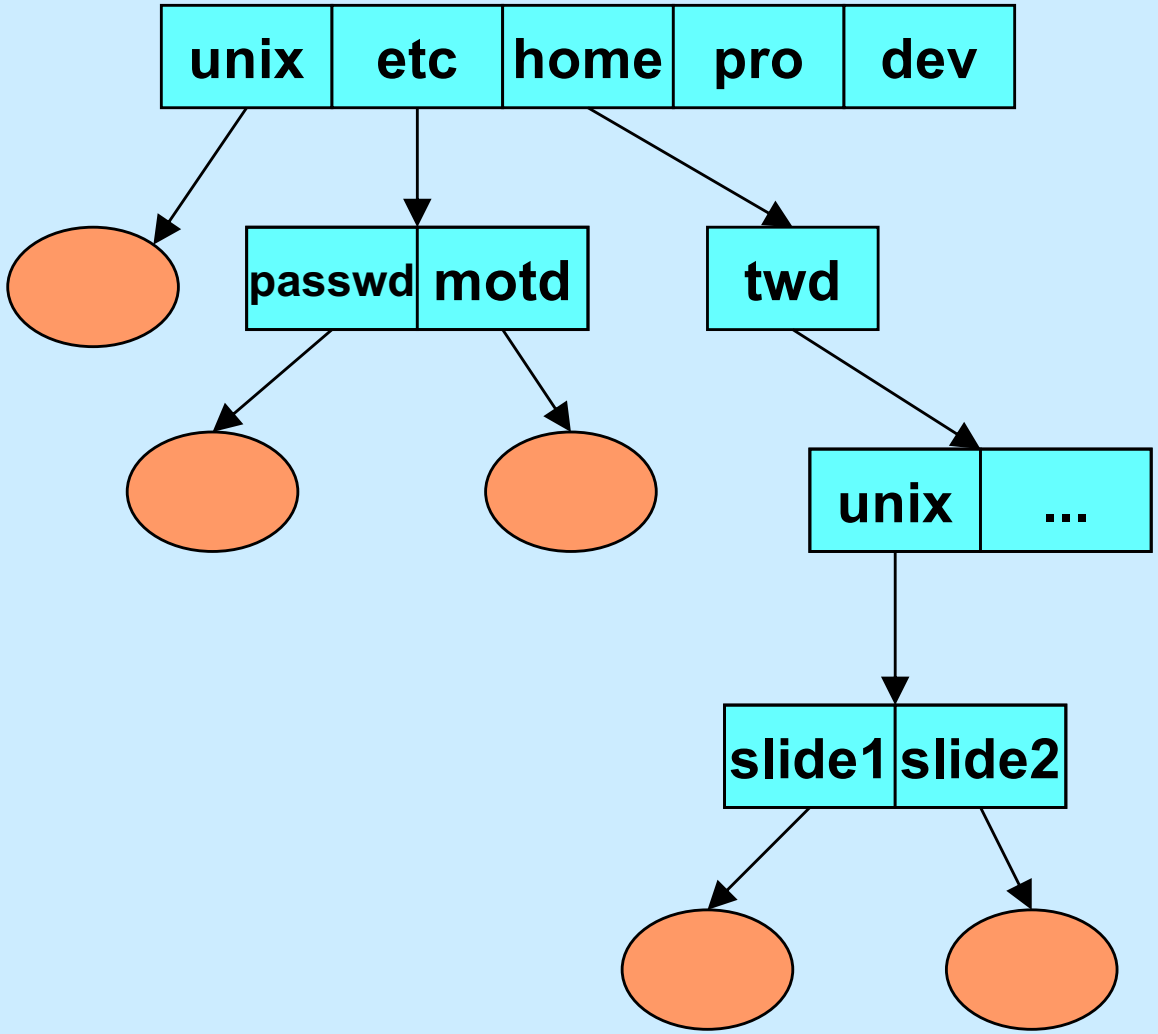
CS 33

Files Part 3

Copyright © 2023 Thomas W. Doeppner. All rights reserved.

Directories



Copyright © 2023 Thomas W. Doeppner. All rights reserved.

Directory Representation

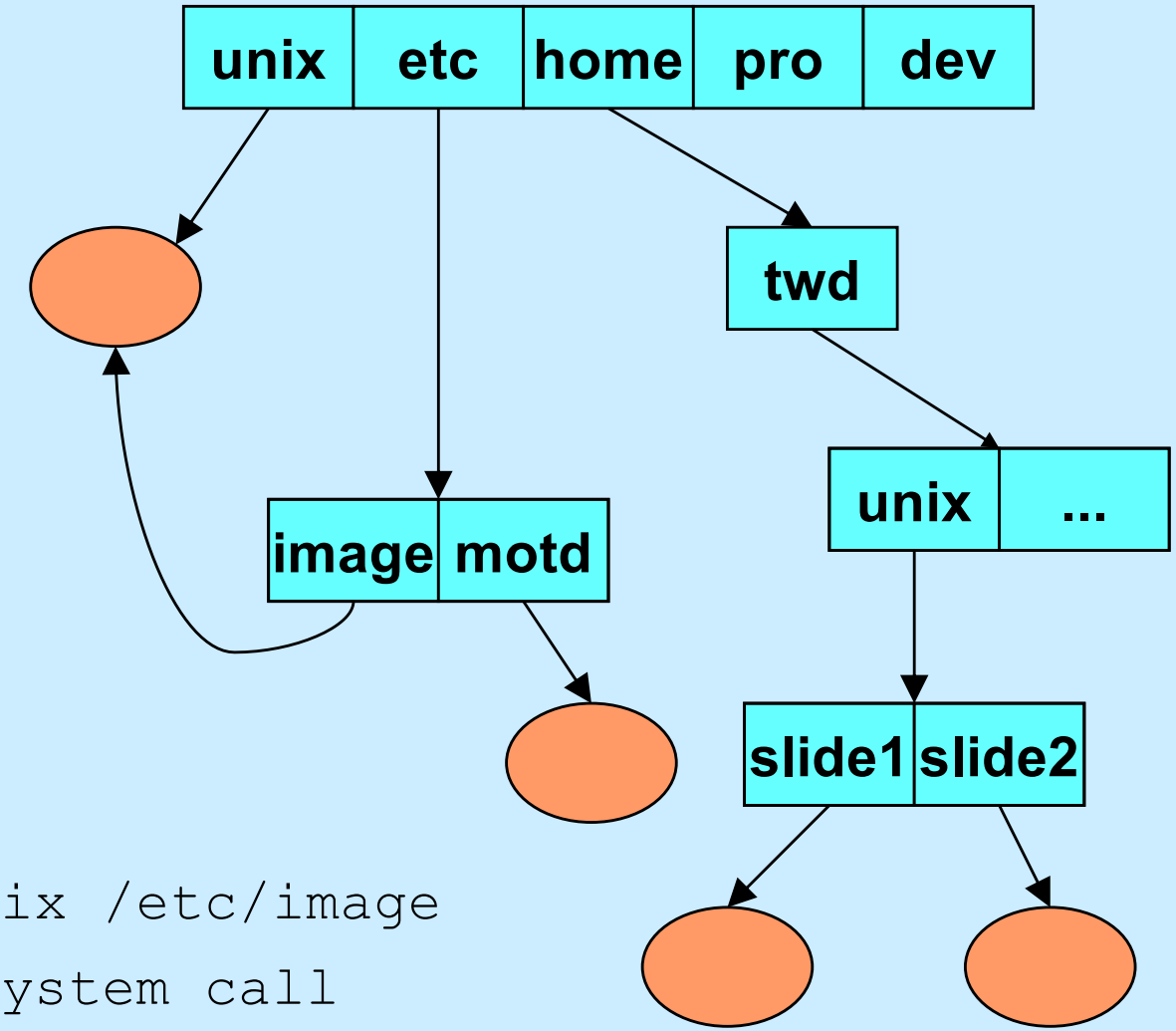
Component Name	Inode Number

directory entry

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

Copyright © 2023 Thomas W. Doeppner. All rights reserved.

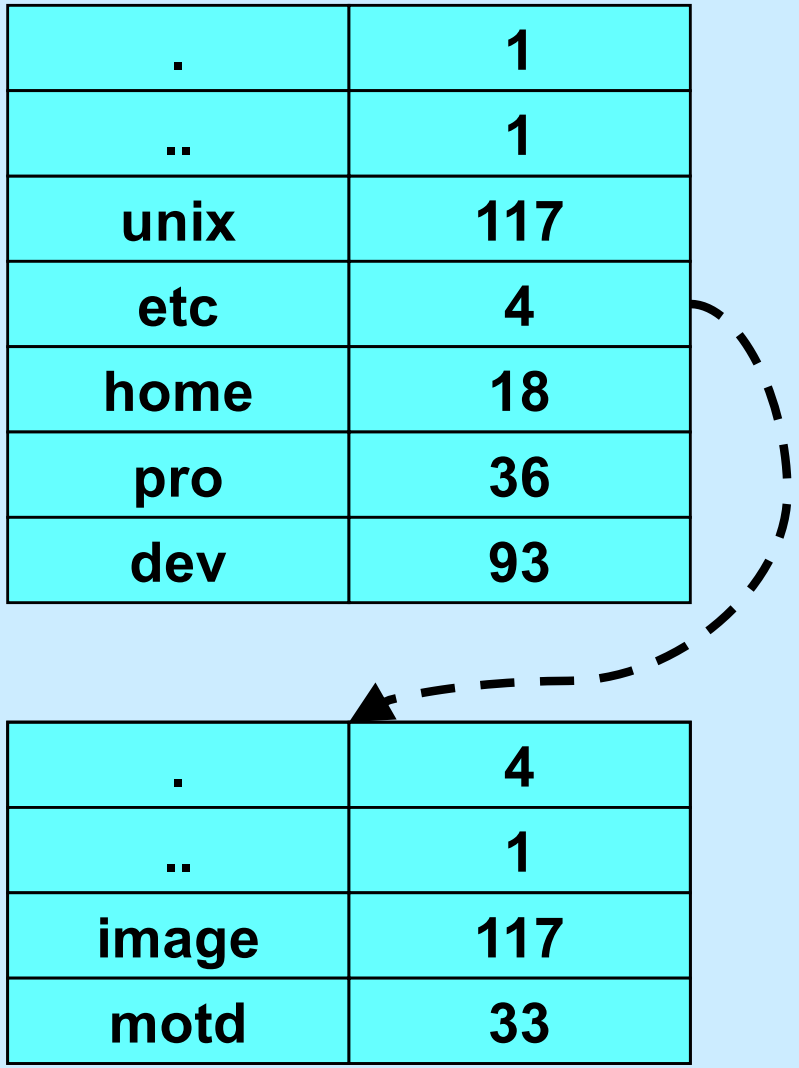
Hard Links




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

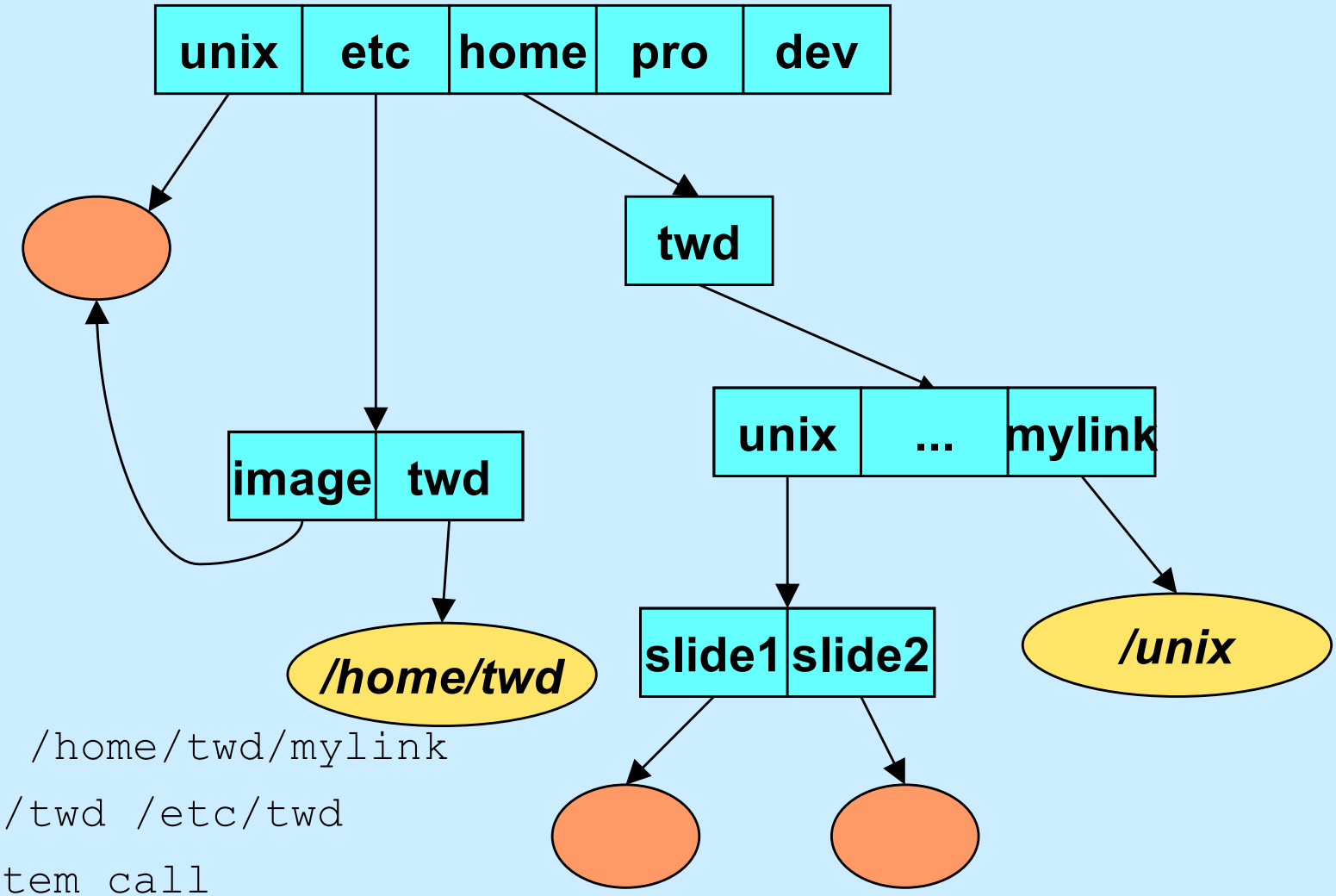

Copyright © 2023 Thomas W. Doeppner. All rights reserved.

Directory Representation



Copyright © 2023 Thomas W. Doeppner. All rights reserved.

Symbolic Links




```
% ln -s /unix /home/twd/mylink
% ln -s /home/twd /etc/twd
# symlink system call
```


Copyright © 2023 Thomas W. Doeppner. All rights reserved.

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?

Copyright © 2023 Thomas W. Doeppner. All rights reserved.

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**

Copyright © 2023 Thomas W. Doeppner. All rights reserved.

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?
```


Copyright © 2023 Thomas W. Doeppner. All rights reserved.

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
```


Copyright © 2023 Thomas W. Doeppner. All rights reserved.

In the Shell ...

```
% program >> file
```


Copyright © 2023 Thomas W. Doeppner. All rights reserved.

File Access Permissions

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

Copyright © 2023 Thomas W. Doeppner. All rights reserved.

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
```


Copyright © 2023 Thomas W. Doeppner. All rights reserved.

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)`

Copyright © 2023 Thomas W. Doeppner. All rights reserved.

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

Copyright © 2023 Thomas W. Doeppner. All rights reserved.

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

Copyright © 2023 Thomas W. Doeppner. All rights reserved.

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

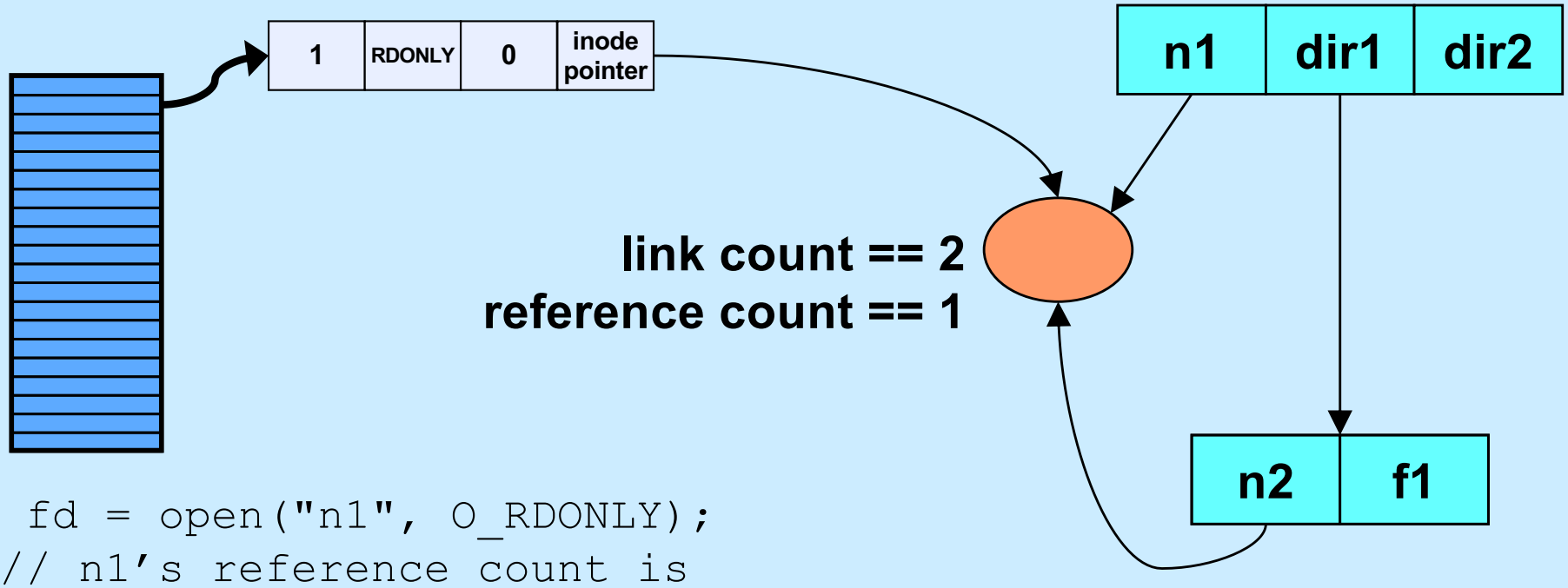
Copyright © 2023 Thomas W. Doeppner. All rights reserved.

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**

Copyright © 2023 Thomas W. Doeppner. All rights reserved.

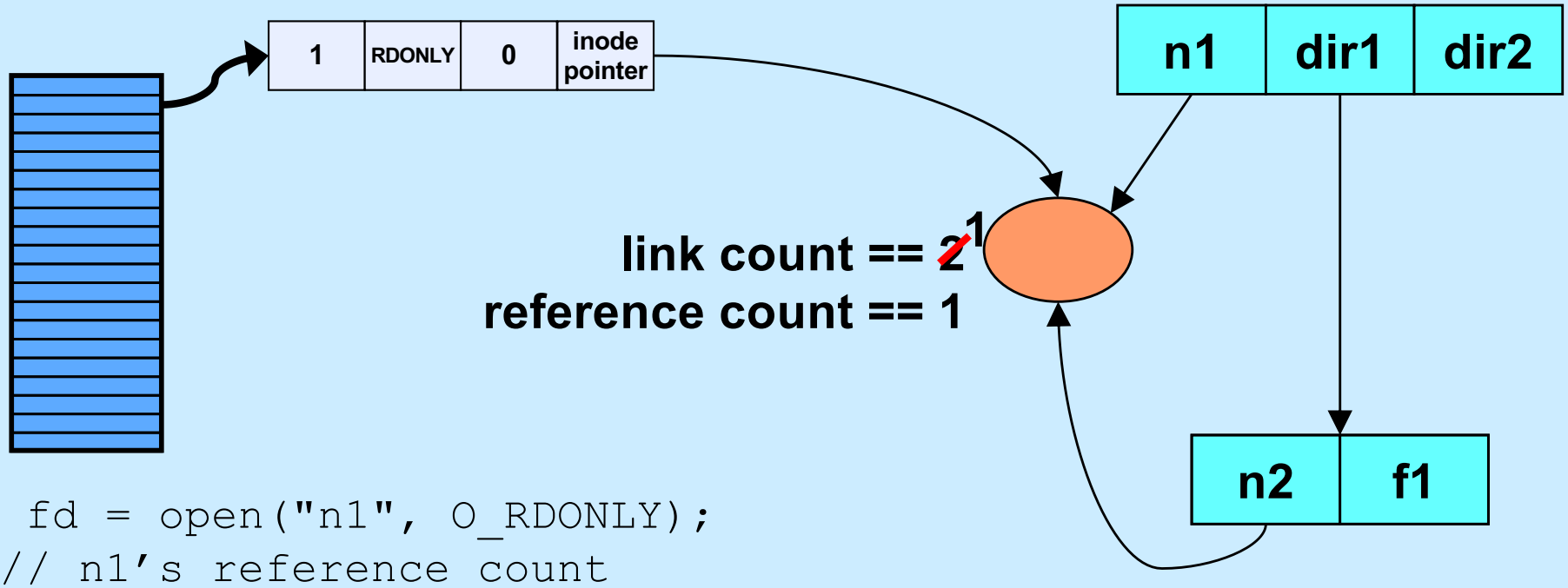
Link and Reference Counts




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


Copyright © 2023 Thomas W. Doeppner. All rights reserved.

Link and Reference Counts

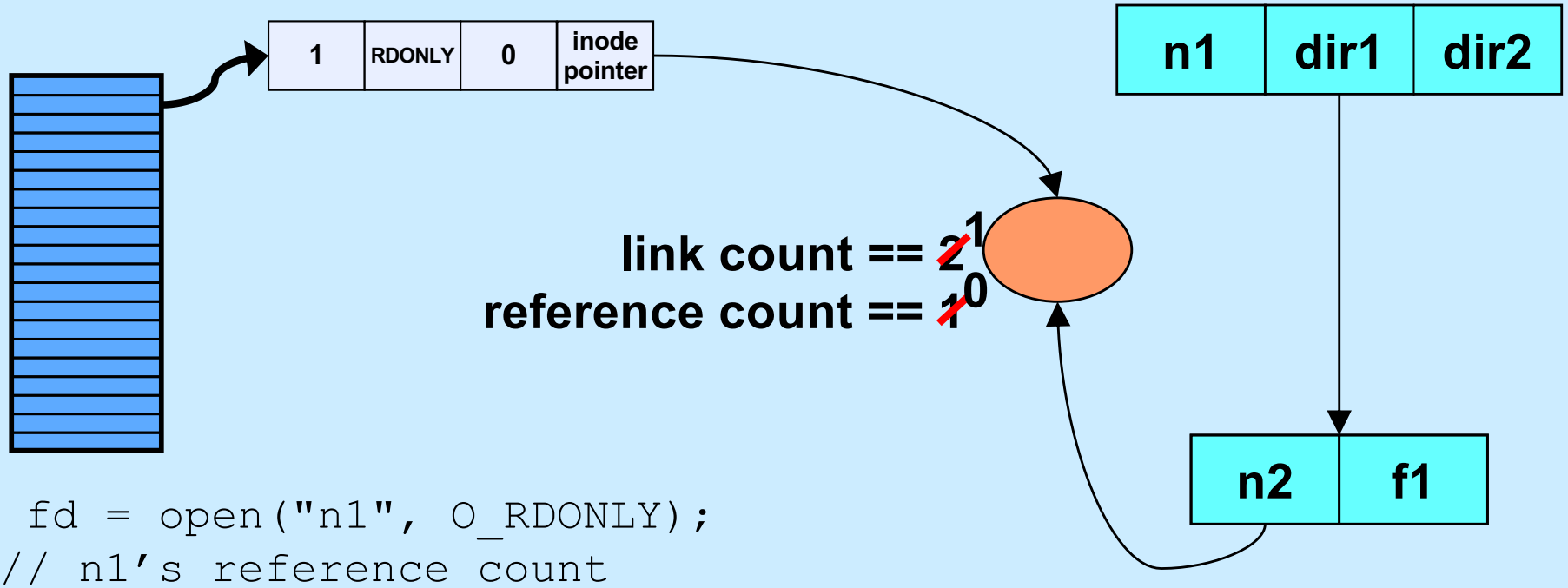



```
int fd = open("n1", O_RDONLY);
    // n1's reference count
    // incremented by 1

unlink("n1");
    // link count decremented by 1
    // same effect in shell via "rm n1"
```


Copyright © 2023 Thomas W. Doeppner. All rights reserved.

Link and Reference Counts



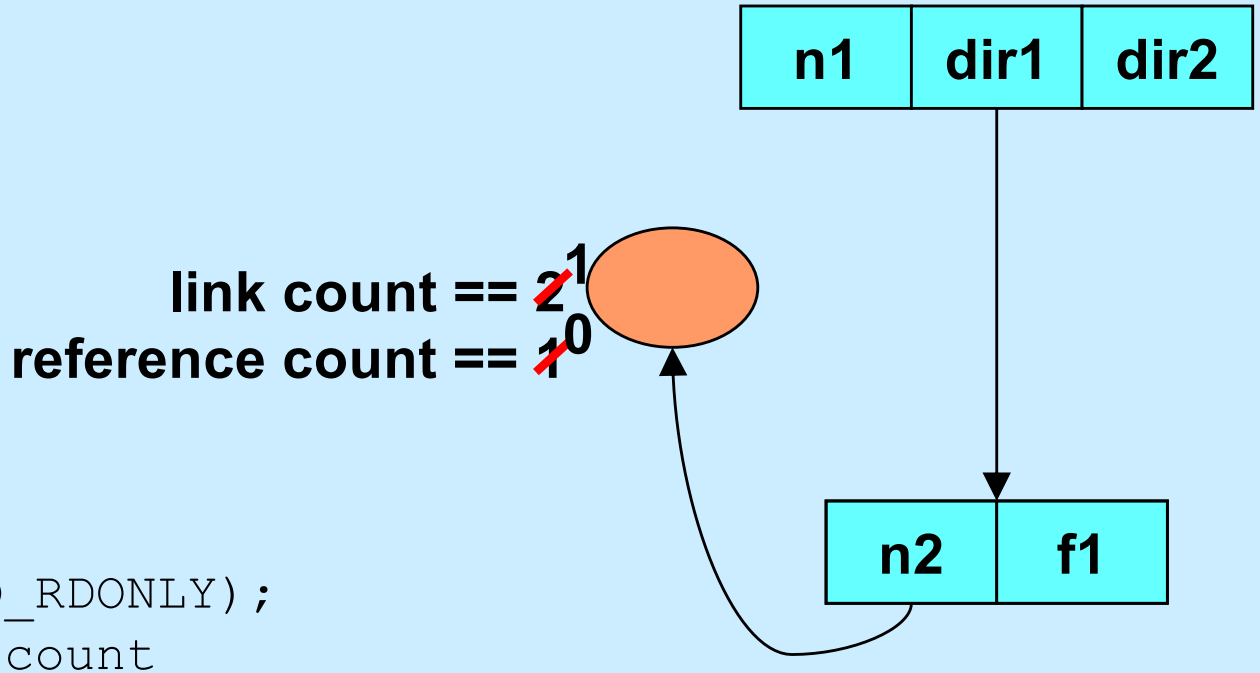

```
int fd = open("n1", O_RDONLY);
   // n1's reference count
   // incremented by 1

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

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


Copyright © 2023 Thomas W. Doeppner. All rights reserved.

Link and Reference Counts



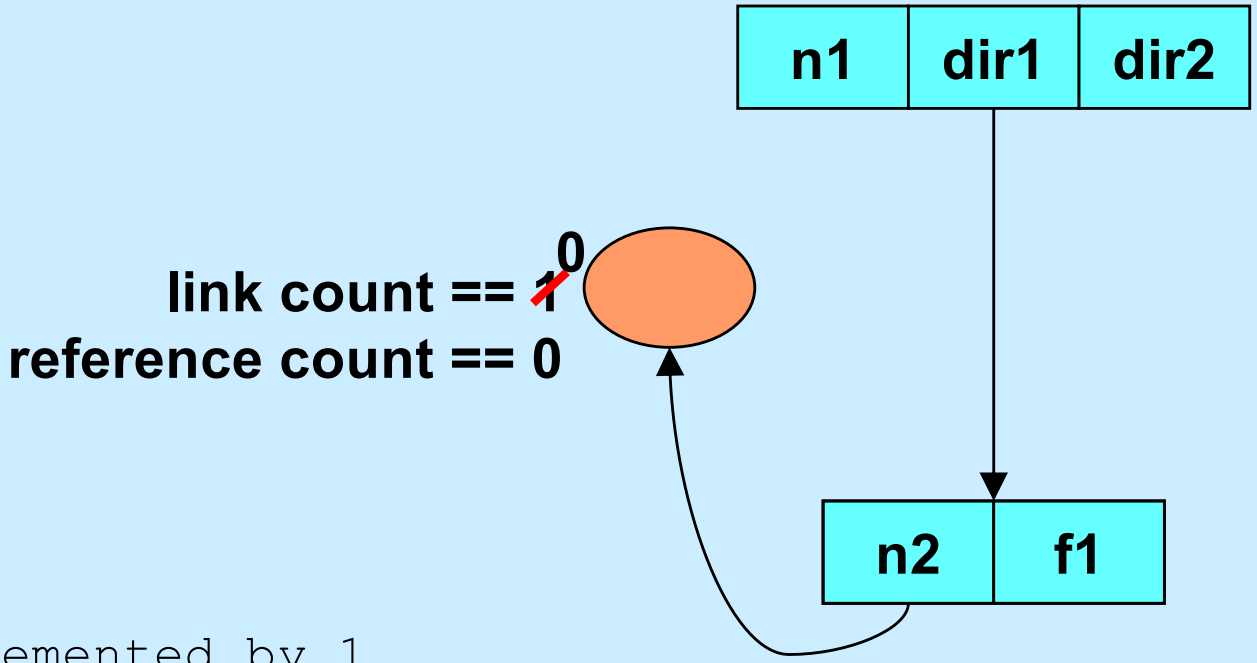

```
int fd = open("n1", O_RDONLY);
   // n1's reference count
   // incremented by 1

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

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


Copyright © 2023 Thomas W. Doeppner. All rights reserved.

Link and Reference Counts




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


Copyright © 2023 Thomas W. Doeppner. All rights reserved.

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.

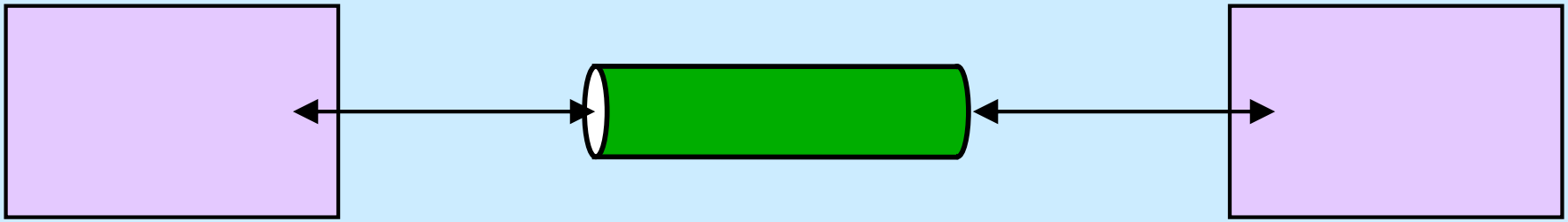
a) This program is doomed to failure, since the file is deleted before it's used

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

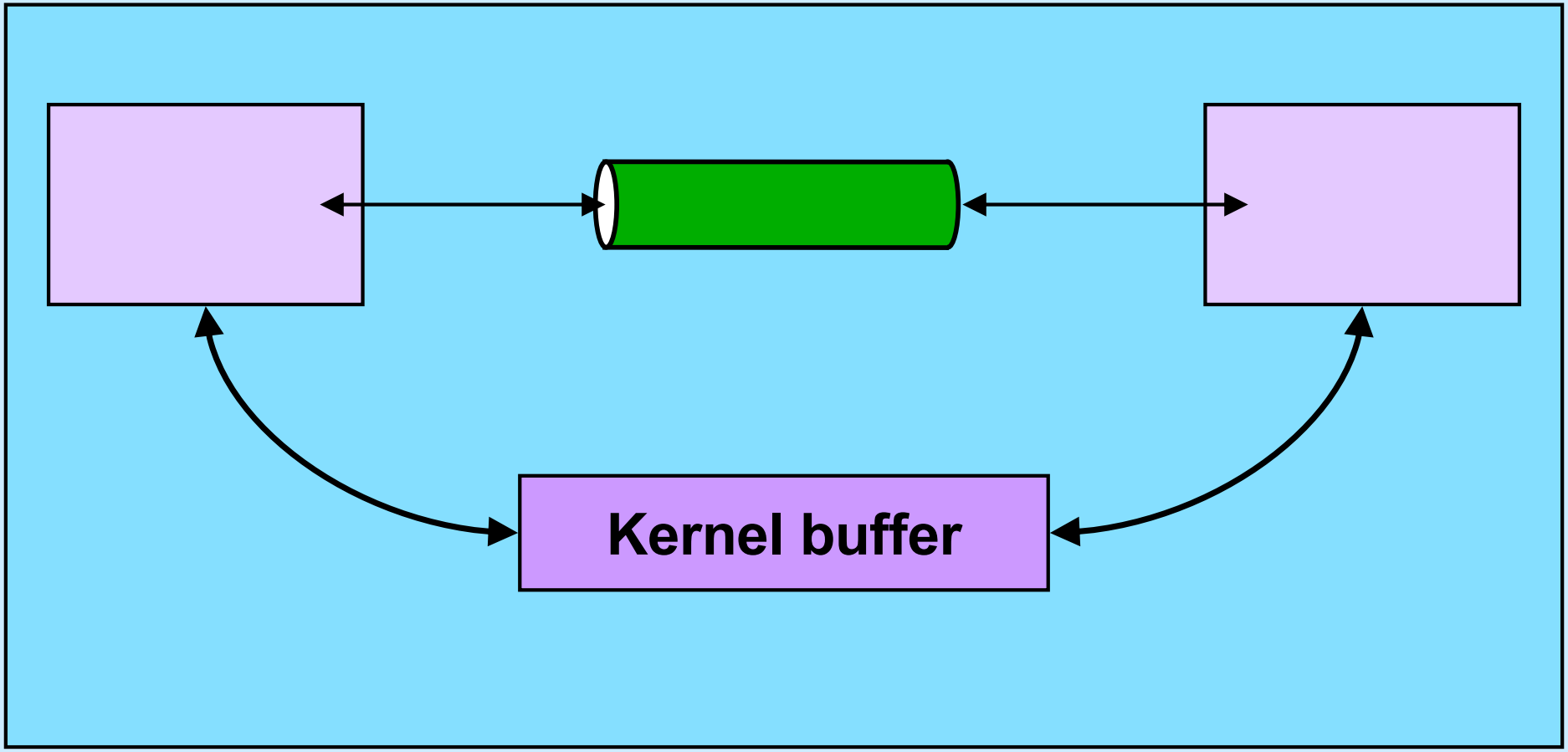
Copyright © 2023 Thomas W. Doeppner. All rights reserved.

Interprocess Communication (IPC): Pipes



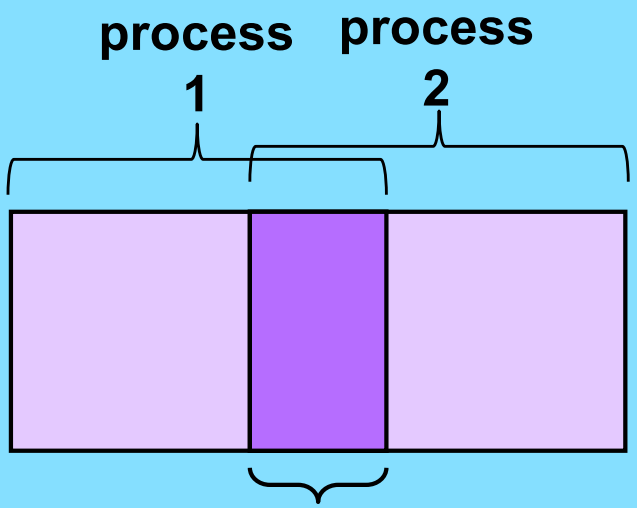
Copyright © 2023 Thomas W. Doeppner. All rights reserved.

Interprocess Communication: Same Machine I



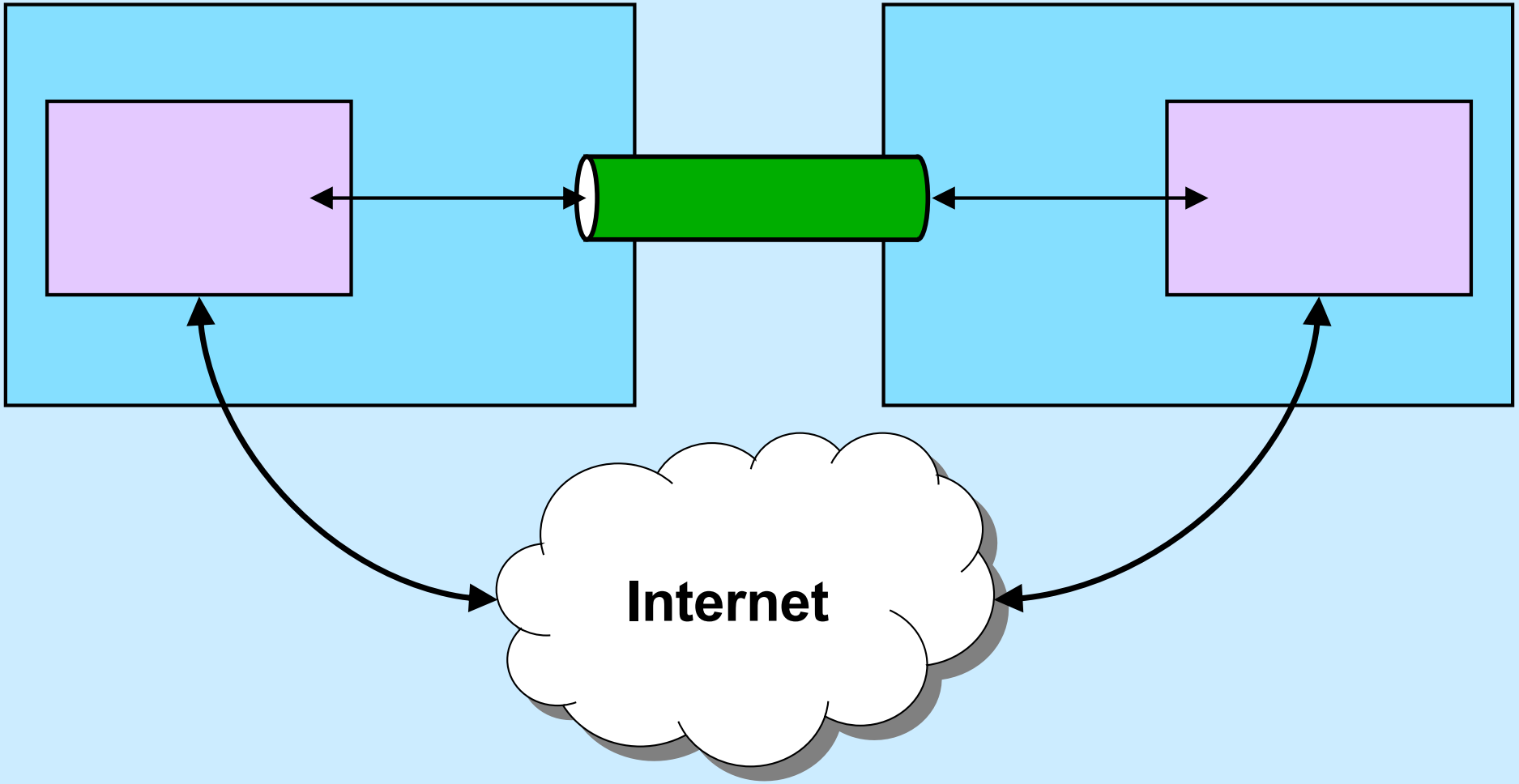
 Copyright © 2023 Thomas W. Doeppner. All rights reserved.

Interprocess Communication: Same Machine II



Copyright © 2023 Thomas W. Doeppner. All rights reserved.

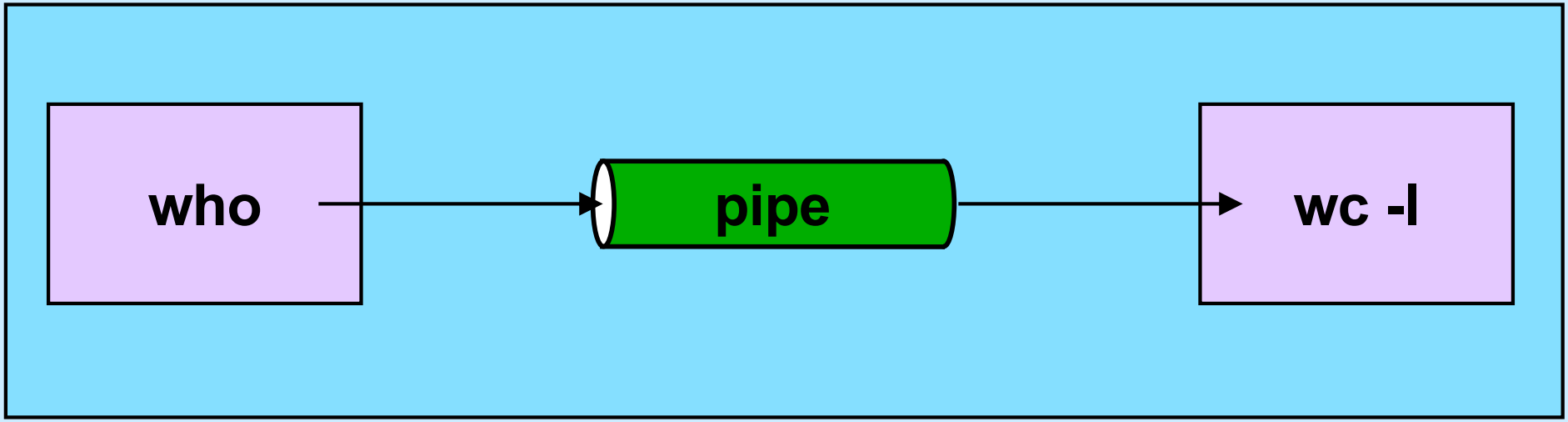
Interprocess Communication: Different Machines



Copyright © 2023 Thomas W. Doeppner. All rights reserved.

Pipes

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

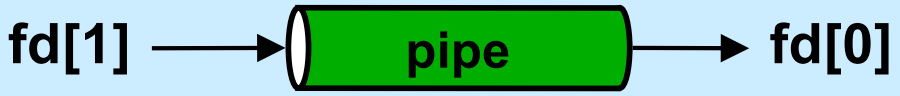


Copyright © 2023 Thomas W. Doeppner. All rights reserved.

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]);
// …
```



Copyright © 2023 Thomas W. Doeppner. All rights reserved.

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

Copyright © 2023 Thomas W. Doeppner. All rights reserved.

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


Copyright © 2023 Thomas W. Doeppner. All rights reserved.

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


Copyright © 2023 Thomas W. Doeppner. All rights reserved.

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!)

Copyright © 2023 Thomas W. Doeppner. All rights reserved.

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!
}
```


Copyright © 2023 Thomas W. Doeppner. All rights reserved.

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!
}
```


Copyright © 2023 Thomas W. Doeppner. All rights reserved.

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

Copyright © 2023 Thomas W. Doeppner. All rights reserved.

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!

Copyright © 2023 Thomas W. Doeppner. All rights reserved.