Programmation Système
Cours 6 — File Locking

Stefano Zacchiroli
zack@pps.univ-paris-diderot.fr

Laboratoire PPS, Université Paris Diderot

2014–2015

Outline

Outline

Process synchronization

Consider again the distributed scheme to assign unique sequential identifiers.

- we store the global counter in a shared file
- each process accesses it as follows:
  1. read current sequence number (n) from file
  2. use sequence number n
  3. write $n + 1$ back to file

Process synchronization

Consider again the distributed scheme to assign unique sequential identifiers.

- we store the global counter in a shared file
- each process accesses it as follows:
  1. read current sequence number (n) from file
  2. use sequence number n
  3. write $n + 1$ back to file

Without synchronization the following might happen:

1. process A: read sequence number (obtains n)
2. process B: read sequence number (obtains n)
3. process A: use sequence number n
4. process A: write $n + 1$ back to file
5. process B: use sequence number n
6. process B: write $n + 1$ back to file

Process synchronization

Consider again the distributed scheme to assign unique sequential identifiers.

- we store the global counter in a shared file
- each process accesses it as follows:
  1. read current sequence number (n) from file
  2. use sequence number n
  3. write $n + 1$ back to file

Without synchronization the following might happen:

1. process A: read sequence number (obtains n)
2. process B: read sequence number (obtains n)
3. process A: use sequence number n
4. process A: write $n + 1$ back to file
5. process B: use sequence number n
6. process B: write $n + 1$ back to file

FAIL.

File locking

File locking is one of the simplest ways to perform synchronization among cooperating processes. With file locking, each process:

File locking protocol

1. place a lock on the file
2. executes its critical section
   - e.g. read sequence number n, use it, write $n + 1$ back
3. remove the lock

The kernel maintains internal locks associated with files on the filesystem and guarantees that only one process at a time can get a file lock — and therefore be in the critical section.

The rationale for associating locks with files is that synchronization is often used in conjunction with file I/O on shared files.

- file locks can also be used for general process synchronization, given their simplicity and the pervasiveness of files on UNIX

Advisory vs mandatory locking

We say mandatory locking when the locking system forbids (at the OS level) a process to perform I/O, unless it has obtained a specific file lock.

We say advisory locking when acquiring locks before entering the critical section is a convention agreed upon by cooperating processes, i.e.:

- acquiring the lock by multiple processes will be prevented...
- ...but performing I/O ignoring the lock won't!

Traditionally, the most common kind of file locking on UNIX is advisory locking.

File locking APIs

There are mainly 2 APIs for placing file locks on UNIX:

- `flock` (AKA BSD locks), which places locks on entire files
- `fcntl` (AKA POSIX locks), which places locks on file regions
  - `fcntl` offers a *superset* of `flock`'s features, but it's also plagued by important design flaws (more on this later)

Pick one, preferably `flock`. Do not mix the two!

File locking APIs — History

- in early UNIX system there was no support for file locking; that made impossible to build safe database systems on UNIX
- `flock` originated on BSD circa 1980. Nowadays, it is also used for general process synchronization (when FS is not bottleneck)
- `fcntl`-based locking descends from System V circa 1984, and is nowadays relatively popular
- POSIX.1 chose to standardize the `fcntl` API[1]

[1] for the wrong reasons, see `https://www.samba.org/samba/news/articles/low_point/tale_two_stds_os2.html`:

> *"AT&T brought [their] behavior to the standards committee as a proposal for byte-range locking, as this was how their current code implementation worked. The committee asked other ISVs if this was how locking should be done. The ISVs who cared about byte range locking were the large database vendors such as Oracle, Sybase and Informix (at the time). All of these companies did their own byte range locking within their own applications [...] their unanimous answer was "we don't care". In the absence of any strong negative feedback on a proposal, the committee added it "as-is", and took as the desired behavior the specifics of the first implementation, the brain-dead one from AT&T."*

Outline

flock

#include <sys/file.h>

int flock(**int** fd, **int** operation);

Returns: *0 if OK, -1 on error*

- `fd` is a file descriptor referencing an open file
- `operation` is an OR of the following flags:

flag	meaning
LOCK_SH	place a shared lock
LOCK_EX	place an exclusive lock
LOCK_UN	unlock instead of locking
LOCK_NB	make a non-blocking lock request

Locks are requested on open files. Hence, to lock a file a process should have the permissions needed to open it.
Other than that, read/write/exec permissions are irrelevant.

flock semantics

At any time, a process can hold 0 or 1 locks on an open file.

Two alternative lock kinds are available:

- a shared lock (AKA "read lock") can be hold by several processes at a time — LOCK_SH
- an exclusive lock (AKA "write lock") can be hold by only one process at a time, and is also incompatible with shared locks — LOCK_EX

Table: compatibility matrix of lock kinds

Process A	**Process B**	
	LOCK_SH	LOCK_EX
LOCK_SH	yes	no
LOCK_EX	no	no

flock semantics (cont.)

At any time, a process can hold 0 or 1 locks on an open file.

Blocking behavior. If the request lock cannot be granted, the process will block. Unblocking will happen as soon as the lock can be granted, atomically with giving the grant.
To avoid blocking, the flag LOCK_NB can be used. In that case, instead of blocking, flock will fail with errno set to EWOULDBLOCK.

Unlock. To release the currently held lock, the flag LOCK_UN can be used. Locks are also automatically released upon file close.

flock — example

```
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <sys/file.h>
#include <unistd.h>
#include "helpers.h"

#define LOCK_PATH       "my-lock"

int main(int argc, char **argv) {
        int fd, lock;

        if (argc < 2 || strlen(argv[1]) < 1) {
                printf("Usage: ./flock ( x | s ) [ n ]\n");
                exit(EXIT_FAILURE);
        }
        lock = (argv[1][0] == 'x') ? LOCK_EX : LOCK_SH;
        if (argc >= 3 && strlen(argv[2]) >= 1 && argv[2][0] == 'n')
                lock |= LOCK_NB;
```

flock — example (cont.)

```
        if ((fd = open(LOCK_PATH, O_RDONLY)) < 0)
                err_sys("open error");
        if (flock(fd, lock) < 0) {
                if (errno == EWOULDBLOCK)
                        err_sys("already locked");
                else
                        err_sys("flock error (acquire)");
        }
        printf("lock acquired, sleeping...\n");
        sleep(8);
        if (flock(fd, LOCK_UN) < 0)
                err_sys("flock error (release)");

        exit(EXIT_SUCCESS);
} // end of flock.c
```

flock — example (cont.)

Demo

- see flock(1) for a much more feature-full version of this utility, to be used in shell scripts

flock context

From flock(2):

> *Locks created by flock are associated with an open file table entry. This means that duplicate file descriptors (created by, for example, fork or dup) refer to the same lock, and this lock may be modified or released using any of these descriptors.*
>
> *[. . .]*
>
> *If a process uses open (or similar) to obtain more than one descriptor for the same file, these descriptors are treated independently by flock. An attempt to lock the file using one of these file descriptors may be denied by a lock that the calling process has already placed via another descriptor.*

flock context (cont.)

Let's unravel that...

Obtained file locks are stored by the kernel in the table of open files

- they are neither stored with the file descriptor itself nor stored in filesystem i-nodes
- ... but different locks pointing to the same file are verified globally (otherwise they won't work!)

These choices have important consequences on flock inheritance:

- upon FD duplication (dup/dup2), locks are inherited by the new file descriptors
- upon fork, we know that the entry in the table of open files is shared; therefore, locks are preserved as well
- upon exec, open files are left untouched; once more, locks are preserved

flock inheritance gotchas

flock context can result in surprising behavior, if we're not careful. Here are some common "gotchas".[2]

```
fd1 = open("foo.txt", O_RDWR);
flock(fd1, LOCK_EX);
fd2 = dup(fd1);
flock(fd2, LOCK_UN);
/* is "foo.txt" locked now? */
```

[2] BSD / Linux behavior

flock inheritance gotchas

flock context can result in surprising behavior, if we're not careful. Here are some common "gotchas".[2]

```
fd1 = open("foo.txt", O_RDWR);
flock(fd1, LOCK_EX);
fd2 = dup(fd1);
flock(fd2, LOCK_UN);
/* is "foo.txt" locked now? */
```

- there is only one lock, stored in the open file table entry pointed by both fd1 and fd2
- it will be released by LOCK_UN

[2] BSD / Linux behavior

flock inheritance gotchas (cont.)

```
fd1 = open("foo.txt", O_RDWR);
fd2 = open("foo.txt", O_RDWR);
flock(fd1, LOCK_EX);
flock(fd2, LOCK_EX);
/* is "foo.txt" locked now? */
```

flock inheritance gotchas (cont.)

```
fd1 = open("foo.txt", O_RDWR);
fd2 = open("foo.txt", O_RDWR);
flock(fd1, LOCK_EX);
flock(fd2, LOCK_EX);
/* is "foo.txt" locked now? */
```

- there are two different locks, as separate open create separate entries in the open file table (possibly for the same file)
- ... but flock consider all entries before granting locks!
- the 2nd flock will block (forever...) as we are trying to acquire 2 *different* exclusive locks on the same file

Warning

A process can lock himself out of a flock lock, if not careful.

flock inheritance gotchas (cont.)

```
fd = open("foo.txt", O_RDWR);
flock(fd, LOCK_EX);
if (fork() == 0)  /* child */
        flock(fd, LOCK_UN);
/* is "foo.txt" locked for the parent? */
/* is "foo.txt" locked for the child? */
```

flock inheritance gotchas (cont.)

```
fd = open("foo.txt", O_RDWR);
flock(fd, LOCK_EX);
if (fork() == 0)  /* child */
        flock(fd, LOCK_UN);
/* is "foo.txt" locked for the parent? */
/* is "foo.txt" locked for the child? */
```

- there is only one lock, inherited through fork
- upon LOCK_UN, the child will release the lock for both himself and the parent

This is actually useful: it allows to transfer a lock from parent to child without race conditions and without having to use explicit synchronization. To do so, after fork the parent should close its file descriptor, leaving the child in control of the lock.

flock — limitations

flock suffers of some limitations:

- granularity: only entire files can be locked
  - fine when files are used only as rendez-vous points for synchronizing access to something else
  - but possibly limiting when synchronizing for shared file I/O
- flock can only do advisory locking
- portability: not in POSIX
  - the implementations of relevant file systems might not support flock locks
  - e.g. in Linux's NFS implementation, support for flock locks has been added only in version 2.6.12 (June 2005)

Outline

Interlude: one syscall to rule them all

Many of the system calls we have seen manipulate FDs for specific purposes. That API model is "1 action ↔ 1 syscall".
An alternative model is "many actions ↔ 1 syscall", i.e. use a single syscall with a command argument used as a dispatcher for several actions. *Restricted* examples of this are:

- `lseek`'s `whence` argument
- `signal`'s `handler` argument
- `flock`'s `operation` argument

trade off: API size ↔ API complexity

Dispatcher syscalls tend to the right in the above trade-off:

pro less clutter in the API namespace

pro easier to extend without breaking ABI compatibility

con diminished code—and API doc.—readability

con harder to detect type error (as in "typed prog. languages")

Interlude: fcntl

`fcntl` is a dispatcher syscall for a wide range of FD manipulations:

- duplication
- flags (e.g. close on exec)
- locking
- request signal notifications

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* arg */);
                          Returns: depends on cmd; often: 0 if OK, -1 on error
```

- `fd` is the FD that will be acted upon
- `cmd` is the desired action (the dispatcher argument)
- `/* arg */` is a variable list of arguments, depending on `cmd`

The portability of `fcntl` varies from command to command (this is possible "thanks" to the extensibility of dispatcher syscalls).

Interlude: fcntl — example (dup)

```
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include "helpers.h"

int main(void) {
        int fd;
        if ((fd = fcntl(STDOUT_FILENO, F_DUPFD, 0)) < 0)
                err_sys("fcntl error");
        if (write(fd, "Hello, ", 7) != 7
            || write(STDOUT_FILENO, "World!\n", 7) != 7)
                err_sys("write error");
        exit(EXIT_SUCCESS);
} // end of fcntl-dup.c

$ ./fcntl-dup
Hello, World!
$
```

Interlude: fcntl — close-on-exec

Associated to each entry in the table of open files, the kernel keeps a list of file descriptor boolean flags.

One such flag[3] is close-on-exec. It states whether the file descriptor should be automatically closed upon exec or not (the default).

fcntl operations F_GETFD (get) and F_SETFD are used to get/set the flags. The close-on-exec bit corresponds to the FD_CLOEXEC constant.

- for F_GETFD the return value is the current file descriptor flags on success; -1 otherwise
- for F_SETFD the return value is 0 on success, -1 otherwise

[3] in fact, the only one standardized upon

Interlude: fcntl — example (close-on-exec)

```
#include <fcntl.h>
#include <unistd.h>
#include "helpers.h"
int main(void) {
        pid_t pid;
        int fdflags;

        if ((pid = fork()) < 0)  err_sys("fork error");
        else if (pid == 0) {     /* 1st child */
                system("echo '1: Hello, World!'");
                exit(EXIT_SUCCESS);
        }
        fdflags = fcntl(STDOUT_FILENO, F_GETFD);
        fdflags |= FD_CLOEXEC;
        if (fcntl(STDOUT_FILENO, F_SETFD, fdflags) < 0)
                err_sys("fcntl error");
        if ((pid = fork()) < 0)  err_sys("fork error");
        else if (pid == 0) {     /* 2nd child */
                system("echo '2: Hello, World!'");
                exit(EXIT_SUCCESS);
        }
        sleep(1);
        exit(EXIT_SUCCESS);
} // end of fcntl-cloexec.c
```

Interlude: `fcntl` — example (close-on-exec)

Demo

Notes:

- assumption: `system` is implemented in terms of exec
  - safe assumption on UNIX systems
- the 2nd child does not print anything on STDOUT, as it's been closed upon exec

Outline

POSIX locks

Contrary to what happens with `flock`, where locks are global, using `fcntl` processes can place locks on byte ranges of an open file, referenced by a FD.

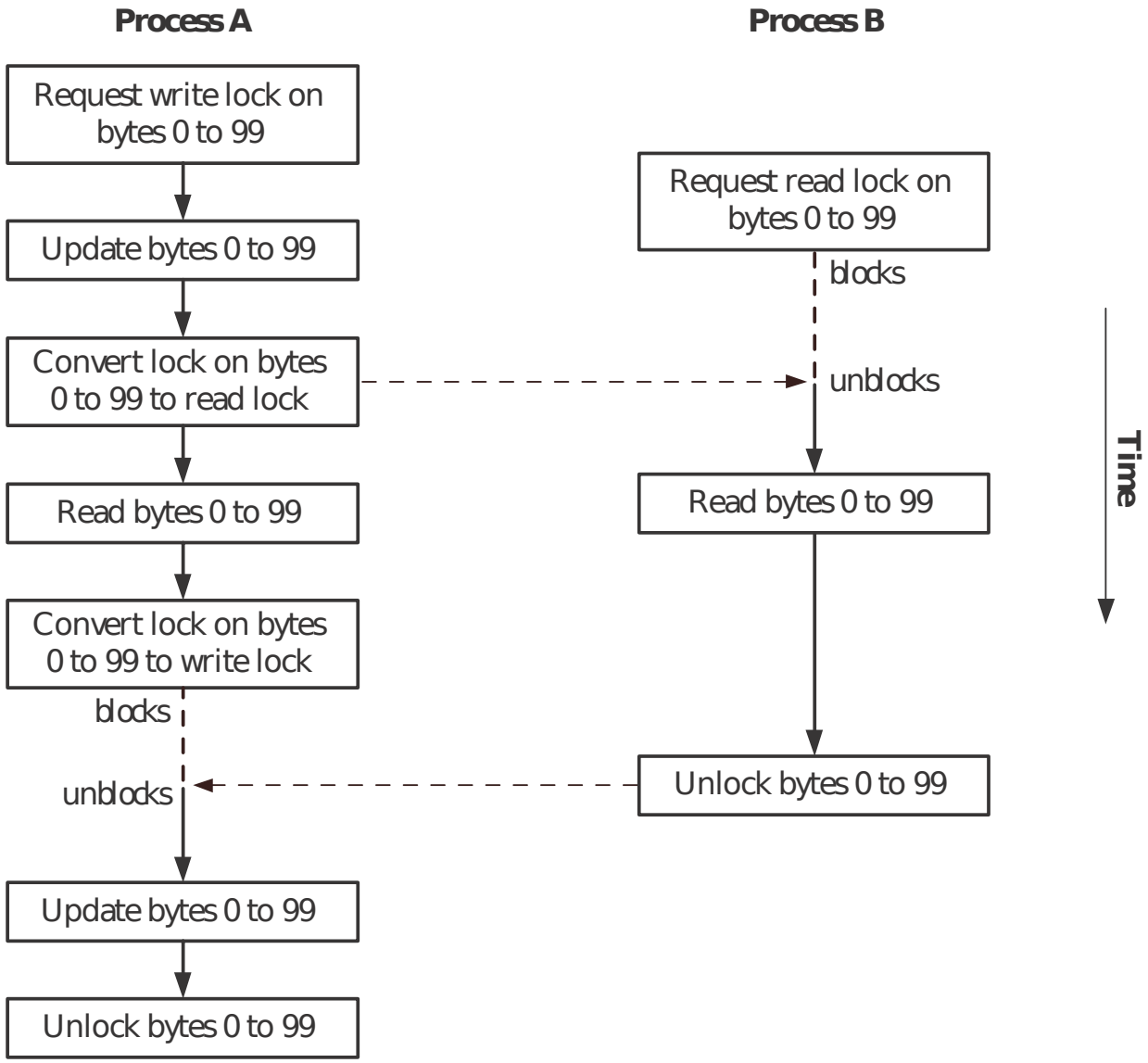
Two kinds of locks are supported: write locks (equivalent to `flock`'s LOCK_EX) and read locks (eq. to LOCK_SH).

POSIX locking is often also called record locking, but that is a misnomer:

- the name is meaningful on OS where the conceptual model of files is record-driven, i.e. a file is a list of records
- that is not the case on UNIX, where files are byte streams, i.e. list of bytes

POSIX locking is also called `fcntl` locking, from the syscall.

POSIX locking — sample usage



TLPI, Figure 55-2

POSIX locking — model

With `flock`, we saw that each process can hold 0 or 1 locks on each of its entries in the open file table, no matter the lock kind.
With POSIX locks the principle remains, but the granularity becomes the byte range, sized from 1 byte to the entire file.

- locks are requested/released on byte ranges
- intuitively, locks "distribute" to each byte of the locked ranges
  - on any byte of an open file, a process can hold zero or one lock (no matter the lock kind)

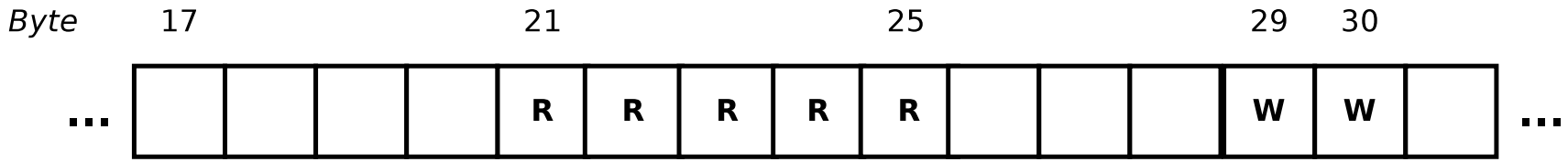


Figure: two ranges locked: one with a read lock (bytes 21–25), the other with a write lock (bytes 29–30)

POSIX locking — model (cont.)

Internally, the kernel represents POSIX locks as ranges (in the mathematical sense), to minimize the size of the representation. Automatic split and merge of ranges is performed when needed.

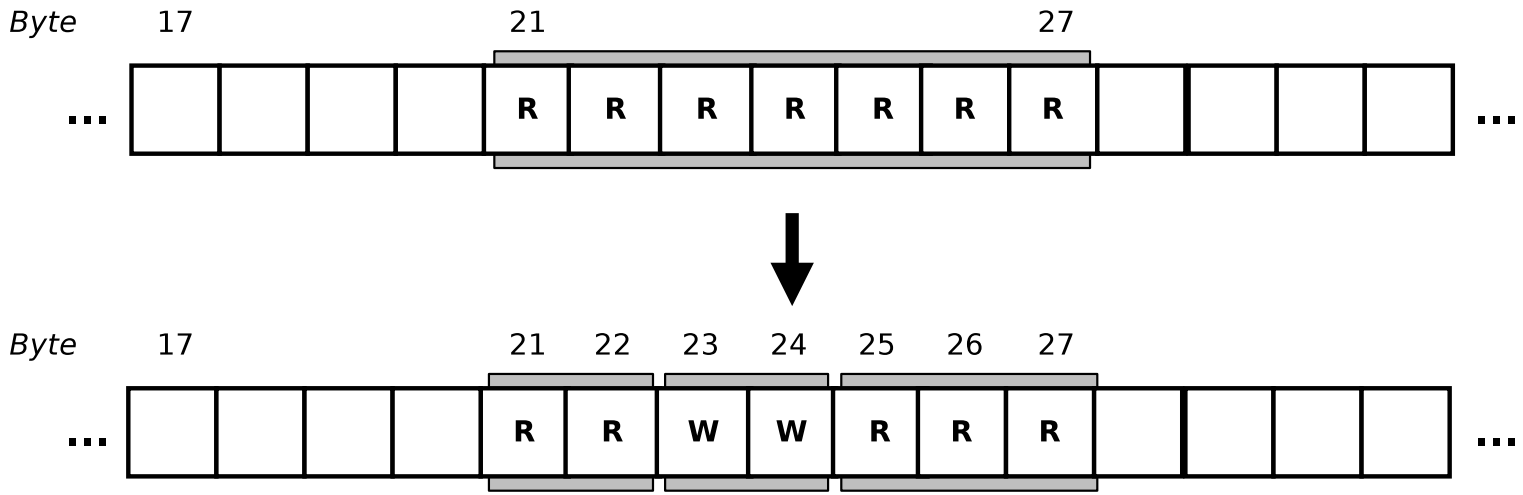


Figure: taking a write lock in the middle of a previously read-lock-ed range

A (Linux-specific) peek on the internal kernel representation can be obtained by looking at /proc/locks:

```
$ head -n 2 /proc/locks
1: POSIX  ADVISORY  WRITE 27134 fe:03:2649029 0 EOF
2: POSIX  ADVISORY  WRITE 5171 fe:03:1014279 1073741824 1073742335
```

POSIX locking — permissions

With `flock` locks, in a sense, the file was a *rendez-vous* point among processes for synchronization purposes:

- the *content* of the file does not matter
- even though it *might* happen that the shared resource, that processes want to access, is that very same file

With POSIX locks the file content is much more important: *parts* of it are now used as rendez-vous point.

Coherently with this intuition, permissions are more fine grained with `fcntl` locks:

- to be able to put a read lock, a process needs read permission on a file
- to be able to put a write lock, a process needs write permission on a file

POSIX locking — ranges

Ranges for POSIX locks are specified by providing:

1. range starting point (absolute position in bytes, inclusive)
2. range length (in bytes)

Starting point is specified as in `lseek`, i.e. by providing an `offset` and a whence argument:

- SEEK_SET, for absolute offsets from the beginning of the file
- SEEK_CUR, for relative offsets to the current file offset
- SEEK_END, for relative (negative) offsets from end of file

POSIX locking — ranges

Ranges for POSIX locks are specified by providing:

1. range starting point (absolute position in bytes, inclusive)
2. range length (in bytes)

Starting point is specified as in `lseek`, i.e. by providing an `offset` and a whence argument:

- SEEK_SET, for absolute offsets from the beginning of the file
- SEEK_CUR, for relative offsets to the current file offset
- SEEK_END, for relative (negative) offsets from end of file

It is allowed to lock bytes which are past EOF. But there is no guarantee that will be enough, as the file could grow more. "EOF-sticky" ranges can be specifying with a length of 0 bytes. Such ranges will always extend to EOF, no matter the file growth.

Corollary: `flock` can be emulated using the $\langle 0, 0 \rangle$ range

POSIX locking — invocation

`fcntl` can be used to request POSIX locking as follows:

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd, struct flock *flock);
```

Returns: *0 if OK, -1 on error*

---

where `cmd` is one of `F_SETLK`, `F_SETLKW`, `F_GETLK`.
The `flock` structure is used to specify range, lock type, as well as a value-return argument:

```
struct flock {
        short l_type;    /* lock type */
        short l_whence; /* how to interpret l_start */
        off_t l_start;  /* range start */
        off_t l_len;    /* range length */
        pid_t l_pid;    /* who holds the lock (for F_GETLK) */
}
```

POSIX locking — modify locks

Two cmd values are used to acquire locks:

F_SETLK acquire or release a lock on the given range, depending on `l_type`

l_type	action
F_RDLCK	acquire a read lock
F_WRLCK	acquire a write lock
F_UNLCK	release a lock

if *any* incompatible lock is held by other processes, `fcntl` will fail with `errno` set to either EAGAIN or EACCESS

F_SETLKW same as above, but with blocking behavior, `fcntl` will block until the lock can be granted

Note: semantics is all or nothing, an incompatible lock on a single byte of the requested range will trigger failure.

POSIX locking — check for locks

With the F_GETLK command we can check if it would be possible to acquire a lock—of the given kind, on the given range.

For F_GETLK the `flock` structure is used as a value-return argument

- `l_type` will be F_UNLCK if the lock would have been permitted
- otherwise, information about one of those ranges will be returned, in particular
  - `l_pid`: PID of the process holding the lock
  - `l_type`: kind of lock that is blocking us
  - range in `l_start` and `l_len`, with `l_whence` always set to SEEK_SET

Warning

Any common combination of F_GETLK with F_SETLK(W) is subject to race conditions, as in between the two the lock situation might change.

POSIX locking — example

As an example of POSIX locking, we fix the race condition in our distributed scheme to assign sequential unique identifiers (dropping `mmap`, as an unrelated change).

To request an identifier each client will:

1. open a well-known file
2. write-lock the part of it that contains the counter
3. read the counter (why now?)
4. update the counter
5. release the lock

POSIX locking — example (cont.)

We use a (non-portable) record-oriented format defined as follows:

1. beginning of file
2. magic number 42 (written as the C string "42\0")
3. next `sizeof(time_t)` bytes: time of last change, in seconds from epoch as returned by `time`
4. next `sizeof(long)` bytes: current value of global counter
5. end of file

As the file format is binary, we need a custom utility to initialize it.

POSIX locking — example (protocol)

```c
#include <fcntl.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include "helpers.h"

#define DB_FILE "counter.data"
#define MAGIC   "42"
#define MAGIC_SIZ       sizeof(MAGIC)

struct glob_id {
        time_t ts;      /* last modification timestamp */
        long val;       /* global counter value */
};
```

POSIX locking — example (library)

```
int glob_id_verify_magic(int fd) {
        char buf[16];
        struct flock lock;

        lock.l_type = F_RDLCK;          /* read lock */
        lock.l_whence = SEEK_SET;       /* abs. position */
        lock.l_start = 0;               /* from begin... */
        lock.l_len = MAGIC_SIZ;         /* ...to magic's end */
        printf("  acquiring read lock...\n");
        if (fcntl(fd, F_SETLKW, &lock) < 0)
                err_sys("fcntl error");

        if (read(fd, buf, MAGIC_SIZ) != MAGIC_SIZ)
                err_sys("read error");
        lock.l_type = F_UNLCK;
        printf("  releasing read lock...\n");
        if (fcntl(fd, F_SETLK, &lock) < 0)
                err_sys("fcntl error");

        return (strncmp(buf, MAGIC, 16) == 0 ? 0 : -1);
}
```

POSIX locking — example (library) (cont.)

```
int glob_id_write(int fd, long val) {
        int rc;
        struct glob_id id;

        id.ts = time(NULL);
        id.val = val;
        if ((rc = write(fd, &id, sizeof(struct glob_id))
             != sizeof(struct glob_id)))
                return rc;
        return 0;
}

// end of fcntl-uid-common.h
```

POSIX locking — example (DB init/reset)

```
#include "fcntl-uid-common.h"
int main(void) {
        int fd;
        struct flock lock;

        if ((fd = open(DB_FILE, O_WRONLY | O_CREAT | O_TRUNC,
                        S_IRUSR | S_IWUSR)) < 0)
                err_sys("open error");
        lock.l_type = F_WRLCK;          /* write lock */
        lock.l_whence = SEEK_SET;       /* abs. position */
        lock.l_start = 0;               /* from begin... */
        lock.l_len = 0;                 /* ...to EOF */
        printf("acquiring write lock...\n");
        if (fcntl(fd, F_SETLKW, &lock) < 0)
                err_sys("fcntl error");

        if (write(fd, MAGIC, MAGIC_SIZ) != MAGIC_SIZ
            || glob_id_write(fd, (long) 0) < 0)
                err_sys("write error");
        exit(EXIT_SUCCESS);
} // end of fcntl-uid-reset.c
```

POSIX locking — example (client)

```
#include "fcntl-uid-common.h"
int main(void) {
        int fd;
        struct glob_id id;
        struct flock lock;

        if ((fd = open(DB_FILE, O_RDWR)) < 0)
                err_sys("open error");
        printf("checking magic number...\n");
        if (glob_id_verify_magic(fd) < 0) {
                printf("invalid magic number: abort.\n");
                exit(EXIT_FAILURE);
        }

        lock.l_type = F_WRLCK;          /* write lock */
        lock.l_whence = SEEK_SET;       /* abs. position */
        lock.l_start = MAGIC_SIZ;       /* from magicno... */
        lock.l_len = 0;                 /* ...to EOF */
        printf("acquiring write lock...\n");
        if (fcntl(fd, F_SETLKW, &lock) < 0)
                err_sys("fcntl error");
```

POSIX locking — example (client) (cont.)

```
        if (lseek(fd, MAGIC_SIZ, SEEK_SET) < 0)
                err_sys("lseek error");
        if (read(fd, &id, sizeof(struct glob_id))
            != sizeof(struct glob_id))
                err_sys("read error (too lazy to retry...)");
        printf("got id: %ld\n", id.val);

        sleep(5);  /* to ease interactive testing */

        if (lseek(fd, MAGIC_SIZ, SEEK_SET) < 0)
                err_sys("lseek error");
        glob_id_write(fd, id.val + 1);

        exit(EXIT_SUCCESS);
} // end of fcntl-uid-get.c
```

POSIX locking — example

Demo

Notes:

- working with record-oriented files is painful!
  - . . . and not very UNIX-y
- as expected:
  - different byte ranges can be locked independently
  - write locks are mutually exclusive
  - read locks block write locks
- locks are automatically released at process termination

Deadlock...

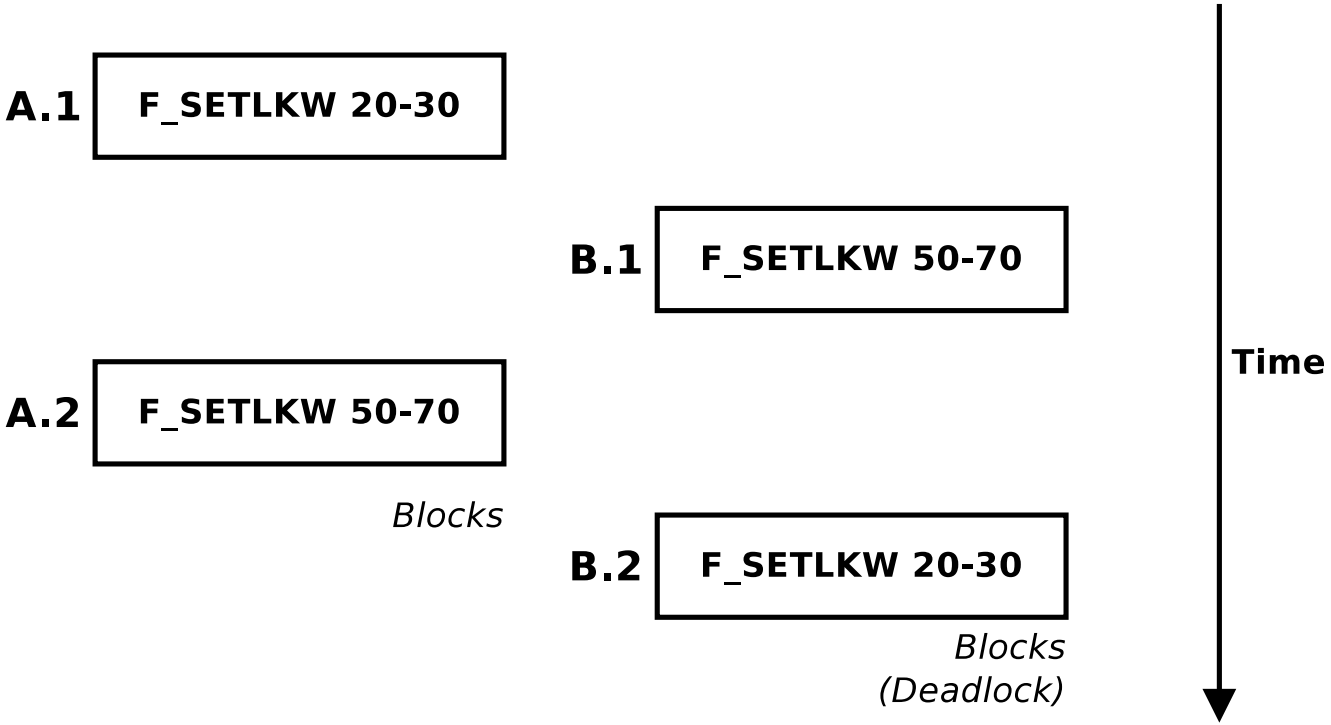
Consider two processes doing the following:

Process A	**Process B**
A.1 F_SETLKW on bytes 20–30	B.1 F_SETLKW on bytes 50–70
A.2 F_SETLKW on bytes 50–70	B.2 F_SETLKW on bytes 20–30

Which of the following action interleaving cause problem?

1. A.1 → A.2 → B.1 → B.2
2. B.1 → B.2 → A.1 → A.2
3. A.1 → B.1 → B.2 → A.2
4. B.1 → A.1 → A.2 → B.2
5. A.1 → B.1 → A.2 → B.2
6. B.1 → A.1 → B.2 → A.2

Deadlock. . . (cont.)



Definition (Deadlock)

A deadlock is a situation in which a circular list of two or more processes are each waiting for the availability of a resource hold by the successor in the list. (and therefore nobody can obtain it)

. . . and deadlock detection

Without assistance from the kernel, a deadlock will leave all involved processes blocked forever.
Many techniques exist to deal with deadlocks, ranging from prevention, to avoidance and detection.

For POSIX locks, the kernel is capable of deadlock detection.

When a deadlock is detected, the kernel choose one `fcntl` call involved—arbitrarily, according to SUSv3—and make it fail with `errno` set to EDEADLK.

Deadlock detection — example

```
#include <fcntl.h>
#include <unistd.h>
#include "helpers.h"

void lockabyte(const char *name, int fd, off_t offset) {
        struct flock lock;

        lock.l_type = F_WRLCK;
        lock.l_start = offset;
        lock.l_whence = SEEK_SET;
        lock.l_len = 1;

        if (fcntl(fd, F_SETLKW, &lock) < 0)
                err_sys(name);

        printf("%s: got the lock, byte %ld\n", name, offset);
}
```

Deadlock detection — example (cont.)

```
int main(void) {
        int fd;
        pid_t pid;

        if ((fd = creat("my-lock", S_IRUSR | S_IWUSR)) < 0)
                err_sys("creat error");
        if (write(fd, "ab", 2) != 2)
                err_sys("write error");
        if ((pid = fork()) < 0) {
                err_sys("fork error");
        } else if (pid == 0) {                          /* child */
                lockabyte("child", fd, 0);
                sleep(1);
                lockabyte("child", fd, 1);
        } else {                                        /* parent */
                lockabyte("parent", fd, 1);
                sleep(1);
                lockabyte("parent", fd, 0);
        }
        exit(EXIT_SUCCESS);
} // end of deadlock.c
```

Deadlock detection — example (cont.)

Demo

Notes:

- as usual `sleep(1)` does not guarantee that the deadlock *will* occur; how can we *force* the deadlock to happen?
  - deadlocks are common causes of hard to debug bugs
- on Linux, it is the most recent `fcntl` invocation that will fail; SUSv3 gives no such guarantee
  - in this example, depending on the timing it will be either the parent or the child to get a `fcntl` error

POSIX locking context and inheritance

Context

POSIX locks are associated to ⟨PID, i-node⟩ pairs. That is a very different context than `flock` locks contexts.

Some consequences:

- when a process terminates, all its locks are released
- a process can no longer lock himself out by opening a file twice, because ⟨PID, i-node⟩ keys don't change
- threads (which share the same PID) cannot use POSIX locks for synchronization purposes :-(

Inheritance

- `fcntl` locks are preserved through exec same PID
- `fcntl` locks are not inherited upon `fork` != pid
  - there is no way to atomically pass locks to children :-(

The problem with POSIX locking

Release

- when a process `close` *any* FD, all POSIX locks on the corresponding file are released
  - there is no way to fool this (dup, dup2, etc.)
  - particularly bad for encapsulation in library code that need to hand out FDs :-((
- even opening/closing the same file independently will release all locks
  - note: hard links count too!
  - very bad for encapsulation: how do you ensure that your code is the only code dealing with a given file? :-(((

This is why POSIX locks are not much “loved” by developers.

Outline

Open File Description (OFD) locks

(Note: it is *description*, not descriptor.)

Open File Description (OFD) locks is an attempt to get the best of the two locking worlds: byte range locks with the sane BSD locking semantics.

- not in POSIX (yet, request pending[4])
- supported in Linux since 3.15 (June 2014)
- supported in glibc since 2.20 (September 2014)

Jeffrey T. Layton
~~File-private POSIX locks~~ Open File Description locks[5]
LWN, https://lwn.net/Articles/586904/

[4] http://austingroupbugs.net/view.php?id=768
[5] renamed from "file-private" to OFD after that article

OFD locks — usage

On supported platforms, `fcntl` can be used to request OFD locks:

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd, struct flock *flock);
```

Returns: *0 if OK, -1 on error*

---

where `cmd` is one of F_*OFD*_SETLK, F_*OFD*_SETLKW, F_*OFD*_GETLK. The `flock` argument has the same format and semantics of POSIX locks.

OFD locks — semantics

From the glibc doc:[6]

> *In contrast to [POSIX locks], Open File Description (OFD) record locks are associated with an open file description rather than a process.*
>
> *[applying a lock] on a region that already has an existing [OFD lock] that was created via the same file descriptor will never cause a lock conflict.*
>
> *Using dup [...] to copy a file descriptor does not give you a new open file description, but rather copies a reference to an existing open file description [...]*
>
> *[OFD locks] can also be used to synchronize file access between threads within the same process by having each thread perform its own open of the file [...]*
>
> *[OFD locks] are automatically freed only upon closing the last file descriptor that refers to the open file description, this locking mechanism avoids the possibility that locks are inadvertently released due to a library routine [opening/closing] a file [...]*

[6] http://www.gnu.org/software/libc/manual/html_node/Open-File-Description-Locks.html