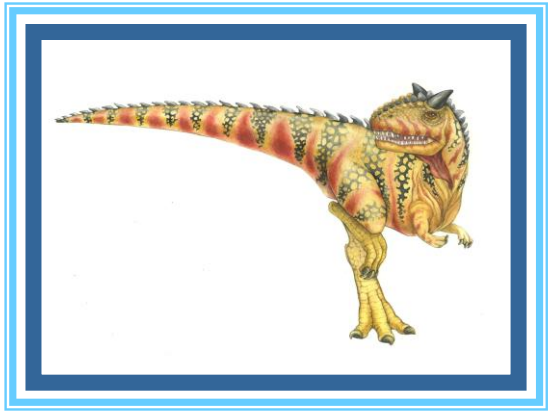
Chapter 12: I/O Systems



Silberschatz, Galvin and Gagne ©2018

Chapter 12: I/O Systems

- Overview
- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem
- Transforming I/O Requests to Hardware Operations
- STREAMS
- Performance

Objectives

- Explore the structure of an operating system's I/O subsystem
- Discuss the principles and complexities of I/O hardware
- Explain the performance aspects of I/O hardware and software

communication between I/O device & OS

Overview

process ⇒ [I/O request code & CPU computation code]

- I/O management is a major component of operating system design and operation
  - Important aspect of computer operation
  - I/O devices vary greatly
  - Various methods to control them
  - Performance management
  - New types of devices frequent
- Ports, busses, device controllers connect to various devices
- **Device drivers** encapsulate device details
  - Present uniform device-access interface to I/O subsystem

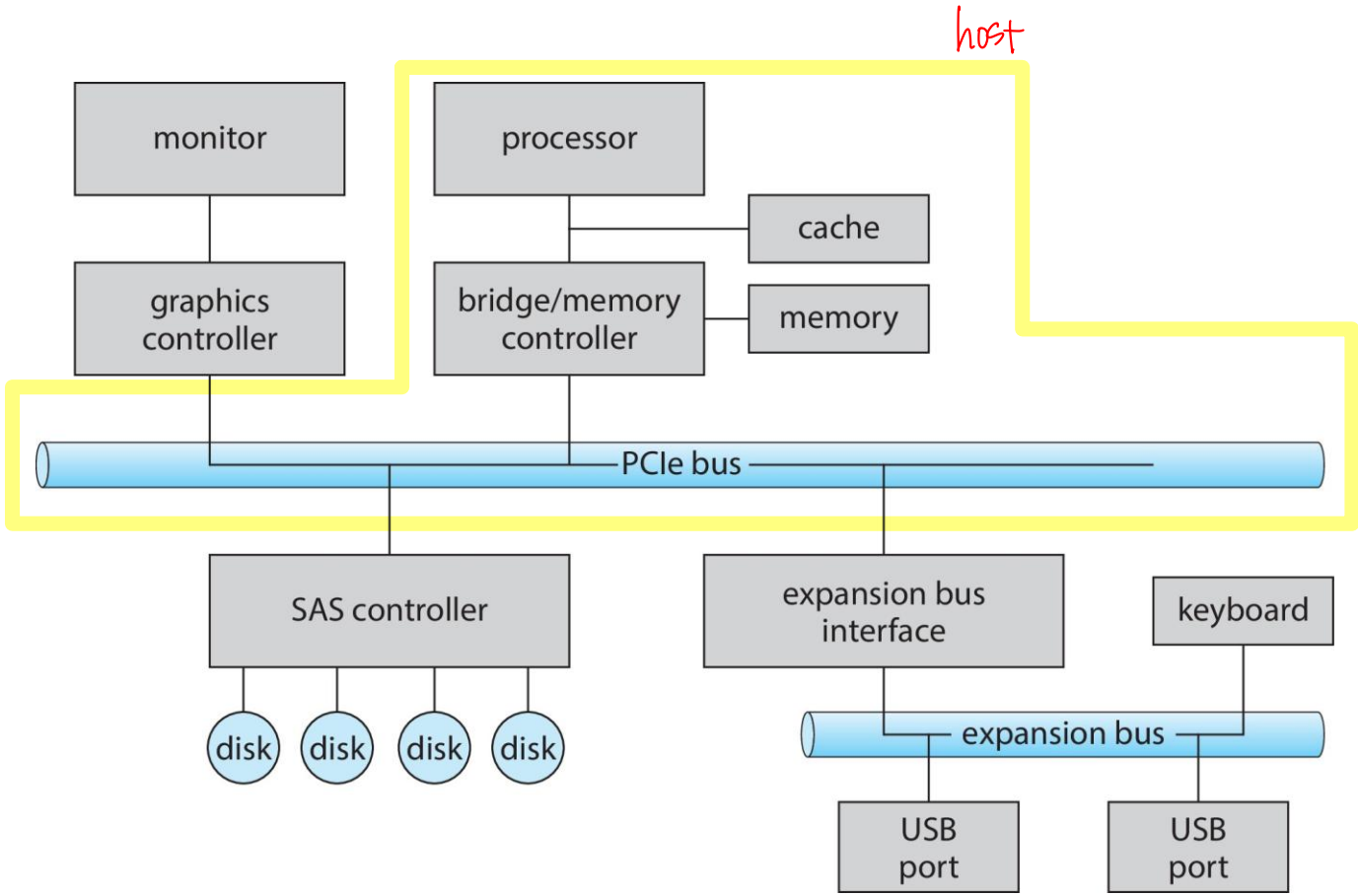
- I/O controller in I/O device communicate with CPU through ports of main board
- Ports are connected with busses

in main board

I/O Hardware

- Incredible variety of I/O devices
  - Storage
  - Transmission
  - Human-interface — I/O device communicates with HOST (main board with CPU & MEM)
- Common concepts – signals from I/O devices interface with computer
  - **Port** – connection point for device (single port, multi devices)
  - **Bus** - daisy chain or shared direct access
    - **PCI** bus common in PCs and servers, PCI Express (**PCIe**)
    - **expansion bus** connects relatively slow devices for human interface (keyboard, mouse, ...)
    - **Serial-attached SCSI** (**SAS**) common disk interface for storage (Small computer system interface)
  - **Controller** (**host adapter**) – electronics that operate port, bus, device
    - Sometimes integrated in I/O device
    - Sometimes separate circuit board (host adapter)
    - Contains processor, microcode, private memory, bus controller, etc (independent system)
      - Some talk to per-device controller with bus controller, microcode, memory, etc

A Typical PC Bus Structure



I/O Hardware (Cont.)

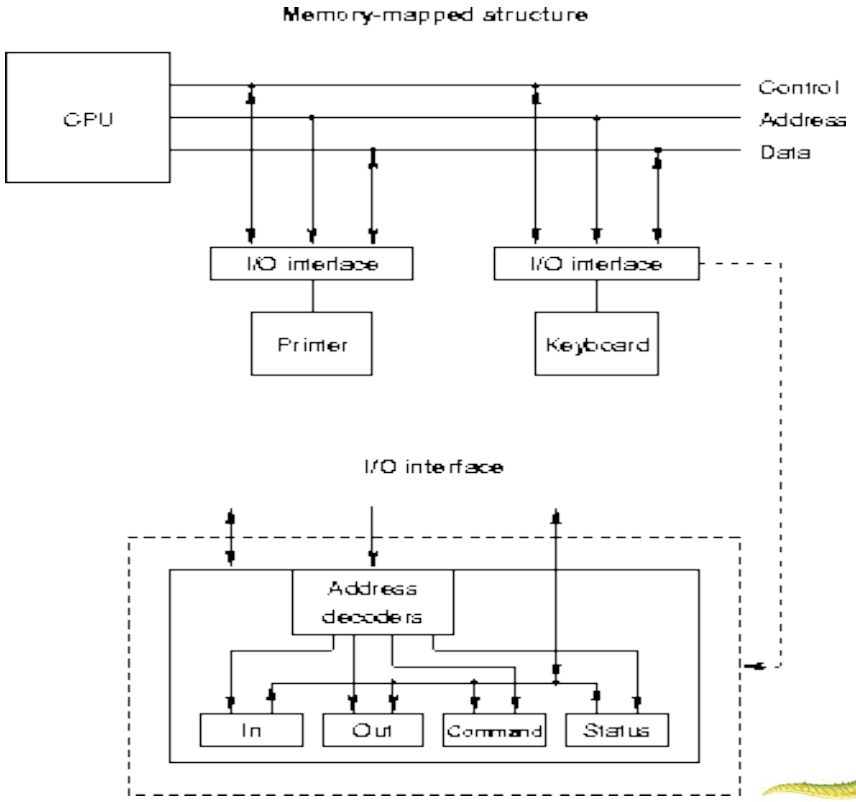
- I/O instructions control devices
- Devices usually have registers where device driver places commands, addresses, and data to write, or read data from registers after command execution
  - Data-in register, data-out register, status register, control register
  - Typically 1-4 bytes, or FIFO buffer
- Devices have addresses, used by
  - Direct I/O instructions
  - **Memory-mapped I/O**
    - Device data and command registers mapped to processor address space
    - Especially for large address spaces (graphics)

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)

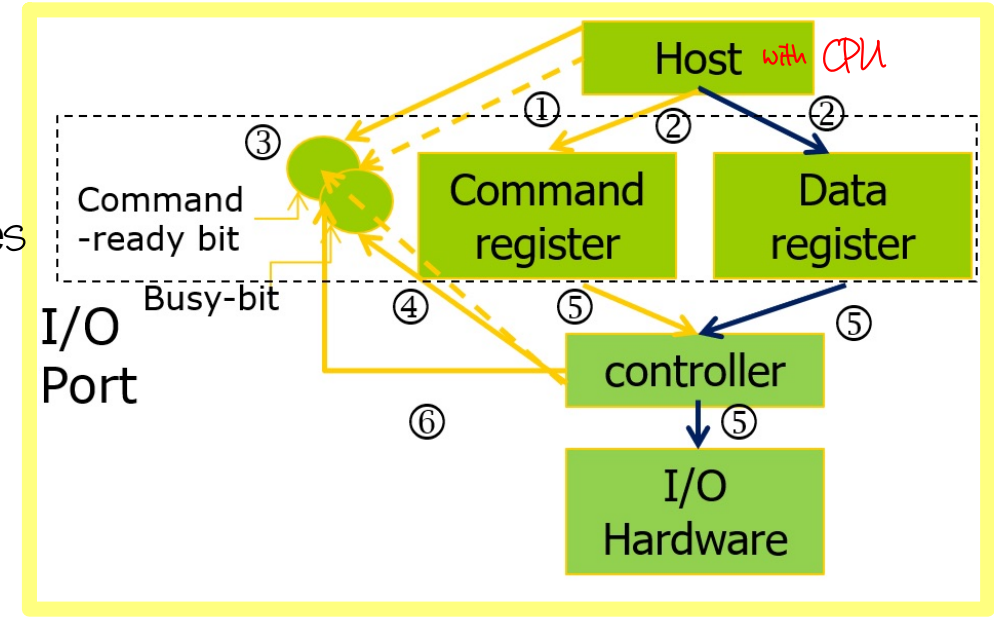
* Typical I/O port has four registers

Data-in	Data-out
Status	Control



Polling

- For each byte of I/O
  1. Read busy bit from status register until 0 ⇒ I/O device is idle
  2. Host sets write bit and if write copies data into data-out register
  3. Host sets command-ready bit
  4. Controller sets busy bit,
  5. Controller executes transfer
  6. Controller clears busy bit, error bit, command-ready bit when transfer done

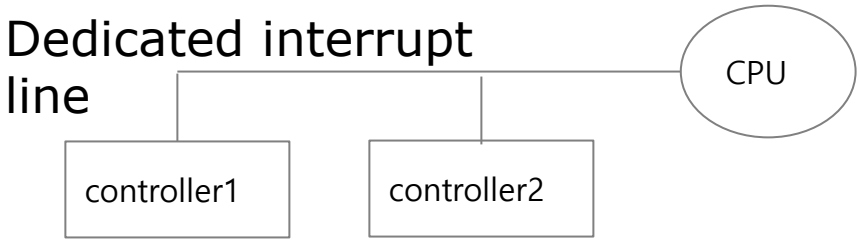


- Step 1 is **busy-wait** cycle to wait for I/O from device
  - Reasonable if device is fast
  - But inefficient if device slow
  - CPU switches to other tasks?
    - But if miss a cycle data overwritten / lost

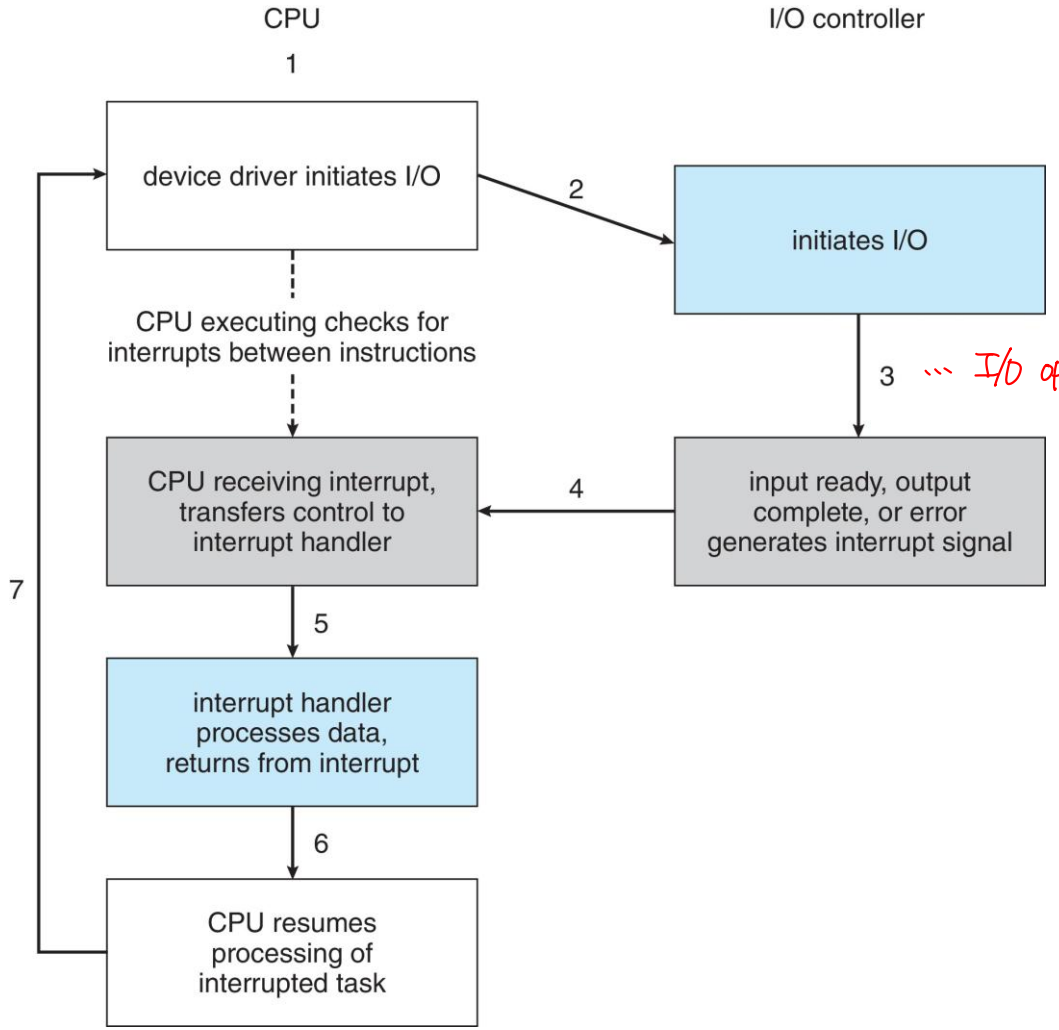
Keyboard

Interrupts

- Polling can happen in 3 instruction cycles
  - Read status, logical-and to extract status bit, branch if not zero
  - How to be more efficient if non-zero infrequently?
- CPU **Interrupt-request line** triggered by I/O device
  - Checked by processor after each instruction
- **Interrupt handler** receives interrupts
  - **Maskable** to ignore or delay some interrupts
- **Interrupt vector** to dispatch interrupt to correct handler
  - Context switch at start and end
  - Based on priority
  - Some **nonmaskable** interrupts are handled immediately
  - Interrupt chaining if more than one device at same interrupt number



Interrupt-Driven I/O Cycle



Interrupts (Cont.)

- Interrupt mechanism also used for **exceptions**
  - Terminate process, crash system due to hardware error
- Page fault executes when memory access error
- System call executes via **trap** to trigger kernel to execute request
- Multi-CPU systems can process interrupts concurrently
  - If operating system designed to handle it
- Used for time-sensitive processing, frequent, must be fast

Latency

- Stressing interrupt management because even single-user systems manage hundreds or interrupts per second and servers hundreds of thousands
- For example, a quiet macOS desktop generated 23,000 interrupts over 10 seconds

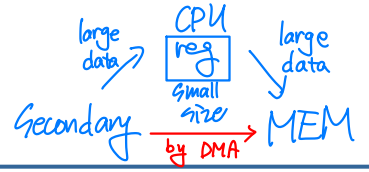
```
Fri Nov 25 13:55:59                                    0:00:10
                      SCHEDULER     INTERRUPTS
-----------------------------------------------
total_samples                13          22998

delays <  10 usecs           12          16243
delays <  20 usecs            1           5312
delays <  30 usecs            0            473
delays <  40 usecs            0            590
delays <  50 usecs            0             61
delays <  60 usecs            0            317
delays <  70 usecs            0              2
delays <  80 usecs            0              0
delays <  90 usecs            0              0
delays < 100 usecs            0              0
total  < 100 usecs           13          22998
```

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

nonmaskable

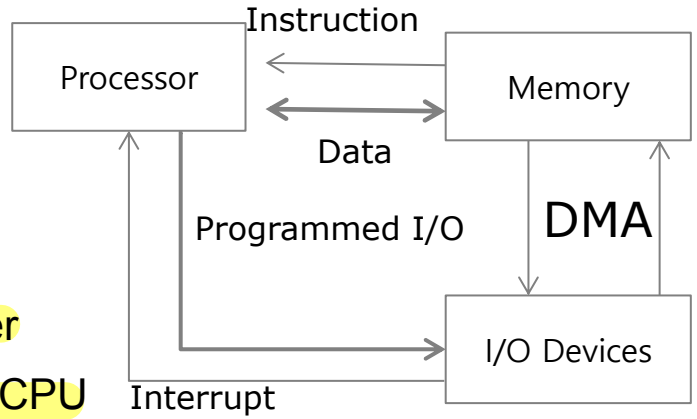
Direct Memory Access



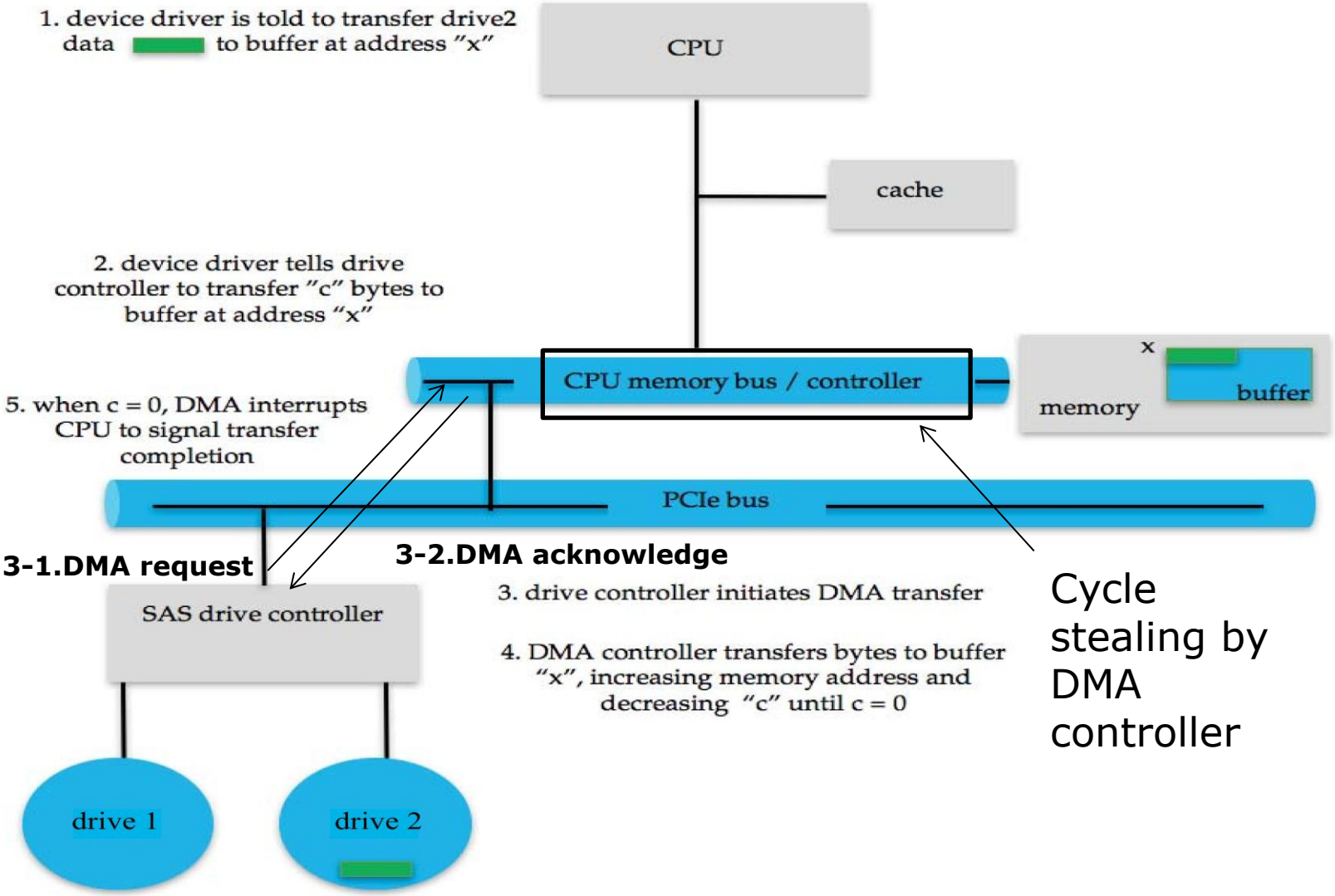
takes much time for large data movement (secondary → MEM) : CPU involved

- Used to avoid **programmed I/O** (one byte at a time) for large data movement
- Requires **DMA** controller
- 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 of command block to DMA controller
  - Bus mastering of DMA controller – grabs bus from CPU
    - **Cycle stealing** from CPU but still much more efficient

      DMA use bus → CPU cannot use bus
  - When done, interrupts to signal completion
- Version that is aware of virtual addresses can be even more efficient - **DVMA**



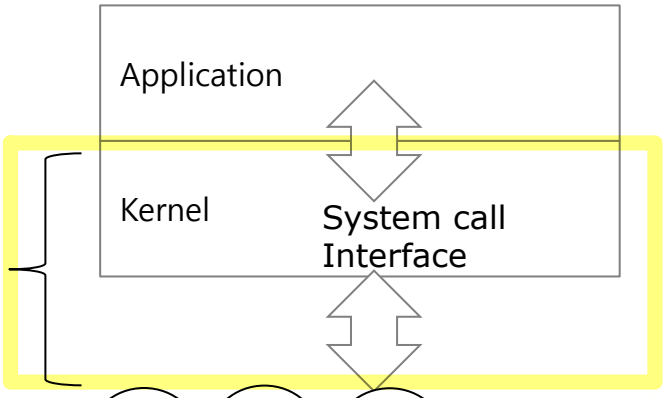
Six Step Process to Perform DMA Transfer



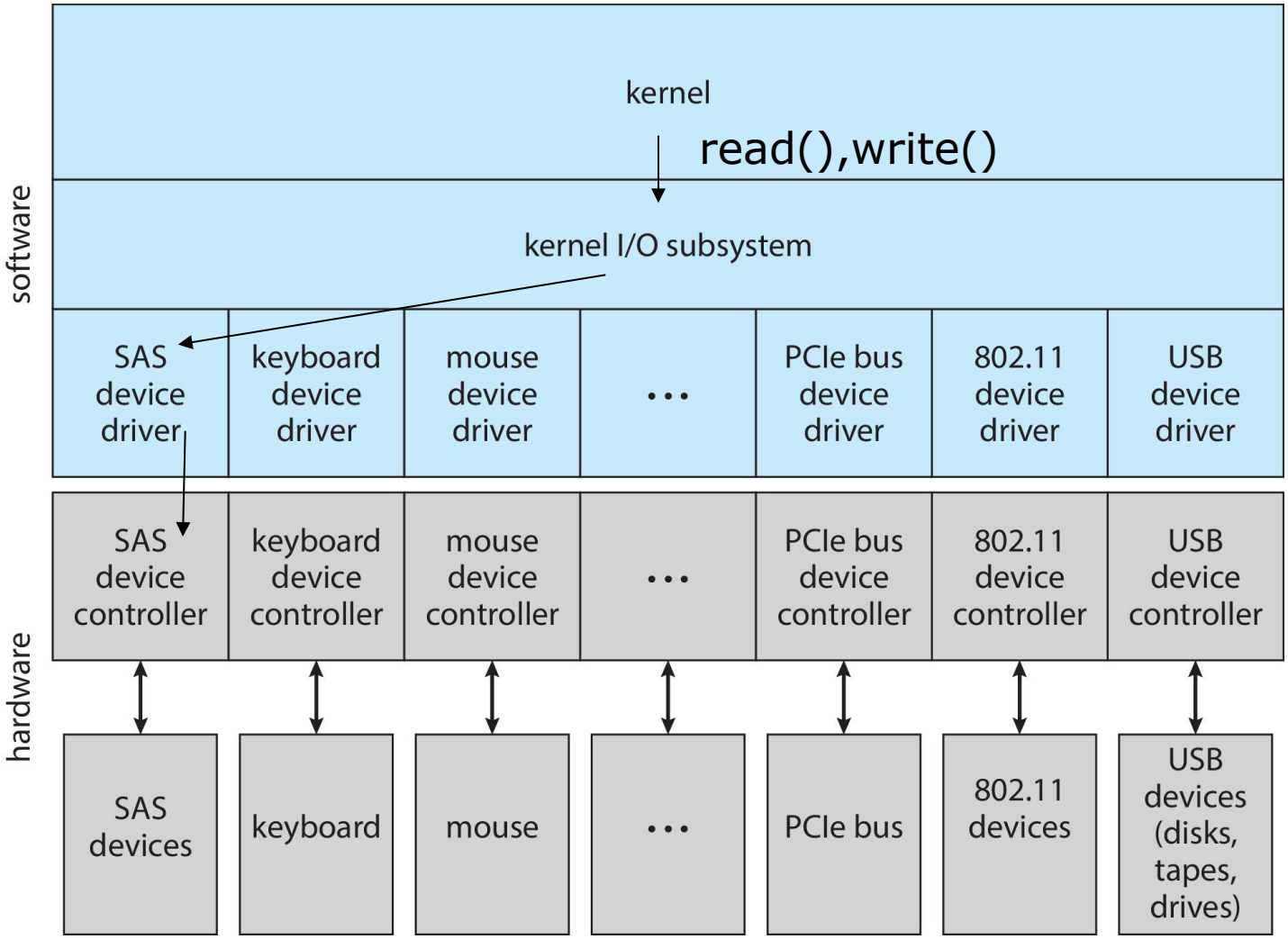
Cycle stealing by DMA controller

Application I/O Interface

- I/O system calls encapsulate device behaviors in generic classes (abstraction)
- Device-driver layer hides differences among I/O controllers from kernel
- New devices talking already-implemented protocols need no extra work
- Each OS has its own I/O subsystem structures and device driver frameworks
- Devices vary in many dimensions
  - **Character-stream** or **block**
  - **Sequential** or **random-access**
  - **Synchronous** or **asynchronous** (or both)
  - **Sharable** or **dedicated**
  - **Speed of operation**
  - **read-write**, **read only**, or **write only**



A Kernel I/O Structure



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 read–write	CD-ROM graphics controller disk

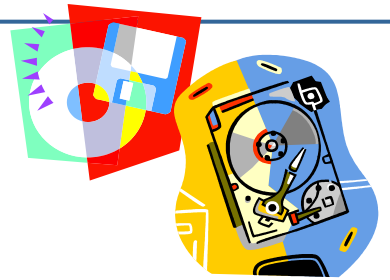
main factor

Characteristics of I/O Devices (Cont.)

- Subtleties of devices handled by device drivers
- Broadly I/O devices can be grouped by the OS into
  - Block I/O
  - Character I/O (Stream)
  - Memory-mapped file access
  - Network sockets
- For direct manipulation of I/O device specific characteristics, usually an escape / back door
  - Unix `ioctl()` call to send arbitrary bits to a device control register and data to device data register
- UNIX and Linux use tuple of "major" and "minor" device numbers to identify type and instance of devices (here major 8 and minors 0-4)

```
% ls -l /dev/sda*
brw-rw---- 1 root disk 8, 0 Mar 16 09:18 /dev/sda
brw-rw---- 1 root disk 8, 1 Mar 16 09:18 /dev/sda1
brw-rw---- 1 root disk 8, 2 Mar 16 09:18 /dev/sda2
brw-rw---- 1 root disk 8, 3 Mar 16 09:18 /dev/sda3
```

Block and Character Devices



- Block devices include disk drives
  - Commands include read, write, seek
  - **Raw I/O**, **direct I/O**, or file-system access
    - Look data as they are in linear arrays
  - Memory-mapped file access possible
    - Data in consecutive addresses in memory = data in consecutive blocks

- Character devices include keyboards, mice, serial ports
  - Commands include `get()`, `put()`
  - Libraries layered on top allow line editing
  - Don't know what data and when they will be given as an input or output
  - Preferable to process streaming data like a/v data or data for printers

Network Devices

kind of character devices

- Varying enough from block and character to have own interface
- Linux, Unix, Windows and many others include **socket** interface
  - Separates network protocol from network operation
  - Includes `select()` functionality
    - Which sockets have the received packets or available buffers to send packets
    - Prevents busy-waiting

read(fd,buf,data)

If fd is file descriptor: read data from file

If fd is socket: read data that came from the device connected through network

socket ⟷ A remote program

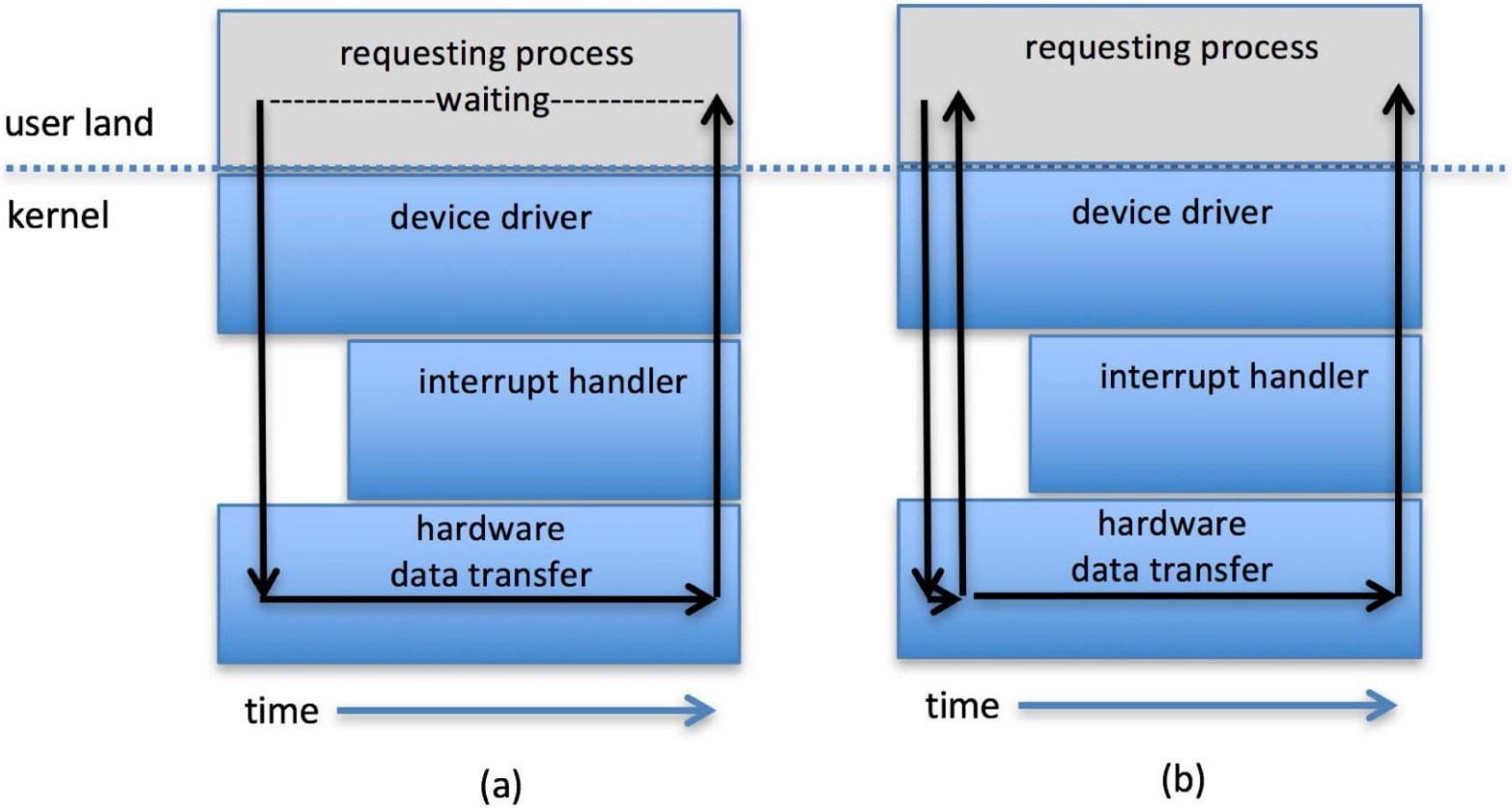
Clocks and Timers

- Provide current time, elapsed time, timer
  - Example of timer use
    - CPU preemption in RR scheduling
    - Flushing the data in cache buffer into disks
    - Some network operations
- Normal resolution about 1/60 second
- Some systems provide higher-resolution timers
- **Programmable interval timer** used for timings, periodic interrupts
  - How to use a timer for many timer events?
    1. Set the timer for the earliest event among them
    2. After the interruption by the earliest event, set the timer for the second earliest event

Nonblocking and Asynchronous I/O

- **Blocking** - process suspended until I/O completed
  - Easy to use and understand
  - Insufficient for some needs
- **Nonblocking** - I/O call returns as much as available
  - User interface, data copy (buffered I/O)
  - Implemented via multi-threading
  - Returns quickly with count of bytes read or written
  - `select()` to find if data ready then `read()` or `write()` to transfer
- **Asynchronous** - process runs while I/O executes
  - Difficult to use
  - I/O subsystem signals process when I/O completed

Two I/O Methods



Synchronous

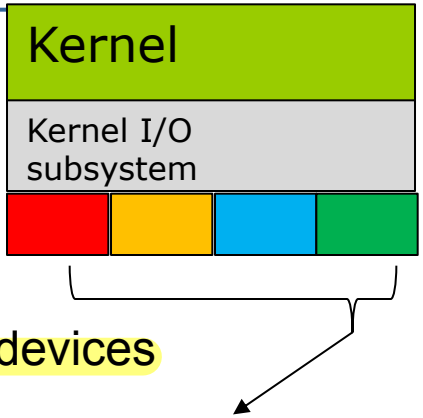
Asynchronous

Vectored I/O

- **Vectored I/O** allows one system call to perform multiple I/O operations
- For example, Unix `readve()` accepts a vector of multiple buffers to read into or write from
- This scatter-gather method better than multiple individual I/O calls
  - Decreases context switching and system call overhead
  - Some versions provide atomicity ∵ no context switching
    - Avoid for example worry about multiple threads changing data as reads / writes occurring

Kernel I/O Subsystem

- **Scheduling**
  - Some I/O request ordering via per-device queue
  - Some OSs try fairness



- **Buffering** - store data in memory while transferring between devices
  - To cope with device speed mismatch
  - To cope with device transfer size mismatch
  - To maintain "copy semantics"
  - **Double buffering** – two copies of the data

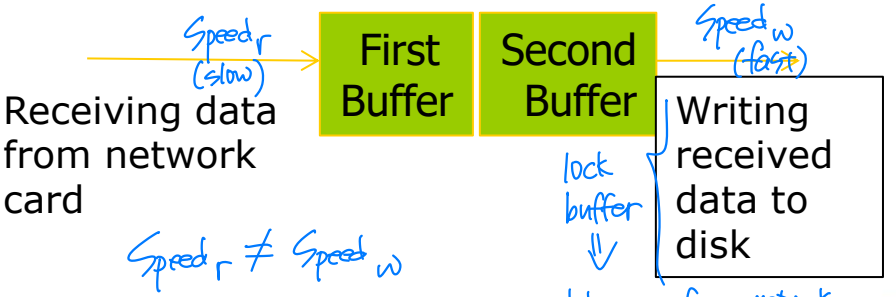
* Copy semantics

```
write(fd,array,100);
memset(array,0,100);
```

system call
update content in MEM

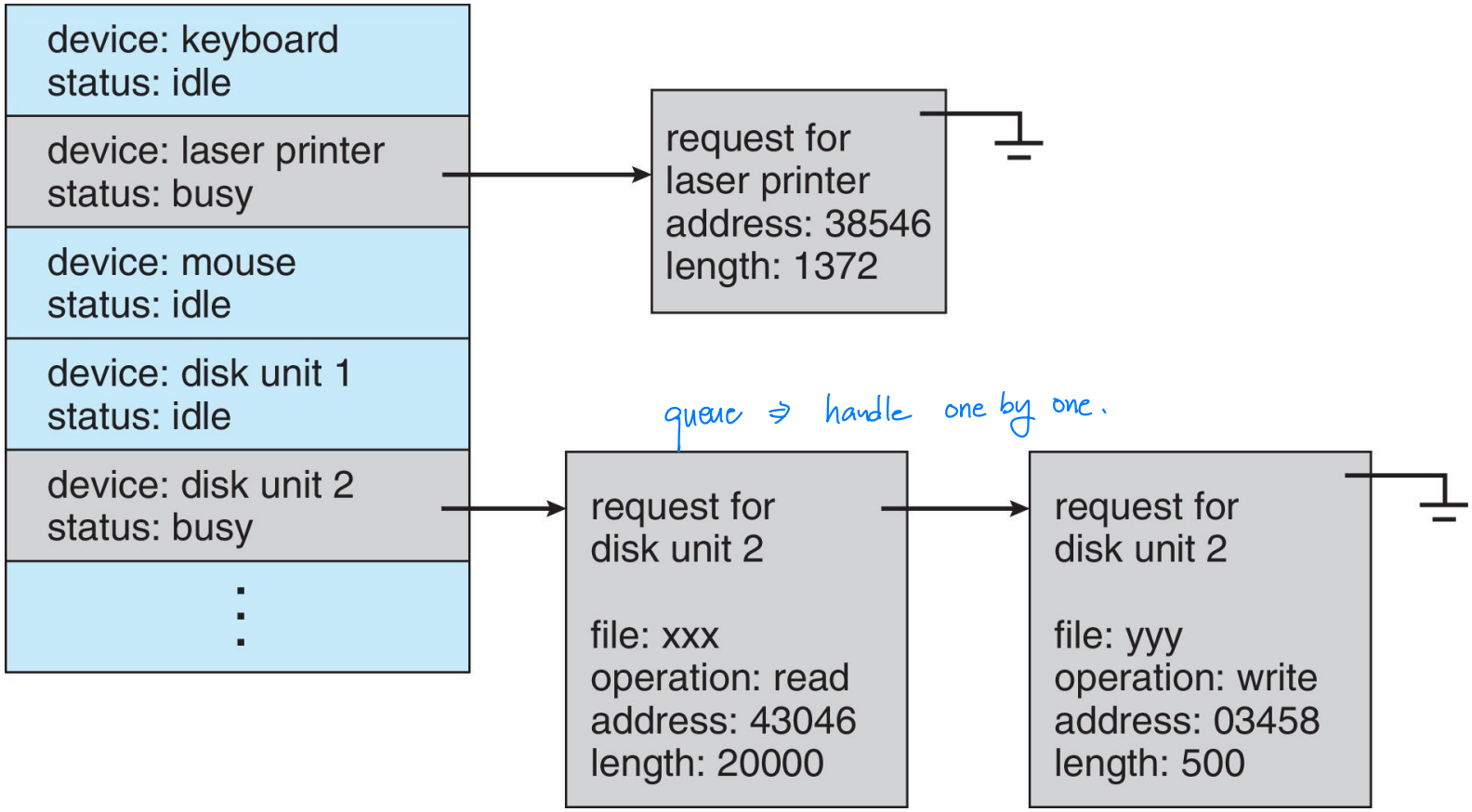
write 명령에서 array의 data를 buffer에 잠시 저장해 연산수행
asynchronous하게 memset이 실행되어도 원래 data 보존 가능
⇒ write 실행에 지장X

* Double buffering

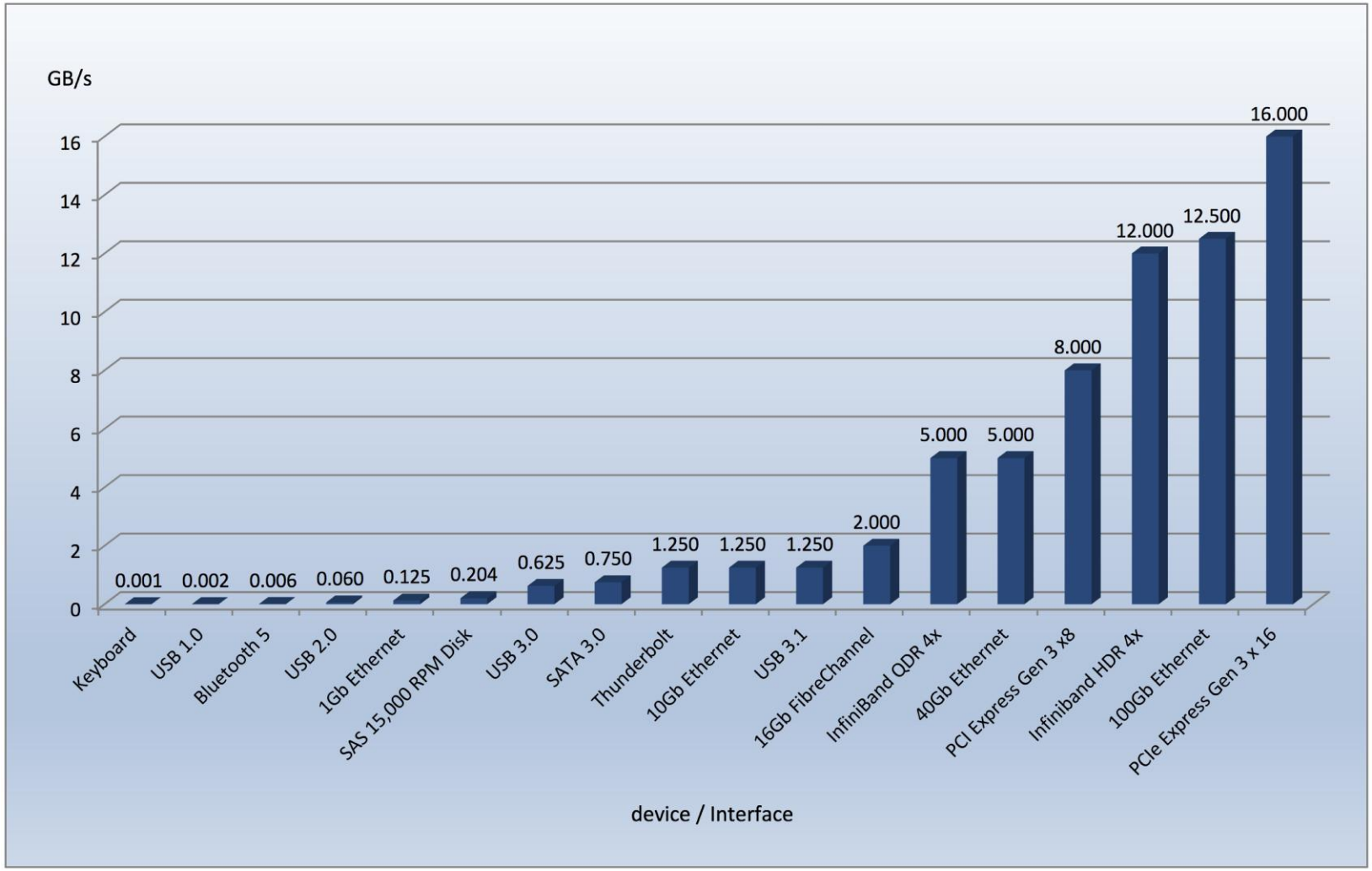


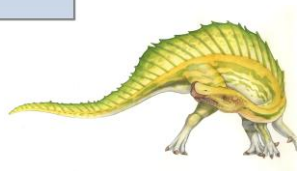
Device-status Table

- To keep track of many I/O requests to support asynchronous I/O



Common PC and Data-center I/O devices and Interface Speeds





Kernel I/O Subsystem (Cont'd)

- **Caching** - faster device holding copy of data
  - Always just a copy
  - Key to performance
  - Sometimes combined with buffering
- **Spooling** - hold output for a device e.g., printer
  - If device can serve only one request at a time
  - i.e., Printing
- **Device reservation** - provides exclusive access to a device
  - System calls for allocation and de-allocation
  - Watch out for deadlock

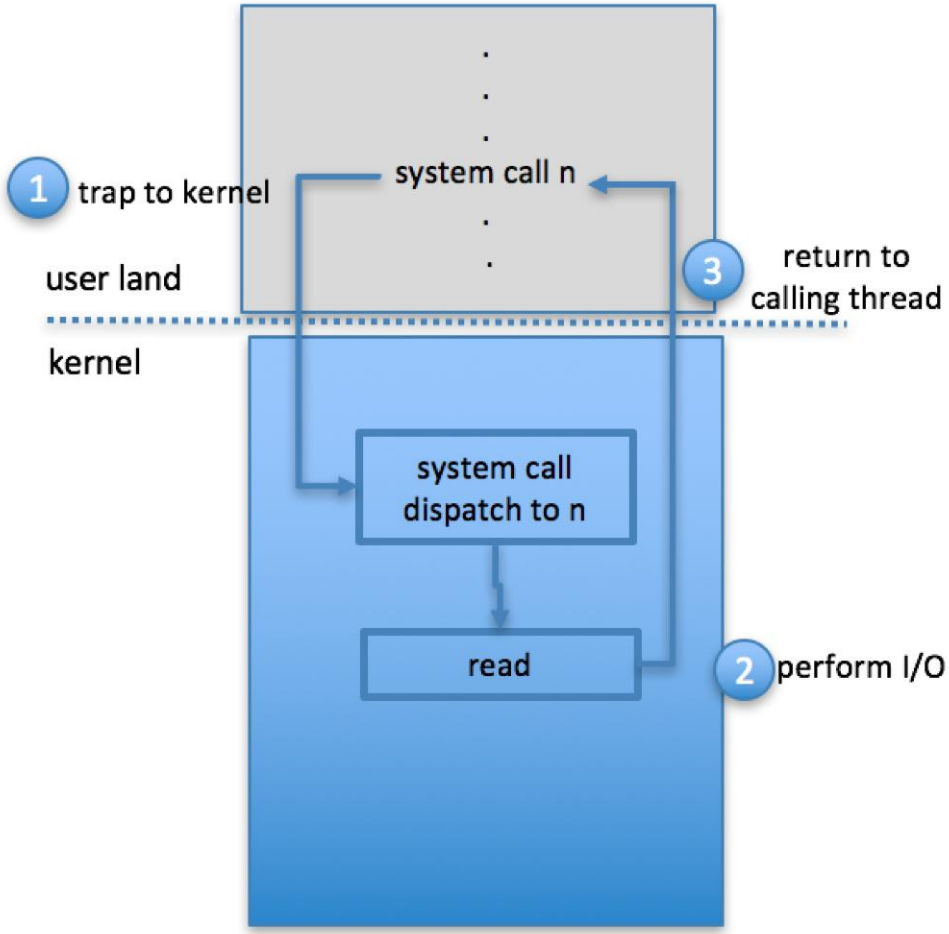
Error Handling

- OS can recover from disk read, device unavailable, transient write failures
  - Retry a read or write, for example
  - Some systems more advanced – Solaris FMA, AIX
    - Track error frequencies, stop using device with increasing frequency of retry-able errors
- Most return an error number or code when I/O request fails
- System error logs hold problem reports

I/O Protection

- User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions
  - All I/O instructions defined to be privileged
  - I/O must be performed via system calls
    - Memory-mapped and I/O port memory locations must be protected too

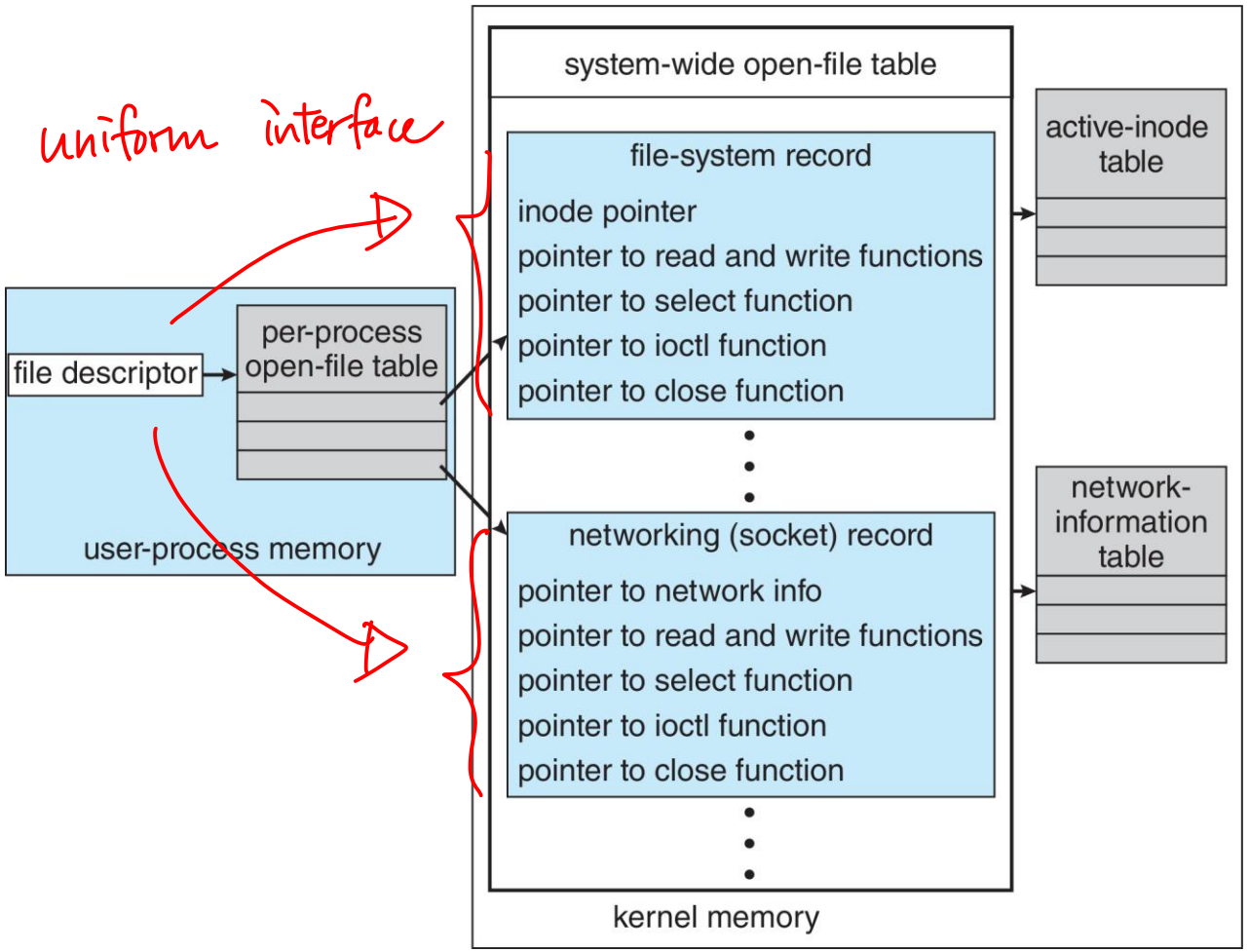
Use of a System Call to Perform I/O



Kernel Data Structures

- Kernel keeps state info for I/O components, including open file tables, network connections, character device state
- Many, many complex data structures to track buffers, memory allocation, “dirty” blocks
- To implement I/O,
  - Linux uses object-oriented methods
    - Supporting uniform interface to encapsulate the difference of the operations
  - Windows uses message passing
    - Message with I/O information passed from user mode into kernel
    - Message modified as it flows through to device driver and back to process

UNIX I/O Kernel Structure



Power Management

- Not strictly domain of I/O, but much is I/O related
- Computers and devices use electricity, generate heat, frequently require cooling
- OSes can help manage and improve use
  - Cloud computing environments move virtual machines between servers
    - Can end up evacuating whole systems and shutting them down
- Mobile computing has power management as first class OS aspect

Power Management (Cont.)

- For example, Android implements
  - Component-level power management
    - Understands relationship between components
    - Build device tree representing physical device topology
      - E.g) System bus → I/O subsystem → {flash, USB storage}
    - Device driver tracks state of device, whether in use
    - Unused component – turn it off
    - All devices in tree branch unused – turn off branch
  - Wake locks – like other locks but prevent sleep of device when lock is held
  - Power collapse – put a device into very deep sleep
    - Marginal power use
    - Only awake enough to respond to external stimuli (button press, incoming call)
- Modern systems use **advanced configuration and power interface** (**ACPI**) firmware providing code that runs as routines called by kernel for device discovery, management, error and power management

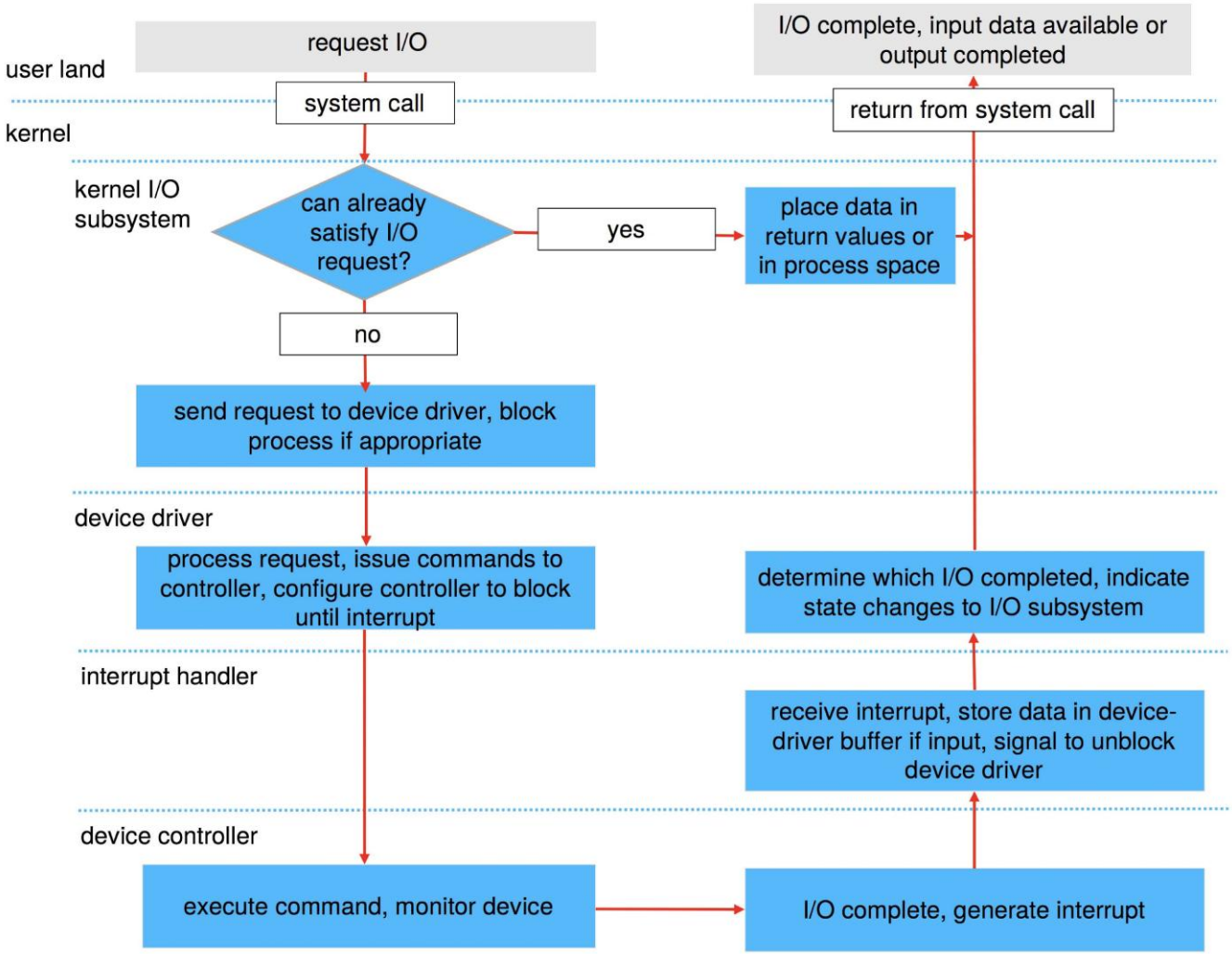
Kernel I/O Subsystem Summary

- In summary, the I/O subsystem coordinates an extensive collection of services that are available to applications and to other parts of the kernel
  - Management of the name space for files and devices
  - Access control to files and devices
  - Operation control (for example, a modem cannot seek())
  - File-system space allocation
  - Device allocation
  - Buffering, caching, and spooling
  - I/O scheduling
  - Device-status monitoring, error handling, and failure recovery
  - Device-driver configuration and initialization
  - Power management of I/O devices
- The upper levels of the I/O subsystem access devices via the uniform interface provided by the device drivers

Transforming I/O Requests to Hardware Operations

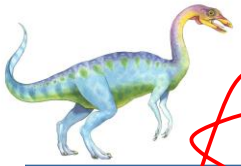
- Consider reading a file from disk for a process:
  - Determine device holding file
  - Translate name to device representation
  - Physically read data from disk into buffer
  - Make data available to requesting process
  - Return control to process

Life Cycle of An I/O Request

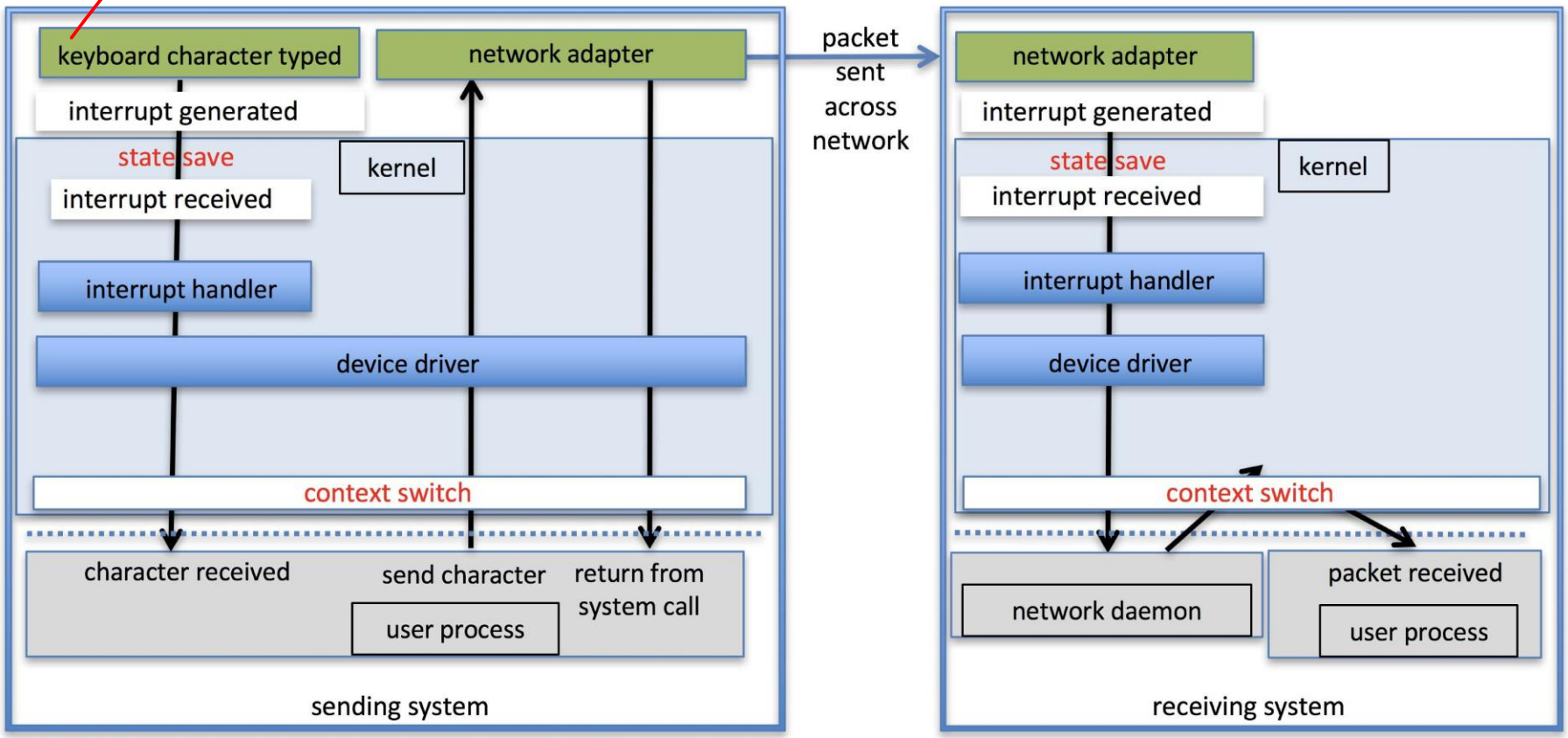


Performance

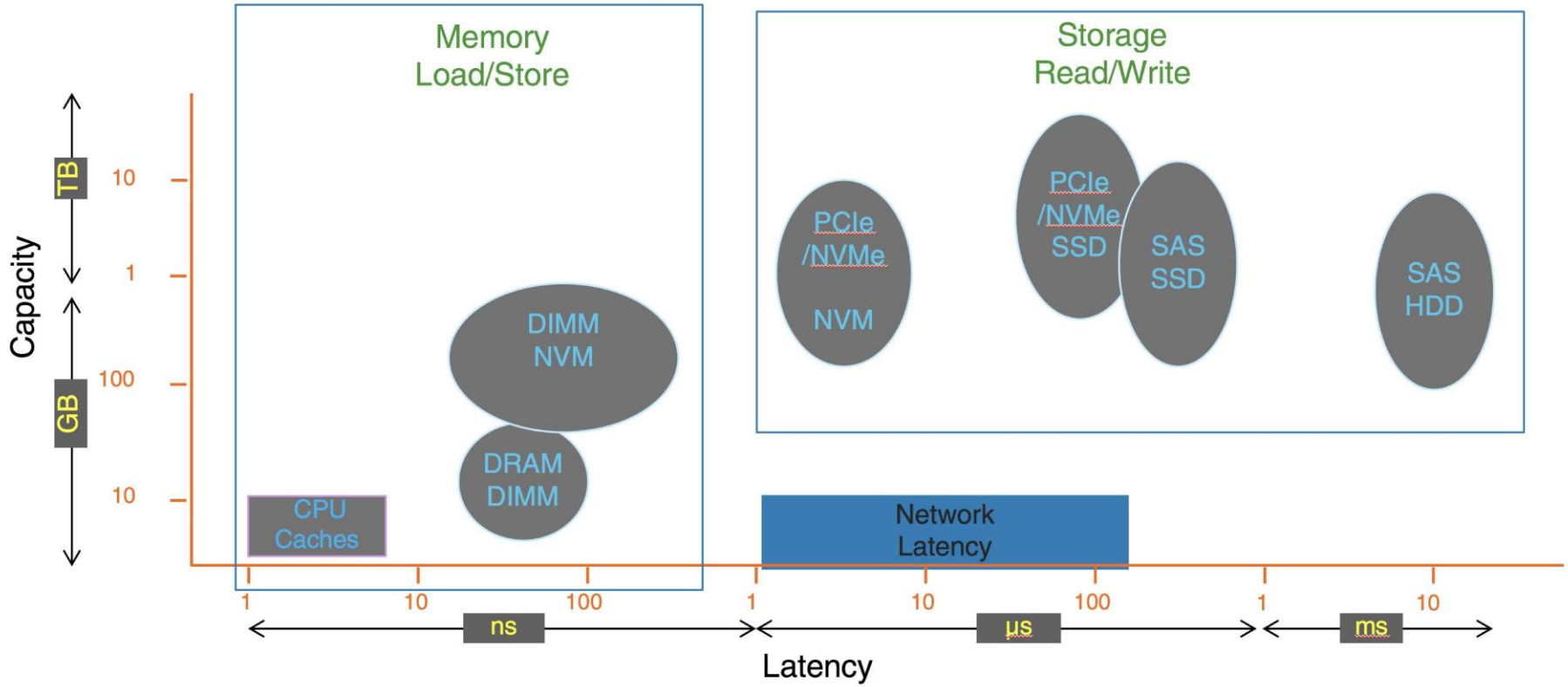
- I/O a major factor in system performance:
  - Demands CPU to execute device driver, kernel I/O code
  - Context switches due to interrupts
  - Data copying
  - Network traffic especially stressful



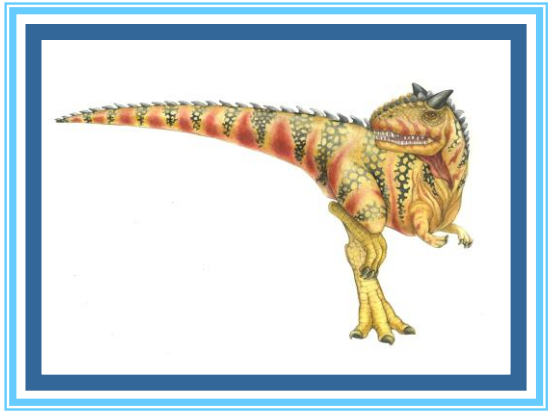
Intercomputer Communications



I/O Performance of Storage (and Network Latency)



End of Chapter 12



Silberschatz, Galvin and Gagne ©2018