CS 423
Operating System Design: Persistence: RAID, FS API
Apr 9

Ram Kesavan

LOGISTICS

MP3 due by Apr 15, 11:59 CT

Midterm Regrades should be done by early next week

AGENDA / LEARNING OUTCOMES

RAID

- 0, 1, 4, 5

File system API

- Basics of files, directories
- Data structures to track “open” files
  - per-process & system-wide

RECAP

I/O SCHEDULING IN DEVICE

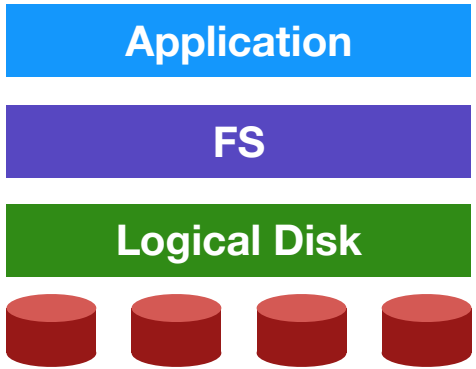
Given a stream of R/W requests, re-order for optimal perf:

- Scheduler in the HDD/SSD firmware
  - OSTEP Ch 37 discusses different scheduling algorithms (HDD only)
  - HDD: optimize for arm movement
  - SSD: optimize for FTL wear-leveling

	HDD		SSD	
	Random	Sequential	Random	Sequential
Access	2-15 ms	2 ms	50-150 us	35-500 us
R Throughput	1-3 MB/s	80-160 MB/s	30-500 MB/s	500-550 MB/s
W Throughput	0.5-1 MB/s	80-160 MB/s	0.2-3 GB/s	450-500 MB/s

SOLUTION 2: RAID

Build logical disk from many physical disks.



Logical disk gives

capacity,

performance,

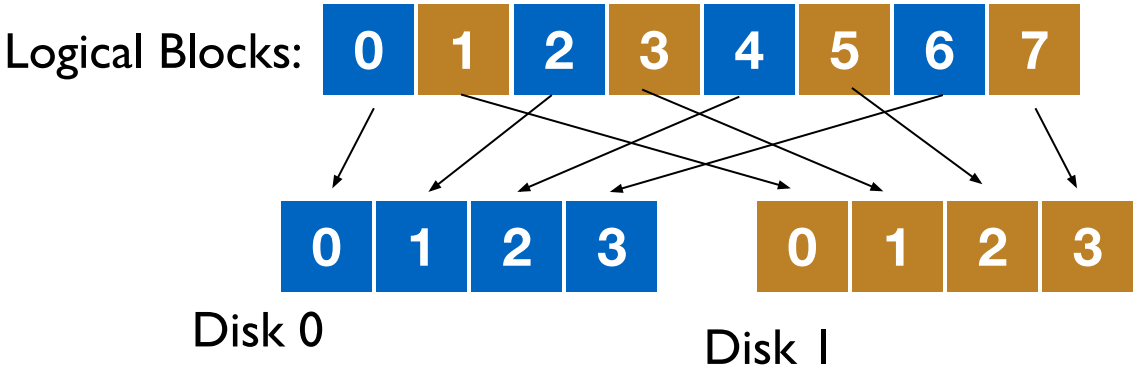
reliability

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

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

RAID-0

Optimize for capacity. No redundancy



RAID 0: STRIPES AND CHUNK SIZE

	Disk 0	Disk 1	Disk 2	Disk 4
	0	1	2	3
stripe:	4	5	6	7
	8	9	10	11
	12	13	14	15

Chunk size = 1block

assume chunk size of 1 for this lecture

	Disk 0	Disk 1	Disk 2	Disk 4
	0 1	2 3	4 5	6 7
stripe:	8 9	10 11	12 13	14 15

Chunk size = 2 blocks

Stripe: blocks/chunks on each disk at the same LBA

RAID-0: ANALYSIS

Capacity: available to file system?

N * C

Reliability: how many failed disks can be tolerated?

0

Latency (random):

D

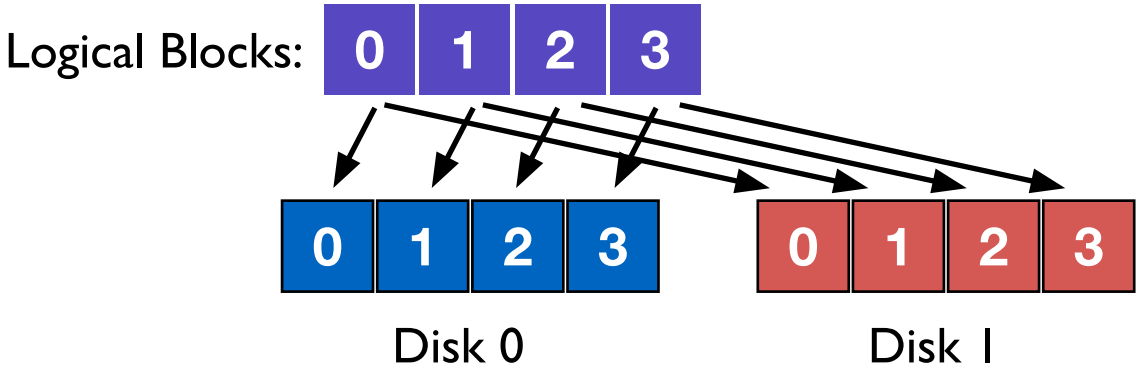
Throughput (sequential, random):

N*S, N*R

More disks improves throughput not latency

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

RAID-1: MIRRORING



Keep two copies of all data.

END RECAP

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

RAID-1: ANALYSIS

Capacity: available to file system?

Reliability: how many failed disks can be tolerated?

Latency (random):

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

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

RAID-1: ANALYSIS

Capacity: available to file system?

N/2 * C

Reliability: how many failed disks can be tolerated?

1 (or a very lucky N/2!)

Latency (random):

D

In practice random write would be slightly higher–why?

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

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

RAID-1: THROUGHPUT

What is steady-state throughput for

- random reads?

- random writes?

- sequential writes?

- sequential reads?

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

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

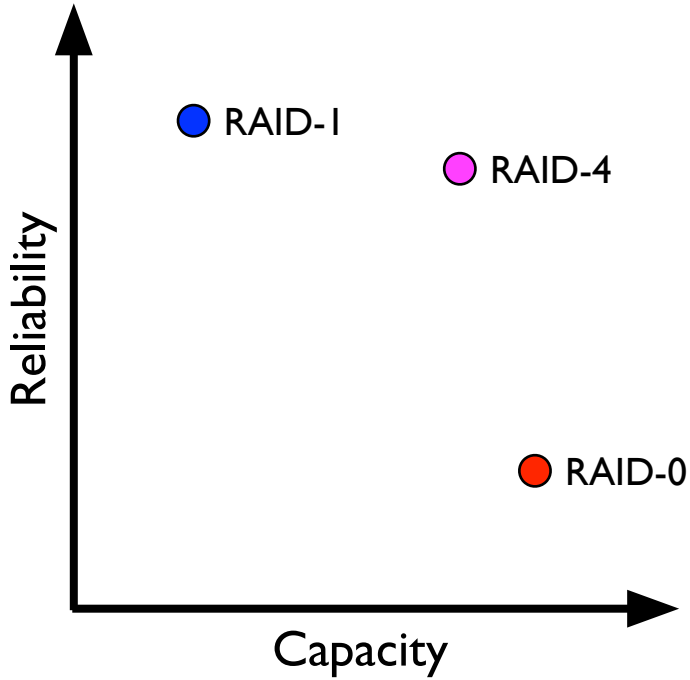
RAID-1: THROUGHPUT

What is steady-state throughput for

- random reads?
  N * R
- random writes?
  N/2 * R
- sequential writes?
  N/2 * S
- sequential reads?
  N/2 * S
  can achieve N*S with larger chunk size
  or clever scheduling

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

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


RAID-1
RAID-4
RAID-0
Reliability
Capacity

RAID-4: STRATEGY

Concept of a **parity** disk

- Each **RAID group** has n data disks + 1 parity disk

Build a math equation across blocks in a stripe

- Each stripe has n + 1 blocks (chunk size = 1)
- Equation has n + 1 variables
- If any n are known, can solve for the missing unknown

The math: ideally cheap to compute, parallelizable

Block on a *bad* disk would be the missing unknown

RAID 4: EXAMPLE

RAID Group: (4,1)
4 data & 1 parity

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 function to compute parity?

Let's assume a block is just a single bit for now!

RAID 4: EXAMPLE

RAID Group: (4,1)

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 function 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

Given C0, C2, C3, and P, how to compute C1?

C1 = XOR(C0, C2, C3, P)
Row0 XOR(0, 1, 1, 0) = 0
Row1 XOR(0, 0, 0, 1) = 1

How does it work if chunk size is 1 or more blocks?

RAID-4: ANALYSIS

Capacity: available to file system?

Reliability: how many failed disks can be tolerated?

Latency (random read):

Latency (random write):

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

RAID-4: ANALYSIS

Capacity: available to file system?

(N - 1) * C

Reliability: how many failed disks can be tolerated?

1 (any of them)

Latency (random read):

D

Latency (random write):

2 * D (assume parity math is fast)

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

RAID-4: THROUGHPUT

Steady-state throughput for

- sequential reads?
- sequential writes?
- random reads?
- random writes?

Important: I/O cost is >> parity computation cost

RAID-4: THROUGHPUT

Steady-state throughput for

- sequential reads? (N - 1) * S
- sequential writes? (N - 1) * S
- random reads? (N - 1) * R
- random writes? R/2

Important: I/O cost is >> parity computation cost

PARITY MATH: ADDITIVE VS SUBTRACTIVE

Write a new C1

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

Additive:

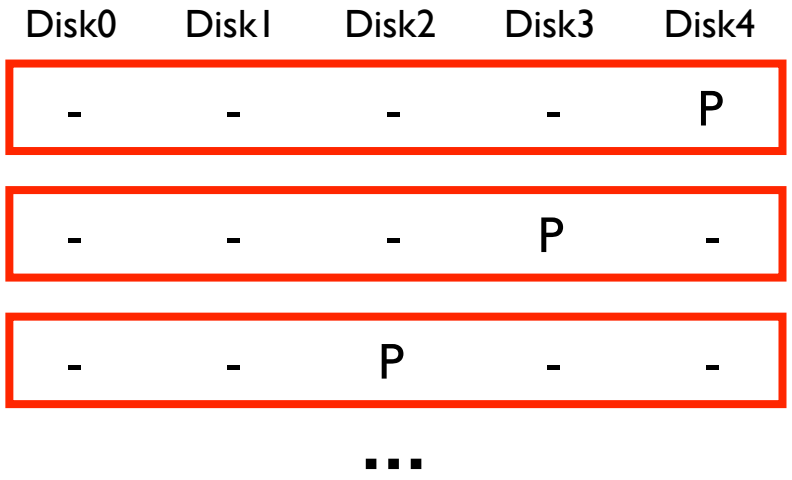
P_{new} = $XOR(C0_{old}, C1_{new}, C2_{old}, C3_{old})$

Subtractive:

$P_{new} = XOR(C1_{old}, C1_{new}, P_{old})$

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-5



Parity disk is the bottleneck: rotate parity across the disks

Parity block location ~= stripe-number modulo n

RAID-5: ANALYSIS

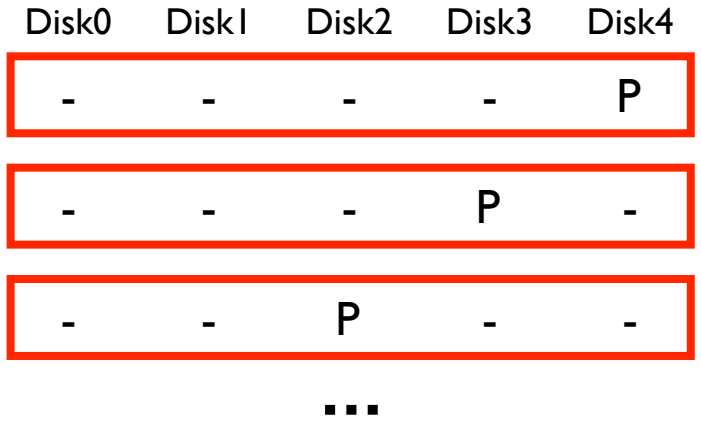
Capacity: available to file system?

Reliability: how many failed disks can be tolerated?

Latency (random read):

Latency (random write):

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



RAID-5: ANALYSIS (SAME AS RAID-4)

Capacity: available to file system?

(N - 1) * C

Reliability: how many failed disks can be tolerated?

1 (any disk)

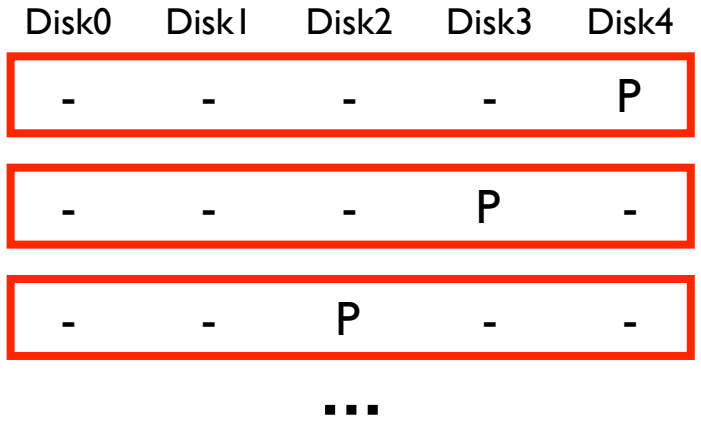
Latency (random read):

D

Latency (random write):

2 * D

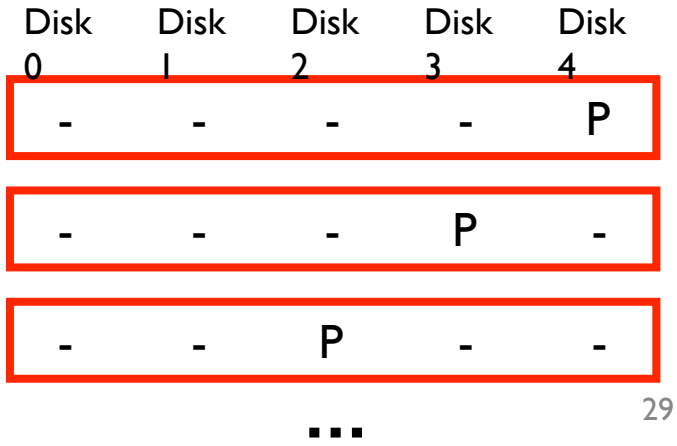
N: #disks
C: capacity of 1 disk
S/R: sequential/random throughput of 1 disk
D: latency of one small I/O operation



RAID-5: THROUGHPUT

What is steady-state throughput for RAID-5?

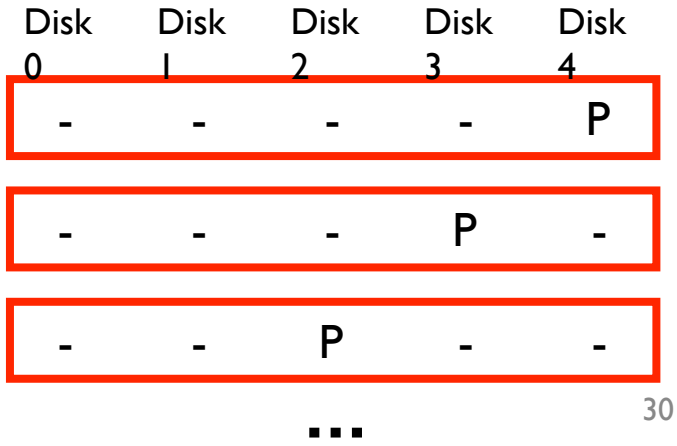
- sequential reads?
- sequential writes?
- random reads?
- random writes?



RAID-5: THROUGHPUT

Steady-state throughput for RAID-5?

- sequential reads? (N - 1) * S
- sequential writes? (N - 1) * S
- random reads? N * R
- random writes? N * R/ 4



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 4 & 5: RECONSTRUCTION

Failed disk needs to be reconstructed

- A spare disk is used to replace the failed disk
- Parity is used to reconstruct each block in that disk

In the meantime, what happens to reads/writes to that disk?

How fast to run reconstruction?

- Too slow → ?
- Too fast → ?

RAID 4 & 5: RECONSTRUCTION

Failed disk needs to be reconstructed

- A spare disk is used to replace the failed disk
- Parity is used to reconstruct each block in that disk

In the meantime, what happens to reads/writes to that disk?

- read: reconstruct-on-the-fly, if it's not already reconstructed
- write: just write the block (no change necessary)

How fast to run reconstruction?

- Too slow → another disk may fail
- Too fast → interferes with application reads/writes

RAID: SUMMARY

RAID: a faster, larger, more reliable disk system

- Used with HDDs or SSDs
- Typically, all disks are similar & of the same size
- Other levels: RAID 10, RAID 01
- RAID 2, RAID 3 (ignore)
- RAID-6, DP (double parity), TP (triple parity)

Single block numbering built over many physical storage devices

Different mapping and redundancy schemes

Different trade-offs

POP QUIZ

https://forms.gle/z86vhxZkrZTuxAD99

If you're done with the pop quiz, here's a fun problem:

When writing to a stripe, writes are issued to many disks in the RAID Group. A random subset of the writes may fail due to disk/system failure, which results in a broken parity equation for that stripe.

How to fix the stripe's consistency:

1. Recompute & write out the parity?
2. Log some info that we can use to recover consistency?

NEXT

Thus far, bottom-up stuff:

- Storage devices
- Storage media
- RAID

Next, top-down stuff:

- Files and File system API
- Different trade-offs

After that: file system

Files & File System

File: sequence of persisted bytes

- sections of a file may be “holes”

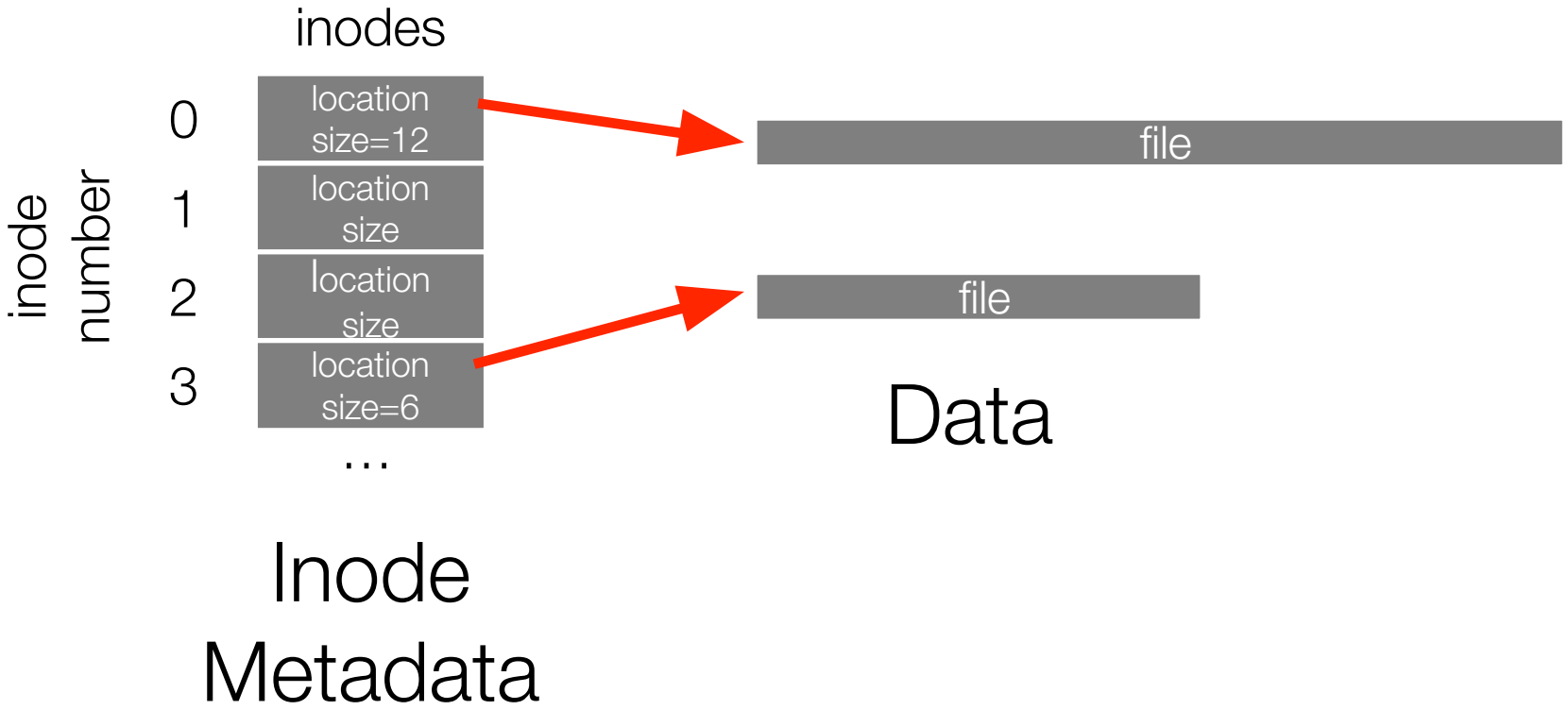
File system refers to both:

- a collection of files (mounted/unmounted together)
- module of the OS that manages this collection of files

Name/address for a file:

- unique id (within the file system): inode number aka inum
- path
- file descriptor

Inodes & Files



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?

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

File Path Name

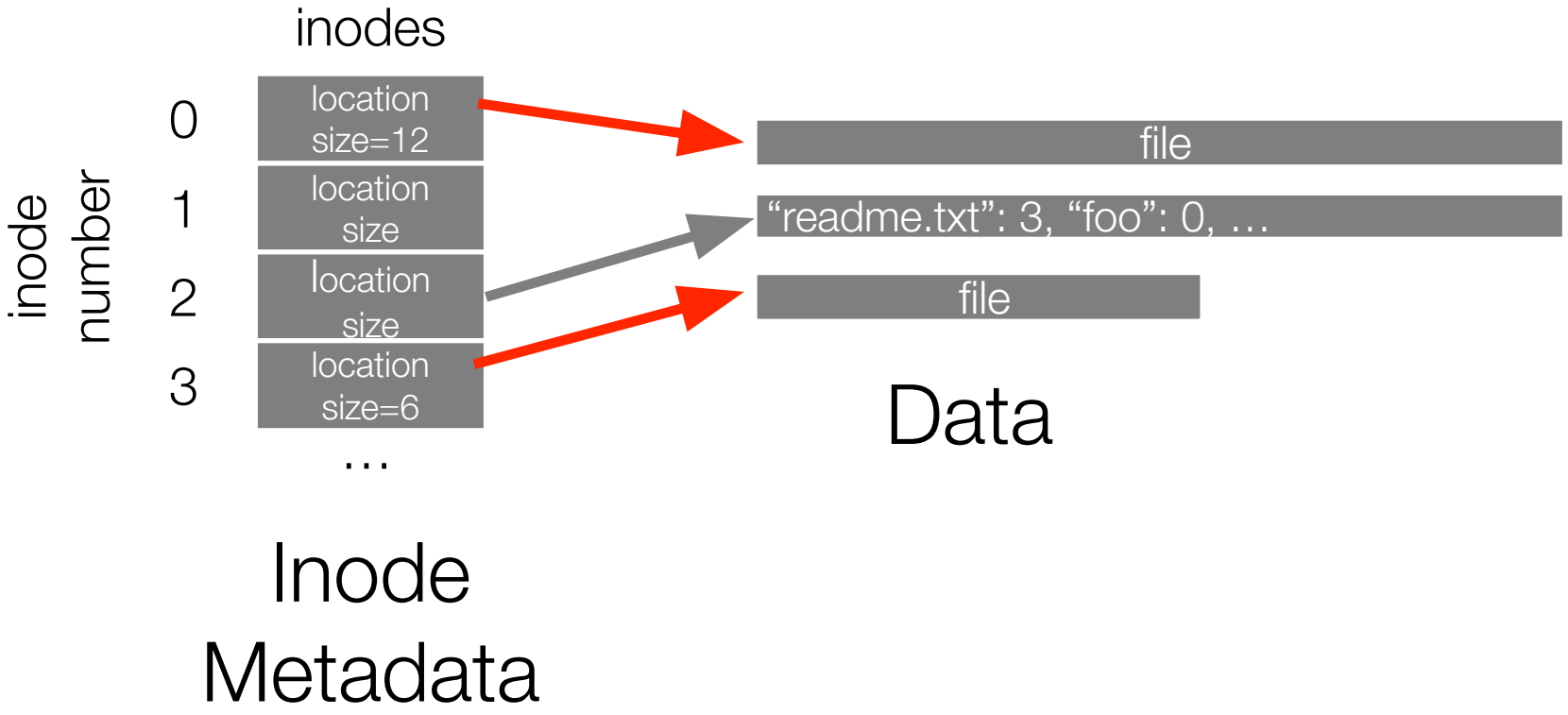
Humans prefer to:

- Use string names instead of inums
- Organize files into hierarchical structures

The file system internally uses only inums

It stores *path-to-inode* mappings in special inodes called *directories (or folders).*

A directory inode



Paths

Directory Tree instead of single root directory

File name needs to be unique within a directory

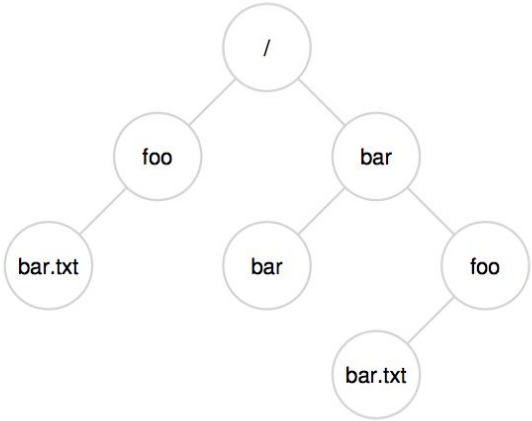
/usr/lib/file.so

/tmp/file.so

Directory stores filename-to-inode mapping

"Directory traversal" needed to access a file

absolute vs relative pathname



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)
```

Disadvantage?

Traversal is expensive: /bar/foo/bar.txt

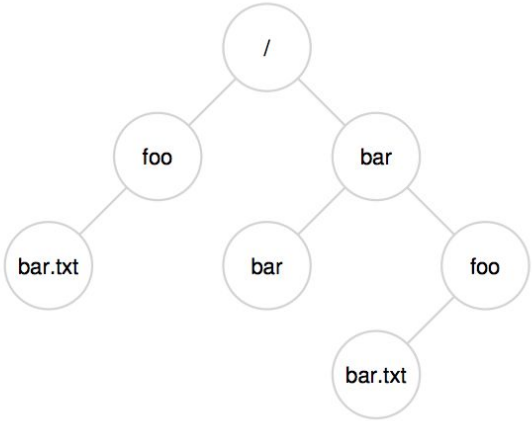
load "/" inode (a fixed inode), search for "bar"

load "bar" inode, search for "foo"

load "foo" inode, search for "bar.txt"

load "bar.txt" inode

Goal: traverse once



File Descriptor (fd)

Solution:

Expensive traversal done once (to **open** file)

The open API returns a file descriptor (fd), which is an int

All subsequent API calls (reads/writes/etc) use fd

An opaque handle that hides the inum within it, and current offset

Each process has a file descriptor table:

maps each fd (handed out) to a pointer into a system-wide file descriptor table

Special file descriptors: 0 (stdin), 1 (stdout), 2 (stderr)

UNIX hands out file descriptors starting from 3

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 & 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	–

OFT: Open File Table to track all open files

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	–	–