7. FILE AND RECORD LOCKING

Introduction

Mandatory and advisory file and record locking both are available on current releases of the operating system. The intent of this capability to is provide a synchronization mechanism for programs accessing the same stores of data simultaneously. Such processing is characteristic of many multi-user 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 operating system users from businesses and campuses across the country.

Advisory file and record locking can be used to coordinate self-synchronizing processes. In mandatory locking, 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 against accessing the data out of sequence.

The remainder of this chapter describes how file and record locking capabilities can be used. 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 manual pages for the **fcntl**(2) system call, the **lockf**(3) library function, and **fcntl**(5) data structures and commands are referred to throughout this section. You should read them before continuing.

Terminology

Before discussing how record locking should be used, let us first define a few terms.

Record

A contiguous set of bytes in a file. The operating system does not impose any record structure on files. This may be done by the programs that use the files.

Cooperating Processes

Processes that work together in some well defined fashion to accomplish the tasks at hand. Processes that share files must request permission to access the files before using them. File access permissions must be carefully set to restrict non-cooperating processes from accessing those files. The term process will be used interchangeably with cooperating process to refer to a task obeying such protocols.

Read (Share) Locks

These are used to gain limited access to sections of files. When a read lock is in place on a record, other processes may also read lock that record, in whole or in part. No other process, however, may have or obtain a write lock on an overlapping section of the file. 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. This access method also permits many processes to read the given record. This might be necessary when searching a file, without the contention involved if a write or exclusive lock were to be used.

Write (Exclusive) Locks

These are used to gain complete control over sections of files. When a write lock is in place on a record, no other process may read or write lock that record, in whole or in part. 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

A form of record locking that does not interact with the I/O subsystem (i.e. **creat**(2), **open**(2), **read**(2), and **write**(2)). The control over records is accomplished by requiring an appropriate record lock request before I/O operations. If appropriate requests are always made by all processes accessing the file, then the accessibility of the file will be controlled by the interaction of these requests. Advisory locking depends on the individual processes to enforce the record locking protocol; it does not require an accessibility check at the time of each I/O request.

Mandatory Locking

A form of record locking that does interact with the I/O subsystem. Access to locked records is enforced by the **creat**(2), **open**(2), **read**(2), and **write**(2) system calls. If a record is locked, then access of that record by any other process is restricted according to the type of lock on the record. The control over records should still be performed explicitly by requesting an appropriate record lock before I/O operations, but an additional check is made by the system before each I/O operation to ensure the record locking protocol is being honored. Mandatory locking offers an extra synchronization check, but at the cost of some additional system overhead.

File Protection

There are access permissions for operating system files to control who may read, write, or execute such a file. These access permissions may only be set by the owner of the file or by the superuser. The permissions of the directory in which the file resides can also affect the final disposition of a file. Note that if the directory permissions allow anyone to write in it, then files within the directory may be removed, even if those files do not have read, write or execute permission for that user. Any information that is worth protecting, is worth protecting properly. If your application warrants the use of record locking, make sure that the permissions on your files and directories are set properly. A record lock, even a mandatory record lock, will only protect the portions of the files that are locked. Other parts of these files might be corrupted if proper precautions are not taken.

Only a known set of programs and/or administrators should be able to read or write a data base. This can be done easily by setting the set-group-ID bit (see **chmod**(1)) of the data base accessing programs. The files can then be accessed by a known set of programs that obey the record locking protocol. An example of such file protection, although record locking is not used, is the **mail**(1) command. In that command only the particular user and the **mail** command can read and write in the unread mail files.

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 done, then the file must be opened with at least read accessibility and likewise for write locks and write accessibility.

For our example, we will open our file for both read and write access:

```
#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 data base 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 for us to perform both locking and I/O functions. We then proceed with the task of setting a lock.

Setting a File Lock

There are several ways for us to set a lock on a file. In part, these methods depend on how the lock interacts with the rest of the program. There are also questions of performance as well as portability. Two methods will be given here, one using the **fcntl**(2) system call, the other using the */usr/group* standards compatible **lockf**(3) library function call.

Locking an entire file is just a special case of record locking. For both these methods the concept and the effect of the lock are the same. The file is locked starting at a byte offset of zero (0) until the end of the maximum file size. This point extends beyond any real end of the file so that no lock can be placed on this file beyond this point. To do this the value of the size of the lock is set to zero. The code using the **fcntl**(2) system call is as follows:

```
#include <fcntl.h>
#define MAX_TRY     10
int try;
struct flock lck;

try = 0;

/* set up the record locking structure, the address of which
* is passed to the fcntl system call.
*/
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 (fcntl(fd, F_SETLK, &lck) < 0) {
        if (errno == EAGAIN || errno == EACCES) {
                /* there might be other errors cases in which
                 * you might try again.
                 */
                if (++try < MAX_TRY) {
                        (void) sleep(2);
                        continue;
                }
                (void) fprintf(stderr,"File busy try again later!\n");
                return;
        }
        perror("fcntl");
        exit(2);
}
        .
        .
        .
```

This portion of code tries to lock a file. This is attempted several times until one of the following things happens:

- the file is locked
- an error occurs
- it gives up trying because MAX_TRY has been exceeded

To perform the same task using the **lockf**(3) function, the code is as follows:

```
#include <unistd.h>
#define MAX_TRY      10
int try;
try = 0;

/* make sure the file pointer
 * is at the beginning of the file.
 */
lseek(fd, 0L, 0);

/* Attempt locking MAX_TRY times before giving up.
 */
while (lockf(fd, F_TLOCK, 0L) < 0) {
        if (errno == EAGAIN || errno == EACCES) {
                /* there might be other errors cases in which
                 * you might try again.
                 */
                if (++try < MAX_TRY) {
                        sleep(2);
                        continue;
                }
                (void) fprintf(stderr,"File busy try again later!\n");
                return;
        }
        perror("lockf");
        exit(2);
}
        .
        .
        .
```

Note that the **lockf**(3) example appears to be simpler, but the **fcntl**(2) example exhibits additional flexibility. Using the **fcntl**(2) method, it is possible to set the type and start of the lock request simply by setting a few structure variables. **lockf**(3) merely sets write (exclusive) locks; an additional system call (**lseek**(2)) is required to specify the start of the lock.

Setting and Removing Record Locks

Locking a record is done the same way as locking a file except for the differing starting point and length of the lock. We will now try to solve an interesting and real problem. There are two records (these records may be in the same or different file) that must be updated simultaneously so that other processes get a consistent view of this information. (This type of problem comes up, for example, when updating the interrecord pointers in a doubly linked list.)

To do this you must decide the following questions:

- What do you want to lock?
- For multiple locks, what order do you want to lock and unlock the records?
- What do you do if you get 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 if one cannot obtain all the required locks. It is because of contention for these records that we have decided to use record locking in the first place. Different programs might:

- wait a certain amount of time, and try again
- abort the procedure and warn the user
- let the process sleep until signaled that the lock has been freed
- do some combination of the above

Let us now look at our example of inserting an entry into a doubly linked list. For the example, we will 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 changed or promoted to a write lock so that the record may be edited.

Promoting a lock (generally from read lock to write lock) is permitted if no other process is holding a read lock in the same section of the file. If there are processes with pending write locks that are sleeping on the same section of the file, the lock promotion succeeds and the other (sleeping) locks wait. Promoting (or demoting) a write lock to a read lock carries no restrictions. In either case, the lock is merely reset with the new lock type. Because the */usr/group* **lockf** function does not have read locks, lock promotion is not applicable to that call. An example of record locking with lock promotion 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 */
};

/* Lock promotion using fcntl(2)
 * When this routine is entered it is assumed that there are read
 * locks on "here" and "next".
 * 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 a -1.
 */
long
set3lock (this, here, next)
long this, here, next;
{
       struct flock lck;

       lck.l_type = F_WRLCK;       /* setting a write lock */
       lck.l_whence = 0;    /* offset l_start from beginning of file */
       lck.l_start = here;
       lck.l_len = sizeof(struct record);

       /* promote lock on "here" to write lock */
       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) {
              /* Lock on "this" failed;
               * demote lock on "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) {
              /* Lock on "next" failed;
               * demote lock on "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);  /* cannot set lock, try again or quit */
       }

       return (this);
}
```

7

The locks on these three records were all set to wait (sleep) if another process was blocking them from being set. This was done with the F_SETLKW command. 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.

Let us now look at a similar example using the **lockf** function. Since there are no read locks, all (write) locks will be referenced generically as locks.

```
/* Lock promotion using lockf(3)
 * When this routine is entered it is assumed that there are
 * no locks on "here" and "next".
 * If locks are obtained:
 *    Set a lock on "this".
 *    Return index to "this" record.
 * If any lock is not obtained:
 *    Remove all other locks.
 *    Return a -1.
 */

#include <unistd.h>

long
set3lock (this, here, next)
long this, here, next;

{

        /* lock "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) {
               /* Lock on "this" failed.
                * Clear lock on "here".
                */
               (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) {
                /* Lock on "next" failed.
                 * Clear lock on "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); /* cannot set lock, try again or quit */

        }

        return (this);
}
```

Locks are removed the way they are set, only the lock type is different (F_UNLCK or F_ULOCK). An unlock cannot be blocked by another process and will only affect locks that were placed by this process. The unlock only affects the section of the file defined in the previous example by **lck**. 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.

Getting Lock Information

One 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. A lock is set up as in the previous examples and the F_GETLK command is used 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. This information includes two pieces of data that have not been discussed yet, **l_pid** and **l_sysid**, that are only used by F_GETLK. (For systems that do not support a distributed architecture the value in **l_sysid** should be ignored.) These fields uniquely identify the process holding the lock.

If a lock passed to **fcntl** using the F_GETLK command would not be blocked by another process' lock, then the **l_type** field is changed to F_UNLCK and the remaining fields in the structure are unaffected. Let us use this capability to print all the segments locked by other processes. Note that if there are several read locks over the same segment only one of these will be found.

```
struct flock lck;

/* Find and print "write lock" blocked segments of this file. */
        (void) printf("sysid   pid type    start   length\n");
        lck.l_whence = 0;
        lck.l_start = 0L;
        lck.l_len = 0L;
        do {
                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 this lock goes to the end of the address
                         * space, no need to look further, so break out.
                         */
                        if (lck.l_len == 0)
                                break;
                        /* otherwise, look for new lock after the one
                         * just found.
                         */
                        lck.l_start += lck.l_len;
                }
        } while (lck.l_type != F_UNLCK);
```

fcntl with the F_GETLK command will always return 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. A routine using **lockf** to test for a lock on a file follows.

```
/* find a blocked record. */

/* seek to beginning of file */
(void) lseek(fd, 0, 0L);
/* set the size of the test region to zero (0)
 * 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 were to seek to a point in the file, the child's file pointer would also be at that location.

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, as described by **l_start**, when using a **l_whence** value of 1. If both the parent and child process set locks on the same file, there is a possibility that a lock will be set using a file pointer that was reset by the other process. This problem appears in the **lockf**(3) function call as well and is a result of the */usr/group* requirements for record locking. If forking is used in a record locking program, the child process should close and reopen the file if either locking method is used. This will result in the creation of a new and separate file pointer that can be manipulated without this problem occurring.

Another solution is to use the **fcntl** system call with a **l_whence** value of 0 or 2. This makes the locking function atomic, so that even processes sharing file pointers can be locked without difficulty.

Deadlock Handling

There is a certain level of deadlock detection/avoidance 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 only valid for processes that are locking files or records on a single system. Deadlocks can only potentially occur 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 will fail and set **errno** to the deadlock error number. If a process wishes to avoid the use of the systems deadlock detection it should set its locks using F_GETLK instead of F_GETLKW.

Selecting Advisory or Mandatory Locking

The use of mandatory locking is not recommended for reasons that will be made clear in a subsequent section. Whether locks are enforced by the I/O system calls is determined at the time the calls are made and the state of the permissions on the file (see **chmod**(2)). For locks to be under mandatory enforcement, the file must be a regular file with the set-group-ID bit on and the group execute permission off. If either condition fails, all record locks are advisory.

Mandatory enforcement can be assured by the following code:

```
#include <sys/types.h>
#include <sys/stat.h>

int mode;
struct stat buf;
              .
              .
              .
        if (stat(filename, &buf) < 0) {
               perror("program");
               exit (2);
        }
        /* get currently set mode */
        mode = buf.st_mode;
        /* remove group execute permission from mode */
        mode &= ~(S_IEXEC>>3);
        /* set 'set group id bit' in mode */
        mode |= S_ISGID;
        if (chmod(filename, mode) < 0) {
               perror("program");
               exit(2);
        }
              .
              .
              .
```

Files that are to be record locked should never have any type of execute permission set on them. This is because the operating system does not obey the record locking protocol when executing a file.

The **chmod**(1) command can also be easily used to set a file to have mandatory locking. This can be done with the command:

chmod +l *filename*

The **ls**(1) command was also changed to show this setting when you ask for the long listing format:

ls -l *filename*

causes the following to be printed.

```
-rw---l---   1 abc       other    1048576 Dec  3 11:44 filename
```

Caveat Emptor—Mandatory Locking

- Mandatory locking only protects those portions of a file that are locked. Other portions of the file that are not locked may be accessed according to normal operating system file permissions.
- If multiple reads or writes are necessary for an atomic transaction, the process should explicitly lock all such pieces before any I/O begins. Thus advisory enforcement is sufficient for all programs that act in this way.
- As stated earlier, arbitrary programs should not have unrestricted access permission to files that are important enough to record lock.
- Advisory locking is more efficient because a record lock check does not have to be performed for every I/O request.

Record Locking and Future Releases of the Operating System

Provisions have been made for file and record locking in a operating system environment. In such an environment the system on which the locking process resides may be remote from the system on which the file and record locks reside. In this way multiple processes on different systems may put locks on a single file that resides on one of these or yet another system. The record locks for a file reside on the system that maintains the file. It is also important to note that deadlock detection/avoidance is only determined by the record locks being held by and for a single system. Therefore, it is necessary that a process only hold record locks on a single system at any given time for the deadlock mechanism to be effective. If a process needs to maintain locks over several systems, it should avoid the **sleep-when-blocked** features of **fcntl** or **lockf** and that 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.

7