Operating System: Chap13 I/O Systems

National Tsing Hua University

2022, Fall Semester

Outline

- Overview
- I/O Hardware
- I/O Methods
- Kernel I/O Subsystem
- Performance
- Application Interface

Overview

- The two main jobs of a computer
  - **I/O** and **Computation**
- **I/O devices**: tape, HD, mouse, joystick, network card, screen, flash disks, etc
- **I/O subsystem**: the methods to control all I/O devices
- Two conflicting trends
  - Standardization of HW/SW interfaces
  - Board variety of I/O devices

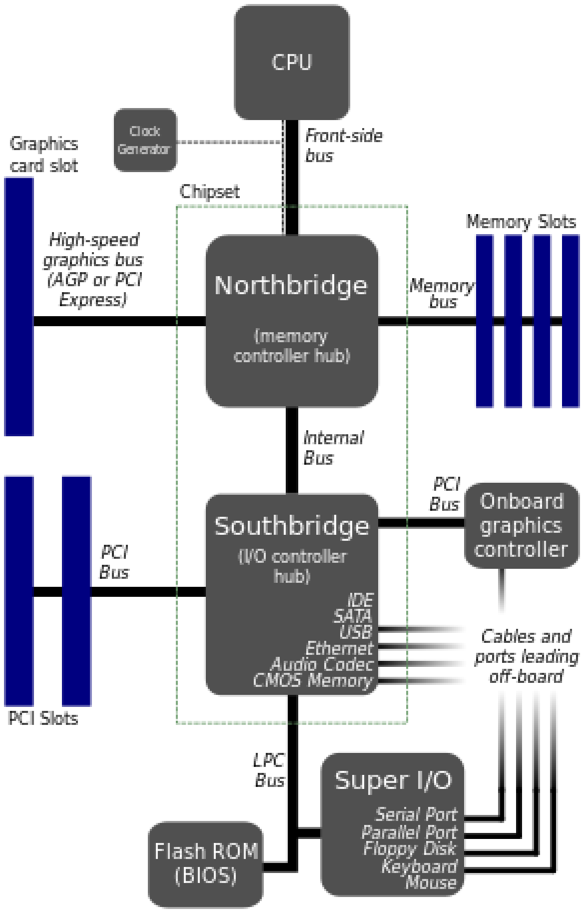
Overview

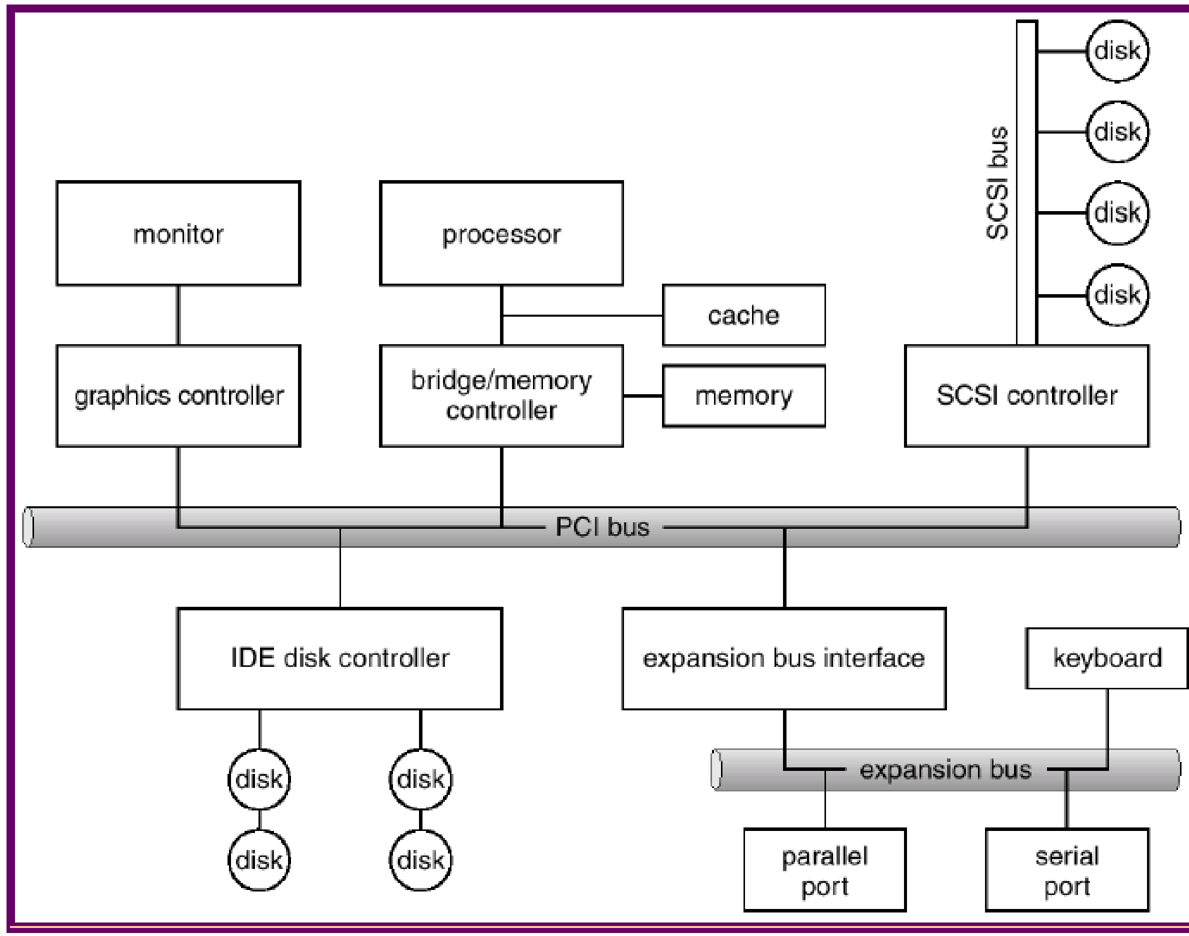
- **Device drivers**: a uniform device-access **interface** to the I/O subsystem
  - Similar to system calls between apps and OS
- Device categories
  - Storage devices: disks, tapes
  - Transmission devices: network cards, modems
  - Human-interface devices: keyboard, screen, mouse
  - Specialized devices: joystick, touchpad

I/O Hardware

- **Port**: A connection point between I/O devices and the host
  - E.g.: USB ports
- **Bus**: A set of wires and a well-defined protocol that specifies messages sent over the wires
  - E.g.: PCI bus
- **Controller**: A collection of electronics that can operate a port, a bus, or a device
  - A controller could have its own processor, memory, etc. (E.g.: SCSI controller)

Typical PC Bus Structure





Basic I/O Method (Port-mapped I/O)

- Each I/O port (device) is identified by a unique port address
- Each I/O port consists of four registers (1~4Bytes)
  - **Data-in register**: read by the host to get input
  - **Data-out register**: written by the host to send output
  - **Status register**: read by the host to check I/O status
  - **Control register**: written by the host to control the device
- Program interact with an I/O port through **special I/O instructions** (different from mem. access)
  - X86: IN, OUT

Device I/O Port Locations on PCs (partial)

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)

I/O Methods Categorization

- Depending on how to address a device:
  - **Port-mapped I/O**
    - Use different address space from memory
    - Access by special I/O instruction (e.g. IN, OUT)
  - **Memory-mapped I/O**
    - Reserve specific memory space for device
    - Access by standard data-transfer instruction (e.g. MOV)
    - ☺ More efficient for large memory I/O (e.g. graphic card)
    - ☹ Vulnerable to accidental modification, error

I/O Methods Categorization

- Depending on how to interact with a device:
  - **Poll** (busy-waiting): processor periodically check status register of a device
  - **Interrupt**: device notify processor of its completion
- Depending on who to control the transfer:
  - **Programmed I/O**: transfer controlled by CPU
  - **Direct memory access** (DMA) I/O: controlled by **DMA controller** (a special purpose controller)
    - Design for large data transfer
    - Commonly used with memory-mapped I/O and interrupt I/O method

Interrupt Vector Table

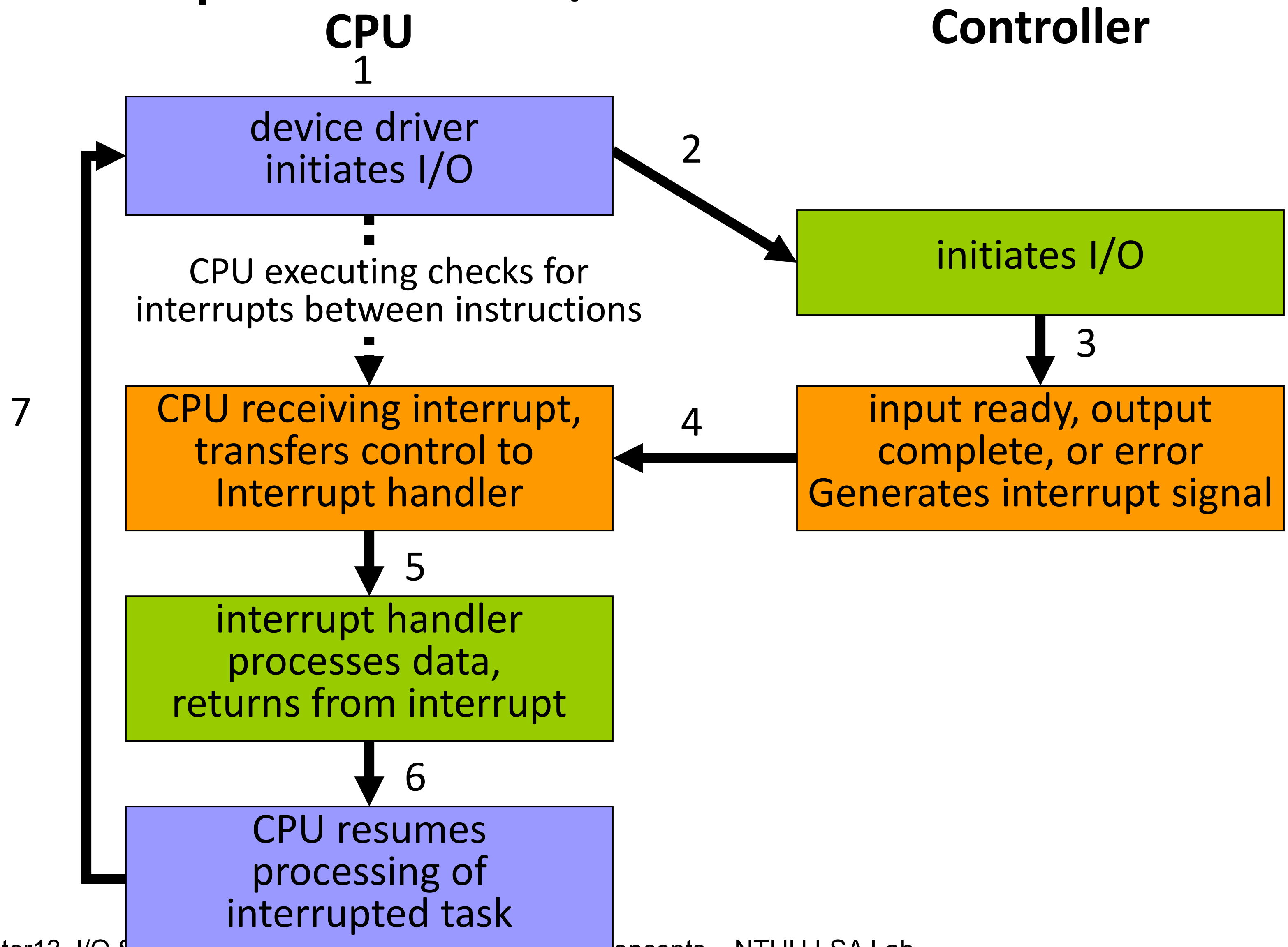
- Intel Pentium Processor:

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
19Ð31	(Intel reserved, do not use)
32Ð255	maskable interrupts

Interrupt Prioritization

- Maskable interrupt: interrupt with priority lower than current priority is not recognized until pending interrupt is complete
- Non-maskable interrupt (NMI): highest-priority, never masked
  - Often used for power-down, memory error

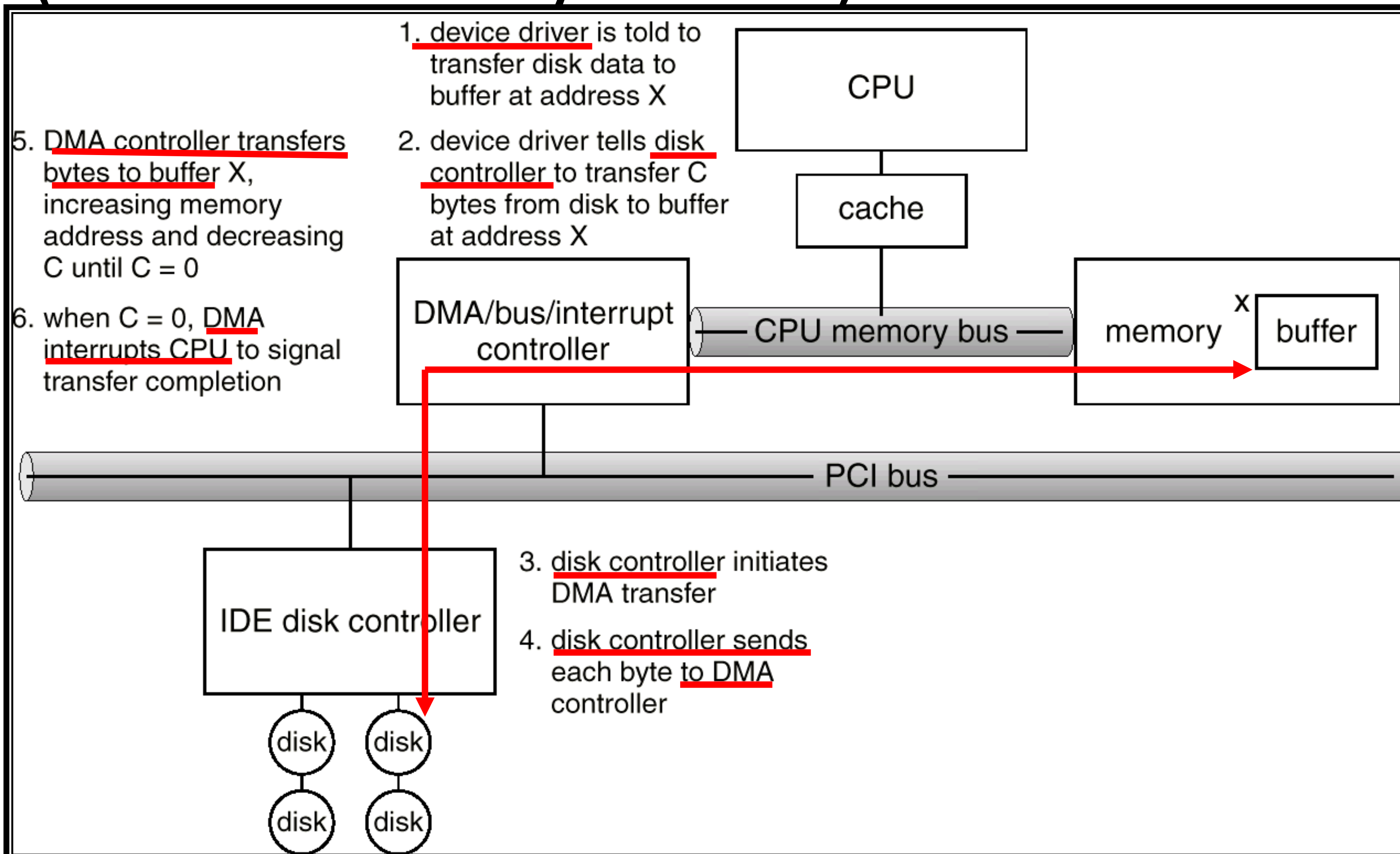
Interrupt-Driven I/O



CPU and device Interrupt handshake

1. Device asserts **interrupt request (IRQ)**
2. CPU checks the **interrupt request line** at the beginning of each instruction cycle
3. Save the status and address of interrupted process
4. CPU acknowledges the interrupt and search the **interrupt vector** table for interrupt handler routines
5. CPU fetches the next instruction from the **interrupt handler routine**
6. Restore interrupted process after executing interrupt handler routine

Six-Step Process to Perform DMA (Direct Memory Access)



Review Slides (I)

- Definition of I/O port? Bus? Controller?
- I/O device and CPU communication?
  - Port-mapped vs. Memory-mapped
  - Poll vs. Interrupt
  - Programmed I/O vs. DMA
- Steps to handle an interrupt I/O and DMA request?

Kernel I/O Subsystem

I/O Subsystem

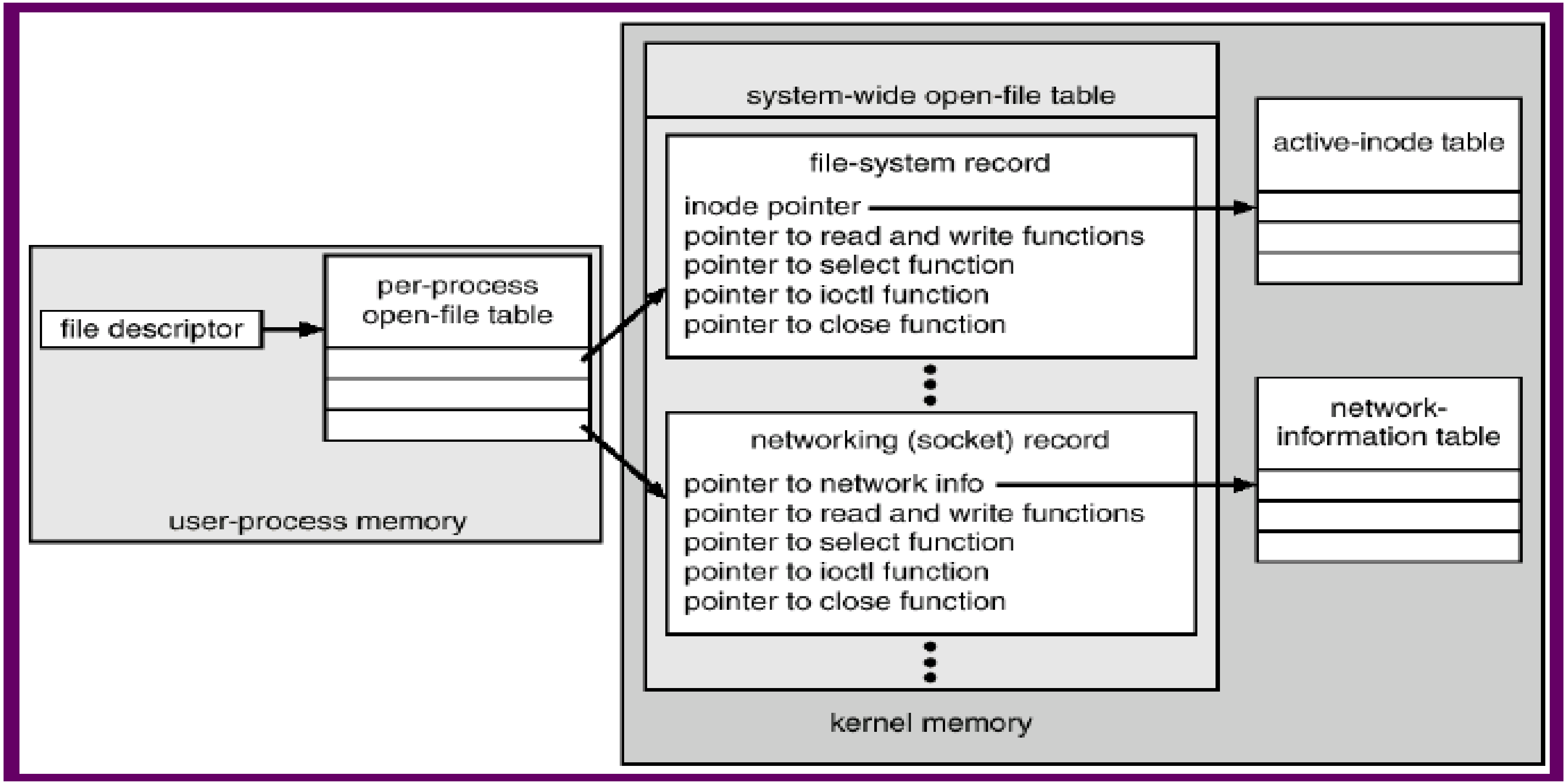
- **I/O Scheduling** – improve system performance by ordering the jobs in I/O queue
  - e.g. disk I/O order scheduling
- **Buffering** – store data in memory while transferring between I/O devices
  - Speed mismatch between devices
  - Devices with different data-transfer sizes
  - Support copy semantics

I/O Subsystem

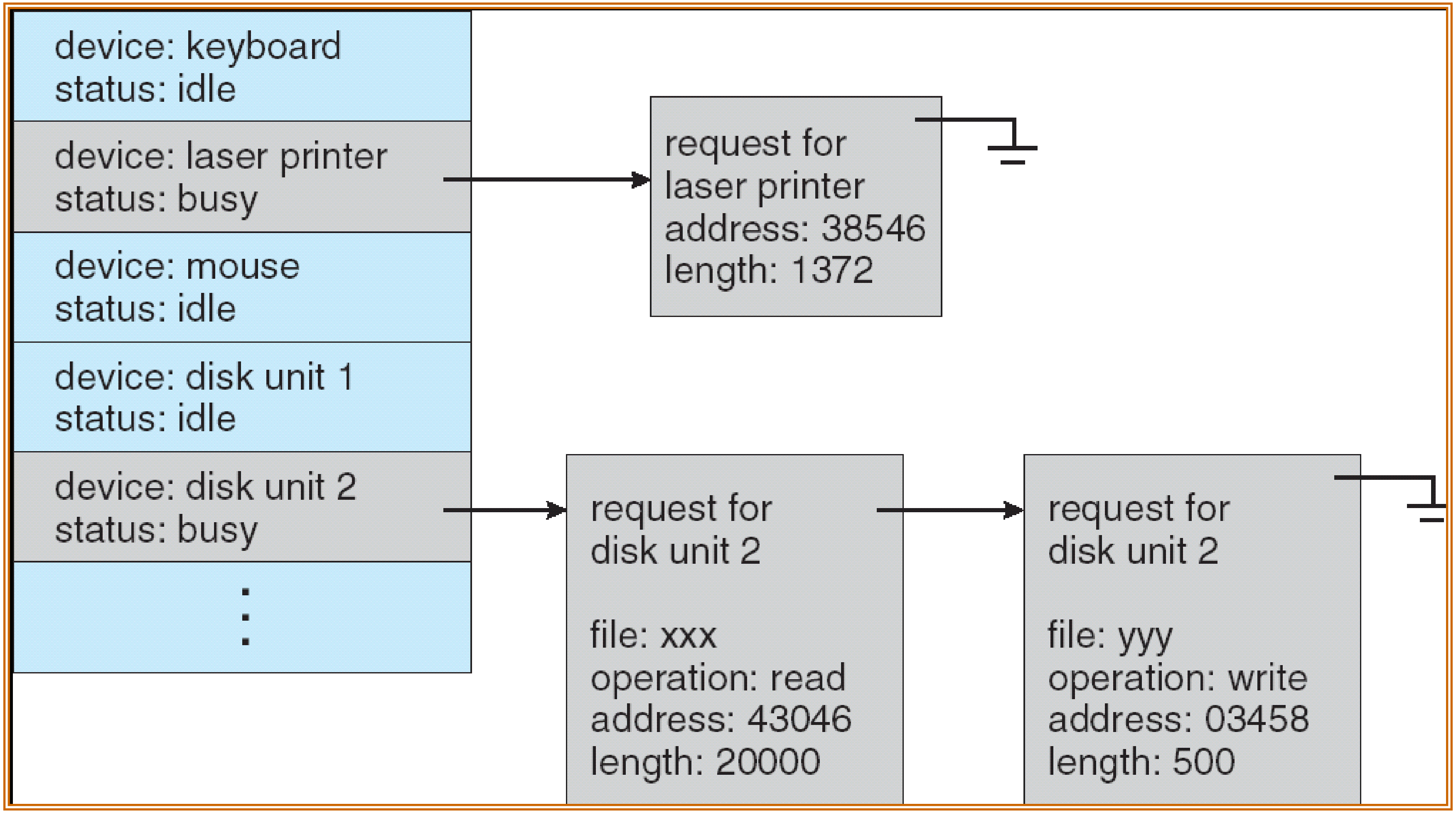
- **Caching** – fast memory that holds copies of data
  - Always just a copy
  - Key to performance
- **Spooling** – holds output for a **device**
  - e.g. printing (cannot accept interleaved files)
- **Error handling** – when I/O error happens
  - e.g. SCSI devices returns error information
- **I/O protection**
  - Privileged instructions

UNIX I/O Kernel Data Structure

- Linux treats all I/O devices like a file



Device-status Table



Blocking and Nonblocking I/O

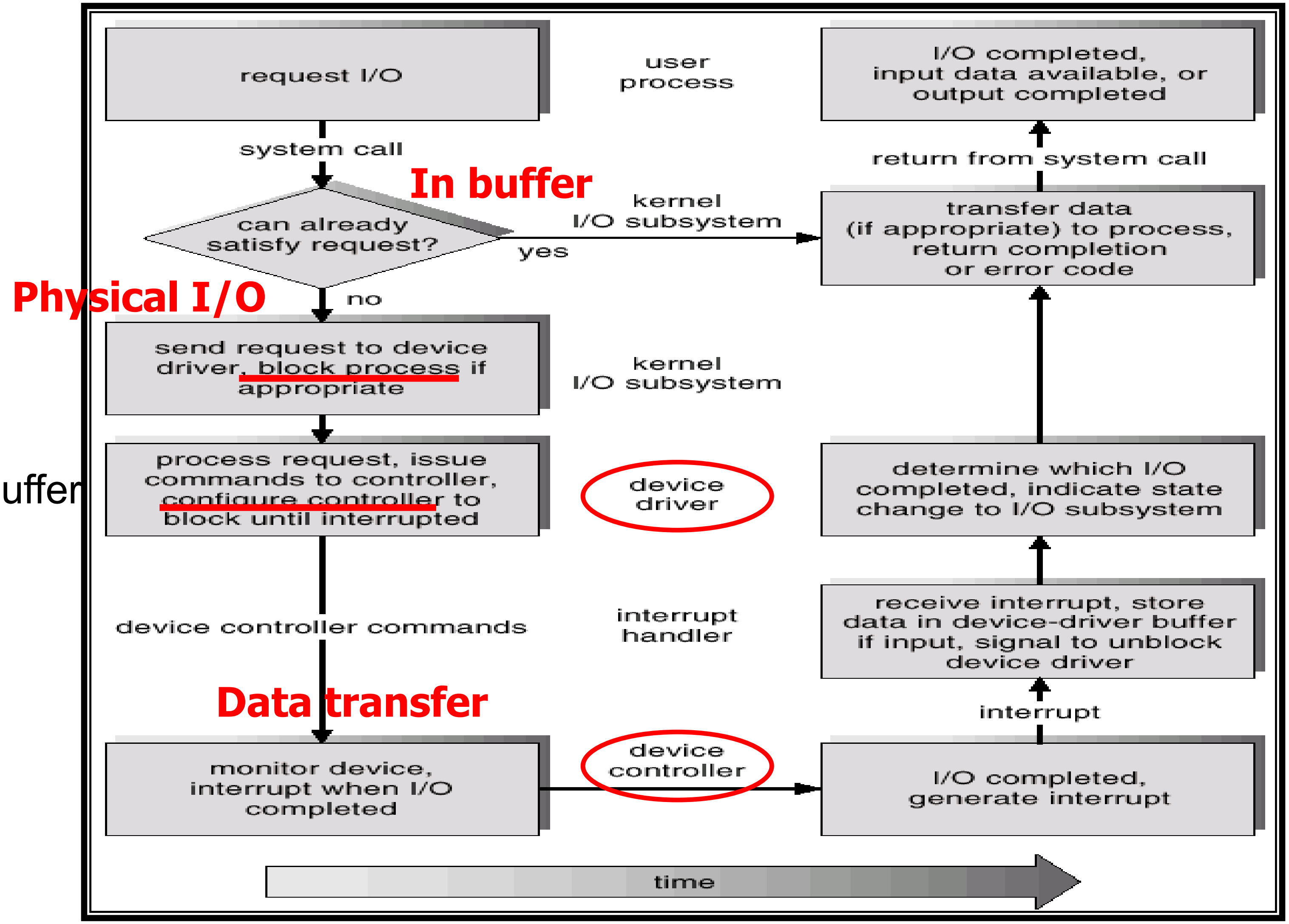
- Blocking - process suspended until I/O completed
  - Easy to use and understand
  - Insufficient for some needs
  - Use for **synchronous** communication & I/O
- Nonblocking
  - Implemented via multi-threading
  - Returns quickly with count of bytes read or written
  - Use for **asynchronous** communication & I/O

Life Cycle of An I/O Request

Check buffer cache

Move process from run queue to wait queue

Allocate kernel buffer
Schedule I/O

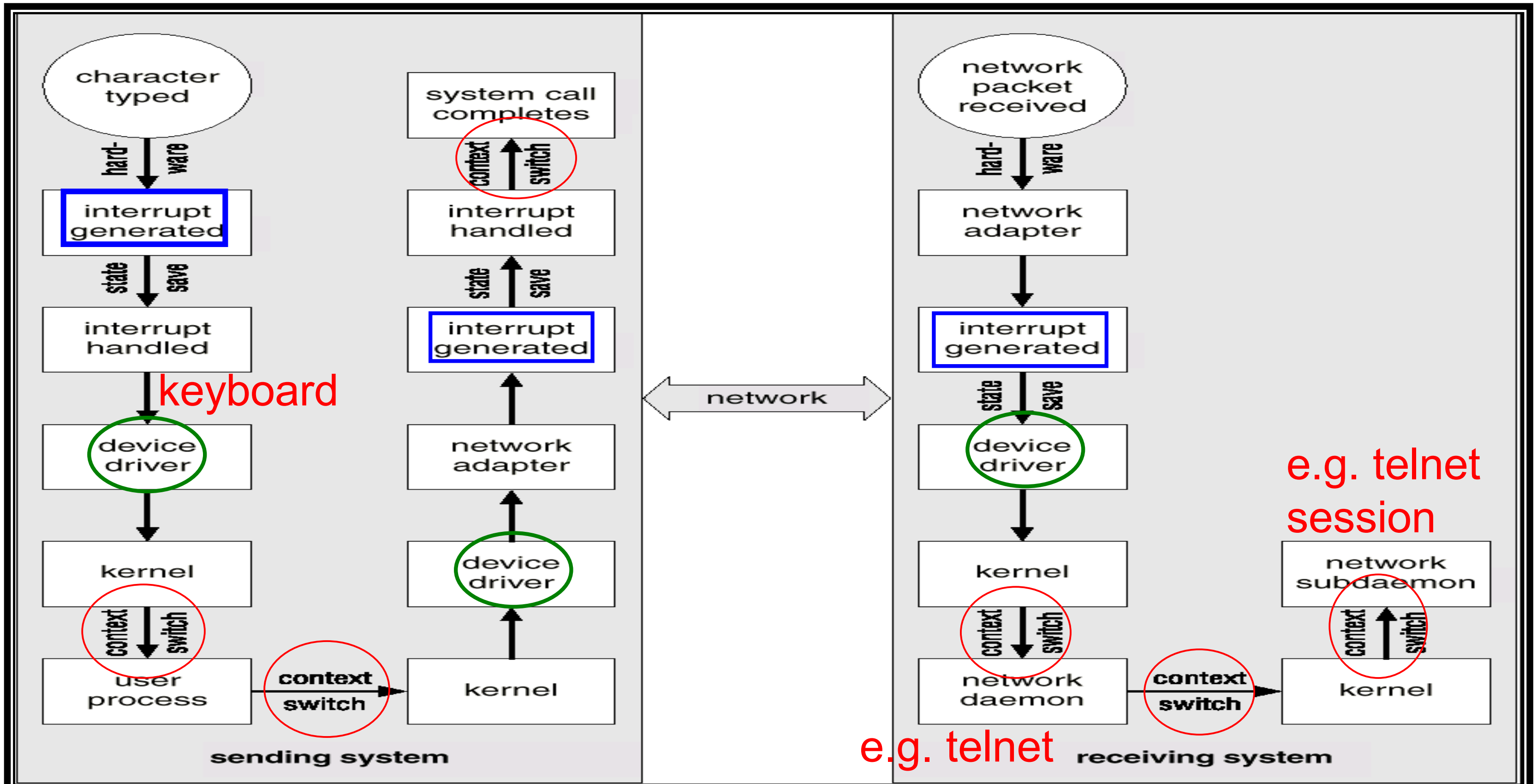


Performance

- I/O is a major factor in **system** performance
  - It places heavy demands on the CPU to execute **device driver code**
  - The resulting **context switches** stress the CPU and its hardware caches
  - I/O loads down the memory bus during **data copy** between controllers and physical memory, …
  - **Interrupt handling** is a relatively expensive task
    - Busy-waiting could be more efficient than interrupt-driven if I/O time is small

Intercomputer Communications

- Network traffic could cause high context switch rate
- Interrupt generated during keyboard & network I/O
- Context switch occurs between prog. & kernel (drivers)



Improving Performance

- Reduce number of context switches
- Reduce data copying
- Reduce interrupts by using large transfers, smart controllers, polling
- Use DMA
- Balance CPU, memory, bus, and I/O performance for highest throughput

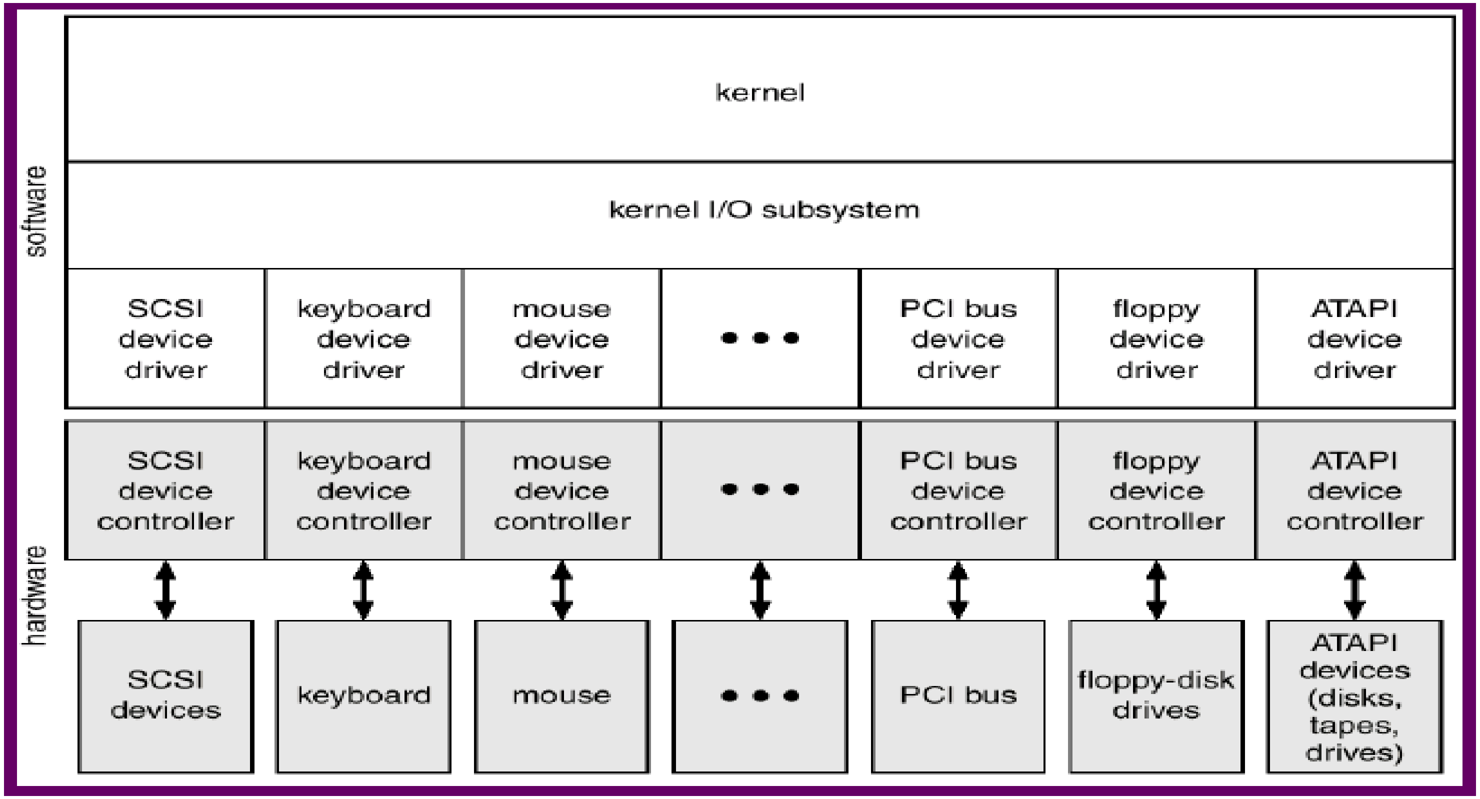
Review Slides (II)

- What are the key I/O services
  - Scheduling
  - Cache
  - Buffering
  - Spooling
  - Error handling
  - I/ protection
- How to improve system performance?

Application I/O Interface

A Kernel I/O Structure

- **Device drivers**: a uniform device-access **interface** to the I/O subsystem; hide the differences among device controllers from the I/O sub-system of OS



Characteristics of I/O Devices

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only readÐwrite	CD-ROM graphics controller disk

I/O Device Class

- Device class is fairly standard across different OS
  - Block I/O
  - Char-stream I/O
  - Memory-mapped file access
  - Network sockets
  - Clock & timer interfaces
- Back-door interfaces (e.g. ioctl())
  - Enable an application to access any functionality implemented by a device driver without the need to invent a new system call

Block & Char Devices

- Block devices: disk drives
  - system calls: read( ), write( ), seek( )
  - Memory-mapped file can be layered on top
- Char-stream devices: mouse, keyboard, serial ports
  - system calls: get( ), put( )
  - Libraries layered on top allow line editing

Network Devices

- Varying enough from block and character to have own interface
  - System call: send(), recv(), select()
  - select() returns which socket is waiting to send or receive, eliminates the need of busy waiting
- Many other approaches
  - pipes, FIFOS, STREAMS, message queues

Textbook Questions

- 13.3: Typically, at the completion of a device I/O, a single interrupt is raised and appropriately handled by the host processor. In certain settings, however, the code that is to be executed at the completion of the I/O can be broken into two separate pieces. The first piece executes immediately after the I/O completes and schedules a second interrupt for the remaining piece of code to be executed at a later time. What is the purpose of using this strategy in the design of interrupt handlers?
- 13.8: Polling for an I/O completion can waste a large number of CPU cycles if the processor iterates a busy-waiting loop many times before the I/O completes. But if the I/O device is ready for service, polling can be much more efficient than is catching and dispatching an interrupt. Describe a hybrid strategy that combines polling, sleeping, and interrupts for I/O device service. For each of these three strategies (pure polling, pure interrupts, hybrid), describe a computing environment in which that strategy is more efficient than is either of the others.

Textbook Questions

- 13.9: Consider the following I/O scenarios on a single-user PC:
  a. A mouse used with a graphical user interface
  b. A tape drive on a multitasking operating system (with no device pre allocation available)
  c. A disk drive containing user files
  d. A graphics card with direct bus connection, accessible through memory-mapped I/O

  For each of these scenarios, would you design the operating system to use buffering, spooling, caching, or a combination? Would you use polled I/O or interrupt-driven I/O? Give reasons for your choices