CS-446/646

I/O & Disks

C. Papachristos
Robotic Workers (RoboWork) Lab
University of Nevada, Reno

I/O & Disks

OS Abstractions

Virtualization	Concurrency	Persistence
Processes	Threads	I/O
Scheduling	Synchronization	Disks
Virtual Memory	Semaphores & Monitors	Filesystems

I/O Management is another major component of OS

- Important aspect of computer operation
- I/O Devices vary greatly
  - Various methods to control them
- New types of Devices pop up constantly

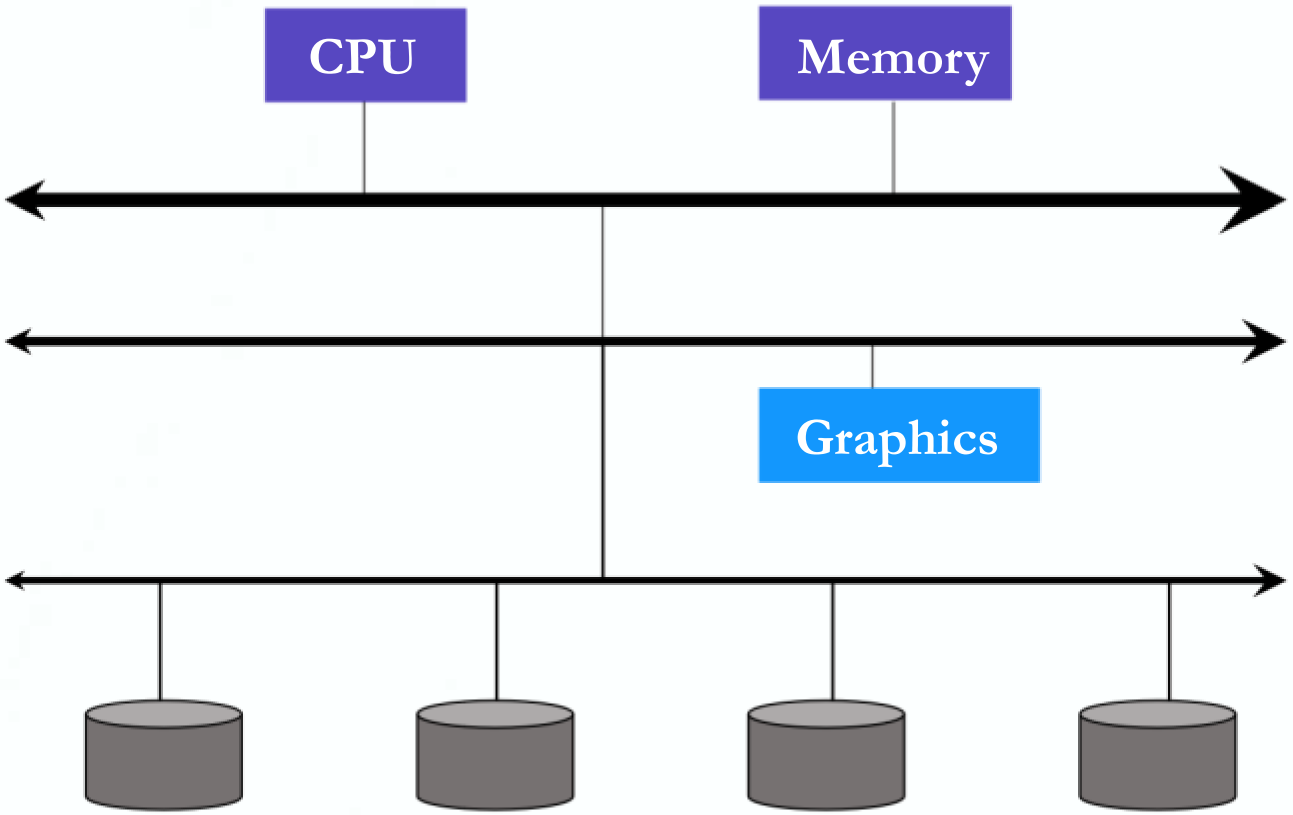
I/O Devices

…

Issues to address:

- How should I/O be integrated into systems?
- What are the general mechanisms?
- How can we manage them efficiently?

Structure of Input/Output (I/O) Device



Memory *Bus*
(proprietary)

General I/O *Bus*
(e.g. PCI)

Peripheral I/O *Bus*
(e.g. SCSI, SATA, USB)

I/O Device Interfaces

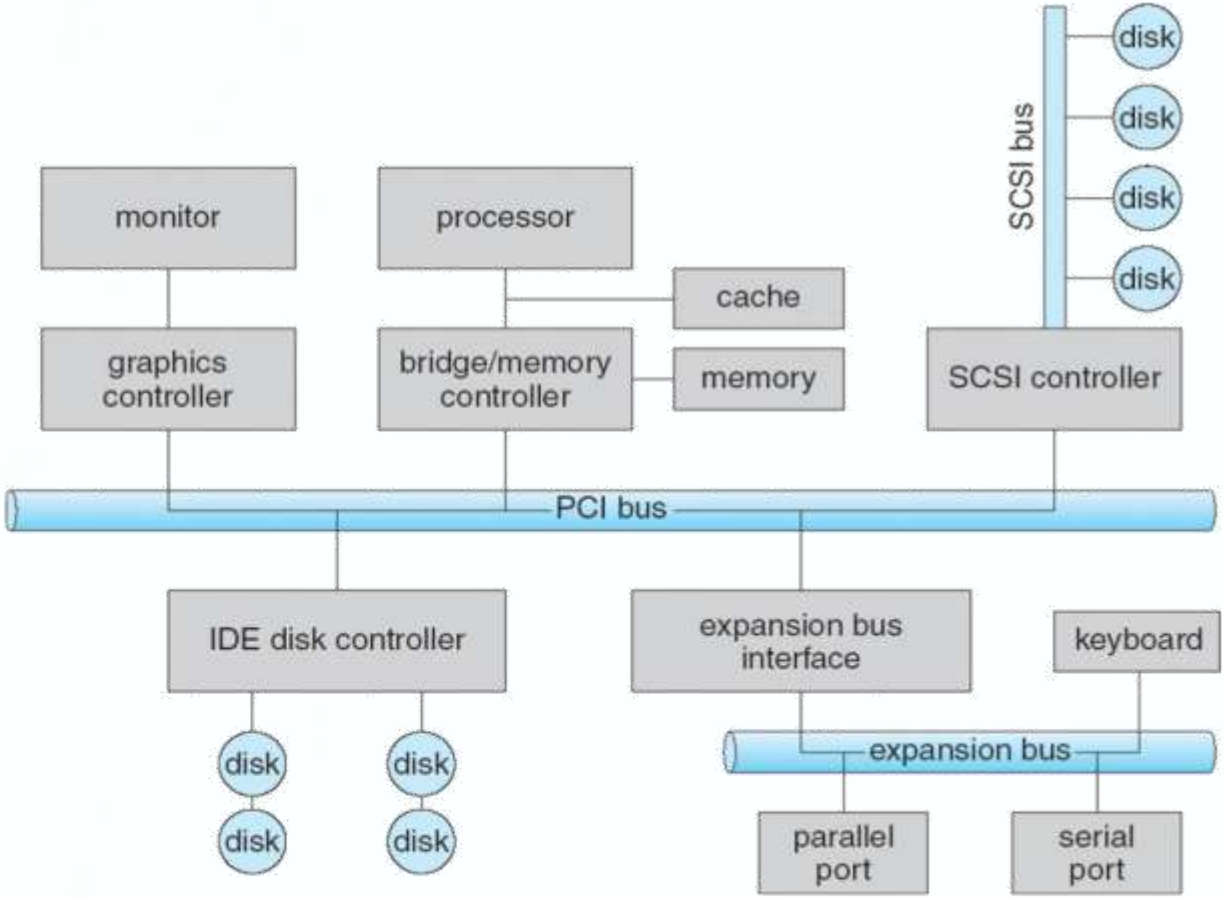
- *Port*

  Connection point (I/O special Address) for Device
  - Serial *Port*
- *Bus*

  Daisy chain for Devices sharing a common set of wires
  - *PCI Bus* (Parallel-Interface) common in PCs and servers
  - *PCI Express (PCIe) Bus* (high-speed Serial-Interface)
  - *Expansion Bus* (connects relatively slow Devices)
- *Controller*

  Electronics that operate *Port*, *Bus*, *Device*
  - Sometimes integrated, sometimes separate circuit board (Host Adapter)
  - Contains *Processor*, *Microcode*, private Memory, *Bus Controller*, etc.
  - Some talk to per-Device *Controller* with *Bus Controller*, *Microcode*, Memory, etc.

I/O Bus – *Example:* ***Peripheral Component Interconnect (PCI) Bus***



Device "Standardized" I/O *Port* Mappings on PCs

"I/O *Port*": Technical term for a specific "special use" Address on the x86's I/O *Bus*

- Legacy
  - Used by older Hardware that was present on pre-PCI systems
    - e.g. Floppy Drive, Serial Port, Parallel Port
  - Modern Architectures utilize "*Memory-Mapped*" I/O (more later) for Device communication
    - do not even have a predefined I/O *Bus*

Port range	Summary
0x0000-0x001F	The first legacy DMA controller, often used for transfers to floppies.
0x0020-0x0021	The first Programmable Interrupt Controller
0x0022-0x0023	Access to the Model-Specific Registers of Cyrix processors.
0x0040-0x0047	The PIT (Programmable Interval Timer)
0x0060-0x0064	The "8042" PS/2 Controller or its predecessors, dealing with keyboards and mice.
0x0070-0x0071	The CMOS and RTC registers
0x0080-0x008F	The DMA (Page registers)
0x0092	The location of the fast A20 gate register
0x00A0-0x00A1	The second PIC
0x00C0-0x00DF	The second DMA controller, often used for soundblasters
0x00E9	Home of the Port E9 Hack. Used on some emulators to directly send text to the hosts' console.
0x0170-0x0177	The secondary ATA harddisk controller.
0x01F0-0x01F7	The primary ATA harddisk controller.
0x0278-0x027A	Parallel port
0x02F8-0x02FF	Second serial port
0x03B0-0x03DF	The range used for the IBM VGA, its direct predecessors, as well as any modern video card in legacy mode.
0x03F0-0x03F7	Floppy disk controller
0x03F8-0x03FF	First serial port

Note: Also see https://wiki.osdev.org/I/O_Ports

Canonical I/O Device

OS reads/writes to these

Device *Registers*			**Interface**
Status	*Command*	*Data*	
Micro-controller (CPU) Memory (DRAM or SRAM or both) Other Hardware-specific Chips			**Internals**

Hardware Interface Of Canonical Device

Registers-based:

- By reading or writing the three *Registers*, OS controls Device behavior

- *Status* : Read the current operating status of the Device
- *Command* : Write to command the Device to perform a certain task
- *Data* : Write data to the Device, or read data from the Device

- Typical interaction example:

```
while (STATUS == BUSY); //wait until device is not busy

write data to data register
write command to command register //doing this starts the device and executes the command

while (STATUS == BUSY); //wait until device is done with your request
```

Device Interaction

How the OS communicates with the Device:

- I/O *Instructions* control Devices
  - **`in`** and **`out`** *Instructions* on x86
    - *"I/O Mapping"*: Special control signal from the CPU to indicate that access is performed to an I/O Port rather than a regular *Memory* location
  - Devices usually have *Registers*
    - Device *Driver* places *Commands*, *Addresses*, and *Data* these, in order to perform read/write
- *"Memory-Mapped"* I/O
  - Device *Registers* available as if they were Memory locations
    - I/O *Ports* *"Memory-Mapped"* within the same unified *Address Space* as ordinary *Memory* (but in a special reserved Address Region)
  - OS performs **`load`**s (to read) or **`store`**s (to write) to the Device, instead of Main *Memory*

x86 I/O *Instructions*

- Used in conjunction with I/O *Port* Mappings

Example: Pintos **threads/io.h**

```
static inline uint8_t inb (uint16_t port) {
  uint8_t data;
  asm volatile ("inb %w1, %b0" : "=a" (data) : "Nd" (port));
  return data;
}

static inline void outb (uint16_t port, uint8_t data) {
  asm volatile ("outb %b0, %w1" : : "a" (data), "Nd" (port));
}

static inline void insw (uint16_t port, void *addr, size_t cnt) {
  asm volatile ("rep insw" : "+D" (addr), "+c" (cnt) : "d" (port) : "memory");
}
```

Example: IDE Disk Driver w/ x86 I/O *Instructions*

```
void IDE_ReadSector(int disk,
                    int off,
                    void *buf) {
  // Select Drive
  outb( 0x1F6 , disk == 0 ? 0xE0 : 0xF0);
  IDEWait();
  // Read length (1 Sector = 512 B)
  outb( 0x1F2 , 1); // 1 Sector
  outb( 0x1F3 , off); // Logical Block Address low
  outb( 0x1F4 , off >> 8); // Logical Block Address mid
  outb( 0x1F5 , off >> 16); // Logical Block Address high
  outb( 0x1F7 , 0x20); // Read command
  insw( 0x1F0 , buf, 256); // Read 256 words
}
```

```
void IDE_Wait() {
  // Discard status 4 times
  inb( 0x1F7 ); inb( 0x1F7 );
  inb( 0x1F7 ); inb( 0x1F7 );
  // Wait for status BUSY flag to clear
  while ((inb( 0x1F7 ) & 0x80) != 0);
}
```

Remember:

0x01F0-0x01F7	The primary ATA harddisk controller.

Memory-Mapped I/O

I/O *Port* Mappings & **in**/**out** *Instructions* are slow and clunky

- *Instruction* format restricts what *Registers* you can use
- Only allows 216 different *Port* numbers
- Per-*Port* access control turns out to not be as useful
  - any *Port* access allows you to disable all *Interrupts*

Devices can achieve same effect with *Physical Addresses*, e.g.:

```
volatile int32_t *device_control = (int32_t *) (0xc0100 + PHYS_BASE);
*device_control = 0x80;  // write
int32_t status = *device_control;  // read
```

- Kernel must ensure specific Mapping of these *Physical* to *Virtual Addresses* across entire OS, and ensure they are *non-Cacheable*

Polling

OS waits until the Device is ready by repeatedly reading the *Status Register*

- Positive: Simple and working
- Negative: Wastes CPU time continuously waiting for the Device
  - Switching to another Ready-state *Task* would be better utilization of the CPU

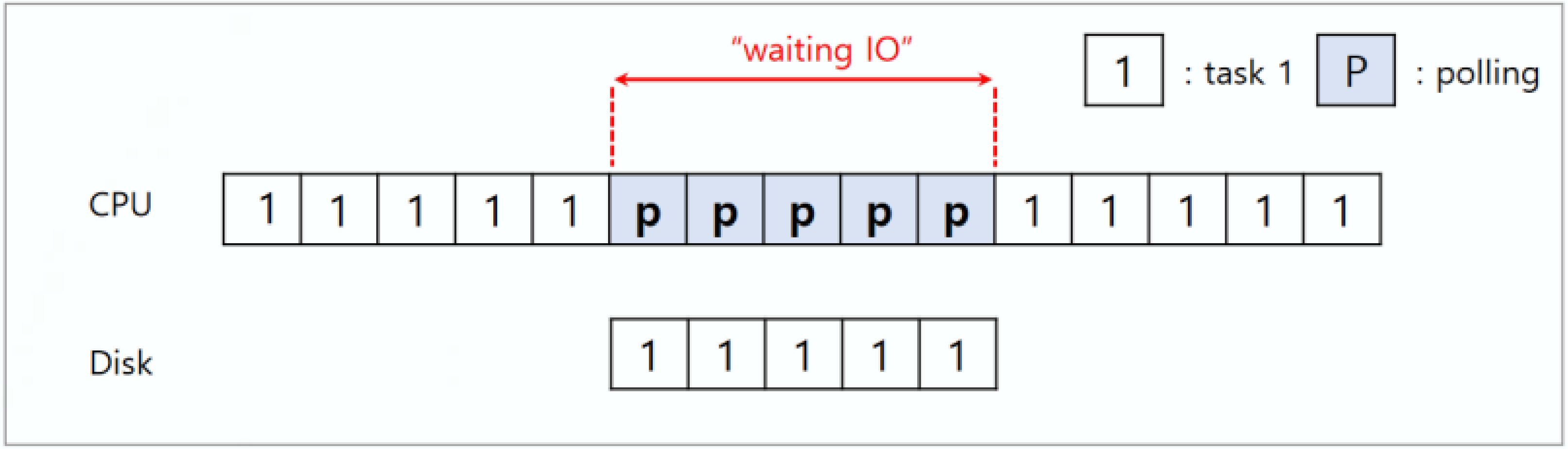


Diagram of CPU utilization when *Polling*

Interrupts

OS puts the I/O-requesting *Process* to Sleep and *Context Switches* to another
When the Device is finished, the *Process* is woken-up via *Interrupt*

- Positive: CPU and the I/O Device are more appropriately utilized
- Negative: …

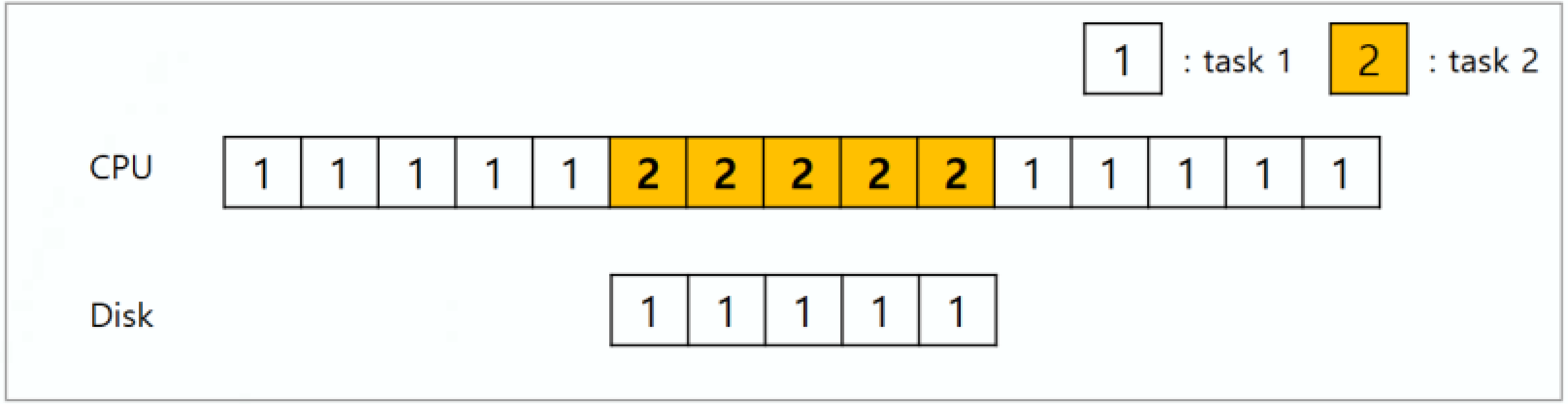


Diagram of CPU utilization with *Interrupts*

Polling vs ***Interrupts***

However, "*Interrupts* is not always the best solution"
- If Device performs very quickly, *Interrupts* will "slow down" the system

e.g. high *Network Packet* arrival rate
- *Network Packets* can arrive faster than OS can process them
- *Interrupts* are very expensive (*Context Switching*)
- *Interrupt Handlers* have high *Priority*
- Worst case: Spend 100% of time in *Interrupt Handlers* never making any progress – "*Receive Livelock*"
- Best case: Adaptive switching between *Interrupts* and *Polling*

Rule-of-Thumb
- If Device is fast → *Polling*
- If it is slow → *Interrupts*

Protocol Variants

Device *Registers*

Status | *Command* | *Data*

Micro-controller (CPU)
Memory (DRAM or SRAM or both)
Other Hardware-specific Chips

- *Status* checking
  - *Polling* –vs– *Interrupts*
- *Data*
  - *Programmed I/O (PIO)* –vs– *Direct Memory Access (DMA)*
- *Control*
  - Special *Command Instructions* –vs– *Memory-Mapped* I/O

Variety is a Challenge

Problem:

- Many different Devices
- Each has its own Protocol

We want to avoid writing a slightly different OS for each piece of Hardware

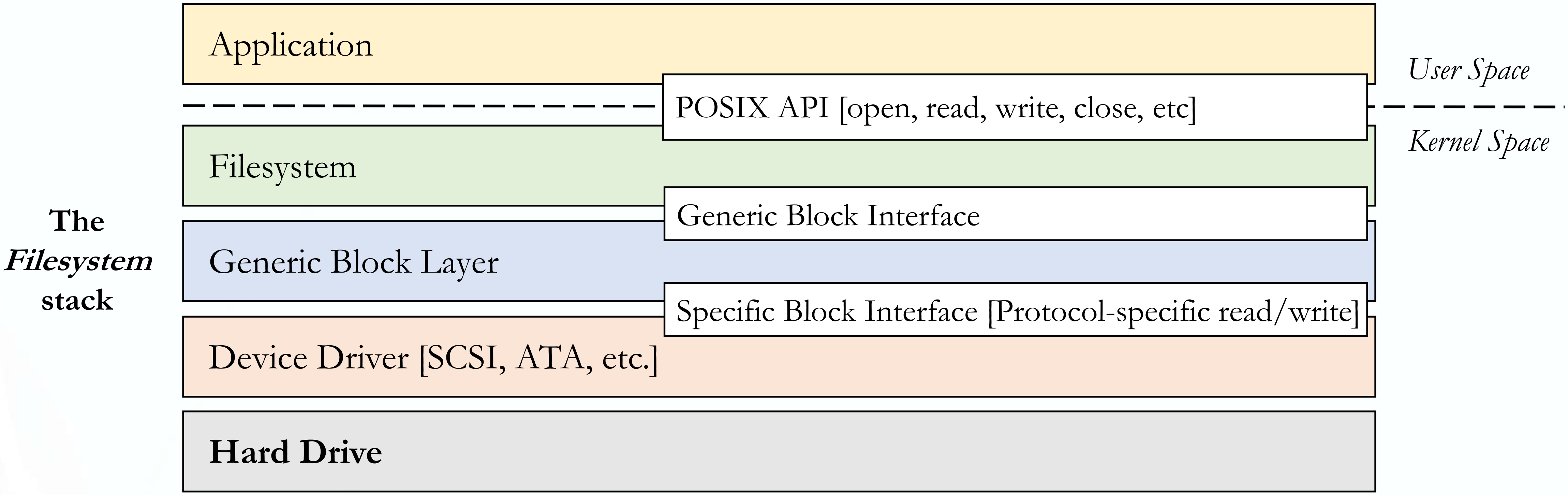
Solution: *Abstraction*

- Build a common Interface
- Write a specific Device *Driver* for each Device
- *Drivers* are 70% of Linux source code

Filesystem Abstraction

Filesystem specifics of which Disk class it is using

- e.g. it issues *Block* read and write request to the *Generic Block Interface* layer



Hard Disks – Basic Interface

- Disk *Interface* represents a linear array of *Sectors*
  - *Sector:* Historically 512 Bytes for *Hard Disk Drives* (HDDs)
  - Written atomically (even if there is a power failure)
  - 4 KB in newer "*Advanced Format*" (AF) Disks – HDDs and *Solid State Drives* (SSDs)
    - "*Torn Write*" – In an untimely power loss, only a portion of a larger write may complete
- *Disk Controller* maps the (linear) *Logical Sector Numbers* to *Physical Sectors*
  - *Physical Sectors* identified by *Surface #, Track #, Sector #* (next slides)
- OS doesn't know *Logical Sector Number* to *Physical Sector* Mapping

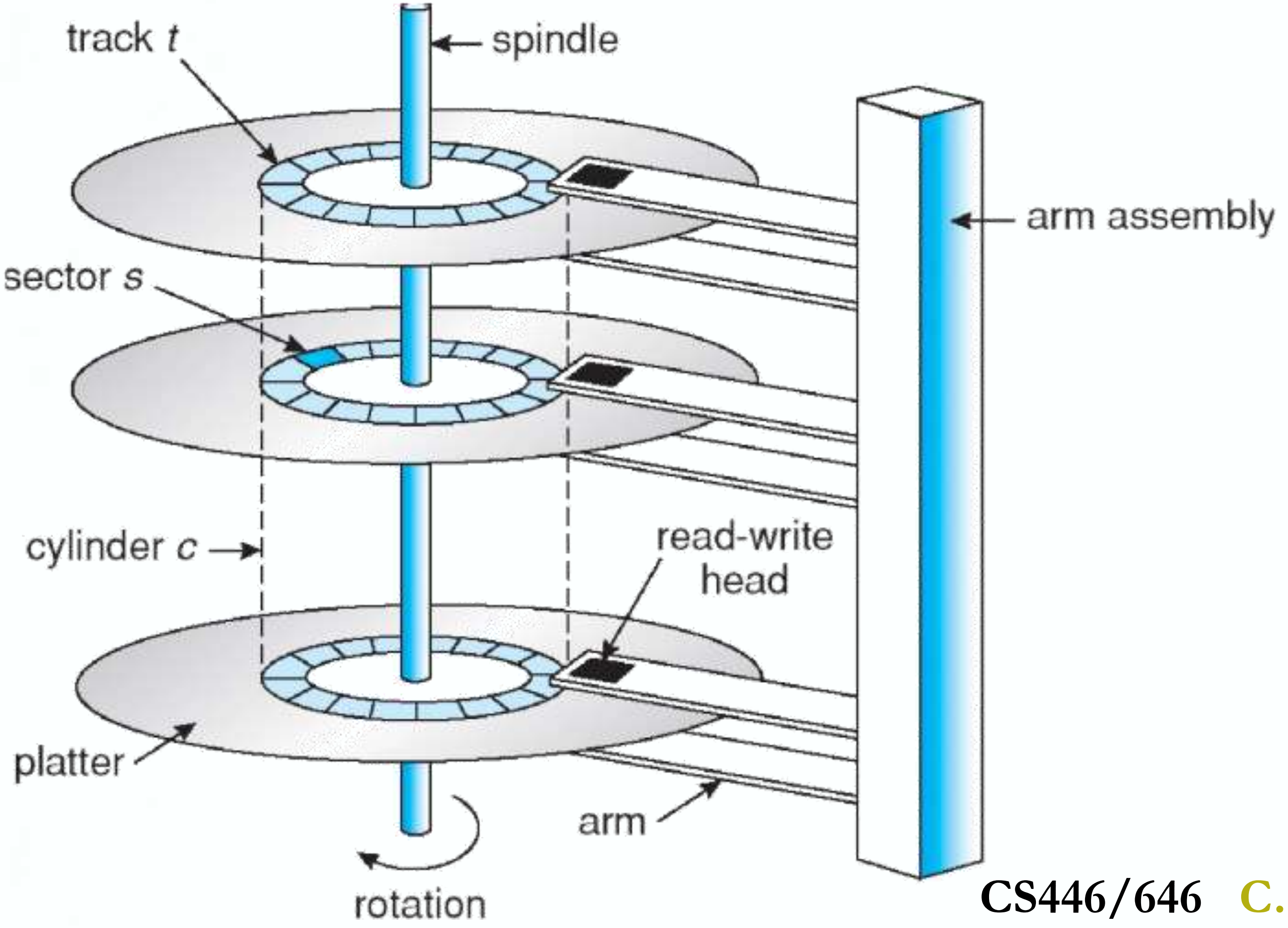
Hard Disks – Basic Geometry

- *Platter* (Aluminum coated with a thin magnetic layer)
  - Disk-shaped
  - Data is stored persistently by inducing magnetic changes to it
  - Each *Platter* has 2 sides, each of which is called a *Surface*

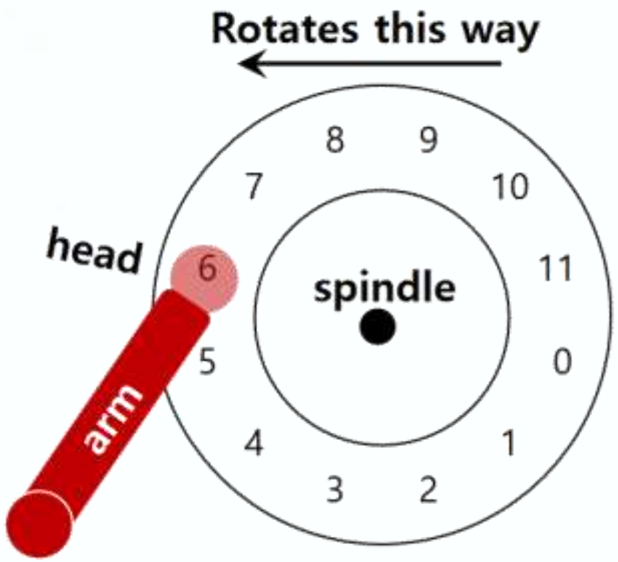
Hard Disks – Basic Geometry

- *Spindle*
  - *Spindle* is connected to a motor that spins the *Platters* around
  - The rate of rotations is measured in *RPM (Revolutions Per Minute)*
    - Typical modern values : 7,200 RPM to 15,000 RPM
- *Track*
  - Concentric circles of *Sectors*
  - Data is encoded on each *Surface* in a *Track*
  - A single *Surface* contains many thousands of *Tracks*
- *Cylinder*
  - A stack of *Tracks* of fixed radius
  - Heads record and sense data along *Cylinders*
  - Generally only one *Head* active at a time

Cylinders, Tracks, Sectors



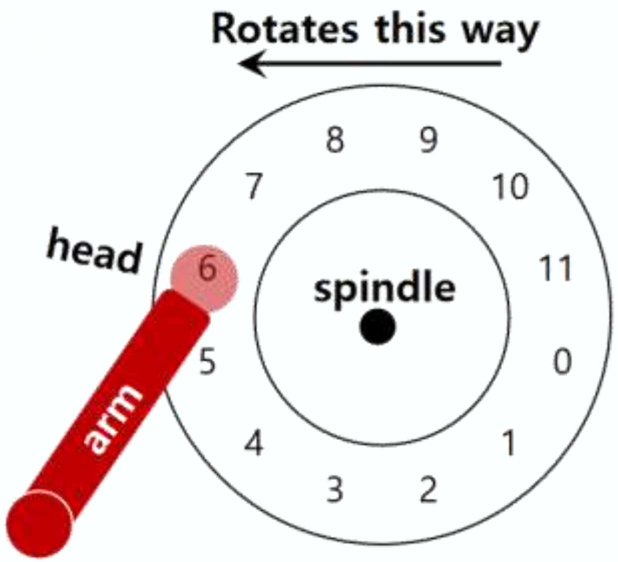
A simple Hard Disk Drive



A single *Track* + a *Head*

- Disk *Head* – One *Head* per *Surface* of the Drive
  - The process of reading and writing is accomplished by the Disk *Head*
  - Attached to a single Disk arm, which moves across the *Surface*

Single-track Latency: The Rotational Delay



A single *Track* + a *Head*

- *Rotational Delay*: Time for the desired *Sector* to rotate
  - Example: Full Rotational delay is R and we start at Sector 6
  - Read sector 0: *Rotational Delay* = R/2
  - Read sector 5: *Rotational Delay* = R-1 (worst case)

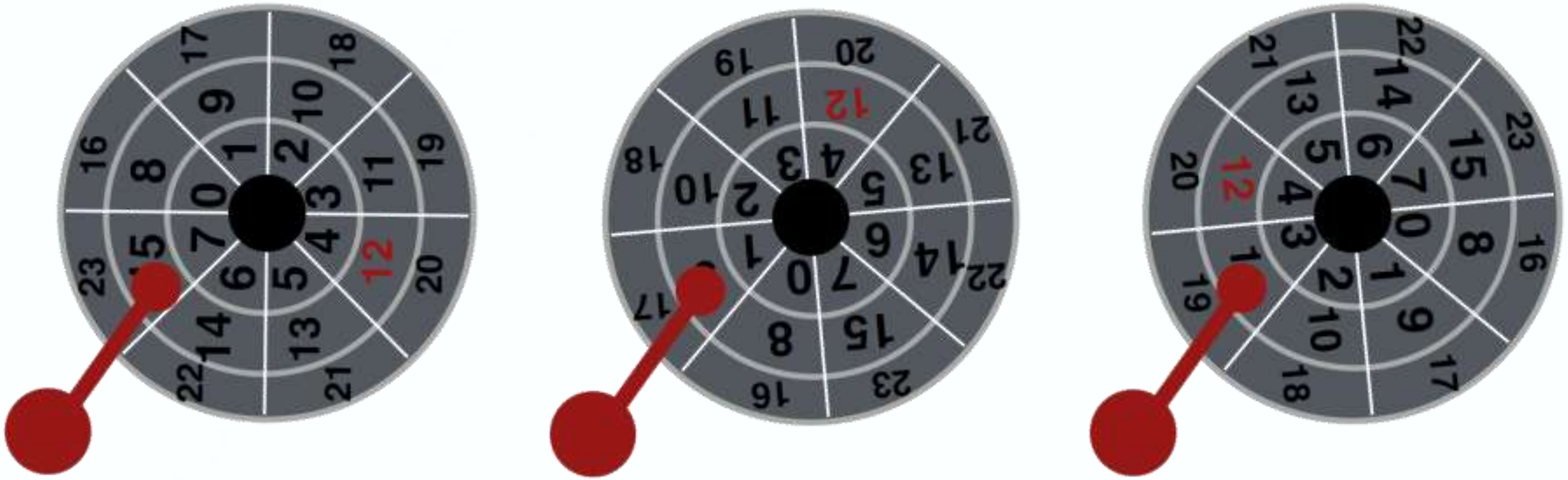
Multiple Tracks: Start a Read

➢ Goal: Read Sector 12

Multiple Tracks: *Seek* to *Track (/Cylinder)*

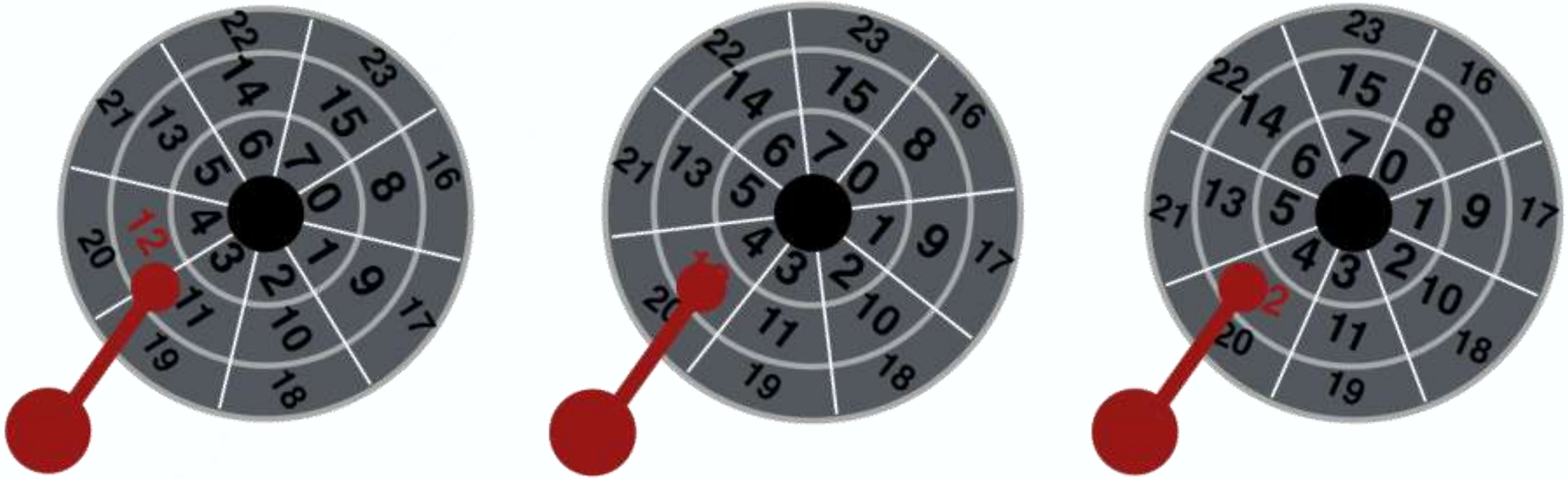
- Goal: Read Sector 12
  - *Seek Time*: Slow (e.g. more than 0.5 – 2 ms)

Multiple Tracks: Wait for *Rotation*



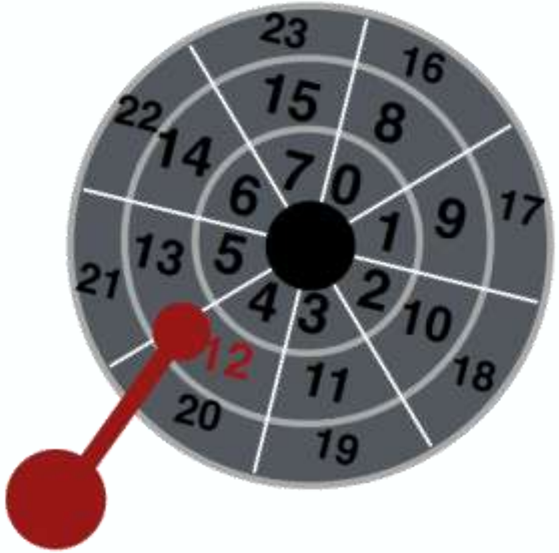
- Goal: Read Sector 12
  - *Rotation Time*: Still slow (depends on mechanical motion)

Multiple Tracks: *Transfer Data*



- Goal: Read Sector 12
  - *Transfer Rate*: Fast (e.g. 125 MB/s)
    - *Transfer Time*: Fast

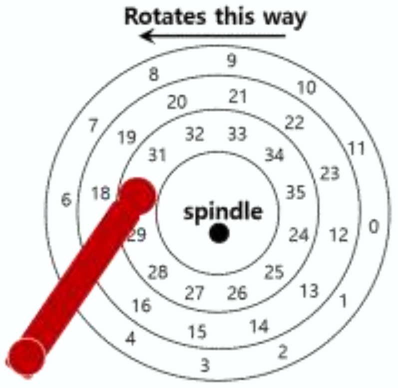
Multiple Tracks: *Transaction Complete*

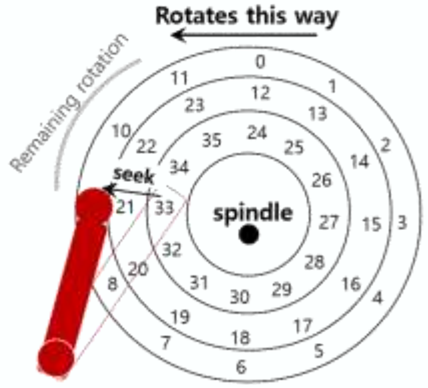


- Goal: Read Sector 12

Disk Latencies

- *Seek* : Move the Disk arm to the correct *Track (/Cylinder)*
- *Seek Time* : Time to move *Head* to *Track* that contains the desired *Sector*
    - One of the most costly Disk operations





- *Rotational Delay* : Disk rotation until the correct *Sector* is reached by the *Head*
    - Still slow
- *Transfer* : Transfer of bits of information from the desired *Sector*
- *Transfer Time* :Time to perform *Transfer*
    - Fast
- *I/O Time* : *Seek* + *Rotation* + *Transfer*

***Seek**, Rotate, Transfer*

- *Acceleration* → *Coasting* → *Deceleration* → *Settling*
- *Acceleration*: The Disk Arm gets moving
- *Coasting*: The Arm is moving at full speed
- *Deceleration*: The Arm slows down
- *Settling*: The Head is carefully positioned over the correct track

- *Seeks* often take several milliseconds!
  - *Settling* alone can take 0.5 to 2ms
  - Entire *Seek* often takes 4 - 10 ms

I/O & Disks

Seek, ***Rotate***, *Transfer*

- Depends on *Revolutions Per Minute (RPM)*
  - 7,200 RPM is common, 15,000 RPM is high-end
- At 7,200 RPM, time to complete 1 revolution?
  - 1 min / 7,200 RPM = 1 second / 120 revolutions = 8.3 ms / revolution
- "Average" revolution time
  - 8.3 ms / 2 = 4.15 ms

Seek, *Rotate*, ***Transfer***

- The final phase of I/O
  - Data is either read from or written to the *Surface*
- Fast – Depends on RPM and *Sector* density
  - 100+ MB/s is typical for maximum transfer rate
- *Example:* Time to transfer 512 B:
  - 512 B * (1 sec / 100 MB) = 5 μs

Workload

- *Seeks* are slow
- *Rotations* are slow
- *Transfers* are fast

What kind of *Workload* is fastest for Disks?

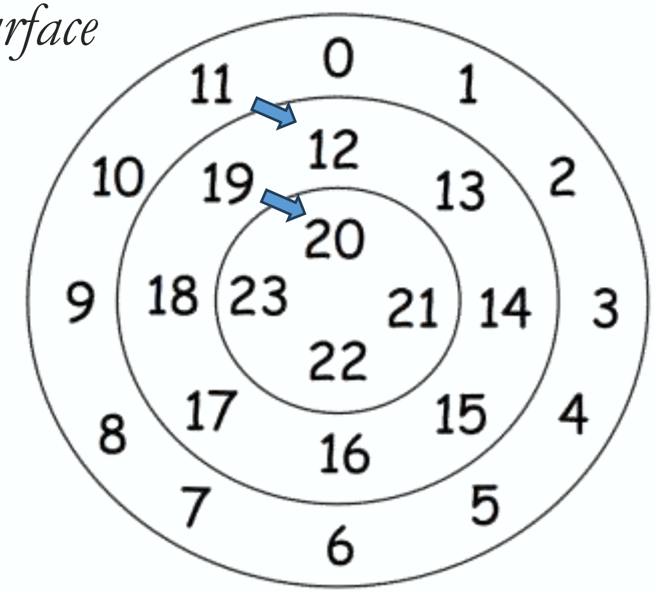
- *Sequential* :
  Access *Sectors* in order (*Transfer*-dominated)
- *Random* :
  Access *Sectors* arbitrarily (*Seek*-&-*Rotation*-dominated)

Sector Mapping

- Mapping of the (linear) *Logical Sectors* to *Physical Sectors*

Logical Sector 0

- The first *Sector* of the first (outermost) *Track* of the first *Surface*
- *Logical Sector* Address incremented within *Track*, then *Tracks* within *Cylinder*, then across *Cylinders*, from outermost to innermost
- *"Track Skew"*



Sector Mapping

➢ *Default Mapping*

Advantages

➢ Simple to implement
➢ *Default Mapping* reduces *Seek Time* for *Sequential* Access

Limitations

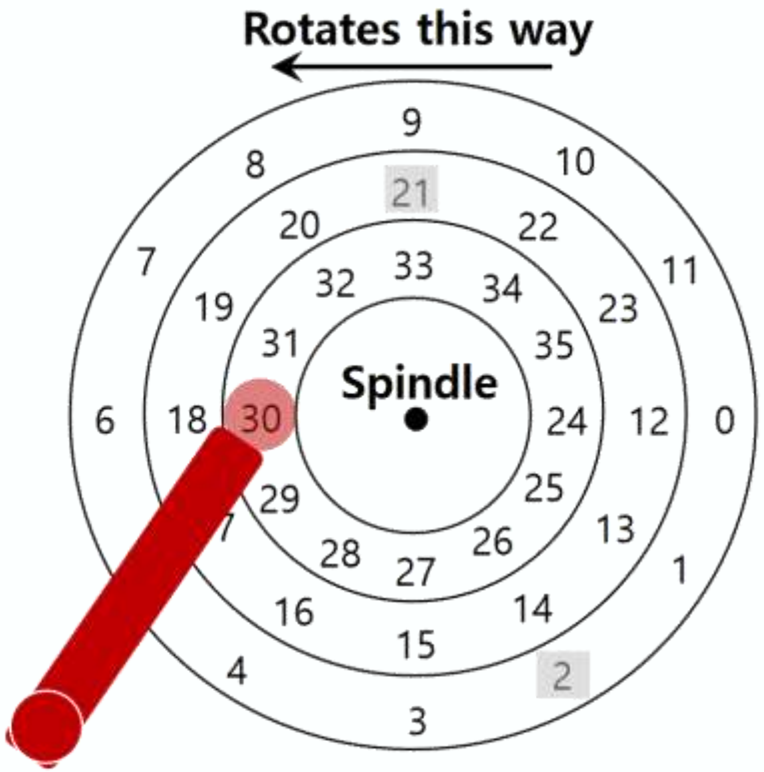
➢ *Filesystem* can't infer mapping
➢ Reverse-engineering of mapping in OS is difficult
  - Number of *Sectors* per *Track* changes (i.e. with radius)
  - Disk Hardware can silently remap *Bad Sectors*

Disk Cache

- Separate internal *Memory* (8MB - 32MB) that is used as Hardware Cache
- *"Read-Ahead"*: Acts as *Track Buffer*
  - Read contents of entire *Track* into *Memory* during *Rotational Delay*
- Write-caching with volatile *Memory*
  - "*Write-Back*" or "*Immediate Reporting*":
    - Claim written to Disk when not actually done yet
      - Faster, but data could be lost on power failure
  - "*Write-through*":
    - *Ack* after data actually written to *Platter*

Disk Scheduling

- *Disk Scheduler* decides which I/O request to schedule next
- Goal: Minimize *Positioning Time*
  - Performed by both OS and Disk itself (Why?)
- Schedule requests in order received (FCFS)
  - Advantage: *Fairness*
  - Disadvantage: High *Seek* cost (+ *Rotation*)
- Handle nearest *Cylinder* next (SSTF)
  - Advantage: Reduces arm movement (*Seek Time*)
  - Disadvantage: Unfair, can *Starve* some requests
- One-direction *Sweeping* of Disk (SCAN / C-SCAN)
- If request comes for a *Block* already serviced in this *Sweep*, queue it for next *Sweep*



Disk Scheduling – FCFS

First Come First Served (FCFS)

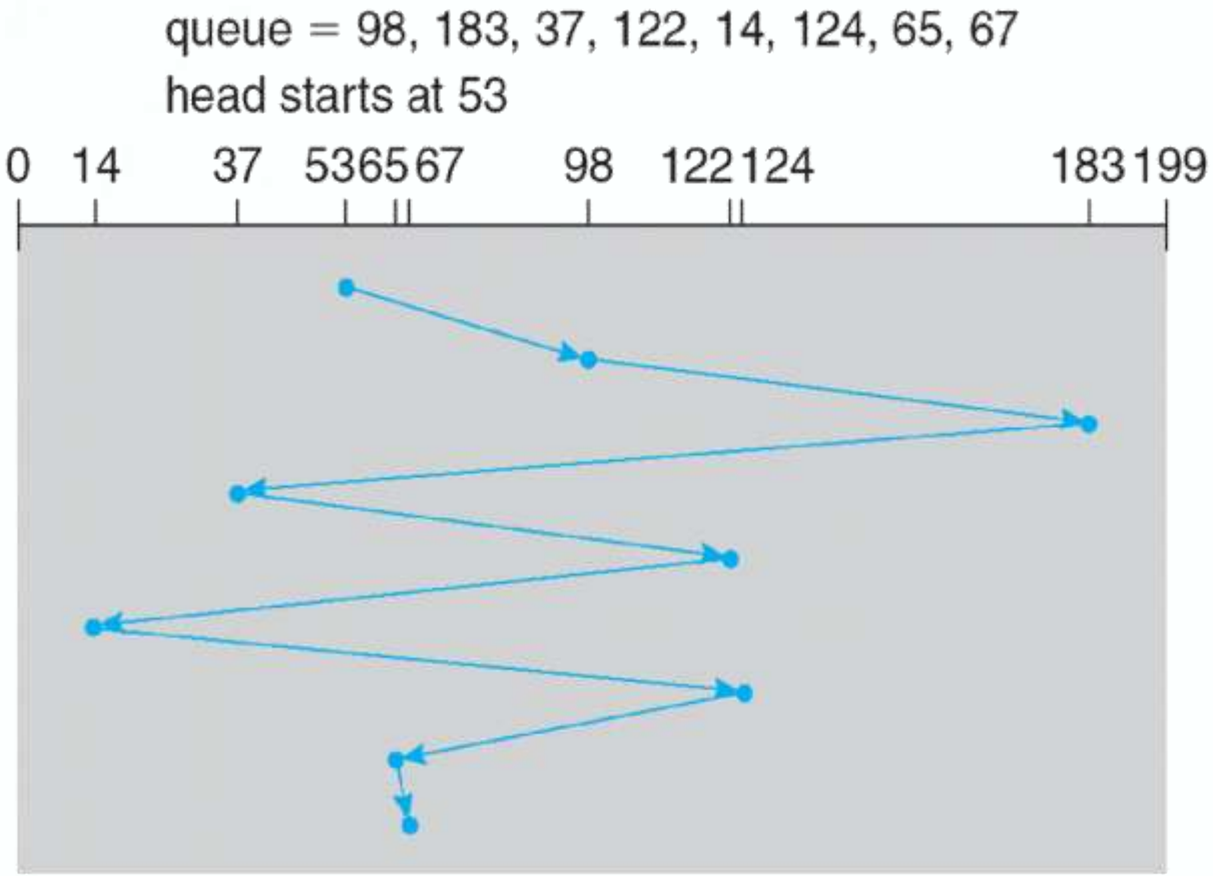
- Process Disk requests in the order they are received

- Advantages
  - Easy to implement
  - Good *Fairness*

- Disadvantages
  - Cannot exploit request *Locality*
  - Increases *Average Latency*, decreasing *Throughput*

Disk Scheduling – FCFS

- *Example:*



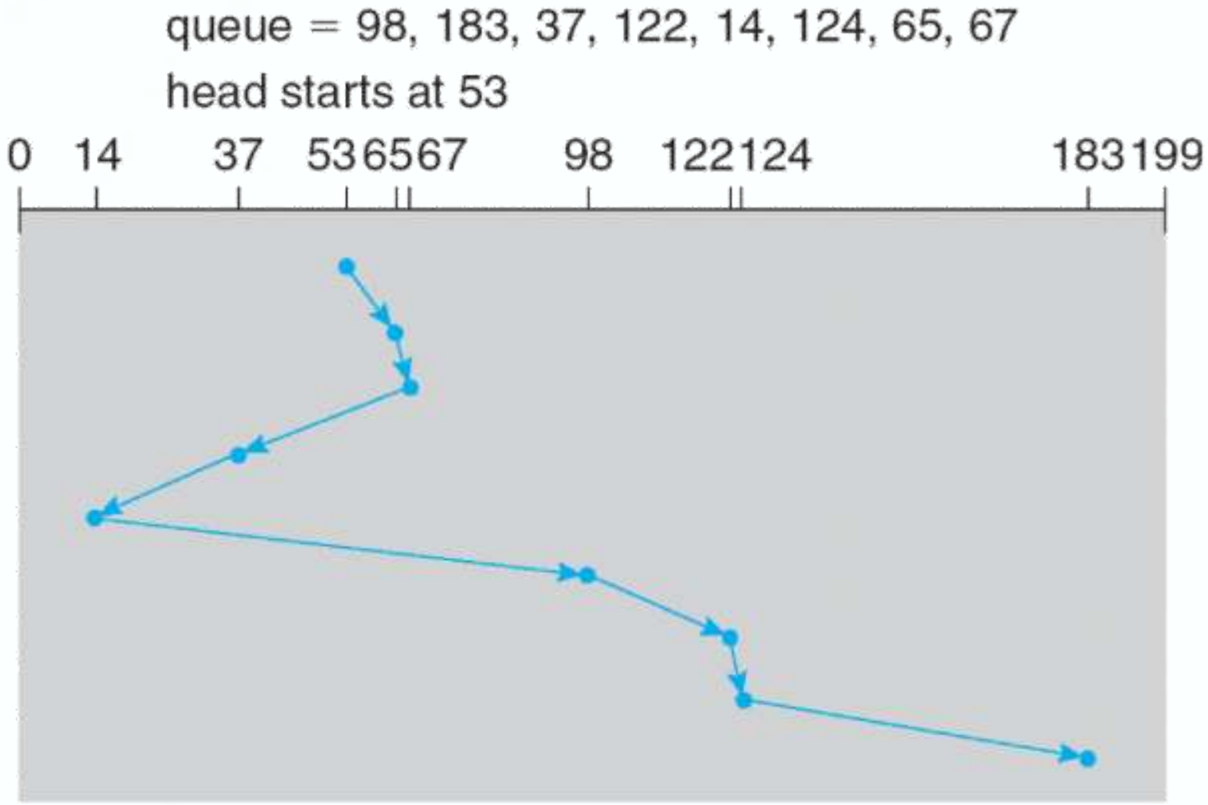
Disk Scheduling **– SSTF (/SPTF)**

Shortest Seek-Time First (SSTF) –or– *Shortest Positioning Time First* (SPTF)

- Order the queue of I/O requests by *Track*
- Pick requests on the nearest *Track* to complete first
- Advantages
  - Exploits *Locality* in Disk requests
  - Higher *Throughput*
- Disadvantages
  - *Starvation*
  - Can't always know which request will be the fastest

Disk Scheduling – SSTF (/SPTF)

➢ *Example:*



"Elevator" Scheduling – SCAN (/C-SCAN)

Sweep across Disk, servicing all requests passed

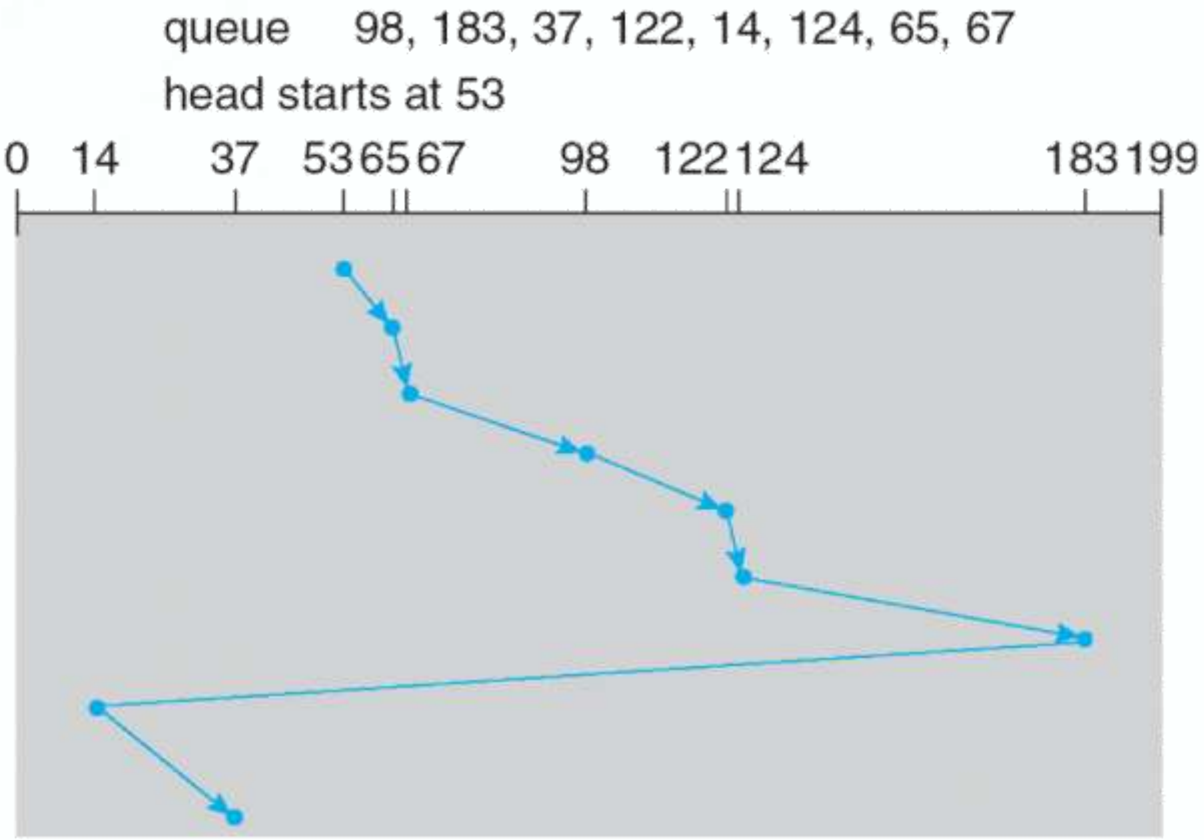
- Like SSTF, but next *Seek* must be in same direction
- Switch directions only if no further requests in same direction

- Advantages
  - Takes advantage of *Locality*
  - Bounded waiting
- Disadvantages
  - Cylinders in the middle get better service
  - Might miss some *Locality* which SSTF could exploit

- *Circular-SCAN* (C-SCAN): Only sweep in one direction
  - Very commonly used algorithm in Unix

Disk Scheduling – C-SCAN

- *Example:*



Disk Technology Trends

- Data → More dense
  - More bits per square inch
  - Disk *Head* closer to *Surface*
  - Create smaller Disks with same *Capacity*
- Disk geometry → Smaller
  - Spin faster → Increase *Bandwidth*, reduce *Rotational Delay*
  - Faster *Seek*
  - More lightweight
- Disk price → Cheaper
  - Density improving more than speed (mechanical limitations)

New *Mass Storage* Technologies

- New Solid-State Memory-based *Mass Storage* technologies avoid *Seek Time* and Rotational Delay
  - NAND Flash
  - Battery-backed DRAM (NVRAM)

Disadvantages

- Price: More expensive than same *Capacity* Disk
- Reliability: More likely to lose data

- Open research question:
  How to effectively use *Flash* in Commercial Storage systems

Flash Memory

Today, we increasingly use *Flash Memory*

- Completely Solid-State (no moving parts)
  - Remembers Data by storing charge
  - Lower power consumption and heat
  - No mechanical *Seek* Times to worry about
- Limited # of overwrites possible
  - *Blocks* wear out after 10,000 (MLC) – 100,000 (SLC) erases
  - Requires *Flash Translation Layer (FTL)* to provide wear leveling, so repeated writes to same *Logical Blocks* don't wear out *Physical Blocks*
  - FTL can seriously impact performance
- Limited durability
  - Charge wears out over time
  - Turning off Device for a year may even lead to loss of data

***Redundant Array of Independent Disks* (RAID)**

Motivation:

- Performance
  - Disks are slow compared to CPU
  - Disk speed improves slowly compared to CPU
- Reliability
  - In single-Disk systems, one Disk failure leads to data loss
- Cost
  - A single fast & reliable Disk is expensive

***Redundant Array of Independent Disks* (RAID)**

Idea:

- Use redundancy to improve Performance and Reliability
  - Redundant array of cheap Disks as one storage unit
  - Fast: Simultaneous read and write of Disks in the array
  - Reliable: Use "*XOR-Magic*" Parity:
    - To detect Errors
    - To rebuild missing data in case of any 1 Disk Failure

- RAID can have different redundancy levels, achieving different Performance and Reliability criteria
  - Seven different RAID levels (RAID *0-6*)

***Redundant Array of Independent Disks* (RAID)**

Evaluating RAID:

- Cost-wise
  - Storage utilization: Data *Capacity* / Total *Capacity*
- Reliability-wise
  - Tolerance against Disk failures
- Performance-wise
  - Perform (large) *Sequential* read, write, read-modify-write
  - Perform (small) *Random* read, write, read-modify-write
  - Measure speedup over a single Disk

***Redundant Array of Independent Disks* (RAID)**

Evaluating RAID:

- Computing Cost:
  - G = Number of Data Disks in a RAID group
  - C = Number of Check/Parity Disks in a RAID group
  - Cost = C/(G+C)

Redundant Array of Independent Disks **(RAID)**

Evaluating RAID:

- Computing Reliability:
  - N = Total number of Disks
  - G = Number of Data Disks in a RAID group
  - C = Number of Check/Parity Disks in a RAID group
  - MTTF (disk) = Mean time to failure for a single Disk
    - Estimated as MTTF (in years) = 1 / AFR (Annual Failure Rate (in percentage))
    - Ex: 114 years (1M hours) = 1 / 0.88%
    - Source: "Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?", FAST'07
  - MTTR(disk) = Mean time to repair for a failed Disk

  Compute:
  - MTTF(group) = Mean time to two failed Disks before first gets repaired in one group
  - MTTF(raid) = Mean time to failure over entire array
  - MTTF(raid) = MTTF(group) / Num. groups

Redundant Array of Independent Disks **(RAID)**

Evaluating RAID:

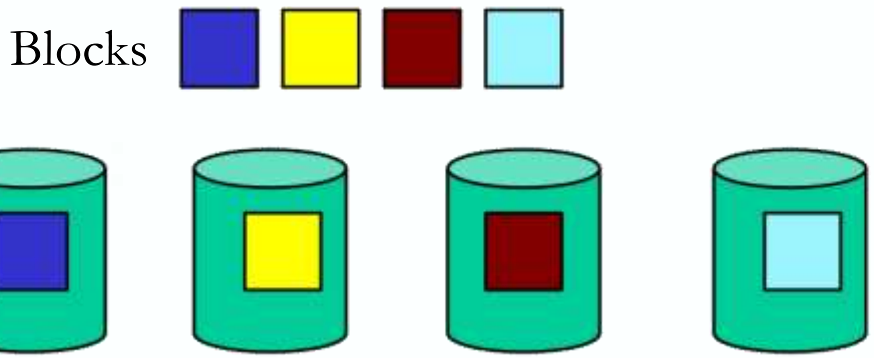
- Computing Reliability:
  - Assume single-error tolerance in one group
    - If another error comes before repair, group fails
  - MTTF(group) = MTTF(1 disk) / Prob[Another failure within MTTR]
    - If Prob ≈ 1, MTTF(group) same as MTTF(1 disk) – no benefit of RAID
    - If Prob ≈ 0, MTTF(group) approaches ∞ – good
  - MTTF(1 disk) = MTTF(disk)/(D+C)
  - MTTF(another disk) = MTTF(disk)/(D+C-1)
  - Prob[Another failure within MTTR] = MTTR/(MTTF(disk)/(D+C-1))
  - MTTF(group) = MTTF(1 disk)/Prob[Another failure within MTTR]
    = (MTTF(disk))2/((D+C)*(D+C-1)*MTTR)
  - Num groups G = N / (D+C)
  - MTTF(raid) = MTTF(group) / G = MTTF(group) / (N/(D+C))
  - Thus: MTTF(raid) = (MTTF(disk))2 / (N * (D+C-1) * MTTR)

RAID *0*: Non-Redundant Striping

Structure:
- A set of Data Sectors striped across (Data) Disks
- No Parity Disks

Advantages:
- Good performance – with N Disks, roughly N-times speedup

Disadvantages:
- Poor Reliability – one Disk failure → Data loss
- MTTF(raid)=MTTF(disk)/N



RAID *0* Performance

Large read of 100 blocks:

- One Disk: 100 * t,
- Raid0: 100/N * t * S
- S: Slowdown. Need to wait for slowest Disk to complete before returning

Performance:

- Large read: N/S
- Large write: N/S
- Large R-M-W: N/S
- Small read: N
- Small write: N
- Small R-M-W: N

RAID *1*: Mirroring

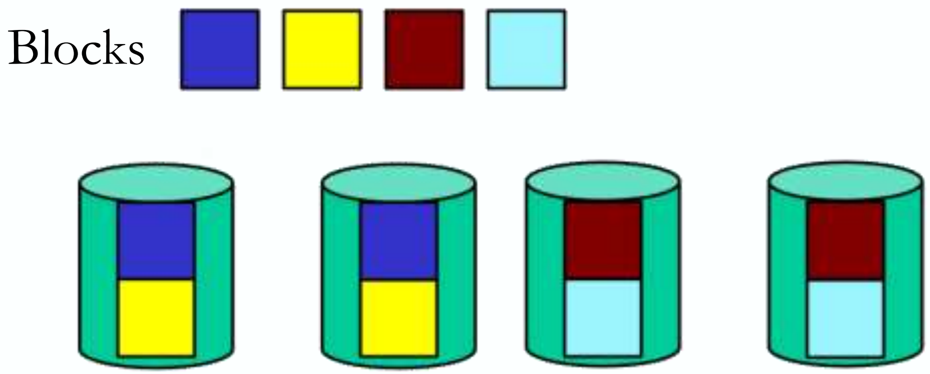
Structure: ➢ Keep a Mirrored (Shadow) copy of each Data Sector

Advantages:

- ➢ Good Reliability: 1 Disk failure OK
- ➢ Good read Performance

Disadvantages:

- ➢ High cost: Each Data Disk requires one Parity Disk



RAID *1* Performance

- Cost = C/(D+C) = 1/(1+1) = 50%
- MTTF(raid) = MTTF(disk)2/(N*MTTR)

Performance

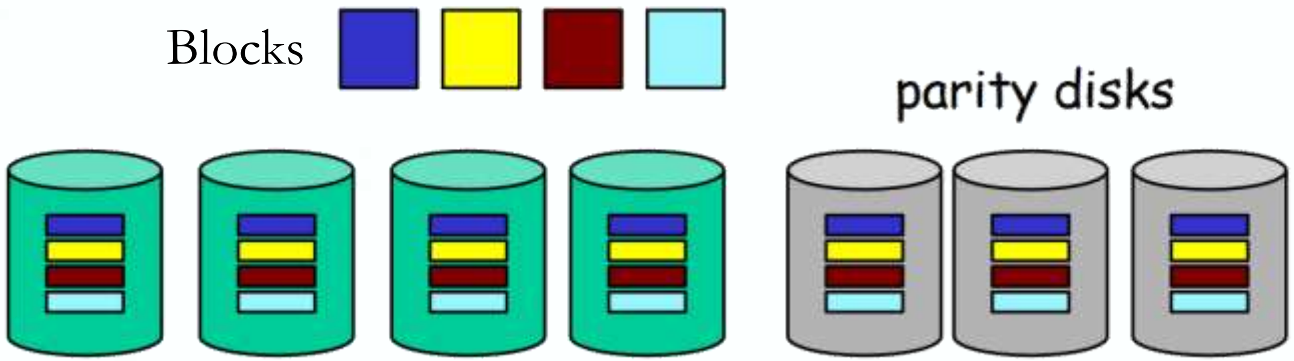
- Large read: N/S
- Large write: N/2S
- Large R-M-W: 2N/3S
  - X sectors, 2 X events (X reads, X writes)
  - Speedup (w.r.t. to 1 Disk) = 2X / (X/(N/S) + X/(N/2S)) = 2N/3S
- Small read: N (no S here since only two Disks)
- Small write: N/2
- Small R-M-W: 2N/3

RAID *2*: Memory-Style Error-Correcting Parity

Structure:
- A Data Sector striped across Data Disks
- Compute Error-Correcting Parity and store in separate Parity Disks

Advantages:
- Good Reliability with higher Storage Utilization than Mirroring

Disadvantages:
- Unnecessary cost: 1 Parity Disk can already detect Failure (coming up next)
- Poor *Random* Performance

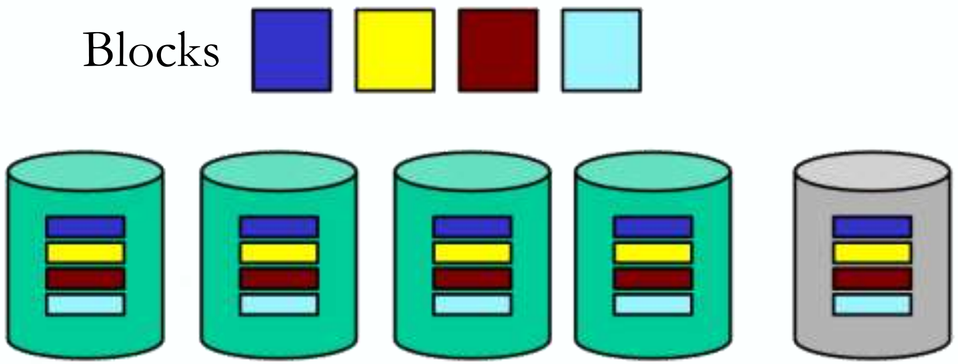


RAID *3*: Bit-Interleaved Parity

Structure:
- A Data Sector striped across Data Disks
- Single Parity Disk (XOR of each stripe of a Data Sector)

Advantages:
- Same Reliability with 1 Disk Failure as RAID *2* (since Disk Controller can determine which Disk Failed)
- Higher Storage Utilization

Disadvantages:
- Poor *Random* read Performance
- Poor *Random* write and read-modify-write Performance (bottleneck: Parity Disk updating)



RAID *4*: Block-Interleaved Parity

Structure: ➢ A set of Data Sectors (Parity Group) striped across Data Disks

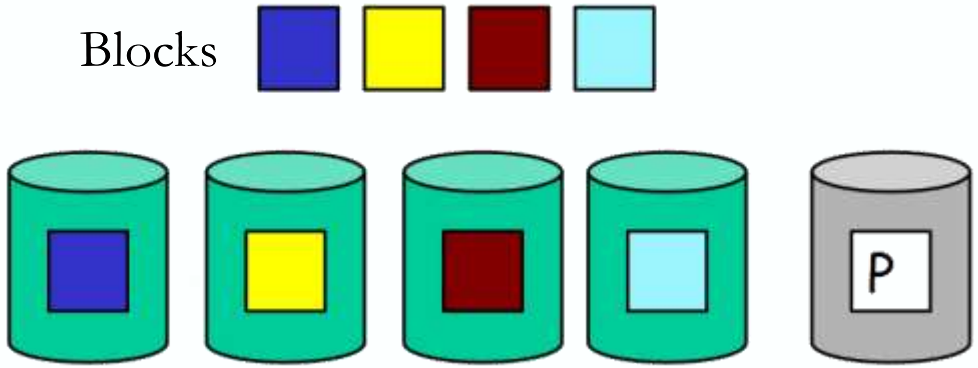
Advantages:

➢ Same Reliability as RAID *3*

➢ Good *Random* read Performance

Disadvantages:

➢ Poor *Random* write and read-modify-write Performance (bottleneck: Parity Disk updating)



RAID *4* Performance

- One Parity Disk (XOR of Data Sectors)
  - Write Data Disk + Parity Disk
  - To update Parity, don't have to read all Disk Sectors
  - Parity = oldParity XOR (changed bits) = oldParity XOR (newData XOR oldData)
- Number of groups: G = N/(D+1(=number of Parity Check Disks))

Performance

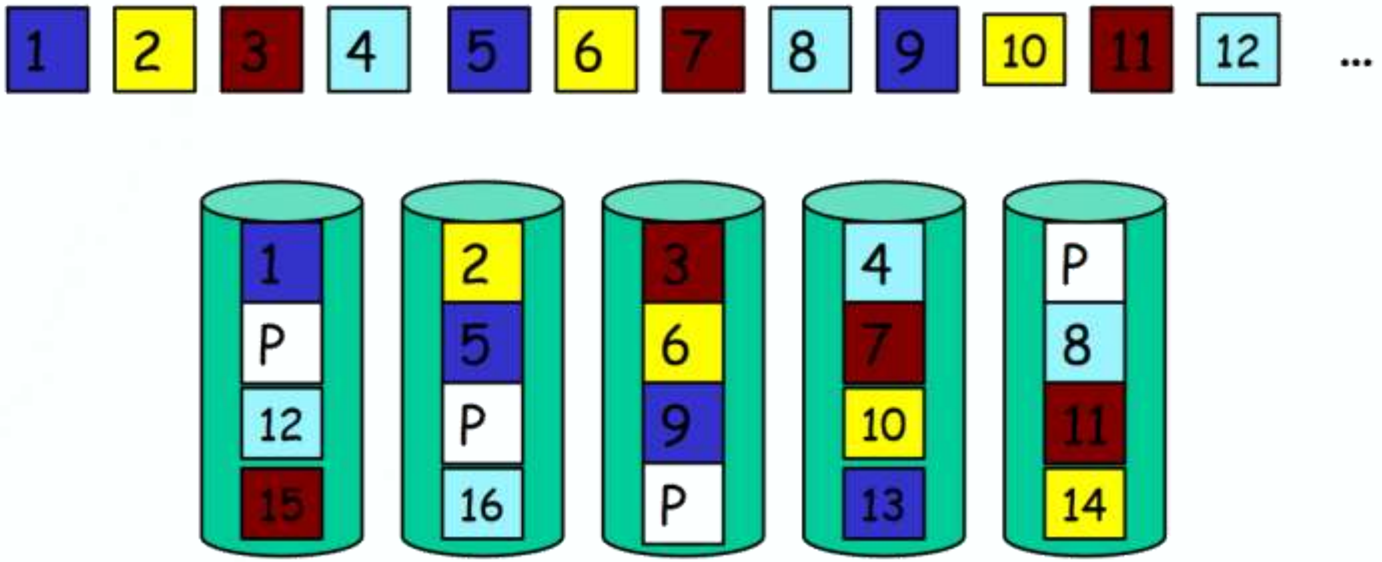
- Large read: (N-G)/S
- Large write: (N-G)/S
- Large R-M-W: (N-G)/S
- Small read: N-G
- Small write: ½*G (for each Block, need a read and a write to Parity Disk)
  - RAID: X sectors: X/((X/1) + (X/1)) = ½
- Small R-M-W: 1*G
  - RAID: X sectors: 2X/((X/1) + (X/1)) = 1

RAID *5*: Block-Interleaved Distributed Parity

Structure: ➢ Parity Sectors distributed across all Disks

Advantages:

- ➢ Same Reliability as RAID *3* & *4* (can tolerate 1 Disk Failure)
- ➢ Good *Small* write and read-modify-write Performance



RAID *5* Performance

- Same as RAID *4* except no single Parity Disk
  - Good small write and read-modify-write Performance

Performance

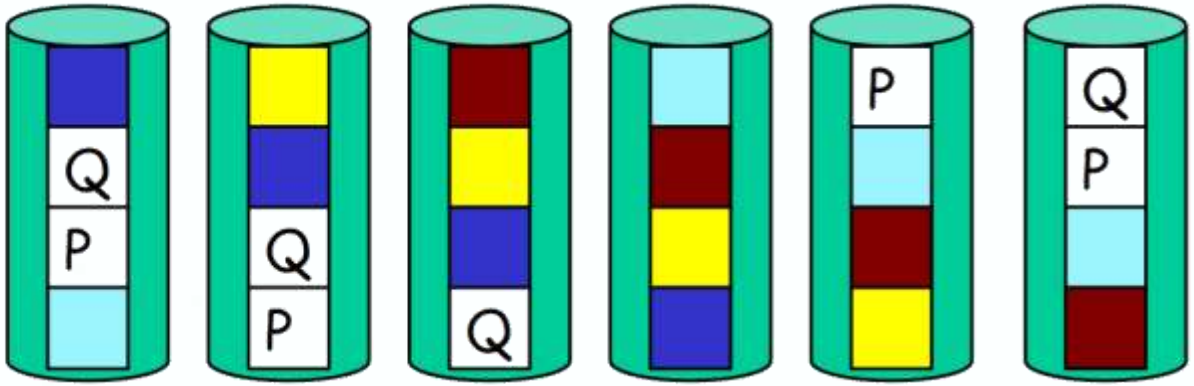
- Large read: (N-G)/S
- Large write: (N-G)/S
- Large R-M-W: (N-G)/S
- Small read: N
- Small write: N/4
  - One disk: X sectors * t
  - Raid 5: (X (read original) + X (read parity) + X (write original) + X (write parity)) / N * t
  - Raid5 can do 4X over all N Disks
- Small R-M-W: N/2
  - Same as small write, except read-original is not wasted

RAID *6*: P+Q Redundancy

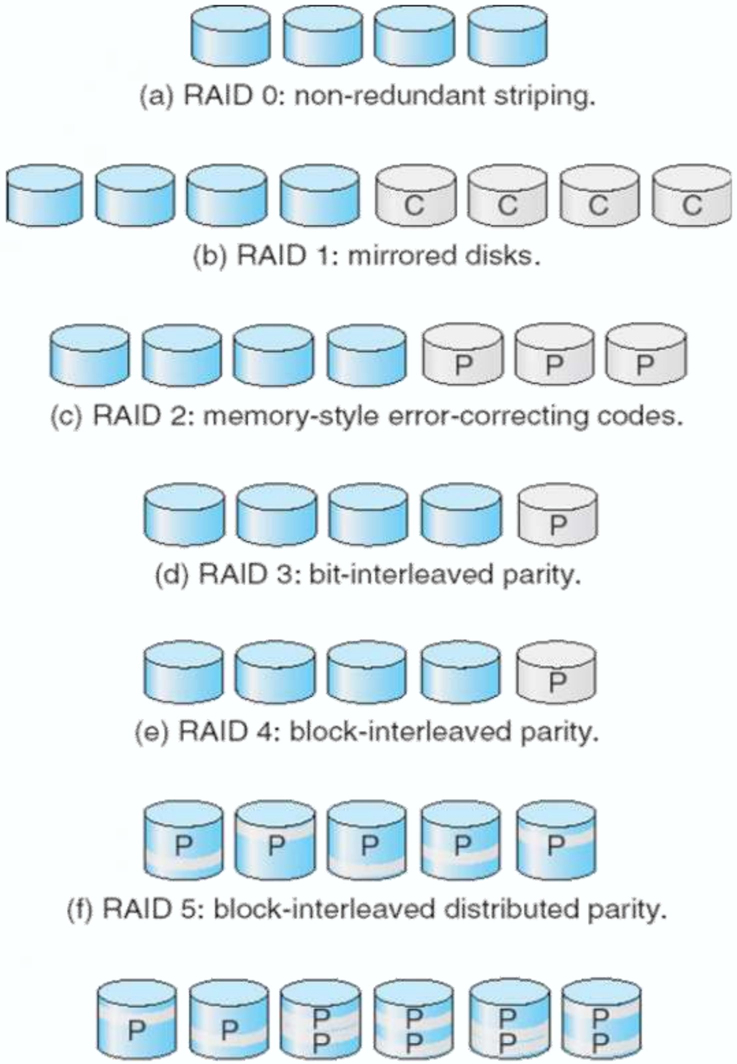
Structure: ➢ Same as RAID *5* except using two Parity Sectors per Parity Group

Advantages:

➢ Can tolerate 2 Disk Failures



RAID Levels



(a) RAID 0: non-redundant striping.

(b) RAID 1: mirrored disks.

(c) RAID 2: memory-style error-correcting codes.

(d) RAID 3: bit-interleaved parity.

(e) RAID 4: block-interleaved parity.

(f) RAID 5: block-interleaved distributed parity.

(g) RAID 6: P + Q redundancy.

CS-446/646

Time for Questions !