CS 33

Files Part 4

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**

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?
```

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
```

In the Shell ...

```
% program >> file
```

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)`

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

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

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) execute the shell command
   `chmod 0644 program`

b) execute the shell command
   `chmod 0755 program`

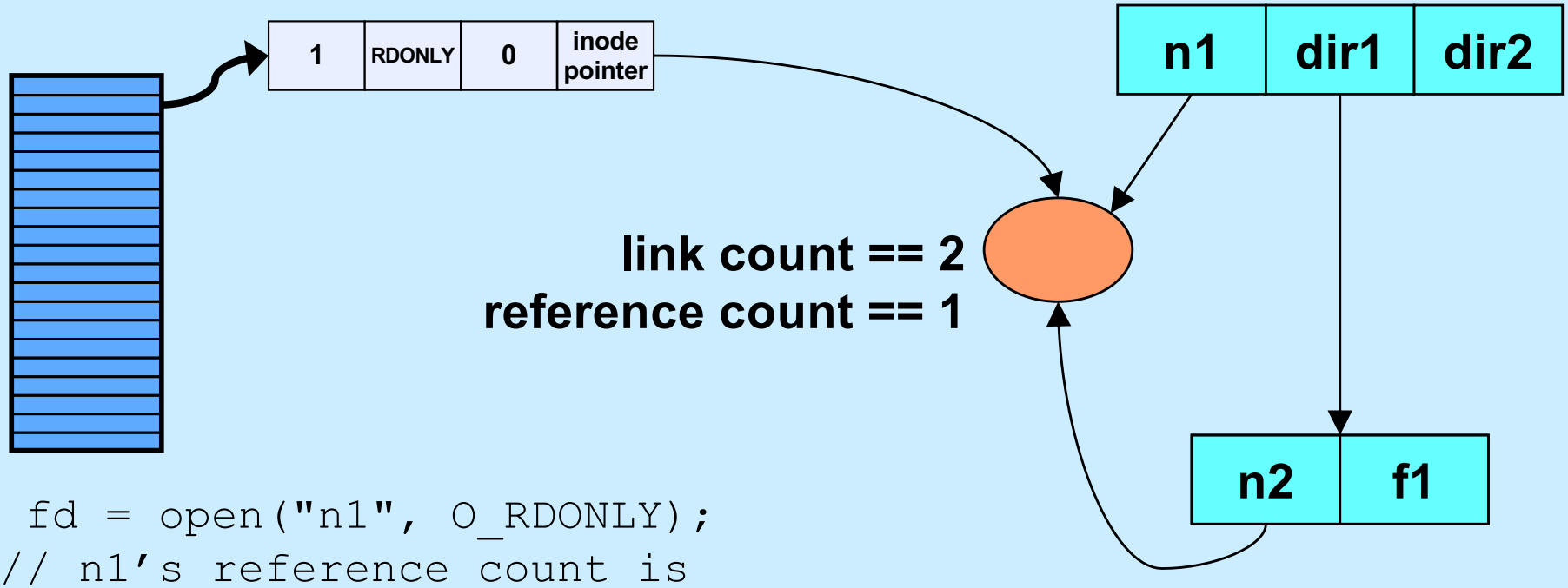
c) find the source code for program and recompile it

d) make an Ed post

Creating a File

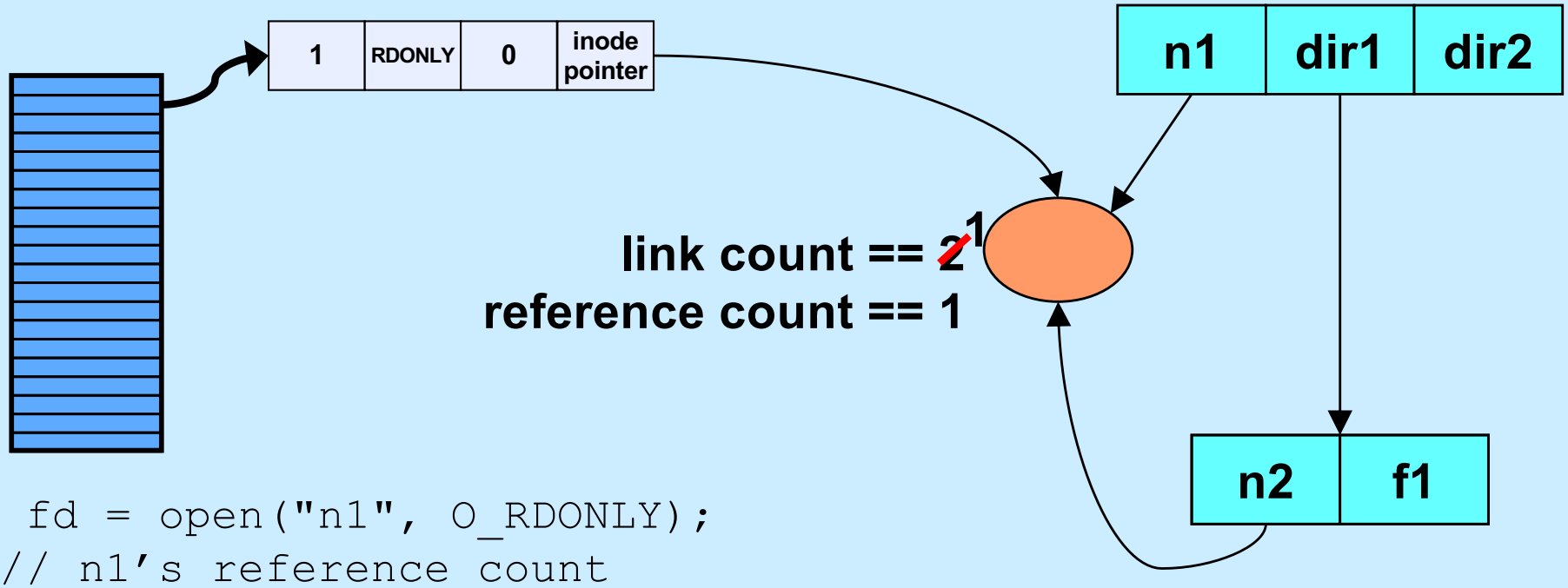
- **Use either *open* or *creat***
  - `open(const char *pathname, int flags, mode_t mode)`
    - » **flags must include O_CREAT**
  - `creat(const char *pathname, mode_t mode)`
    - » **open is preferred**
- **The *mode* parameter helps specify the permissions of the newly created file**
  - **permissions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

Link and Reference Counts




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

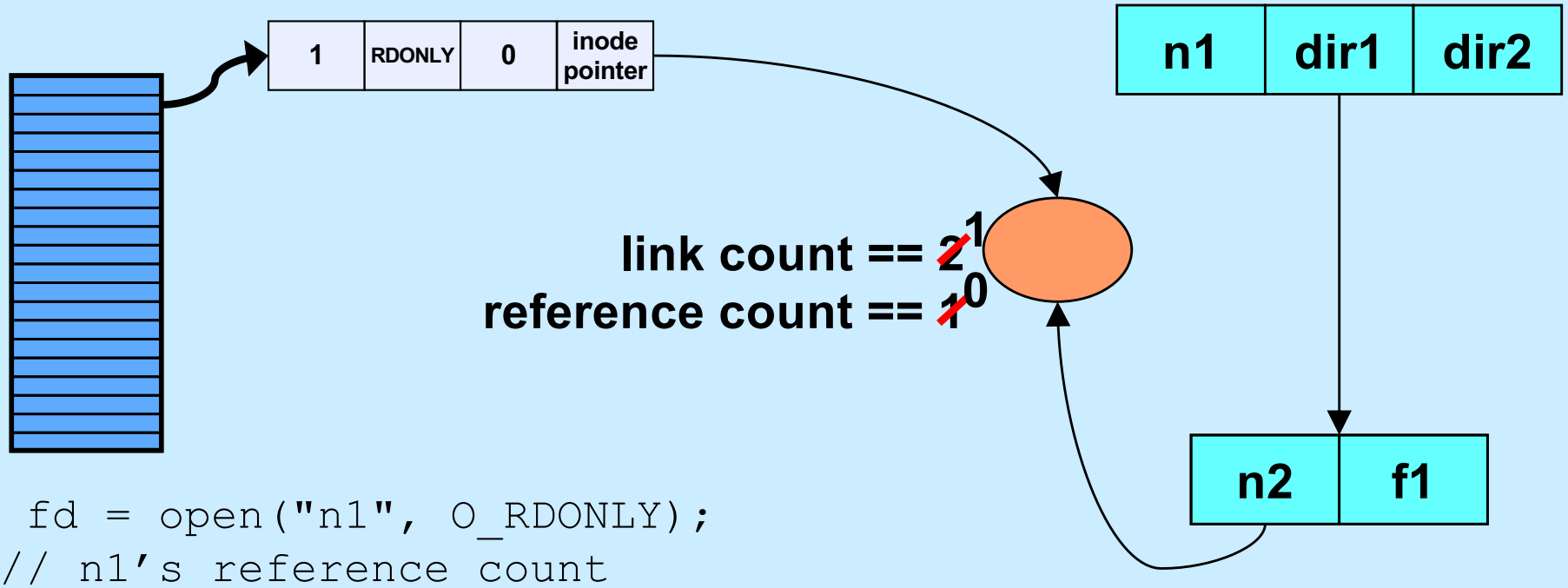
Link and Reference Counts




```
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

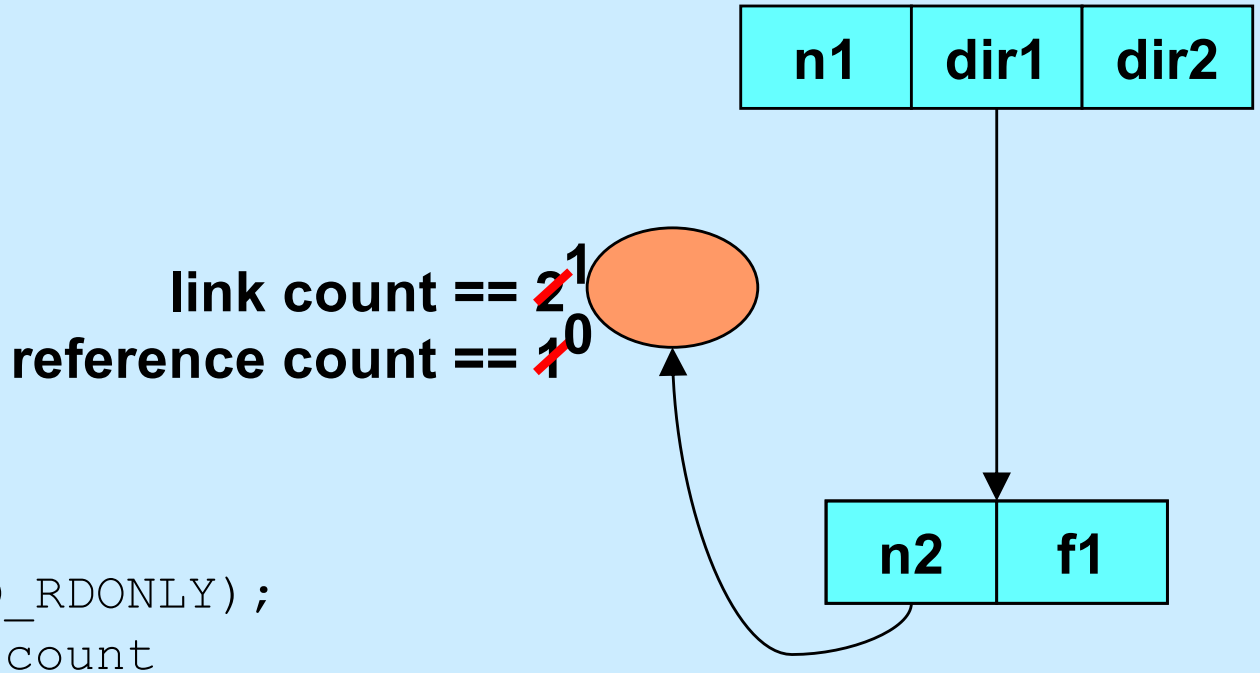



```
int fd = open("n1", O_RDONLY);
   // n1's reference count
   // incremented by 1

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

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

Link and Reference Counts

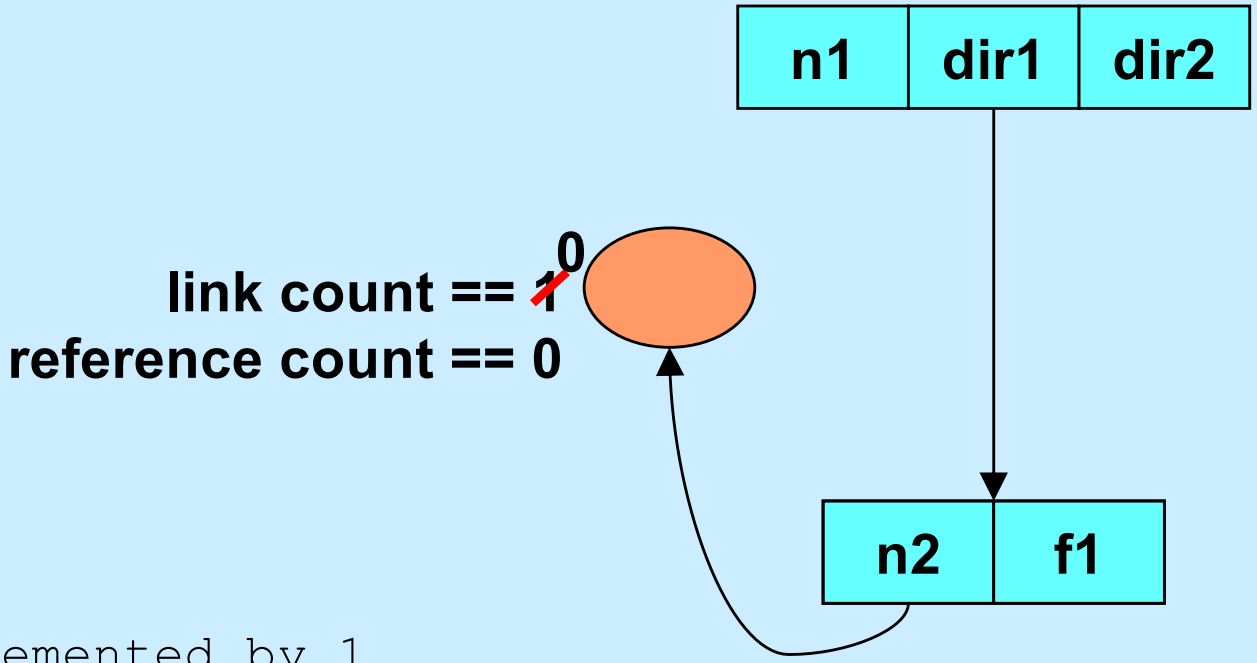



```
int fd = open("n1", O_RDONLY);
   // n1's reference count
   // incremented by 1

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
```

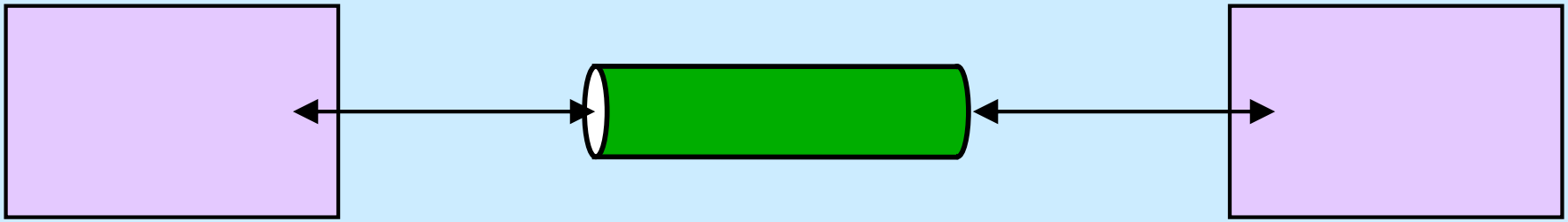
Quiz 2

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

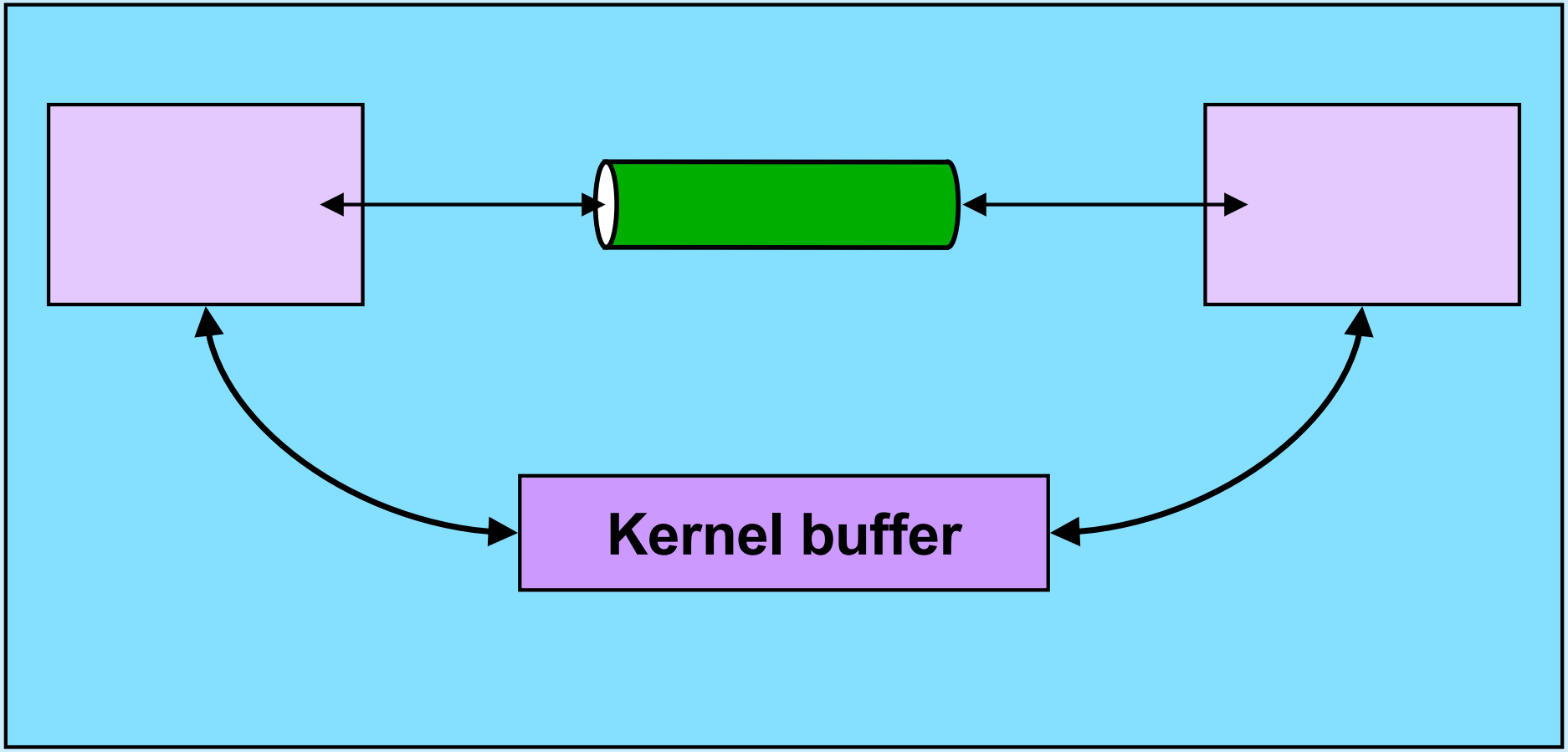
Assume that *PutStuffInFile* writes to the given file, and *GetStuffFromFile* reads from the file.

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

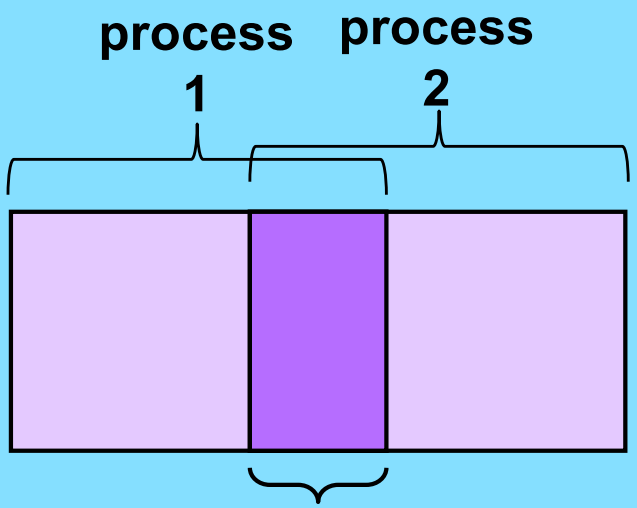
Interprocess Communication (IPC): Pipes



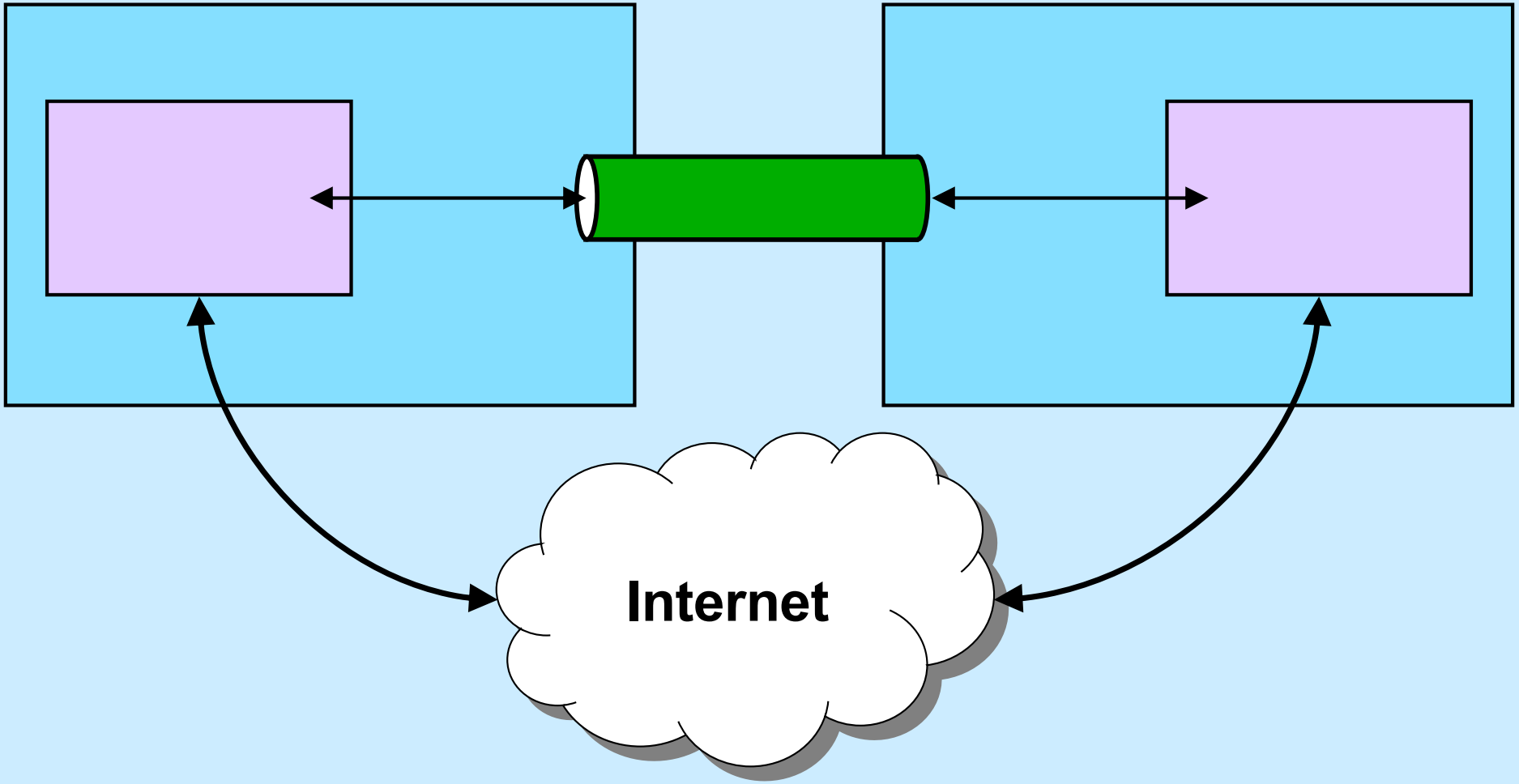
Interprocess Communication: Same Machine I



Interprocess Communication: Same Machine II

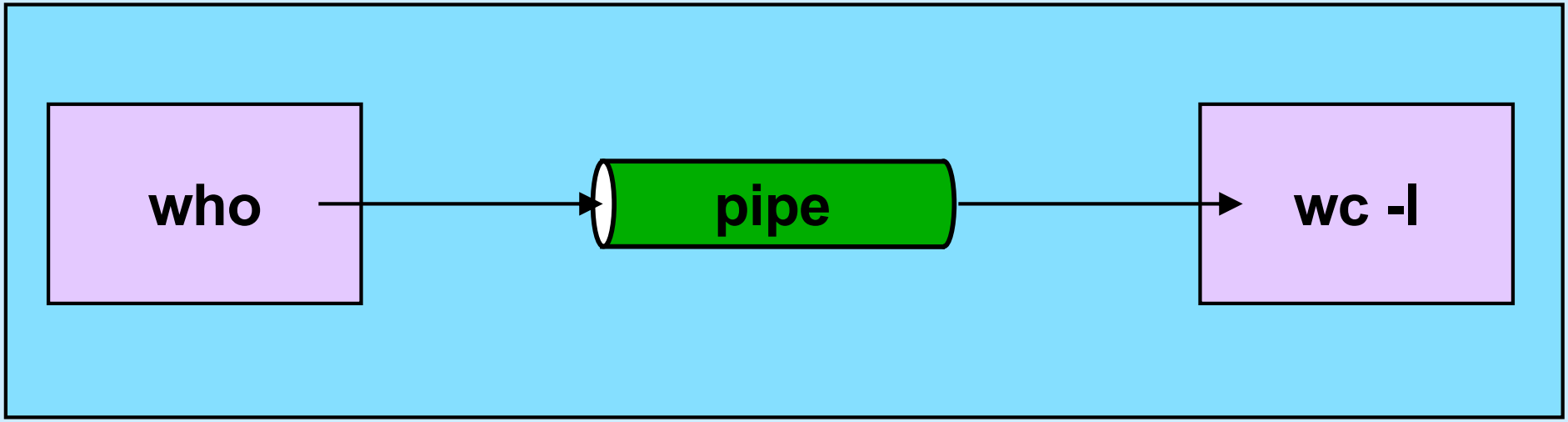


Interprocess Communication: Different Machines



Pipes

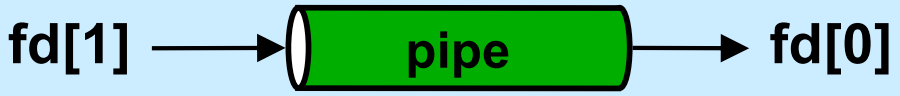
```
$cslab2e who | wc -l
```



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" (done in shell 2)
        } ... 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; D | B; D | C; D = (A | B | C); D
- 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

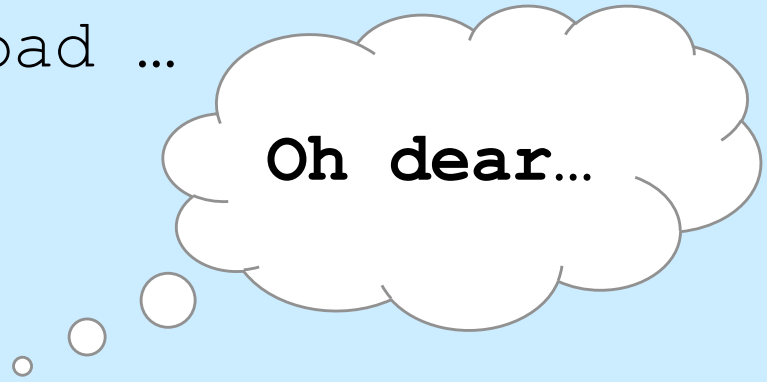
Signals Part 1

An Interlude Between Shells

- Shell 1
  - it can run programs
  - it can redirect I/O
- Signals
  - a mechanism for coping with exceptions and external events
  - the mechanism needed for shell 2
- Shell 2
  - it can control running programs

Whoops ...

```
$ SometimesUsefulProgram xyz
Are you sure you want to proceed? Y
Are you really sure? Y
Reformatting of your disk will begin
in 3 seconds.
Everything you own will be deleted.
There's little you can do about it.
Too bad …
```



A Gentler Approach

- Signals
  - get a process's attention
    - » send it a signal
  - process must either deal with it or be terminated
    - » in some cases, the latter is the only option

Stepping Back …

- What are we trying to do?
  - interrupt the execution of a program
    - » cleanly terminate it

      or
    - » cleanly change its course
  - not for the faint of heart
    - » it's difficult
    - » it gets complicated
    - » (not done in Windows)

Signals

- Generated (by OS) in response to
  - exceptions (e.g., arithmetic errors, addressing problems)
    - » synchronous signals
  - external events (e.g., timer expiration, certain keystrokes, actions of other processes)
    - » asynchronous signals
- Effect on process:
  - termination (possibly producing a core dump)
  - invocation of a function that has been set up to be a signal handler
  - suspension of execution
  - resumption of execution

Signal Types

SIGABRT	*abort* called	term, core
SIGALRM	alarm clock	term
SIGCHLD	death of a child	ignore
SIGCONT	continue after stop	cont
SIGFPE	erroneous arithmetic operation	term, core
SIGHUP	hangup on controlling terminal	term
SIGILL	illegal instruction	term, core
SIGINT	interrupt from keyboard	term
SIGKILL	kill	forced term
SIGPIPE	write on pipe with no one to read	term
SIGQUIT	quit	term, core
SIGSEGV	invalid memory reference	term, core
SIGSTOP	stop process	forced stop
SIGTERM	software termination signal	term
SIGTSTP	stop signal from keyboard	stop
SIGTTIN	background read attempted	stop
SIGTTOU	background write attempted	stop
SIGUSR1	application-defined signal 1	stop
SIGUSR2	application-defined signal 2	stop

Sending a Signal

- `int kill(pid_t pid, int sig)`
  - send signal *sig* to process *pid*
- Also
  - *kill* shell command
  - type ctrl-c
    - » sends signal 2 (SIGINT) to current process
  - type ctrl-\
    - » sends signal 3 (SIGQUIT) to current process
  - type ctrl-z
    - » sends signal 20 (SIGTSTP) to current process
  - do something bad
    - » bad address, bad arithmetic, etc.

Handling Signals

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signo,
    sighandler_t handler);

sighandler_t OldHandler;

OldHandler = signal(SIGINT, NewHandler);
```

Special Handlers

- **SIG_IGN**
  - **ignore the signal**
  - `signal(SIGINT, SIG_IGN);`
- **SIG_DFL**
  - **use the default handler**
    - » **usually terminates the process**
  - `signal(SIGINT, SIG_DFL);`

Example

```
void sigloop() {
  while(1)
    ;
}

int main() {
  void handler(int);
  signal(SIGINT, handler);
  sigloop();
  return 1;
}
void handler(int signo) {
  printf("I received signal %d. "
     "Whoopee!!\n", signo);
}
```

Digression: Core Dumps

- Core dumps
  - files (called “core”) that hold the contents of a process’s address space after termination by a signal
  - they’re large and rarely used, so they’re often disabled by default
  - use the ulimit command in bash to enable them

```
ulimit -c unlimited
```

  - use gdb to examine the process (post-mortem debugging)

```
gdb sig core
```

sigaction

```
int sigaction(int sig, const struct sigaction *new,
              struct sigaction *old);
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
};

int main() {
    struct sigaction act; void myhandler(int);
    sigemptyset(&act.sa_mask); // zeroes the mask
    act.sa_flags = 0;
    act.sa_handler = myhandler;
    sigaction(SIGINT, &act, NULL);
    …
}
```

Example

```
int main() {
  void handler(int);
  struct sigaction act;
  act.sa_handler = handler;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  sigaction(SIGINT, &act, 0);

  while(1)
    ;
  return 1;
}

void handler(int signo) {
  printf("I received signal %d. "
     "Whoopee!!\n", signo);
}
```

Quiz 3

```
int main() {
  void handler(int);
  struct sigaction act;
  act.sa_handler = hand
  sigemptyset(&act.sa_m
  act.sa_flags = 0;
  sigaction(SIGINT, &ac

  while(1)
    ;
  return 1;
}

void handler(int signo) {
  printf("I received signal %d. "
     "Whoopee!!\n", signo);
}
```

You run the example program, then quickly type ctrl-C. What is the most likely explanation if the program then terminates?

a) **this "can't happen"; thus there's a problem with the system**
b) **you're really quick or the system is really slow (or both)**
c) **what we've told you so far isn't quite correct**