Technical Note TN2037

Exclusive File Access in Mac OS X

CONTENTS

This document discusses the issues surrounding obtaining exclusive file access in current versions of Mac OS X and how it differs from classic Mac OS. Exclusive file access is an issue which affects all Mac OS X developers, Carbon, Cocoa, Java, and BSD.

[May 01 2002]

Overview

Opening a file from classic Mac OS (pre Mac OS X) with `fsWrPerm`, `fsRdWrPerm`, or the default `fsCurPerm`, meant that any other application trying to open that same file with write access would not be able to do so. Usually an `fsRdWrPerm` error would be returned when other attempts were made to open the file for write access, though attempts to open such a file for read only access would succeed. This default behavior allows for one "writer" and multiple "readers" of the file.

Mac OS X's BSD subsystem does not enforce file read/write privileges in the same way as classic Mac OS. Opening a file for writing does not ensure other processes can not write to the same file. The default behavior of BSD allows for multiple "writers" to a single file. In the current implementation, all of the File Manager calls in Mac OS X call through to the underlying BSD file system. As a result, opening a file via `PBHOpenDF, PBHOpenRF, PBHOpen, PBOpenFork, FSOpenFork, HOpen,` etc. on a local volume and passing in a permissions value of `fsCurPerm`, `fsWrPerm`, or `fsRdWrPerm` does not guarantee exclusive file access on Mac OS X. On Mac OS X subsequent Open calls to open a file with write permission may succeed without error. Similarly the `PBLockRange()` routines may not actually guarentee byte ranges cannot be modified by other processes. Because these routines may return without error, you should check the availability of exclusive file access before making any assumptions about the underlying file access. If the 'supports advisory locks' feature is not available your application will not know if the file is already in use by another application.

AppleShare servers and Personal File Sharing on Mac OS X do enforce exclusive file access and range locking for volumes accessed over the network. However, this functionality is only available when accessing files over a networked file sharing connection and is not available to applications running on the server itself.

Guidelines for working with non-exclusivity

We realize that many applications rely on the behavior of the classic Mac OS File Manager to prevent multiple applications from writing to the same file (or to control write access through byte range locking). Since that behavior is not implemented in all versions of Mac OS X, some common workarounds that you may wish to use in your code are described below.
BSD was designed without exclusive locks in order to prevent denial of service attacks in which one process opens a file with an exclusive lock which may be required by another process, effectively blocking the other process.

Checking Availability of Exclusive File Access

Mac OS X will enforce exclusive file access, i.e. one writer and many readers of a file, through it's application frameworks, Carbon, Cocoa, and Java, by enforcing BSD advisory locks as though they are exclusive. The 'supports advisory locks' feature is defined if both the OS and file system for the volume in question support advisory locks. In this case, the default behavior of the application frameworks is to open files with exclusive access when opened as writable. Applications built on these frameworks automatically get this functionallity and do not need to be modified. When the conditions are met to support exclusive file access, `PBLockRange` will also call down through to the BSD advisory locks. Since `PBLockRange` will be based on BSD advisory locks at this point, range locks can be applied to local files as well as those on file servers.

Since not all versions of Carbon on Mac OS X support exclusive file access nor do all file systems support BSD advisory locks, you should check a couple things before making assumptions about the underlying file access behavior. You should only assume these features are available if the gestalt bit, `gestaltFSSupportsExclusiveLocks`, as well as the `GetVolParms` bit, `bSupportsExclusiveLocks`, are both set. For instance, the Carbon Framework File Manager routines support advisory locks by default when `SupportsExclusiveFileAccess` returns true.

```
#ifndef gestaltFSSupportsExclusiveLocks
   #define    gestaltFSSupportsExclusiveLocks    15
   #define    bSupportsExclusiveLocks            18
#endif

Boolean    SupportsExclusiveFileAccess( short vRefNum )
{
   OSErr                    err;
   GetVolParmsInfoBuffer    volParmsBuffer;
   HParamBlockRec           hPB;
   long                     response;
   Boolean                  exclusiveAccess    = false;

   err = Gestalt( gestaltSystemVersion, &response );
   if ( (err == noErr) && (response < 0x01000) )
   {
      err = Gestalt( gestaltMacOSCompatibilityBoxAttr, &response );
      if ( (err != noErr)
      || ((response & (1 << gestaltMacOSCompatibilityBoxPresent)) == 0) )
         return( true );        //    Running on Mac OS 9, not in Classic
   }

   err = Gestalt( gestaltFSAttr, &response );
   if ( (err == noErr)
        && (response & (1L << gestaltFSSupportsExclusiveLocks)) )
   {
      hPB.ioParam.ioVRefNum     = vRefNum;
      hPB.ioParam.ioNamePtr     = NULL;
      hPB.ioParam.ioBuffer      = (Ptr) &volParmsBuffer;
      hPB.ioParam.ioReqCount    = sizeof( volParmsBuffer );
      err = PBHGetVolParmsSync( &hPB );
      if ( err == noErr )
         exclusiveAccess =
                (volParmsBuffer.vMExtendedAttributes
                & (1L << bSupportsExclusiveLocks)) != 0;
   }

   return( exclusiveAccess );
}
```

To check if a volume supports byte range locking via `PBLockRange` you should check the `bHasOpenDeny` bit returned from GetVolParms. See Technical Note FL37 for more information about `PBLockRange` details.

```
      hPB.ioParam.ioVRefNum     = vRefNum;
      hPB.ioParam.ioNamePtr     = NULL;
      hPB.ioParam.ioBuffer      = (Ptr) &volParmsBuffer;
      hPB.ioParam.ioReqCount    = sizeof( volParmsBuffer );
      err = PBHGetVolParmsSync( &hPB );
      if ( err == noErr )
         supportsByteRangeLocking =
             (volParmsBuffer.vMAttrib & (1L << bHasOpenDeny)) != 0;
```

Back to top

Common Workarounds

The following two techniques are frequently used to work around this issue on platforms that do not enforce exclusive file access:

Lockfiles

A common approach used by many developers is to create a "lockfile" in the same directory as the file being opened. Whenever opening a file, "foo", for instance, with write access you first try to create a lockfile, "foo.lock", in the same location. If the file creation fails because the file already exists, you assume "foo" is already open by another application. Upon closing "foo" the application is also responsible for deleting "foo.lock". A strength of this technique is it only makes one assumption about the underlying file system: the file creation operation is atomic. The obvious weakness is that since there is no OS support for this method, each application is responsible for implementing its own lock file mechanism, and there are no agreed upon standards or conventions for the naming of lock files

The included sample, GrabBag, implements a variation of this "lockfile" technique. Not only does it create the lockfile, it stores its ProcessSerialNumber in the file. Before opening a file, the code checks if a lockfile exists, and if it does, verifies the PSN in the file is valid. This helps guard against files being left in a locked state in the event of an application crash.

> **IMPORTANT:**
> Workarounds should be requalified with a system supporting the 'supports advisory locks' feature. It will be announced when it becomes available.

Edit a Copy

Another work around relies on operating on a unique copy of the file. When a file is opened for editing, a duplicate of the file is created in the `/tmp` directory with a unique name, and opened. When the user tries to save the document, the modification date of the original is matched against the date cached during the open of the file. If it has changed, you know the file was modified.

Back to top

Mac OS X Solutions

BSD Advisory Locking

Although Mac OS X's BSD subsystem does not implement provisions for exclusive write access, (i.e. mandatory locks), it does provide advisory locks. An advisory lock is a voluntary locking mechanism in which the underlying file system maintains a linked list of record locks. As long as your application and other applications respect the locks, only one application at a time will have write access to a particular file. Since these locks are voluntary it is the choice/responsibility of the application developer to respect or ignore advisory locks. If you would like to use advisory locks, this can be done by following the instructions later in this document. By accessing files through the application frameworks (Carbon, Cocoa, Java), in versions of the OS supporting the advisory locks feature in frameworks, this will be provided automatically if you use the framework's file access methods.

> **IMPORTANT:**
> All applications should respect and use advisory locks.

Applications that call BSD file I/O functions directly will not gain this behavior for free, and therefore should be revised to set and respect advisory locks by specifying the appropriate flags when opening a file.

i.e. You should evaluate changing calls from:

```
    fd = open( "./foo", O_RDWR );
```

to:

```
    fd = open( "./foo", O_RDWR + O_EXLOCK + O_NONBLOCK );
```

- Where, **`O_EXLOCK`** means Atomically obtain an exclusive lock, and **`O_NONBLOCK`** means Do not block on open or for data to become available or Do not wait for the device or file to be ready or available.

Back to top

Implementing Advisory Locking

Anywhere you are calling the System.framework version of `open(2)` with write access, you should modify the parameters to include the `"O_EXLOCK + O_NONBLOCK"` flags, and handle errors being returned, where they may have succeeded in the past. The `open(2)` call will then fail if the file has already been opened for exclusive access by another process.

Advisory locks are associated with a process and a file. This has two implications:

- When a process terminates all its locks are released.
- Whenever a descriptor is closed, any locks on the file referenced by that descriptor are released.

Implementing Byte Range Locking

BSD also provides advisory byte range locking support through the `fcntl()` function. By using advisory locking, your application will be able to work in a cooperative manner with Carbon, Classic, and other applications in the future. In these circumstances, files should be opened with the `O_EXLOCK` set and then ranges locked through the `fcntl()` call.

Stevens' "Advanced Programming in the Unix Environment" (page 367) describes some techniques for using the Unix service `fcntl()` to lock portions of a file for reading and writing. (Stevens, 1999, p. 367)

WARNING:
A file lock request which is blocked can be interrupted by a signal. In this case the lock operation returns `EINTR`. Thus you may think you got a lock when you really did not. A solution is to block signals when locking. Another solution is to test the value returned by the lock operation and relock if the value is `EINTR`. Another solution, which we adopt here, is to do nothing about it.

Record Locking is the term normally used to describe the ability of a process to prevent other processes from modifying a region of a file while the first process is reading or modifying that portion of the file. BSD provides access to its record locking mechanism through the `fcntl` function:

```
    #include <sys/types.h>
    #include <unistd.h>
    #include <fcntl.h>

    /*
     * Returns:
     *    -1 on error
     */
    int fcntl(int filedes, int cmd, ... /* struct flock *flockptr */ );
```

We'll start with the third argument (`flockptr`), which points to a `flock` structure:

```
struct flock {
    short l_type;     /* F_RDLCK (shared read lock), or
                       * F_WRLCK (shared write lock), or
                       * F_UNLCK (unlocking a region)
                       */
    off_t l_start;    /* offset in bytes, relative to l_whence */
    short l_whence;   /* SEEK_SET: file's offset is set to
                       *           l_start bytes from beginning of file
                       * SEEK_CUR: file's offset is set to its current
                       *           value plus the l_start (which can
                       *           be + or -)
                       * SEEK_END: file's offset is set to the size of
                       *           the file plus the l_start (which can
                       *           be + or -)
                       */
    off_t l_len;      /* length of region, in bytes
                       * special case: if (l_len == 0), it means that
                       * the lock extends to the largest possible
                       * offset of the file. This allows us to lock a
                       * region starting anywhere in the file, up
                       * through and including any data that is
                       * appended to the file
                       */
    pid_t l_pid;      /* returned when cmd = F_GETLK */
}
```

This structure describes:

- The type of lock desired (i.e. read lock, write lock, unlock)
- The starting byte offset of the region being locked or unlocked (`l_start and l_whence`)
- The size of the region (`l_len`)

To lock an entire file, set `l_start` and `l_whence` to point to the beginning of the file (i.e. `l_start= 0`, `l_whence= SEEK_SET`), and specify a length (`l_len`) of `0`.

Any number of processes can have a shared read lock on a given byte, but only one process can have an exclusive write lock on a given byte. To obtain a read lock the descriptor must be open for reading, and the region cannot have an exclusive write lock. To obtain a write lock the descriptor must be open for writing, and the region cannot have an exclusive write lock nor any read locks.

Now, we will describe the second parameter (`cmd`) for `fcntl`. The possible commands and what they mean are described in the following table:

Command	Meaning
F_GETLK	Determine if the lock described by `flockptr` is blocked by some other lock. If a lock exists that would prevent ours from being created, the information on that existing lock overwrites the information pointed to by `flockptr`. If no lock exists that would prevent ours from being created, the structure pointed to by `flockptr` is left unchanged except for the `l_type member`, which is set to `F_UNLCK`.
F_SETLK	Set the lock described by `flockptr`. If we are unable to obtain a lock (because of previous locks already granted for the region) then `fcntl` returns `-1` and `errno` is set to either `EACCES` or `EAGAIN`.
F_SETLKW	This command is a blocking version of `F_SETLK` (the W in the command means "wait"). If the requested read lock or write lock cannot be granted because another process currently has some part of the requested region locked, the calling process is put to sleep. This sleep is interrupted is a signal is caught.

Be aware that testing for a lock with `F_GETLK` and then trying to obtain that lock with `F_SETLK` or `F_SETLKW` is not an atomic operation. We have no guarantee that between the two `fcntrl` calls some other process won't come in and obtain the same lock.

To save ourselves the trouble of allocating a `flock` structure and filling in all the elements each time, Stevens defines the function `lock_reg` and a number of macros that call it. Notice that the macros shorten the number of parameters by two, and save us from having to remember the `F_*` constants mentioned above.

```
#define read_lock(fd, offset, whence, len)    \
         lock_reg  (fd, F_SETLK,  F_RDLCK, offset, whence, len)

#define readw_lock(fd, offset, whence, len)   \
         lock_reg  (fd, F_SETLKW, F_RDLCK, offset, whence, len)

#define write_lock(fd, offset, whence, len)   \
         lock_reg  (fd, F_SETLK,  F_WRLCK, offset, whence, len)

#define writew_lock(fd, offset, whence, len)  \
         lock_reg  (fd, F_SETLKW, F_WRLCK, offset, whence, len)

#define un_lock(fd, offset, whence, len)      \
         lock_reg  (fd, F_SETLK,  F_UNLCK, offset, whence, len)


pid_t    lock_test(int, int , off_t , int , off_t );

#define    is_readlock(fd, offset, whence, len) \
            lock_test(fd, F_RDLCK, offset, whence, len)
#define    is_writelock(fd, offset, whence, len) \
            lock_test(fd, F_WRLCK, offset, whence, len)


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

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

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


pid_t    lock_test(int fd, int type, off_t offset, int whence, off_t len)
{
  struct flock lock;
  lock.l_type = type;     /* F_RDLCK or F_WRLCK */
  lock.l_start = offset;  /* byte offset relative to l_whence */
  lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
  lock.l_len = len;       /* #bytes (0 means to EOF) */
  if (fcntl(fd,F_GETLK,&lock) < 0){
    perror("fcntl"); exit(1);}
  if (lock.l_type == F_UNLCK)
    return (0);        /* false, region is not locked by another process */
  return (lock.l_pid); /* true, return pid of lock owner */
}
```

There are three important rules regarding automatic inheritance and release of record locks:

- Locks are associated with a process and a file. When a process terminates, all its locks are released. Whenever a descriptor is closed, any locks on the file referenced by that descriptor for that process are released.
- Locks are never inherited by the child across a fork (otherwise we could end up with two processes sharing a write lock)
- Locks may be inherited by a new program across an `exec`. This is not required by BSD and is therefore machine dependent

Back to top

References

Stevens, Richard W. (1999). Advanced Programming in the Unix Environment
Massachusetts: Addison Wesley Longman, Inc.
ISBN: 0201563177

Back to top

Downloadables

	Acrobat version of this Note (68K)	Download
	GrabBag, a Carbon application demonstrating the use of "lockfiles" (200 K)	Download

Back to top