Carnegie Mellon University

Filesystems — Interface

MARCH 27, 2025

Sara McAllister
sjmcalli@cs.cmu.edu
saramcallister.github.io

Slides: saramcallister.github.io/files/2025-03-27-files.pdf

Situating this lecture – Operating Systems (ECE 344)

Date and recording	Lecture Note
1/9	Introduction
1/10	Architecture Support
1/16	Architecture Support
1/17	Architecture Support
1/23	Processes
1/24	Processes
1/30	Threads
1/31	Threads
2/6	Threads, Synchronization (I)
2/7	Synchronization (I), Synchronization (II)
2/13	Synchronization (II)
2/14	Synchronization (II)
2/28	Memory Management (I)
3/6	Memory Management (I)
3/7	Memory Management (I)
3/14	Memory Management (II): Paging
3/20	Memory Management (II): Paging
3/21	Memory Management (III): Replacement

What we've covered so far:

Processes

Threads

Storage Devices

Hard disk drives

Cycle-head-sector (CHS) addressing

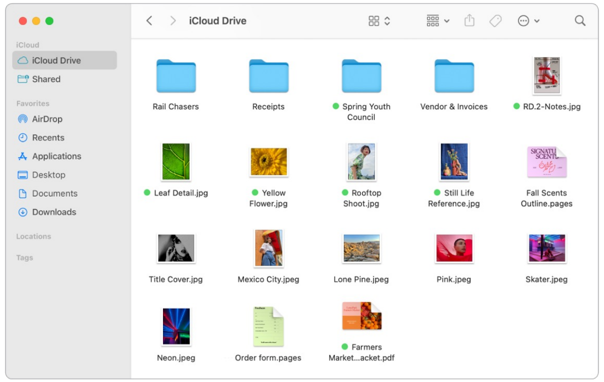
Logical block addresses (LBAs)

0	1	2	3	4

How can we abstract storage devices?

How do we abstract persistent storage?

Filesystems: a system for organizing, managing, and accessing *files* and *directories* stored on persistent media



Local filesystems (UNIX)

Enable

File sharing platforms

Why do we need filesystems?

- Durability across restarts and crashes
- Naming and organization
- Sharing data between processes and users

What makes filesystems challenging?

- Crash recovery (next lecture)
- **Performance + concurrency**
- **Sharing + security**

What we're covering today: Filesystem interface

What are files?

`open` and fds

Directories and ownership

What are files?

A file is a sequence of bytes, logically grouped together

- Allow users/ processes to access all files the same way

In UNIX, “Everything is a file”

- `/dev/sda2` (disk partition)
- `/dev/tty2` (terminal)
- `/proc/$$/status` (current process status)

Typical file metadata

Name - 14 characters? 8.3? 255?

Identifier - "file number" (usually internal)

Size - two meanings (next lecture)

Protection - Who c

Time, date, last mo

```
struct stat {
    dev_t     st_dev;          /* ID of device containing file */
    ino_t     st_ino;          /* Inode number */
    mode_t    st_mode;         /* File type and mode */
    nlink_t   st_nlink;        /* Number of hard links */
    uid_t     st_uid;          /* User ID of owner */
    gid_t     st_gid;          /* Group ID of owner */
    dev_t     st_rdev;         /* Device ID (if special file) */
    off_t     st_size;         /* Total size, in bytes */
    blksize_t st_blksize;      /* Block size for filesystem I/O */
    blkcnt_t  st_blocks;       /* Number of 512B blocks allocated */
    struct timespec st_atim;   /* Time of last access */
    struct timespec st_mtim;   /* Time of last modification */
    struct timespec st_ctim;   /* Time of last status change */

};
```

Operations on Files

Write, Read - often via position pointer/ “cursor”

Seek - adjust position pointer for next access

Append - write at end of file (implicit synchronization)

Rename - change name of file inside directory and (maybe) move a file between 2 directories

I/O to a file

Need to read and write to a file

So how do you read and write to one?

Try 1: `read("example.txt", buffer, num_bytes);`

- What if the file is large?

I/O to a file

Need to read and write to a file

So how do you read and write to one?

Try 1: `read("example.txt", buffer, num_bytes);`

- What if the file is large?

Try 2: `read("example.txt", buffer, num_bytes, start_loc);`

- What is notably inefficient about this?
- What's the solution?

What we're covering today: Filesystem interface

What are files?

`open` and fds

Directories and ownership

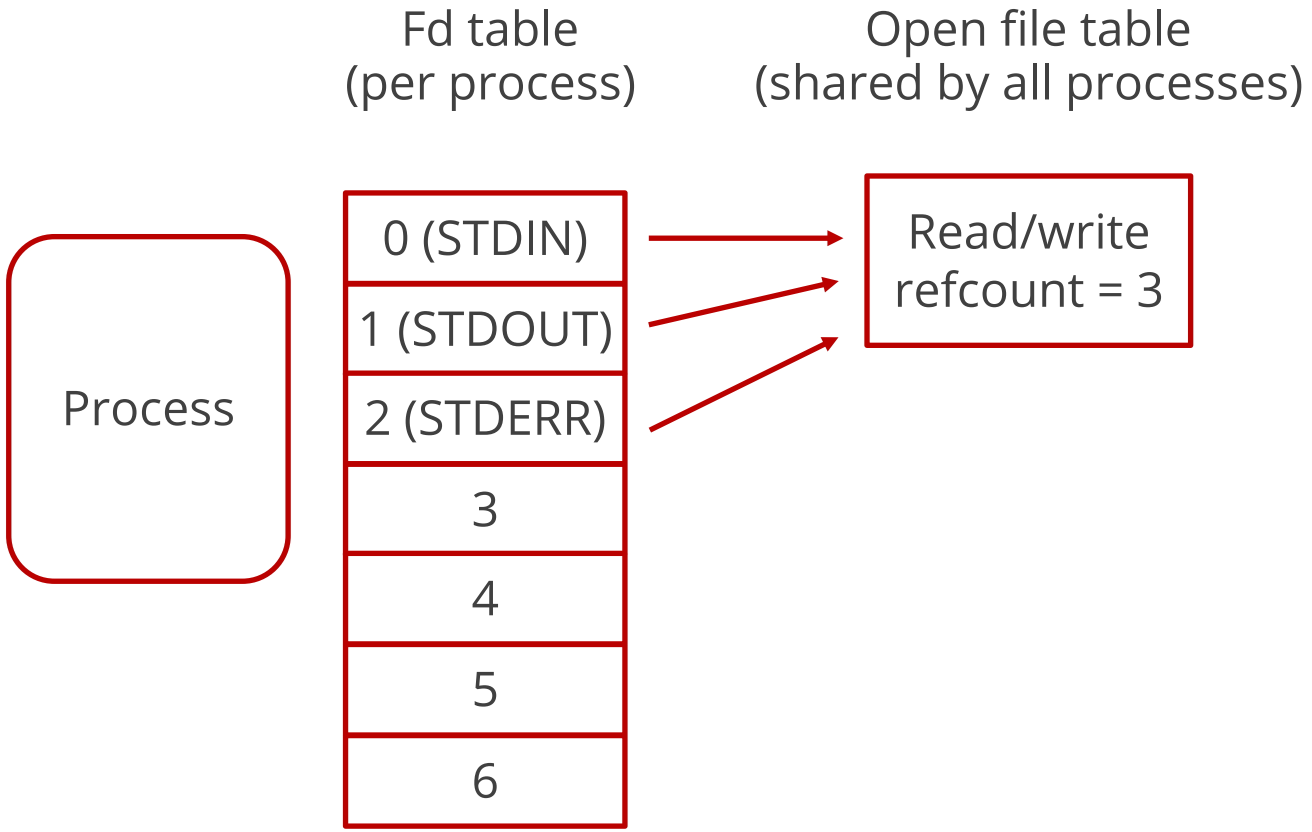
Solution: Open-file state

Add an `open` operation + "state"

"Open-file" structure (file descriptor) stores

- Filesystem / partition
- Filesystem-relative file number
- Operations allowed: eg read vs write
- Cursor position

Open files (UNIX model)



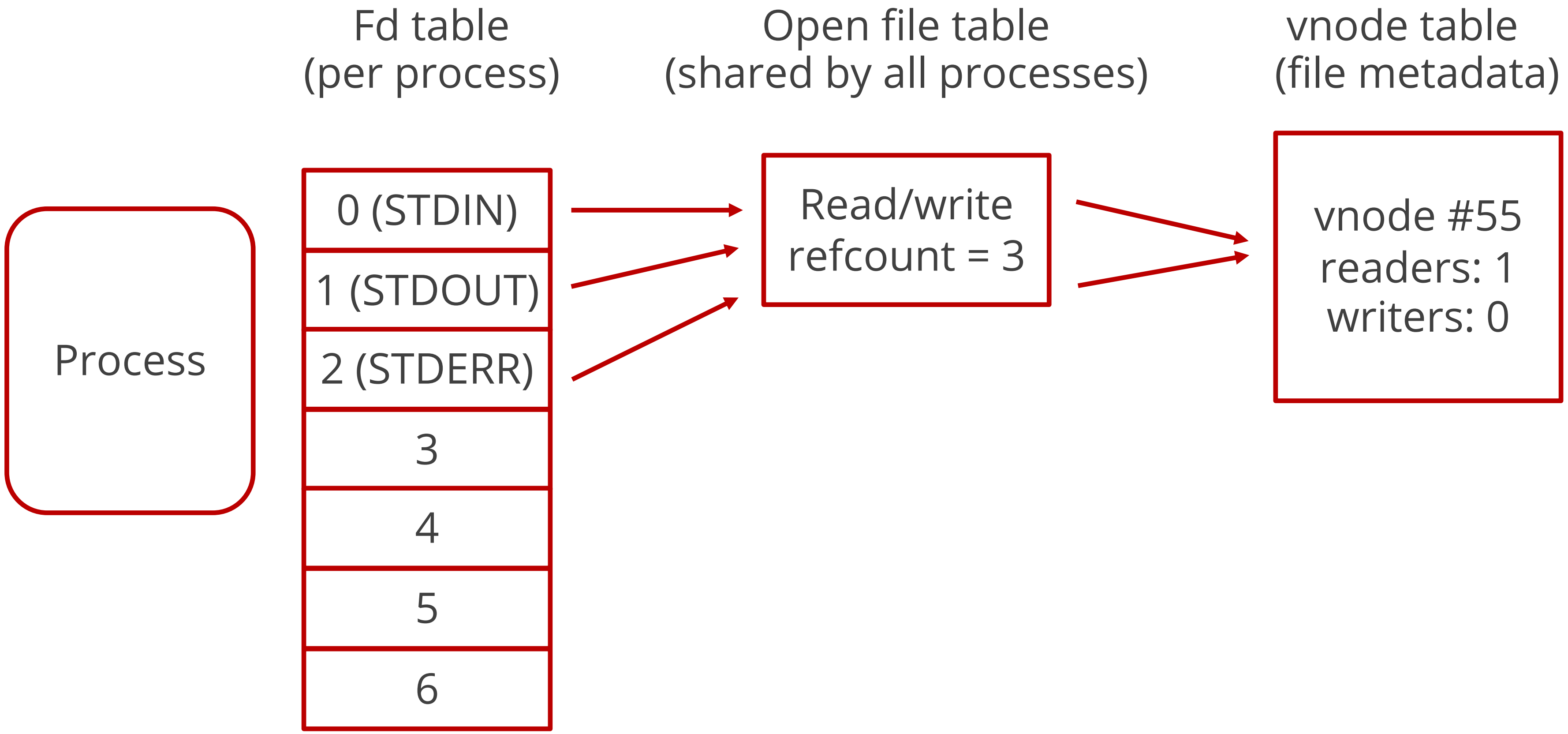
Open files (Unix model)

"In-core" file state - avoid going to disk repeatedly

- Mirror of on-disk structure (File number, size, permissions, modification time, …)
- Any other filesystem specific information

Shared when file is opened multiple times

Open files (UNIX model)



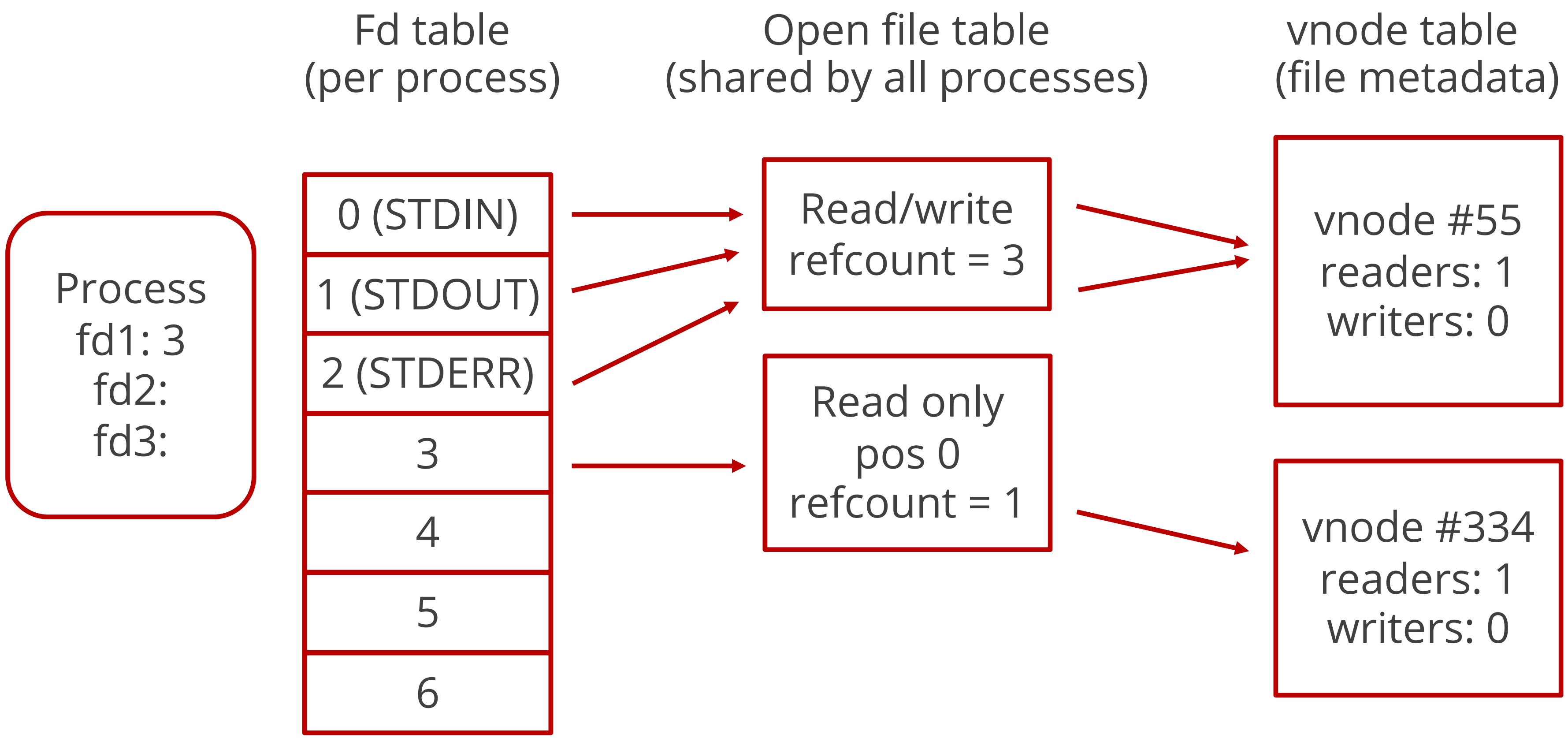
Open files - example

```
char buf[10];

int fd1 = open("foo.c", O_RDONLY, 0);
int fd2 = dup(fd1);
int fd3 = open("foo.c", O_RDONLY, 0);
```

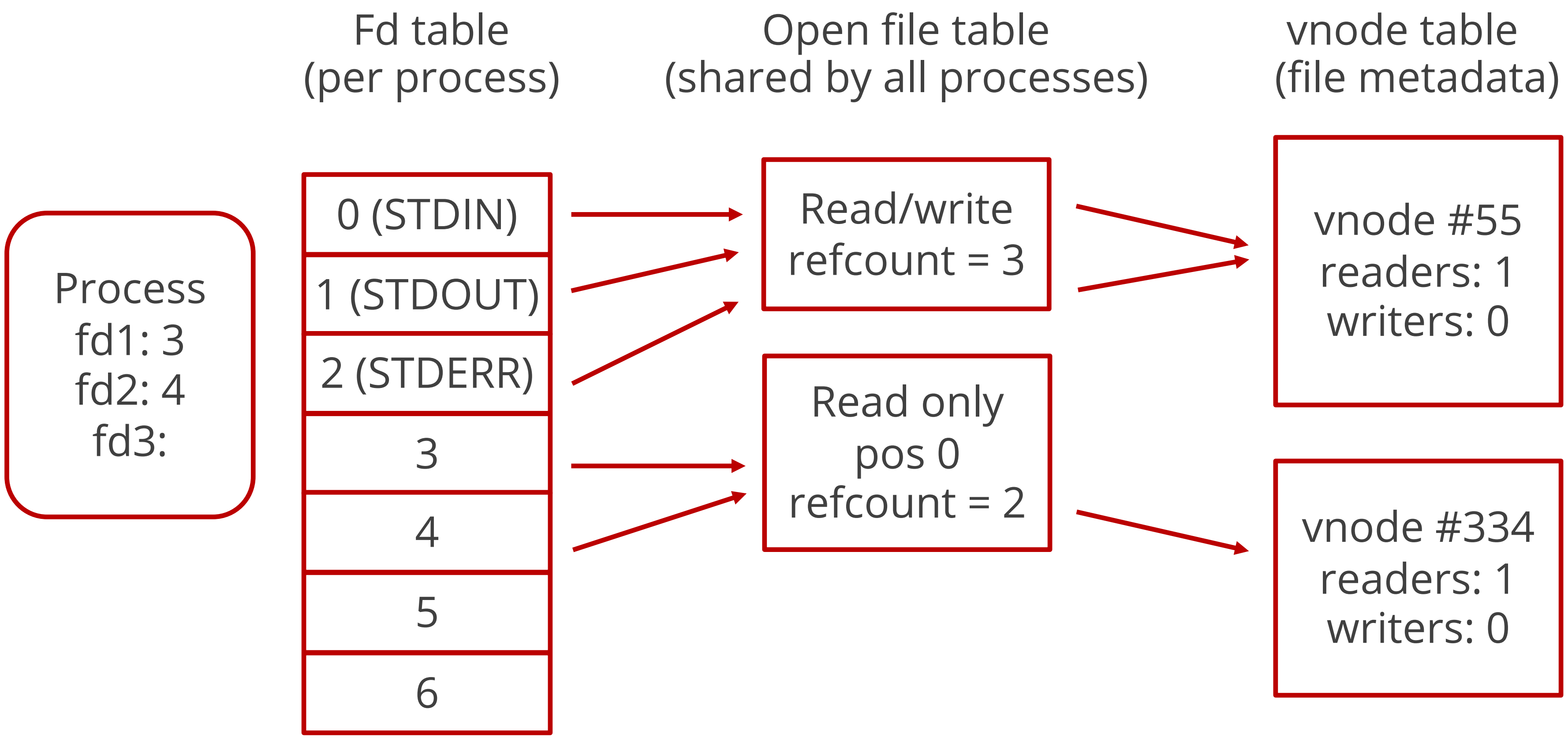
What's wrong with this example code?

“Open File” vs “In-Core File”



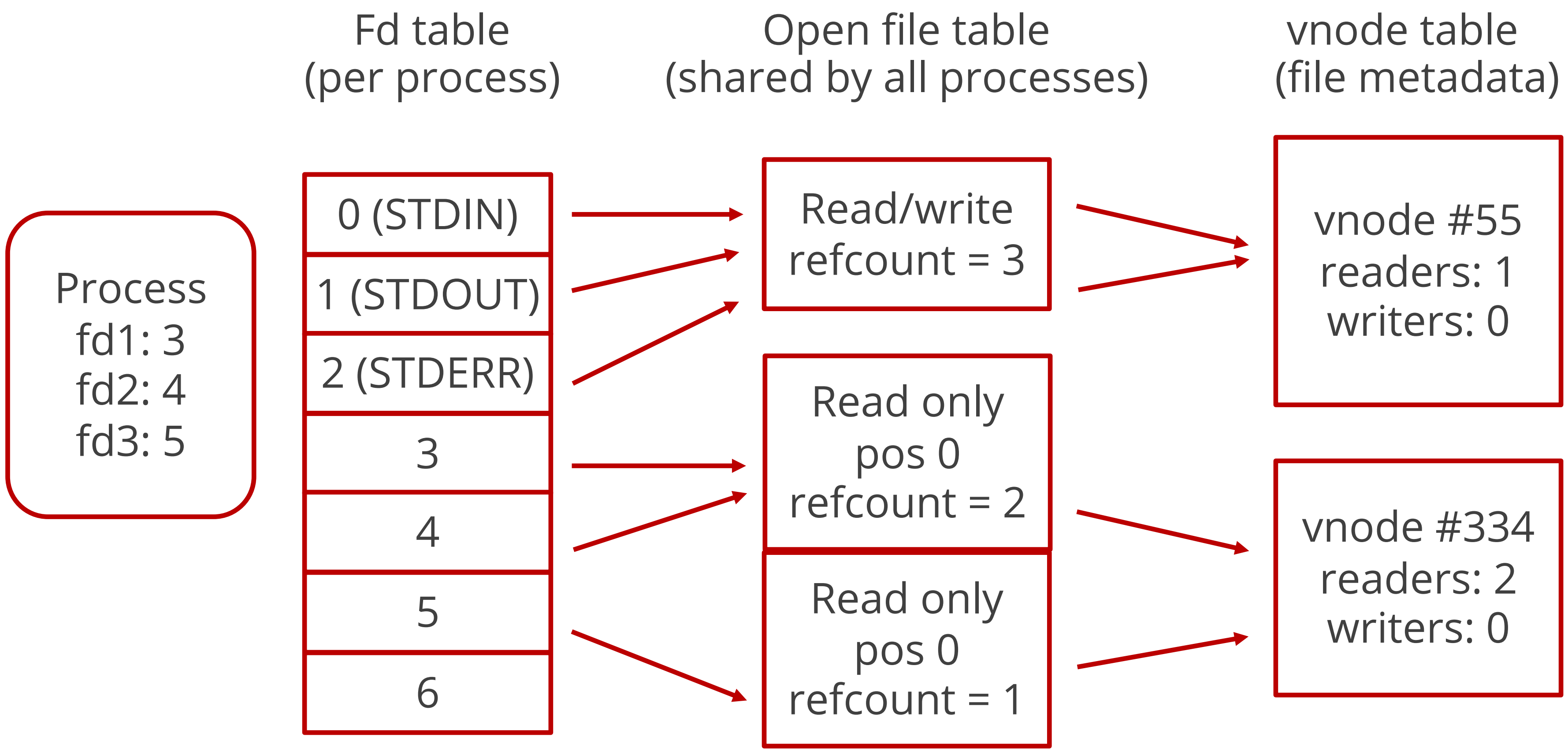

```
int fd1 = open("foo.c", O_RDONLY, 0);
```

"Open File" vs "In-Core File"




```
int fd2 = dup(fd1);
```

“Open File” vs “In-Core File”




```
int fd3 = open(“foo.c”, O_RDONLY, 0);
```

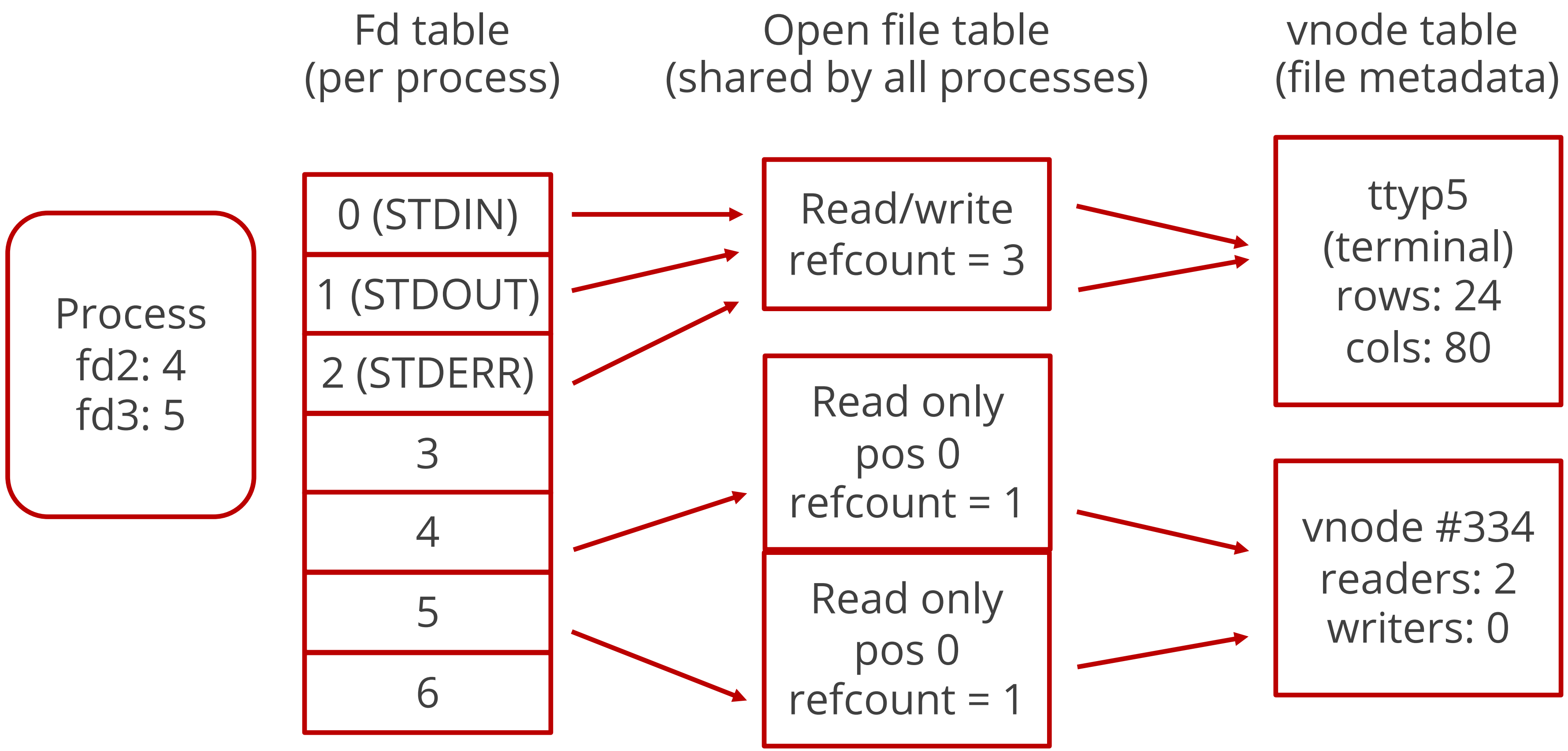
Open files - example

```
char buf[10];

int fd1 = open(“foo.c”, O_RDONLY, 0);
int fd2 = dup(fd1);
int fd3 = open(“foo.c”, O_RDONLY, 0);
read(fd1, &buf, sizeof(buf));

off_t pos2 = lseek(fd2, 0, SEEK_CUR); /* ? */
off_t pos3 = lseek(fd3, 0, SEEK_CUR); /* ? */
```

“Open File” vs “In-Core File”




```
close(fd1);
```

Let’s use file descriptors

printf writes to STDOUT. Add to this C code so the second printf writes to “error.txt” using `open` and `close`

```
printf("STDOUT is the command line\n”);




printf(“error.txt is STDOUT\n”);
```

Challenge: Are there any correctness or performance implications of how `open` works when multithreading?

What we're covering today: Filesystem interface

What are files?

`open` and fds

Directories and ownership

Directory Types

Single-level

- Flat global namespace - only *one* test.c
- Ok for floppy disks (maybe)

Directory Types

Single-level

- Flat global namespace - only *one* test.c
- Ok for floppy disks (maybe)

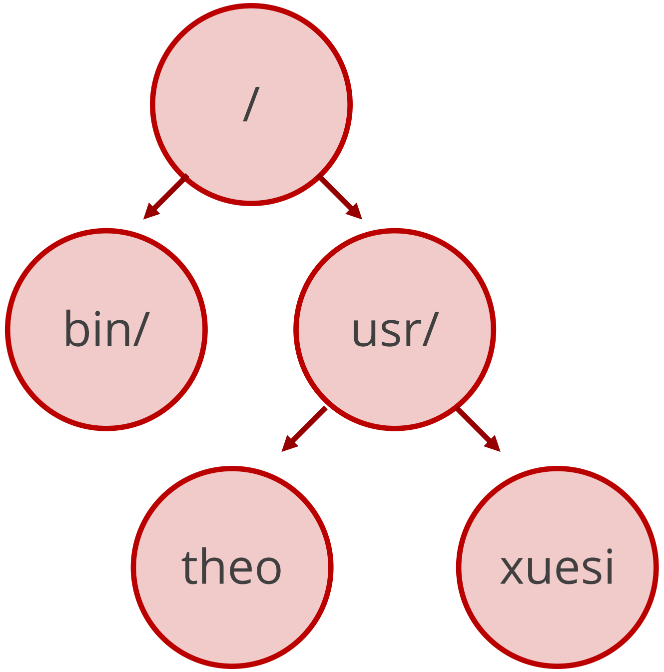
Two-level

- Every user has a directory
- One test.c *per user*
- Typical of early timesharing

Tree Directories

Directories are special files

- Created with special system calls - `mkdir`
- Format understood + maintained by OS



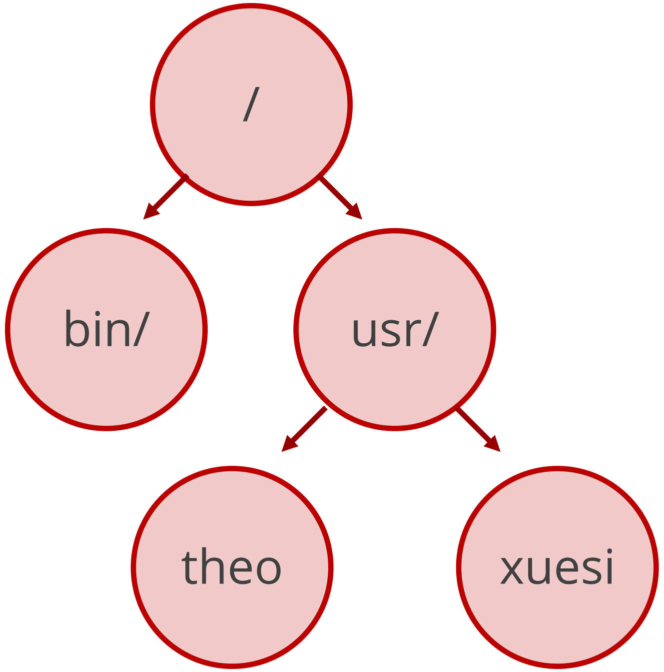
Tree Directories

Directories are special files

- Created with special system calls - `mkdir`
- Format understood + maintained by OS

Absolute Pathname

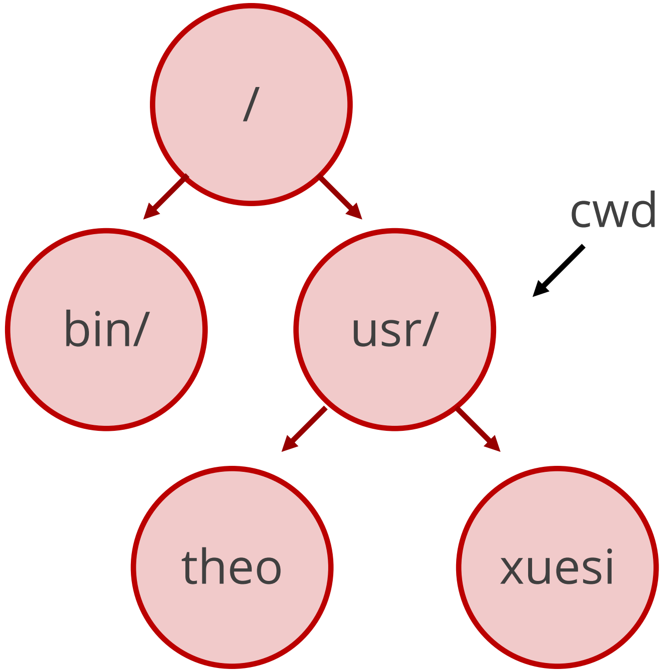
- Sequence of directory names
- Starting from “root”
- Ending with a file name
- `/usr/theo`



Tree Directories

Current directory (".", cwd)

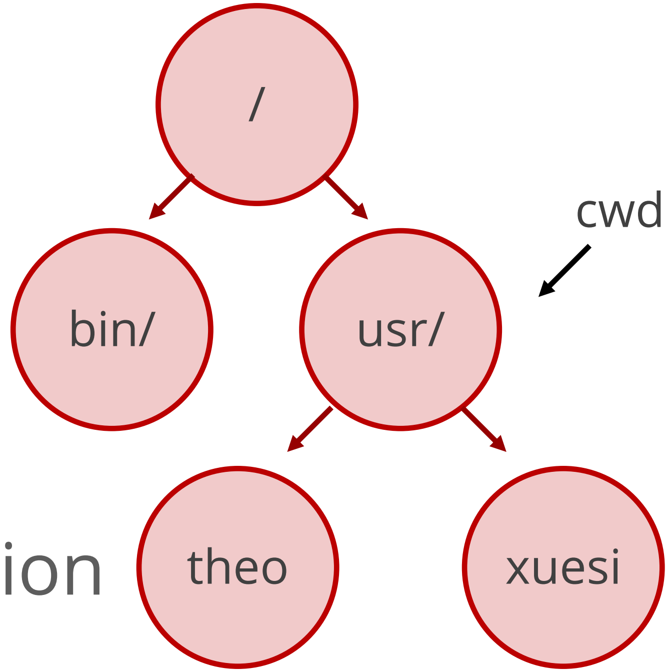
- "Where I am now" (e.g. /usr/)
- Start of *relative* pathname
  - ./xuesi -> /usr/xuesi
  - ../bin -> /bin/
- Each process has a cwd



Tree Directories

Current directory (“.”, cwd)

- “Where I am now” (e.g. /usr/)
- Start of *relative* pathname
  - ./xuesi -> /usr/xuesi
  - ../bin -> /bin/
- Each process has a cwd



Almost everything is a file description

```
(12:49 sjmcalli@pembroke2 ~) > ls -l /proc/$$/cwd
lrwxrwxrwx 1 sjmcalli pdl 0 Mar 25 12:46 /proc/43576/cwd -> /h/sjmcalli
```

Protection - typical baseline

Files specifies *owner, group*

- Permissions for owner, permissions for group members
- Permissions for “other” / “world"

Traditional Unix encoding

- `chmod 777 example.txt`
- `r w x    r - x    - - x = 0751` (octal)
- `1 1 1    1 0 1    0 0 1`
- V7 Unix: 3 16-bit words specified all permission info
  - permission bits, user #, group #

What we've covered today: Filesystem interface

After the lecture today, you should be able to:

- Explain what a file is and operations that exist on files
- Discuss why we need an open syscall and file descriptors
- Compare different directory types

Next time: How do filesystems enable the file abstraction?