# Systems Programming

Prof. Shih-Wei Li

Department of Computer Science and Information Engineering
National Taiwan University

# What we will cover today?

- File/Record lock
- Blocking/Non-Blocking I/Os, and I/O multiplexing
- Network programming
- Programming Assignment 1

# File Accesses From Multiple Processes

• *SEEK_SET*: the file's offset is set to *offset* bytes from the beginning of the file

Process A

Process B

```
lseek ( fd, 100, SEEK_SET);
read ( fd, &int_var, 4 );
int_var = int_var - 200;
```

- More than one process may access the same file at the same time
  - Think about the examples for atomic operations that we showed last week

```
lseek ( fd, 100, SEEK_SET);
read ( fd, &int_var, 4 );
int_var = int_var - 200;
lseek ( fd, 100, SEEK_SET);
write ( fd, &int_var, 4 );
```

```
lseek ( fd, 100, SEEK_SET);
write ( fd, &int_var, 4 );
```

# File Accesses From Multiple Processes

- More than one process may access the same file at the same time
  - Think about the examples for atomic operations that we showed last week
  - ***Problem:*** a process may overwrite the old data while that old data is still being read by another process

Process A

```
lseek ( fd, 100, SEEK_SET);
read ( fd, &int_var, 4 );
int_var = int_var - 200;
```

Process B

```
lseek ( fd, 100, SEEK_SET);
read ( fd, &int_var, 4 );
int_var = int_var - 200;
lseek ( fd, 100, SEEK_SET);
write ( fd, &int_var, 4 );
```

```
lseek ( fd, 100, SEEK_SET);
write ( fd, &int_var, 4 );
```

# File/Record Locking in Unix

- Unix provides lock support for avoiding file inconsistency:
  - ***File locking:*** a mechanism that ensures exclusive access to the entire files
  - ***Record locking (or byte range locking)***: a mechanism that ensures exclusive access to only a specified segment (ranges of bytes) of a file

# File/Record Locking in Unix

- Unix System allows a process to *exclusively lock* a file to prevent other processes from reading from or writing to the same file
  - When a lock is granted to a process, the process has the right to read/write the file
  - When a lock is denied, the process cannot get the lock until the lock holder releases the lock
- Three ways to use file/record locking in Unix to support *advisory locking* (important)
  - *file locking*:
    - *flock()*: lock *an entire file*
  - *record locking*:
    - *fcntl()*: lock *regions (arbitrary byte ranges) in a file*
    - *lockf()*: built on top of *fcntl()*, providing simplified interface

# Advisory Locking v.s. Mandatory Locking

- Advisory lock:
  - fcntl(), flock(), lockf()
  - A cooperative locking scheme, all participating processes must follow the locking protocol; a process calls the file lock functions when accessing shared files
  - Problem: cannot prevent a process not holding an advisory lock from accessing shared files
- Mandatory lock:
  - The OS kernel checks and verifies every open, read, and write against a shared file that the calling process does not violate a lock
  - Mandatory lock is not part of SUS:
    - Linux 2.4.22 and Solaris 0 provide mandatory record locking (Linux implementation is unreliable!), FreeBSD and Mac OS X 10.3 do not

# Advisory Locking v.s. Mandatory Locking

Assume that lock() is to support advisory locking

Process 1

```
int fd = open(PATH, ..);
// use lock() to acquire a lock
int ret = lock(fd, …);
if (!ret)
{
    // Lock acquired. Read the file now
    while (count)
        int bytes = read(fd, buf, sizeof(buf));
}
ret = lock(fd, …);
close(fd);
```

Process 2

```
int fd = open(PATH, ..);
// do not use lock()
int bytes = write(fd, buf, sizeof(buf));
close(fd);
```

Process 2 can write to the file (specified by PATH) and ignore the advisory lock that Process 1 holds.

https://loonytek.com/2015/01/15/advisory-file-locking-differences-between-posix-and-bsd-locks/

# Advisory Locking v.s. Mandatory Locking

Assume that lock() is to support advisory locking

Process 1

```
int fd = open(PATH, ..);
// use lock() to acquire a lock
int ret = lock(fd, …);
if (!ret)
{
    // Lock acquired. Read the file now
    while (count)
        int bytes = read(fd, buf, sizeof(buf));
}
ret = lock(fd, …);
close(fd);
```

Process 2

```
int fd = open(PATH, ..);
// use lock() to acquire a lock
int ret = lock(fd, …);
if (!ret)
{
    int bytes = write(fd, buf, sizeof(buf));
}
ret = lock(fd, …);
close(fd);
```

Process 2 can only write the file (specified by PATH) only if it gets the write exclusive lock.

https://loonytek.com/2015/01/15/advisory-file-locking-differences-between-posix-and-bsd-locks/

# flock

```
#include <sys/file.h>
// returns: 0 on success, –1 on error
int flock(int fd, int operation);
```

- flock() apply or remove a lock on an open file
- The argument operation is one of the following:
  - *LOCK_SH*: place a *shared lock* - more than one process may hold a shared lock for a given file at a given time
  - *LOCK_EX*: place an *exclusive lock* - only one process may hold an exclusive lock for a given file at a given time
  - *LOCK_UN*: remove an existing lock held by this process

# flock

```
int main(int argc, char *argv[]) {
    int fd = open(argv[1], O_RDWR | O_CREAT, 0666);
    if (fd == -1) {
        exit(1);
    }

    if (flock(fd, LOCK_EX) == -1) {
        close(fd);
        exit(1);
    }

    .....

    if (flock(fd, LOCK_UN) == -1) {
        close(fd);
        exit(1);
    }

    close(fd);
    return 0;
}
```

# File I/O: fcntl

- … is the ISO C way to specify that the number and types of the remaining arguments may vary

```
#include <fcntl.h>
// The value returned on a success call depends on cmd; -1 is returned on error
int fcntl(int fd, int cmd, … /* int arg */ );
```

- *fcntl()* performs an operation (specified by *cmd*) for an open file descriptor *fd*
- There are 11 different *cmd* values supported by fcntl for five different purposes (check Chapter 3.14 for more details)
  - Duplicate an existing descriptor (just as dup!)
  - Get/set *file descriptor flags* (the “fd flags” from the per-process file descriptor table entry)
  - Get/set *file status flags* (from the open file table entry)
  - Get/set asynchronous I/O ownership
  - ***Get/set file record locks***
- *fcntl()* can take an optional third argument *arg*, determined by *cmd*

# Rewind: Unix kernel support for File I/O

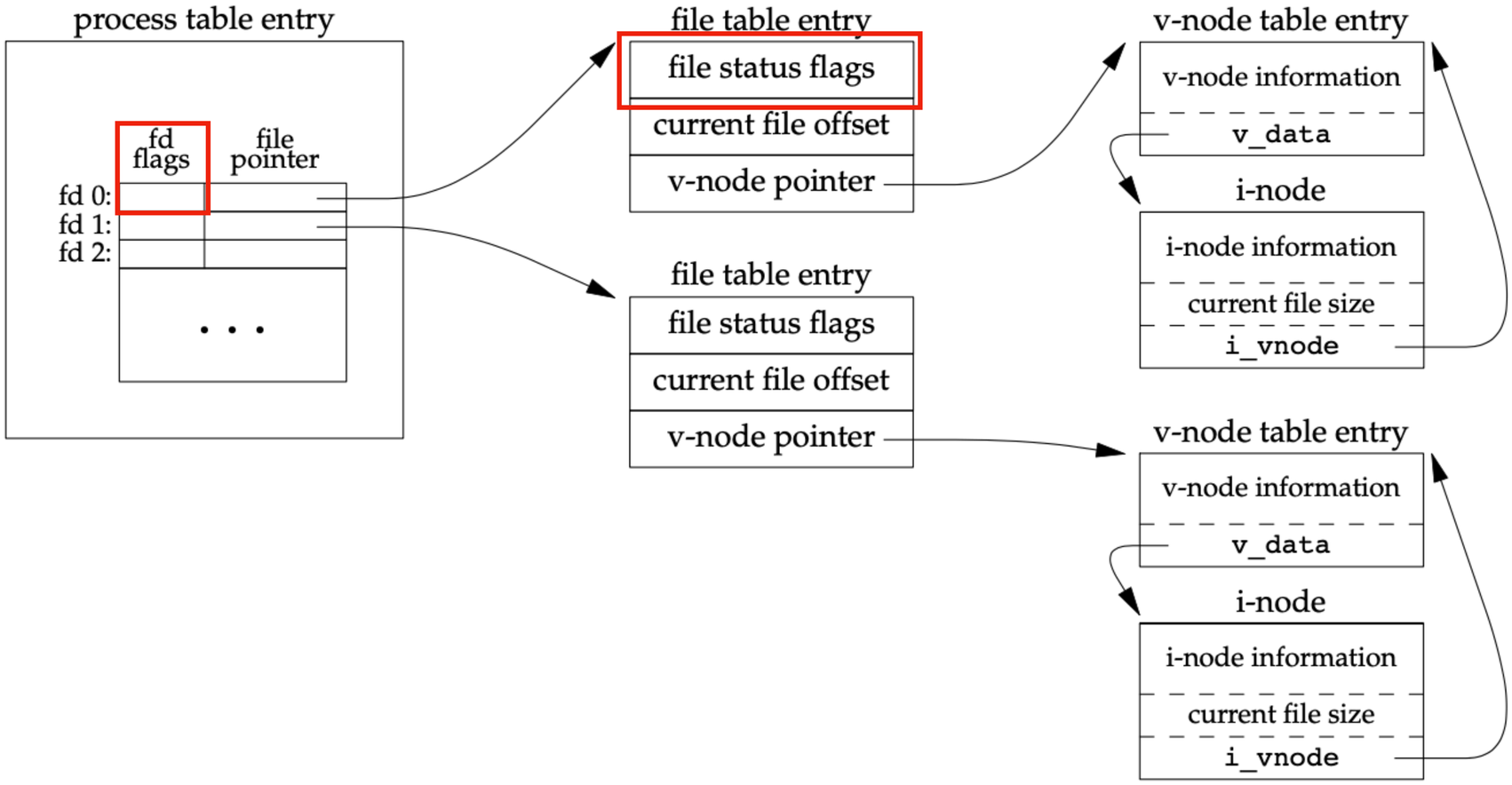



**Figure 3.7** Kernel data structures for open files

# File I/O: fcntl

```
#include <fcntl.h>
// The value returned on a success call depends on cmd if OK; -1 is returned on error
int fcntl(int fd, int cmd, … /* int arg */ );
```

- *fcntl()* returns the file status flags for *fd*: *cmd*: F_GETFL
- *fcntl()* sets the file status flag to the value of the third argument : *cmd*: F_SETFL

| File status flag | Description |
|---|---|
| O_RDONLY | open for reading only |
| O_WRONLY | open for writing only |
| O_RDWR | open for reading and writing |
| O_EXEC | open for execute only |
| O_SEARCH | open directory for searching only |
| O_APPEND | append on each write |
| O_NONBLOCK | nonblocking mode |
| O_SYNC | wait for writes to complete (data and attributes) |
| O_DSYNC | wait for writes to complete (data only) |
| O_RSYNC | synchronize reads and writes |
| O_FSYNC | wait for writes to complete (FreeBSD and Mac OS X only) |
| O_ASYNC | asynchronous I/O (FreeBSD and Mac OS X only) |

**Figure 3.10** File status flags for `fcntl`

# File I/O: fcntl

```
#include        <sys/types.h>
#include        <fcntl.h>
#include        "ourhdr.h"

int
main(int argc, char *argv[])
{
        int             accmode, val;

        if (argc != 2)
                err_quit("usage: a.out <descriptor#>");

        if ( (val = fcntl(atoi(argv[1]), F_GETFL, 0)) < 0)
                err_sys("fcntl error for fd %d", atoi(argv[1]));

        accmode = val & O_ACCMODE;
        if      (accmode == O_RDONLY)   printf("read only");
        else if (accmode == O_WRONLY)   printf("write only");
        else if (accmode == O_RDWR)             printf("read write");
        else err_dump("unknown access mode");

        if (val & O_APPEND)             printf(", append");
        if (val & O_NONBLOCK)           printf(", nonblocking");
#if     !defined(_POSIX_SOURCE) && defined(O_SYNC)
        if (val & O_SYNC)               printf(", synchronous writes");
#endif
        putchar('\n');
        exit(0);
}
```

# File I/O: fcntl

```
#include        <sys/types.h>
#include        <fcntl.h>
#include        "ourhdr.h"

int
main(int argc, char *argv[])
{
        int             accmode, val;

        if (argc != 2)
                err_quit("usage: a.out <descriptor#>");

        if ( (val = fcntl(atoi(argv[1]), F_GETFL, 0)) < 0)
                err_sys("fcntl error for fd %d", atoi(argv[1]));

        accmode = val & O_ACCMODE;
        if      (accmode == O_RDONLY)   printf("read only");
        else if (accmode == O_WRONLY)   printf("write only");
        else if (accmode == O_RDWR)             printf("read write");
        else err_dump("unknown access mode");

        if (val & O_APPEND)             printf(", append");
        if (val & O_NONBLOCK)           printf(", nonblocking");
#if     !defined(_POSIX_SOURCE) && defined(O_SYNC)
        if (val & O_SYNC)               printf(", synchronous writes");
#endif
        putchar('\n');
        exit(0);
}
```

```
$ ./a.out 0 < /dev/tty
read only
$ ./a.out 1 > temp.foo
$ cat temp.foo
write only
$ ./a.out 2 2>>temp.foo
write only, append
$ ./a.out 5 5<>temp.foo
read write
```

# File I/O: fcntl

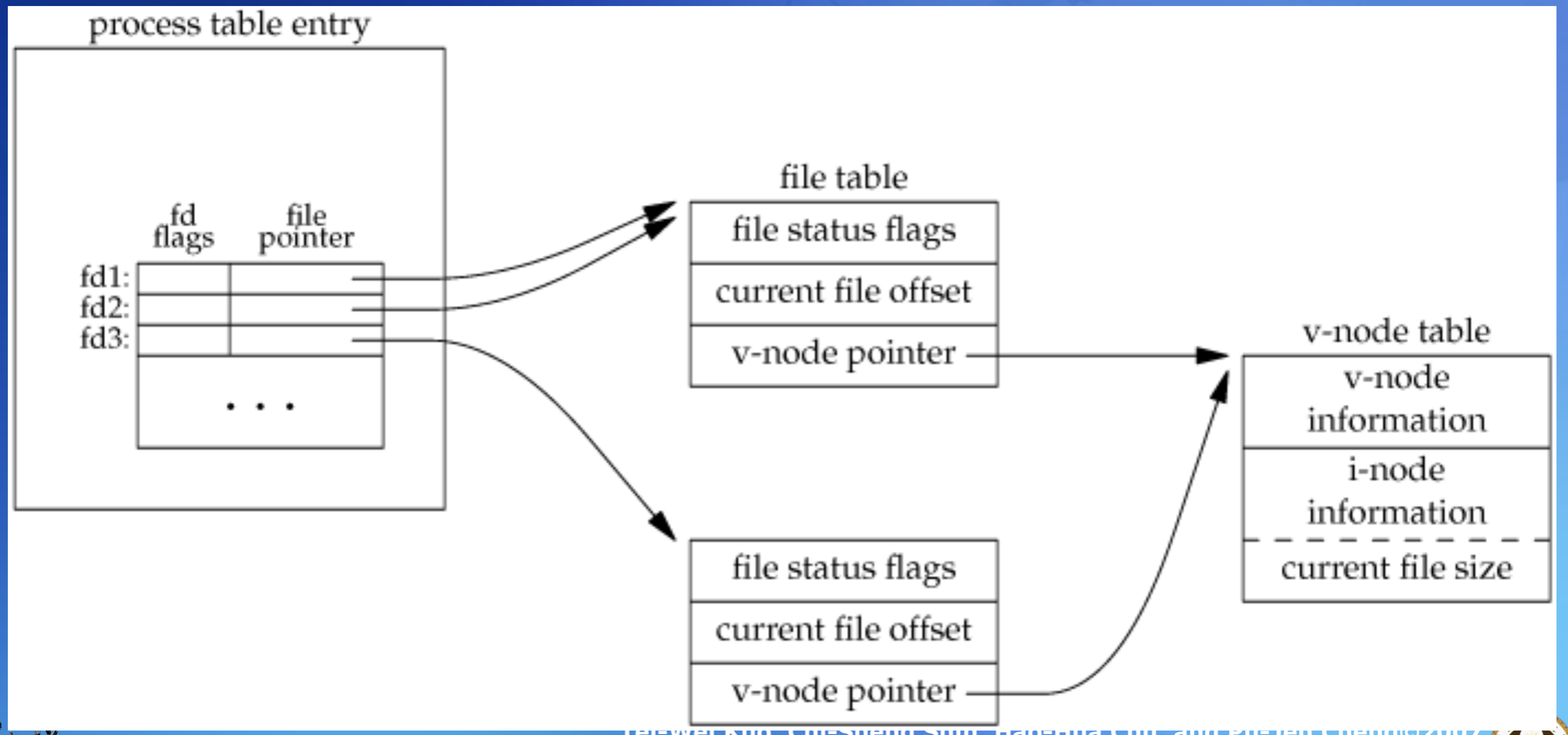
```
#include "apue.h"
#include <fcntl.h>

void
set_fl(int fd, int flags) /* flags are file status flags to turn on */
{
    int     val;

    if ((val = fcntl(fd, F_GETFL, 0)) < 0)
        err_sys("fcntl F_GETFL error");

    val |= flags;       /* turn on flags */

    if (fcntl(fd, F_SETFL, val) < 0)
        err_sys("fcntl F_SETFL error");
}
```

```
val &= ~flags;      /* turn flags off */
```

**Figure 3.12** Turn on one or more of the file status flags for a descriptor

# Which descriptors are affected by an fcntl on fd1 with F_SETFD and F_SETFL?

```
fd1 = open(pathname, oflags);
fd2 = dup(fd1);
fd3 = open(pathname, oflags);
```



# fcntl record locking: the parameters

```
#include <fcntl.h>
// depends on cmd if OK, –1 on error
int fcntl(int fd, int cmd, …, /* struct flock *flockptr */ );
```

- The function fcntl supports three commands (specified by *cmd*) for file locking (***record locking***):
- fcntl takes a pointer (flockptr) to a struct ***flock***, which contains the lock information
- *fcntl()* supports three different commands (***cmd***) for record locks: *F_SETLK*, *F_SETLKW*, and *F_GETLK*

```
struct flock {
        short l_type;    /* F_RDLCK, F_WRLCK, F_UNLCK */
        short l_whence; /* SEEK_SET, SEEK_CUR, or SEEK_END, same as
        the whence in lseek*/
        off_t l_start;    /* offset in bytes relative to whence */
        off_t l_len;    /* length, in bytes, 0 means lock to EOF*/
        pid_t l_pid;   /* filled in by F_GETLK, ignore otherwise */
}
```

- *l_type* in the struct flock can be:
  - *F_RDLCK*: a shared read lock
  - *F_WRLCK*: an exclusive write lock
  - *F_UNLCK*: unlocking a region

# fcntl record locking: types of Lock

- ***Shared read lock:***
  - Any number of processes can have a shared read lock on a given byte (or bytes)
  - If there are one or more read locks on a given byte(s), there can't be any write locks on that byte(s)
  - *i.e., supports multiple readers but no writers*
- ***Exclusive write lock:***
  - Only one single process can get an exclusive write lock on a given byte (or bytes)
  - If there is an exclusive write lock on a given byte(s), there cannot be any read/write locks on that byte(s)
  - *i.e., supports a single writer but no reader*

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

**Figure 14.3** Compatibility between different lock types

From: Figure 14.3 in APUE 3rd edition

# fcntl record locking

```
#include <fcntl.h>
// depends on cmd if OK, –1 on error
int fcntl(int fd, int cmd, …, /* struct flock *flockptr */ );
```

```
struct flock {
        short l_type;    /* F_RDLCK, F_WRLCK, F_UNLCK */
        short l_whence; /* SEEK_SET, SEEK_CUR, or SEEK_END, same as
        the whence in lseek*/
        off_t l_start;    /* offset in bytes relative to whence */
        off_t l_len;    /* length, in bytes, 0 means lock to EOF */
        pid_t l_pid;   /* filled in by F_GETLK, ignore otherwise */
}
```

- Supported commands for fcntl:
  - *F_GETLK*: Determine whether the lock described by *flockptr* can be placed on the file:
    - If yes, returns *F_UNLCK* in the l_type field
    - If not (the file is locked), *fcntl()* returns details about one of those locks that prevents the placement: by *updating* the *l_type*, *l_whence*, *l_start*, *l_pid,* and *l_len* fields of *flockptr*

# fcntl record locking

```
#include <fcntl.h>
// depends on cmd if OK, –1 on error
int fcntl(int fd, int cmd, …, /* struct flock *flockptr */ );
```

```
struct flock {
        short l_type;    /* F_RDLCK, F_WRLCK, F_UNLCK */
        short l_whence; /* SEEK_SET, SEEK_CUR, or SEEK_END, same as
        the whence in lseek*/
        off_t l_start;    /* offset in bytes relative to whence */
        off_t l_len;    /* length, in bytes, 0 means lock to EOF*/
        pid_t l_pid;   /* filled in by F_GETLK, ignore otherwise */
}
```

- Supported commanc
  - *F_GETLK*: Determ d on the file:
    - If yes, returns *F*
    - If not (the file is hat prevents the placement: by *flockptr*

**F_GETLK** (*struct flock **)

On input to this call, *lock* describes a lock we would like to place on the file. If the lock could be placed, **fcntl**() does not actually place it, but returns **F_UNLCK** in the *l_type* field of *lock* and leaves the other fields of the structure unchanged.

If one or more incompatible locks would prevent this lock being placed, then **fcntl**() returns details about one of those locks in the *l_type*, *l_whence*, *l_start*, and *l_len* fields of *lock*. If the conflicting lock is a traditional (process-associated) record lock, then the *l_pid* field is set to the PID of the process holding that lock. If the conflicting lock is an open file description lock, then *l_pid* is set to -1. Note that the returned information may already be out of date by the time the caller inspects it.

# fcntl record locking

```
#include <fcntl.h>
// depends on cmd if OK, –1 on error
int fcntl(int fd, int cmd, …, /* struct flock *flockptr */ );
```

```
struct flock {
        short l_type;    /* F_RDLCK, F_WRLCK, F_UNLCK */
        short l_whence; /* SEEK_SET, SEEK_CUR, or SEEK_END, same as
        the whence in lseek*/
        off_t l_start;    /* offset in bytes relative to whence */
        off_t l_len;    /* length, in bytes, 0 means lock to EOF*/
        pid_t l_pid;   /* filled in by F_GETLK, ignore otherwise */
}
```

- Supported commands for fcntl:
  - *F_SETLK*: Set the lock described by *flockptr*; if we fail to set the lock (e.g., the lock is held by another process); fcntl returns -1 immediately and updates the errno (EACCESS or EAGAIN)
    - You can acquire a lock by specifying a lock type of F_RDLCK or F_WRLCK on the bytes specified by the *l_whence*, *l_start*, and *l_len* fields of *flockptr*
    - You can release a lock by specifying a lock type of F_UNLCK

# fcntl record locking

```
#include <fcntl.h>
// depends on cmd if OK, –1 on error
int fcntl(int fd, int cmd, …, /* struct flock *flockptr */ );
```

```
struct flock {
        short l_type;    /* F_RDLCK, F_WRLCK, F_UNLCK */
        short l_whence; /* SEEK_SET, SEEK_CUR, or SEEK_END, same as
        the whence in lseek*/
        off_t l_start;    /* offset in bytes relative to whence */
        off_t l_len;    /* length, in bytes, 0 means lock to EOF*/
        pid_t l_pid;   /* filled in by F_GETLK, ignore otherwise */
}
```

- Supported commands for fcntl:
  - *F_SETLKW*: This command is equivalent to F_SETLK, except that if a shared or exclusive lock is held on the file, the caller shall wait (or block) until the lock to be released

# File/record locking: example use case

- From: https://www.gnu.org/software/libc/manual/html_node/File-Locks.html
- Consider a program that can be run simultaneously by several different users, that logs status information to a common file:
  - A game that uses a file to keep track of high scores
  - A program that records usage or accounting information for billing purposes
- Having multiple copies of the program simultaneously writing to the file could cause the contents of the file to become mixed up
  - Prevent such problem by setting a write lock on the file before actually writing to the file
- If  the program also needs to read the file and wants to make sure that the contents of the file are in a consistent state
  - Use a read lock; while the read lock is set, no other process can lock that part of the file for writing

# fcntl record locking: remarks

- When setting or releasing a lock on a file, the system combines or splits adjacent areas as required

- If we lock byte 150, the kernel coalesces the adjacent locked regions into a single region from byte 100-199

From: Figure 14.4 in APUE 3rd edition

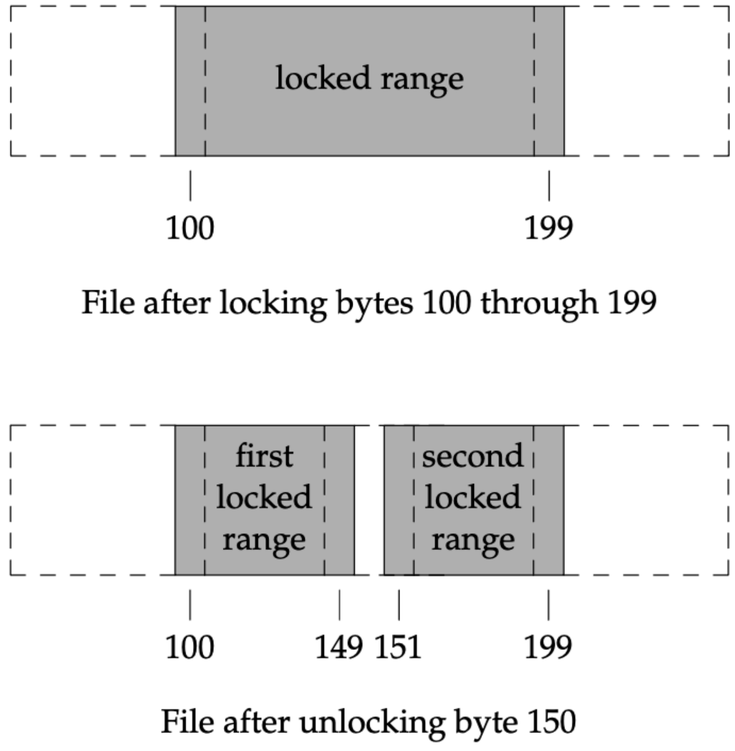



**Figure 14.4** File byte-range lock diagram

# fcntl record locking: remarks

- File Access:
  - To obtain a read lock, the file descriptor must be open for reading
  - To obtain a write lock, the file descriptor must be open for writing

# fcntl record locking: remarks

- Non-atomic operations:
  - Testing for a lock with F_GETLK, and then trying to obtain that lock with F_SETLK or F_SETLKW takes two calls to fcntl():
    - Some process might set the lock between the two fcntl() calls

```c
struct flock fl = {
    .l_type = F_WRLCK,
    .l_whence = SEEK_SET,
    .l_start = 0,
    .l_len = 0
};

// First, check if the file is locked (Assume that fd was opened before here)
if (fcntl(fd, F_GETLK, &fl) == -1) {
    perror("fcntl F_GETLK");
    exit(1);
}

if (fl.l_type == F_UNLCK) {
    // Now try to acquire the lock
    fl.l_type = F_WRLCK;
    if (fcntl(fd, F_SETLK, &fl) == -1) {
        if (errno == EACCES || errno == EAGAIN) {
            printf("Failed to acquire lock (already locked)\n");
        } else {
            perror("fcntl F_SETLK");
            exit(1);
        }
    }  else {
        …
        // Release the lock
        fl.l_type = F_UNLCK;
        if (fcntl(fd, F_SETLK, &fl) == -1) {
            perror("fcntl unlock");
            exit(1);
        }
    }
} else {
    printf("%s: File is already locked\n", process_name);
}
```

# fcntl record locking: remarks

- Release of Locks:
  - When a process terminates, all its locks are released
  - Locks are associated with a process and a file, so *when a file descriptor is closed*, *any locks on the file* referenced by that descriptor for that *process are released*

# fcntl record locking: code examples

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

int
lock_reg(int fd, int cmd, int type, off_t offset, int whence, off_t len)
{
    struct flock    lock;
    lock.l_type = type;     /* F_RDLCK, F_WRLCK, F_UNLCK */
    lock.l_start = offset;  /* byte offset, relative to l_whence */
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;       /* #bytes (0 means to EOF) */
    return(fcntl(fd, cmd, &lock));
}
```

```
#define read_lock(fd, offset, whence, len) \
                    lock_reg((fd), F_SETLK, F_RDLCK, (offset), (whence), (len))
#define write_lock(fd, offset, whence, len) \
                    lock_reg((fd), F_SETLK, F_WRLCK, (offset), (whence), (len))
```

# fcntl record locking: code examples

What happens to the locks on fd1?

```
fd1 = open(pathname, …);
read_lock(fd1, …);
fd2 = dup(fd1);
close(fd2);
```

```
fd1 = open(pathname, …);
read_lock(fd1, …);
fd2 = open(pathname, …);
close(fd2);
```

# fcntl record locking: code examples

What happens to the locks on fd1?

```
fd1 = open(pathname, …);
read_lock(fd1, …);
fd2 = dup(fd1);
close(fd2);
```

If a process closes *any* file descriptor referring to a file, then all of the process's locks on that file are released, regardless of the file descriptor(s) on which the locks were obtained. This is bad: it means that a process can lose its locks on a file such as */etc/passwd* or */etc/mtab* when for some reason a library function decides to open, read, and close the same file.

```
fd1 = open(pathname, …);
read_lock(fd1, …);
fd2 = open(pathname, …);
close(fd2);
```

# Example: FreeBSD Implementation for record locking

The kernel does not know what descriptors own what record lock —> when a file descriptor is closed, the OS kernel releases all of the associate file's file locks (linked to the descriptor's respective inode)

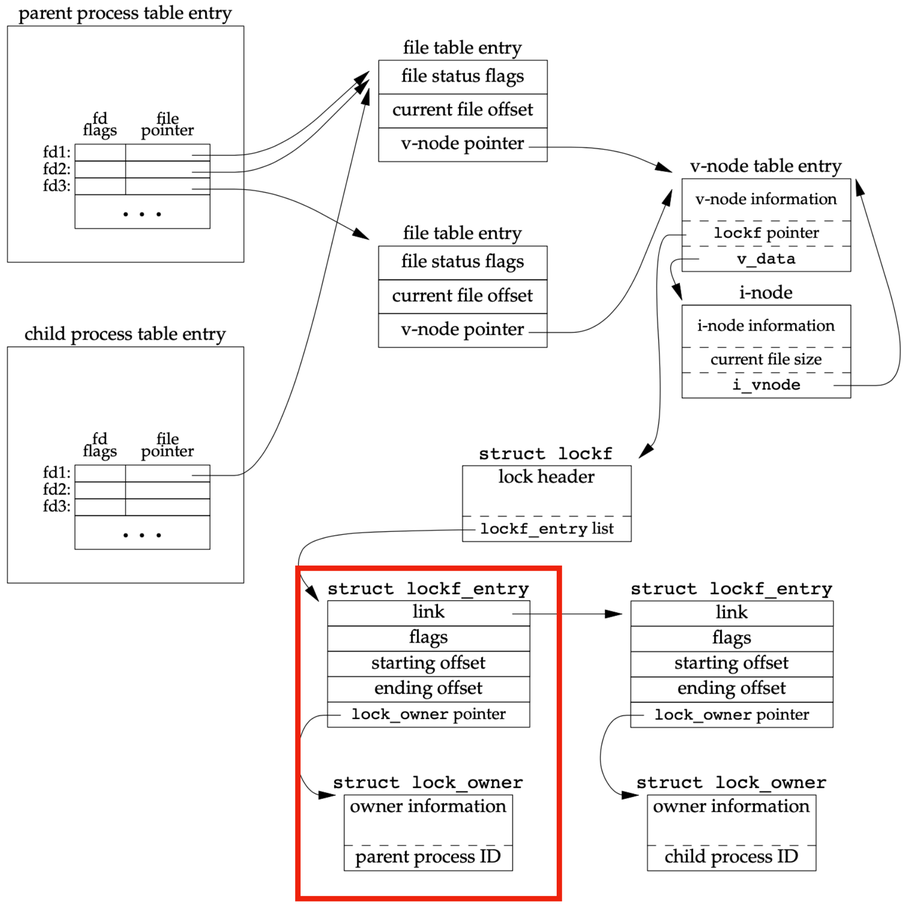



**Figure 14.8** The FreeBSD data structures for record locking

# Demo: lock.c

- To produce a scenario where “lock contention” (i.e., multiple processes try to lock the same file)

# Fast & Slow System Calls

- Fast system calls:
  - Those who take a known amount of time to finish: do not block by external resources
  - Example: read files from a local disk
- Slow system calls:
  - Those who wait for an *indefinite* amount of time to finish (e.g., block forever)
  - Examples: *read*ing from terminal devices or network devices, reading from or writing to a pipe (chap. 15), waiting for a network connection, etc.

# Blocking v.s. Nonblocking I/O

- ***Blocking I/O:***
  - The I/O functions that do not return until their action is completed (slow system calls)
- ***Nonblocking I/O:***
  - Nonblocking I/O lets us issue an I/O operation, such as an open, read, or write, and return as quickly as possible without waiting:
    - Caveat: if the operation cannot be completed, the call returns immediately with an error
    - Ex: non-blocking reads: reads as many bytes as possible without suspending the process
  - It is possible to set a file descriptor to use nonblocking I/O:
    - Two ways: open() or fcntl() (use the cmd: *F_SETFL*) with the *O_NONBLOCK* flag (use with caution)

# Review: Turn on one or more flags with fcntl()

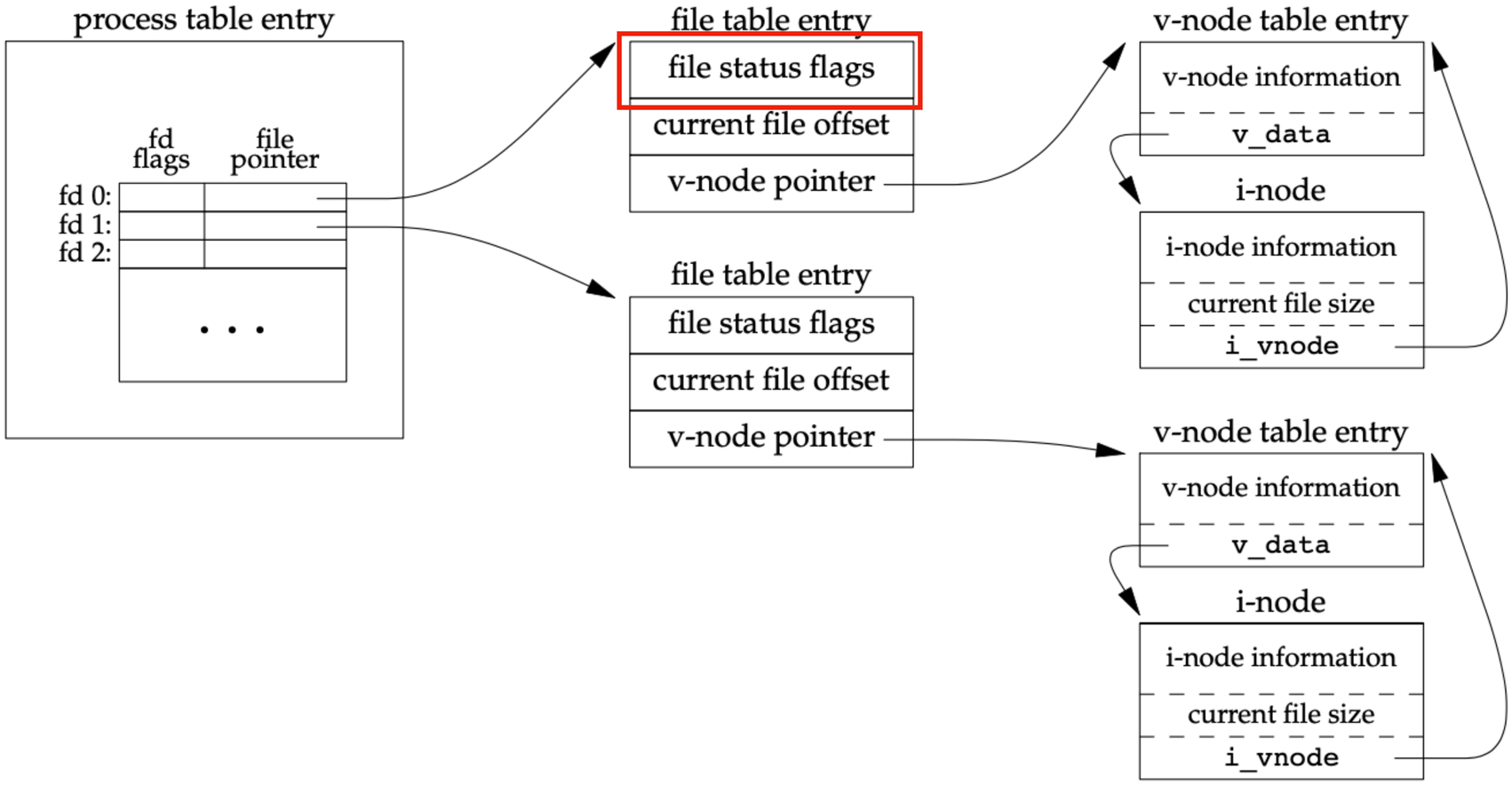
```
#include "apue.h"
#include <fcntl.h>

void
set_fl(int fd, int flags) /* flags are file status flags to turn on */
{
    int     val;

    if ((val = fcntl(fd, F_GETFL, 0)) < 0)
        err_sys("fcntl F_GETFL error");

    val |= flags;       /* turn on flags */

    if (fcntl(fd, F_SETFL, val) < 0)
        err_sys("fcntl F_SETFL error");
}
```

**Figure 3.12** Turn on one or more of the file status flags for a descriptor

**Turn on flags:**
val | flags

**clear the flags:**
val & ~flags

# Review: file status flags of open files




From APUE 3rd Edition: Figure 3.7

**Figure 3.7** Kernel data structures for open files

# Nonblocking I/O Code Example

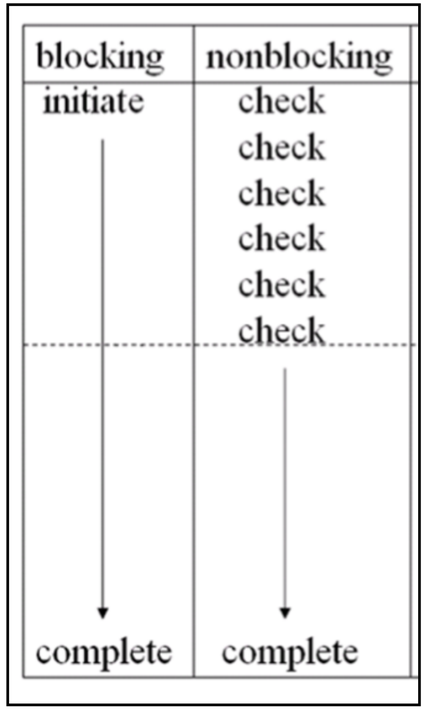
```
#include <errno.h>
#include <fcntl.h>
char    buf[500000];
int main(void)
{
    int     ntowrite, nwrite;
    char    *ptr;
    ntowrite = read(STDIN_FILENO, buf, sizeof(buf));
    fprintf(stderr, "read %d bytes\n", ntowrite);
    set_fl(STDOUT_FILENO, O_NONBLOCK);  /* set nonblocking */
    ptr = buf;
    while (ntowrite > 0) {
        errno = 0;
        nwrite = write(STDOUT_FILENO, ptr, ntowrite);
        fprintf(stderr, "nwrite = %d, errno = %d\n", nwrite, errno);
        if (nwrite > 0) {
            ptr += nwrite;
            ntowrite -= nwrite;
        }
    }
    clr_fl(STDOUT_FILENO, O_NONBLOCK);  /* clear nonblocking */
    exit(0);
}
```

From: Figure 14.1 in APUE 3rd edition

# Nonblocking I/O Code Example

stdout to a regular file, the write executes once

```
$ ls -l /etc/services                                   print file size
-rw-r--r--  1 root     677959 Jun 23  2009 /etc/services
$ ./a.out < /etc/services > temp.file                   try a regular file first
read 500000 bytes
nwrite = 500000, errno = 0                              a single write
$ ls -l temp.file                                       verify size of output file
-rw-rw-r--  1 sar      500000 Apr  1 13:03 temp.file
```




output to the terminal, the write returns a partial count (not 500,000) sometimes and an error at other times

```
$ ./a.out < /etc/services 2>stderr.out          output to terminal
                                                lots of output to terminal ...
$ cat stderr.out
read 500000 bytes
nwrite = 999, errno = 0
nwrite = -1, errno = 35
nwrite = -1, errno = 35
nwrite = -1, errno = 35
nwrite = -1, errno = 35
nwrite = 1001, errno = 0
nwrite = -1, errno = 35
nwrite = 1002, errno = 0
nwrite = 1004, errno = 0
nwrite = 1003, errno = 0
nwrite = 1003, errno = 0
nwrite = 1005, errno = 0
nwrite = -1, errno = 35                         61 of these errors
   :
nwrite = 1006, errno = 0
nwrite = 1004, errno = 0
nwrite = 1005, errno = 0
nwrite = 1006, errno = 0
nwrite = -1, errno = 35                         108 of these errors
   :
nwrite = 1006, errno = 0
nwrite = 1005, errno = 0
nwrite = 1005, errno = 0
nwrite = -1, errno = 35                         681 of these errors
   :
                                                and so on ...
nwrite = 347, errno = 0
```

What is 2>stderr.out?

35 is EAGAIN

# Nonblocking I/O Code Example

```
#include <errno.h>
#include <fcntl.h>
char    buf[500000];
int main(void)
{
    int     ntowrite, nwrite;
    char    *ptr;
    ntowrite = read(STDIN_FILENO, buf, sizeof(buf));
    fprintf(stderr, "read %d bytes\n", ntowrite);
    set_fl(STDOUT_FILENO, O_NONBLOCK);  /* set nonblocking */
    ptr = buf;
    while (ntowrite > 0) {
        errno = 0;
        nwrite = write(STDOUT_FILENO, ptr, ntowrite);
        fprintf(stderr, "nwrite = %d, errno = %d\n", nwrite, errno);
        if (nwrite > 0) {
            ptr += nwrite;
            ntowrite -= nwrite;
        }
    }
    clr_fl(STDOUT_FILENO, O_NONBLOCK);  /* clear nonblocking */
    exit(0);
}
```

polling

From: Figure 14.1 in APUE 3rd edition

# Terminal Device (Ch 18.2)

- Each terminal device has an input queue and an output queue
  - The shell redirects standard input to the terminal
  - The input queue size is bounded by MAX_INPUT
- When the output queue gets filled up:
  - *Blocking:* the process is put to sleep until room is available (the process does nothing)
  - *Nonblocking:* ***polling*** - the program loops and checks if it can do the output
    - A waste of CPU time on a multi-user system because most of the time, there is nothing to be done (i.e., the queue is full)

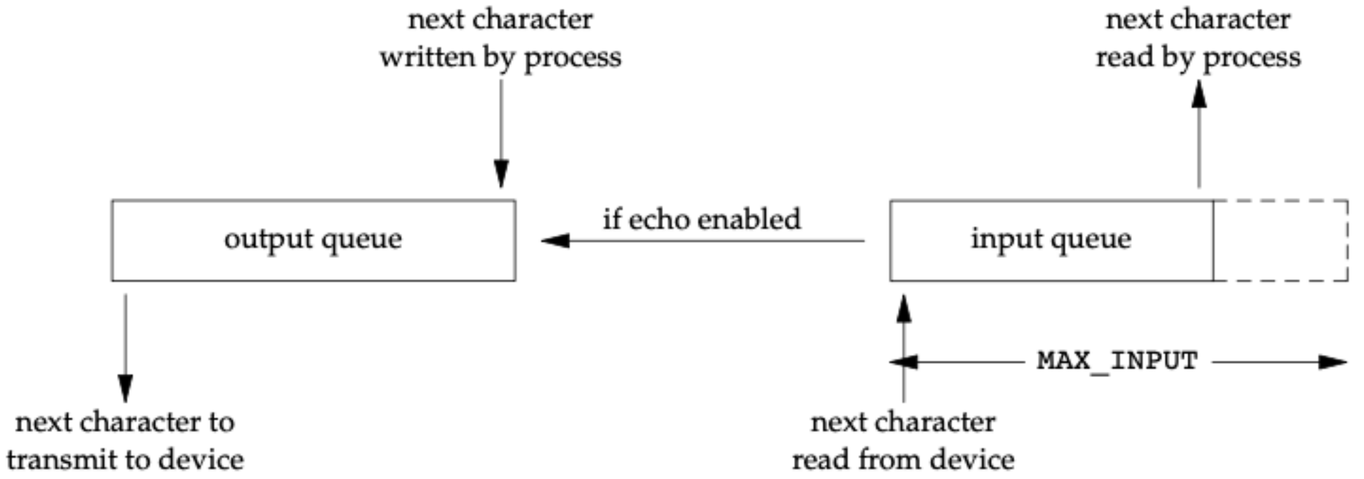
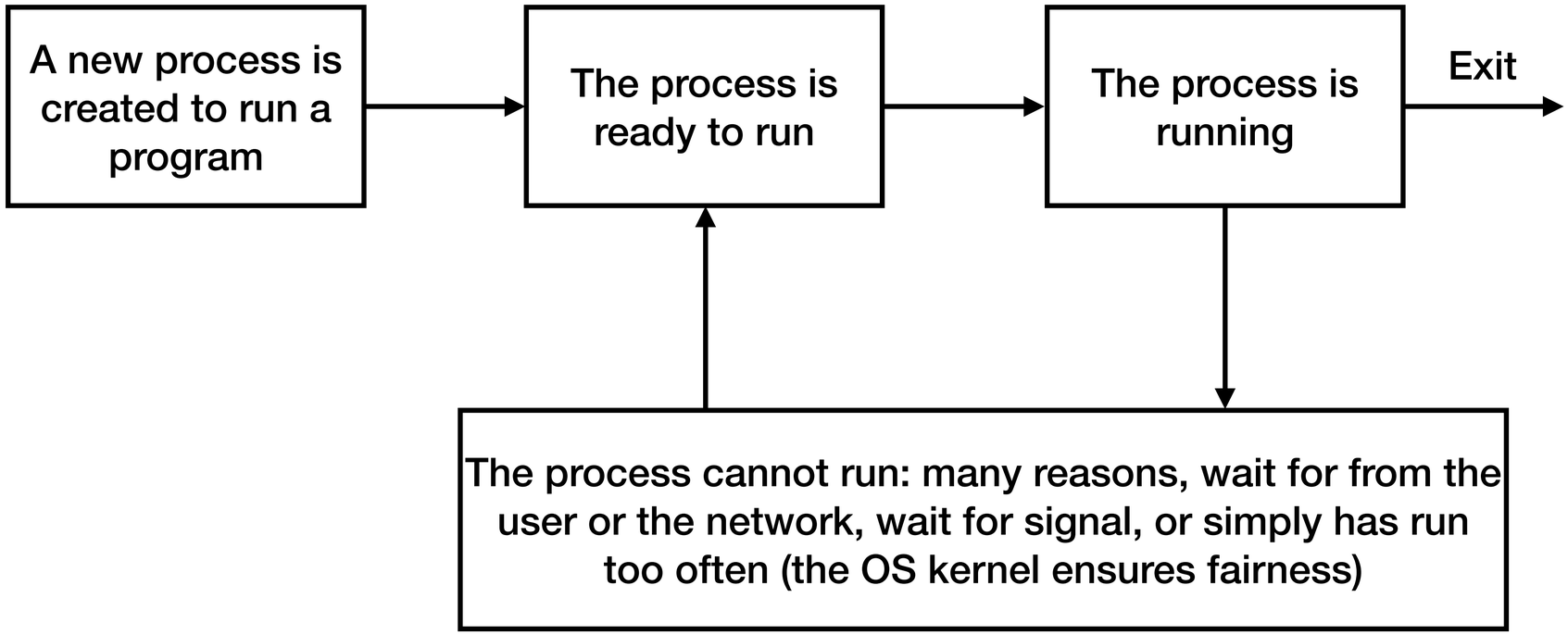



**Figure 18.1** Logical picture of input and output queues for a terminal device

From: Figure 18.1 in APUE 3rd edition

# Review: Process Scheduling

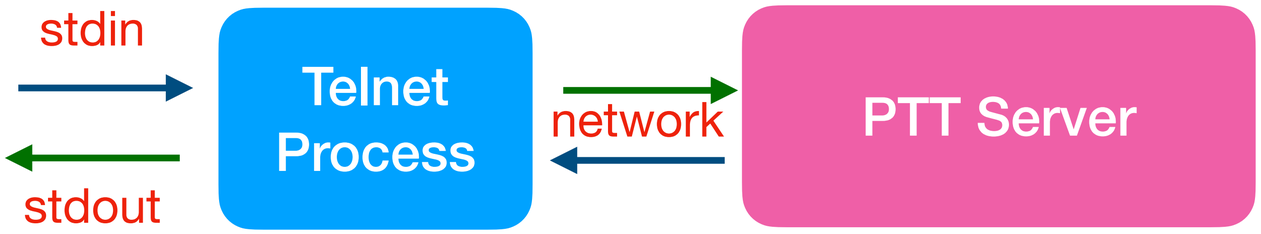
**Process Life Cycle**



# I/O Multiplexing

Telnet Process

```
n = read(STDIN_FILENO, buf, BUFSIZ);
    …
n = read(network_fd, netbuf, BUFSIZ);
    …
```




The telnet process reads from stdin and writes to the network to the telnet server (PTT); the telnet process also reads from the network and writes to stdout

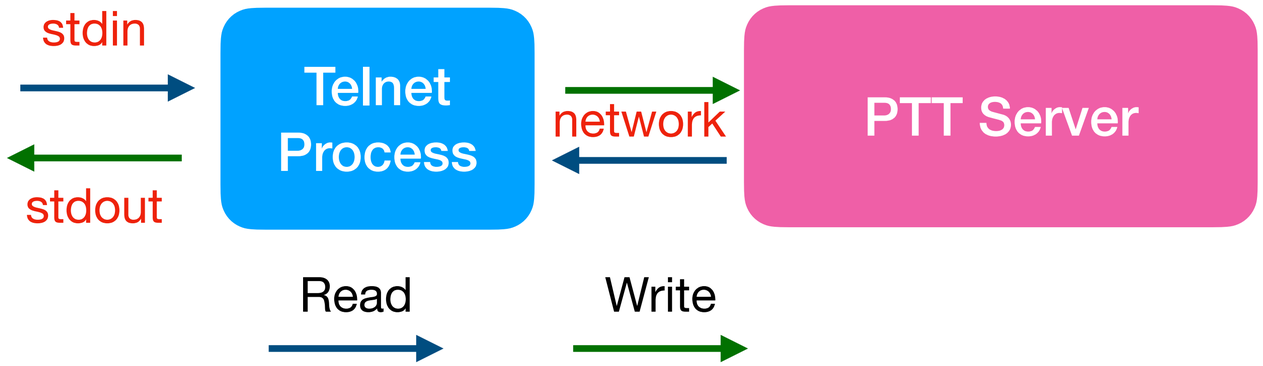
What are the pros/cons when using blocking I/O?

Read two file descriptors: stdin and network input

# I/O Multiplexing

Telnet Process

```
n = read(STDIN_FILENO, buf, BUFSIZ); /* block */
    …
n = read(network_fd, netbuf, BUFSIZ); /* block */
    …
```




**Problem:** blocking I/O here cannot handle multiple I/O descriptors (file/socket descriptors) at the same time — *I/O on any one descriptor can result in blocking — bad for performance*

# I/O Multiplexing

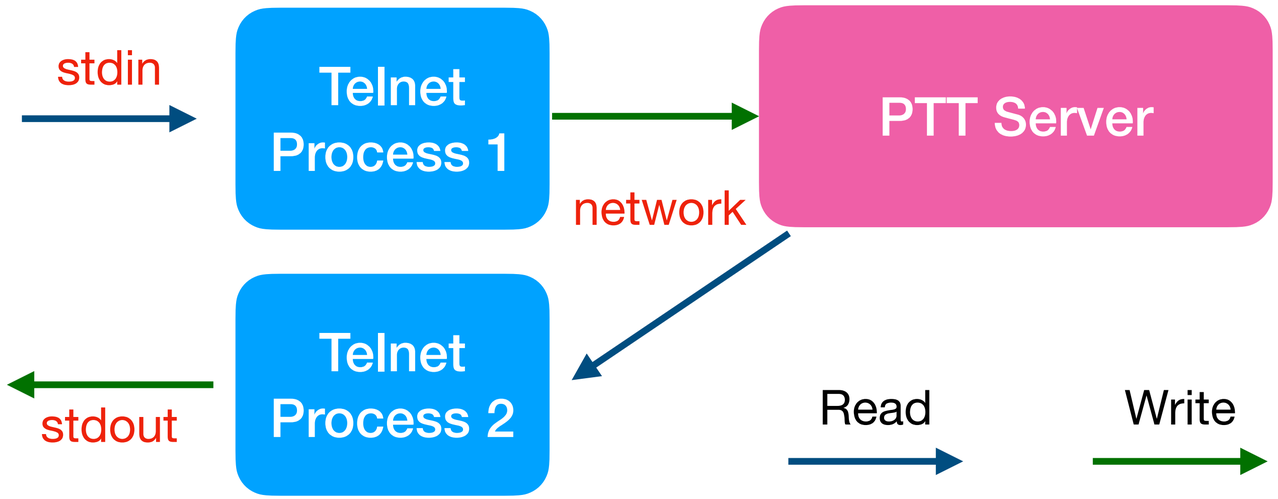
Telnet Process1

```
n = read(STDIN_FILENO, buf, BUFSIZ);
    …
```

Telnet Process2

```
n = read(network_fd, netbuf, BUFSIZ);
    …
```

Blocking I/O, to read from one descriptor and write to another descriptor



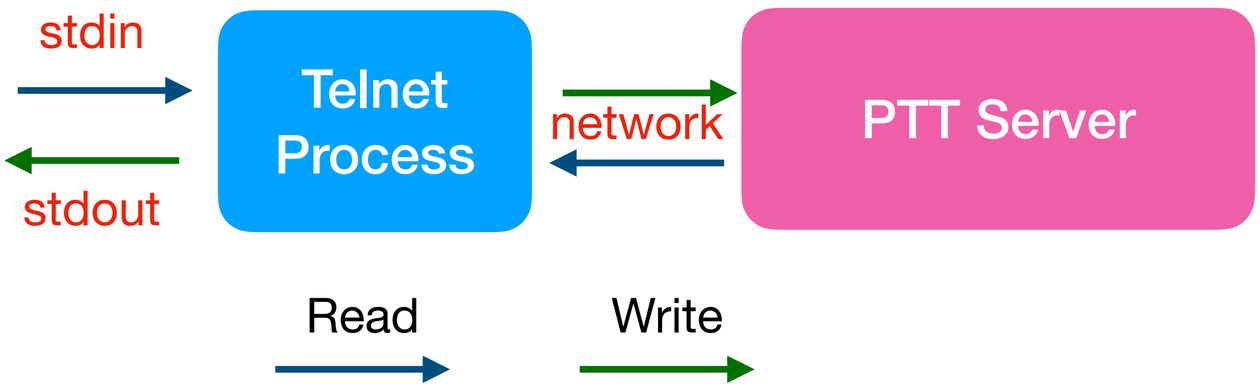

**Idea1. multi-process:** run two processes, let each process do one blocking read

**Problem:** waste of resources to allocating processes, extra overhead needed for process communications & state synchronization

# I/O Multiplexing

Telnet Process

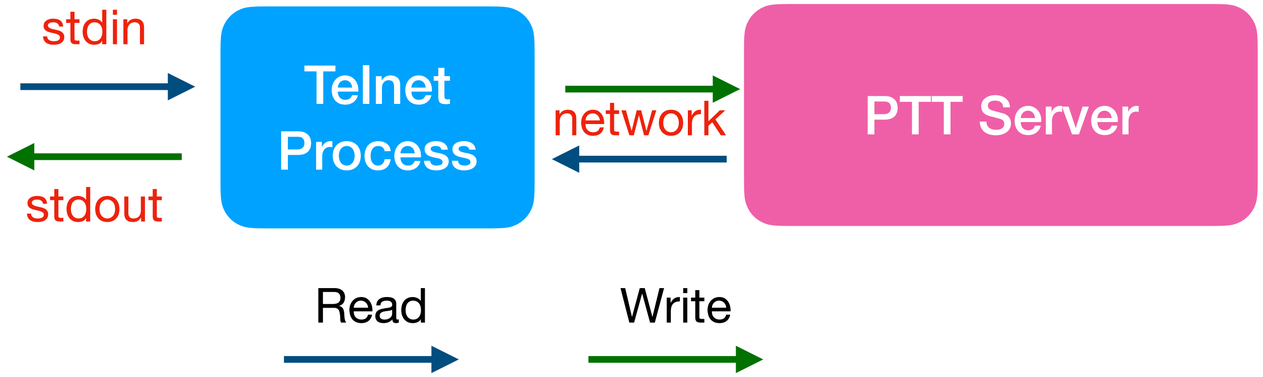
```
While (1) {
n = read(STDIN_FILENO, buf, BUFSIZ);
   /* if read nothing, do something else, then check STDIN */
    …
n = read(network_fd, netbuf, BUFSIZ);
   /* if read nothing, do something else, then check network_fd */
    …
}
```




**Idea2. nonblocking I/O:** set the two I/O descriptors for read to ***non-blocking*** - for a file descriptor, if the data is present, we read it and process it; if no data is present, the read returns immediately - ***polling*** for I/Os

**Problem:** waste of CPU resources if the operating cannot be carried in most of the time

# I/O Multiplexing




**Idea3. I/O Multiplexing:** use the functions ***select*** and ***poll***

***Goal:*** to avoid busy-wait; handle multiple I/O sources at the same time

- I/O multiplexing is generally employed when:
  - An application needs to handle multiple I/O file descriptors at the same time — e.g. file and network I/O (socket) descriptors
  - I/O on any of the descriptors can result in blocking

# I/O Multiplexing: select

```
#include <sys/select.h>
// returns: count of ready descriptors, 0 on timeout before any descriptors is ready, –1 on error
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

- The *select()* function supports I/O multiplexing in all POSIX-compatible platforms — arguments to *select()* tell the kernel:
  - What descriptors we're interested in
  - Which conditions we are interested in for each descriptor: read from a given descriptor, write to a given descriptor, or an exception condition for a given descriptor
  - How long we want to wait: wait forever, wait for a fixed amount of time, not wait at all

# I/O Multiplexing: select

```
#include <sys/select.h>
// returns: count of ready descriptors, 0 on timeout before any descriptors is ready, –1 on error
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

- The *select()* function supports I/O multiplexing in all POSIX-compatible platforms — arguments to *select()* tell the kernel:
  - What descriptors we're interested in: ***nfds*** — set to the highest-numbered file descriptor in any of the three sets, plus 1; the indicated file descriptors in each set are checked, up to this limit
  - Which conditions we are interested in for each descriptor: read from a given descriptor, write to a given descriptor, or an exception condition for a given descriptor: ***readfds, writefds, exceptfds*** — the pointers to *descriptor sets* (bitmap of file descriptors)
  - How long we want to wait: wait forever, wait for a fixed amount of time, not wait at all: ***timeout***

# I/O Multiplexing: select

- *Descriptor sets* (readfds, writefds, exceptfds) are stored in the *fd_set* data type: the bitmap for each possible file descriptor

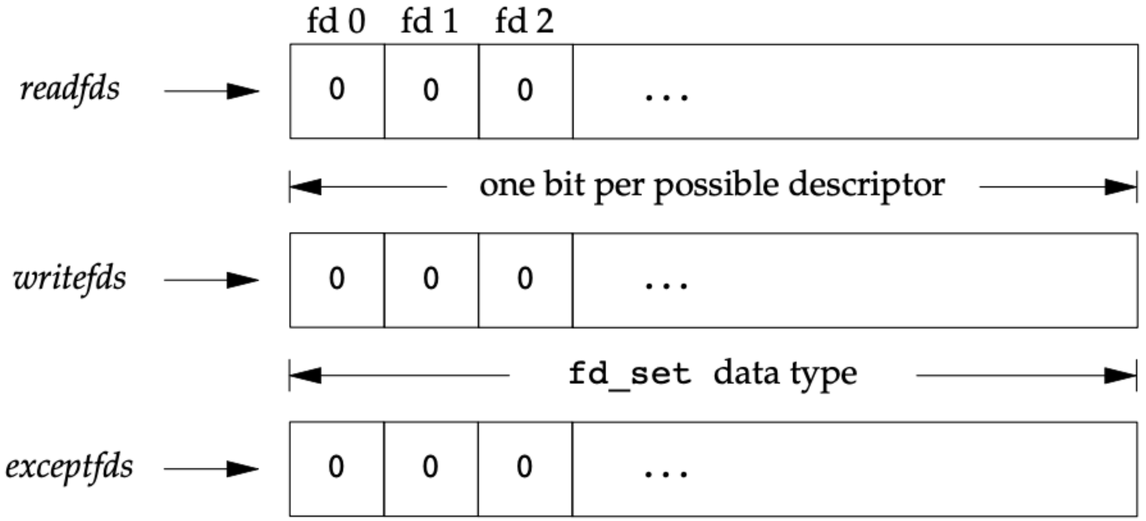



**Figure 14.15** Specifying the read, write, and exception descriptors for `select`

From: Figure 14.15 in APUE 3rd edition

```
#include <sys/select.h>
int FD_ISSET(int fd, fd_set *fdset);
void FD_CLR(int fd, fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_ZERO(fd_set *fdset);
```

Returns: nonzero if *fd* is in set, 0 otherwise

Helper macros/functions for the fd_set

# I/O Multiplexing: select

```
#include <sys/select.h>
// returns: count of ready descriptors, 0 on timeout before any descriptors is ready, –1 on error
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

- The kernel tells us (1) The total number of descriptors (sum of all three sets) that are ready (return value of *select()*), (2) What descriptors are ready for each of the conditions (updates to the read, write, and exception set)
  - A given descriptor in the read-and-write set is *ready*: the read/write from/to the descriptor *will not block*
  - A given descriptor in the exception set is *ready*: an exception condition is pending on that descriptor

# I/O Multiplexing: select

```
fd_set  readset, writeset;

FD_ZERO(&readset);
FD_ZERO(&writeset);
FD_SET(0, &readset);
FD_SET(3, &readset);
FD_SET(1, &writeset);
FD_SET(2, &writeset);
select(4, &readset, &writeset, NULL, NULL);
```

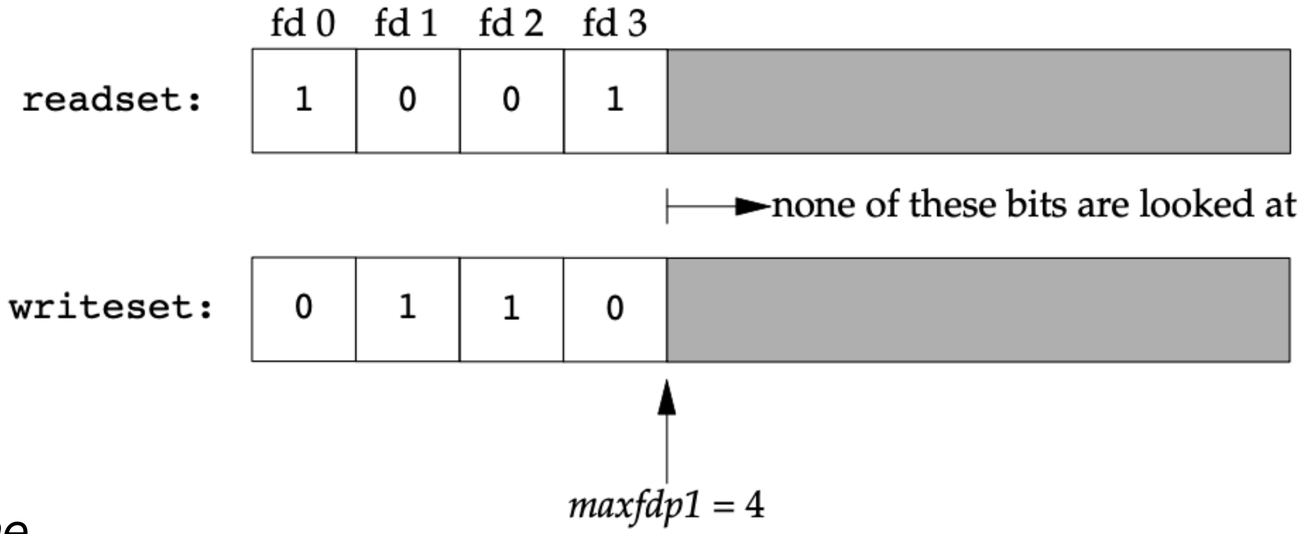



**Figure 14.16** Example descriptor sets for `select`

From: Figure 14.16 in APUE 3rd edition

The positive return value of *select* is the *sum of the descriptors in the three sets that are ready* — if the same descriptor is ready to be read *and* written, it will be counted twice in the return value!

# I/O Multiplexing: select

```
int main()
{
        int     i;
        struct timeval  timeout;
        struct fd_set   master_set, working_set;
        char    buf[1024];

        FD_ZERO( &master_set );
        FD_SET( 0, &master_set );
        timeout.tv_sec = 5;
        timeout.tv_usec = 0;
        i = 0;

        while ( 1 )
        {
                memcpy( &working_set, &master_set, sizeof( master_set ) );
                select( 1, &working_set, NULL, NULL, &timeout );
                if ( FD_ISSET( 0, &working_set ) )
                {
                        fgets( buf, sizeof( buf ), stdin );
                        fputs( buf, stdout );
                }
                printf( "iteration: %d\n", i ++ );
        }
        return 0;
}
```

Note: the kernel updates the working_set here - you need to backup and restore each time!

# I/O Multiplexing: select

```
#include <sys/select.h>
// returns: count of ready descriptors, 0 on timeout, –1 on error
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

- We can set nfds to *FD_SETSIZE*, a constant in <sys/select.h> that specifies the maximum number of descriptors (set to 1024 in Linux)

- The pointers to the descriptor set can be NULL pointers if we are not interested in that condition; if all three pointers are NULL, select works as a *higher-precision timer* (in microseconds) than is provided by sleep (in seconds)

```
// the select call in the following sleeps for the time specified timeval
int totalfds = select(0, NULL, NULL, NULL, &timeval);
```

# I/O Multiplexing: select

```
#include <sys/select.h>
// returns: count of ready descriptors, 0 on timeout, –1 on error
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

- We can set nfd … cifies the maximum number of des…
- The pointers to … nterested in that condition
  - If all three p… er (in microseconds) than is prov…

**The timeout**
The *timeout* argument for **select**() is a structure of the following type:

```
struct timeval {
    time_t      tv_sec;         /* seconds */
    suseconds_t tv_usec;        /* microseconds */
};
```

The corresponding argument for **pselect**() is a **timespec**(3) structure.

On Linux, **select**() modifies *timeout* to reflect the amount of time not slept; most other implementations do not do this. (POSIX.1 permits either behavior.) This causes problems both when Linux code which reads *timeout* is ported to other operating systems, and when code is ported to Linux that reuses a *struct timeval* for multiple **select**()s in a loop without reinitializing it. Consider *timeout* to be undefined after **select**() returns.

https://man7.org/linux/man-pages/man2/select.2.html

# I/O Multiplexing: poll

```
#include <poll.h>
// returns: count of ready descriptors, 0 on timeout, –1 on error
int poll(struct pollfd fdarray[], nfds_t nfds, int timeout);
```

```
struct pollfd {
    int fd; // the file descriptor to check, or < 0 to ignore
    short events; // events of interest on fd
    short revents; // events that occurred on fd
};
```

- The *poll* function is similar to *select* but has a different programmer interface:
  - *nfds*: specify the number of items in the *fdarray*
  - Build an array of pollfd structures to specify the descriptors (***fds***) and the conditions that of interest
    - *The caller sets* the ***events*** of interests (a bit mask, Figure 14.17 in APUE 3rd edition has more details)
    - ***revents*** is *set by the kernel* to specify which events have occurred for each descriptor (a bit mask)
  - timeout: how long (in milliseconds) to wait before un-suspending the process:
    - -1: wait forever; 0: do not wait; >0: wait (milliseconds)

# I/O Multiplexing: poll

```
#include <poll.h>
// returns: count of ready descriptors, 0 on timeout, –1 on error
int poll(struct pollfd fdarray[], nfds_t nfds, int timeout);
```

```
struct pollfd {
    int fd; // the file descriptor to check, or < 0 to ignore
    short events; // events of interest on fd
    short revents; // events that occurred on fd
};
```

| Name | Input to events? | Result from revents? | Description |
|---|---|---|---|
| POLLIN | • | • | Data other than high priority data can be read without blocking (equivalent to POLLRDNORM \| POLLRDBAND). |
| POLLRDNORM | • | • | Normal data can be read without blocking. |
| POLLRDBAND | • | • | Priority data can be read without blocking. |
| POLLPRI | • | • | High-priority data can be read without blocking. |
| POLLOUT | • | • | Normal data can be written without blocking. |
| POLLWRNORM | • | • | Same as POLLOUT. |
| POLLWRBAND | • | • | Priority data can be written without blocking. |
| POLLERR | | • | An error has occurred. |
| POLLHUP | | • | A hangup has occurred. |
| POLLNVAL | | • | The descriptor does not reference an open file. |

**Figure 14.17** The events and revents flags for poll

From: Figure 14.17 in APUE 3rd edition

# I/O Multiplexing: select v.s. poll

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
fd_set *exceptfds, struct timeval *timeout);
```

```
int poll(struct pollfd fdarray[], nfds_t nfds, int timeout);
```

- Alternatives:
  - *pselect*: (nanoseconds timeout, signal mask)
  - *epoll*: good for speed and scalability

| | select | poll |
|---|---|---|
| kernel updates to input arguments | update the descriptor set | update revents (not events) |
| fd traversal | returns the # of ready descriptors, you have to enumerate the sets | could only put fds of interests in the fdarray |
| supported conditions | read, write, error | more than 3 condition types |

# Comparison of I/O Models

| | Blocking I/O | Non-blocking I/O (polling) | I/O Multiplexing |
|---|---|---|---|
| How to use? | FD without O_NONBLOCK flag | FD with O_NONBLOCK flag | select()/poll() |
| Handle Multiple FDs | No | Yes | Yes |
| CPU Usage | Low | High | Low |
| Responsiveness | Slow | Fast | Fast |

# Backup

# FreeBSD Implementation for record locking

```
fd1 = open(pathname, ...);
/* parent write locks byte 0 */
write_lock(fd1, 0, SEEK_SET, 1);
/* parent */
if ((pid = fork()) > 0) {
    fd2 = dup(fd1);
    fd3 = open(pathname, ...);
} else if (pid == 0) {
/* child read locks byte 1 */
    read_lock(fd1, 1, SEEK_SET, 1);
}
pause();
```

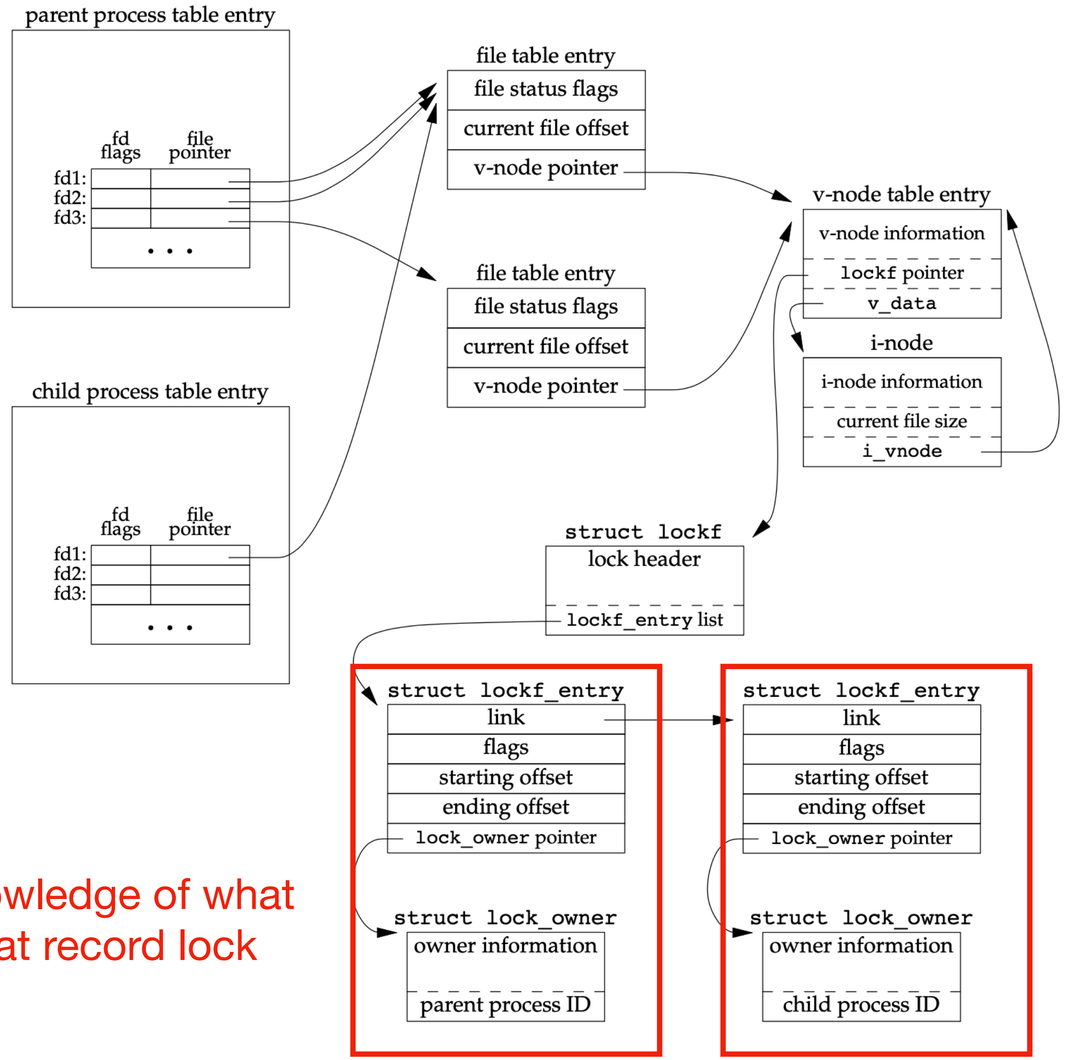



**Figure 14.8** The FreeBSD data structures for record locking

The kernel has no knowledge of what descriptors own what record lock

# Comparison of I/O Models