Chapter 4

File and Record Locking

This chapter describes how you can use file and record locking capabilities. Examples are given for the correct use of record locking. Misconceptions about the amount of protection that record locking affords are dispelled. Record locking should be viewed as a synchronization mechanism, not a security mechanism.

The chapter includes these topics:

- "Overview of File and Record Locking" presents an introduction to locking mechanisms.
- "Terminology" defines some common terms about locking.
- "Using Record Locking" covers using access permissions, locking files, and getting lock information.
- "Selecting Advisory or Mandatory Locking" describes mandatory locking and record locking across systems.

Overview of File and Record Locking

Mandatory and advisory file and record locking are available on many current releases of the UNIX operating system. The intent of these capabilities is to provide a synchronization mechanism for programs accessing the same stores of data simultaneously. Such processing is characteristic of many multiuser applications, and the need for a standard method of dealing with the problem has been recognized by standards advocates like */usr/group*, an organization of UNIX system users from businesses and campuses across the country.

Advisory file and record locking can be used to coordinate independent, unrelated processes. In mandatory locking, on the other hand, the standard I/O subroutines and I/O system calls enforce the locking protocol. In this way, at the cost of a little efficiency, mandatory locking double-checks the programs to avoid accessing the data out of sequence.

The engineering reference data for record locking is found in the fcntl(2), lockf(3), and fcntl(5) reference pages. You may want to skim those pages before continuing.

Terminology

Before discussing record locking mechanisms, first consider a few terms.

Record

A record is any contiguous sequence of bytes in a file. The UNIX operating system does not impose any record structure on files. The boundaries of records are defined by the programs that use the files. Within a single file, a record as defined by one process can overlap partially or completely on a record as defined by some other process.

Cooperating Processes

Cooperating processes work together in a well-defined fashion to accomplish the tasks at hand. Processes can coordinate their use of resources of different types in many ways, including semaphores in shared memory. Processes that share files must request permission to access the files before using them. File access permissions must be carefully set to prevent noncooperating processes from accessing those files. The term "process" is used interchangeably with "cooperating process" to refer to a task obeying such protocols.

Read (Shared) Lock

A read lock keeps a record from changing while one or more processes read the data. If a process holds a read lock, it may assume that no other process will be writing or updating that record at the same time. When a read lock is in place on a record, other processes may also read-lock that record or a record that overlaps the locked record. No other process, however, may have or obtain a write lock on that record or an overlapping record.

Write (Exclusive) Lock

A write lock is used to gain complete control over a record. When a write lock is in place on a record, no other process may read- or write-lock that record or a record that overlaps it. If a process holds a write lock it may assume that no other process will be reading or writing that record at the same time.

Advisory Locking

Advisory locking has no effect on the file I/O subsystem functions or the processes that use them. File I/O functions include the **creat()**, **open()**, **read()**, and **write()** calls.

Advisory control is accomplished by requiring all cooperating processes to make an appropriate record lock request before performing any I/O operation. When all processes use advisory locking, the accessibility of the file is controlled by the interaction of these requests. Advisory locking depends on the individual processes to enforce the record locking protocol; it is not enforced by the file I/O subsystem.

Mandatory Locking

Mandatory record locking is enforced by the file I/O subsystem and so is effective on unrelated processes that may not be part of the cooperating group. Access to locked records is enforced by the **creat()**, **open()**, **read()**, and **write()** system calls. When a record is locked, access to that record by any other process is restricted according to the type of lock on the record. Cooperating processes should still request an appropriate record lock before an I/O operation, but an additional check is made by IRIX before each I/O operation to ensure the record locking protocol is being honored. Mandatory locking offers security against unplanned file use by unrelated programs, but it imposes additional system overhead on access to the controlled files.

Lock Promotion and Demotion

Promoting a read lock to write-lock status is permitted if no other process is holding a read lock in the same record. If processes exist with pending write locks that are waiting for the same record, the lock promotion succeeds and the other (sleeping) processes wait. Promoting (or demoting) a write lock to a read lock carries no restrictions. In either case, the lock is reset with the new lock type.

Because the **lockf()** function does not support read locks, lock promotion is not applicable to locks set with that call.

Using File Permissions

The access permissions for each UNIX file control who may read, write, or execute the file. These access permissions may be set only by the owner of the file or by the superuser. The permissions of the directory in which the file resides can also affect the access permissions for a file. Note that if the permissions for a directory allow anyone to write in the directory, then files within that directory may be removed even by a user who does not have read, write, or execute permission for those files.

If your application warrants the use of record locking, make sure that the permissions on your files and directories are also set properly. A record lock, even a mandatory record lock, protects only the records that are locked, while they are locked. Other parts of the files can be corrupted if proper precautions are not taken.

Only a known set of programs and/or users should be able to read or write a database. This can be enforced through file permissions as follows:

1. Using the *chown* facility (see the chown(1) and chown(2) reference pages), set the ownership of the critical directories and files to reflect the authorized group ID.
2. Using the *chmod* facility (see also the chmod(1) and chmod(2) reference pages), set the file permissions of the critical directories and files so that only members of the authorized group have write access ("775" permissions).
3. Using the *chown* facility, set the accessing program executable files to be owned by the authorized group.
4. Using the *chmod* facility, set the set-GID bit for each accessing program executable file, and to permit execution by anyone ("2755" permissions).

Users who are not members of the authorized group cannot modify the critical directories and files. However, when an ordinary user executes one of the accessing programs, the program automatically adopts the group ID of its owner. The accessing program can create and modify files in the critical directory, but other programs started by an ordinary user cannot.

Using Record Locking

This section covers the following topics:

- "Opening a File for Record Locking"
- "Setting a File Lock"
- "Setting and Removing Record Locks"
- "Getting Lock Information"
- "Deadlock Handling"

Opening a File for Record Locking

The first requirement for locking a file or segment of a file is having a valid open file descriptor. If read locks are to be used, then the file must be opened with at least read access; likewise for write locks and write access.

Example 4-1 opens a file for both read and write access.

Example 4-1 Opening a File for Locked Use

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
int fd;    /* file descriptor */
char *filename;
main(argc, argv)
int argc;
char *argv[];
{
    extern void exit(), perror();
    /* get database file name from command line and open the
     * file for read and write access.
     */
    if (argc < 2) {
        (void) fprintf(stderr, "usage: %s filename\n", argv[0]);
        exit(2);
    }
    filename = argv[1];
    fd = open(filename, O_RDWR);
    if (fd < 0) {
        perror(filename);
        exit(2);
    }
}
```

The file is now open to perform both locking and I/O functions. The next step is to set a lock.

Setting a File Lock

Several ways exist to set a lock on a file. These methods depend upon how the lock interacts with the rest of the program. Issues of portability and performance need to be considered. Two methods for setting a lock are given here: using the **fcntl()** system call and using the */usr/group* standards-compatible **lockf()** library function call.

Locking an entire file is just a special case of record locking—one record is locked, which has the size of the entire file. The file is locked starting at a byte offset of zero and size of the maximum file size. This size is beyond any real end-of-file so that no other lock can be placed on the file.

Since a whole-file lock is a common operation, the **fcntl()** function treats a lock size of 0 as meaning "size of file." Example 4-2 contains a function that attempts a specified number of times to obtain a whole-file lock using **fcntl()**. When the lock is placed, it returns 1; if the lock cannot be placed, it returns 0..

Example 4-2 Setting a Whole-File Lock With fcntl()

```
#include <fcntl.h>
#define MAX_TRY 10
int
lockWholeFile(int fd, int tries)
{
    int limit = (tries)?tries:MAX_TRY;
    int try = 0;
    struct flock lck;
    /* set up the record locking structure for whole file */
    lck.l_type = F_WRLCK; /* setting a write lock */
    lck.l_whence = 0; /* offset l_start from beginning of file */
    lck.l_start = 0L;
    lck.l_len = 0L; /* until the end of the file address space */
    /* Attempt locking MAX_TRY times before giving up. */
    while (0 > fcntl(fd, F_SETLK, &lck))
    {
        if ((++try < limit) && (errno == EAGAIN || errno == EACCES))
        {
            (void) sginap(1);
            continue;
        }
        else
        { /* not EAGAIN|EACCES, or too many failures */
            perror("fcntl");
            return 0;
        }
    }
    return 1;
}
```

Example 4-3 shows a version of the same function that performs the same task using the **lockf()** function.

Example 4-3 Setting a Whole-File Lock With lockf()

```
#include <unistd.h>
#define MAX_TRY 10
int
lockWholeFile(int fd, int tries)
{
    int limit = (tries)?tries:MAX_TRY;
    int try = 0;
    /* make sure the file ptr is at the beginning of file. */
    lseek(fd, 0L, 0);
    /* Attempt locking MAX_TRY times before giving up. */
    while (0 > lockf(fd, F_TLOCK, 0L))
    {
        if ((++try < limit) && (errno == EAGAIN || errno == EACCES))
        {
            (void) sginap(1);
            continue;
        }
        else
        { /* not EAGAIN|EACCES, or too many failures */
            perror("lockf");
            return 0;
        }
    }
    return 1;
}
```

The following differences between Example 4-2 and Example 4-3 that should be noted:

- In Example 4-2, the type of lock (F_WRLCK) is specified in the *l_type* field. **fcntl()** supports both read and write locks.
- In Example 4-3, the type of lock is not specified—**lockf()** supports only write (exclusive) locks.
- In Example 4-3, the operation type (F_TLOCK) specifies that the function should return if the lock cannot be placed immediately (the F_LOCK operation would sleep until the lock could be placed). The comparable feature in Example 4-2 is the use of F_SETLK, where F_SETLKW would sleep.
- In Example 4-2, the starting location of the record is set as the sum of two fields, *l_whence* and *l_start*. In Example 4-3, the start of the record is set implicitly by the current file position, so **lseek()** is called to ensure the correct position before **lockf()** is called.

Setting and Removing Record Locks

Locking a record is done the same way as locking a file except that the record does not encompass the entire file contents. This section examines an example problem of dealing with two records (which may be either in the same file or in different files) that must be updated simultaneously so that other processes get a consistent view of the information they contain. This type of problem occurs, for example, when updating the inter-record pointers in a doubly linked list.

To deal with multiple locks, consider the following questions:

- What do you want to lock?
- For multiple locks, in what order do you want to lock and unlock the records?
- What do you do if you succeed in getting all the required locks?
- What do you do if you fail to get all the locks?

In managing record locks, you must plan a failure strategy for the case in which you cannot obtain all the required locks. It is because of contention for these records that you have decided to use record locking in the first place. Different programs might:

- wait a certain amount of time, and try again
- end the procedure and warn the user

- let the process sleep until signaled that the lock has been freed
- a combination of the above

Look now at the example of inserting an entry into a doubly linked list. All the following examples assume that a record is declared as follows:

```
struct record {
.../* data portion of record */...
    long prev;    /* index to previous record in the list */
    long next;    /* index to next record in the list */
};
```

For the example, assume that the record after which the new record is to be inserted has a read lock on it already. The lock on this record must be promoted to a write lock so that the record may be edited. Example 4-4 shows a function that can be used for this.

Example 4-4 Record Locking With Promotion Using fcntl()

```
/*
|| This function is called with a file descriptor and the
|| offsets to three records in it: this, here, and next.
|| The caller is assumed to hold read locks on both here and next.
|| This function promotes these locks to write locks.
|| If write locks on "here" and "next" are obtained
||    Set a write lock on "this".
||    Return index to "this" record.
|| If any write lock is not obtained:
||    Restore read locks on "here" and "next".
||    Remove all other locks.
||    Return -1.
*/
long set3Locks(int fd, long this, long here, long next)
{
    struct flock lck;
    lck.l_type = F_WRLCK;    /* setting a write lock */
    lck.l_whence = 0;        /* offsets are absolute */
    lck.l_len = sizeof(struct record);
    /* Promote the lock on "here" to write lock */
    lck.l_start = here;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        return (-1);
    }
    /* Lock "this" with write lock */
    lck.l_start = this;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        /* Failed to lock "this"; return "here" to read lock. */
        lck.l_type = F_RDLCK;
        lck.l_start = here;
        (void) fcntl(fd, F_SETLKW, &lck);
        return (-1);
    }
    /* Promote lock on "next" to write lock */
    lck.l_start = next;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        /* Failed to promote "next"; return "here" to read lock... */
        lck.l_type = F_RDLCK;
        lck.l_start = here;
        (void) fcntl(fd, F_SETLK, &lck);
        /* ...and remove lock on "this".  */
        lck.l_type = F_UNLCK;
        lck.l_start = this;
        (void) fcntl(fd, F_SETLK, &lck);
        return (-1)
    }
    return (this);
}
```

Example 4-4 uses the F_SETLKW command to **fcntl()**, with the result that the calling process will sleep if there are conflicting locks at any of the three points. If the F_SETLK command was used instead, the **fcntl()** system calls would fail if blocked. The program would then have to be changed to handle the blocked condition in each of the error return sections (as in Example 4-2).

It is possible to unlock or change the type of lock on a subsection of a previously set lock; this may cause an additional lock (two locks for one system call) to be used by the operating system. This occurs if the subsection is from the middle of the previously set lock.

Example 4-5 shows a similar example using the **lockf()** function. Since it does not support read locks, all (write) locks are referenced generically as locks.

Example 4-5 Record Locking Using lockf()

```
/*
|| This function is called with a file descriptor and the
|| offsets to three records in it: this, here, and next.
|| The caller is assumed to hold no locks on any of the records.
|| This function tries to lock “here” and “next” using lockf().
|| If locks on “here” and “next” are obtained
||    Set a lock on “this”.
||    Return index to “this” record.
|| If any lock is not obtained:
||    Remove all other locks.
||    Return -1.
*/
long set3Locks(int fd, long this, long here, long next)
{
    /* Set a lock on “here” */
    (void) lseek(fd, here, 0);
    if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
        return (-1);
    }
    /* Lock “this” */
    (void) lseek(fd, this, 0);
    if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
        /* Failed to lock “this”; clear “here” lock. */
        (void) lseek(fd, here, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        return (-1);
    }
    /* Lock “next” */
    (void) lseek(fd, next, 0);
    if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
        /* Failed to lock “next”; release “here”... */
        (void) lseek(fd, here, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        /* ...and remove lock on “this”.  */
        (void) lseek(fd, this, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        return (-1)
    }
    return (this);
}
```

Locks are removed in the same manner as they are set; only the lock type is different (F_UNLCK or F_ULOCK). An unlock cannot be blocked by another process. An unlock can affect only locks that were placed by the unlocking process.

Getting Lock Information

You can determine which processes, if any, are blocking a lock from being set. This can be used as a simple test or as a means to find locks on a file. To find this information, set up a lock as in the previous examples and use the F_GETLK command in the **fcntl()** call. If the lock passed to **fcntl()** would be blocked, the first blocking lock is returned to the process through the structure passed to **fcntl()**. That is, the lock data passed to **fcntl()** is overwritten by blocking lock information.

The returned information includes two pieces of data, *l_pidf* and *l_sysid*, that are used only with F_GETLK. These fields uniquely identify the process holding the lock. (For systems that do not support a distributed architecture, the value in *l_sysid* can be ignored.)

If a lock passed to **fcntl()** using the F_GETLK command is not blocked by another lock, the *l_type* field is changed to F_UNLCK and the remaining fields in the structure are unaffected.

Example 4-6 shows how to use this capability to print all the records locked by other processes. Note that if several read locks occur over the same record, only one of these is found.

Example 4-6 Detecting Contending Locks Using fcntl()

```
/*
|| This function takes a file descriptor and prints a report showing
|| all locks currently set on that file. The loop variable is the
|| l_start field of the flock structure. The function asks fcntl()
|| for the first lock that would block a lock from l_start to the end
|| of the file (l_len==0). When no lock would block such a lock,
|| the returned l_type contains F_UNLCK and the loop ends.
|| Otherwise the contending lock is displayed, l_start is set to
|| the end-point of that lock, and the loop repeats.
*/
void printAllLocksOn(int fd)
{
    struct flock lck;
    /* Find and print "write lock" blocked segments of file. */
    (void) printf("sysid pid type start length\n");
    lck.l_whence = 0;
    lck.l_start = 0L;
    lck.l_len = 0L;
    for( lck.l_type = 0; lck.l_type != F_UNLCK; )
    {
        lck.l_type = F_WRLCK;
        (void) fcntl(fd, F_GETLK, &lck);
        if (lck.l_type != F_UNLCK)
        {
            (void) printf("%5d %5d %c %8d %8d\n",
                          lck.l_sysid,
                          lck.l_pid,
                          (lck.l_type == F_WRLCK) ? 'W' : 'R',
                          lck.l_start,
                          lck.l_len);
            if (lck.l_len == 0)
                break; /* this lock goes to end of file, stop */
            lck.l_start += lck.l_len;
        }
    }
}
```

fcntl() with the F_GETLK command always returns correctly (that is, it will not sleep or fail) if the values passed to it as arguments are valid.

The **lockf()** function with the F_TEST command can also be used to test if there is a process blocking a lock. This function does not, however, return the information about where the lock actually is and which process owns the lock. Example 4-7 shows a code fragment that uses **lockf()** to test for a lock on a file.

Example 4-7 Testing for Contending Lock Using lockf()

```
/* find a blocked record. */
/* seek to beginning of file */
(void) lseek(fd, 0, 0L);
/* set the size of the test region to zero
 * to test until the end of the file address space.
 */
if (lockf(fd, F_TEST, 0L) < 0) {
    switch (errno) {
    case EACCES:
    case EAGAIN:
        (void) printf("file is locked by another process\n");
        break;
    case EBADF:
        /* bad argument passed to lockf */
        perror("lockf");
        break;
    default:
        (void) printf("lockf: unknown error <%d>\n", errno);
        break;
    }
}
```

When a process forks, the child receives a copy of the file descriptors that the parent has opened. The parent and child also share a common file pointer for each file. If the parent seeks to a point in the file, the child's file pointer is also set to that location. Similarly, when a share group of processes is created using **sproc()**, and the **sproc()** flag PR_SFDS is used to keep the open-file table synchronized for all processes (see the sproc(2) reference page), then there is a single file pointer for each file and it is shared by every process in the share group.

This feature has important implications when using record locking. The current value of the file pointer is used as the reference for the offset of the beginning of the lock, in **lockf()** at all times and in **fcntl()** when using an *l_whence* value of 1. Since there is no way to perform the sequence *lseek(); fcntl();* as an atomic operation, there is an obvious potential for race conditions—a lock might be set using a file pointer that was just changed by another process.

The solution is to have the child process close and reopen the file. This creates a distinct file descriptor for the use of that process. Another solution is to always use the **fcntl()** function for locking with an *l_whence* value of 0 or 2. This makes the locking function independent of the file pointer (processes might still contend for the use of the file pointer for other purposes such as direct-access input).

Deadlock Handling

A certain level of deadlock detection/avoidance is built into the record locking facility. This deadlock handling provides the same level of protection granted by the */usr/group* standard **lockf()** call. This deadlock detection is valid only for processes that are locking files or records on a single system.

Deadlocks can potentially occur only when the system is about to put a record locking system call to sleep. A search is made for constraint loops of processes that would cause the system call to sleep indefinitely. If such a situation is found, the locking system call fails and sets *errno* to the deadlock error number.

If a process wishes to avoid using the system's deadlock detection, it should set its locks using F_GETLK instead of F_GETLKW.

Selecting Advisory or Mandatory Locking

As discussed under "Advisory Locking" on page 149, file locking is usually an in-memory service of the IRIX kernel. The kernel keeps a table of locks that have been placed. Processes anywhere in the system update the table by calling fcntl() or lockf() to request locks. When all processes that use a file do this, and respect the results, file integrity can be maintained.

It is possible to extend file locking by making it mandatory on all processes, whether or not they were designed to be part of the cooperating group. Mandatory locking, as defined under "Mandatory Locking" on page 149, is enforced by the file I/O function calls. As a result, an independent process that calls **write()** to update a locked record will be blocked. The **write()** function tests for a contending lock internally. Of course, the same test is made by **write()** when called by a cooperating process, in which case it represents unnecessary overhead.

Mandatory locking is enforced on a file-by-file basis, triggered by a bit in the file inode that is set by *chmod* (see the chmod(1) and chmod(2) reference pages). In order to enforce mandatory locking on a particular file, turn on the set-group-ID bit along with a nonexecutable group permission ("2644" permissions, for example). This must be done before the file is opened; a change has no effect on a file that is already open.

Example 4-8 shows a fragment of code that sets mandatory lock mode on a given filename.

Example 4-8 Setting Mandatory Locking Permission Bits

```
#include <sys/types.h>
#include <sys/stat.h>
int setMandatoryLocking(char *filename)
{
   int mode;
   struct stat buf;
   if (stat(filename, &buf) < 0)
   {
      perror("stat(2)");
      return error;
   }
   mode = buf.st_mode;
   /* ensure group execute permission 0010 bit is off */
   mode &= ~(S_IEXEC>>3);
   /* turn on 'set group id bit' in mode */
   mode |= S_ISGID;
   if (chmod(filename, mode) < 0)
   {
      perror("chmod(2)");
      return error;
   }
   return 0;
}
```

When IRIX opens a file, it checks to see whether both of two conditions are true:

- Set group-ID bit is 1
- Group execute permission is 0

When both are true, the file is marked for mandatory locking, and each use of **creat()**, **open()**, **read()**, and **write()** tests for contending locks.

Some points to remember about mandatory locking:

- Mandatory locking does not protect against file truncation with the **truncate()** function (see the truncate(2) reference page), which does not look for locks on the truncated portion of the file.
- Mandatory locking protects only those portions of a file that are locked. Other portions of the file that are not locked may be accessed according to normal UNIX system file permissions.
- Advisory locking is more efficient because a record lock check does not have to be performed for every I/O request.

Record Locking Across Multiple Systems

Record locking, whether mandatory or advisory, is effective only within a single copy of the IRIX kernel. Locking is effective within a multiprocessor, because processes running in different CPUs of the multiprocessor share a single copy of the IRIX kernel.

Record locking is also effective on processes that execute in different systems connected by a network using NFS to mount the files. However, deadlock detection is not possible between processes in different systems.

If a process needs to maintain locks over several systems, it is suggested that the process avoid the sleep-when-blocked features of **fcntl()** or **lockf()** and that the process maintain its own deadlock detection. If the process uses the sleep-when-blocked feature, then a timeout mechanism should be provided by the process so that it does not hang waiting for a lock to be cleared.