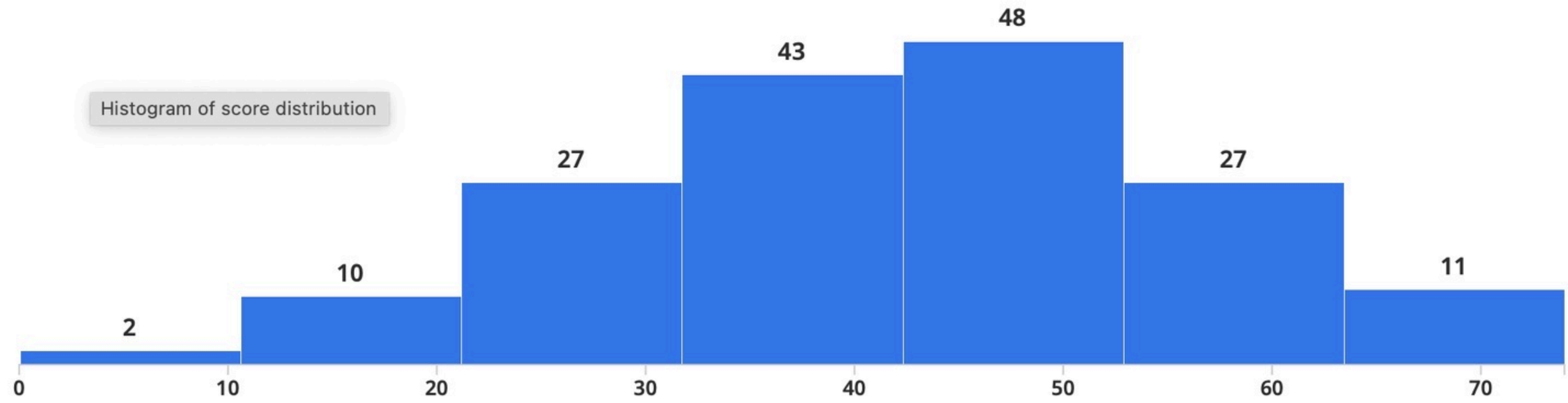
Prelim 2

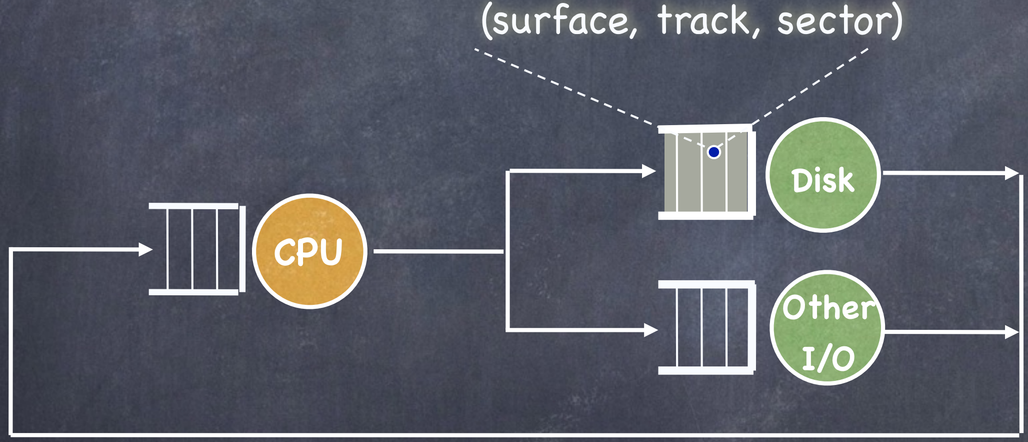


Minimum	Median	Maximum	Mean	Std Dev
7.5	42.75	74.0	42.18	13.93

Median: B

Disk Head Scheduling

- In a multiprogramming/time sharing environment, a queue of disk I/Os can form



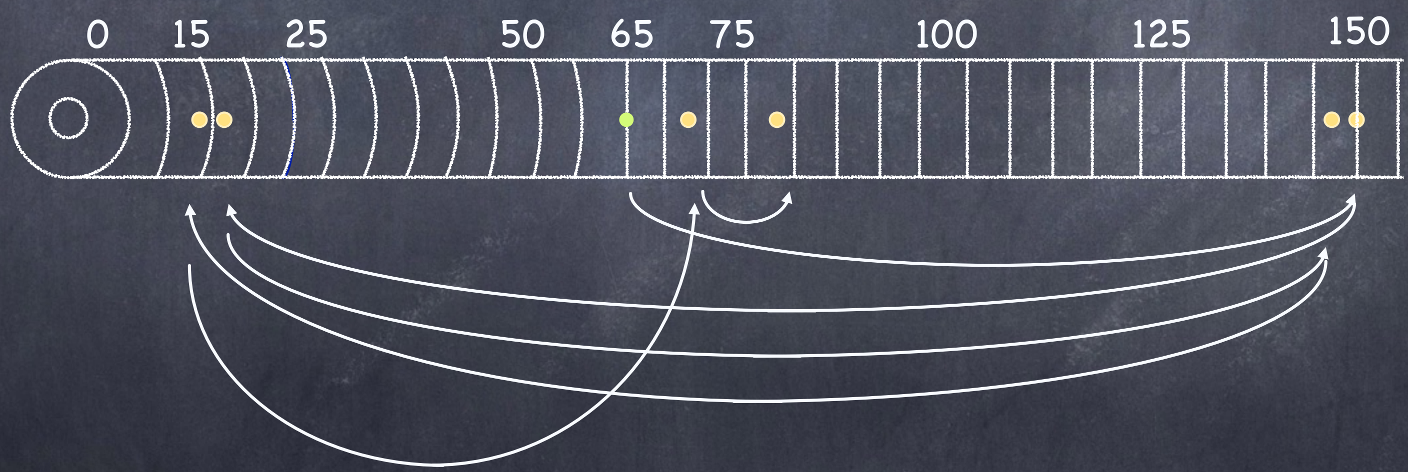
Read about disk scheduling algorithms in class notes and in Chapter 37 of 3 Easy Pieces

- OS maximizes disk I/O throughput by minimizing head movement through disk head scheduling
  - and this time we have a good sense of tasks' length!

FCFS

- Assume a queue of request exists to read/write tracks

and the head is on track 65



FCFS scheduling results in disk head moving 550 tracks

and makes no use of what we know about the length of the tasks!

SSTF:
Shortest Seek Time First

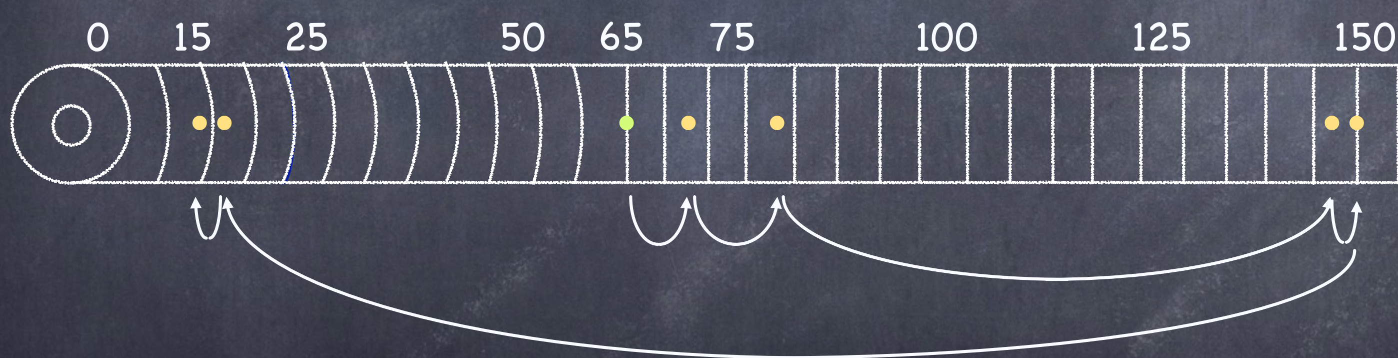
- Greedy scheduling

Rearrange queue from:

....	83	72	14	147	16	150

to:

....	14	16	150	147	83	72



Head moves 221 tracks BUT

- OS knows blocks, not tracks (easily fixed)
- starvation

SCAN Scheduling
"Elevator"

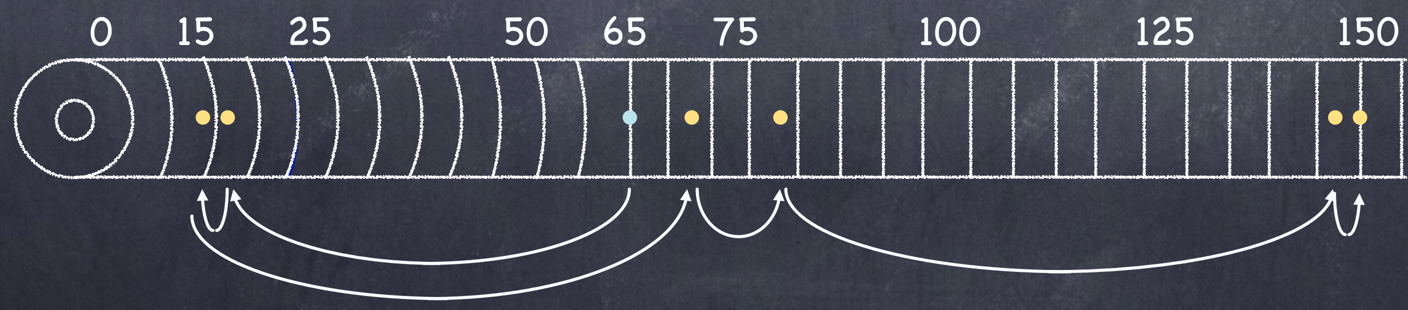
- Move the head in one direction until all requests have been serviced, and then reverse
  - sweeps disk back and forth

Rearrange queue from:

....	83	72	14	147	16	150

to:

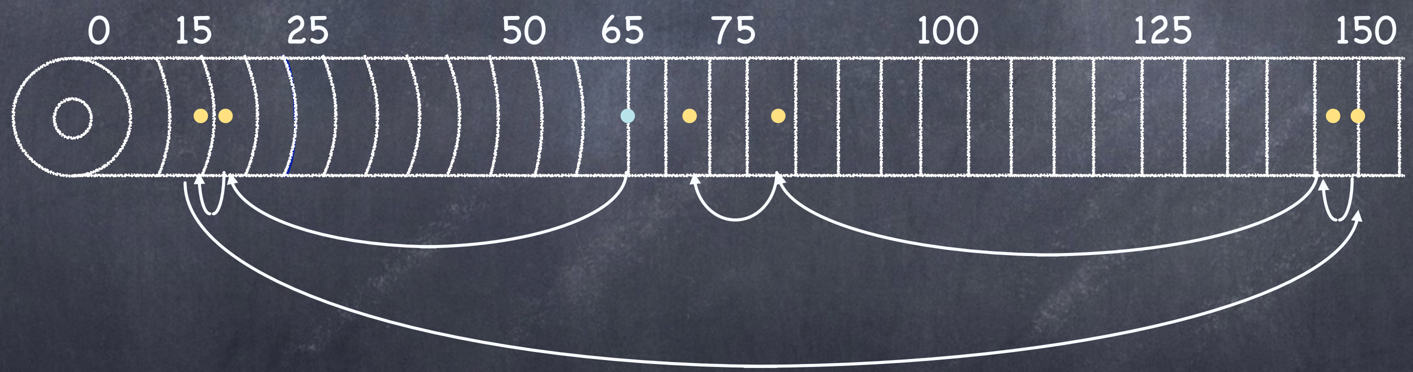
....	150	147	83	72	14	16



Head moves 187 tracks.

C-SCAN scheduling

- Circular SCAN
  - sweeps disk in one direction (from outer to inner track), then resets to outer track and repeats



- More uniform wait time than SCAN
  - moves head to serve requests that are likely to have waited longer

OS Outsources Scheduling Decisions

- Selecting which track to serve next should include rotation time (not just seek time!)
  - SPTF: Shortest Positioning Time First
- Hard for the OS to estimate rotation time accurately
  - Hierarchical decision process
    - OS sends disk controller a batch of “reasonable” requests
    - disk controller makes final scheduling decisions

Back to Storage…

What qualities we want from storage?

- Reliable: It returns the data you stored
- Fast: It returns the data you stored promptly
- Affordable: It does not break the bank
- Plenty: It holds everything you need

What we may instead get is a SLED!

- Single, Large, Expensive Disk

Read about disk scheduling algorithms in class notes and in Chapter 38 of 3 Easy Pieces

RAID

Redundant Array of Inexpensive* Disks

* In industry, "inexpensive" has been replaced by "independent" :-)

E Pluribus Unum

- Implement the abstraction of a faster, bigger and more reliable disk using a collection of slower, smaller, and more likely to fail disks
  - different configurations offer different tradeoffs
- Key feature: transparency
  - The Power of Abstraction™
  - to the OS looks like a single, large, highly performant and highly reliable single disk (a SLED, hopefully with lower-case "e"!)
    - a linear array of blocks
    - mapping needed to get to actual disk
    - cost: one logical I/O may translate into multiple physical I/Os
- In the box:
  - microcontroller, DRAM (to buffer blocks) [sometimes non-volatile memory, parity logic]

Failure Model

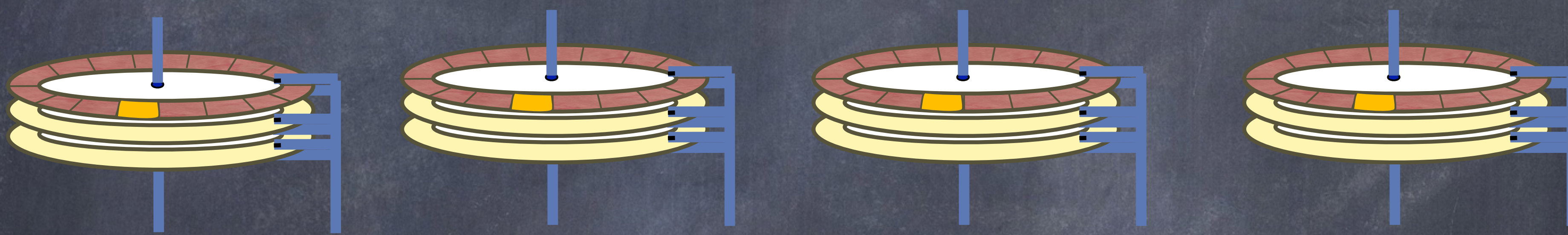
- RAID adopts the strong, somewhat unrealistic Fail-Stop failure model (electronic failure, wear out, head damage)
  - component works correctly until it crashes, permanently
    - disk is either working: all sectors can be read and written
    - or has failed: it is permanently lost
  - failure of the component is immediately detected
    - RAID controller can immediately observe a disk has failed and accesses return error codes
- In reality, disks can also suffer from isolated sector failures
  - Permanent: physical malfunction (magnetic coating, scratches, contaminants)
  - Transient: data is corrupted, but new data can be successfully read from/written to sector

How to Evaluate a RAID

- Capacity
  - what fraction of the sum of the storage of its constituent disks does the RAID make available?
- Reliability
  - How many disk faults can a specific RAID configuration tolerate?
- Performance
  - Workload dependent

RAID-0: Striping

Spread blocks across disks using round robin



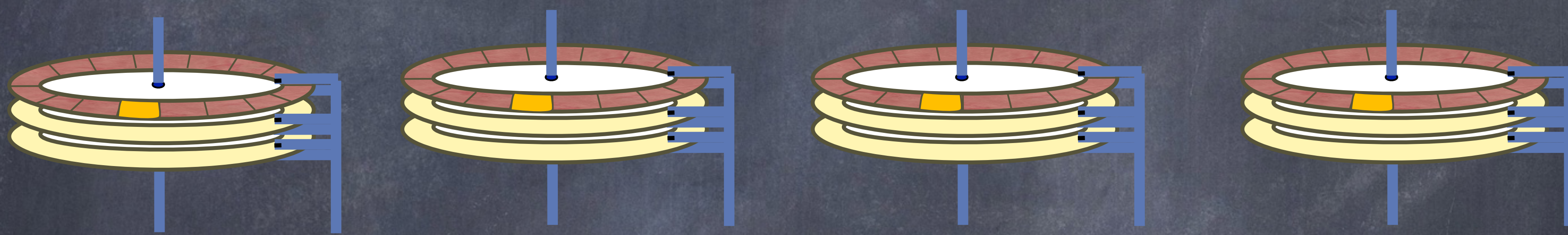
Stripe	0	1	2	3
Stripe	4	5	6	7
Stripe	8	9	10	11
Stripe	12	13	14	15

+ Excellent parallelism
  - can read/write from multiple disks

– Worst-case positioning time
  - wait for largest across all disks

RAID-0: Striping (Big Chunk Edition)

Spread blocks across disks using round robin



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

\+ improve positioning time — decrease parallelism

RAID-0: Evaluation

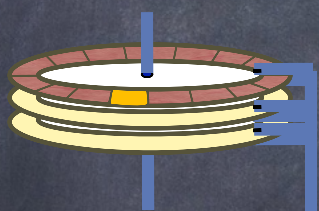
- Capacity
  - Excellent: N disks, each holding B blocks support the abstraction of a single disk with NxB blocks
- Reliability
  - Poor: Striping reduces reliability
    - Any disk failure causes data loss
- Performance
  - Workload dependent, of course
  - We'll consider two workloads
    - Sequential: single disk transfers S MB/s
    - Random: single disk transfer R MB/s
    - S >> R

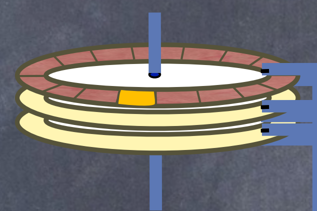
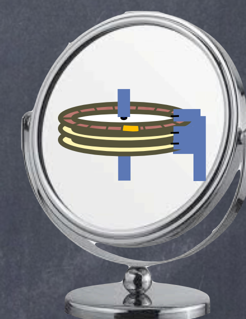
RAID-0: Performance

- Single-block read/write throughput
  - about the same as accessing a single disk
- Latency
  - Read: T ms (latency of one I/O op to disk)
  - Write: T ms
- Steady-state read/write throughput
  - Sequential: N x S MB/s
  - Random: N x R MB/s

RAID-1: Mirroring

Each block is replicated twice



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

Read from any

Write to both

RAID-1: Evaluation

- Capacity
  - Poor: N disks of B blocks yield (N x B)/2 blocks
- Reliability
  - Good: Can tolerate the loss (not corruption!) of any one disk
- Performance
  - Fine for reads: can choose any disk
  - Poor for writes: every logical write requires writing to both disks
    - suffers worst seek+rotational delay of the two writes

RAID-1: Performance

- Steady-state throughput
  - Sequential Writes: N/2 x S MB/s
    - Each logical Write involves two physical Writes
  - Sequential Reads: as low as N/2 x S MB/s

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

Suppose we want to read
0, 1, 2, 3, 4, 5, 6, 7

RAID-1: Performance

- Steady-state throughput
  - Sequential Writes: N/2 x S MB/s
    - Each logical Write involves two physical Writes
  - Sequential Reads: as low as N/2 x S MB/s

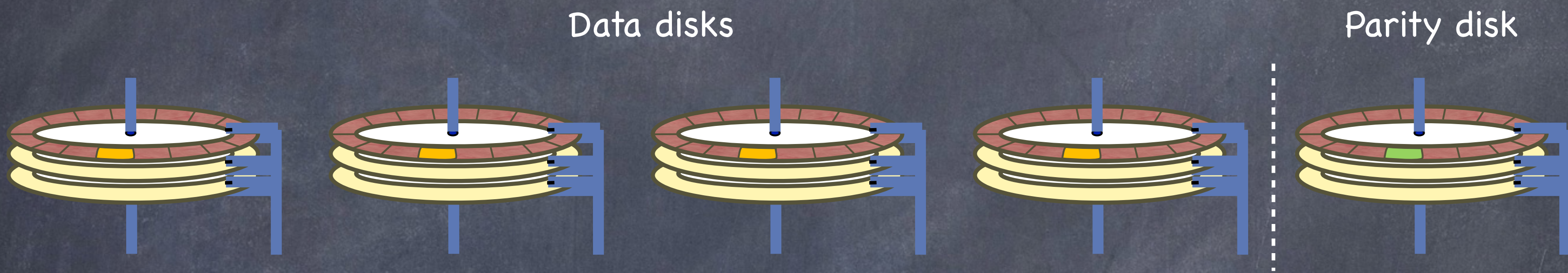
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Suppose we want to read
0, 1, 2, 3, 4, 5, 6, 7

Each disk only delivers half of his bandwidth:
half of its blocks are skipped!

  - Random Writes: N/2 x R MB/s
    - Each logical Write involves two physical Writes
  - Random Reads: N x R MB/s
    - Reads can be distributed across all disks
- Latency for Reads and Writes: T ms

RAID-4: Block Striped, with Parity



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

1	1	0
0	1	0
0	0	1

1	0	0
1	1	0
0	1	1

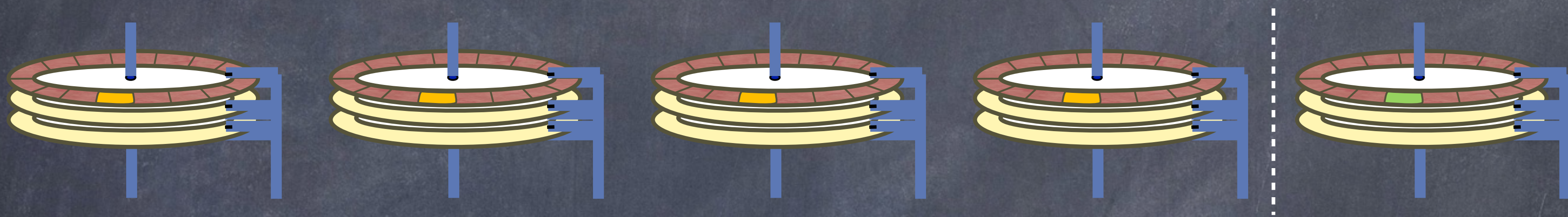
1	0	0
0	1	0
1	0	1

1	1	0
1	1	1
0	0	1

0	0	0
0	0	1
1	1	0

RAID-4: Block Striped, with Parity

Data disks   Parity disk



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

1	1	0
0	1	0
0	0	1

1	0	0
1	1	0
0	1	1

1	0	0
0	1	0
1	0	1

1	1	0
1	1	1
0	0	1

0	0	0
0	0	1
1	1	0

Disk controller can identify faulty disk

- single parity disk can detect and correct errors

RAID-4: Evaluation

- Capacity
  - N disks of B blocks yield (N-1) x B blocks
- Reliability
  - Tolerates the failure of any one disk
- Performance
  - Fine for sequential read/write accesses and random reads
  - Random writes are a problem!

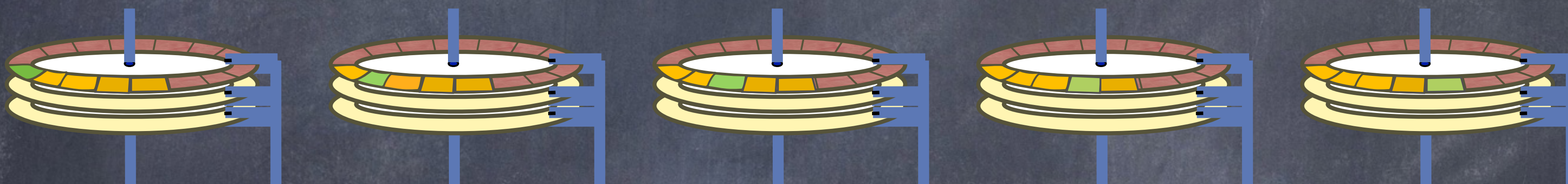
RAID-4: Performance

- Sequential Reads: (N-1) x S MB/s
- Sequential Writes: (N-1) x S MB/s
  - compute & write parity block once for the full stripe
- Random Read: (N-1) x R MB/s
- Random Writes: R/2 MB/s (N is gone! Yikes!)
  - need to read block B_{old} from disk and parity block P_{old}
  - Compute $P_{new} = (B_{old} \text{ XOR } B_{new}) \text{ XOR } P_{old}$
  - Write back B_{new} and P_{new}
  - Every write must go through parity disk, eliminating any chance of parallelism
  - Every logical I/O requires two physical I/Os at parity disk: can at most achieve 1/2 of its random transfer rate (i.e. R/2)

- Latency: Reads: T ms; Writes: 2T ms

RAID-5: Rotating Parity (avoids the bottleneck)

Parity and Data distributed across all disks



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-5: Evaluation

- Capacity & Reliability
  - As in Raid-4
- Performance
  - Sequential read/write accesses as in RAID-4
    - (N-1) x S MB/s
  - Random Reads are slightly better
    - N x R MB/s (instead of (N-1) x R MB/s)
  - Random Writes much better than RAID-4: R/2 x N/2
    - as in RAID-4 writes involve two operations at every disk: each disk can achieve at most R/2
    - but, without a bottleneck parity disk, we can issue up to N/2 writes in parallel (each involving 2 disks)

SSDs

Why care?

HDD

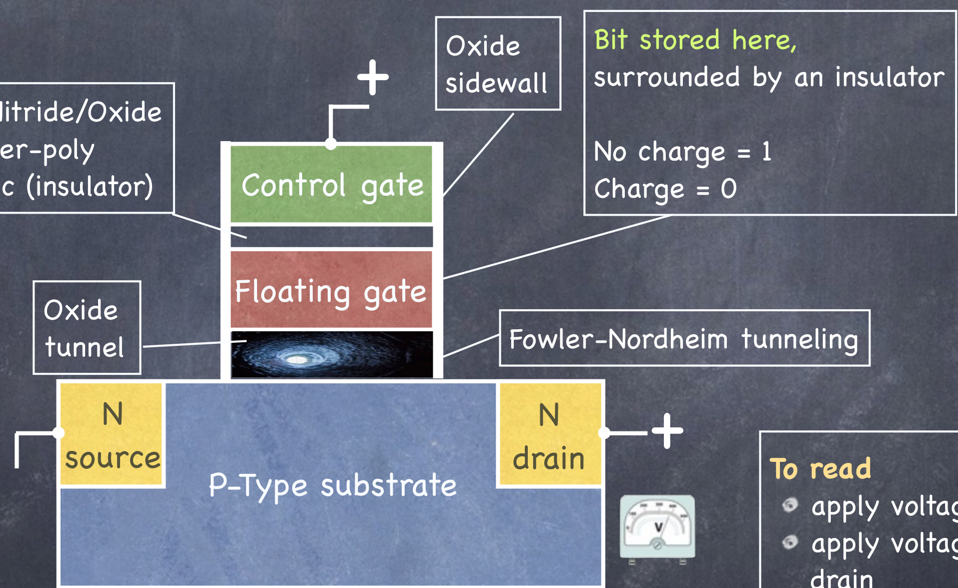
- Require seek, rotate, transfer on each I/O
- Not parallel (one active head)
- Brittle (moving parts)
- Slow (mechanical)
- Poor random I/O (10s of ms)

SSD

- No seeks
- Parallel
- No moving parts
- Random reads take 10s of μs
- Wears out!

Flash Storage

To write 0
- apply positive voltage to drain
- apply even stronger positive voltage to control gate
- some electrons are tunneled into floating gate



To write 1
- apply positive voltage to drain
- apply negative voltage to control gate
- electrons are forced out of floating gate into source

To read
- apply voltage to control gate
- apply voltage across source and drain
- measure current between source and drain to determine whether electrons in gate
  - if electrons in floating gate, must apply higher voltage to control gate to have current
  - measured current can encode more than a single bit

The SSD Storage Hierarchy

Cell
1 to 4 bits

Page
2 KB to 8 KB
not to be confused with a VM page

Block
64 to 256 pages
not to be confused with a disk block

Plane/Bank
Many blocks (Several Ks)

Flash Chip
Several banks that can be accessed in parallel

Basic Flash Operations

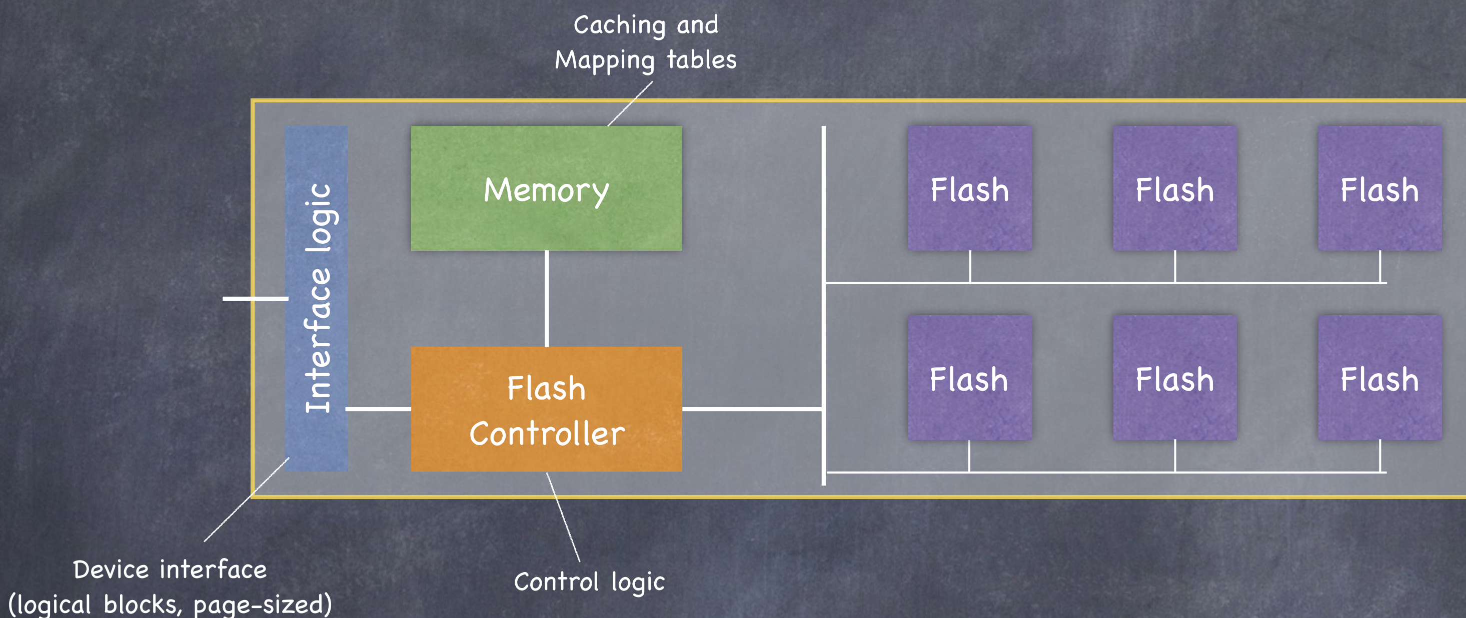
- Read (a page)
  - 10s of μs, independent of the previously read page
    - great for random access!
- Erase (a block)
  - sets the entire block (with all its pages) to 1 (!)
  - very coarse way to write 1s...
  - 1.5 to 2 ms (on a fast single level cell)
- Program (a page)
  - can change some bits in a page of an erased block to 0
  - 100s of μs
  - changing a 0 bit back to 1 requires erasing the entire block!

Using Flash Memory

- Need to map reads and writes to logical blocks to read, program, and erase operations on flash

Flash Translation Layer (FTL)

From Flash to SSD



- Flash Translation Layer
  - tries to minimize
    - write amplification: $\left[\frac{\text{write traffic (bytes) to flash chips}}{\text{write traffic (bytes) from client to SSD}}\right]$
    - wear out: practices wear leveling
    - disturbance: when many reads occur from pages of the same block, value of nearby cells can be affected

File Systems

The File System Abstraction

- Addresses need for long-term information storage:
  - store large amounts of information
  - do it in a way that outlives processes (RAM will not do)
  - can support concurrent access from multiple processes
- Presents applications with persistent, named data
- Two main components:
  - files
  - directories

The File

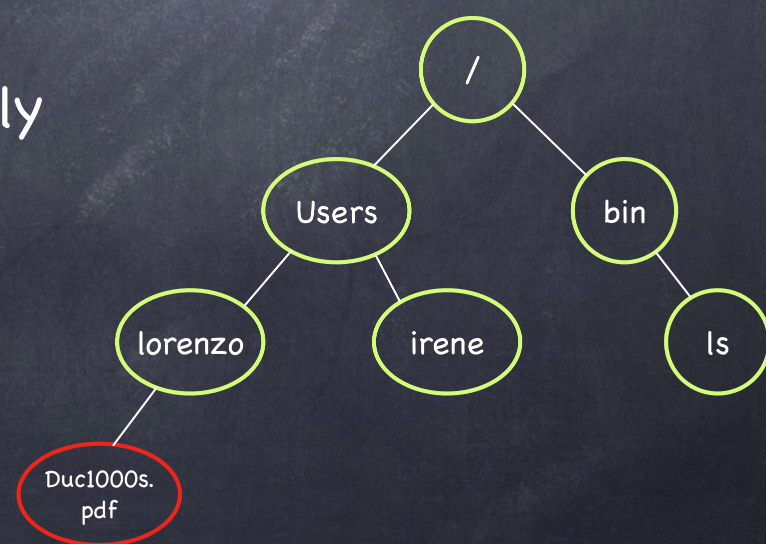
- A file is a named collection of data. In fact, it has many names, depending on context:
  - i-node number: low-level name assigned to the file by the file system
  - path: human friendly string
    - must be mapped to inode number, somehow
  - file descriptor
    - dynamically assigned handle a process uses to refer to i-node
- A file has two parts
  - data – what a user or application puts in it
    - array of untyped bytes
  - metadata – information added and managed by the OS
    - size, owner, security info, modification time, etc.

The Directory

- A special file that stores mappings between human-friendly names of files and their inode numbers

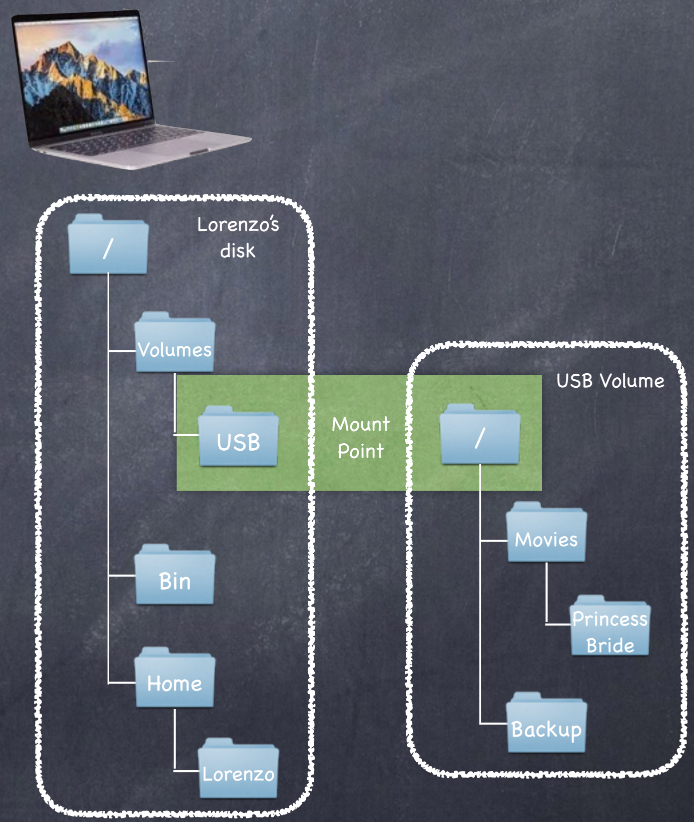
```
Argo% ls -i
2968458 Applications/        3123638 Dropbox (Old)/           4689728 Pictures/       4687176 gems/
2968461 Code/                3123878 Incompatible Software/   4687155 Public/         4687697 mercurial/
2968464 Desktop/             3123881 Library/                 4687159 Sites/          4687700 profiles.bin
2968978 Documents/           4687153 Mail/                    4687168 Synology/       4687701 src/
3121827 Downloads/           4689724 Movies/                  4687170 bin/            4689710 uninstall-mpi-cups.sh
3123562 Dropbox/             4689726 Music/                   4687175 fun/
Argo%
```

  - Has its own inode, of course
  - Mapping may of course also apply to human-friendly names of directories and their inodes
    - directory tree
    - / indicates the root



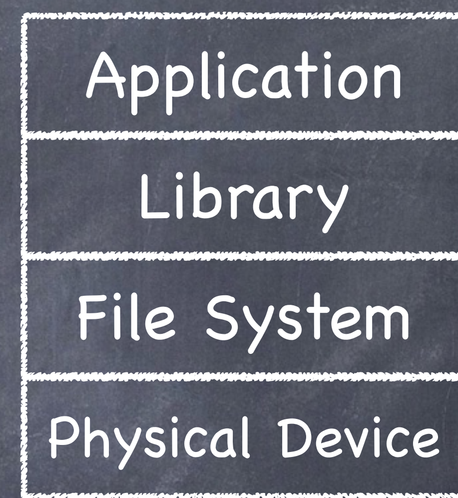
Mount

- Mount: allows multiple file systems on multiple volumes to form a single logical hierarchy
  - a mapping from some path in existing file system to the root directory of the mounted file system



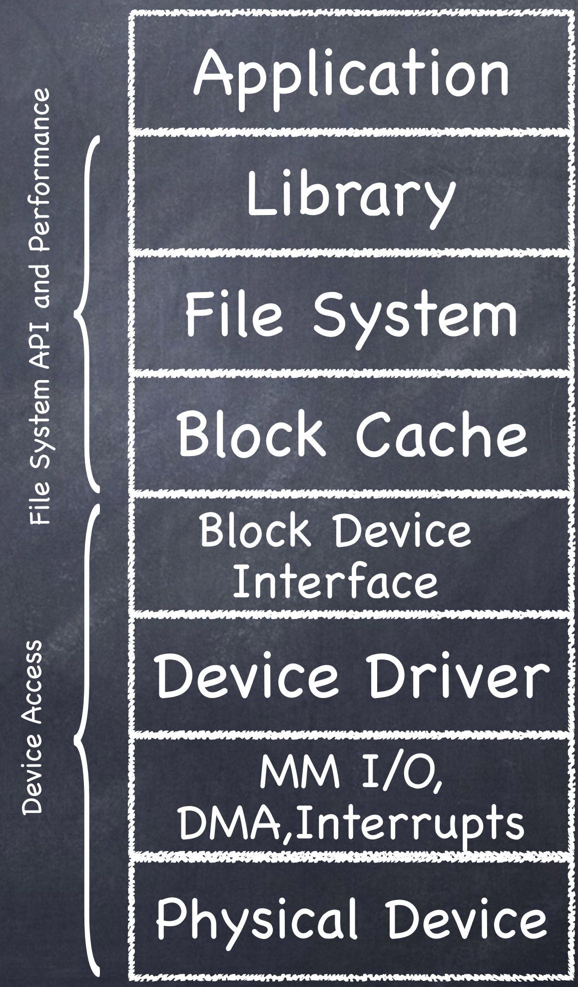
The Abstraction Stack

- I/O systems are accessed through a series of layered abstractions



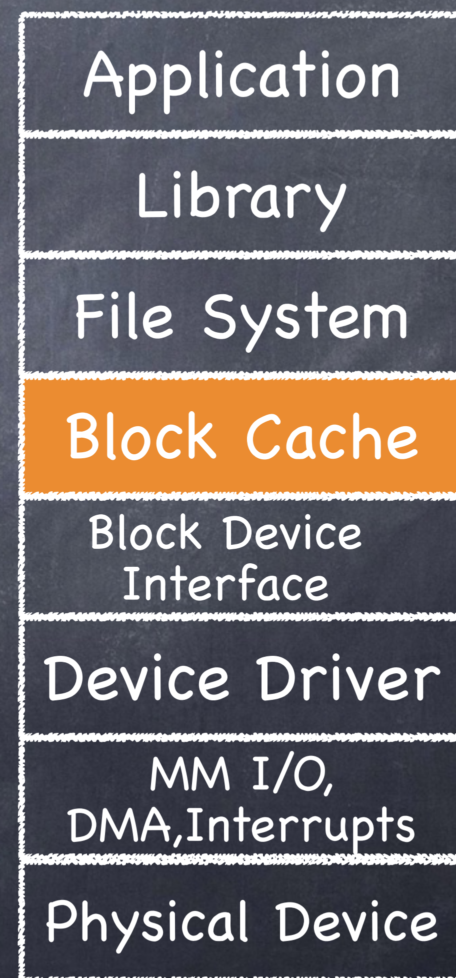
The Abstraction Stack

- I/O systems are accessed through a series of layered abstractions



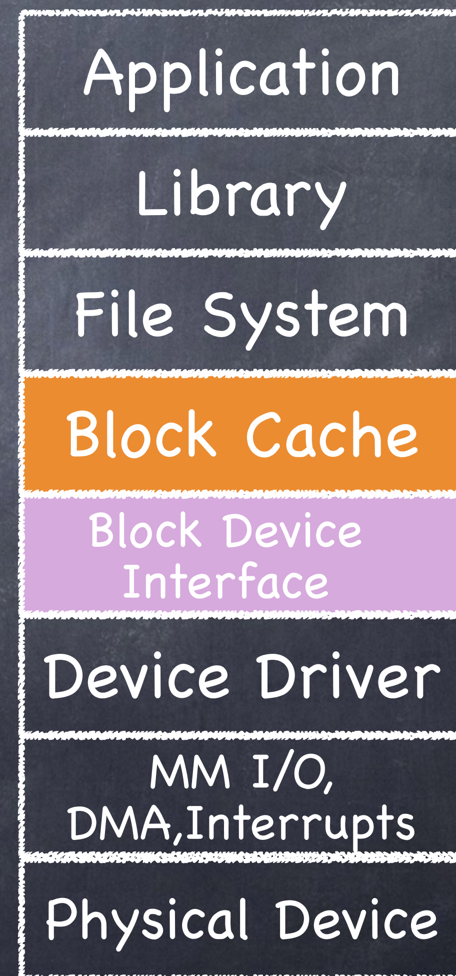
The Abstraction Stack

- I/O systems are accessed through a series of layered abstractions
  - Caches blocks recently read from disk
  - Buffers recently written blocks



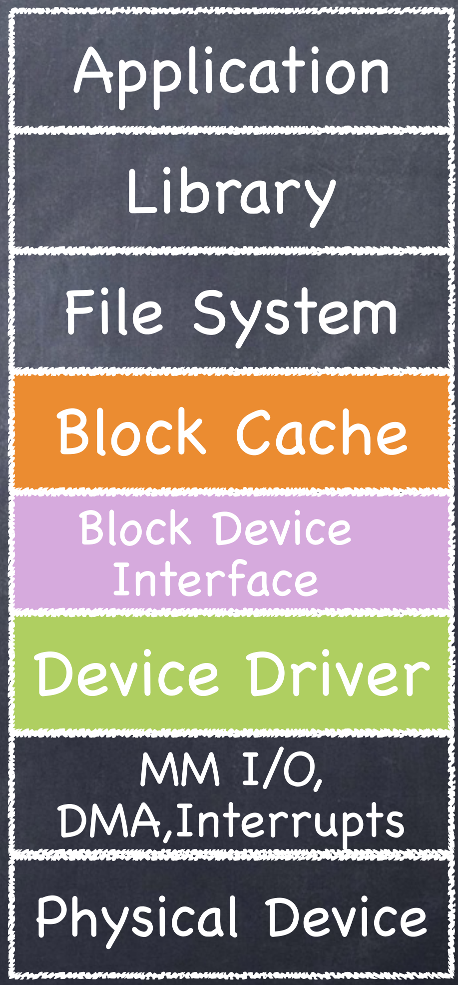
The Abstraction Stack

- I/O systems are accessed through a series of layered abstractions
  - Caches blocks recently read from disk
  - Buffers recently written blocks
  - Single interface to many devices, allows data to be read/written in fixed sized blocks



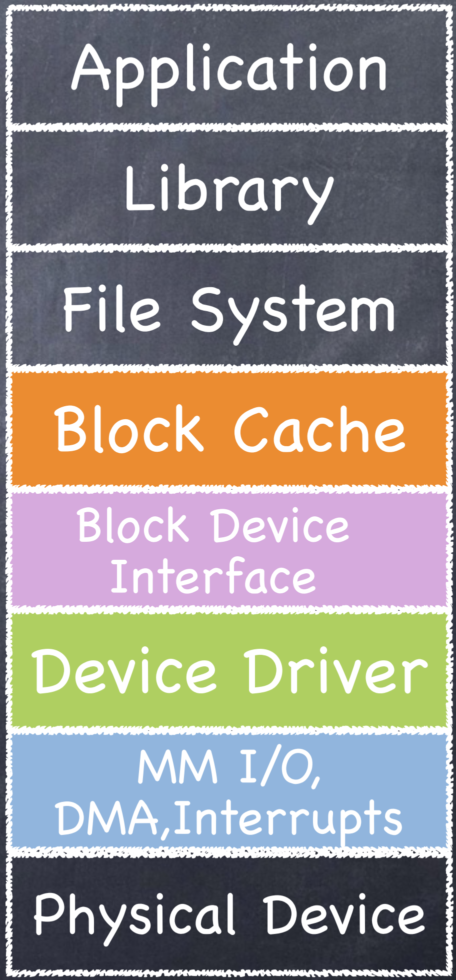
The Abstraction Stack

- I/O systems are accessed through a series of layered abstractions
  - Caches blocks recently read from disk
  - Buffers recently written blocks
  - Single interface to many devices, allows data to be read/written in fixed sized blocks
  - Translates OS abstractions and hw specific details of I/O devices



The Abstraction Stack

- I/O systems are accessed through a series of layered abstractions
  - Caches blocks recently read from disk
  - Buffers recently written blocks
  - Single interface to many devices, allows data to be read/written in fixed sized blocks
  - Translates OS abstractions and hw specific details of I/O devices
  - Control registers, bulk data transfer, OS notifications



File System API

- Creating a file
  - path / flags / permissions
  - int fd = open("foo", O_CREAT|O_RDWR|O_TRUNC, S_IRUSR|S_IWUSR);
  - returns a file descriptor, a per-process integer that grants process a capability to perform certain operations on the file
  - int close(int fd); closes the file
- Reading/Writing
  - ssize_t read (int fd, void *buf, size_t count);
  - ssize_t write (int fd, void *buf, size_t count);
    - return number of bytes read/written
  - off_t lseek (int fd, off_t offset, int whence);
    - repositions file's offset (initially 0, updates on reads and writes)
      - to offset bytes from beginning of file (SEEK_SET)
      - to offset bytes from current location (SEEK_CUR)
      - to offset bytes after the end of the file (SEEK_END)

File System API

- Writing synchronously
  - int fsynch (int fd);
  - flushes to disk all dirty data for file referred to by fd
  - if file is newly created, must fsynch also its directory!
- Getting file's metadata
  - stat() , fstat() — return a stat structure

```
 struct stat {
    dev_t st_dev;       /* ID of device containing file */
    ino_t st_ino;       /* inode number */
    mode_t st_mode;     /* protection */
    nlink_t st_nlink;   /* number of hard links */
    uid_t st_uid;       /* user ID of owner */
    gid_t st_gid;       /* group ID of owner */
    dev_t st_rdev;      /* device ID (if special file) */
    off_t st_size;      /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t st_blocks; /* number of blocks allocated */
    time_t st_atime;    /* time of last access */
    time_t st_mtime;    /* time of last modification */
    time_t st_ctime;    /* time of last status change */
};
```

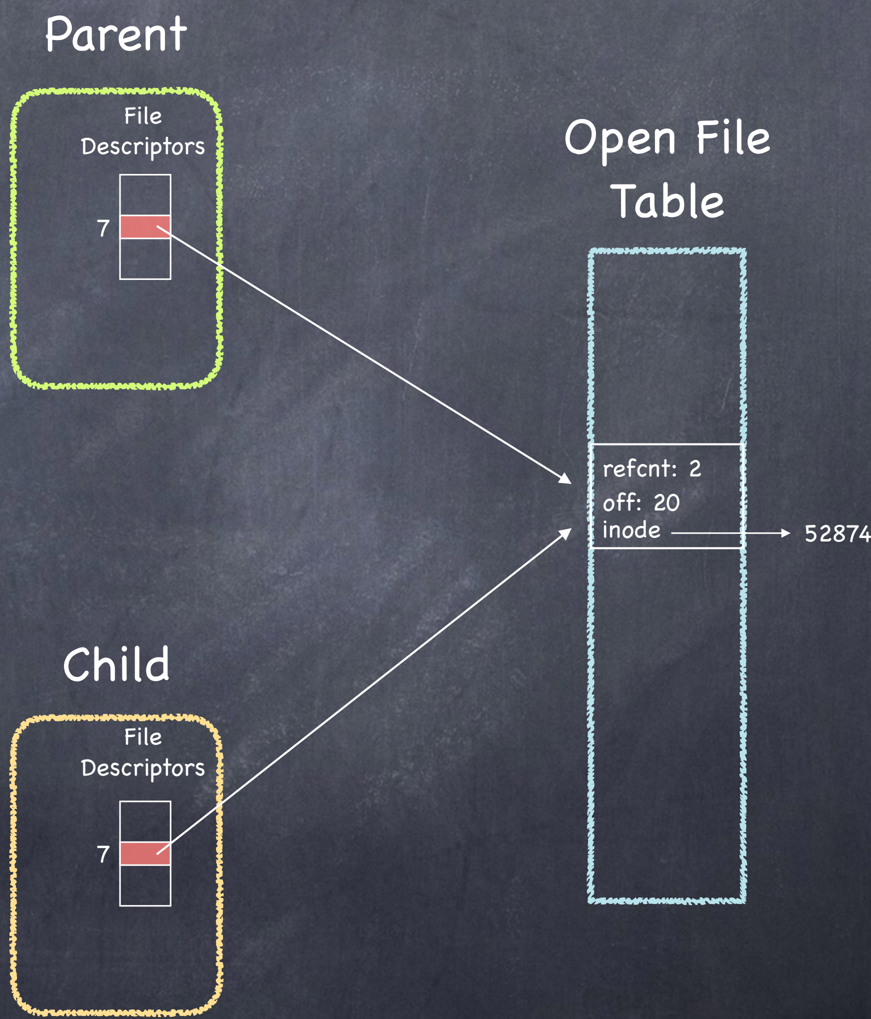
retrieved from file's inode

- on disk, per-file data structure
- may be cached in memory

Old Friends

- Remember fork()?

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



What does this code print?

```
child: offset 10
parent: offset 20
```