Persistent Storage

&

File Systems

Storage Devices

- We focus on two types of persistent storage
  - magnetic disks
    - servers, workstations, laptops
  - flash memory
    - smart phones, tablets, cameras, laptops
- Other exist(ed)
  - tapes
  - drums
  - clay tablets

Interacting with a Device

Abstraction

(what the user sees)

Interacting with a Device

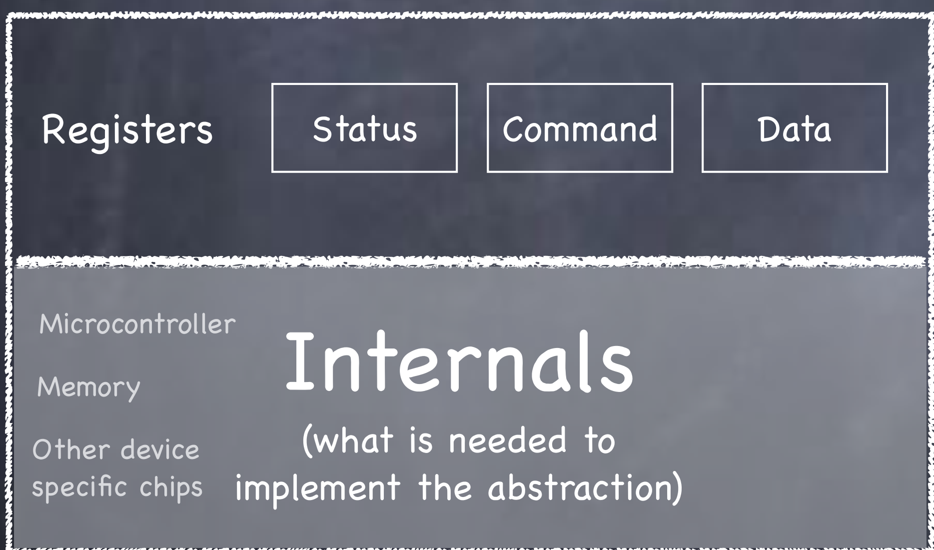
Interface

(what the OS sees)

Internals

(what is needed to implement the abstraction)

Interacting with a Device



Interacting with a Device

- OS controls device by reading/writing registers

```
while (STATUS == BUSY)
      ; // wait until device is not busy
write data to DATA register
write command to COMMAND register
     // starts device and executes command
while (STATUS == BUSY)
     ; // wait until device is done with request
```

Tuning It Up

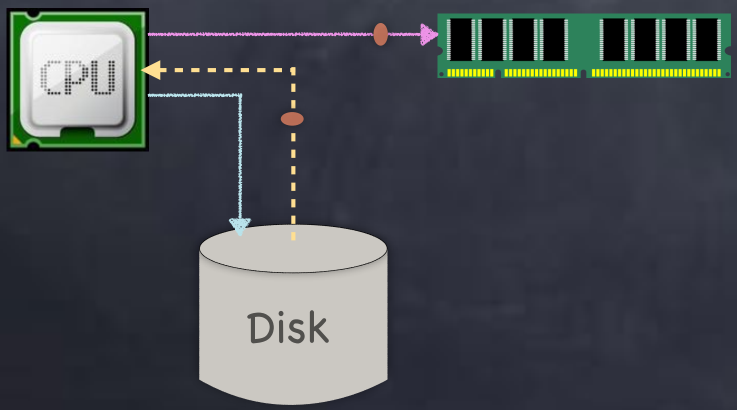
- CPU is polling
  - use interrupts
  - run another process while device is busy
  - what if device returns very quickly?
- CPU is copying all the data to and from DATA
  - use Direct Memory Access (DMA)

```
while (STATUS == BUSY)
      ; // wait until device is not busy
write data to DATA register
write command to COMMAND register
     // starts device and executes command
while (STATUS == BUSY)
     ; // wait until device is done with request
```

From interrupt-driven I/O to DMA

- Interrupt driven I/O
  - Device ⟷ CPU ⟷ RAM

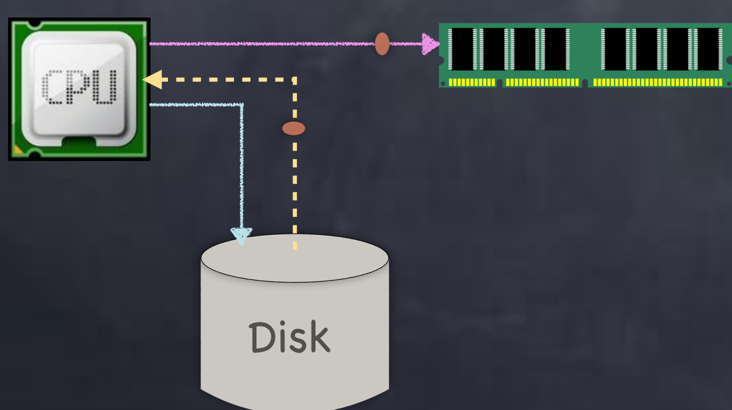
    for $(i = 1 \ldots n)$
    - CPU issues read request
    - device interrupts CPU with data
    - CPU writes data to memory



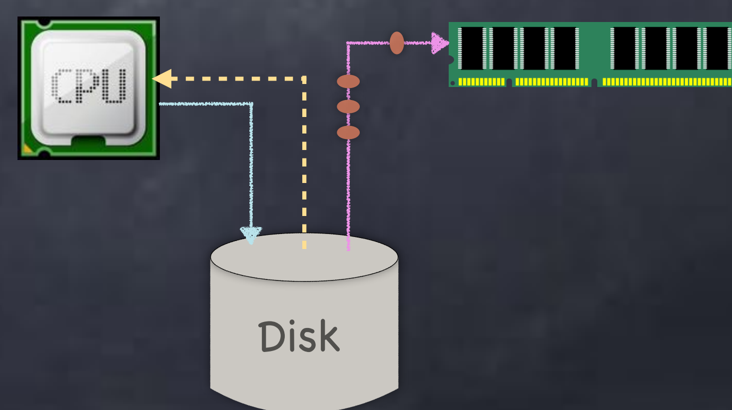
From interrupt-driven I/O to DMA

- Interrupt driven I/O
  - Device ⟷ CPU ⟷ RAM

    for $(i = 1 \ldots n)$
    - CPU issues read request
    - device interrupts CPU with data
    - CPU writes data to memory



- + Direct Memory Access
  - Device ⟷ RAM
    - CPU sets up DMA request
    - Device puts data on bus & RAM accepts it
    - Device interrupts CPU when done



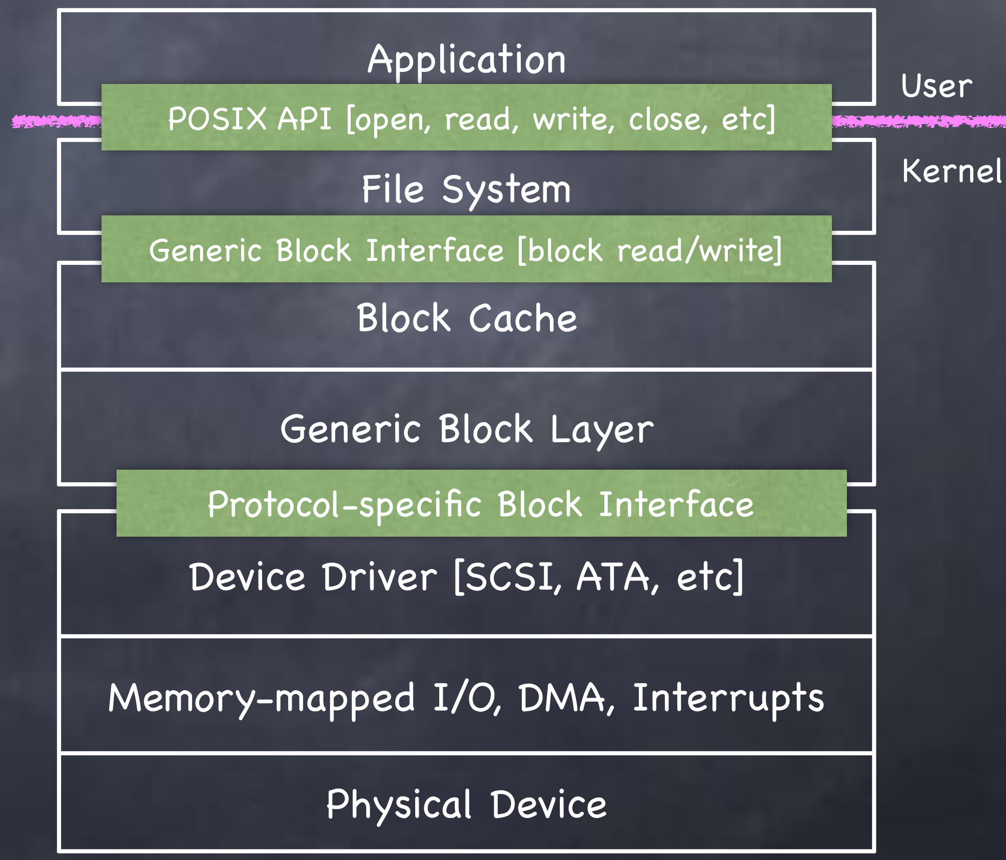
Communicating with devices

- Explicit I/O instructions (privileged)
  - in and out instructions in x86
- Memory-mapped I/O
  - map device registers to memory location
  - use memory load and store instructions to read/write to registers

How can the OS handle a multitude of devices?

- Abstraction!
  - Encapsulate device specific interactions in a device driver
  - Implement device neutral interfaces above device drivers
- Humans are about 70% water...
  - ...OSs are about 70% device drivers!

File System Stack (simplified)



Magnetic disk

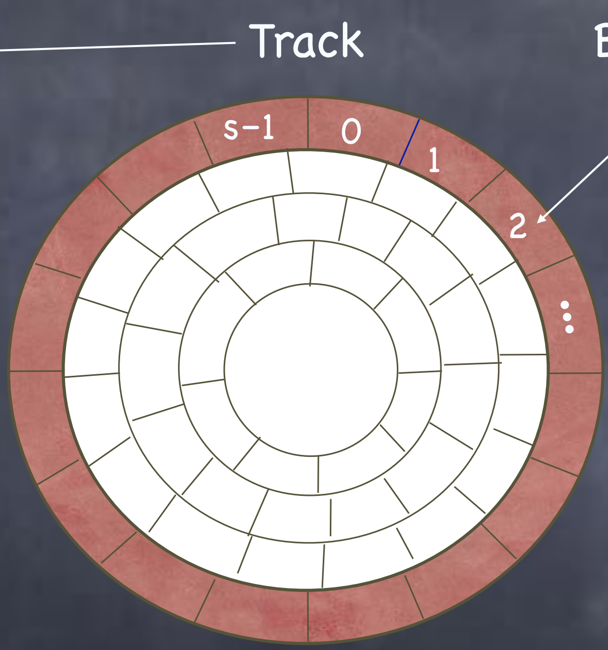
- Store data magnetically on thin metallic film bonded to rotating disk of glass, ceramic, or aluminum

Disk Drive Schematic

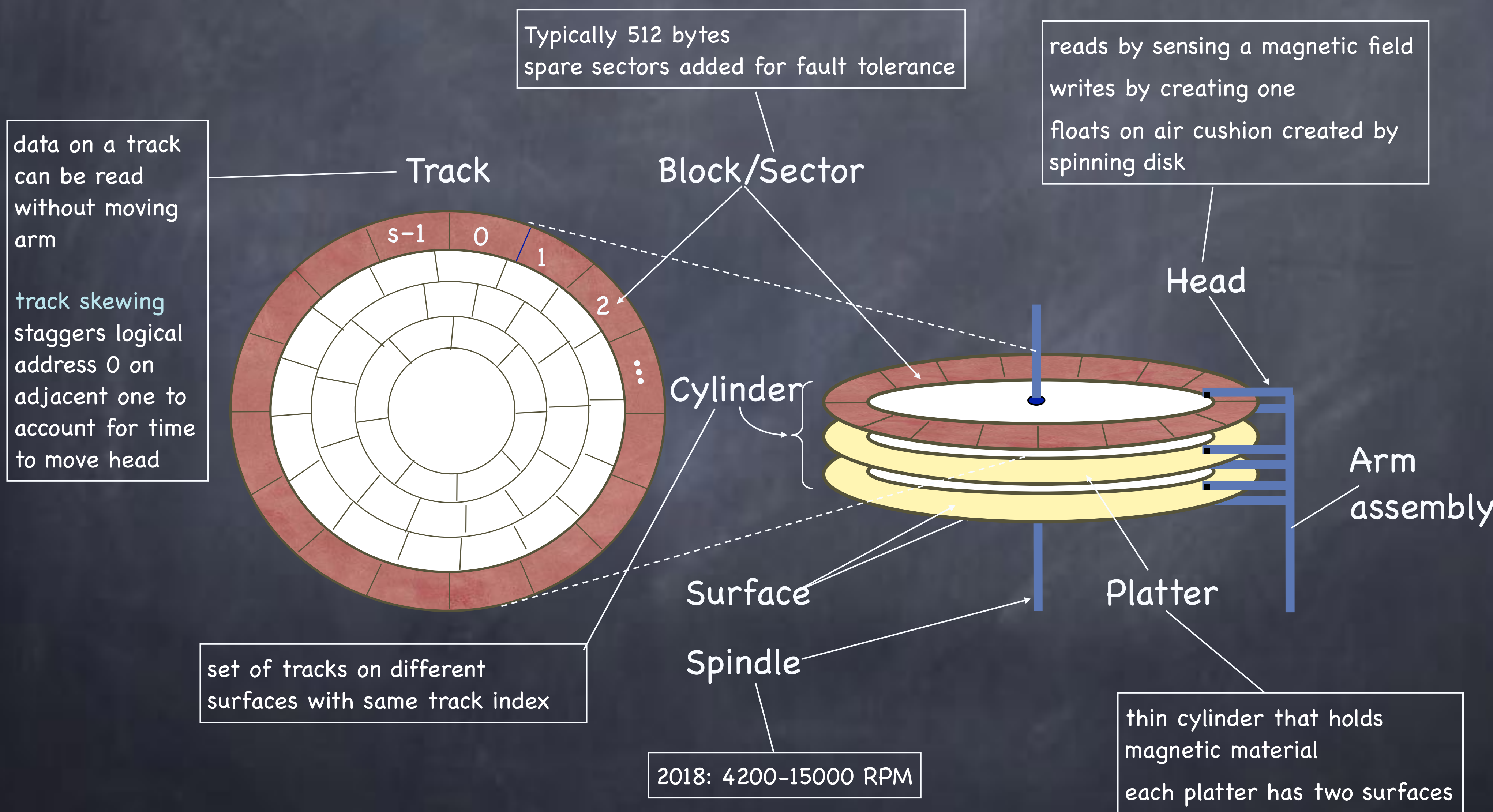
Typically 512 bytes
spare sectors added for fault tolerance

data on a track can be read without moving arm

track skewing staggers logical address 0 on adjacent one to account for time to move head

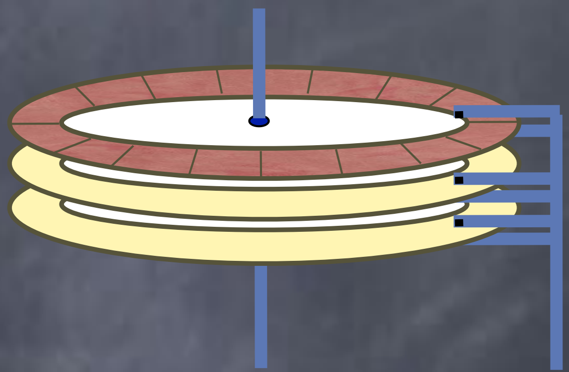


Disk Drive Schematic



Disk Read/Write

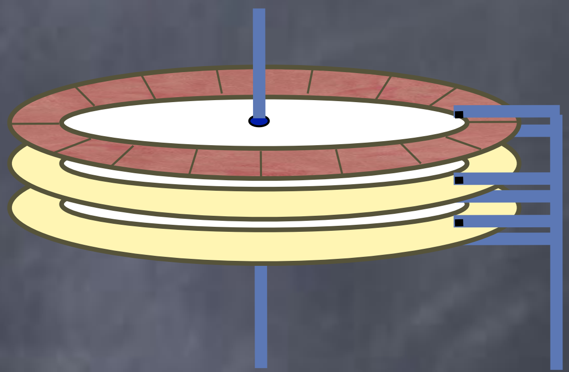
- Present disk with a sector address
  - Old: CHS = (cylinder, head, sector)
  - New abstraction: Logical Block Address (LBA)
    - linear addressing 0...N-1
- Heads move to appropriate track
  - seek
  - settle
- Appropriate head is enabled
- Wait for sector to appear under head
  - rotational latency
- Read/Write sector
  - transfer time



Disk access time:

Disk Read/Write

- Present disk with a sector address
  - Old: CHS = (cylinder, head, sector)
  - New abstraction: Logical Block Address (LBA)
    - linear addressing 0…N-1
- Heads move to appropriate track
  - seek (and though shalt approximately find)
  - settle (fine adjustments)
- Appropriate head is enabled
- Wait for sector to appear under head
  - rotational latency
- Read/Write sector
  - transfer time

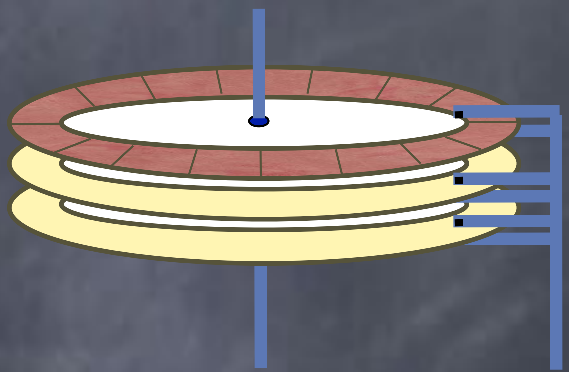


Disk access time:

seek time +

Disk Read/Write

- Present disk with a sector address
  - Old: CHS = (cylinder, head, sector)
  - New abstraction: Logical Block Address (LBA)
    - linear addressing 0…N-1
- Heads move to appropriate track
  - seek (and though shalt approximately find)
  - settle (fine adjustments)
- Appropriate head is enabled
- Wait for sector to appear under head
  - rotational latency
- Read/Write sector
  - transfer time



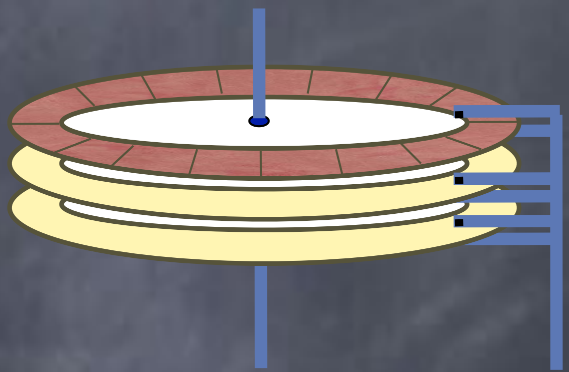
Disk access time:

seek time +

rotation time +

Disk Read/Write

- Present disk with a sector address
  - Old: CHS = (cylinder, head, sector)
  - New abstraction: Logical Block Address (LBA)
    - linear addressing 0…N-1
- Heads move to appropriate track
  - seek (and though shalt approximately find)
  - settle (fine adjustments)
- Appropriate head is enabled
- Wait for sector to appear under head
  - rotational latency
- Read/Write sector
  - transfer time



Disk access time:

seek time +

rotation time +

transfer time

Seek time: A closer look

- Minimum: time to go from one track to the next
  - 0.3-1.5 ms
- Maximum: time to go from innermost to outermost track
  - more than 10ms; up to over 20ms
- Average: average across seeks between each possible pair of tracks
  - approximately time to seek 1/3 of the way across disk

Why? Details in the notes and 3EP readings

Seek time: A closer look

- Minimum: time to go from one track to the next
  - 0.3-1.5 ms
- Maximum: time to go from innermost to outermost track
  - more than 10ms; up to over 20ms
- Average: average across seeks between each possible pair of tracks
  - approximately time to seek 1/3 of the way across disk
- Head switch time: time to move from track i on one surface to the same track on a different surface
  - range similar to minimum seek time

Rotation time: A closer look

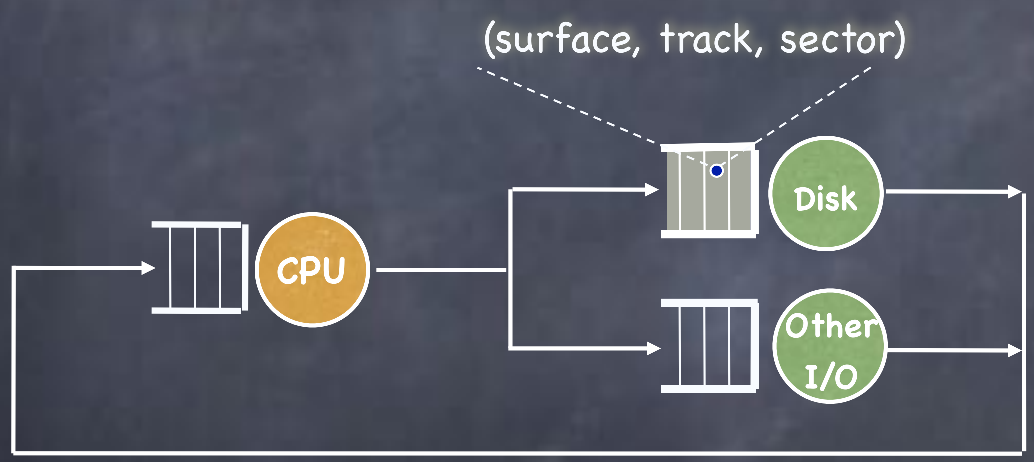
- Today most disk rotate at 4,200 to 15,000 RPM
  - ≈ 15ms to 4ms per rotation
  - good estimate for rotational latency is half that amount
- Head starts reading as soon as it settles on a track
  - track buffering to avoid "shoulda coulda" if any of the sectors flying under the head turn out to be needed

Transfer time: A closer look

- Surface transfer time
  - Time to transfer one or more sequential sectors to/from surface after head reads/writes first sector
  - Much smaller than seek time or rotational latency
    - 512 bytes at 100MB/s ≈ 5μs (0.005 ms)
  - Lower for outer tracks than inner ones
    - same RPM, but more sectors/track: higher bandwidth!
- Host transfer time
  - time to transfer data between host memory and disk buffer
    - 60MB/s (USB 2.0); 640 MB/s (USB 3.0); 25.GB/s (Fibre Channel 256GFC)

Disk Head Scheduling

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

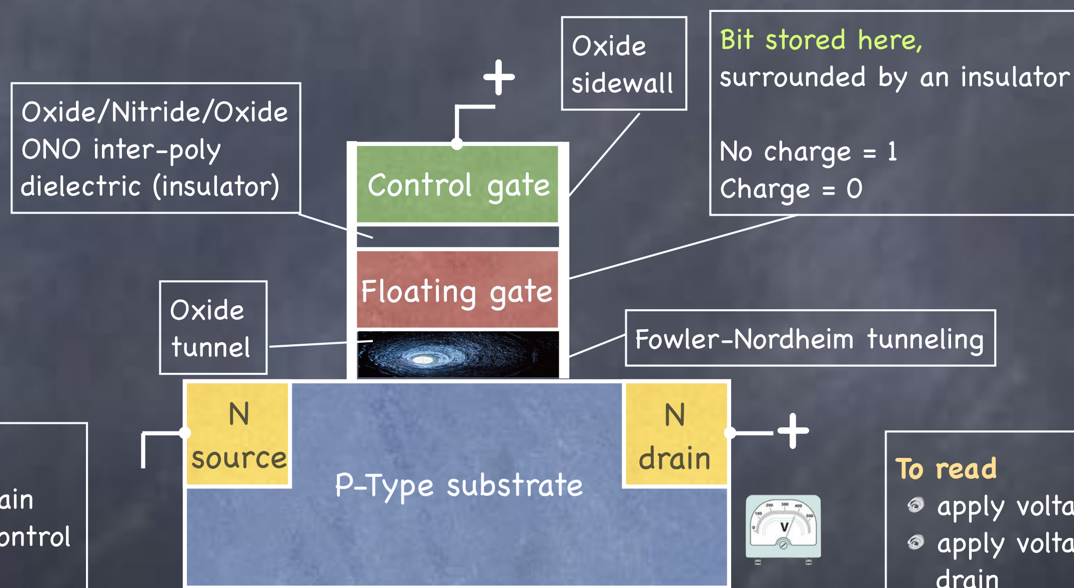


Read about disk scheduling algorithms in class notes and in Chapter 37 of 3 Easy Pieces

- 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!

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 ato 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!

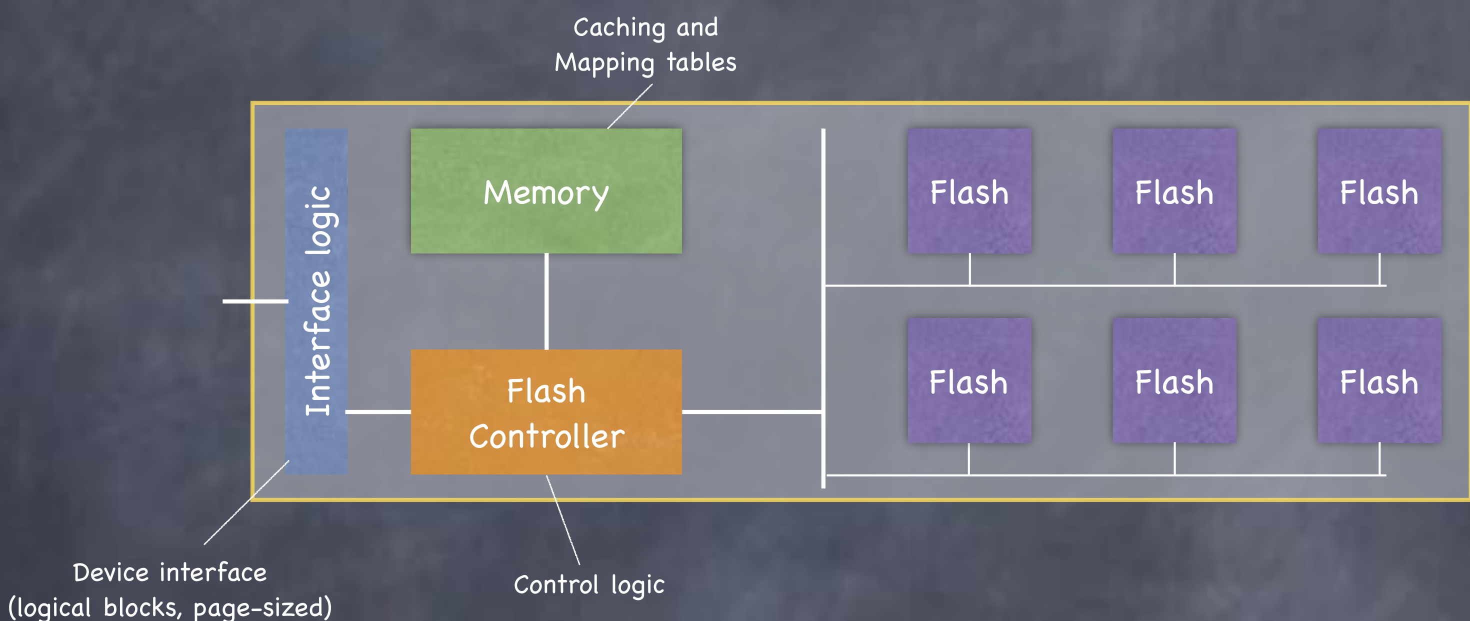
Using Flash Memory

- 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

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

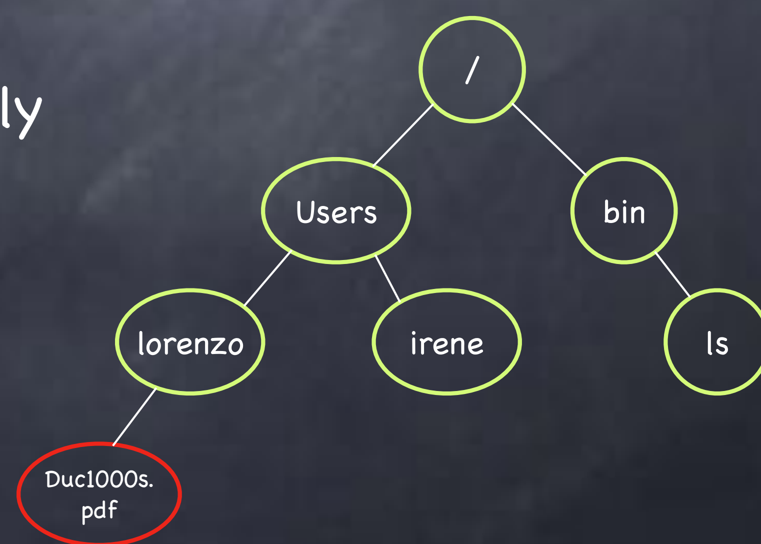
- A file is a named collection of data. In fact, it has many names, depending on context:
  - i-node number: low-level name assigned to the file by the file system
  - path: human friendly string
    - must be mapped to inode number, somehow
  - file descriptor
    - dynamically assigned handle aprocess 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

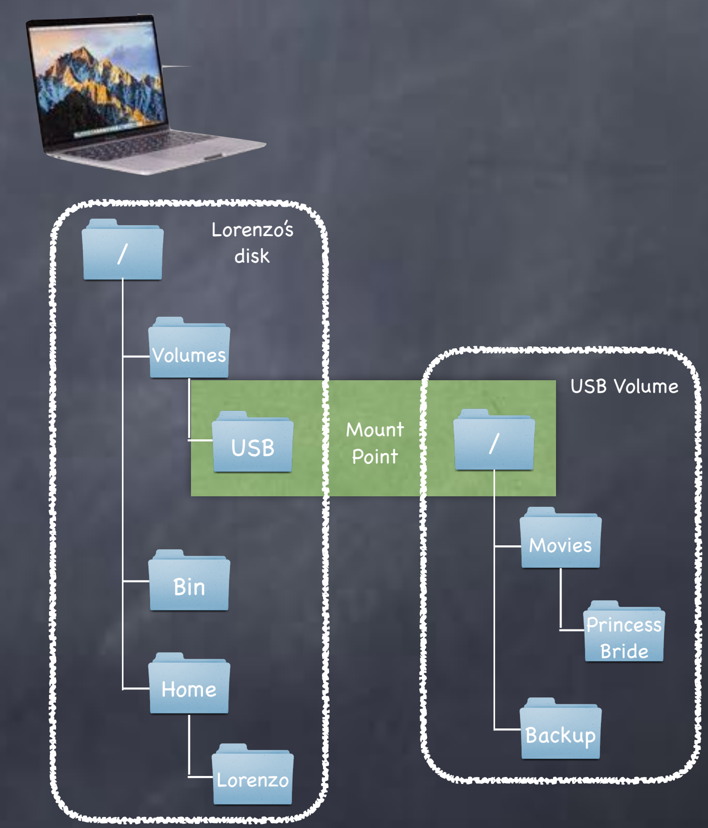
```
Argo% ls -i
2968458 Applications/        3123638 Dropbox (Old)/          4689728 Pictures/     4687176 gems/
2968461 Code/                3123878 Incompatible Software/  4687155 Public/       4687697 mercurial/
2968464 Desktop/             3123881 Library/                4687159 Sites/        4687700 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



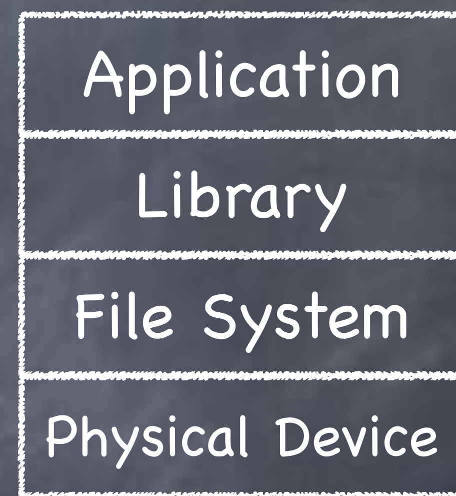
Mount

- 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



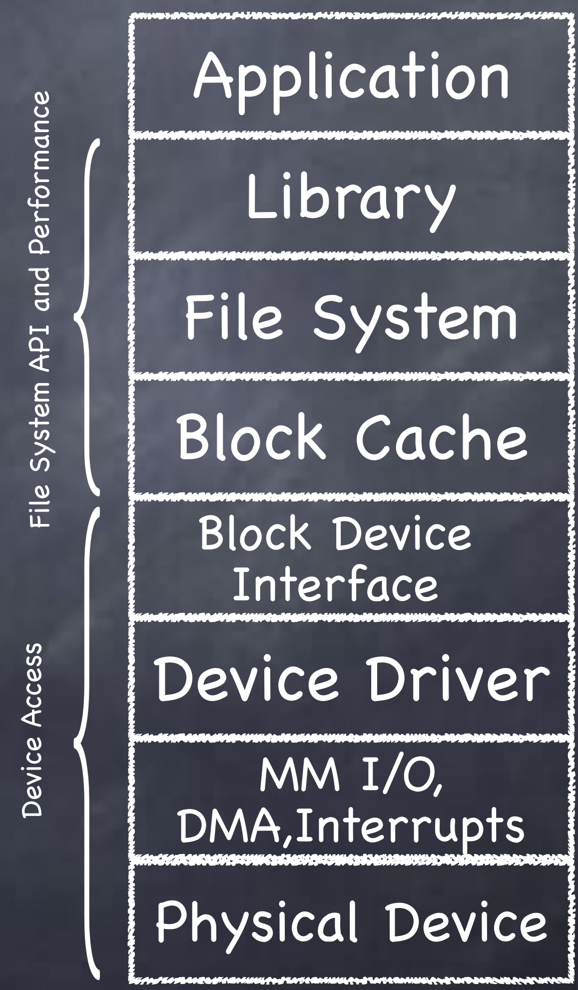
The Abstraction Stack

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



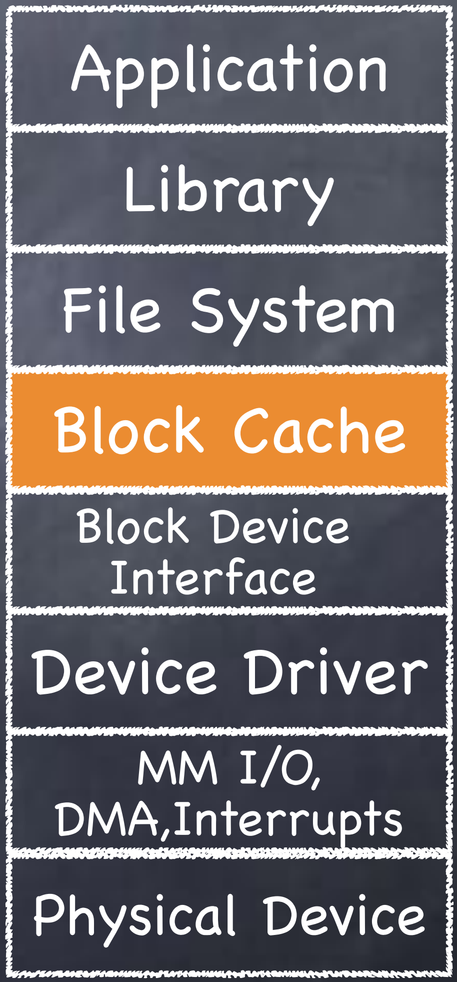
The Abstraction Stack

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



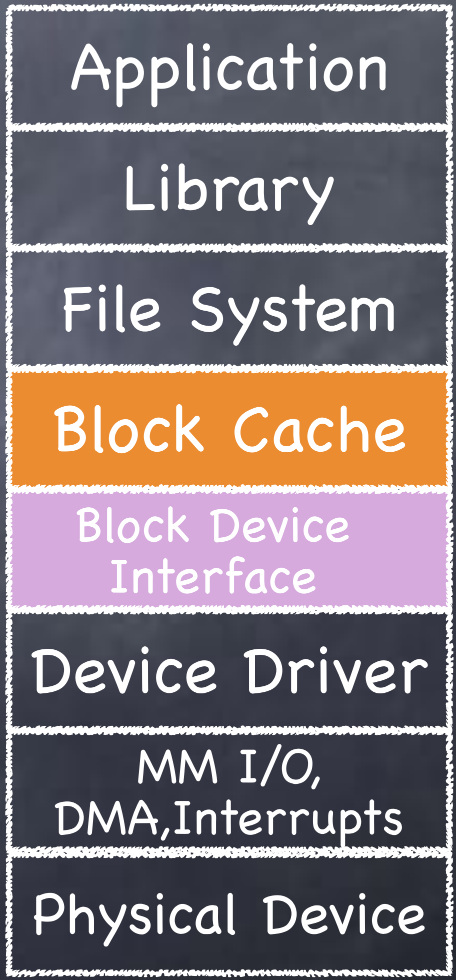
The Abstraction Stack

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



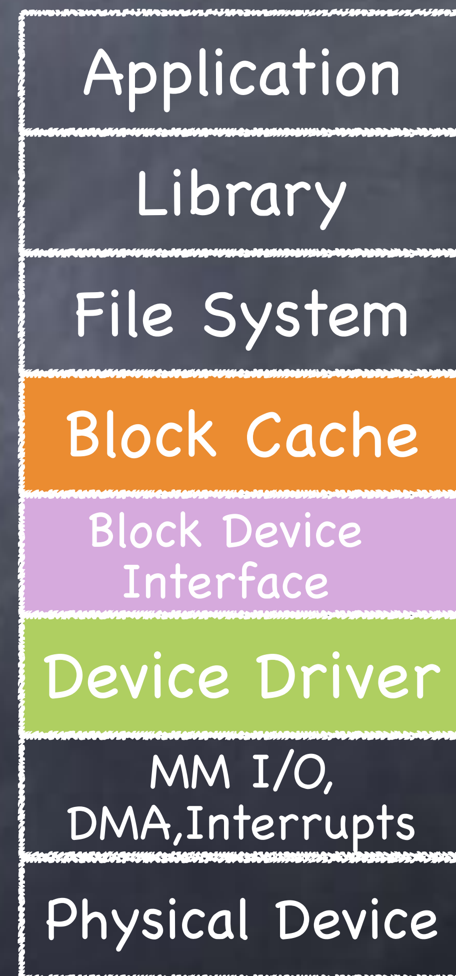
The Abstraction Stack

- 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



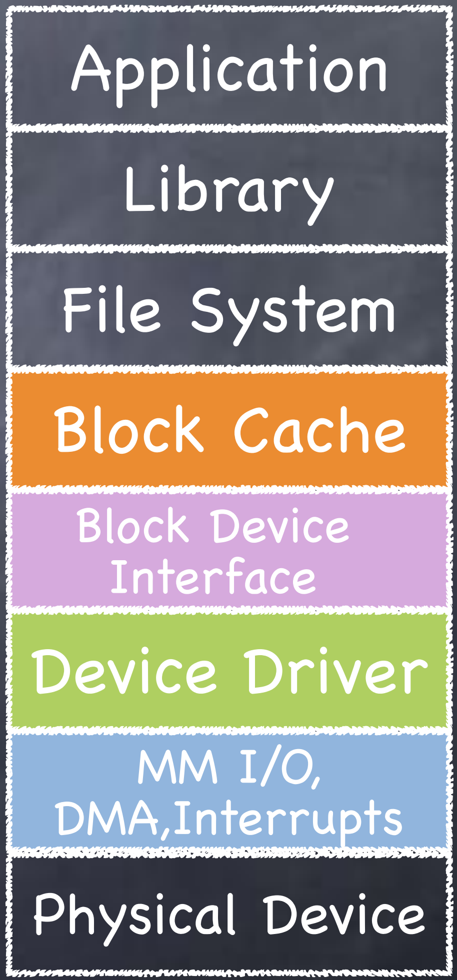
The Abstraction Stack

- 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 */
};
```

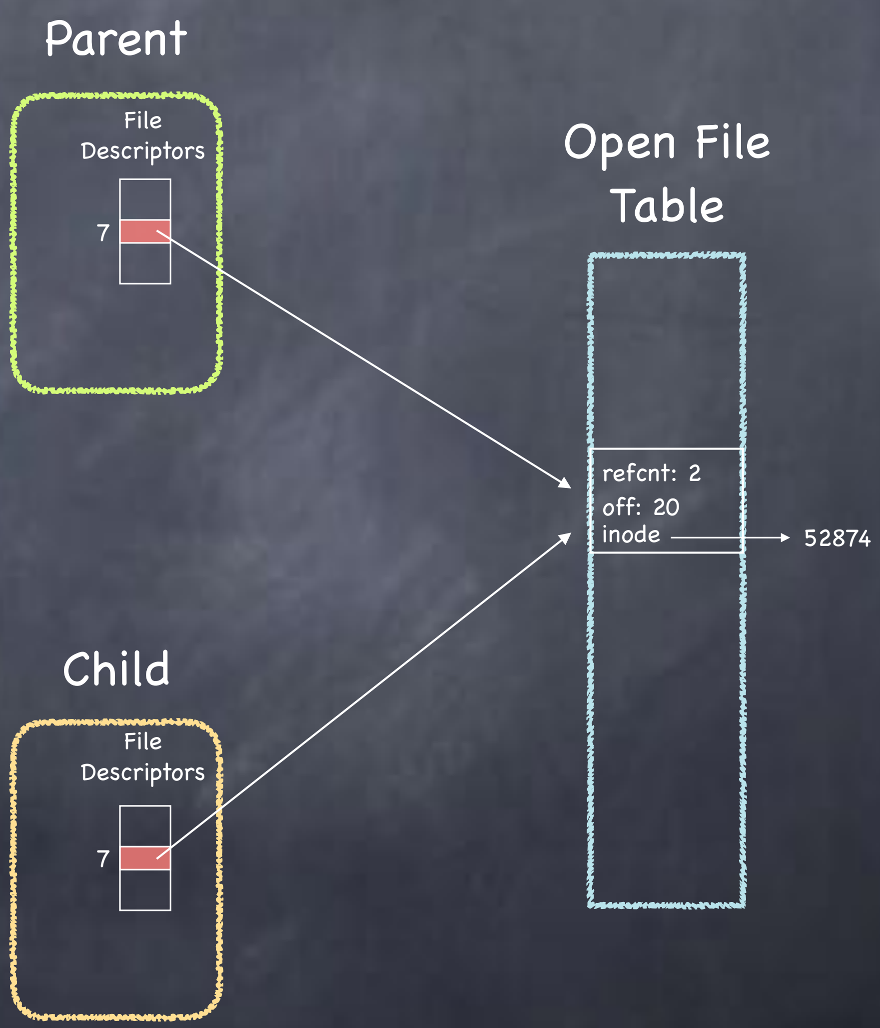
retrieved from file's inode

- 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
```

The Directory

- The directory holds mappings between human-friendly names (HFNs) and inode numbers
- It stores two types of mappings:
  - Hard links
    - map a file's HFN (its local path) to the file's inode number
  - Symbolic (soft) links
    - Logically, map a file's HFN (its local path) to the HFN of a different file
    - Implementation: maps a file's HFN to the number of an inode that contains the HFN of a different file

Hard links

- Creating file foo adds a hard link for file foo in the file's directory
- Command ln oldpath newpath
  - adds to the directory a hard link mapping HFN newpath to the inode number of the file with HFN oldpath
  - Now two HFNs are mapping to the same inode!
  - calls int link(const char *oldpath, const char *newpath)
- Removing a file through the rm [file] command invokes a call to int unlink(const char *pathname)
  - removes from directory the hard link between pathname and corresponding inode number
- File's inode stores the number of hard links to it
  - inode reclaimed (file deleted) only when link count = 0; if file opened, wait to reclaim until file is closed

Hard link No-Nos

- Creating a hard link to a directory
  - may create a cycle in the directory tree!
- Creating a hard link to files in other volumes
  - inode numbers are unique only within a single file system