14

Advanced I/O

14.1 Introduction

This chapter covers numerous topics and functions that we lump under the term *advanced I/O*: nonblocking I/O, record locking, I/O multiplexing (the `select` and `poll` functions), asynchronous I/O, the `readv` and `writev` functions, and memory-mapped I/O (`mmap`). We need to cover these topics before describing interprocess communication in Chapter 15, Chapter 17, and many of the examples in later chapters.

14.2 Nonblocking I/O

In Section 10.5, we said that system calls are divided into two categories: the "slow" ones and all the others. The slow system calls are those that can block forever. They include

- Reads that can block the caller forever if data isn't present with certain file types (pipes, terminal devices, and network devices)
- Writes that can block the caller forever if the data can't be accepted immediately by these same file types (e.g., no room in the pipe, network flow control)
- Opens that block until some condition occurs on certain file types (such as an open of a terminal device that waits until an attached modem answers the phone, or an open of a FIFO for writing only, when no other process has the FIFO open for reading)
- Reads and writes of files that have mandatory record locking enabled

- Certain `ioctl` operations
- Some of the interprocess communication functions (Chapter 15)

We also said that system calls related to disk I/O are not considered slow, even though the read or write of a disk file can block the caller temporarily.

Nonblocking I/O lets us issue an I/O operation, such as an `open`, `read`, or `write`, and not have it block forever. If the operation cannot be completed, the call returns immediately with an error noting that the operation would have blocked.

There are two ways to specify nonblocking I/O for a given descriptor.

1. If we call `open` to get the descriptor, we can specify the `O_NONBLOCK` flag (Section 3.3).
2. For a descriptor that is already open, we call `fcntl` to turn on the `O_NONBLOCK` file status flag (Section 3.14). Figure 3.12 shows a function that we can call to turn on any of the file status flags for a descriptor.

> Earlier versions of System V used the flag `O_NDELAY` to specify nonblocking mode. These versions of System V returned a value of 0 from the `read` function if there wasn't any data to be read. Since this use of a return value of 0 overlapped with the normal UNIX System convention of 0 meaning the end of file, POSIX.1 chose to provide a nonblocking flag with a different name and different semantics. Indeed, with these older versions of System V, when we get a return of 0 from `read`, we don't know whether the call would have blocked or whether the end of file was encountered. We'll see that POSIX.1 requires that `read` return –1 with `errno` set to `EAGAIN` if there is no data to read from a nonblocking descriptor. Some platforms derived from System V support both the older `O_NDELAY` and the POSIX.1 `O_NONBLOCK`, but in this text we'll use only the POSIX.1 feature. The older `O_NDELAY` is intended for backward compatibility and should not be used in new applications.
>
> 4.3BSD provided the `FNDELAY` flag for `fcntl`, and its semantics were slightly different. Instead of affecting only the file status flags for the descriptor, the flags for either the terminal device or the socket were also changed to be nonblocking, thereby affecting all users of the terminal or socket, not just the users sharing the same file table entry (4.3BSD nonblocking I/O worked only on terminals and sockets). Also, 4.3BSD returned `EWOULDBLOCK` if an operation on a nonblocking descriptor could not complete without blocking. Today, BSD-based systems provide the POSIX.1 `O_NONBLOCK` flag and define `EWOULDBLOCK` to be the same as `EAGAIN`. These systems provide nonblocking semantics consistent with other POSIX-compatible systems: changes in file status flags affect all users of the same file table entry, but are independent of accesses to the same device through other file table entries. (Refer to Figures 3.7 and 3.9.)

Example

Let's look at an example of nonblocking I/O. The program in Figure 14.1 reads up to 500,000 bytes from the standard input and attempts to write it to the standard output. The standard output is first set to be nonblocking. The output is in a loop, with the results of each `write` being printed on the standard error. The function `clr_fl` is similar to the function `set_fl` that we showed in Figure 3.12. This new function simply clears one or more of the flag bits.

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

char    buf[500000];

int
main(void)
{
    int     ntowrite, nwrite;
    char    *ptr;

    ntowrite = read(STDIN_FILENO, buf, sizeof(buf));
    fprintf(stderr, "read %d bytes\n", ntowrite);

    set_fl(STDOUT_FILENO, O_NONBLOCK);  /* set nonblocking */

    ptr = buf;
    while (ntowrite > 0) {
        errno = 0;
        nwrite = write(STDOUT_FILENO, ptr, ntowrite);
        fprintf(stderr, "nwrite = %d, errno = %d\n", nwrite, errno);

        if (nwrite > 0) {
            ptr += nwrite;
            ntowrite -= nwrite;
        }
    }

    clr_fl(STDOUT_FILENO, O_NONBLOCK);  /* clear nonblocking */

    exit(0);
}
```

Figure 14.1 Large nonblocking `write`

If the standard output is a regular file, we expect the `write` to be executed once:

```
$ ls -l /etc/services                              print file size
-rw-r--r--  1 root     677959 Jun 23  2009 /etc/services
$ ./a.out < /etc/services > temp.file              try a regular file first
read 500000 bytes
nwrite = 500000, errno = 0                         a single write
$ ls -l temp.file                                  verify size of output file
-rw-rw-r--  1 sar      500000 Apr  1 13:03 temp.file
```

But if the standard output is a terminal, we expect the `write` to return a partial count sometimes and an error at other times. This is what we see:

```
$ ./a.out < /etc/services 2>stderr.out          output to terminal
                                                lots of output to terminal ...
$ cat stderr.out
read 500000 bytes
nwrite = 999, errno = 0
nwrite = -1, errno = 35
nwrite = -1, errno = 35
nwrite = -1, errno = 35
nwrite = -1, errno = 35
nwrite = 1001, errno = 0
nwrite = -1, errno = 35
nwrite = 1002, errno = 0
nwrite = 1004, errno = 0
nwrite = 1003, errno = 0
nwrite = 1003, errno = 0
nwrite = 1005, errno = 0
nwrite = -1, errno = 35                         61 of these errors
   .
   .
   .
nwrite = 1006, errno = 0
nwrite = 1004, errno = 0
nwrite = 1005, errno = 0
nwrite = 1006, errno = 0
nwrite = -1, errno = 35                         108 of these errors
   .
   .
   .
nwrite = 1006, errno = 0
nwrite = 1005, errno = 0
nwrite = 1005, errno = 0
nwrite = -1, errno = 35                         681 of these errors
   .
   .
   .
                                                and so on ...
nwrite = 347, errno = 0
```

On this system, the `errno` of 35 is `EAGAIN`. The amount of data accepted by the terminal driver varies from system to system. The results will also vary depending on how you are logged in to the system: on the system console, on a hard-wired terminal, on a network connection using a pseudo terminal. If you are running a windowing system on your terminal, you are also going through a pseudo terminal device. □

In this example, the program issues more than 9,000 `write` calls, even though only 500 are needed to output the data. The rest just return an error. This type of loop, called *polling*, is a waste of CPU time on a multiuser system. In Section 14.4, we'll see that I/O multiplexing with a nonblocking descriptor is a more efficient way to do this.

Sometimes, we can avoid using nonblocking I/O by designing our applications to use multiple threads (see Chapter 11). We can allow individual threads to block in I/O calls if we can continue to make progress in other threads. This can sometimes simplify the design, as we shall see in Chapter 21; at other times, however, the overhead of synchronization can add more complexity than is saved from using threads.

14.3 Record Locking

What happens when two people edit the same file at the same time? In most UNIX systems, the final state of the file corresponds to the last process that wrote the file. In some applications, however, such as a database system, a process needs to be certain that it alone is writing to a file. To provide this capability for processes that need it, commercial UNIX systems provide record locking. (In Chapter 20, we develop a database library that uses record locking.)

Record locking is the term normally used to describe the ability of a process to prevent other processes from modifying a region of a file while the first process is reading or modifying that portion of the file. Under the UNIX System, "record" is a misnomer; the UNIX kernel does not have a notion of records in a file. A better term is *byte-range locking*, given that it is a range of a file (possibly the entire file) that is locked.

History

One of the criticisms of early UNIX systems was that they couldn't be used to run database systems, because they did not support locking portions of files. As UNIX systems found their way into business computing environments, various groups added support for record locking (differently, of course).

Early Berkeley releases supported only the `flock` function. This function locks only entire files, not regions of a file.

Record locking was added to System V Release 3 through the `fcntl` function. The `lockf` function was built on top of this, providing a simplified interface. These functions allowed callers to lock arbitrary byte ranges in a file, ranging from the entire file down to a single byte within the file.

POSIX.1 chose to standardize on the `fcntl` approach. Figure 14.2 shows the forms of record locking provided by various systems. Note that the Single UNIX Specification includes `lockf` in the XSI option.

System	Advisory	Mandatory	`fcntl`	`lockf`	`flock`
SUS	•		•	XSI	
FreeBSD 8.0	•		•	•	•
Linux 3.2.0	•	•	•	•	•
Mac OS X 10.6.8	•		•	•	•
Solaris 10	•	•	•	•	•

Figure 14.2 Forms of record locking supported by various UNIX systems

We describe the difference between advisory locking and mandatory locking later in this section. In this text, we describe only the POSIX.1 `fcntl` locking.

> Record locking was originally added to Version 7 in 1980 by John Bass. The system call entry into the kernel was a function named `locking`. This function provided mandatory record locking and propagated through many versions of System III. Xenix systems picked up this function, and some Intel-based System V derivatives, such as OpenServer 5, continued to support it in a Xenix-compatibility library.

`fcntl` Record Locking

Let's repeat the prototype for the `fcntl` function from Section 3.14.

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* struct flock *flockptr */ );
```

Returns: depends on *cmd* if OK (see following), –1 on error

For record locking, *cmd* is `F_GETLK`, `F_SETLK`, or `F_SETLKW`. The third argument (which we'll call *flockptr*) is a pointer to an `flock` structure.

```
struct flock {
  short  l_type;   /* F_RDLCK, F_WRLCK, or F_UNLCK */
  short  l_whence; /* SEEK_SET, SEEK_CUR, or SEEK_END */
  off_t  l_start;  /* offset in bytes, relative to l_whence */
  off_t  l_len;    /* length, in bytes; 0 means lock to EOF */
  pid_t  l_pid;    /* returned with F_GETLK */
};
```

This structure describes

- The type of lock desired: `F_RDLCK` (a shared read lock), `F_WRLCK` (an exclusive write lock), or `F_UNLCK` (unlocking a region)
- The starting byte offset of the region being locked or unlocked (`l_start` and `l_whence`)
- The size of the region in bytes (`l_len`)
- The ID (`l_pid`) of the process holding the lock that can block the current process (returned by `F_GETLK` only)

Numerous rules apply to the specification of the region to be locked or unlocked.

- The two elements that specify the starting offset of the region are similar to the last two arguments of the `lseek` function (Section 3.6). Indeed, the `l_whence` member is specified as `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`.
- Locks can start and extend beyond the current end of file, but cannot start or extend before the beginning of the file.
- If `l_len` is 0, it means that the lock extends to the largest possible offset of the file. This allows us to lock a region starting anywhere in the file, up through and including any data that is appended to the file. (We don't have to try to guess how many bytes might be appended to the file.)
- To lock the entire file, we set `l_start` and `l_whence` to point to the beginning of the file and specify a length (`l_len`) of 0. (There are several ways to specify the beginning of the file, but most applications specify `l_start` as 0 and `l_whence` as `SEEK_SET`.)

We previously mentioned two types of locks: a shared read lock (`l_type` of `F_RDLCK`) and an exclusive write lock (`F_WRLCK`). The basic rule is that any number of processes can have a shared read lock on a given byte, but only one process can have an exclusive write lock on a given byte. Furthermore, if there are one or more read locks on a byte, there can't be any write locks on that byte; if there is an exclusive write lock on a byte, there can't be any read locks on that byte. We show this compatibility rule in Figure 14.3.

		Request for	
		read lock	write lock
Region currently has	no locks	OK	OK
	one or more read locks	OK	denied
	one write lock	denied	denied

Figure 14.3 Compatibility between different lock types

The compatibility rule applies to lock requests made from different processes, not to multiple lock requests made by a single process. If a process has an existing lock on a range of a file, a subsequent attempt to place a lock on the same range by the same process will replace the existing lock with the new one. Thus, if a process has a write lock on bytes 16–32 of a file and then tries to place a read lock on bytes 16–32, the request will succeed, and the write lock will be replaced by a read lock.

To obtain a read lock, the descriptor must be open for reading; to obtain a write lock, the descriptor must be open for writing.

We can now describe the three commands for the `fcntl` function.

`F_GETLK` Determine whether the lock described by *flockptr* is blocked by some other lock. If a lock exists that would prevent ours from being created, the information on that existing lock overwrites the information pointed to by *flockptr*. If no lock exists that would prevent ours from being created, the structure pointed to by *flockptr* is left unchanged except for the `l_type` member, which is set to `F_UNLCK`.

`F_SETLK` Set the lock described by *flockptr*. If we are trying to obtain a read lock (`l_type` of `F_RDLCK`) or a write lock (`l_type` of `F_WRLCK`) and the compatibility rule prevents the system from giving us the lock (Figure 14.3), `fcntl` returns immediately with `errno` set to either `EACCES` or `EAGAIN`.

> Although POSIX allows an implementation to return either error code, all four implementations described in this text return `EAGAIN` if the locking request cannot be satisfied.

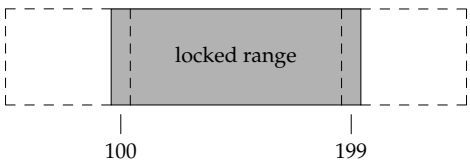
This command is also used to clear the lock described by *flockptr* (`l_type` of `F_UNLCK`).

`F_SETLKW` This command is a blocking version of `F_SETLK`. (The `W` in the command name means *wait*.) If the requested read lock or write lock cannot be granted because another process currently has some part of the requested region locked, the calling process is put to sleep. The process wakes up either when the lock becomes available or when interrupted by a signal.

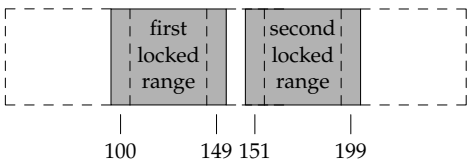
Be aware that testing for a lock with `F_GETLK` and then trying to obtain that lock with `F_SETLK` or `F_SETLKW` is not an atomic operation. We have no guarantee that, between the two `fcntl` calls, some other process won't come in and obtain the same lock. If we don't want to block while waiting for a lock to become available to us, we must handle the possible error returns from `F_SETLK`.

> Note that POSIX.1 doesn't specify what happens when one process read locks a range of a file, a second process blocks while trying to get a write lock on the same range, and a third processes then attempts to get another read lock on the range. If the third process is allowed to place a read lock on the range just because the range is already read locked, then the implementation might starve processes with pending write locks. Thus, as additional requests to read lock the same range arrive, the time that the process with the pending write-lock request has to wait is extended. If the read-lock requests arrive quickly enough without a lull in the arrival rate, then the writer could wait for a long time.

When setting or releasing a lock on a file, the system combines or splits adjacent areas as required. For example, if we lock bytes 100 through 199 and then unlock byte 150, the kernel still maintains the locks on bytes 100 through 149 and bytes 151 through 199. Figure 14.4 illustrates the byte-range locks in this situation.



File after locking bytes 100 through 199



File after unlocking byte 150

Figure 14.4 File byte-range lock diagram

If we were to lock byte 150, the system would coalesce the adjacent locked regions into a single region from byte 100 through 199. The resulting picture would be the first diagram in Figure 14.4, the same as when we started.

Example — Requesting and Releasing a Lock

To save ourselves from having to allocate an `flock` structure and fill in all the elements each time, the function `lock_reg` in Figure 14.5 handles all these details.

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

int
lock_reg(int fd, int cmd, int type, off_t offset, int whence, off_t len)
{
    struct flock    lock;

    lock.l_type = type;     /* F_RDLCK, F_WRLCK, F_UNLCK */
    lock.l_start = offset;  /* byte offset, relative to l_whence */
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;       /* #bytes (0 means to EOF) */

    return(fcntl(fd, cmd, &lock));
}
```

Figure 14.5 Function to lock or unlock a region of a file

Since most locking calls are to lock or unlock a region (the command `F_GETLK` is rarely used), we normally use one of the following five macros, which are defined in `apue.h` (Appendix B).

```
#define read_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_RDLCK, (offset), (whence), (len))
#define readw_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_RDLCK, (offset), (whence), (len))
#define write_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_WRLCK, (offset), (whence), (len))
#define writew_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_WRLCK, (offset), (whence), (len))
#define un_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_UNLCK, (offset), (whence), (len))
```

We have purposely defined the first three arguments to these macros in the same order as the `lseek` function. □

Example — Testing for a Lock

Figure 14.6 defines the function `lock_test` that we'll use to test for a lock.

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

pid_t
lock_test(int fd, int type, off_t offset, int whence, off_t len)
{
```

```
    struct flock    lock;

    lock.l_type = type;     /* F_RDLCK or F_WRLCK */
    lock.l_start = offset;  /* byte offset, relative to l_whence */
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;       /* #bytes (0 means to EOF) */

    if (fcntl(fd, F_GETLK, &lock) < 0)
        err_sys("fcntl error");

    if (lock.l_type == F_UNLCK)
        return(0);      /* false, region isn't locked by another proc */
    return(lock.l_pid); /* true, return pid of lock owner */
}
```

Figure 14.6 Function to test for a locking condition

If a lock exists that would block the request specified by the arguments, this function returns the process ID of the process holding the lock. Otherwise, the function returns 0 (false). We normally call this function from the following two macros (defined in `apue.h`):

```
#define is_read_lockable(fd, offset, whence, len) \
          (lock_test((fd), F_RDLCK, (offset), (whence), (len)) == 0)
#define is_write_lockable(fd, offset, whence, len) \
          (lock_test((fd), F_WRLCK, (offset), (whence), (len)) == 0)
```

Note that the `lock_test` function can't be used by a process to see whether it is currently holding a portion of a file locked. The definition of the `F_GETLK` command states that the information returned applies to an existing lock that would prevent us from creating our own lock. Since the `F_SETLK` and `F_SETLKW` commands always replace a process's existing lock if it exists, we can never block on our own lock; thus, the `F_GETLK` command will never report our own lock. □

Example — Deadlock

Deadlock occurs when two processes are each waiting for a resource that the other has locked. The potential for deadlock exists if a process that controls a locked region is put to sleep when it tries to lock another region that is controlled by a different process.

Figure 14.7 shows an example of deadlock. The child locks byte 0 and the parent locks byte 1. Each then tries to lock the other's already locked byte. We use the parent–child synchronization routines from Section 8.9 (`TELL_xxx` and `WAIT_xxx`) so that each process can wait for the other to obtain its lock.

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

static void
lockabyte(const char *name, int fd, off_t offset)
```

```
{
    if (writew_lock(fd, offset, SEEK_SET, 1) < 0)
        err_sys("%s: writew_lock error", name);
    printf("%s: got the lock, byte %lld\n", name, (long long)offset);
}

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

    /*
     * Create a file and write two bytes to it.
     */
    if ((fd = creat("templock", FILE_MODE)) < 0)
        err_sys("creat error");
    if (write(fd, "ab", 2) != 2)
        err_sys("write error");

    TELL_WAIT();
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {          /* child */
        lockabyte("child", fd, 0);
        TELL_PARENT(getppid());
        WAIT_PARENT();
        lockabyte("child", fd, 1);
    } else {                        /* parent */
        lockabyte("parent", fd, 1);
        TELL_CHILD(pid);
        WAIT_CHILD();
        lockabyte("parent", fd, 0);
    }
    exit(0);
}
```

Figure 14.7 Example of deadlock detection

Running the program in Figure 14.7 gives us

```
$ ./a.out
parent: got the lock, byte 1
child: got the lock, byte 0
parent: writew_lock error: Resource deadlock avoided
child: got the lock, byte 1
```

When a deadlock is detected, the kernel has to choose one process to receive the error return. In this example, the parent was chosen, but this is an implementation detail. On some systems, the child always receives the error. On other systems, the parent always gets the error. On some systems, you might even see the errors split between the child and the parent as multiple lock attempts are made. □

Implied Inheritance and Release of Locks

Three rules govern the automatic inheritance and release of record locks.

1. Locks are associated with a process and a file. This has two implications. The first is obvious: when a process terminates, all its locks are released. The second is far from obvious: whenever a descriptor is closed, any locks on the file referenced by that descriptor for that process are released. This means that if we make the calls

   ```
   fd1 = open(pathname, ...);
   read_lock(fd1, ...);
   fd2 = dup(fd1);
   close(fd2);
   ```

   after the `close(fd2)`, the lock that was obtained on `fd1` is released. The same thing would happen if we replaced the `dup` with `open`, as in

   ```
   fd1 = open(pathname, ...);
   read_lock(fd1, ...);
   fd2 = open(pathname, ...)
   close(fd2);
   ```

   to open the same file on another descriptor.
2. Locks are never inherited by the child across a `fork`. This means that if a process obtains a lock and then calls `fork`, the child is considered another process with regard to the lock that was obtained by the parent. The child has to call `fcntl` to obtain its own locks on any descriptors that were inherited across the `fork`. This constraint makes sense because locks are meant to prevent multiple processes from writing to the same file at the same time. If the child inherited locks across a `fork`, both the parent and the child could write to the same file at the same time.
3. Locks are inherited by a new program across an `exec`. Note, however, that if the close-on-exec flag is set for a file descriptor, all locks for the underlying file are released when the descriptor is closed as part of an `exec`.

FreeBSD Implementation

Let's take a brief look at the data structures used in the FreeBSD implementation. This should help clarify rule 1, which states that locks are associated with a process and a file.

Consider a process that executes the following statements (ignoring error returns):

```
fd1 = open(pathname, ...);
write_lock(fd1, 0, SEEK_SET, 1);    /* parent write locks byte 0 */
if ((pid = fork()) > 0) {           /* parent */
    fd2 = dup(fd1);
    fd3 = open(pathname, ...);
} else if (pid == 0) {
    read_lock(fd1, 1, SEEK_SET, 1); /* child read locks byte 1 */
}
pause();
```

Figure 14.8 shows the resulting data structures after both the parent and the child have paused.

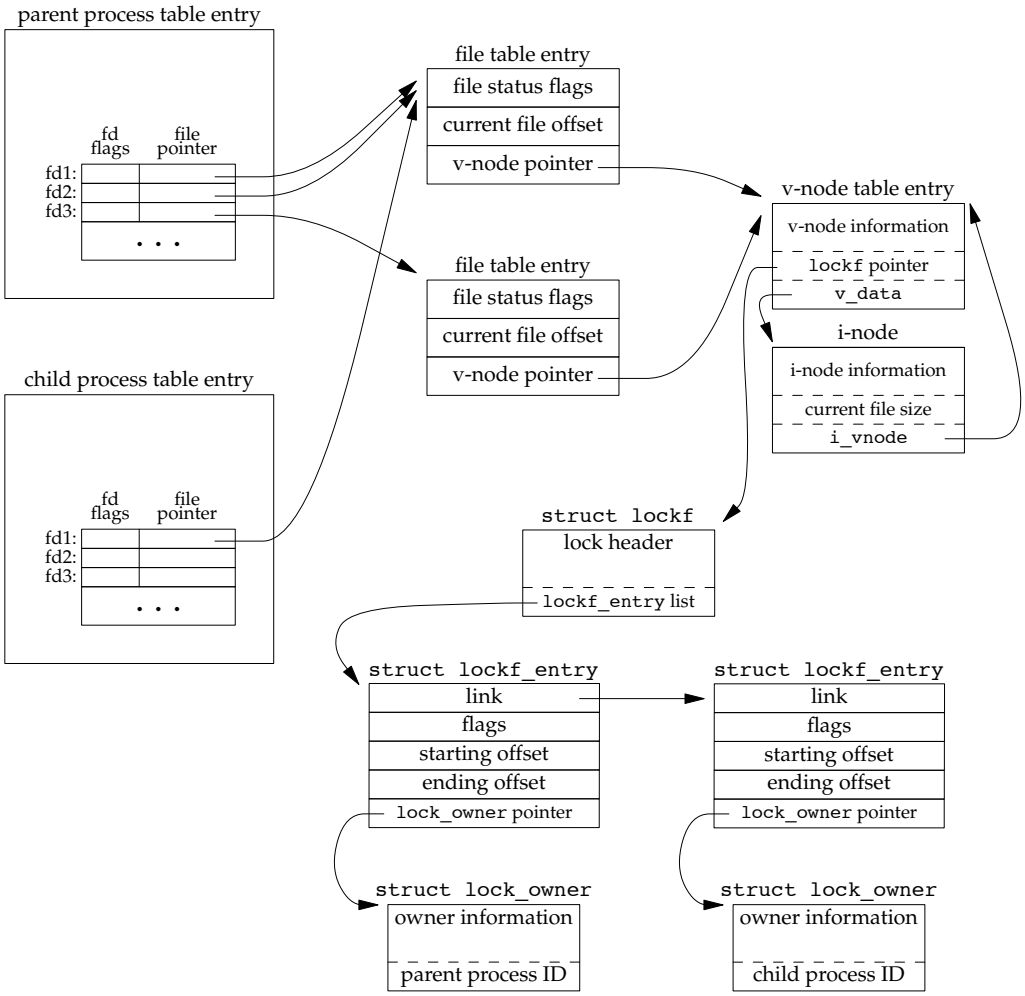


Figure 14.8 The FreeBSD data structures for record locking

We've shown the data structures that result from the `open`, `fork`, and `dup` calls earlier (Figures 3.9 and 8.2). What is new here are the `lockf` structures that are linked together from the i-node structure. Each `lockf` structure describes one locked region (defined by an offset and length) for a given process. We show two of these structures: one for the parent's call to `write_lock` and one for the child's call to `read_lock`. Each structure contains the corresponding process ID.

In the parent, closing any one of `fd1`, `fd2`, or `fd3` causes the parent's lock to be released. When any one of these three file descriptors is closed, the kernel goes through

the linked list of locks for the corresponding i-node and releases the locks held by the calling process. The kernel can't tell (and doesn't care) which descriptor of the three was used by the parent to obtain the lock.

Example

In the program in Figure 13.6, we saw how a daemon can use a lock on a file to ensure that only one copy of the daemon is running. Figure 14.9 shows the implementation of the `lockfile` function used by the daemon to place a write lock on a file.

```
#include <unistd.h>
#include <fcntl.h>

int
lockfile(int fd)
{
    struct flock fl;

    fl.l_type = F_WRLCK;
    fl.l_start = 0;
    fl.l_whence = SEEK_SET;
    fl.l_len = 0;
    return(fcntl(fd, F_SETLK, &fl));
}
```

Figure 14.9 Place a write lock on an entire file

Alternatively, we could define the `lockfile` function in terms of the `write_lock` function:

```
#define lockfile(fd) write_lock((fd), 0, SEEK_SET, 0)
```

□

Locks at End of File

We need to use caution when locking or unlocking byte ranges relative to the end of file. Most implementations convert an `l_whence` value of `SEEK_CUR` or `SEEK_END` into an absolute file offset, using `l_start` and the file's current position or current length. Often, however, we need to specify a lock relative to the file's current length, but we can't call `fstat` to obtain the current file size, since we don't have a lock on the file. (There's a chance that another process could change the file's length between the call to `fstat` and the lock call.)

Consider the following sequence of steps:

```
writew_lock(fd, 0, SEEK_END, 0);
write(fd, buf, 1);
un_lock(fd, 0, SEEK_END);
write(fd, buf, 1);
```

This sequence of code might not do what you expect. It obtains a write lock from the current end of the file onward, covering any future data we might append to the file.

Assuming that we are at end of file when we perform the first `write`, this operation will extend the file by one byte, and that byte will be locked. The unlock operation that follows has the effect of removing the locks for future writes that append data to the file, but it leaves a lock on the last byte in the file. When the second write occurs, the end of file is extended by one byte, but this byte is not locked. The state of the file locks for this sequence of steps is shown in Figure 14.10.

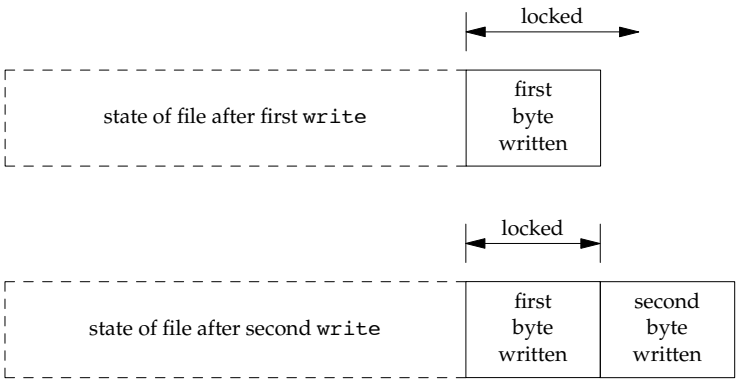


Figure 14.10 File range lock diagram

When a portion of a file is locked, the kernel converts the offset specified into an absolute file offset. In addition to specifying an absolute file offset (`SEEK_SET`), `fcntl` allows us to specify this offset relative to a point in the file: current (`SEEK_CUR`) or end of file (`SEEK_END`). The kernel needs to remember the locks independent of the current file offset or end of file, because the current offset and end of file can change, and changes to these attributes shouldn't affect the state of existing locks.

If we intended to remove the lock covering the byte we wrote in the first write, we could have specified the length as –1. Negative length values represent the bytes before the specified offset.

Advisory versus Mandatory Locking

Consider a library of database access routines. If all the functions in the library handle record locking in a consistent way, then we say that any set of processes using these functions to access a database are *cooperating processes*. It is feasible for these database access functions to use advisory locking if they are the only ones being used to access the database. But advisory locking doesn't prevent some other process that has write permission for the database file from writing whatever it wants to the database file. This rogue process would be an uncooperating process, since it's not using the accepted method (the library of database functions) to access the database.

Mandatory locking causes the kernel to check every `open`, `read`, and `write` to verify that the calling process isn't violating a lock on the file being accessed. Mandatory locking is sometimes called *enforcement-mode locking*.

We saw in Figure 14.2 that Linux 3.2.0 and Solaris 10 provide mandatory record locking, but FreeBSD 8.0 and Mac OS X 10.6.8 do not. Mandatory record locking is not part of the Single UNIX Specification. On Linux, if you want mandatory locking, you need to enable it on a per file system basis by using the `-o mand` option to the `mount` command.

Mandatory locking is enabled for a particular file by turning on the set-group-ID bit and turning off the group-execute bit. (Recall Figure 4.12.) Since the set-group-ID bit makes no sense when the group-execute bit is off, the designers of SVR3 chose this way to specify that the locking for a file is to be mandatory locking and not advisory locking.

What happens to a process that tries to `read` or `write` a file that has mandatory locking enabled and that part of the file is currently locked by another process? The answer depends on the type of operation (`read` or `write`), the type of lock held by the other process (read lock or write lock), and whether the descriptor for the `read` or `write` is nonblocking. Figure 14.11 shows the eight possibilities.

Type of existing lock on region held by other process	Blocking descriptor, tries to		Nonblocking descriptor, tries to	
	`read`	`write`	`read`	`write`
read lock	OK	blocks	OK	`EAGAIN`
write lock	blocks	blocks	`EAGAIN`	`EAGAIN`

Figure 14.11 Effect of mandatory locking on `read`s and `write`s by other processes

In addition to the `read` and `write` functions in Figure 14.11, the `open` function can be affected by mandatory record locks held by another process. Normally, `open` succeeds, even if the file being opened has outstanding mandatory record locks. The next `read` or `write` follows the rules listed in Figure 14.11. But if the file being opened has outstanding mandatory record locks (either read locks or write locks), and if the flags in the call to `open` specify either `O_TRUNC` or `O_CREAT`, then `open` returns an error of `EAGAIN` immediately, regardless of whether `O_NONBLOCK` is specified.

Only Solaris treats the `O_CREAT` flag as an error case. Linux allows the `O_CREAT` flag to be specified when opening a file with an outstanding mandatory lock. Generating the `open` error for `O_TRUNC` makes sense, because the file cannot be truncated if it is read locked or write locked by another process. Generating the error for `O_CREAT`, however, makes little sense; this flag says to create the file only if it doesn't already exist, but it has to exist to be record locked by another process.

This handling of locking conflicts with `open` can lead to surprising results. While developing the exercises in this section, a test program was run that opened a file (whose mode specified mandatory locking), established a read lock on an entire file, and then went to sleep for a while. (Recall from Figure 14.11 that a read lock should prevent writing to the file by other processes.) During this sleep period, the following behavior was seen in other typical UNIX System programs.

- The same file could be edited with the `ed` editor, and the results written back to disk! The mandatory record locking had no effect at all. Using the system call trace feature provided by some versions of the UNIX System, it was seen that `ed`

wrote the new contents to a temporary file, removed the original file, and then renamed the temporary file to be the original file. The mandatory record locking has no effect on the `unlink` function, which allowed this to happen.

> Under FreeBSD 8.0 and Solaris 10, we can obtain the system call trace of a process with the `truss`(1) command. Linux 3.2.0 provides the `strace`(1) command for the same purpose. Mac OS X 10.6.8 provides the `dtruss`(1m) command to trace system calls, but its use requires superuser privileges.

- The `vi` editor was never able to edit the file. It could read the file's contents, but whenever we tried to write new data to the file, `EAGAIN` was returned. If we tried to append new data to the file, the `write` blocked. This behavior from `vi` is what we expect.
- Using the Korn shell's `>` and `>>` operators to overwrite or append to the file resulted in the error "cannot create."
- Using the same two operators with the Bourne shell resulted in an error for `>`, but the `>>` operator just blocked until the mandatory lock was removed, and then proceeded. (The difference in the handling of the append operator occurs because the Korn shell `open`s the file with `O_CREAT` and `O_APPEND`, and we mentioned earlier that specifying `O_CREAT` generates an error. The Bourne shell, however, doesn't specify `O_CREAT` if the file already exists, so the `open` succeeds but the next `write` blocks.)

Results will vary, depending on the version of the operating system you are using. The bottom line, as demonstrated by this exercise, is to be wary of mandatory record locking. As seen with the `ed` example, it can be circumvented.

Mandatory record locking can also be used by a malicious user to hold a read lock on a file that is publicly readable. This can prevent anyone from writing to the file. (Of course, the file has to have mandatory record locking enabled for this to occur, which may require the user to be able to change the permission bits of the file.) Consider a database file that is world readable and has mandatory record locking enabled. If a malicious user were to hold a read lock on the entire file, the file could not be written to by other processes.

Example

We can run the program in Figure 14.12 to determine whether our system supports mandatory locking.

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

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

```
    pid_t           pid;
    char            buf[5];
    struct stat     statbuf;

    if (argc != 2) {
        fprintf(stderr, "usage: %s filename\n", argv[0]);
        exit(1);
    }
    if ((fd = open(argv[1], O_RDWR | O_CREAT | O_TRUNC, FILE_MODE)) < 0)
        err_sys("open error");
    if (write(fd, "abcdef", 6) != 6)
        err_sys("write error");

    /* turn on set-group-ID and turn off group-execute */
    if (fstat(fd, &statbuf) < 0)
        err_sys("fstat error");
    if (fchmod(fd, (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
        err_sys("fchmod error");

    TELL_WAIT();

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {   /* parent */
        /* write lock entire file */
        if (write_lock(fd, 0, SEEK_SET, 0) < 0)
            err_sys("write_lock error");

        TELL_CHILD(pid);

        if (waitpid(pid, NULL, 0) < 0)
            err_sys("waitpid error");
    } else {                /* child */
        WAIT_PARENT();      /* wait for parent to set lock */

        set_fl(fd, O_NONBLOCK);

        /* first let's see what error we get if region is locked */
        if (read_lock(fd, 0, SEEK_SET, 0) != -1)    /* no wait */
            err_sys("child: read_lock succeeded");
        printf("read_lock of already-locked region returns %d\n",
          errno);

        /* now try to read the mandatory locked file */
        if (lseek(fd, 0, SEEK_SET) == -1)
            err_sys("lseek error");
        if (read(fd, buf, 2) < 0)
            err_ret("read failed (mandatory locking works)");
        else
            printf("read OK (no mandatory locking), buf = %2.2s\n",
              buf);
    }
    exit(0);
}
```

Figure 14.12 Determine whether mandatory locking is supported

This program creates a file and enables mandatory locking for the file. The program then splits into parent and child, with the parent obtaining a write lock on the entire file. The child first sets its descriptor to be nonblocking and then attempts to obtain a read lock on the file, expecting to get an error. This lets us see whether the system returns `EACCES` or `EAGAIN`. Next, the child rewinds the file and tries to `read` from the file. If mandatory locking is provided, the `read` should return `EACCES` or `EAGAIN` (since the descriptor is nonblocking). Otherwise, the `read` returns the data that it read. Running this program under Solaris 10 (which supports mandatory locking) gives us

```
$ ./a.out temp.lock
read_lock of already-locked region returns 11
read failed (mandatory locking works): Resource temporarily unavailable
```

If we look at either the system's headers or the `intro`(2) manual page, we see that an `errno` of 11 corresponds to `EAGAIN`. Under FreeBSD 8.0, we get

```
$ ./a.out temp.lock
read_lock of already-locked region returns 35
read OK (no mandatory locking), buf = ab
```

Here, an `errno` of 35 corresponds to `EAGAIN`. Mandatory locking is not supported. □

Example

Let's return to the first question posed in this section: what happens when two people edit the same file at the same time? The normal UNIX System text editors do not use record locking, so the answer is still that the final result of the file corresponds to the last process that wrote the file.

Some versions of the `vi` editor use advisory record locking. Even if we were using one of these versions of `vi`, it still doesn't prevent users from running another editor that doesn't use advisory record locking.

If the system provides mandatory record locking, we could modify our favorite editor to use it (if we have the editor's source code). Not having the source code for the editor, we might try the following. We write our own program that is a front end to `vi`. This program immediately calls `fork`, and the parent just waits for the child to complete. The child opens the file specified on the command line, enables mandatory locking, obtains a write lock on the entire file, and then executes `vi`. While `vi` is running, the file is write locked, so other users can't modify it. When `vi` terminates, the parent's `wait` returns and our front end terminates.

A small front-end program of this type can be written, but it doesn't work. The problem is that it is common practice for editors to read their input file and then close it. A lock is released on a file whenever a descriptor that references that file is closed. As a result, when the editor closes the file after reading its contents, the lock is gone. There is no way to prevent this from happening in the front-end program. □

We'll use record locking in Chapter 20 in our database library to provide concurrent access to multiple processes. We'll also provide some timing measurements to see how record locking affects a process.

14.4 I/O Multiplexing

When we read from one descriptor and write to another, we can use blocking I/O in a loop, such as

```
while ((n = read(STDIN_FILENO, buf, BUFSIZ)) > 0)
    if (write(STDOUT_FILENO, buf, n) != n)
        err_sys("write error");
```

We see this form of blocking I/O over and over again. What if we have to read from two descriptors? In this case, we can't do a blocking `read` on either descriptor, as data may appear on one descriptor while we're blocked in a `read` on the other. A different technique is required to handle this case.

Let's look at the structure of the `telnet`(1) command. In this program, we read from the terminal (standard input) and write to a network connection, and we read from the network connection and write to the terminal (standard output). At the other end of the network connection, the `telnetd` daemon reads what we typed and presents it to a shell as if we were logged in to the remote machine. The `telnetd` daemon sends any output generated by the commands we type back to us through the `telnet` command, to be displayed on our terminal. Figure 14.13 shows a picture of this arrangement.

Figure 14.13 Overview of `telnet` program

The `telnet` process has two inputs and two outputs. We can't do a blocking `read` on either of the inputs, as we never know which input will have data for us.

One way to handle this particular problem is to divide the process in two pieces (using `fork`), with each half handling one direction of data. We show this in Figure 14.14. (The `cu`(1) command provided with System V's `uucp` communication package was structured like this.)

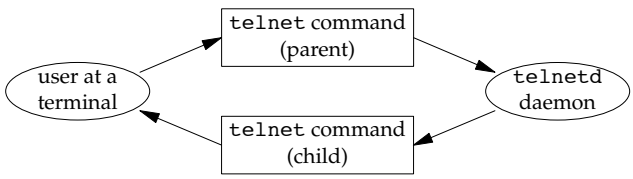


Figure 14.14 The `telnet` program using two processes

If we use two processes, we can let each process do a blocking `read`. But this leads to a problem when the operation terminates. If an end of file is received by the child (the

network connection is disconnected by the `telnetd` daemon), then the child terminates and the parent is notified by the `SIGCHLD` signal. But if the parent terminates (the user enters an end-of-file character at the terminal), then the parent has to tell the child to stop. We can use a signal for this (`SIGUSR1`, for example), but it does complicate the program somewhat.

Instead of two processes, we could use two threads in a single process. This avoids the termination complexity, but requires that we deal with synchronization between the threads, which could add more complexity than it saves.

We could use nonblocking I/O in a single process by setting both descriptors to be nonblocking and issuing a `read` on the first descriptor. If data is present, we read it and process it. If there is no data to read, the call returns immediately. We then do the same thing with the second descriptor. After this, we wait for some amount of time (a few seconds, perhaps) and then try to read from the first descriptor again. This type of loop is called *polling*. The problem is that it wastes CPU time. Most of the time, there won't be data to read, so we waste time performing the `read` system calls. We also have to guess how long to wait each time around the loop. Although it works on any system that supports nonblocking I/O, polling should be avoided on a multitasking system.

Another technique is called *asynchronous I/O*. With this technique, we tell the kernel to notify us with a signal when a descriptor is ready for I/O. There are two problems with this approach. First, although systems provide their own limited forms of asynchronous I/O, POSIX chose to standardize a different set of interfaces, so portability can be an issue. (In the past, POSIX asynchronous I/O was an optional facility in the Single UNIX Specification, but these interfaces are required as of SUSv4.) System V provides the `SIGPOLL` signal to support a limited form of asynchronous I/O, but this signal works only if the descriptor refers to a STREAMS device. BSD has a similar signal, `SIGIO`, but it has similar limitations: it works only on descriptors that refer to terminal devices or networks.

The second problem with this technique is that the limited forms use only one signal per process (`SIGPOLL` or `SIGIO`). If we enable this signal for two descriptors (in the example we've been talking about, reading from two descriptors), the occurrence of the signal doesn't tell us which descriptor is ready. Although the POSIX.1 asynchronous I/O interfaces allow us to select which signal to use for notification, the number of signals we can use is still far less than the number of possible open file descriptors. To determine which descriptor is ready, we would need to set each file descriptor to nonblocking mode and try the descriptors in sequence. We discuss asynchronous I/O in Section 14.5.

A better technique is to use *I/O multiplexing*. To do this, we build a list of the descriptors that we are interested in (usually more than one descriptor) and call a function that doesn't return until one of the descriptors is ready for I/O. Three functions—`poll`, `pselect`, and `select`—allow us to perform I/O multiplexing. On return from these functions, we are told which descriptors are ready for I/O.

> POSIX specifies that `<sys/select.h>` be included to pull the information for `select` into your program. Older systems require that you include `<sys/types.h>`, `<sys/time.h>`, and `<unistd.h>`. Check the `select` manual page to see what your system supports.

I/O multiplexing was provided with the `select` function in 4.2BSD. This function has always worked with any descriptor, although its main use has been for terminal I/O and network I/O. SVR3 added the `poll` function when the STREAMS mechanism was added. Initially, `poll` worked only with STREAMS devices. In SVR4, support was added to allow `poll` to work on any descriptor.

14.4.1 `select` and `pselect` Functions

The `select` function lets us do I/O multiplexing under all POSIX-compatible platforms. The arguments we pass to `select` tell the kernel

- Which descriptors we're interested in.
- Which conditions we're interested in for each descriptor. (Do we want to read from a given descriptor? Do we want to write to a given descriptor? Are we interested in an exception condition for a given descriptor?)
- How long we want to wait. (We can wait forever, wait a fixed amount of time, or not wait at all.)

On the return from `select`, the kernel tells us

- The total count of the number of descriptors that are ready
- Which descriptors are ready for each of the three conditions (read, write, or exception condition)

With this return information, we can call the appropriate I/O function (usually `read` or `write`) and know that the function won't block.

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

int select(int maxfdp1, fd_set *restrict readfds,
           fd_set *restrict writefds, fd_set *restrict exceptfds,
           struct timeval *restrict tvptr);
```

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

Let's look at the last argument first. It specifies how long we want to wait in terms of seconds and microseconds (recall Section 4.20). There are three conditions.

tvptr `== NULL`

Wait forever. This infinite wait can be interrupted if we catch a signal. Return is made when one of the specified descriptors is ready or when a signal is caught. If a signal is caught, `select` returns –1 with `errno` set to `EINTR`.

tvptr->tv_sec `==` 0 `&&` *tvptr->tv_usec* `==` 0

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

tvptr->tv_sec `!=` 0 || *tvptr->tv_usec* `!=` 0

Wait the specified number of seconds and microseconds. Return is made when one of the specified descriptors is ready or when the timeout value expires. If the timeout expires before any of the descriptors is ready, the return value is 0. (If the system doesn't provide microsecond resolution, the *tvptr->tv_usec* value is rounded up to the nearest supported value.) As with the first condition, this wait can also be interrupted by a caught signal.

POSIX.1 allows an implementation to modify the `timeval` structure, so after `select` returns, you can't rely on the structure containing the same values it did before calling `select`. FreeBSD 8.0, Mac OS X 10.6.8, and Solaris 10 all leave the structure unchanged, but Linux 3.2.0 will update it with the time remaining if `select` returns before the timeout value expires.

The middle three arguments—*readfds*, *writefds*, and *exceptfds*—are pointers to *descriptor sets*. These three sets specify which descriptors we're interested in and for which conditions (readable, writable, or an exception condition). A descriptor set is stored in an `fd_set` data type. This data type is chosen by the implementation so that it can hold one bit for each possible descriptor. We can consider it to be just a big array of bits, as shown in Figure 14.15.

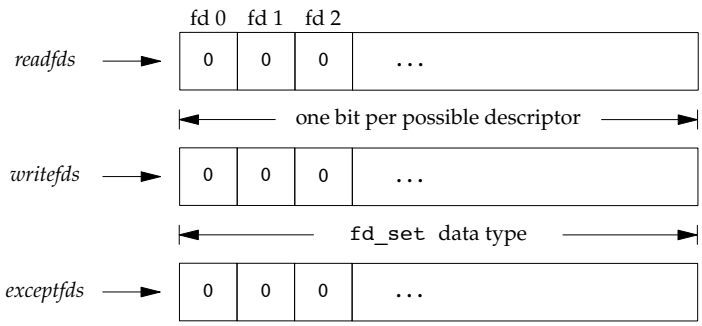


Figure 14.15 Specifying the read, write, and exception descriptors for `select`

The only thing we can do with the `fd_set` data type is allocate a variable of this type, assign a variable of this type to another variable of the same type, or use one of the following four functions on a variable of this type.

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

int FD_ISSET(int fd, fd_set *fdset);

                                        Returns: nonzero if fd is in set, 0 otherwise

void FD_CLR(int fd, fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_ZERO(fd_set *fdset);
```

These interfaces can be implemented as either macros or functions. An `fd_set` is set to all zero bits by calling `FD_ZERO`. To turn on a single bit in a set, we use `FD_SET`. We can clear a single bit by calling `FD_CLR`. Finally, we can test whether a given bit is turned on in the set with `FD_ISSET`.

After declaring a descriptor set, we must zero the set using `FD_ZERO`. We then set bits in the set for each descriptor that we're interested in, as in

```
fd_set   rset;
int      fd;

FD_ZERO(&rset);
FD_SET(fd, &rset);
FD_SET(STDIN_FILENO, &rset);
```

On return from `select`, we can test whether a given bit in the set is still on using `FD_ISSET`:

```
if (FD_ISSET(fd, &rset)) {
      .
      .
      .
}
```

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 is 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 one second; the actual resolution depends on the system's clock.) Exercise 14.5 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 14.16 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).

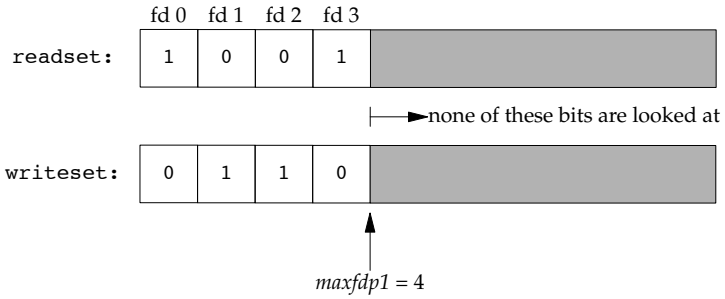


Figure 14.16 Example descriptor sets for `select`

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 4.2.) 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.

14.4.2 `poll` Function

The `poll` function is similar to `select`, but the programmer interface is different. This function was originally introduced in System V to support the STREAMS subsystem, but 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 from Figure 2.21.) 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 which 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 14.17. On return, the revents member is set by the kernel, thereby specifying which events have occurred for each descriptor. (Note that poll doesn't change the events member. This behavior differs from that of select, which modifies its arguments to indicate what is ready.)

Name	Input to events?	Result from revents?	Description
POLLIN	•	•	Data other than high priority data can be read without blocking (equivalent to POLLRDNORM | POLLRDBAND).
POLLRDNORM	•	•	Normal data can be read without blocking.
POLLRDBAND	•	•	Priority data 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	•	•	Priority data 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 14.17 The events and revents flags for poll

The first four rows of Figure 14.17 test for readability, the next three test for writability, and the final three are for exception conditions. The last three rows in Figure 14.17 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.

> The poll event names containing the term *BAND* refer to priority bands in STREAMS. Refer to Rago [1993] for more information about STREAMS and priority bands.

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 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.

14.5 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. As mentioned in the previous section, these forms of asynchronous I/O are limited: they don't work with all file types and they allow the use of only one signal. 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.

Version 4 of the Single UNIX Specification moved the general asynchronous I/O mechanism from the real-time extensions to the base specification. This mechanism addresses the limitations that exist with these older asynchronous I/O facilities.

Before we look at the different ways to use asynchronous I/O, we need to discuss the costs. When we decide to use asynchronous I/O, we complicate the design of our application by choosing to juggle multiple concurrent operations. A simpler approach may be to use multiple threads, which would allow us to write the program using a synchronous model, and let the threads run asynchronous to each other.

We incur additional complexity when we use the POSIX asynchronous I/O interfaces:

- We have to worry about three sources of errors for every asynchronous operation: one associated with the submission of the operation, one associated with the result of the operation itself, and one associated with the functions used to determine the status of the asynchronous operations.
- The interfaces themselves involve a lot of extra setup and processing rules compared to their conventional counterparts, as we shall see.

  We can't really call the non-asynchronous I/O function calls "synchronous," because although they are synchronous with respect to the program flow, they aren't synchronous with respect to the I/O. Recall the discussion of synchronous writes in Chapter 3. We call a write "synchronous" if the data we write is persistent when we return from the call to the `write` function. We also can't differentiate the conventional I/O function calls from the asynchronous ones by referring to the conventional calls as the "standard" I/O calls, because this confuses them with the function calls in the standard I/O library. To avoid confusion, we'll refer to the `read` and `write` functions as the "conventional" I/O function calls in this section.

- Recovering from errors can be difficult. For example, if we submit multiple asynchronous writes and one fails, how should we proceed? If the writes are related, we might have to undo the ones that succeeded.

14.5.1 System V Asynchronous I/O

System V provides a limited form of asynchronous I/O that 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 14.18. These constants are defined in `<stropts.h>`.

> Interfaces related to the STREAMS mechanism were marked obsolescent in SUSv4, so we don't cover them in any detail. See Rago [1993] for more information about STREAMS.

Constant	Description
`S_INPUT`	We can read data (other than high-priority data) without blocking.
`S_RDNORM`	We can read normal data without blocking.
`S_RDBAND`	We can read priority data without blocking.
`S_BANDURG`	If this constant is specified with `S_RDBAND`, the `SIGURG` signal is generated instead of `SIGPOLL` when we can read priority data without blocking.
`S_HIPRI`	We can read high-priority data without blocking.
`S_OUTPUT`	We can write normal data without blocking.
`S_WRNORM`	Same as `S_OUTPUT`.
`S_WRBAND`	We can write priority data without blocking.
`S_MSG`	The `SIGPOLL` signal message has reached the stream head.
`S_ERROR`	The stream has an error.
`S_HANGUP`	The stream has hung up.

Figure 14.18 Conditions for generating `SIGPOLL` signal

In addition to calling `ioctl` to specify the conditions that should generate the `SIGPOLL` signal, we 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`.

14.5.2 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 3.10).

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, such as TCP connections.

14.5.3 POSIX Asynchronous I/O

The POSIX asynchronous I/O interfaces give us a consistent way to perform asynchronous I/O, regardless of the type of file. These interfaces were adopted from the real-time draft standard, which themselves were an option in the Single UNIX Specification. In Version 4, the Single UNIX Specification moved these interfaces to the base, so they are now required to be supported by all platforms.

The asynchronous I/O interfaces use AIO control blocks to describe I/O operations. The `aiocb` structure defines an AIO control block. It contains at least the fields shown in the following structure (implementations might include additional fields):

```
struct aiocb {
  int             aio_fildes;      /* file descriptor */
  off_t           aio_offset;      /* file offset for I/O */
  volatile void  *aio_buf;         /* buffer for I/O */
  size_t          aio_nbytes;      /* number of bytes to transfer */
  int             aio_reqprio;     /* priority */
  struct sigevent aio_sigevent;    /* signal information */
  int             aio_lio_opcode;  /* operation for list I/O */
};
```

The `aio_fildes` field is the file descriptor open for the file to be read or written. The read or write starts at the offset specified by `aio_offset`. For a read, data is copied to the buffer that begins at the address specified by `aio_buf`. For a write, data is copied from this buffer. The `aio_nbytes` field contains the number of bytes to read or write.

Note that we have to provide an explicit offset when we perform asynchronous I/O. The asynchronous I/O interfaces don't affect the file offset maintained by the operating system. This won't be a problem as long as we never mix asynchronous I/O functions with conventional I/O functions on the same file in a process. Also note that if we write to a file opened in append mode (with `O_APPEND`) using an asynchronous interface, the `aio_offset` field in the AIO control block is ignored by the system.

The other fields don't correspond to the conventional I/O functions. The `aio_reqprio` field is a hint that gives applications a way to suggest an ordering for the asynchronous I/O requests. The system has only limited control over the exact ordering, however, so there is no guarantee that the hint will be honored. The `aio_lio_opcode` field is used only with list-based asynchronous I/O, which we'll

discuss shortly. The aio_sigevent field controls how the application is notified about the completion of the I/O event. It is described by a sigevent structure.

```
struct sigevent {
  int             sigev_notify;                 /* notify type */
  int             sigev_signo;                  /* signal number */
  union sigval    sigev_value;                  /* notify argument */
  void (*sigev_notify_function)(union sigval); /* notify function */
  pthread_attr_t *sigev_notify_attributes;      /* notify attrs */
};
```

The sigev_notify field controls the type of notification. It can take on one of three values.

SIGEV_NONE The process is not notified when the asynchronous I/O request completes.

SIGEV_SIGNAL The signal specified by the sigev_signo field is generated when the asynchronous I/O request completes. If the application has elected to catch the signal and has specified the SA_SIGINFO flag when establishing the signal handler, the signal is queued (if the implementation supports queued signals). The signal handler is passed a siginfo structure whose si_value field is set to sigev_value (again, if SA_SIGINFO is used).

SIGEV_THREAD The function specified by the sigev_notify_function field is called when the asynchronous I/O request completes. It is passed the sigev_value field as its only argument. The function is executed in a separate thread in a detached state, unless the sigev_notify_attributes field is set to the address of a pthread attribute structure specifying alternative attributes for the thread.

To perform asynchronous I/O, we need to initialize an AIO control block and call either the aio_read function to make an asynchronous read or the aio_write function to make an asynchronous write.

```
#include <aio.h>

int aio_read(struct aiocb *aiocb);

int aio_write(struct aiocb *aiocb);
```

Both return: 0 if OK, –1 on error

When these functions return success, the asynchronous I/O request has been queued for processing by the operating system. The return value bears no relation to the result of the actual I/O operation. While the I/O operation is pending, we have to be careful to ensure that the AIO control block and data buffer remain stable; their underlying memory must remain valid and we can't reuse them until the I/O operation completes.

To force all pending asynchronous writes to persistent storage without waiting, we can set up an AIO control block and call the aio_fsync function.

```
#include <aio.h>

int aio_fsync(int op, struct aiocb *aiocb);
```

Returns: 0 if OK, –1 on error

The `aio_fildes` field in the AIO control block indicates the file whose asynchronous writes are synched. If the *op* argument is set to `O_DSYNC`, then the operation behaves like a call to `fdatasync`. Otherwise, if *op* is set to `O_SYNC`, the operation behaves like a call to `fsync`.

Like the `aio_read` and `aio_write` functions, the `aio_fsync` operation returns when the synch is scheduled. The data won't be persistent until the asynchronous synch completes. The AIO control block controls how we are notified, just as with the `aio_read` and `aio_write` functions.

To determine the completion status of an asynchronous read, write, or synch operation, we need to call the `aio_error` function.

```
#include <aio.h>

int aio_error(const struct aiocb *aiocb);
```

Returns: (see following)

The return value tells us one of four things.

`0`	The asynchronous operation completed successfully. We need to call the `aio_return` function to obtain the return value from the operation.
–1	The call to `aio_error` failed. In this case, `errno` tells us why.
`EINPROGRESS`	The asynchronous read, write, or synch is still pending.
anything else	Any other return value gives us the error code corresponding to the failed asynchronous operation.

If the asynchronous operation succeeded, we can call the `aio_return` function to get the asynchronous operation's return value.

```
#include <aio.h>

ssize_t aio_return(const struct aiocb *aiocb);
```

Returns: (see following)

Until the asynchronous operation completes, we need to be careful to avoid calling the `aio_return` function. The results are undefined until the operation completes. We also need to be careful to call `aio_return` only one time per asynchronous I/O operation. Once we call this function, the operating system is free to deallocate the record containing the I/O operation's return value.

The `aio_return` function will return –1 and set `errno` if `aio_return` itself fails. Otherwise, it will return the results of the asynchronous operation. In this case, it will return whatever `read`, `write`, or `fsync` would have returned on success if one of those functions had been called.

We use asynchronous I/O when we have other processing to do and we don't want to block while performing the I/O operation. However, when we have completed the processing and find that we still have asynchronous operations outstanding, we can call the `aio_suspend` function to block until an operation completes.

```
#include <aio.h>

int aio_suspend(const struct aiocb *const list[], int nent,
                const struct timespec *timeout);
```

Returns: 0 if OK, –1 on error

One of three things can cause `aio_suspend` to return. If we are interrupted by a signal, it returns –1 with `errno` set to `EINTR`. If the time limit specified by the optional *timeout* argument expires without any of the I/O operations completing, then `aio_suspend` returns –1 with `errno` set to `EAGAIN` (we can pass a null pointer for the *timeout* argument if we want to block without a time limit). If any of the I/O operations complete, `aio_suspend` returns 0. If all asynchronous I/O operations are complete when we call `aio_suspend`, then `aio_suspend` will return without blocking.

The *list* argument is a pointer to an array of AIO control blocks and the *nent* argument indicates the number of entries in the array. Null pointers in the array are skipped; the other entries must point to AIO control blocks that have been used to initiate asynchronous I/O operations.

When we have pending asynchronous I/O operations that we no longer want to complete, we can attempt to cancel them with the `aio_cancel` function.

```
#include <aio.h>

int aio_cancel(int fd, struct aiocb *aiocb);
```

Returns: (see following)

The *fd* argument specifies the file descriptor with the outstanding asynchronous I/O operations. If the *aiocb* argument is `NULL`, then the system attempts to cancel all outstanding asynchronous I/O operations on the file. Otherwise, the system attempts to cancel the single asynchronous I/O operation described by the AIO control block. We say that the system "attempts" to cancel the operations, because there is no guarantee that the system will be able to cancel any operations that are in progress.

The `aio_cancel` function can return one of four values:

`AIO_ALLDONE`	All of the operations completed before the attempt to cancel them.
`AIO_CANCELED`	All of the requested operations have been canceled.
`AIO_NOTCANCELED`	At least one of the requested operations could not be canceled.
–1	The call to `aio_cancel` failed. The error code will be stored in `errno`.

If an asynchronous I/O operation is successfully canceled, calling the `aio_error` function on the corresponding AIO control block will return the error `ECANCELED`. If the operation can't be canceled, then the corresponding AIO control block is unchanged by the call to `aio_cancel`.

One additional function is included with the asynchronous I/O interfaces, although it can be used in either a synchronous or an asynchronous manner. The `lio_listio` function submits a set of I/O requests described by a list of AIO control blocks.

```
#include <aio.h>

int lio_listio(int mode, struct aiocb *restrict const list[restrict],
               int nent, struct sigevent *restrict sigev);
```

Returns: 0 if OK, –1 on error

The *mode* argument determines whether the I/O is truly asynchronous. When it is set to `LIO_WAIT`, the `lio_listio` function won't return until all of the I/O operations specified by the list are complete. In this case, the *sigev* argument is ignored. When the *mode* argument is set to `LIO_NOWAIT`, then the `lio_listio` function returns as soon as the I/O requests are queued. The process is notified asynchronously when all of the I/O operations complete, as specified by the *sigev* argument. If we don't want to be notified, we can set *sigev* to `NULL`. Note that the individual AIO control blocks themselves may also enable asynchronous notification when an individual operation completes. The asynchronous notification specified by the *sigev* argument is in addition to these, and is sent only when all of the I/O operations complete.

The *list* argument points to a list of AIO control blocks specifying the I/O operations to perform. The *nent* argument specifies the number of elements in the array. The list of AIO control blocks can contain `NULL` pointers; these entries are ignored.

In each AIO control block, the `aio_lio_opcode` field specifies whether the operation is a read (`LIO_READ`), a write (`LIO_WRITE`), or a no-op (`LIO_NOP`), which is ignored. A read is treated as if the corresponding AIO control block had been passed to the `aio_read` function. Similarly, a write is treated as if the AIO control block had been passed to `aio_write`.

Implementations can limit the number of asynchronous I/O operations we are allowed to have outstanding. The limits are runtime invariants, and are summarized in Figure 14.19.

Name	Description	Minimum acceptable value
`AIO_LISTIO_MAX`	maximum number of I/O operations in a single list I/O call	`_POSIX_AIO_LISTIO_MAX (2)`
`AIO_MAX`	maximum number of outstanding asynchronous I/O operations	`_POSIX_AIO_MAX (1)`
`AIO_PRIO_DELTA_MAX`	maximum amount by which a process can decrease its asynchronous I/O priority level	0

Figure 14.19 POSIX.1 runtime invariant values for asynchronous I/O

We can determine the value of AIO_LISTIO_MAX by calling the sysconf function with the *name* argument set to _SC_IO_LISTIO_MAX. Similarly, we can determine the value of AIO_MAX by calling sysconf with the *name* argument set to _SC_AIO_MAX, and we can get the value of AIO_PRIO_DELTA_MAX by calling sysconf with its argument set to _SC_AIO_PRIO_DELTA_MAX.

The POSIX asynchronous I/O interfaces were originally introduced to provide real-time applications with a way to avoid being blocked while performing I/O operations. Now we'll look at an example of how to use the interfaces.

Example

We don't discuss real-time programming in this text, but because the POSIX asynchronous I/O interfaces are now part of the base specification in the Single UNIX Specification, we'll look at how to use them. To compare the asynchronous I/O interfaces with their conventional counterparts, we'll look at the task of translating a file from one format to another.

The program shown in Figure 14.20 translates a file using the ROT-13 algorithm that the USENET news system, popular in the 1980s, used to obscure text that might be offensive or contain spoilers or joke punchlines. The algorithm rotates the characters 'a' to 'z' and 'A' to 'Z' by 13 positions, but leaves all other characters unchanged.

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

#define BSZ 4096

unsigned char buf[BSZ];

unsigned char
translate(unsigned char c)
{
    if (isalpha(c)) {
        if (c >= 'n')
            c -= 13;
        else if (c >= 'a')
            c += 13;
        else if (c >= 'N')
            c -= 13;
        else
            c += 13;
    }
    return(c);
}

int
main(int argc, char* argv[])
{
    int ifd, ofd, i, n, nw;
```

```
    if (argc != 3)
        err_quit("usage: rot13 infile outfile");
    if ((ifd = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s", argv[1]);
    if ((ofd = open(argv[2], O_RDWR|O_CREAT|O_TRUNC, FILE_MODE)) < 0)
        err_sys("can't create %s", argv[2]);

    while ((n = read(ifd, buf, BSZ)) > 0) {
        for (i = 0; i < n; i++)
            buf[i] = translate(buf[i]);
        if ((nw = write(ofd, buf, n)) != n) {
            if (nw < 0)
                err_sys("write failed");
            else
                err_quit("short write (%d/%d)", nw, n);
        }
    }

    fsync(ofd);
    exit(0);
}
```

Figure 14.20 Translate a file using ROT-13

The I/O portion of the program is straightforward: we read a block from the input file, translate it, and then write the block to the output file. We repeat this until we hit the end of file and `read` returns zero. The program in Figure 14.21 shows how to perform the same task using the equivalent asynchronous I/O functions.

```
#include "apue.h"
#include <ctype.h>
#include <fcntl.h>
#include <aio.h>
#include <errno.h>

#define BSZ 4096
#define NBUF 8

enum rwop {
    UNUSED = 0,
    READ_PENDING = 1,
    WRITE_PENDING = 2
};

struct buf {
    enum rwop     op;
    int           last;
    struct aiocb  aiocb;
    unsigned char data[BSZ];
};

struct buf bufs[NBUF];
```

```
unsigned char
translate(unsigned char c)
{
    /* same as before */
}

int
main(int argc, char* argv[])
{
    int                 ifd, ofd, i, j, n, err, numop;
    struct stat         sbuf;
    const struct aiocb  *aiolist[NBUF];
    off_t               off = 0;

    if (argc != 3)
        err_quit("usage: rot13 infile outfile");
    if ((ifd = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s", argv[1]);
    if ((ofd = open(argv[2], O_RDWR|O_CREAT|O_TRUNC, FILE_MODE)) < 0)
        err_sys("can't create %s", argv[2]);
    if (fstat(ifd, &sbuf) < 0)
        err_sys("fstat failed");

    /* initialize the buffers */
    for (i = 0; i < NBUF; i++) {
        bufs[i].op = UNUSED;
        bufs[i].aiocb.aio_buf = bufs[i].data;
        bufs[i].aiocb.aio_sigevent.sigev_notify = SIGEV_NONE;
        aiolist[i] = NULL;
    }

    numop = 0;
    for (;;) {
        for (i = 0; i < NBUF; i++) {
            switch (bufs[i].op) {
            case UNUSED:
                /*
                 * Read from the input file if more data
                 * remains unread.
                 */
                if (off < sbuf.st_size) {
                    bufs[i].op = READ_PENDING;
                    bufs[i].aiocb.aio_fildes = ifd;
                    bufs[i].aiocb.aio_offset = off;
                    off += BSZ;
                    if (off >= sbuf.st_size)
                        bufs[i].last = 1;
                    bufs[i].aiocb.aio_nbytes = BSZ;
                    if (aio_read(&bufs[i].aiocb) < 0)
                        err_sys("aio_read failed");
                    aiolist[i] = &bufs[i].aiocb;
```

```
            numop++;
        }
        break;

    case READ_PENDING:
        if ((err = aio_error(&bufs[i].aiocb)) == EINPROGRESS)
            continue;
        if (err != 0) {
            if (err == -1)
                err_sys("aio_error failed");
            else
                err_exit(err, "read failed");
        }

        /*
         * A read is complete; translate the buffer
         * and write it.
         */
        if ((n = aio_return(&bufs[i].aiocb)) < 0)
            err_sys("aio_return failed");
        if (n != BSZ && !bufs[i].last)
            err_quit("short read (%d/%d)", n, BSZ);
        for (j = 0; j < n; j++)
            bufs[i].data[j] = translate(bufs[i].data[j]);
        bufs[i].op = WRITE_PENDING;
        bufs[i].aiocb.aio_fildes = ofd;
        bufs[i].aiocb.aio_nbytes = n;
        if (aio_write(&bufs[i].aiocb) < 0)
            err_sys("aio_write failed");
        /* retain our spot in aiolist */
        break;

    case WRITE_PENDING:
        if ((err = aio_error(&bufs[i].aiocb)) == EINPROGRESS)
            continue;
        if (err != 0) {
            if (err == -1)
                err_sys("aio_error failed");
            else
                err_exit(err, "write failed");
        }

        /*
         * A write is complete; mark the buffer as unused.
         */
        if ((n = aio_return(&bufs[i].aiocb)) < 0)
            err_sys("aio_return failed");
        if (n != bufs[i].aiocb.aio_nbytes)
            err_quit("short write (%d/%d)", n, BSZ);
        aiolist[i] = NULL;
        bufs[i].op = UNUSED;
```

```
                numop--;
                break;
            }
        }
        if (numop == 0) {
            if (off >= sbuf.st_size)
                break;
        } else {
            if (aio_suspend(aiolist, NBUF, NULL) < 0)
                err_sys("aio_suspend failed");
        }
    }

    bufs[0].aiocb.aio_fildes = ofd;
    if (aio_fsync(O_SYNC, &bufs[0].aiocb) < 0)
        err_sys("aio_fsync failed");
    exit(0);
}
```

Figure 14.21 Translate a file using ROT-13 and asynchronous I/O

Note that we use eight buffers, so we can have up to eight asynchronous I/O requests pending. Surprisingly, this might actually reduce performance—if the reads are presented to the file system out of order, it can defeat the operating system's read-ahead algorithm.

Before we can check the return value of an operation, we need to make sure the operation has completed. When `aio_error` returns a value other than `EINPROGRESS` or –1, we know the operation is complete. Excluding these values, if the return value is anything other than 0, then we know the operation failed. Once we've checked these conditions, it is safe to call `aio_return` to get the return value of the I/O operation.

As long as we have work to do, we can submit asynchronous I/O operations. When we have an unused AIO control block, we can submit an asynchronous read request. When a read completes, we translate the buffer contents and then submit an asynchronous write request. When all AIO control blocks are in use, we wait for an operation to complete by calling `aio_suspend`.

When we write a block to the output file, we retain the same offset at which we read the data from the input file. Consequently, the order of the writes doesn't matter. This strategy works only because each character in the input file has a corresponding character in the output file at the same offset; we neither add nor delete characters in the output file. (This insight might help solve Exercise 14.8.)

We don't use asynchronous notification in this example, because it is easier to use a synchronous programming model. If we had something else to do while the I/O operations were in progress, then the additional work could be folded into the `for` loop. If we needed to prevent this additional work from delaying the task of translating the file, however, then we might have to structure the code to use some form of asynchronous notification. With multiple tasks, we need to prioritize the tasks before deciding how the program should be structured. □

14.6 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 fd, const struct iovec *iov, int iovcnt);

ssize_t writev(int fd, 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 2.11). Figure 14.22 shows a diagram relating the arguments to these two functions and the iovec structure.

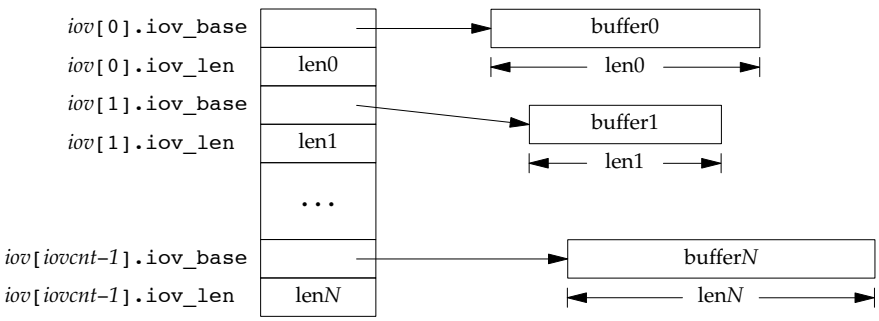


Figure 14.22 The iovec structure for readv and writev

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 option of the Single UNIX Specification.

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 14.23 shows the results from the three methods just described.

Operation	Linux (Intel x86)			Mac OS X (Intel x86)		
	User	System	Clock	User	System	Clock
two `writes`	0.06	2.04	2.13	0.85	8.33	13.83
buffer copy, then one `write`	0.03	1.13	1.16	0.70	4.87	9.25
one `writev`	0.04	1.21	1.26	0.43	5.34	9.24

Figure 14.23 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 14.23. We used `times` (Section 8.17) 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 3.6.

Next, note that the sum of the CPU times (user plus system) is slightly 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.

> Don't infer too much about the relative performance of Linux and Mac OS X from the numbers shown in Figure 14.23. The two computers were very different: they had different processor generations, 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.

14.7 `readn` and `writen` Functions

Pipes, FIFOs, and some devices—notably terminals and networks—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 return less than we specified. This may be caused by kernel output buffers becoming full, 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 `readn` and `writen` functions to read and write *N* bytes of data, respectively, letting these functions handle a return value that's possibly 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 fd, void *buf, size_t nbytes);

ssize_t writen(int fd, 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 14.24 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 the 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 14.24 The `readn` and `writen` functions

14.8 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`. Version 4 of the Single UNIX Specification moved the `mmap` function from an option to the base specification. All POSIX-conforming systems are required to support it.

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

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

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

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

The *addr* argument lets us specify the address where we want the mapped region to start. We normally set this value 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 *fd* 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 14.25 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 14.26 shows a memory-mapped file. (Recall the memory layout of a typical process, shown in Figure 7.6.) In this figure, "start addr" is the return value from `mmap`. We have shown

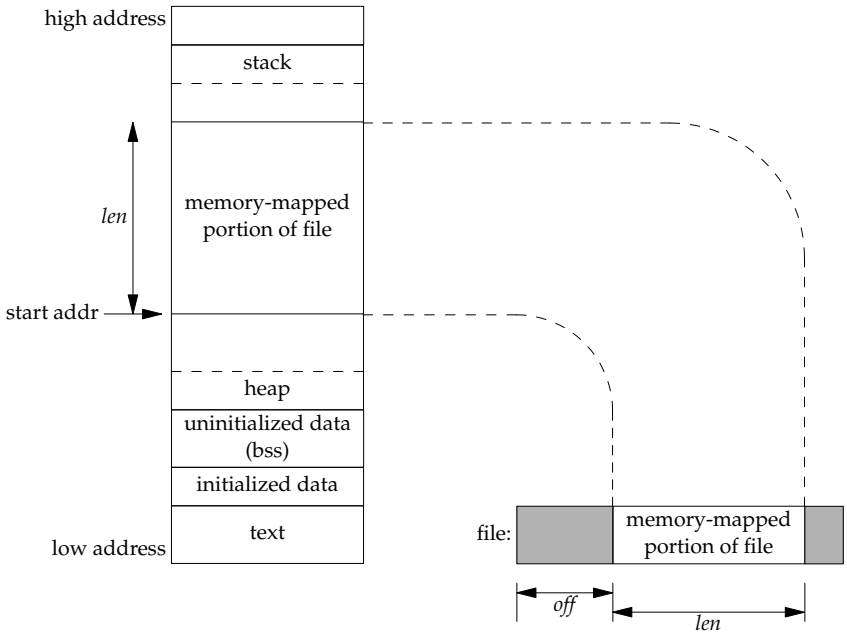


Figure 14.26 Example of a memory-mapped file

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 usually 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.

> This requirement is usually imposed by the system implementations. Although the Single UNIX Specification no longer requires that this condition be satisfied, all the platforms covered in this book, except FreeBSD 8.0, have this requirement. FreeBSD 8.0 allows us to use any address alignment and offset alignment as long as the alignments match.

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 14.27.

Two signals are normally used with mapped regions. `SIGSEGV` is 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 14.25). Be aware that implementations may require the address argument to be an integral multiple of the system's page size.

When we modify pages that we've mapped into our address space using the `MAP_SHARED` flag, the changes aren't written back to the file immediately. Instead, the

kernel daemons decide when dirty pages are written back based on (a) system load and (b) configuration parameters meant to limit data loss in the event of a system failure. When the changes are written back, they are written in units of pages. Thus, if we modify only one byte in a page, when the change is written back to the file, the entire page will be written.

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.

> The `msync` function is included in the XSI option in the Single UNIX Specification. As such, all UNIX systems must support it.

A memory-mapped region is automatically unmapped when the process terminates or we can unmap a region directly by calling the `munmap` function. Closing the file descriptor used when we mapped the region does not unmap the region.

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

int munmap(void *addr, size_t len);
```

Returns: 0 if OK, –1 on error

The `munmap` function 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 sometime after 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 14.27 copies a file (similar to the `cp`(1) command) using memory-mapped I/O.

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

#define COPYINCR (1024*1024*1024)   /* 1 GB */

int
main(int argc, char *argv[])
{
    int         fdin, fdout;
    void        *src, *dst;
    size_t      copysz;
    struct stat sbuf;
    off_t       fsz = 0;

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

    if (ftruncate(fdout, sbuf.st_size) < 0) /* set output file size */
        err_sys("ftruncate error");

    while (fsz < sbuf.st_size) {
        if ((sbuf.st_size - fsz) > COPYINCR)
            copysz = COPYINCR;
        else
            copysz = sbuf.st_size - fsz;

        if ((src = mmap(0, copysz, PROT_READ, MAP_SHARED,
          fdin, fsz)) == MAP_FAILED)
            err_sys("mmap error for input");
        if ((dst = mmap(0, copysz, PROT_READ | PROT_WRITE,
          MAP_SHARED, fdout, fsz)) == MAP_FAILED)
            err_sys("mmap error for output");

        memcpy(dst, src, copysz);   /* does the file copy */
        munmap(src, copysz);
        munmap(dst, copysz);
        fsz += copysz;
    }
    exit(0);
}
```

Figure 14.27 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 `ftruncate` 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 successful, but the first reference to the associated memory region generates a `SIGBUS` signal.

We then call `mmap` for each file, to map the file into memory, and finally call `memcpy` to copy data from the input buffer to the output buffer. We copy at most 1 GB of data at a time to limit the amount of memory we use (it might not be possible to map the entire contents of a very large file if the system doesn't have enough memory). Before mapping the next sections of the files, we unmap the previous sections.

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 depends 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 14.28 shows the results. The times are given in seconds and the size of the file copied was 300 MB. Note that we don't synch the data to disk before exiting.

Operation	Linux 3.2.0 (Intel x86)			Solaris 10 (SPARC)		
	User	System	Clock	User	System	Clock
`read/write`	0.01	0.54	5.67	0.29	10.60	43.67
`mmap/memcpy`	0.08	0.65	22.54	1.89	8.56	38.42

Figure 14.28 Timing results comparing `read/write` versus `mmap/memcpy`

For both Linux 3.2.0 and Solaris 10, the total CPU time (user + system) is almost the same for both approaches. On Solaris, copying using `mmap` and `memcpy` takes more user time but less system time than copying using `read` and `write`. On Linux, the results are similar for the user time, but the system time for using `read` and `write` is slightly better than using `mmap` and `memcpy`. The two versions do the same work, but they go about it differently.

The major difference is that with `read` and `write`, we execute a lot more system calls and do more copying than 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. This copying occurs as a result of page fault handling when we reference memory pages that don't yet exist (there is one fault per page read and one fault per page written). If the overhead for the

system call and extra copying differs from the page fault overhead, then one approach will perform better than the other.

On Linux 3.2.0, as far as elapsed time is concerned, the two versions of the program show a large difference in clock time: the version using `read` and `write` completes four times faster than the version using `mmap` and `memcpy`. However, on Solaris 10, the version with `mmap` and `memcpy` is faster than the version with `read` and `write`. If the CPU times are almost the same, then why would the clock times differ? One possibility is that we might have to wait longer for I/O to complete in one version. This wait time is not counted as CPU processing time. Another possibility is that some system processing might not be counted against our program—the processing done by system daemons to write pages to disk, for example. As we need to allocate pages for reading and writing, these system daemons will help make pages available. If the page writes are random instead of sequential, then it will take longer to write them out to disk, so we will need to wait longer before the pages become available for us to reuse. □

Depending on the system, memory-mapped I/O can be more efficient when copying one regular file to another. There are limitations. We can't use this technique 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 is the manipulation of a frame buffer device that references a bitmapped 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.

14.9 Summary

In this chapter, we've described numerous advanced I/O functions, many 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)
- I/O multiplexing—the `select` and `poll` functions (we'll use these in many of the later examples)
- Asynchronous I/O
- 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 while 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 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.

14.6 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.

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

14.8 Rewrite the program in Figure 14.21 to make it a filter: read from the standard input and write to the standard output, but use the asynchronous I/O interfaces. What must you change to make it work properly? Keep in mind that you should get the same results whether the standard output is attached to a terminal, a pipe, or a regular file.

14.9 Recall Figure 14.23. 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 14.27 to copy a file and determine whether the last-access time for the input file is updated.

14.11 In the program from Figure 14.27, `close` the input file after calling `mmap` to verify that closing the descriptor does not invalidate the memory-mapped I/O.