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

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
```


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)`

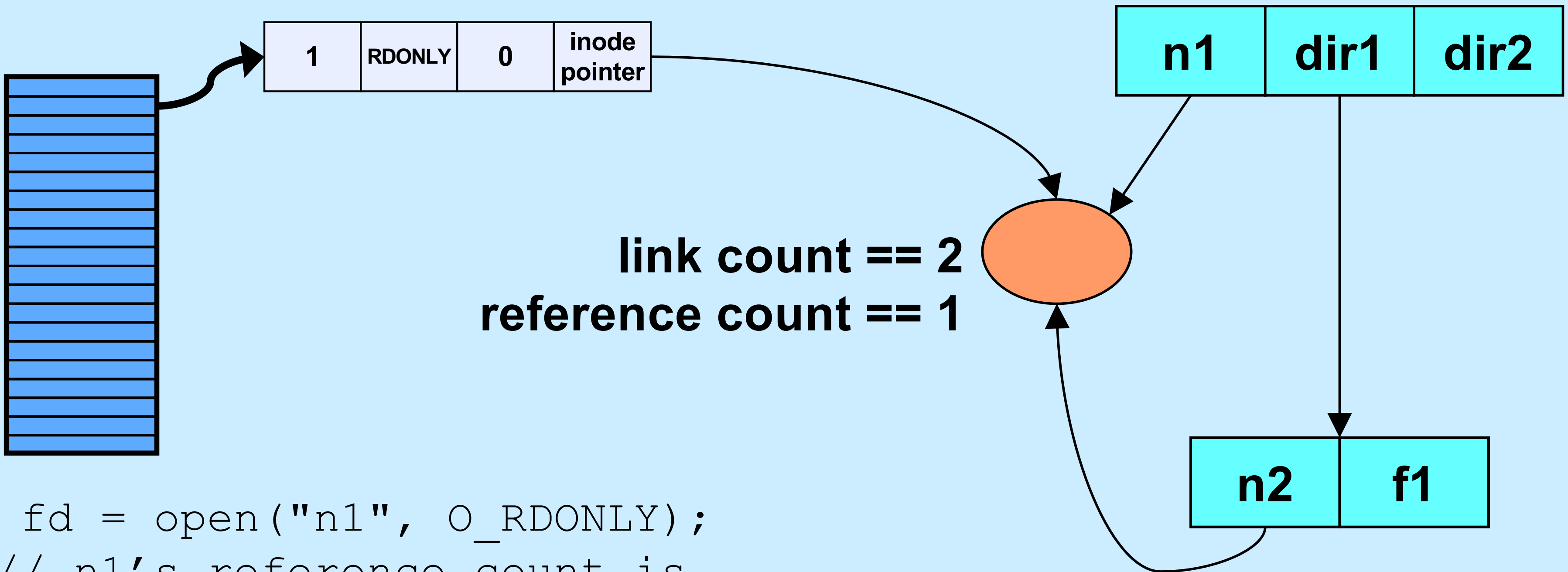
Umask

- **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

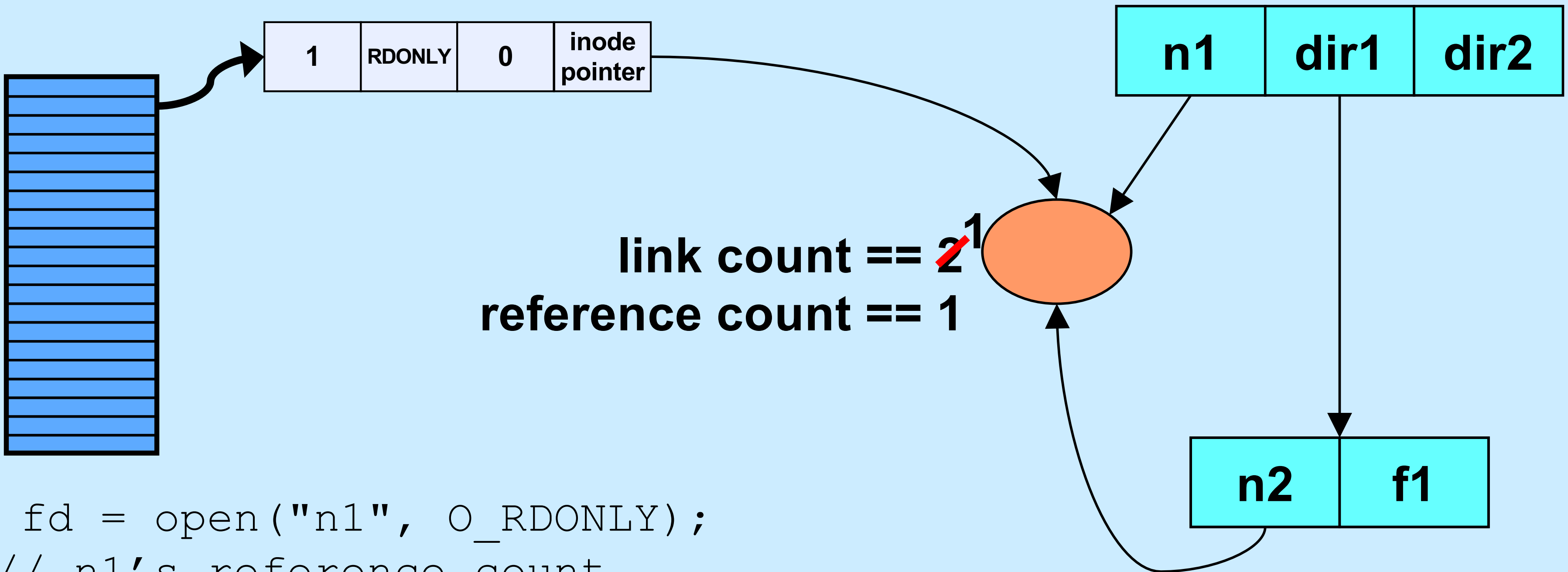
- **Use either *open* or *creat***
  - `open(const char *pathname, int flags, mode_t mode)`
    - » **flags must include O_CREAT**
  - `creat(const char *pathname, mode_t mode)`
    - » **open is preferred**
- **The *mode* parameter helps specify the permissions of the newly created file**
  - **permissions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

Link and Reference Counts




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

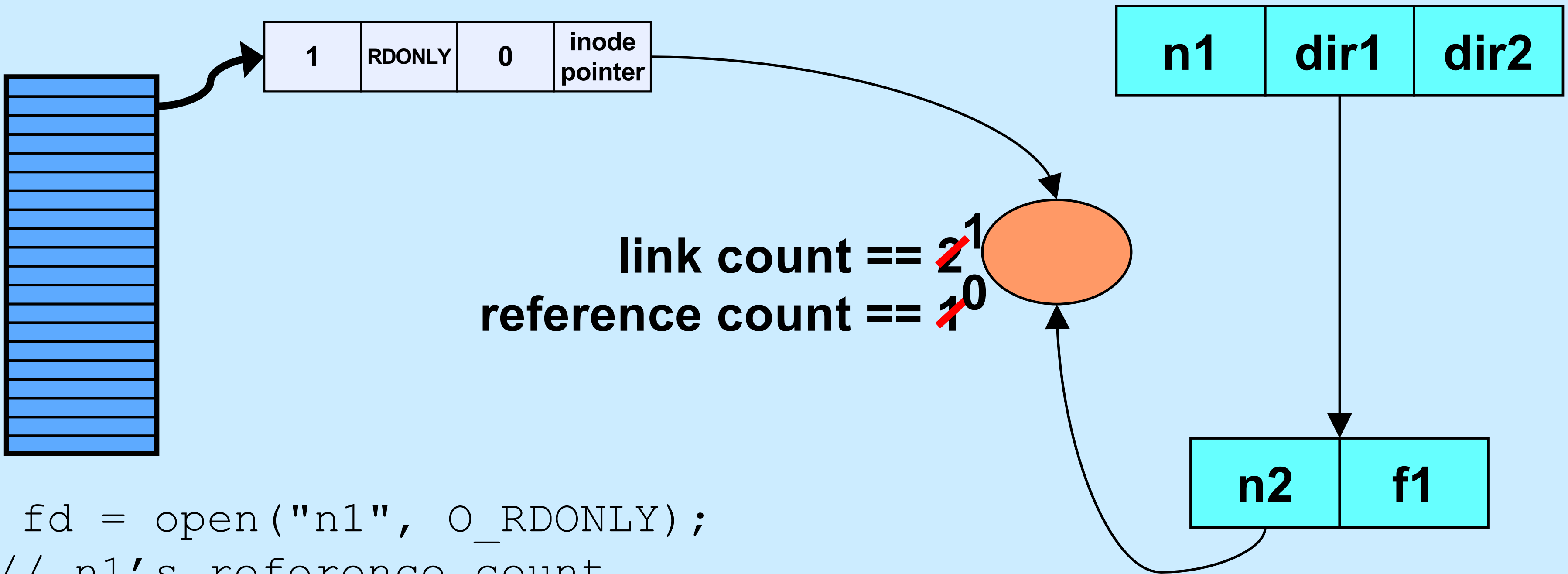
Link and Reference Counts




```
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

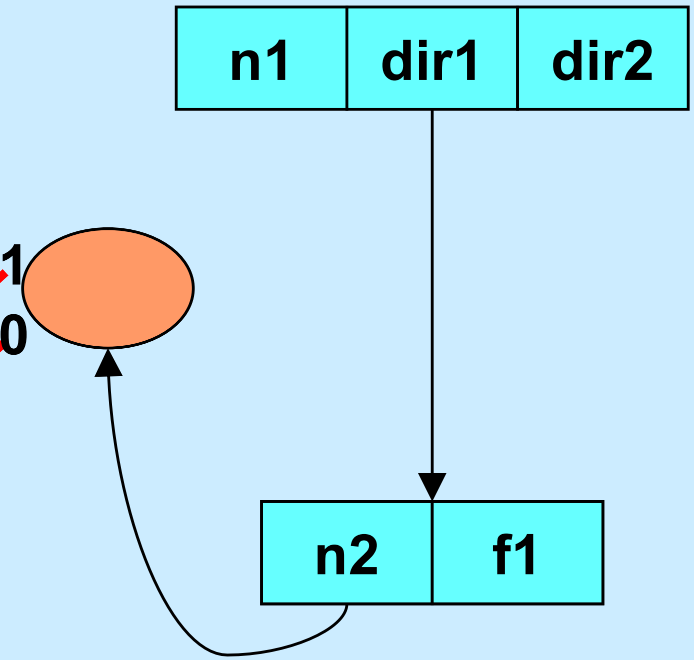



```
int fd = open("n1", O_RDONLY);
   // n1's reference count
   // incremented by 1

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

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

Link and Reference Counts



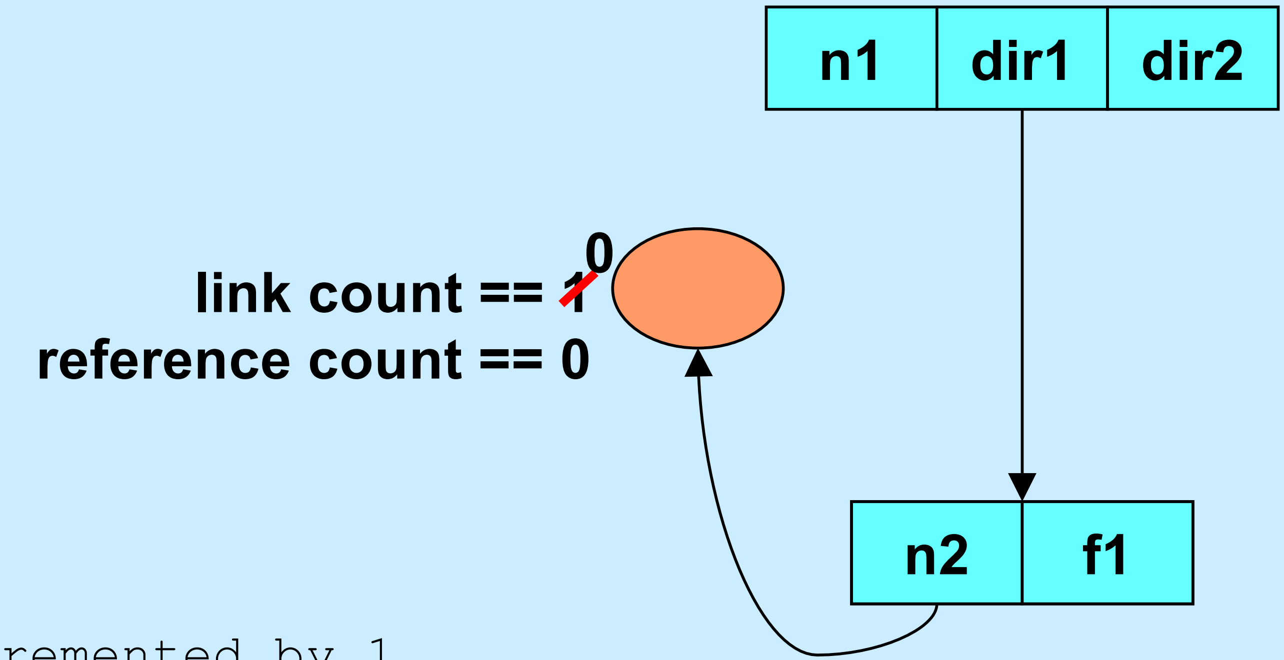
link count == ~~2~~ 1
reference count == ~~1~~ 0

```
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;
}
```

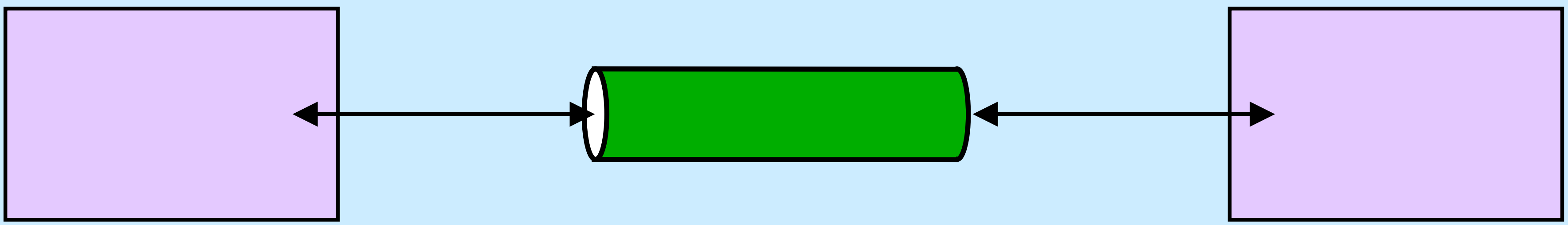
Assume that *PutStuffInFile* writes to the given file, and *GetStuffFromFile* reads from the file.

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

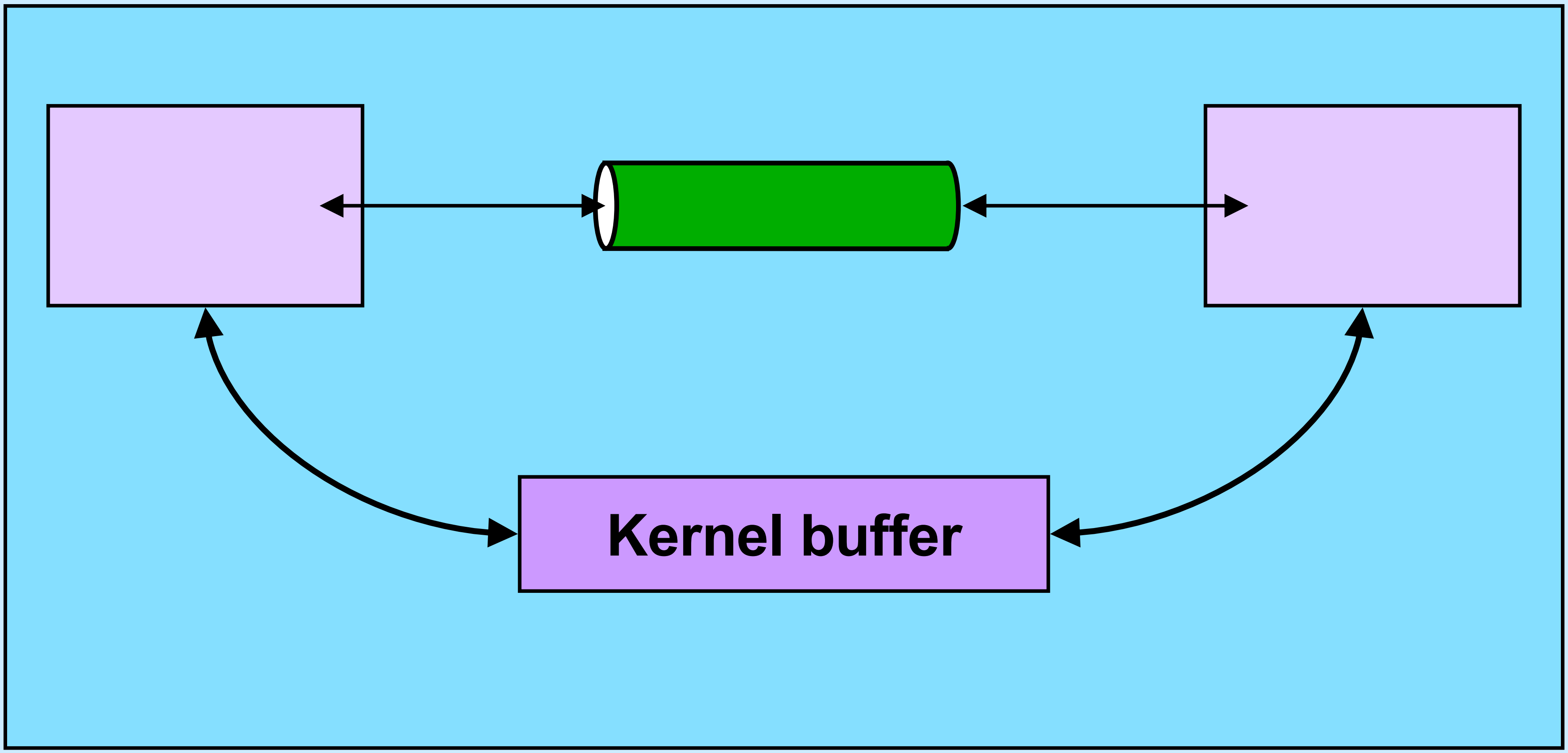
b) Because the file is used after the unlink call, it won't be deleted

c) The file will be deleted when the program terminates

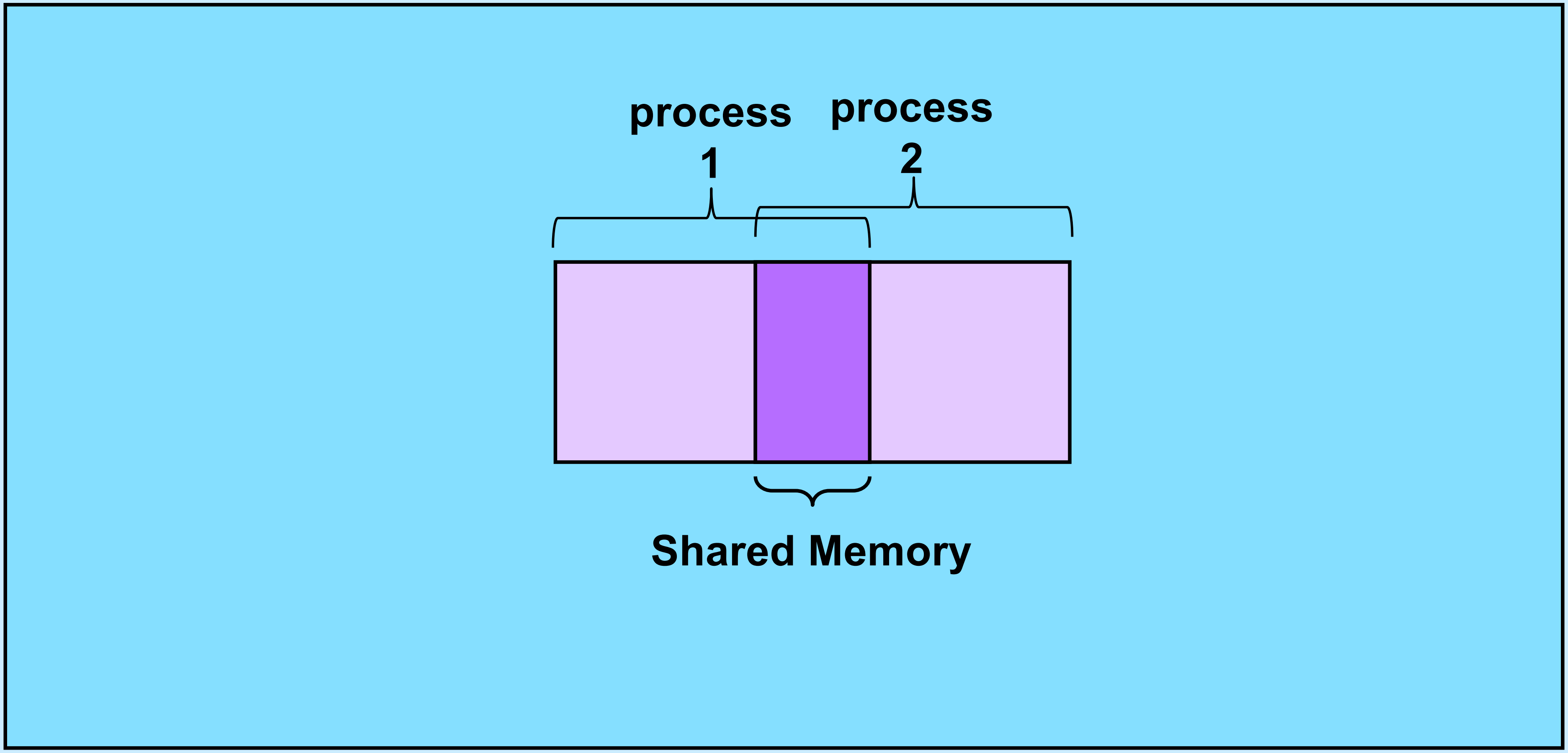
Interprocess Communication (IPC): Pipes



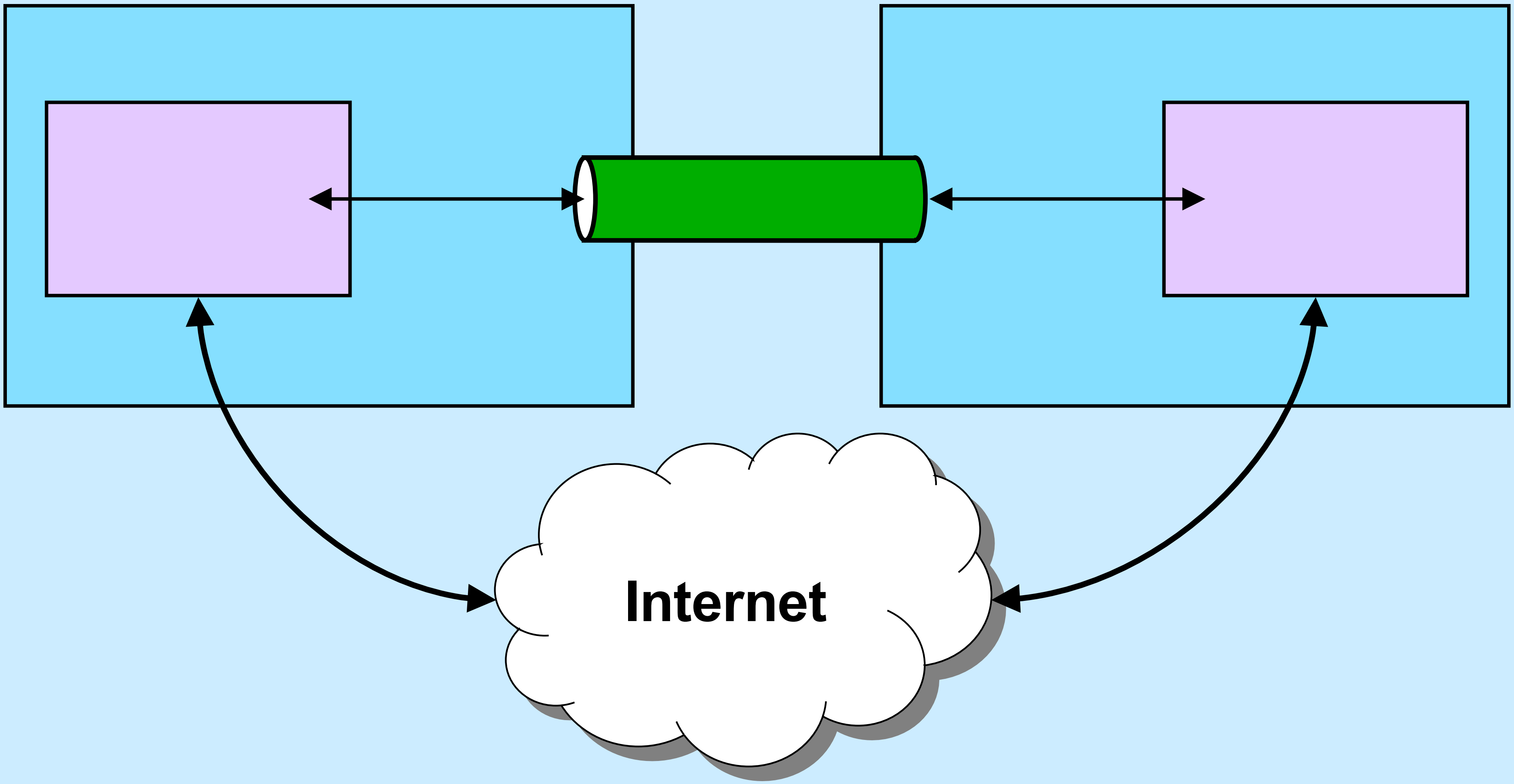
Interprocess Communication: Same Machine I



Interprocess Communication: Same Machine II

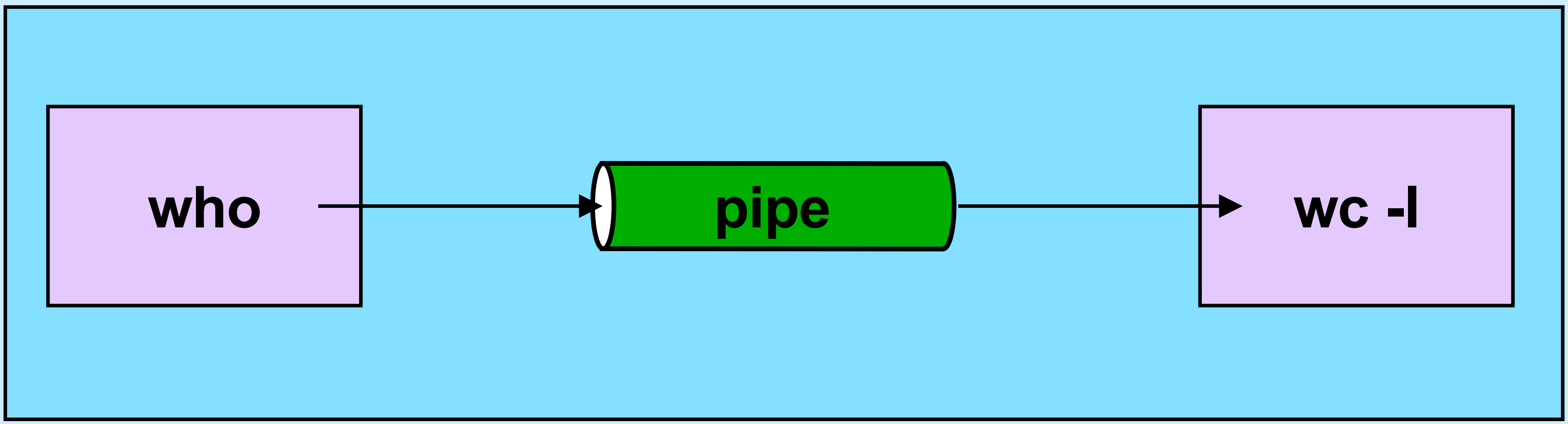


Interprocess Communication: Different Machines



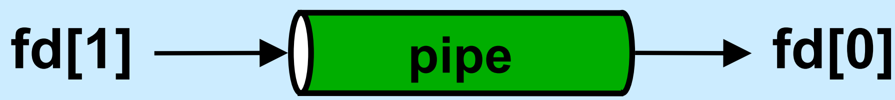
Pipes

```
$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]);
// …
```

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(...);
```

whoops!

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

(whoops!)

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

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!

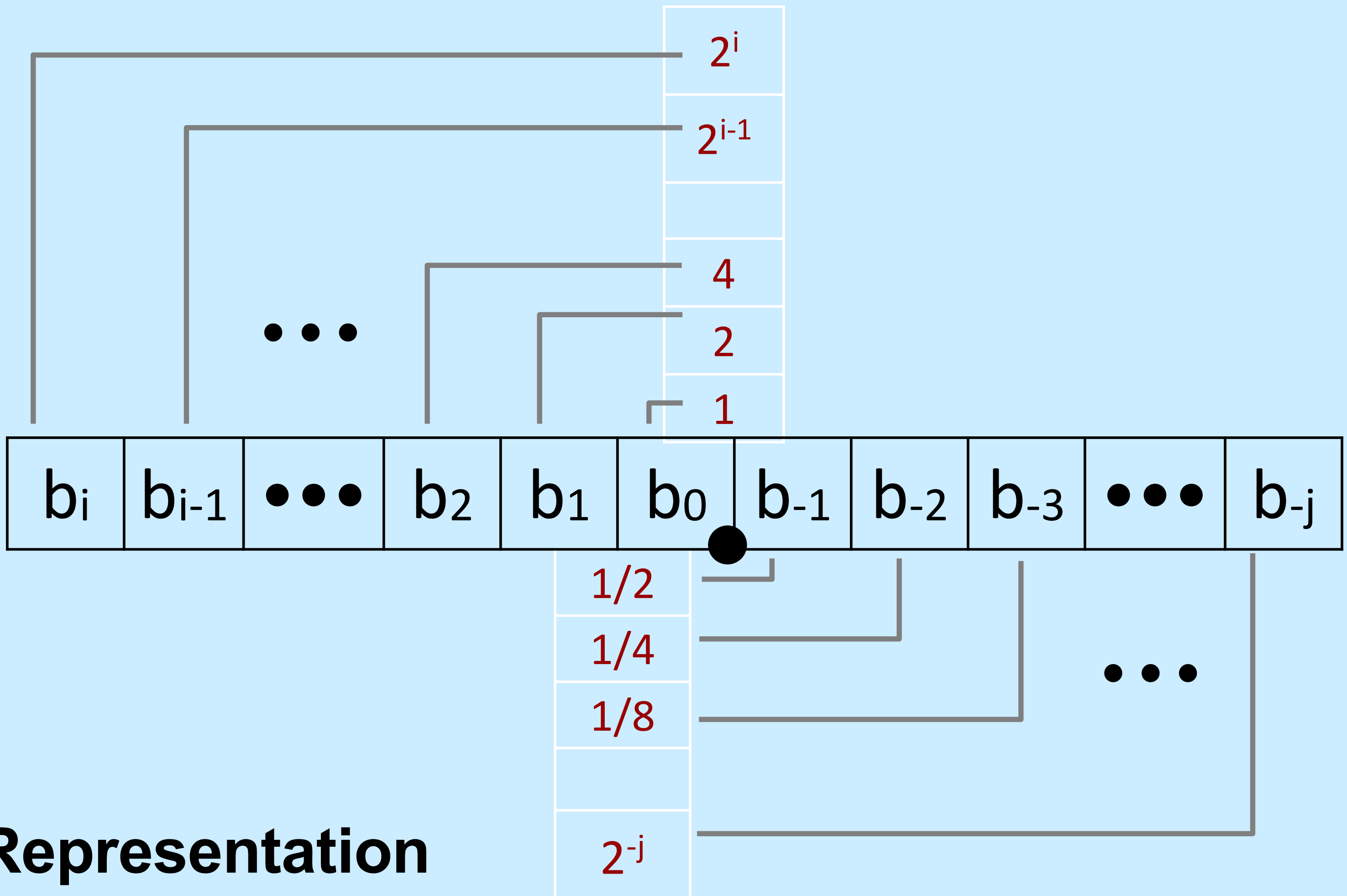
CS 33

Data Representation (Part 3)

Fractional binary numbers

- What is 1011.101_2?

Fractional Binary Numbers



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

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 $0.0101010101[01]\ldots_2$
    - » 1/5 $0.001100110011[0011]\ldots_2$
    - » 1/10 $0.0001100110011[0011]\ldots_2$

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

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

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

Precision options

- **Single precision: 32 bits**

s	exp	frac
1	8-bits	23-bits

- **Double precision: 64 bits**

s	exp	frac
1	11-bits	52-bits

- **Extended precision: 80 bits (Intel only)**

s	exp	frac
1	15-bits	64-bits

“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”

Normalized Encoding Example

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

- **Significand**

  M **=** `1.`$\underline{\texttt{1101101101101}}_2$

  `frac` **=** $\underline{\texttt{1101101101101}}\texttt{0000000000}_2$

- **Exponent**

  E **= 13**

  bias **= 127**

  exp **= 140 =** `10001100`$_2$

- **Result:**

s	exp	frac
`0`	`10001100`	`11011011011010000000000`

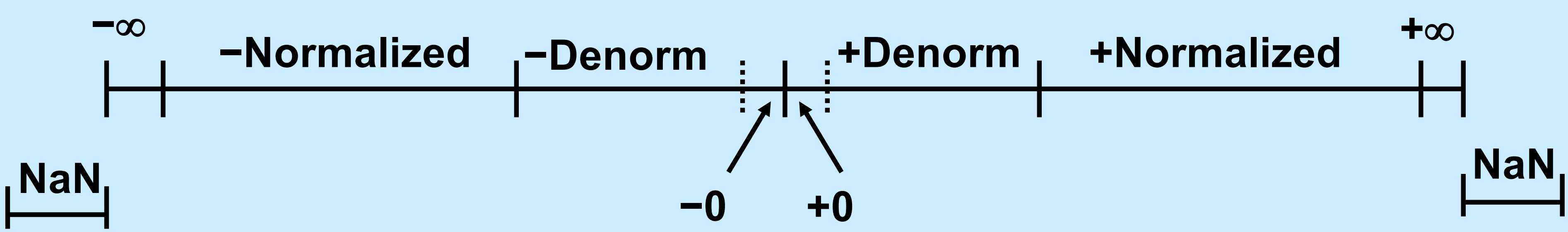
Denormalized Values

- 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

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

Visualization: Floating-Point Encodings

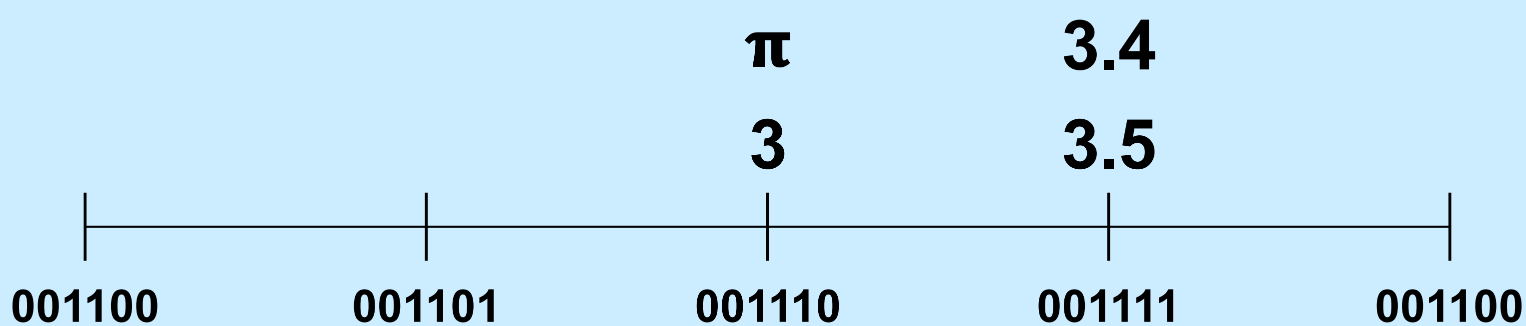


Mapping Real Numbers to Float

- 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

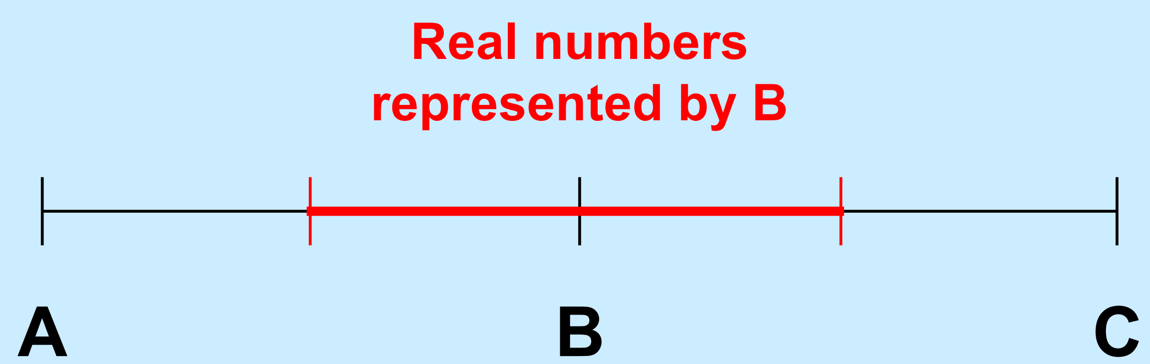


Mapping Real Numbers to Float

- 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



+/− 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

      $+\infty\ /-0 = -\infty$

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

Floating Point

- **Single precision (float)**

s	exp	frac
1	8-bits	23-bits

  – **range: $\pm 1.8\times 10^{-38}$ – $\pm 3.4\times 10^{38}$, ~7 decimal digits**

- **Double Precision (double)**

s	exp	frac
1	11-bits	52-bits

  – **range: $\pm 2.23\times 10^{-308}$ – $\pm 1.8\times 10^{308}$, ~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!**
      - $2 +_f (1e38 +_f -1e38) = 2$
      - $(2 +_f 1e38) +_f -1e38 = 0$

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