Advanced Programming in the UNIX Environment — *File Locking*

Hop Lee
hoplee@bupt.edu.cn

School of Information and Communication Engineering

Table of Contents

Understanding File Locking I

- Working with data records within a file of a multi-processing system can present the conflict challenge. If one process must update records while another process is doing the same, then some form of coordination is required to prevent chaos. One UNIX solution to this problem is the **file locking facility**.
- There are two basic forms of file locking. They are
  - Lock files
  - Locked regions
- The first form requires that a process create and open a lock file before it writes to the protected data file. If a process fails to create the lock file, then it sleeps for a while and tries again.

Understanding File Locking II

- The UNIX kernel also will permit a process to lock regions of a data file. A region consists of one or more bytes at a specified starting offset. The offset can extend beyond the end of the current file size.
- In this way, all processes agree to tell the kernel which regions of the file they are about to update. If a requested lock region is in conflict with presently granted locks on that file, the requesting process is put to sleep until the conflict is removed. When all processes obey this procedure, the integrity of the file is preserved.
- File locking under UNIX occurs under one of two lock enforcement models:
  1. Advisory locking—No enforcement
  2. Mandatory locking—Enforced locking

Understanding File Locking III

- The lock file and lock region methods just discussed require process cooperation to maintain the integrity of the data file. Cooperative locking methods are known as advisory locking. The UNIX kernel cannot enforce such cooperative methods. Consequently, when advisory locking methods are employed, processes that disobey the locking convention can corrupt the data file.
- Many UNIX kernels also support mandatory locking of files. When a process attempts to write to a region of a file that has enforced locking enabled, all other processes are prevented from interfering. Similarly, the writing process is blocked from executing until its conflicts with other processes have vanished.

The Lock File Technique I

- The lock file technique is a coarse-grained locking technique, since it implies that the entire file is locked. The technique is simple, however:
  1. Attempt to create and open the lock file.
  2. If step 1 fails, sleep for a while and repeat step 1.
  3. If step 1 succeeds, then you have successfully locked the resource.
- The success of this method depends on the fact that the creation and opening of the lock file must be atomic. This is easily accomplished with the UNIX `open` call, when the options `O_CREAT|O_EXCL` are used together.
- Example (`advio/lockfile2.c`).

The Lock File Technique II

- One of the things that you probably noticed about running the program `lockfile` from Listing 6 was that when locks were enabled, the test took much longer to run.
- The reason for this has to do with the need for the `Lock` function in line 20 to call upon `sleep`(3) when it was unsuccessful creating the lock file. While you could omit the `sleep` function call, this would be unwelcome on a multiuser system. The real problem lies in the fact that this is a polling method.

The Lock File Technique III

- Another limitation of the lock file method is that it is reliable only on a local file system. If your lock file is created on an NFS file system, NFS cannot guarantee that your `open` flags `O_CREAT|O_EXCL` will be respected (the operation may not be atomic). The operation must be atomic to be a reliable lock indicator.
- Additionally, the lock file technique can only operate at a file level. Successful locking with a lock file implies that the process has access to update the entire data file. All other processes must wait, even if they want to update different parts of the same file.

Using an Advisory Lock on the Entire File I

- An improvement over the file locking method was the creation of a UNIX kernel service that would allow a process to lock or unlock an entire file. Additionally, it was desirable to indicate when a file was being read or written. When a file is locked for reading, other processes can safely read the file concurrently. However, while the file remains read-locked, write-lock requests are blocked to ensure the safety of the data being read. Once all read locks are released, a write lock can be established on the file.
- This kernel service provides the following benefits to the programmer:
  1. Higher performance, since `sleep` is not called
  2. Finer lock granularity: read and write locks

Using an Advisory Lock on the Entire File II

- The file locking service is provided by the `flock` function on a BSD platform. This function provides the programmer with the following file locking capabilities:
  1. Shared locks—for reading
  2. Exclusive locks—for writing
- The function synopsis for the `flock` is as follows:

```
1 #include <sys/file.h>
2 int flock(int fd, int operation);
3 #define   LOCK_SH   0x01   /* shared file lock */
4 #define   LOCK_EX   0x02   /* exclusive file lock */
5 #define   LOCK_NB   0x04   /* don't block when locking */
6 #define   LOCK_UN   0x08   /* unlock file */
```

- The `flock` function has a few advantages over the lock file technique.
  1. No additional lock file is involved.

Using an Advisory Lock on the Entire File III

2. `sleep` is not called for retry attempts, providing improved performance.
3. Finer-grained locking allows locks to be shared or exclusive.
4. Allows locks to be held on NFS mounted file systems.

Record Locking I

- The BSD `flock` approach provides improved performance over the lock file but still suffers from the fact that it locks the entire file.
- Even better performance can be obtained when the regions of the file are locked instead of the entire file. System V provided the `lockf` function to accomplish this. Later, POSIX defined yet another application interface using the `fcntl` function.

```
1 #include <sys/lockf.h> /* AIX */
2 #include <unistd.h>
3 int lockf(int fd,int request,off_t size);
4 #define F_ULOCK   0   /* unlock a region */
5 #define F_LOCK    1   /* lock a region */
6 #define F_TLOCK   2   /* test and lock a region */
7 #define F_TEST    3   /* test region for lock */
```

Record Locking II

- The `lockf` function uses the current offset in the file open on *fd*. The *request* to lock a region of the file starts at this implied offset and includes *size* bytes. If *size* is negative, the region works backward from the current offset.
- The `lockf` function requires that the file descriptor *fd* must be open for write (`O_WRONLY`) or for read/write (`O_RDWR`). A file that is open only for reading cannot obtain a locked region with `lockf`.
- When a process provides multiple lock requests for overlapping regions that are already locked, the lock regions are merged.
- The POSIX method for locking files uses the `fcntl` application interface. The function synopsis for `fcntl` as it applies to file locking is as follows:

Record Locking III

```
1 #include <fcntl.h>
2 int fcntl(int fd, int cmd, struct flock *lck);
3 struct flock {
4   off_t  l_start;   /* starting offset */
5   off_t  l_len;     /* len = 0 means until end of file */
6   pid_t  l_pid;     /* lock owner (F_GETLK only) */
7   short  l_type;    /* F_RDLCK, F_WRLCK or F_UNLCK */
8   short  l_whence;  /* SEEK_SET, SEEK_CUR or SEEK_END */
9 };
```

- *cmd*:

  `F_GETLK` Get current lock information about *fd*.

  `F_SETLK` Set lock information about *fd*.

  `F_SETLKW` The same as `F_SETLK` except that the operation will block until the operation can succeed.

Record Locking IV

- The `fcntl` interface permits two different locks to be applied when the *cmd* argument is `F_SETLK` or `F_SETLKW`:
  1. Shared locks—`F_RDLCK`
  2. Write locks—`F_WRLCK`
- The following code segment shows how a region of an open file descriptor fd would be locked:

```
1  int fd;                      /* Open file descriptor */
2  struct flock lck;            /* Lock structure */
3  lck.l_start = 0;             /* Start at beginning of file */
4  lck.l_len = 0;               /* Lock entire file */
5  lck.l_type = F_RDLCK;        /* Shared lock */
6  lck.l_whence = SEEK_SET; /* Absolute offset */
7  if ( fcntl(fd, F_SETLKW,&lck) == -1 ) {
8      /* Error handling */
9      ......
10 }
```

Record Locking V

This example locks the entire file with a shared (read) lock on the file descriptor *fd*. Since `F_SETLKW` was used, this function call will block until it is successful.

Mandatory Locking I

- The discussions so far have covered only advisory locking. As long as all processes cooperate and use the locking conventions in agreement, the integrity of the file is maintained. However, if one or more processes do not obey the locking convention established, then updates to the file can result in file corruption and data loss.
- To enable mandatory locking on a file, the `setgid` bit is established for the file without the execution permission being given. More precisely, permission bit `S_ISGID` must be enabled, and `S_IXGRP` must be reset.
- With mandatory locking enabled, all read/write I/O calls will be affected as follows:

Mandatory Locking II

1. Any write call will be blocked if another process has a conflicting region locked with a shared or exclusive lock.
2. Any read attempt will be blocked if another process has a conflicting region locked with an exclusive lock.

- Mandatory locking provides the following benefits:
  1. All processes are forced to synchronize their access to the file, whether they explicitly lock regions or not.
  2. The application does not have to issue lock and unlock requests for simple updates.
- Mandatory locking suffers from the following disadvantages:
  1. Additional UNIX kernel overhead is required to check locks for every read and write system call on the file.
  2. A malicious process can hold an exclusive lock on the file indefinitely, preventing any reads or writes to the file.

Mandatory Locking III

3. A malicious process can hold a shared lock on a mandatory lock to deny any process to write to the file.
4. Mandatory locks may not be supported on NFS mounted file systems.
5. Mandatory locks are not supported on all UNIX platforms.

Summary

- This chapter covered the different forms of locking, from the primitive file-based locks to the more advanced region locks. The next chapter will look at the basic UNIX functions that allow your programs to manage files and to obtain property information about them.

The End

The End of Chapter 11.