CS 33

Files Part 3

Setting File Permissions

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *path, mode_t mode)
```

- sets the file permissions of the given file to those specified in *mode*
- only the owner of a file and the superuser may change its permissions
- nine combinable possibilities for *mode* (*read/write/execute* for *user*, *group*, and *others*)
  - `S_IRUSR (0400), S_IWUSR (0200), S_IXUSR (0100)`
  - `S_IRGRP (040), S_IWGRP (020), S_IXGRP (010)`
  - `S_IROTH (04), S_IWOTH (02), S_IXOTH (01)`

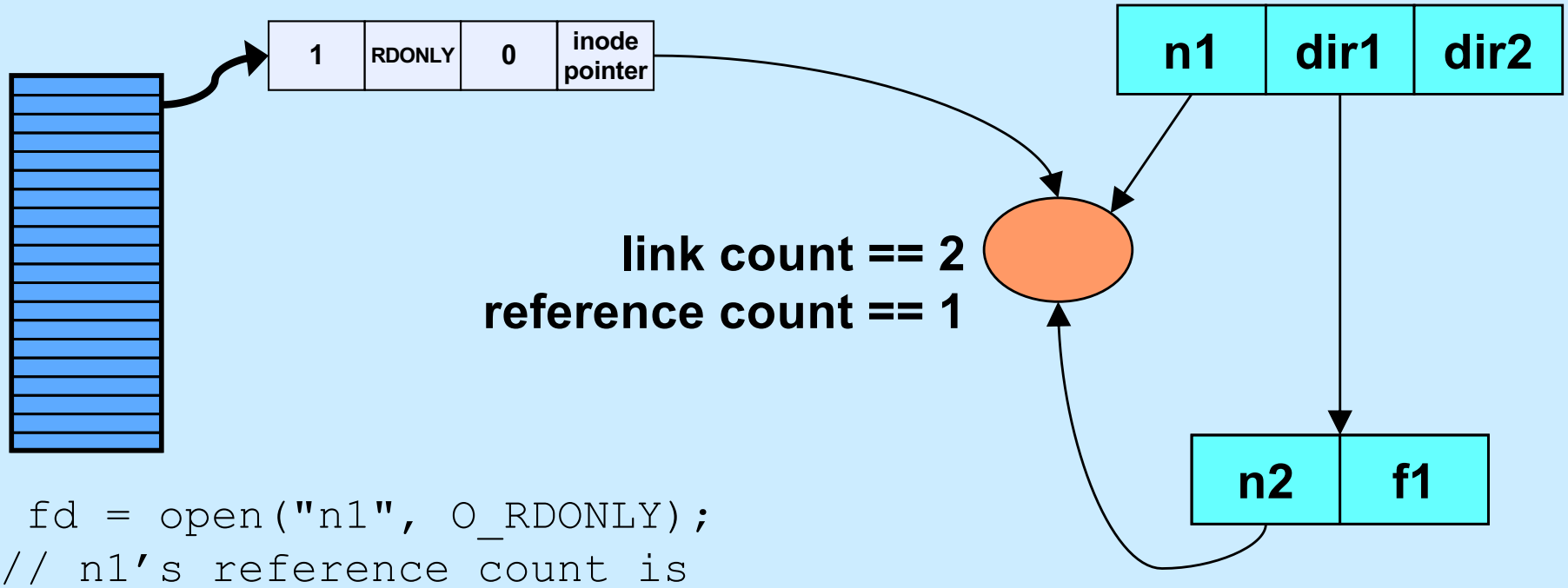
Umask

- Standard programs create files with "maximum needed permissions" as mode
  - compilers: 0777
  - editors: 0666
- Per-process parameter, *umask*, used to turn off undesired permission bits
  - e.g., turn off all permissions for others, write permission for group: set umask to 027
    - » compilers: permissions = 0777 & ~(027) = 0750
    - » editors: permissions = 0666 & ~(027) = 0640
  - set with *umask* system call or (usually) shell command

Creating a File

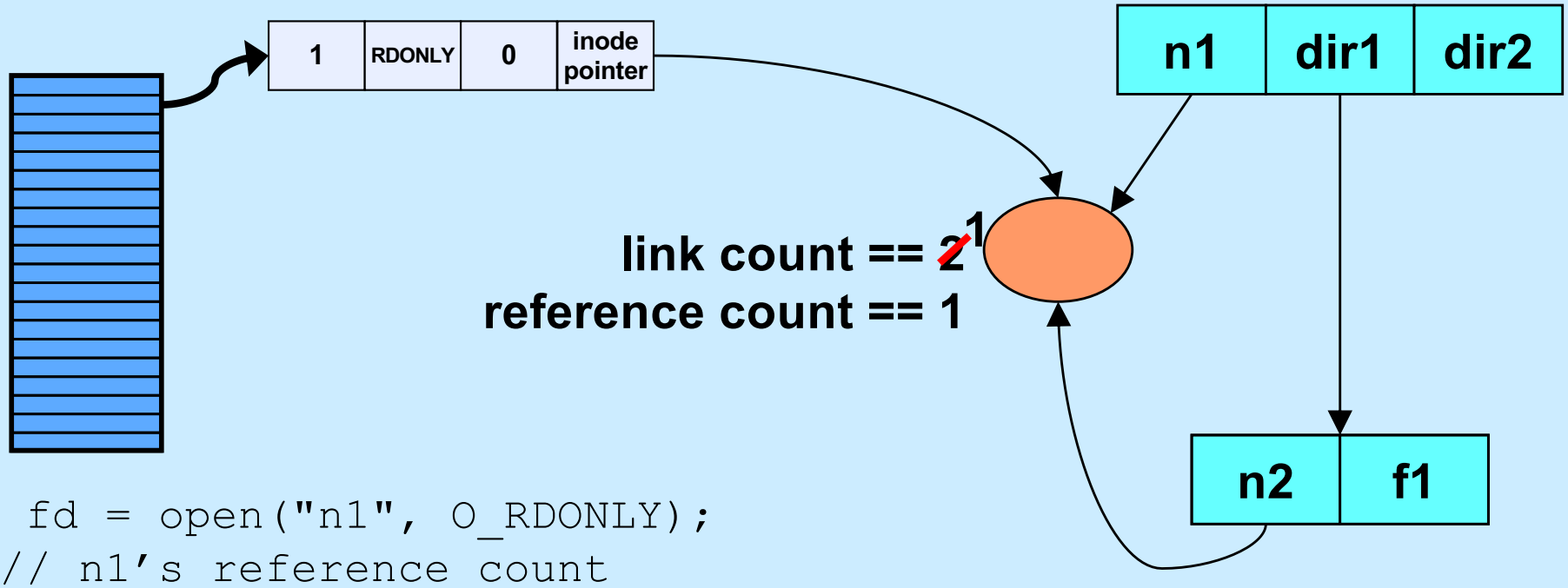
- **Use either *open* or *creat***
  - `open(const char *pathname, int flags, mode_t mode)`
    - » **flags must include O_CREAT**
  - `creat(const char *pathname, mode_t mode)`
    - » **open is preferred**
- **The *mode* parameter helps specify the permissions of the newly created file**
  - **permissions = mode & ~umask**

Link and Reference Counts




```
int fd = open("n1", O_RDONLY);
   // n1's reference count is
   // incremented by 1
```

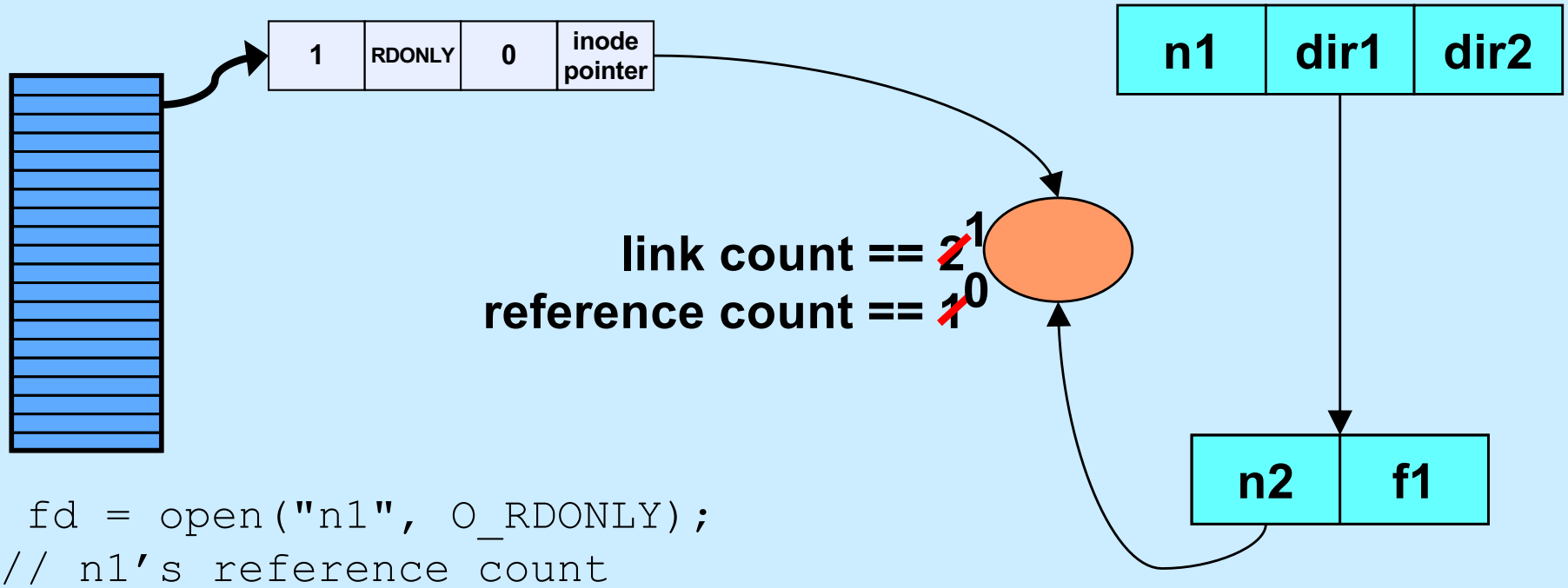
Link and Reference Counts




```
int fd = open("n1", O_RDONLY);
    // n1's reference count
    // incremented by 1

unlink("n1");
    // link count decremented by 1
    // same effect in shell via "rm n1"
```

Link and Reference Counts

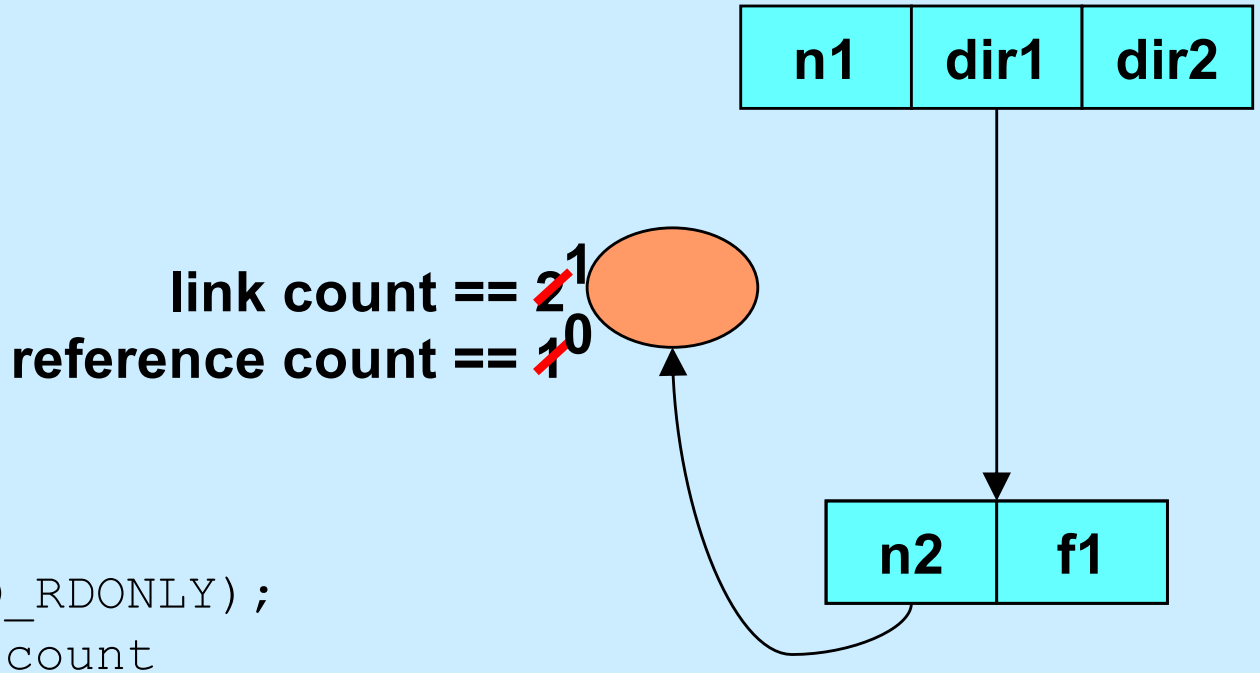



```
int fd = open("n1", O_RDONLY);
   // n1's reference count
   // incremented by 1

unlink("n1");
   // link count decremented by 1

close(fd);
   // reference count decremented by 1
```

Link and Reference Counts

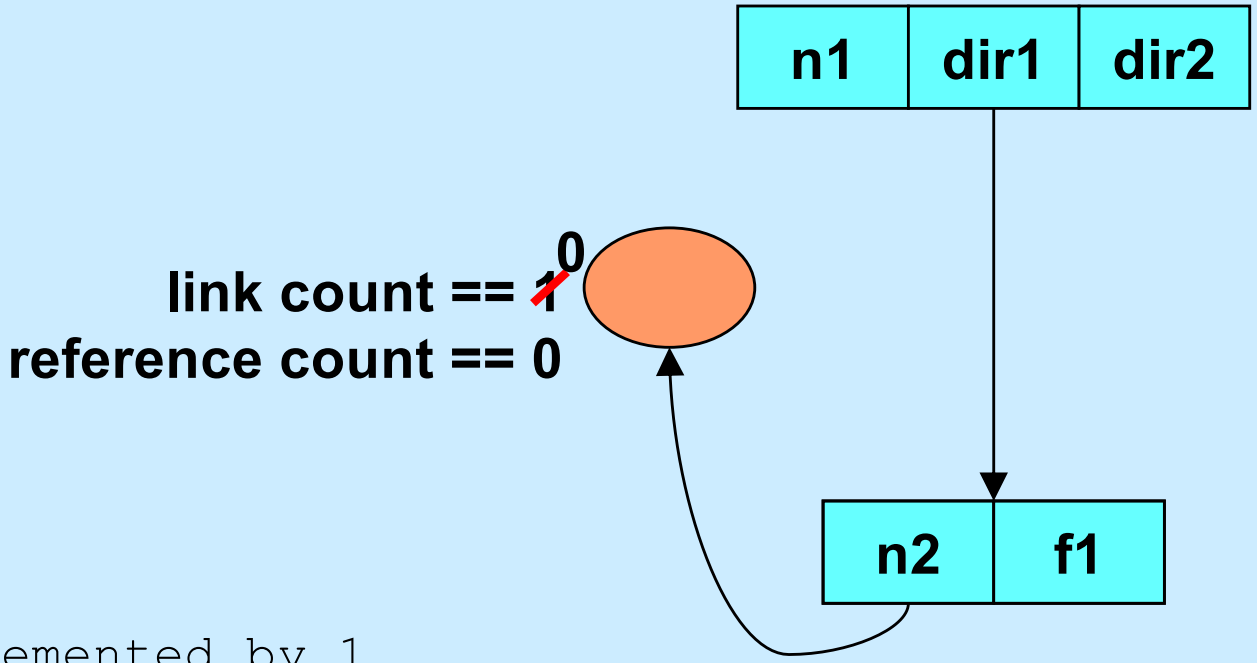



```
int fd = open("n1", O_RDONLY);
   // n1's reference count
   // incremented by 1

unlink("n1");
   // link count decremented by 1

close(fd);
   // reference count decremented by 1
```

Link and Reference Counts




```
unlink("dir1/n2");
   // link count decremented by 1
```

Quiz 1

```
int main() {
  int fd = open("file", O_RDWR|O_CREAT, 0666);
  unlink("file");
  PutStuffInFile(fd);
  GetStuffFromFile(fd);
  return 0;
}
```

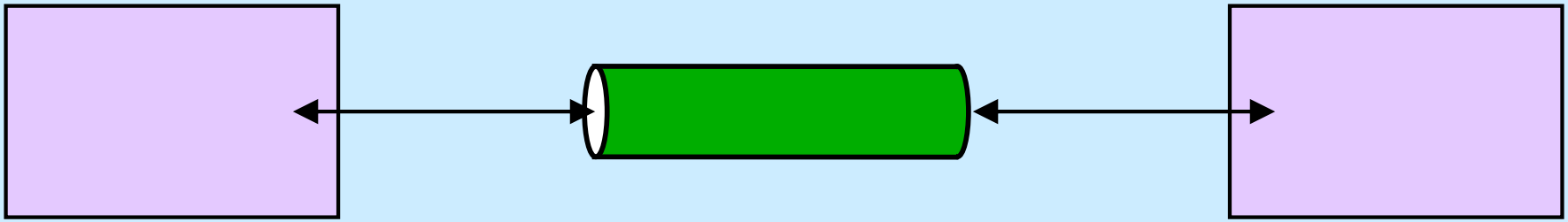
Assume that *PutStuffInFile* writes to the given file, and *GetStuffFromFile* reads from the file.

a) This program is doomed to failure, since the file is deleted before it's used

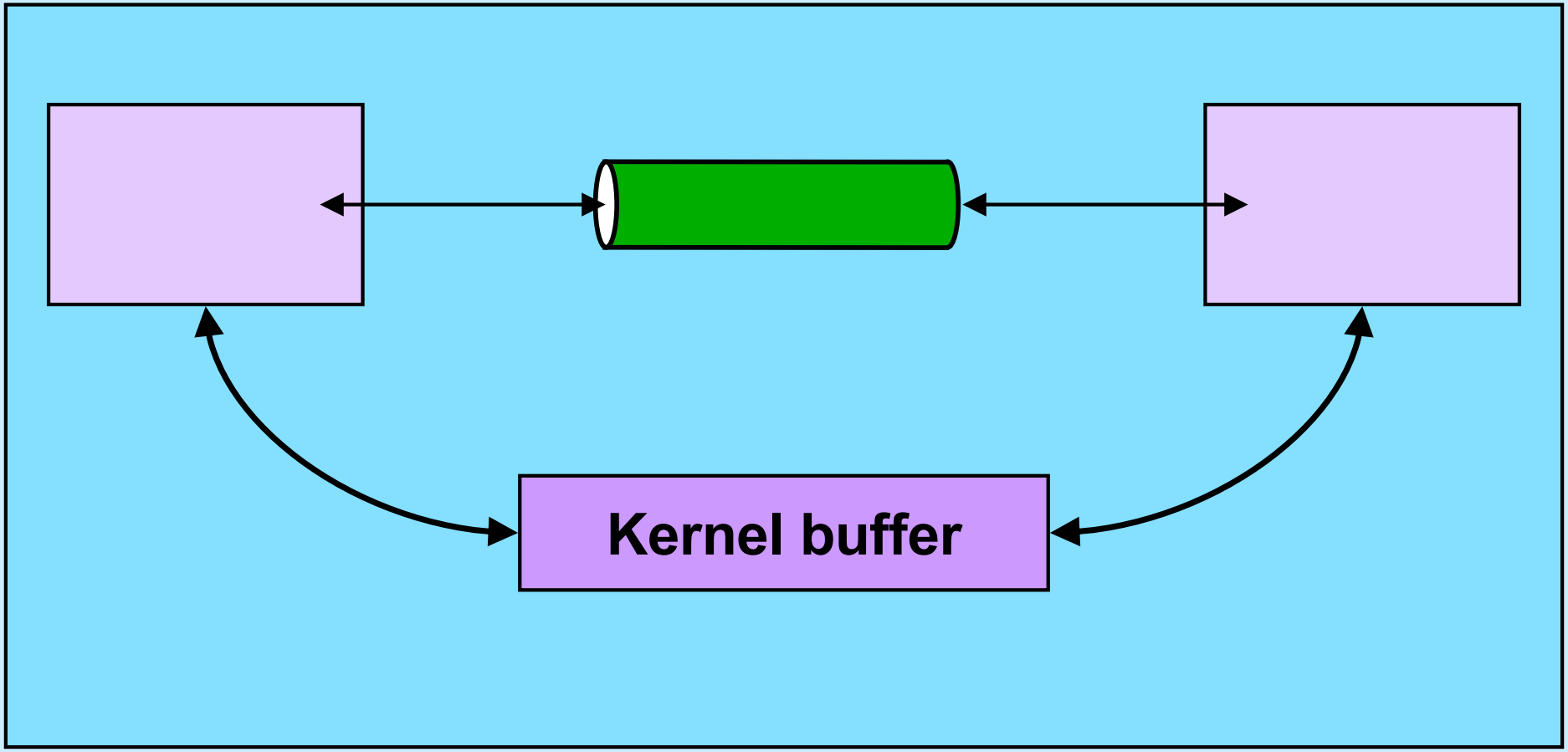
b) Because the file is used after the unlink call, it won't be deleted

c) The file will be deleted when the program terminates

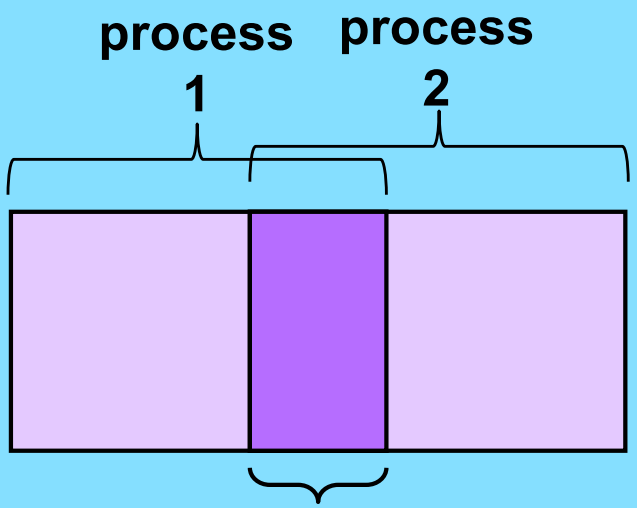
Interprocess Communication (IPC): Pipes



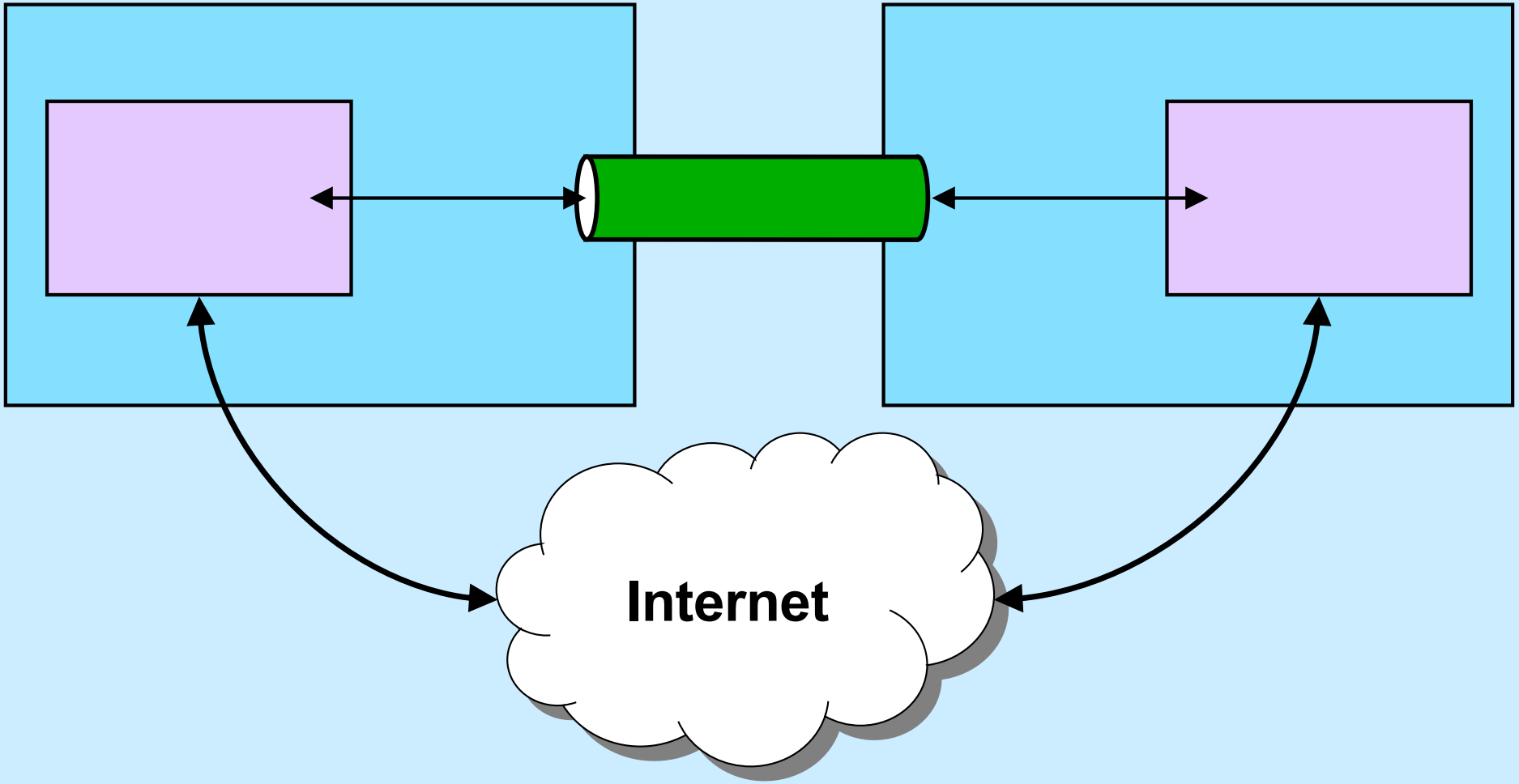
Interprocess Communication: Same Machine I



Interprocess Communication: Same Machine II

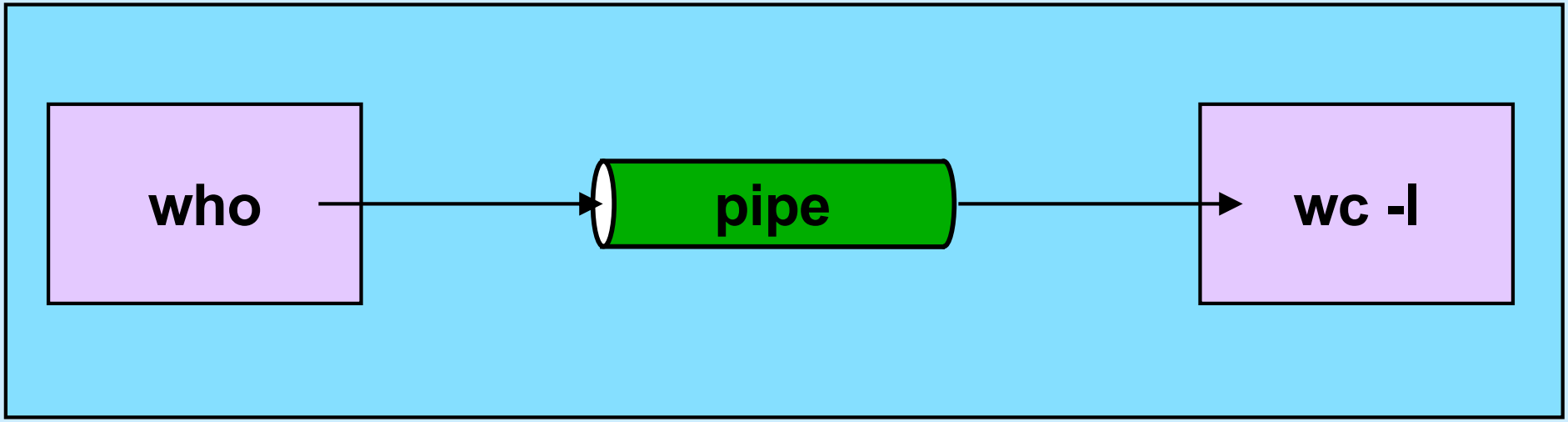


Interprocess Communication: Different Machines



Pipes

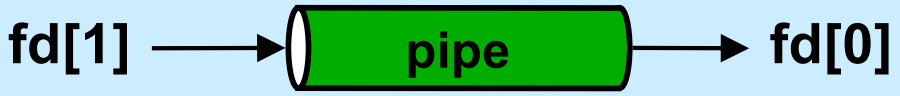
```
$cslab2e who | wc -l
```



Using Pipes in C

```
$cslab2e who | wc -l
```

```
int fd[2];
pipe(fd);
if (fork() == 0) {
  close(fd[0]);
  close(1);
  dup(fd[1]); close(fd[1]);
  execl("/usr/bin/who", "who", 0); // who sends output to pipe
}
if (fork() == 0) {
  close(fd[1]);
  close(0);
  dup(fd[0]); close(fd[0]);
  execl("/usr/bin/wc", "wc", "-l", 0); // wc's input is from pipe
}
close(fd[1]); close(fd[0]);
// …
```



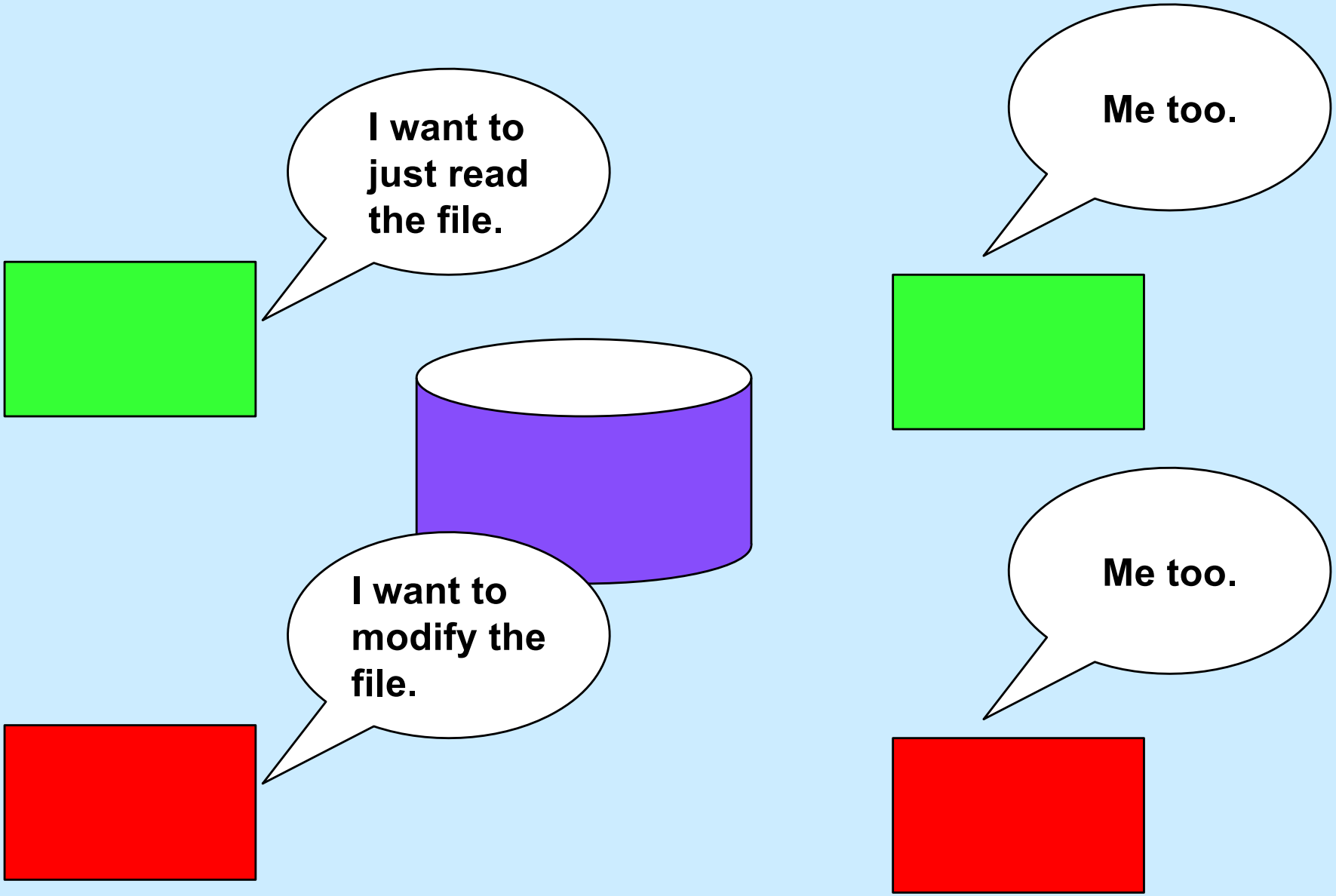
Sharing Files

- You're doing a project with a partner
- You code it as one 15,000-line file
  - the first 7,500 lines are yours
  - the second 7,500 lines are your partner's
- You edit the file, changing 6,000 lines
  - it's now 5am
- Your partner completes her changes at 5:01am
- At 5:02am you look at the file
  - your partner's changes are there
  - yours are not

Lessons

- Never work with a partner
- Use more than one file
- Read up on git
- Use an editor and file system that support file locking

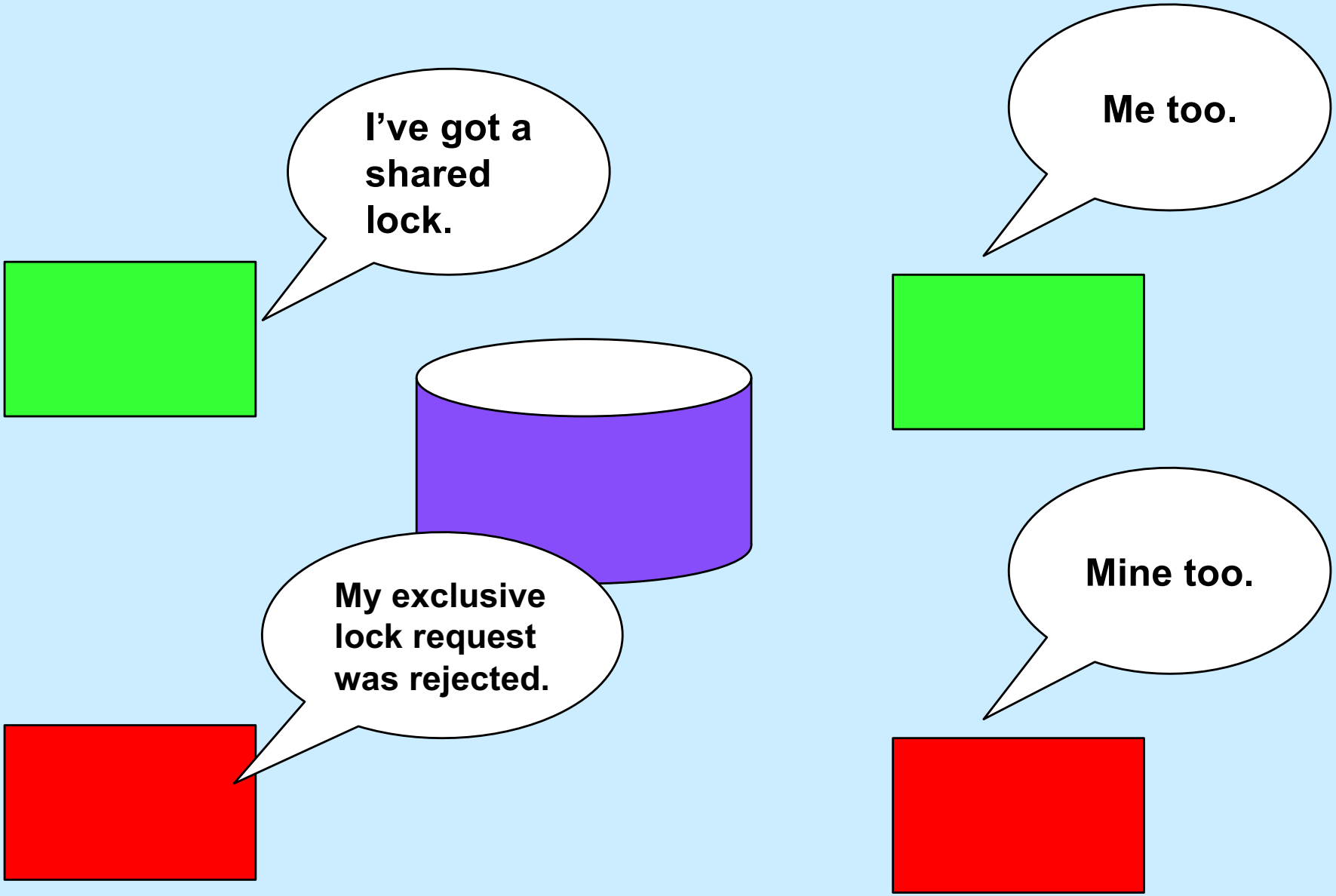
What We Want ...



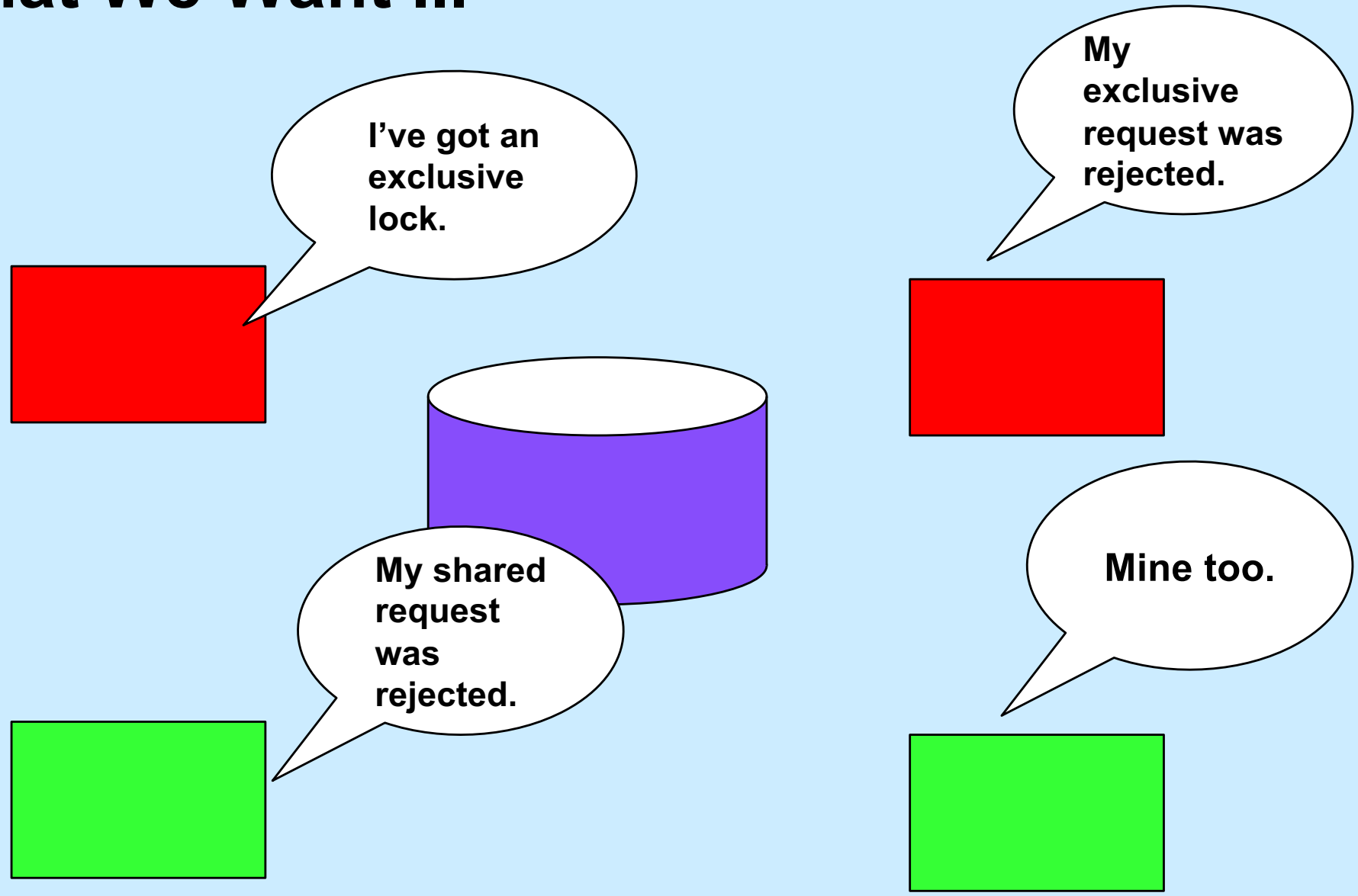
Types of Locks

- Shared (readers) locks
  - any number may have them at same time
  - may not be held when an exclusive lock is held by others
- Exclusive (writers) locks
  - only one at a time
  - may not be held when any other lock is held by others

What We Want ...



What We Want ...



Locking Files

- **Early Unix didn't support file locking**
- **How did people survive?**
  - `open("file.lck", O_RDWR|O_CREAT|O_EXCL, 0666);`
    - » **operation fails if *file.lck* exists, succeeds (and creates file.lck) otherwise**
    - » **requires cooperative programs**

Locking Files (continued)

- How it's done in "modern" Unix
  - "advisory locks" may be placed on files
    - » may request shared (readers) or exclusive (writers) lock
      - *fcntl* system call
    - » either succeeds or fails
    - » *open*, *read*, *write* always work, regardless of locks
    - » a lock applies to a specified range of bytes, not necessarily to the whole file
    - » requires cooperative programs
  - "mandatory locks" supported as a per-file option
    - » set along with permission bits
    - » if set, file can't be used unless process possesses appropriate locks

Locking Files (still continued)

- **How to:**

```
struct flock fl;
fl.l_type = F_RDLCK;        // read lock
// fl.l_type = F_WRLCK;     // write lock
// fl.l_type = F_UNLCK;     // unlock
fl.l_whence = SEEK_SET;     // starting where
fl.l_start = 0;             // offset
fl.l_len = 0;               // how much? (0 = whole file)
fd = open("file", O_RDWR);
if (fcntl(fd, F_SETLK, &fl) == -1)
  if ((errno == EACCES) || (errno == EAGAIN))
    // didn't get lock
  else
    // something else is wrong
else
  // got the lock!
```

Quiz 2

- **Your program currently has a shared lock on a portion of a file. It would like to "upgrade" the lock to be an exclusive lock. Would there be any problems with adding an option to *fcntl* that would allow the holder of a shared lock to wait until it's possible to upgrade to an exclusive lock, then do the upgrade?**

  a) **at least one major problem**

  b) **either no problems whatsoever or some easy-to-deal-with problems**

Shell 1: Artisanal Coding

```
while ((line = get_a_line()) != 0) {
    tokens = parse_line(line);
    for (int i=0; i < ntokens; i++) {
        if (strcmp(tokens[i], ">") == 0) {
            // handle output redirection
        } else if (strcmp(tokens[i], "<") == 0) {
            // handle input redirection
        } else if (strcmp(tokens[i], "&") == 0) {
            // handle "no wait"
        } ... else {
            // handle other cases
        }
    }
    if (fork() == 0) {
        // ...
        execv(...);
    }
    // ...
}
```

Shell 1: Non-Artisanal Coding (1)

```
while ((line = get_a_line()) != 0) {
    tokens = parse_line(line);
    for (int i=0; i < ntokens; i++) {
        // handle "normal" case
    }
    if (fork() == 0) {
        // ...
        execv(...);
    }
    // ...
}
```

Shell 1: Non-Artisanal Coding (2)

```
next_line: while ((line = get_a_line()) != 0) {
    tokens = parse_line(line);
    for (int i=0; i < ntokens; i++) {
        if (redirection_symbol(token[i])) {
            // ...
            if (fork() == 0) {
                // ...
                execv(...);
            }
            // ...
            goto next_line;
        }
        // handle "normal" case
    }
    if (fork() == 0) {
        // ...
        execv(...);
    }
    // ...
}
```

whoops!

(whoops!)

Shell 1: Non-Artisanal Coding (3)

```
next_line: while ((line = get_a_line()) != 0) {
    tokens = parse_line(line);
    for (int i=0; i < ntokens; i++) {
        if (redirection_symbol(token[i])) {
            // ...
            if (fork() == 0) {
                // ...
                execv(...);
            }
            // ... deal with &
            goto next_line;
        }
        // handle "normal" case
    }
    if (fork() == 0) {
        // ...
        execv(...);
    }
    // ... also deal with & here!
}
```

Shell 1: Non-Artisanal Coding (Worse)

```
next_line: while ((line = get_a_line()) != 0) {
tokens = parse_line(line);
for (int i=0; i < ntokens; i++) {
if (redirection_symbol(token[i])) {
// ...
if (fork() == 0) {
// ...
execv(...);
}
// ... deal with &
goto next_line;
}
// handle "normal" case
}
if (fork() == 0) {
// ...
execv(...);
}
// ... also deal with & here!
}
```

Artisanal Programming

- Factor your code!
  - A; FE | B; FE | C; FE = (A | B | C); FE
- Format as you write!
  - don't run the formatter only just before handing it in
  - your code should always be well formatted

- If you have a tough time understanding your code, you'll have a tougher time debugging it and TAs will have an even tougher time helping you

It's Your Code

- Be proud of it!
  - it not only works; it shows skillful artisanship
- It's not enough to merely work
  - others have to understand it
    - » (not to mention you ...)
  - you (and others) have to maintain it
    - » shell 2 is coming soon!