Programmation Systèmes
Cours 8 — IPC: File Locking

Stefano Zacchiroli
zack@pps.jussieu.fr

Laboratoire PPS, Université Paris Diderot - Paris 7

15 novembre 2011

Outline

Outline

1. Synchronization and file locking
2. `flock`
3. `fcntl`-based locking

Process synchronization

Consider again the need of distributing unique sequential identifiers. We would like to implement it without a central server, relying on process cooperation.

- we can store the global counter in a shared file
- each processes can access 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 need of distributing unique sequential identifiers. We would like to implement it without a central server, relying on process cooperation.

- we can store the global counter in a shared file
- each processes can access 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:

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 are also used for general process synchronization, given its simplicity and the pervasiveness of filesystem on UNIX

Advisory and mandatory locking

We speak about mandatory locking when the locking system forbids a process to perform I/O, unless it has obtained a specific file lock.

We speak about advisory locking when acquiring locks before proceeding into the critical section is a convention agreed upon by cooperating processes.

Traditionally, on UNIX the most common kind of file locking is advisory locking. We will focus on it.

File locking APIs

There are two main APIs for placing file locks on UNIX:

- `flock`, which places locks on entire files
- `fcntl` (AKA record locking), which can be used to place locks on regions of files
  - `fcntl` offers a *superset* of `flock`'s features, but it's also plagued by some design issues...

Pick one, do not mix the two!

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 often used for general process synchronization.
- `fcntl`-based locking descends from System V circa 1984 and is nowadays a popular file locking API.
- POSIX.1 chose to standardize the `fcntl` approach

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

Note: 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.

Lock kinds. Two *alternative* kinds of locks are supported:

- a shared lock (AKA “read lock”) can be hold by several processes at a time
- an exclusive lock (AKA “write lock”) can be hold by only one process at a time and also inhibits the presence of shared locks

Blocking behavior. If the request lock cannot be granted, the process will be blocked. Unblocking will happen as soon as the lock can be granted, atomically with it.
To avoid blocking, the flag LOCK_NB can be used. With it, instead of blocking, `flock` will fail with `errno` set to EWOULDBLOCK.

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

flock — example

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

flock context

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

- they are neither stored in the file descriptor itself
- nor stored in filesystem i-nodes

This choice has 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".[1]

```
flock(fd1, LOCK_EX);
fd2 = dup(fd);
flock(fd2, LOCK_UN);
```

[1] as flock is not standard, we show BSD / Linux behavior

flock inheritance gotchas

flock context can result in surprising behavior, if we're not careful. Here are some common "gotchas".[1]

```
flock(fd1, LOCK_EX);
fd2 = dup(fd);
flock(fd2, LOCK_UN);
```

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

[1] as flock is not standard, we show 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);
```

flock inheritance gotchas (cont.)

```
fd1 = open("foo.txt", O_RDWR);
fd2 = open("foo.txt", O_RDWR);
flock(fd1, LOCK_EX);
flock(fd2, LOCK_EX);
```

- there are two different locks, as separate open create separate entries in the open file table (possibly for the same file)
- the 2nd `flock` will block (forever. . .) as we are trying to acquire two *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)
        flock(fd, LOCK_UN);
```

flock inheritance gotchas (cont.)

```
fd = open("foo.txt", O_RDWR);
flock(fd, LOCK_EX);
if (fork() == 0)
        flock(fd, LOCK_UN);
```

- 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. To do so, after fork the parent should close its file descriptor, leaving the child in control of the lock.

flock — limitations

flock suffers from a number of limitations, that have motivated standardizing fnctl-based locking only.

- granularity: only entire files can be locked
  - This is fine when files are used only as rendez-vous points for synchronized access to something else.
  - But it is a serious limitation when synchronizing for shared file I/O.
- flock can only do advisory locking
- the implementations of important file systems (such as NFS) do not support flock locks
  - e.g. in Linux's NFS, support for flock locks is available only since version 2.6.12

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".
A different model is "1 syscall ↔ many actions", i.e. using 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 of the above trade-off:

pro less clutter in the API namespace

pro easier to extend, by adding new commands

cons diminished code—and API doc.—readability

cons 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

```c
#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)

```c
#include <fcntl.h>
#include <unistd.h>
#include "apue.h"

#define HELLO   "Hello, "
#define WORLD   "World!\n"
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, 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[2] is close-on-exec. It states whether the file descriptor should be 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

[2] in fact, the only one defined

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

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

`fcntl` locking

Contrary wrt what happened with `flock`, where locks were global, with `fcntl` process can place locks on byte ranges of an open file, referenced by a FD.

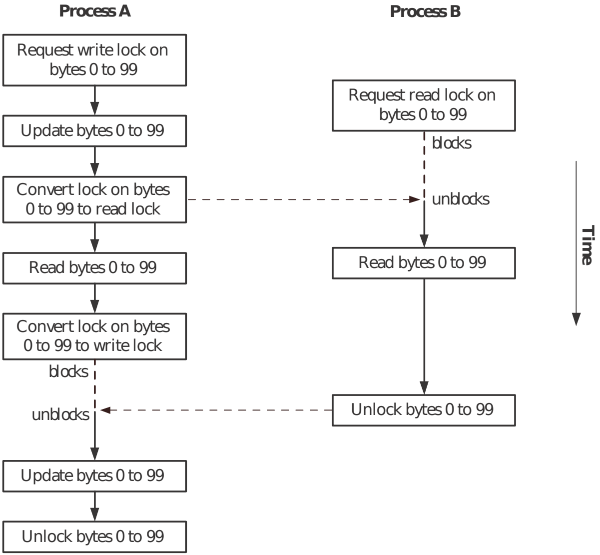
Two kinds of locks are supported: write locks (equivalent to `flock`'s exclusive locks) and read locks (eq. to shared locks).

`fcntl` locking is often 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

`fcntl` locking is also called POSIX locking, as it is the mechanism blessed by POSIX.1 to do file locking.

fcntl locking — sample usage



TLPI, Figure 55-2

`fcntl` 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 `fcntl` the principle remains, but the granularity is the byte range, sized from 1 byte to the entire file.

- locks are requested/released on byte ranges
- conceptually, 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)

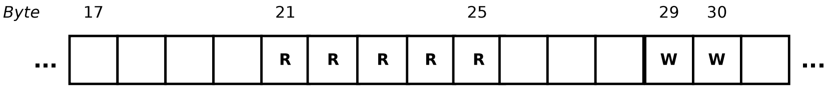


Figure: two ranges locked: one with a read lock (bytes 21–25), the other with a write lock (bytes 29–30)

fcntl locking — model (cont.)

Internally, the kernel represents fcntl locks as ranges, to minimize the size of the representation.
Automatic split and merge of ranges is performed when needed.

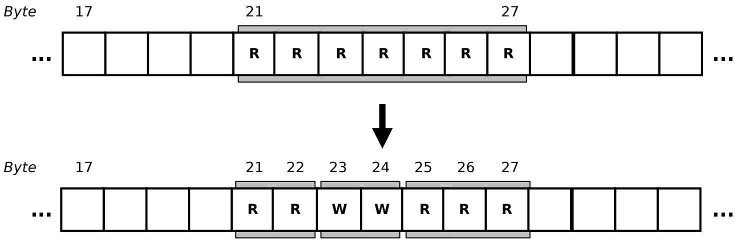


Figure: taking a write lock in between a previously read-lock-ed range

A (Linux-specific) peek on the 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
```

`fcntl` locking — permissions

In some sense, with `flock` locks the file is mostly used as a *rendez-vous* point among processes for synchronization purposes.

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

With `fcntl` 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 from a file
- to be able to put a write lock, a process needs write permission to a file

`fcntl` locking — ranges

Ranges for `fcntl` 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

`fcntl` locking — ranges

Ranges for `fcntl` 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.

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

fcntl 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) */
}
```

fcntl 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.

fcntl 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

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.

fcntl locking — example

As an example of fcntl locking, we give an implementation of the distributed scheme to assign sequential unique identifiers.

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 here?)
4. update the counter
5. release the lock

`fcntl` 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.

fcntl locking — example (protocol)

```
#include <fcntl.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include "apue.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 */
};
```

fcntl 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);
}
```

fcntl 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 */
```

fcntl 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 */
```

fcntl 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");
```

fcntl 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);

	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 */
```

fcntl 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

fcntl locking — release and inheritance

fcntl locks are associated with both a process and an i-node. That is substantially different from flock locks. 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 the ⟨pid, i-node⟩ keys are the same

Inheritance

- fcntl locks are preserved through exec
- fcntl locks are not inherited upon fork
  - there is no way to atomically pass locks to children :-(

Release

- when a process close a FD, all its fcntl locks on the corresponding file are released
  - there is no way to fool this (dup, dup2, etc.) :-(
  - particularly bad for library code that need to hand out FDs :-(

Deadlock. . .

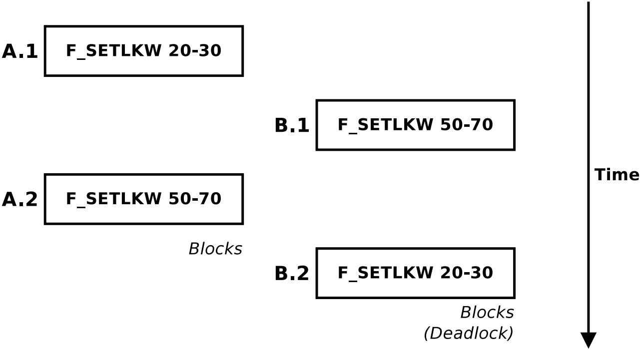
Consider two processes doing the following:

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 `fcntl` locking, 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 "apue.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("fcntl error");

        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 */
```

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 heisenbug
- on Linux, it is the most recent `fcntl` invocation that will fail; SUSv3 gives no such guarantee

lockf

SUSv3 offers a wrapper function to ease record locking:

```
#include <unistd.h>

int lockf(int fd, int cmd, off_t len);
```

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

`lockf` locks a sequence of bytes of length `len`, starting at the current file offset. Locks can be requested with the cmd-s F_LOCK (blocking) and F_TLOCK (non-blocking, “T” for “try”), released with F_ULOCK, and tested with F_TEST.

Unfortunately, SUSv3 does not specify whether `lockf` is implemented in terms of `fcntl` and hence its possible interactions with `fcntl`.