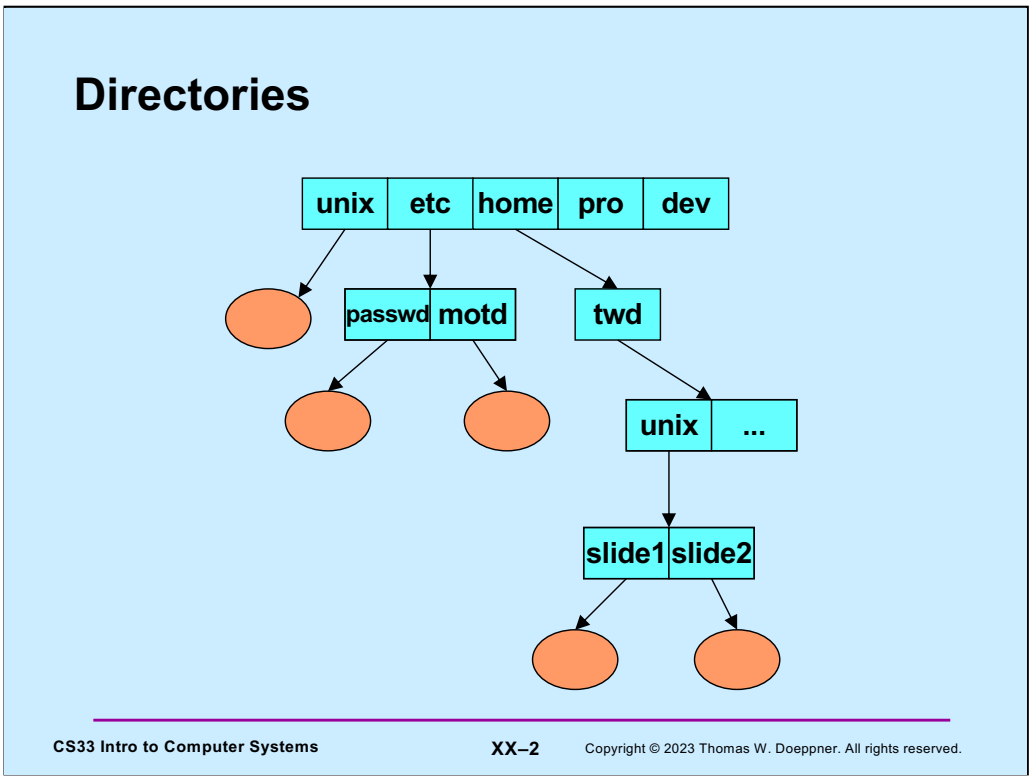
CS 33

Files Part 3



Here is a portion of a Unix directory tree. The ovals represent files, the rectangles represent directories (which are really just special cases of files).

Directory Representation

Component Name	Inode Number

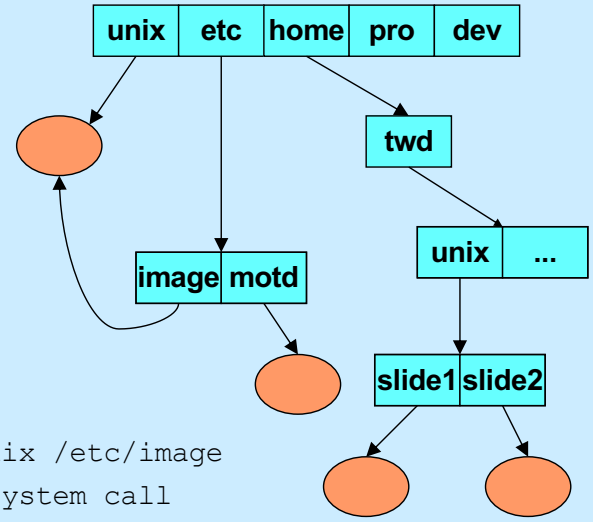
directory entry

.	1
..	1
unix	117
etc	4
home	18
pro	36
dev	93

A simple implementation of a directory consists of an array of pairs of **component name** and **inode number**, where the latter identifies the target file's *inode* to the operating system (an inode is data structure maintained by the operating system that represents a file). Note that every directory contains two special entries, "." and "..". The former refers to the directory itself, the latter to the directory's parent (in the case of the slide, the directory is the root directory and has no parent, thus its ".." entry is a special case that refers to the directory itself).

While this implementation of a directory was used in early file systems for Unix, it suffers from a number of practical problems (for example, it doesn't scale well for large directories). It provides a good model for the semantics of directory operations, but directory implementations on modern systems are more complicated than this (and are beyond the scope of this course).

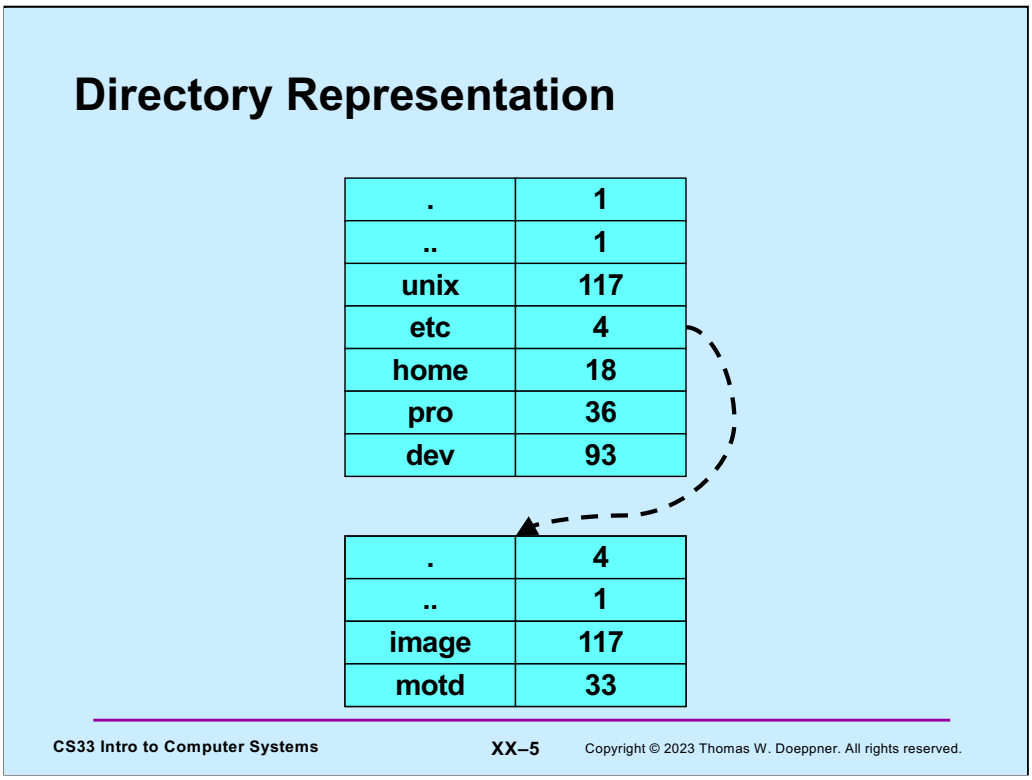
Hard Links




```
$ ln /unix /etc/image
# link system call
```


Here are two directory entries referring to the same file. This is done, via the shell, through the **ln** command which creates a (hard) link to its first argument, giving it the name specified by its second argument.

The shell's "ln" command is implemented using the **link** system call.

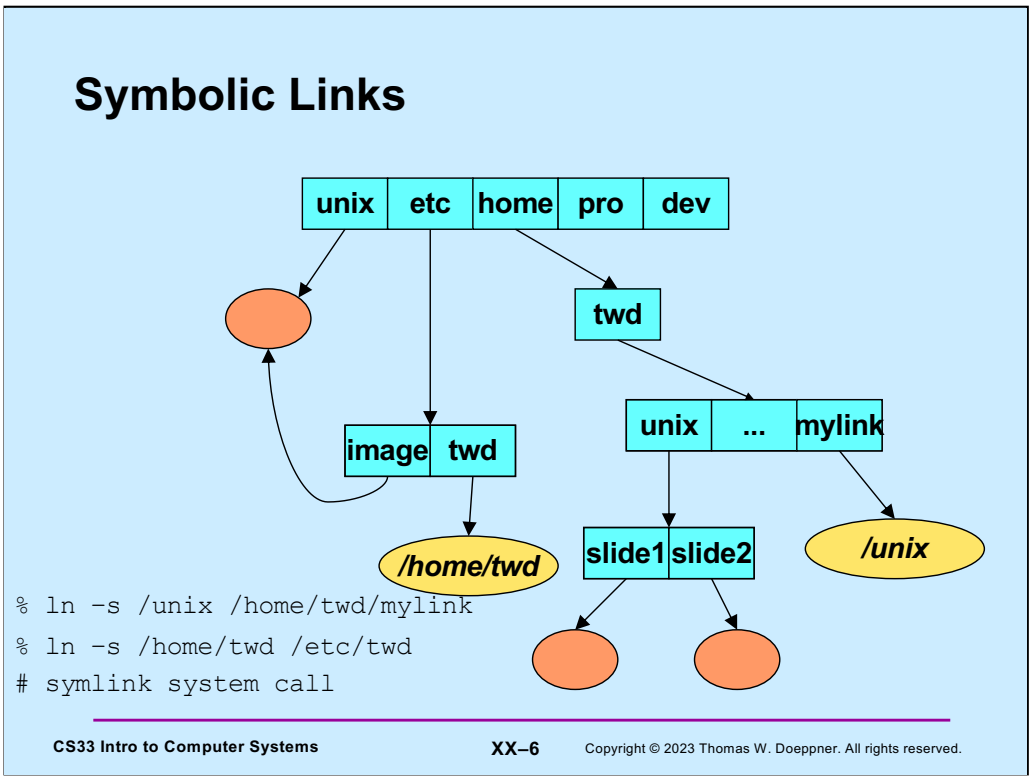


Here are the (abbreviated) contents of both the **root** (/) and **/etc** directories, showing how **/unix** and **/etc/image** are the same file. Note that if the directory entry **/unix** is deleted (via the shell's "rm" command), the file (represented by inode 117) continues to exist, since there is still a directory entry referring to it. However, if **/etc/image** is also deleted, then the file has no more links and is removed. To implement this, the file's inode contains a link count, indicating the total number of directory entries that refer to it. A file is actually deleted only when its inode's link count reaches zero.

Note: suppose a file is open, i.e. is being used by some process, when its link count becomes zero. Rather than delete the file while the process is using it, the file will continue to exist until no process has it open. Thus the inode also contains a reference count indicating how many times it is open: in particular, how many system file table entries point to it. A file is deleted when and only when both the link count and this reference count become zero.

The shell's "rm" command is implemented using the **unlink** system call.

Note that **/etc/..** refers to the root directory.



Differing from a hard link, a **symbolic link** (often called soft link) is a special kind of file containing the name of another file. When the kernel processes such a file, rather than simply retrieving its contents, it makes use of the contents by replacing the portion of the directory path that it has already followed with the contents of the soft-link file and then following the resulting path. Thus referencing **/home/twd/mylink** results in the same file as referencing **/unix**. Referencing **/etc/twd/unix/slide1** results in the same file as referencing ***/home/twd/unix/slide1***.

The shell's "ln" command with the "-s" flag is implemented using the **symlink** system call.

Working Directory

- **Maintained in kernel for each process**
  - **paths not starting from "/" start with the working directory**
  - **changed by use of the *chdir* system call**
    - » ***cd* shell command**
  - **displayed (via shell) using "pwd"**
    - » **how is this done?**

The **working directory** is maintained in the kernel for each process. Whenever a process attempts to follow a path that doesn't start with "/", it starts at its working directory (rather than at "/").

Open

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path, int options [, mode_t mode])
```

- options
  - » O_RDONLY — open for reading only
  - » O_WRONLY — open for writing only
  - » O_RDWR — open for reading and writing
  - » O_APPEND — set the file offset to *end of file* prior to each *write*
  - » O_CREAT — if the file does not exist, then create it, setting its mode to *mode* adjusted by *umask*
  - » O_EXCL — if O_EXCL and O_CREAT are set, then *open* fails if the file exists
  - » O_TRUNC — delete any previous contents of the file

Here's a partial list of the options available as the second argument to open. (Further options are often available, but they depend on the version of Unix.) Note that the first three options are mutually exclusive: one, and only one, must be supplied. We discuss the third argument to *open*, mode, in the next few slides.

Appending Data to a File (1)

```
int fd = open("file", O_WRONLY);
lseek(fd, 0, SEEK_END);
    // sets the file location to the end
write(fd, buffer, bsize);
    // does this always write to the
    // end of the file?
```


We'd like to write data to the end of a file. One approach, shown here, is to use the **lseek** system call to set the file location in the file context structure to the end of the file. Once this is done, then when we write to the file, we're writing to its end and thus are appending data to the file.

However, this assumes that no other program is writing data to the end of the file at the same time. If another program were doing this, then the file could grow between our calls to **lseek** and **write**. If this happens, then the write would no longer be to the end of the file but would overwrite the data written by the other program.

Appending Data to a File (2)

```
int fd = open("file", O_WRONLY | O_APPEND);
write(fd, buffer, bsize);
    // this is guaranteed to write to the
    // end of the file
```


By using the O_APPEND option of open, we make certain that writes on this file descriptor are always to the end of file. If another program is doing this at the same time, the operating system makes certain that one doesn't start until after the other ends.

In the Shell ...

```
% program >> file
```

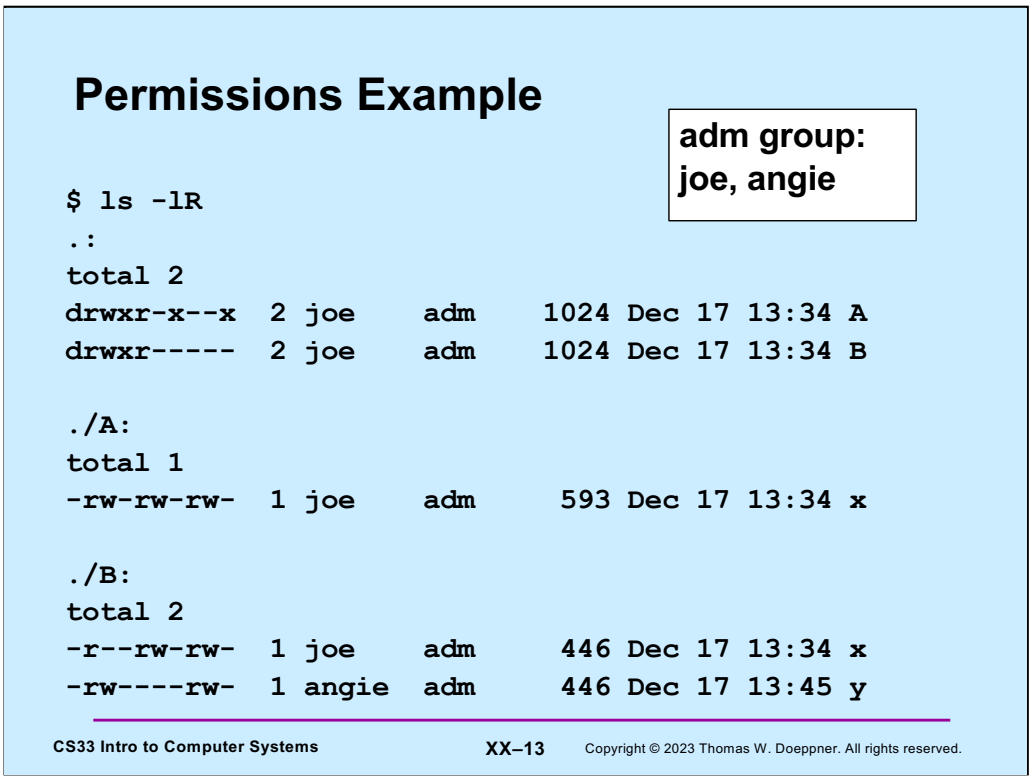

The ">>" operator tells the shell to open file with the O_APPEND flag so that writes are always to the end of the file.

File Access Permissions

- Who's allowed to do what?
  - who
    - » user (owner)
    - » group
    - » others (rest of the world)
  - what
    - » read
    - » write
    - » execute

Each file has associated with it a set of access permissions indicating, for each of three classes of principals, what sorts of operations on the file are allowed. The three classes are the owner of the file, known as **user**, the group owner of the file, known simply as **group**, and everyone else, known as **others**. The operations are grouped into the classes **read**, **write**, and **execute**, with their obvious meanings. The access permissions apply to directories as well as to ordinary files, though the meaning of execute for directories is not quite so obvious: one must have **execute** permission for a directory file in order to follow a path through it.

The system, when checking permissions, first determines the smallest class of principals the requester belongs to: user (smallest), group, or others (largest). It then, within the chosen class, checks for appropriate permissions.



The `ls -lR` command lists the contents of the current directory, its subdirectories, their subdirectories, etc. in long format (the `l` causes the latter, the `R` the former).

In the current directory are two subdirectories, **A** and **B**, with access permissions as shown in the slide. Note that the permissions are given as a string of characters: the first character indicates whether or not the file is a directory, the next three characters are the permissions for the owner of the file, the next three are the permissions for the members of the file's group's members, and the last three are the permissions for the rest of the world.

Quiz: the users **joe** and **angie** are members of the **adm** group; **leo** is not.

- May **leo** list the contents of directory *A*?
- May **leo** read *A/x*?
- May **angie** list the contents of directory *B*?
- May **angie** modify *B/y*?
- May **joe** modify *B/x*?

- May **joe** read B/y?

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, for example, 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.

Permission Bits

- It's worth your while to remember this!
  - read: 4
  - write: 2
  - execute: 1
  - read/write: 6
  - read/write/execute: 7

  - user:group:others
    - » 0751
      - rwx for user, rx for group, x for others
    - » 0640
      - rw for user, r for group, nothing for others

For each category (user, group, other), three bits represent their permissions. Thus these are usually viewed as octal digits.

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.)

Quiz 1

You get the following message when you attempt to execute ./program (a file that you own):

```
bash: ./program: Permission denied
```

You're first response should be:

a) find the source code for program and recompile it

b) execute the shell command
`chmod 0644 program`

c) execute the shell command
`chmod 0755 program`

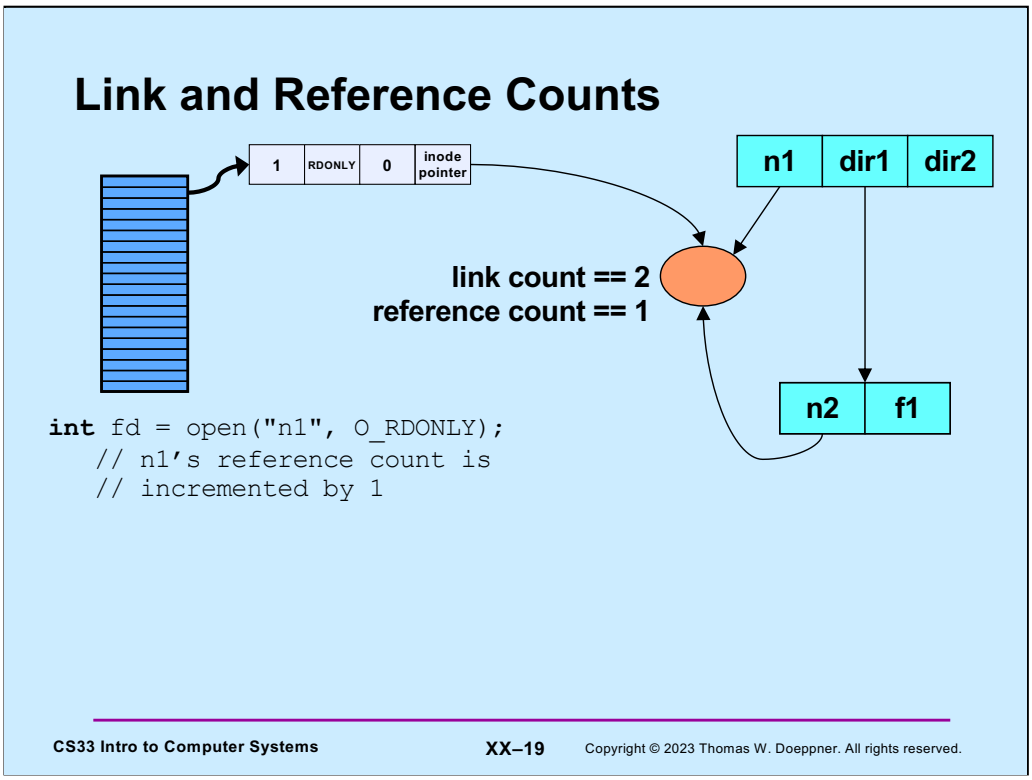
d) make an Ed post

Creating a File

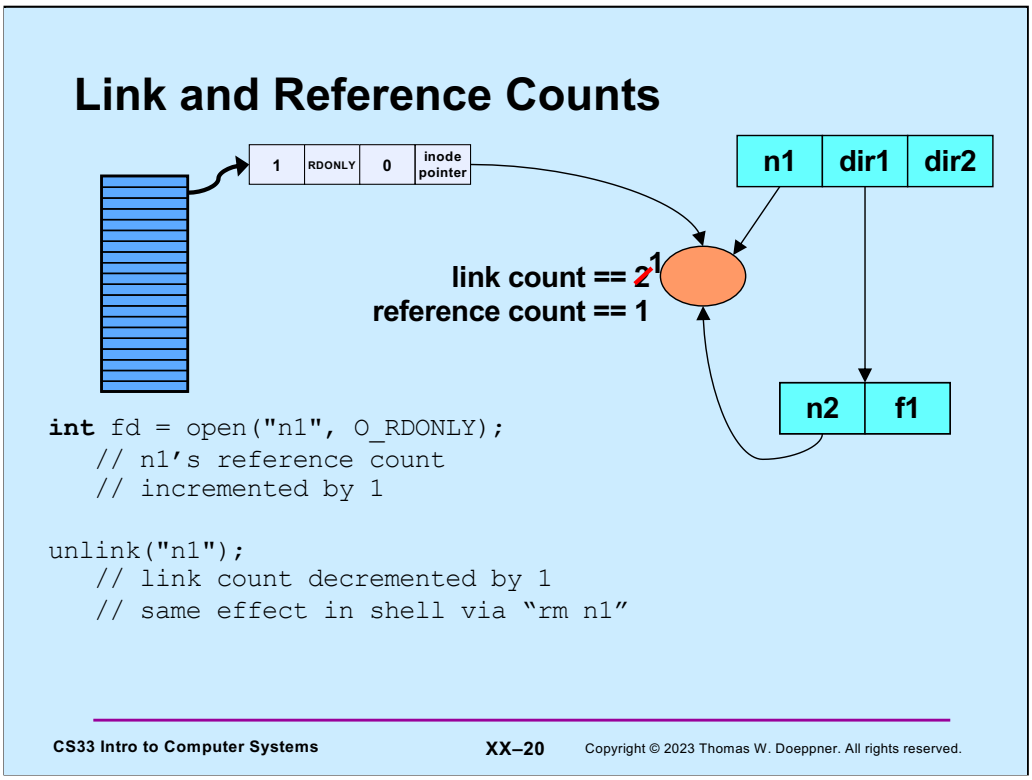
- **Use either *open* or *creat***
  - `open(const char *pathname, int flags, mode_t mode)`
    - » **flags must include O_CREAT**
  - `creat(const char *pathname, mode_t mode)`
    - » **open is preferred**
- **The *mode* parameter helps specify the permissions of the newly created file**
  - **permissions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

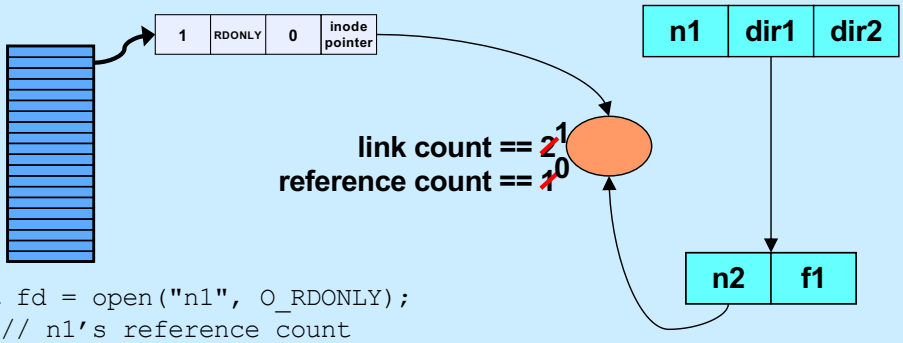


A file's link count is the number of directory entries that refer to it. There's a separate reference count that's the number of file context structures that refer to it (via the inode pointer – see slide XVII-9). These counts are maintained in the file's inode, which contains all information used by the operating system to refer to the file (on disk).



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

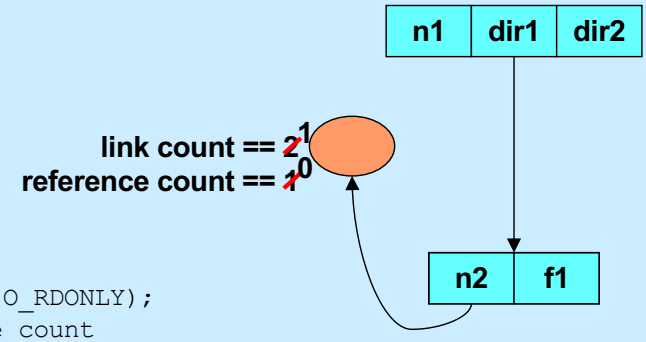



```
int fd = open("n1", O_RDONLY);
   // n1's reference count
   // incremented by 1

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

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

Link and Reference Counts



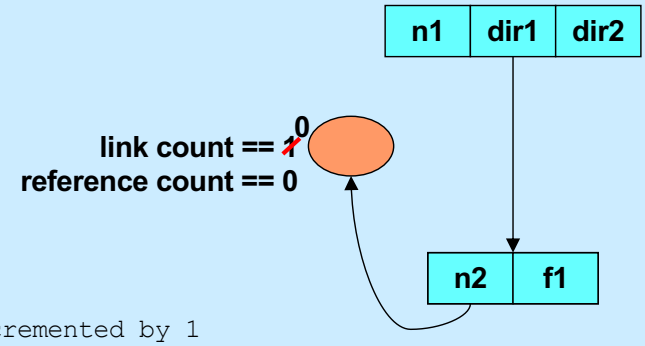

```
int fd = open("n1", O_RDONLY);
   // n1's reference count
   // incremented by 1

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




```
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 2

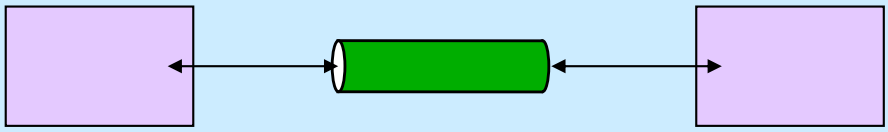
```
int main() {
  int fd = open("file", O_RDWR|O_CREAT, 0666);
  unlink("file");
  PutStuffInFile(fd);
  GetStuffFromFile(fd);
  return 0;
}
```

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

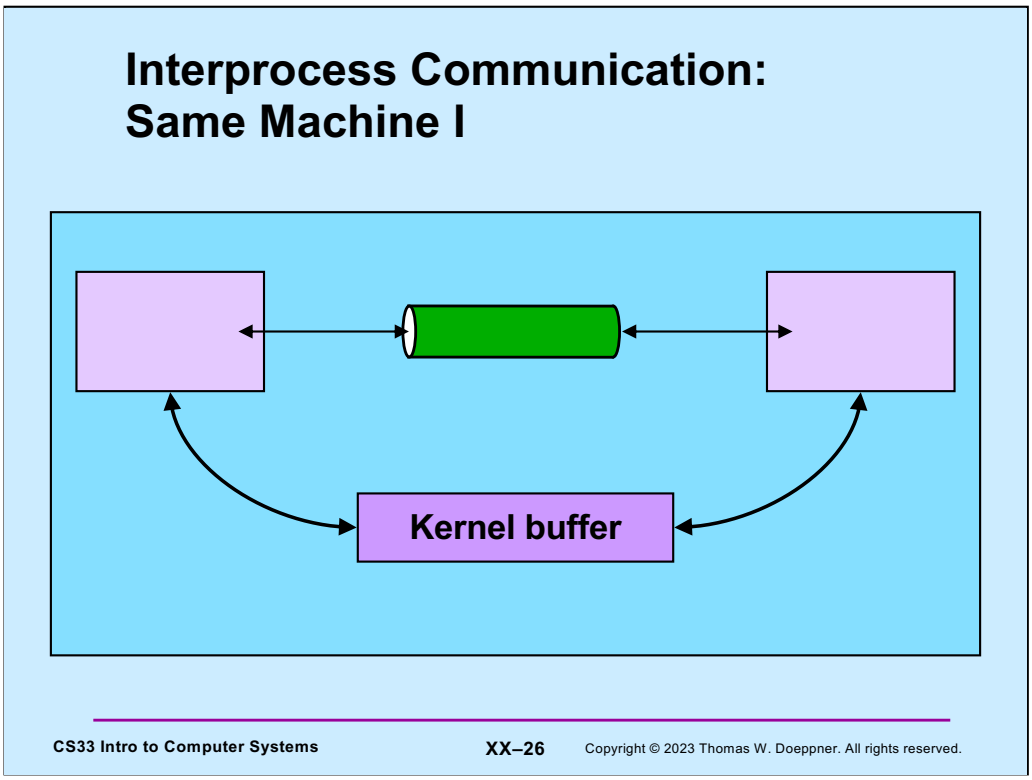
a) **This program is doomed to failure, since the file is deleted before it's used**
b) **Because the file is used after the unlink call, it won't be deleted**
c) **The file will be deleted when the program terminates**

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

Interprocess Communication (IPC): Pipes

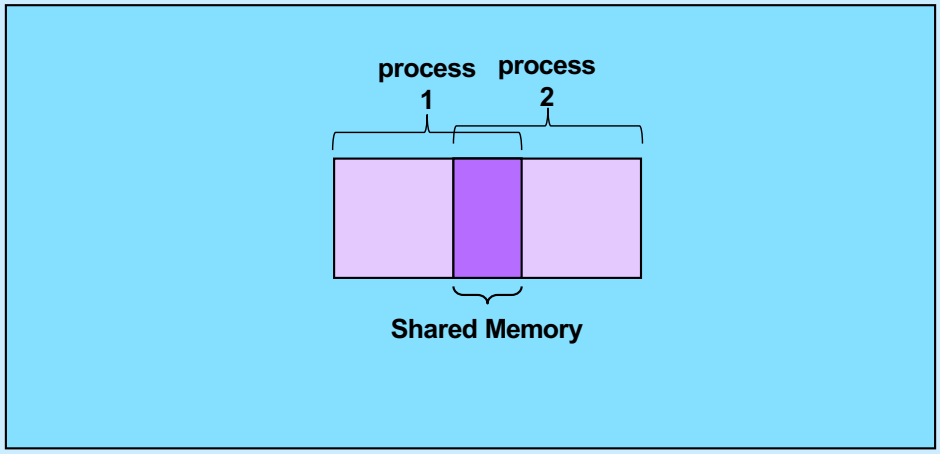


A rather elegant way for different processes to communicate is via a pipe: one process puts data into a pipe, another process reads the data from the pipe.

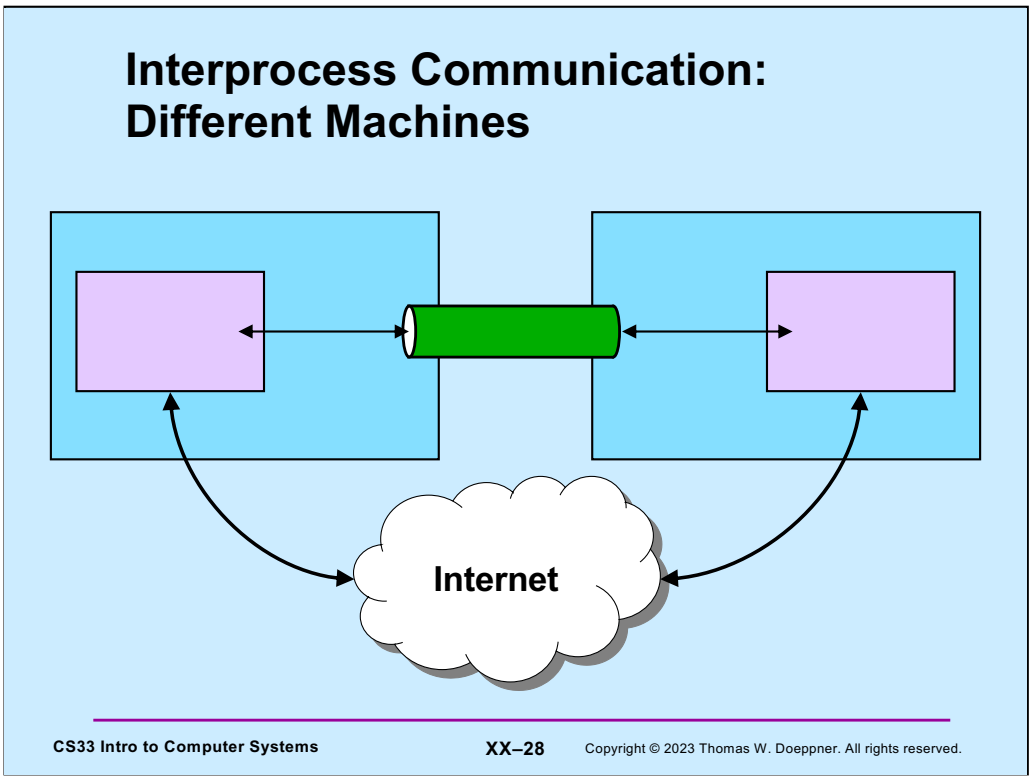


The implementation of a pipe involves the sending process using a write system call to transfer data into a kernel buffer. The receiving process fetches the data from the buffer via a read system call. We'll cover some of the details about how this works when we discuss multithreaded programming later in the semester.

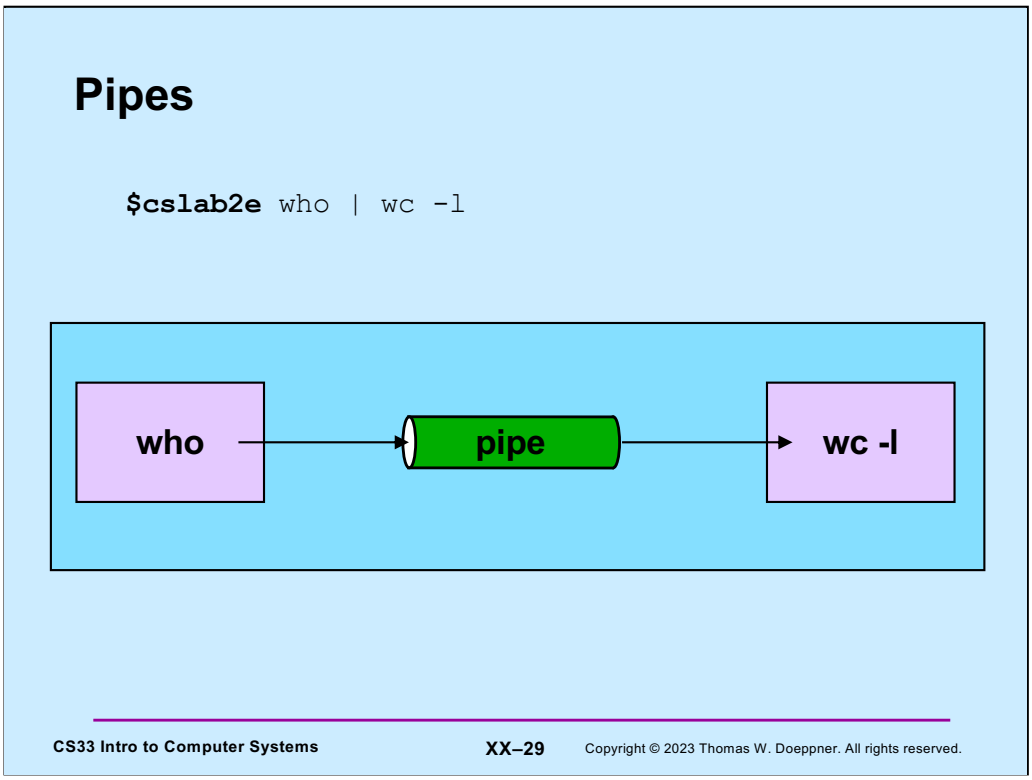
Interprocess Communication: Same Machine II



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.



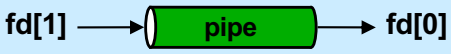
The pipe abstraction can also be made to work between processes on different machines. We discuss this later in the semester.



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]);
// …
```


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.

Quiz 3

We would like the output of prog1 be the input of prog2. Rather than use a pipe, we do the following:

```
$ prog1 >file &
$ prog2 <file
```

Would this work?

a) never
b) sometimes
c) always

Recall that when an ampersand (&) is placed at the end of a command, the shell doesn't wait for it to complete, but immediately goes on to the next command.

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.

Also note that “artisanal” anything is always better than “non-artisanal” anything.

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!)

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!