

CS 33

Files Part 3

The File Abstraction

- **A file is a simple array of bytes**
- **A file is made larger by writing beyond its current end**
- **Files are named by paths in a naming tree**
- **System calls on files are synchronous**
- **Files are permanent**

Naming

- **(almost) everything has a path name**
 - files
 - directories
 - devices (known as *special files*)
 - » keyboards
 - » displays
 - » disks
 - » etc.

I/O System Calls

- **int** file_descriptor = open(pathname, mode [, permissions])
- **int** close(file_descriptor)
- **ssize_t** count = read(file_descriptor, buffer_address, buffer_size)
- **ssize_t** count = write(file_descriptor, buffer_address, buffer_size)
- **off_t** position = lseek(file_descriptor, offset, whence)

Standard File Descriptors

```
int main( ) {
    char buf[BUFSIZE];
    int n;
    const char *note = "Write failed\n";

    while ((n = read(0, buf, sizeof(buf))) > 0)
        if (write(1, buf, n) != n) {
            write(2, note, strlen(note));
            exit(1);
        }
    return(0);
}
```

Standard I/O Library

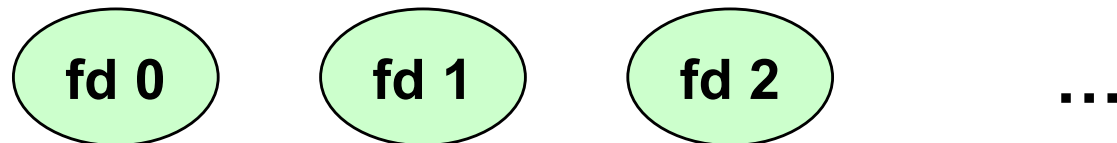
Formatting



Buffering



Syscalls



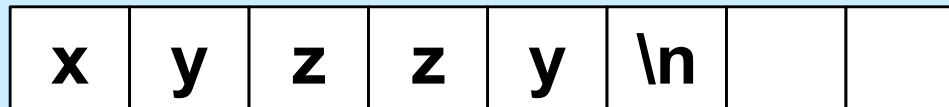
Standard I/O

```
FILE *stdin;           // declared in stdio.h
FILE *stdout;          // declared in stdio.h
FILE *stderr;          // declared in stdio.h

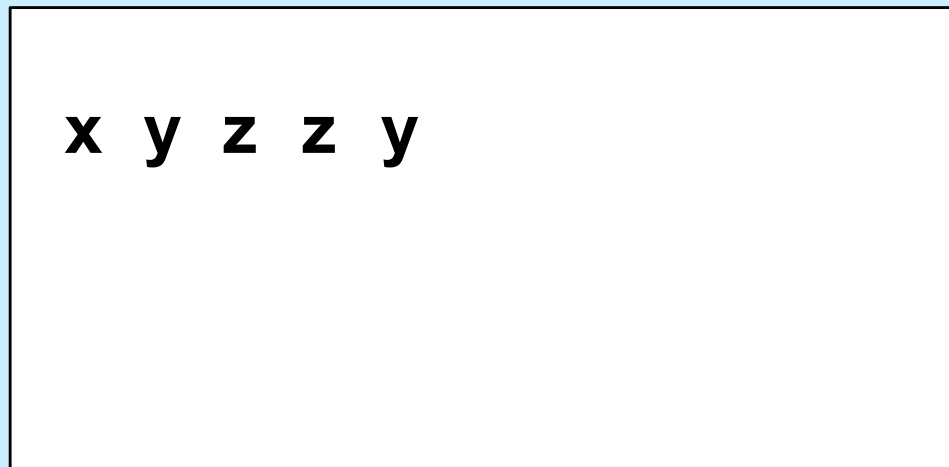
scanf("%d", &in);      // read via f.d. 0
printf("%d\n", in);    // write via f.d. 1
fprintf(stderr, "there was an error\n");
// write via f.d. 2
```

Buffered Output

```
printf("xy");  
printf("zz");  
printf("y\n");
```



buffer



display

Unbuffered Output

```
fprintf(stderr, "xy");  
fprintf(stderr, "zz");  
fprintf(stderr, "y\n");
```



x y z z y

display

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

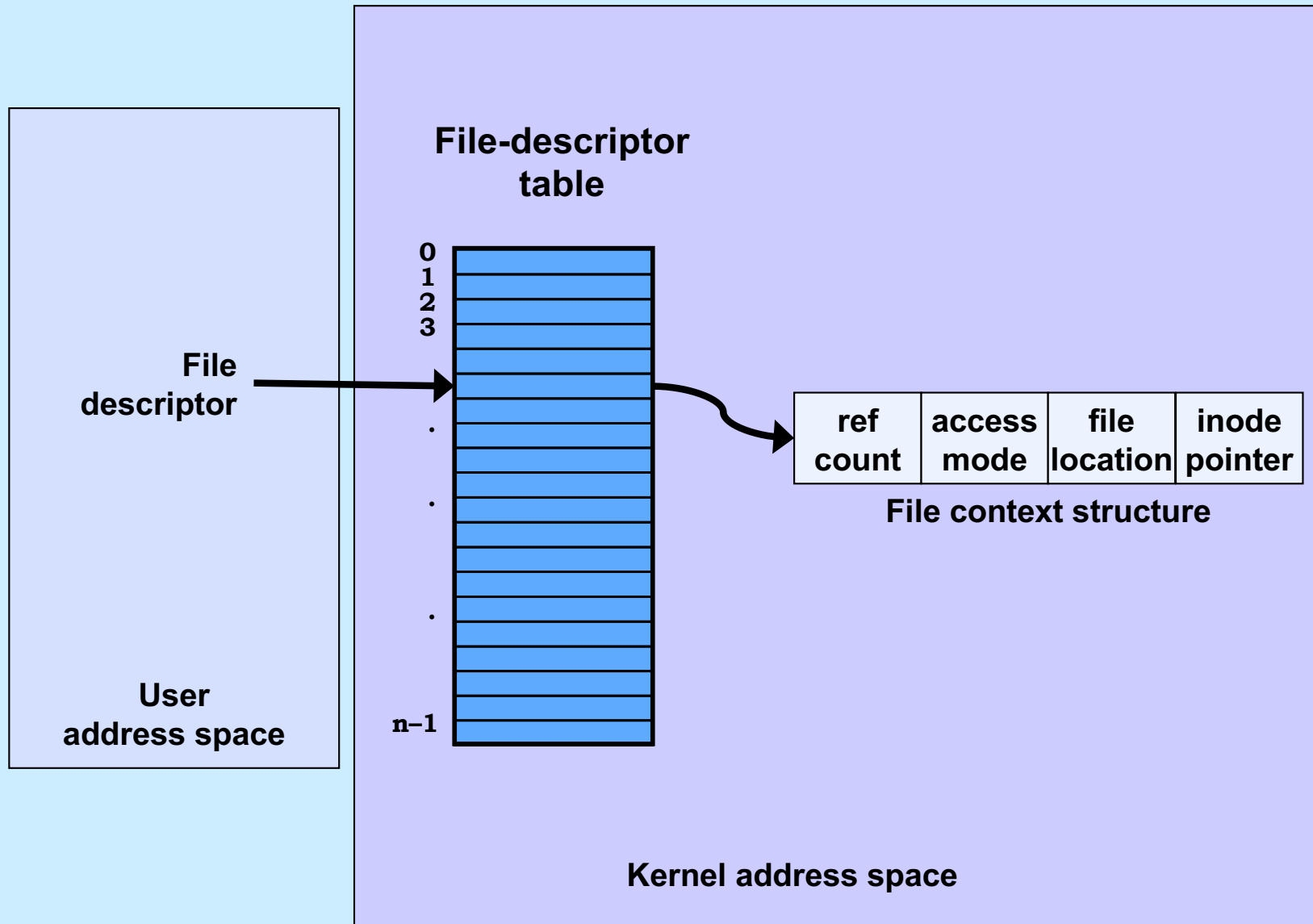
Redirecting Stdout in C

```
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);
}

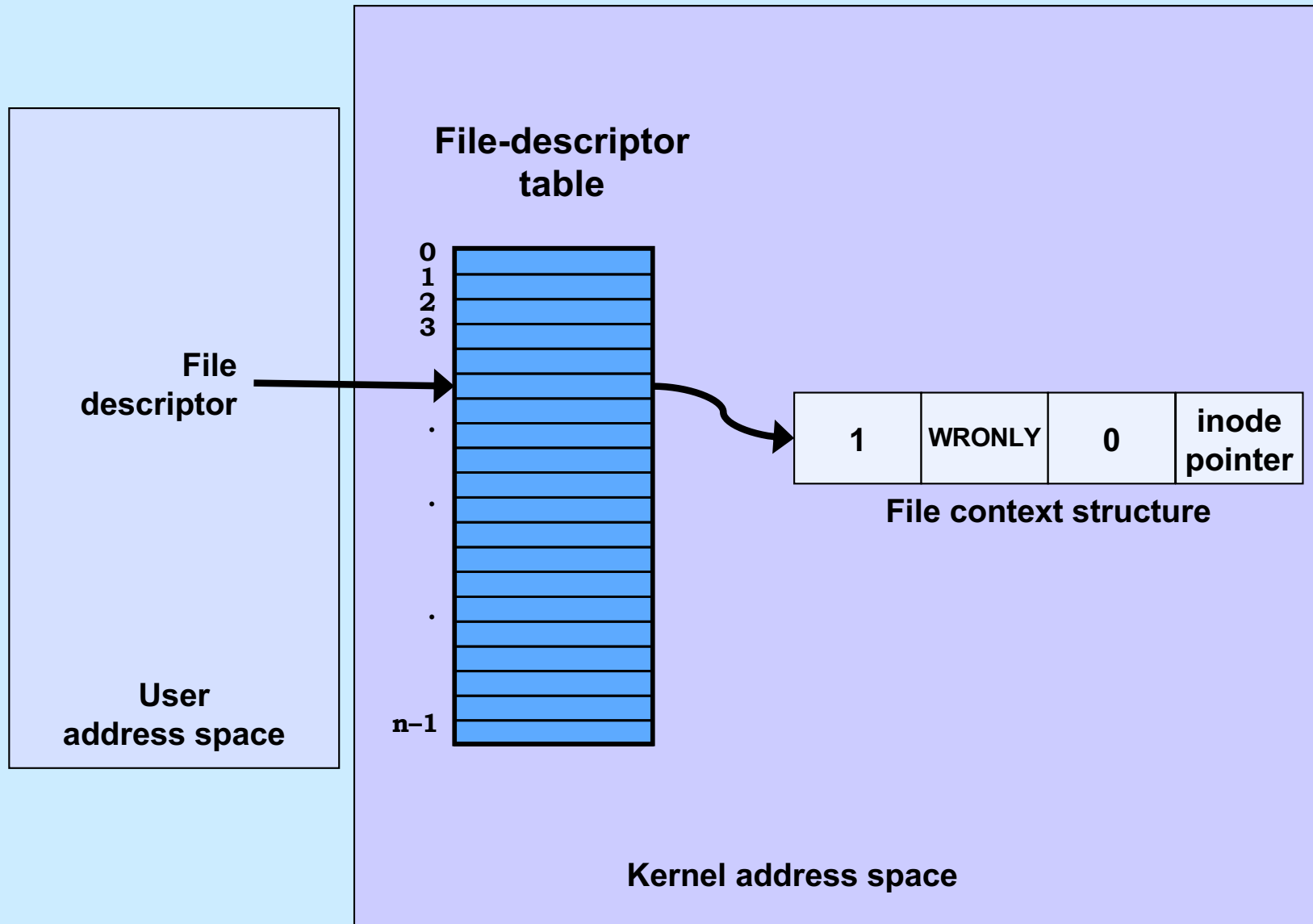
/* parent continues here */

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

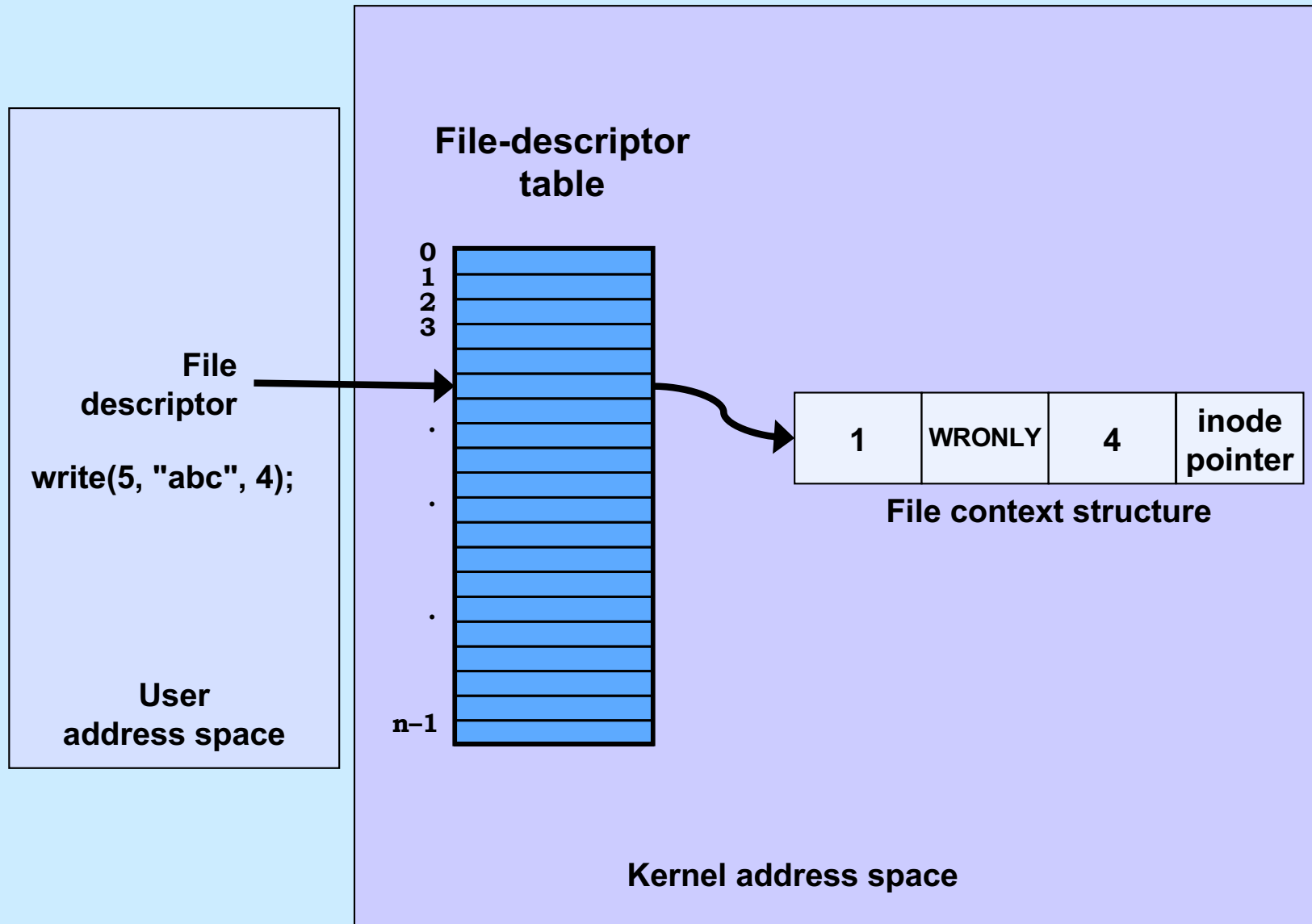
File-Descriptor Table



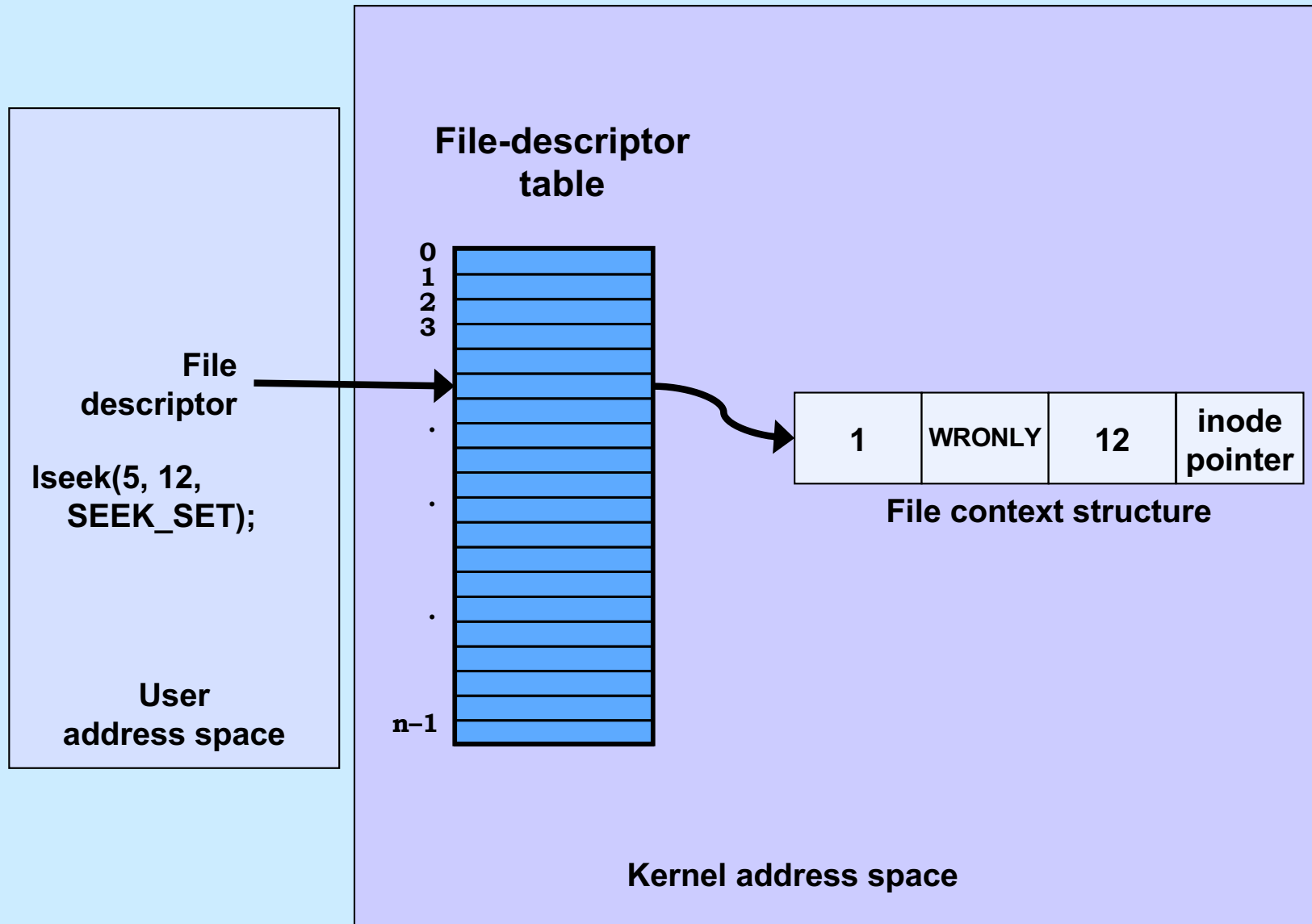
File Location



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

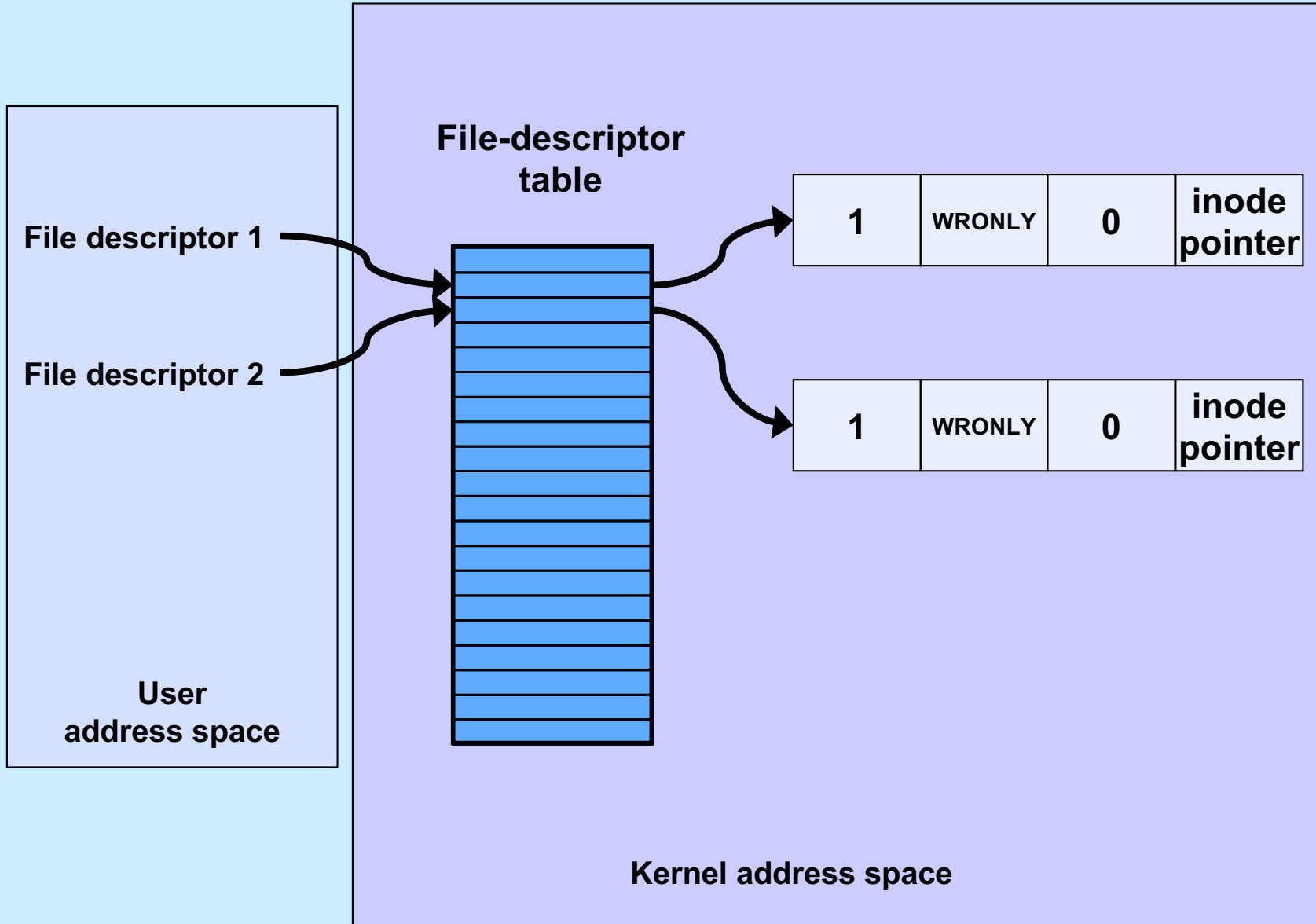
```
if (fork() == 0) {
    /* set up file descriptors 1 and 2 in the child process */
    close(1);
    close(2);
    if (open("/home/twd/Output", O_WRONLY) == -1) {
        exit(1);
    }
    if (open("/home/twd/Output", O_WRONLY) == -1) {
        exit(1);
    }
    char *argv[] = {"echon", 2, NULL};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

From the Shell ...

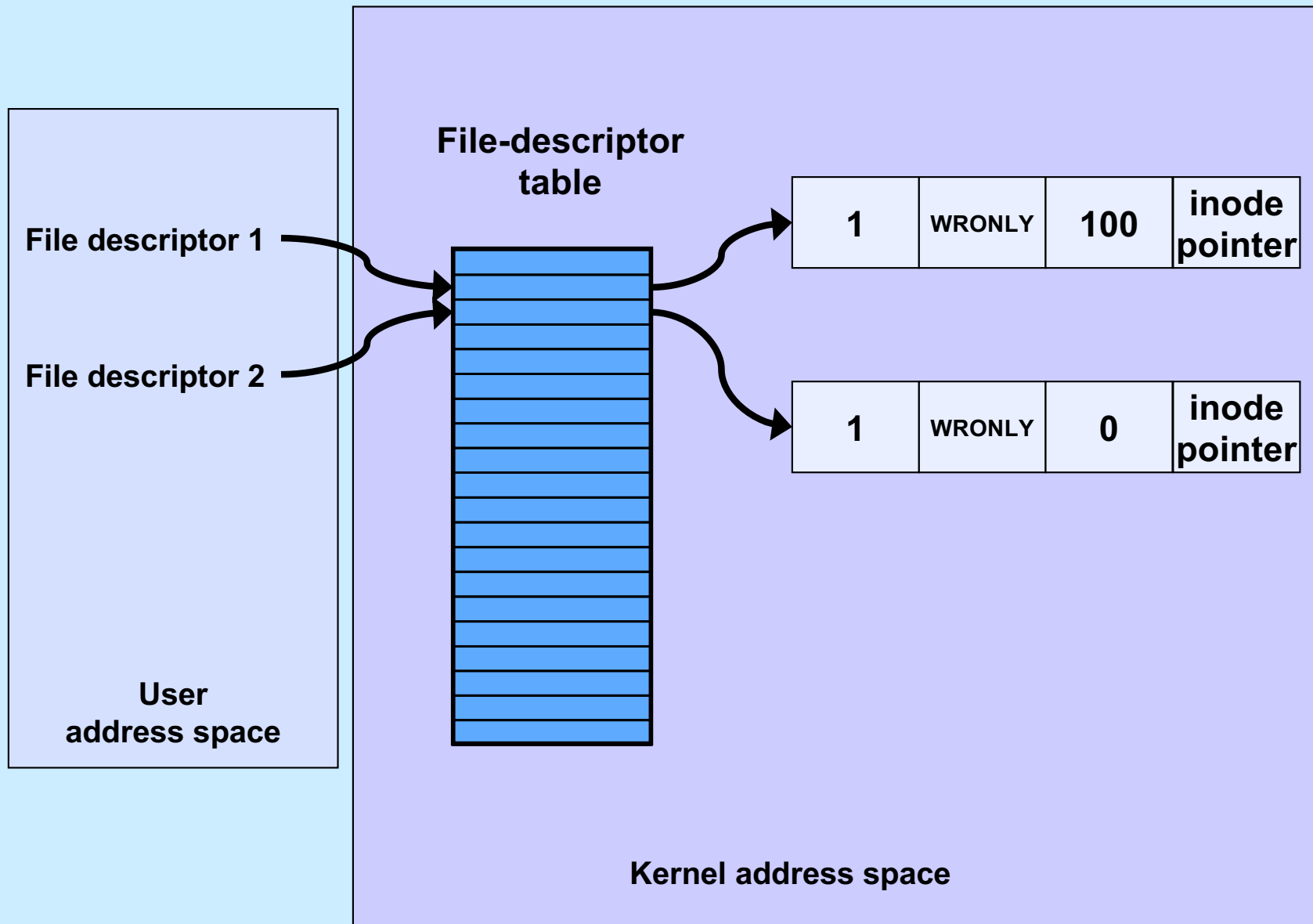
```
$ echo 1 >Output 2>Output
```

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

Redirected Output



Redirected Output After Write



Quiz 1

- **Suppose we run**

```
$ echo 3 >Output 2>Output
```

- **The input line is**

```
X
```

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

a) X\nX\nreps too large, reduced to 2\n

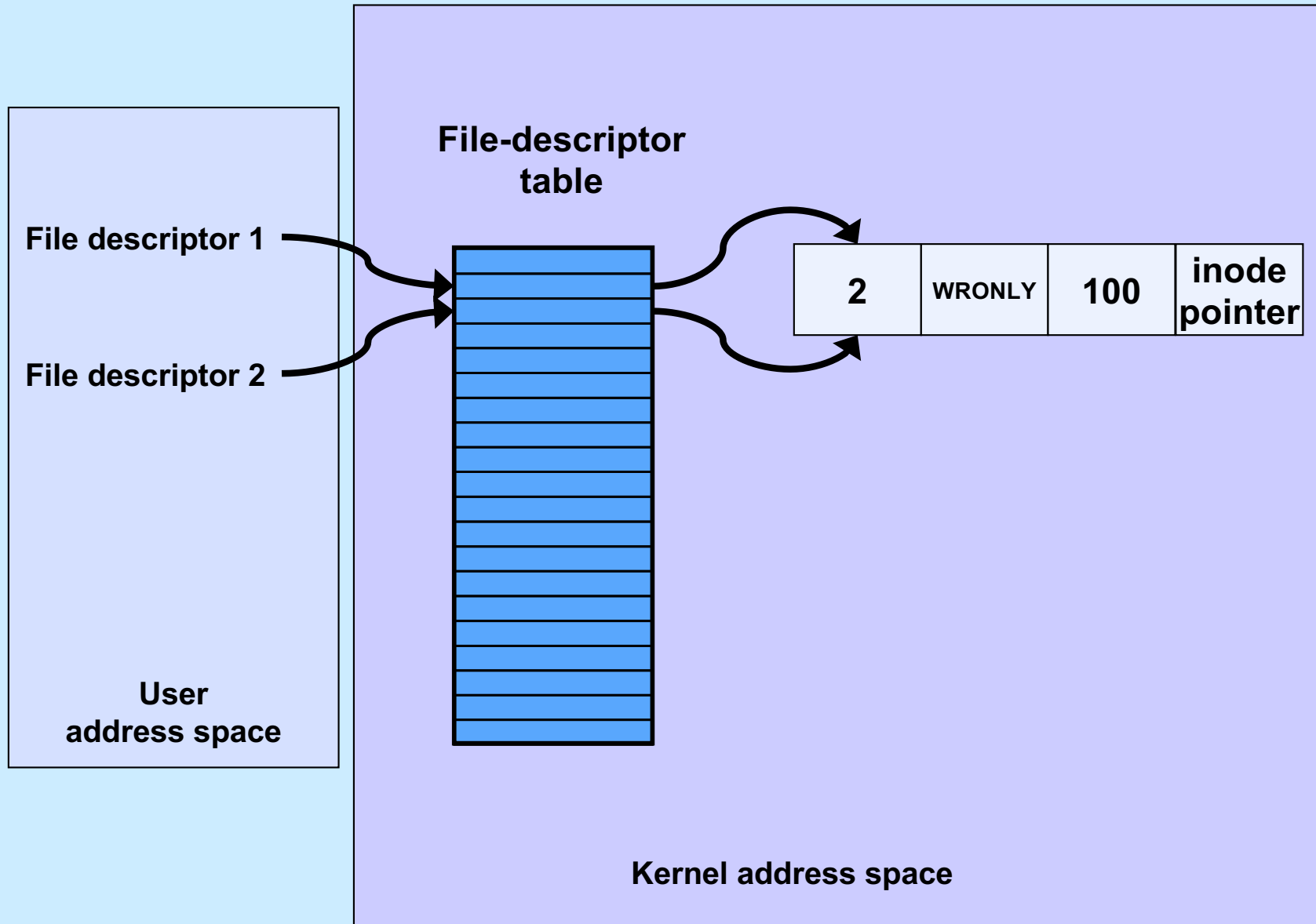
b) X\nX\n too large, reduced to 2\n

c) reps too large, reduced to 2\nX\nX\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
```

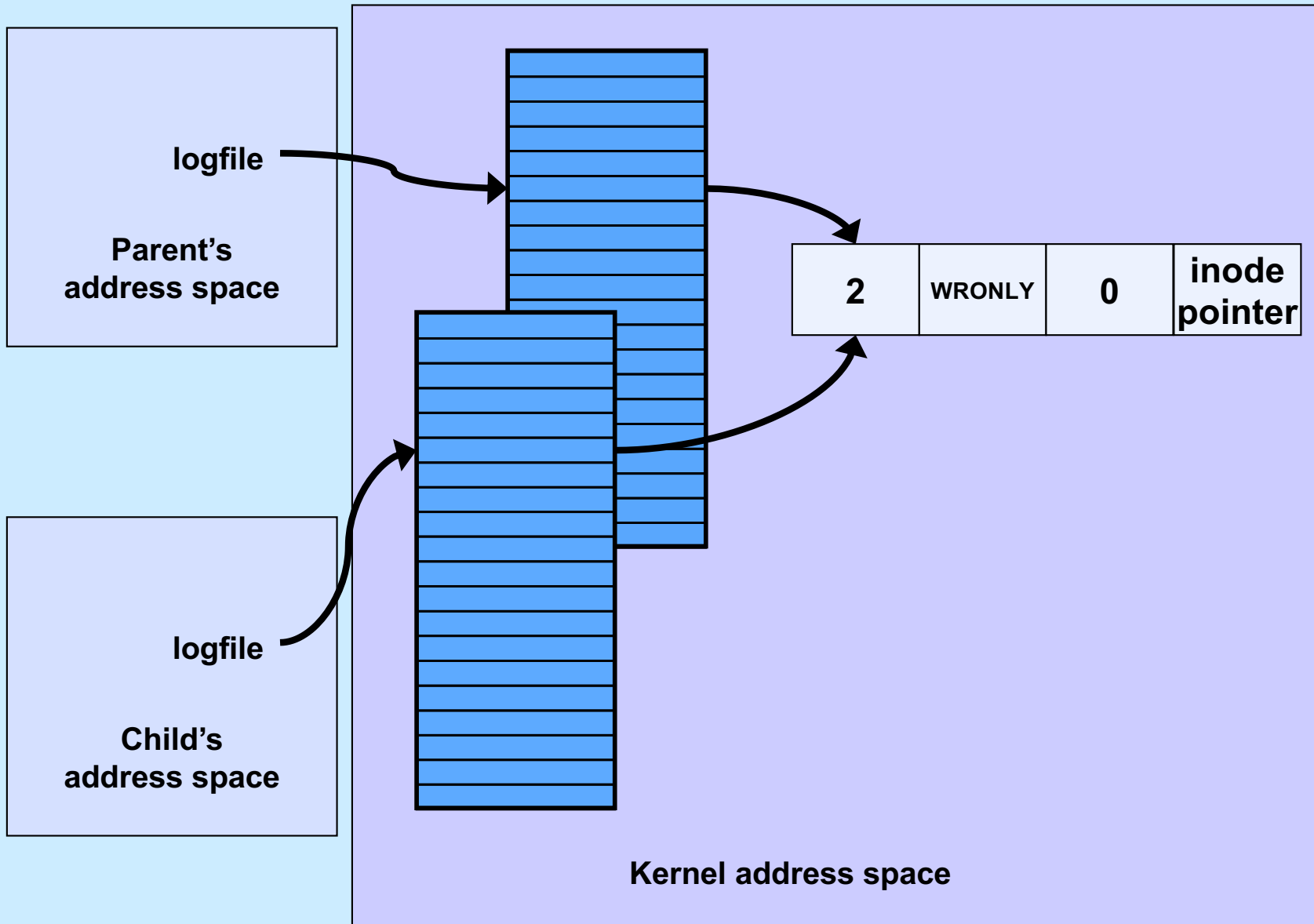
Fork and File Descriptors

```
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");  
}
```

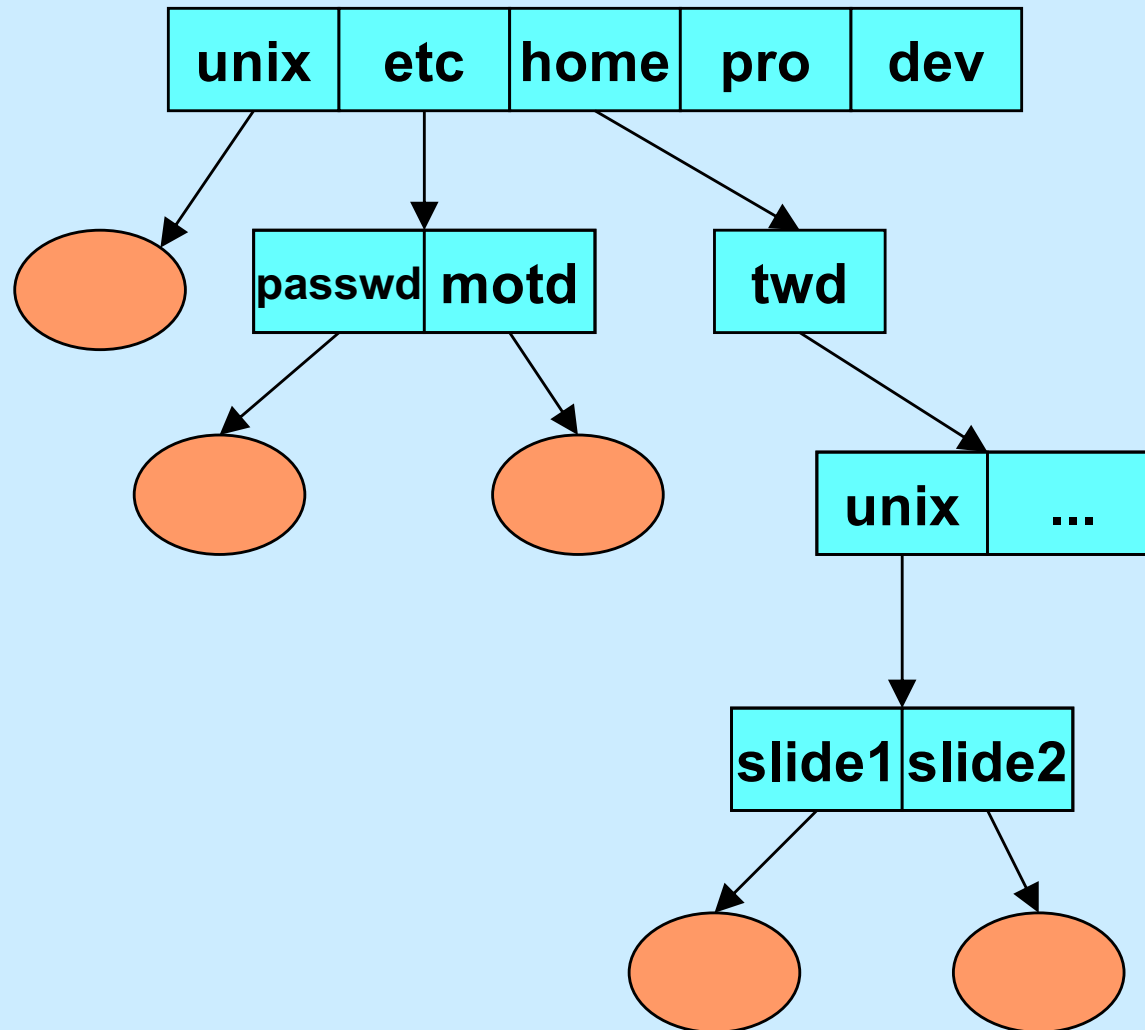
Suppose the program is run as:

```
$ 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



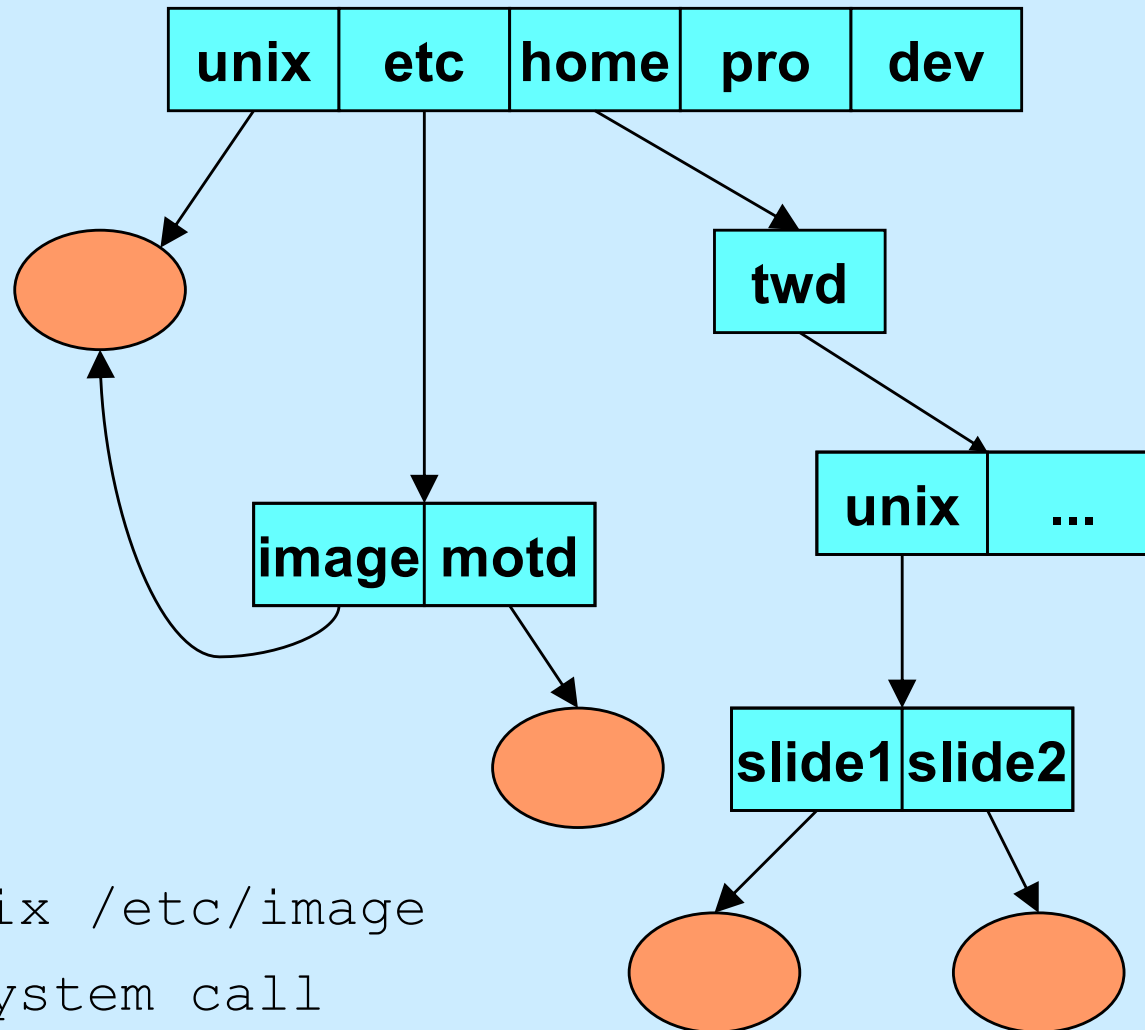
Directory Representation

Component Name	Inode Number
----------------	--------------

directory entry

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

Hard Links

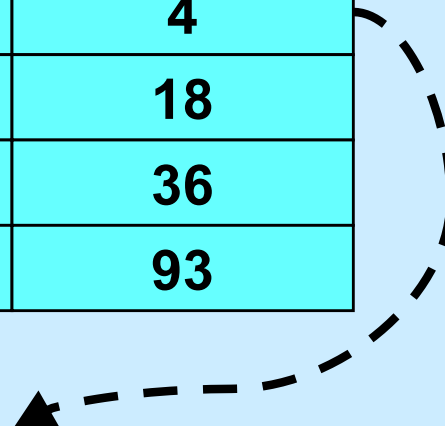


```
$ ln /unix /etc/image  
# link system call
```

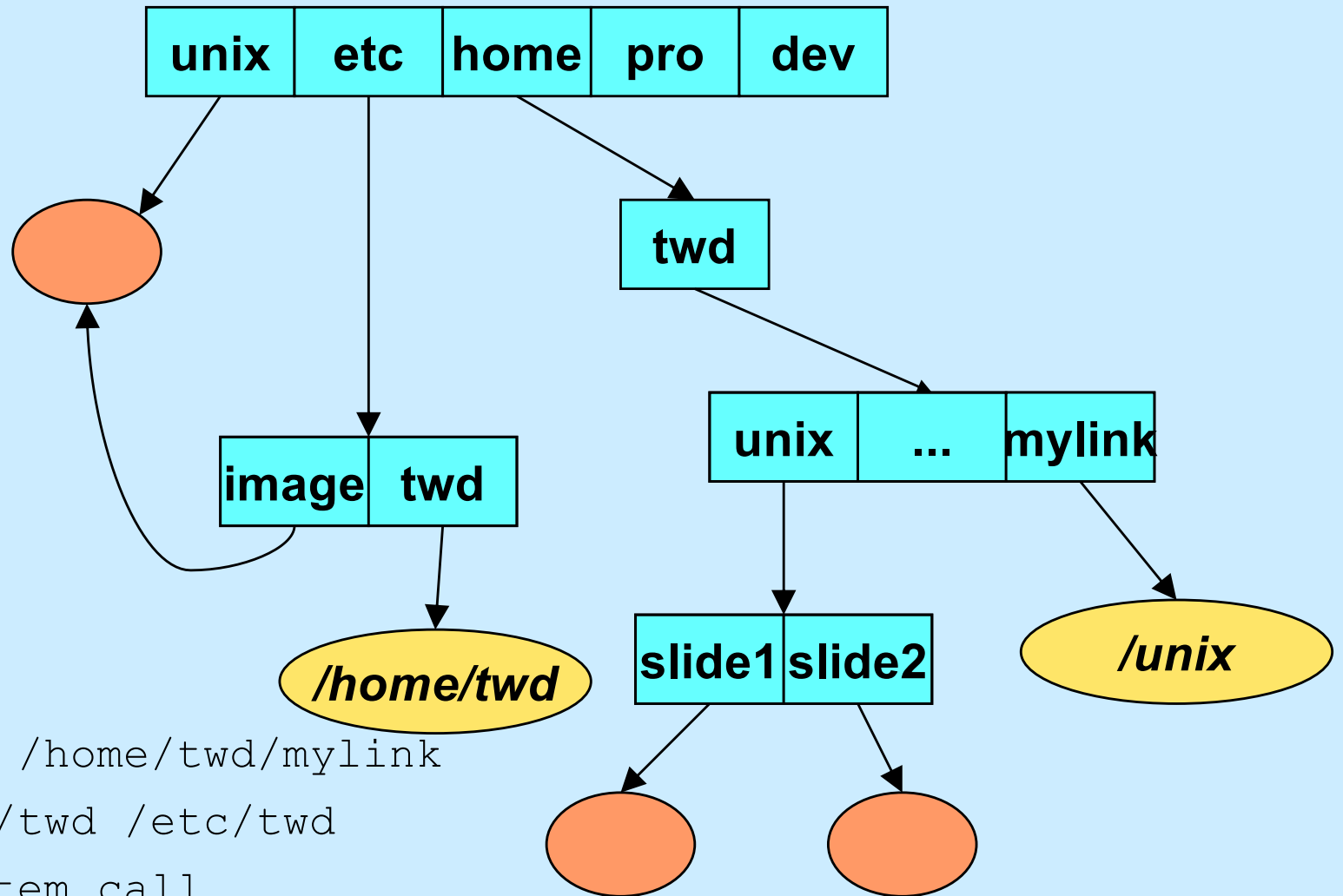
Directory Representation

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

.	4
..	1
image	117
motd	33



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 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 = $0777 \& \sim(027) = 0750$
 - » editors: permissions = $0666 \& \sim(027) = 0640$
 - set with *umask* system call or (usually) shell command

Quiz 3

You get the following message when you attempt to execute `./program` (a file that you own):

```
bash: ./program: Permission denied
```

Your 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**