CS 471 Operating Systems

Yue Cheng

George Mason University

Spring 2019

Review: RAID

RAID

- **Idea**: Build an awesome disk from small, cheap disks

- **Metrics**: Capacity, performance, reliability

RAID

- **Idea**: Build an awesome disk from small, cheap disks
- **Metrics**: Capacity, performance, reliability
- The art of tradeoff navigation

RAID Levels

- **RAID-0:**
  - No redundancy, perf & capacity upper-bound
- **RAID-1:**
  - Mirroring
- **RAID-4:**
  - Parity disk
- **RAID-5:**
  - Parity disk (rotated among disks)

File System Abstraction

What is a File?

- File: Array of bytes
  - Ranges of bytes can be read/written
- File system (FS) consists of many files
- Files need names so programs can choose the right one

File Names

- Three types of names (abstractions)
  - **inode** (low-level names)
  - **path** (human readable)
  - **file descriptor** (runtime state)

Inodes

- Each file has exactly one inode number
- Inodes are unique (at a given time) within a FS
- Numbers may be recycled after deletes

Inodes

- Each file has exactly one inode number
- Inodes are unique (at a given time) within a FS
- Numbers may be recycled after deletes
- Show inodes via `stat`
  - `$ stat <file or dir>`

‘stat’ Example

```
PROMPT>: stat test.dat
File: ‘test.dat’  Size: 5     Blocks: 8     IO Block: 4096   regular file
Device: 803h/2051d        Inode: 119341128   Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1001/     yue)   Gid: ( 1001/     yue)
Context: unconfined_u:object_r:user_home_t:s0
Access: 2015-12-17 04:12:47.935716294 -0500
Modify: 2014-12-12 19:25:32.669625220 -0500
Change: 2014-12-12 19:25:32.669625220 -0500
Birth: -
```

Path (multiple directories)

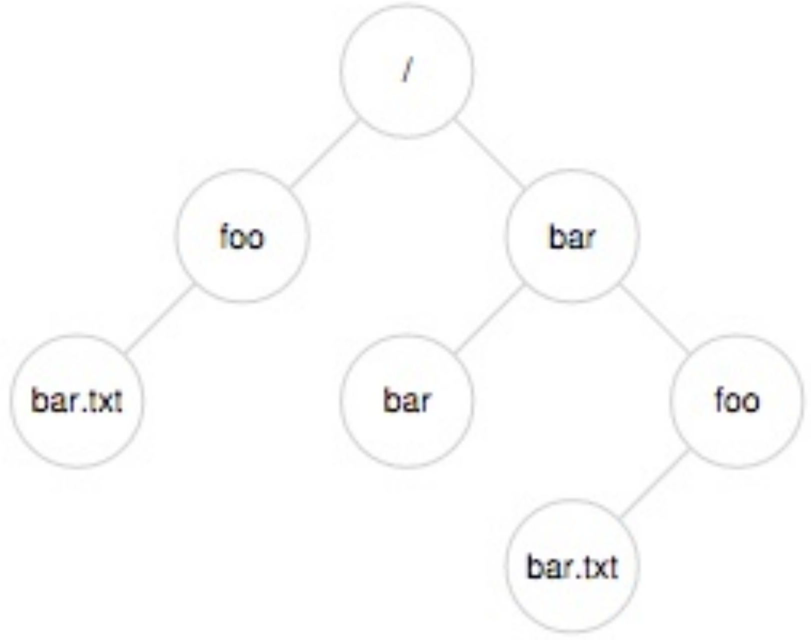
- A directory is a file
  - Associated with an inode

- Contains a list of <user-readable name, low-level name> pairs

Path (multiple directories)

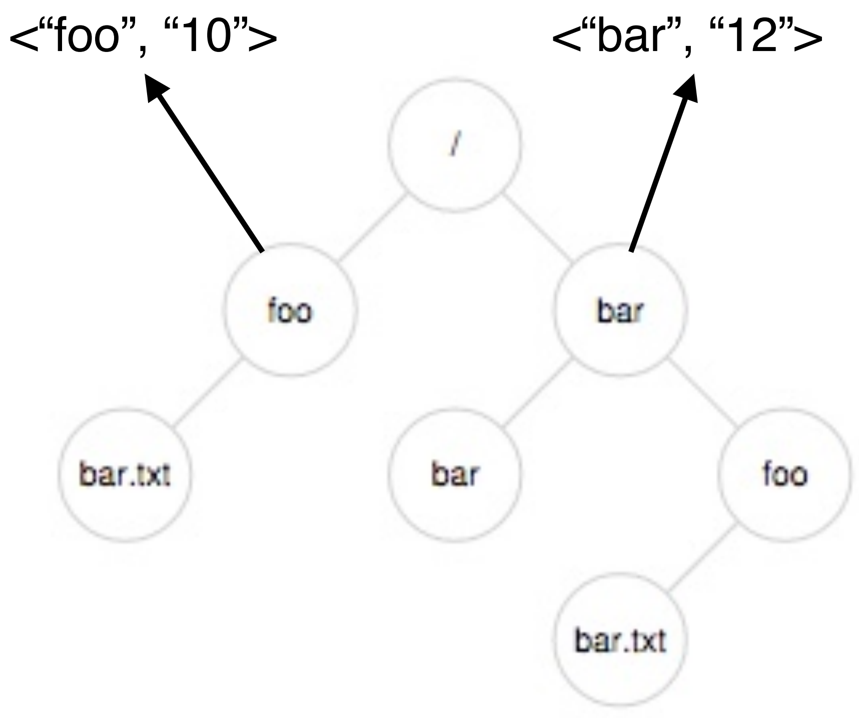
- A directory is a file
  - Associated with an inode

- Contains a list of <user-readable name, low-level name> pairs



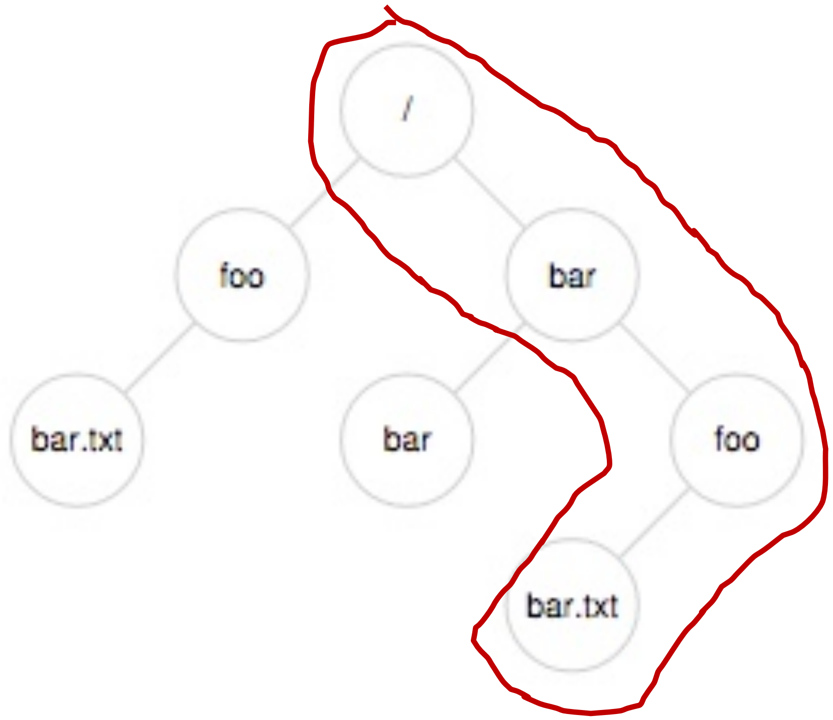
Path (multiple directories)

- A directory is a file
  - Associated with an inode
- Contains a list of <user-readable name, low-level name> pairs



Path (multiple directories)

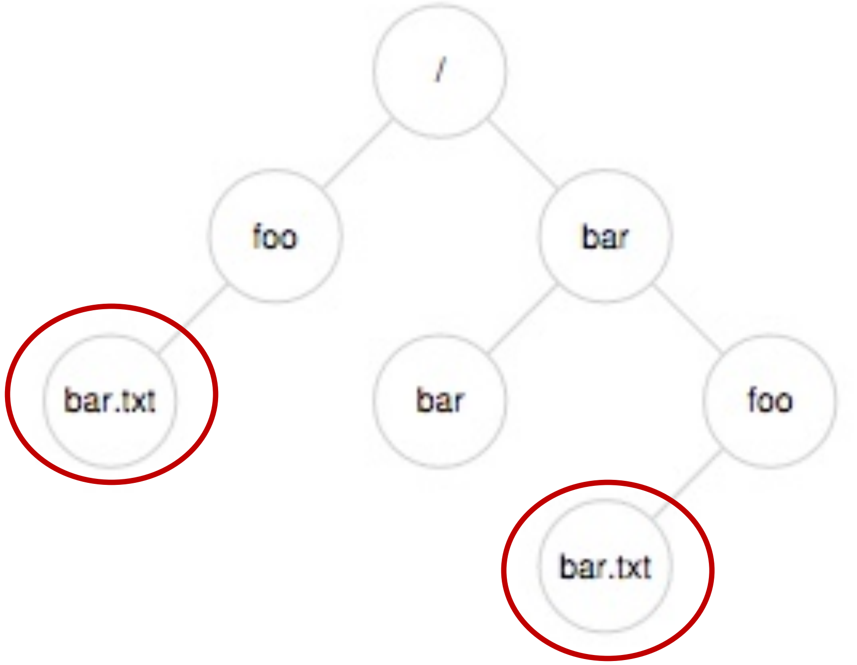
- A directory is a file
  - Associated with an inode
- Contains a list of <user-readable name, low-level name> pairs
- Directory tree: reads for getting final inode called **traversal**



[traverse /bar/foo/bar.txt]

File Naming

- Directories and files can have the same name as long as they are in different locations of the file-system tree



- .txt, .c, etc.
  - Naming convention
  - In UNIX-like OS, no enforcement for extension name

Special Directory Entries

```
prompt> ls -al
total 216
drwxr-xr-x   19 yue   staff    646 Nov 23 16:28 .
drwxr-xr-x+ 40 yue   staff   1360 Nov 15 01:41 ..
-rw-r--r--@  1 yue   staff   1064 Aug 29 21:48 common.h
-rwxr-xr-x   1 yue   staff   9356 Aug 30 14:03 cpu
-rw-r--r--@  1 yue   staff    258 Aug 29 21:48 cpu.c
-rwxr-xr-x   1 yue   staff   9348 Sep  6 12:12 cpu_bound
-rw-r--r--   1 yue   staff    245 Sep  5 13:10 cpu_bound.c
…
```

File System Interfaces

Creating Files

- UNIX system call: open( )

```
int fd = open(char *path, int flag, mode_t mode);
-OR-
int fd = open(char *path, int flag);
```

File Descriptor (`fd`)

- `open()` returns a file descriptor (`fd`)
  - A `fd` is an integer
  - Private per process

- An opaque handle that gives caller the power to perform certain operations

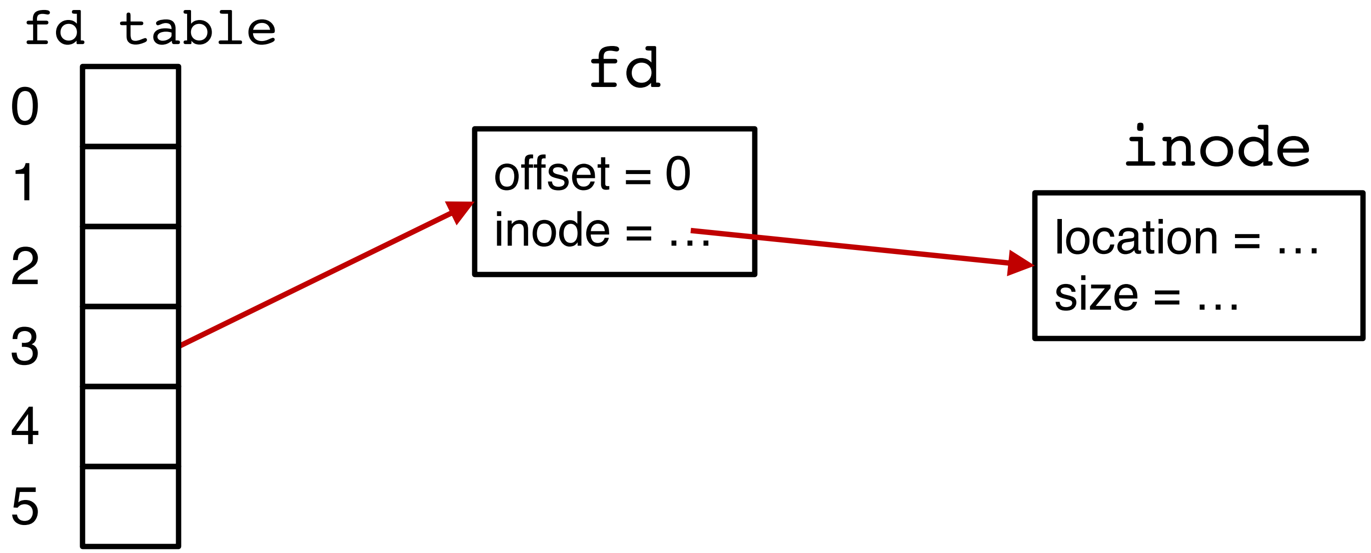
- You can think of a `fd` as a pointer to an object of the file
  - By owning such an object, you can call other “methods” to access the file

open( ) Example

```
int fd1 = open(“file.txt”, O_CREAT);  // return 3
read(fd1, buf, 8);
int fd2 = open(“file.txt”, O_WRONLY); // return 4
int fd3 = dup(fd2);                   // return 5
```

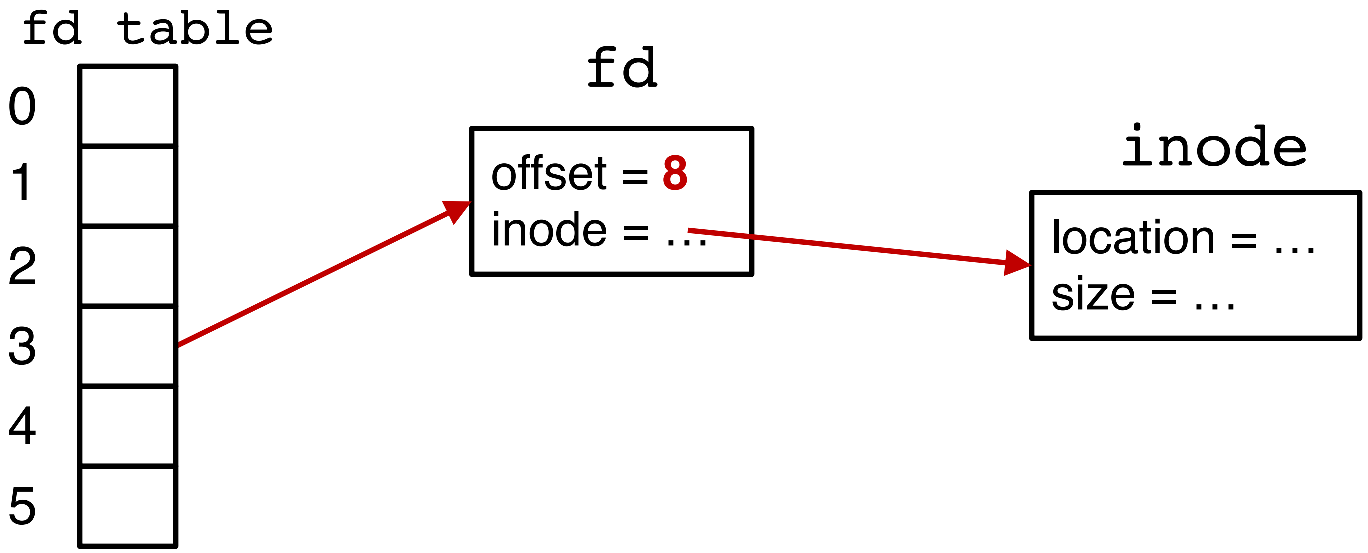
open( ) Example

```
int fd1 = open("file.txt", O_CREAT);  // return 3
```



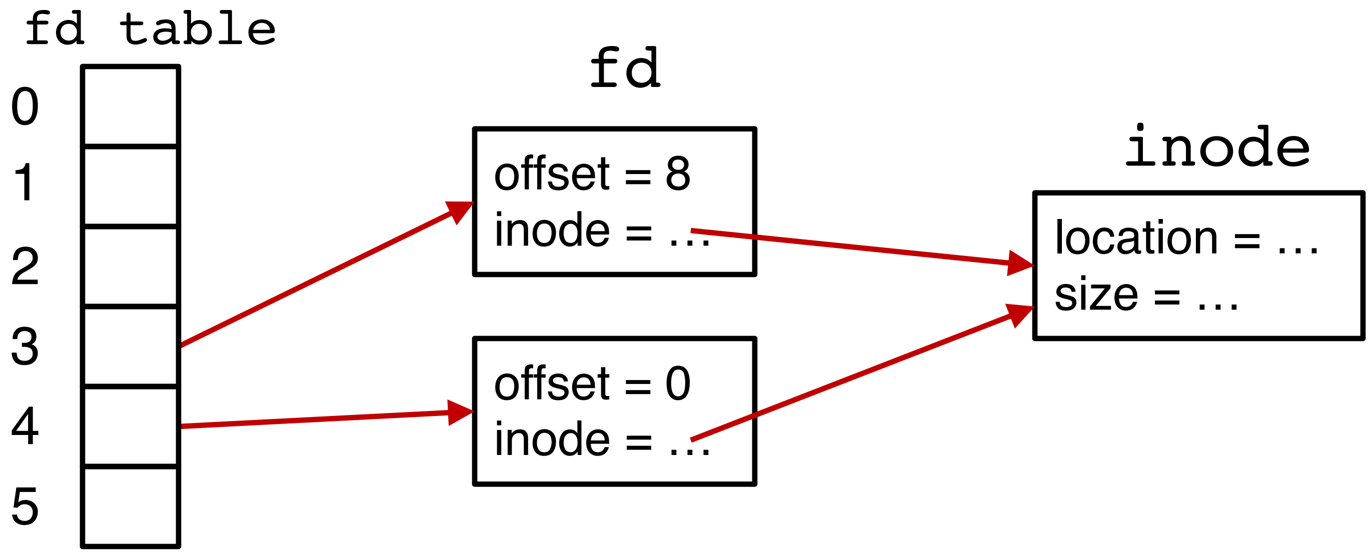
open( ) Example

```
int fd1 = open("file.txt", O_CREAT);  // return 3
read(fd1, buf, 8);
```



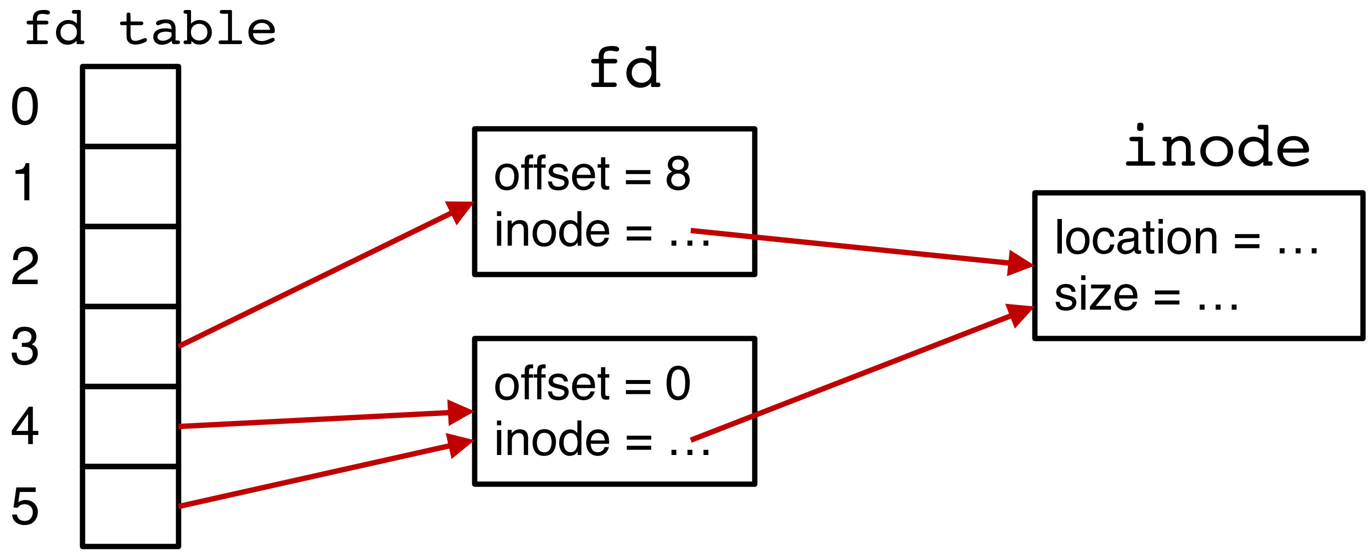
open( ) Example

```
int fd1 = open("file.txt", O_CREAT);  // return 3
read(fd1, buf, 8);
int fd2 = open("file.txt", O_WRONLY); // return 4
```



open( ) Example

```
int fd1 = open(“file.txt”, O_CREAT);  // return 3
read(fd1, buf, 8);
int fd2 = open(“file.txt”, O_WRONLY); // return 4
int fd3 = dup(fd2);                   // return 5
```



UNIX File Read and Write APIs

```
int fd = open(char *path, int flag, mode_t mode);
-OR-
int fd = open(char *path, int flag);

ssize_t sz = read(int fd, void *buf, size_t count);

ssize_t sz = write(int fd, void *buf, size_t count);

int ret = close(int fd);
```

Reading and Writing Files

```
prompt> echo hello > file.txt
prompt> cat file.txt
hello
prompt>
```

Reading and Writing Files

```
prompt> strace cat file.txt
...
open("file.txt", O_RDONLY)              = 3
read(3, "hello\n", 65536)               = 6
write(1, "hello\n", 6)                  = 6
read(3, "", 65536)                      = 0
close(3)                                = 0
...
prompt>
```

Reading and Writing Files

Open the file with read only mode

```
prompt> strace cat file.txt
...
open("file.txt", O_RDONLY)                 = 3
read(3, "hello\n", 65536)                  = 6
write(1, "hello\n", 6)                     = 6
read(3, "", 65536)                         = 0
close(3)                                   = 0
...
prompt>
```

Reading and Writing Files

Open the file with read only mode

Read content from file

```
prompt> strace cat file.txt
...
open("file.txt", O_RDONLY)                   = 3
read(3, "hello\n", 65536)                    = 6
write(1, "hello\n", 6)                       = 6
read(3, "", 65536)                           = 0
close(3)                                     = 0
...
prompt>
```

Reading and Writing Files

Open the file with read only mode

Read content from file

Write string to std output `fd` 1

```
prompt> strace cat file.txt
...
open("file.txt", O_RDONLY)          = 3
read(3, "hello\n", 65536)           = 6
write(1, "hello\n", 6)              = 6
read(3, "", 65536)                  = 0
close(3)                            = 0
...
prompt>
```

Reading and Writing Files

Open the file with read only mode

Read content from file

Write string to std output `fd` 1

`cat` tries to read more but reaches EOF

```
prompt> strace cat file.txt
...
open("file.txt", O_RDONLY)            = 3
read(3, "hello\n", 65536)             = 6
write(1, "hello\n", 6)                = 6
read(3, "", 65536)                    = 0
close(3)                              = 0
...
prompt>
```

Reading and Writing Files

Open the file with read only mode

Read content from file

Write string to std output `fd` 1

`cat` tries to read more but reaches EOF

`cat` done with file ops and closes the file

```
prompt> strace cat file.txt
...
open("file.txt", O_RDONLY)              = 3
read(3, "hello\n", 65536)               = 6
write(1, "hello\n", 6)                  = 6
read(3, "", 65536)                      = 0
close(3)                                = 0
...
prompt>
```

Non-Sequential File Operations

```
off_t offset = lseek(int fd, off_t offset, int whence);
```

Non-Sequential File Operations

```
off_t offset = lseek(int fd, off_t offset, int whence);
```

whence:

- If whence is SEEK_SET, the offset is set to `offset` bytes
- If whence is SEEK_CUR, the offset is set to its current location plus `offset` bytes
- If whence is SEEK_END, the offset is set to the size of the file plus `offset` bytes

Non-Sequential File Operations

```
off_t offset = lseek(int fd, off_t offset, int whence);
```

whence:

- If whence is `SEEK_SET`, the offset is set to `offset` bytes
- If whence is `SEEK_CUR`, the offset is set to its current location plus `offset` bytes
- If whence is `SEEK_END`, the offset is set to the size of the file plus `offset` bytes

Note: Calling `lseek()` does not perform a disk seek!

Writing Immediately with `fsync()`

```
int fd = fsync(int fd);
```

- `fsync(fd)` forces buffers to flush to disk, and (usually) tells the disk to flush its write cache too
  - To make the data durable and persistent

- **Write buffering** improves performance

Renaming Files

```
prompt> mv file.txt new_name.txt
```

Renaming Files

```
prompt> strace mv file.txt new_name.txt
...
rename("file.txt", "new_name.txt")   = 0
...
prompt>
```

Renaming Files

System call `rename()` **atomically** renames a file

```
prompt> strace mv file.txt new_name.txt
...
rename("file.txt", "new_name.txt")   = 0
...
prompt>
```

Renaming Files

System call `rename()` **atomically** renames a file

```
prompt> strace mv file.txt new_name.txt
...
rename("file.txt", "new_name.txt")   = 0
...
prompt>
```

What if user program crashes?
File system does extra work to guarantee atomicity.

File Renaming Example

```
prompt> vim file.txt
```

```
int fd = open(“.file.txt.swp”,O_WRONLY|O_CREAT|O_TRUNC,S_IRUSR|S_IWUSR);
```

Using `vim` to edit a file and then save it

File Renaming Example

```
prompt> vim file.txt
… vim editing session …
```

```
int fd = open(".file.txt.swp",O_WRONLY|O_CREAT|O_TRUNC,S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version of file (editing…)
```

Using `vim` to edit a file and then save it

File Renaming Example

```
prompt> vim file.txt
… vim editing session …
prompt>
```

:wq

```
int fd = open(“.file.txt.swp”,O_WRONLY|O_CREAT|O_TRUNC,S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version of file
fsync(fd);               // make data durable
close(fd);               // close tmp file
rename(“.file.txt.swp”, “file.txt”);// change name and replacing old file
```

Using `vim` to edit a file and then save it

Deleting Files

```
prompt> rm file.txt
```

Deleting Files

```
prompt> strace rm file.txt
...
unlink("file.txt")            = 0
...
prompt>
```

Deleting Files

System call `unlink()` is called to delete a file

```
prompt> strace rm file.txt
...
unlink("file.txt")                = 0
...
prompt>
```

Deleting Files

System call `unlink()` is called to delete a file

```
prompt> strace rm file.txt
...
unlink("file.txt")              = 0
...
prompt>
```

Directories are deleted when `unlink()` is called

File descriptors are deleted when ???

Deleting Files

System call `unlink()` is called to delete a file

```
prompt> strace rm file.txt
...
unlink("file.txt")                    = 0
...
prompt>
```

Directories are deleted when `unlink()` is called

File descriptors are deleted when `close()`, or process quits

Demo: Hard Links vs. Symbolic Links

Concurrency

- How can multiple processes avoid updating the same file at the same time?

- Normal locks don't work, as developers may have developed their programs independently

Concurrency

- How can multiple processes avoid updating the same file at the same time?

- Normal locks don't work, as developers may have developed their programs independently

- Use **`flock()`**, e.g.
  - `flock(fd, LOCK_EX)`
  - `flock(fd, LOCK_UN)`