Lecture 11
I/O and Storage

Prof. Yinqian Zhang

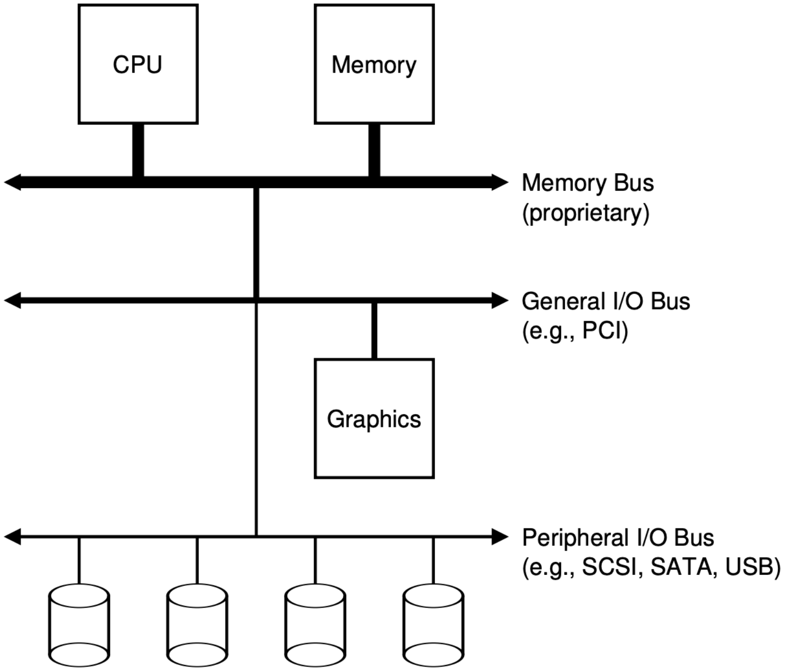
Fall 2025

I/O Management

- So far, we have learned how to manage CPU and memory
- Challenges of I/O management
  - Diverse devices: each device is slightly different
    - How can we standardize the interfaces to these devices?
  - Unreliable device: media failures and transmission errors
    - How can we make them reliable?
  - Unpredictable and slow devices
    - How can we manage them if we do not know what they will do or how they will perform?

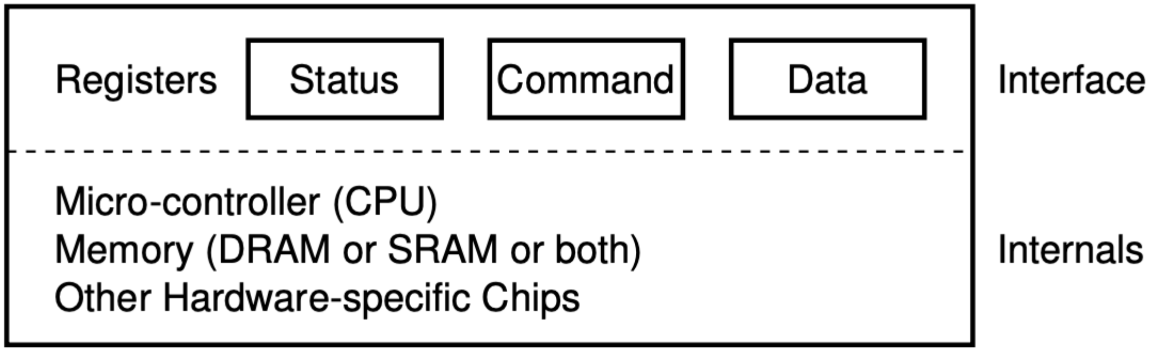
A Classic View of Computer System

- A **single CPU** attached to the **main memory** of the system via memory bus or interconnect.
- Some devices (**graphics** and some other **higher-performance I/O devices**) are connected to the system via a general I/O bus (e.g., PCI)
- Finally, a peripheral bus, such as SCSI, SATA, or USB, connects slow devices to the system, including **disks**, **mice**, and **keyboards**



A Canonical View of Devices

- Interface
  - The hardware interface a device present to the rest of the system
  - **Status** registers: check the current status of the device
  - **Command** register: tell the device to perform a certain task
  - **Data** register: pass data to the device or get data from the device.
- Internal structures
  - Implementation of the abstract of the device



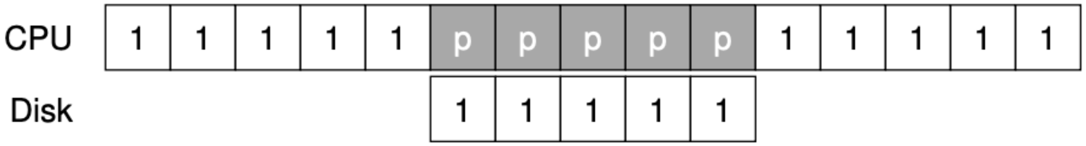
Basic I/O: Polling

- To write a byte of data to device
  - Step 1: OS waits until the device is ready to receive a command by repeatedly reading the status register;
  - Step 2: OS sends some data down to the data register;
  - Step 3: OS writes a command to the command register
  - Step 4: OS waits for the device to finish by again polling it in a loop, waiting to see if it is finished

```
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
```

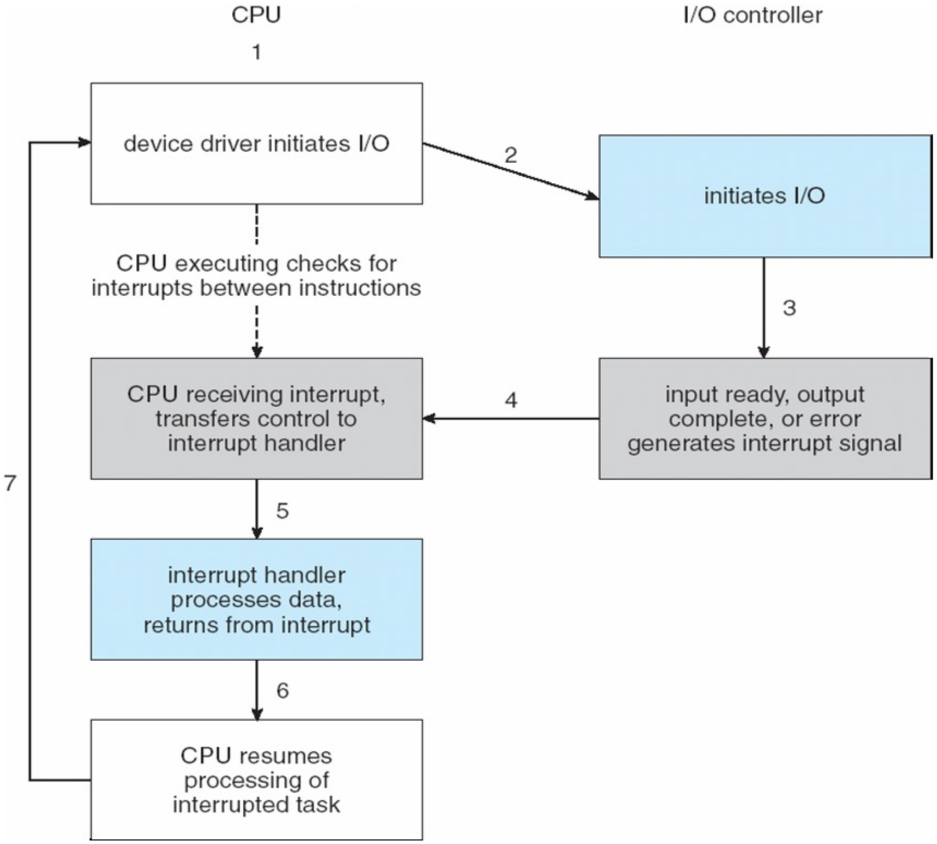
Issues of Polling

- Polling: frequent checking the status of I/O devices
- Polling is inefficient and inconvenient
  - Polling wastes CPU time waiting for slow devices to complete its activity
  - If CPU switches to other tasks, data may be overwritten
    - e.g., keyboard data overflow the buffer



Efficient I/O: Interrupts

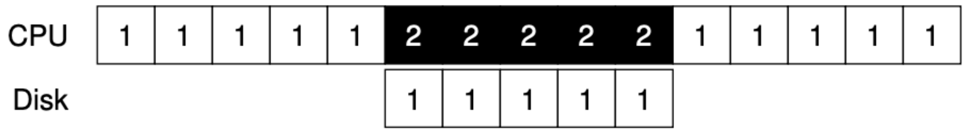
- Instead of polling the device repeatedly, the OS can issue a request, put the calling process to sleep, and context switch to another task.
- When the device is finally finished with the operation, it will raise a hardware interrupt, causing the CPU to jump into the OS at a predetermined **interrupt handler**



CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
7
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task

Polling or Interrupt?

- Polling works better for fast devices
  - Data fetched with first poll
- Interrupt works better for slow devices
  - Context switch is expensive
- Hybird approach if speed of the device is not known or unstable
  - Polls for a while
  - Then use interrupts



Hardware Support for Interrupts

- **Interrupt-request line,** a CPU wire, triggered by I/O device
  - Checked by processor after each instruction
  - Save CPU state and jumps to the interrupt handler

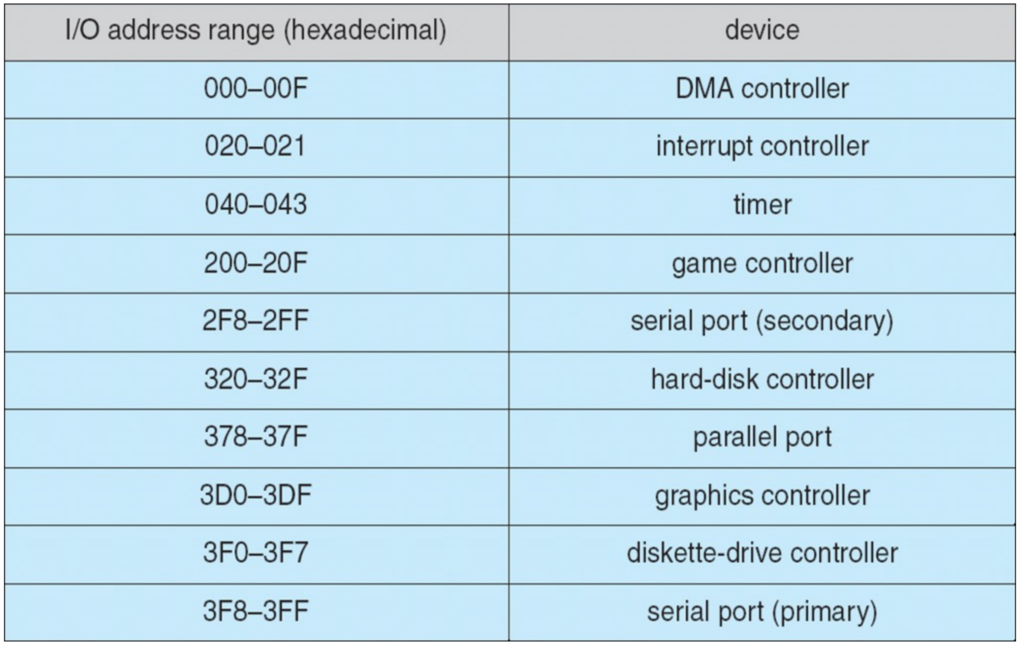
- **Interrupt-controller hardware**
  - Defer interrupt handling during critical processing
  - Dispatch to proper interrupt handler
  - Support multi-level interrupts, high- and low-priority interrupts

Software Support for Interrupts

- **Interrupt handler** receives interrupts
  - **Maskable** to ignore or delay some interrupts
  - Some are **nonmaskable**, e.g., unrecoverable memory errors.
- A table of **interrupt vectors** to specify interrupt-handling routine
  - Dispatch interrupt to correct handler
  - **Interrupt chaining** if more than one device at the same interrupt number
    - Interrupt handlers on the corresponding chain are called one by one
  - The size of the interrupt table (i.e., number of interrupt vectors) and length of interrupt chains are results of system design trade-off.
  - Priority of interrupts: high-priority interrupts can preempt low-priority interrupts

Programmed I/O

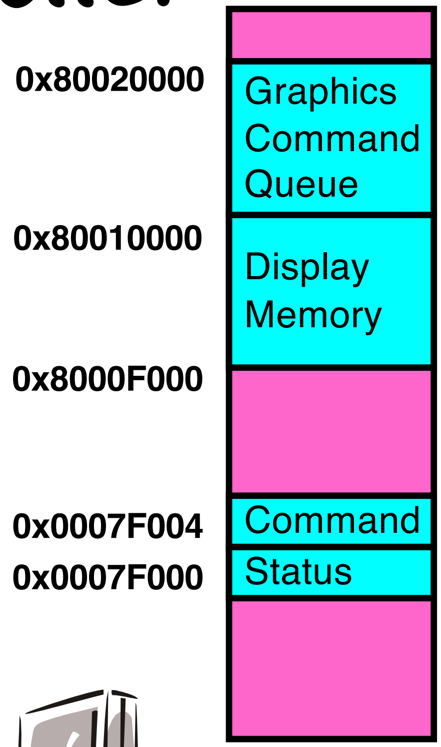
- Explicit I/O instructions
  - in/out instructions on x86: out 0x21,AL
  - I/O instructions are privileged instructions
- Memory-mapped I/O
  - Registers/memory appear in physical address space
  - I/O accomplished with load and store instructions
  - I/O protection with address translation

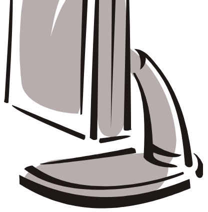


I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Memory-Mapped Display Controller

- Memory-Mapped I/O
  - Hardware maps control registers and display memory into physical address space
    - Addresses set by HW jumpers or at boot time
  - Simply writing to display memory (also called the “frame buffer”) changes image on screen
    - Addr: 0x8000F000 — 0x8000FFFF
  - Writing graphics description to cmd queue
    - Say enter a set of triangles describing some scene
    - Addr: 0x80010000 — 0x8001FFFF
  - Writing to the command register may cause on-board graphics hardware to do something
    - Say render the above scene
    - Addr: 0x0007F004



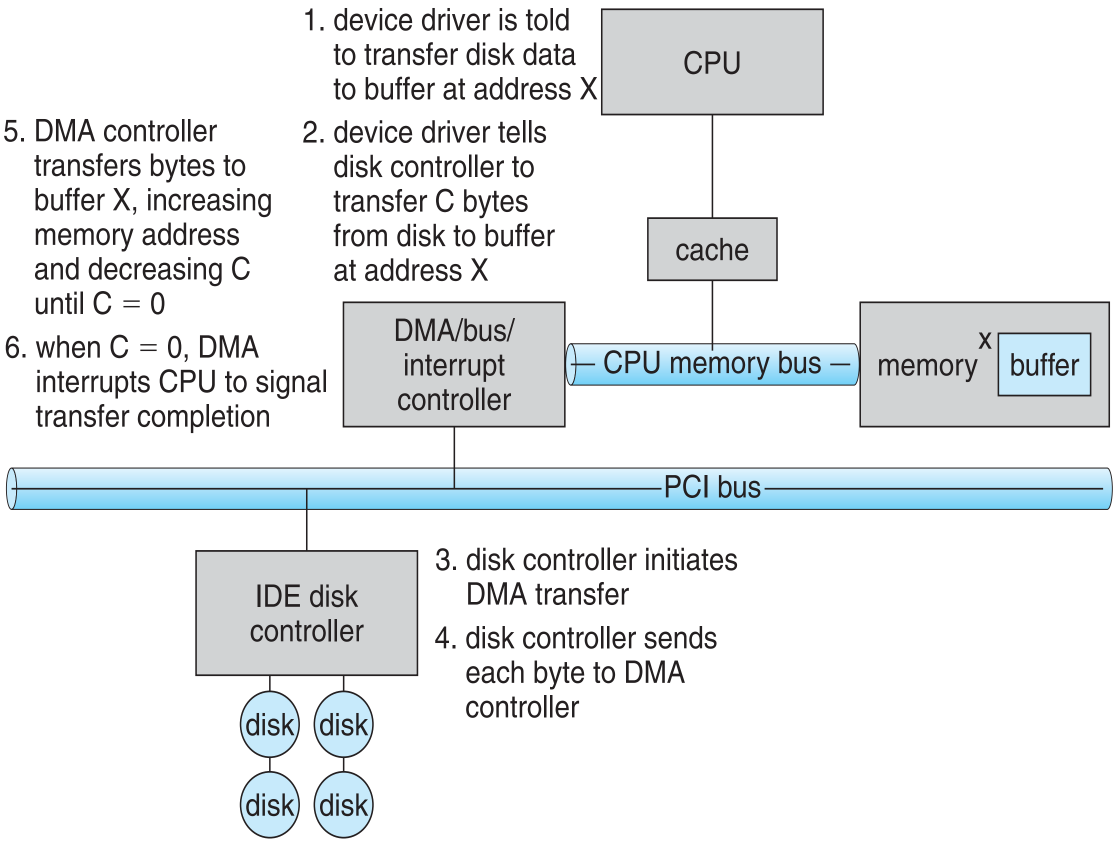


Physical Address Space

More Efficient Data Movement: DMA

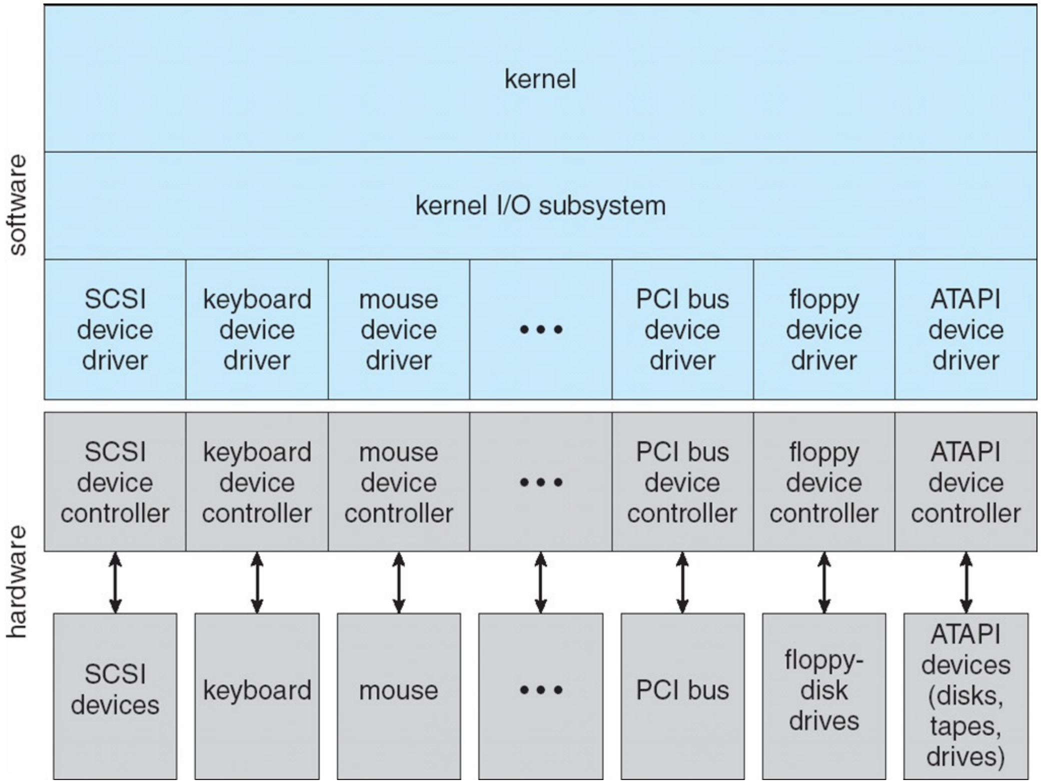
- DMA is used to avoid **programmed I/O** for large data movement
  - Programmed I/O (PIO): when CPU is involved in data movement
  - PIO consumes CPU time
  - bypasses CPU to transfer data directly between I/O device and memory
- OS writes DMA command block into memory
  - Source and destination addresses
  - Read or write mode
  - Count of bytes
  - Writes location to DMA controller
  - Bus mastering of DMA controller – grabs bus from CPU
    - **Cycle stealing** from CPU but still much more efficient
  - When done, interrupts to signal completion

More Efficient Data Movement: DMA



1. device driver is told to transfer disk data to buffer at address X
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. when C = 0, DMA interrupts CPU to signal transfer completion

Kernel I/O Structure



software

kernel

kernel I/O subsystem

SCSI device driver | keyboard device driver | mouse device driver | • • • | PCI bus device driver | floppy device driver | ATAPI device driver

hardware

SCSI device controller | keyboard device controller | mouse device controller | • • • | PCI bus device controller | floppy device controller | ATAPI device controller

SCSI devices | keyboard | mouse | • • • | PCI bus | floppy-disk drives | ATAPI devices (disks, tapes, drives)

Device Drivers

- Device Driver: Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with the ioctl() system call
- Device Drivers typically divided into two pieces:
  - Top half: accessed in call path from system calls
    - implements a set of standard, cross-device calls like open(), close(), read(), write(), ioctl(), strategy()
    - This is the kernel's interface to the device driver
    - Top half will *start* I/O to device, may put thread to sleep until finished
  - Bottom half: run as interrupt routine
    - Gets input or transfers next block of output
    - May wake sleeping threads if I/O now complete

Life Cycle of An I/O Request

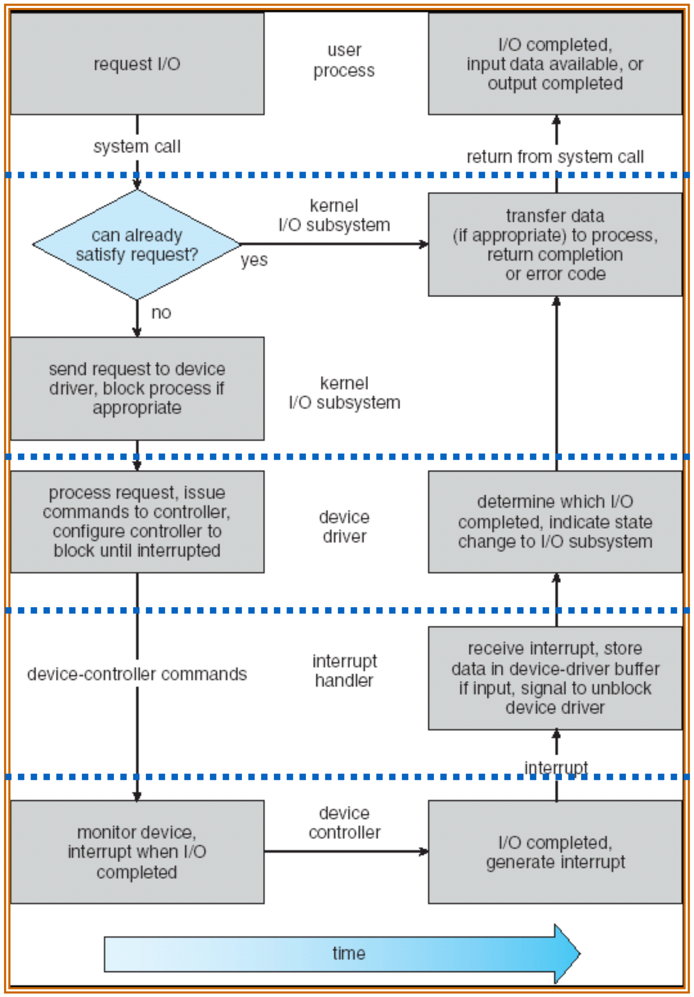
User Program

Kernel I/O Subsystem

Device Driver Top Half

Device Driver Bottom Half

Device Hardware

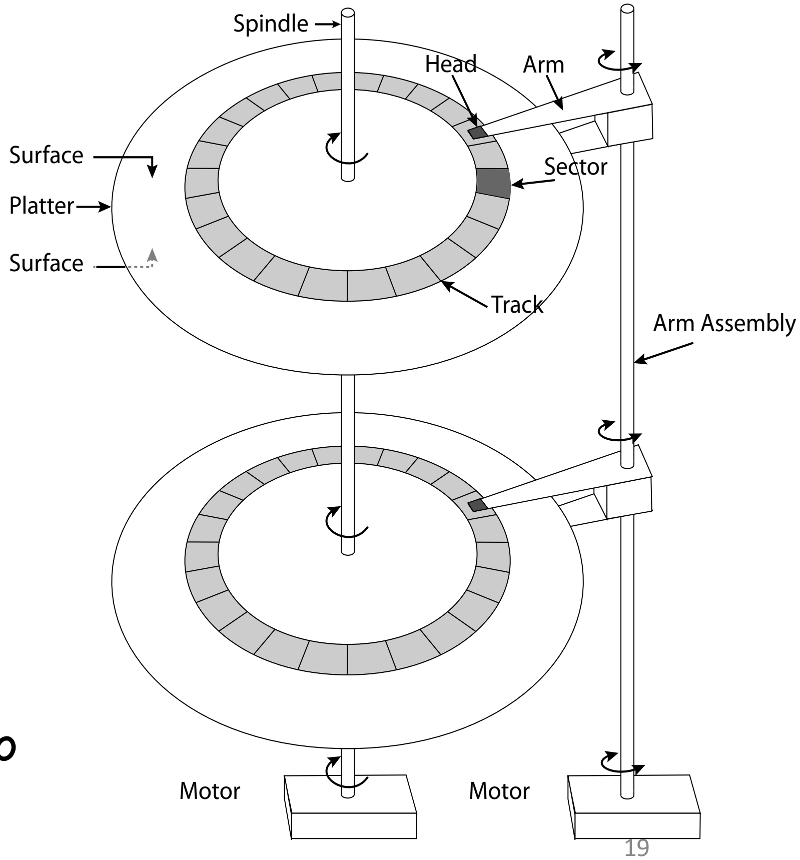

request I/O
user process
I/O completed, input data available, or output completed
system call
return from system call
can already satisfy request?
yes
no
kernel I/O subsystem
transfer data (if appropriate) to process, return completion or error code
send request to device driver, block process if appropriate
kernel I/O subsystem
process request, issue commands to controller, configure controller to block until interrupted
device driver
determine which I/O completed, indicate state change to I/O subsystem
device-controller commands
interrupt handler
receive interrupt, store data in device-driver buffer if input, signal to unblock device driver
interrupt
monitor device, interrupt when I/O completed
device controller
I/O completed, generate interrupt
time

Storage Devices

- Magnetic disks
  - Storage that rarely becomes corrupted
  - Large capacity at low cost
  - Block level random access
  - Poor performance for random access
  - Better performance for sequential access
- Solid State Disk
  - Storage that rarely becomes corrupted
  - Capacity at intermediate cost (5-20x disk)
  - Block level random access
  - Good performance for reads; worse for random writes

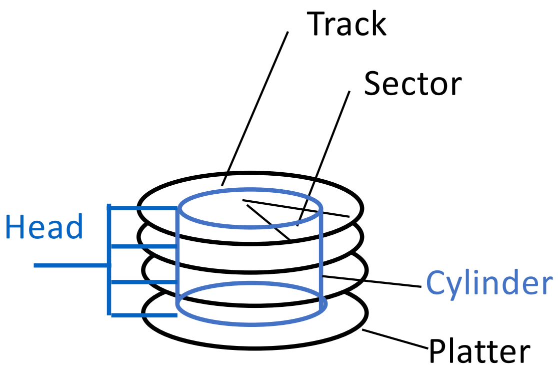
The Amazing Magnetic Disk

- Unit of Transfer: Sector
  - Ring of sectors form a track
  - Stack of tracks form a cylinder
  - Heads position on cylinders
- Disk Tracks ~ 1μm (micron) wide
  - Wavelength of light is ~ 0.5μm
  - Resolution of human eye: 50μm
  - 100K tracks on a typical 2.5″ disk
- Separated by unused guard regions
  - Reduces likelihood neighboring tracks are corrupted during writes (still a small non-zero chance)

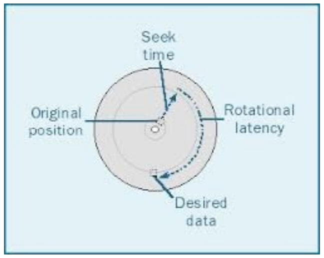


Magnetic Disks

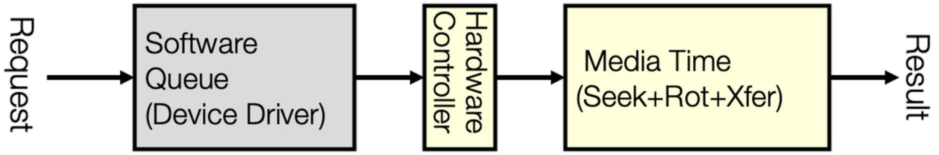
- Cylinders: all the tracks under the head at a given point on all surface
- Read/write data is a three-stage process:
  - Seek time: position the head/arm over the proper track
  - Rotational latency: wait for desired sector to rotate under r/w head
  - Transfer time: transfer a block of bits (sector) under r/w head



Seek time = 4-8ms
One rotation = 1-2ms
(3600-7200 RPM)



Disk Latency = Queuing Time + Controller time +
Seek Time + Rotation Time + Transfer Time



Request

Software
Queue
(Device Driver)

Hardware
Controller

Media Time
(Seek+Rot+Xfer)

Result

Disk Performance Example

- Assumptions:
  - Ignoring queuing and controller times for now
  - Avg seek time of 5ms,
  - 7200RPM ⇒ Time for rotation: 60000 (ms/minute) / 7200(rev/min) ≅ 8ms
  - Transfer rate 4MByte/s, sector size 1 Kbyte ⇒ 1024 bytes/4×10^6 (bytes/s) = 256 × 10^{-6} sec ≅ .26 ms
- Read sector from random place on disk:
  - Seek (5ms) + Rot. Delay (4ms) + Transfer (0.26ms)
  - *Approx* 10ms to fetch/put data: 100 KByte/sec
- Read sector from random place in the same cylinder:
  - Rot. Delay (4ms) + Transfer (0.26ms)
  - *Approx* 5ms to fetch/put data: 200 KByte/sec
- Read next sector on same track:
  - Transfer (0.26ms): 4 MByte/sec

Key to using disk effectively (especially for file systems) is to *minimize seek and rotational delays*

Typical Numbers for Magnetic Disk

Parameter	Info / Range
Space/Density	Space: 8TB (Seagate), 10TB (Hitachi) in 3½ inch form factor! Areal Density: ≥ 1Terabit/square inch! (SMR, Helium, …)
Average seek time	Typically 5-10 milliseconds. Depending on reference locality, actual cost may be 25-33% of this number.
Average rotational latency	Most laptop/desktop disks rotate at 3600-7200 RPM (16-8 ms/rotation). Server disks up to 15,000 RPM. Average latency is halfway around disk so 8-4 milliseconds
Controller time	Depends on controller hardware
Transfer rate	Typically 50 to 100 MB/s.
Cost	Used to drop by a factor of two every 1.5 years (or even faster); now slowing down

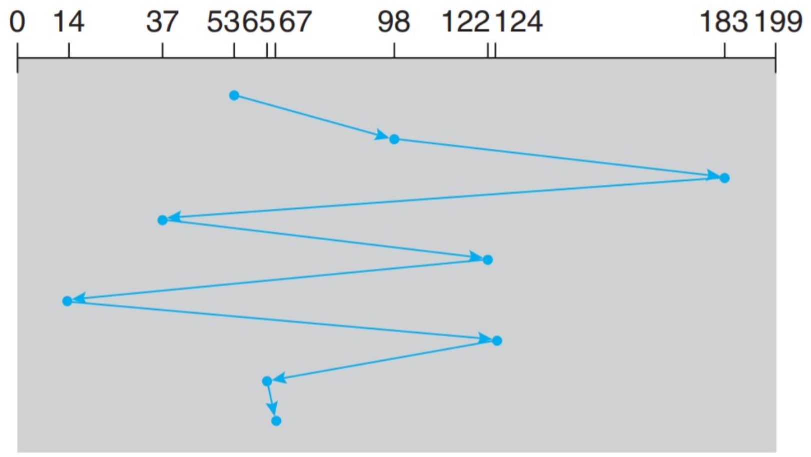
Disk Scheduling

- There are many sources of disk I/O request
  - OS
  - System processes
  - User processes
- OS should think how to use hardware efficiently?
  - Disk bandwidth
  - Access time
- Given a sequence of access cylinders in the HDD
  - 98, 183, 37, 122, 14, 124, 65, 67
  - Head point: 53
  - Pages: 0 ~ 199
- Minimize seek time
  - Seek time ≈ seek distance
- How to minimize the total head movement distance?
  - Minimize the total number of cylinders.

Disk Scheduling: FIFO

- FIFO Order
  - Fair among requesters, but order of arrival may be to random spots on the disk ⇒ Very long seeks
- The head movement distance = ?

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53

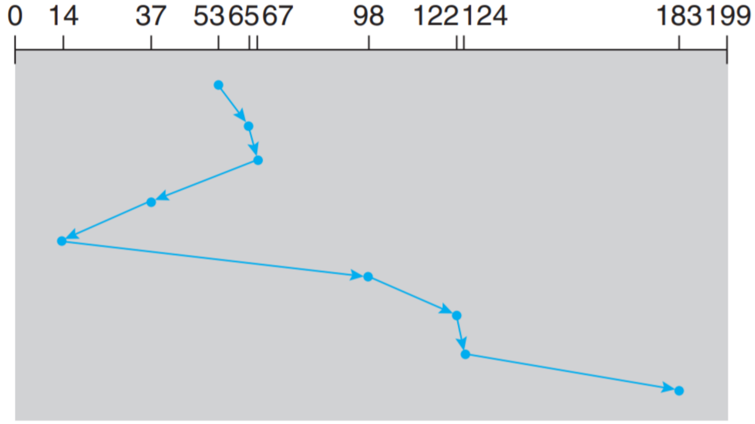


Disk Scheduling: SSTF

- Shortest Seek Time First order
    - Shortest Seek Time First selects the request with the minimum seek time from the current head position
    - SSTF scheduling is a form of **SJF** scheduling; may cause **starvation** of some requests
- The head movement distance = ?

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



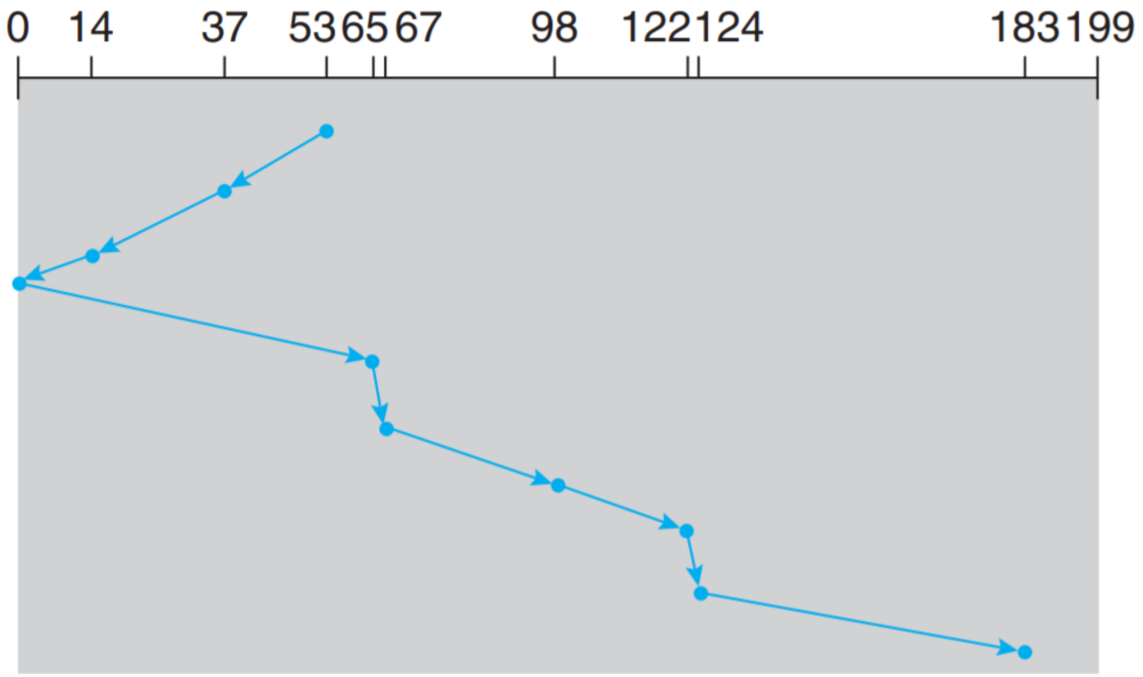
Disk Scheduling: SCAN

- SCAN order
  - SCAN algorithm a.k.a., elevator algorithm
  - The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
  - But note that if requests are uniformly dense, largest density at other end of disk and those wait the longest
- The head movement distance = ?

Disk Scheduling: SCAN

queue = 98, 183, 37, 122, 14, 124, 65, 67

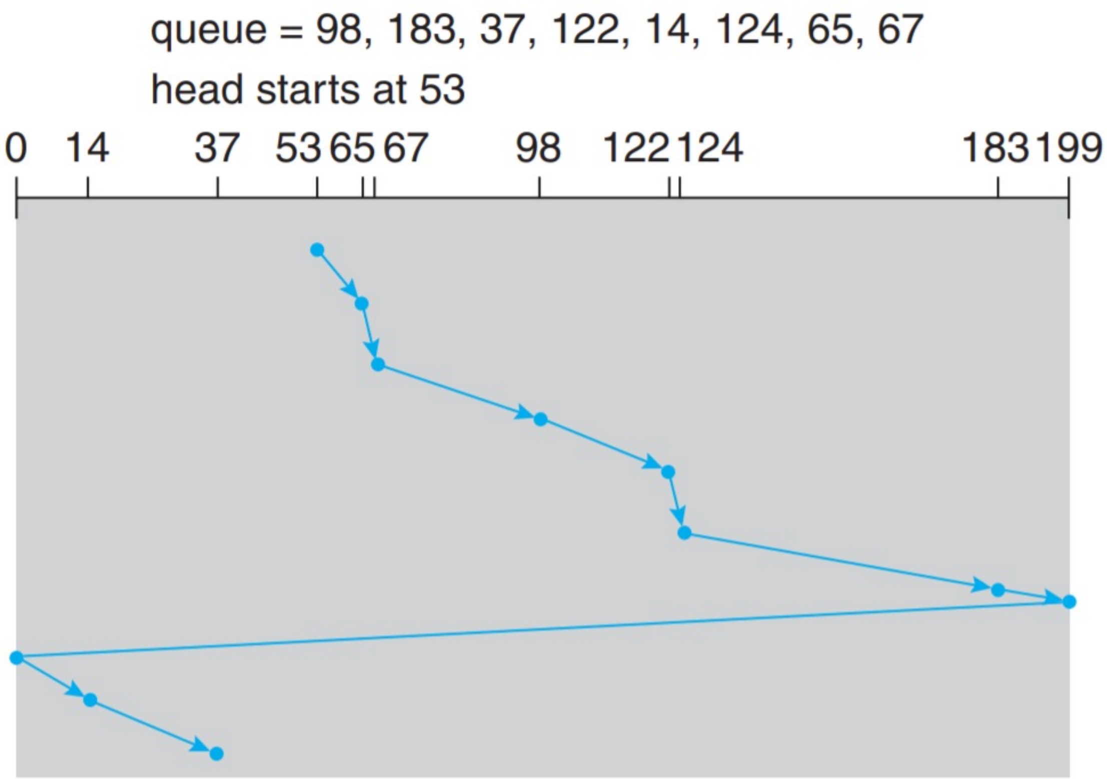
head starts at 53



Disk Scheduling: C-SCAN

- C-SCAN order
  - Provides a more uniform wait time than SCAN
  - The head moves from one end of the disk to the other, servicing requests as it goes
    - When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip
  - Treats the cylinders as a circular list that wraps around from the last cylinder to the first one
- The head movement distance =?

Disk Scheduling: C-SCAN



queue = 98, 183, 37, 122, 14, 124, 65, 67

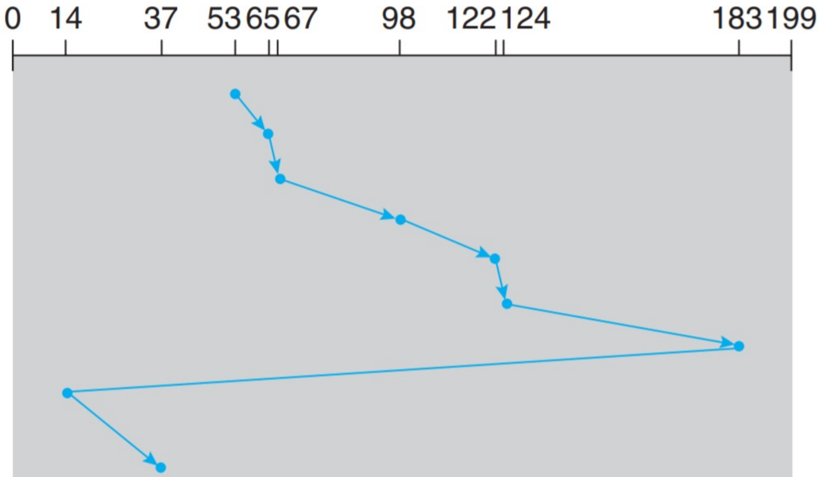
head starts at 53

Disk Scheduling: LOOK, C-LOOK

- Look and C-LOOK order
  - LOOK a variant of SCAN, C-LOOK a variant of C-SCAN
  - Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk
- C-LOOK: the head movement distance = ?

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



Selecting a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk
  - Less starvation
- Either SSTF or LOOK is a reasonable choice for the default algorithm
- Performance depends on the number and types of requests
- The disk-scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm if necessary

Thank you!