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.

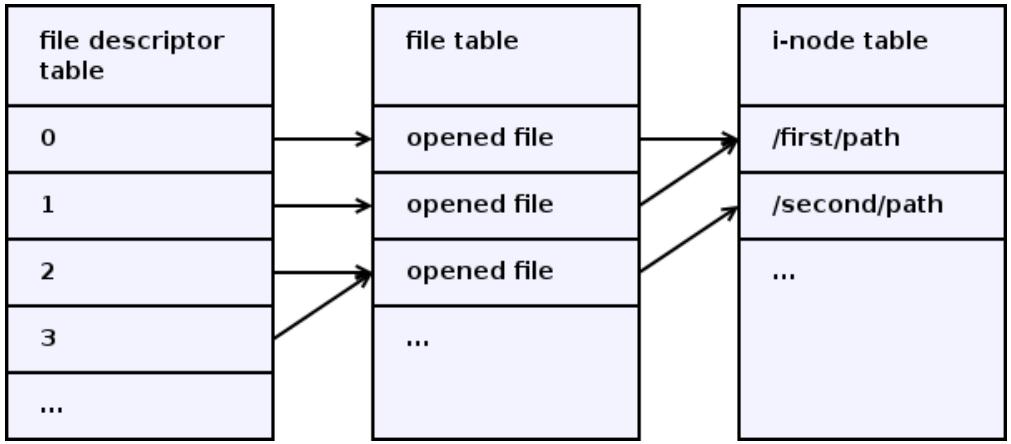
Differing features

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:

```
#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);
```

функция lockf

`lockf(3)` функция является упрощенной версией блокировки записей POSIX.

Характеристики:

- указано в POSIX (XSI)
- может применяться к диапазону байтов (опционально автоматически расширяясь при добавлении данных в будущем).
- ассоциируется с `[i-node, pid]`парой вместо файлового объекта
- поддерживает только эксклюзивные блокировки
- работает на NFS (в Linux)

Поскольку `lockf`блокировки связаны с `[i-node, pid]`парой, они имеют те же проблемы, что и блокировки записей POSIX, описанные выше.

Взаимодействие между `lockf`блокировками и другими типами блокировок не задается POSIX. В Linux `lockf`это просто оболочка для блокировок записей POSIX.

Пример использования:

```
#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);
}
```

Замки описания открытого файла (fcntl)

Блокировки описания открытых файлов специфичны для Linux и сочетают в себе преимущества блокировок BSD и блокировок записей POSIX. Они предоставляются`fcntl(2)`, см. Раздел “Open file description locks (non-POSIX)” на справочной странице.

Характеристики:

- Специфичная для Linux, не указанная в POSIX
- может быть применен к диапазону байтов
- связанный с объектом file
- гарантируйте атомарное переключение между режимами блокировки (exclusive и shared)
- работа с NFS (в Linux)

Таким образом, блокировки описания открытых файлов сочетают в себе преимущества блокировок BSD и блокировок записей POSIX: они обеспечивают как атомарное переключение между режимами блокировки, так и возможность синхронизации как потоков, так и процессов.

Эти блокировки доступны начиная с ядра 3.15.

API такой же, как и для блокировок записей POSIX (см. Выше). Он `struct flock`тоже использует. Разница только в `fcntl`именах команд:

- `F_OFD_SETLK` вместо `F_SETLK`
- `F_OFD_SETLKW` вместо `F_SETLKW`
- `F_OFD_GETLK` вместо `F_GETLK`

Эмуляция блокировок описания открытых файлов

Что у нас есть для многопоточности и атомарности?

- Блокировки BSD позволяют синхронизировать потоки, но не позволяют переключать атомарный режим.
- Блокировки записей POSIX не позволяют синхронизировать потоки, но позволяют переключать атомарный режим.
- Открыть описание файла блокировки позволяют оба, но доступны только на последних ядрах Linux.

Если вам нужны обе функции, но вы не можете использовать блокировки описания открытых файлов (например, вы используете какую-то встроенную систему с устаревшим ядром Linux), вы можете *эмулировать* их поверх блокировок записей POSIX.

Вот один из возможных подходов:

- Реализуйте свой собственный API для блокировки файлов. Убедитесь, что все потоки всегда используют этот API вместо `fcntl()`прямого использования. Убедитесь, что потоки никогда не открывают и не закрывают файлы блокировки напрямую.

- В API реализуйте синглтон для всего процесса (общий для всех потоков), содержащий все приобретенные в данный момент блокировки.

- Связать два дополнительных объекта с каждой полученной блокировкой:

- счетчик

- RW-мьютекс, например `pthread_rwlock`

Теперь вы можете реализовать операции блокировки следующим образом:

- Получение блокировки
- Сначала приобретите RW-мьютекс. Если пользователь запросил общий режим, получите блокировку чтения. Если пользователь запросил эксклюзивный режим, получите блокировку записи.
- Проверьте счетчик. Если она равна нулю, также приобретите блокировку файла с помощью `fcntl()`.
- Увеличьте счетчик.
- Отпирающий замок
- Уменьшите счетчик.
- Если счетчик становится равным нулю, отпустите блокировку файла с помощью `fcntl()`.
- Отпустите RW-мьютекс.

Такой подход делает возможной синхронизацию потоков и процессов.

Тестовая программа

Я подготовил небольшую программу, которая помогает изучить поведение различных типов блокировок.

Программа запускает два потока или процесса, оба из которых ждут получения блокировки, затем спят в течение одной секунды, а затем снимают блокировку. Он имеет три параметра:

- режим блокировки: `flock`(блокировки BSD), `lockf`, `fcntl_posix`(блокировки записей POSIX), `fcntl_linux`(блокировки описания открытого файла)
- режим доступа: `same_fd`(блокировка доступа через один и тот же дескриптор), `dup_fd`(блокировка доступа через дублированные дескрипторы), `two_fds`(блокировка доступа через два дескриптора, открытых независимо для одного и того же пути)

- режим параллелизма: `threads`(блокировка доступа из двух потоков), `processes`(блокировка доступа из двух процессов)

Ниже вы можете найти несколько примеров.

Потоки не сериализуются, если они используют блокировки BSD для дублированных дескрипторов:

```
$ ./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
```

Но они сериализуются, если используются на двух независимых дескрипторах:

```
$ ./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
```

Потоки не сериализуются, если они используют блокировки записей POSIX для двух независимых дескрипторов:

```
$ ./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
```

Но процессы сериализуются:

```
$ ./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
```

Инструменты командной строки

Для получения и снятия блокировок файлов из командной строки можно использовать следующие инструменты:

- **flock**

  Предоставляется `util-linux`посылка. Использует `flock()`функцию.

  Есть два способа использовать этот инструмент:

  - выполните команду, удерживая блокировку:

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

    `flock` получит блокировку, выполнит команду и отпустит блокировку.

  - откройте файловый дескриптор в bash и используйте `flock`его для получения и снятия блокировки вручную:

    ```
    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
    ```

Вы можете попробовать запустить эти два фрагмента параллельно в разных терминалах и увидеть, что пока один спит, удерживая блокировку, другой блокируется в flock.

- **lockfile**

  Предоставляется procmailпосылка.

  Запускает заданную команду, удерживая блокировку. Можно использовать либо flock()lockf()функцию , либо fcntl()функцию, в зависимости от того, что доступно в системе.

Существует также два способа проверки приобретенных в настоящее время замков:

- **lslocks**

  Предоставляется util-linuxпакетом.

  Перечисляет все текущие блокировки файлов во всей системе. Позволяет выполнять фильтрацию по PID и настраивать формат вывода.

  Пример вывода:

```
COMMAND           PID   TYPE   SIZE MODE  M      START        END PATH
containerd       4498  FLOCK   256K WRITE 0          0          0 /var/lib/docker/containerd/...
dockerd          4289  FLOCK   256K WRITE 0          0          0 /var/lib/docker/volumes/...
(undefined)        -1 OFDLCK        READ  0          0          0 /dev...
dockerd          4289  FLOCK    16K WRITE 0          0          0 /var/lib/docker/builder/...
dockerd          4289  FLOCK    16K WRITE 0          0          0 /var/lib/docker/buildkit/...
dockerd          4289  FLOCK    16K WRITE 0          0          0 /var/lib/docker/buildkit/...
dockerd          4289  FLOCK    32K WRITE 0          0          0 /var/lib/docker/buildkit/...
(unknown)        4417  FLOCK        WRITE 0          0          0 /run...
```

- **/proc/locks**

  Файл в procfsвиртуальной файловой системе, который показывает текущие блокировки файлов всех типов. lslocksИнструменты полагаются на этот файл.

  Пример содержимого:

```
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
```

Обязательная блокировка

Linux имеет ограниченную поддержку обязательной блокировки файлов. См. Раздел “Обязательная блокировка” на fcntl(2)справочной странице.

Обязательная блокировка активируется для файла при выполнении всех этих условий:

- Раздел был смонтирован с mandопцией.
- Бит set-group-ID включен, а бит group-execute выключен для файла.
- Получена блокировка записи POSIX.

Обратите внимание, что бит set-group-ID имеет свое обычное значение повышения привилегий, когда бит group-execute включен, и специальное значение включения обязательной блокировки, когда бит group-execute выключен.

Когда активирована обязательная блокировка, она влияет на регулярные системные вызовы файла:

- При получении эксклюзивной или общей блокировки все системные вызовы, *изменяющие* файл (напримерopen(), and truncate()), блокируются до тех пор, пока блокировка не будет снята.
- Когда приобретается эксклюзивная блокировка, все системные вызовы, которые *считываются* из файла (напримерread()), блокируются до тех пор, пока блокировка не будет снята.

Однако в документации упоминается, что текущая реализация не является надежной, в частности:

- гонки возможны, когда блокировки приобретаются одновременно с read()или write()
- гонки возможны при использовании mmap()

Поскольку обязательные блокировки не разрешены для каталогов и игнорируются unlink()rename()вызовами and, вы не можете предотвратить удаление или переименование файлов с помощью этих блокировок.

Пример использования

Ниже вы можете найти пример использования обязательной блокировки.

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);
}
```

Сборкаfcntl_lock:

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

Смонтируйте раздел и создайте файл с включенной обязательной блокировкой:

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

Получить блокировку в первом терминале:

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

Попробуйте прочитать файл во втором терминале:

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

ALSO ON **GAVV.GITHUB.IO**

Detecting USB devices with libudev

5 years ago • 3 comments

Below you can find code snippets that match USB devices using libudev. A …

Working on a new network transport …

3 years ago • 9 comments

Intro Last few years I was working on Roc, an open-source toolkit for …

Reusing UNIX domain socket …

5 years ago • 4 comments

Unix domain sockets are a networkless version of Internet sockets. They …

Roc 0.1 released: real-time streaming over …

3 years ago • 1 comment

What is Roc? I'm happy to announce the first release of Roc, version 0.1.0! Roc …

Pu ho

5 ye

Tab Abo com