55

FILE LOCKING

Previous chapters have covered various techniques that processes can use to synchronize their actions, including signals (Chapters 20 to 22) and semaphores (Chapters 47 and 53). In this chapter, we look at further synchronization techniques designed specifically for use with files.

55.1 Overview

A frequent application requirement is to read data from a file, make some change to that data, and then write it back to the file. As long as just one process at a time ever uses a file in this way, then there are no problems. However, problems can arise if multiple processes are simultaneously updating a file. Suppose, for example, that each process performs the following steps to update a file containing a sequence number:

1. Read the sequence number from the file.
2. Use the sequence number for some application-defined purpose.
3. Increment the sequence number and write it back to the file.

The problem here is that, in the absence of any synchronization technique, two processes could perform the above steps at the same time with (for example) the consequences shown in Figure 55-1 (here, we assume that the initial value of the sequence number is 1000).

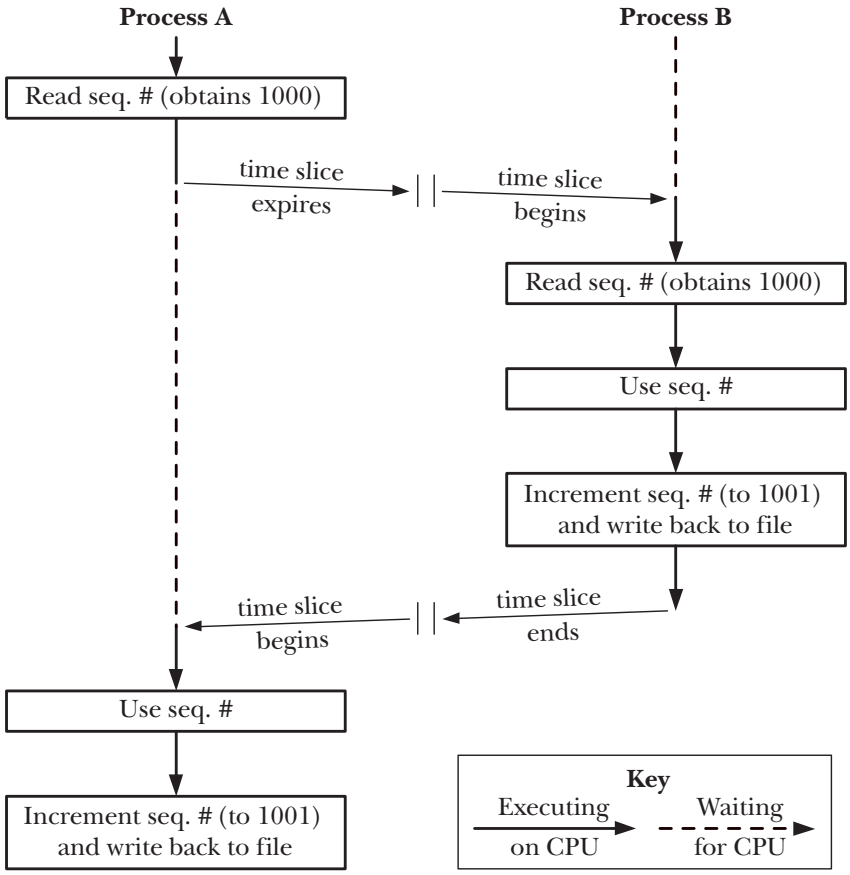

Figure 55-1: Two processes updating a file at the same time without synchronization

The problem is clear: at the end of these steps, the file contains the value 1001, when it should contain the value 1002. (This is an example of a race condition.) To prevent such possibilities, we need some form of interprocess synchronization.

Although we could use (say) semaphores to perform the required synchronization, using file locks is usually preferable, because the kernel automatically associates locks with files.

> [Stevens & Rago, 2005] dates the first UNIX file locking implementation to 1980, and notes that *fcntl()* locking, upon which we primarily focus in this chapter, appeared in System V Release 2 in 1984.

In this chapter, we describe two different APIs for placing file locks:

- *flock()*, which places locks on entire files; and
- *fcntl()*, which places locks on regions of a file.

The *flock()* system call originated on BSD; *fcntl()* originated on System V.

The general method of using *flock()* and *fcntl()* is as follows:

1. Place a lock on the file.
2. Perform file I/O.
3. Unlock the file so that another process can lock it.

Although file locking is normally used in conjunction with file I/O, we can also use it as a more general synchronization technique. Cooperating processes can follow a convention that locking all or part of a file indicates access by a process to some shared resource other than the file itself (e.g., a shared memory region).

Mixing locking and *stdio* functions

Because of the user-space buffering performed by the *stdio* library, we should be cautious when using *stdio* functions with the locking techniques described in this chapter. The problem is that an input buffer might be filled before a lock is placed, or an output buffer may be flushed after a lock is removed. There are a few ways to avoid these problems:

- Perform file I/O using *read()* and *write()* (and related system calls) instead of the *stdio* library.
- Flush the *stdio* stream immediately after placing a lock on the file, and flush it once more immediately before releasing the lock.
- Perhaps at the cost of some efficiency, disable *stdio* buffering altogether using *setbuf()* (or similar).

Advisory and mandatory locking

In the remainder of this chapter, we'll distinguish locks as being either advisory or mandatory. By default, file locks are *advisory*. This means that a process can simply ignore a lock placed by another process. In order for an advisory locking scheme to be workable, each process accessing the file must cooperate, by placing a lock before performing file I/O. By contrast, a *mandatory* locking system forces a process performing I/O to abide by the locks held by other processes. We say more about this distinction in Section 55.4.

55.2 File Locking with *flock()*

Although *fcntl()* provides a superset of the functionality provided by *flock()*, we nevertheless describe *flock()* because it is still used in some applications, and because it differs from *fcntl()* in some of the semantics of inheritance and release of locks.

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

int flock(int fd, int operation);
```

Returns 0 on success, or −1 on error

The *flock()* system call places a single lock on an entire file. The file to be locked is specified via an open file descriptor passed in *fd*. The *operation* argument specifies one of the values LOCK_SH, LOCK_EX, or LOCK_UN, which are described in Table 55-1.

By default, *flock()* blocks if another process already holds an incompatible lock on a file. If we want to prevent this, we can OR (|) the value into *operation*. In this case, if another process already holds an incompatible lock on the file, *flock()* doesn't block, but instead returns −1, with *errno* set to EWOULDBLOCK.

Table 55-1: Values for the *operation* argument of *flock()*

Value	Description
LOCK_SH	Place a *shared* lock on the file referred to by *fd*
LOCK_EX	Place an *exclusive* lock on the file referred to by *fd*
LOCK_UN	Unlock the file referred to by *fd*
LOCK_NB	Make a nonblocking lock request

Any number of processes may simultaneously hold a shared lock on a file. However, only one process at a time can hold an exclusive lock on a file. (In other words, exclusive locks deny both exclusive and shared locks by other processes.) Table 55-2 summarizes the compatibility rules for *flock()* locks. Here, we assume that process A is the first to place the lock, and the table indicates whether process B can then place a lock.

Table 55-2: Compatibility of *flock()* locking types

Process A	Process B	
	LOCK_SH	LOCK_EX
LOCK_SH	Yes	No
LOCK_EX	No	No

A process can place a shared or exclusive lock regardless of the access mode (read, write, or read-write) of the file.

An existing shared lock can be converted to an exclusive lock (and vice versa) by making another call to *flock()* specifying the appropriate value for *operation*. Converting a shared lock to an exclusive lock will block if another process holds a shared lock on the file, unless LOCK_NB was also specified.

A lock conversion is *not* guaranteed to be atomic. During conversion, the existing lock is first removed, and then a new lock is established. Between these two steps, another process's pending request for an incompatible lock may be granted. If this occurs, then the conversion will block, or, if LOCK_NB was specified, the conversion will fail and the process will lose its original lock. (This behavior occurred in the original BSD *flock()* implementation and also occurs on many other UNIX implementations.)

> Although it is not part of SUSv3, *flock()* appears on most UNIX implementations. Some implementations require the inclusion of either <fcntl.h> or <sys/fcntl.h> instead of <sys/file.h>. Because *flock()* originates on BSD, the locks that it places are sometimes known as *BSD file locks*.

Listing 55-1 demonstrates the use of *flock()*. This program locks a file, sleeps for a specified number of seconds, and then unlocks the file. The program takes up to three command-line arguments. The first of these is the file to lock. The second specifies the lock type (shared or exclusive) and whether or not to include the LOCK_NB (nonblocking) flag. The third argument specifies the number of seconds to sleep between acquiring and releasing the lock; this argument is optional and defaults to 10 seconds.

Listing 55-1: Using *flock()*

filelock/t_flock.c

```
#include <sys/file.h>
#include <fcntl.h>
#include "curr_time.h"                  /* Declaration of currTime() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int fd, lock;
    const char *lname;

    if (argc < 3 || strcmp(argv[1], "--help") == 0 ||
            strchr("sx", argv[2][0]) == NULL)
        usageErr("%s file lock [sleep-time]\n"
                 "    'lock' is 's' (shared) or 'x' (exclusive)\n"
                 "        optionally followed by 'n' (nonblocking)\n"
                 "    'secs' specifies time to hold lock\n", argv[0]);

    lock = (argv[2][0] == 's') ? LOCK_SH : LOCK_EX;
    if (argv[2][1] == 'n')
        lock |= LOCK_NB;

    fd = open(argv[1], O_RDONLY);               /* Open file to be locked */
    if (fd == -1)
        errExit("open");

    lname = (lock & LOCK_SH) ? "LOCK_SH" : "LOCK_EX";

    printf("PID %ld: requesting %s at %s\n", (long) getpid(), lname,
            currTime("%T"));

    if (flock(fd, lock) == -1) {
        if (errno == EWOULDBLOCK)
            fatal("PID %ld: already locked - bye!", (long) getpid());
        else
            errExit("flock (PID=%ld)", (long) getpid());
    }

    printf("PID %ld: granted    %s at %s\n", (long) getpid(), lname,
            currTime("%T"));

    sleep((argc > 3) ? getInt(argv[3], GN_NONNEG, "sleep-time") : 10);

    printf("PID %ld: releasing  %s at %s\n", (long) getpid(), lname,
            currTime("%T"));
    if (flock(fd, LOCK_UN) == -1)
        errExit("flock");

    exit(EXIT_SUCCESS);
}
```

filelock/t_flock.c

Using the program in Listing 55-1, we can conduct a number of experiments to explore the behavior of *flock()*. Some examples are shown in the following shell session. We begin by creating a file, and then start an instance of our program that sits in the background and holds a shared lock for 60 seconds:

```
$ touch tfile
$ ./t_flock tfile s 60 &
[1] 9777
PID 9777: requesting LOCK_SH at 21:19:37
PID 9777: granted    LOCK_SH at 21:19:37
```

Next, we start another instance of the program that successfully requests a shared lock and then releases it:

```
$ ./t_flock tfile s 2
PID 9778: requesting LOCK_SH at 21:19:49
PID 9778: granted    LOCK_SH at 21:19:49
PID 9778: releasing  LOCK_SH at 21:19:51
```

However, when we start another instance of the program that makes a nonblocking requests for an exclusive lock, the request immediately fails:

```
$ ./t_flock tfile xn
PID 9779: requesting LOCK_EX at 21:20:03
PID 9779: already locked - bye!
```

When we start another instance of the program that makes a blocking request for an exclusive lock, the program blocks. When the background process that was holding a shared lock for 60 seconds releases its lock, the blocked request is granted:

```
$ ./t_flock tfile x
PID 9780: requesting LOCK_EX at 21:20:21
PID 9777: releasing  LOCK_SH at 21:20:37
PID 9780: granted    LOCK_EX at 21:20:37
PID 9780: releasing  LOCK_EX at 21:20:47
```

55.2.1 Semantics of Lock Inheritance and Release

As shown in Table 55-1, we can release a file lock via an *flock()* call that specifies *operation* as `LOCK_UN`. In addition, locks are automatically released when the corresponding file descriptor is closed. However, the story is more complicated than this. A file lock obtained via *flock()* is associated with the open file description (Section 5.4), rather than the file descriptor or the file (i-node) itself. This means that when a file descriptor is duplicated (via *dup()*, *dup2()*, or an *fcntl()* `F_DUPFD` operation), the new file descriptor refers to the same file lock. For example, if we have obtained a lock on the file referred to by *fd*, then the following code (which omits error checking) releases that lock:

```
flock(fd, LOCK_EX);             /* Gain lock via 'fd' */
newfd = dup(fd);                /* 'newfd' refers to same lock as 'fd' */
flock(newfd, LOCK_UN);          /* Frees lock acquired via 'fd' */
```

If we have acquired a lock via a particular file descriptor, and we create one or more duplicates of that descriptor, then—if we don't explicitly perform an unlock operation—the lock is released only when all of the duplicate descriptors have been closed.

However, if we use *open()* to obtain a second file descriptor (and associated open file description) referring to the same file, this second descriptor is treated independently by *flock()*. For example, a process executing the following code will block on the second *flock()* call:

```
fd1 = open("a.txt", O_RDWR);
fd2 = open("a.txt", O_RDWR);
flock(fd1, LOCK_EX);
flock(fd2, LOCK_EX);              /* Locked out by lock on 'fd1' */
```

Thus, a process can lock itself out of a file using *flock()*. As we'll see later, this can't happen with record locks obtained by *fcntl()*.

When we create a child process using *fork()*, that child obtains duplicates of its parent's file descriptors, and, as with descriptors duplicated via *dup()* and so on, these descriptors refer to the same open file descriptions and thus to the same locks. For example, the following code causes a child to remove a parent's lock:

```
flock(fd, LOCK_EX);               /* Parent obtains lock */
if (fork() == 0)                  /* If child... */
    flock(fd, LOCK_UN);           /* Release lock shared with parent */
```

These semantics can sometimes be usefully exploited to (atomically) transfer a file lock from a parent process to a child process: after the *fork()*, the parent closes its file descriptor, and the lock is under sole control of the child process. As we'll see later, this isn't possible using record locks obtained by *fcntl()*.

Locks created by *flock()* are preserved across an *exec()* (unless the close-on-exec flag was set for the file descriptor and that file descriptor was the last one referencing the underlying open file description).

The Linux semantics described above conform to the classical BSD implementation of *flock()*. On some UNIX implementations, *flock()* is implemented using *fcntl()*, and we'll see later that the inheritance and release semantics of *fcntl()* locks differ from those of *flock()* locks. Because the interactions between locks created by *flock()* and *fcntl()* are undefined, an application should use only one of these locking methods on a file.

55.2.2 Limitations of *flock()*

Placing locks with *flock()* suffers from a number of limitations:

- Only whole files can be locked. Such coarse locking limits the potential for concurrency among cooperating processes. If, for example, we have multiple processes, each of which would like to simultaneously access different parts of the same file, then locking via *flock()* would needlessly prevent these processes from operating concurrently.
- We can place only advisory locks with *flock()*.
- Many NFS implementations don't recognize locks granted by *flock()*.

All of these limitations are addressed by the locking scheme implemented by *fcntl()*, which we describe in the next section.

Historically, the Linux NFS server did not support *flock()* locks. Since kernel 2.6.12, the Linux NFS server supports *flock()* locks by implementing them as an *fcntl()* lock on the entire file. This can cause some strange effects when mixing BSD locks on the server and BSD locks on the client: the clients usually won't see the server's locks, and vice versa.

55.3 Record Locking with *fcntl()*

Using *fcntl()* (Section 5.2), we can place a lock on any part of a file, ranging from a single byte to the entire file. This form of file locking is usually called *record locking*. However, this term is a misnomer, because files on the UNIX system are byte sequences, with no concept of record boundaries. Any notion of records within a file is defined purely within an application.

Typically, *fcntl()* is used to lock byte ranges corresponding to the application-defined record boundaries within the file; hence the origin of the term *record locking*. The terms *byte range*, *file region*, and *file segment* are less commonly used, but more accurate, descriptions of this type of lock. (Because this is the only kind of locking specified in the original POSIX.1 standard and in SUSv3, it is sometimes also called POSIX file locking.)

SUSv3 requires record locking to be supported for regular files, and permits it to be supported for other file types. Although it generally makes sense to apply record locks only to regular files (since, for most other file types, it isn't meaningful to talk about byte ranges for the data contained in the file), on Linux, it is possible to apply a record lock to any type of file descriptor.

Figure 55-2 shows how record locking might be used to synchronize access by two processes to the same region of a file. (In this diagram, we assume that all lock requests are blocking, so that they will wait if a lock is held by another process.)

The general form of the *fcntl()* call used to create or remove a file lock is as follows:

```
struct flock flockstr;

/* Set fields of 'flockstr' to describe lock to be placed or removed */

fcntl(fd, cmd, &flockstr);          /* Place lock defined by 'fl' */
```

The *fd* argument is an open file descriptor referring to the file on which we wish to place a lock.

Before discussing the *cmd* argument, we first describe the *flock* structure.

The *flock* structure

The *flock* structure defines the lock that we wish to acquire or remove. It is defined as follows:

```
struct flock {
    short l_type;       /* Lock type: F_RDLCK, F_WRLCK, F_UNLCK */
    short l_whence;     /* How to interpret 'l_start': SEEK_SET,
                           SEEK_CUR, SEEK_END */
```

```
    off_t l_start;      /* Offset where the lock begins */
    off_t l_len;        /* Number of bytes to lock; 0 means "until EOF" */
    pid_t l_pid;        /* Process preventing our lock (F_GETLK only) */
};
```

The *l_type* field indicates the type of lock we want to place. It is specified as one of the values in Table 55-3.

Semantically, read (`F_RDLCK`) and write (`F_WRLCK`) locks correspond to the shared and exclusive locks applied by *flock()*, and they follow the same compatibility rules (Table 55-2): any number of processes can hold read locks on a file region, but only one process can hold a write lock, and that lock excludes read and write locks by other processes. Specifying *l_type* as `F_UNLCK` is analogous to the *flock()* `LOCK_UN` operation.

Table 55-3: Lock types for *fcntl()* locking

Lock type	Description
`F_RDLCK`	Place a read lock
`F_WRLCK`	Place a write lock
`F_UNLCK`	Remove an existing lock

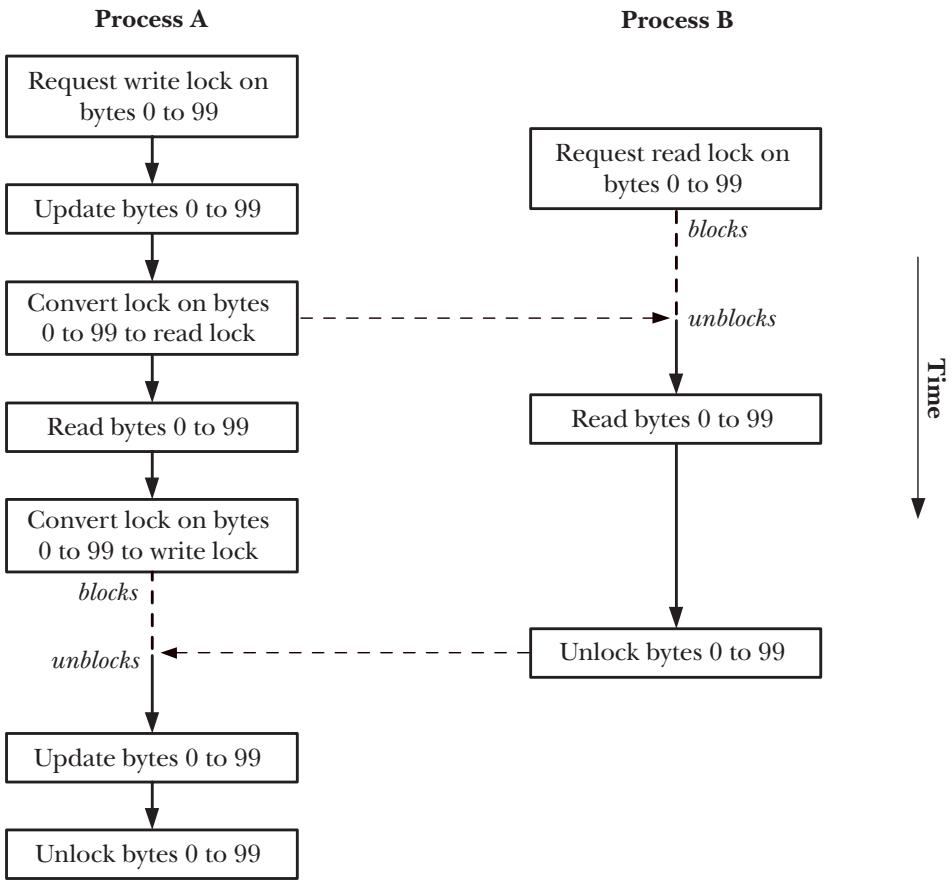

Figure 55-2: Using record locks to synchronize access to the same region of a file

In order to place a read lock on a file, the file must be open for reading. Similarly, to place a write lock, the file must be open for writing. To place both types of locks, we open the file read-write (O_RDWR). Attempting to place a lock that is incompatible with the file access mode results in the error EBADF.

The *l_whence*, *l_start*, and *l_len* fields together specify the range of bytes to be locked. The first two of these fields are analogous to the *whence* and *offset* arguments to *lseek()* (Section 4.7). The *l_start* field specifies an offset within the file that is interpreted with respect to one of the following:

- the start of the file, if *l_whence* is SEEK_SET;
- the current file offset, if *l_whence* is SEEK_CUR; or
- the end of the file, if *l_whence* is SEEK_END.

In the last two cases, *l_start* can be a negative number, as long as the resulting file position doesn't lie before the start of the file (byte 0).

The *l_len* field contains an integer specifying the number of bytes to lock, starting from the position defined by *l_whence* and *l_start*. It is possible to lock nonexistent bytes past the end of the file, but it is not possible to lock bytes before the start of the file.

Since kernel 2.4.21, Linux allows a negative value to be supplied in *l_len*. This is a request to lock the *l_len* bytes preceding the position specified by *l_whence* and *l_start* (i.e., bytes in the range *(l_start – abs(l_len))* through to *(l_start – 1)*). SUSv3 permits, but doesn't require, this feature. Several other UNIX implementations also provide it.

In general, applications should lock the minimum range of bytes necessary. This allows greater concurrency for other processes simultaneously trying to lock different regions of the same file.

> The term *minimum range* needs qualification in some circumstances. Mixing record locks and calls to *mmap()* can have unpleasant consequences on network file systems such as NFS and CIFS. The problem occurs because *mmap()* maps files in units of the system page size. If a file lock is page-aligned, then all is well, since the lock will cover the entire region corresponding to a dirty page. However, if the lock is not page-aligned, then there is a race condition—the kernel may write into the area that is not covered by the lock if any part of the mapped page has been modified.

Specifying 0 in *l_len* has the special meaning "lock all bytes from the point specified by *l_start* and *l_whence* through to the end of the file, no matter how large the file grows." This is convenient if we don't know in advance how many bytes we are going to add to a file. To lock the entire file, we can specify *l_whence* as SEEK_SET and both *l_start* and *l_len* as 0.

The *cmd* argument

When working with file locks, three possible values may be specified for the *cmd* argument of *fcntl()*. The first two are used for acquiring and releasing locks:

F_SETLK
: Acquire (*l_type* is F_RDLCK or F_WRLCK) or release (*l_type* is F_UNLCK) a lock on the bytes specified by *flockstr*. If an incompatible lock is held by another

process on any part of the region to be locked, *fcntl()* fails with the error EAGAIN. On some UNIX implementations, *fcntl()* fails with the error EACCES in this case. SUSv3 permits either possibility, and a portable application should test for both values.

F_SETLKW
: This is the same as F_SETLK, except that if another process holds an incompatible lock on any part of the region to be locked, then the call blocks until the lock can be granted. If we are handling signals and have not specified SA_RESTART (Section 21.5), then an F_SETLKW operation may be interrupted (i.e., fail with the error EINTR). We can take advantage of this behavior to use *alarm()* or *setitimer()* to set a timeout on the lock request.

Note that *fcntl()* locks either the entire region specified or nothing at all. There is no notion of locking just those bytes of the requested region that are currently unlocked.

The remaining *fcntl()* operation is used to determine whether we can place a lock on a given region:

F_GETLK
: Check if it would be possible to acquire the lock specified in *flockstr*, but don't actually acquire it. The *l_type* field must be F_RDLCK or F_WRLCK. The *flockstr* structure is treated as a value-result argument; on return, it contains information informing us whether or not the specified lock could be placed. If the lock would be permitted (i.e., no incompatible locks exist on the specified file region), then F_UNLCK is returned in the *l_type* field, and the remaining fields are left unchanged. If one or more incompatible locks exist on the region, then *flockstr* returns information about *one* of those locks (it is indeterminate which), including its type (*l_type*), range of bytes (*l_start* and *l_len*; *l_whence* is always returned as SEEK_SET), and the process ID of the process holding the lock (*l_pid*).

Note that there are potential race conditions when combining the use of F_GETLK with a subsequent F_SETLK or F_SETLKW. By the time we perform the latter operation, the information returned by F_GETLK may already be out of date. Thus, F_GETLK is less useful than it first appears. Even if F_GETLK says that it is possible to place a lock, we must still be prepared for an error return from F_SETLK or for F_SETLKW to block.

> The GNU C library also implements the function *lockf()*, which is just a simplified interface layered on top of *fcntl()*. (SUSv3 specifies *lockf()*, but doesn't specify the relationship between *lockf()* and *fcntl()*. However, most UNIX systems implement *lockf()* on top of *fcntl()*.) A call of the form *lockf(fd, operation, size)* is equivalent to a call to *fcntl()* with *l_whence* set to SEEK_CUR, *l_start* set to 0, and *l_len* set to *size*; that is, *lockf()* locks a sequence of bytes starting at the current file offset. The *operation* argument to *lockf()* is analogous to the *cmd* argument to *fcntl()*, but different constants are used for acquiring, releasing, and testing for the presence of locks. The *lockf()* function places only exclusive (i.e., write) locks. See the *lockf(3)* manual page for further details.

Details of lock acquisition and release

Note the following points regarding the acquisition and release of locks created with *fcntl()*:

- Unlocking a file region always immediately succeeds. It is not an error to unlock a region on which we don't currently hold a lock.
- At any time, a process can hold just one type of lock on a particular region of a file. Placing a new lock on a region we have already locked either results in no change (if the lock type is the same as the existing lock) or atomically converts the existing lock to the new mode. In the latter case, when converting a read lock to a write lock, we need to be prepared for the possibility that the call will yield an error (`F_SETLK`) or block (`F_SETLKW`). (This differs from *flock()*, whose lock conversions are not atomic.)
- A process can never lock itself out of a file region, even when placing locks via multiple file descriptors referring to the same file. (This contrasts with *flock()*, and we say more on this point in Section 55.3.5.)
- Placing a lock of a different mode in the middle of a lock we already hold results in three locks: two smaller locks in the previous mode are created on either side of the new lock (see Figure 55-3). Conversely, acquiring a second lock adjacent to or overlapping an existing lock in the same mode results in a single coalesced lock covering the combined area of the two locks. Other permutations are possible. For example, unlocking a region in the middle of a larger existing lock leaves two smaller locked regions on either side of the unlocked region. If a new lock overlaps an existing lock with a different mode, then the existing lock is shrunk, because the overlapping bytes are incorporated into the new lock.
- Closing a file descriptor has some unusual semantics with respect to file region locks. We describe these semantics in Section 55.3.5.

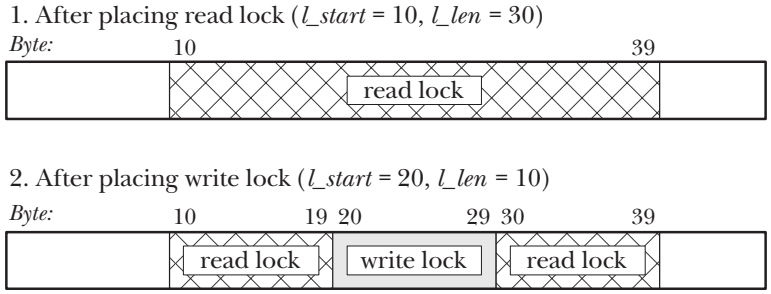

Figure 55-3: Splitting of an existing read lock by a write lock by the same process

55.3.1 Deadlock

When using `F_SETLKW`, we need to be aware of the type of scenario illustrated in Figure 55-4. In this scenario, each process's second lock request is blocked by a lock held by the other process. Such a scenario is referred to as a *deadlock*. If unchecked by the kernel, this would leave both processes blocked forever. To prevent this possibility, the kernel checks each new lock request made via `F_SETLKW` to see if it would result in a deadlock situation. If it would, then the kernel selects one

of the blocked processes and causes its *fcntl()* call to unblock and fail with the error EDEADLK. (On Linux, the process making the most recent *fcntl()* call is selected, but this is not required by SUSv3, and may not hold true on future versions of Linux or on other UNIX implementations. Any process using F_SETLKW must be prepared to handle an EDEADLK error.)

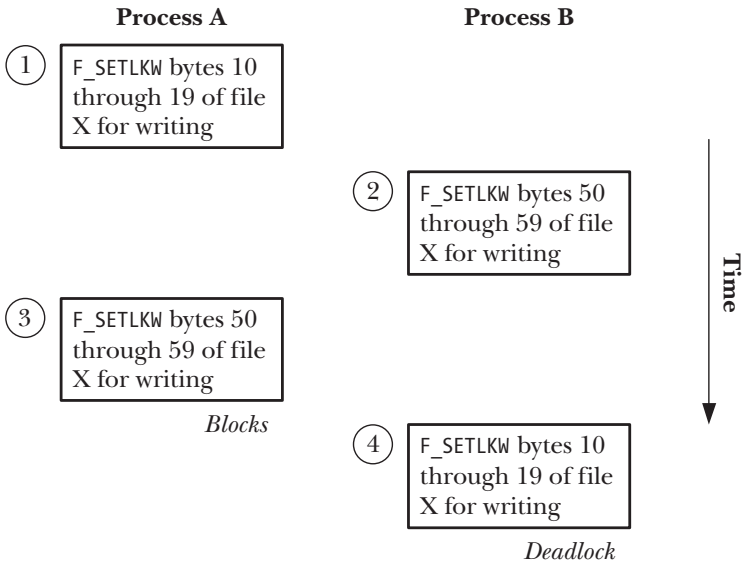

Figure 55-4: Deadlock when two processes deny each other's lock requests

Deadlock situations are detected even when placing locks on multiple different files, as are circular deadlocks involving multiple processes. (By *circular deadlock*, we mean, for example, process A waiting to acquire a lock on a region locked by process B, process B waiting on a lock held by process C, and process C waiting on a lock held by process A.)

55.3.2 Example: An Interactive Locking Program

The program shown in Listing 55-2 allows us to interactively experiment with record locking. This program takes a single command-line argument: the name of a file on which we wish to place locks. Using this program, we can verify many of our previous statements regarding the operation of record locking. The program is designed to be used interactively and accepts commands of this form:

---

cmd lock start length [*whence*]

For *cmd*, we can specify *g* to perform an F_GETLK, *s* to perform an F_SETLK, or *w* to perform an F_SETLKW. The remaining arguments are used to initialize the *flock* structure passed to *fcntl()*. The *lock* argument specifies the value for the *l_type* field and is *r* for F_RDLCK, *w* for F_WRLCK, or *u* for F_UNLCK. The *start* and *length* arguments are integers specifying the values for the *l_start* and *l_len* fields. Finally, the optional *whence* argument specifies the value for the *l_whence* field, and may be *s* for SEEK_SET (the default), *c* for SEEK_CUR, or *e* for SEEK_END. (For an explanation of why we cast the *l_start* and *l_len* fields to *long long* in the *printf()* call in Listing 55-2, see Section 5.10.)

Listing 55-2: Experimenting with record locking

filelock/i_fcntl_locking.c

```
#include <sys/stat.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

#define MAX_LINE 100

static void
displayCmdFmt(void)
{
    printf("\n    Format: cmd lock start length [whence]\n\n");
    printf("    'cmd' is 'g' (GETLK), 's' (SETLK), or 'w' (SETLKW)\n");
    printf("    'lock' is 'r' (READ), 'w' (WRITE), or 'u' (UNLOCK)\n");
    printf("    'start' and 'length' specify byte range to lock\n");
    printf("    'whence' is 's' (SEEK_SET, default), 'c' (SEEK_CUR), "
           "or 'e' (SEEK_END)\n\n");
}

int
main(int argc, char *argv[])
{
    int fd, numRead, cmd, status;
    char lock, cmdCh, whence, line[MAX_LINE];
    struct flock fl;
    long long len, st;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s file\n", argv[0]);

    fd = open(argv[1], O_RDWR);
    if (fd == -1)
        errExit("open (%s)", argv[1]);

    printf("Enter ? for help\n");

    for (;;) {          /* Prompt for locking command and carry it out */
        printf("PID=%ld> ", (long) getpid());
        fflush(stdout);

        if (fgets(line, MAX_LINE, stdin) == NULL)       /* EOF */
            exit(EXIT_SUCCESS);
        line[strlen(line) - 1] = '\0';          /* Remove trailing '\n' */

        if (*line == '\0')
            continue;                           /* Skip blank lines */

        if (line[0] == '?') {
            displayCmdFmt();
            continue;
        }

        whence = 's';                   /* In case not otherwise filled in */
```

```
        numRead = sscanf(line, "%c %c %lld %lld %c", &cmdCh, &lock,
                        &st, &len, &whence);
        fl.l_start = st;
        fl.l_len = len;

        if (numRead < 4 || strchr("gsw", cmdCh) == NULL ||
                strchr("rwu", lock) == NULL || strchr("sce", whence) == NULL) {
            printf("Invalid command!\n");
            continue;
        }

        cmd = (cmdCh == 'g') ? F_GETLK : (cmdCh == 's') ? F_SETLK : F_SETLKW;
        fl.l_type = (lock == 'r') ? F_RDLCK : (lock == 'w') ? F_WRLCK : F_UNLCK;
        fl.l_whence = (whence == 'c') ? SEEK_CUR :
                      (whence == 'e') ? SEEK_END : SEEK_SET;

        status = fcntl(fd, cmd, &fl);           /* Perform request... */

        if (cmd == F_GETLK) {                   /* ... and see what happened */
            if (status == -1) {
                errMsg("fcntl - F_GETLK");
            } else {
                if (fl.l_type == F_UNLCK)
                    printf("[PID=%ld] Lock can be placed\n", (long) getpid());
                else                            /* Locked out by someone else */
                    printf("[PID=%ld] Denied by %s lock on %lld:%lld "
                            "(held by PID %ld)\n", (long) getpid(),
                            (fl.l_type == F_RDLCK) ? "READ" : "WRITE",
                            (long long) fl.l_start,
                            (long long) fl.l_len, (long) fl.l_pid);
            }
        } else {                /* F_SETLK, F_SETLKW */
            if (status == 0)
                printf("[PID=%ld] %s\n", (long) getpid(),
                        (lock == 'u') ? "unlocked" : "got lock");
            else if (errno == EAGAIN || errno == EACCES)        /* F_SETLK */
                printf("[PID=%ld] failed (incompatible lock)\n",
                        (long) getpid());
            else if (errno == EDEADLK)                          /* F_SETLKW */
                printf("[PID=%ld] failed (deadlock)\n", (long) getpid());
            else
                errMsg("fcntl - F_SETLK(W)");
        }
    }
}
```

filelock/i_fcntl_locking.c

In the following shell session logs, we demonstrate the use of the program in Listing 55-2 by running two instances to place locks on the same 100-byte file (`tfile`). Figure 55-5 shows the state of granted and queued lock requests at various points during this shell session log, as noted in the commentary below.

We start a first instance (process A) of the program in Listing 55-2, placing a read lock on bytes 0 to 39 of the file:

```
Terminal window 1
$ ls -l tfile
-rw-r--r--    1 mtk      users         100 Apr 18 12:19 tfile
$ ./i_fcntl_locking tfile
Enter ? for help
PID=790> s r 0 40
[PID=790] got lock
```

Then we start a second instance of the program (process B), placing a read lock on a bytes 70 through to the end of the file:

```
Terminal window 2
$ ./i_fcntl_locking tfile
Enter ? for help
PID=800> s r -30 0 e
[PID=800] got lock
```

At this point, things appear as shown in part *a* of Figure 55-5, where process A (process ID 790) and process B (process ID 800) hold locks on different parts of the file.

Now we return to process A, where we try to place a write lock on the entire file. We first employ F_GETLK to test whether the lock can be placed and are informed that there is a conflicting lock. Then we try placing the lock with F_SETLK, which also fails. Finally, we try placing the lock with F_SETLKW, which blocks.

```
PID=790> g w 0 0
[PID=790] Denied by READ lock on 70:0 (held by PID 800)
PID=790> s w 0 0
[PID=790] failed (incompatible lock)
PID=790> w w 0 0
```

At this point, things appear as shown in part *b* of Figure 55-5, where process A and process B each hold a lock on different parts of the file, and process A has a queued lock request on the whole file.

We continue in process B, by trying to place a write lock on the entire file. We first test whether the lock can be placed using F_GETLK, which informs us that there is a conflicting lock. We then try placing the lock using F_SETLKW.

```
PID=800> g w 0 0
[PID=800] Denied by READ lock on 0:40
(held by PID 790)
PID=800> w w 0 0
[PID=800] failed (deadlock)
```

Part *c* of Figure 55-5 shows what happened when process B made a blocking request to place a write lock on the entire file: a deadlock. At this point, the kernel selected one of the lock requests to fail—in this case, the request by process B, which then receives the EDEADLK error from its *fcntl()* call.

We continue in process B, by removing all of its locks on the file:

```
PID=800> s u 0 0
[PID=800] unlocked
[PID=790] got lock
```

As we see from the last line of output, this allowed process A's blocked lock request to be granted.

It is important to realize that even though process B's deadlocked request was canceled, it still held its other lock, and so process A's queued lock request remained blocked. Process A's lock request is granted only when process B removes its other lock, bringing about the situation shown in part *d* of Figure 55-5.

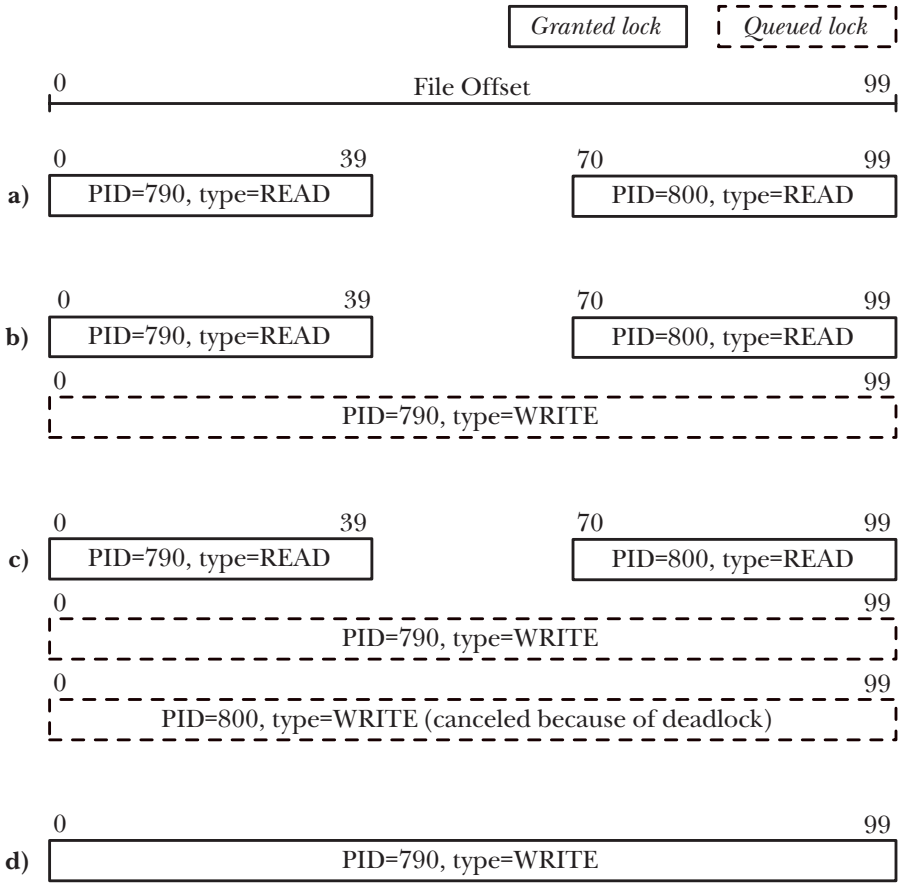

Figure 55-5: State of granted and queued lock requests while running `i_fcntl_locking.c`

55.3.3 Example: A Library of Locking Functions

Listing 55-3 provides a set of locking functions that we can use in other programs. These functions are as follows:

- The *lockRegion()* function uses `F_SETLK` to place a lock on the open file referred to by the file descriptor *fd*. The *type* argument specifies the lock type (`F_RDLCK` or `F_WRLCK`). The *whence*, *start*, and *len* arguments specify the range of bytes to lock. These arguments provide the values for the similarly named fields of the *flockstr* structure that is used to place the lock.
- The *lockRegionWait()* function is like *lockRegion()*, but makes a blocking lock request; that is, it uses `F_SETLKW`, rather than `F_SETLK`.

- The *regionIsLocked()* function tests whether a lock can be placed on a file. The arguments of this function are as for *lockRegion()*. This function returns 0 (false) if no process holds a lock that conflicts with the lock specified in the call. If one of more processes hold conflicting locks, then this function returns a nonzero value (i.e., true)—the process ID of one the processes holding a conflicting lock.

Listing 55-3: File region locking functions

filelock/region_locking.c

```
#include <fcntl.h>
#include "region_locking.h"             /* Declares functions defined here */

/* Lock a file region (private; public interfaces below) */

static int
lockReg(int fd, int cmd, int type, int whence, int start, off_t len)
{
    struct flock fl;

    fl.l_type = type;
    fl.l_whence = whence;
    fl.l_start = start;
    fl.l_len = len;

    return fcntl(fd, cmd, &fl);
}

int                     /* Lock a file region using nonblocking F_SETLK */
lockRegion(int fd, int type, int whence, int start, int len)
{
    return lockReg(fd, F_SETLK, type, whence, start, len);
}

int                     /* Lock a file region using blocking F_SETLKW */
lockRegionWait(int fd, int type, int whence, int start, int len)
{
    return lockReg(fd, F_SETLKW, type, whence, start, len);
}

/* Test if a file region is lockable. Return 0 if lockable, or
   PID of process holding incompatible lock, or -1 on error. */

pid_t
regionIsLocked(int fd, int type, int whence, int start, int len)
{
    struct flock fl;

    fl.l_type = type;
    fl.l_whence = whence;
    fl.l_start = start;
    fl.l_len = len;
```

```
    if (fcntl(fd, F_GETLK, &fl) == -1)
        return -1;

    return (fl.l_type == F_UNLCK) ? 0 : fl.l_pid;
}
```

filelock/region_locking.c

55.3.4 Lock Limits and Performance

SUSv3 allows an implementation to place fixed, system-wide upper limits on the number of record locks that can be acquired. When this limit is reached, *fcntl()* fails with the error ENOLCK. Linux doesn't set a fixed upper limit on the number of record locks that may be acquired; we are merely limited by availability of memory. (Many other UNIX implementations are similar.)

How quickly can record locks be acquired and released? There is no fixed answer to this question, since the speed of these operations is a function of the kernel data structure used to maintain record locks and the location of a particular lock within that data structure. We look at this structure in a moment, but first we consider some requirements that influence its design:

- The kernel needs to be able to merge a new lock with any existing locks (held by the same process) of the same mode that may lie on either side of the new lock.
- A new lock may completely replace one or more existing locks held by the calling process. The kernel needs to be able to easily locate all of these locks.
- When creating a new lock with a different mode in the middle of an existing lock, the job of splitting the existing lock (Figure 55-3) should be simple.

The kernel data structure used to maintain information about locks is designed to satisfy these requirements. Each open file has an associated linked list of locks held against that file. Locks within the list are ordered, first by process ID, and then by starting offset. An example of such a list is shown in Figure 55-6.

> The kernel also maintains *flock()* locks and file leases in the linked list of locks associated with an open file. (We briefly describe file leases when discussing the /proc/locks file in Section 55.5.) However, these types of locks are typically far fewer in number and therefore less likely to impact performance, so we ignore them in our discussion.

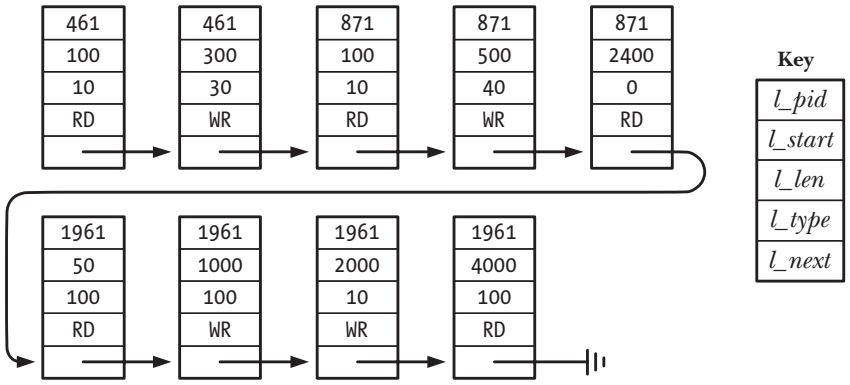

Figure 55-6: Example of a record lock list for a single file

Whenever a new lock is added to this data structure, the kernel must check for conflicts with any existing lock on the file. This search is carried out sequentially, starting at the head of the list.

Assuming a large number of locks distributed randomly among many processes, we can say that the time required to add or remove a lock increases roughly linearly with the number of locks already held on the file.

55.3.5 Semantics of Lock Inheritance and Release

The semantics of *fcntl()* record lock inheritance and release differ substantially from those for locks created using *flock()*. Note the following points:

- Record locks are not inherited across a *fork()* by a child process. This contrasts with *flock()*, where the child inherits a reference to the *same* lock and can release this lock, with the consequence that the parent also loses the lock.
- Record locks are preserved across an *exec()*. (However, note the effect of the close-on-exec flag, described below.)
- All of the threads in a process share the same set of record locks.
- Record locks are associated with both a process and an i-node (refer to Section 5.4). An unsurprising consequence of this association is that when a process terminates, all of its record locks are released. Less expected is that whenever a process closes a file descriptor, *all* locks held by the process on the corresponding file are released, regardless of the file descriptor(s) through which the locks were obtained. For example, in the following code, the *close(fd2)* call releases the lock held by the calling process on `testfile`, even though the lock was obtained via the file descriptor *fd1*:

```
struct flock fl;

fl.l_type = F_WRLCK;
fl.l_whence = SEEK_SET;
fl.l_start = 0;
fl.l_len = 0;

fd1 = open("testfile", O_RDWR);
fd2 = open("testfile", O_RDWR);

if (fcntl(fd1, cmd, &fl) == -1)
    errExit("fcntl");

close(fd2);
```

The semantics described in the last point apply no matter how the various descriptors referring to the same file were obtained and no matter how the descriptor is closed. For example, *dup()*, *dup2()*, and *fcntl()* can all be used to obtain duplicates of an open file descriptor. And, as well as performing an explicit *close()*, a descriptor can be closed by an *exec()* call if the close-on-exec flag was set, or via a *dup2()* call, which closes its second file descriptor argument if that descriptor is already open.

The semantics of *fcntl()* lock inheritance and release are an architectural blemish. For example, they make the use of record locks from library packages problematic, since a library function can't prevent the possibility that its caller will close a file descriptor referring to a locked file and thus remove a lock obtained by the library code. An alternative implementation scheme would have been to associate a lock with a file descriptor rather than with an i-node. However, the current semantics are the historical and now standardized behavior of record locks. Unfortunately, these semantics greatly limit the utility of *fcntl()* locking.

> With *flock()*, a lock is associated only with an open file description, and remains in effect until either any process holding a reference to the lock explicitly releases the lock or all file descriptors referring to the open file description are closed.

55.3.6 Lock Starvation and Priority of Queued Lock Requests

When multiple processes must wait in order to place a lock on a currently locked region, a couple of questions arise.

Can a process waiting to place a write lock be starved by a series of processes placing read locks on the same region? On Linux (as on many other UNIX implementations), a series of read locks can indeed starve a blocked write lock, possibly indefinitely.

When two or more processes are waiting to place a lock, are there any rules that determine which process obtains the lock when it becomes available? For example, are lock requests satisfied in FIFO order? And do the rules depend on the types of locks being requested by each process (i.e., does a process requesting a read lock have priority over one requesting a write lock or vice versa, or neither)? On Linux, the rules are as follows:

- The order in which queued lock requests are granted is indeterminate. If multiple processes are waiting to place locks, then the order in which they are satisfied depends on how the processes are scheduled.
- Writers don't have priority over readers, and vice versa.

Such statements don't necessarily hold true on other systems. On some UNIX implementations, lock requests are served in FIFO order, and readers have priority over writers.

55.4 Mandatory Locking

The kinds of locks we have described so far are *advisory*. This means that a process is free to ignore the use of *fcntl()* (or *flock()*) and simply perform I/O on the file. The kernel doesn't prevent this. When using advisory locking, it is up to the application designer to:

- set appropriate ownership (or group ownership) and permissions for the file, so as to prevent noncooperating process from performing file I/O; and
- ensure that the processes composing the application cooperate by obtaining the appropriate lock on the file before performing I/O.

Linux, like many other UNIX implementations, also allows *fcntl()* record locks to be *mandatory*. This means that every file I/O operation is checked to see whether it is compatible with any locks held by other processes on the region of the file on which I/O is being performed.

> Advisory mode locking is sometimes referred to as *discretionary locking*, while mandatory locking is sometimes referred to as *enforcement-mode locking*. SUSv3 doesn't specify mandatory locking, but it is available (with some variation in the details) on most modern UNIX implementations.

In order to use mandatory locking on Linux, we must enable it on the file system containing the files we wish to lock and on each file to be locked. We enable mandatory locking on a file system by mounting it with the (Linux-specific) *–o mand* option:

```
# mount -o mand /dev/sda10 /testfs
```

From a program, we can achieve the same result by specifying the `MS_MANDLOCK` flag when calling *mount(2)* (Section 14.8.1).

We can check whether a mounted file system has mandatory locking enabled by looking at the output of the *mount(8)* command with no options:

```
# mount | grep sda10
/dev/sda10 on /testfs type ext3 (rw,mand)
```

Mandatory locking is enabled on a file by the combination of having the set-group-ID permission bit turned on and the group-execute permission turned off. This combination of permission bits was otherwise meaningless and unused in earlier UNIX implementations. In this way, later UNIX systems added mandatory locking without needing to change existing programs or add new system calls. From the shell, we can enable mandatory locking on a file as follows:

```
$ chmod g+s,g-x /testfs/file
```

From a program, we can enable mandatory locking for a file by setting permissions appropriately using *chmod()* or *fchmod()* (Section 15.4.7).

When displaying permissions for a file whose permission bits are set for mandatory locking, *ls(1)* displays an *S* in the group-execute permission column:

```
$ ls -l /testfs/file
-rw-r-Sr--    1 mtk      users       0 Apr 22 14:11 /testfs/file
```

Mandatory locking is supported for all native Linux and UNIX file systems, but may not be supported on some network file systems or on non-UNIX file systems. For example, Microsoft's VFAT file system has no set-group-ID permission bit, so mandatory locking can't be used on VFAT file systems.

Effect of mandatory locking on file I/O operations

If mandatory locking is enabled for a file, what happens when a system call that performs data transfer (e.g., *read()* or *write()*) encounters a lock conflict (i.e., an attempt is made to write to a region that is currently read or write locked, or to read from a region that is currently write locked)? The answer depends on whether the file has been opened in blocking or nonblocking mode. If the file was opened

in blocking mode, the system call blocks. If the file was opened with the O_NONBLOCK flag, the system call immediately fails with the error EAGAIN. Similar rules apply for *truncate()* and *ftruncate()*, if the bytes they are attempting to add or remove from the file overlap a region currently locked (for reading or writing) by another process.

If we have opened a file in blocking mode (i.e., O_NONBLOCK is not specified in the *open()* call), then I/O system calls can be involved in deadlock situations. Consider the example shown in Figure 55-7, involving two processes that open the same file for blocking I/O, obtain write locks on different parts of the file, and then each attempt to write to the region locked by the other process. The kernel resolves this situation in the same way that deadlock between two *fcntl()* calls is resolved (Section 55.3.1): it selects one of the processes involved in the deadlock and causes its *write()* system call to fail with the error EDEADLK.

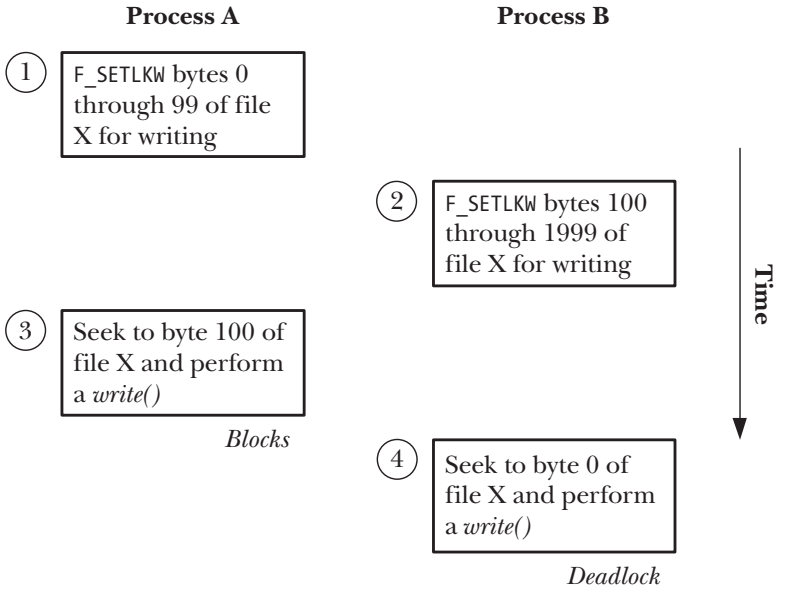

Figure 55-7: Deadlock when mandatory locking is in force

Attempts to *open()* a file with the O_TRUNC flag always fail immediately (with the error EAGAIN) if any other process holds a read or write lock on any part of the file.

It is not possible to create a shared memory mapping (i.e., *mmap()* with the MAP_SHARED flag) on a file if any other process holds a mandatory read or write lock on *any* part of the file. Conversely, it is not possible to place a mandatory lock on *any* part of a file that is currently involved in a shared memory mapping. In both cases, the relevant system call fails immediately with the error EAGAIN. The reason for these restrictions becomes clear when we consider the implementation of memory mappings. In Section 49.4.2, we saw that a shared file mapping both reads from and writes to a file (and the latter operation, in particular, conflicts with any type of lock on the file). Furthermore, this file I/O is performed by the memory-management subsystem, which has no knowledge of the location of any file locks in the system. Thus, to prevent a mapping from updating a file on which a mandatory lock is held, the kernel performs a simple check—testing at the time of the *mmap()* call whether there are locks anywhere in the file to be mapped (and vice versa for *fcntl()*).

Mandatory locking caveats

Mandatory locks do less for us than we might at first expect, and have some potential shortcomings and problems:

- Holding a mandatory lock on a file doesn't prevent another process from deleting it, since all that is required to unlink a file is suitable permissions on the parent directory.
- Careful consideration should be applied before enabling mandatory locks on a publicly accessible file, since not even privileged processes can override a mandatory lock. A malicious user could continuously hold a lock on the file in order to create a denial-of-service attack. (While in most cases, we could make the file accessible once more by turning off the set-group-ID bit, this may not be possible if, for example, the mandatory file lock is causing the system to hang.)
- There is a performance cost associated with the use of mandatory locking. For each I/O system call made on a file with mandatory locking enabled, the kernel must check for lock conflicts on the file. If the file has a large number of locks, this check can slow I/O system calls significantly.
- Mandatory locking also incurs a cost in application design. We need to handle the possibility that each I/O system call can return `EAGAIN` (for nonblocking I/O) or `EDEADLK` (for blocking I/O).
- As a consequence of some kernel race conditions in the current Linux implementation, there are circumstances in which system calls that perform I/O operations can succeed despite the presence of mandatory locks that should deny those operations.

In summary, the use of mandatory locks is best avoided.

55.5 The `/proc/locks` File

We can view the set of locks currently held in the system by examining the contents of the Linux-specific `/proc/locks` file. Here is an example of the information we can see in this file (in this case, for four locks):

```
$ cat /proc/locks
1: POSIX  ADVISORY  WRITE 458 03:07:133880 0 EOF
2: FLOCK  ADVISORY  WRITE 404 03:07:133875 0 EOF
3: POSIX  ADVISORY  WRITE 312 03:07:133853 0 EOF
4: FLOCK  ADVISORY  WRITE 274 03:07:81908 0 EOF
```

The `/proc/locks` file displays information about locks created by both *flock()* and *fcntl()*. The eight fields shown for each lock are as follows (from left to right):

1. The ordinal number of the lock within the set of all locks held for this file. (Refer to Section 55.3.4.)
2. The type of lock. Here, `FLOCK` indicates a lock created by *flock()*, and `POSIX` indicates a lock created by *fcntl()*.
3. The mode of the lock, either `ADVISORY` or `MANDATORY`.

4. The type of lock, either `READ` or `WRITE` (corresponding to shared and exclusive locks for *fcntl()*).
5. The process ID of the process holding the lock.
6. Three colon-separated numbers that identify the file on which the lock is held. These numbers are the major and minor device numbers of the file system on which the file resides, followed by the i-node number of the file.
7. The starting byte of the lock. This is always 0 for *flock()* locks.
8. The ending byte of the lock. Here, `EOF` indicates that the lock runs to the end of the file (i.e., *l_len* was specified as 0 for a lock created by *fcntl()*). For *flock()* locks, this column is always `EOF`.

> In Linux 2.4 and earlier, each line of `/proc/locks` includes five additional hexadecimal values. These are pointer addresses used by the kernel to record locks in various lists. These values are not useful in application programs.

Using the information in `/proc/locks`, we can find out which process is holding a lock, and on what file. The following shell session shows how to do this for lock number 3 in the list above. This lock is held by process ID 312, on the i-node 133853 on the device with major ID 3 and minor ID 7. We begin by using *ps(1)* to list information about the process with process ID 312:

```
$ ps -p 312
  PID TTY          TIME CMD
  312 ?        00:00:00 atd
```

The above output shows that the program holding the lock is *atd*, the daemon that executes scheduled batch jobs.

In order to find the locked file, we first search the files in the `/dev` directory, and thus determine that the device with ID 3:7 is `/dev/sda7`:

```
$ ls -li /dev/sda7 | awk '$6 == "3," && $7 == 10'
  1311 brw-rw----    1 root   disk    3,  7 May 12  2006 /dev/sda7
```

We then determine the mount point for the device `/dev/sda7` and search that part of the file system for the file whose i-node number is 133853:

```
$ mount | grep sda7
/dev/sda7 on / type reiserfs (rw)            Device is mounted on /
$ su                                         So we can search all directories
Password:
# find / -mount -inum 133853                 Search for i-node 133853
/var/run/atd.pid
```

The *find −mount* option prevents *find* from descending into subdirectories under `/` that are mount points for other file systems.

Finally, we display the contents of the locked file:

```
# cat /var/run/atd.pid
312
```

Thus, we see that the *atd* daemon is holding a lock on the file `/var/run/atd.pid`, and that the content of this file is the process ID of the process running *atd*. This daemon

is employing a technique to ensure that only one instance of the daemon is running at a time. We describe this technique in Section 55.6.

We can also use /proc/locks to obtain information about blocked lock requests, as demonstrated in the following output:

```
$ cat /proc/locks
1: POSIX  ADVISORY  WRITE 11073 03:07:436283 100 109
1: -> POSIX  ADVISORY  WRITE 11152 03:07:436283 100 109
2: POSIX  MANDATORY WRITE 11014 03:07:436283 0 9
2: -> POSIX  MANDATORY WRITE 11024 03:07:436283 0 9
2: -> POSIX  MANDATORY READ  11122 03:07:436283 0 19
3: FLOCK  ADVISORY  WRITE 10802 03:07:134447 0 EOF
3: -> FLOCK  ADVISORY  WRITE 10840 03:07:134447 0 EOF
```

Lines shown with the characters -> immediately after a lock number represent lock requests blocked by the corresponding lock number. Thus, we see one request blocked on lock 1 (an advisory lock created with *fcntl()*), two requests blocked on lock 2 (a mandatory lock created with *fcntl()*), and one request blocked on lock 3 (a lock created with *flock()*).

> The /proc/locks file also displays information about any file leases that are held by processes on the system. File leases are a Linux-specific mechanism available in Linux 2.4 and later. If a process takes out a lease on a file, then it is notified (by delivery of a signal) if another process tries to *open()* or *truncate()* that file. (The inclusion of *truncate()* is necessary because it is the only system call that can be used to change the contents of a file without first opening it.) File leases are provided in order to allow Samba to support the *opportunistic locks* (*oplocks*) functionality of the Microsoft SMB protocol and to allow NFS version 4 to support *delegations* (which are similar to SMB oplocks). Further details about file leases can be found under the description of the F_SETLEASE operation in the *fcntl(2)* manual page.

55.6 Running Just One Instance of a Program

Some programs—in particular, many daemons—need to ensure that only one instance of the program is running on the system at a time. A common method of doing this is to have the daemon create a file in a standard directory and place a write lock on it. The daemon holds the file lock for the duration of its execution and deletes the file just before terminating. If another instance of the daemon is started, it will fail to obtain a write lock on the file. Consequently, it will realize that another instance of the daemon must already be running, and terminate.

> Many network servers use an alternative convention of assuming that a server instance is already running if the well-known socket port to which the server binds is already in use (Section 61.10).

The /var/run directory is the usual location for such lock files. Alternatively, the location of the file may be specified by a line in the daemon's configuration file.

Conventionally, a daemon writes its own process ID into the lock file, and hence the file is often named with an extension .pid (for example, *syslogd* creates the file /var/run/syslogd.pid). This is useful if some application needs to find the

process ID of the daemon. It also allows an extra sanity check—we can verify whether that process ID exists using *kill(pid, 0)*, as described in Section 20.5. (In older UNIX implementations that did not provide file locking, this was used as an imperfect, but usually practicable, way of assessing whether an instance of the daemon really was still running, or whether an earlier instance had simply failed to delete the file before terminating.)

There are many minor variations in the code used to create and lock a process ID lock file. Listing 55-4 is based on ideas presented in [Stevens, 1999] and provides a function, *createPidFile()*, that encapsulates the steps described above. We would typically call this function with a line such as the following:

```
if (createPidFile("mydaemon", "/var/run/mydaemon.pid", 0) == -1)
    errExit("createPidFile");
```

One subtlety in the *createPidFile()* function is the use of *ftruncate()* to erase any previous string in the lock file. This is done because the last instance of the daemon may have failed to delete the file, perhaps because of a system crash. In this case, if the process ID of the new daemon instance is small, we might otherwise not completely overwrite the previous contents of the file. For example, if our process ID is 789, then we would write just `789\n` to the file, but a previous daemon instance might have written `12345\n`. If we did not truncate the file, then the resulting content would be `789\n5\n`. Erasing any existing string may not be strictly necessary, but it is tidier and removes any potential for confusion.

The *flags* argument can specify the constant `CPF_CLOEXEC`, which causes *createPidFile()* to set the close-on-exec flag (Section 27.4) for the file descriptor. This is useful for servers that restart themselves by calling *exec()*. If the file descriptor was not closed during the *exec()*, then the restarted server would think that a duplicate instance of the server is already running.

Listing 55-4: Creating a PID lock file to ensure just one instance of a program is started

filelock/create_pid_file.c

```
#include <sys/stat.h>
#include <fcntl.h>
#include "region_locking.h"             /* For lockRegion() */
#include "create_pid_file.h"            /* Declares createPidFile() and
                                           defines CPF_CLOEXEC */
#include "tlpi_hdr.h"

#define BUF_SIZE 100            /* Large enough to hold maximum PID as string */

/* Open/create the file named in 'pidFile', lock it, optionally set the
   close-on-exec flag for the file descriptor, write our PID into the file,
   and (in case the caller is interested) return the file descriptor
   referring to the locked file. The caller is responsible for deleting
   'pidFile' file (just) before process termination. 'progName' should be the
   name of the calling program (i.e., argv[0] or similar), and is used only for
   diagnostic messages. If we can't open 'pidFile', or we encounter some other
   error, then we print an appropriate diagnostic and terminate. */
```

```
int
createPidFile(const char *progName, const char *pidFile, int flags)
{
    int fd;
    char buf[BUF_SIZE];

    fd = open(pidFile, O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    if (fd == -1)
        errExit("Could not open PID file %s", pidFile);

    if (flags & CPF_CLOEXEC) {

        /* Set the close-on-exec file descriptor flag */

        flags = fcntl(fd, F_GETFD);                     /* Fetch flags */
        if (flags == -1)
            errExit("Could not get flags for PID file %s", pidFile);

        flags |= FD_CLOEXEC;                            /* Turn on FD_CLOEXEC */

        if (fcntl(fd, F_SETFD, flags) == -1)            /* Update flags */
            errExit("Could not set flags for PID file %s", pidFile);
    }

    if (lockRegion(fd, F_WRLCK, SEEK_SET, 0, 0) == -1) {
        if (errno  == EAGAIN || errno == EACCES)
            fatal("PID file '%s' is locked; probably "
                    "'%s' is already running", pidFile, progName);
        else
            errExit("Unable to lock PID file '%s'", pidFile);
    }

    if (ftruncate(fd, 0) == -1)
        errExit("Could not truncate PID file '%s'", pidFile);

    snprintf(buf, BUF_SIZE, "%ld\n", (long) getpid());
    if (write(fd, buf, strlen(buf)) != strlen(buf))
        fatal("Writing to PID file '%s'", pidFile);

    return fd;
}
```

filelock/create_pid_file.c

55.7 Older Locking Techniques

In older UNIX implementations that lacked file locking, a number of *ad hoc* locking techniques were employed. Although all of these have been superseded by *fcntl()* record locking, we describe them here since they still appear in some older programs. All of these techniques are advisory in nature.

`open(file, O_CREAT | O_EXCL,...)` plus `unlink(file)`

SUSv3 requires that an *open()* call with the flags `O_CREAT` and `O_EXCL` perform the steps of checking for the existence of a file and creating it atomically (Section 5.1). This means that if two processes attempt to create a file specifying these flags, it is guaranteed that only one of them will succeed. (The other process will receive the error `EEXIST` from *open()*.) Used in conjunction with the *unlink()* system call, this provides the basis for a locking mechanism. Acquiring the lock is performed by successfully opening the file with the `O_CREAT` and `O_EXCL` flags, followed by an immediate *close()*. Releasing the lock is performed using *unlink()*. Although workable, this technique has several limitations:

- If the *open()* fails, indicating that some other process has the lock, then we must retry the *open()* in some kind of loop, either polling continuously (which wastes CPU time) or with a delay between each attempt (which means that there may be some delay between the time the lock becomes available and when we actually acquire it). With *fcntl()*, we can use `F_SETLKW` to block until the lock becomes free.
- Acquiring and releasing locks using *open()* and *unlink()* involves file-system operations that are rather slower than the use of record locks. (On one of the author's x86-32 systems running Linux 2.6.31, acquiring and releasing 1 million locks on an *ext3* file using the technique described here required 44 seconds. Acquiring and releasing 1 million record locks on the same byte of a file required 2.5 seconds.)
- If a process accidentally exits without deleting the lock file, the lock is not released. There are *ad hoc* techniques for handling this problem, including checking the last modification time of the file and having the lock holder write its process ID to the file so that we can check if the process still exists, but none of these techniques is foolproof. By comparison, record locks are released automatically when a process terminates.
- If we are placing multiple locks (i.e., using multiple lock files), deadlocks are not detected. If a deadlock arises, the processes involved in the deadlock will remain blocked indefinitely. (Each process will be spinning, checking to see if it can obtain the lock it requires.) By contrast, the kernel provides deadlock detection for *fcntl()* record locks.
- NFS version 2 doesn't support `O_EXCL` semantics. Linux 2.4 NFS clients also fail to implement `O_EXCL` correctly, even for NFS version 3 and later.

`link(file, lockfile)` plus `unlink(lockfile)`

The fact that the *link()* system call fails if the new link already exists has also been used as a locking mechanism, again employing *unlink()* to perform the unlock function. The usual approach is to have each process that needs to acquire the lock create a unique temporary filename, typically one including the process ID (and possibly the hostname, if the lock file is created on a network file system). To acquire the lock, this temporary file is linked to some agreed-upon standard pathname. (The semantics of hard links require that the two pathnames reside in the same file system.)

If the *link()* call succeeds, we have obtained the lock. If it fails (EEXIST), then another process has the lock and we must try again later. This technique suffers the same limitations as the *open(file, O_CREAT | O_EXCL,...)* technique described above.

open(file, O_CREAT | O_TRUNC | O_WRONLY, 0) plus unlink(file)

The fact that calling *open()* on an existing file fails if O_TRUNC is specified and write permission is denied on the file can be used as the basis of a locking technique. To obtain a lock, we use the following code (which omits error checking) to create a new file:

```
fd = open(file, O_CREAT | O_TRUNC | O_WRONLY, (mode_t) 0);
close(fd);
```

> For an explanation of why we use the *(mode_t)* cast in the *open()* call above, see Appendix C.

If the *open()* call succeeds (i.e., the file didn't previously exist), we have the lock. If it fails with EACCES (i.e., the file exists and has no permissions for anyone), then another process has the lock, and we must try again later. This technique suffers the same limitations as the previous techniques, with the added caveat that we can't employ it in a program with superuser privileges, since the *open()* call will always succeed, regardless of the permissions that are set on the file.

55.8 Summary

File locks allow processes to synchronize access to a file. Linux provides two file locking system calls: the BSD-derived *flock()* and the System V–derived *fcntl()*. Although both system calls are available on most UNIX implementations, only *fcntl()* locking is standardized in SUSv3.

The *flock()* system call locks an entire file. Two types of locks may be placed: shared locks, which are compatible with shared locks held by other processes, and exclusive locks, which prevent other processes from placing any type of lock.

The *fcntl()* system call places locks ("record locks") on any region of a file, ranging from a single byte to the entire file. Two types of locks may be placed: read locks and write locks, which have similar compatibility semantics to the shared and exclusive locks placed via *flock()*. If a blocking (F_SETLKW) lock request would bring about a deadlock situation, then the kernel causes *fcntl()* to fail (with the error EDEADLK) in one of the affected processes.

Locks placed using *flock()* and *fcntl()* are invisible to one another (except on systems that implement *flock()* using *fcntl()*). The locks placed via *flock()* and *fcntl()* have different semantics with respect to inheritance across *fork()* and release when file descriptors are closed.

The Linux-specific /proc/locks file displays the file locks currently held by all processes on the system.

Further information

An extensive discussion of *fcntl()* record locking can be found in [Stevens & Rago, 2005] and [Stevens, 1999]. Some details of the implementation of *flock()* and *fcntl()*

locking on Linux are provided in [Bovet & Cesati, 2005]. [Tanenbaum, 2007] and [Deitel et al., 2004] describe deadlocking concepts in general, including coverage of deadlock detection, avoidance, and prevention.

55.9 Exercises

55-1. Experiment by running multiple instances of the program in Listing 55-1 (`t_flock.c`) to determine the following points about the operation of *flock()*:

a) Can a series of processes acquiring shared locks on a file starve a process attempting to place an exclusive lock on the file?

b) Suppose that a file is locked exclusively, and other processes are waiting to place both shared and exclusive locks on the file. When the first lock is released, are there any rules determining which process is next granted a lock? For example, do shared locks have priority over exclusive locks or vice versa? Are locks granted in FIFO order?

c) If you have access to some other UNIX implementation that provides *flock()*, try to determine the rules on that implementation.

55-2. Write a program to determine whether *flock()* detects deadlock situations when being used to lock two different files in two processes.

55-3. Write a program to verify the statements made in Section 55.2.1 regarding the semantics of inheritance and release of *flock()* locks.

55-4. Experiment by running the programs in Listing 55-1 (`t_flock.c`) and Listing 55-2 (`i_fcntl_locking.c`) to see whether locks granted by *flock()* and *fcntl()* have any effect on one another. If you have access to other UNIX implementations, try the same experiment on those implementations.

55-5. In Section 55.3.4, we noted that, on Linux, the time required to add or check for the existence of a lock is a function of the position of the lock in the list of all locks on the file. Write two programs to verify this:

a) The first program should acquire (say) 40,001 write locks on a file. These locks are placed on alternating bytes of the file; that is, locks are placed on bytes 0, 2, 4, 6, and so on through to (say) byte 80,000. Having acquired these locks, the process then goes to sleep.

b) While the first program is sleeping, the second program loops (say) 10,000 times, using `F_SETLK` to try to lock one of the bytes locked by the previous program (these lock attempts always fail). In any particular execution, the program always tries to lock byte *N * 2* of the file.

Using the shell built-in *time* command, measure the time required by the second program for *N* equals 0, 10,000, 20,000, 30,000, and 40,000. Do the results match the expected linear behavior?

55-6. Experiment with the program in Listing 55-2 (`i_fcntl_locking.c`) to verify the statements made in Section 55.3.6 regarding lock starvation and priority for *fcntl()* record locks.

55-7. If you have access to other UNIX implementations, use the program in Listing 55-2 (`i_fcntl_locking.c`) to see if you can establish any rules for *fcntl()* record locking regarding starvation of writers and regarding the order in which multiple queued lock requests are granted.

55-8. Use the program in Listing 55-2 (`i_fcntl_locking.c`) to demonstrate that the kernel detects circular deadlocks involving three (or more) processes locking the same file.

55-9. Write a pair of programs (or a single program that uses a child process) to bring about the deadlock situation with mandatory locks described in Section 55.4.

55-10. Read the manual page of the *lockfile(1)* utility that is supplied with *procmail*. Write a simple version of this program.