PERSISTENCE: FILE API

Shivaram Venkataraman

CS 537, Spring 2020

ADMINISTRIVIA

Midterm grades are up!

Project 4a: due tomorrow at 10pm

Discussion today:

- Some debugging hints (valgrind), P4b preview

AGENDA / LEARNING OUTCOMES

How do we achieve resilience against disk errors?

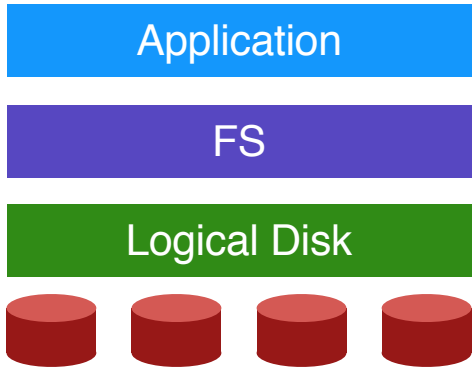
How to name and organize data on a disk?

What is the API programs use to communicate with OS?

RECAP

RAID

Build logical disk from many physical disks.



Logical disk gives

capacity,

performance,

reliability

RAID: **R**edundant **A**rray of **I**nexpensive **D**isks

METRICS

Capacity: how much space can apps use?

Reliability: how many disks can we safely lose? (assume fail stop)

Performance: how long does each workload take? (latency, throughput)

Normalize each to characteristics of one disk

Different **RAID levels** make different trade-offs

RAID LEVEL COMPARISONS

	Reliability	Capacity	Read latency	Write Latency	Seq Read	Seq Write	Rand Read	Rand Write
RAID-0	0	C*N	D	D	N * S	N * S	N * R	N * R
RAID-1	1	C*N/2	D	D	N/2 * S	N/2 * S	N * R	N/2 * R

RAID-4 STRATEGY

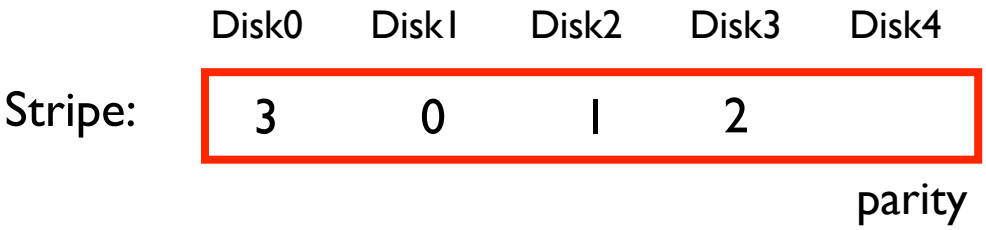
Use **parity** disk

If an equation has N variables, and N-1 are known,
you can solve for the unknown.

Treat sectors across disks in a stripe as an equation.

Data on bad disk is like an unknown in the equation.

RAID 4: EXAMPLE



What functions can we use to compute parity?

RAID-4: ANALYSIS

What is capacity?

How many disks can fail?

Latency (read, write)?

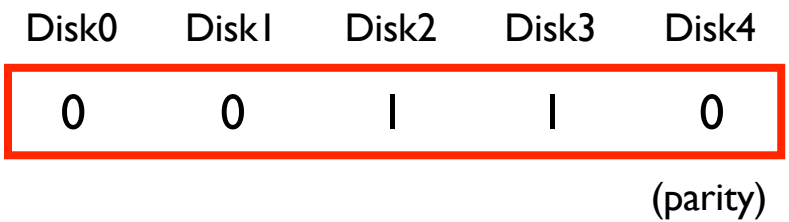
Disk0	Disk1	Disk2	Disk3	Disk4 parity
1	0	1	1	1
0	1	1	0	0
1	1	0	1	1

N := number of disks
C := capacity of 1 disk
S := sequential throughput of 1 disk
R := random throughput of 1 disk
D := latency of one small I/O operation

RAID-4: THROUGHPUT

What is steady-state throughput for

- sequential reads?
- sequential writes?
- random reads?
- random writes? (next page!)



RAID-4: ADDITIVE VS SUBTRACTIVE

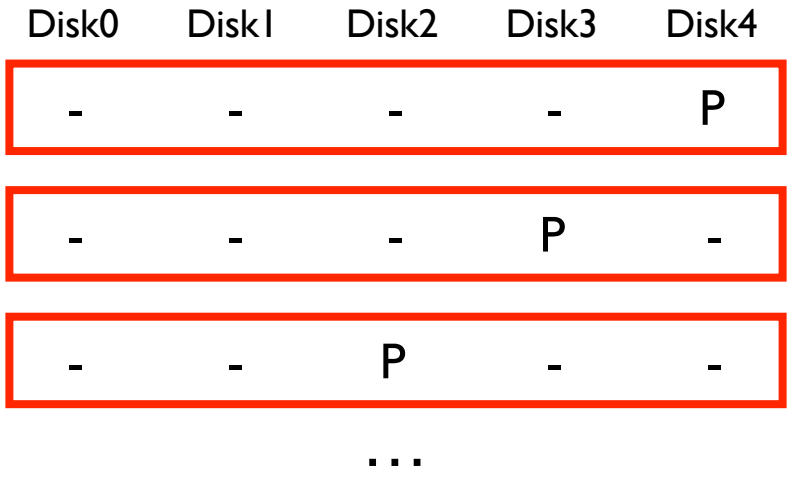
C0	C1	C2	C3	P0
0	0	1	1	XOR(0,0,1,1)

Additive Parity

Subtractive Parity

$$P_{new} = (C_{old} \oplus C_{new}) \oplus P_{old}$$

RAID-5



Rotate parity across different disks

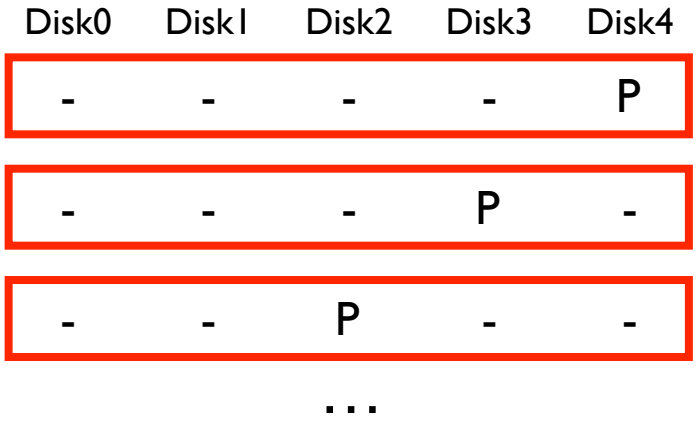
RAID-5: ANALYSIS

What is capacity?

How many disks can fail?

Latency (read, write)?

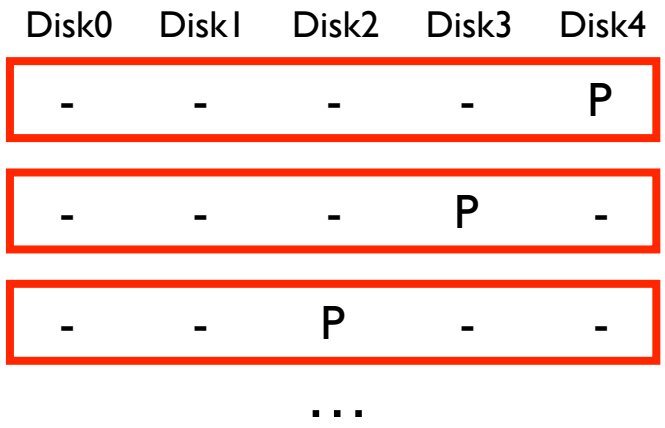
N := number of disks
C := capacity of 1 disk
S := sequential throughput of 1 disk
R := random throughput of 1 disk
D := latency of one small I/O operation



RAID-5: THROUGHPUT

What is steady-state throughput for RAID-5?

- sequential reads?
- sequential writes?
- random reads?
- random writes? (next page!)



RAID-5 RANDOM WRITES

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

RAID LEVEL COMPARISONS

	Reliability	Capacity	Read latency	Write Latency	Seq Read	Seq Write	Rand Read	Rand Write
RAID-0	0	C*N	D	D	N * S	N * S	N * R	N * R
RAID-1	1	C*N/2	D	D	N/2 * S	N/2 * S	N * R	N/2 * R
RAID-4	1	(N-1) * C	D	2D	(N-1)*S	(N-1)*S	(N-1)*R	R/2
RAID-5	1	(N-1) * C	D	2D	(N-1)*S	(N-1)*S	N * R	N/4 * R

SUMMARY

RAID: a faster, larger, more reliable disk system

One logical disk built from many physical disk

Different mapping and redundancy schemes present different trade-offs

DISKS → FILES

WHAT IS A FILE?

Array of persistent bytes that can be read/written

File system consists of many files

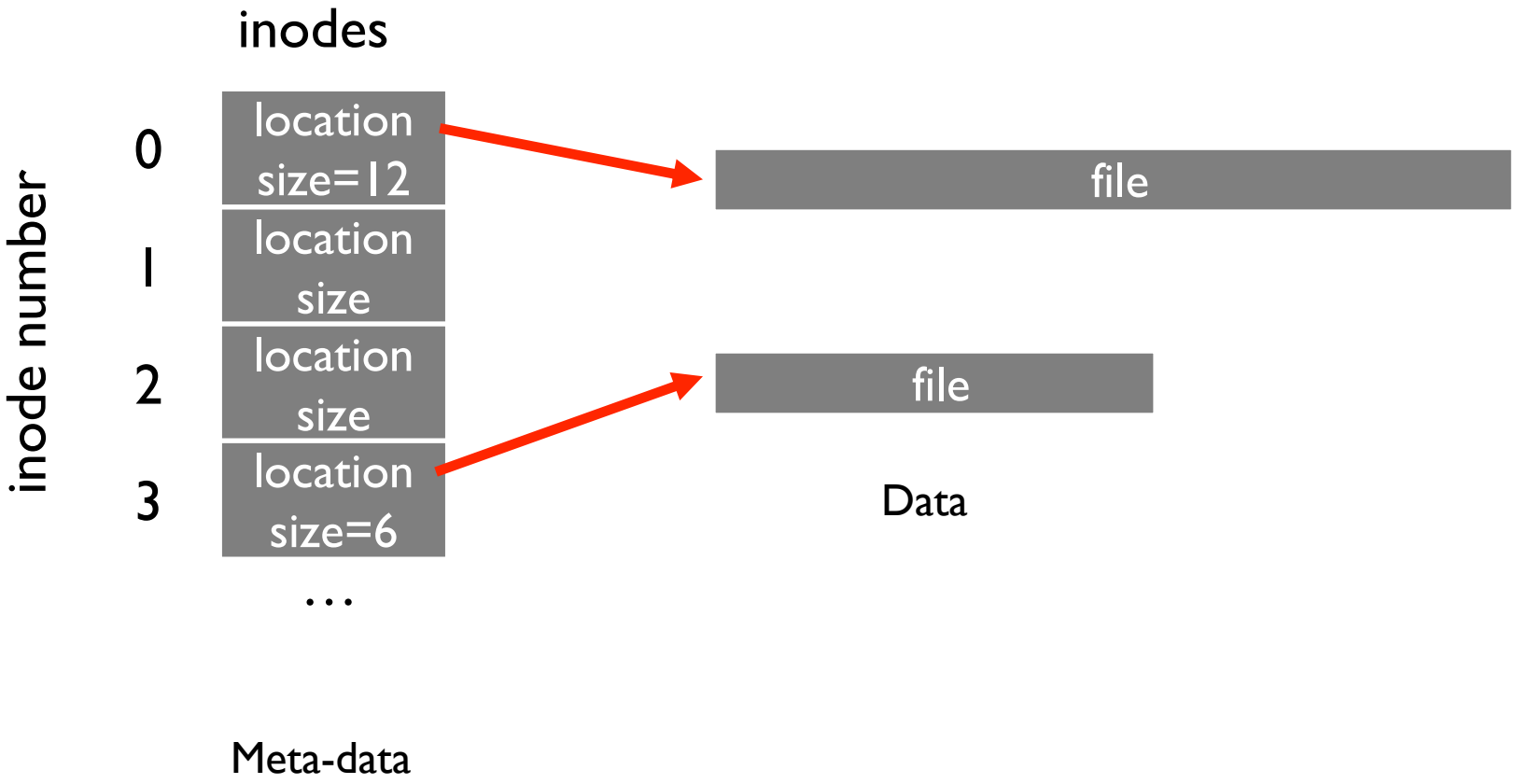
Refers to collection of files

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?

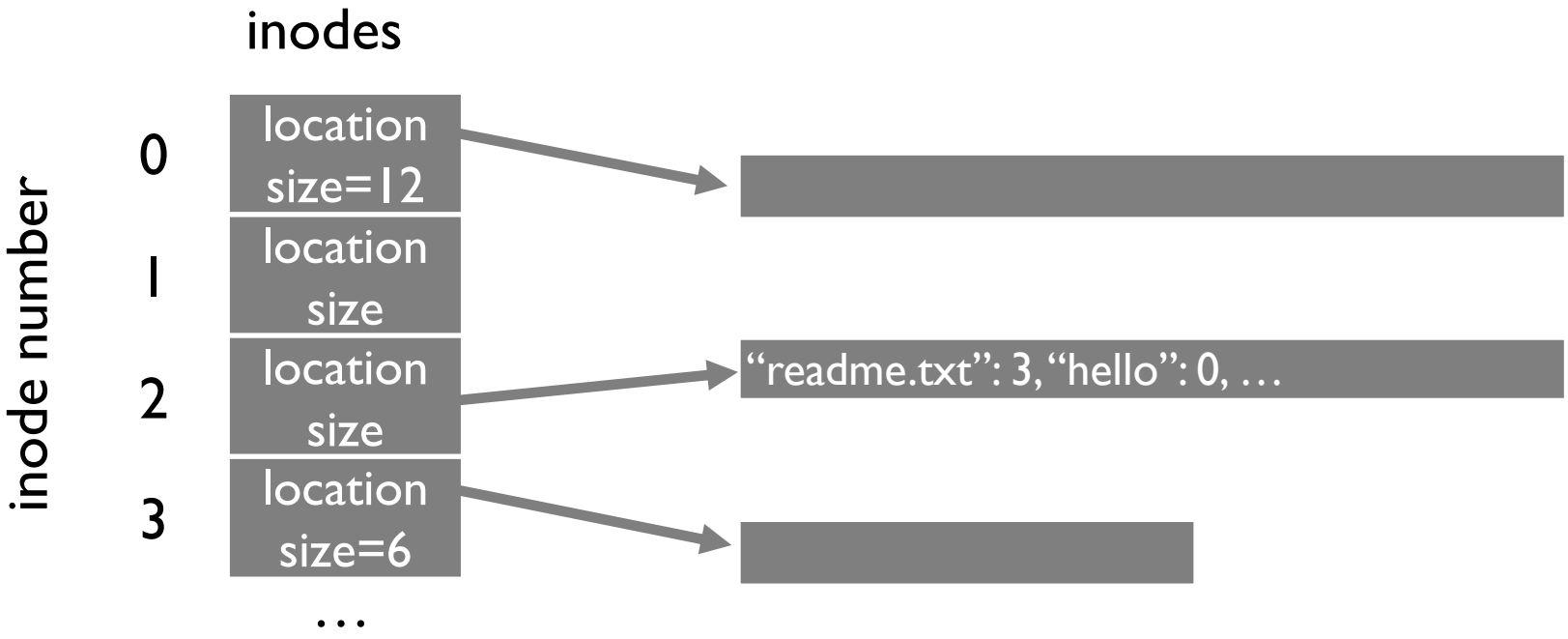
- names hard to remember
- no organization or meaning to inode numbers
- semantics of offset across multiple processes?

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

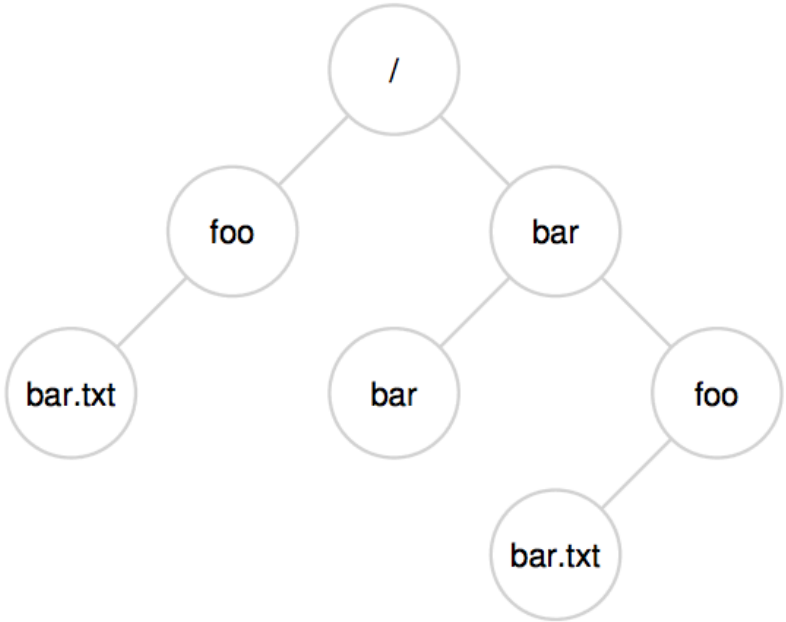
Directory Tree instead of single root directory

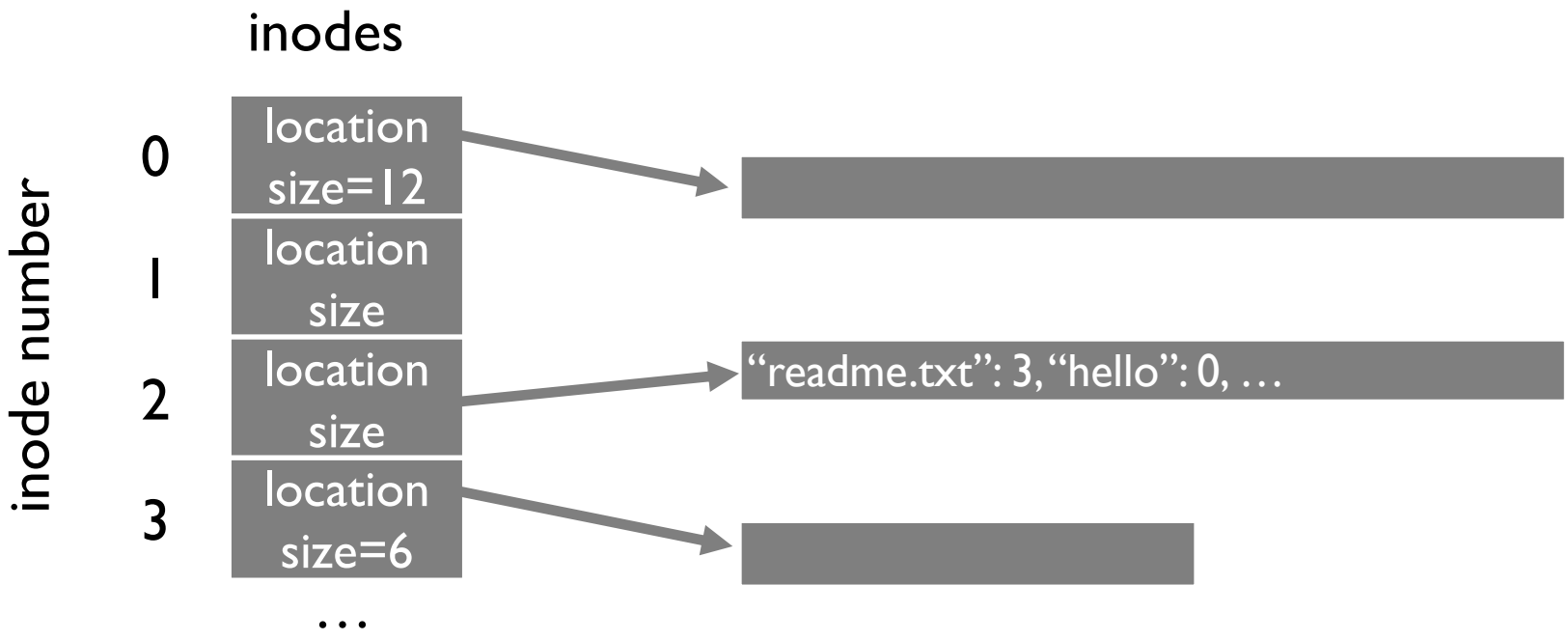
File name needs to be unique within a directory

/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

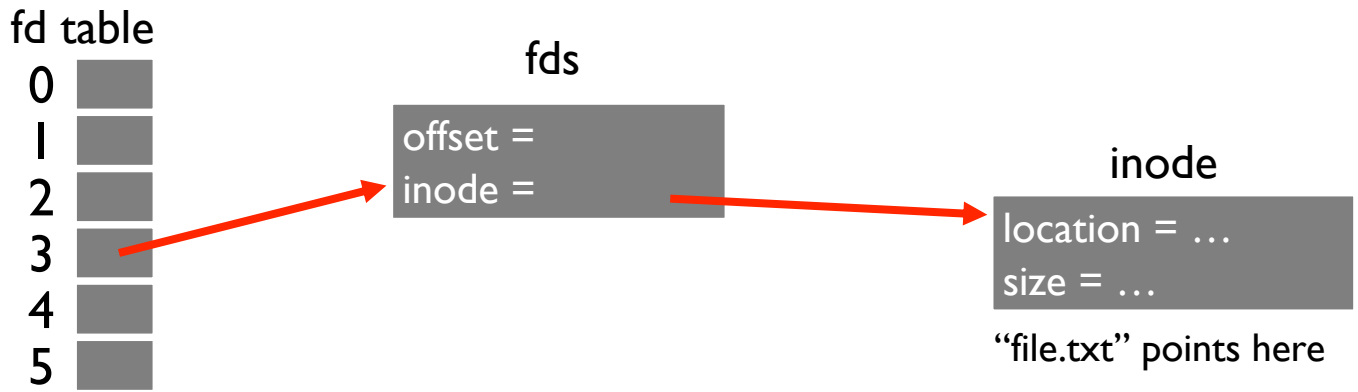
FD TABLE (XV6)

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

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

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

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;
};
```

QUIZ 24

https://tinyurl.com/cs537-sp20-quiz24

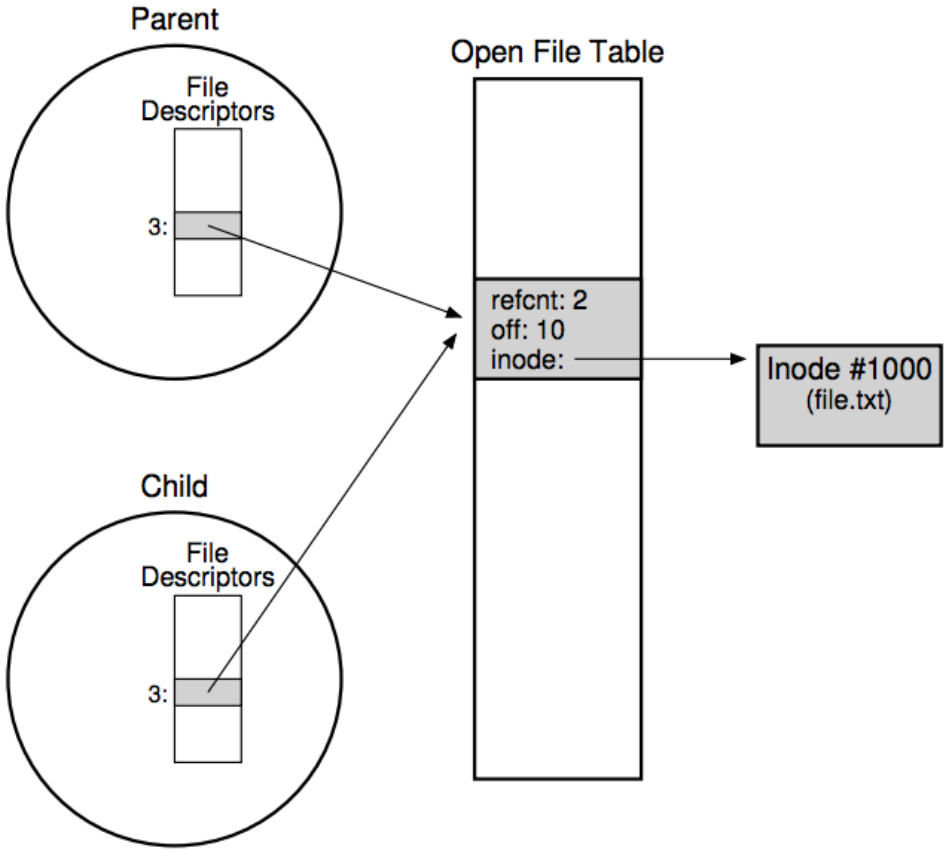
```
int fd1 = open("file.txt"); // returns 12
int fd2 = open("file.txt"); // returns 13
read(fd1, buf, 16);
int fd3 = dup(fd2);         // returns 14
read(fd2, buf, 16);
lseek(fd1, 100, SEEK_SET);
```

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

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

Discussion: Debugging parallel code, P4b