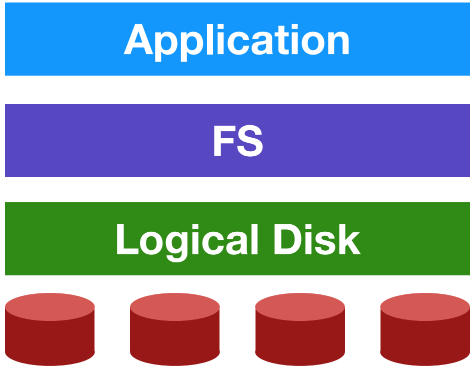
CS 423 Operating System Design: Persistence: RAID, FS API 04/16

Ram Alagappan

RECAP

SOLUTION 2: RAID

Build logical disk from many physical disks.



Logical disk gives

capacity,

performance,

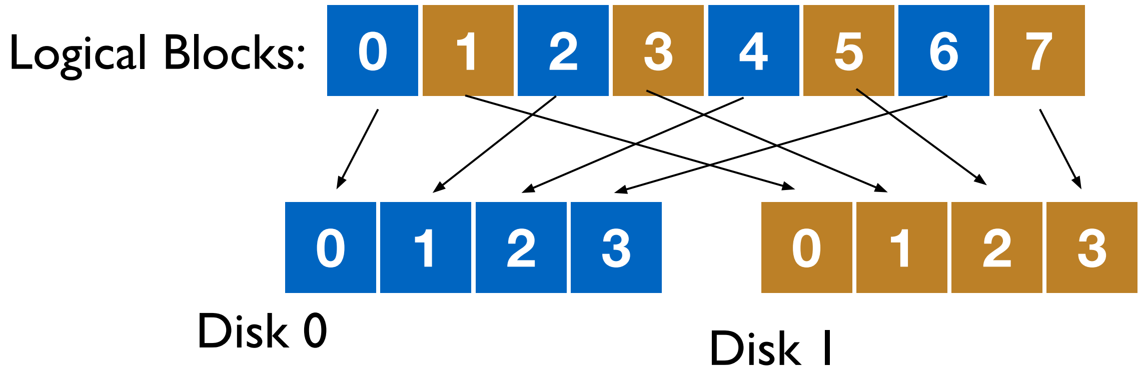
reliability

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

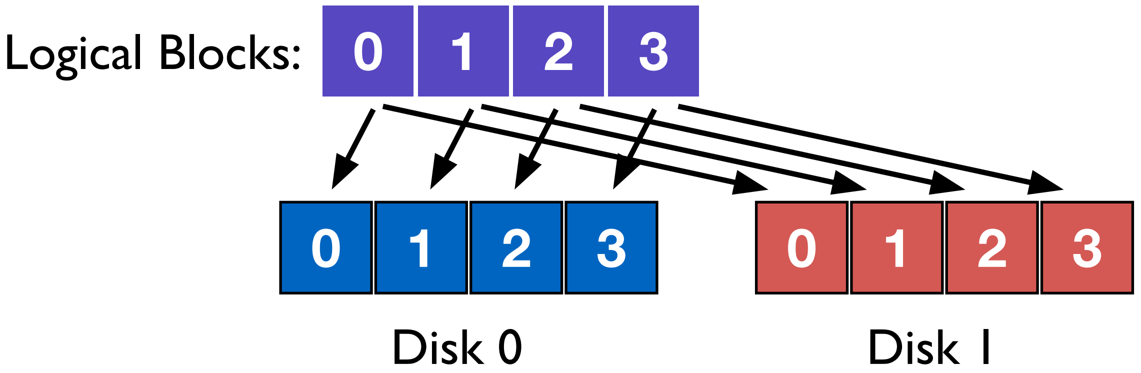
Transparency: no changes to the FS, host, Apps → ease of deployment

RAID-0: STRIPING

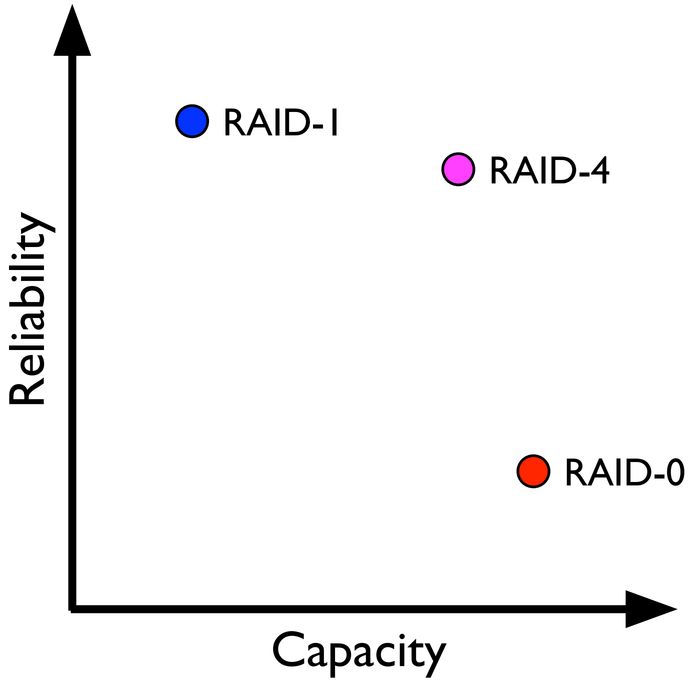
Optimize for capacity. No redundancy



RAID-1: MIRRORING



Keep two copies of all data.


RAID-1
RAID-4
RAID-0
Reliability
Capacity

END RECAP

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

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

What functions can we use to compute parity?

C0	C1	C2	C3	P
0	0	1	1	XOR(0,0,1,1) = 0
0	1	0	0	XOR(0,1,0,0) = 1

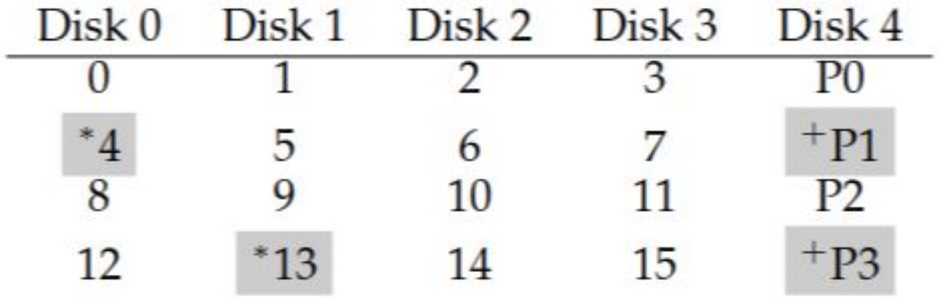
RAID-4: ADDITIVE VS SUBTRACTIVE

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

Additive Parity:
read all old blocks, recalculate parity block value

Subtractive Parity

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



Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
*4	5	6	7	+P1
8	9	10	11	P2
12	*13	14	15	+P3

RAID-4: ANALYSIS

What is the capacity?	(N-1) * C
How many disks can fail?	1
Latency (read, write)?	read: D, write: 2D

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

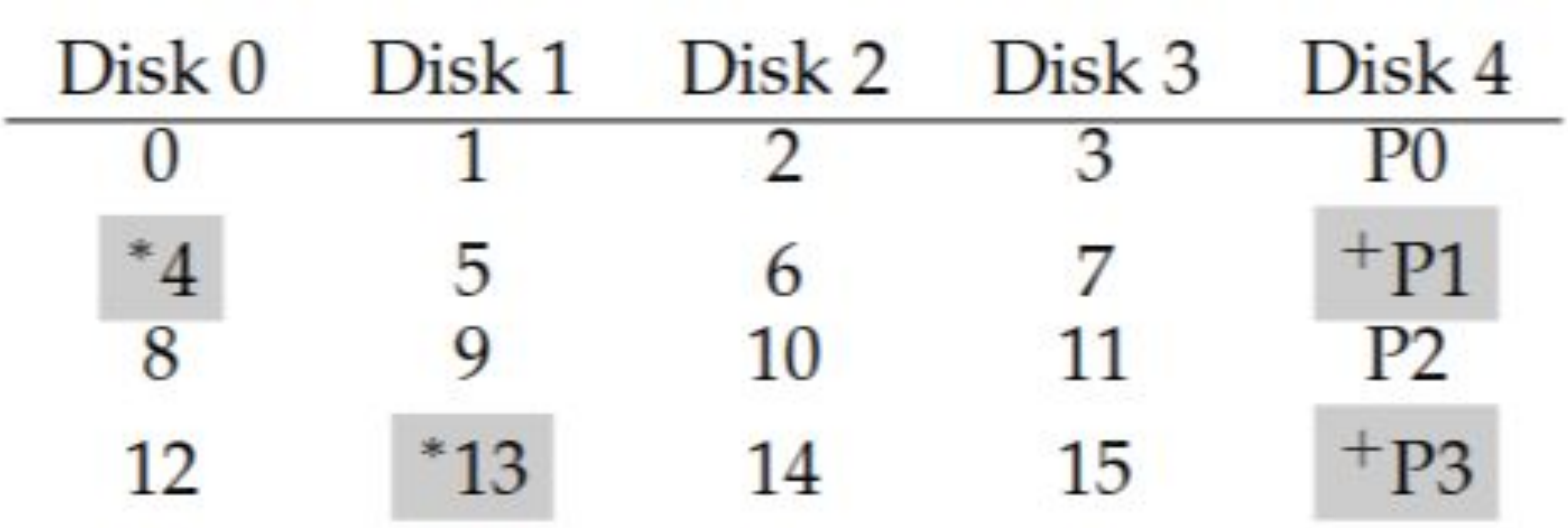
RAID-4: THROUGHPUT

What is steady-state throughput for

- sequential reads? (N-1) * S
- sequential writes? (N-1) * S
- random reads?
- random writes?

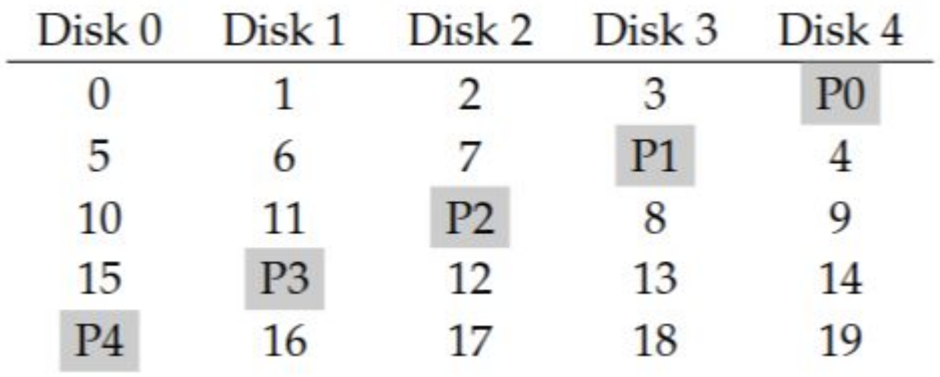
Discuss for two mins…

RAID-4 RANDOM WRITE



Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
*4	5	6	7	+P1
8	9	10	11	P2
12	*13	14	15	+P3

RAID-5



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

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

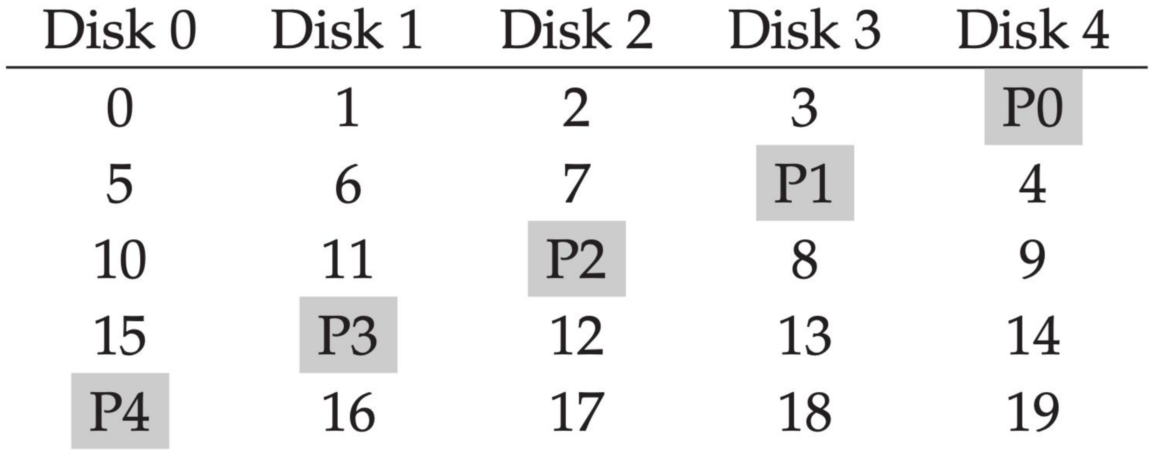
RAID-5: THROUGHPUT

What is steady-state throughput for RAID-5?

- sequential reads? (N-1) * S
- sequential writes? (N-1) * S
- random reads?
- random writes?

Discuss for two mins…

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-1 Layout: Mirroring

2 disks

Disk 0	Disk 1
0	0
1	1
2	2
3	3

4 disks

Disk 0	Disk 1	Disk 2	Disk 4
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

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

RAID 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

Plan

- This lecture: Files and FS API
- Next: File system implementation
- After: RAID/Other topics

What is a File?

Array of persistent bytes that can be read/written

File system consists of many files

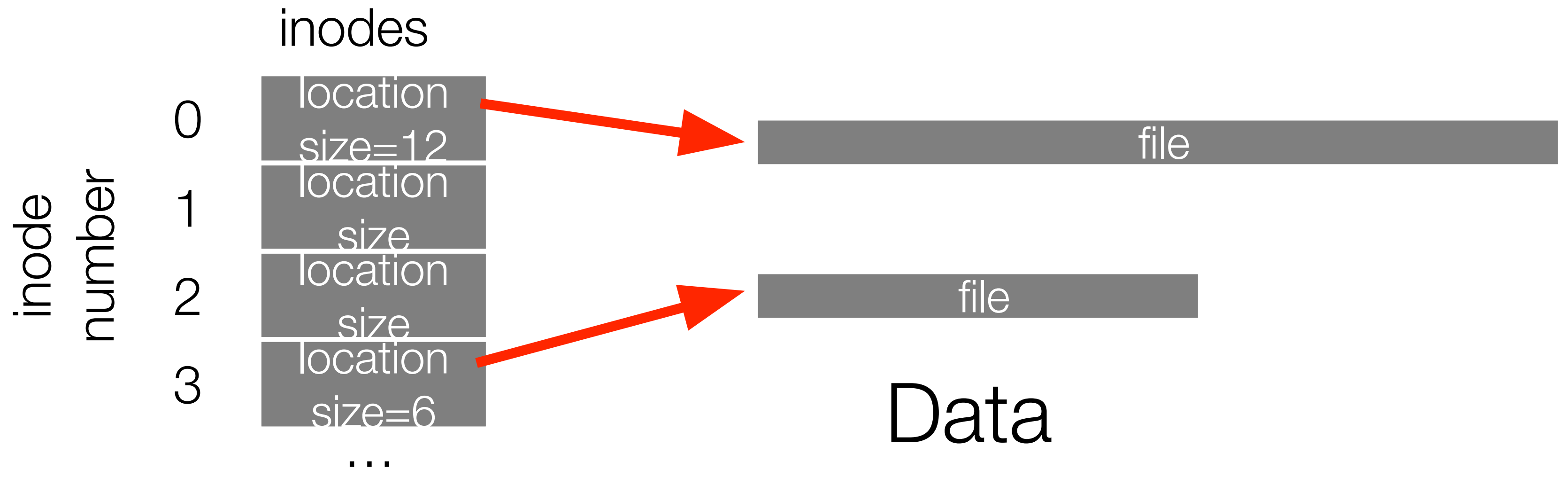
  Refers to collection of files

  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?

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?

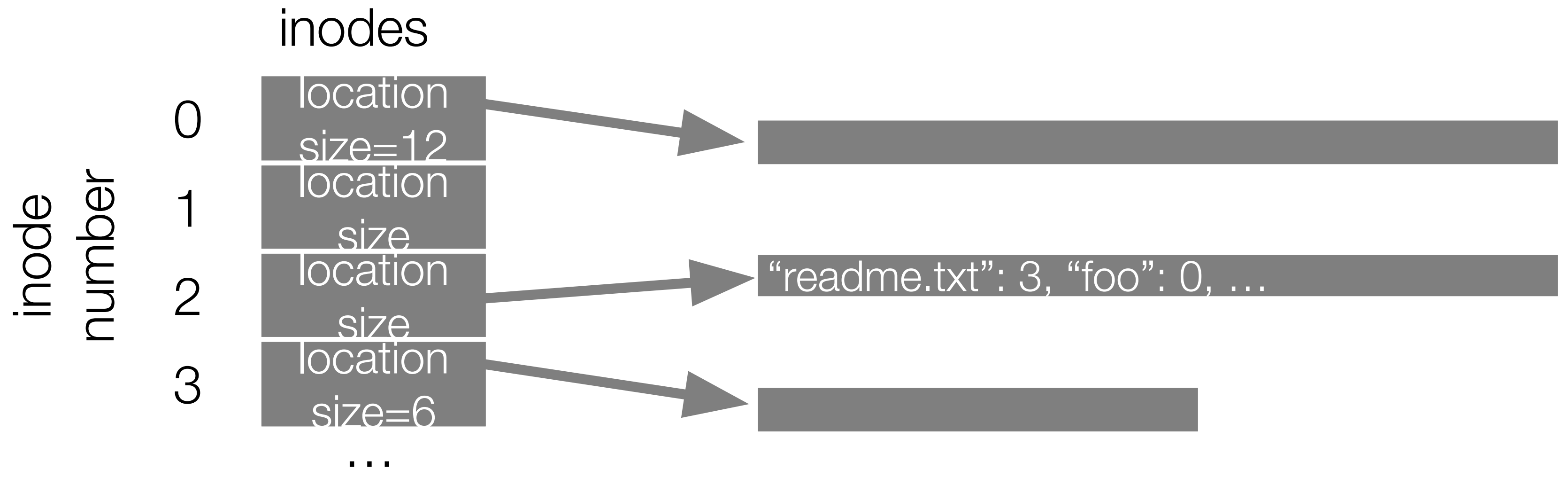
- names hard to remember
- no organization or meaning to inode numbers
- semantics of offset across multiple processes?

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, “foo”: 0, …

Paths

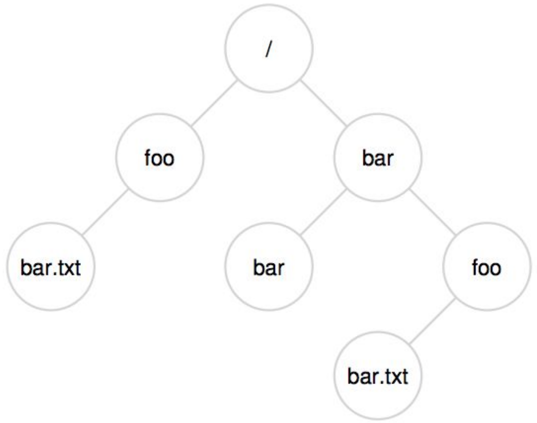
Directory Tree instead of single root directory

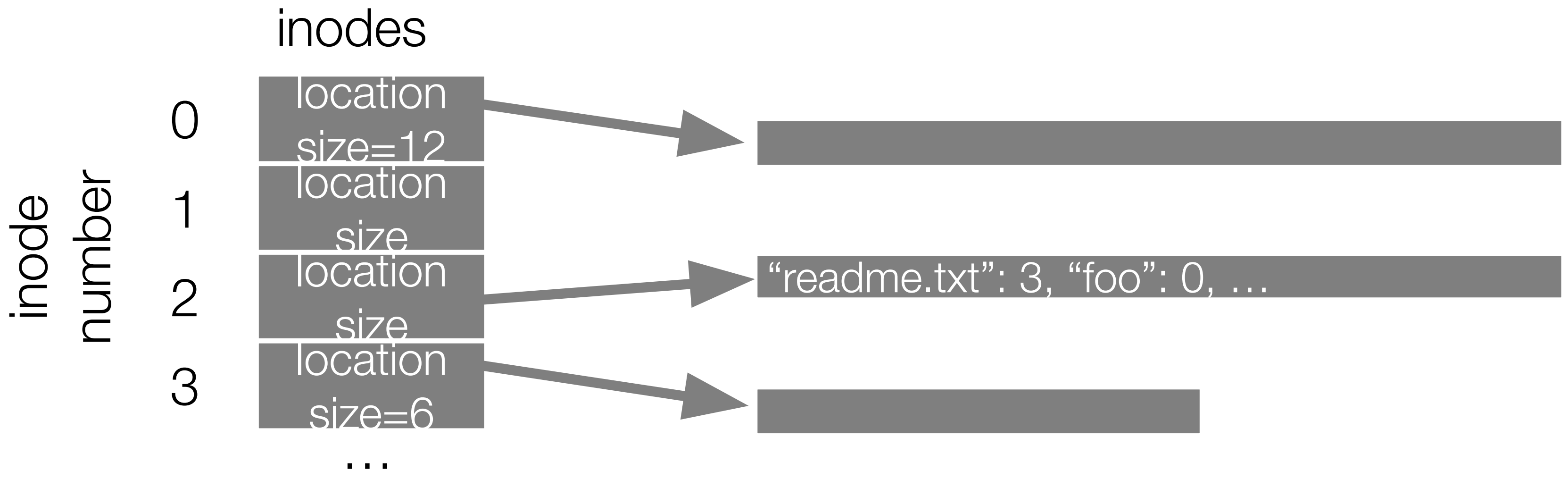
File name needs to be unique within a directory

/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

First time a process opens a file, what will be the fd in Unix/Linux?

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

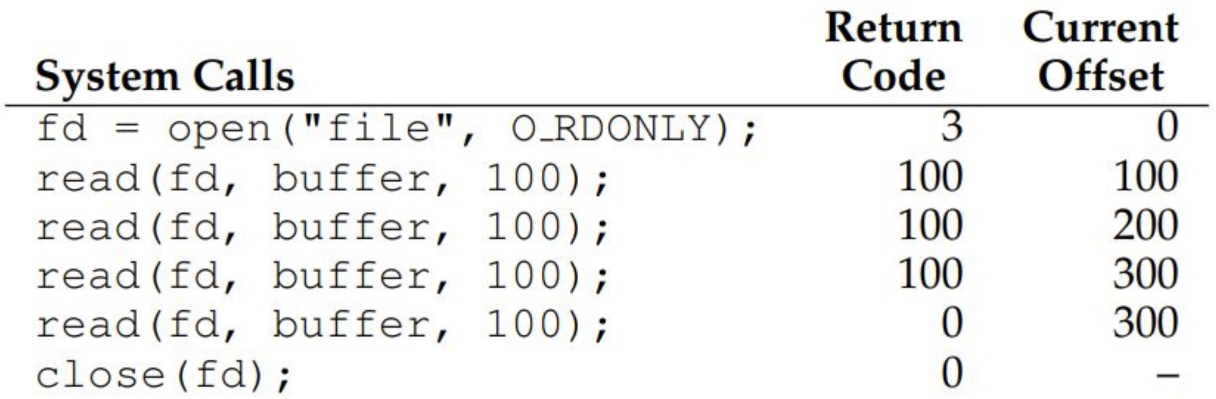
FD Table (xv6)

```
struct file {
  int ref;
  char readable;
  char writable;
  struct inode *ip;
  uint off;
};
```

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

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

FD offsets



System Calls	Return Code	Current Offset
`fd = open("file", O_RDONLY);`	3	0
`read(fd, buffer, 100);`	100	100
`read(fd, buffer, 100);`	100	200
`read(fd, buffer, 100);`	100	300
`read(fd, buffer, 100);`	0	300
`close(fd);`	0	–

FD Offsets

System Calls	Return Code	OFT[10] Current Offset	OFT[11] Current Offset
`fd1 = open("file", O_RDONLY);`	3	0	–
`fd2 = open("file", O_RDONLY);`	4	0	0
`read(fd1, buffer1, 100);`	100	100	0
`read(fd2, buffer2, 100);`	100	100	100
`close(fd1);`	0	–	100
`close(fd2);`	0	–	–

LSEEK and READ

```
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.

Assume head is on track 1
Suppose we do lseek to X and the sector for X is on track 4
Where is head immediately after lseek?

Entries in OFT

When a process opens its first file (say whose inode is 10),

What will be the values in the file struct?

```
struct file {
  int ref;
  char readable;
  char writable;
  struct inode *ip;
  uint off;
};
```

What if another process opens the same file?

How will the values inside file struct change?

Shared Entries in OFT

Fork:

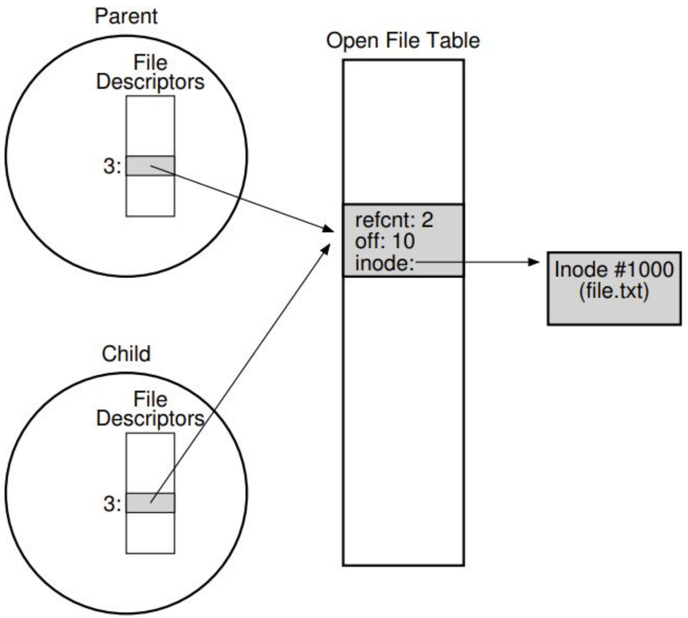
```
int main(int argc, char *argv[]) {
    int fd = open("file.txt", O_RDONLY);
    assert(fd >= 0);
    int rc = fork();
    if (rc == 0) {
        rc = lseek(fd, 10, SEEK_SET);
        printf("child: offset %d\n", rc);
    } else if (rc > 0) {
        (void) wait(NULL);
        printf("parent: offset %d\n",
               (int) lseek(fd, 0, SEEK_CUR));
    }
    return 0;
}
```

What is the parent trying to print?

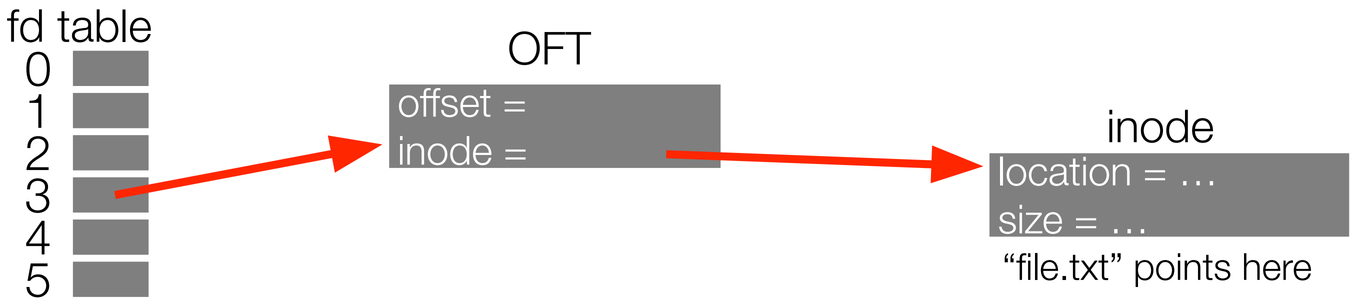
What value will be printed?

Shared Entries in OFT

What's happening here?



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
```

DUP

```
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);
```

How many entries in the OFT (assume no other process)?

Offset for fd1?

Offset for fd2?

Offset for fd3

Fsync

File system keeps newly written data in memory for a while

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

Rename

rename(char *old, char *new):

Do we need to copy/move data?

How does the FS implement this?

Does it matter whether the old and new names are in the same directory or different directories?

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?

Paths and Links

(Hard) Link

Inode has a field called "nlinks"

When is it incremented?

When is it decremented?

Deleting Files

What is the 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

Symbolic or soft links

A different type of link

Hard links don’t work with directory and cannot be cross-FS

touch foo; echo hello > foo;

Hardlink: ln foo foo2

Stat foo; what will be the size and inode?

Stat foo2; what will be the size and inode?

Softlink: ln –s foo bar

Stat bar; what will be the size and inode?

Atomic File Update

Say you want to update file.txt atomically

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

How to do?

Next Lecture

File system implementation

A simple ext2 like FS

Then, fsck, journaling, FFS, …