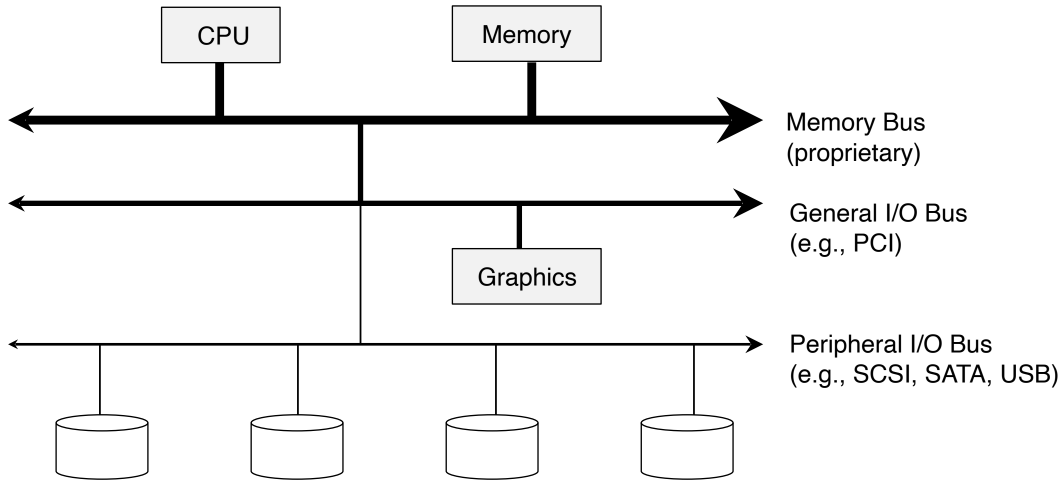
Mass Storage

Persistence

- 36 - I/O Devices
- 37 - Hard Disk Drives
- 38 - RAID
- 39 - File and Directories

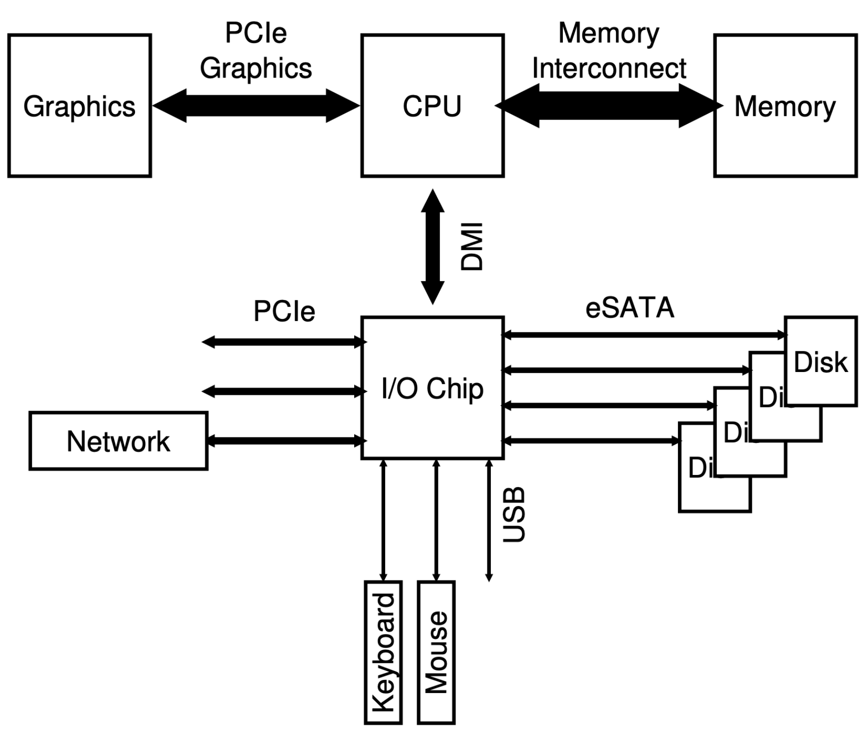
Classic I/O Architecture



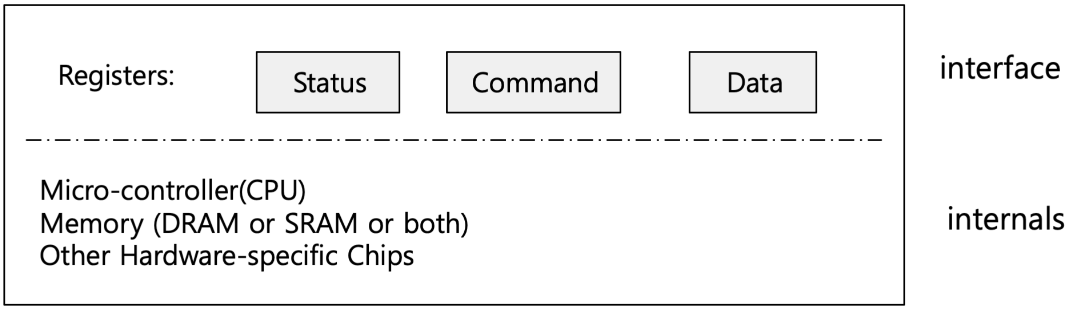
Prototypical System Architecture

- How should I/O devices be integrated into systems?
- What are the general mechanisms?
- How can we make them efficient?

Modern Architecture



A Canonical Device



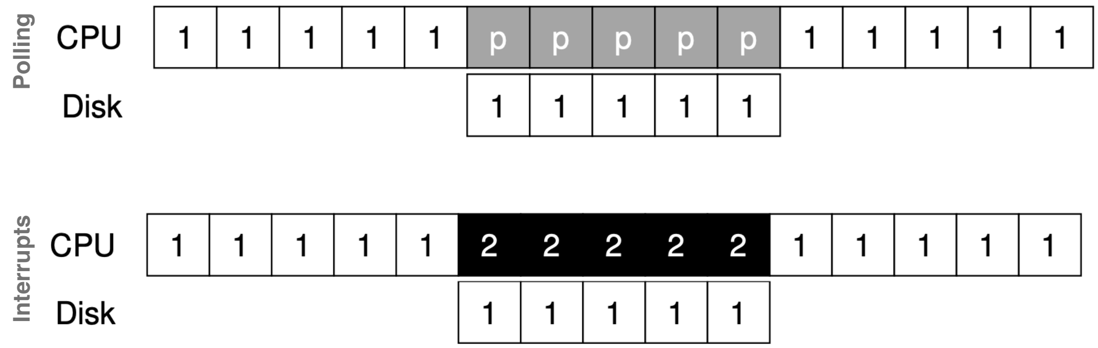
- status register: read current device state
- command register: send commands to device
- data register: read or write data a word at a time

Devices: Polling for Response

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

- Simple
- Inefficient
  - CPU occupied doing nothing

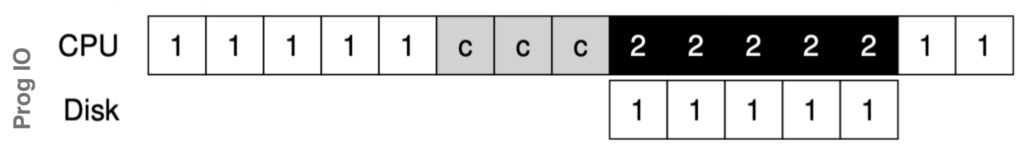
Devices: interrupts



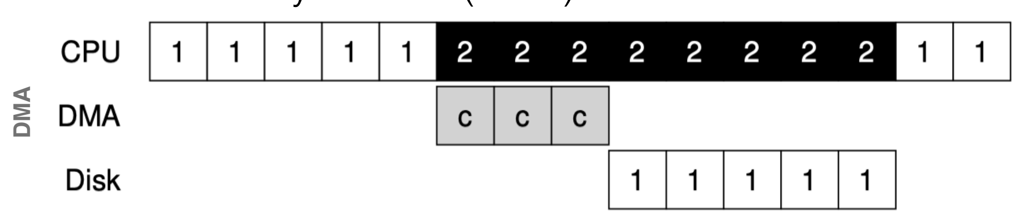
- Send request
- Do something else
- Reschedule process only when interrupt signals finished

Efficiency Issues With Interrupts

- Fast jobs: first poll might have found job finished
  - Hybrid: poll for a bit, then block
- Livelock: per-packet interrupts might monopolize CPU
  - Coalescing: device delays to combine multiple interrupts
- Writing large blocks to device is a poor use of CPU

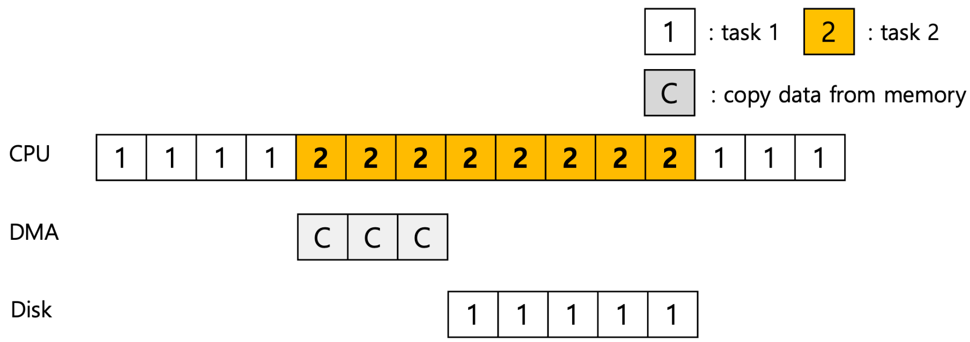


  - Direct Memory Access (DMA)



DMA

- Starting
  - write address, length of data block to device data registers
  - start by writing to control register
  - *do something else*
- Finish
  - raise interrupt to signal finish



Communicating w/ devices

- Specific I/O instructions
  - instructions `in` and `out` on x86
- Memory-mapped I/O
  - each register mapped to specific kernel address
  - kernel uses ordinary `load` and `store`

File Systems Stack

- Want any file system to write to any device

Application

POSIX API [open, read, write, close, etc.]

File System | Raw

Generic Block Interface [block read/write]

Generic Block Layer

Specific Block Interface [protocol-specific read/write]

Device Driver [SCSI, ATA, etc.]

user

kernel mode

- *more than 70% of os code is in device drivers*

Example Device: IDE interface

- wait for drive:
  - read status register until `READY` and not `BUSY`
- sector count, logical block address, drive number to command registers
- start I/O
  - issue read/write to command register
- data transfer (writes)
  - wait until `READY` and `DRQ` (drive request for data)
  - write data to port
- handle interrupts
  - per sector transferred, or batch
- error handling
  - read status register

```
Control Register:
  Address 0x3F6 = 0x08 (0000 1RE0): R=reset,
                  E=0 means "enable interrupt"

Command Block Registers:
  Address 0x1F0 = Data Port
  Address 0x1F1 = Error
  Address 0x1F2 = Sector Count
  Address 0x1F3 = LBA low byte
  Address 0x1F4 = LBA mid byte
  Address 0x1F5 = LBA hi  byte
  Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive
  Address 0x1F7 = Command/status

Status Register (Address 0x1F7):
    7      6     5     4    3    2     1     0
   BUSY  READY FAULT SEEK  DRQ  CORR IDDEX ERROR

Error Register (Address 0x1F1): (check when ERROR==
    7      6     5     4    3    2     1    0
   BBK    UNC   MC   IDNF  MCR  ABRT TONF AMNF

   BBK  = Bad Block
   UNC  = Uncorrectable data error
   MC   = Media Changed
   IDNF = ID mark Not Found
   MCR  = Media Change Requested
   ABRT = Command aborted
   T0NF = Track 0 Not Found
   AMNF = Address Mark Not Found
```

Example IDE Driver

```
void ide_rw(struct buf *b) {
  acquire(&ide_lock);
  for (struct buf **pp = &ide_queue; *pp; pp=&(*pp)->qnext)
    ;                                // walk queue
  *pp = b;                           // add request to end
  if (ide_queue == b)                // if q is empty
    ide_start_request(b);            // send req to disk
  while ((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
    sleep(b, &ide_lock);             // wait for completion
  release(&ide_lock);
}
```

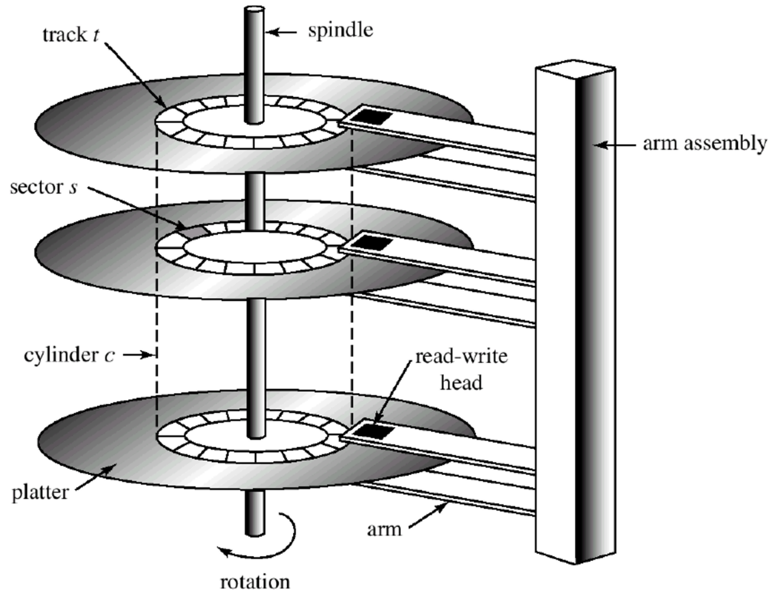
```
static void ide_start_request(struct buf *b) {
  ide_wait_ready();
  outb(0x3f6, 0);                  // generate interrupt
  outb(0x1f2, 1);                  // how many sectors?
  outb(0x1f3, b->sector & 0xff);   // LBA goes here ...
  outb(0x1f4, (b->sector >> 8) & 0xff);   // ... and here
  outb(0x1f5, (b->sector >> 16) & 0xff);  // ... and here!
  outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
  if(b->flags & B_DIRTY){
    outb(0x1f7, IDE_CMD_WRITE);    // this is a WRITE
    outsl(0x1f0, b->data, 512/4);  // transfer data too!
  } else {
    outb(0x1f7, IDE_CMD_READ);     // this is a READ (no data)
  }
}
```

Persistence

- 36 - I/O Devices
- 37 - Hard Disk Drives
- 38 - RAID
- 39 - File and Directories

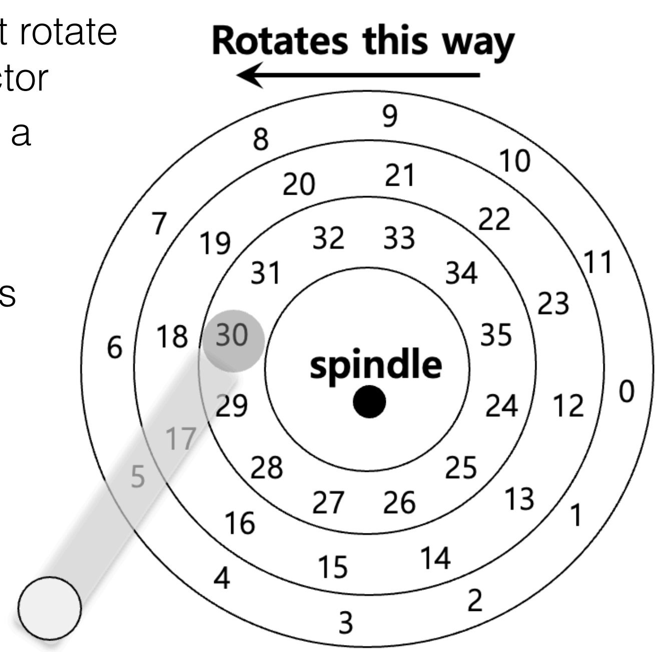
Magnetic Hard Drives

- *platter* has set of concentric tracks
- each track divided into sectors
- sectors read by read-write head



Computing the Cost

- Cost is:
  + seek time: move to correct track
  + rotational delay: disk must rotate until we get to correct sector
  + transfer time: time to read a sector
- Also, disk has:
  - track cache: head always reading, remembering
  - scheduler: more later…



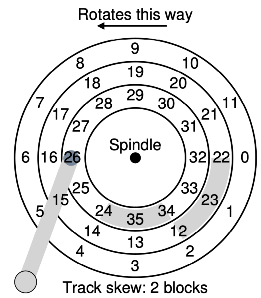
I/O Speeds

- I/O time defined as:
  - $T_{I/O} = T_{seek} + T_{rotation} + T_{transfer}$
- Rate of I/O:
  - $R_{I/O} = \dfrac{Size_{transfer}}{T_{I/O}}$
- Workload types
  - random - need a seek
  - sequential - consecutive blocks should not require seek

Example

- Examples:
  - WD 6TB Red Plus, 5400 RPM, SATA 6Gb/sec, 128 MB cache (2024)
- 5400 RPM, 100 sectors/track, sector 4KB, seek time 2 msec:
  - 5400 RPM $\Rightarrow \dfrac{1}{5400/60} = 11.1$msec/rot $\Rightarrow$ avg rot latency = 5.50 msec
  - $t_{transfer} = 11.1$msec$/100$ = 0.11 msec
  - seek time = 3.00 msec
  - tota = 8.61 msec
  - *Implies*: $1000/8.61 = 116$sectors/sec $= 116 \times 4096$ = **475 MB/sec**
- But…they claim much higher average throughput
  - constantly reading/caching everything under head
  - locality, locality, locality.
  - sequential I/O is a Good Thing

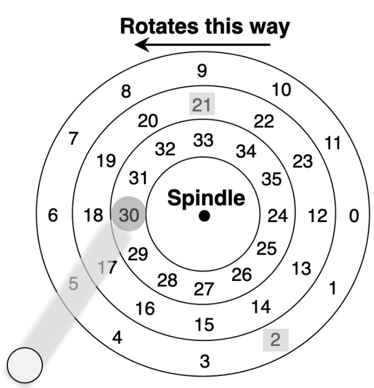
Optimizations



- track cache:
  - read head always reading
- track *skew*:
  - sectors laid out so if cross track boundaries, no extra delay
- When to ack:
  - write-back
    - ack when data in memory *dangerous! but fast!*
  - write-through
    - ack when data on disk *safe*

Disk Scheduling

- Shortest-seek-time First (SSTF)
  - order the request queue by track
  - pick requests on the nearest queue



SSTF: Scheduling Request 21 and 2

Issue the request to 21 → issue the request to 2

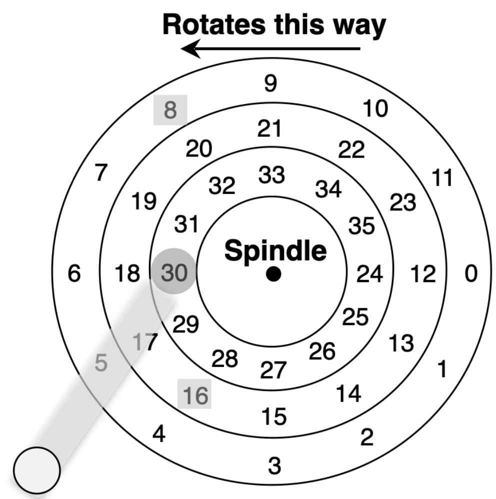
- Downsides
  - OS doesn't know drive geometry
  - *starvation…*

Elevator

- Move across the disk servicing requests in order of tracks
  - SCAN: back and forth across tracks
    - outer-to-inner, then inner-to-outer
    - If request arrives for track on current sweep, it is queued until next sweep
  - F-SCAN
    - Freeze queue while doing a sweep
    - Avoids starvation of distant requests
  - C-SCAN (circular scan)
    - Sweep from outer-to-inner, reset, then outer-to-inner, etc.

How to Account for Positioning?

- If seeks much slower than rot. lat.:
  - optimize for shorter seeks
  - request **16 is next**
  - SSTF is fine
- If seeks much faster than rot. lat.:
  - optimize for smaller rotation lat.
  - 8 **is next**
- SPTF:
  - Shortest positioning time first
  - OS does not have information
- On-disk scheduler
  - efficient SPTF
  - I/O merging



SSTF: Sometimes Not Good Enough

Sequential vs Random Example

- sequential (`S`) vs random (`R`). Assume:
  - **Sequential** : transfer 10 MB on average as continuous data.
  - **Random** : transfer 10 KB on average.
  - Average seek time: 7 ms
  - Average rotational delay: 3 ms
  - Transfer rate of disk: 50 MB/s

- Results:
  - `S` = $\frac{Amount\ of\ Data}{Time\ to\ access} = \frac{10\ MB}{210\ ms}$ = 47.62 MB /s
  - `R` = $\frac{Amount\ of\ Data}{Time\ to\ access} = \frac{10\ KB}{10.195\ ms}$ = 0.981 MB /s