Advanced Programming in the UNIX Environment

Week 12, Segment 3:
Resource Locking

Department of Computer Science
Stevens Institute of Technology

Jan Schaumann
jschauma@stevens.edu
https://stevens.netmeister.org/631/

Resource Locking

Ways we have learned so far to ensure only one process has exclusive access to a resource:

- open file using `O_CREAT|O_EXCL`, then immediately `unlink(2)` it
- create a “lockfile” – if file exists, somebody else is using the resource
- use of a semaphore

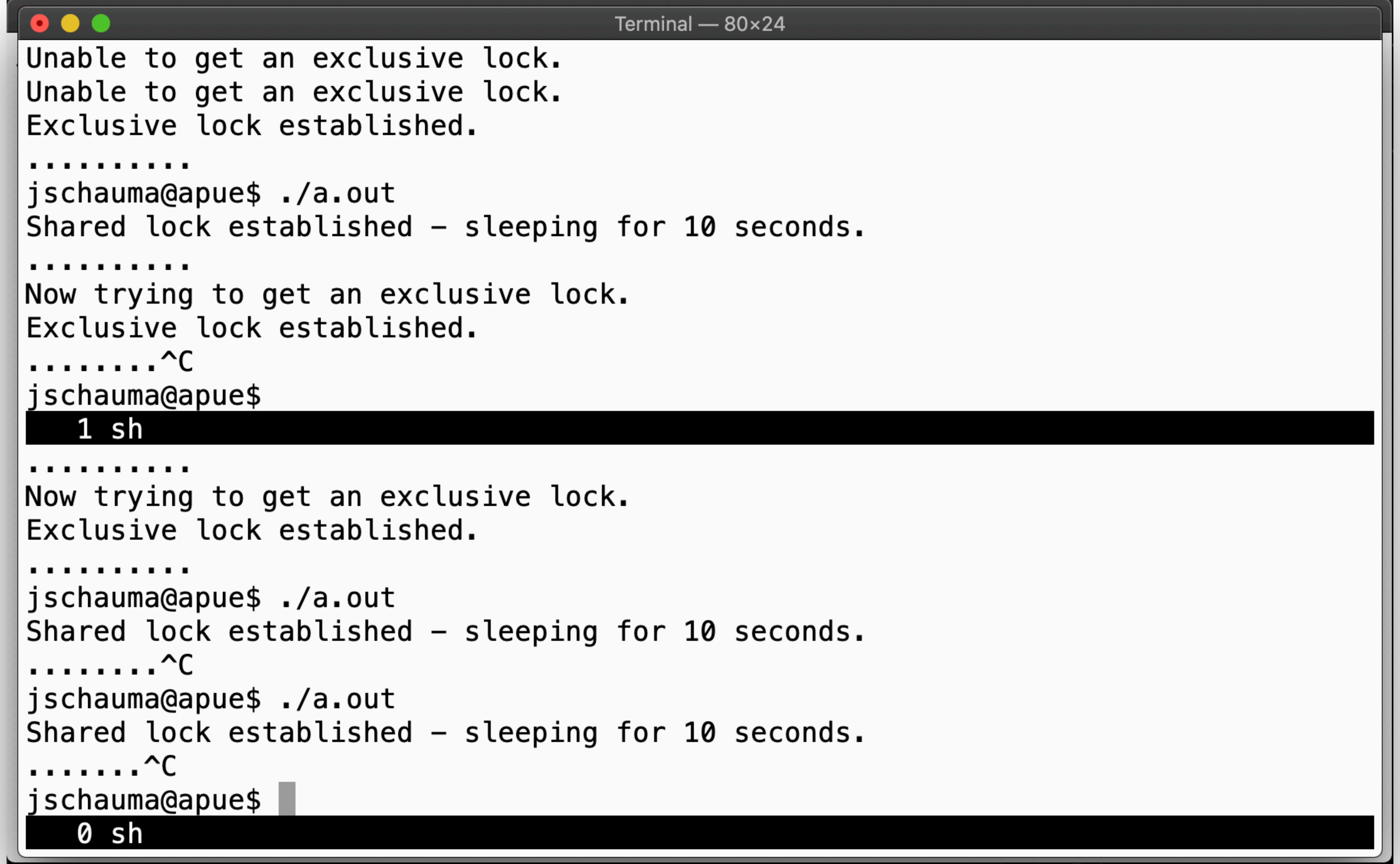
flock(2)

```
#include <fnctl.h>

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

Returns: 0 on success, -1 on error

- applies or removes an *advisory* lock on the file associated with the file descriptor `fd`
- *operation* can be `LOCK_NB` and any one of:
  - `LOCK_SH`
  - `LOCK_EX`
  - `LOCK_UN`
- locks entire file
- see flockfile(3) for locking stdio streams



```
Unable to get an exclusive lock.
Unable to get an exclusive lock.
Exclusive lock established.
..........
jschauma@apue$ ./a.out
Shared lock established - sleeping for 10 seconds.
..........
Now trying to get an exclusive lock.
Exclusive lock established.
........^C
jschauma@apue$
   1 sh
..........
Now trying to get an exclusive lock.
Exclusive lock established.
..........
jschauma@apue$ ./a.out
Shared lock established - sleeping for 10 seconds.
........^C
jschauma@apue$ ./a.out
Shared lock established - sleeping for 10 seconds.
.......^C
jschauma@apue$
   0 sh
```

Advisory Record Locking

Record locking is done using fcntl(2), using one of F_GETLK, F_SETLK or F_SETLKW and passing a

```
struct flock {
        short l_type;   /* F_RDLCK, F_WRLCK, or F_UNLCK */
        off_t l_start;  /* offset in bytes from l_whence */
        short l_whence; /* SEEK_SET, SEEK_CUR, or SEEK_END */
        off_t l_len;    /* length, in bytes; 0 means "lock to EOF" */
        pid_t l_pid;    /* returned by F_GETLK */
}
```

Lock types are:

- F_RDLCK – Non-exclusive (read) lock; fails if write lock exists.
- F_WRLCK – Exclusive (write) lock; fails if any lock exists.
- F_UNLCK – Releases our lock on specified range.

Advisory Record Locking

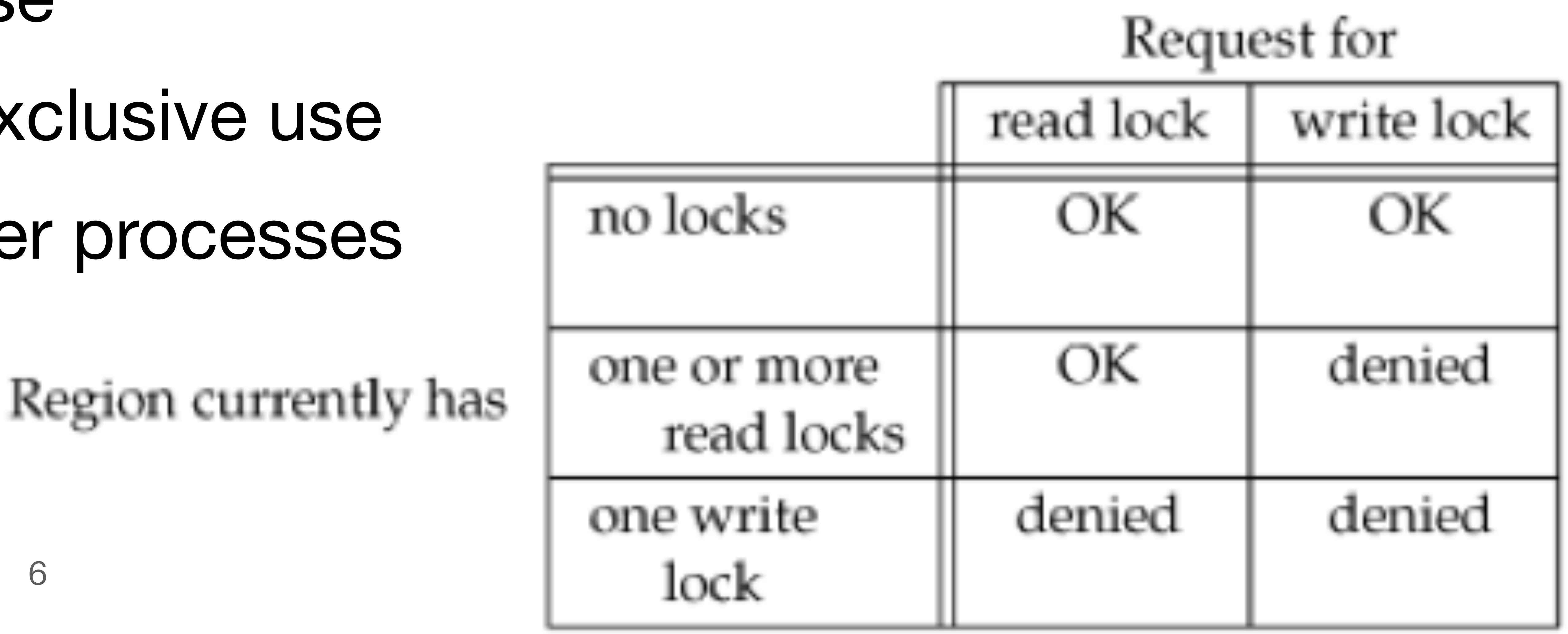
```
#include <unistd.h>

int lockf(int fd, int value, off_t size);
```

Returns: 0 on success, -1 on error

value can be:

- `F_ULOCK` – unlock locked sections
- `F_LOCK` – lock a section for exclusive use
- `F_TLOCK` – test and lock a section for exclusive use
- `F_TEST` – test a section for locks by other processes



		Request for	
		read lock	write lock
Region currently has	no locks	OK	OK
	one or more read locks	OK	denied
	one write lock	denied	denied

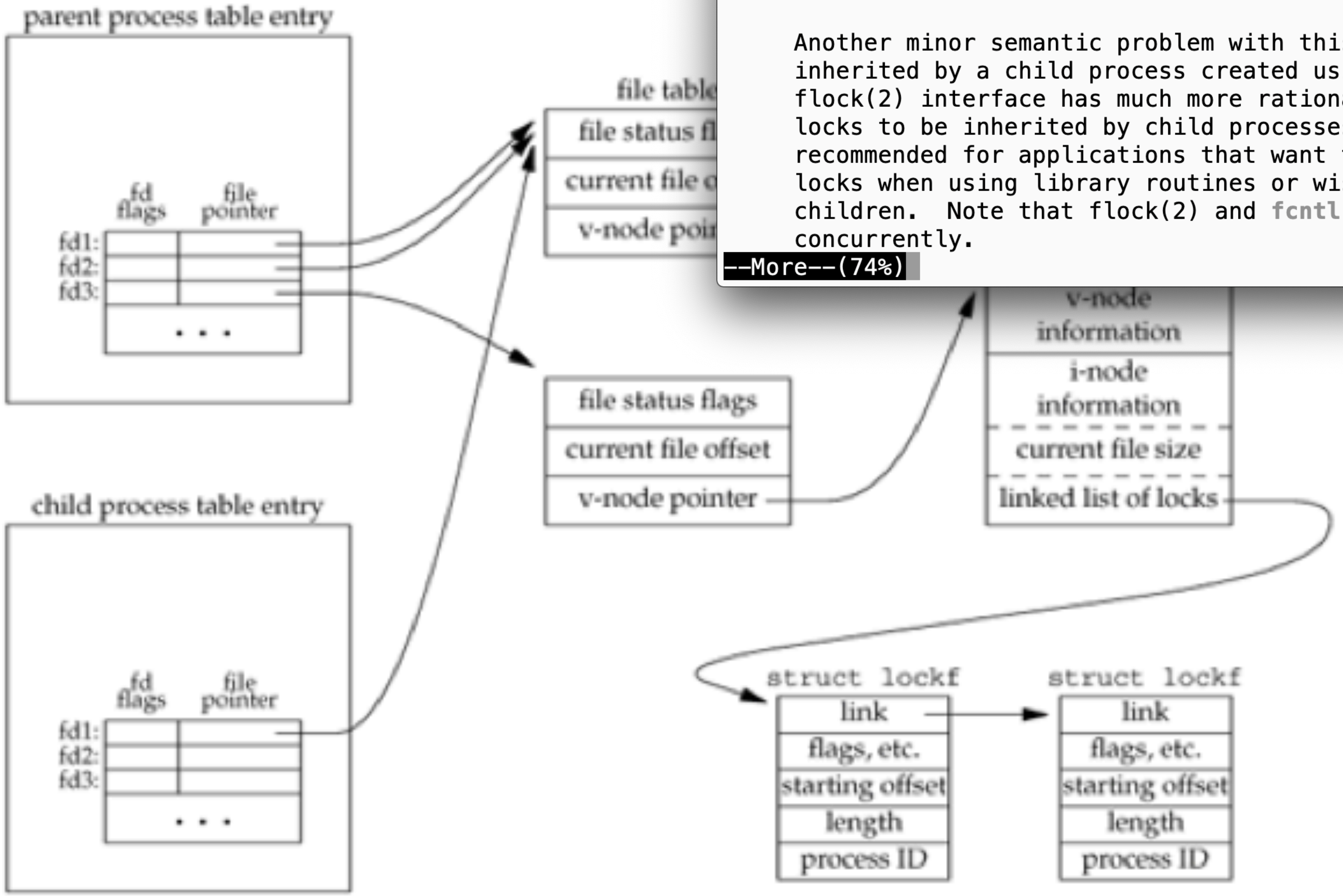
Advisory Record Locking

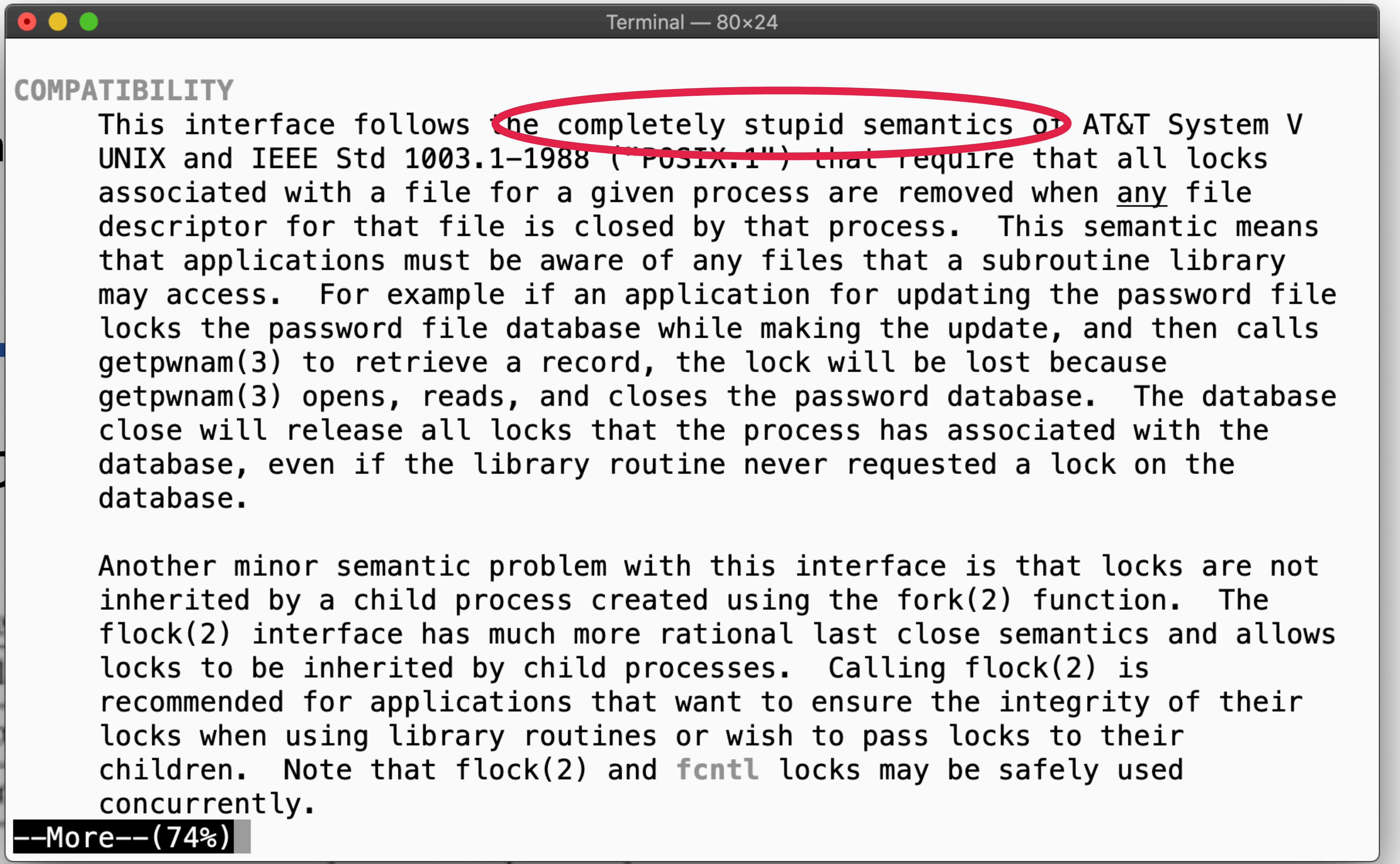
Locks are:

- not inherited across `fork(2)`
- inherited across `exec(2)`
- released upon `exec(2)` if `close-on-exec` is set
- released if a process terminates
- released if a filedescriptor is closed (!)

Advisory Record Locking

Locks are associated with a *file and process p*





```
COMPATIBILITY
     This interface follows the completely stupid semantics of AT&T System V
     UNIX and IEEE Std 1003.1-1988 ("POSIX.1") that require that all locks
     associated with a file for a given process are removed when any file
     descriptor for that file is closed by that process.  This semantic means
     that applications must be aware of any files that a subroutine library
     may access.  For example if an application for updating the password file
     locks the password file database while making the update, and then calls
     getpwnam(3) to retrieve a record, the lock will be lost because
     getpwnam(3) opens, reads, and closes the password database.  The database
     close will release all locks that the process has associated with the
     database, even if the library routine never requested a lock on the
     database.

     Another minor semantic problem with this interface is that locks are not
     inherited by a child process created using the fork(2) function.  The
     flock(2) interface has much more rational last close semantics and allows
     locks to be inherited by child processes.  Calling flock(2) is
     recommended for applications that want to ensure the integrity of their
     locks when using library routines or wish to pass locks to their
     children.  Note that flock(2) and fcntl locks may be safely used
     concurrently.
--More--(74%)
```

"Mandatory” Locking

- not implemented on all UNIX flavors
    - `chmod g+s,g-x file`

- possible to be circumvented:

```
$ mandatory-lock /tmp/file &
$ echo foo > /tmp/file2
$ rm /tmp/file
$ mv /tmp/file2 /tmp/file
```

Resource Locking

- Most locking mechanisms discussed here are *advisory*: they require the cooperation of the processes.
- Any form of locking carries the risk of a deadlock; code carefully and defensively to account for this!
- Try locking STDOUT for concurrent writes from multiple processes. Try locking streams.
- Verify that locks are per file descriptor, allowing e.g., a parent process to lose a lock when a child unlocks it.
- Rewrite flock.c to use fcntl(2).
- What happens if you try to lock a region that extends beyond the current end of the file?
- How do flock(2) and fcntl(2) locks interact?