sequentially with getservent.

```
#include <netdb.h>

struct servent *getservbyname(const char *name, const char *proto);

struct servent *getservbyport(int port, const char *proto);

struct servent *getservent(void);

                                    All return: pointer if OK, NULL on error

void setservent(int stayopen);

void endservent(void);
```

The servent structure is defined to have at least the following members:

```
struct servent {
  char   *s_name;      /* service name */
  char  **s_aliases;   /* pointer to alternate service name array */
  int     s_port;      /* port number */
  char   *s_proto;     /* name of protocol */
   .
   .
   .
};
```

POSIX.1 defines several new functions to allow an application to map from a host name and a service name to an address and vice versa. These functions replace the older gethostbyname and gethostbyaddr functions.

The getaddrinfo function allows us to map a host name and a service name to an address.

```
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *restrict host,
                const char *restrict service,
                const struct addrinfo *restrict hint,
                struct addrinfo **restrict res);

                              Returns: 0 if OK, nonzero error code on error

void freeaddrinfo(struct addrinfo *ai);
```

We need to provide the host name, the service name, or both. If we provide only one name, the other should be a null pointer. The host name can be either a node name or the host address in dotted-decimal notation.

The getaddrinfo function returns a linked list of addrinfo structures. We can use freeaddrinfo to free one or more of these structures, depending on how many structures are linked together using the ai_next field.

The addrinfo structure is defined to include at least the following members:

```
struct addrinfo {
  int              ai_flags;       /* customize behavior */
  int              ai_family;      /* address family */
  int              ai_socktype;    /* socket type */
  int              ai_protocol;    /* protocol */
  socklen_t        ai_addrlen;     /* length in bytes of address */
  struct sockaddr *ai_addr;        /* address */
  char            *ai_canonname;   /* canonical name of host */
  struct addrinfo *ai_next;        /* next in list */
    .
    .
    .
};
```

We can supply an optional *hint* to select addresses that meet certain criteria. The hint is a template used for filtering addresses and uses only the ai_family, ai_flags, ai_protocol, and ai_socktype fields. The remaining integer fields must be set to 0, and the pointer fields must be null. Figure 16.6 summarizes the flags we can use in the ai_flags field to customize how addresses and names are treated.

Flag	Description
AI_ADDRCONFIG	Query for whichever address type (IPv4 or IPv6) is configured.
AI_ALL	Look for both IPv4 and IPv6 addresses (used only with AI_V4MAPPED).
AI_CANONNAME	Request a canonical name (as opposed to an alias).
AI_NUMERICHOST	Return the host address in numeric format.
AI_NUMERICSERV	Return the service as a port number.
AI_PASSIVE	Socket address is intended to be bound for listening.
AI_V4MAPPED	If no IPv6 addresses are found, return IPv4 addresses mapped in IPv6 format.

Figure 16.6 Flags for addrinfo structure

If getaddrinfo fails, we can't use perror or strerror to generate an error message. Instead, we need to call gai_strerror to convert the error code returned into an error message.

```
#include <netdb.h>

const char *gai_strerror(int error);

                      Returns: a pointer to a string describing the error
```

The getnameinfo function converts an address into a host name and a service name.

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *restrict addr,
                socklen_t alen, char *restrict host,
                socklen_t hostlen, char *restrict service,
                socklen_t servlen, unsigned int flags);

                                   Returns: 0 if OK, nonzero on error
```

The socket address (*addr*) is translated into a host name and a service name. If *host* is non-null, it points to a buffer *hostlen* bytes long that will be used to return the host name. Similarly, if *service* is non-null, it points to a buffer *servlen* bytes long that will be used to return the service name.

The *flags* argument gives us some control over how the translation is done. Figure 16.7 summarizes the supported flags.

Flag	Description
`NI_DGRAM`	The service is datagram based instead of stream based.
`NI_NAMEREQD`	If the host name can't be found, treat this as an error.
`NI_NOFQDN`	Return only the node name portion of the fully-qualified domain name for local hosts.
`NI_NUMERICHOST`	Return the numeric form of the host address instead of the name.
`NI_NUMERICSERV`	Return the numeric form of the service address (i.e., the port number) instead of the name.

Figure 16.7 Flags for the `getnameinfo` function

Example

Figure 16.8 illustrates the use of the `getaddrinfo` function.

```
#include "apue.h"
#include <netdb.h>
#include <arpa/inet.h>
#if defined(BSD) || defined(MACOS)
#include <sys/socket.h>
#include <netinet/in.h>
#endif

void
print_family(struct addrinfo *aip)
{
    printf(" family ");
    switch (aip->ai_family) {
    case AF_INET:
        printf("inet");
        break;
    case AF_INET6:
        printf("inet6");
        break;
    case AF_UNIX:
        printf("unix");
        break;
    case AF_UNSPEC:
        printf("unspecified");
        break;
    default:
        printf("unknown");
    }
```

```
}

void
print_type(struct addrinfo *aip)
{
    printf(" type ");
    switch (aip->ai_socktype) {
    case SOCK_STREAM:
        printf("stream");
        break;
    case SOCK_DGRAM:
        printf("datagram");
        break;
    case SOCK_SEQPACKET:
        printf("seqpacket");
        break;
    case SOCK_RAW:
        printf("raw");
        break;
    default:
        printf("unknown (%d)", aip->ai_socktype);
    }
}

void
print_protocol(struct addrinfo *aip)
{
    printf(" protocol ");
    switch (aip->ai_protocol) {
    case 0:
        printf("default");
        break;
    case IPPROTO_TCP:
        printf("TCP");
        break;
    case IPPROTO_UDP:
        printf("UDP");
        break;
    case IPPROTO_RAW:
        printf("raw");
        break;
    default:
        printf("unknown (%d)", aip->ai_protocol);
    }
}

void
print_flags(struct addrinfo *aip)
{
    printf("flags");
    if (aip->ai_flags == 0) {
        printf(" 0");
    } else {
        if (aip->ai_flags & AI_PASSIVE)
            printf(" passive");
        if (aip->ai_flags & AI_CANONNAME)
            printf(" canon");
        if (aip->ai_flags & AI_NUMERICHOST)
            printf(" numhost");
#if defined(AI_NUMERICSERV)
        if (aip->ai_flags & AI_NUMERICSERV)
            printf(" numserv");
#endif
#if defined(AI_V4MAPPED)
        if (aip->ai_flags & AI_V4MAPPED)
            printf(" v4mapped");
#endif
#if defined(AI_ALL)
        if (aip->ai_flags & AI_ALL)
            printf(" all");
#endif
    }
}

int
main(int argc, char *argv[])
{
    struct addrinfo     *ailist, *aip;
    struct addrinfo     hint;
    struct sockaddr_in  *sinp;
    const char          *addr;
    int                 err;
    char                abuf[INET_ADDRSTRLEN];

    if (argc != 3)
        err_quit("usage: %s nodename service", argv[0]);
    hint.ai_flags = AI_CANONNAME;
    hint.ai_family = 0;
    hint.ai_socktype = 0;
    hint.ai_protocol = 0;
    hint.ai_addrlen = 0;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(argv[1], argv[2], &hint, &ailist)) != 0)
        err_quit("getaddrinfo error: %s", gai_strerror(err));
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        print_flags(aip);
        print_family(aip);
        print_type(aip);
        print_protocol(aip);
        printf("\n\thost %s", aip->ai_canonname?aip->ai_canonname:"-");
        if (aip->ai_family == AF_INET) {
            sinp = (struct sockaddr_in *)aip->ai_addr;
            addr = inet_ntop(AF_INET, &sinp->sin_addr, abuf,
                INET_ADDRSTRLEN);
            printf(" address %s", addr?addr:"unknown");
            printf(" port %d", ntohs(sinp->sin_port));
        }
        printf("\n");
    }
    exit(0);
}
```

Figure 16.8 Print host and service information

This program illustrates the use of the getaddrinfo function. If multiple protocols provide the given service for the given host, the program will print more than one entry. In this example, we print out the address information only for the protocols that work with IPv4 (ai_family equals AF_INET). If we wanted to restrict the output to the AF_INET protocol family, we could set the ai_family field in the hint.

When we run the program on one of the test systems, we get

```
$ ./a.out harry nfs
flags canon family inet type stream protocol TCP
     host harry address 192.168.1.105 port 2049
flags canon family inet type datagram protocol UDP
     host harry address 192.168.1.105 port 2049
```

□

16.3.4 Associating Addresses with Sockets

The address associated with a client's socket is of little interest, and we can let the system choose a default address for us. For a server, however, we need to associate a well-known address with the server's socket on which client requests will arrive. Clients need a way to discover the address to use to contact a server, and the simplest scheme is for a server to reserve an address and register it in /etc/services or with a name service.

We use the bind function to associate an address with a socket.

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

int bind(int sockfd, const struct sockaddr *addr, socklen_t len);

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

There are several restrictions on the address we can use:

- The address we specify must be valid for the machine on which the process is running; we can't specify an address belonging to some other machine.
- The address must match the format supported by the address family we used to create the socket.

- The port number in the address cannot be less than 1,024 unless the process has the appropriate privilege (i.e., is the superuser).
- Usually, only one socket endpoint can be bound to a given address, although some protocols allow duplicate bindings.

For the Internet domain, if we specify the special IP address INADDR_ANY, the socket endpoint will be bound to all the system's network interfaces. This means that we can receive packets from any of the network interface cards installed in the system. We'll see in the next section that the system will choose an address and bind it to our socket for us if we call connect or listen without first binding an address to the socket.

We can use the getsockname function to discover the address bound to a socket.

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

int getsockname(int sockfd, struct sockaddr *restrict addr,
                socklen_t *restrict alenp);

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

Before calling getsockname, we set *alenp* to point to an integer containing the size of the sockaddr buffer. On return, the integer is set to the size of the address returned. If the address won't fit in the buffer provided, the address is silently truncated. If no address is currently bound to the socket, the results are undefined.

If the socket is connected to a peer, we can find out the peer's address by calling the getpeername function.

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

int getpeername(int sockfd, struct sockaddr *restrict addr,
                socklen_t *restrict alenp);

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

Other than returning the peer's address, the getpeername function is identical to the getsockname function.

16.4 Connection Establishment

If we're dealing with a connection-oriented network service (SOCK_STREAM or SOCK_SEQPACKET), then before we can exchange data, we need to create a connection between the socket of the process requesting the service (the client) and the process providing the service (the server). We use the connect function to create a connection.

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

int connect(int sockfd, const struct sockaddr *addr, socklen_t len);

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

The address we specify with connect is the address of the server with which we wish to communicate. If *sockfd* is not bound to an address, connect will bind a default address for the caller.

When we try to connect to a server, the connect request might fail for several reasons. The machine to which we are trying to connect must be up and running, the server must be bound to the address we are trying to contact, and there must be room in the server's pending connect queue (we'll learn more about this shortly). Thus, applications must be able to handle connect error returns that might be caused by transient conditions.

Example

Figure 16.9 shows one way to handle transient connect errors. This is likely with a server that is running on a heavily loaded system.

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

#define MAXSLEEP 128

int
connect_retry(int sockfd, const struct sockaddr *addr, socklen_t alen)
{
    int nsec;

    /*
     * Try to connect with exponential backoff.
     */
    for (nsec = 1; nsec <= MAXSLEEP; nsec <<= 1) {
        if (connect(sockfd, addr, alen) == 0) {
            /*
             * Connection accepted.
             */
            return(0);
        }

        /*
         * Delay before trying again.
         */
        if (nsec <= MAXSLEEP/2)
            sleep(nsec);
    }
    return(-1);
}
```

Figure 16.9 Connect with retry

This function shows what is known as an *exponential backoff* algorithm. If the call to connect fails, the process goes to sleep for a short time and then tries again, increasing the delay each time through the loop, up to a maximum delay of about 2 minutes. □

If the socket descriptor is in nonblocking mode, which we discuss further in Section 16.8, connect will return −1 with errno set to the special error code EINPROGRESS if the connection can't be established immediately. The application can use either poll or select to determine when the file descriptor is writable. At this point, the connection is complete.

The connect function can also be used with a connectionless network service (SOCK_DGRAM). This might seem like a contradiction, but it is an optimization instead. If we call connect with a SOCK_DGRAM socket, the destination address of all messages we send is set to the address we specified in the connect call, relieving us from having to provide the address every time we transmit a message. In addition, we will receive datagrams only from the address we've specified.

A server announces that it is willing to accept connect requests by calling the listen function.

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

int listen(int sockfd, int backlog);

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

The *backlog* argument provides a hint to the system of the number of outstanding connect requests that it should enqueue on behalf of the process. The actual value is determined by the system, but the upper limit is specified as SOMAXCONN in <sys/socket.h>.

On Solaris, the SOMAXCONN value in <sys/socket.h> is ignored. The particular maximum depends on the implementation of each protocol. For TCP, the default is 128.

Once the queue is full, the system will reject additional connect requests, so the *backlog* value must be chosen based on the expected load of the server and the amount of processing it must do to accept a connect request and start the service.

Once a server has called listen, the socket used can receive connect requests. We use the accept function to retrieve a connect request and convert that into a connection.

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

int accept(int sockfd, struct sockaddr *restrict addr,
           socklen_t *restrict len);

                   Returns: file (socket) descriptor if OK, −1 on error
```

The file descriptor returned by accept is a socket descriptor that is connected to the client that called connect. This new socket descriptor has the same socket type and address family as the original socket (*sockfd*). The original socket passed to accept is not associated with the connection, but instead remains available to receive additional connect requests.

If we don't care about the client's identity, we can set the *addr* and *len* parameters to NULL. Otherwise, before calling accept, we need to set the *addr* parameter to a buffer large enough to hold the address and set the integer pointed to by *len* to the size of the buffer. On return, accept will fill in the client's address in the buffer and update the integer pointed to by *len* to reflect the size of the address.

If no connect requests are pending, accept will block until one arrives. If *sockfd* is in nonblocking mode, accept will return −1 and set errno to either EAGAIN or EWOULDBLOCK.

All four platforms discussed in this text define EAGAIN to be the same as EWOULDBLOCK.

If a server calls accept and no connect request is present, the server will block until one arrives. Alternatively, a server can use either poll or select to wait for a connect request to arrive. In this case, a socket with pending connect requests will appear to be readable.

Example

Figure 16.10 shows a function we can use to allocate and initialize a socket for use by a server process.

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

int
initserver(int type, const struct sockaddr *addr, socklen_t alen,
  int qlen)
{
    int fd;
    int err = 0;

    if ((fd = socket(addr->sa_family, type, 0)) < 0)
        return(-1);
    if (bind(fd, addr, alen) < 0) {
        err = errno;
        goto errout;
    }
    if (type == SOCK_STREAM || type == SOCK_SEQPACKET) {
        if (listen(fd, qlen) < 0) {
            err = errno;
            goto errout;
        }
    }
    return(fd);

errout:
    close(fd);
    errno = err;
    return(-1);
}
```

Figure 16.10 Initialize a socket endpoint for use by a server

We'll see that TCP has some strange rules regarding address reuse that make this example inadequate. Figure 16.20 shows a version of this function that bypasses these rules, solving the major drawback with this version. □

16.5 Data Transfer

Since a socket endpoint is represented as a file descriptor, we can use `read` and `write` to communicate with a socket, as long as it is connected. Recall that a datagram socket can be "connected" if we set the default peer address using the `connect` function. Using `read` and `write` with socket descriptors is significant, because it means that we can pass socket descriptors to functions that were originally designed to work with local files. We can also arrange to pass the socket descriptors to child processes that execute programs that know nothing about sockets.

Although we can exchange data using `read` and `write`, that is about all we can do with these two functions. If we want to specify options, receive packets from multiple clients, or send out-of-band data, we need to use one of the six socket functions designed for data transfer.

Three functions are available for sending data, and three are available for receiving data. First, we'll look at the ones used to send data.

The simplest one is `send`. It is similar to `write`, but allows us to specify flags to change how the data we want to transmit is treated.

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

ssize_t send(int sockfd, const void *buf, size_t nbytes, int flags);

                        Returns: number of bytes sent if OK, −1 on error
```

Like `write`, the socket has to be connected to use `send`. The *buf* and *nbytes* arguments have the same meaning as they do with `write`.

Unlike `write`, however, `send` supports a fourth *flags* argument. Two flags are defined by the Single UNIX Specification, but it is common for implementations to support additional ones. They are summarized in Figure 16.11.

Flag	Description	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
MSG_DONTROUTE	Don't route packet outside of local network.		•	•	•	•
MSG_DONTWAIT	Enable nonblocking operation (equivalent to using O_NONBLOCK).		•	•	•	
MSG_EOR	This is the end of record if supported by protocol.	•	•	•	•	
MSG_OOB	Send out-of-band data if supported by protocol (see Section 16.7).	•	•	•	•	•

Figure 16.11 Flags used with send socket calls

If send returns success, it doesn't necessarily mean that the process at the other end of the connection receives the data. All we are guaranteed is that when send succeeds, the data has been delivered to the network drivers without error.

With a protocol that supports message boundaries, if we try to send a single message larger than the maximum supported by the protocol, send will fail with errno set to EMSGSIZE. With a byte-stream protocol, send will block until the entire amount of data has been transmitted.

The sendto function is similar to send. The difference is that sendto allows us to specify a destination address to be used with connectionless sockets.

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

ssize_t sendto(int sockfd, const void *buf, size_t nbytes, int flags,
               const struct sockaddr *destaddr, socklen_t destlen);

                              Returns: number of bytes sent if OK, -1 on error
```

With a connection-oriented socket, the destination address is ignored, as the destination is implied by the connection. With a connectionless socket, we can't use send unless the destination address is first set by calling connect, so sendto gives us an alternate way to send a message.

We have one more choice when transmitting data over a socket. We can call sendmsg with a msghdr structure to specify multiple buffers from which to transmit data, similar to the writev function (Section 14.7).

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

ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);

                              Returns: number of bytes sent if OK, -1 on error
```

POSIX.1 defines the msghdr structure to have at least the following members:

```
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 */
  .
  .
  .
};
```

We saw the iovec structure in Section 14.7. We'll see the use of ancillary data in Section 17.4.2.

The recv function is similar to read, but allows us to specify some options to control how we receive the data.

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

ssize_t recv(int sockfd, void *buf, size_t nbytes, int flags);

                                   Returns: length of message in bytes,
          0 if no messages are available and peer has done an orderly shutdown,
                                                             or -1 on error
```

The flags that can be passed to `recv` are summarized in Figure 16.12. Only three are defined by the Single UNIX Specification.

Flag	Description	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
`MSG_OOB`	Retrieve out-of-band data if supported by protocol (see Section 16.7).	•	•	•	•	•
`MSG_PEEK`	Return packet contents without consuming packet.	•	•	•	•	•
`MSG_TRUNC`	Request that the real length of the packet be returned, even if it was truncated.			•		
`MSG_WAITALL`	Wait until all data is available (`SOCK_STREAM` only).	•	•	•	•	•

Figure 16.12 Flags used with `recv` socket calls

When we specify the `MSG_PEEK` flag, we can peek at the next data to be read without actually consuming it. The next call to `read` or one of the `recv` functions will return the same data we peeked at.

With `SOCK_STREAM` sockets, we can receive less data than we requested. The `MSG_WAITALL` flag inhibits this behavior, preventing `recv` from returning until all the data we requested has been received. With `SOCK_DGRAM` and `SOCK_SEQPACKET` sockets, the `MSG_WAITALL` flag provides no change in behavior, because these message-based socket types already return an entire message in a single read.

If the sender has called `shutdown` (Section 16.2) to end transmission, or if the network protocol supports orderly shutdown by default and the sender has closed the socket, then `recv` will return 0 when we have received all the data.

If we are interested in the identity of the sender, we can use `recvfrom` to obtain the source address from which the data was sent.

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

ssize_t recvfrom(int sockfd, void *restrict buf, size_t len, int flags,
                 struct sockaddr *restrict addr,
                 socklen_t *restrict addrlen);

                                   Returns: length of message in bytes,
          0 if no messages are available and peer has done an orderly shutdown,
                                                             or -1 on error
```

If *addr* is non-null, it will contain the address of the socket endpoint from which the data was sent. When calling `recvfrom`, we need to set the *addrlen* parameter to point to an integer containing the size in bytes of the socket buffer to which *addr* points. On return, the integer is set to the actual size of the address in bytes.

Because it allows us to retrieve the address of the sender, `recvfrom` is usually used with connectionless sockets. Otherwise, `recvfrom` behaves identically to `recv`.

To receive data into multiple buffers, similar to `readv` (Section 14.7), or if we want to receive ancillary data (Section 17.4.2), we can use `recvmsg`.

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

ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

Returns: length of message in bytes,
0 if no messages are available and peer has done an orderly shutdown,
or −1 on error

The `msghdr` structure (which we saw used with `sendmsg`) is used by `recvmsg` to specify the input buffers to be used to receive the data. We can set the *flags* argument to change the default behavior of `recvmsg`. On return, the `msg_flags` field of the `msghdr` structure is set to indicate various characteristics of the data received. (The `msg_flags` field is ignored on entry to `recvmsg`). The possible values on return from `recvmsg` are summarized in Figure 16.13. We'll see an example that uses `recvmsg` in Chapter 17.

Flag	Description	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
MSG_CTRUNC	Control data was truncated.	•	•	•	•	•
MSG_DONTWAIT	`recvmsg` was called in nonblocking mode.			•		•
MSG_EOR	End of record was received.	•	•	•	•	•
MSG_OOB	Out-of-band data was received.	•	•	•	•	•
MSG_TRUNC	Normal data was truncated.	•	•	•	•	•

Figure 16.13 Flags returned in `msg_flags` by `recvmsg`

Example—Connection-Oriented Client

Figure 16.14 shows a client command that communicates with a server to obtain the output from a system's `uptime` command. We call this service "remote uptime" (or "ruptime" for short).

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

#define MAXADDRLEN  256
```

```
#define BUFLEN      128

extern int connect_retry(int, const struct sockaddr *, socklen_t);

void
print_uptime(int sockfd)
{
    int     n;
    char    buf[BUFLEN];

    while ((n = recv(sockfd, buf, BUFLEN, 0)) > 0)
        write(STDOUT_FILENO, buf, n);
    if (n < 0)
        err_sys("recv error");
}

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    int             sockfd, err;

    if (argc != 2)
        err_quit("usage: ruptime hostname");
    hint.ai_flags = 0;
    hint.ai_family = 0;
    hint.ai_socktype = SOCK_STREAM;
    hint.ai_protocol = 0;
    hint.ai_addrlen = 0;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(argv[1], "ruptime", &hint, &ailist)) != 0)
        err_quit("getaddrinfo error: %s", gai_strerror(err));
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = socket(aip->ai_family, SOCK_STREAM, 0)) < 0)
            err = errno;
        if (connect_retry(sockfd, aip->ai_addr, aip->ai_addrlen) < 0) {
            err = errno;
        } else {
            print_uptime(sockfd);
            exit(0);
        }
    }
    fprintf(stderr, "can't connect to %s: %s\n", argv[1],
      strerror(err));
    exit(1);
}
```

Figure 16.14 Client command to get uptime from server

This program connects to a server, reads the string sent by the server, and prints the string on the standard output. Since we're using a SOCK_STREAM socket, we can't be guaranteed that we will read the entire string in one call to recv, so we need to repeat the call until it returns 0.

The getaddrinfo function might return more than one candidate address for us to use if the server supports multiple network interfaces or multiple network protocols. We try each one in turn, giving up when we find one that allows us to connect to the service. We use the connect_retry function from Figure 16.9 to establish a connection with the server. □

Example—Connection-Oriented Server

Figure 16.15 shows the server that provides the uptime command's output to the client program from Figure 16.14.

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

#define BUFLEN  128
#define QLEN 10

#ifndef HOST_NAME_MAX
#define HOST_NAME_MAX 256
#endif

extern int initserver(int, struct sockaddr *, socklen_t, int);

void
serve(int sockfd)
{
    int     clfd;
    FILE    *fp;
    char    buf[BUFLEN];

    for (;;) {
        clfd = accept(sockfd, NULL, NULL);
        if (clfd < 0) {
            syslog(LOG_ERR, "ruptimed: accept error: %s",
              strerror(errno));
            exit(1);
        }
        if ((fp = popen("/usr/bin/uptime", "r")) == NULL) {
            sprintf(buf, "error: %s\n", strerror(errno));
            send(clfd, buf, strlen(buf), 0);
        } else {
            while (fgets(buf, BUFLEN, fp) != NULL)
                send(clfd, buf, strlen(buf), 0);
            pclose(fp);
        }
        close(clfd);
    }
}

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    int             sockfd, err, n;
    char            *host;

    if (argc != 1)
        err_quit("usage: ruptimed");
#ifdef _SC_HOST_NAME_MAX
    n = sysconf(_SC_HOST_NAME_MAX);
    if (n < 0)  /* best guess */
#endif
        n = HOST_NAME_MAX;
    host = malloc(n);
    if (host == NULL)
        err_sys("malloc error");
    if (gethostname(host, n) < 0)
        err_sys("gethostname error");
    daemonize("ruptimed");
    hint.ai_flags = AI_CANONNAME;
    hint.ai_family = 0;
    hint.ai_socktype = SOCK_STREAM;
    hint.ai_protocol = 0;
    hint.ai_addrlen = 0;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(host, "ruptime", &hint, &ailist)) != 0) {
        syslog(LOG_ERR, "ruptimed: getaddrinfo error: %s",
          gai_strerror(err));
        exit(1);
    }
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = initserver(SOCK_STREAM, aip->ai_addr,
          aip->ai_addrlen, QLEN)) >= 0) {
            serve(sockfd);
            exit(0);
        }
    }
    exit(1);
}
```

Figure 16.15 Server program to provide system uptime

To find out its address, the server needs to get the name of the host on which it is running. Some systems don't define the _SC_HOST_NAME_MAX constant, so we use HOST_NAME_MAX in this case. If the system doesn't define HOST_NAME_MAX, we define it ourselves. POSIX.1 states that the minimum value for the host name is 255 bytes, not including the terminating null, so we define HOST_NAME_MAX to be 256 to include the terminating null.

The server gets the host name by calling gethostname and looks up the address for the remote uptime service. Multiple addresses can be returned, but we simply choose the first one for which we can establish a passive socket endpoint. Handling multiple addresses is left as an exercise.

We use the initserver function from Figure 16.10 to initialize the socket endpoint on which we will wait for connect requests to arrive. (Actually, we use the version from Figure 16.20; we'll see why when we discuss socket options in Section 16.6.) □

Example—Alternate Connection-Oriented Server

Previously, we stated that using file descriptors to access sockets was significant, because it allowed programs that knew nothing about networking to be used in a networked environment. The version of the server shown in Figure 16.16 illustrates this point. Instead of reading the output of the uptime command and sending it to the client, the server arranges to have the standard output and standard error of the uptime command be the socket endpoint connected to the client.

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

#define QLEN 10

#ifndef HOST_NAME_MAX
#define HOST_NAME_MAX 256
#endif

extern int initserver(int, struct sockaddr *, socklen_t, int);

void
serve(int sockfd)
{
    int     clfd, status;
    pid_t   pid;

    for (;;) {
        clfd = accept(sockfd, NULL, NULL);
        if (clfd < 0) {
            syslog(LOG_ERR, "ruptimed: accept error: %s",
              strerror(errno));
```

```
                exit(1);
            }
            if ((pid = fork()) < 0) {
                syslog(LOG_ERR, "ruptimed: fork error: %s",
                  strerror(errno));
                exit(1);
            } else if (pid == 0) {  /* child */
                /*
                 * The parent called daemonize (Figure 13.1), so
                 * STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO
                 * are already open to /dev/null.  Thus, the call to
                 * close doesn't need to be protected by checks that
                 * clfd isn't already equal to one of these values.
                 */
                if (dup2(clfd, STDOUT_FILENO) != STDOUT_FILENO ||
                  dup2(clfd, STDERR_FILENO) != STDERR_FILENO) {
                    syslog(LOG_ERR, "ruptimed: unexpected error");
                    exit(1);
                }
                close(clfd);
                execl("/usr/bin/uptime", "uptime", (char *)0);
                syslog(LOG_ERR, "ruptimed: unexpected return from exec: %s",
                  strerror(errno));
            } else {        /* parent */
                close(clfd);
                waitpid(pid, &status, 0);
            }
        }
    }

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    int             sockfd, err, n;
    char            *host;

    if (argc != 1)
        err_quit("usage: ruptimed");
#ifdef _SC_HOST_NAME_MAX
    n = sysconf(_SC_HOST_NAME_MAX);
    if (n < 0)  /* best guess */
#endif
        n = HOST_NAME_MAX;
    host = malloc(n);
    if (host == NULL)
        err_sys("malloc error");
    if (gethostname(host, n) < 0)
        err_sys("gethostname error");
    daemonize("ruptimed");
    hint.ai_flags = AI_CANONNAME;
    hint.ai_family = 0;
    hint.ai_socktype = SOCK_STREAM;
    hint.ai_protocol = 0;
    hint.ai_addrlen = 0;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(host, "ruptime", &hint, &ailist)) != 0) {
        syslog(LOG_ERR, "ruptimed: getaddrinfo error: %s",
          gai_strerror(err));
        exit(1);
    }
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = initserver(SOCK_STREAM, aip->ai_addr,
          aip->ai_addrlen, QLEN)) >= 0) {
            serve(sockfd);
            exit(0);
        }
    }
    exit(1);
}
```

Figure 16.16 Server program illustrating command writing directly to socket

Instead of using popen to run the uptime command and reading the output from the pipe connected to the command's standard output, we use fork to create a child process and then use dup2 to arrange that the child's copy of STDIN_FILENO is open to /dev/null and that both STDOUT_FILENO and STDERR_FILENO are open to the socket endpoint. When we execute uptime, the command writes the results to its standard output, which is connected to the socket, and the data is sent back to the ruptime client command.

The parent can safely close the file descriptor connected to the client, because the child still has it open. The parent waits for the child to complete before proceeding, so that the child doesn't become a zombie. Since it shouldn't take too long to run the uptime command, the parent can afford to wait for the child to exit before accepting the next connect request. This strategy might not be appropriate if the child takes a long time, however. □

The previous examples have used connection-oriented sockets. But how do we choose the appropriate type? When do we use a connection-oriented socket, and when do we use a connectionless socket? The answer depends on how much work we want to do and what kind of tolerance we have for errors.

With a connectionless socket, packets can arrive out of order, so if we can't fit all our data in one packet, we will have to worry about ordering in our application. The maximum packet size is a characteristic of the communication protocol. Also, with a connectionless socket, the packets can be lost. If our application can't tolerate this loss, we should use connection-oriented sockets.

Tolerating packet loss means that we have two choices. If we intend to have reliable communication with our peer, we have to number our packets and request retransmission from the peer application when we detect a missing packet. We will also have to identify duplicate packets and discard them, since a packet might be delayed and appear to be lost, but show up after we have requested retransmission.

The other choice we have is to deal with the error by letting the user retry the command. For simple applications, this might be adequate, but for complex applications, this usually isn't a viable alternative, so it is generally better to use connection-oriented sockets in this case.

The drawbacks to connection-oriented sockets are that more work and time are needed to establish a connection, and each connection consumes more resources from the operating system.

Example—Connectionless Client

The program in Figure 16.17 is a version of the uptime client command that uses the datagram socket interface.

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

#define BUFLEN      128
#define TIMEOUT     20

void
sigalrm(int signo)
{
}

void
print_uptime(int sockfd, struct addrinfo *aip)
{
    int     n;
    char    buf[BUFLEN];

    buf[0] = 0;
    if (sendto(sockfd, buf, 1, 0, aip->ai_addr, aip->ai_addrlen) < 0)
        err_sys("sendto error");
    alarm(TIMEOUT);
    if ((n = recvfrom(sockfd, buf, BUFLEN, 0, NULL, NULL)) < 0) {
        if (errno != EINTR)
            alarm(0);
        err_sys("recv error");
    }
    alarm(0);
    write(STDOUT_FILENO, buf, n);
}
```

```
int
main(int argc, char *argv[])
{
    struct addrinfo     *ailist, *aip;
    struct addrinfo     hint;
    int                 sockfd, err;
    struct sigaction    sa;

    if (argc != 2)
        err_quit("usage: ruptime hostname");
    sa.sa_handler = sigalrm;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    if (sigaction(SIGALRM, &sa, NULL) < 0)
        err_sys("sigaction error");
    hint.ai_flags = 0;
    hint.ai_family = 0;
    hint.ai_socktype = SOCK_DGRAM;
    hint.ai_protocol = 0;
    hint.ai_addrlen = 0;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(argv[1], "ruptime", &hint, &ailist)) != 0)
        err_quit("getaddrinfo error: %s", gai_strerror(err));

    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = socket(aip->ai_family, SOCK_DGRAM, 0)) < 0) {
            err = errno;
        } else {
            print_uptime(sockfd, aip);
            exit(0);
        }
    }

    fprintf(stderr, "can't contact %s: %s\n", argv[1], strerror(err));
    exit(1);
}
```

Figure 16.17 Client command using datagram service

The `main` function for the datagram-based client is similar to the one for the connection-oriented client, with the addition of installing a signal handler for `SIGALRM`. We use the `alarm` function to avoid blocking indefinitely in the call to `recvfrom`.

With the connection-oriented protocol, we needed to connect to the server before exchanging data. The arrival of the connect request was enough for the server to determine that it needed to provide service to a client. But with the datagram-based protocol, we need a way to notify the server that we want it to perform its service on our behalf. In this example, we simply send the server a 1-byte message. The server will receive it, get our address from the packet, and use this address to transmit its