CMPT 300

Introduction to Operating Systems

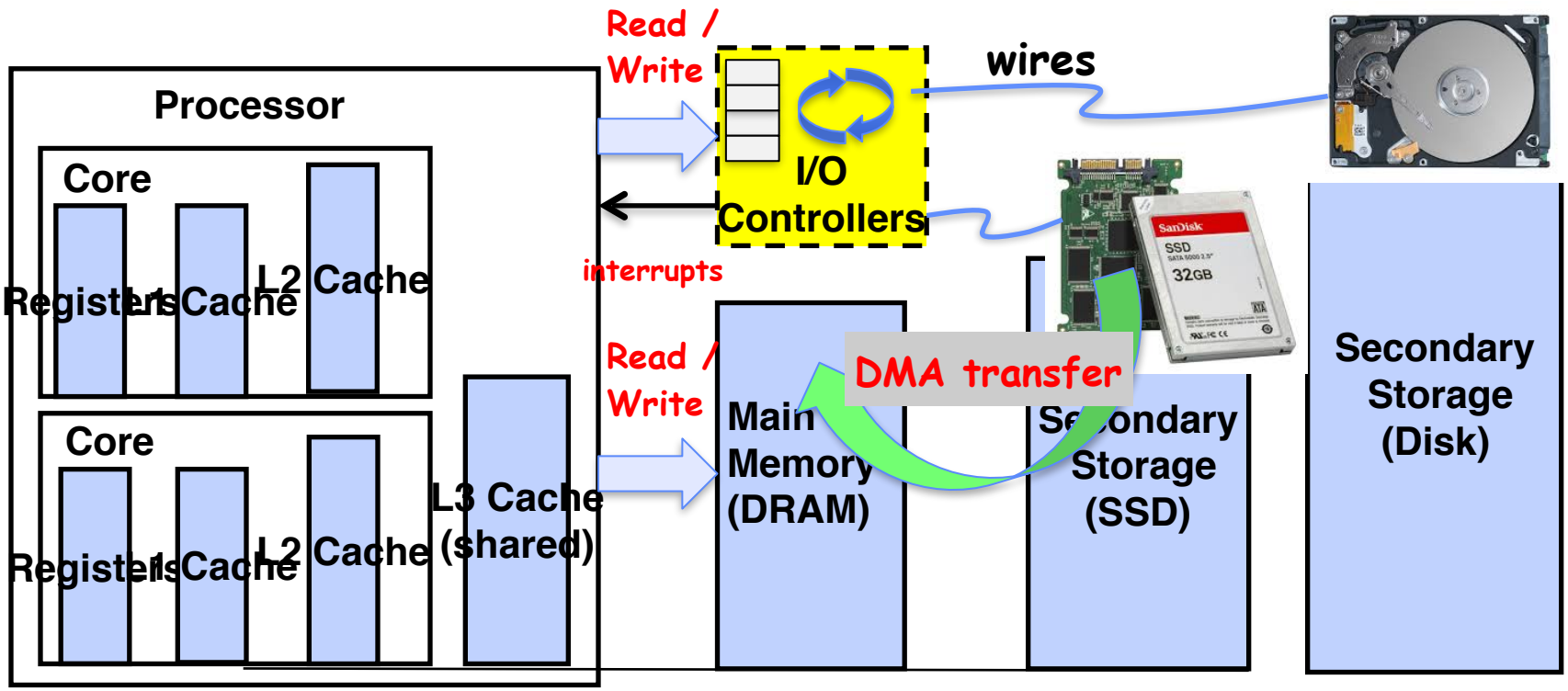
I/O

Acknowledgement: some slides are taken from Anthony D. Joseph's course material at UC Berkeley and Dr. Janice Reagan's course material at SFU

Outline

- **Overview**
- Principles of I/O hardware
- Principles of I/O software
- Disks

In a picture



- I/O devices you recognize are supported by I/O Controllers
- Processors accesses them by reading and writing IO registers as if they were memory
  - Write commands and arguments, read status and results

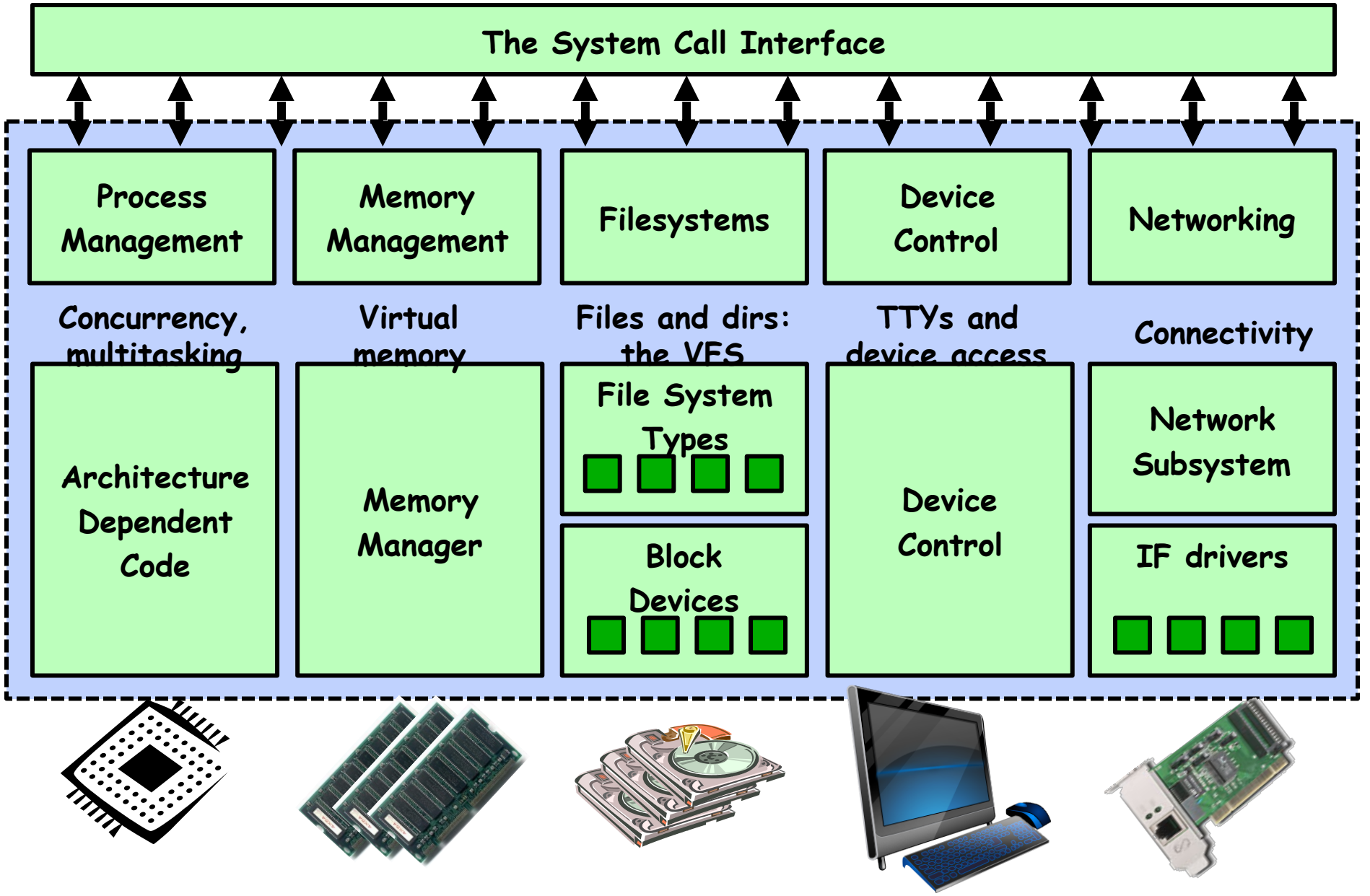
The Requirements of I/O

- So far in this course:
  - We have learned how to manage CPU, memory
- What about I/O?
  - Without I/O, computers are useless (disembodied brains?)
  - But… thousands of devices, each slightly different
    - » How can we standardize the interfaces to these devices?
  - Devices unreliable: media failures and transmission errors
    - » How can we make them reliable???
  - Devices unpredictable and/or slow
    - » How can we manage them if we don't know what they will do or how they will perform?

Operational Parameters for I/O

- Data granularity: Byte vs. Block
  - Some devices provide single byte at a time (e.g., keyboard)
  - Others provide whole blocks (e.g., disks, networks, etc.)
- Access pattern: Sequential vs. Random
  - Some devices must be accessed sequentially (e.g., tape)
  - Others can be accessed "randomly" (e.g., disk, cd, etc.)
    - » Fixed overhead to start sequential transfer (more later)
- Transfer Notification: Polling vs. Interrupts
  - Some devices require continual monitoring
  - Others generate interrupts when they need service
- Transfer Mechanism: Programmed IO and DMA

Kernel Device Structure



The Goal of the I/O Subsystem

- Provide Uniform Interfaces, Despite Wide Range of Different Devices
  - This code works on many different devices:

```
   FILE fd = fopen("/dev/something","rw");
for (int i = 0; i < 10; i++) {
   fprintf(fd,"Count %d\n",i);
}
close(fd);
```

  - Why? Because code that controls devices ("device driver") implements standard interface.
- We will try to get a flavor for what is involved in actually controlling devices in rest of lecture
  - Can only scratch surface!

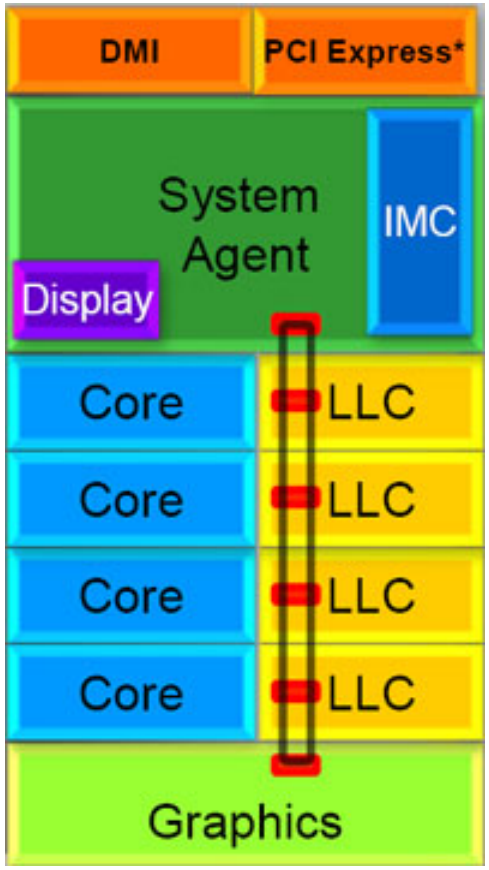
Want Standard Interfaces to Devices

- **Block Devices:** e.g. disk drives, tape drives, DVD-ROM
  - Access blocks of data
  - Commands include `open()`, `read()`, `write()`, `seek()`
  - Raw I/O or file-system access
  - Memory-mapped file access possible
- **Character Devices:** e.g. keyboards, mice, serial ports, some USB devices
  - Single characters at a time
  - Commands include `get()`, `put()`
  - Libraries layered on top allow line editing
- **Network Devices:** e.g. Ethernet, Wireless, Bluetooth
  - Different enough from block/character to have own interface
  - Unix and Windows include **socket** interface
    - » Separates network protocol from network operation
    - » Includes `select()` functionality
  - Usage: pipes, FIFOs, streams, queues, mailboxes

How Does User Deal with Timing?

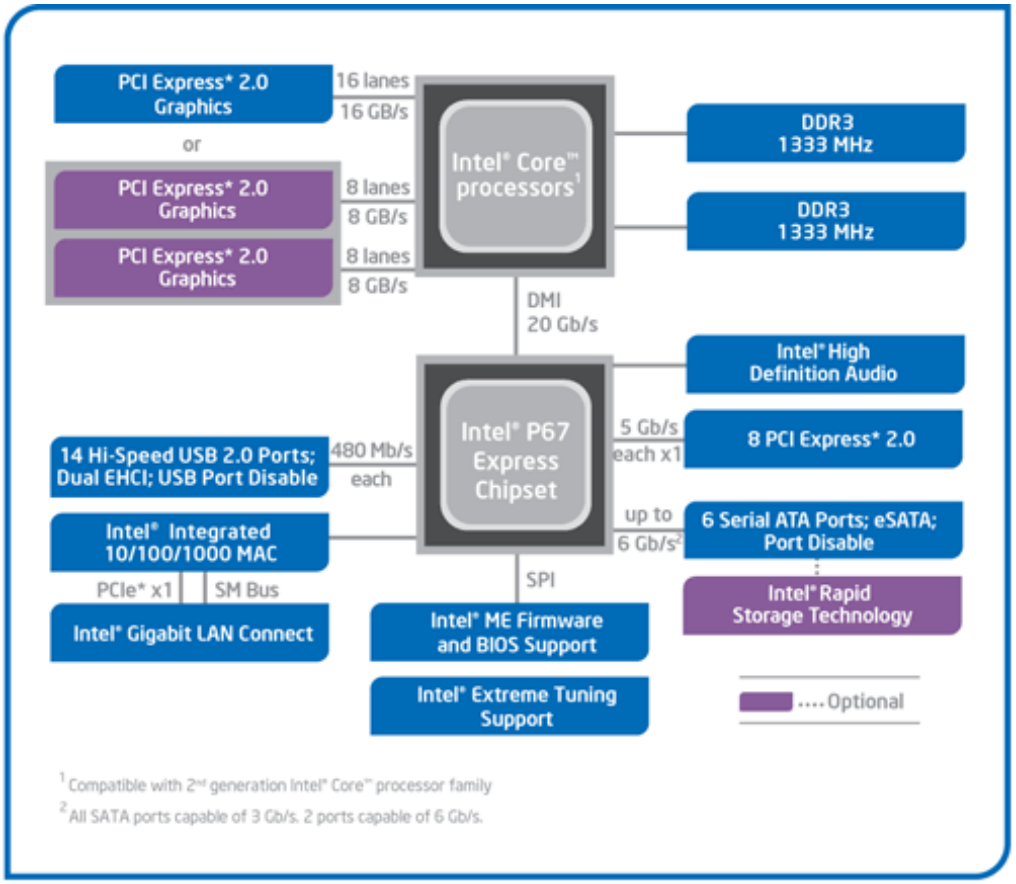
- **Blocking Interface:** "Wait"
  - When request data (e.g. `read()` system call), put process to sleep until data is ready
  - When write data (e.g. `write()` system call), put process to sleep until device is ready for data
- **Non-blocking Interface:** "Don't Wait"
  - Returns quickly from read or write request with count of bytes successfully transferred
  - Read may return nothing, write may write nothing
- **Asynchronous Interface:** "Tell Me Later"
  - When request data, take pointer to user's buffer, return immediately; later kernel fills buffer and notifies user
  - When send data, take pointer to user's buffer, return immediately; later kernel takes data and notifies user

Chip-scale features of Recent x86 (SandyBridge)



- Significant pieces:
  - Four OOO cores
    - » New Advanced Vector eXtensions (256-bit FP)
    - » AES instructions
    - » Instructions to help with Galois-Field mult
    - » 4 μ-ops/cycle
  - Integrated GPU
  - System Agent (Memory and Fast I/O)
  - Shared L3 cache divided in 4 banks
  - On-chip Ring bus network
    - » Both coherent and non-coherent transactions
    - » High-BW access to L3 Cache
- Integrated I/O
  - Integrated memory controller (IMC)
    - » Two independent channels of DDR3 DRAM
  - High-speed PCI-Express (for Graphics cards)
  - DMI Connection to SouthBridge (PCH)

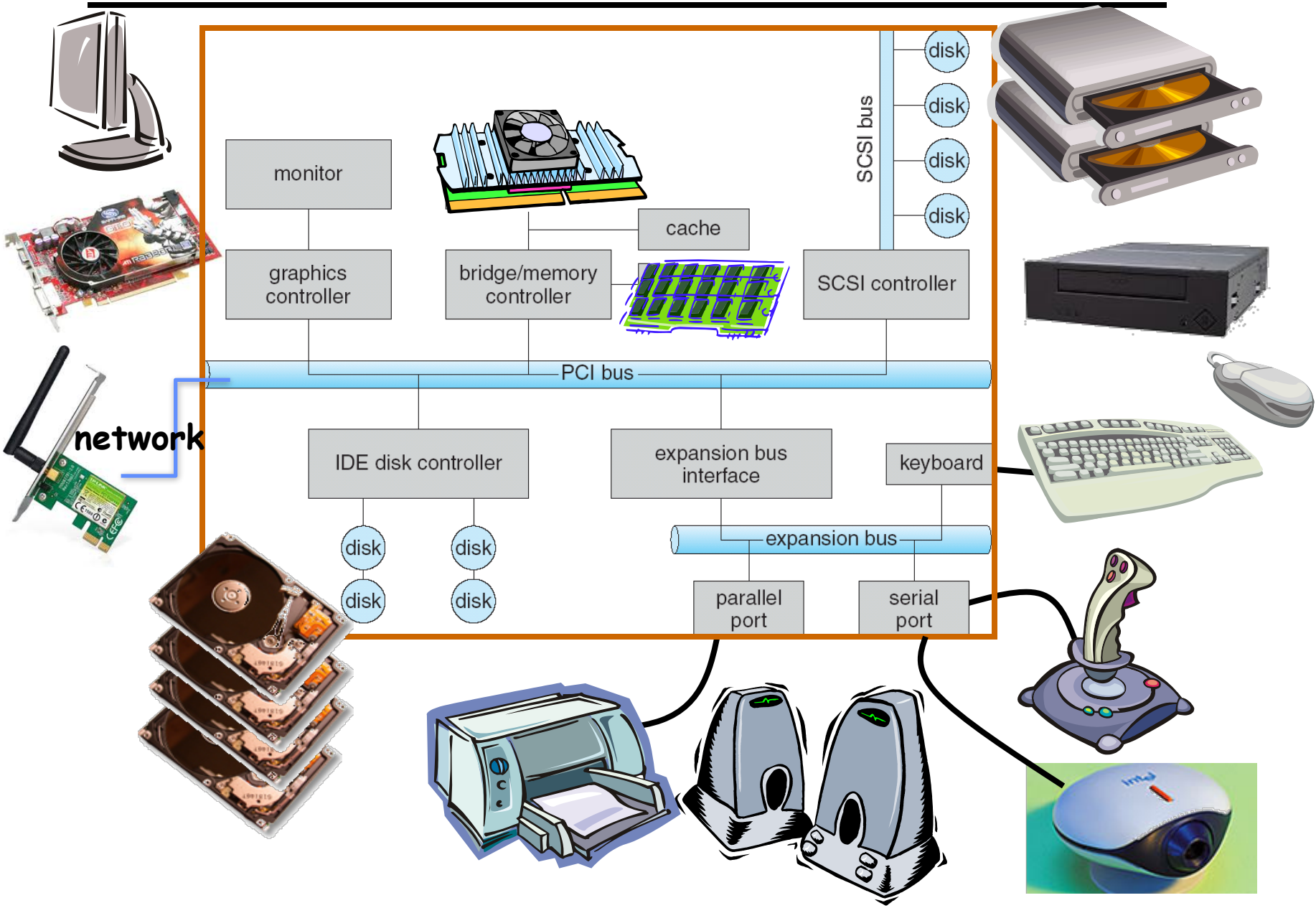
SandyBridge I/O: PCH



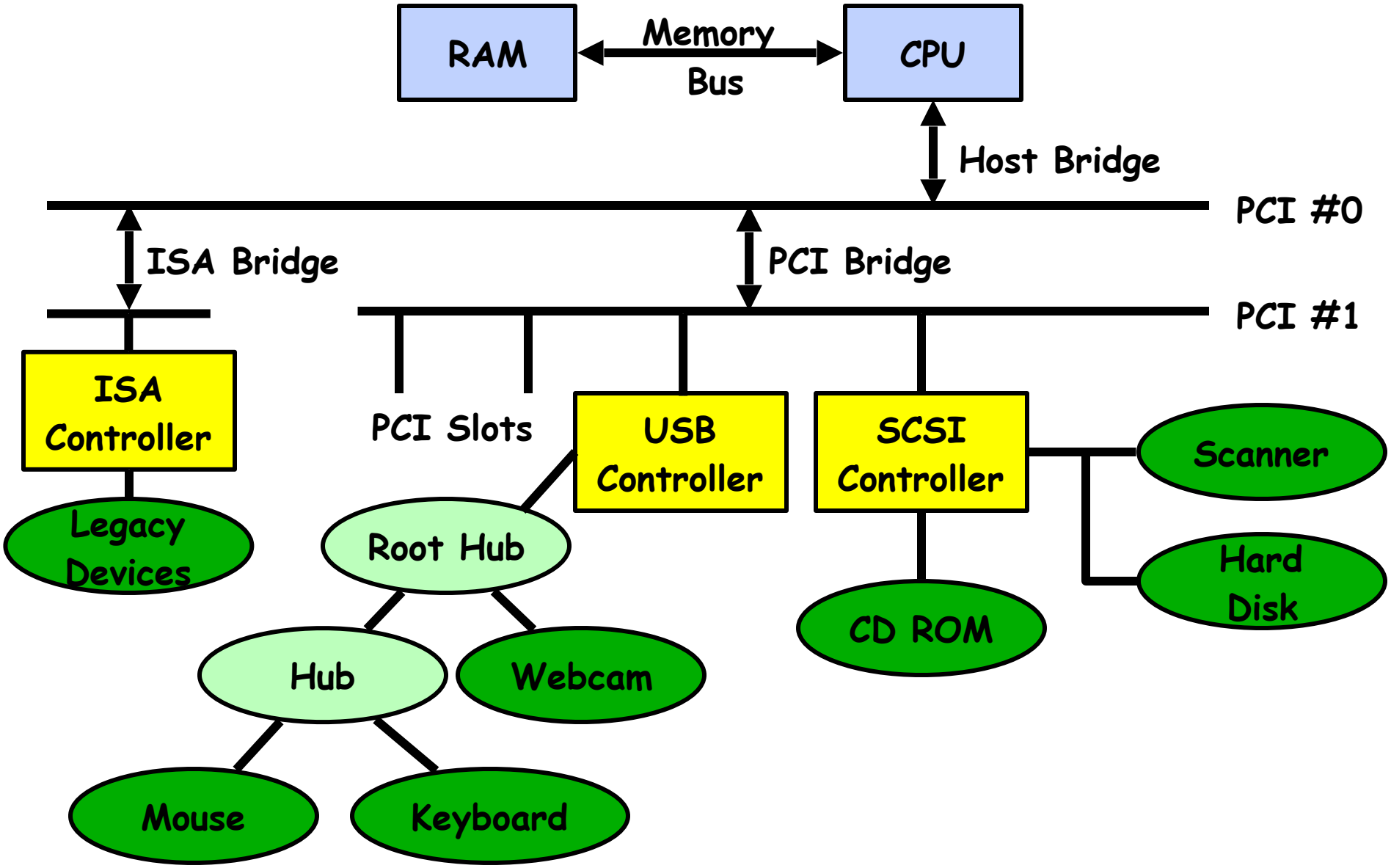
SandyBridge
System Configuration

- Platform Controller Hub
  - Used to be "SouthBridge," but no "NorthBridge" now
  - Connected to processor with proprietary bus
    - » Direct Media Interface
  - Code name "Cougar Point" for SandyBridge processors
- Types of I/O on PCH:
  - USB
  - Ethernet
  - Audio
  - BIOS support
  - More PCI Express (lower speed than on Processor)
  - Sata (for Disks)

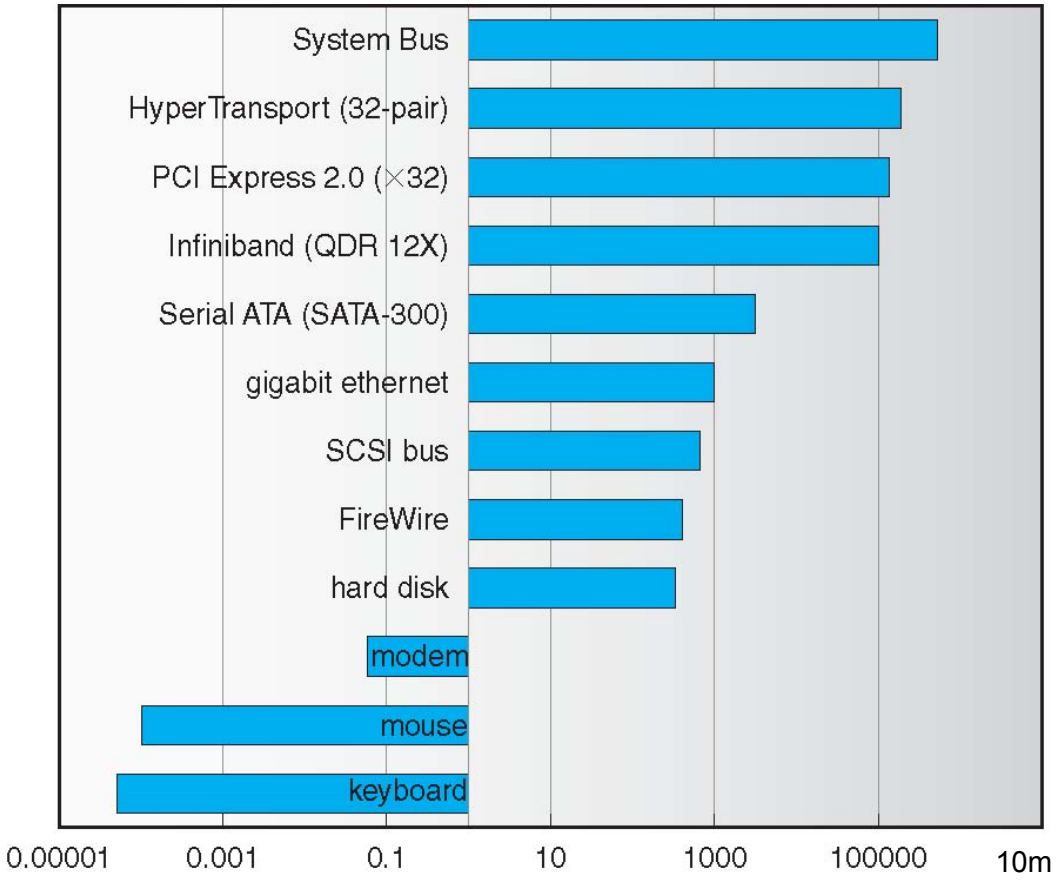
Modern I/O Systems



Example: PCI Architecture

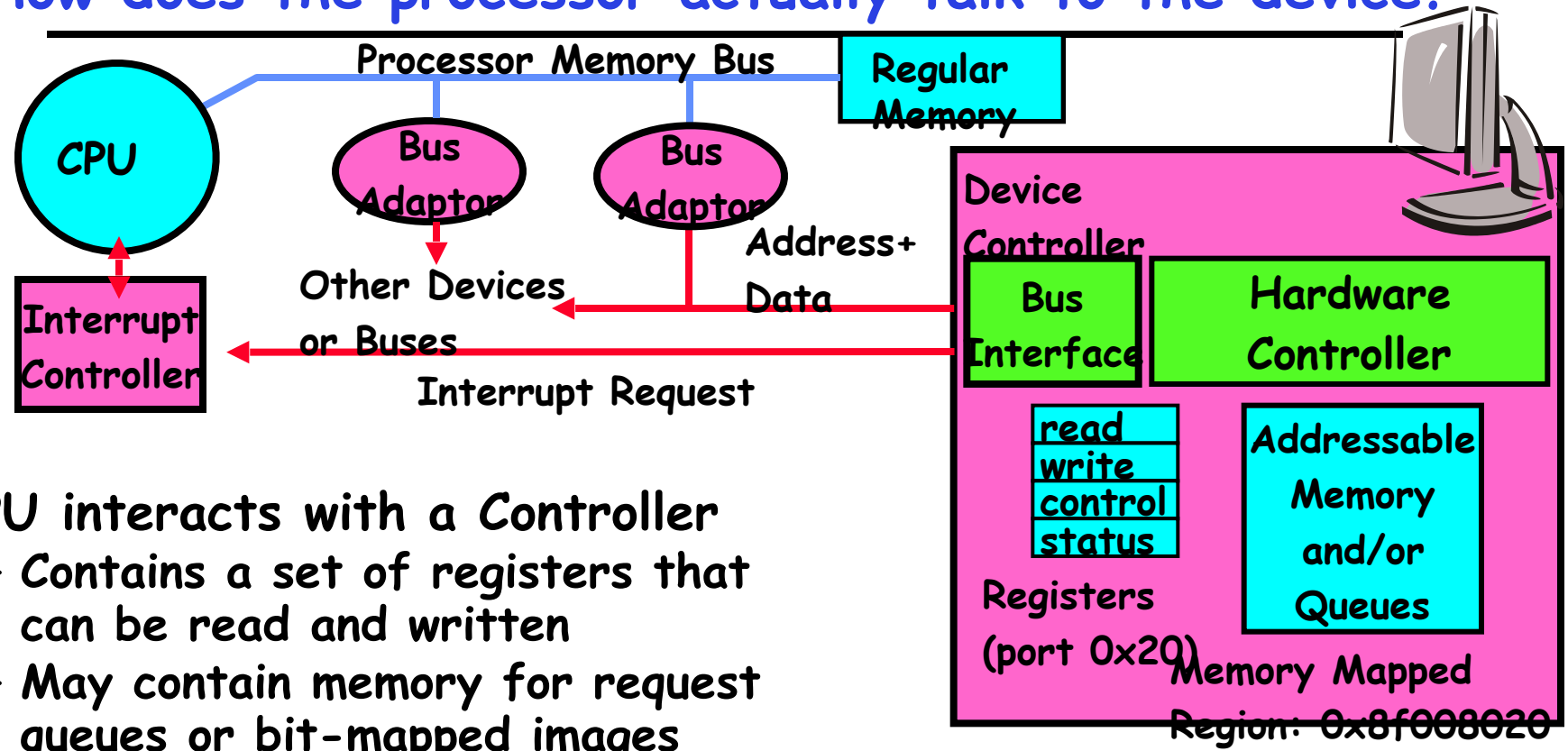


Example Device-Transfer Rates in Mb/s (Sun Enterprise 6000)



- Device Rates vary over 12 orders of magnitude !!!
  - System better be able to handle this wide range
  - Better not have high overhead/byte for fast devices!
  - Better not waste time waiting for slow devices

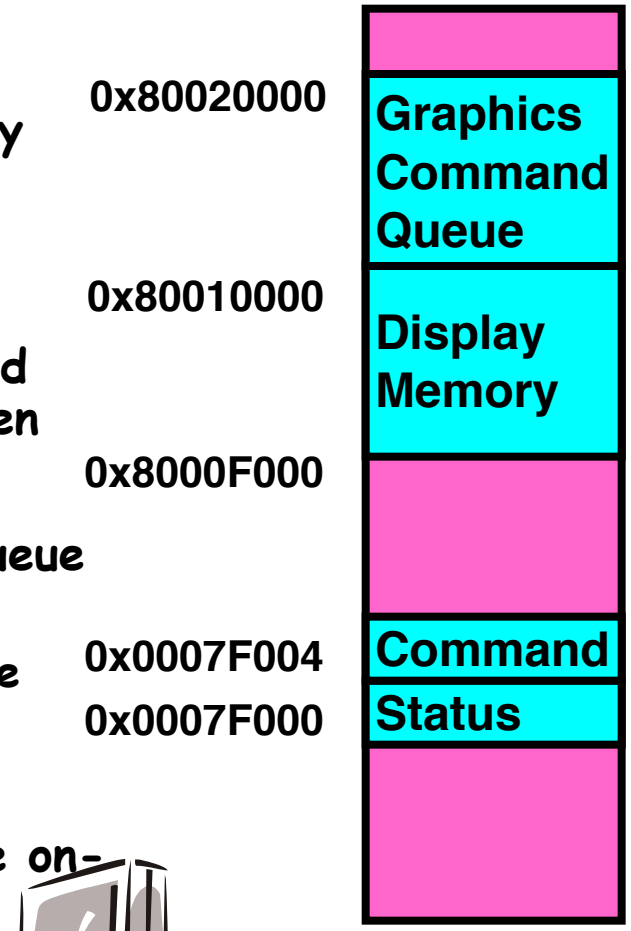
How does the processor actually talk to the device?

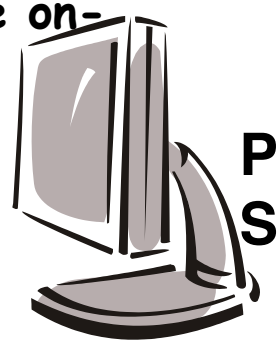


- CPU interacts with a Controller
  - Contains a set of registers that can be read and written
  - May contain memory for request queues or bit-mapped images
- Regardless of the complexity of the connections and buses, processor accesses registers in two ways:
  - I/O instructions: in/out instructions
    - » Example from the Intel architecture: `out 0x21,AL`
  - Memory mapped I/O: load/store instructions
    - » Registers/memory appear in physical address space
    - » I/O accomplished with load and store instructions

Example: Memory-Mapped Display Controller

- Memory-Mapped:
  - Hardware maps control registers and display memory into physical address space
    - » Addresses set by hardware jumpers or programming at boot time
  - Simply writing to display memory (also called the "frame buffer") changes image on screen
    - » Addr: 0x8000F000—0x8000FFFF
  - Writing graphics description to command-queue area
    - » Say enter a set of triangles that describe 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
- Can protect with address translation





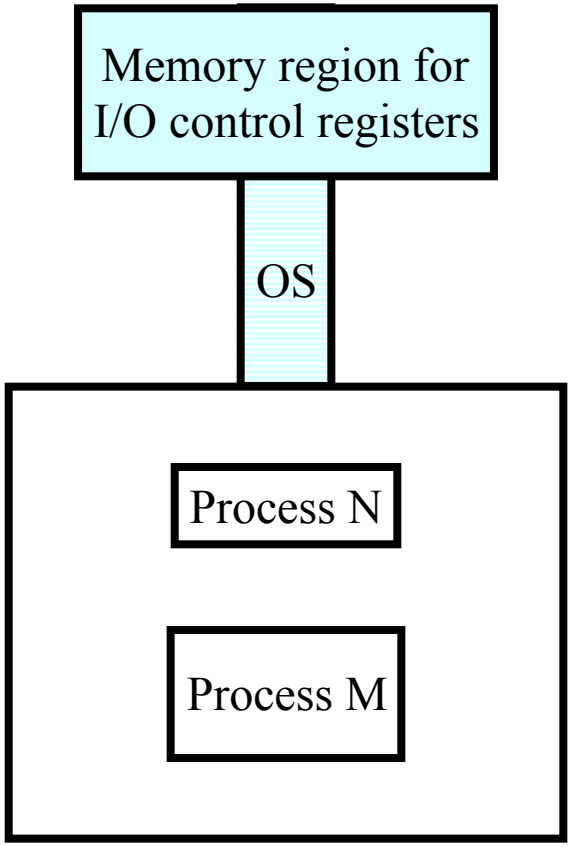
Physical Address Space

Direct I/O

- Each control register is assigned a port number PORT
- Use special assembler language I/O instructions
  - IN REG, PORT: reads in control register PORT and stores result in CPU register REG
  - OUT PORT, REG: writes content of REG to control register PORT

Memory-mapped I/O

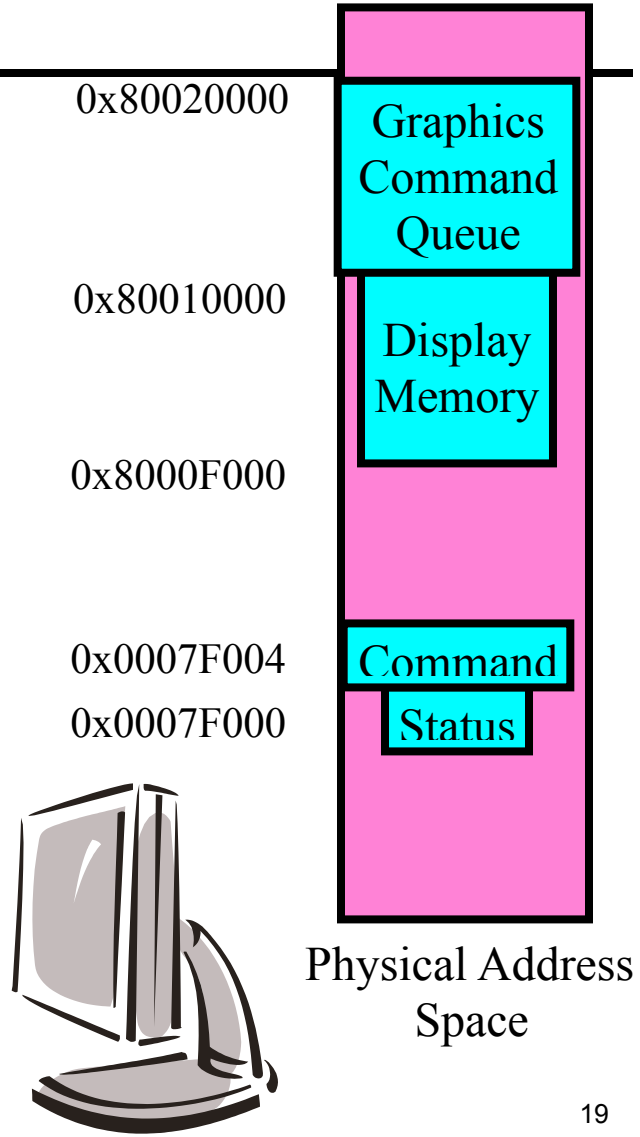
- Map all I/O control registers into the memory space
- Memory map will have a block of addresses that physically corresponds the registers on the I/O controllers rather than to locations in main memory
- When you read from/ write to mem region for I/ O control registers, the request does not go to memory; it is transparently sent to the I/O device



Memory map

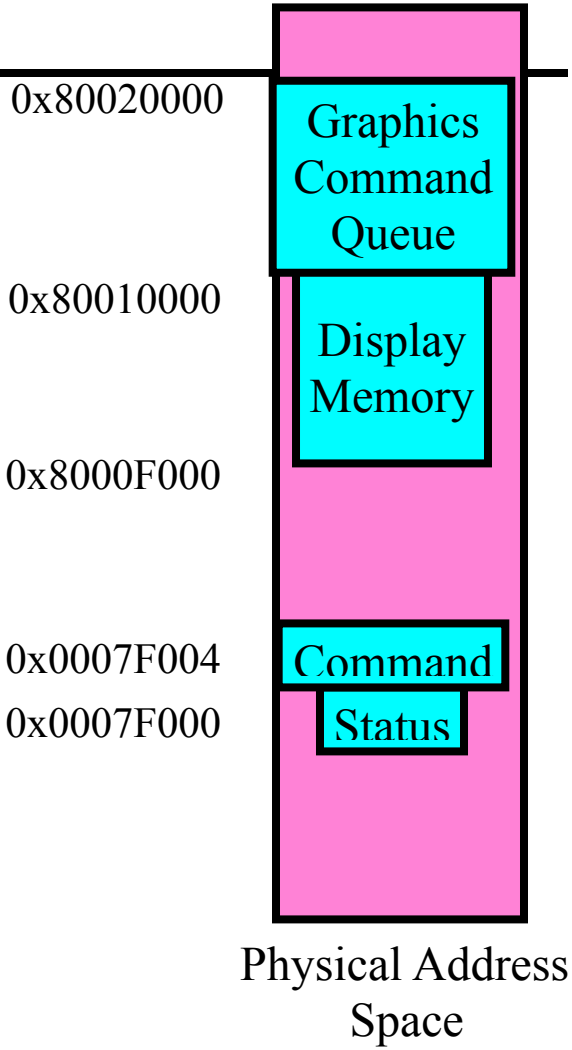
Example: Memory-Mapped Display Controller

- Memory-Mapped:
  - Hardware maps control registers and display memory into physical address space
    - Addresses set by hardware jumpers or programming at boot time
  - Simply writing to display memory (also called the "frame buffer") changes image on screen
    - Addr: 0x8000F000—0x8000FFFF



- Writing graphics description to command-queue area
  - Say enter a set of triangles that describe 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

- Can protect with page tables



Advantages: memory mapped I/O

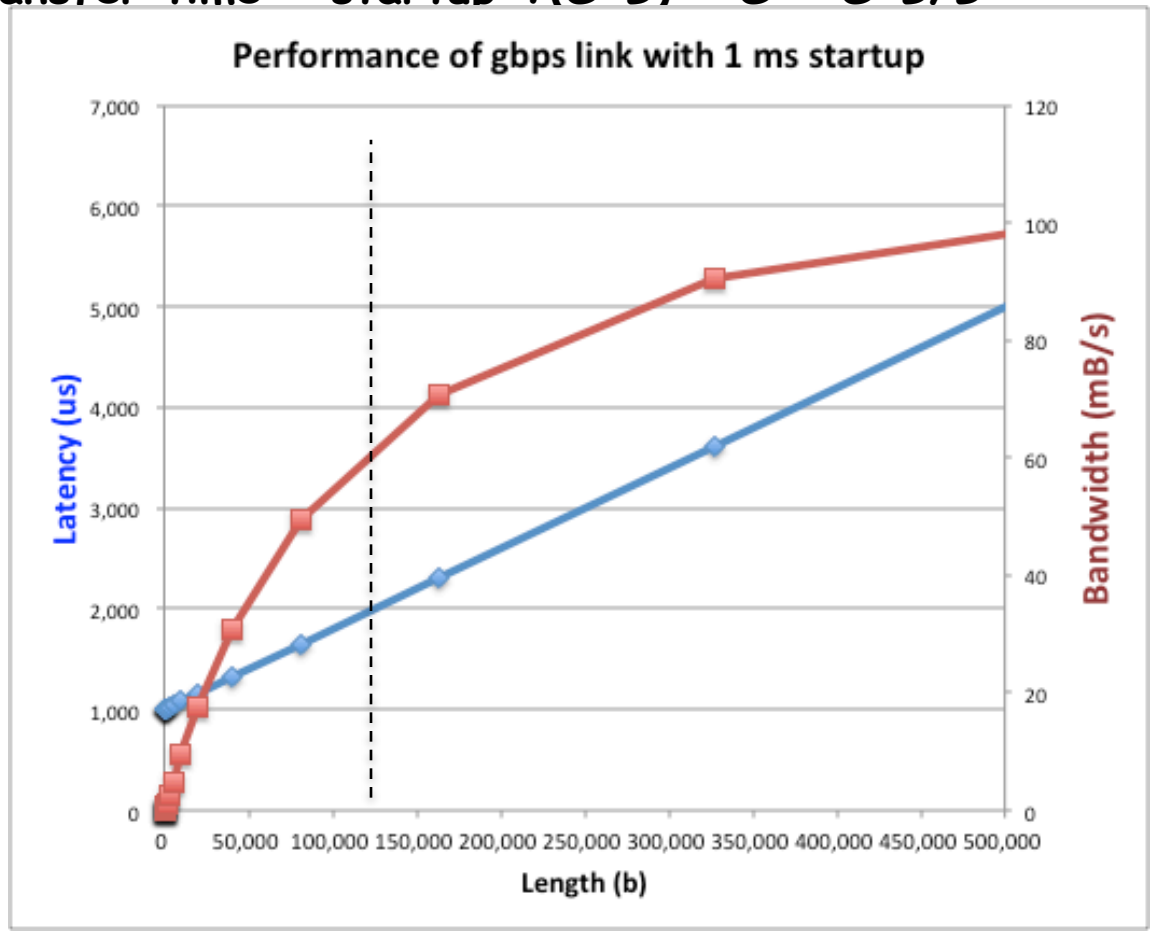
- Allows device drivers and low level control software to be written in C rather than assembler
- Every instruction that can access memory can also access controller registers, reducing the number of instructions needed for I/O
- Can use virtual memory mechanism to protect I/O from user processes
  - Memory region for I/O control registers are mapped to kernel space

I/O Device Notifying the OS

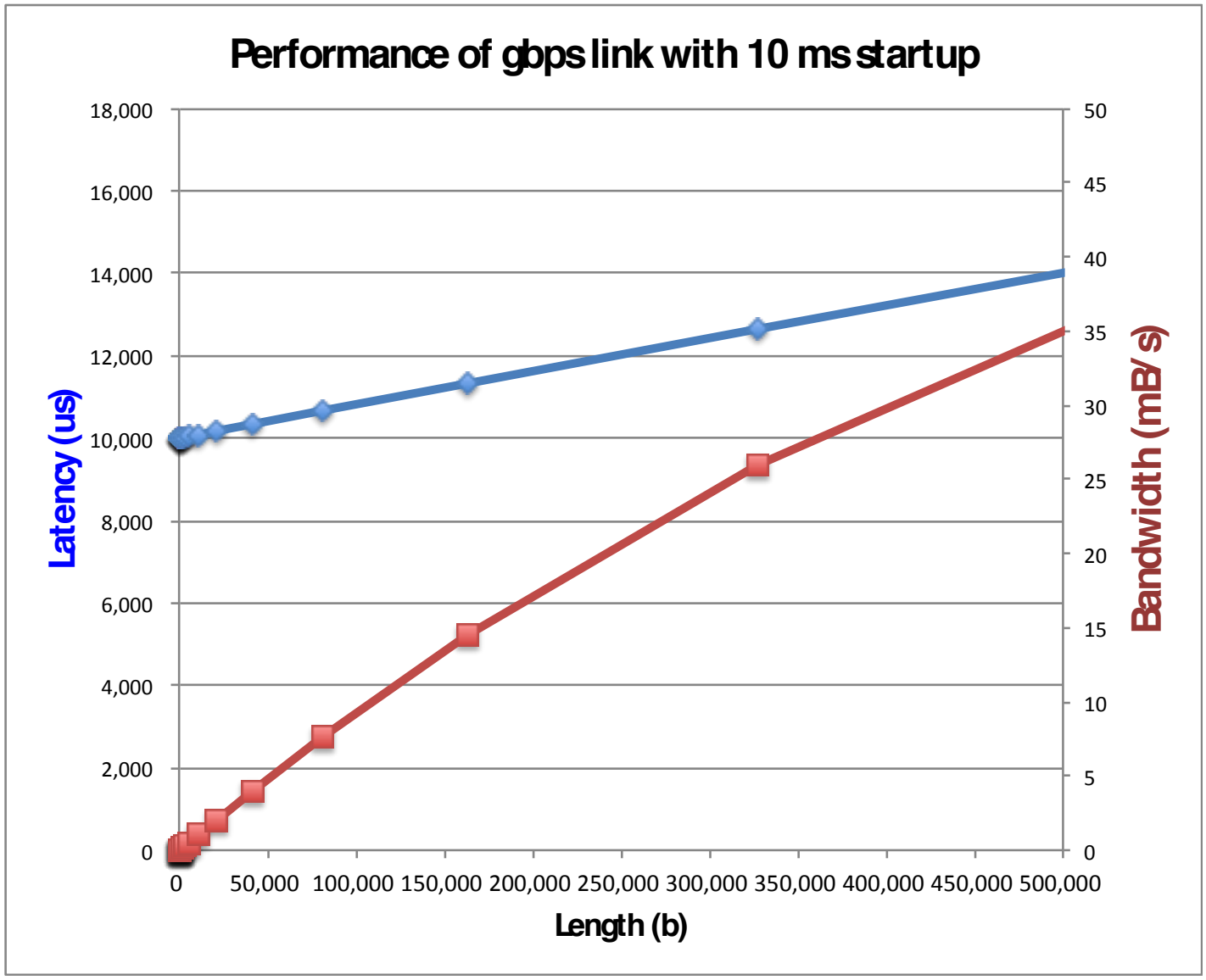
- The OS needs to know when:
  - The I/O device has completed an operation
  - The I/O operation has encountered an error
- I/O Interrupt:
  - Device generates an interrupt whenever it needs service
  - Pro: handles unpredictable events well
  - Con: interrupts relatively high overhead
- Polling:
  - OS periodically checks a device-specific status register
    - » I/O device puts completion information in status register
  - Pro: low overhead
  - Con: may waste many cycles on polling if infrequent or unpredictable I/O operations
- Actual devices combine both polling and interrupts
  - For instance – High-bandwidth network adapter:
    - » Interrupt for first incoming packet
    - » Poll for following packets until hardware queues are empty

Example (fast network)

- Consider a gpbs link (125 MB/s)
- With a startup cost S = 1 ms
- Theorem: half-power point occurs at n=S*B:
  - When transfer time = startup T(S*B) = S + S*B/B



Example: at 10 ms startup (disk)



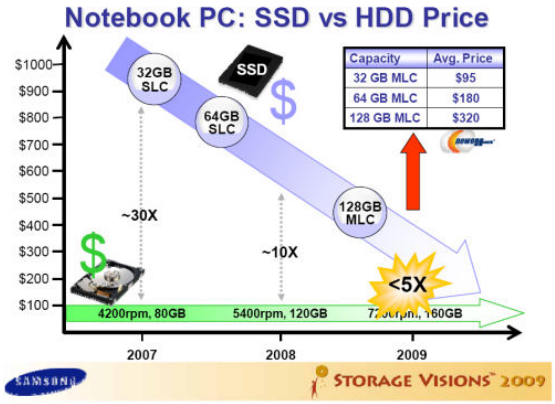
What determines peak BW for I/O ?

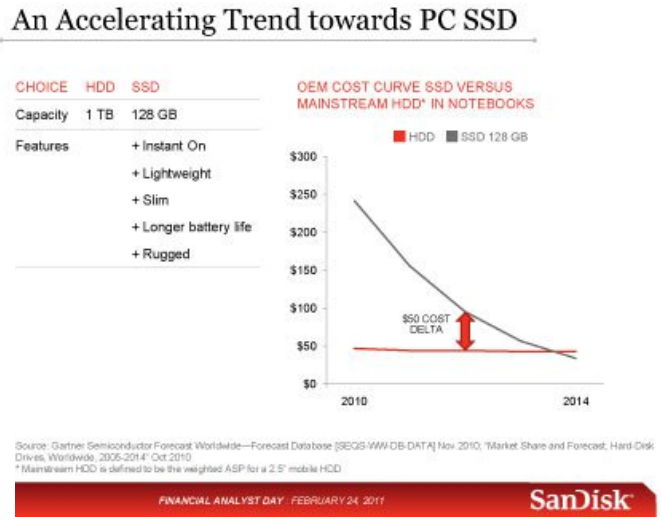
- Bus Speed
  - PCI-X: 1064 MB/s = 133 MHz x 64 bit (per lane)
  - ULTRA WIDE SCSI: 40 MB/s
  - Serial Attached SCSI & Serial ATA & IEEE 1394 (firewire) : 1.6 Gbps full duplex (200 MB/s)
  - USB 1.5 – 12 mb/s
- Device Transfer Bandwidth
  - Rotational speed of disk
  - Write / Read rate of nand flash
  - Signaling rate of network link
- Whatever is the bottleneck in the path

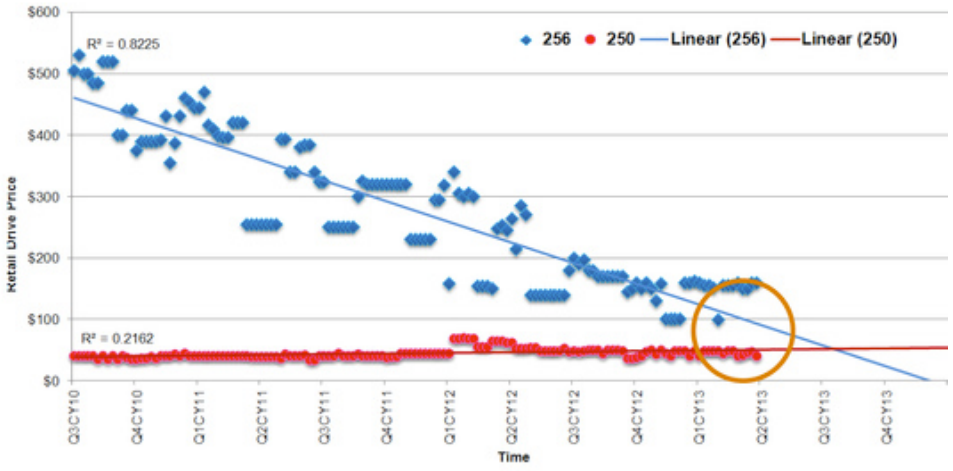
Storage Devices

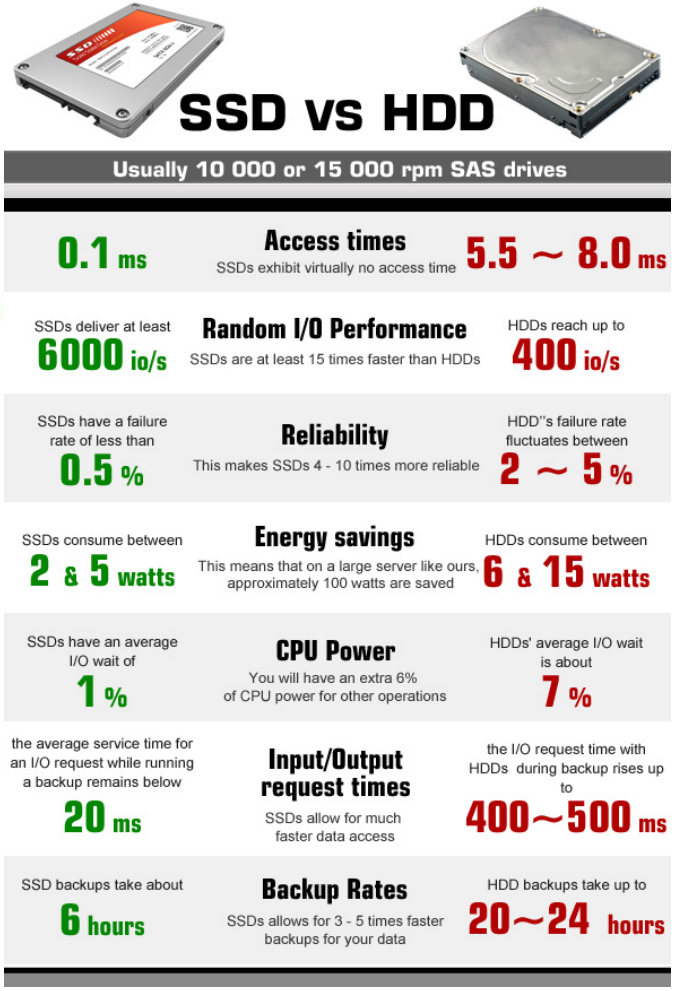
- Magnetic disks
  - Storage that rarely becomes corrupted
  - Large capacity at low cost
  - Block level random access
  - Slow performance for random access
  - Better performance for streaming access
- Flash memory
  - Storage that rarely becomes corrupted
  - Capacity at intermediate cost (50x disk ???)
  - Block level random access
  - Good performance for reads; worse for random writes
  - Erasure requirement in large blocks
  - Wear patterns

Are we in an inflection point?

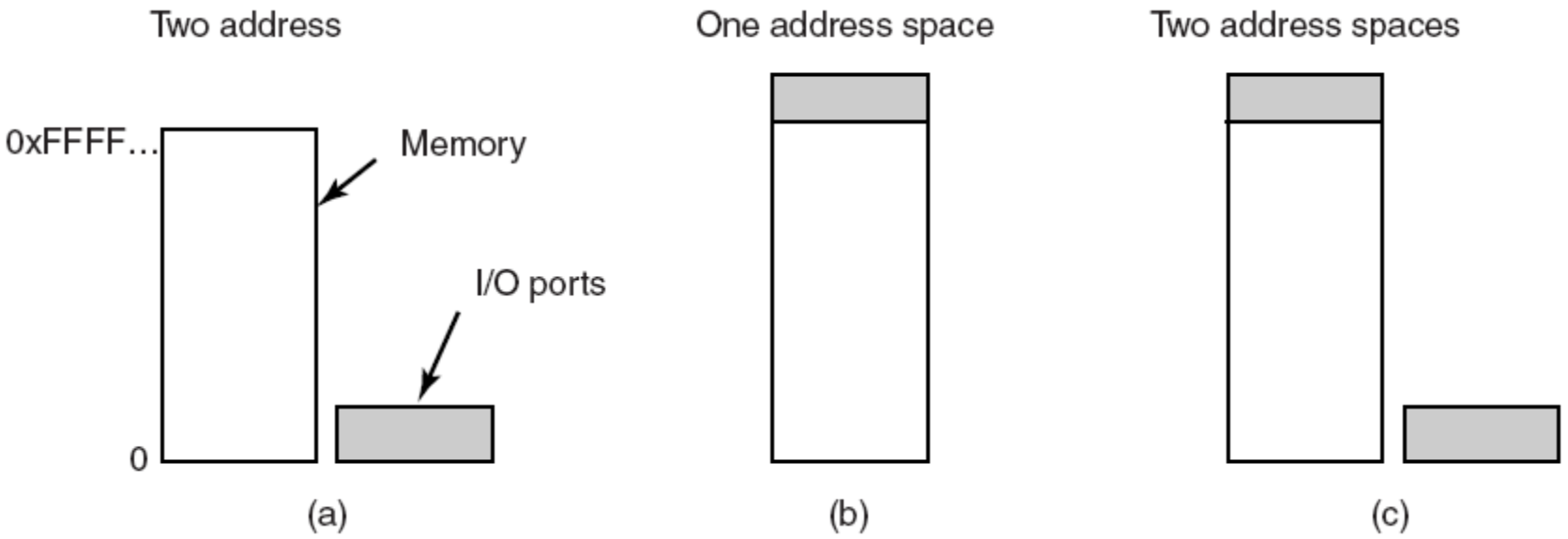








Memory and I/O space



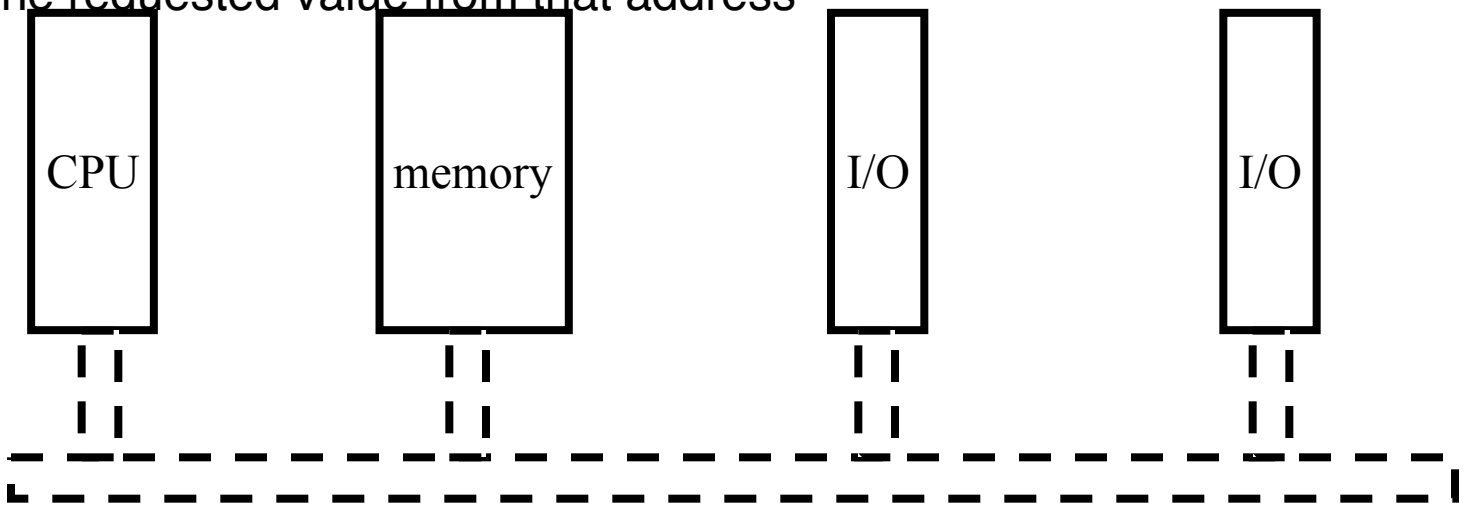
- (a) Separate I/O and memory space.
- (b) Memory-mapped I/O: map device memory (data buffers and control registers) into CPU memory; each device memory address is assigned a unique CPU memory address
- (c) Hybrid: data buffers are memory-mapped; control registers have separate memory space (I/O ports)

Disadvantages: memory mapped I/O

- Need additional complexity in the OS
  - Cannot cache controller registers
  - Changes made in cache do not affect the controller!
  - Must assure that the memory range reserved for memory mapped control registers cannot be cached. (disable caching)
- All memory modules and I/O devices must examine all memory references

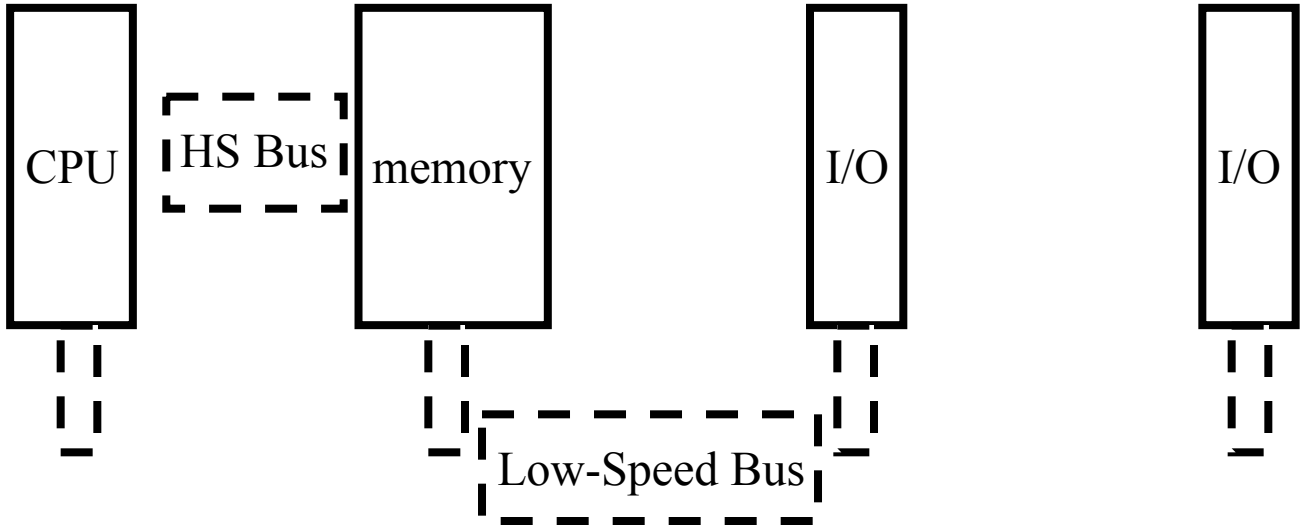
Single Bus: memory mapping

- CPU sends requested address along bus
- Bus carries one request/reply at a time
- Each I/O device controller checks if requested address is in thier memory space
- Device controller whose address space does contain the address replies with the requested value from that address



Memory Bus: memory mapping

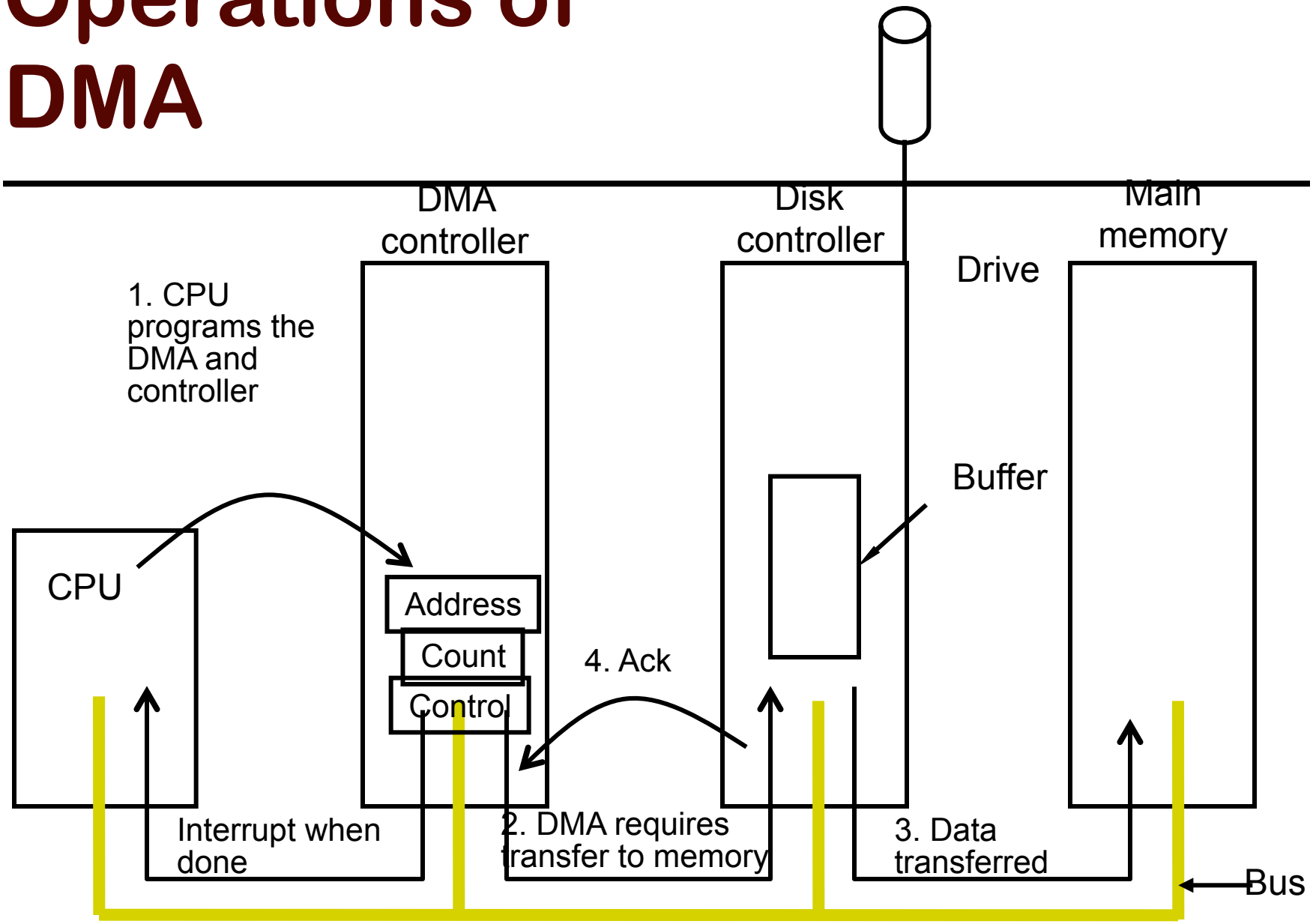
- Most CPUs have a high-speed bus for memory access, and a low-speed bus for peripheral I/O device access.
- CPU first sends memory request to the memory bus, and if that fails (address not found in memory), send it to the I/O bus.



Direct Memory Access (DMA)

- Request data from I/O without DMA
  - Device controller reads data from device
  - It interrupts CPU when a byte/block of data available
  - CPU reads controller's buffer into main memory
  - Too many interruptions, expensive
- DMA: direct memory access
  - A DMA controller with registers read/written by CPU
  - CPU programs the DMA: what to transfer where
    - Source, destination and size
  - DMA interrupts CPU only after all the data are transferred.

Operations of DMA



DMA Details

1. CPU programs DMA controller by setting registers
   - Address, count, control
2. DMA controller initiates the transfer by issuing a read request over the bus to the disk controller
3. Write to memory in another standard bus cycle
4. When the write is done, disk controller sends an acknowledgement signal to DMA controller
   - If there is more to transfer, go to step 2 and loop
5. DMA controller interrupts CPU when transfer is complete.
   - CPU doesn't need to copy anything.

Transfer Modes

- Word-at-a-time (cycle stealing)
  - DMA controller acquires the bus, transfer one word, and releases the bus
  - CPU waits for bus if data is transferring
  - Cycle stealing: steal an occasional bus cycle from CPU once in a while
- Burst mode
  - DMA holds the bus until a series of transfers complete
  - More efficient since acquiring bus takes time
  - Block the CPU from using bus for a substantial amount of time

Outline

- Overview
- Principles of I/O hardware
- **Principles of I/O software**
- Disks

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

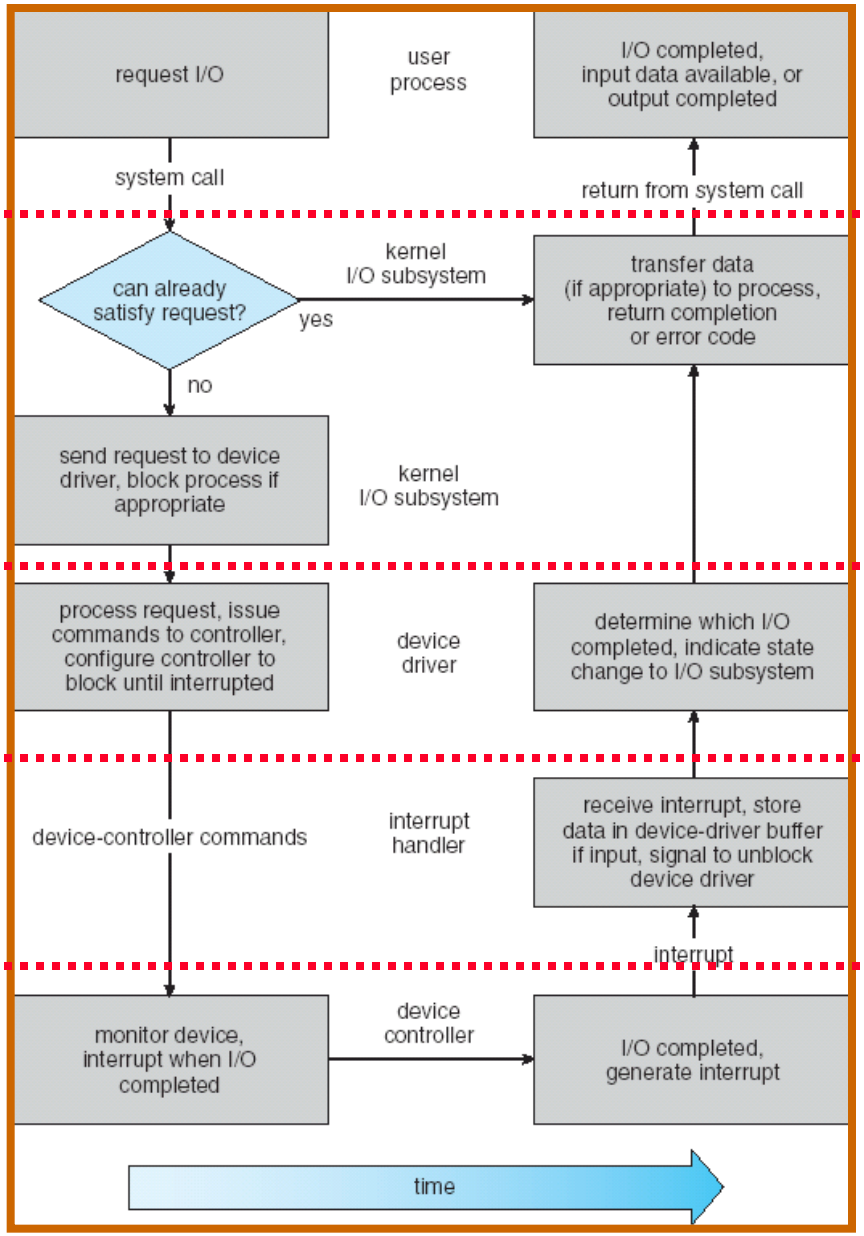
User Program

Kernel I/O Subsystem

Device Driver Top Half

Device Driver Bottom Half

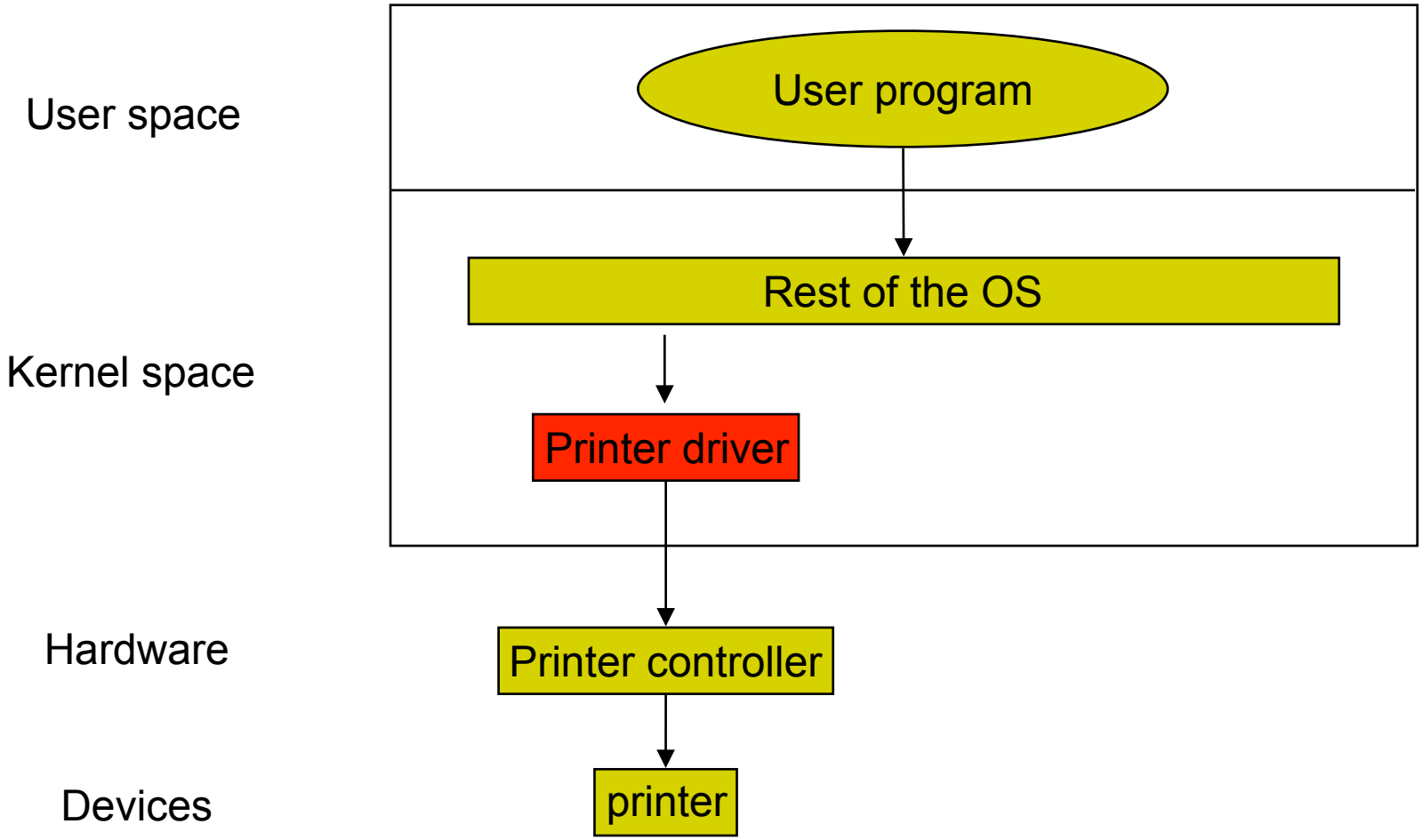
Device Hardware



Basic Performance Concepts

- **Response Time or Latency:** Time to perform an operation (s)
- **Bandwidth or Throughput:** Rate at which operations are performed (op/s)
  - Files: mB/s, Networks: mb/s, Arithmetic: GFLOP/s
- **Start up or "Overhead":** time to initiate an operation
- Most I/O operations are roughly linear
  - Latency (n) = Ovhd + n/Bandwidth

Logical Position of Device Drivers

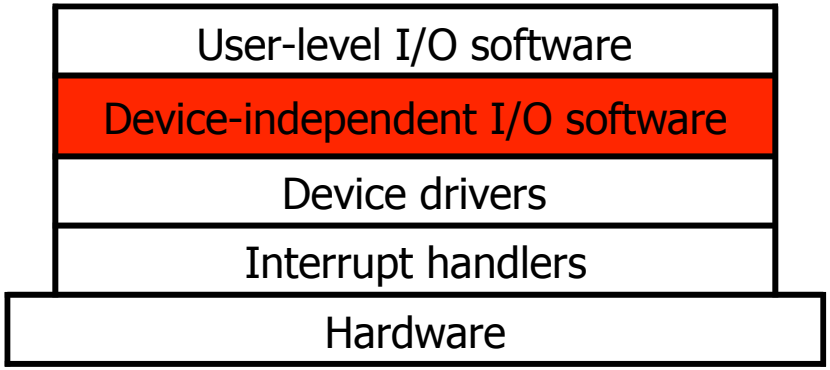


How to Install a Driver?

- Re-compile and re-link the kernel
  - Drivers and OS are in a single binary program
  - Used when devices rarely change
- Dynamically loaded during OS initialization
  - Used when devices often change
- Dynamically loaded during operation
  - Plug-and-Play

Device-Independent I/O Software

- Why device-independent I/O software?
  - Perform I/O functions common to all devices
  - Provide a uniform interface to user-level software
- It provides:
  - Uniform interfacing for devices drivers
  - Buffering
  - Error reporting
  - Allocating and releasing dedicated devices
  - Providing a device-independent block size



Uniform Interfacing for Device Drivers

- New device → modify OS, not good
- Provide the same interface for all drivers
  - Easy to plug a new driver
  - In reality, not absolutely identical, but most functions are common
- Name I/O devices in a uniform way
  - Mapping symbolic device names onto the proper driver
  - Treat device name as file name in UNIX
    - E.g., hard disk /dev/disk0 is a special file. Its i-node contains the major device number, which is used to locate the appropriate driver, and minor device number.

Uniform Interfacing for Device Drivers

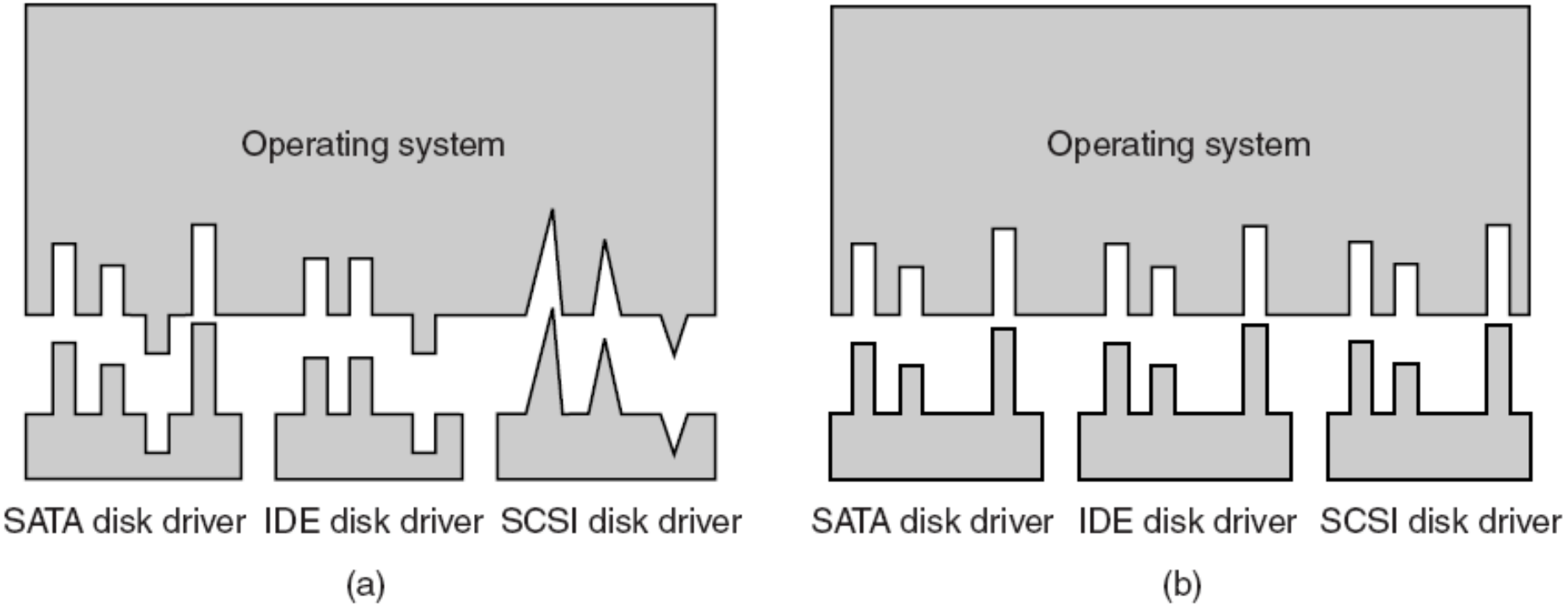
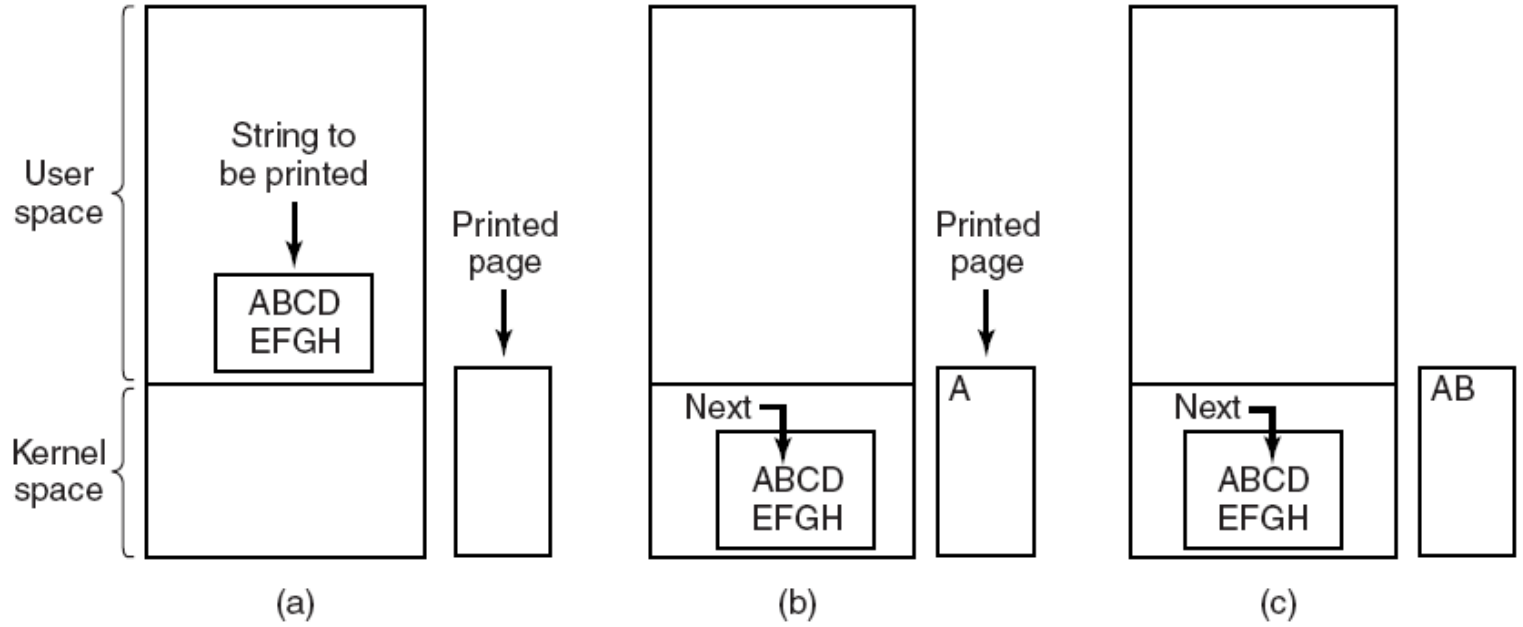


Figure 5-14. (a) Without a standard driver interface. (b) With a standard driver interface.

Types of I/O

- Synchronous I/O
  - Programmed I/O:
    - Process busy-waits (polls) while I/O is completed
- Asynchronous I/O
  - Interrupt driven I/O:
    - CPU issues an I/O command to I/O device
    - CPU enters wait state
    - CPU continues with other processing (same or more likely different process)
    - I/O device generates an interrupt when it finishes and the CPU finishes processing the interrupt before continuing with its present calculations.
  - Direct Memory Access (DMA)

Programmed I/O: Writing a String to Printer




```
copy_from_user(buffer, p, count);                    /* p is the kernel buffer */
for (i = 0; i < count; i++) {                        /* loop on every character */
        while (*printer_status_reg != READY) ;       /* loop until ready */
        *printer_data_register = p[i];               /* output one character */
}
return_to_user( );
```

Programmed I/O

- First the data are copied to the kernel. Then the operating system enters a tight loop outputting the characters one at a time.
  - After outputting a character, the CPU continuously polls the device in a while loop to see if it is ready to accept another one.
- Busy waiting wastes CPU time while waiting for IO to complete

Interrupt-Driven I/O

```
copy_from_user(buffer, p, count);
enable_interrupts( );
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler( );
```

(a)

```
if (count == 0) {
    unblock_user( );
} else {
    *printer_data_register = p[i];
    count = count – 1;
    i = i + 1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

(b)

- (a) Code executed at the time the print system call is made. Buffer is copied to kernel space; 1st char is copied to printer as soon as it is ready to accept a char
- (b) ISR for printer interrupt. When printer has printed the 1st char, it generates an interrupt to run the ISR; if no more chars to print, it unblocks the user process; otherwise, it prints the next char and returns from the interrupt. Each interrupt grabs one char from the kernel buffer and prints it.

I/O using DMA

```
copy_from_user(buffer, p, count);
set_up_DMA_controller( );
scheduler( );
```

(a)

```
acknowledge_interrupt( );
unblock_user( );
return_from_interrupt( );
```

(b)

- (a) Code executed when the print system call is made.
- (b) ISR for printer interrupt
- Let the DMA controller feed the chars to printer one at a time to free up the CPU

Interrupt Handlers

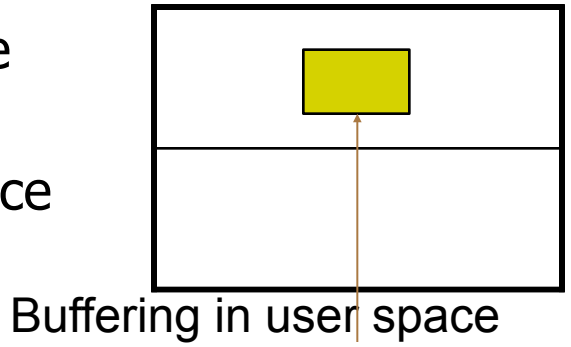
- Hide I/O interrupts deep in OS
  - Device driver starts I/O and blocks (e.g., *down* a mutex)
  - Interrupt wakes up driver
- Process an interrupt
  - Save registers (which to where?)
  - Set up context (TLB, MMU, page table)
  - Run the handler (usually the handler will be blocked)
  - Choose a process to run next
  - Load the context for the newly selected process
  - Run the process

Buffering for Input

- Motivation: consider a process that wants to read data from a modem
  - User process handles one character at a time.
  - It blocks if a character is not available
  - Each arriving character causes an interrupt
  - User process is unblocked and reads the character.
  - Try to read another character and block again.
  - Many short runs in a process: inefficient!
    - Overhead of context switching

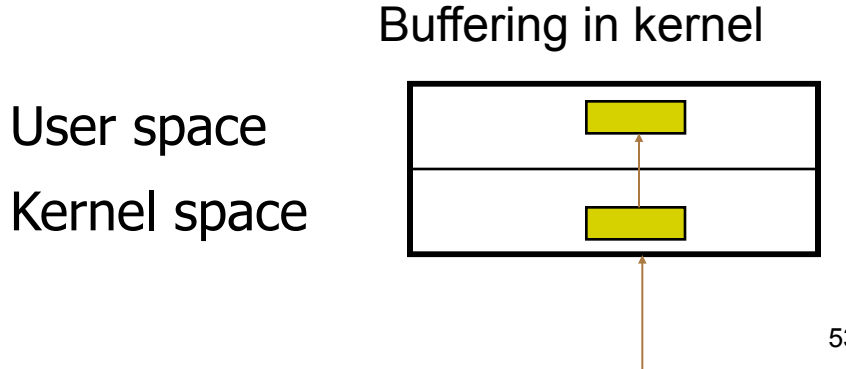
Buffering in User Space

- Set a buffer in user process' space
- User process is waked up only if the buffer is filled up by interrupt service procedure. More efficient.
- Can the buffer be paged out to disk?
  - If yes, where to put the next character?
  - No, by locking page in memory: the pool of other (available) pages shrink



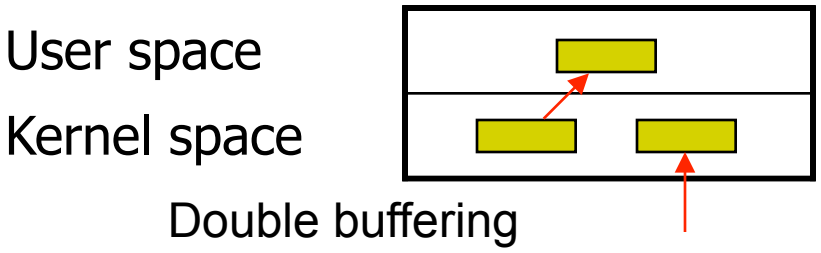
Buffering in Kernel

- Two buffers: one in kernel and one in user
- Interrupt handler puts characters into the buffer in kernel space
  - Kernel buffers are never paged to disk
- When full, copy the kernel buffer to user buffer
  - But where to store the new arrived characters when the user-space page is being loaded from disk?



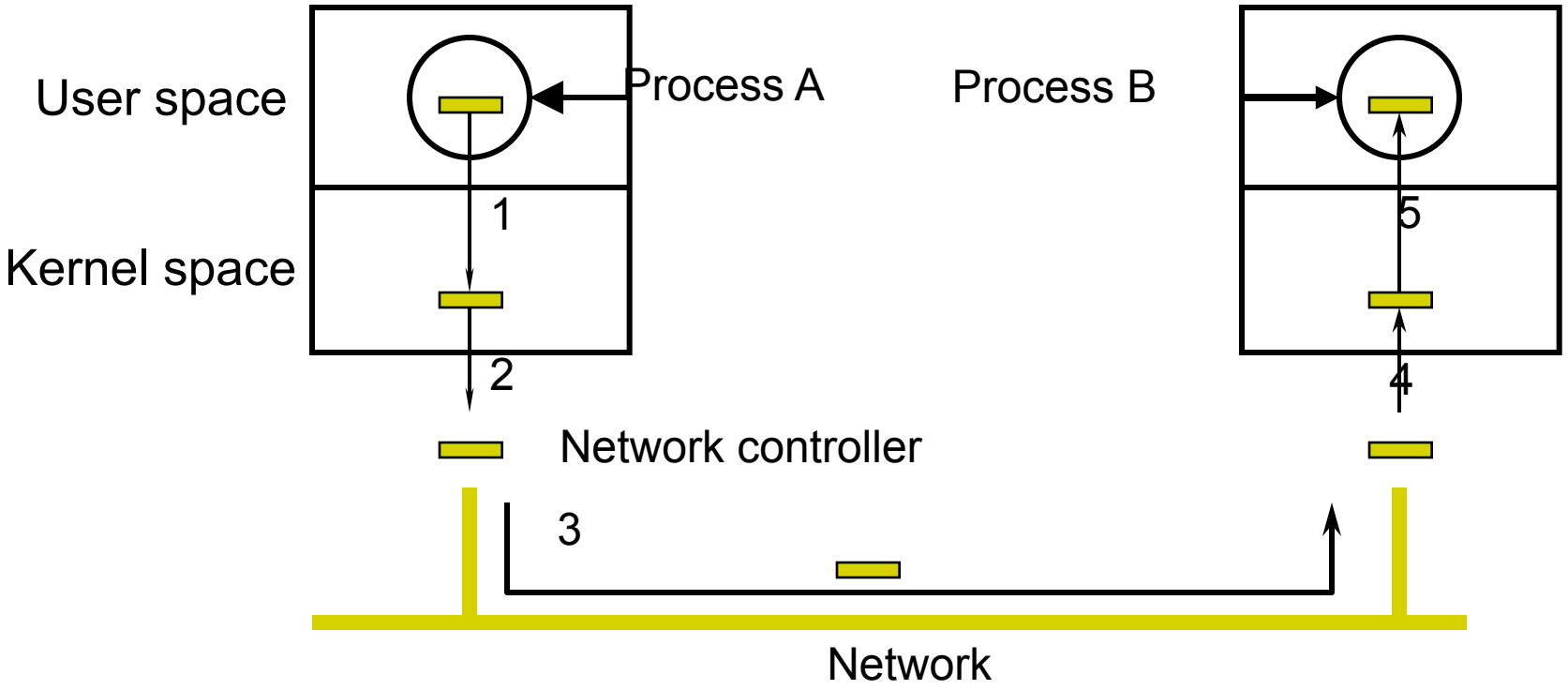
Double Buffering in Kernel

- Two kernel buffers
- When the first one fills up, but before it has been emptied, the second one is used.
- Buffers are used in turn: while one is being copied to user space, the other is accumulating new input



Downside of Data Buffering

- Many sequential buffering steps slow down transmission



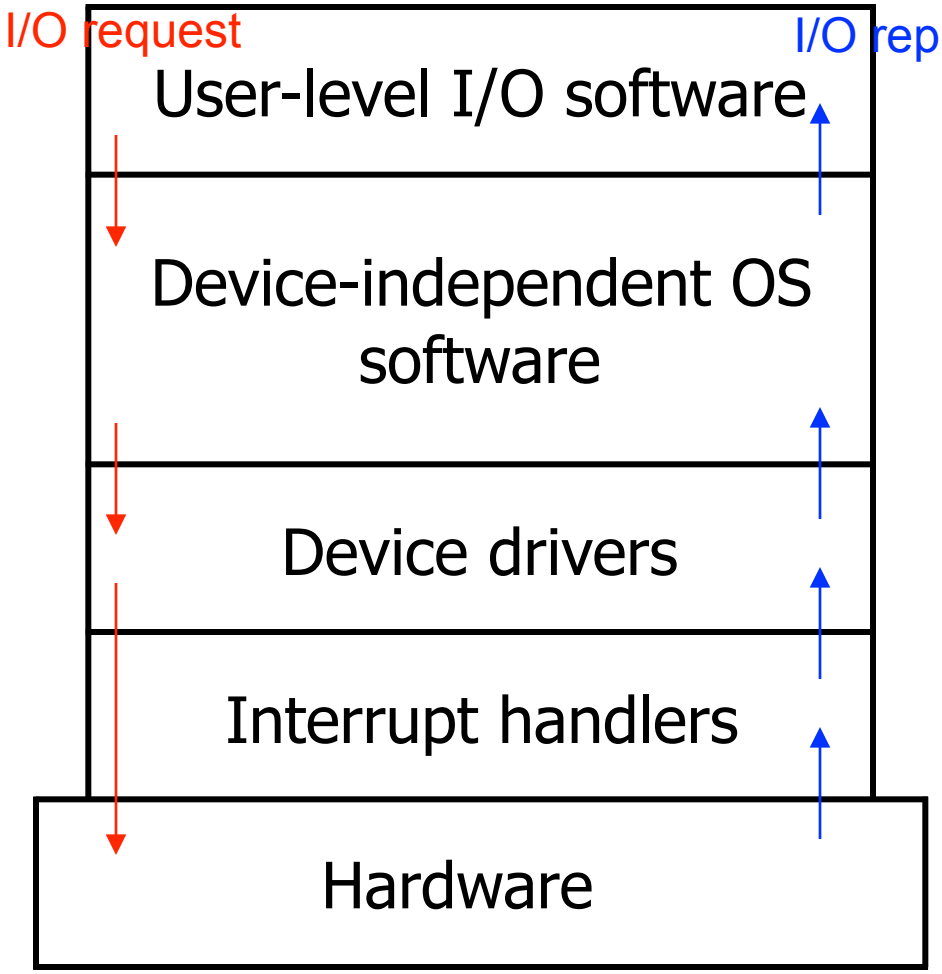
Handling I/O Errors

- Programming errors: ask for something impossible
  - E.g. writing a keyboard, reading a printer
  - Invalid parameters, like buffer address
  - Report an error code to caller
- Actual I/O error
  - E.g. write a damaged disk block
  - Handled by device driver and/or device-independent software
- System error
  - E.g. root directory or free block list is destroyed
  - display message, terminate system

Allocating Dedicated Devices

- Before using a device, make the system call *open*
- When the device is unavailable
  - The call fails, or
  - The caller is blocked and put on a queue
- Release the device by making the *close* system call

Summary: I/O Software



Make I/O call; format I/O; spooling

Naming, protection, blocking, buffering, allocation

Setup device registers; check status

Wake up driver when I/O completed

Perform I/O operation

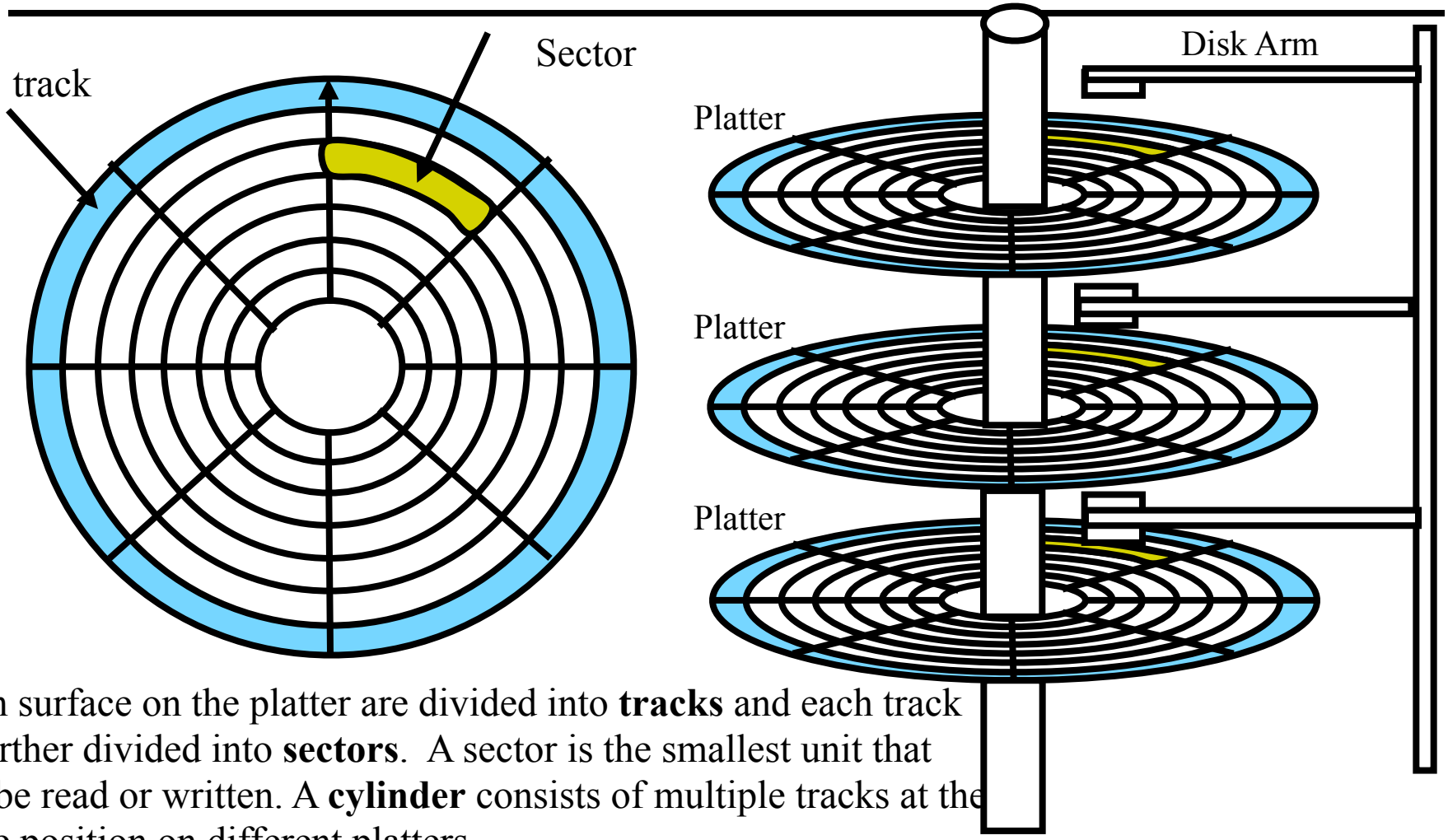
Outline

- Overview
- Principles of I/O hardware
- Principles of I/O software
- **Disks**

Types of Disks

- Magnetic disks
  - Hard disks and floppy disks
  - Reads/writes are equally fast
  - Ideal secondary memory
  - Highly reliable storage
- Optical disks
  - CD-ROM, CD-R: 600MB
  - DVD: 4.7-17GB
- Flash disks
  - USB drive

Disk Geometry



Each surface on the platter are divided into **tracks** and each track is further divided into **sectors**. A sector is the smallest unit that can be read or written. A **cylinder** consists of multiple tracks at the same position on different platters.

Properties

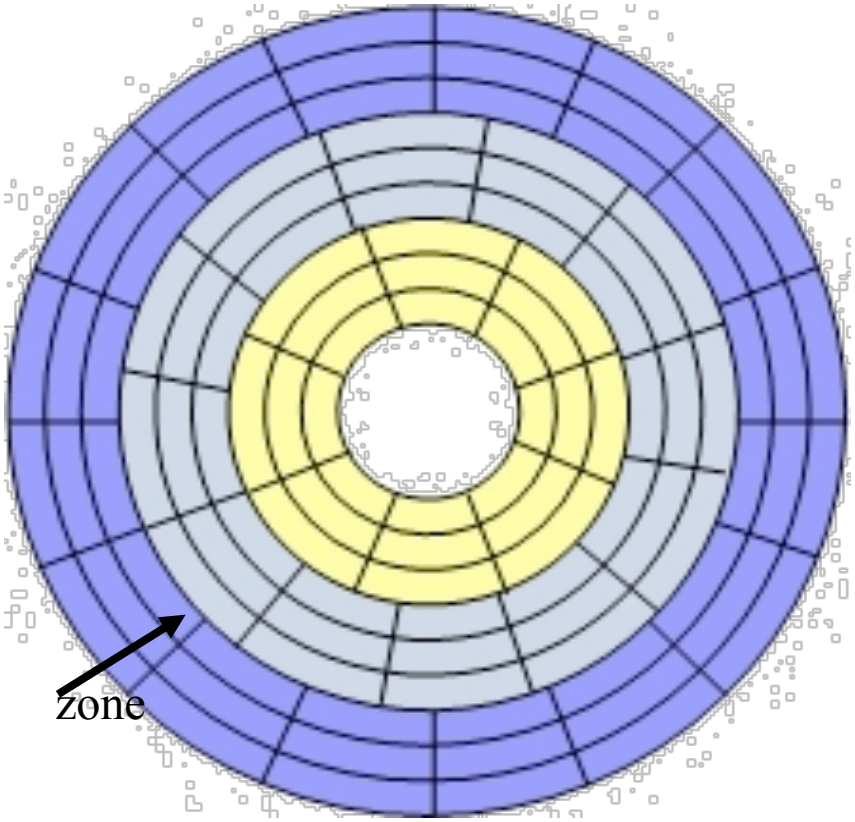
- Independently addressable element: sector
  - A block is a group of sectors. OS always transfers multiple blocks.
- A disk can access directly any given block of information it contains (random access). Can access any file either sequentially or randomly.
- A disk can be rewritten in place: it is possible to read/modify/write a block from the disk
- Typical numbers (depending on the disk size):
  - 500 to more than 20,000 tracks per surface
  - 32 to 800 sectors per track

Comparison of old and new disks

Parameter	IBM 360-KB floppy disk	WD 18300 hard disk
Number of cylinders	40	10601
Tracks per cylinder	2	12
Sectors per track	9	281 (avg)
Sectors per disk	720	35742000
Bytes per sector	512	512
Disk capacity	360 KB	18.3 GB
Seek time (adjacent cylinders)	6 msec	0.8 msec
Seek time (average case)	77 msec	6.9 msec
Rotation time	200 msec	8.33 msec
Motor stop/start time	250 msec	20 sec
Time to transfer 1 sector	22 msec	17 μsec

- Figure 5-18. Disk parameters for the original IBM PC 360-KB floppy disk and a Western Digital WD 18300 hard disk.

Zones



- Real disks will have zones with more sectors towards the outer edge and fewer toward the inner edge
- Most disks present a *virtual geometry* to the OS, which assumes a constant number of sectors per track. The controller maps the OS requested sector to the physical sector on the disk

Physical vs. Virtual Geomery

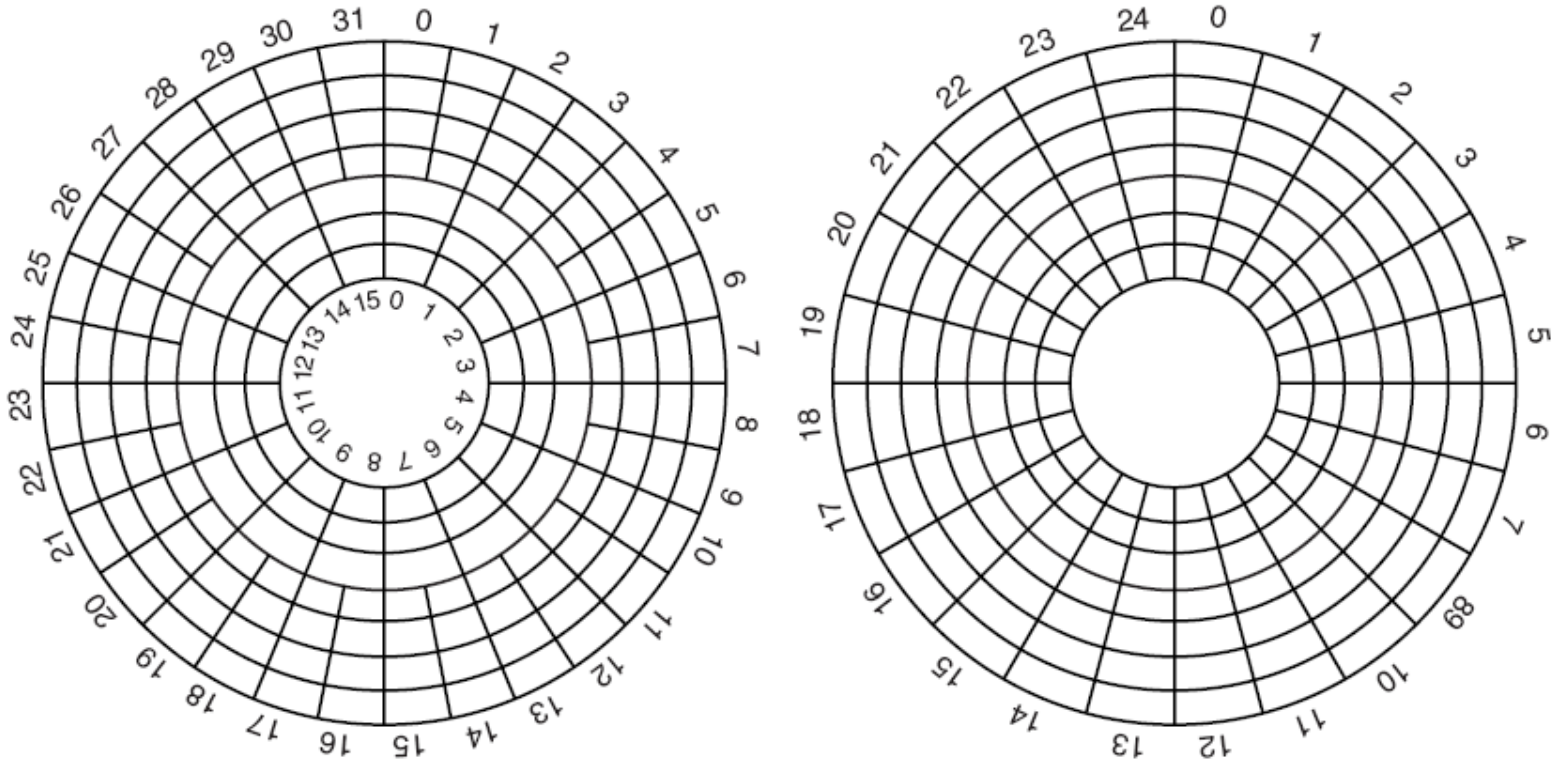


Figure 5-19. (a) Physical geometry of a disk with two zones.
(b) A possible virtual geometry for this disk.

Cylinders

- In the disk there are multiple platters (often two sided). And there are heads to read each side of each platter
- All the heads move in and out together.
- If we consider one head it is above a particular track on a particular platter of the disk
- If we consider the whole disk, A cylinder is the group of tracks (track n on each side of each platter) that can be read when the heads are in a particular position (above a certain track)

Sectors

- Each sector contains
  - Preamble: synchronization marker
  - Sector information, cylinder and sector number
  - Data
  - Error detection/correction information
- Whole sector is read to buffer in controller
- Error detection/correction is performed
- Data is transferred to its destination memory address from the disk controller's buffer

Format of a Sector

A disk sector

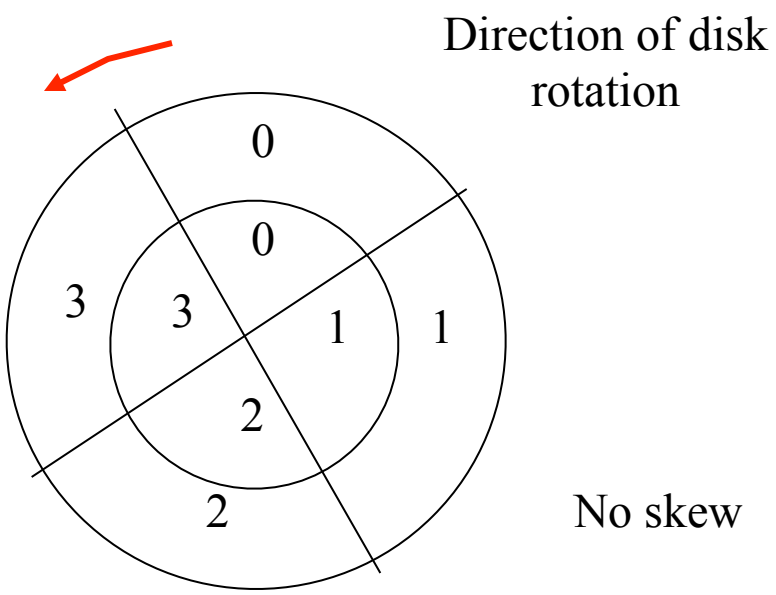
- Preamble: recognize the start of the sector. It also contains the cylinder and sector numbers.
- Data: most disks use 512-byte sectors
- ECC (Error Correcting Code): can be used to recover from errors
- Gap between sectors

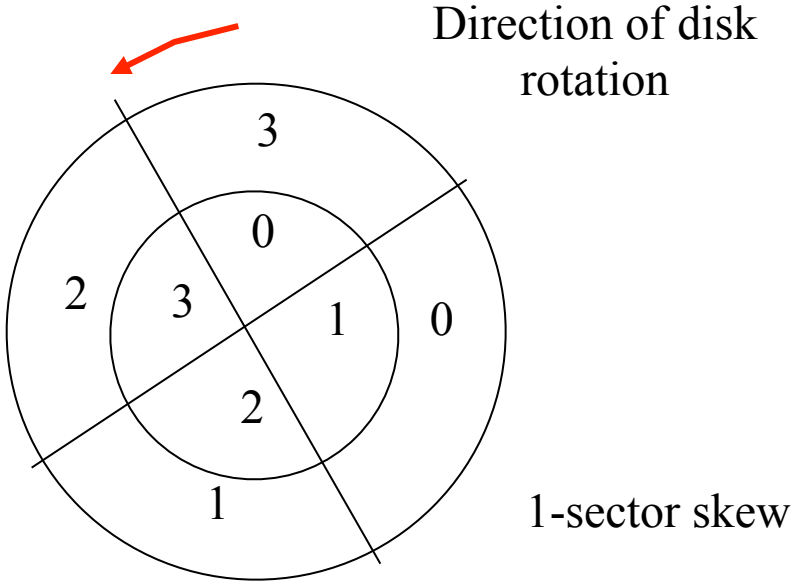
Cost of Read / Write A Disk Block

- Seek time
  - Time to move the arm to the proper cylinder
  - Dominate the other two times for most disks
  - E.g., 0.8 msec for adjacent cylinders
- Rotational delay
  - Time for the proper sector to rotate under the head
  - E.g, 0.03 msec for adjacent sectors
- Data transfer time
  - E.g., 17 μsec for one sector

Cylinder Skew

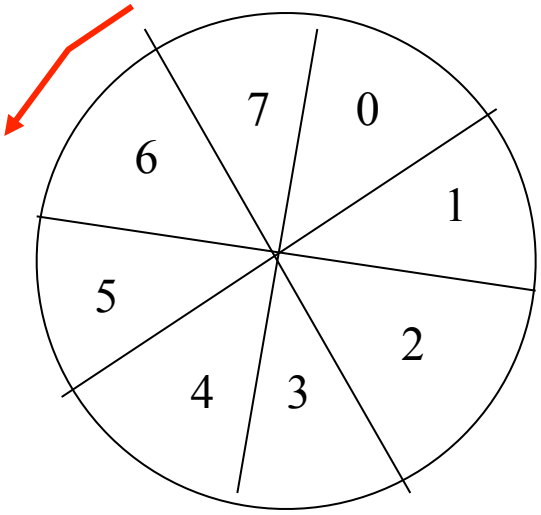
- The position of sector 0 on each track is offset from the previous track. This offset is called *cylinder skew*.
- Allow the disk to read multiple tracks in one continuous operation without losing data



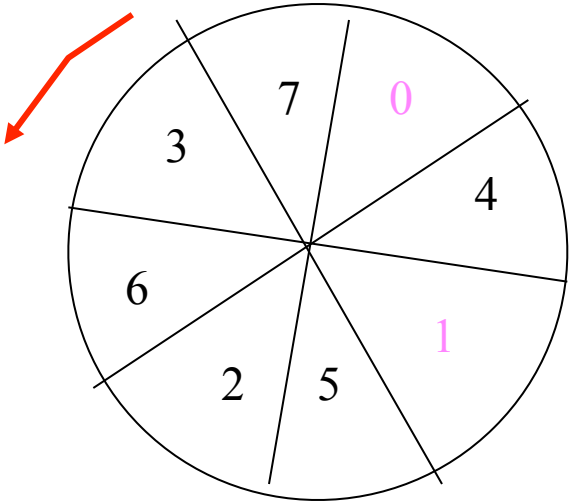


Sector Interleaving

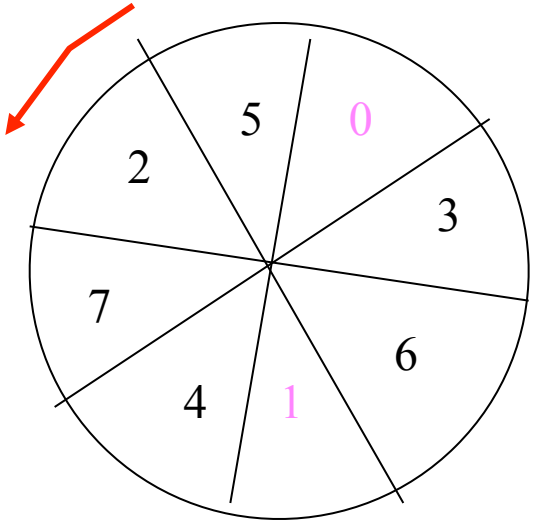
- Consider a controller with one sector buffer. A request of reading two consecutive sectors. When the controller is busy with transferring one sector of data to memory, the next sector will fly by the head.
- Solution: sector interleaving



No interleaving



Single interleaving



Double interleaving

Disk Scheduling

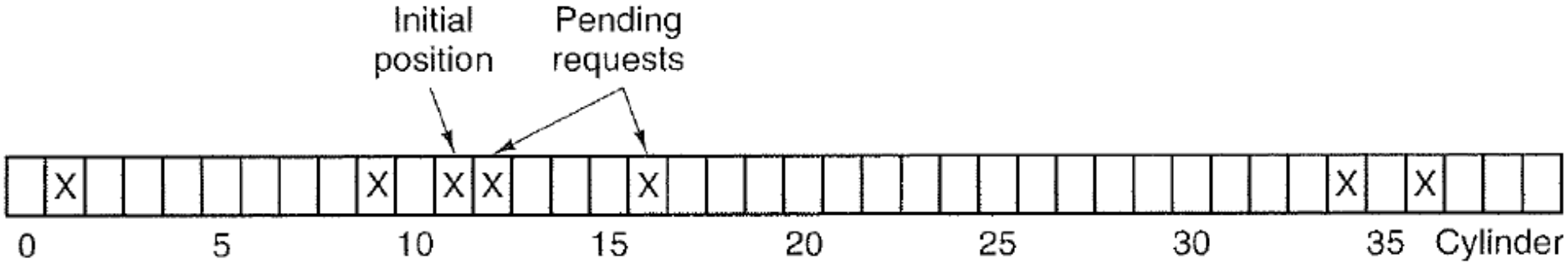
- Want to schedule disk requests to optimize performance. Must consider
  - Seek time (time to move the arm to the proper cylinder)
  - Rotational delay (time for the proper sector to rotate under the head)
  - Data transfer time
- Different approaches to the order in which disk accesses are processed

First Come First Serve

- Requests are removed from the queue in the order that they arrived.
  - For a small number of processes, each process will have clusters of nearby accesses so some improvement over random scheduling may occur
  - For a large number of processes, many areas on the disk may be in demand. May perform very similarly to random request order

FCFS Example

- Consider a disk with 40 cylinders. Requests for cylinder # 11, 1, 36, 16, 34, 9, 12 come in that order

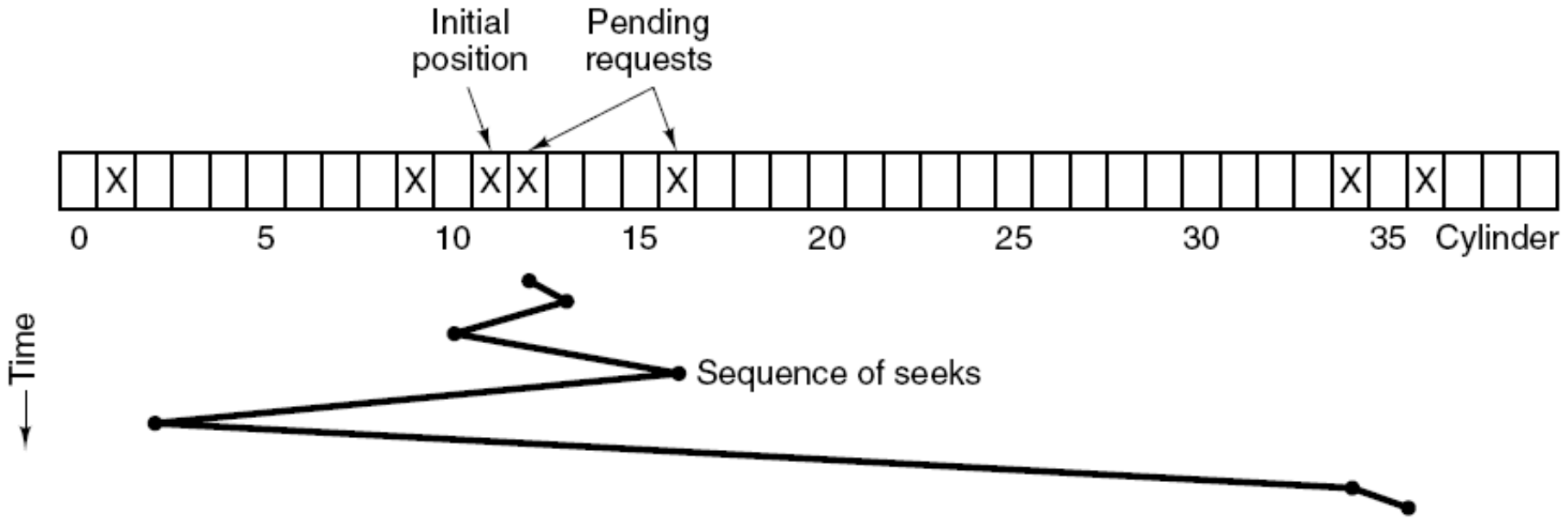


- From initial position of 11, the disk arm serves requests in the order of (1, 36, 16, 34, 9, 12) with movements of (10, 35, 20, 18, 25, 3), total of 111 cylinders

Shortest seek first (SSF)

- Choose the request in the queue whose location on the disk is closest to the present location of the head (shortest seek time)
  - More efficient than FCFS, transfer time cannot be changed so minimizing seek time will help optimize the system
  - Can cause starvation, If there are many requests in one area of the disk, processes using other parts of the disk may never have their requests filled.
  - On a busy system the arm will tend to stay near the center of the disk
  - Need a tie breaking algorithm (what if there are two requests the same distance away in different directions)

SSF Example



- From initial position of 11, the disk arm serves requests in the order of (12, 9, 16, 1, 34, 36) with movements of (1, 3, 7, 15, 33, 2), total of 61 cylinders

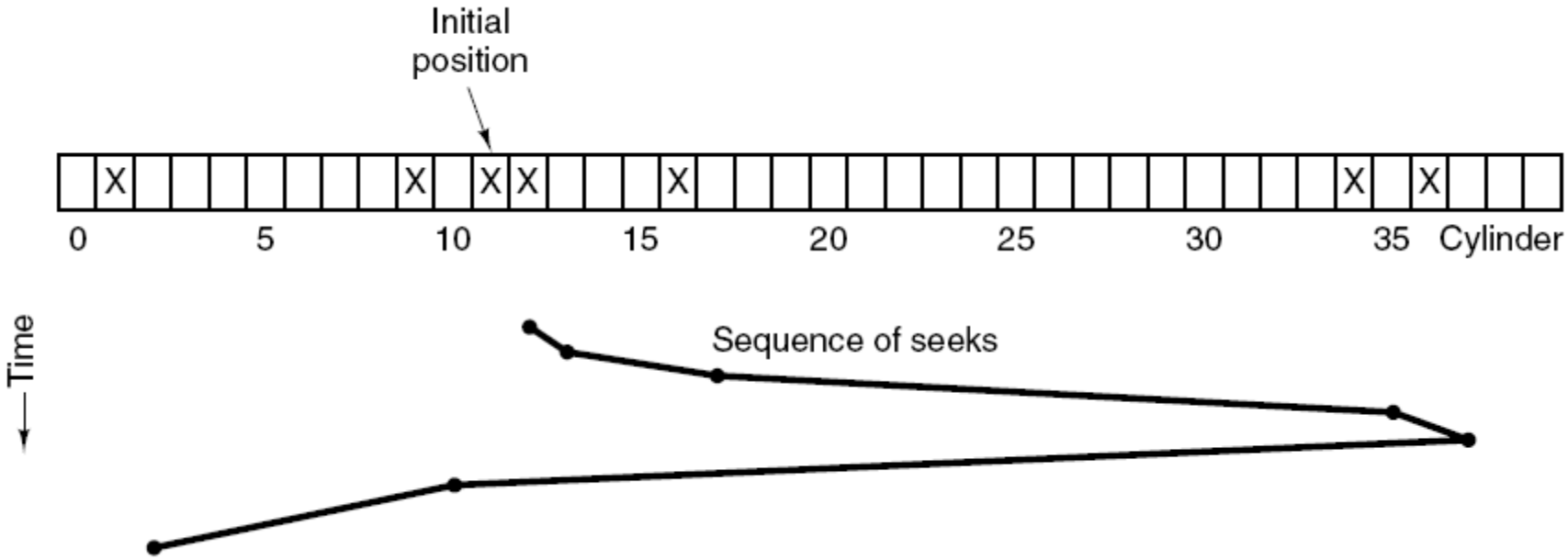
Problem with SSF

- Suppose more requests keep coming in while the requests are being processed.
  - For example, if, after going to cylinder 16, a new request for cylinder 8 is present, that request will have priority over cylinder 1. If a request for cylinder 13 then comes in, the arm will next go to 13, instead of 1.
- With a heavily loaded disk, the arm will tend to stay in the middle of the disk most of the time, so requests at either extreme will have to wait a long time
  - Requests far from the middle may get poor service.

Elevator Algorithm (SCAN)

- Keep moving in the same direction until there are no more outstanding requests in that direction, then switch directions.

SCAN Algorithm Example

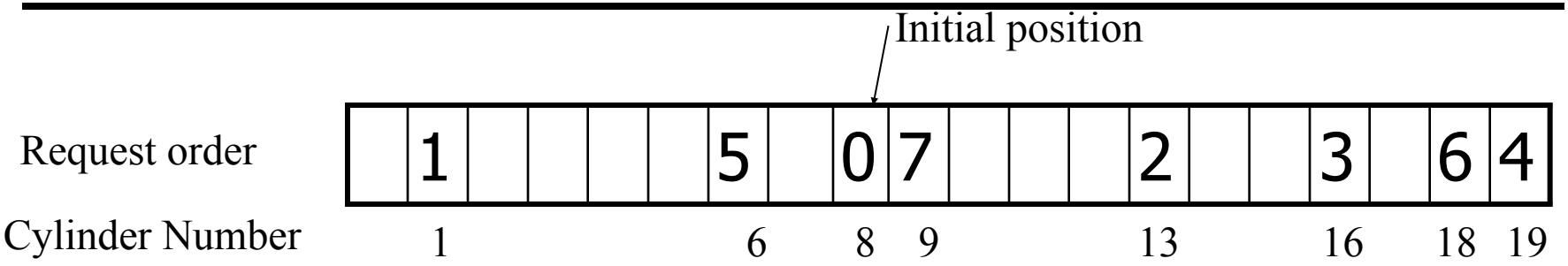


- From initial position of 11, the disk arm serves requests in the order of (12, 16, 34, 36, 9, 1) with movements of (1, 4, 18, 2, 27, 8), total of 61 cylinders

Circular SCAN

- A variant of SCAN
- Always scan in the same direction. When the highest numbered cylinder with a pending request has been serviced, the arm goes to the lowest-numbered cylinder with a pending request and then continues moving in an upward direction.
- Q: What is the upper bound of disk arm movement distance for serving one request for SCAN? For C-SCAN?
- A: both twice the number of total cylinders

Quiz



What is the sequence of servicing requests for FCFS, SSF, SCAN and C-SCAN?

FCFS: cylinder 8→1→13→16→19→6→18→9, total 59 cylinders

SSF: cylinder 8→9→6→1→13→16→18→19, total 27 cylinders

SCAN: cylinder 8→9→13→16→18→19→6→1, total 29 cylinders

Assume the direction is initially UP.

C-SCAN: cylinder 8→9→13→16→18→19→1→6, total 34 cylinders

Assume the direction is initially UP.

Exercise

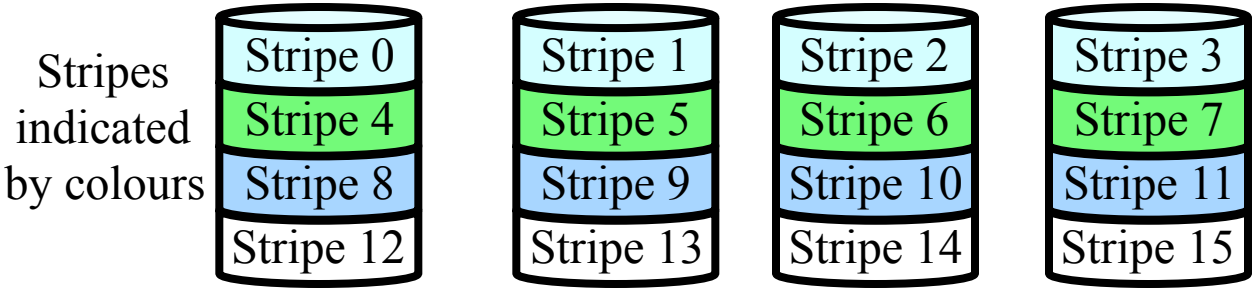
- Workout the sequence of servicing requests for FCFS, SSF and SCAN for the following order of requests (initial position of disk head is 53):
  - 98, 183, 37, 122, 14, 124, 65, 67
- Answer:
- FCFS: http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/diskschedulingfcfs.htm
- SSF: http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/diskschedulingsstf.htm
- SCAN:
- http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/diskschedulingscan.htm

RAID

- Redundant Array of Inexpensive Disks
  - A set of physical disk drives seen as a single logical drive by the system (OS)
  - Data (individual files) are distributed across multiple physical drives
    - Access can be faster, access multiple disks to get the data
    - Controller controls mapping and setup of RAID structure on the group of disks
    - OS sees the equivalent of a single disk
  - Different levels of optimization, different approaches

RAID Level 0

- Individual disk controllers are replaced by a single RAID 0 controller than simultaneously manages all disks. It is capable of simultaneously transferring from all the disks
- Each disk is divided into stripes. A stripe may be a block, a sector, or some other unit.
- When a large write to disk is requested the RAID 0 controller will break the requested data into strips. The first strip will be placed on the first disk, the second on the second disk and so on in a round robin fashion.

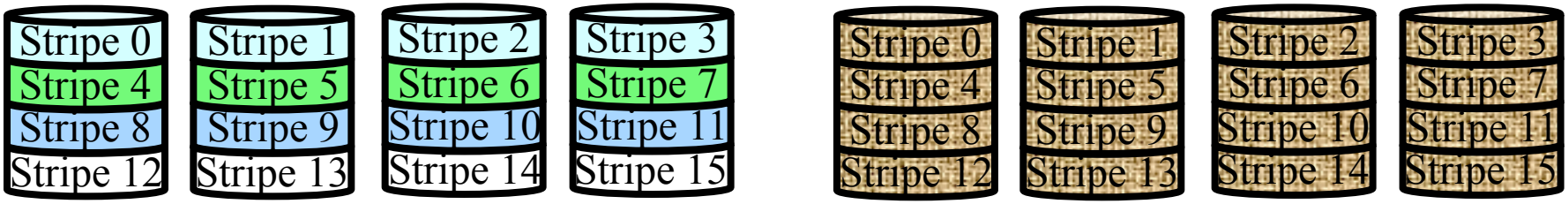


RAID Level 0

- Dividing the data between N disks allows the RAID 0 controller to read/write the data N time faster
- If two requests are pending there is a good chance they are on different disks and can be serviced simultaneously. This reduces the average time in the I/O queue
- Works best for large read/write requests
- Decreases mean time to failure over single large disk
- Also called striping, no redundancy (so not true RAID)

RAID Level 1

- All data is duplicated, each logical strip is mapped to two different disks (same data stored in the two strips).
- Each disk has a mirror disk that contains the same data copy.
- To recover from failure on one disk read the data from the mirror disk



RAID Level 1

- Each disk has a mirror disk that contains the same data.
- A read request can be serviced by either disk containing the data (choose faster of the two available reads)
- A write request requires both disks containing the data to be updated. (limited by slower or two writes)
- Expensive, requires double the storage capacity
- Useful, providing real time backup
- If the bulk of I/O requests are reads can approach double the access speed of RAID0
- (Details omitted for RAID2-6)

Summary

- Hardware Principle
  - Device controller: between devices and OS
  - Memory mapped I/O Vs. I/O port number
  - DMA vs. Interrupt
- Software Principle
  - Programmed I/O: waste CPU time
  - Interrupts: overheads
  - DMA: offload I/O from CPU

Summary (Cont.)

- Four layers of I/O software
  - Interrupt handlers: context switch, wake up driver when I/O completed
  - Device drivers: set up device registers, issue commands, check status and errors
  - Device-independent software: naming, protection, buffering, allocating
  - User-space software: make I/O call, format I/O, spooling

Summary (Cont.)

- Disks
  - Structure: cylinder → track → sector
  - Disk scheduling algorithms: FIFO, SSTF, SCAN, C-SCAN