File locking in Linux

29 Jul 2016

`linux` `posix` `ipc`

Table of contents

Introduction

File locking is a mutual-exclusion mechanism for files. Linux supports two major kinds of file locks:

- advisory locks
- mandatory locks

Below we discuss all lock types available in POSIX and Linux and provide usage examples.

Advisory locking

Traditionally, locks are advisory in Unix. They work only when a process explicitly acquires and releases locks, and are ignored if a process is not aware of locks.

There are several types of advisory locks available in Linux:

- BSD locks (flock)
- POSIX record locks (fcntl, lockf)
- Open file description locks (fcntl)

All locks except the `lockf` function are reader-writer locks, i.e. support exclusive and shared modes.

Note that `flockfile` and friends have nothing to do with the file locks. They manage internal mutex of the `FILE` object from stdio.

Reference:

- File Locks, GNU libc manual
- Open File Description Locks, GNU libc manual
- File-private POSIX locks, an LWN article about the predecessor of open file description locks

Common features

The following features are common for locks of all types:

- All locks support blocking and non-blocking operations.
- Locks are allowed only on files, but not directories.
- Locks are automatically removed when the process exits or terminates. It's guaranteed that if a lock is acquired, the process acquiring the lock is still alive.

Differing features

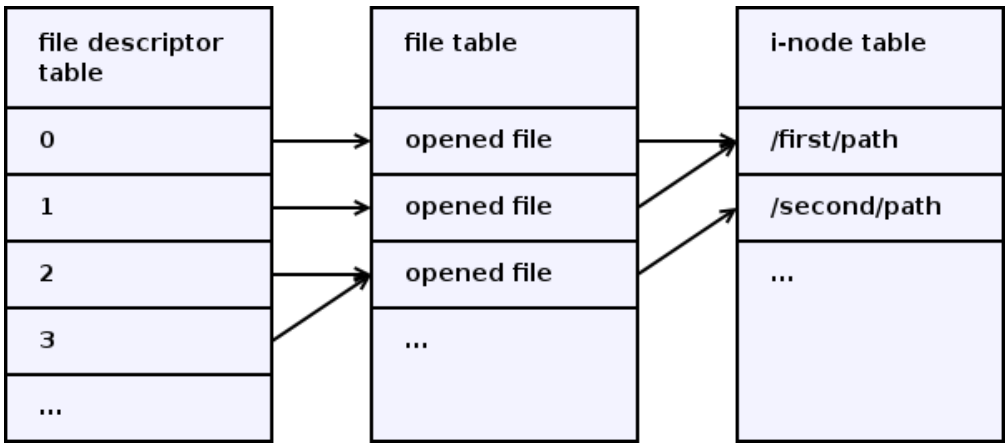
This table summarizes the difference between the lock types. A more detailed description and usage examples are provided below.

	BSD locks	lockf function	POSIX record locks	Open file description locks
Portability	widely available	POSIX (XSI)	POSIX (base standard)	Linux 3.15+
Associated with	File object	[i-node, pid] pair	[i-node, pid] pair	File object
Applying to byte range	no	yes	yes	yes

Support exclusive and shared modes	yes	no	yes	yes
Atomic mode switch	no	-	yes	yes
Works on NFS (Linux)	Linux 2.6.12+	yes	yes	yes

File descriptors and i-nodes

A *file descriptor* is an index in the per-process file descriptor table (in the left of the picture). Each file descriptor table entry contains a reference to a *file object*, stored in the file table (in the middle of the picture). Each file object contains a reference to an i-node, stored in the i-node table (in the right of the picture).



A file descriptor is just a number that is used to refer a file object from the user space. A file object represents an opened file. It contains things likes current read/write offset, non-blocking flag and another non-persistent state. An i-node represents a filesystem object. It contains things like file meta-information (e.g. owner and permissions) and references to data blocks.

File descriptors created by several `open()` calls for the same file path point to different file objects, but these file objects point to the same i-node. Duplicated file descriptors created by `dup2()` or `fork()` point to the same file object.

A BSD lock and an Open file description lock is associated with a file object, while a POSIX record lock is associated with an `[i-node, pid]` pair. We'll discuss it below.

BSD locks (flock)

The simplest and most common file locks are provided by `flock(2)`.

Features:

- not specified in POSIX, but widely available on various Unix systems
- always lock the entire file
- associated with a file object
- do not guarantee atomic switch between the locking modes (exclusive and shared)
- up to Linux 2.6.11, didn't work on NFS; since Linux 2.6.12, flock() locks on NFS are emulated using fcntl() POSIX record byte-range locks on the entire file (unless the emulation is

disabled in the NFS mount options)

The lock acquisition is associated with a file object, i.e.:

- duplicated file descriptors, e.g. created using `dup2` or `fork`, share the lock acquisition;
- independent file descriptors, e.g. created using two `open` calls (even for the same file), don't share the lock acquisition;

This means that with BSD locks, threads or processes can't be synchronized on the same or duplicated file descriptor, but nevertheless, both can be synchronized on independent file descriptors.

`flock()` doesn't guarantee atomic mode switch. From the man page:

> Converting a lock (shared to exclusive, or vice versa) is not guaranteed to be atomic: the existing lock is first removed, and then a new lock is established. Between these two steps, a pending lock request by another process may be granted, with the result that the conversion either blocks, or fails if LOCK_NB was specified. (This is the original BSD behaviour, and occurs on many other implementations.)

This problem is solved by POSIX record locks and Open file description locks.

Usage example:

```c
#include <sys/file.h>

// acquire shared lock
if (flock(fd, LOCK_SH) == -1) {
    exit(1);
}

// non-atomically upgrade to exclusive lock
// do it in non-blocking mode, i.e. fail if can't upgrade immediately
if (flock(fd, LOCK_EX | LOCK_NB) == -1) {
    exit(1);
}

// release lock
// lock is also released automatically when close() is called or process exits
if (flock(fd, LOCK_UN) == -1) {
    exit(1);
}
```

POSIX record locks (fcntl)

POSIX record locks, also known as process-associated locks, are provided by `fcntl(2)`, see "Advisory record locking" section in the man page.

Features:

- specified in POSIX (base standard)
- can be applied to a byte range
- associated with an `[i-node, pid]` pair instead of a file object
- guarantee atomic switch between the locking modes (exclusive and shared)
- work on NFS (on Linux)

The lock acquisition is associated with an `[i-node, pid]` pair, i.e.:

- file descriptors opened by the same process for the same file share the lock acquisition (even independent file descriptors, e.g. created using two `open` calls);
- file descriptors opened by different processes don't share the lock acquisition;

This means that with POSIX record locks, it is possible to synchronize processes, but not threads. All threads belonging to the same process always share the lock acquisition of a file, which means that:

- the lock acquired through some file descriptor by some thread may be released through another file descriptor by another thread;
- when any thread calls `close` on any descriptor referring to given file, the lock is released for the whole process, even if there are other opened descriptors referring to this file.

This problem is solved by Open file description locks.

Usage example:

```
#include <fcntl.h>

struct flock fl;
memset(&fl, 0, sizeof(fl));

// lock in shared mode
fl.l_type = F_RDLCK;

// lock entire file
fl.l_whence = SEEK_SET; // offset base is start of the file
fl.l_start = 0;         // starting offset is zero
fl.l_len = 0;           // len is zero, which is a special value representing end
                        // of file (no matter how large the file grows in future)

fl.l_pid = 0; // F_SETLK(W) ignores it; F_OFD_SETLK(W) requires it to be zero

// F_SETLKW specifies blocking mode
if (fcntl(fd, F_SETLKW, &fl) == -1) {
    exit(1);
}

// atomically upgrade shared lock to exclusive lock, but only
```

```
// for bytes in range [10; 15)
//
// after this call, the process will hold three lock regions:
//  [0; 10)        - shared lock
//  [10; 15)       - exclusive lock
//  [15; SEEK_END) - shared lock
fl.l_type = F_WRLCK;
fl.l_start = 10;
fl.l_len = 5;

// F_SETLKW specifies non-blocking mode
if (fcntl(fd, F_SETLK, &fl) == -1) {
    exit(1);
}

// release lock for bytes in range [10; 15)
fl.l_type = F_UNLCK;

if (fcntl(fd, F_SETLK, &fl) == -1) {
    exit(1);
}

// close file and release locks for all regions
// remember that locks are released when process calls close()
// on any descriptor for a lock file
close(fd);
```

lockf function

`lockf(3)` function is a simplified version of POSIX record locks.

Features:

- specified in POSIX (XSI)
- can be applied to a byte range (optionally automatically expanding when data is appended in future)
- associated with an `[i-node, pid]` pair instead of a file object
- supports only exclusive locks
- works on NFS (on Linux)

Since `lockf` locks are associated with an `[i-node, pid]` pair, they have the same problems as POSIX record locks described above.

The interaction between `lockf` and other types of locks is not specified by POSIX. On Linux, `lockf` is just a wrapper for POSIX record locks.

Usage example:

```c
#include <unistd.h>

// set current position to byte 10
if (lseek(fd, 10, SEEK_SET) == -1) {
    exit(1);
}

// acquire exclusive lock for bytes in range [10; 15)
// F_LOCK specifies blocking mode
if (lockf(fd, F_LOCK, 5) == -1) {
    exit(1);
}

// release lock for bytes in range [10; 15)
if (lockf(fd, F_ULOCK, 5) == -1) {
    exit(1);
}
```

Open file description locks (fcntl)

Open file description locks are Linux-specific and combine advantages of the BSD locks and POSIX record locks. They are provided by `fcntl(2)`, see "Open file description locks (non-POSIX)" section in the man page.

Features:

- Linux-specific, not specified in POSIX
- can be applied to a byte range
- associated with a file object
- guarantee atomic switch between the locking modes (exclusive and shared)
- work on NFS (on Linux)

Thus, Open file description locks combine advantages of BSD locks and POSIX record locks: they provide both atomic switch between the locking modes, and the ability to synchronize both threads and processes.

These locks are available since the 3.15 kernel.

The API is the same as for POSIX record locks (see above). It uses `struct flock` too. The only difference is in `fcntl` command names:

- `F_OFD_SETLK` instead of `F_SETLK`
- `F_OFD_SETLKW` instead of `F_SETLKW`
- `F_OFD_GETLK` instead of `F_GETLK`

Emulating Open file description locks

What do we have for multithreading and atomicity so far?

- BSD locks allow thread synchronization but don't allow atomic mode switch.
- POSIX record locks don't allow thread synchronization but allow atomic mode switch.
- Open file description locks allow both but are available only on recent Linux kernels.

If you need both features but can't use Open file description locks (e.g. you're using some embedded system with an outdated Linux kernel), you can *emulate* them on top of the POSIX record locks.

Here is one possible approach:

- Implement your own API for file locks. Ensure that all threads always use this API instead of using `fcntl()` directly. Ensure that threads never open and close lock-files directly.
- In the API, implement a process-wide singleton (shared by all threads) holding all currently acquired locks.
- Associate two additional objects with every acquired lock:
- a counter
- an RW-mutex, e.g. `pthread_rwlock`

Now, you can implement lock operations as follows:

- Acquiring lock
- First, acquire the RW-mutex. If the user requested the shared mode, acquire a read lock. If the user requested the exclusive mode, acquire a write lock.
- Check the counter. If it's zero, also acquire the file lock using `fcntl()`.
- Increment the counter.
- Releasing lock
- Decrement the counter.
- If the counter becomes zero, release the file lock using `fcntl()`.
- Release the RW-mutex.

This approach makes possible both thread and process synchronization.

Test program

I've prepared a small program that helps to learn the behavior of different lock types.

The program starts two threads or processes, both of which wait to acquire the lock, then sleep for one second, and then release the lock. It has three parameters:

- lock mode: `flock` (BSD locks), `lockf`, `fcntl_posix` (POSIX record locks), `fcntl_linux` (Open file description locks)

- access mode: `same_fd` (access lock via the same descriptor), `dup_fd` (access lock via duplicated descriptors), `two_fds` (access lock via two descriptors opened independently for the same path)
- concurrency mode: `threads` (access lock from two threads), `processes` (access lock from two processes)

Below you can find some examples.

Threads are not serialized if they use BSD locks on duplicated descriptors:

```
$ ./a.out flock dup_fd threads
13:00:58 pid=5790 tid=5790 lock
13:00:58 pid=5790 tid=5791 lock
13:00:58 pid=5790 tid=5790 sleep
13:00:58 pid=5790 tid=5791 sleep
13:00:59 pid=5790 tid=5791 unlock
13:00:59 pid=5790 tid=5790 unlock
```

But they are serialized if they are used on two independent descriptors:

```
$ ./a.out flock two_fds threads
13:01:03 pid=5792 tid=5792 lock
13:01:03 pid=5792 tid=5794 lock
13:01:03 pid=5792 tid=5792 sleep
13:01:04 pid=5792 tid=5792 unlock
13:01:04 pid=5792 tid=5794 sleep
13:01:05 pid=5792 tid=5794 unlock
```

Threads are not serialized if they use POSIX record locks on two independent descriptors:

```
$ ./a.out fcntl_posix two_fds threads
13:01:08 pid=5795 tid=5795 lock
13:01:08 pid=5795 tid=5796 lock
13:01:08 pid=5795 tid=5795 sleep
13:01:08 pid=5795 tid=5796 sleep
13:01:09 pid=5795 tid=5795 unlock
13:01:09 pid=5795 tid=5796 unlock
```

But processes are serialized:

```
$ ./a.out fcntl_posix two_fds processes
13:01:13 pid=5797 tid=5797 lock
13:01:13 pid=5798 tid=5798 lock
13:01:13 pid=5797 tid=5797 sleep
13:01:14 pid=5797 tid=5797 unlock
13:01:14 pid=5798 tid=5798 sleep
13:01:15 pid=5798 tid=5798 unlock
```

Command-line tools

The following tools may be used to acquire and release file locks from the command line:

- **`flock`**

  Provided by `util-linux` package. Uses `flock()` function.

  There are two ways to use this tool:

  - run a command while holding a lock:

    ```
    flock my.lock sleep 10
    ```

    `flock` will acquire the lock, run the command, and release the lock.

  - open a file descriptor in bash and use `flock` to acquire and release the lock manually:

    ```
    set -e            # die on errors
    exec 100>my.lock  # open file 'my.lock' and link file descriptor 100 to it
    flock -n 100      # acquire a lock
    echo hello
    sleep 10
    echo goodbye
    flock -u -n 100   # release the lock
    ```

  You can try to run these two snippets in parallel in different terminals and see that while one is sleeping while holding the lock, another is blocked in flock.

- **`lockfile`**

  Provided by `procmail` package.

  Runs the given command while holding a lock. Can use either `flock()`, `lockf()`, or `fcntl()` function, depending on what's available on the system.

There are also two ways to inspect the currently acquired locks:

- **`lslocks`**

  Provided by `util-linux` package.

  Lists all the currently held file locks in the entire system. Allows to perform filtering by PID and to configure the output format.

  Example output:

  ```
  COMMAND           PID   TYPE   SIZE MODE  M      START        END PATH
  containerd       4498  FLOCK   256K WRITE 0          0          0 /var/lib/docker
  dockerd          4289  FLOCK   256K WRITE 0          0          0 /var/lib/docker
  (undefined)        -1 OFDLCK        READ  0          0          0 /dev...
  ```

```
dockerd          4289  FLOCK    16K WRITE 0          0          0 /var/lib/docker
dockerd          4289  FLOCK    16K WRITE 0          0          0 /var/lib/docker
dockerd          4289  FLOCK    16K WRITE 0          0          0 /var/lib/docker
dockerd          4289  FLOCK    32K WRITE 0          0          0 /var/lib/docker
(unknown)        4417  FLOCK        WRITE 0          0          0 /run...
```

- `/proc/locks`

  A file in `procfs` virtual file system that shows current file locks of all types. The `lslocks` tools relies on this file.

  Example content:

```
16: FLOCK  ADVISORY  WRITE 4417 00:17:23319 0 EOF
27: FLOCK  ADVISORY  WRITE 4289 08:03:9441686 0 EOF
28: FLOCK  ADVISORY  WRITE 4289 08:03:9441684 0 EOF
29: FLOCK  ADVISORY  WRITE 4289 08:03:9441681 0 EOF
30: FLOCK  ADVISORY  WRITE 4289 08:03:8528339 0 EOF
31: OFDLCK ADVISORY  READ  -1 00:06:9218 0 EOF
43: FLOCK  ADVISORY  WRITE 4289 08:03:8536567 0 EOF
52: FLOCK  ADVISORY  WRITE 4498 08:03:8520185 0 EOF
```

Mandatory locking

Linux has limited support for mandatory file locking. See the "Mandatory locking" section in the `fcntl(2)` man page.

A mandatory lock is activated for a file when all of these conditions are met:

- The partition was mounted with the `mand` option.
- The set-group-ID bit is on and group-execute bit is off for the file.
- A POSIX record lock is acquired.

Note that the set-group-ID bit has its regular meaning of elevating privileges when the group-execute bit is on and a special meaning of enabling mandatory locking when the group-execute bit is off.

When a mandatory lock is activated, it affects regular system calls on the file:

- When an exclusive or shared lock is acquired, all system calls that *modify* the file (e.g. `open()` and `truncate()`) are blocked until the lock is released.
- When an exclusive lock is acquired, all system calls that *read* from the file (e.g. `read()`) are blocked until the lock is released.

However, the documentation mentions that current implementation is not reliable, in particular:

- races are possible when locks are acquired concurrently with `read()` or `write()`
- races are possible when using `mmap()`

Since mandatory locks are not allowed for directories and are ignored by `unlink()` and `rename()` calls, you can't prevent file deletion or renaming using these locks.

Example usage

Below you can find a usage example of mandatory locking.

`fcntl_lock.c`:

```c
#include <sys/fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    if (argc != 2) {
        fprintf(stderr, "usage: %s file\n", argv[0]);
        exit(1);
    }
```

```
    int fd = open(argv[1], O_RDWR);
    if (fd == -1) {
        perror("open");
        exit(1);
    }

    struct flock fl = {};
    fl.l_type = F_WRLCK;
    fl.l_whence = SEEK_SET;
    fl.l_start = 0;
    fl.l_len = 0;

    if (fcntl(fd, F_SETLKW, &fl) == -1) {
        perror("fcntl");
        exit(1);
    }

    pause();
    exit(0);
}
```

Build `fcntl_lock`:

```
$ gcc -o fcntl_lock fcntl_lock.c
```

Mount the partition and create a file with the mandatory locking enabled:

```
$ mkdir dir
$ mount -t tmpfs -o mand,size=1m tmpfs ./dir
$ echo hello > dir/lockfile
$ chmod g+s,g-x dir/lockfile
```

Acquire a lock in the first terminal:

```
$ ./fcntl_lock dir/lockfile
(wait for a while)
^C
```

Try to read the file in the second terminal:

```
$ cat dir/lockfile
(hangs until ^C is pressed in the first terminal)
hello
```