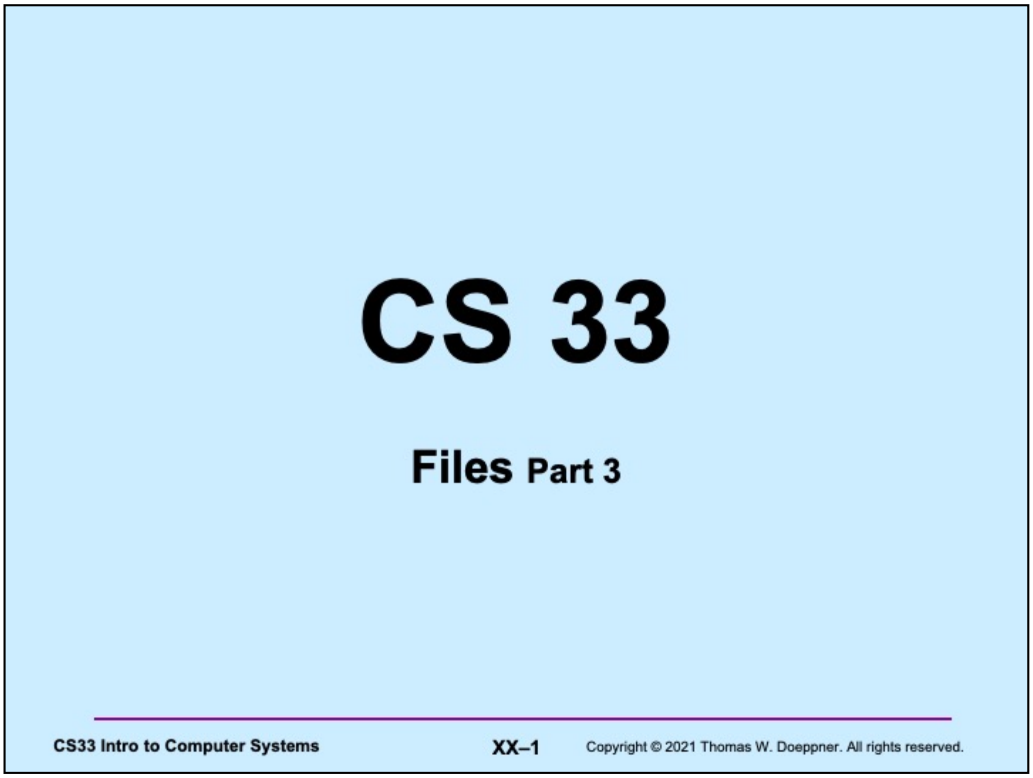
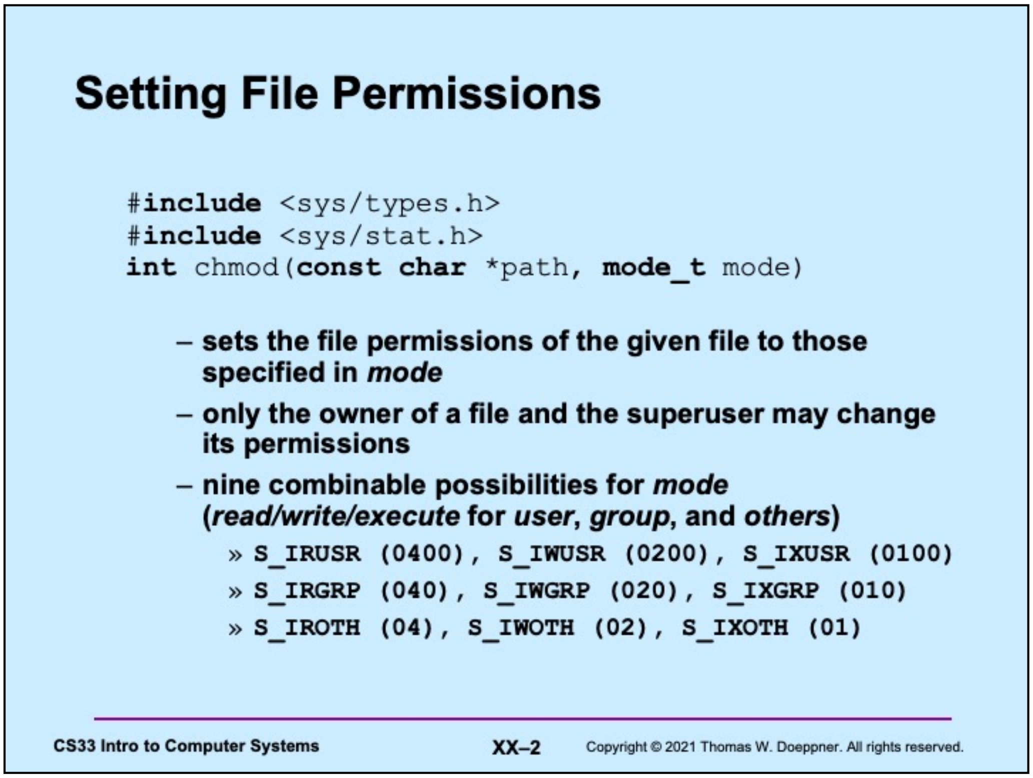


CS 33

Files Part 3

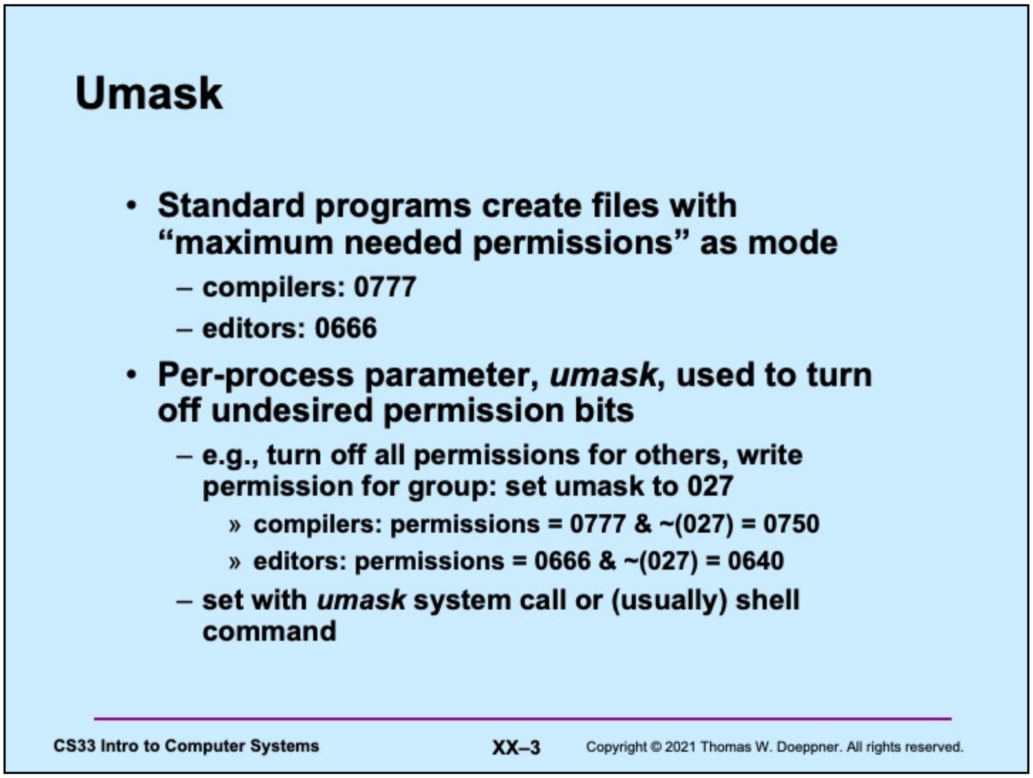


Setting File Permissions

```
#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)`

The **chmod** system call (and the similar **chmod** shell command) is used to change the permissions of a file. Note that the symbolic names for the permissions are rather cumbersome; what is often done is to use their numerical equivalents instead. Thus the combination of read/write/execute permission for the user (0700), read/execute permission for the group (050), and execute-only permission for others (01) can be specified simply as 0751.



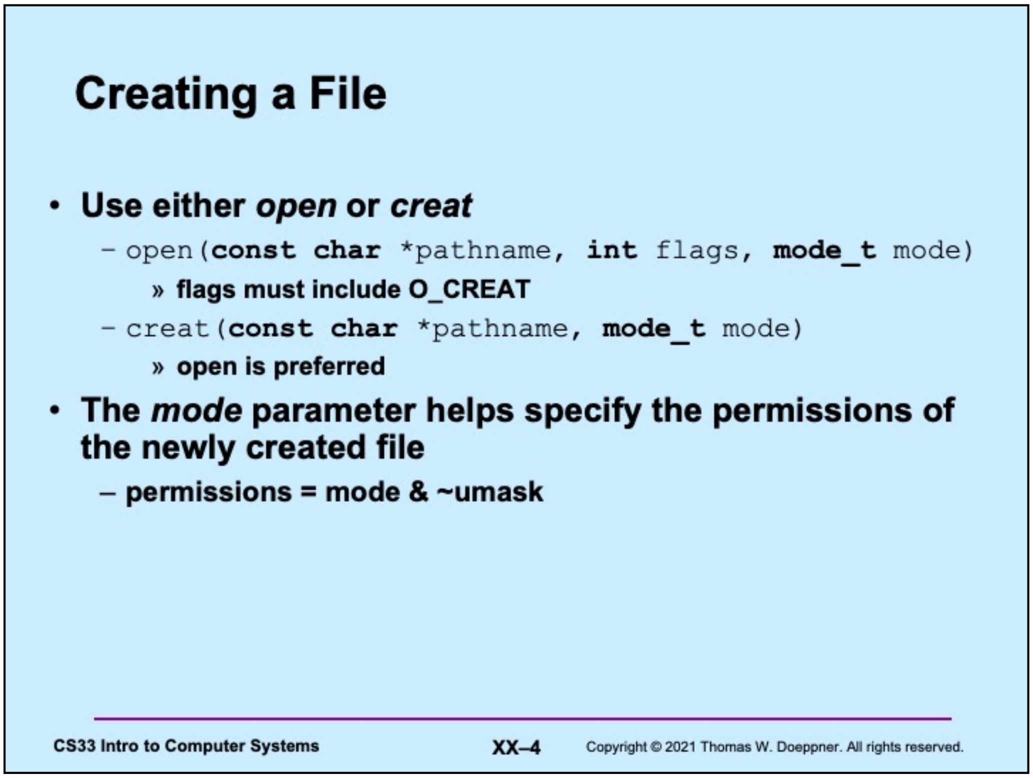
Umask

- **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

The **umask** (often called the "creation mask") allows programs to have wired into them a standard set of maximum needed permissions as their file-creation modes. Users then have, as part of their environment (via a per-process parameter that is inherited by child processes from their parents), a limit on the permissions given to each of the classes of security principals. This limit (the **umask**) looks like the 9-bit permissions vector associated with each file, but each one-bit indicates that the corresponding permission is not to be granted. Thus, if **umask** is set to 022, then, whenever a file is created, regardless of the settings of the mode bits in the *open* or *creat* call, write permission for *group* and *others* is not to be included with the file's access permissions.

You can determine the current setting of **umask** by executing the **umask** shell command without any arguments.

(Recall that numbers written with a leading 0 are in octal (base-8) notation.)

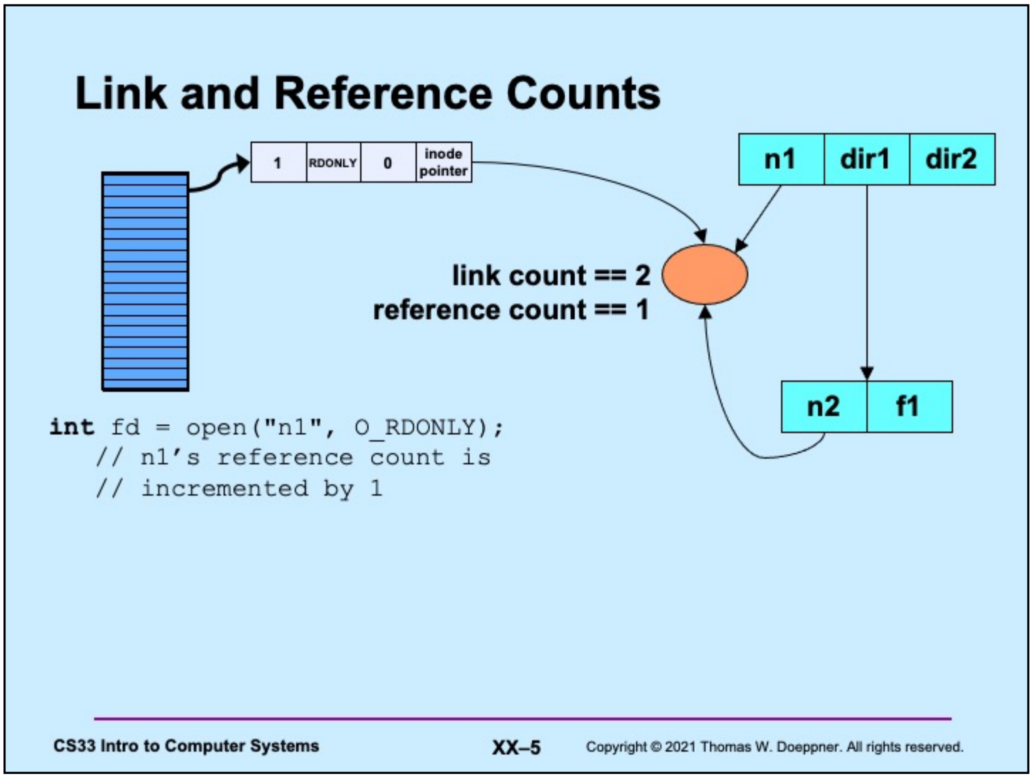


Creating a File

- **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**

Originally in Unix one created a file only by using the **creat** system call. A separate O_CREAT flag was later given to **open** so that it, too, can be used to create files. The **creat** system call fails if the file already exists. For **open**, what happens if the file already exists depends upon the use of the flags O_EXCL and O_TRUNC. If O_EXCL is included with the flags (e.g., **open("newfile", O_CREAT|O_EXCL, 0777)),** then, as with **creat**, the call fails if the file exists. Otherwise, the call succeeds and the (existing) file is opened. If O_TRUNC is included in the flags, then, if the file exists, its previous contents are eliminated and the file (whose size is now zero) is opened.

When a file is created by either **open** or **creat**, the file's initial access permissions are the bitwise AND of the mode parameter and the complement of the process's **umask** (explained in the previous slide).

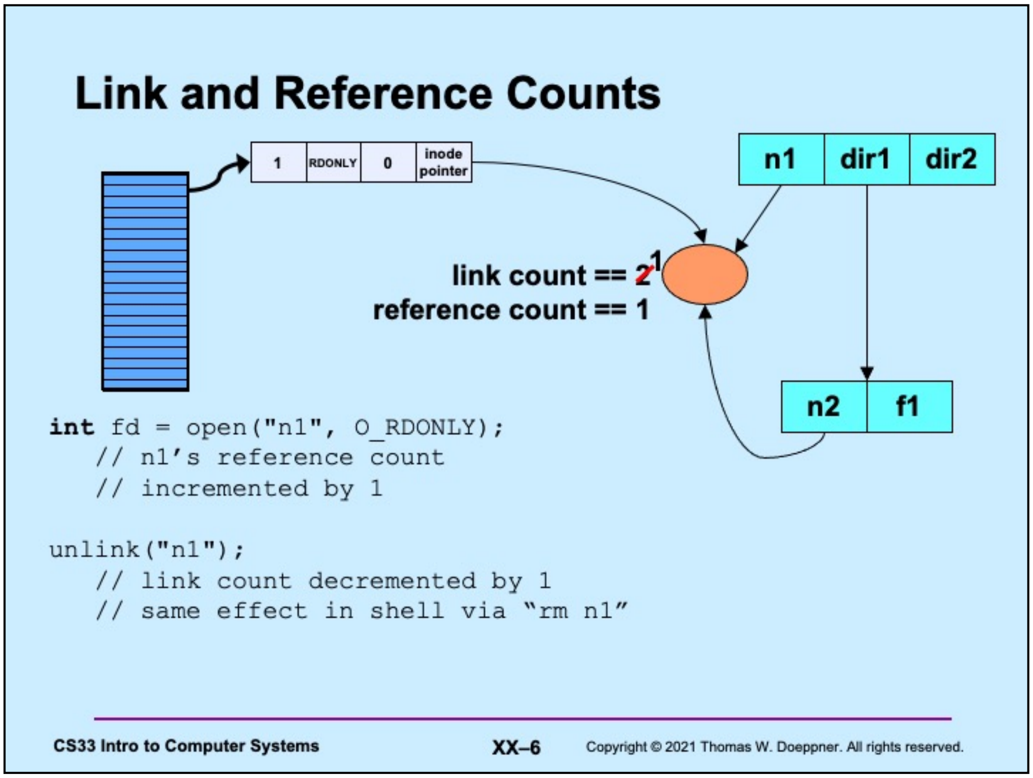


Link and Reference Counts

link count == 2
reference count == 1

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

A file’s link count is the number of directory entries that refer to it. There’s a separate reference count that’s the number of file context structures that refer to it (via the inode pointer – see slide XVII-5).



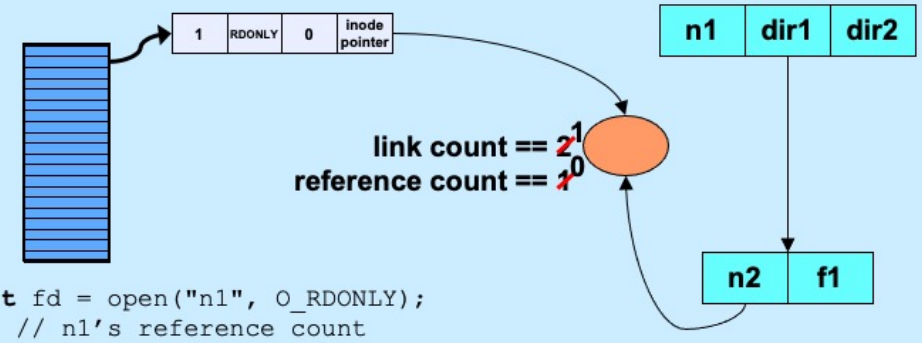
Link and Reference Counts

```
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"
```

Note that the shell's rm command is implemented using unlink; it simply removes the directory entry, reducing the file's link count by 1.

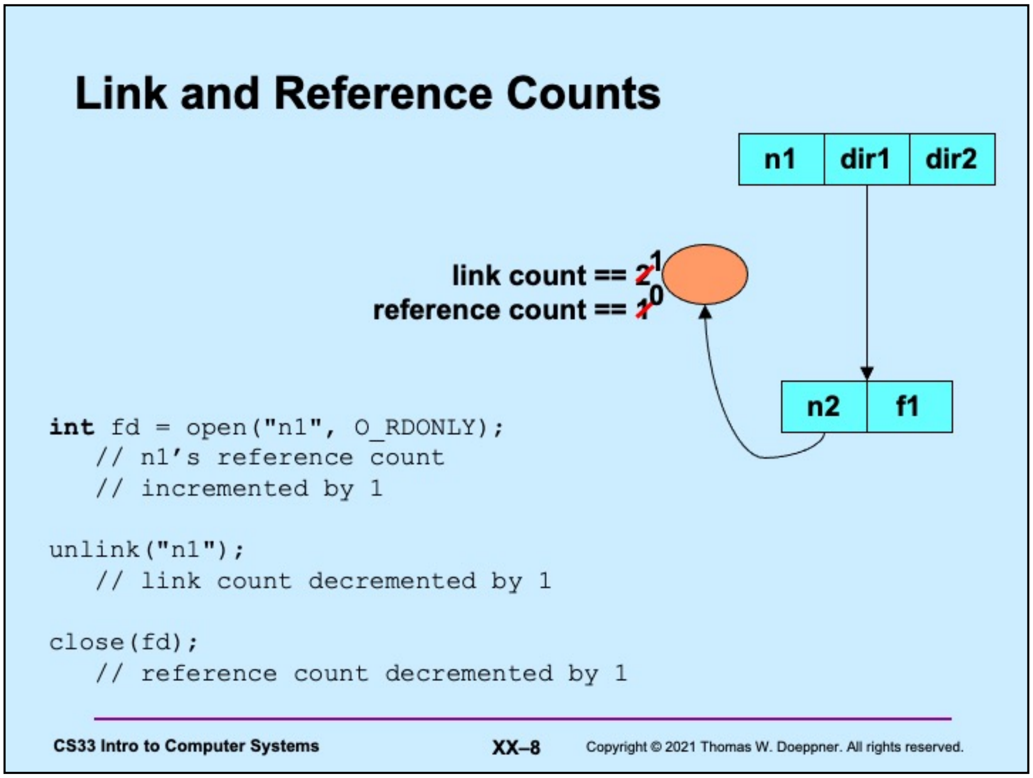
Link and Reference Counts



```
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
```



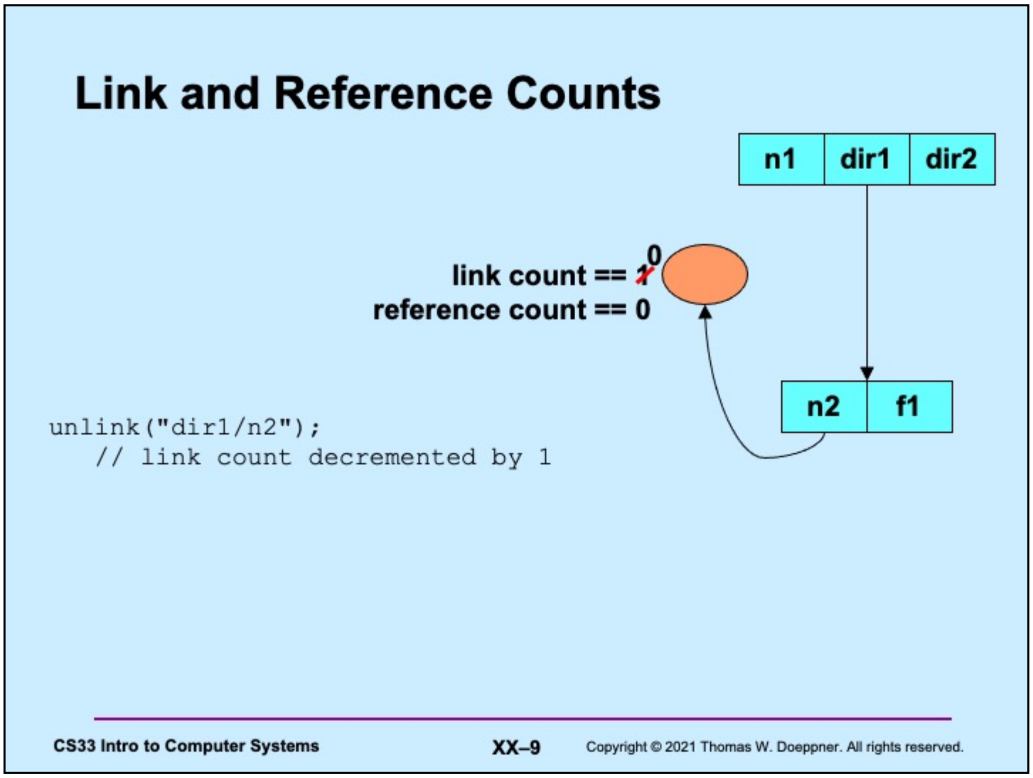
Link and Reference Counts

```
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
```

A file is deleted if and only if both its link and reference counts are zero.



Link and Reference Counts

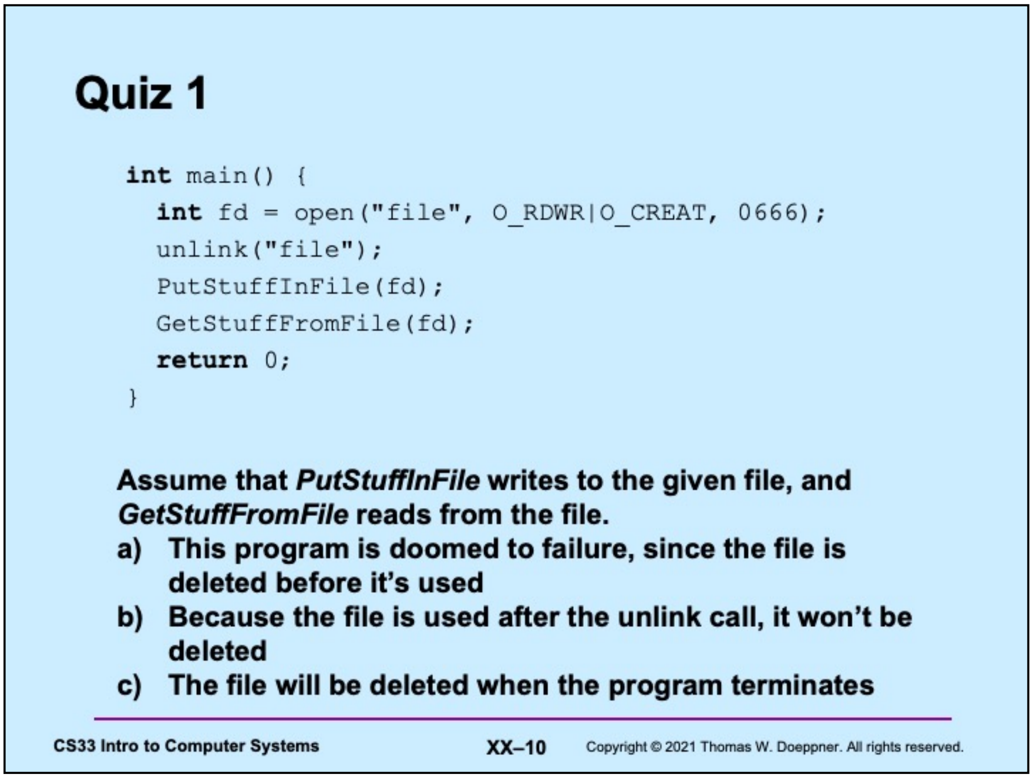
n1 | dir1 | dir2

n2 | f1

link count == ~~1~~ 0
reference count == 0

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

A file is deleted if and only if both its link and reference counts are zero.



Quiz 1

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

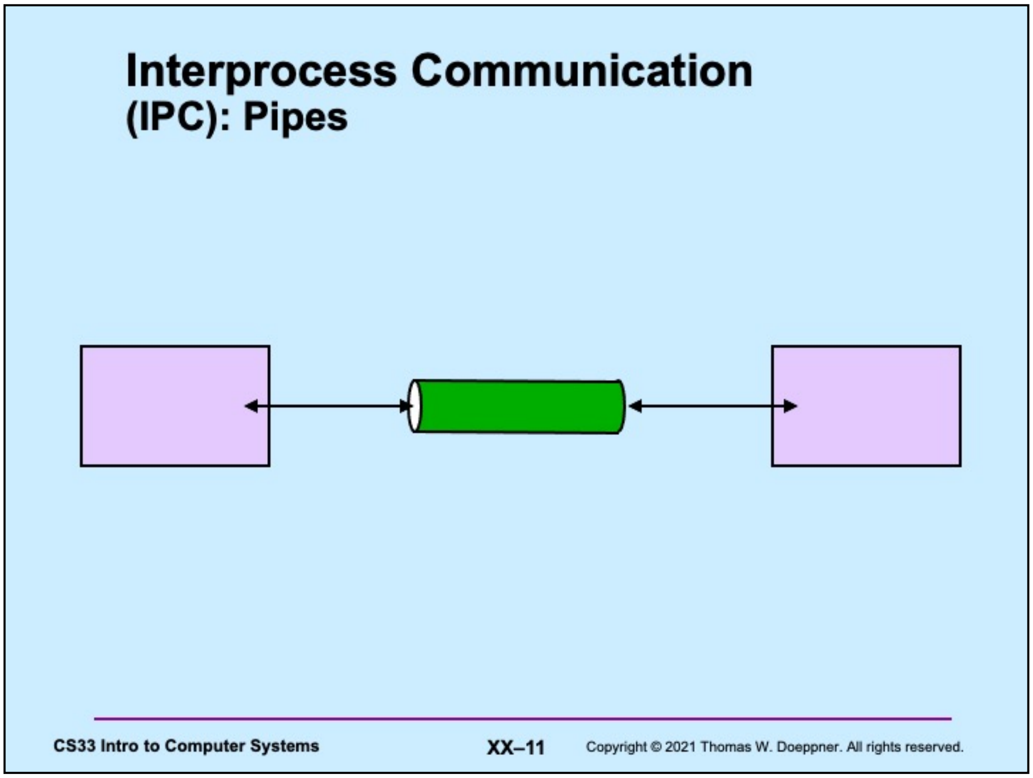
Assume that *PutStuffInFile* writes to the given file, and *GetStuffFromFile* reads from the file.

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

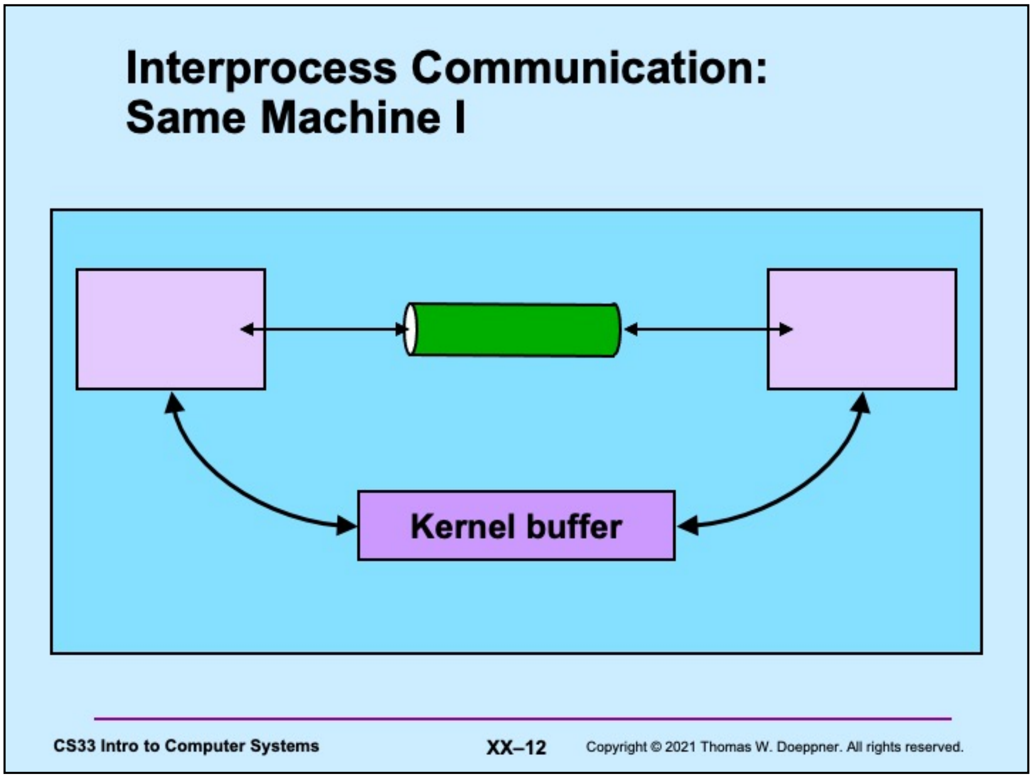
c) The file will be deleted when the program terminates

Note that when a process terminates, all its open files are automatically closed.



Interprocess Communication (IPC): Pipes

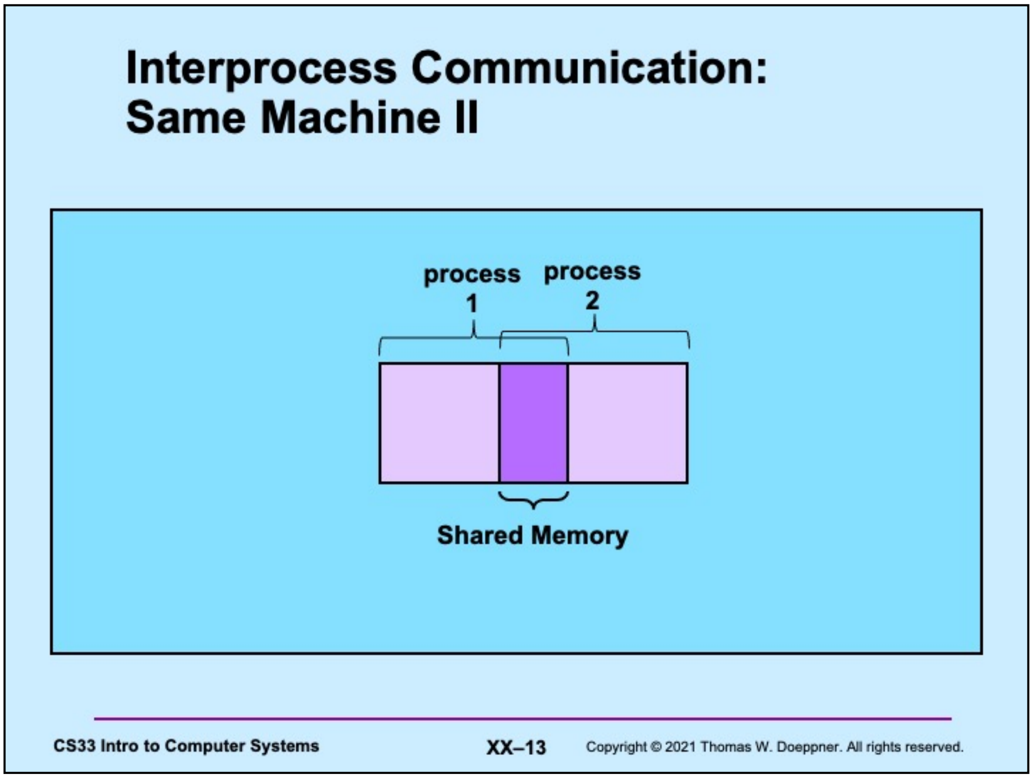
A rather elegant way for different processes to communicate is via a pipe: one process puts data into a pipe, another process reads the data from the pipe.



Interprocess Communication: Same Machine I

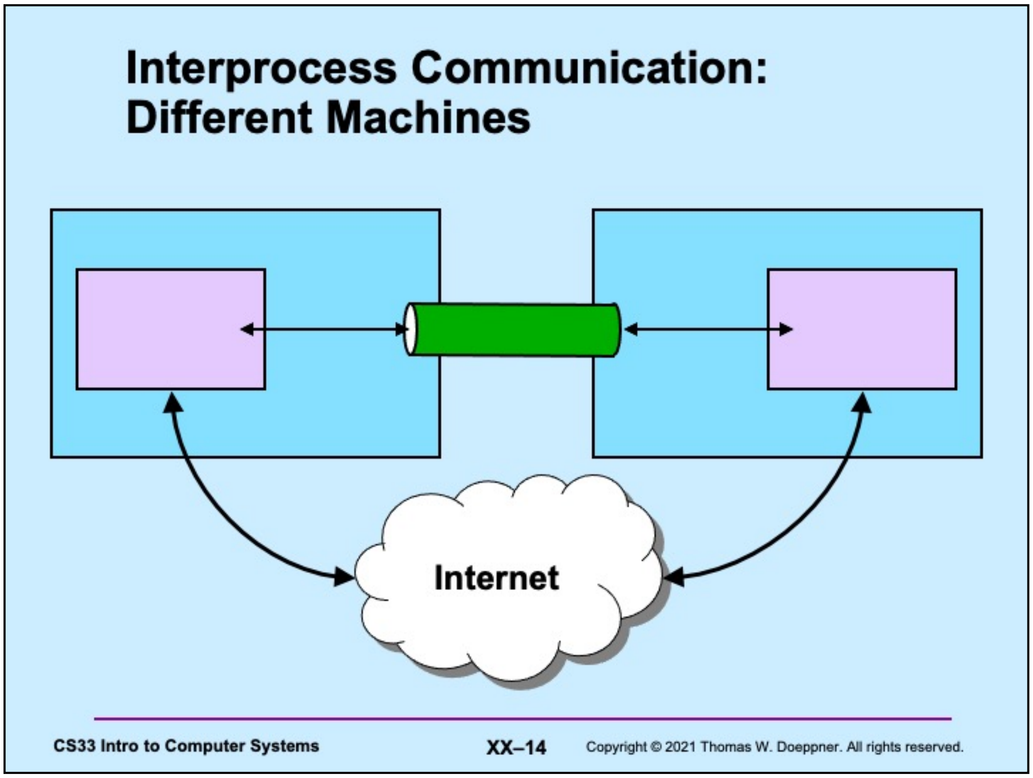
Kernel buffer

The implementation of a pipe involves the sending process using a write system call to transfer data into a kernel buffer. The receiving process fetches the data from the buffer via a read system call.



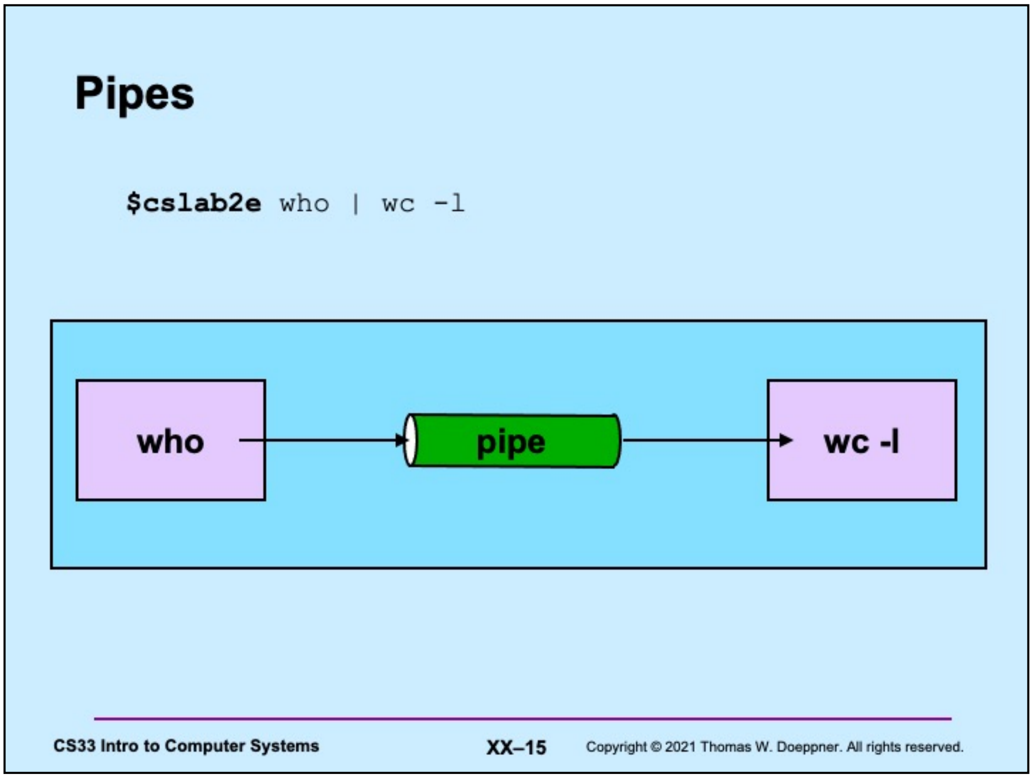
Interprocess Communication: Same Machine II

Another way for processes to communicate is for them to arrange to have some memory in common via which they share information. We discuss this approach later in the semester.



Interprocess Communication: Different Machines

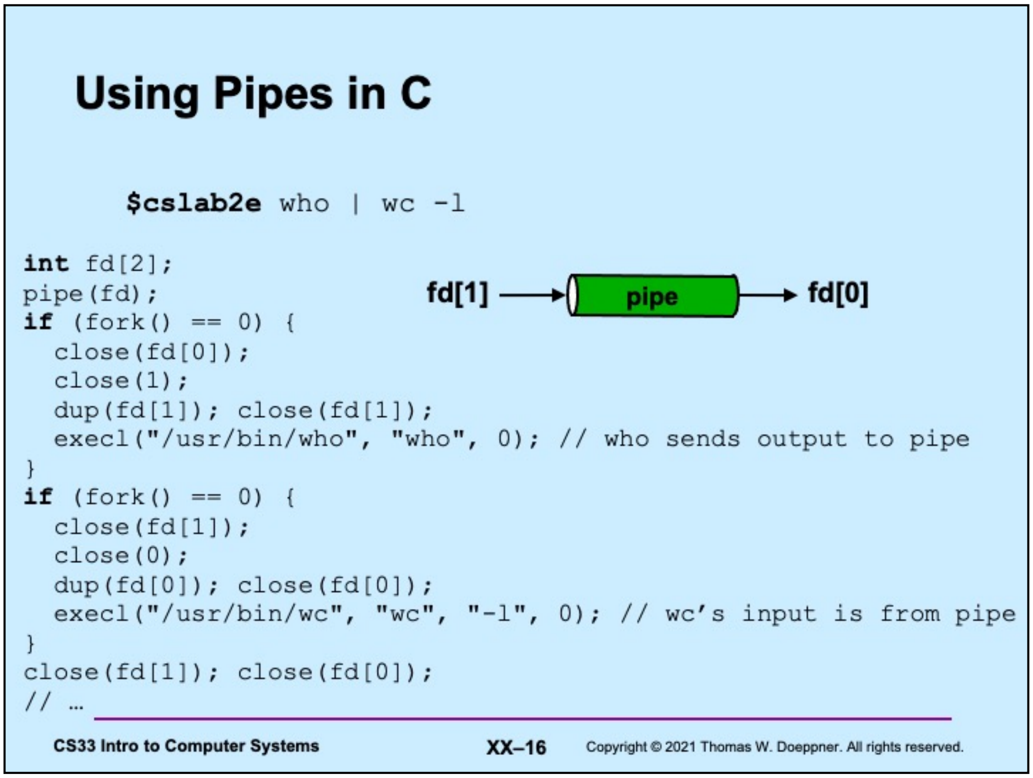
The pipe abstraction can also be made to work between processes on different machines. We discuss this later in the semester.



Pipes

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

The vertical bar ("|") is the pipe symbol in the shell. The syntax shown above represents creating two processes, one running who and the other running wc. The standard output of who is setup to be the pipe; the standard input of wc is setup to be the pipe. Thus, the output of who becomes the input of wc. The "-l" argument to wc tells it to count and print out the number of lines that are input to it. The who command writes to standard output the login names of all logged in users. The combination of the two produces the number of users who are currently logged in.



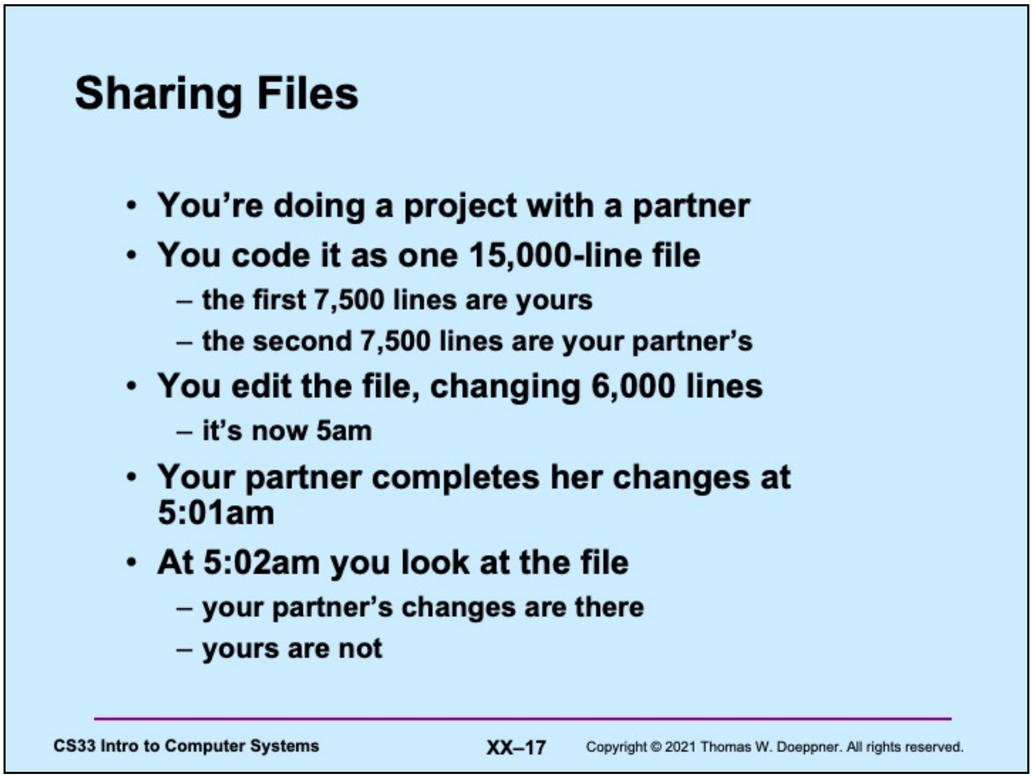
Using Pipes in C

```
$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]);
// …
```

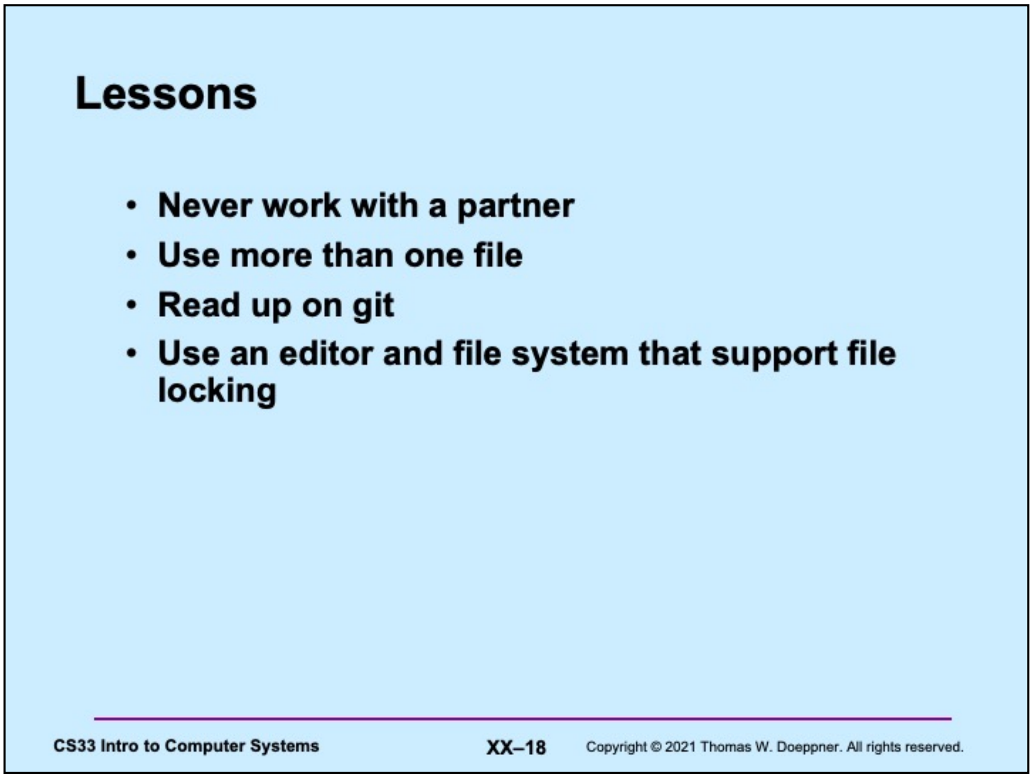
CS33 Intro to Computer Systems XX–16

The **pipe** system call creates a “pipe” in the kernel and sets up two file descriptors. One, in fd[1], is for writing to the pipe; the other, in fd[0], is for reading from the pipe. The input end of the pipe is set up to be **stdout** for the process running **who**, and the output end of the pipe is closed, since it’s not needed. Similarly, the input end of the pipe is set up to be **stdin** for the process running **wc**, and the input end is closed. Since the parent process (running the shell) has no further need for the pipe, it closes both ends. When neither end of the pipe is open by any process, the system deletes it. If a process reads from a pipe for which no process has the input end open, the read returns 0, indicating end of file. If a process writes to a pipe for which no process has the output end open, the write returns -1, indicating an error and **errno** is set to EPIPE; the process also receives the SIGPIPE signal, which we explain in the next lecture.



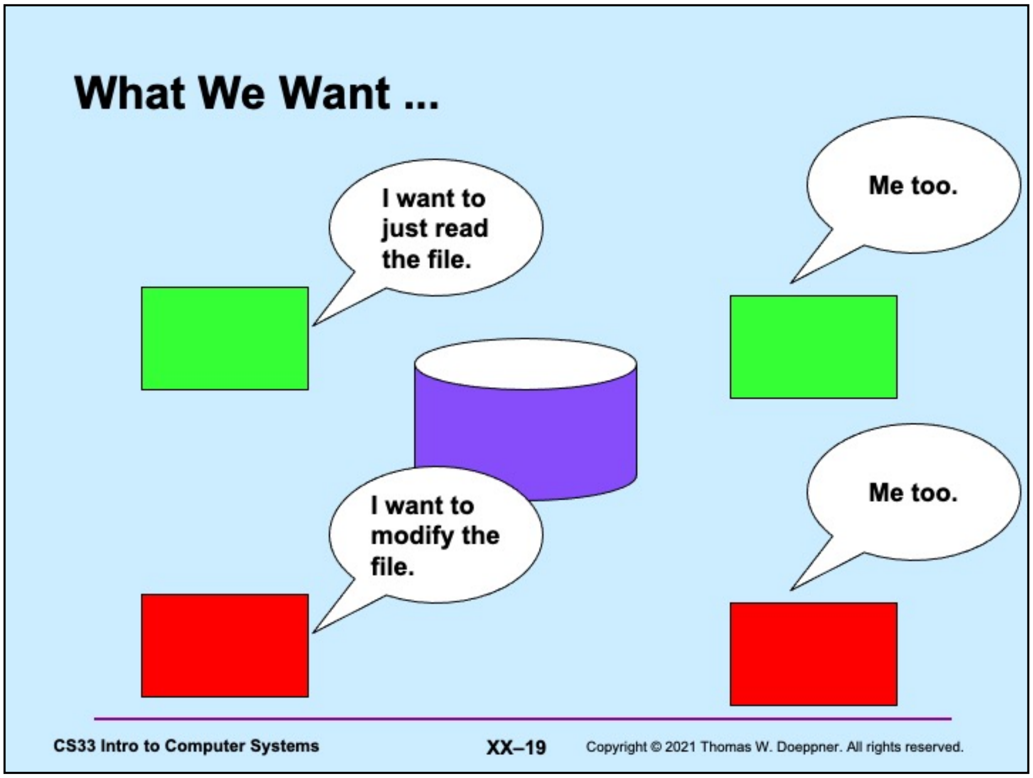
Sharing Files

- 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



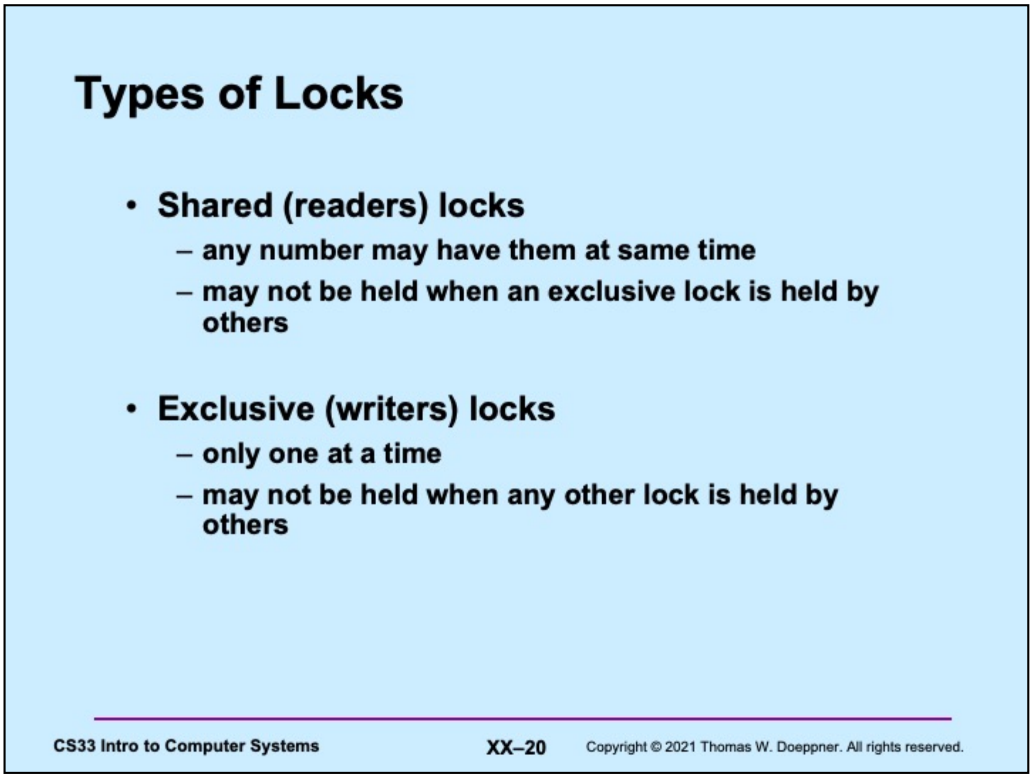
Lessons

- 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 ...

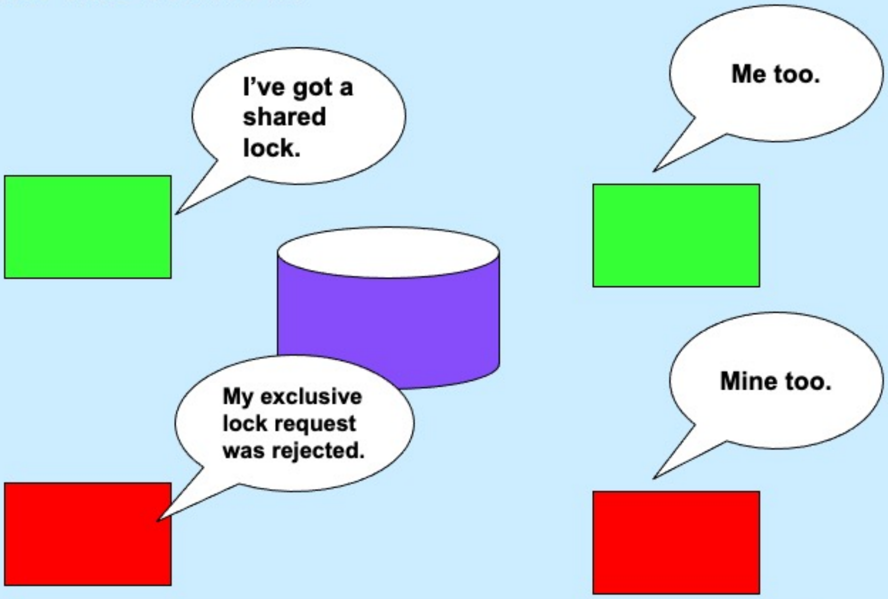
I want to just read the file.
Me too.
I want to modify the file.
Me too.



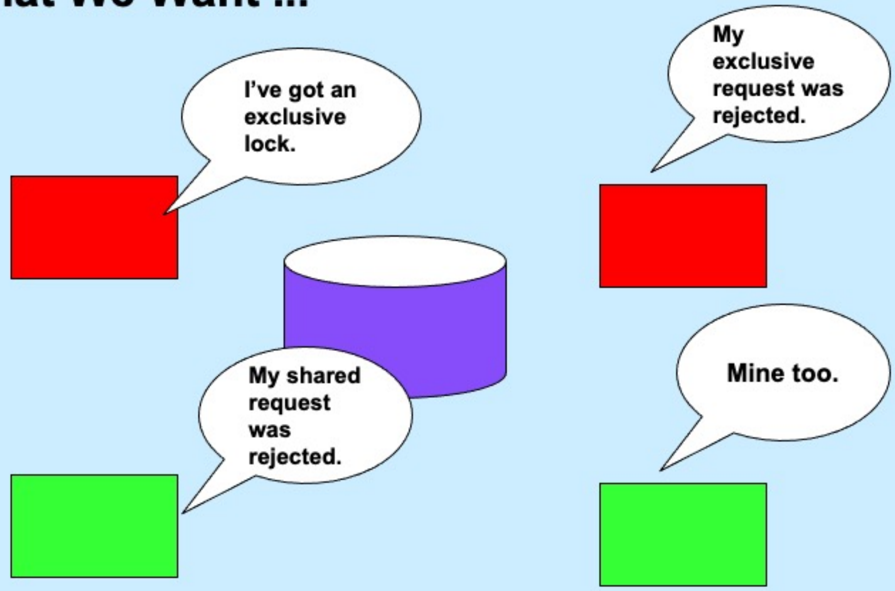
Types of Locks

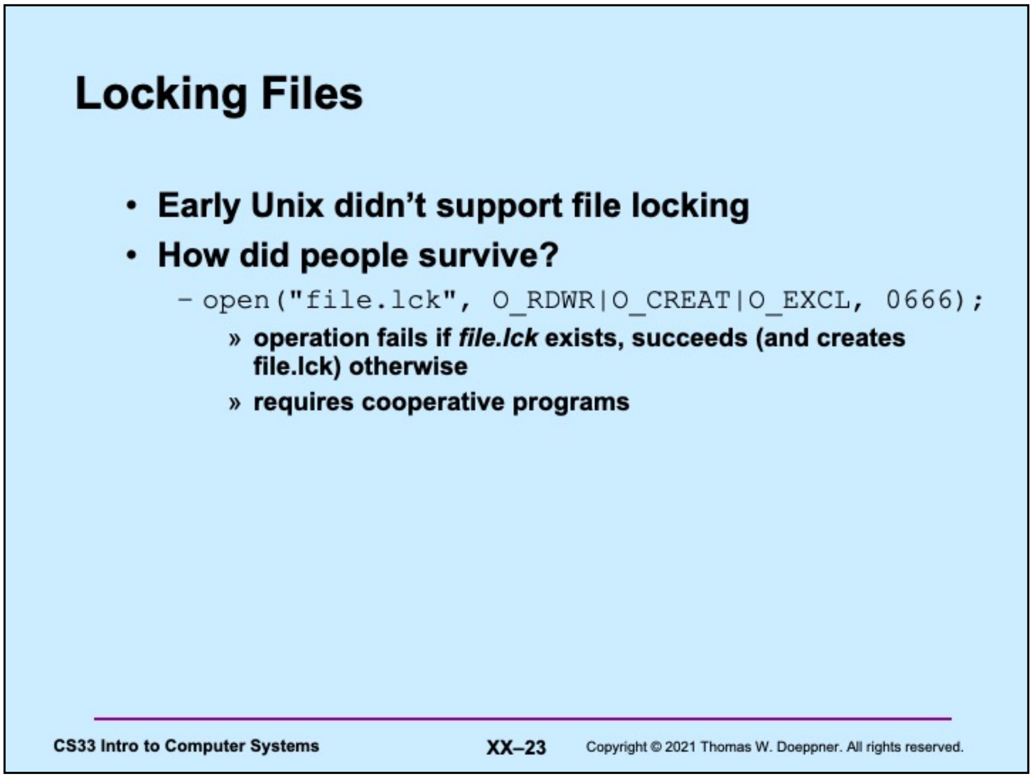
- **Shared (readers) locks**
  - **any number may have them at same time**
  - **may not be held when an exclusive lock is held by others**
- **Exclusive (writers) locks**
  - **only one at a time**
  - **may not be held when any other lock is held by others**

What We Want ...


I've got a shared lock.
Me too.
My exclusive lock request was rejected.
Mine too.

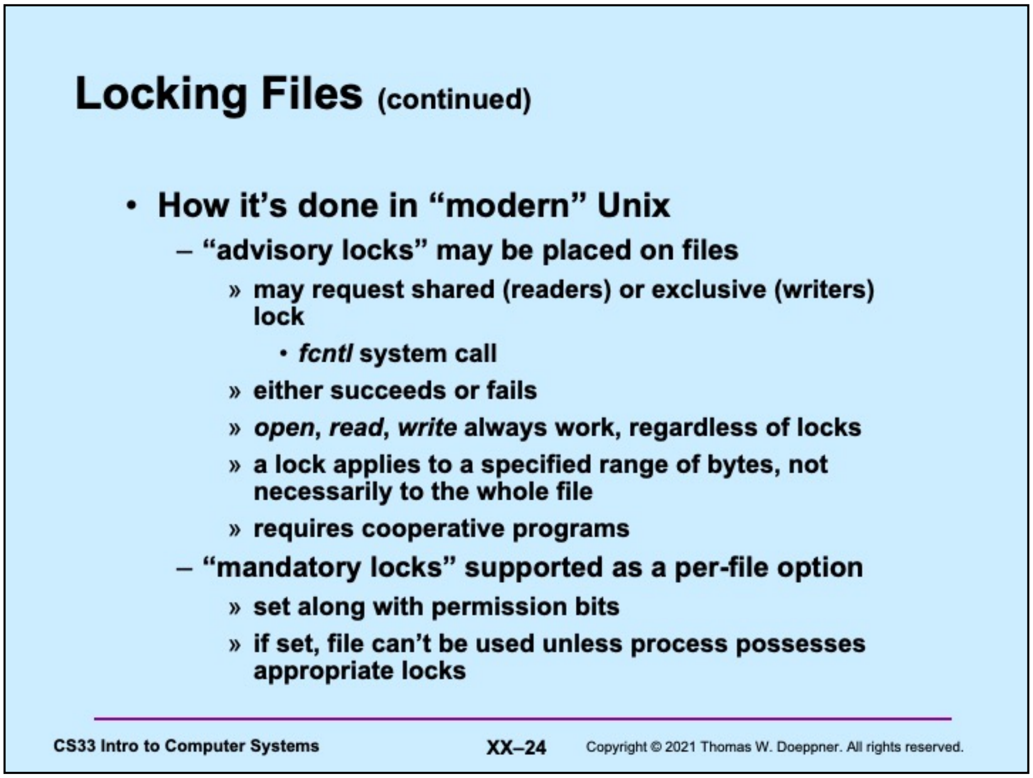
What We Want ...


I’ve got an exclusive lock.
My exclusive request was rejected.
My shared request was rejected.
Mine too.



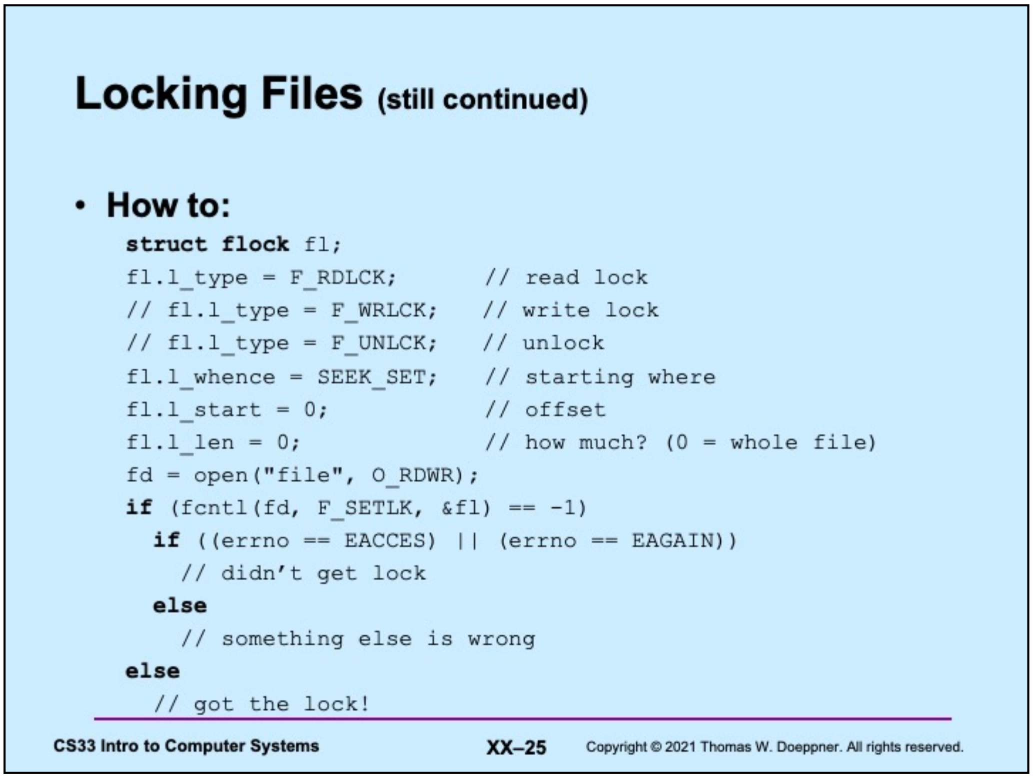
Locking Files

- 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



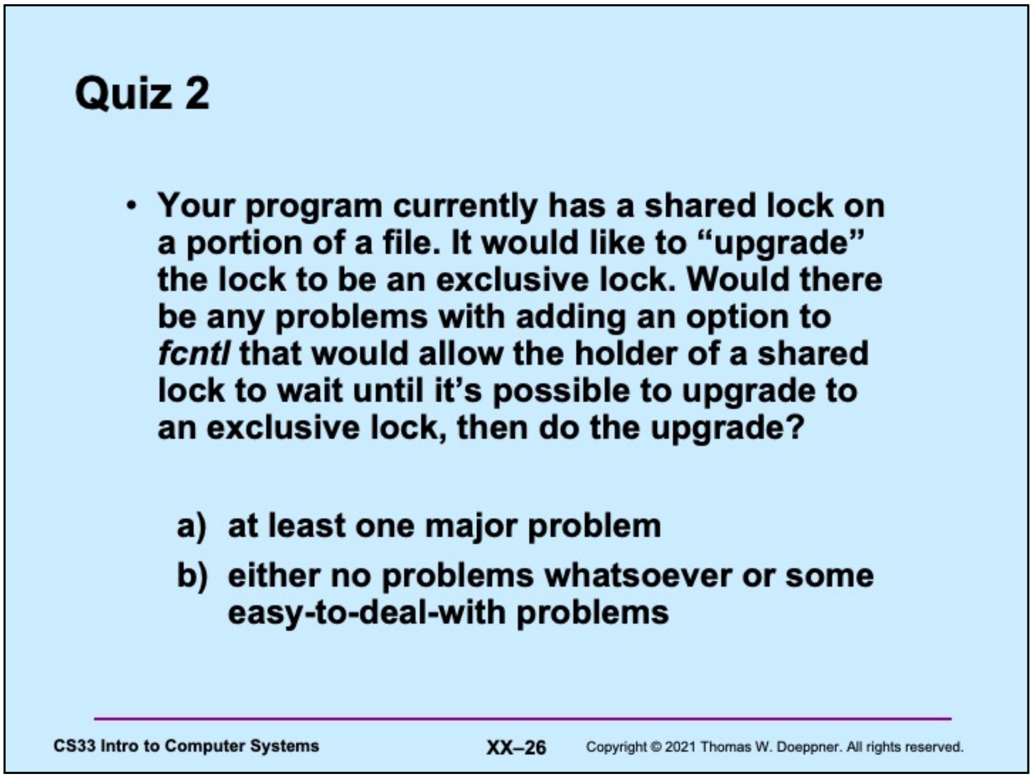
Locking Files (still continued)

- **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!
```

Alternatively, one may use l_type values of F_RDLCKW and F_WRLCKW to wait until the lock may be obtained, rather than to return an error if it can’t be obtained.

Whether the lock is mandatory or advisory depends upon the per-file settings.

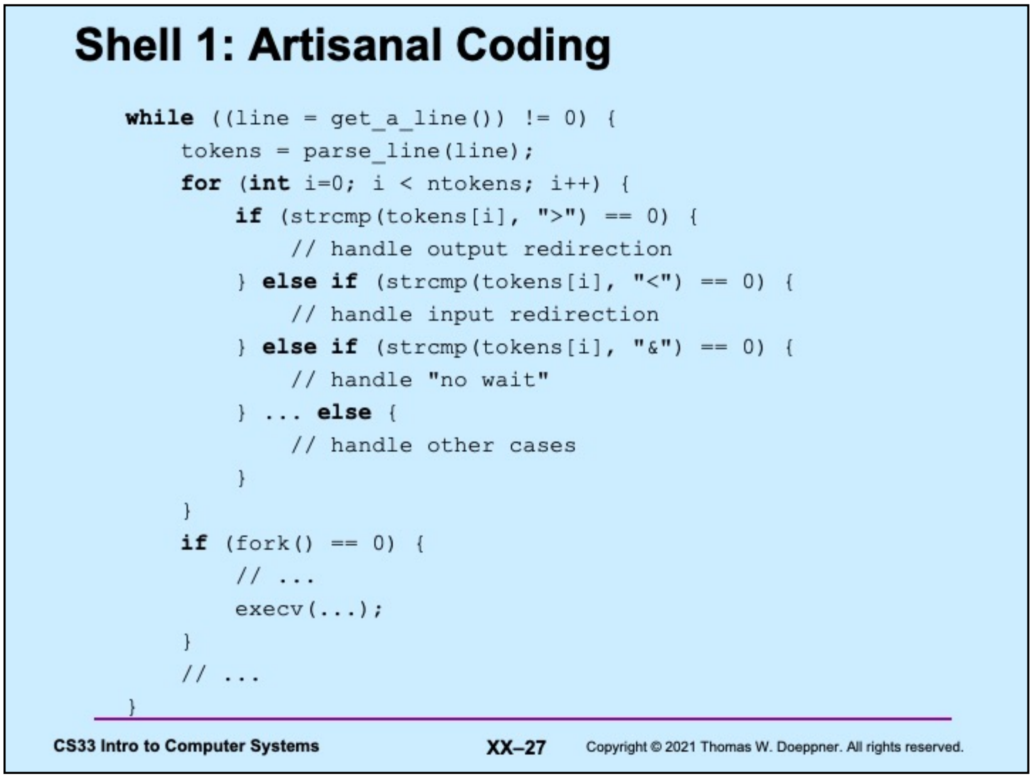


Quiz 2

- 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

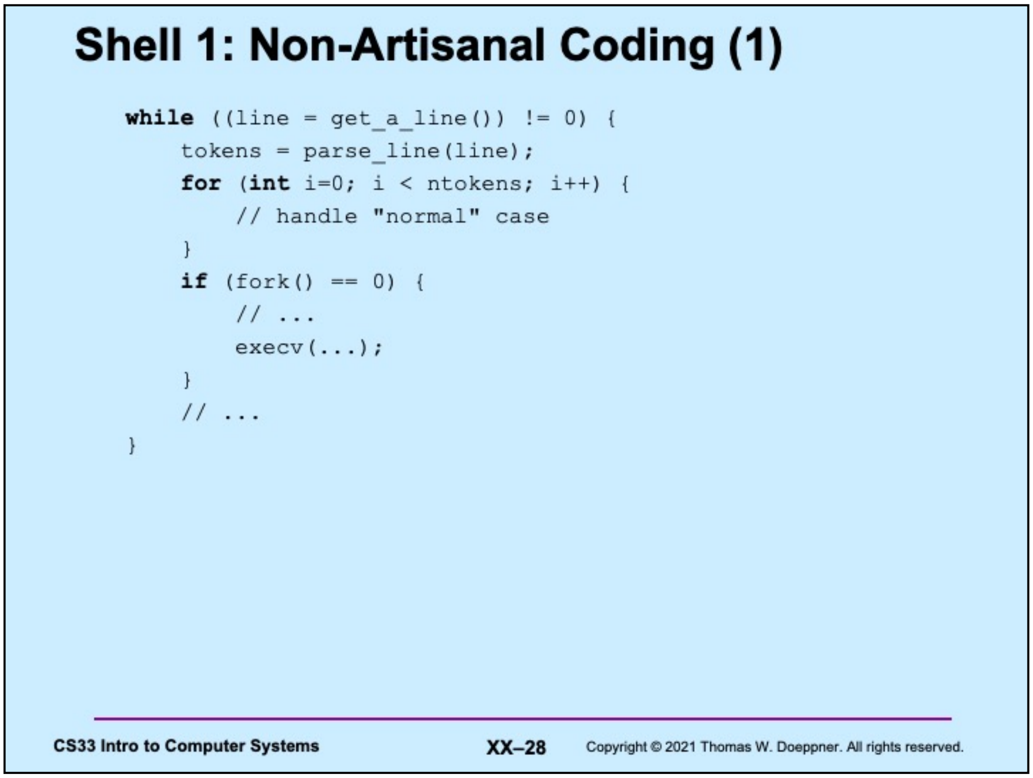


Shell 1: Artisanal Coding

```
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(...);
    }
    // ...
}
```

This is, of course, over simplified. The complete program should be 200 or so lines long.

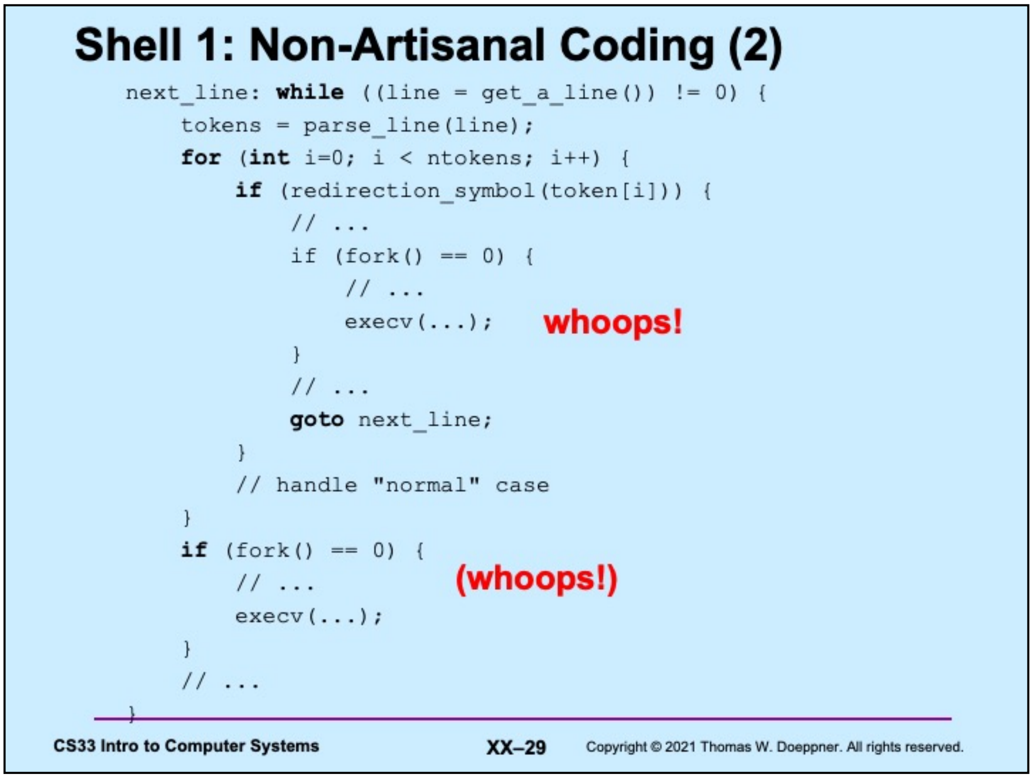
Note that "handle x" might simply involve taking note of x, then dealing with it later.



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(...);
    }
    // ...
}
```

One first writes the code assuming no redirection symbols and no &s. That's perfectly reasonable.



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(...);
    }
    // ...
}
```

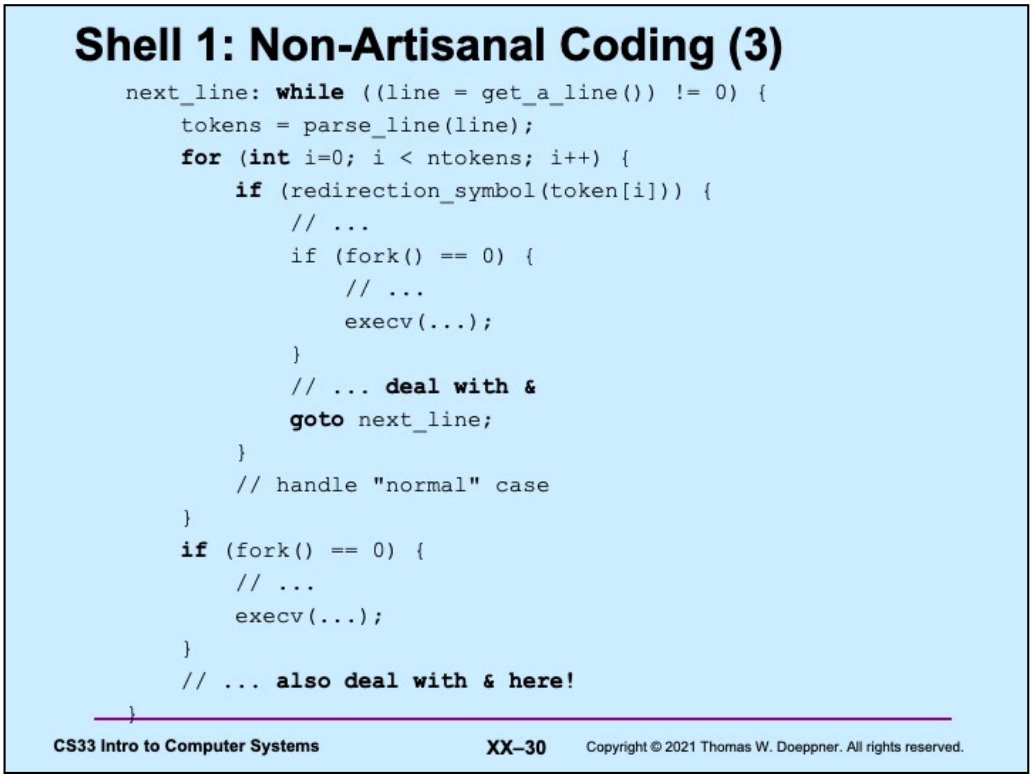
whoops!

(whoops!)

CS33 Intro to Computer Systems XX–29

The next step is to deal with redirection symbols. Rather than modify the fork/exec code so as to work for both cases, it's copied into the new case and modified there. Thus, we now have two versions of the fork/exec code to maintain. If we find a bug in one, we need to remember to fix it in both.

At this point it's becoming difficult for you to debug your code, and really difficult for TAs to figure out what you're doing so they can help you.

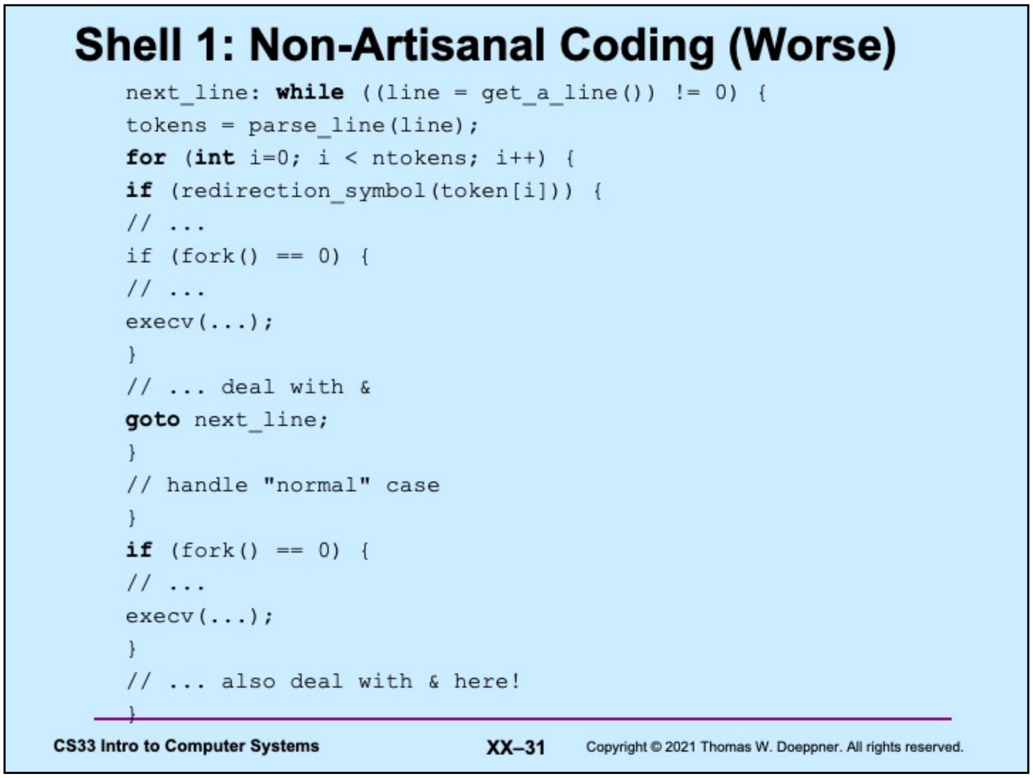


Shell 1: Non-Artisanal Coding (3)

```
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!
}
```

We now have to handle & in multiple places.

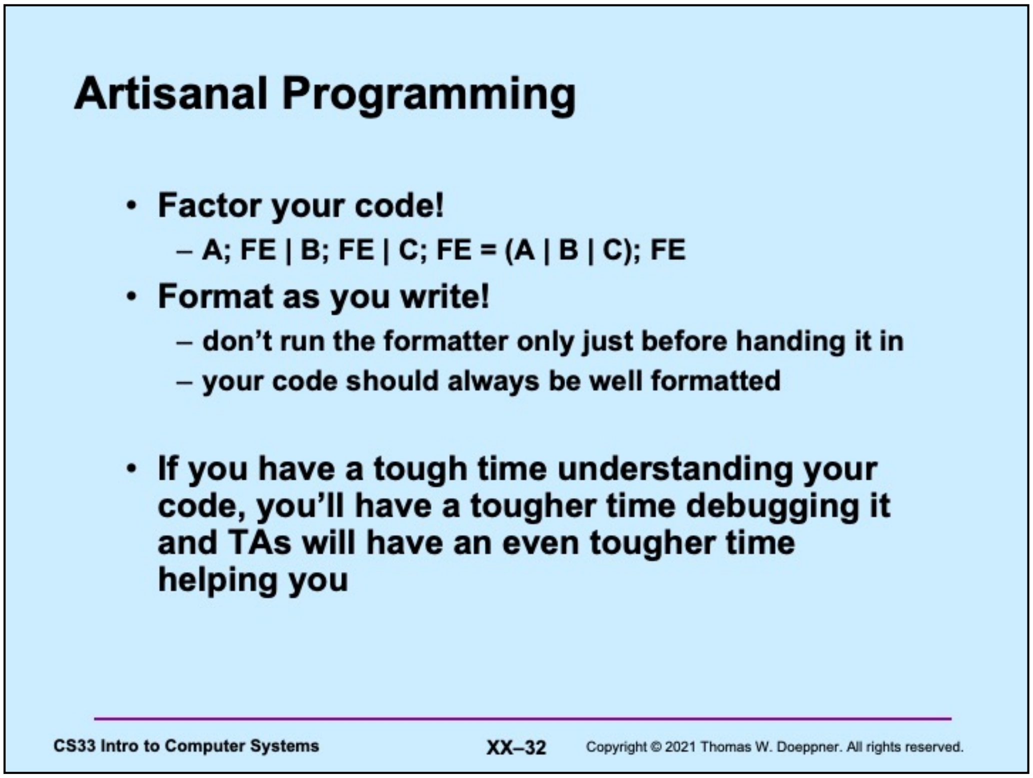
If done this way, you could well have a 700-line program (the artisanal code took around 200 lines).



Shell 1: Non-Artisanal Coding (Worse)

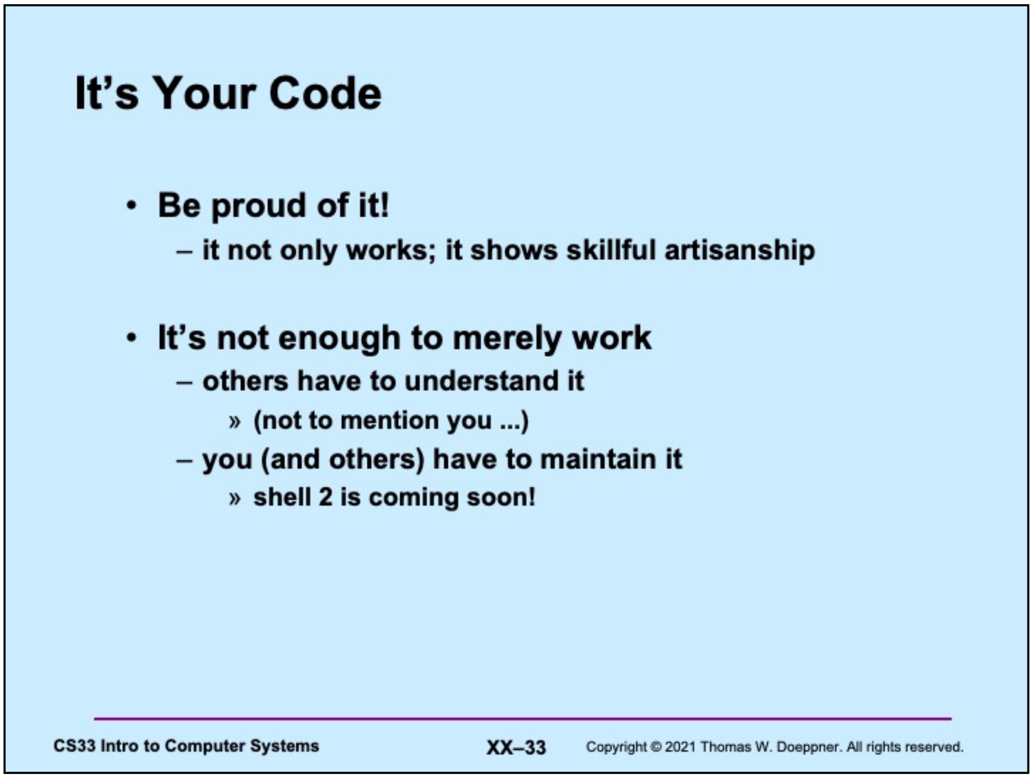
```
next_line: while ((line = get_a_line()) != 0) {
tokens = parse_line(line);
for (int i=0; i < ntokens; i++) {
if (redirection_symbol(token[i])) {
// ...
if (fork() == 0) {
// ...
execv(...);
}
// ... deal with &
goto next_line;
}
// handle "normal" case
}
if (fork() == 0) {
// ...
execv(...);
}
//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

If the code is poorly formatted, it's even tougher to understand.



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!