PERSISTENCE: FILE API

Shivaram Venkataraman

CS 537, Spring 2023

Administrivia

Project 5

Project 6, extra slip days

Midterm 2: Today!

AGENDA / LEARNING OUTCOMES

How to name and organize data on a disk?

What is the API programs use to communicate with OS?

RECAP

Disks Summary

- Disks: seek between tracks, rotate within a track
- I/O time: rotation + seek + transfer
- Sequential vs random throughput
- Scheduling: SSTF, SCAN, C-SCAN

QUIZ 15

https://tinyurl.com/cs537-fa24-q15

Assume the following disk characteristics:

Average Seek time: 7ms
Average rotational delay: 3ms
Transfer rate of disk: 50 MB/s
Untitled Title

Description (optional)

What is the throughput rate for a sequential read of 10 MB (i.e. one seek and rotation)? *

- 67.22 MB/s
- 53.71 MB/s
- 42.13 MB/s
- 47.62 MB/s
- None of the above

DISKS → FILES

WHAT IS A FILE?

Array of persistent bytes that can be read/written

File system consists of many files

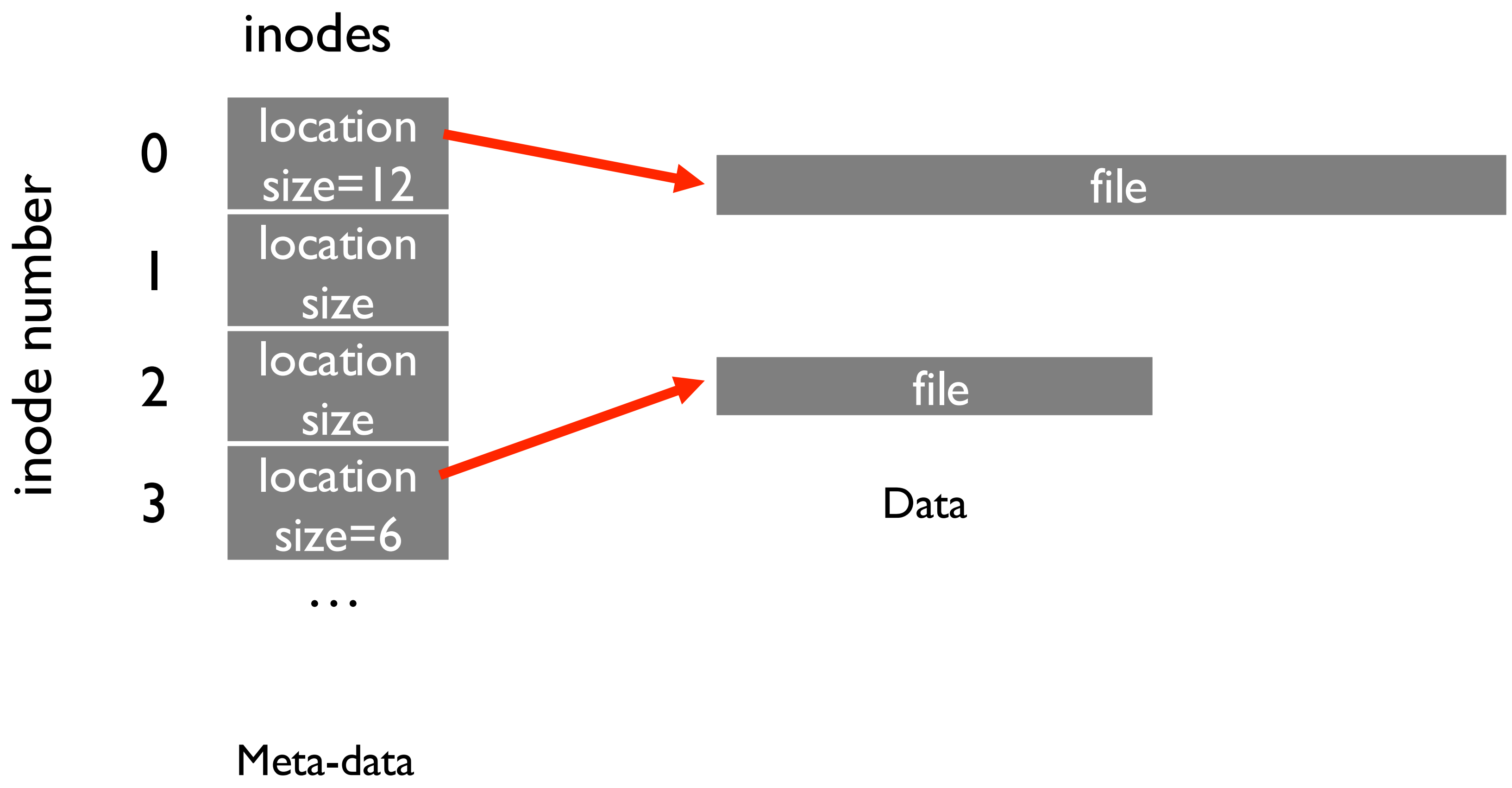
Refers to collection of files

Also refers to part of OS that manages those files

Files need names to access correct one

Three types of names

- Unique id: inode numbers
- Path
- File descriptor


inodes
inode number
0
1
2
3
location
size=12
location
size
location
size
location
size=6
...
file
file
Data
Meta-data

FILE API (ATTEMPT 1)

```
read(int inode, void *buf, size_t nbyte)
write(int inode, void *buf, size_t nbyte)
seek(int inode, off_t offset)
```

Disadvantages?

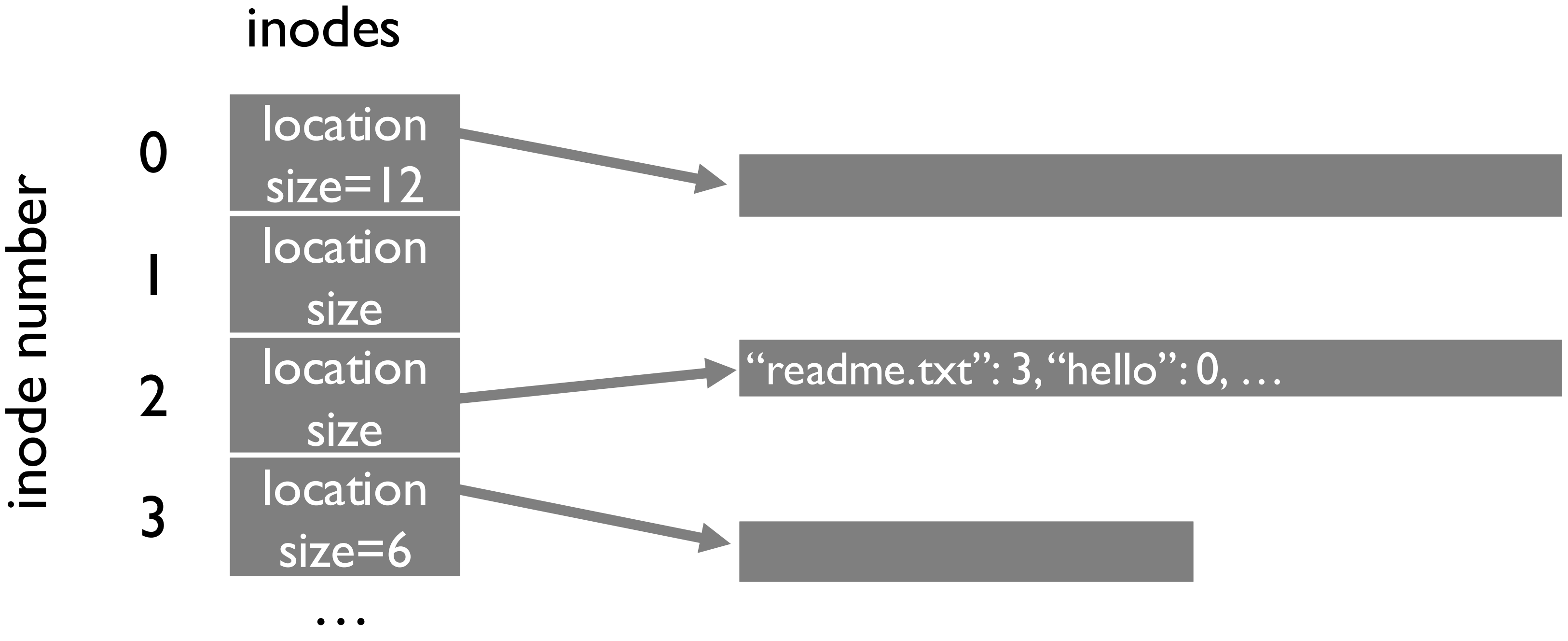
- names hard to remember
- no organization or meaning to inode numbers
- semantics of offset across multiple processes?

PATHS

String names are friendlier than number names

File system still interacts with inode numbers

Store *path-to-inode* mappings in a special file or rather a Directory!



inodes
inode number
0
1
2
3
location
size=12
location
size
location
size
location
size=6
…
“readme.txt”: 3, “hello”: 0, …

PATHS

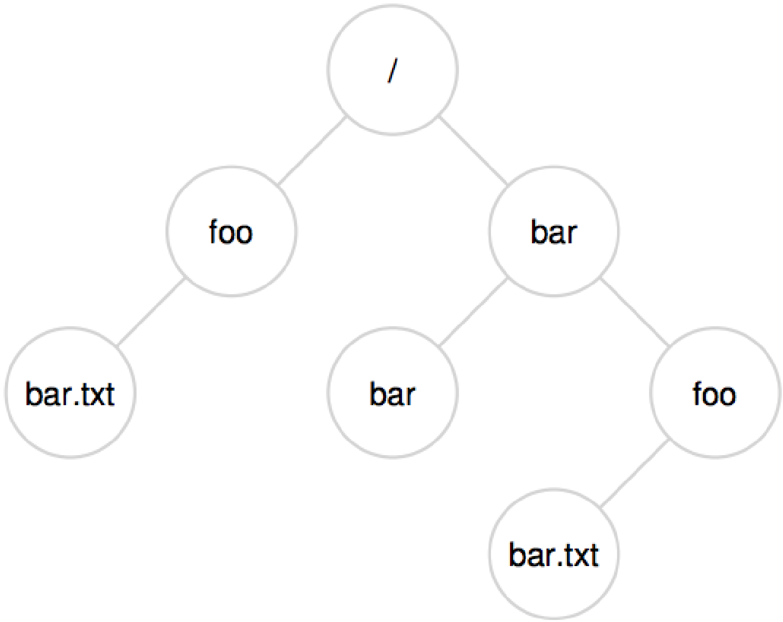
Directory Tree instead of single root directory

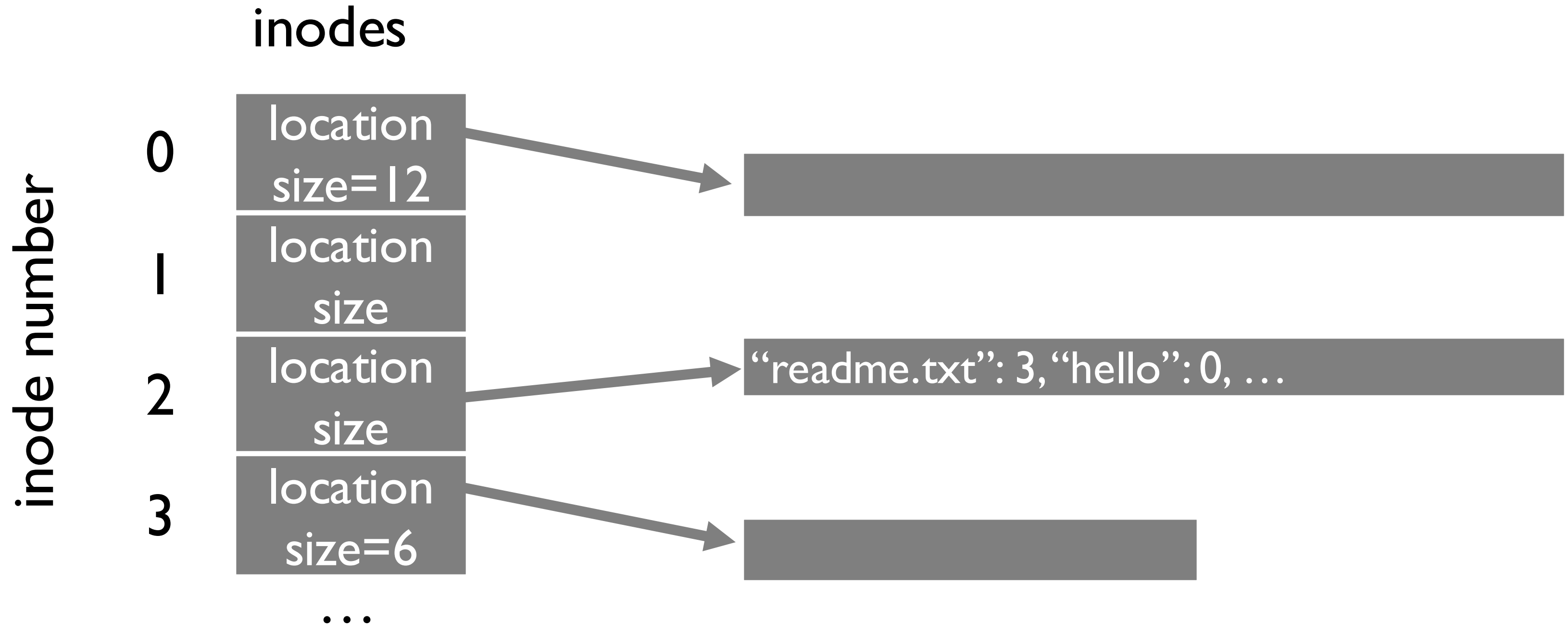
File name needs to be unique within a directory

/usr/lib/file.so

/tmp/file.so

Store file-to-inode mapping in each directory





Reads for getting final inode called "traversal"

Example: read /hello

FILE API (ATTEMPT 2)

```
read(char *path, void *buf, off_t offset, size_t nbyte)
write(char *path, void *buf, off_t offset, size_t nbyte)
```

Disadvantages?

Expensive traversal!

Goal: traverse once

File Descriptor (FD)

Idea:

- Do expensive traversal once (open file)
- Store inode in descriptor object (kept in memory).
- Do reads/writes via descriptor, which tracks offset

Each process:

- File-descriptor table contains pointers to open file descriptors

Integers used for file I/O are indexes into this table

- stdin: 0, stdout: 1, stderr: 2

FILE API (ATTEMPT 3)

```
int fd = open(char *path, int flag, mode_t mode)
read(int fd, void *buf, size_t nbyte)
write(int fd, void *buf, size_t nbyte)
close(int fd)
```

advantages:
- string names
- hierarchical
- traverse once
- offsets precisely defined

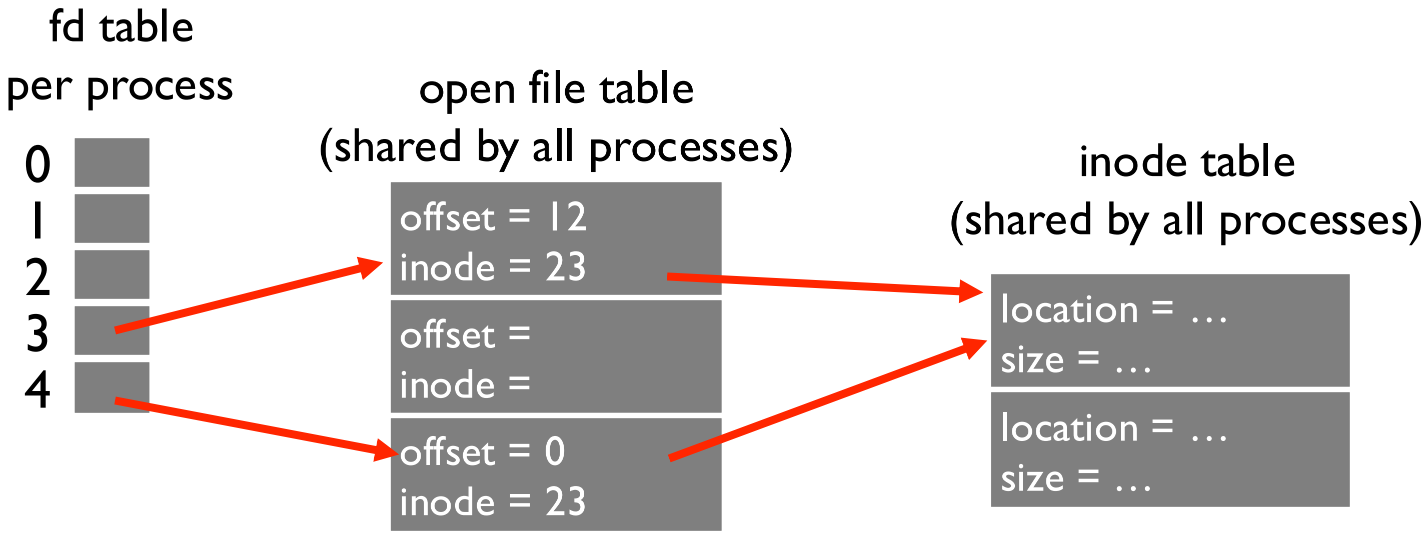
FD TABLE (XV6)

```
struct file {
  ...
  struct inode *ip;
  uint off;
};
```

```
struct {
    struct spinlock lock;
    struct file file[NFILE];
} ftable;
```

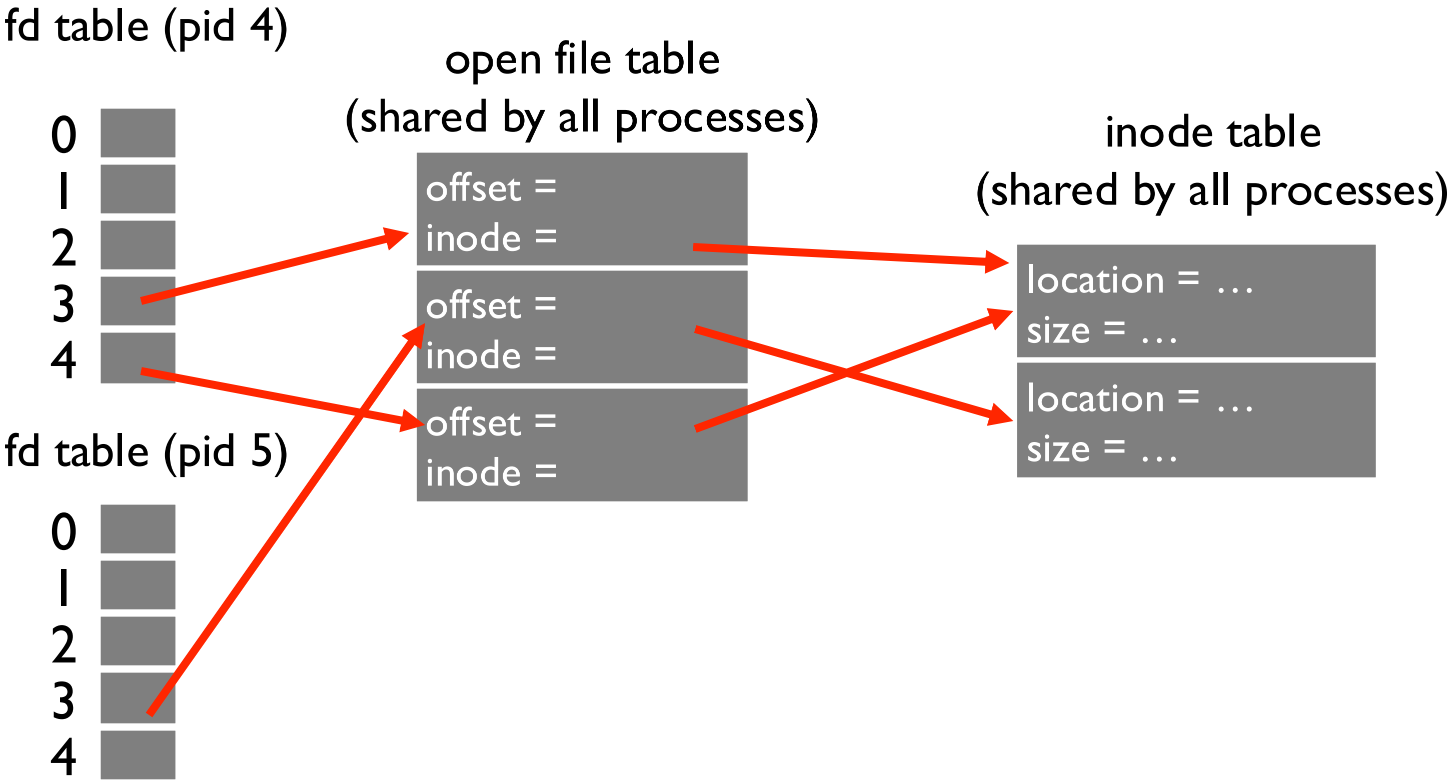
```
// Per-process state
struct proc {
  ...
  struct file *ofile[NOFILE];  // Open files
  ...
}
```

FD TABLE



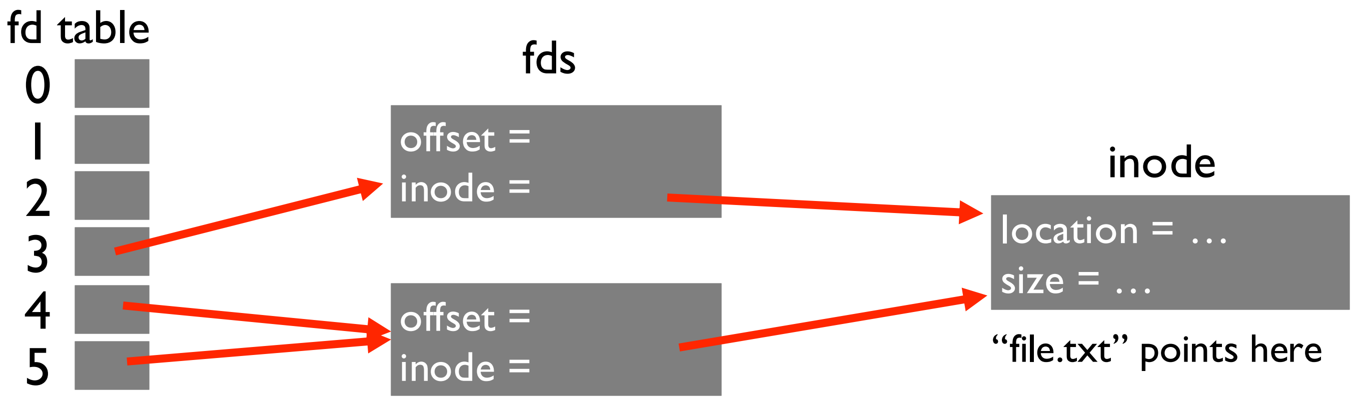

```
int fd1 = open("file.txt"); // returns 3
read(fd1, buf, 12);
int fd2 = open("file.txt"); // returns 4
```

FD TABLE



fd table (pid 4)
0
1
2
3
4
fd table (pid 5)
0
1
2
3
4
open file table
(shared by all processes)
offset =
inode =
offset =
inode =
offset =
inode =
inode table
(shared by all processes)
location = …
size = …
location = …
size = …

DUP




```
int fd1 = open(“file.txt”); // returns 3
read(fd1, buf, 12);
int fd2 = open(“file.txt”); // returns 4
int fd3 = dup(fd2);         // returns 5
```

READ NOT SEQUENTIALLY

```
off_t lseek(int filedesc, off_t offset, int 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.
```

```
struct file {
  ...
  struct inode *ip;
  uint off;
};
```

PRACTICE

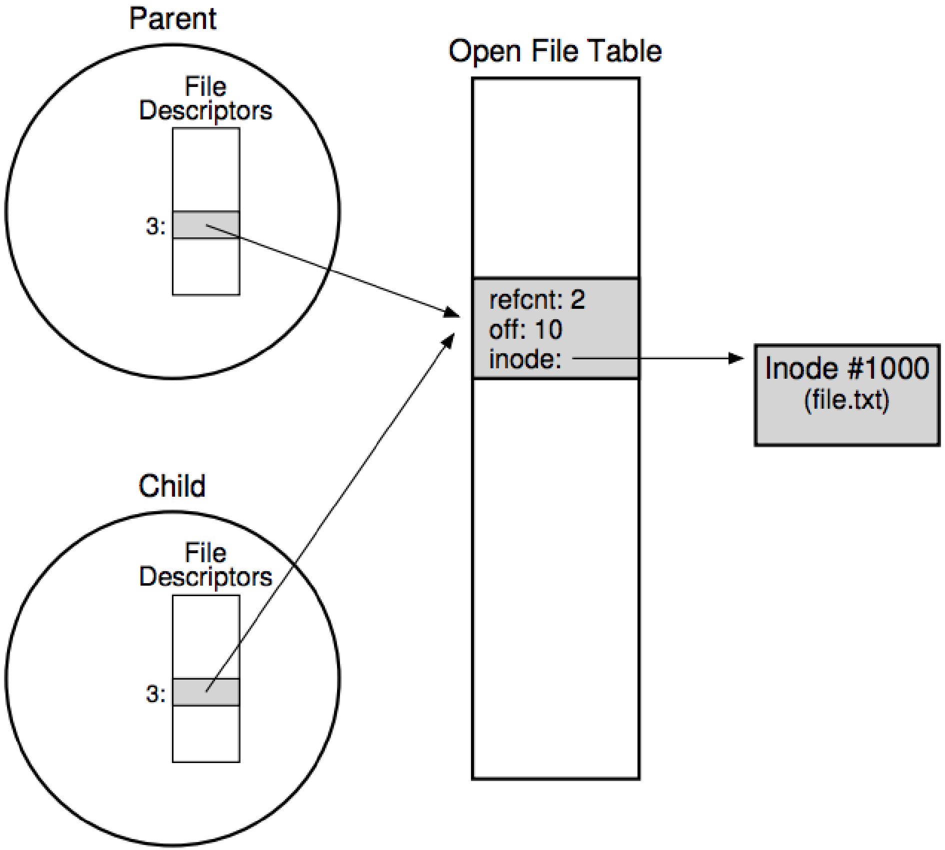
```
int fd1 = open("file.txt"); // returns 12
int fd2 = open("file.txt"); // returns 13
read(fd1, buf, 16);
int fd3 = dup(fd2);         // returns 14
read(fd2, buf, 16);
lseek(fd1, 100, SEEK_SET);
```

Offset for fd1?

Offset for fd2?

Offset for fd3?

WHAT HAPPENS ON FORK?



COMMUNICATING REQUIREMENTS: FSYNC

File system keeps newly written data in memory for awhile

Write buffering improves performance (why?)

But what if system crashes before buffers are flushed?

fsync(int fd) forces buffers to flush to disk, tells disk to flush its write cache

Makes data durable

DELETING FILES

There is no system call for deleting files!

Inode (and associated file) is **garbage collected** when there are no references

Paths are deleted when: `unlink()` is called

FDs are deleted when: `close()` or process quits

RENAME

rename(char *old, char *new):

- deletes an old link to a file
- creates a new link to a file

Just changes name of file, does not move data

Even when renaming to new directory

What can go wrong if system crashes at wrong time?

Atomic File Update

Say application wants to update file.txt atomically

If crash, should see only old contents or only new contents

1. write new data to file.txt.tmp file
2. fsync file.txt.tmp
3. rename file.txt.tmp over file.txt, replacing it

DIRECTORY CALLS

- mkdir()
- readdir()

LINKS

Hard links: Both path names use same inode number

File does not disappear until all hard links removed; cannot link directories

```
$ echo hello > a.txt
$ ln a.txt b.txt
$ cat b.txt
hello
$ ls -li .
```

LINKS

Soft or symbolic links: Point to second path name; can softlink to dirs

```
ln -s oldfile softlink
```

Confusing behavior:"file does not exist"!

Confusing behavior: "cd linked dir; cd .., in different parent! "

LINKS

```
fariha@node0:dir$ cat ~/a.txt
hello
fariha@node0:dir$ ls -i ~/a.txt
4000077 /users/fariha/a.txt
fariha@node0:dir$ ln ~/a.txt b.txt
fariha@node0:dir$ ls -li .
total 4
4000077 -rw-r--r-- 2 fariha advosuwmadison-P 6 Nov  7 07:34 b.txt
fariha@node0:dir$ cat b.txt
hello
fariha@node0:dir$ ln -s ~/a.txt sym.txt
fariha@node0:dir$ ls -li .
total 4
4000077 -rw-r--r-- 2 fariha advosuwmadison-P  6 Nov  7 07:34 b.txt
4014092 lrwxrwxrwx 1 fariha advosuwmadison-P 19 Nov  7 07:35 sym.txt → /users/fariha/a.txt
fariha@node0:dir$ cat sym.txt
hello
fariha@node0:dir$ rm ~/a.txt
fariha@node0:dir$ cat b.txt
hello
fariha@node0:dir$ cat sym.txt
cat: sym.txt: No such file or directory
fariha@node0:dir$
```

LINKS

```
fariha@node0:dir$ ls -l /usr/bin/python
lrwxrwxrwx 1 root root 7 Oct 11  2021 /usr/bin/python -> python3
fariha@node0:dir$ ls -l /usr/bin/python3
lrwxrwxrwx 1 root root 10 Aug 18  2022 /usr/bin/python3 -> python3.10
fariha@node0:dir$
```

PERMISSIONS, ACCESS CONTROL

```
fariha@node0:dir$ ls -la
total 12
drwxr-xr-x  2 fariha advosuwmadison-P 4096 Nov  7 07:43 .
drwxr-xr-x 12 fariha advosuwmadison-P 4096 Nov  7 07:35 ..
-rw-r--r--  1 fariha advosuwmadison-P    6 Nov  7 07:34 b.txt
-rw-------  1 fariha advosuwmadison-P    0 Nov  7 07:43 file
lrwxrwxrwx  1 fariha advosuwmadison-P   19 Nov  7 07:35 sym.txt -> /users/fariha/a.txt
```

SUMMARY

Using multiple types of name provides convenience and efficiency

Special calls (fsync, rename) let developers communicate requirements to file system

Next class: Directory features, Filesystem implementation