EE3233 Systems Programming for Engrs

Reference: M. Kerrisk, The Linux Programming Interface

Lecture 17

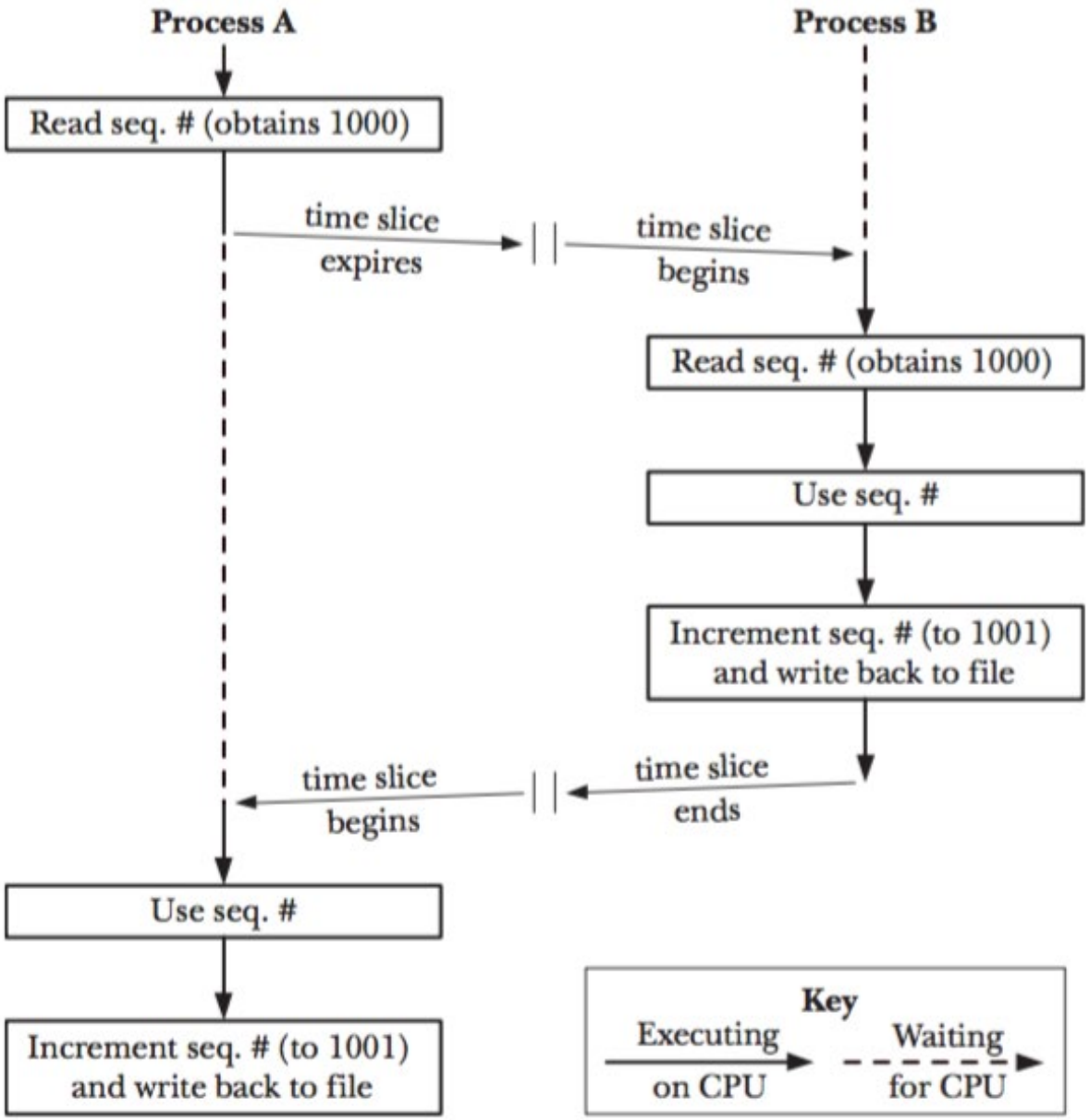
System V File Locking

ECE Electrical & Computer Engineering

Overview

- As long as just one process at a time ever uses a file, then no problem
- However, what if multiple processes are simultaneously updating a file
- Suppose that each process performs following steps to update a file containing a sequence number
  1. Read the sequence number from the file
  2. Use the sequence number for some application-defined purpose
  3. Increment the sequence number and write it back to the file

Problem with no synchronization



Problem with no synchronization

- The file contains the value 1001 at the end of these steps
  - It should contain the value 1002
- To prevent, we need synchronization
  - We could use semaphore to perform the required synchronization, but
  - Using file locks is usually preferable because kernel automatically associates locks with files
- Two API for placing file locks
  - *flock()* : places locks on entire files
  - *fcntl()* : places locks on regions of a file
- General method of using *flock()* and *fcntl()* is as follows:
  - Place a lock on the file
  - Perform file I/O
  - Unlock the file so that another file process can lock it

Advisory and Mandatory Locking

- Advisory
  - By default, file locks are advisory
  - A process can simply ignore a lock placed by another process
  - Each process accessing the file must cooperate for the advisory scheme to work: place a lock before performing file I/O
- Mandatory
  - Forces a process performing I/O to abide by the locks held by other processes

File Locking with *flock()*

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

Int flock (int fd, int operation);
                                    returns 0 on success, or -1 on error
```

- Places a single lock on an entire file
- *fd* : file to be locked is specified via an open descriptor passed in *fd*
- *operation* : one of values
  - **LOCK_SH** : Place a shared lock on the file referred to by *fd*
  - **LOCK_EX** : Place an exclusive lock on the file referred to by *fd*
  - **LOCK_UN** : Unlock the file referred to by *fd*
  - **LOCK_NB** : Make a nonblocking lock request

File Locking with *flock()*

- Any number of processes may simultaneously hold a shared lock on a file
  - However, only one process at a time can hold an exclusive lock on a file
- A process can place a shared or exclusive lock regardless of the access mode of the file

Process A	Process B	
	LOCK_SH	LOCK_EX
LOCK_SH	Yes	No
LOCK_EX	No	No

Example: Using flock()

filelock/t_flock.c

```
#include <sys/file.h>
#include <fcntl.h>
#include "curr_time.h"                  /* Declaration of currTime() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int fd, lock;
    const char *lname;

    if (argc < 3 || strcmp(argv[1], "--help") == 0 ||
            strchr("sx", argv[2][0]) == NULL)
        usageErr("%s file lock [sleep-time]\n"
                "    'lock' is 's' (shared) or 'x' (exclusive)\n"
                "        optionally followed by 'n' (nonblocking)\n"
                "    'secs' specifies time to hold lock\n", argv[0]);

    lock = (argv[2][0] == 's') ? LOCK_SH : LOCK_EX;
    if (argv[2][1] == 'n')
        lock |= LOCK_NB;
```

```
fd = open(argv[1], O_RDONLY);                /* Open file to be locked */
if (fd == -1)
    errExit("open");

lname = (lock & LOCK_SH) ? "LOCK_SH" : "LOCK_EX";

printf("PID %ld: requesting %s at %s\n", (long) getpid(), lname,
        currTime("%T"));

if (flock(fd, lock) == -1) {
    if (errno == EWOULDBLOCK)
        fatal("PID %ld: already locked - bye!", (long) getpid());
    else
        errExit("flock (PID=%ld)", (long) getpid());
}

printf("PID %ld: granted    %s at %s\n", (long) getpid(), lname,
        currTime("%T"));

sleep((argc > 3) ? getInt(argv[3], GN_NONNEG, "sleep-time") : 10);

printf("PID %ld: releasing  %s at %s\n", (long) getpid(), lname,
        currTime("%T"));
if (flock(fd, LOCK_UN) == -1)
    errExit("flock");

exit(EXIT_SUCCESS);
```

Results

```
$ touch tfile
$ ./t_flock tfile s 60 &
[1] 9777
PID 9777: requesting LOCK_SH at 21:19:37
PID 9777: granted    LOCK_SH at 21:19:37
$ ./t_flock tfile s 2
PID 9778: requesting LOCK_SH at 21:19:49
PID 9778: granted    LOCK_SH at 21:19:49
PID 9778: releasing  LOCK_SH at 21:19:51

$ ./t_flock tfile xn
PID 9779: requesting LOCK_EX at 21:20:03
PID 9779: already locked - bye!

$ ./t_flock tfile x
PID 9780: requesting LOCK_EX at 21:20:21
PID 9777: releasing  LOCK_SH at 21:20:37
PID 9780: granted    LOCK_EX at 21:20:37
PID 9780: releasing  LOCK_EX at 21:20:47
```

Semantics of Lock Inheritance and Release

- Locks are automatically released when the corresponding file descriptor is closed
- When a file descriptor is duplicated (via *dup()*, *dup2()*, *fcntl()* **F_DUPFD** operation), the new file descriptor refers to the same file lock

```
flock (fd, LOCK_EX);          /* Gain lock via ‘fd’ */
newfd = dup (fd);             /* ‘newfd’ refers to same lock as ‘fd’ */
flock (newfd, LOCK_UN);       /* Frees lock acquired via ‘fd’ */
```

Semantics of Lock Inheritance and Release

- When we create a child process using fork(), that child obtains duplicates of its parent's file descriptors
- Following code causes a child to remove a parent's lock

```
flock (fd, LOCK_EX);              /* Parent obtains lock */
If (fork() == 0)                  /* If child …  */
      flock (fd, LOCK_UN);        /* Release lock shared with parent */
```

- After *fork()*, the parent closes its file descriptor, and the lock is under sole control of the child process

Limitations of *flock()*

- Only whole files can be locked
  - Such coarse locking limits the potential for concurrency among cooperating processes
  - For example, multiple processes would like to simultaneously access different parts of the same file, then locking via *flock()* would needlessly prevent these processes from operating concurrently

Record Locking with *fcntl()*

- Places a lock on any part of a file ranging from a single byte to the entire file

Record Locking with *fcntl()*

- Places a lock on any part of a file ranging from a single byte to the entire file

General form of the *fcntl()*

```
struct flock flockstr;

    fcntl (fd, cmd, &flockstr);
```

Lock types for *fcntl()* locking

Lock type	Description
F_RDLCK	Place a read lock
F_WRLCK	Place a write lock
F_UNLCK	Remove an existing lock

flock structure

- struct flock {

    short *l_type*;

        /* Lock type: F_RDLCK, F_WRLCK, F_UNLCK */

    short *l_whence*;

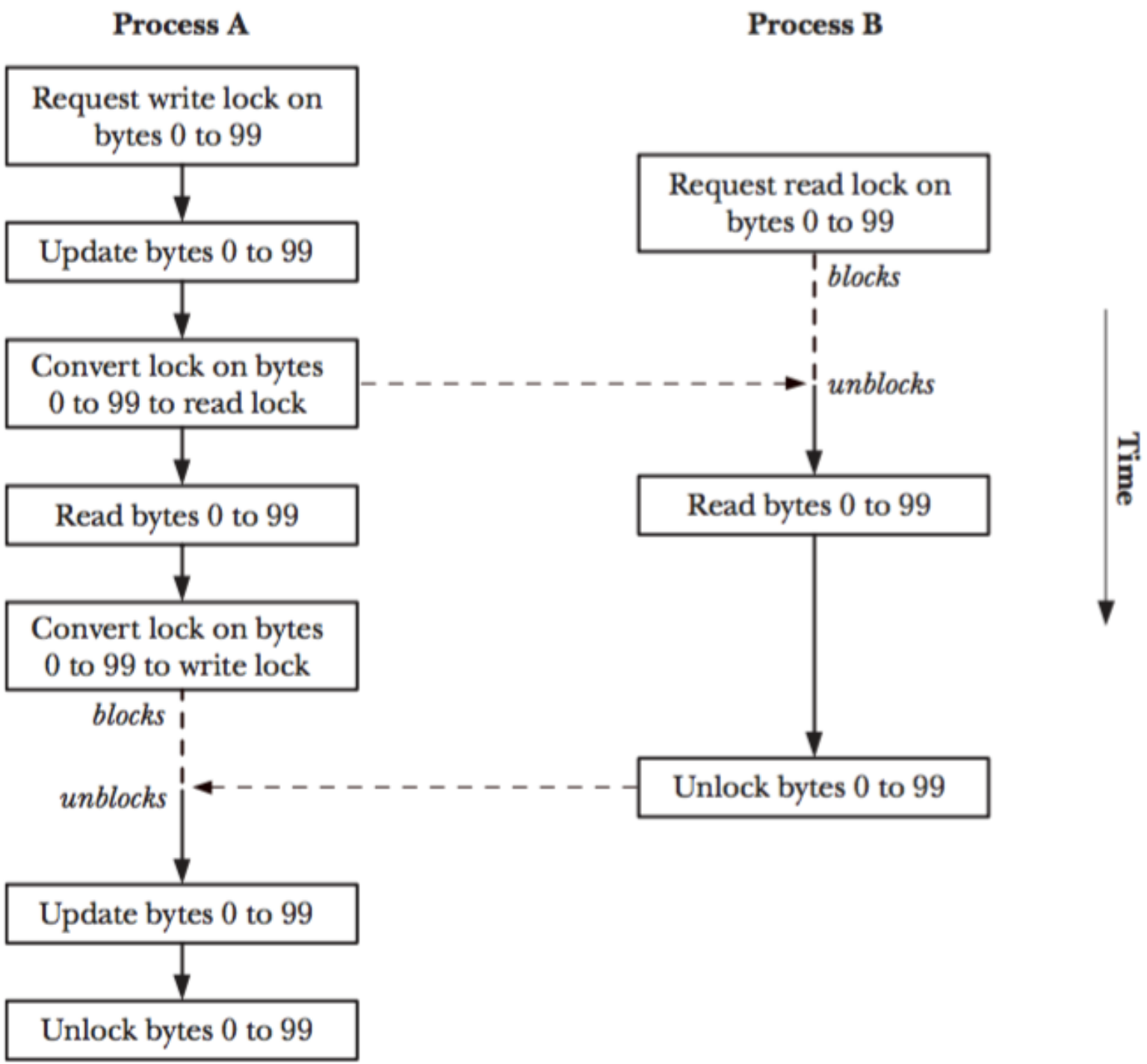
        /* How to interpret 'l_start': SEEK_SET, SEEK_CUR, SEEK_END */

    off_t *l_start*; /* Offset where the lock begins */

    off_t *l_len*; /* Number of bytes to lock; 0 means "until EOF" */

    pid_t *l_pid*; /* Process preventing our lock */

  }


Process A
Process B
Request write lock on bytes 0 to 99
Update bytes 0 to 99
Convert lock on bytes 0 to 99 to read lock
Read bytes 0 to 99
Convert lock on bytes 0 to 99 to write lock
blocks
unblocks
Update bytes 0 to 99
Unlock bytes 0 to 99
Request read lock on bytes 0 to 99
blocks
unblocks
Read bytes 0 to 99
Unlock bytes 0 to 99
Time

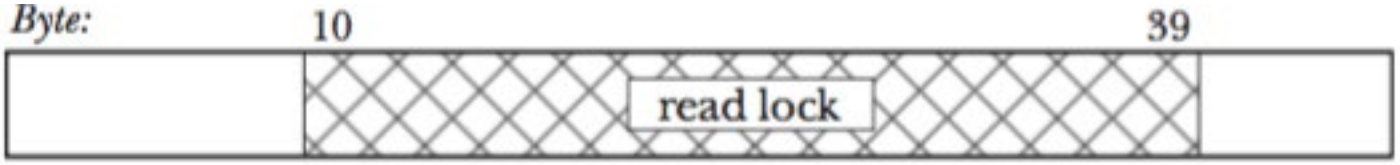
cmd argument

- **F_SETLK**
  - Acquire (*l_type* is **F_RDLCK** or **F_WRLCK**) or release (*l_type* is **F_UNLCK**)
  - If an incompatible lock is held by another process on any part of the region to be locked, *fcntl()* fails
- **F_SETLKW**
  - Same as **F_SETLK**, except that if another process holds an incompatible lock on any part of the region to be blocked, then the call blocks until the lock can be granted
- **F_GETLK**
  - Check if it would be possible to acquire the lock specified in *flockstr*, but don't actually acquire it
  - If the lock would be permitted, then **F_UNLCK** is returned in the *l_type*
  - If locks exist on the region, then *flockstr* returns information about the lock including *l_type*, *l_start*, *l_len*, *l_whence*, *l_pid*

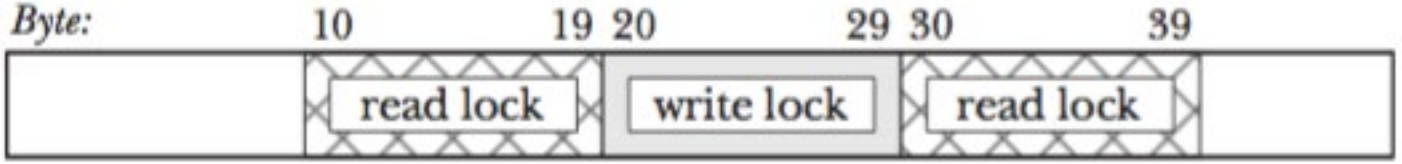
fcntl()

- Unlocking a file region always immediately succeeds
- At any time, a process can hold just one type of lock on a particular region of a file
- A process can never lock itself out of a file region
- Placing a lock of a different mode in the middle of a lock we already hold results in three locks

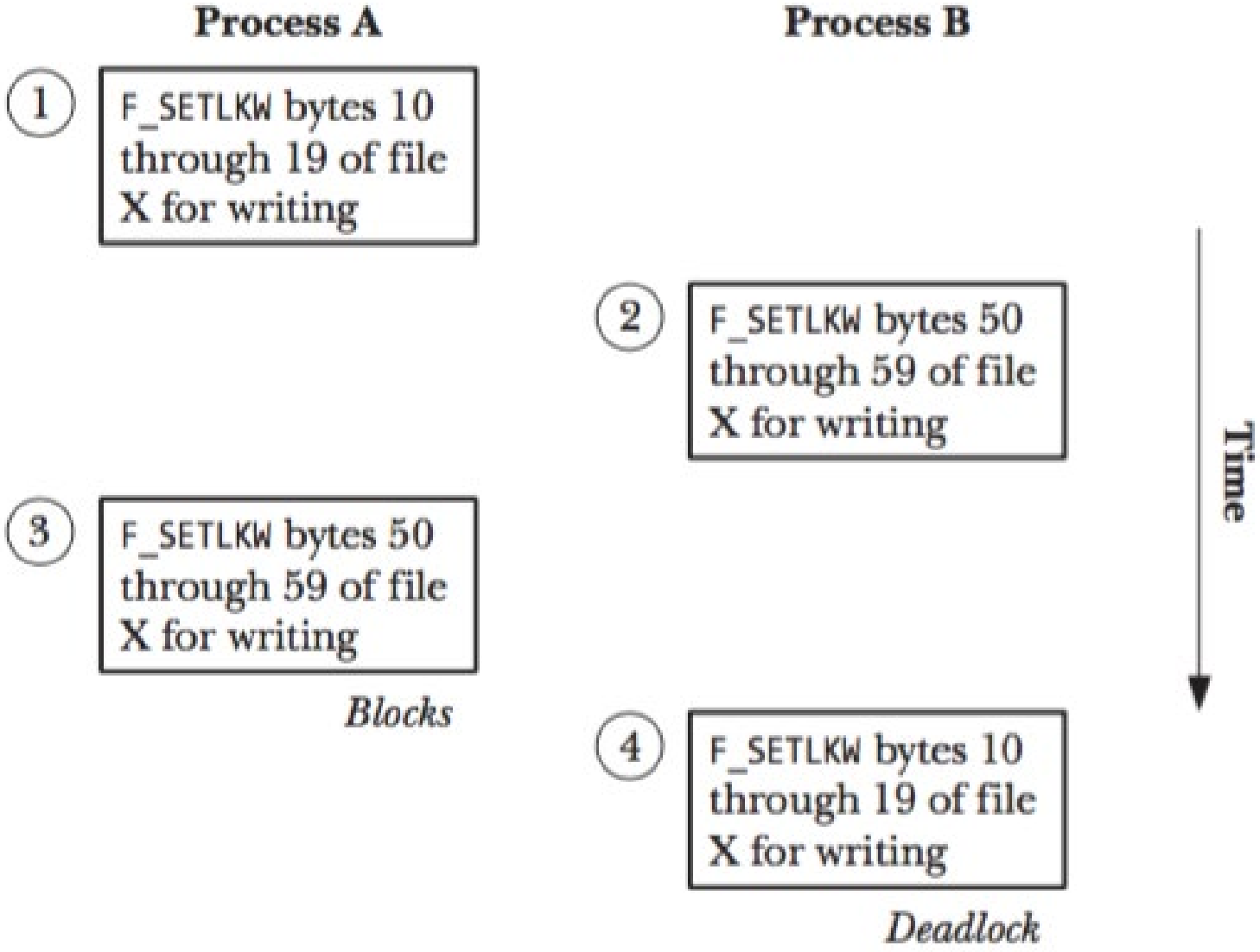
1. After placing read lock (l_start = 10, l_len = 30)



2. After placing write lock (l_start = 20, l_len = 10)



deadlock()


Process A
Process B
1
F_SETLKW bytes 10 through 19 of file X for writing
2
F_SETLKW bytes 50 through 59 of file X for writing
3
F_SETLKW bytes 50 through 59 of file X for writing
Blocks
4
F_SETLKW bytes 10 through 19 of file X for writing
Deadlock
Time

Example: An Interactive Locking Program

filelock/i_fcntl_locking.c

```
#include <sys/stat.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

#define MAX_LINE 100

static void
displayCmdFmt(void)
{
    printf("\n    Format: cmd lock start length [whence]\n\n");
    printf("    'cmd' is 'g' (GETLK), 's' (SETLK), or 'w' (SETLKW)\n");
    printf("    'lock' is 'r' (READ), 'w' (WRITE), or 'u' (UNLOCK)\n");
    printf("    'start' and 'length' specify byte range to lock\n");
    printf("    'whence' is 's' (SEEK_SET, default), 'c' (SEEK_CUR), "
           "or 'e' (SEEK_END)\n\n");
}
```

```
int
main(int argc, char *argv[])
{
    int fd, numRead, cmd, status;
    char lock, cmdCh, whence, line[MAX_LINE];
    struct flock fl;
    long long len, st;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s file\n", argv[0]);

    fd = open(argv[1], O_RDWR);
    if (fd == -1)
        errExit("open (%s)", argv[1]);

    printf("Enter ? for help\n");

    for (;;) {          /* Prompt for locking command and carry it out */
        printf("PID=%ld> ", (long) getpid());
        fflush(stdout);

        if (fgets(line, MAX_LINE, stdin) == NULL)       /* EOF */
            exit(EXIT_SUCCESS);
        line[strlen(line) - 1] = '\0';          /* Remove trailing '\n' */

        if (*line == '\0')
            continue;                           /* Skip blank lines */

        if (line[0] == '?') {
            displayCmdFmt();
            continue;
        }

        whence = 's';                   /* In case not otherwise filled in */
```

```
numRead = sscanf(line, "%c %c %lld %lld %c", &cmdCh, &lock,
                 &st, &len, &whence);
fl.l_start = st;
fl.l_len = len;

if (numRead < 4 || strchr("gsw", cmdCh) == NULL ||
        strchr("rwu", lock) == NULL || strchr("sce", whence) == NULL) {
    printf("Invalid command!\n");
    continue;
}

cmd = (cmdCh == 'g') ? F_GETLK : (cmdCh == 's') ? F_SETLK : F_SETLKW;
fl.l_type = (lock == 'r') ? F_RDLCK : (lock == 'w') ? F_WRLCK : F_UNLCK;
fl.l_whence = (whence == 'c') ? SEEK_CUR :
              (whence == 'e') ? SEEK_END : SEEK_SET;

status = fcntl(fd, cmd, &fl);           /* Perform request... */
```

```
        if (cmd == F_GETLK) {                   /* ... and see what happened */
            if (status == -1) {
                errMsg("fcntl - F_GETLK");
            } else {
                if (fl.l_type == F_UNLCK)
                    printf("[PID=%ld] Lock can be placed\n", (long) getpid());
                else                            /* Locked out by someone else */
                    printf("[PID=%ld] Denied by %s lock on %lld:%lld "
                            "(held by PID %ld)\n", (long) getpid(),
                            (fl.l_type == F_RDLCK) ? "READ" : "WRITE",
                            (long long) fl.l_start,
                            (long long) fl.l_len, (long) fl.l_pid);
            }
        } else {                /* F_SETLK, F_SETLKW */
            if (status == 0)
                printf("[PID=%ld] %s\n", (long) getpid(),
                        (lock == 'u') ? "unlocked" : "got lock");
            else if (errno == EAGAIN || errno == EACCES)        /* F_SETLK */
                printf("[PID=%ld] failed (incompatible lock)\n",
                        (long) getpid());
            else if (errno == EDEADLK)                          /* F_SETLKW */
                printf("[PID=%ld] failed (deadlock)\n", (long) getpid());
            else
                errMsg("fcntl - F_SETLK(W)");
        }
    }
}
```

Example: An Interactive Locking Program

```
Terminal window 1
$ ls -l tfile
-rw-r--r--    1 mtk      users         100 Apr 18 12:19 tfile
$ ./i_fcntl_locking tfile
Enter ? for help
PID=790> s r 0 40
[PID=790] got lock
```

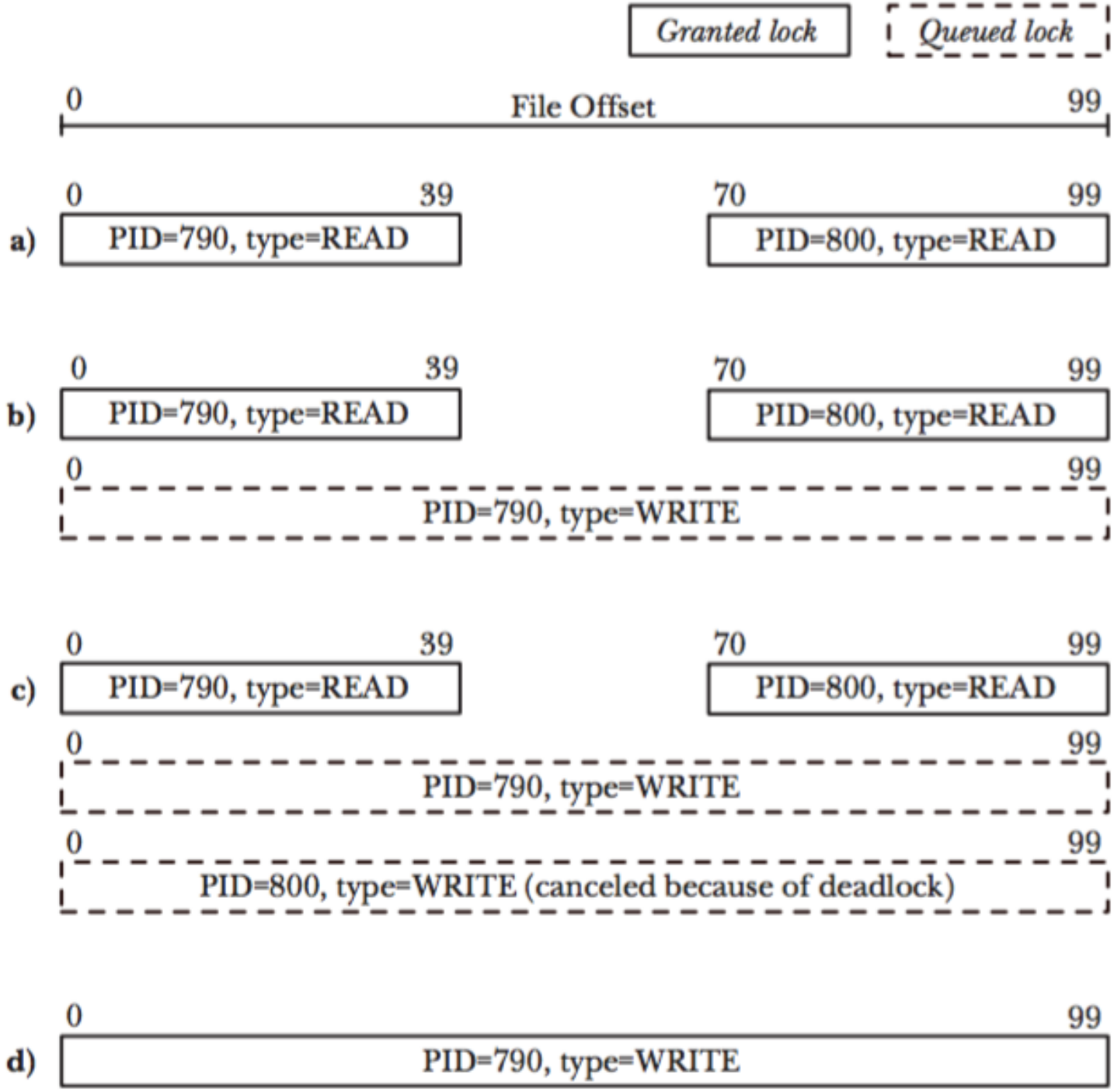
```
Terminal window 2
$ ./i_fcntl_locking tfile
Enter ? for help
PID=800> s r -30 0 e
[PID=800] got lock
```

```
PID=790> g w 0 0
[PID=790] Denied by READ lock on 70:0 (held by PID 800)
PID=790> s w 0 0
[PID=790] failed (incompatible lock)
PID=790> w w 0 0
```

```
PID=800> g w 0 0
[PID=800] Denied by READ lock on 0:40
(held by PID 790)
PID=800> w w 0 0
[PID=800] failed (deadlock)
```

```
PID=800> s u 0 0
[PID=800] unlocked
```

```
[PID=790] got lock
```


Granted lock
Queued lock
0
File Offset
99
a)
0
39
PID=790, type=READ
70
99
PID=800, type=READ
b)
0
39
PID=790, type=READ
70
99
PID=800, type=READ
0
99
PID=790, type=WRITE
c)
0
39
PID=790, type=READ
70
99
PID=800, type=READ
0
99
PID=790, type=WRITE
0
99
PID=800, type=WRITE (canceled because of deadlock)
d)
0
99
PID=790, type=WRITE