CS 33

Files Part 2

A Program

```
int main(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr, "Usage: echon reps\n");
    exit(1);
  }
  int reps = atoi(argv[1]);
  if (reps > 2) {
    fprintf(stderr, "reps too large, reduced to 2\n");
    reps = 2;
  }
  char buf[256];
  while (fgets(buf, 256, stdin) != NULL)
    for (int i=0; i<reps; i++)
      fputs(buf, stdout);
  return(0);
}
```

From the Shell ...

```
$ echon 1
```

- ***stdout*** **(fd 1) and** ***stderr*** **(fd 2) go to the display**
- ***stdin*** **(fd 0) comes from the keyboard**

```
$ echon 1 > Output
```

- ***stdout*** **goes to the file "Output" in the current directory**
- ***stderr*** **goes to the display**
- ***stdin*** **comes from the keyboard**

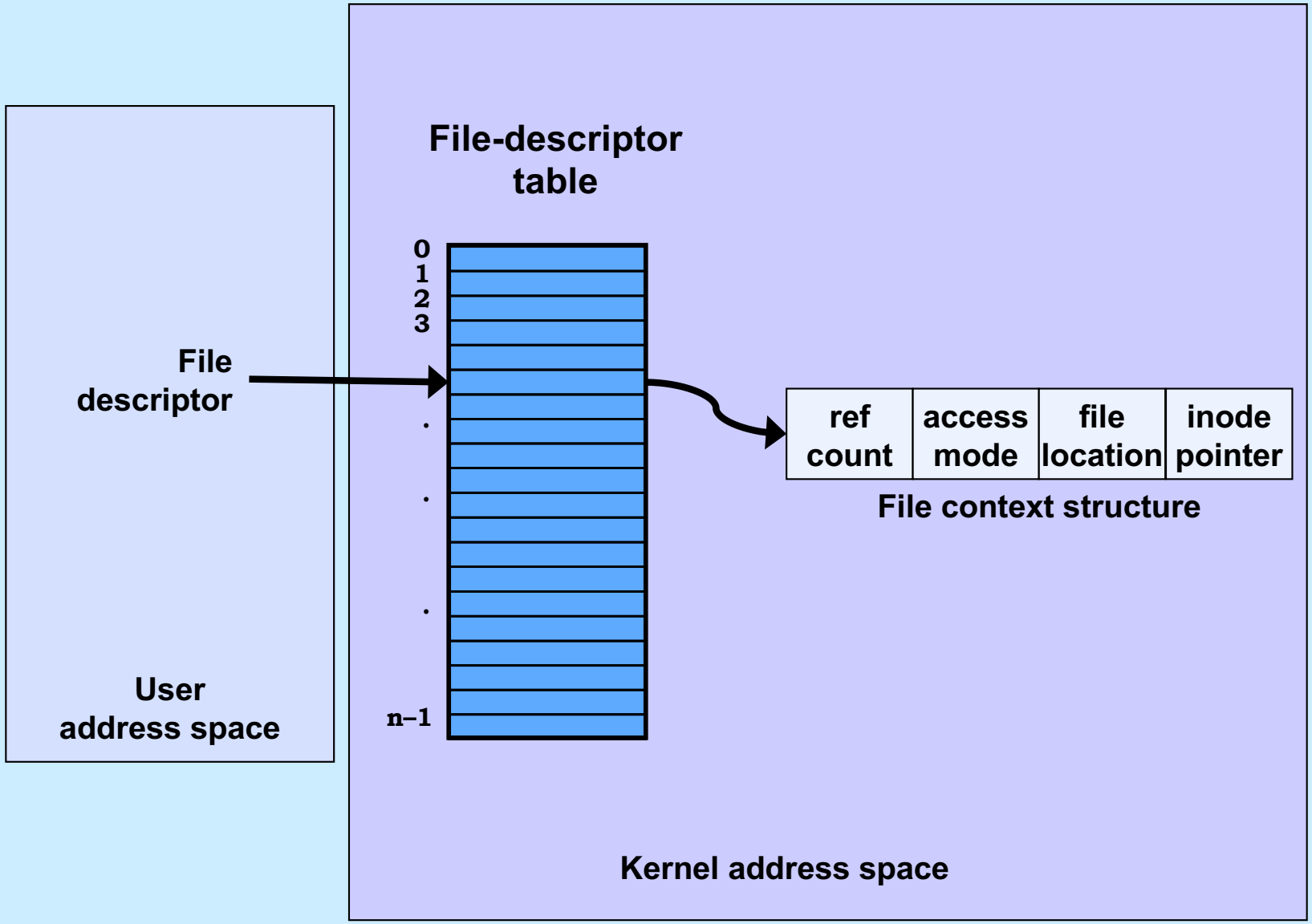
```
$ echon 1 < Input
```

- ***stdin*** **comes from the file "Input" in the current directory**

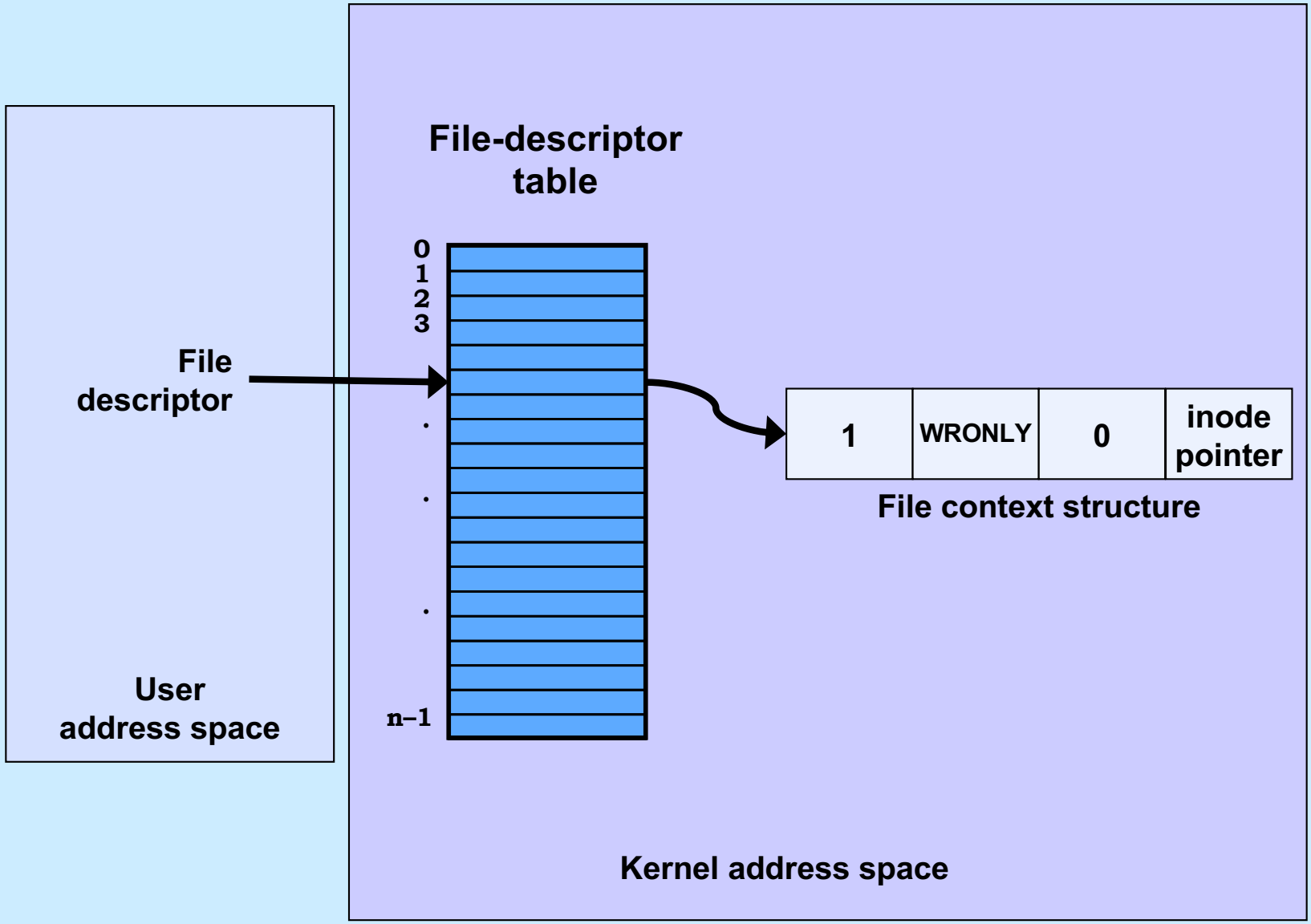
Redirecting Stdout in C

```
if ((pid = fork()) == 0) {
   /* set up file descriptor 1 in the child process */
   close(1);
   if (open("/home/twd/Output", O_WRONLY) == -1) {
      perror("/home/twd/Output");
      exit(1);
   }
   char *argv[] = {"echon", "2", NULL};
   execv("/home/twd/bin/echon", argv);
   exit(1);
}

/* parent continues here */

waitpid(pid, 0, 0);     // wait for child to terminate
```

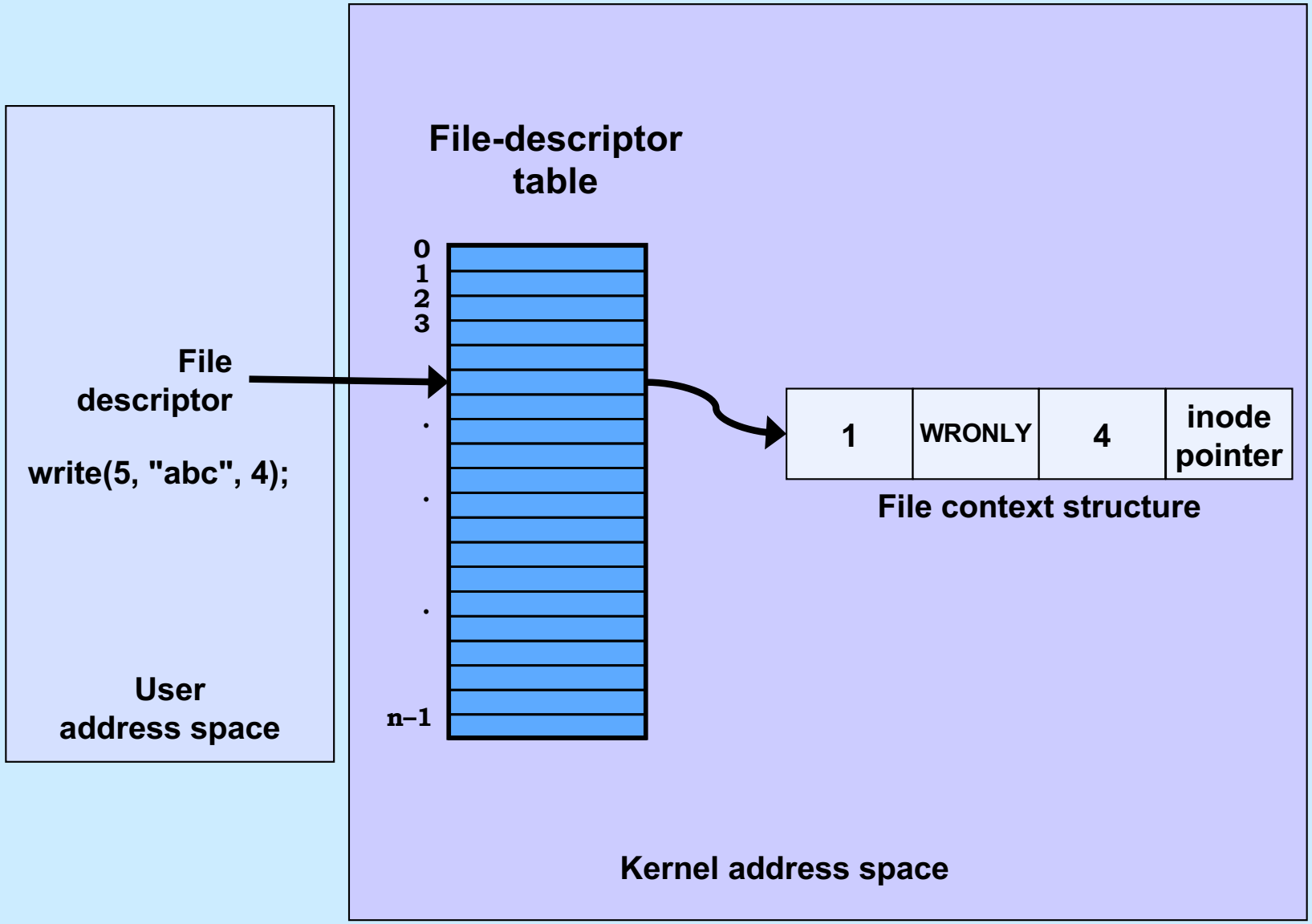
File-Descriptor Table



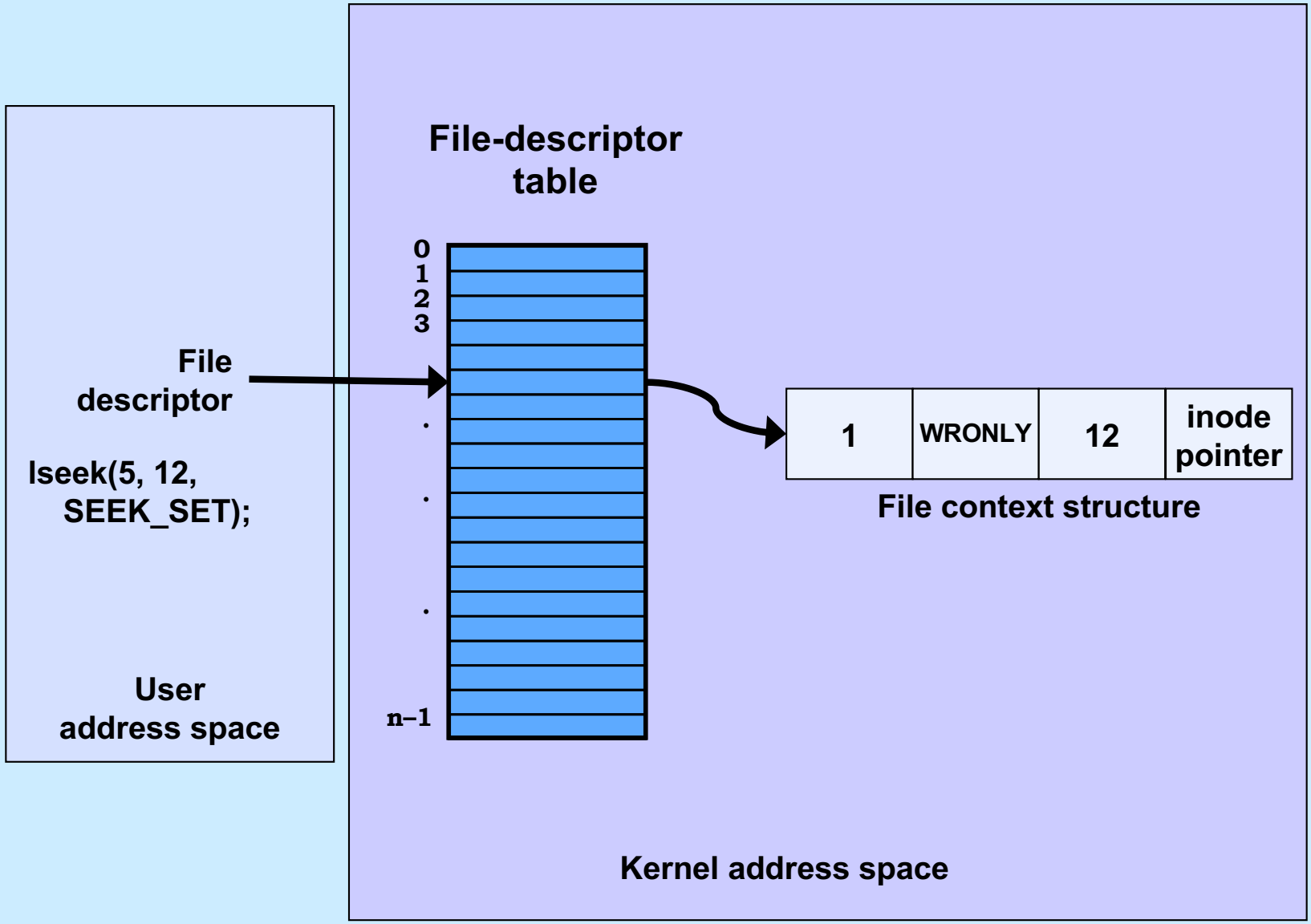
File Location



File Location



File Location



Allocation of File Descriptors

- **Whenever a process requests a new file descriptor, the lowest-numbered file descriptor not already associated with an open file is selected; thus**

```
#include <fcntl.h>
#include <unistd.h>

close(0);
fd = open("file", O_RDONLY);
```

  - **will always associate *file* with file descriptor 0 (assuming that *open* succeeds)**

Redirecting Output … Twice

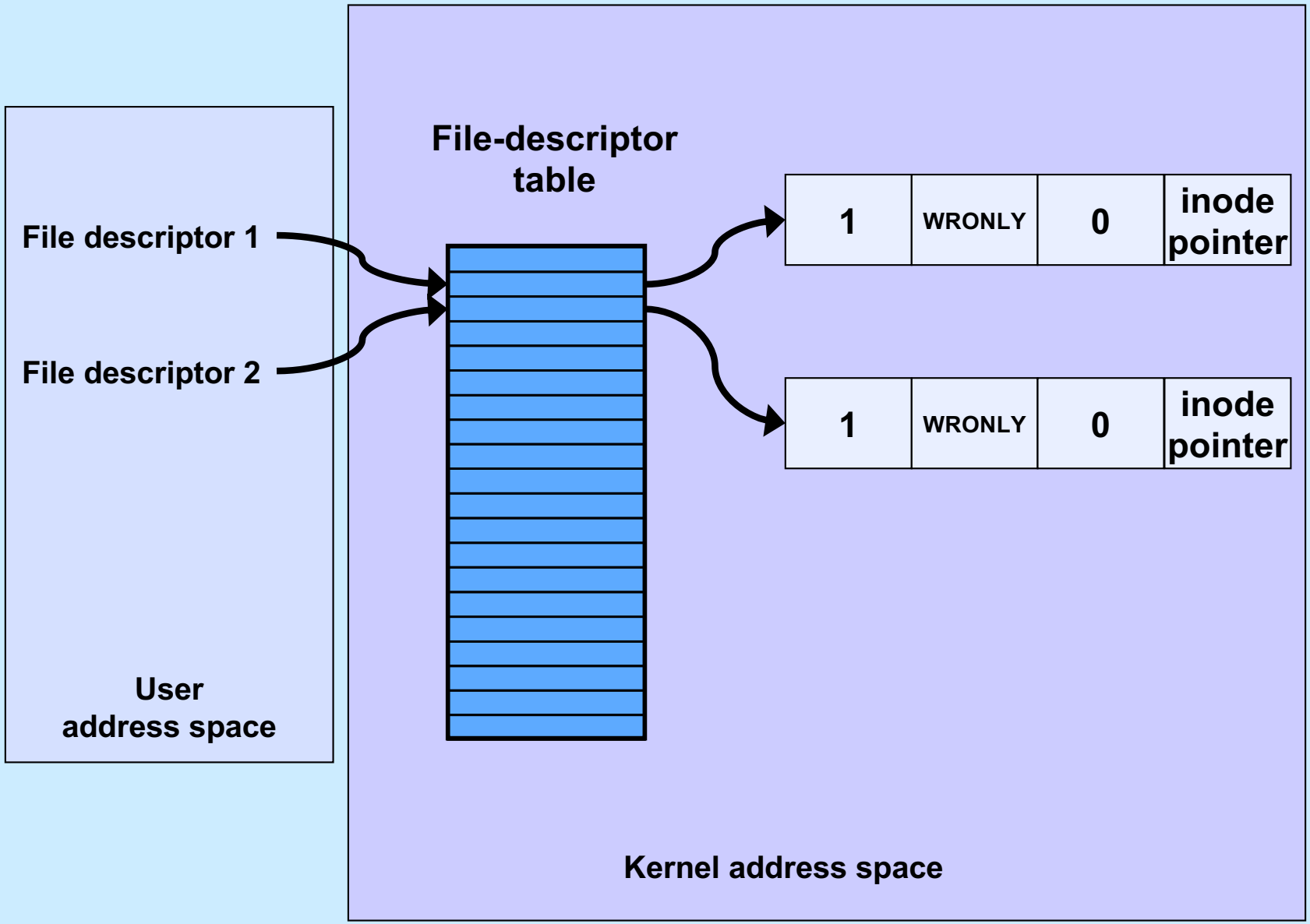
```
if (fork() == 0) {
   /* set up file descriptors 1 and 2 in the child process */
   close(1);
   close(2);
   if (open("/home/twd/Output", O_WRONLY) == -1) {
      exit(1);
   }
   if (open("/home/twd/Output", O_WRONLY) == -1) {
      exit(1);
   }
   char *argv[] = {"echon", 2, NULL};
   execv("/home/twd/bin/echon", argv);
   exit(1);
}
/* parent continues here */
```

From the Shell ...

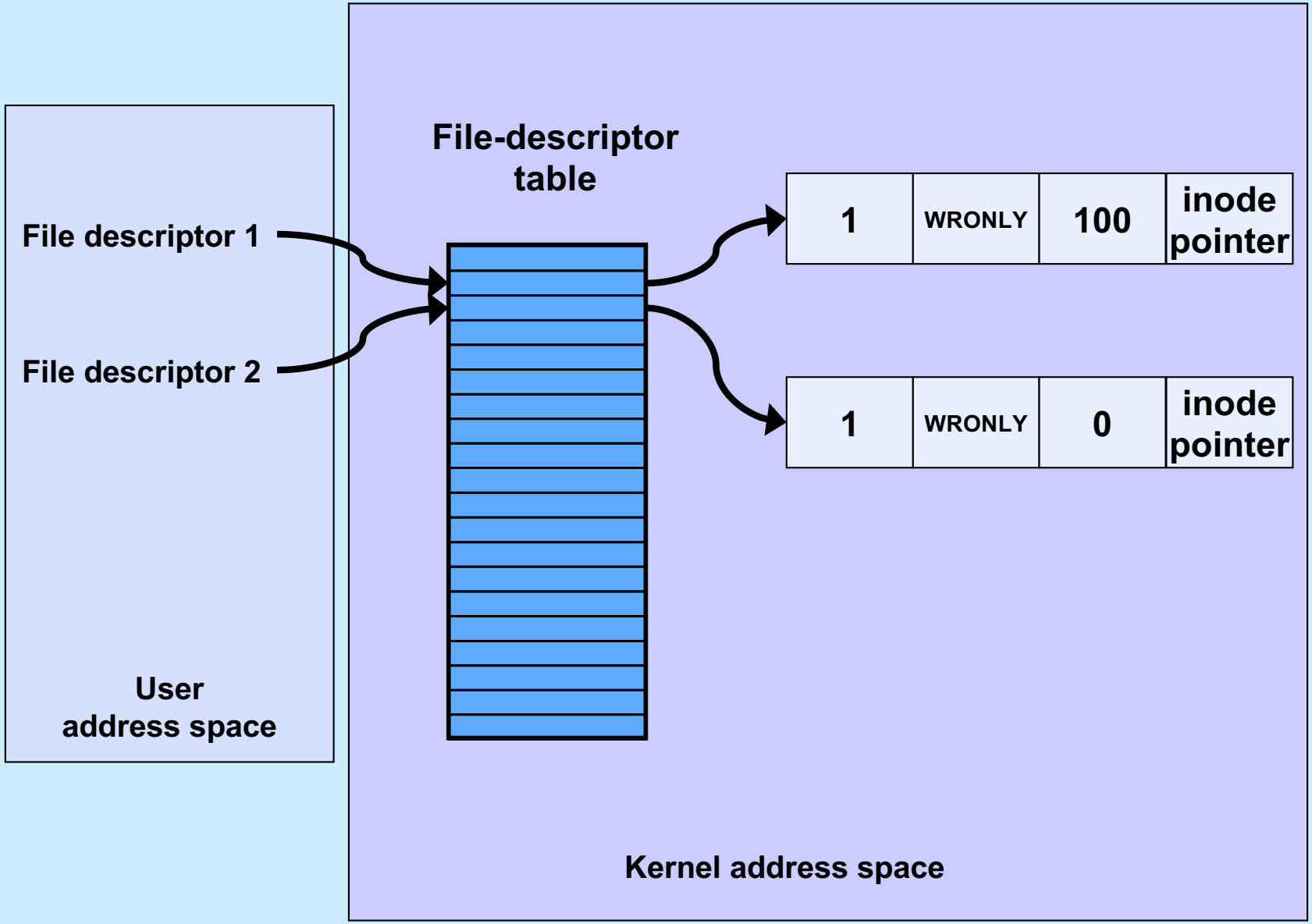
```
$ echon 1 >Output 2>Output
```

- **both stdout and stderr go to Output file**

Redirected Output



Redirected Output After Write



Quiz 1

- **Suppose we run**

```
$ echon 3 >Output 2>Output
```

- **The input line is**

```
X
```

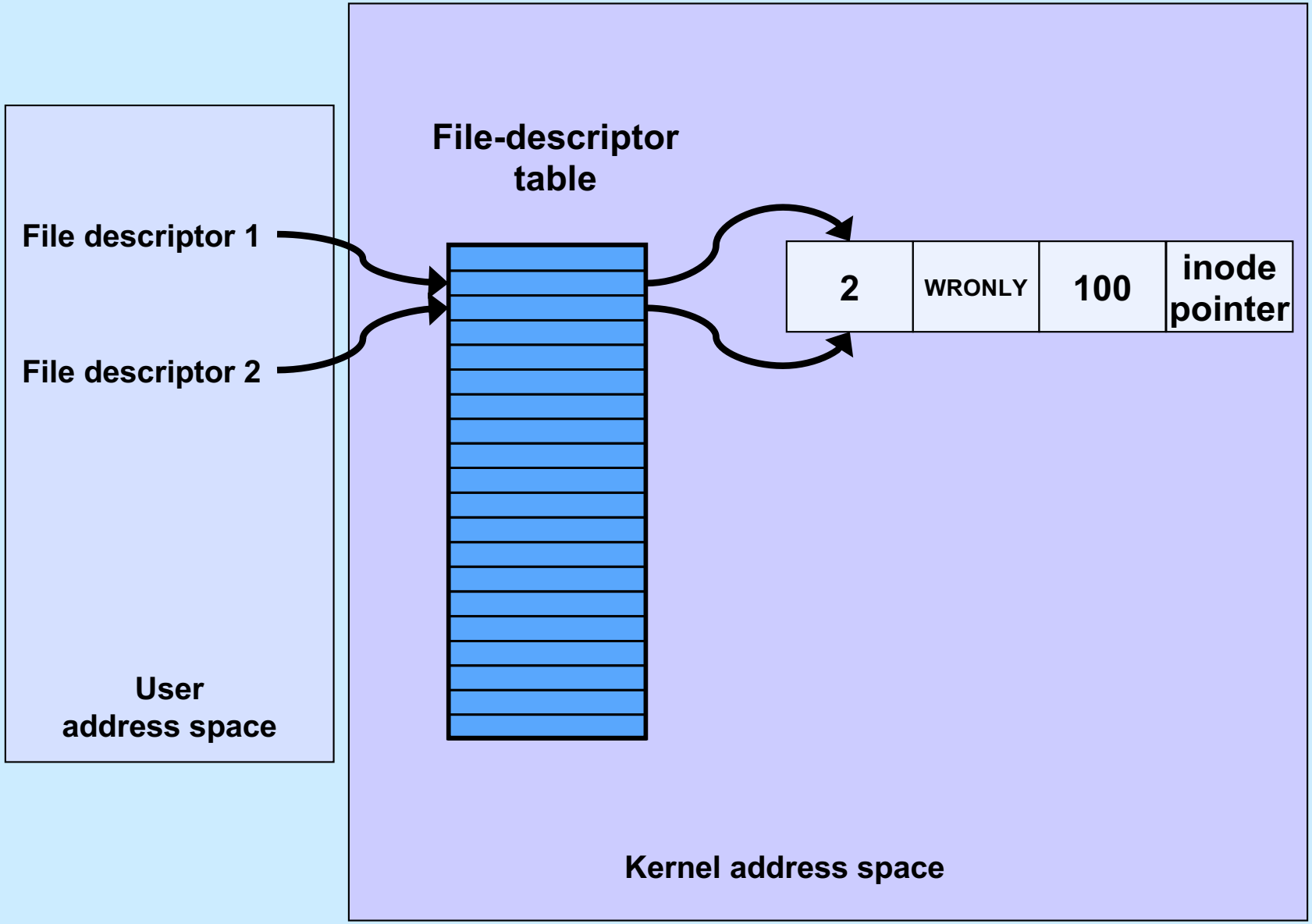
- **What is the final content of Output?**

```
a) reps too large, reduced to 2\nX\nX\n
b) X\nX\nreps too large, reduced to 2\n
c) X\nX\n too large, reduced to 2\n
```

Sharing Context Information

```
if (fork() == 0) {
   /* set up file descriptors 1 and 2 in the child process */
   close(1);
   close(2);
   if (open("/home/twd/Output", O_WRONLY) == -1) {
      exit(1);
   }
   dup(1); /* set up file descriptor 2 as a duplicate of 1 */
   char *argv[] = {"echon", 2};
   execv("/home/twd/bin/echon", argv);
   exit(1);
}
/* parent continues here */
```

Redirected Output After Dup



From the Shell ...

```
$ echon 3 >Output 2>&1
```

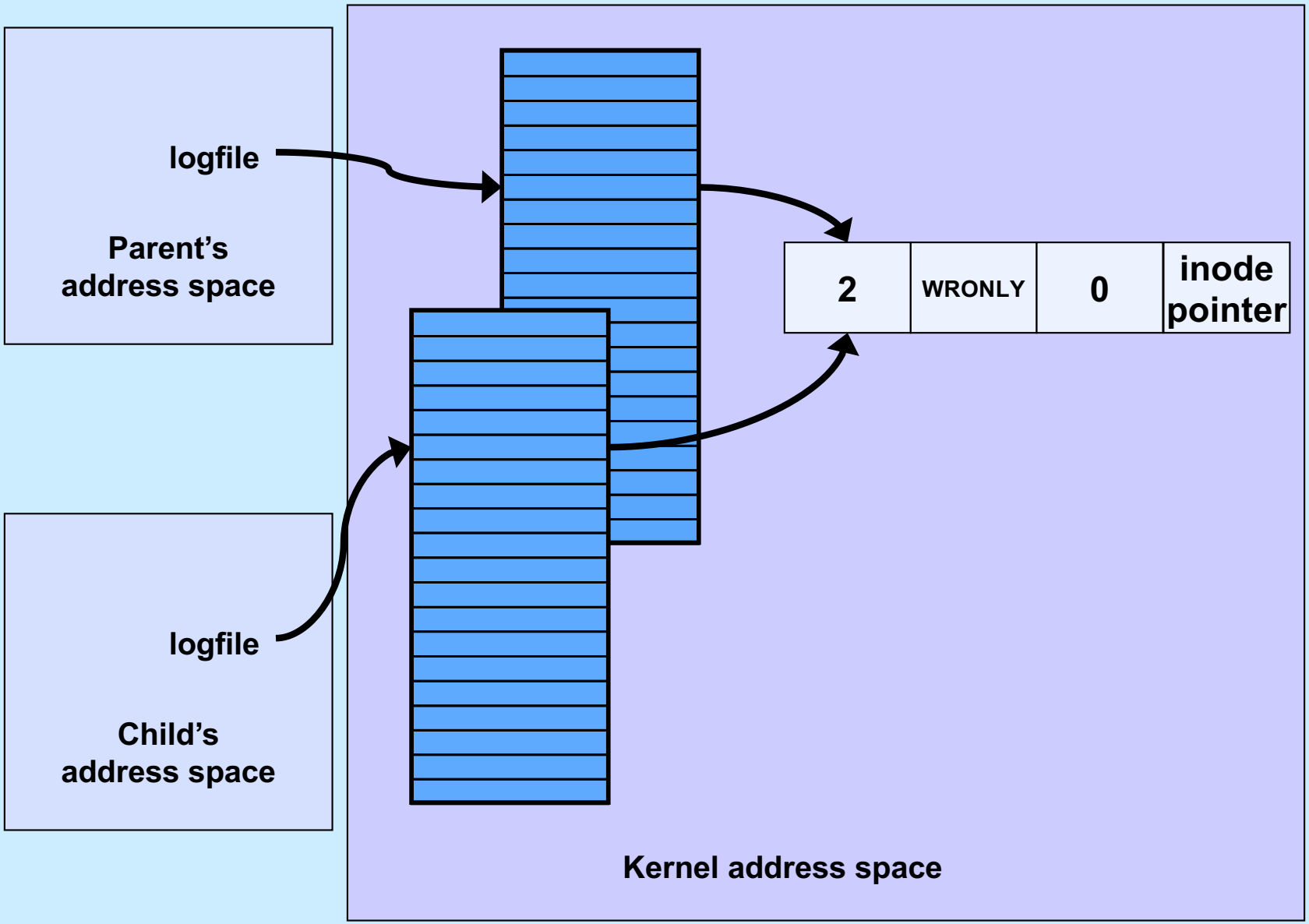
- **stdout goes to Output file, stderr is the dup of fd 1**
- **with input "X\n" it now produces in Output:**

```
reps too large, reduced to 2\nX\nX\n
```

Fork and File Descriptors

```
int logfile = open("log", O_WRONLY);
if (fork() == 0) {
   /* child process computes something, then does: */
   write(logfile, LogEntry, strlen(LogEntry));
   …
   exit(0);
}

/* parent process computes something, then does: */

write(logfile, LogEntry, strlen(LogEntry));
…
```

File Descriptors After Fork



Quiz 2

```
int main() {
  if (fork() == 0) {
    fprintf(stderr, "Child");
    exit(0);
  }
  fprintf(stderr, "Parent");
}
```

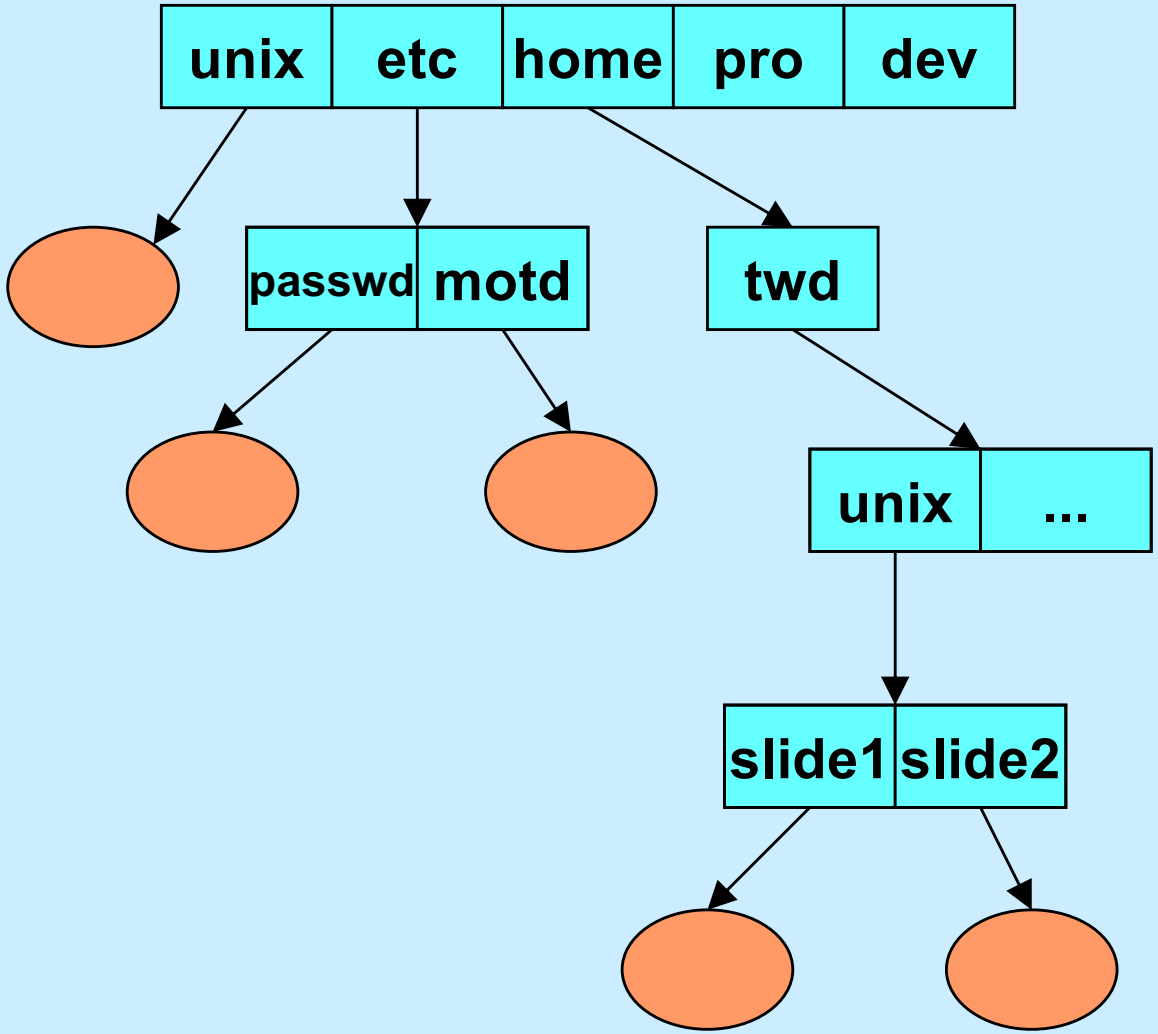
Suppose the program is run as:

```
$ prog >file 2>&1
```

What is the final content of file? (Assume writes are "atomic".)

a) either "Childt" or "Parent"
b) either "Child" or "Parent"
c) either "ChildParent" or "ParentChild"

Directories



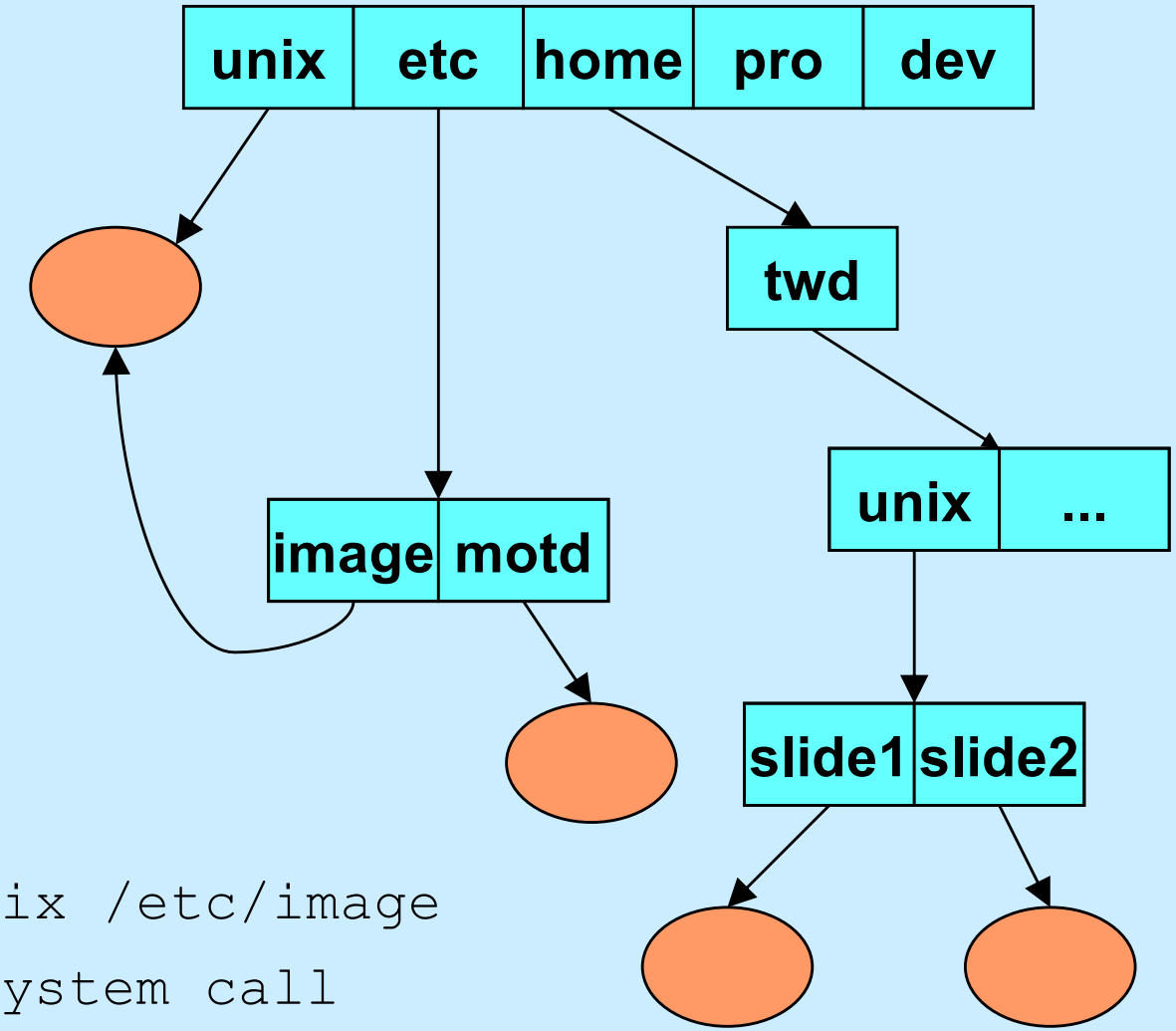
Directory Representation

Component Name	Inode Number

directory entry

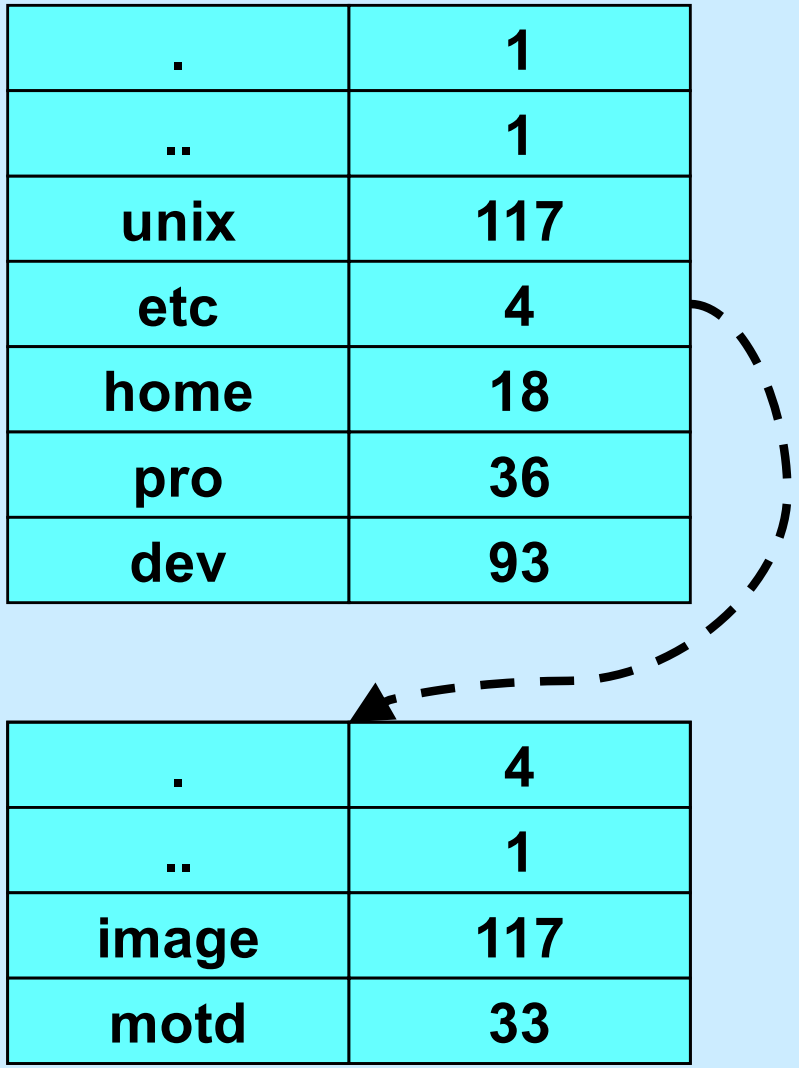
.	1
..	1
unix	117
etc	4
home	18
pro	36
dev	93

Hard Links

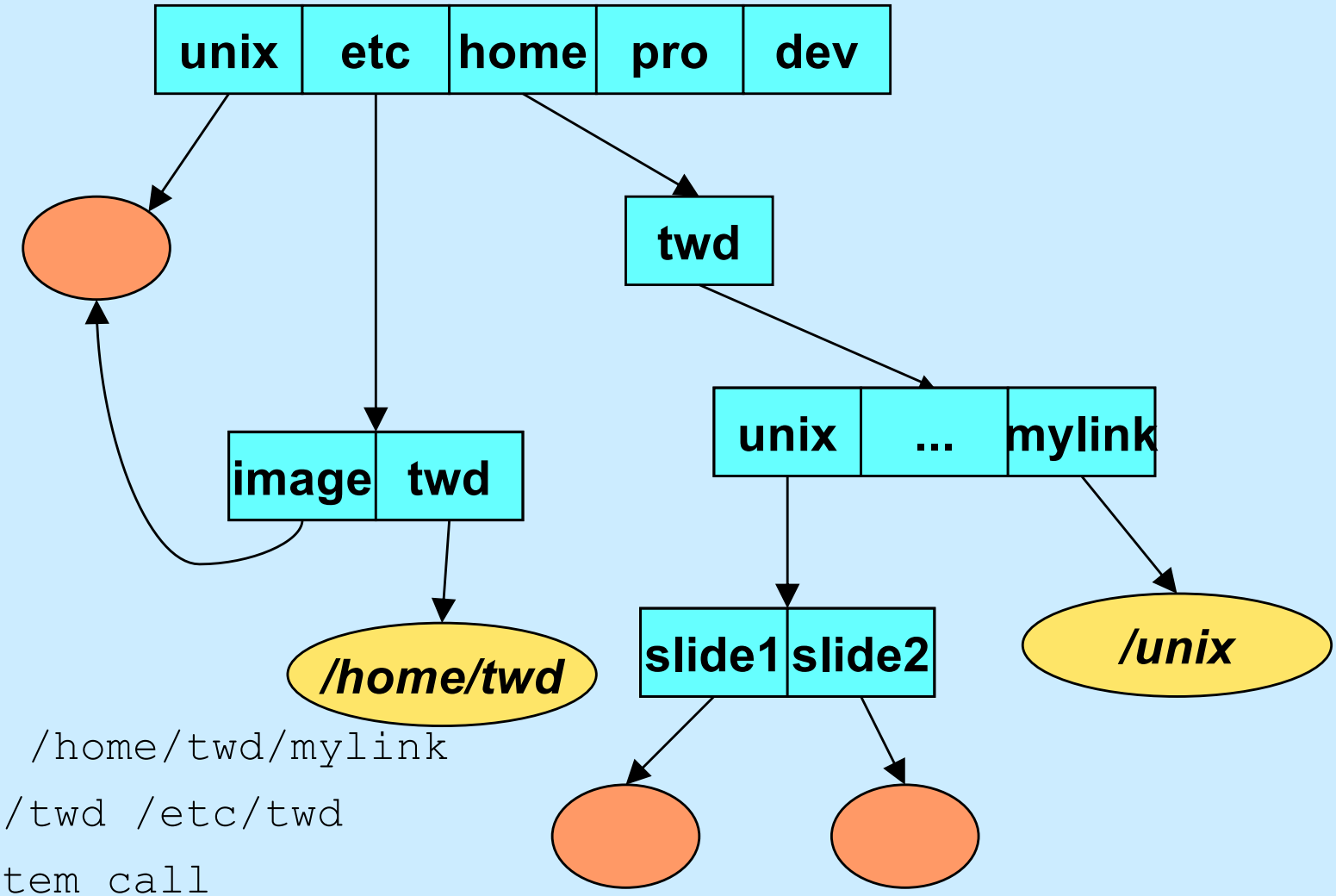



```
$ ln /unix /etc/image
# link system call
```

Directory Representation



Symbolic Links




```
% ln -s /unix /home/twd/mylink
% ln -s /home/twd /etc/twd
# 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?

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 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 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: tom, trina

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

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

./B:
total 2
-r--rw-rw-  1 tom    adm      446 Dec 17 13:34 x
-rw----rw-  1 trina  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)`

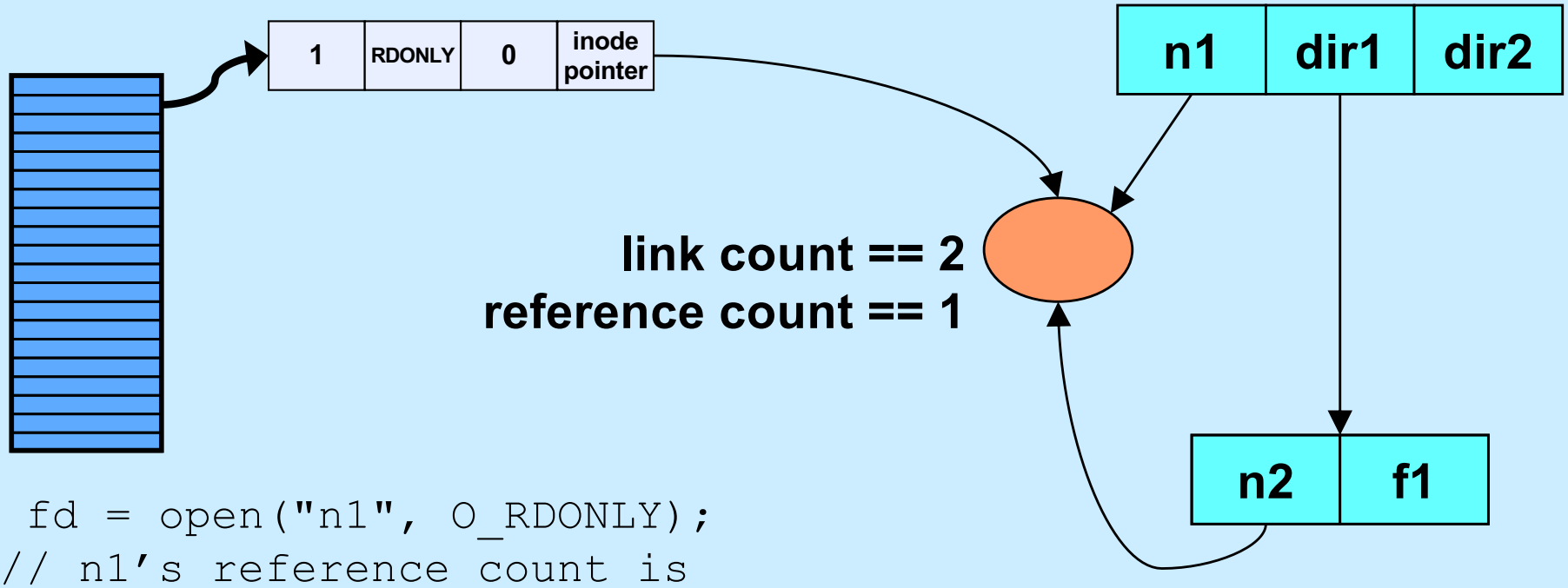
Umask

- 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

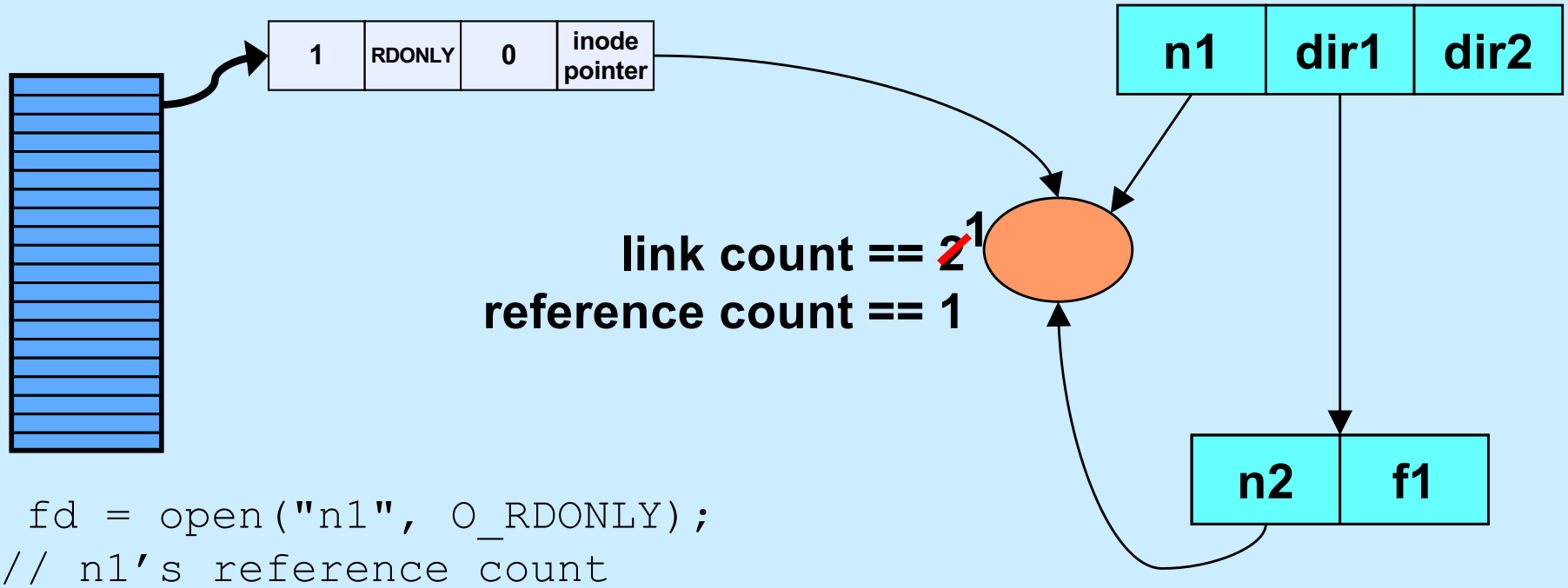
- **Use either *open* or *creat***
  - `open(const char *pathname, int flags, mode_t mode)`
    - » **flags must include O_CREAT**
  - `creat(const char *pathname, mode_t mode)`
    - » **open is preferred**
- **The *mode* parameter helps specify the permissions of the newly created file**
  - **permissions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

Link and Reference Counts



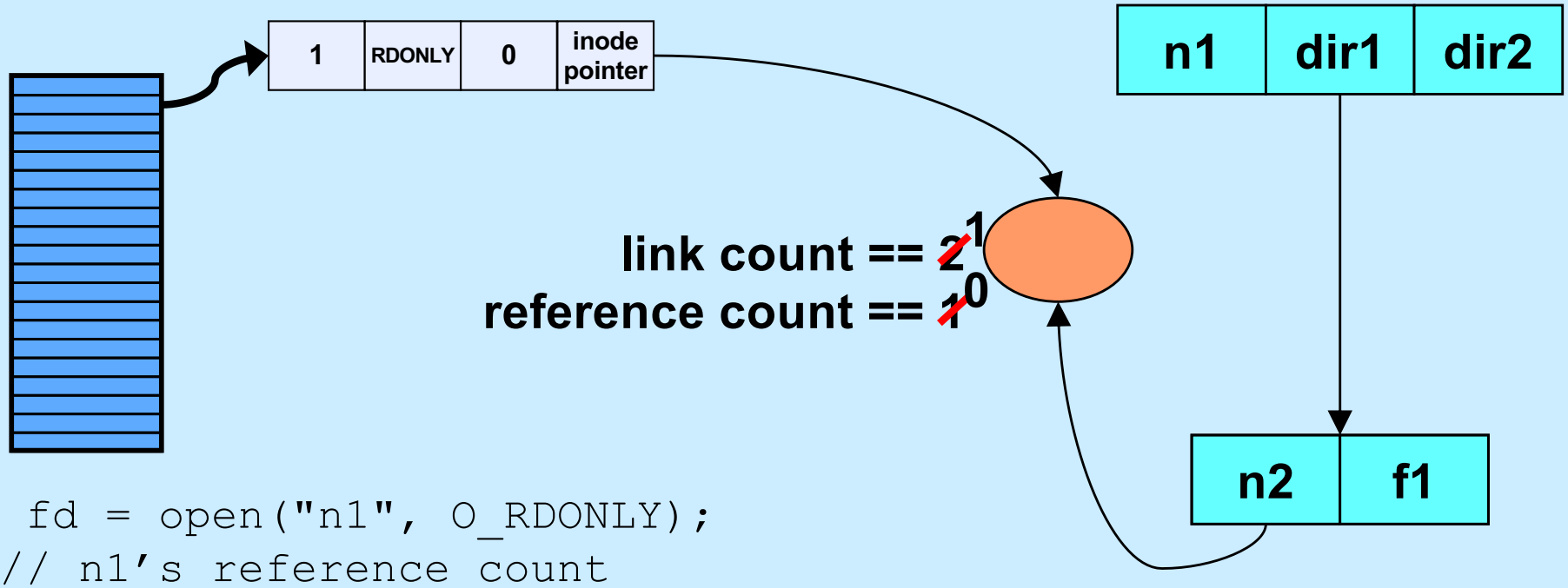

```
int fd = open("n1", O_RDONLY);
   // n1's reference count is
   // incremented by 1
```

Link and Reference Counts



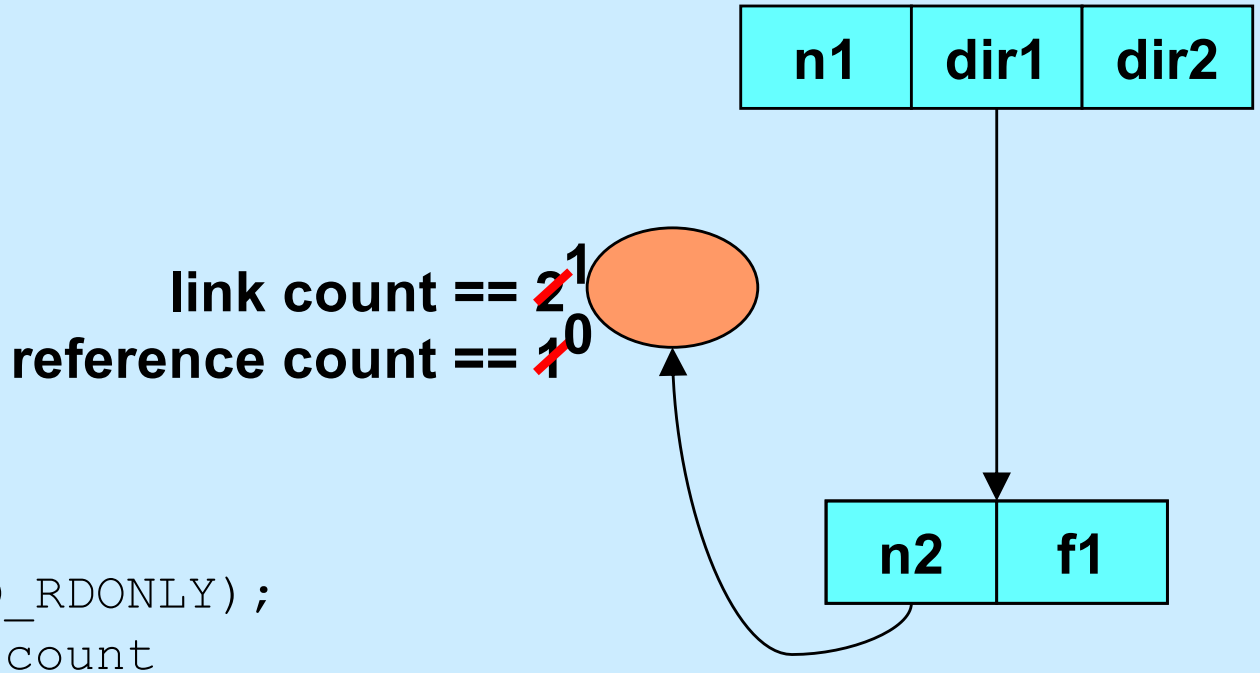

```
int fd = open("n1", O_RDONLY);
    // n1's reference count
    // incremented by 1

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

Link and Reference Counts



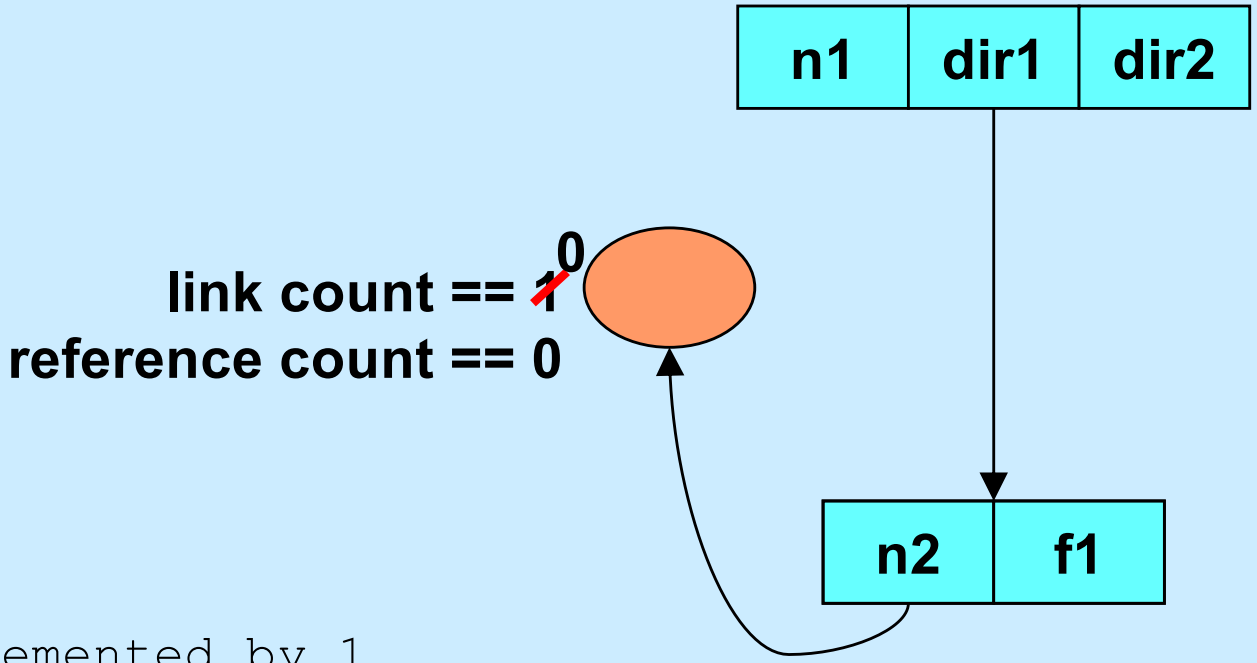

```
int fd = open("n1", O_RDONLY);
   // n1's reference count
   // incremented by 1

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

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

Link and Reference Counts




```
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 3

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

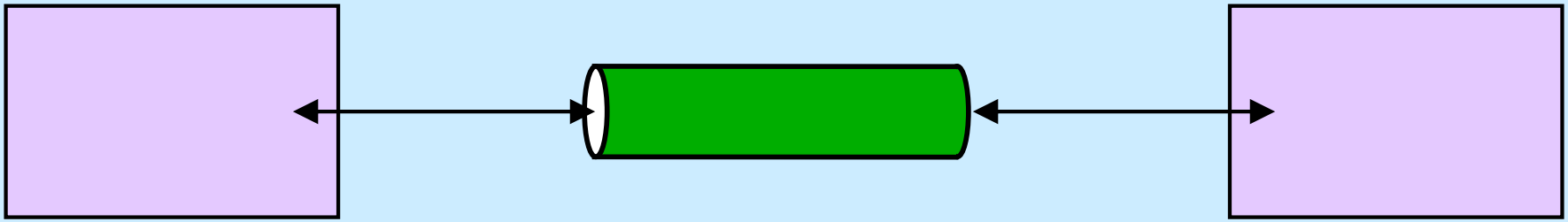
Assume that *PutStuffInFile* writes to the given file, and *GetStuffFromFile* reads from the file.

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

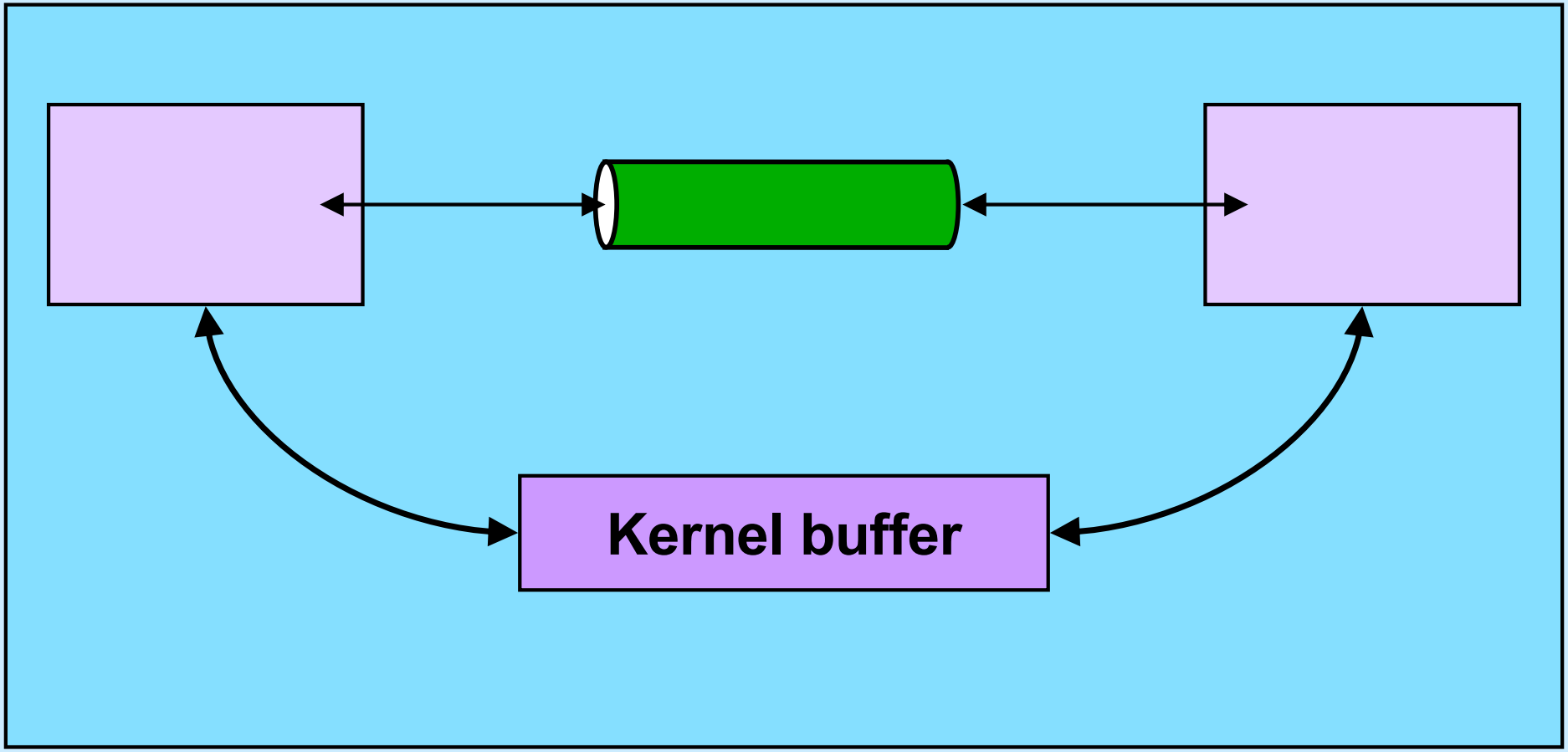
b) Because the file is used after the unlink call, it won't be deleted

c) The file will be deleted when the program terminates

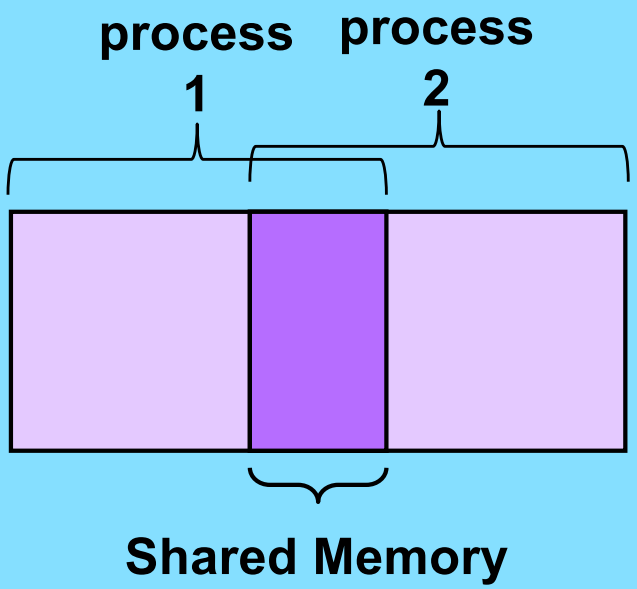
Interprocess Communication (IPC): Pipes



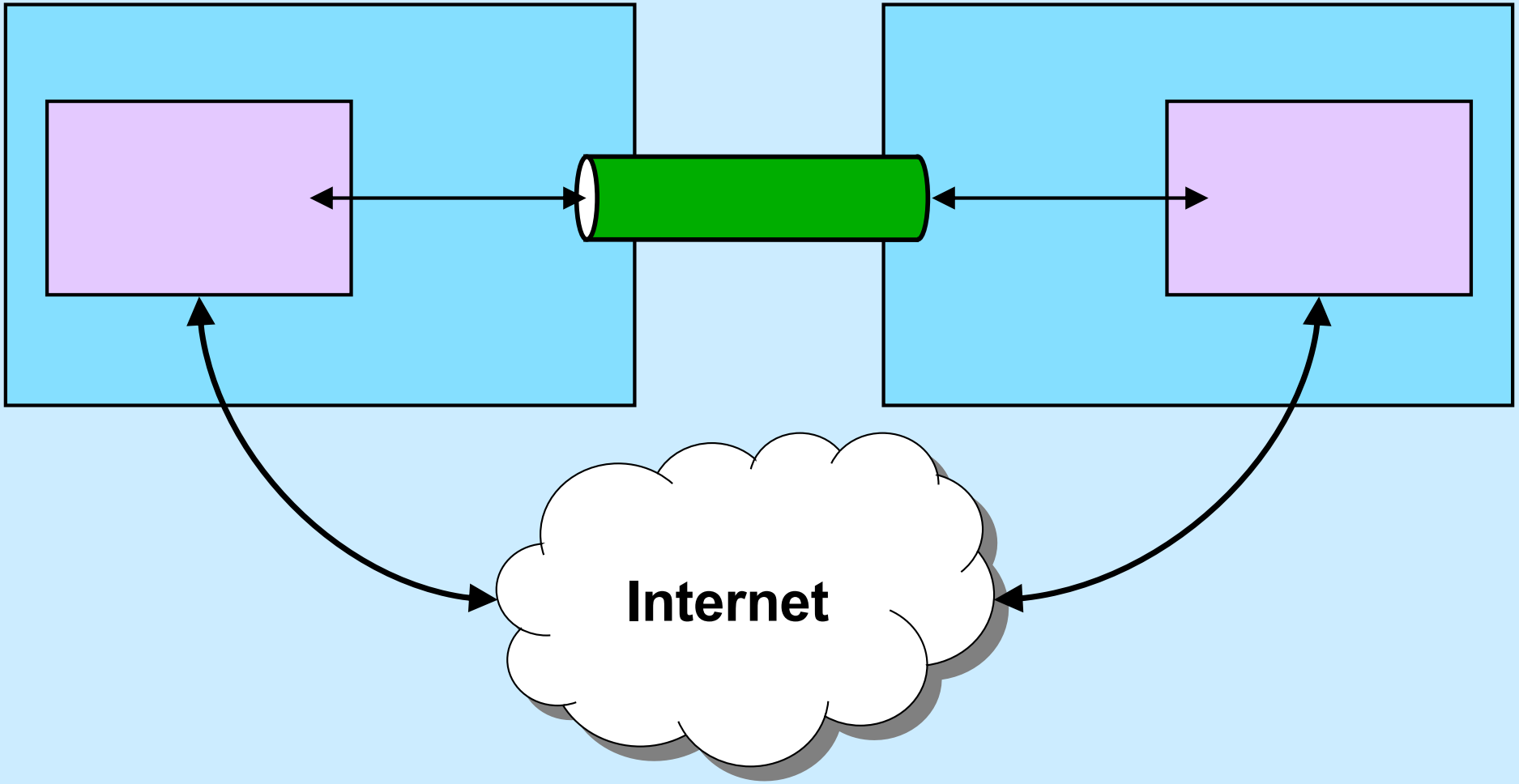
Interprocess Communication: Same Machine I



Interprocess Communication: Same Machine II

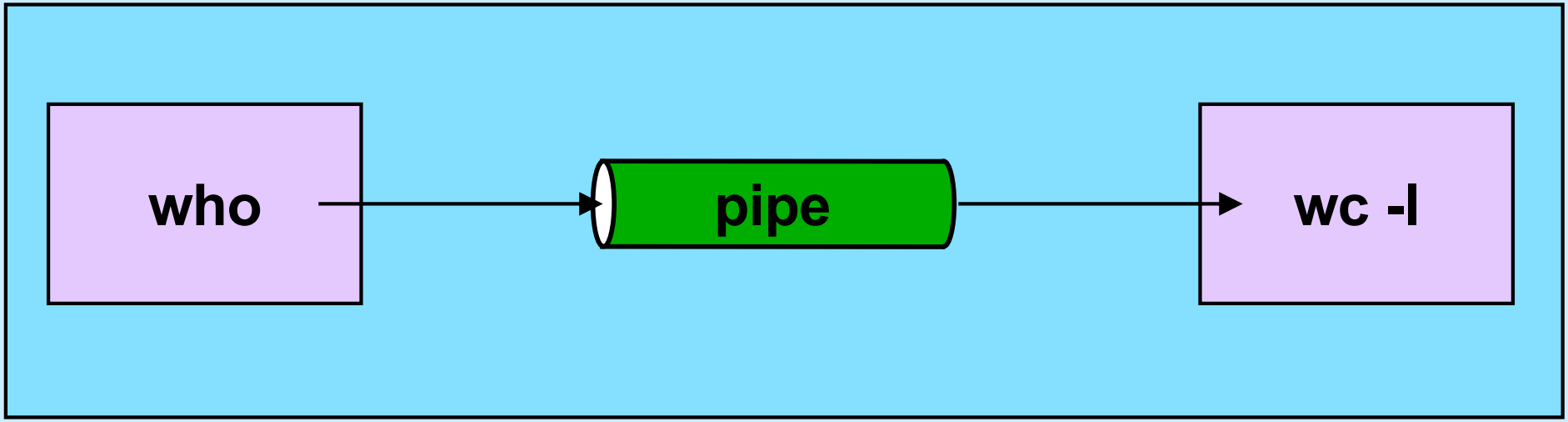


Interprocess Communication: Different Machines



Pipes

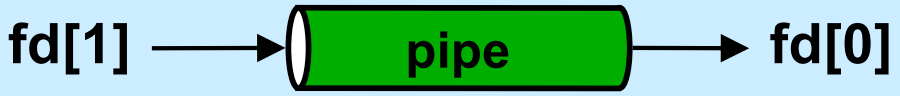
```
$cslab2e who | wc -l
```



Intramachine IPC

```
$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]);
// …
```



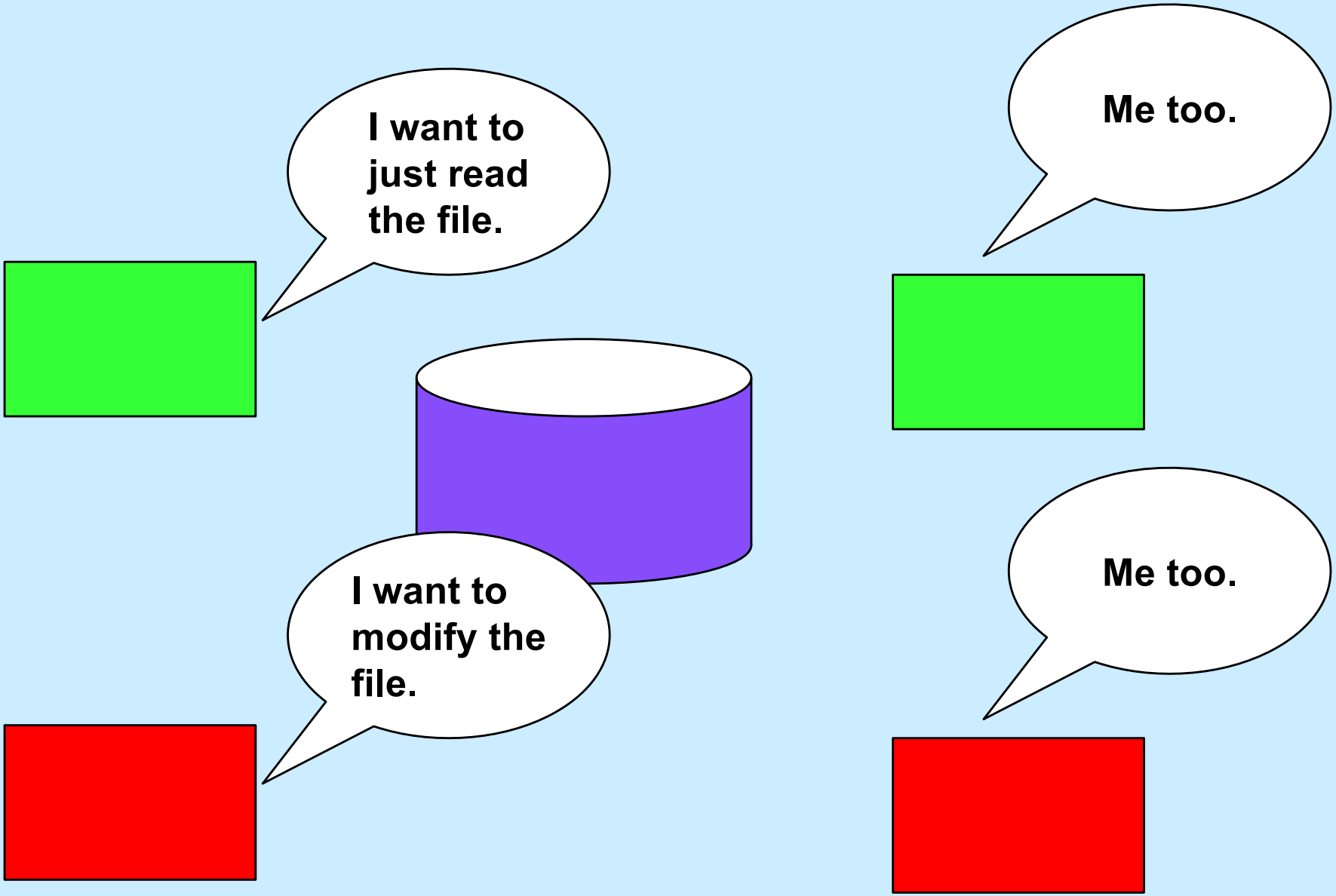
Sharing Files

- You're doing a project with a partner
- You code it as one 15,000-line file
  - the first 7,500 lines are yours
  - the second 7,500 lines are your partner's
- You edit the file, changing 6,000 lines
  - it's now 5am
- Your partner completes her changes at 5:01am
- At 5:02am you look at the file
  - your partner's changes are there
  - yours are not

Lessons

- Never work with a partner
- Use more than one file
- Read up on git
- Use an editor and file system that support file locking

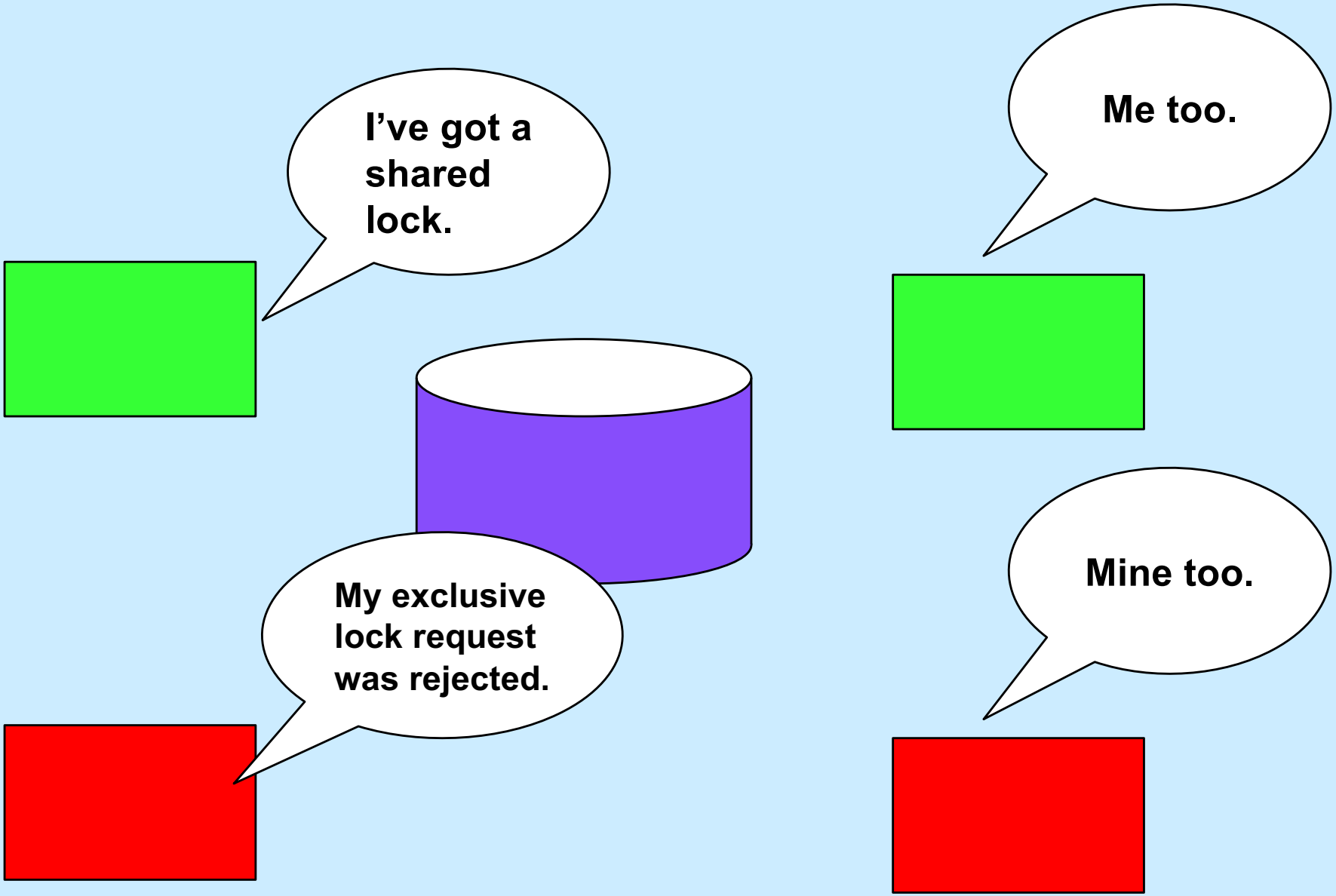
What We Want ...



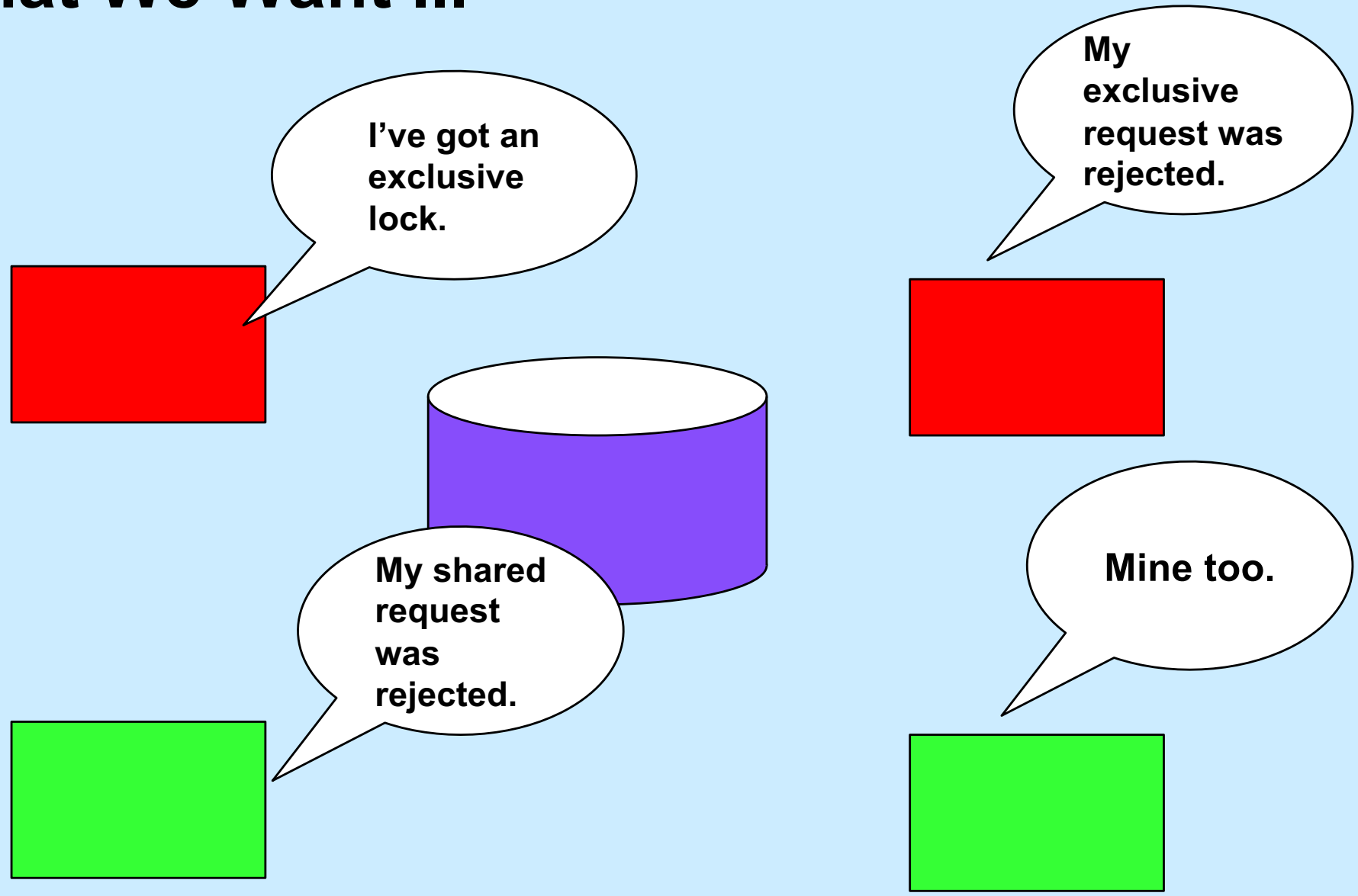
Types of Locks

- Shared (readers) locks
  - any number may have them at same time
  - may not be held when an exclusive lock is held
- Exclusive (writers) locks
  - only one at a time
  - may not be held when a shared lock is held

What We Want ...



What We Want ...



Locking Files

- **Early Unix didn't support file locking**
- **How did people survive?**
  - `open("file.lck", O_RDWR|O_CREAT|O_EXCL, 0666);`
    - » **operation fails if *file.lck* exists, succeeds (and creates file.lck) otherwise**
    - » **requires cooperative programs**

Locking Files (continued)

- How it's done in "modern" Unix
  - "advisory locks" may be placed on files
    - » may request shared (readers) or exclusive (writers) lock
      - *fcntl* system call
    - » either succeeds or fails
    - » *open*, *read*, *write* always work, regardless of locks
    - » a lock applies to a specified range of bytes, not necessarily to the whole file
    - » requires cooperative programs
  - "mandatory locks" supported as a per-file option
    - » set along with permission bits
    - » if set, file can't be used unless process possesses appropriate locks

Locking Files (still continued)

- **How to:**

```
struct flock fl;
fl.l_type = F_RDLCK;        // read lock
// fl.l_type = F_WRLCK;     // write lock
// fl.l_type = F_UNLCK;     // unlock
fl.l_whence = SEEK_SET;     // starting where
fl.l_start = 0;             // offset
fl.l_len = 0;               // how much? (0 = whole file)
fd = open("file", O_RDWR);
if (fcntl(fd, F_SETLK, &fl) == -1)
  if ((errno == EACCES) || (errno == EAGAIN))
    // didn't get lock
  else
    // something else is wrong
else
  // got the lock!
```

Quiz 4

- **Your program currently has a shared lock on a portion of a file. It would like to "upgrade" the lock to be an exclusive lock. Would there be any problems with adding an option to *fcntl* that would allow the holder of a shared lock to wait until it's possible to upgrade to an exclusive lock, then do the upgrade?**

  a) **at least one major problem**

  b) **either no problems whatsoever or some easy-to-deal-with problems**