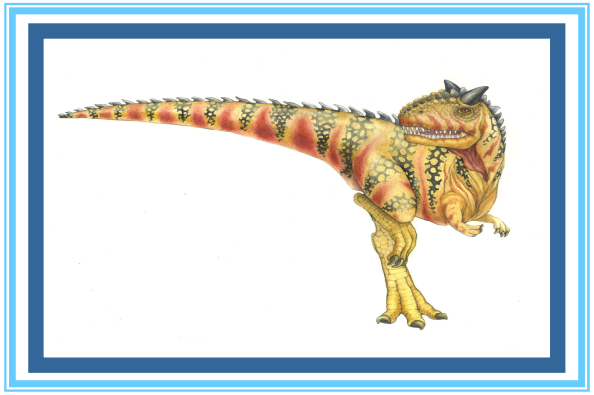
Chapter 11: I/O Systems

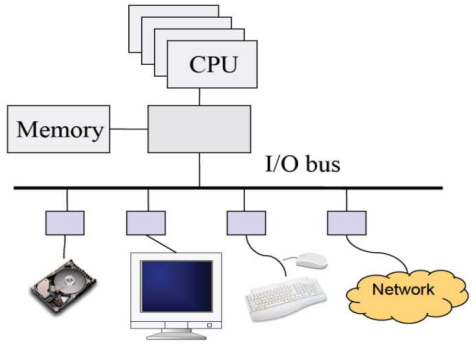
School of Computing, Gachon Univ.
Jungchan Cho

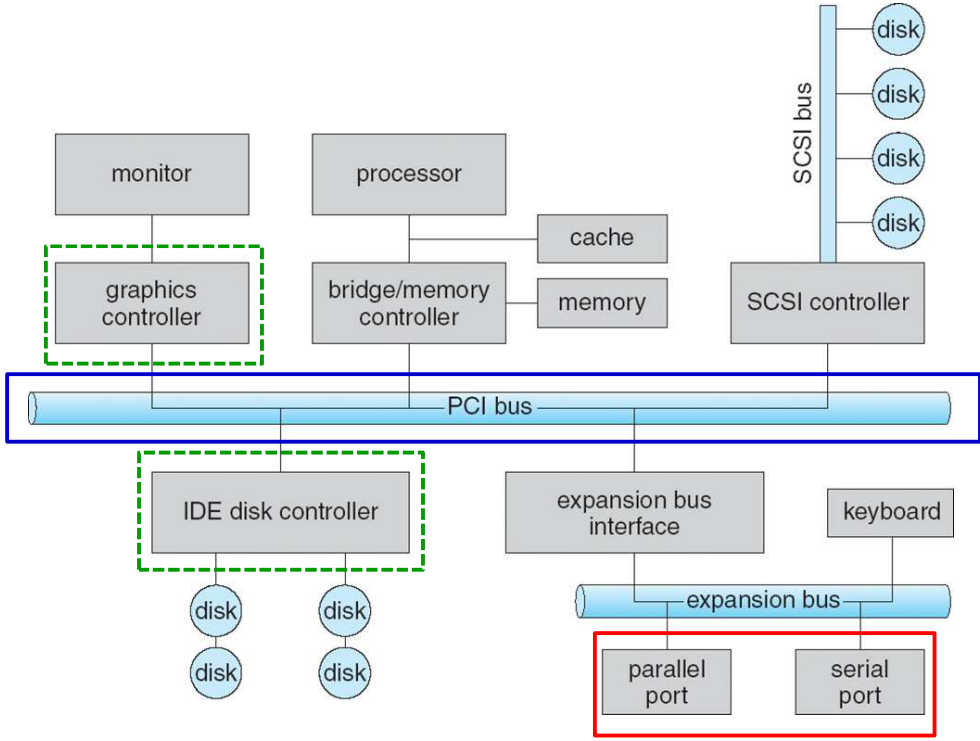


Most slides from "Operating System Concepts – 10th Edition".
Many slides are taken from lecture notes of Prof. Joon Yoo.

I/O Hardware

- Incredible variety (a large number) of I/O devices
  - Storage (e.g., disks), Transmission (e.g., network, Bluetooth), Human-interface (e.g., keyboard, mouse, monitor), …
- Basic I/O hardware
  - **I/O Device**
  - **Port**
    - connection point for device (e.g., serial port)
    - typically consists of device **registers**
  - **Bus**
    - Shared wires that transfer data between components inside computer
  - **Device Controller** (Host adaptor)





Device Controller

- **Device controller**
  - Collection of electronics (e.g., processor, register/memory) that can operate on port, a bus, or a device
  - Sometimes integrated: Simple, single chip
    - e.g., Serial-port controller (for mouse, keyboard)
  - Sometimes separate circuit board
    - e.g., SCSI controller

- How does processor give commands to I/O devices?
  - **Device controllers** usually have **registers** where OS (device driver) places commands, addresses, and data to write, or read data from registers after command execution

How does processor communicate with I/O?

- Techniques for Performing I/O
  - **Direct I/O instructions**
    - CPU issues I/O command by directly writing instructions into device's registers
    - CPU busy waits for completion (polling)
  - **Memory Mapped I/O**
    - Device-control registers are mapped into memory address space
    - CPU executes I/O requests by read/write at their mapped locations in memory
  - **Direct Memory Access (DMA)**
    - CPU asks DMA controller to perform device-to-memory transfer
    - DMA controller issues I/O command to device controller and transfers into memory
    - CPU module is interrupted after completion

I/O Hardware Communication Problems

- **How to send command/data to controller?**
  - **Direct (Port-mapped) I/O vs. Memory mapped I/O**
- How do we know I/O is ready?
  - Polling vs. Interrupt-driven I/O
- Lot's of I/O data
  - Programmed I/O vs. Direct Memory Access (DMA)

Direct I/O (port-mapped I/O)

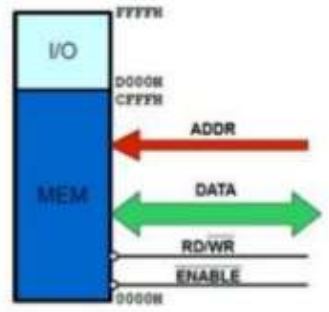
- **Direct I/O** instructions (I/O-mapped or port-mapped I/O)
  - Each **device controller register** is assigned a port number address **PORT** by OS
  - Use **special I/O instructions** to *directly* transfer control/data via control line (bus) to **PORT**

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)

Device I/O PORT Locations

- Direct I/O instructions can be slow
  - CPU wants to write millions of bytes to graphical card I/O
  - Must issue millions of special I/O instructions

Memory Mapped I/O



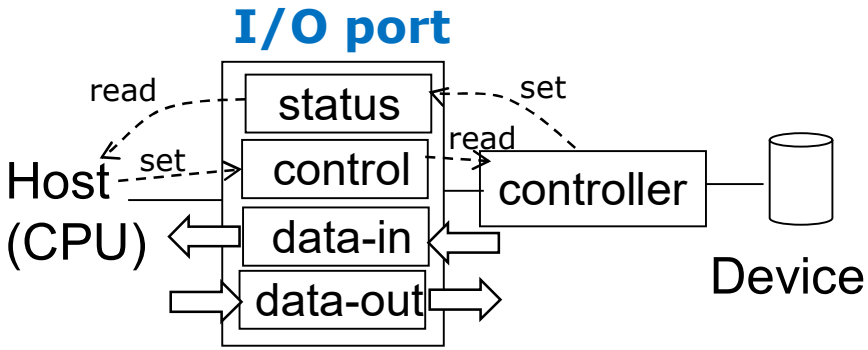
- **Memory-mapped I/O**
  - Device **command registers** and **memory** are mapped to certain kernel **memory address space**
  - CPU executes I/O requests to read and write the device control registers at the **memory space** (mapped to I/O)
  - **Faster** especially for large address spaces (e.g., graphics)
    - Better to write millions of bytes to graphics memory (Memory-mapped I/O) than to issue millions of special I/O instructions (Direct I/O)

I/O Hardware Problems

- How to send command/data to controller?
  - Direct (Port-mapped) I/O vs. Memory mapped I/O
- **How do we know I/O device is ready?**
  - **Polling vs. Interrupt-driven I/O**
- Lot's of I/O data
  - Programmed I/O vs. Direct Memory Access (DMA)

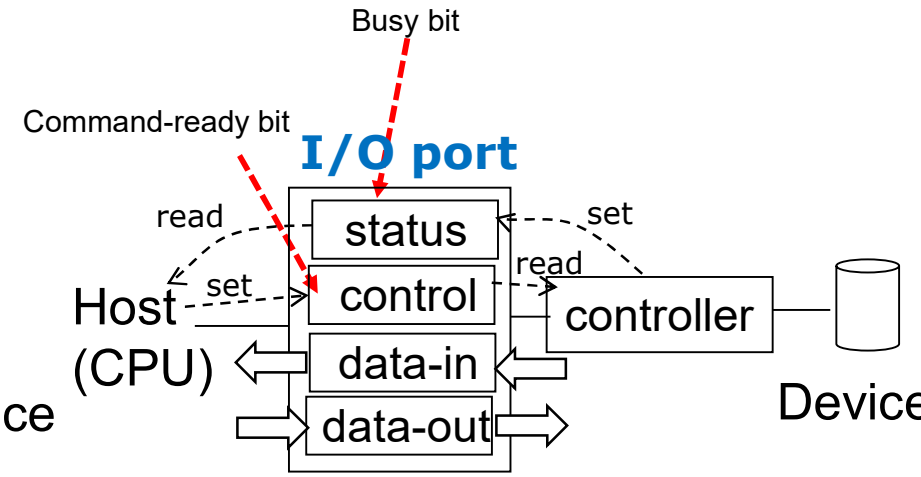
I/O Port Registers

- I/O Port Registers (1-4 bytes)
  - **Status** register
    - **busy-bit**: device is idle(0) or busy(1)
    - byte is available to *read from data-in*
    - device error has occurred
  - **Control** register
    - **command-ready bit**: ready (1) or not ready (0) to *start a command*
    - *change mode* of a device
  - **Data-in** register ⇦
    - Host *reads* to get input from I/O
  - **Data-out** register ⇨
    - Host *writes* to send output to I/O



Polling

- Polling summary: CPU is *active*, I/O device is *passive*
- **Busy bit** in **status** register
  - set (1): device is busy working
  - clear (0): device is ready to accept next command
- **Command-ready bit** in **control** register
  - Host sets command-ready bit when command is available for controller to execute
- **Busy-wait** (**Polling**) cycle to wait for I/O from device
  - 1. CPU requests I/O operation
    - If busy bit is set (1), repeatedly check until **busy bit** is clear (0)
  - 2. If **busy bit** is clear (0), I/O controller performs operation
    - host writes byte to **data-out register**, then sets (1) **command-ready bit**
    - I/O controller notices **command-ready bit** is set (1) and then sets (1) status **busy bit**, then reads data-out register and does I/O to device
  - 3. I/O complete
    - I/O controller clears (0) **command-ready bit**, clears (0) **busy bit**

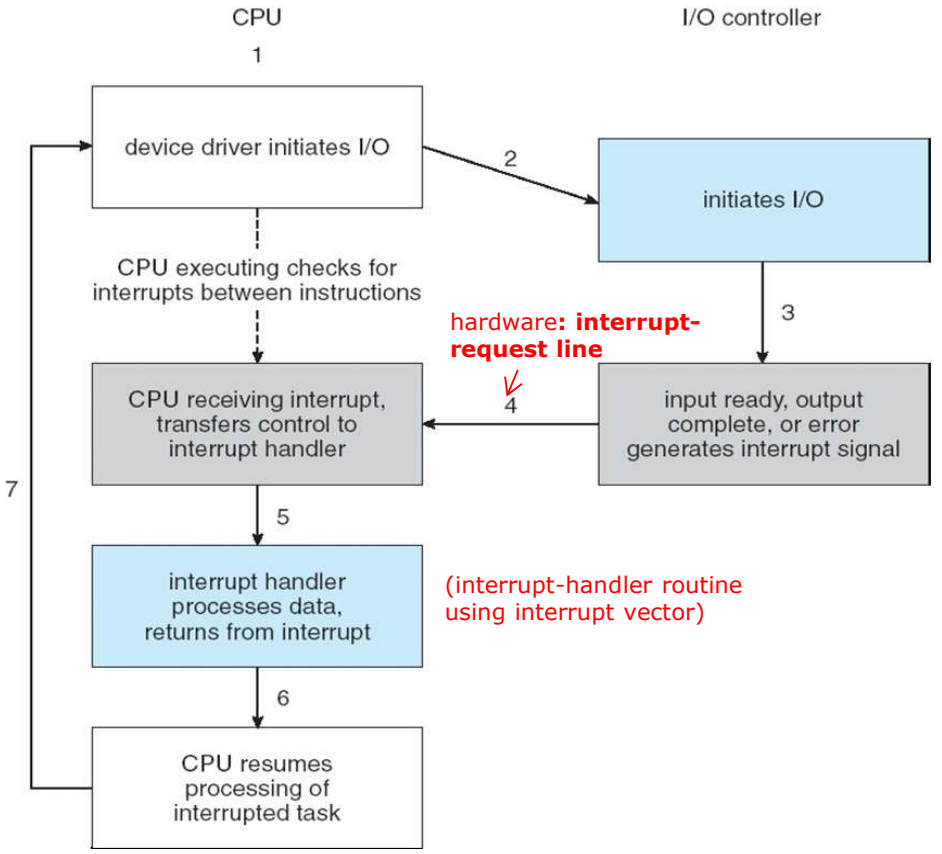


Polling

- Simple to implement if I/O is fast (e.g., character devices)
- But if CPU has to wait too long?
  - Busy-waiting loop; may incur low CPU utilization. Better to switch to another job.
- May be efficient to arrange hardware controller to notify CPU when the device becomes ready
  - CPU can now do other jobs while waiting
  - This notification is called **Interrupt**

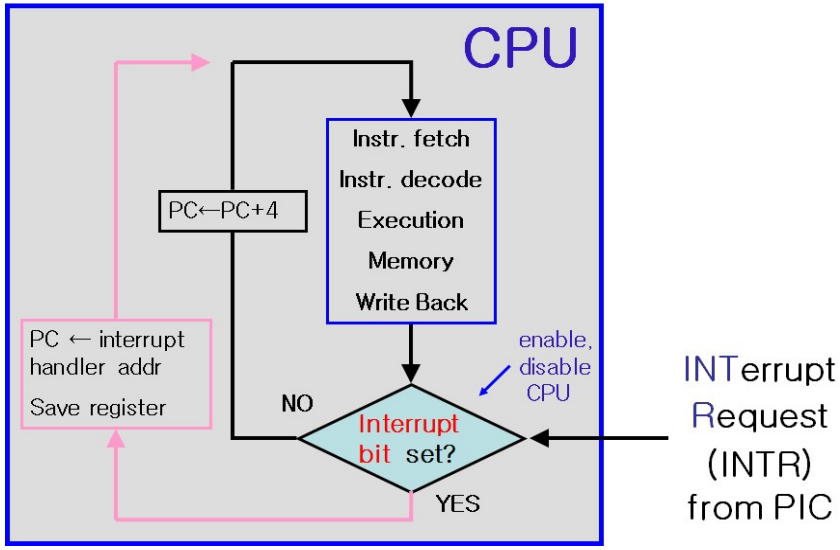
Interrupt-Driven I/O Cycle

- **Interrupt** summary: CPU is *passive*, I/O device is *active*
- CPU issues command to device, gets interrupt on completion on a wire called **interrupt-request line**
  - Interrupt-request line is connected to CPU hardware
  - When CPU detects that a controller has asserted a signal on the interrupt-request line, then CPU performs **interrupt handler routine**.
- Approach
  1. CPU issues read command (via device driver)
  2,3. I/O controller gets data **while CPU does other work**
  4. I/O complete. **I/O controller interrupts CPU**
  5. Interrupt handler process data using interrupt vector
  6, 7. CPU resumes



Interrupts

- CPU hardware has a **Interrupt request line** triggered by I/O device
- At boot time, OS probes hardware buses to determine what devices are present – install corresponding **interrupt handlers** into **interrupt vector**
- At run time, CPU receives interrupts - **Interrupt vector** dispatches interrupt to correct **Interrupt handler**



- Interrupt mechanism used for
  - **I/O device controllers**: e.g., output has completed, input data are available, failure has been detected.
  - **exceptions (traps):** e.g., divide-by-zero, accessing protected memory space, attempt to execute privileged instruction from user mode

Intel Pentium Processor Event-Vector Table

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

By device controller

Interrupt Handling Example: Page Fault

- Page fault for virtual memory access
  - Process references a page with invalid bit (trap)
  - Interrupt occurs (via interrupt-request line to CPU)
    - suspend current process, check interrupt vector, and jump to page-fault interrupt handler in kernel
  - **(page fault) Interrupt handler**
    - saves state of process (in _____)
    - moves the process to waiting queue (process now in _____ state)
    - schedules a disk I/O operation to fetch page
    - schedules another process to resume execution
  - Return from interrupt

Interrupt Handling Example: System Call

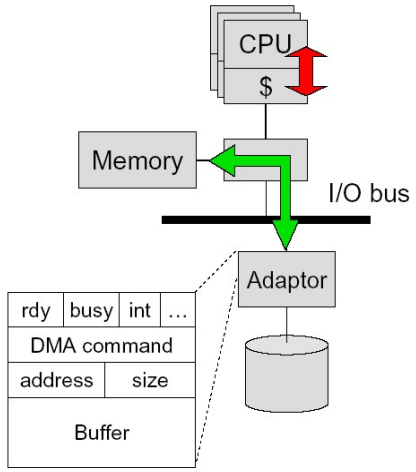
- System call executes (trap) to trigger kernel to execute request
  - Program library (API) issues system call
  - process executes software interrupt (trap) to execute desired service
  - **Interrupt handler**
    - saves state of user process
    - switch to kernel mode
    - dispatch kernel routine to run desired service

I/O Hardware Problems

- How to send command/data to controller?
  - Direct (Port-mapped) I/O vs. Memory mapped I/O
- How do we know I/O is ready?
  - Polling vs. Interrupt-driven I/O
- **Lot's of I/O data**
  - **Programmed I/O vs. Direct Memory Access (DMA)**

Direct Memory Access (DMA)

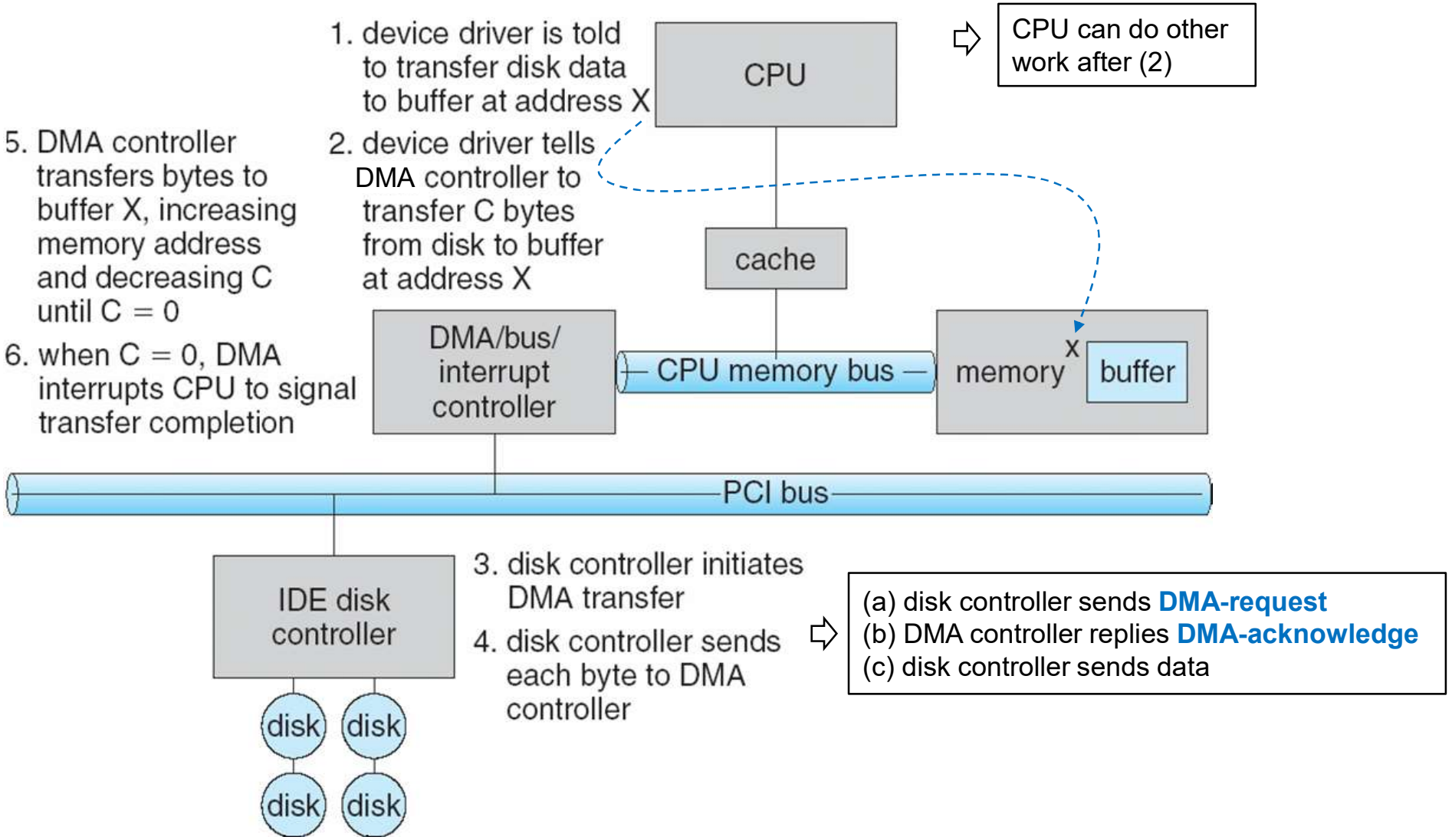
- **Programmed I/O (PIO)**
  - CPU commands to transmit data one byte at a time
    - e.g., polling – CPU watch status bit and feed data to register one byte at a time



- **Direct Memory Access (DMA)**
  - Involves special hardware element: **DMA controller**
  - Avoids PIO: Bypasses CPU to transfer data ***directly*** between I/O device and memory – Less burden for CPU

- More efficient than CPU requesting data from I/O controller byte by byte, e.g., for **block devices** like **disks** or **network interfaces**
  - CPU can do other work after sending command to DMA controller
  - DMA controller interrupts CPU after completing block transfer

Direct Memory Access (DMA)



Application I/O Interface

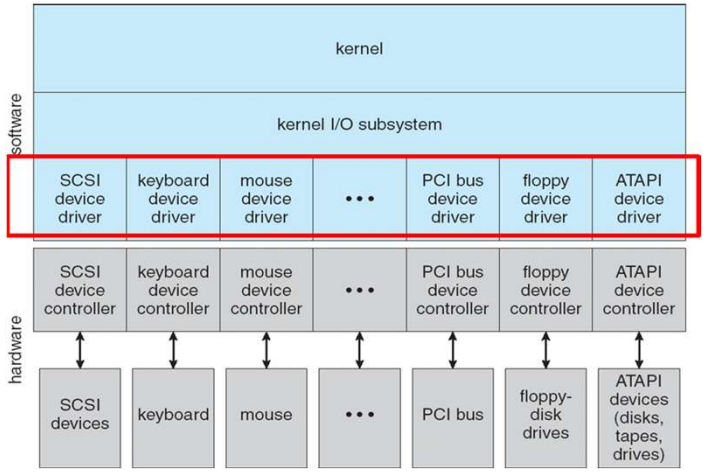
- **Wide variety of peripherals (external devices)**
  - Delivering different amounts of data
  - At different speeds
  - In different formats

  - Storage devices
    - Disk, tapes
  - Transmission/Communication devices
    - Network card, modem
  - Human interface devices
    - Screen, keyboard, mouse
  - Specialized devices
    - Joystick

- Question:
  - There are wide variety of I/O devices. How does OS deal with this complexity?

Device Drivers

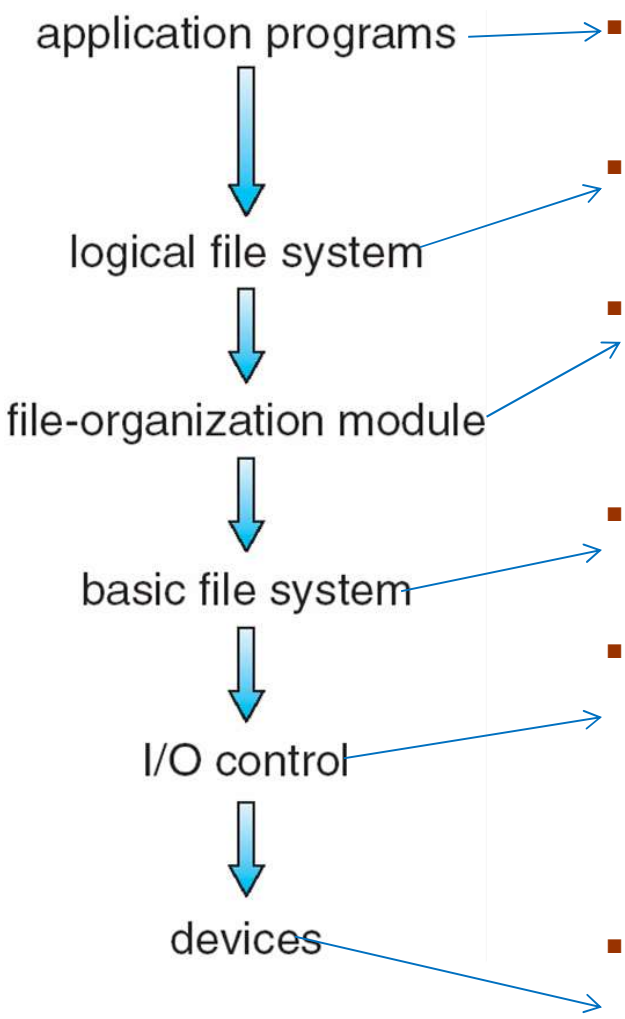
- **Device Driver**: Device-specific software code to control an IO device
  - Custom-tailored to specific devices (e.g., graphic card, network, mouse)
  - Export one of the standard interfaces to the OS (e.g., Windows, Linux, MAC)
  - **Main purpose:** Hide the difference among device controllers from the I/O subsystem of the kernel



Device Drivers

- **Benefits**
  - Makes I/O subsystem independent of hardware
  - Simplifies job of OS developer – simple interface to I/O
  - Benefits hardware manufacturers – why?
- **Example:** Hardware manufacturer designs new device
  - Device drivers are typically written by device's manufacturer
  - Option 1: New devices can be compatible with existing host controller/driver interface
  - Option 2: Write new device drivers to interface the new hardware to OS
  - Each type of OS has its own standards – need multiple device drivers to support various OSes (e.g., Windows, Linux, Mac, ...)

Example: File I/O



- Read file “foo.c”
- Search directory (metadata), FCB
- Translate file/pointer into physical block#
  - “foo.c” → block#313
- Send command to device driver
  - read block#313
- **Device driver**
  - Translate high-level commands (“read block #313”) to low-level commands (hardware-specific instructions; “read track#13 sector#11”)
  - Send command to device controller
- **Device controller**
  - Execute hardware I/O command

Summary

- **How does processor communicate with I/O?**
  - Techniques for Performing I/O
    - **Direct I/O instructions**
    - **Memory Mapped I/O**
    - **Direct Memory Access (DMA)**

- **How to send command/data to controller?**
  - Direct (Port-mapped) I/O vs. Memory mapped I/O

- **How do we know I/O is ready?**
  - Polling vs. Interrupt-driven I/O

- **Lot's of I/O data**
  - Programmed I/O vs. Direct Memory Access (DMA)

- Appendix
  - Application I/O Interface
    - Block and Character Devices
    - Network Devices (Socket)
  - Kernel I/O Subsystem

Application I/O Interface

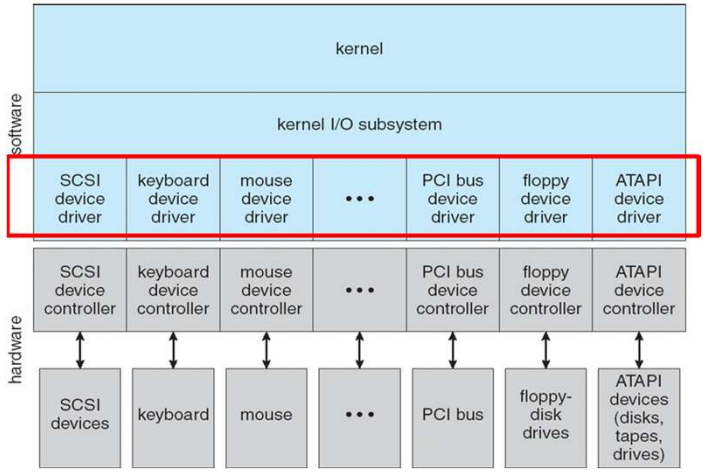
- **Wide variety of peripherals (external devices)**
  - Delivering different amounts of data
  - At different speeds
  - In different formats

  - Storage devices
    - Disk, tapes
  - Transmission/Communication devices
    - Network card, modem
  - Human interface devices
    - Screen, keyboard, mouse
  - Specialized devices
    - Joystick

- Question:
  - There are wide variety of I/O devices. How does OS deal with this complexity?

Device Drivers

- Devices are grouped into a few conventional types:
  - e.g.,
  - block I/O,
  - character-stream I/O,
  - memory-mapped file access, network sockets,
  - graphical display,
  - audio, ...



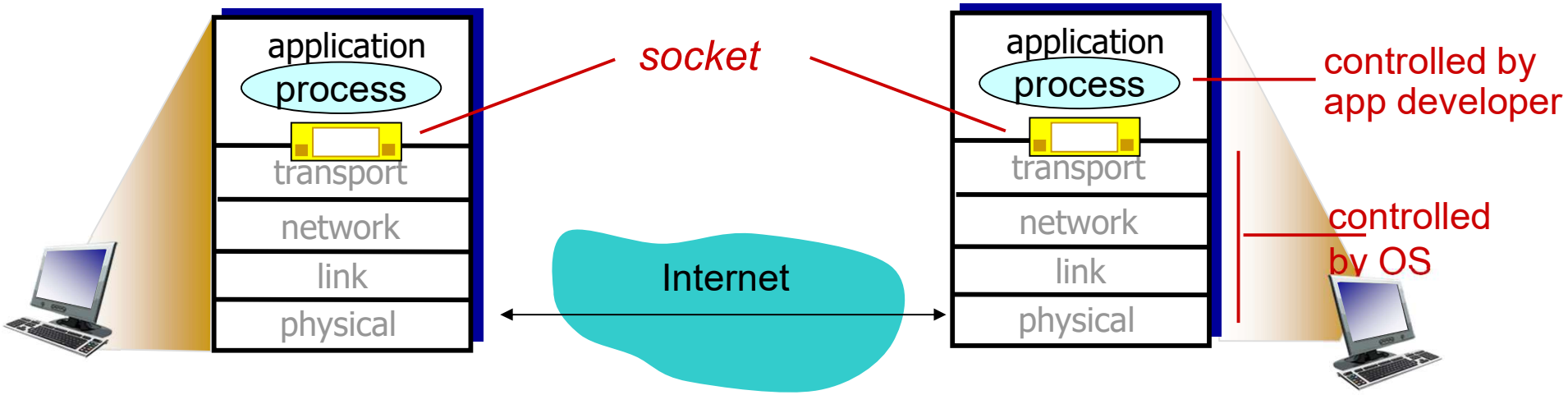
Block and Character Devices

- **Character devices** typically transfers **Bytes** one by one (e.g., keyboards, mice, serial ports)
  - produce data for input “spontaneously”

- **Block devices** transfers **Block** of bytes, (e.g., disk drives)
  - Commands include read, write, seek
  - **Memory-mapped file** access possible on block-device drivers
    - File mapped to virtual memory and clusters brought via demand paging
  - **Direct Memory Access (DMA)**

Network Devices (Socket)

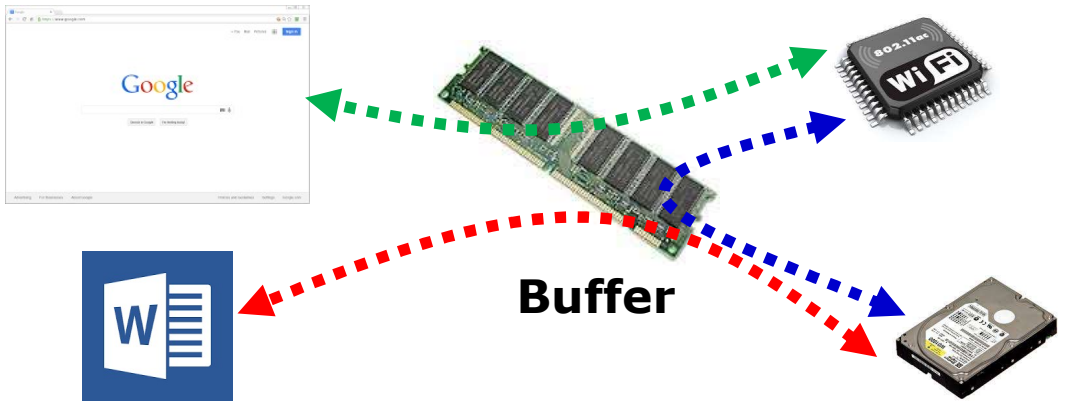
- UNIX and Windows include **socket** interface via system call
  - Separates network protocol (e.g., TCP/IP, Wi-Fi, Ethernet) from network operation
  - Create a socket, connect a local socket to a remote address, send/receive packets over the connection



Kernel I/O Subsystem

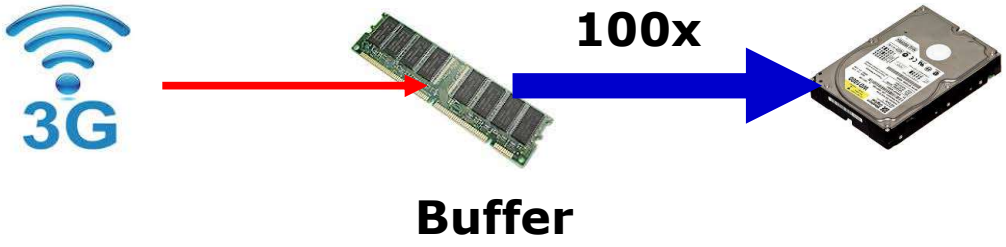
- **Scheduling**
  - I/O scheduling can improve overall system performance
    - e.g., Disk scheduling (Ch 11.2)

- **Buffering**
  - **Buffer:** memory area that stores data being transferred between two devices or between a device and application



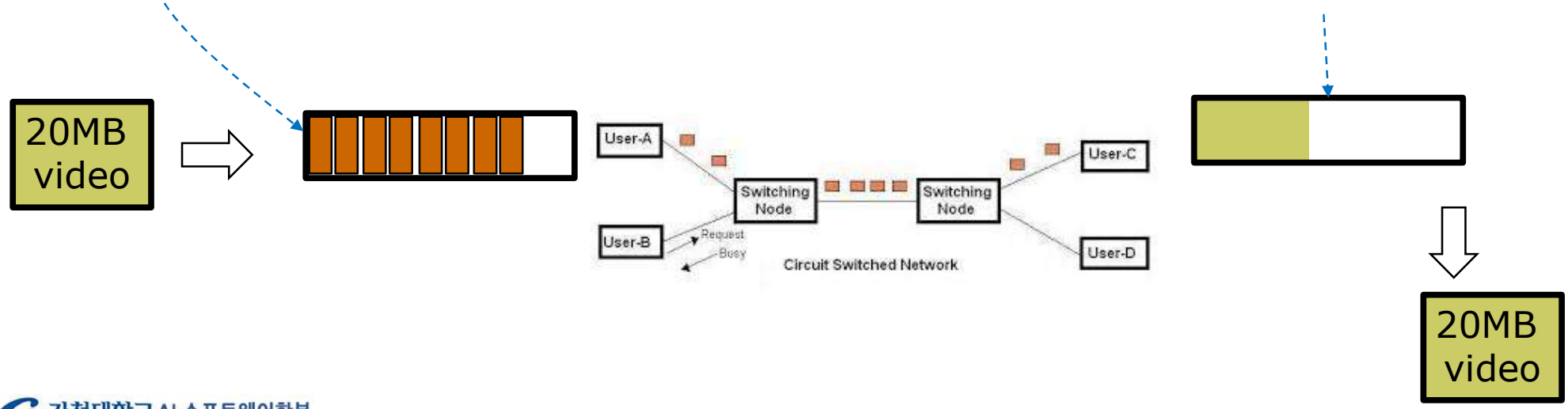
Kernel I/O Subsystem

- Reasons for using buffering (1): Cope with speed mismatch between the producer and consumer of a data stream
  - e.g., Slow network (e.g., 3G tethering) vs. faster hard disk
  - File is being received via 3G tethering for storage on hard disk. Say hard disk is 100x faster than 3G network
  - Buffer is created in main memory to accumulate bytes received from 3G – after 3G fills the buffer, disk write is requested



Kernel I/O Subsystem

- Reasons for using buffering (2): Provide adaptations for devices that have different data-transfer size (e.g., computer networks)
  - At the sending side, a large message (e.g., 20MB Video) is fragmented into small network packets (e.g., 1KB packets) and stored in **send buffer**
  - Packets are sent over the network (e.g., 20,000 packets)
  - At the receiving side, the packets are placed in **reassembly buffer** to form the original source file



Life Cycle of An I/O Request

Read Ch 12.5