CS 33

Files Part 3

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.

Permissions Example

**adm group:
joe, angie**

```
$ ls -lR
.:
total 2
drwxr-x--x  2 joe    adm    1024 Dec 17 13:34 A
drwxr-----  2 joe    adm    1024 Dec 17 13:34 B

./A:
total 1
-rw-rw-rw-  1 joe    adm     593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-  1 joe    adm     446 Dec 17 13:34 x
-rw----rw-  1 angie  adm     446 Dec 17 13:45 y
```


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.

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.

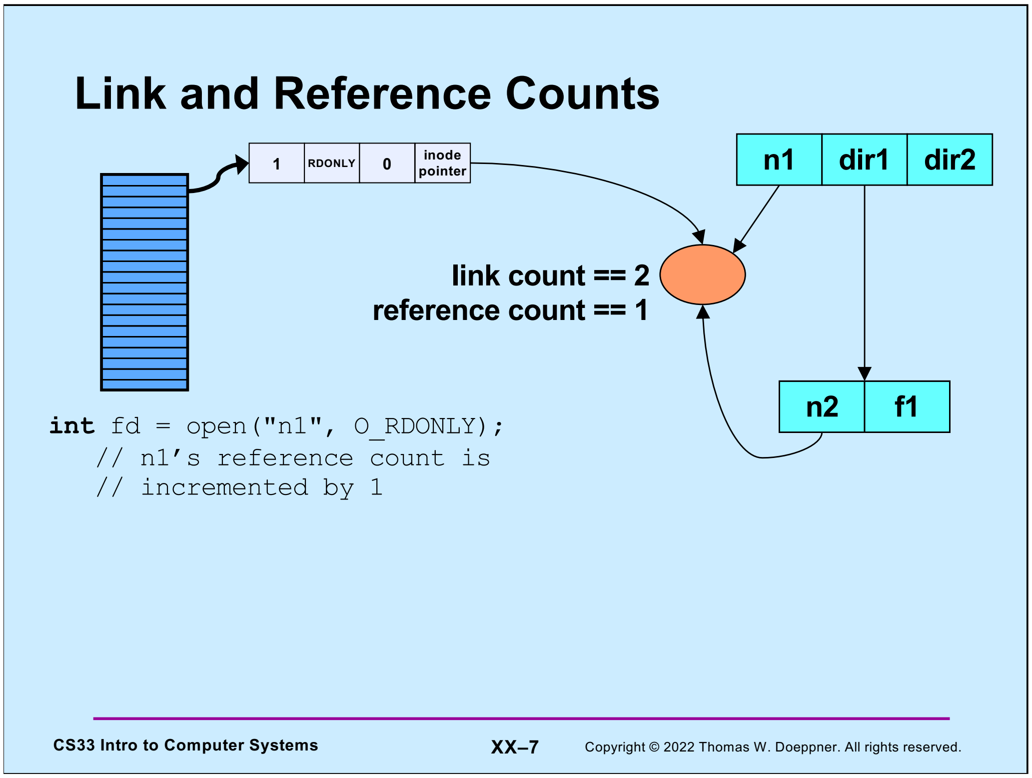
(Recall that numbers written with a leading 0 are in octal (base-8) notation.)

Creating a File

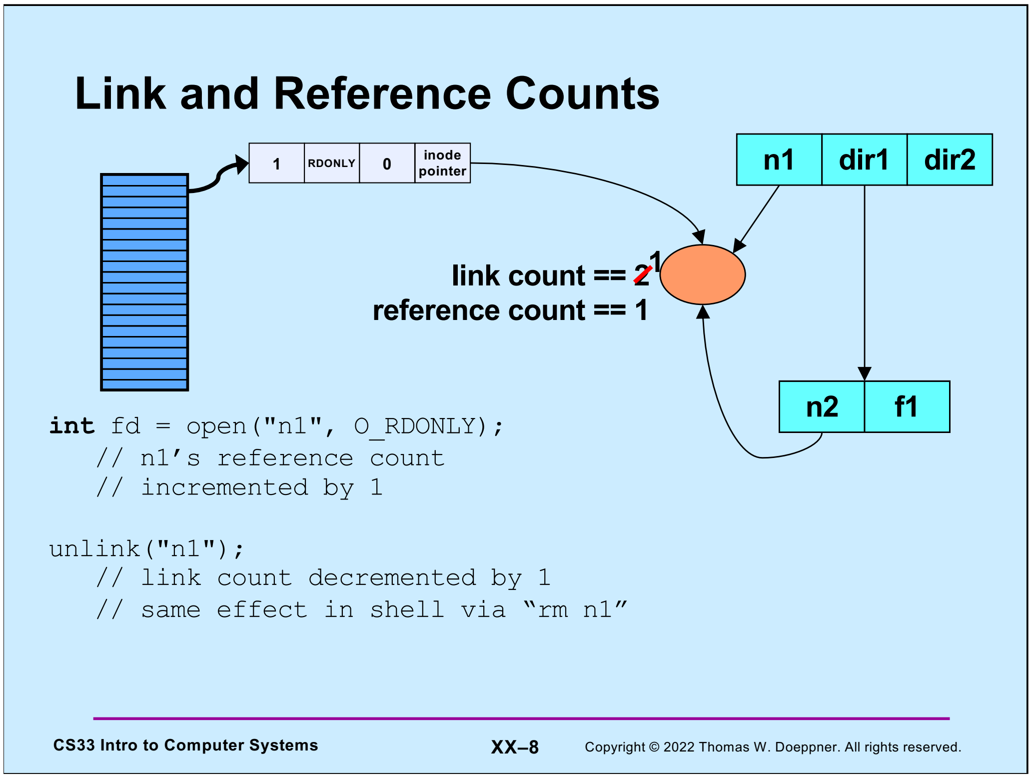
- **Use either *open* or *creat***
  - `open(const char *pathname, int flags, mode_t mode)`
    - » **flags must include O_CREAT**
  - `creat(const char *pathname, mode_t mode)`
    - » **open is preferred**
- **The *mode* parameter helps specify the permissions of the newly created file**
  - **– permissions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

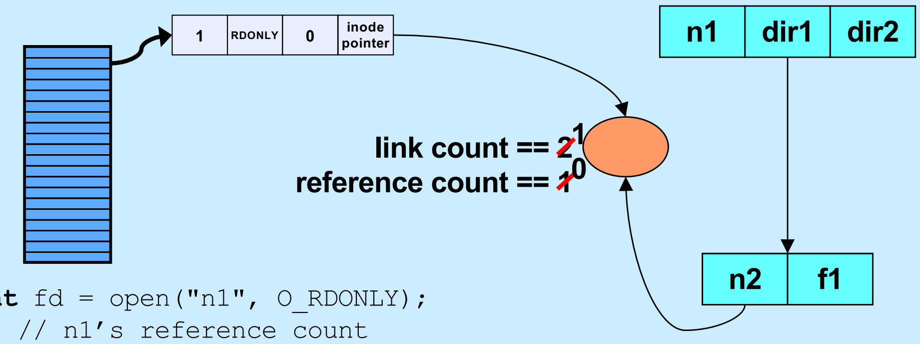


A file's link count is the number of directory entries that refer to it. There's a separate reference count that's the number of file context structures that refer to it (via the inode pointer – see slide XVII-9). These counts are maintained in the file's inode, which contains all information used by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

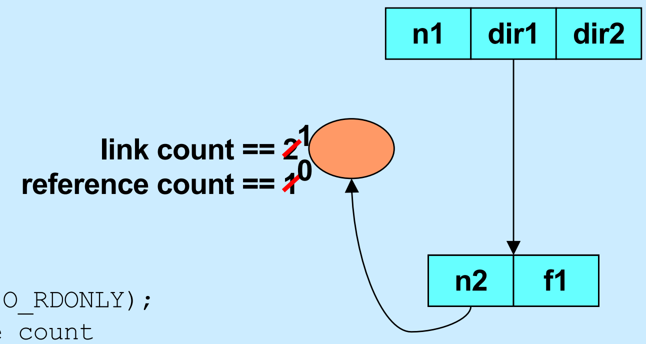



```
int fd = open("n1", O_RDONLY);
   // n1's reference count
   // incremented by 1

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

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

Link and Reference Counts



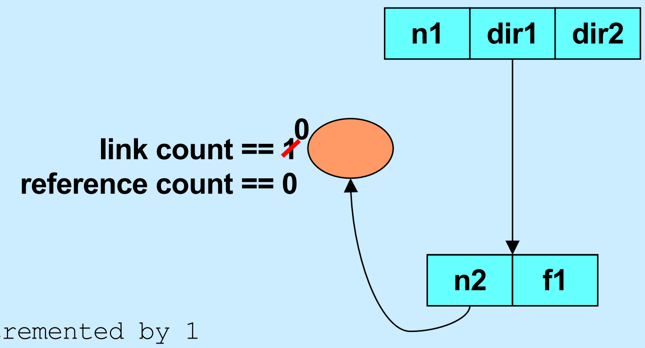

```
int fd = open("n1", O_RDONLY);
   // n1's reference count
   // incremented by 1

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 1

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

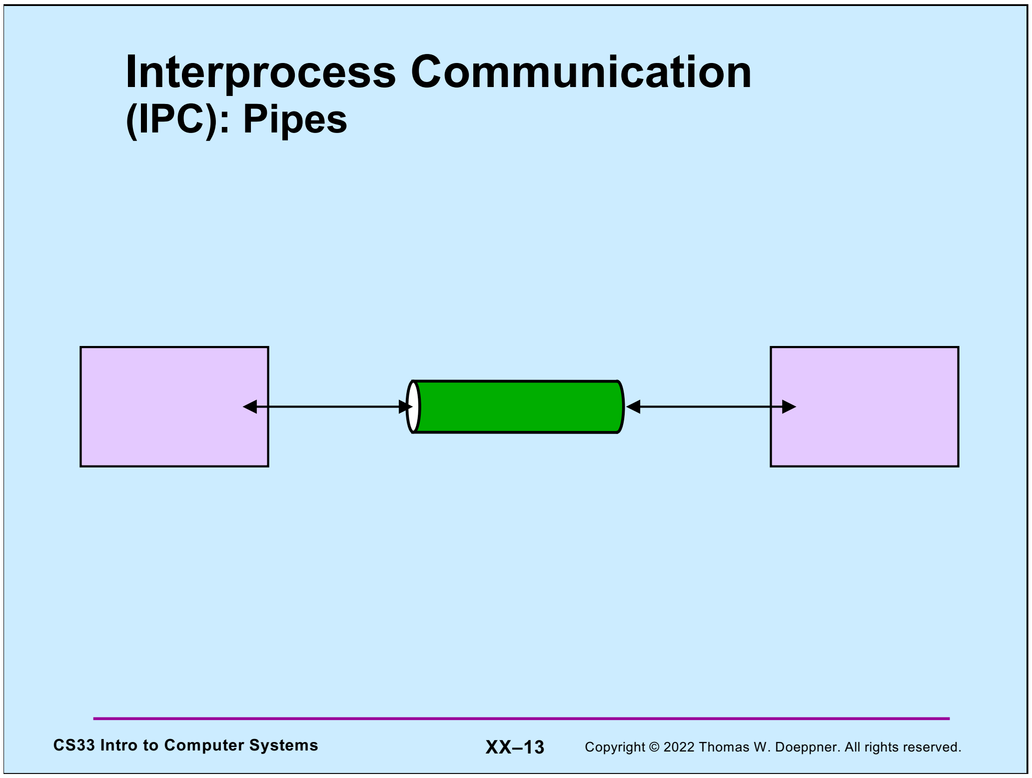
Assume that *PutStuffInFile* writes to the given file, and *GetStuffFromFile* reads from the file.

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

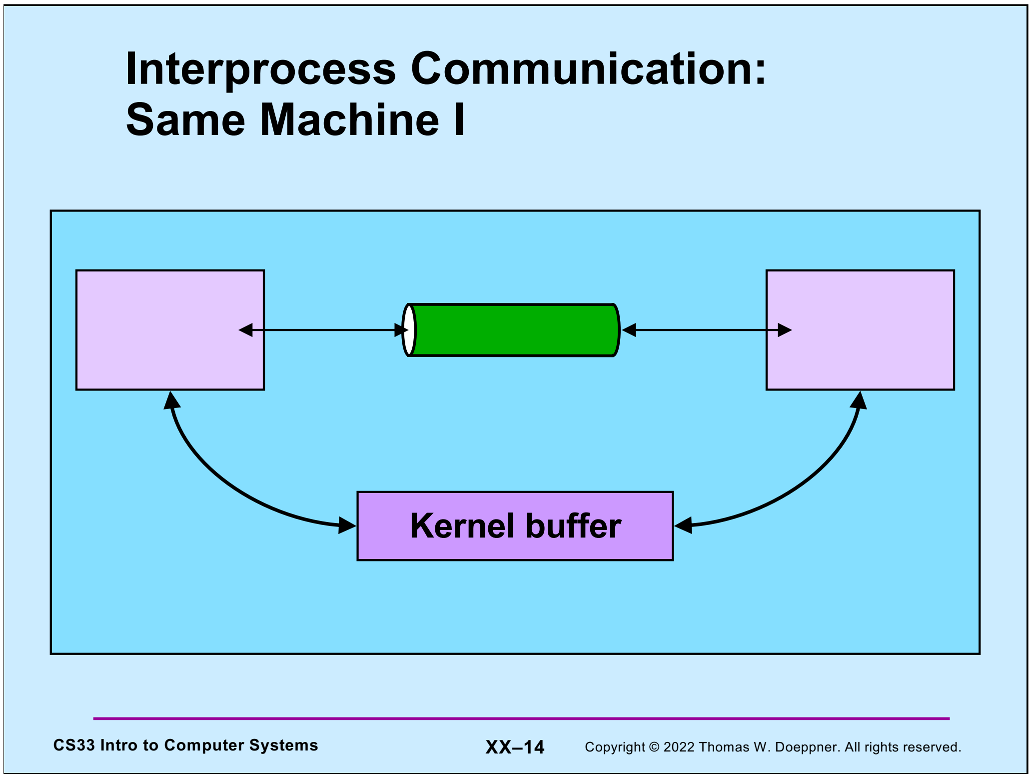
b) Because the file is used after the unlink call, it won't be deleted

c) The file will be deleted when the program terminates

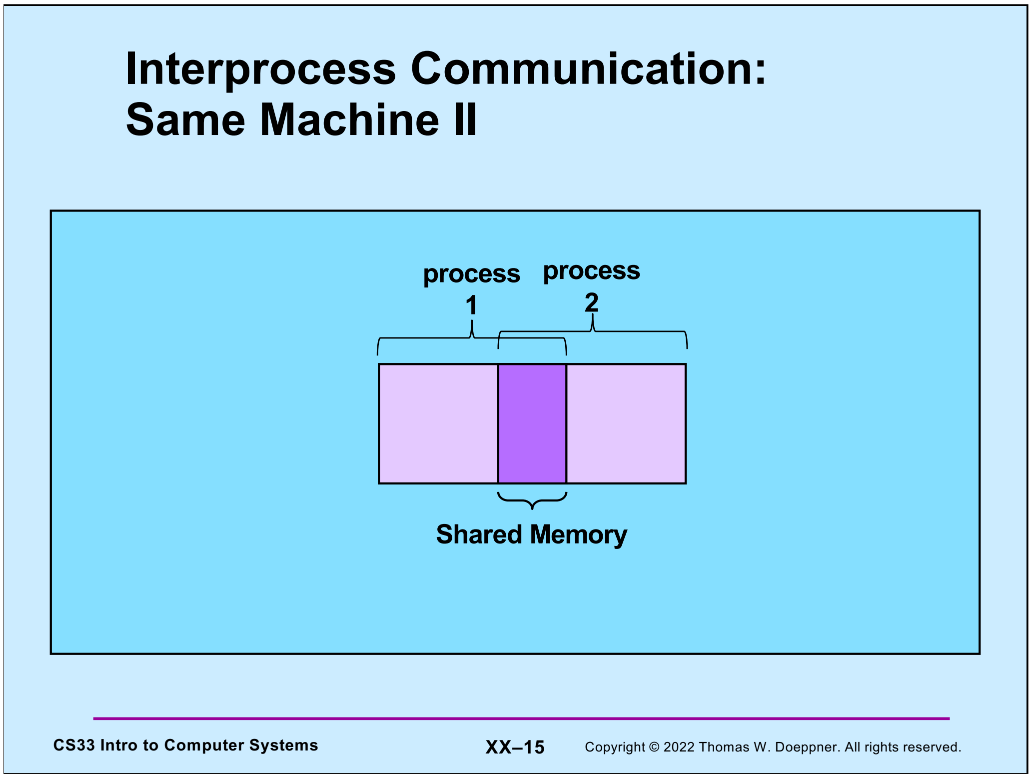
Note that when a process terminates, all its open files are automatically closed.



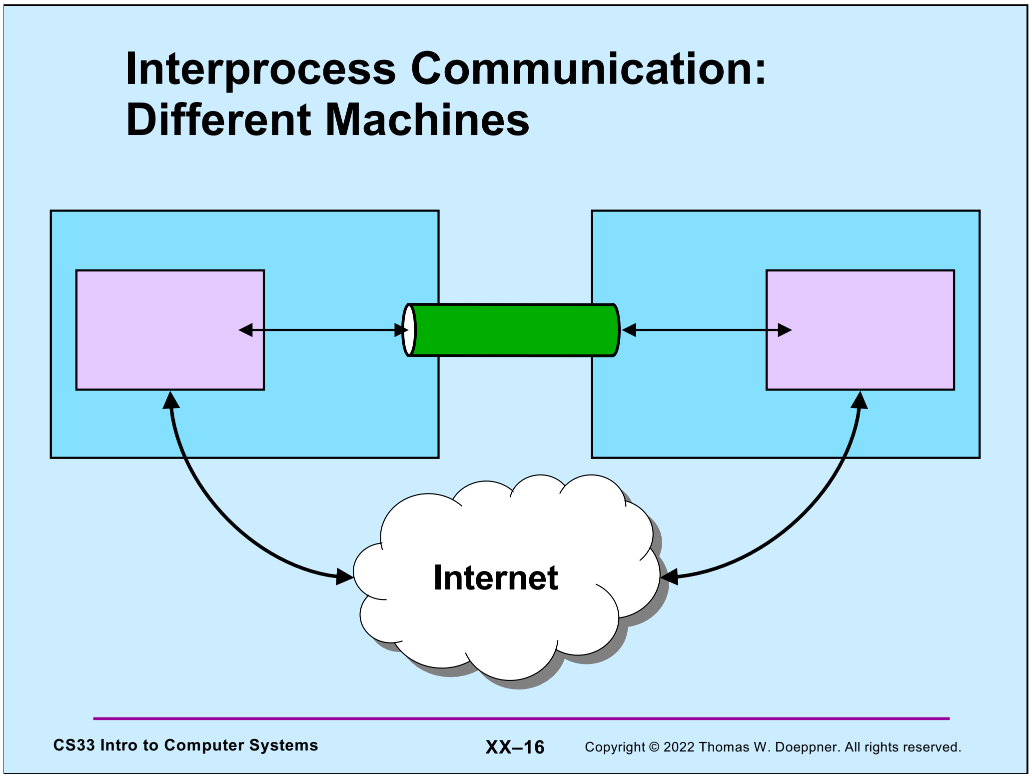
A rather elegant way for different processes to communicate is via a pipe: one process puts data into a pipe, another process reads the data from the pipe.



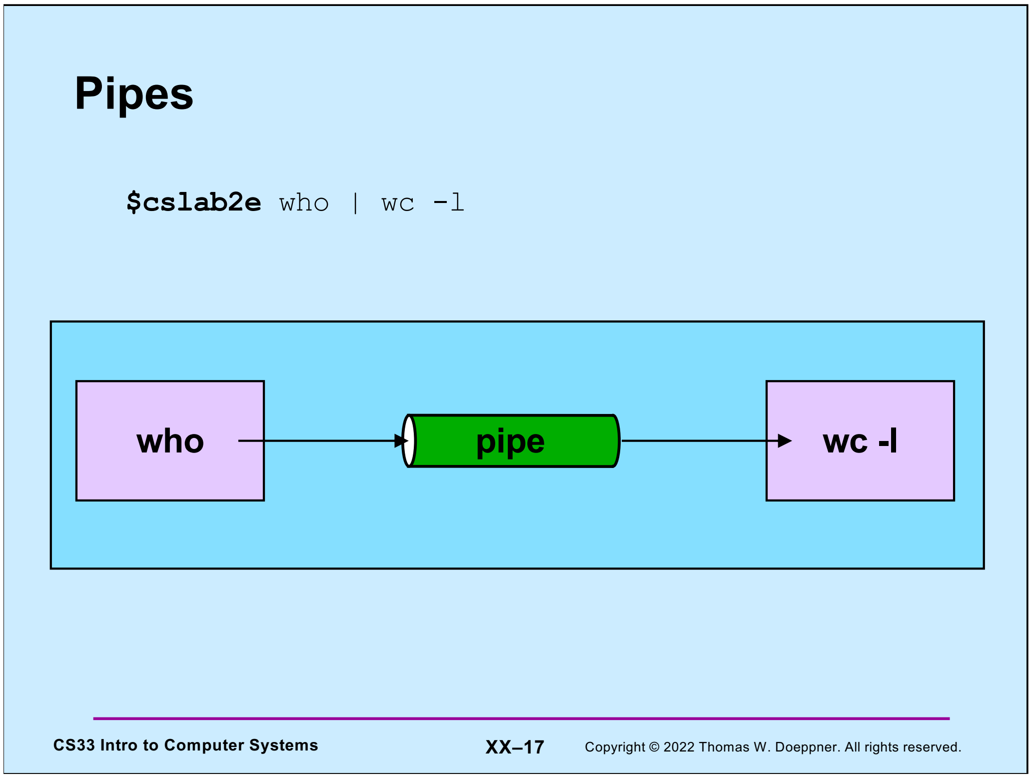
The implementation of a pipe involves the sending process using a write system call to transfer data into a kernel buffer. The receiving process fetches the data from the buffer via a read system call.



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.



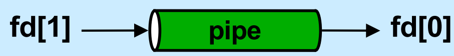
The pipe abstraction can also be made to work between processes on different machines. We discuss this later in the semester.



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.

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(...);      whoops!
            }
            // ...
            goto next_line;
        }
        // handle "normal" case
    }
    if (fork() == 0) {
        // ...           (whoops!)
        execv(...);
    }
    // ...
}
```


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!

CS 33

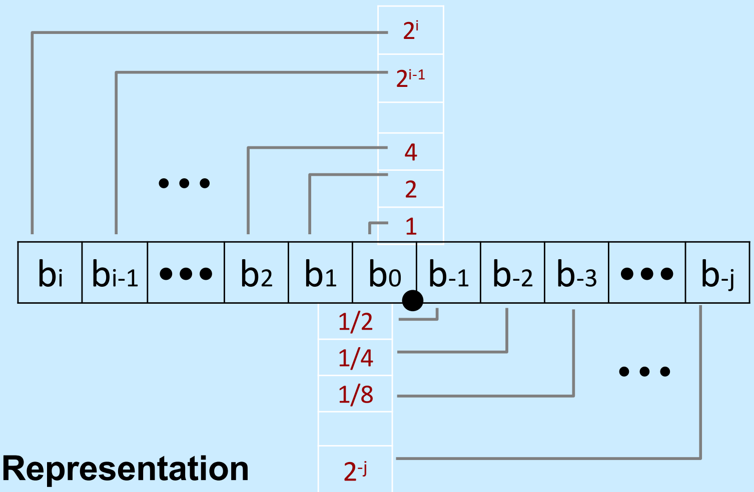
Data Representation (Part 3)

Fractional binary numbers

- What is 1011.101_2?

Supplied by CMU.

Fractional Binary Numbers



- **Representation**
  - **bits to right of "binary point" represent fractional powers of 2**
  - **represents rational number:** $\sum_{k=-j}^{i} b_k \times 2^k$

Supplied by CMU.

Representable Numbers

- **Limitation #1**
  - can exactly represent only numbers of the form $n/2^k$
    - » other rational numbers have repeating bit representations
  - value representation
    - » 1/3 `0.0101010101[01]…`$_2$
    - » 1/5 `0.001100110011[0011]…`$_2$
    - » 1/10 `0.0001100110011[0011]…`$_2$

- **Limitation #2**
  - just one setting of decimal point within the *w* bits
    - » limited range of numbers (very small values? very large?)

Supplied by CMU.

IEEE Floating Point

- IEEE Standard 754
  - established in 1985 as uniform standard for floating point arithmetic
    - before that, many idiosyncratic formats
  - supported on all major CPUs
- Driven by numerical concerns
  - nice standards for rounding, overflow, underflow
  - hard to make fast in hardware
    - numerical analysts predominated over hardware designers in defining standard

Supplied by CMU.

IEEE is the Institute for Electrical and Electronics Engineers (pronounced "eye triple e").

Floating-Point Representation

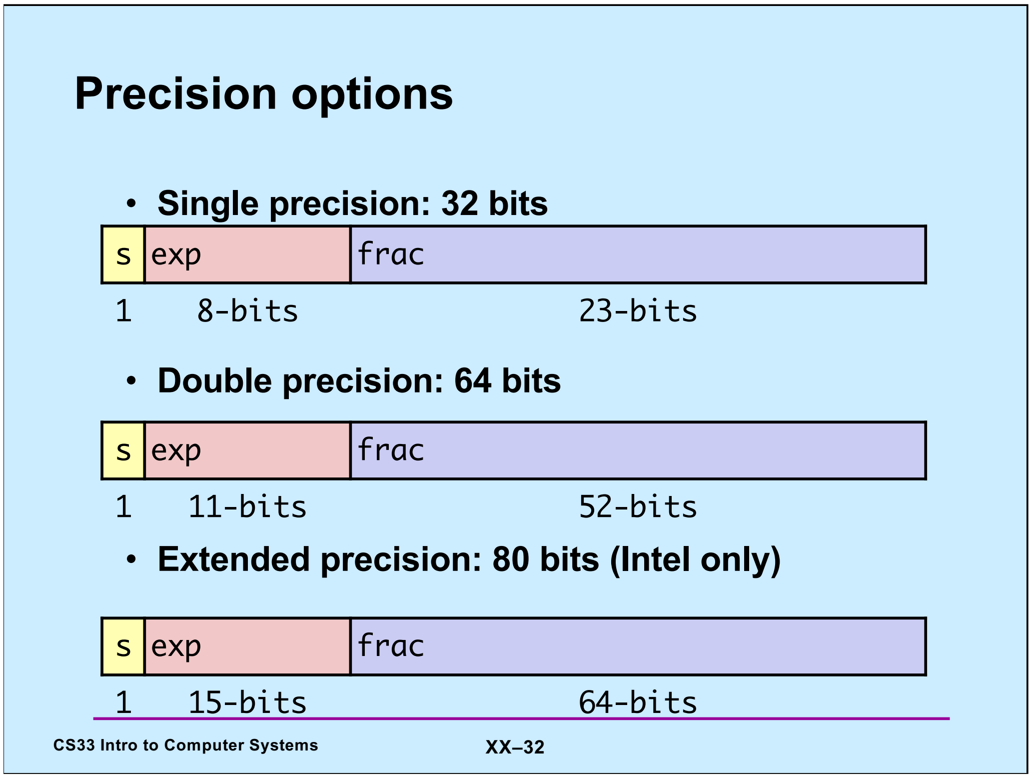
- **Numerical Form:**

$$(-1)^s\ M\ \ 2^E$$

  - sign bit s determines whether number is negative or positive
  - significand M normally a fractional value in range [1.0,2.0)
  - exponent E weights value by power of two
- **Encoding**
  - MSB `s` is sign bit s
  - `exp` field encodes E (but is not equal to E)
  - `frac` field encodes M (but is not equal to M)

s	exp	frac

Supplied by CMU.



Supplied by CMU.

On x86 hardware, all floating-point arithmetic is done with 80 bits, then reduced to either 32 or 64 as required.

“Normalized” Values

- When: exp ≠ 000…0 and exp ≠ 111…1
- Exponent coded as biased value: E = Exp – Bias
  - exp: unsigned value exp
  - bias = 2^{k-1} - 1, where k is number of exponent bits
    - » single precision: 127 (Exp: 1…254, E: -126…127)
    - » double precision: 1023 (Exp: 1…2046, E: -1022…1023)
- Significand coded with implied leading 1: M = $1.xxx…x_2$
  - xxx…x: bits of frac
  - minimum when frac=000…0 (M = 1.0)
  - maximum when frac=111…1 (M = 2.0 – ε)
  - get extra leading bit for “free”

Supplied by CMU.

Normalized Encoding Example

- **Value:** `float F = 15213.0;`
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$

- **Significand**
  - *M* = `1.`$\underline{1101101101101}_2$
  - `frac` = $\underline{1101101101101}0000000000_2$

- **Exponent**
  - *E* = **13**
  - *bias* = **127**
  - *exp* = **140** = 10001100_2

- **Result:**

s	exp	frac
0	10001100	11011011011010000000000

Supplied by CMU.

Denormalized Values

- Condition: exp = `000…0`
- Exponent value: E = –Bias + 1 (instead of E = 0 – Bias)
- Significand coded with implied leading 0:
  $M = 0.xxx\ldots x_2$
  - `xxx…x`: bits of `frac`, range [0,1)
- Cases
  - `exp` = `000…0`, `frac` = `000…0`
    - » represents zero value
    - » note distinct values: +0 and –0 (why?)
  - `exp` = `000…0`, `frac` ≠ `000…0`
    - » numbers closest to 0.0
    - » equispaced

Supplied by CMU.

For denormalized values, there's a single exponent value, which is 1- Bias. The significand is in a range of values greater than or equal to zero, but less than one.

Special Values

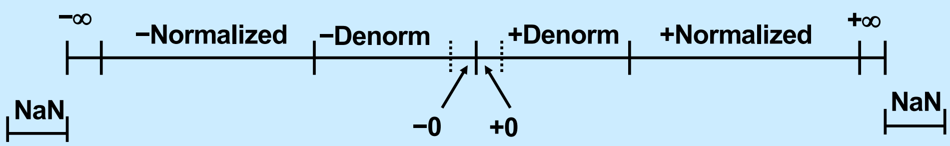
- **Condition: `exp` = 111…1**

- **Case: `exp` = 111…1, `frac` = 000…0**
  - represents value ∞ (infinity)
  - operation that overflows
  - both positive and negative
  - e.g., 1.0/0.0 = −1.0/−0.0 = +∞, 1.0/−0.0 = −∞

- **Case: `exp` = 111…1, `frac` ≠ 000…0**
  - not-a-number (NaN)
  - represents case when no numeric value can be determined
  - e.g., sqrt(–1), ∞ − ∞, ∞ × 0

Supplied by CMU.

Visualization: Floating-Point Encodings



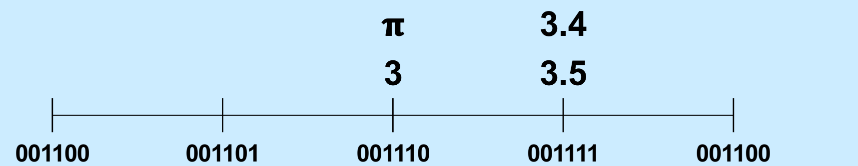
Supplied by CMU.

Mapping Real Numbers to Float

- The real number 3 is represented as
  0 011 10
- The real number 3.5 is represented as
  0 011 11
- How is the real number 3.4 represented?

  0 011 11
- How is the real number π represented?

  0 011 10



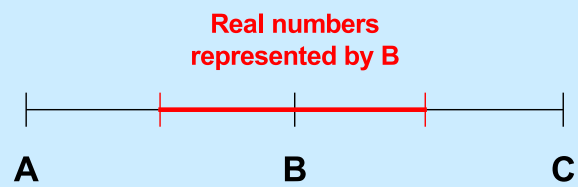
For the sake of this slide and example, assume that we have a six-bit representation of floating-point numbers. In this encoding there is one sign bit, 3 exponent bits (with a bias of 3) and 2 fraction bits. Thus 0 011 10 is 2^{3-3} * 1.5.

Mapping Real Numbers to Float

- If R is a real number, it's mapped to the floating-point number whose value is closest to R

Floats are Sets of Values

- If A, B, and C are successive floating-point values
  - e.g., 010001, 010010, and 010011
- B represents all real numbers from midway between A and B through midway between B and C



What about values that are equidistant from A and B or from B and C? There are rules for rounding such values that we don't have time to get into.

A special case is 0. Positive 0 represents a range of values that are greater than or equal to 0. Negative 0 represents a range of values that are less than or equal to zero.

+/− Zero

- Only one zero for ints
  - an int is a single number, not a range of numbers, thus there can be only zero
- Floating-point zero
  - a range of numbers around the real 0
  - it really matters which side of 0 we're on!
    - » a very large negative number divided by a very small negative number should be positive
      $-\infty/-0 = +\infty$
    - » a very large positive number divided by a very small negative number should be negative
      $+\infty/-0 = -\infty$

CS33 Intro to Computer Systems XX–41

It's important to remember that a floating-point value is not a single number, but a range of numbers.

Significance

- Normalized numbers
  - for a particular exponent value E and an S-bit significand, the range from 2^E up to 2^{E+1} is divided into 2^S equi-spaced floating-point values
    - » thus each floating-point value represents $1/2^S$ of the range of values with that exponent
    - » all bits of the signifcand are important
    - » we say that there are S significant bits – for reasonably large S, each floating-point value covers a rather small part of the range
      - high accuracy
      - for S=23 (32-bit float), accurate to one in 2^{23} (.0000119% accuracy)

Significance

- **Unnormalized numbers**
  - high-order zero bits of the significand aren't important
  - in 32-bit floating point, 0 00000000 00000000000000000000001 represents 2^{-149}
    - » it is the only value with that exponent: 1 significant bit (either 2^{-149} or 0)
  - 0 00000000 00000000000000000000010 represents 2^{-148}
    0 00000000 00000000000000000000011 represents $1.5*2^{-148}$
    - » only two values with exponent -148: 2 significant bits (encoding those two values, as well as 2^{-149} and 0)
  - fewer significant bits mean less accuracy
  - 0 00000000 00000000000000000000001 represents a range of values from $.5*2^{-9}$ to $1.5*2^{-9}$
  - 50% accuracy

Recall that the bias for the exponent of 8-bit IEEE FP is 7, thus for unnormalized numbers the actual exponent is -6 (-bias+1). The significand has an implied leading 0, thus 0 0000 001 represents $2^{-6} * 2^{-3}$.

With 8-bit IEEE FP. the value 0 0000 01 is interpreted as 2-9, But the number represented could be 50% or 50% more.

Floating Point

- **Single precision (float)**

s	exp	frac
1	8-bits	23-bits

  - **range: $\pm 1.8\times10^{-38}$ – $\pm 3.4\times10^{38}$, ~7 decimal digits**

- **Double Precision (double)**

s	exp	frac
1	11-bits	52-bits

  - **range: $\pm 2.23\times10^{-308}$ – $\pm 1.8\times10^{308}$, ~16 decimal digits**

Quiz 2

Suppose *f*, declared to be a `float`, is assigned the largest possible floating-point positive value (other than +∞). What is the value of *g = f+1.0*?

a) 0
b) f
c) +∞
d) NaN

Float is not Rational …

- Floating addition
  - commutative: $a +_f b = b +_f a$
    - » yes!
  - associative: $a +_f (b +_f c) = (a +_f b) +_f c$
    - » no!
      - $2 +_f (1e38 +_f -1e38) = 2$
      - $(2 +_f 1e38) +_f -1e38 = 0$

Note that the floating-point numbers in this and the next two slides are expressed in base 10, not base 2.

In this and the next few slides, $+_f$ means floating-point addition (as opposed to addition of real numbers) and $*_f$ means floating-point multiplication.

Float is not Rational …

- **Multiplication**
  - commutative: $a *_f b = b *_f a$
    - » yes!
  - associative: $a *_f (b *_f c) = (a *_f b) *_f c$
    - » no!
      - $1e37 *_f (1e37 *_f 1e\text{-}37) = 1e37$
      - $(1e37 *_f 1e37) *_f 1e\text{-}37 = +\infty$

Float is not Rational …

- More …
  - multiplication distributes over addition:
    $a *_f (b +_f c) = (a *_f b) +_f (a *_f c)$
    - » no!
    - » $1e38 *_f (1e38 +_f -1e38) = 0$
    - » $(1e38 *_f 1e38) +_f (1e38 *_f -1e38) = NaN$
  - insignificance:
    $x = y +_f 1$
    $z = 2 /_f (x -_f y)$
    $z == 2?$
    - » not necessarily!
      - consider y = 1e38

If y is 1e38 and we're using single-precision floating-point arithmetic, then z would be $+\infty$ (since $x -_f y$ would be 0).