mandatory-locking.txt

Mandatory File Locking For The Linux Operating System

Andy Walker <andy@lysaker.kvaerner.no>

15 April 1996
(Updated September 2007)

0. Why you should avoid mandatory locking

The Linux implementation is prey to a number of difficult-to-fix race conditions which in practice make it not dependable:

- The write system call checks for a mandatory lock only once at its start. It is therefore possible for a lock request to be granted after this check but before the data is modified. A process may then see file data change even while a mandatory lock was held.
- Similarly, an exclusive lock may be granted on a file after the kernel has decided to proceed with a read, but before the read has actually completed, and the reading process may see the file data in a state which should not have been visible to it.
- Similar races make the claimed mutual exclusion between lock and mmap similarly unreliable.

1. What is mandatory locking?

Mandatory locking is kernel enforced file locking, as opposed to the more usual cooperative file locking used to guarantee sequential access to files among processes. File locks are applied using the flock() and fcntl() system calls (and the lockf() library routine which is a wrapper around fcntl().) It is normally a process' responsibility to check for locks on a file it wishes to update, before applying its own lock, updating the file and unlocking it again. The most commonly used example of this (and in the case of sendmail, the most troublesome) is access to a user's mailbox. The mail user agent and the mail transfer agent must guard against updating the mailbox at the same time, and prevent reading the mailbox while it is being updated.

In a perfect world all processes would use and honour a cooperative, or "advisory" locking scheme. However, the world isn't perfect, and there's a lot of poorly written code out there.

In trying to address this problem, the designers of System V UNIX came up with a "mandatory" locking scheme, whereby the operating system kernel would block attempts by a process to write to a file that another process holds a "read" -or- "shared" lock on, and block attempts to both read and write to a file that a process holds a "write " -or- "exclusive" lock on.

The System V mandatory locking scheme was intended to have as little impact as possible on existing user code. The scheme is based on marking individual files as candidates for mandatory locking, and using the existing fcntl()/lockf() interface for applying locks just as if they were normal, advisory locks.

Note 1: In saying "file" in the paragraphs above I am actually not telling the whole truth. System V locking is based on fcntl(). The granularity of fcntl() is such that it allows the locking of byte ranges in files, in addition to entire files, so the mandatory locking rules also have byte level granularity.

Note 2: POSIX.1 does not specify any scheme for mandatory locking, despite borrowing the fcntl() locking scheme from System V. The mandatory locking scheme is defined by the System V Interface Definition (SVID) Version 3.

2. Marking a file for mandatory locking

A file is marked as a candidate for mandatory locking by setting the group-id bit in its file mode but removing the group-execute bit. This is an otherwise meaningless combination, and was chosen by the System V implementors so as not to break existing user programs.

Note that the group-id bit is usually automatically cleared by the kernel when a setgid file is written to. This is a security measure. The kernel has been modified to recognize the special case of a mandatory lock candidate and to refrain from clearing this bit. Similarly the kernel has been modified not to run mandatory lock candidates with setgid privileges.

3. Available implementations

I have considered the implementations of mandatory locking available with SunOS 4.1.x, Solaris 2.x and HP-UX 9.x.

Generally I have tried to make the most sense out of the behaviour exhibited by these three reference systems. There are many anomalies.

All the reference systems reject all calls to open() for a file on which another process has outstanding mandatory locks. This is in direct contravention of SVID 3, which states that only calls to open() with the O_TRUNC flag set should be rejected. The Linux implementation follows the SVID definition, which is the "Right Thing", since only calls with O_TRUNC can modify the contents of the file.

HP-UX even disallows open() with O_TRUNC for a file with advisory locks, not just mandatory locks. That would appear to contravene POSIX.1.

mmap() is another interesting case. All the operating systems mentioned prevent mandatory locks from being applied to an mmap()'ed file, but HP-UX also disallows advisory locks for such a file. SVID actually specifies the paranoid HP-UX behaviour.

In my opinion only MAP_SHARED mappings should be immune from locking, and then only from mandatory locks - that is what is currently implemented.

SunOS is so hopeless that it doesn't even honour the O_NONBLOCK flag for mandatory locks, so reads and writes to locked files always block when they should return EAGAIN.

I'm afraid that this is such an esoteric area that the semantics described below are just as valid as any others, so long as the main points seem to agree.

4. Semantics

1. Mandatory locks can only be applied via the fcntl()/lockf() locking interface - in other words the System V/POSIX interface. BSD style locks using flock() never result in a mandatory lock.

2. If a process has locked a region of a file with a mandatory read lock, then other processes are permitted to read from that region. If any of these processes attempts to write to the region it will block until the lock is released, unless the process has opened the file with the O_NONBLOCK flag in which case the system call will return immediately with the error status EAGAIN.

3. If a process has locked a region of a file with a mandatory write lock, all attempts to read or write to that region block until the lock is released, unless a process has opened the file with the O_NONBLOCK flag in which case the system call will return immediately with the error status EAGAIN.

4. Calls to open() with O_TRUNC, or to creat(), on a existing file that has any mandatory locks owned by other processes will be rejected with the error status EAGAIN.

5. Attempts to apply a mandatory lock to a file that is memory mapped and shared (via mmap() with MAP_SHARED) will be rejected with the error status EAGAIN.

6. Attempts to create a shared memory map of a file (via mmap() with MAP_SHARED) that has any mandatory locks in effect will be rejected with the error status EAGAIN.

5. Which system calls are affected?

Those which modify a file's contents, not just the inode. That gives read(), write(), readv(), writev(), open(), creat(), mmap(), truncate() and ftruncate(). truncate() and ftruncate() are considered to be "write" actions for the purposes of mandatory locking.

The affected region is usually defined as stretching from the current position for the total number of bytes read or written. For the truncate calls it is defined as the bytes of a file removed or added (we must also consider bytes added, as a lock can specify just "the whole file", rather than a specific range of bytes.)

Note 3: I may have overlooked some system calls that need mandatory lock checking in my eagerness to get this code out the door. Please let me know, or better still fix the system calls yourself and submit a patch to me or Linus.

6. Warning!

Not even root can override a mandatory lock, so runaway processes can wreak havoc if they lock crucial files. The way around it is to change the file

permissions (remove the setgid bit) before trying to read or write to it.
Of course, that might be a bit tricky if the system is hung :-(