PERSISTENCE:
I/O DEVICES AND HARD DISKS

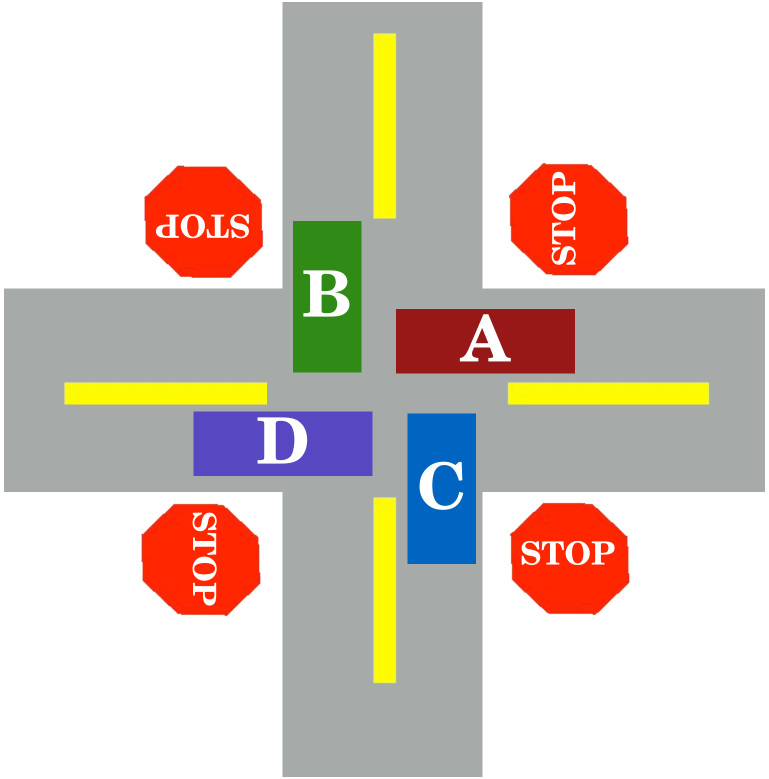
Kai Mast
CS 537
Fall 2022

DEADLOCK THEORY

Deadlocks can only happen when these four conditions hold:

1. mutual exclusion
2. hold-and-wait
3. no preemption
4. circular wait

Can eliminate deadlock by eliminating any one condition



1. MUTUAL EXCLUSION

Problem: Threads claim exclusive control of resources that they require
Strategy: Eliminate locks!

Try to replace locks with atomic primitive:

```
// returns 0 on failure, 1 on success
int cmp_and_swap(int *addr, int expected, int new) {
      // Does this atomically using hardware (for example, cmpxchg on x86)
      if *addr == expected {
           *addr = new;
           return 1;
       } else {
           return 0;
       }
}
```

LOCK-FREE ALGORITHMS

```
void add(int *val, int amt) {
  mutex_lock(&m);
  *val += amt;
  mutex_unlock(&m);
}
```

```
void add(int *val, int amt) {
  int old;
  do {
     old = *val;
  } while(!cmp_and_swap(val, old, old+amt);
}
```

T1: add(&val, 2);

val = 10;

T2: add(&val, 3);

```
T1: add(&val, 2);
old = 10;
```

```
T2: add(&val, 3);

old = 10;
*val == 10 => success; *val = 13;
return true;
```

```
*val != 10 => fail
old = 13;
*val == 13 => success; *val=15;
```

LOCK-FREE ALGORITHM: LINKED LIST INSERT

```
void insert(list_t *l, int val) {
  node_t *n = malloc(sizeof(*n));
  n->val = val;
  lock(&l->mutex);
  n->next = l->head;
  l->head = n;
  unlock(&l-<m);
}
```

```
void insert (list_t *l, int val) {
  node_t *n = malloc(sizeof(*n));
  n->val = val;
  do {
    n->next = l->head;
  } while (!cmp_and_swap(&l->head,
                          n->next, n));
}
```

LOCK-FREE ALGORITHM: LINKED LIST INSERT

```
void insert (list_t *l, int val) {
    node_t *n = malloc(sizeof(*n));
    n->val = val;
    do {
        n->next = l->head;
    } while (!cmp_and_swap(&l->head, n->next, n));
}
```

Assume scheduling: T1, T2, T2, T1… (one line each)

T1: insert(2); initially head = 0x0 **T2: insert(3);**

T1:

```
node_t *n = malloc(...); // 0x100

n->val = val; // n->val == 2

n->next = l->head; // n->next == 0x0

cmp_and_swap => success! (l->head == 0x100)
```

T2:

```
node_t *n = malloc(...); // 0x200

n->val = val; // n->val == 3

n->next = l->head; // n->next == 0x0

cmp_and_swap => fail (n->next != l->head)
n->next = l->head; // n->next == 0x100
cmp_and_swap => success (l->head == 0x200)
```

2. HOLD-AND-WAIT

Problem: Threads hold resources while waiting for additional resources

Strategy: Acquire all locks atomically

Can release locks over time, but cannot acquire again until all have been released

How? Use a meta lock:

```
lock(&meta);
lock(&L1);
lock(&L2);
lock(&L3);
…
unlock(&meta);
// CS1
unlock(&L1);
// CS 2
Unlock(&L2);
```

```
lock(&meta);
lock(&L2);
lock(&L1);
unlock(&meta);

// CS1
unlock(&L1);

// CS2
Unlock(&L2);
```

```
lock(&meta);
lock(&L1);
unlock(&meta);

// CS1
unlock(&L1);
```

2. HOLD-AND-WAIT

Disadvantages?

Must know ahead of time which locks will be needed
Must be conservative (acquire any lock possibly needed)
Degenerates to just having one big lock (reduces concurrency)

```
lock(&meta);
lock(&L1);
lock(&L2);
lock(&L3);
…
unlock(&meta);
// CS1
unlock(&L1);
// CS 2
Unlock(&L2);
```

```
lock(&meta);
lock(&L2);
lock(&L1);
unlock(&meta);

// CS1
unlock(&L1);

// CS2
Unlock(&L2);
```

```
lock(&meta);
lock(&L1);
unlock(&meta);

// CS1
unlock(&L1);
```

3. NO PREEMPTION

Problem: Resources (e.g., locks) cannot be forcibly removed from other threads

Strategy: if thread can not get what it wants, release what it holds

```
top:
    lock(A);
    if (trylock(B) == -1) {
        unlock(A);
        goto top;
    }
    // use A...
```

Disadvantages?

Potential Livelock

No processes make progress, but state of involved processes constantly changes

Classic solution: Exponential random back-off

4. CIRCULAR WAIT

Circular chain such that each thread holds a resource (e.g., lock) requested by next thread in chain

Practical Solution:

- Decide which locks must be acquired before others
- If A before B, never acquire A if B is already held!
- Document and write code accordingly
- Works well if system has distinct layers

Example: Lock Ordering in Xv6

Creating a file requires simultaneously holding:

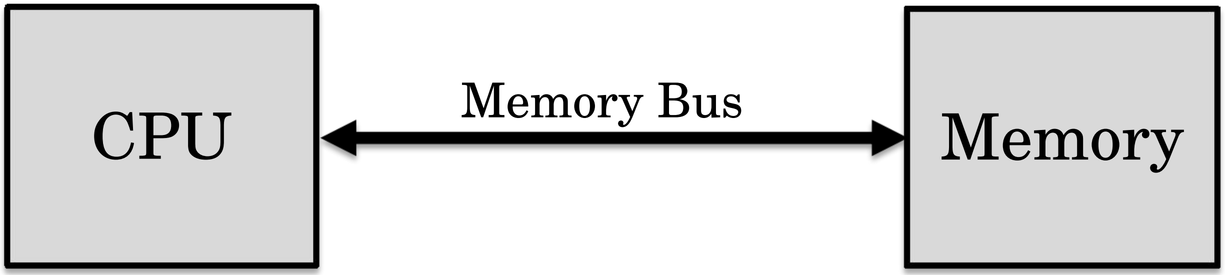
- a lock on the directory,
- a lock on the new file's inode,
- a lock on a disk block buffer,
- idelock,
- ptable.lock

Always acquires locks in the order listed

I/O DEVICES

(Book Chapter 36)

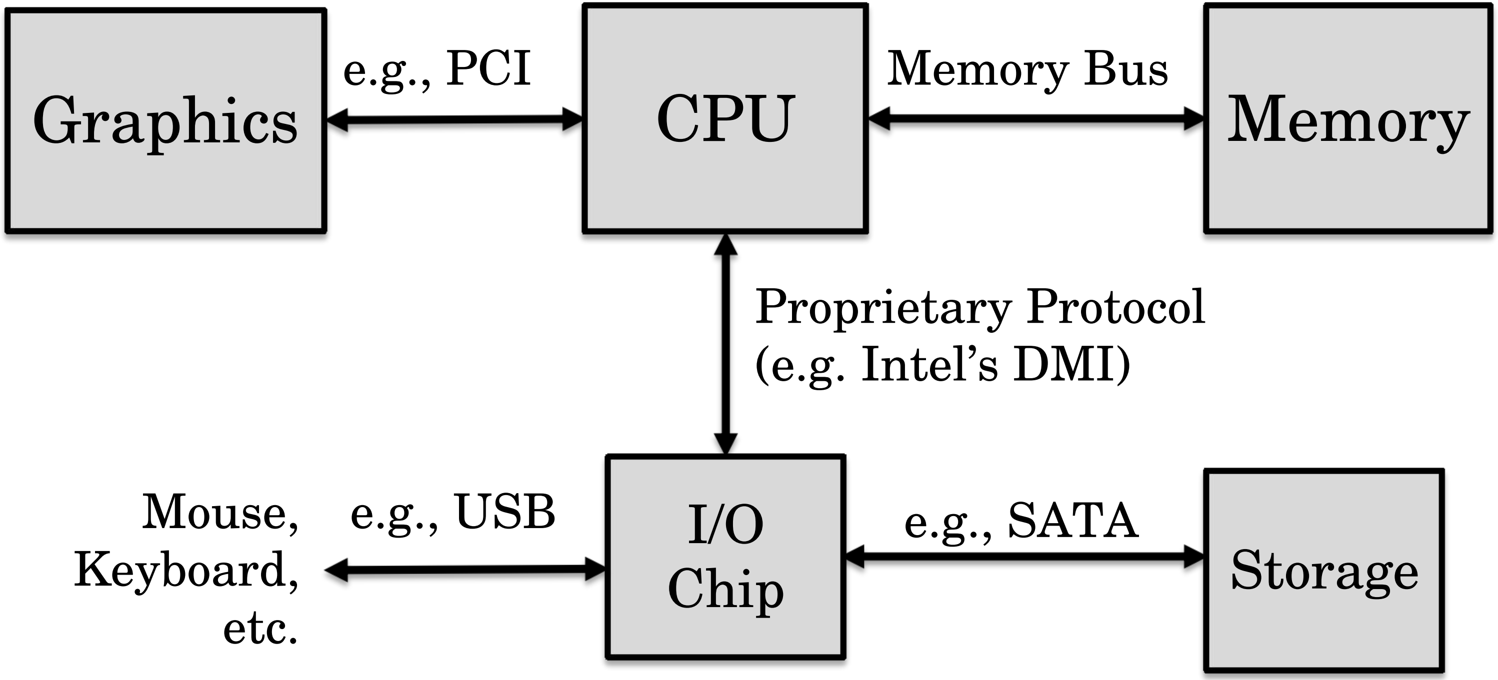
COMPUTERS SO FAR



Memory is volatile

- Data is lost when removing power
- Need hardware that provides persistence

MODERN SYSTEM ARCHITECTURE



AGENDA

Today

- Interaction with I/O devices
- Hard disk drives

Remainder of the Semester

- Other types of storage, e.g., SSDs
- RAID
- File systems
- Distributed File systems

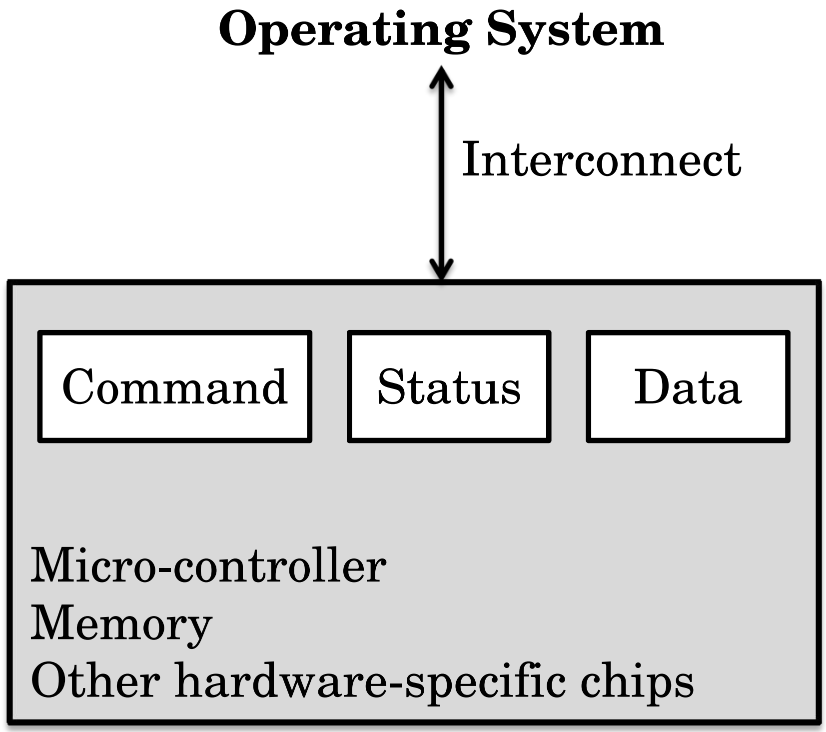
DEVICE INTERFACE

OS controls device. Why?

- Security
- Virtualization
  - Support different kinds of hardware
  - Allow multiple processes to access the hardware at the same time

Device internals have varying levels of complexity

- e.g., GPU (high-complexity) vs. Keyboard (low-complexity)

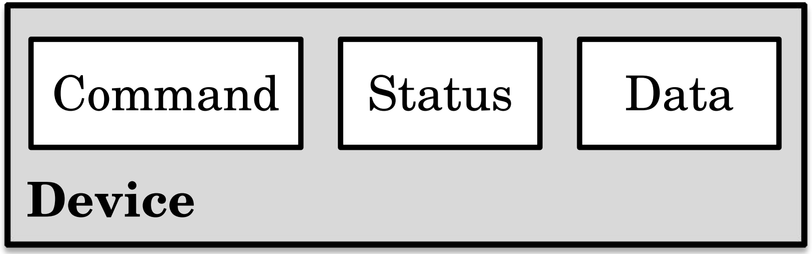


Generic hardware device

ACCESSING DEVICE REGISTERS

Approach 1: Special I/O instructions
- Addition to the CPU's instruction set that allows accessing hardware
- e.g, `IN` and `OUT` in x86

Approach 2: Memory-Mapped I/O
- Device registers are mapped into memory
- OS can write to device registers like to any other memory location
- *Not* the same as `mmap`!



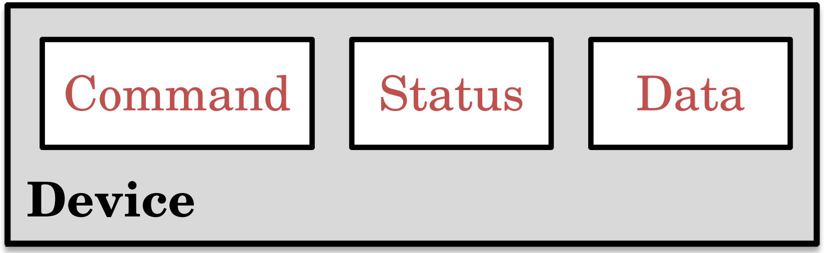
SIMPLE HARDWARE ACCESS PROTOCOL

```
// wait until device is not busy
while (Status == BUSY) {}

for (size of data)
    move data -> Data register

write command to Command register

// wait until device is done with request
while (Status == BUSY) {}
```



Problems

1. Spinning wastes CPU cycles (sometimes called *polling*)
2. Data movement is very CPU intensive

SPIN-FREE DEVICE ACCESS

```
system-call:
// wait until device is not busy
sem_wait(device_ready);

for (size of data)
    move data -> Data register

write command to Command register

// wait until device is done
sem_wait(device_ready)
```

```
interrupt-handler:
sem_post(device_ready)
```

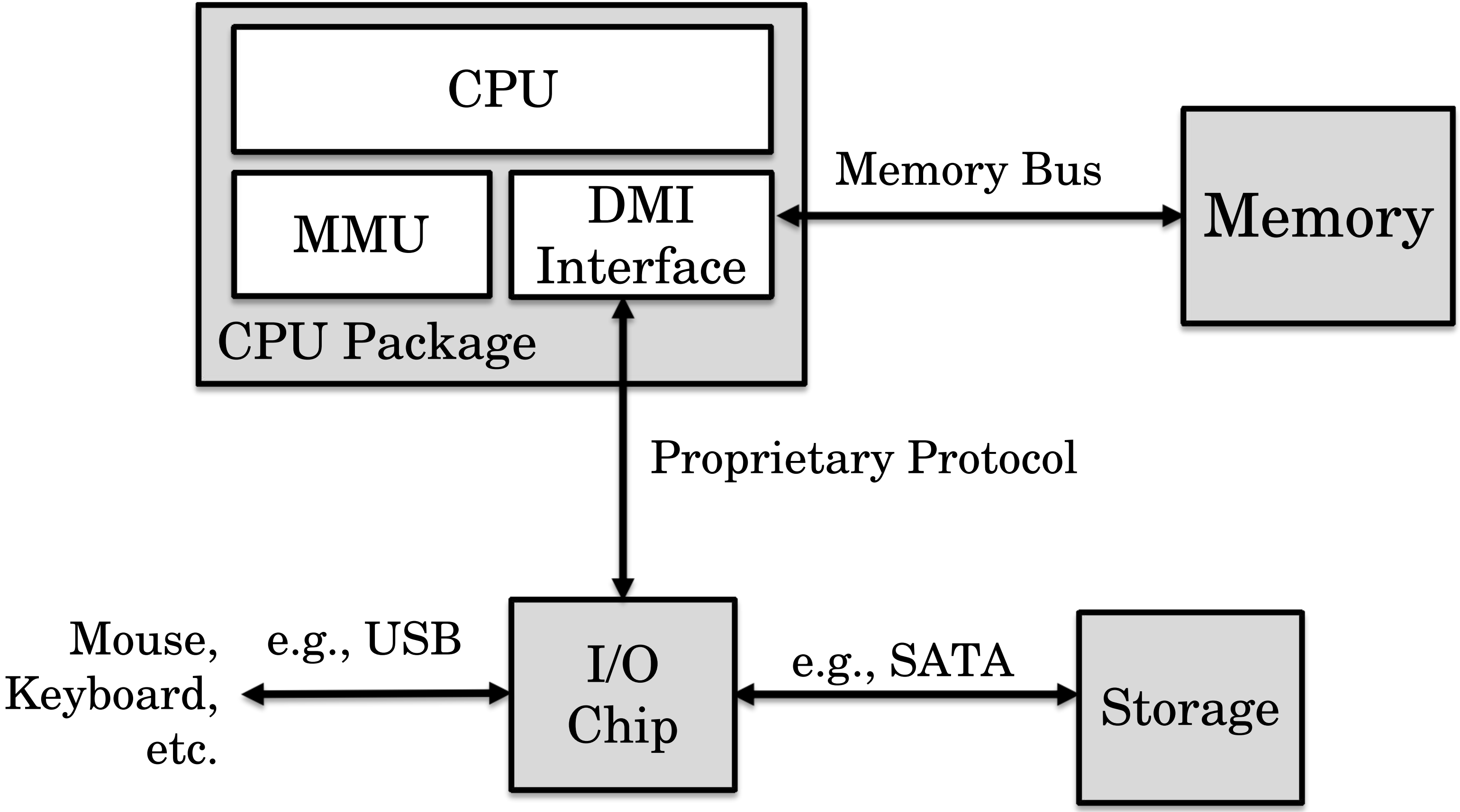
Approach

- Instead of spinning, go to sleep
- Process state changes to BLOCKED
- Device notifies when it is done using an interrupt

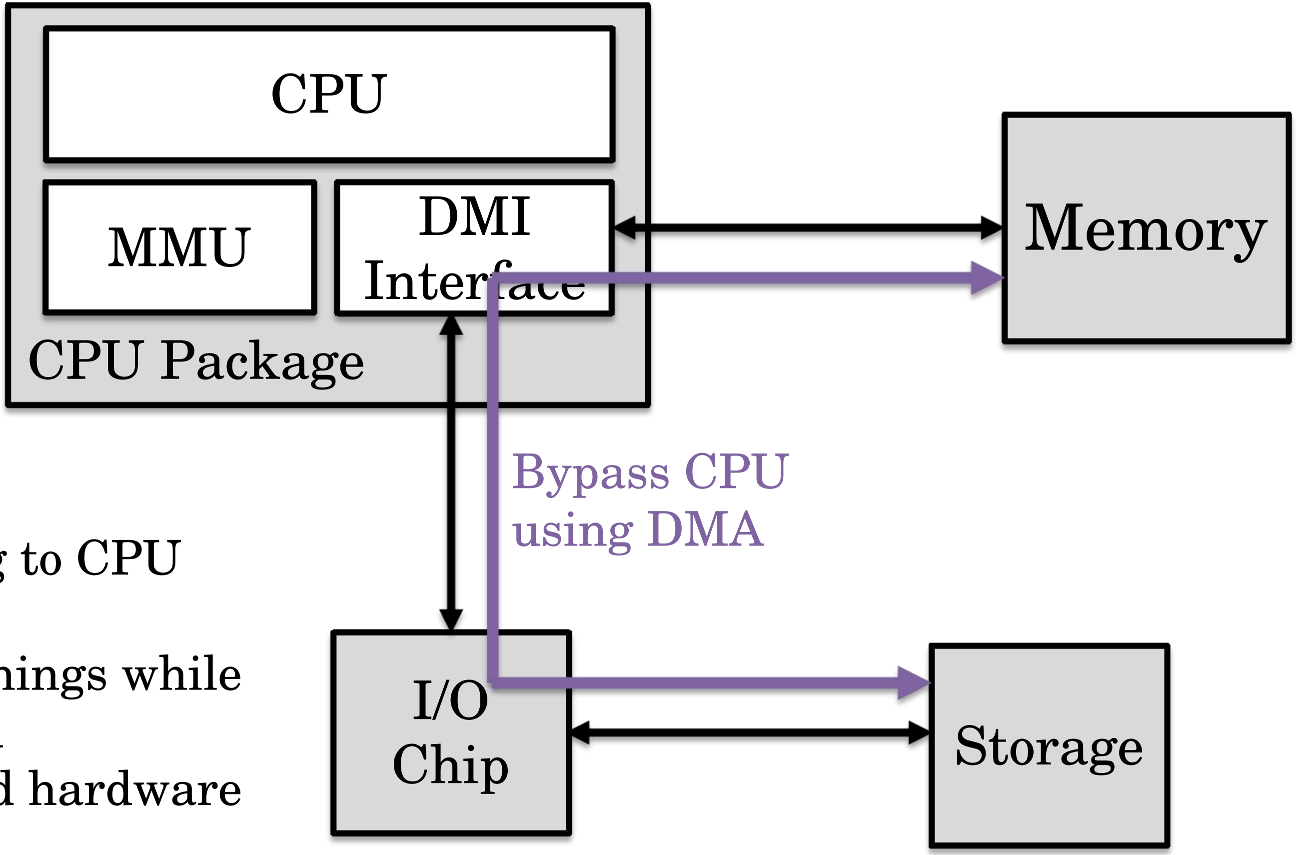
Caveats?

- If device is very fast, we get very frequent interrupts
- Leads to context switch overhead

MODERN DIRECT MEMORY ACCESS



MODERN DIRECT MEMORY ACCESS



- Faster than copying to CPU and then to disk
- CPU can do other things while data is being moved
- Requires specialized hardware

HARD DISKS

(Book Chapter 37)

TYPES OF STORAGE DEVICES

Tape Drives

Hard Disks Drives

Solid State Drives

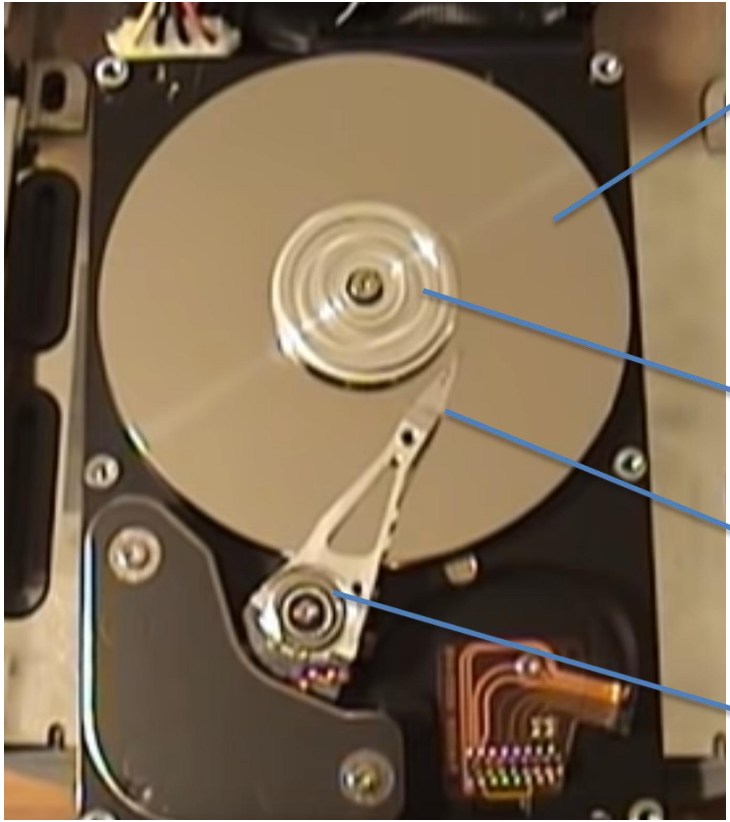
Persistent Memory (e.g. Intel Optane)

Speed

Cost

Size

HARD DISK DRIVES



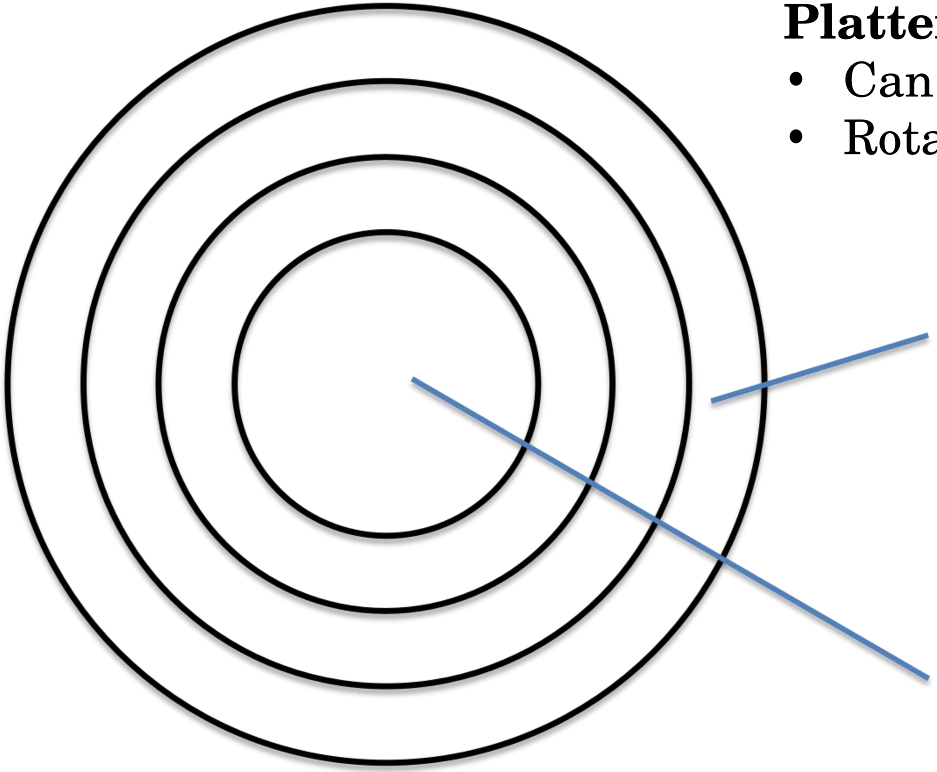
Platter
- Double-sided
- A drive can have multiple platters

Spindle

Read/write Head

Disk Arm
- Moves between tracks of the platter

HARD DISK PLATTER



Platter

- Can be read or written
- Rotates at fixed speeds (e.g., 7200 RPM)

Track

- Split into fixed-size sectors (often 512 bytes)
- Contain redundant encoding to recover from corrupted bits

Spindle

HARD DISK CONTROLLER

Controller executes operations stored in command buffer

- Writes output to status or data registers

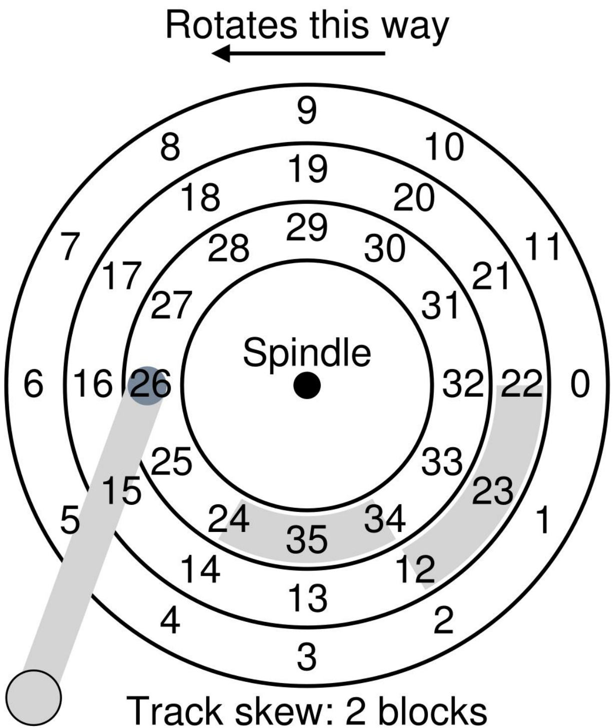
Tracks/sectors are exposed to the OS as a linear array

- Controller translates accesses to disk into internal actions
- Need to figure out which track and sector the data is located at

Controllers can keep track of multiple actions at once

- Allows for higher throughput
- Can schedule actions to minimize platter and arm movements

TRACK SKEW



Sectors on different tracks are offset on most disks

- See, for example, the "gap" between sectors 11 and 12

Why?

- We can change tracks without stopping the platter rotation
- Allows for faster sequential reads

HARD DISK ACCESS SPEED

To perform an I/O operation

- **seek:** move disk arm to correct track
- **wait (rotation):** wait for sector to rotate under arm
- **transfer:** read/write data

Example: Read one sector (512B)

- avg. seek: 7ms
- avg. rotate: 3ms
- avg. transfer: ~0ms (200Mb/s)
- throughput is 512b/10ms ≈ 50kb/s

Why so slow?

- Random I/O are dominated by seek and rotation
- Sequential accesses can be much faster