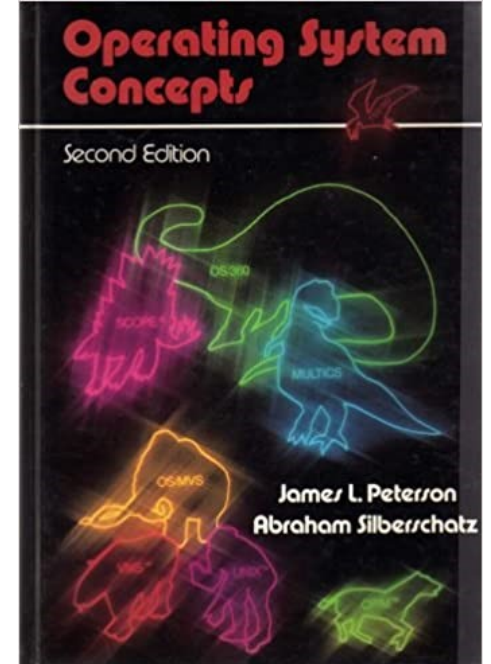


03/09

# Chapter 2: System Structures

School of Computing, Gachon Univ.  
Joon Yoo



# Objectives

---

- To discuss the concepts of multitasking, interrupt, protection, system calls, and caching
- To discuss the various ways of structuring an operating system

# Chapter 2: System Structures

---

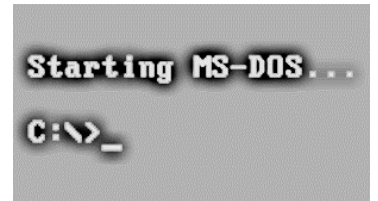
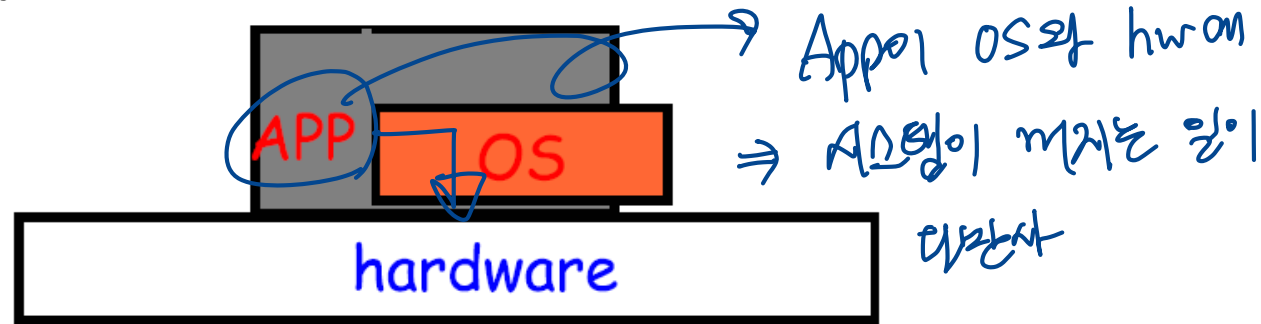
## ■ Basic Operations

- Multitasking (Ch. 1.4)
- Interrupt (Ch. 1.2)
- Protection
  - ▶ Dual-mode operation (Ch. 1.4.2)
  - ▶ Types of System Calls
- Cache Management (Ch. 1.5.5)

## ■ Operating System Structure & Data Structures

# Primitive Operating Systems

- Just a library of standard services [no protection]



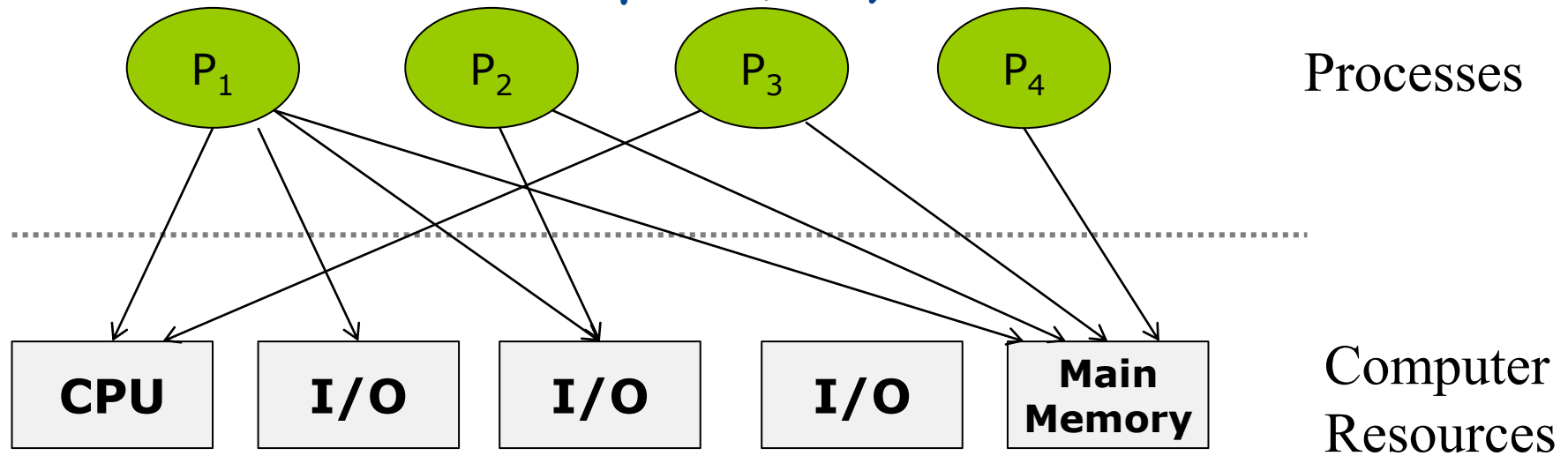
- Standard interface above hardware-specific drivers, etc.
- Simplifying assumptions 추정, 상정 행동에 기반함!
  - System runs **one program** at a time 여러개 X
- Problem: Poor utilization** ⇒ 매우 느리고 불편함. ..
  - ... of hardware (e.g., CPU idle while waiting for disk)
  - ... of human user (must wait for each program to finish)

그러한 불편함을 해결하기 위해 →

# Multitasking (Ch. 1.4)

- Several jobs are kept in main memory at the same time, and these processes share the CPU time, I/O devices and other resources.

공유된 자원 (CPU, I/O...) 등을 같이 사용한다!



- Managing and controlling several processes simultaneously are challenging for OS

즉, OS가 Multitasking을 진행하는 것임!

# Multitasking

- **Multitasking** (=Multiprogramming, Multiprocessing) needed for efficiency

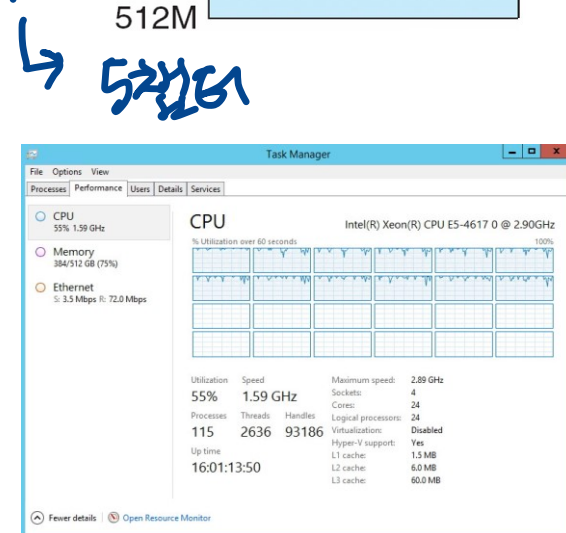
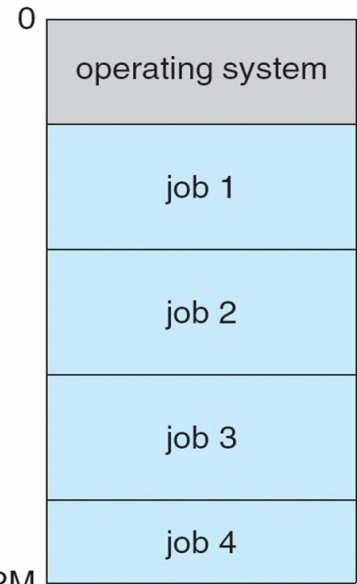
- Users frequently have multiple processes loaded on **Main memory**
- Single process *cannot* keep **CPU** and **I/O devices** busy at all times

- Advantage1: **CPU utilization**

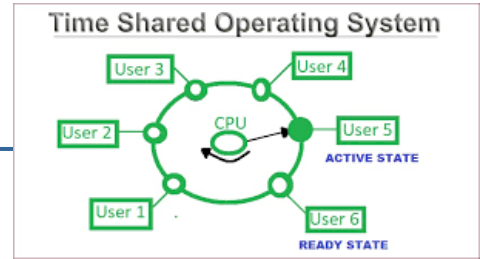
- Keep several **processes** (=job, task) in memory simultaneously
- One process selected and run via **scheduling** → **선행 우선순위 정하기**
- When it has to wait (e.g., for I/O operation), OS **switches** to another process
- As long as at least one process needs to execute, the CPU never stays idle → **CPU 놓지 않지!**
- What happens if it is not switched?

(Hint: Waste CPU cycles (why?))

↳ **CPU utilization 이 향상된다.**



# Multitasking



## Advantage 2: **Timesharing** (of **CPU** resource)

- A logical extension in which CPU switches jobs so **frequently** that the user have an **illusion of multitasking** *사실 하나씩만 실행하고 있지만*

<https://giphy.com/>

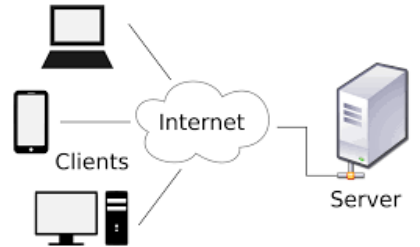
- Allows many users to share the computer **simultaneously**

*switch를 계속 해주니까*



If several jobs ready to run at the same time, which process do we run first? ⇒ **CPU Job scheduling** (Ch. 5)

*switch를 여러개가 동시에 실행되는 것처럼 보인다 ...!*



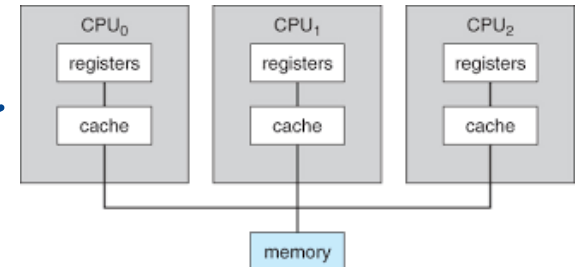
*여러 사용자가 동시에 사용할 수 있게!*

*⇒ CPU가 여러개*

## c.f., **Multiprocessing** vs. **Multiprocessor** (멀티프로세서)

- Multiprocessor: Using multiple CPUs on a single computer

*⇒ CPU가 개이고, 거기서 Utilization & Timesharing을 해주는 것*



# Challenging Issues of Multitasking

---

- At any given time, each resource (e.g., CPU core) can serve only **one** process
- Multiple processes can be executed while sharing memory
- Using I/O device takes a long time, and CPU is idle and underutilized
- An unauthorized process causes a system fault although other processes have no problem

ANS  
⇒



# What does an OS do? (Ch. 1.5)

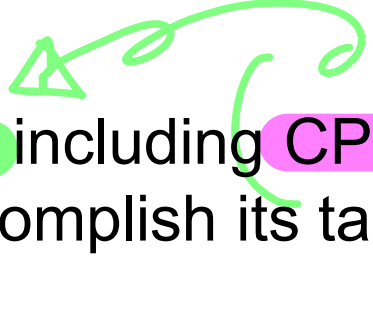
---

- At any given time, each resource (e.g. CPU) can serve only **one** process
  - OS → **Process Management**
- Multiple processes can be executed while sharing memory
  - OS → **Memory Management**
- Using I/O device takes a long time, and CPU is idle and underutilized
  - OS → **I/O Management**
- An unauthorized process causes a system fault although other processes have no problem
  - OS → **Protection**

이런 것들을 전부 OS가 해결해야 하는 일이라는 것임 .

# Process Management (Ch. 1.5.1)

---

- A **Process** : a program in memory
    - A process needs certain **resources**, including **CPU time**, **memory space**, **files**, and **I/O devices**, to accomplish its task
  - OS Functionality (Ch. 3-7)
    - **Create/run/terminate processes/threads** (Ch. 3-4)
    - Who goes first? **CPU scheduling** (Ch. 5)
    - Mechanisms for **resource** sharing and synchronization (Ch. 6-8)
- 

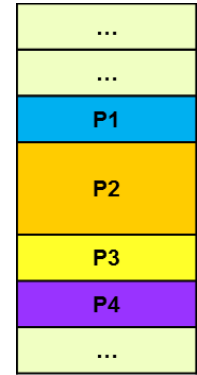
# Memory Management (Ch. 1.5.2)

## ■ Main memory (RAM)

- A large array of bytes, each byte with its own address
- CPU reads both **instructions** and **data** from main memory – it must first be loaded to main memory *↳ 지시, 설명*

## ■ Multitasking

- Multiple processes (= tasks, jobs) **in main memory**
- Improves utilization of CPU and response to its users



## ■ Memory management (Ch. 9-10)

- Keeping track of which parts of memory are currently being used and by whom
- Deciding which processes and data to move into and out of memory
- Allocating and deallocating memory space as needed

# File & Storage Management (Ch. 1.5.3-4)

- OS provides uniform, logical view of storage
  - Abstracts physical storage (**disk**) to logical storage (**file**)

실제 물리적인 저장 공간인 **disk** 에 있는  
내용들을 논리적인 형태로 보여주는 것이 **file** 이다.

Physical storage



OS

Logical storage

File  
System



- OS Functionality
  - *File-system management* (Ch. 13-15)
  - *Mass-storage (disk) management* (Ch. 11)

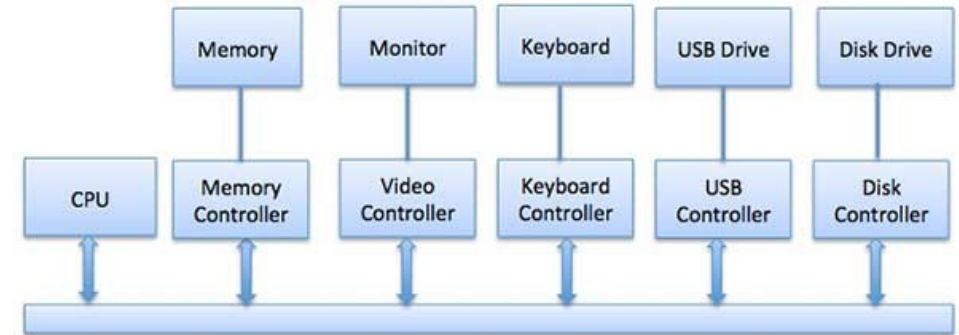
# I/O Subsystem (Ch. 1.2.3)

## ■ I/O Devices *Input, Output Devices*

- mouse, keyboards, touch pad, disk drives, display adapters, USB devices, network connections, audio I/O, printers
- connected by system **bus**



- There is always a **device driver** and **device controller** for each I/O device



# I/O Subsystem

## ■ Device controller (in I/O Device – HW or SW?)

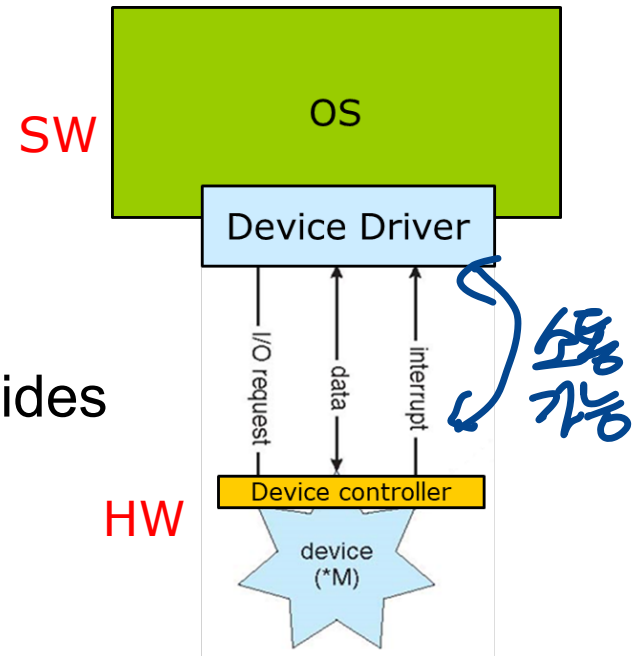
- In charge of one or more specific type of device (e.g., disk, USB, network controller).
- Think of controller as a small CPU for a device
- Move the data between the I/O devices and its local buffer (small memory in I/O device)

Input → local buffer → Main memory

## ■ Device driver (in OS – HW or SW?)

- A device driver for each device controller
- A program that understands the controller and provides OS with an interface to device controller.

∴ USB 같은 경우는 driver가 OS를 자동으로 인식하기 때문에 따로 설치할 필요가 없음



# I/O Subsystem contd.

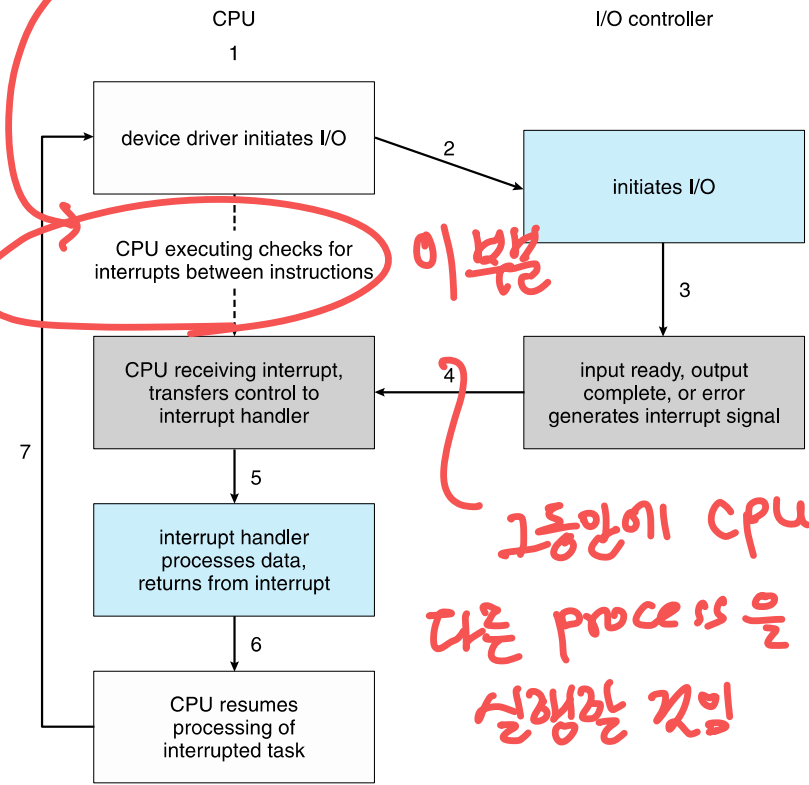
## I/O Operation

- ① **I/O request** from application (e.g., read file from disk): **device driver** delivers command to **device controller**
- ② **controller** determines which action to take
  - ▶ e.g., “read block#369 from hard disk”
- ③ **device controller** takes action
  - ▶ e.g., read block from disk, and transfer data from the device to memory
- ④ device controller informs the CPU via an **interrupt** that it has finished
- ⑤ OS takes control and **handles** interrupt

작업이 끝났다는 것을 알려주기 위해 interrupt를 사용

→ I/O 작업 인식하려면 잠깐 쉬게 된다.

같은 설명  
다른 그림



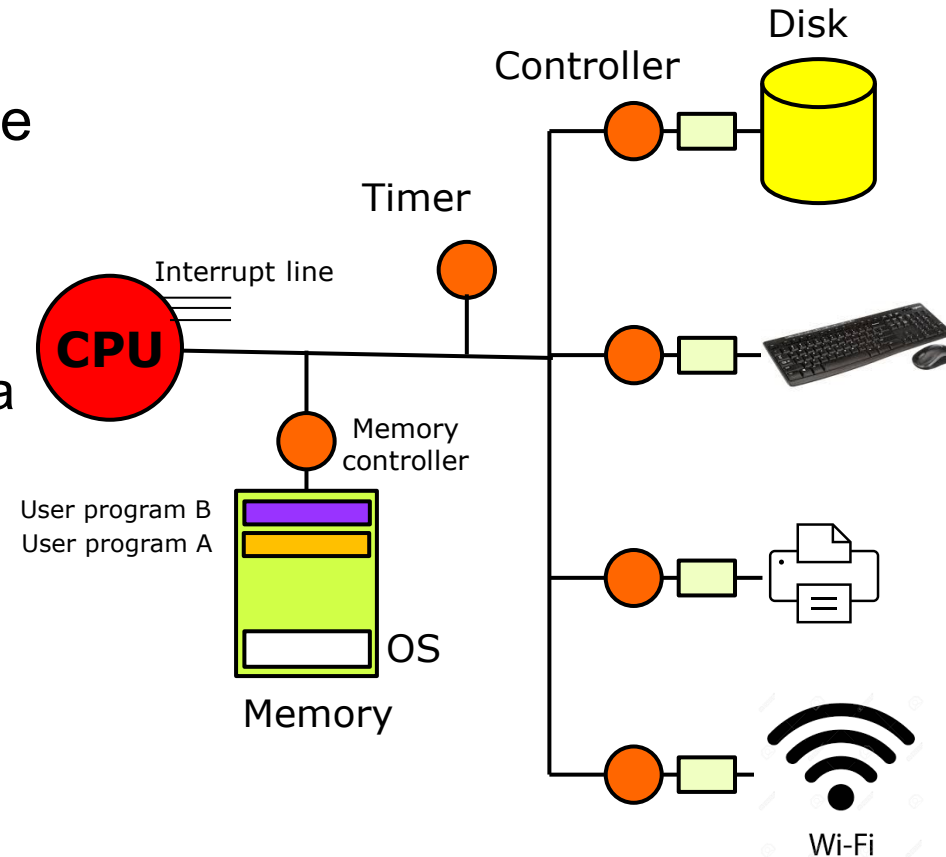
이부분

그동안 CPU는  
다른 process를  
실행할 것임

# I/O Subsystem contd.

## ■ I/O Operation

- ① **I/O request** from application (e.g., read file from disk): device driver delivers command to device controller
- ② controller determines which action to take
  - ▶ e.g., “read block#369 from hard disk”
- ③ **device controller** takes action
  - ▶ e.g., read block from disk, and transfer data from the device to memory
- ④ device controller informs the CPU via an **interrupt** that it has finished
- ⑤ OS takes control and **handles** interrupt





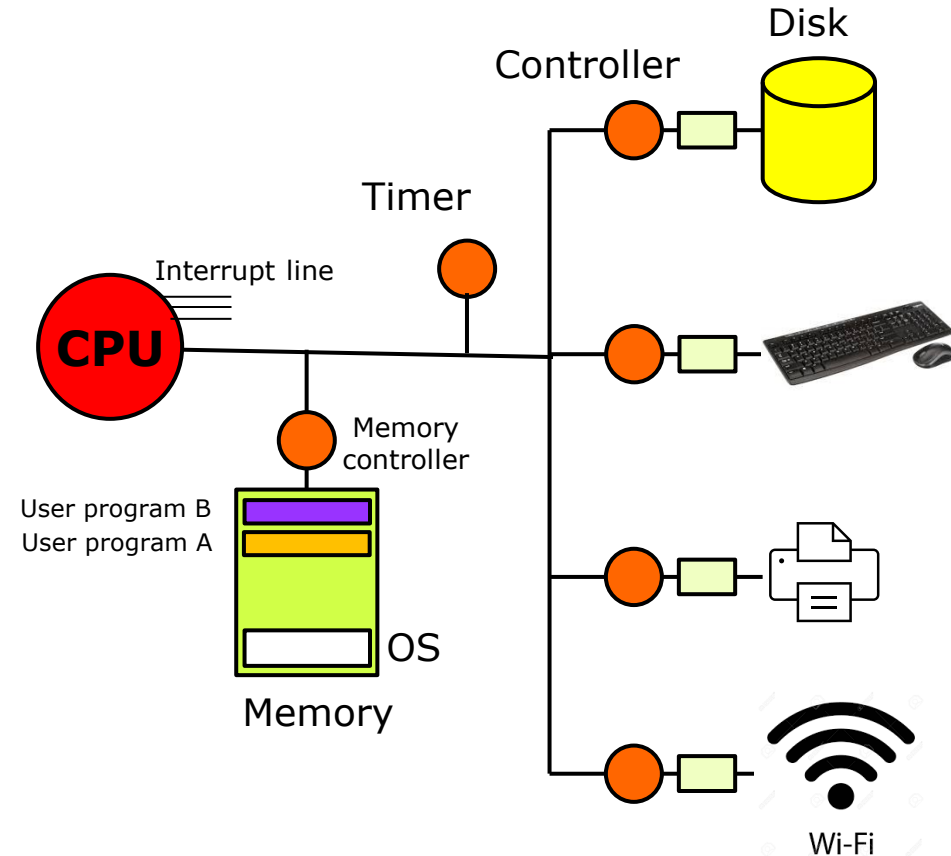
# Chapter 2: System Structures

---

- Basic Operations
  - Multitasking (Ch. 1.4)
  - **Interrupt (Ch. 1.2.1)**
  - Protection
    - ▶ Dual-mode operation (Ch. 1.4.2)
    - ▶ Types of System Calls
  - Cache Management (Ch. 1.5.5)
- Operating System Structure & Data Structures

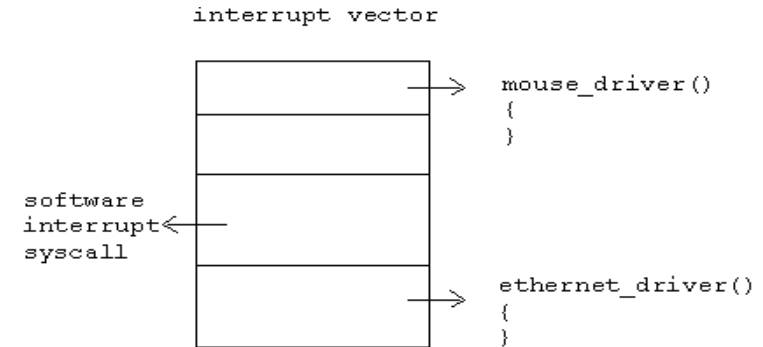
# Interrupt (Ch. 1.2.1)

- Question:
  - Device controller needs to inform CPU that device has finished its operation
    - ▶ e.g., I/O request (read file from disk) → disk controller finished reading file
  - How? Through **Interrupt!**
- The CPU runs an instruction – after each instruction it checks if the interrupt line is set!
- If an interrupt occurs:
  - An Interrupt line is set
  - The CPU will **stop** anything it is doing and it will jump to an **interrupt handler**.



# Interrupt Handler

- When the CPU is interrupted, the CPU
  - transfer execution to **interrupt vector**
    - ▶ The interrupt vector has the **addresses** of the **service routine** for each type of interrupt
  - execute **interrupt service routine**
    - ▶ e.g., mouse driver, network driver
  - on completion of the service routine, the **CPU** resumes the previously executed computation (which was interrupted)



Vector of pointers to function with all interrupt drivers.  
CPU knows where to jump to.

# Interrupt

---

- An operating system is **interrupt driven**
  - **Interrupt** is a key part of OS-hardware interaction
- **Hardware and Software Interrupt**
  - **Hardware Interrupt: Hardware I/O devices** interrupt CPU
    - ▶ Hardware may trigger an interrupt at any time by sending a signal to CPU
    - ▶ Disk operation completed, mouse movement, keyboard type, Internet data packet, disk/CD/Tape driver, timer interrupts (used for scheduling multiple processes), sound (microphone)
  - **Software Interrupt (= traps): Software programs** interrupt CPU
    - ▶ **system calls**: application programs uses to call kernel functions
    - ▶ **exceptions**: divide by zero exception

# Chapter 2: System Structures

---

- Basic Operations
  - Multitasking (Ch. 1.4)
  - Interrupt (Ch. 1.2)
  - Protection
    - ▶ Dual-mode operation (Ch. 1.4.2)
    - ▶ Types of System Calls
  - Cache Management (Ch. 1.5.5)
- Operating System Structure & Data Structures

# Dual-Mode Operation (Ch. 1.4.2)

---

## ■ Motivation

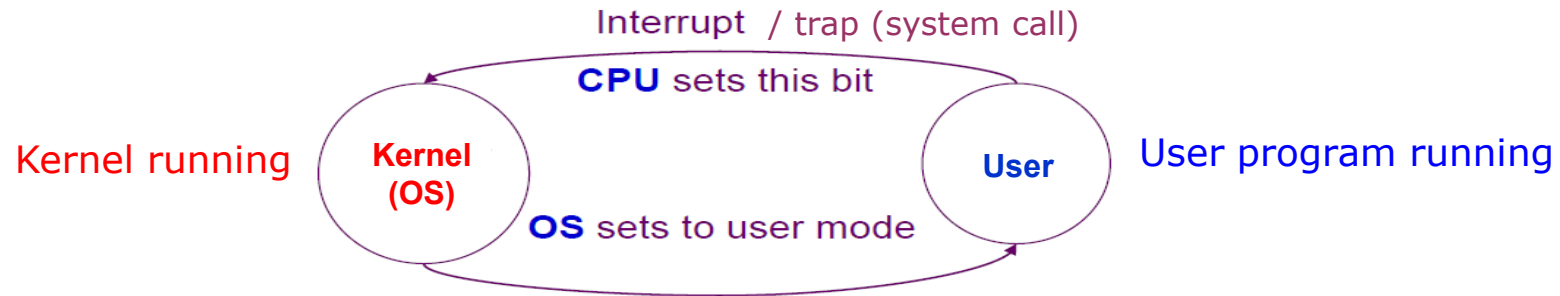
- OS and users share hardware/software of computer system
- Malicious program should not harm other programs (including OS)
- Should distinguish between OS code and user code

## ■ Dual-mode operation

- **Kernel mode** (=supervisor, system, privileged mode) vs. **User mode**
- **Mode bit** provided by hardware support (e.g., CPU)
  - ▶ kernel(0) or user(1)

# Dual-Mode Operation

- CPU **Mode bit** added to computer hardware to indicate the current mode: **kernel (0)** or **user (1)**.



- Life cycle of instruction
  - OS running (kernel mode)
  - User programs running (user mode)
- Example
  - System boot time (\_\_\_\_\_ mode)
  - Starts user application (\_\_\_\_\_ mode)
  - Hardware interrupt occurs (\_\_\_\_\_ mode)

# Dual-Mode Operation

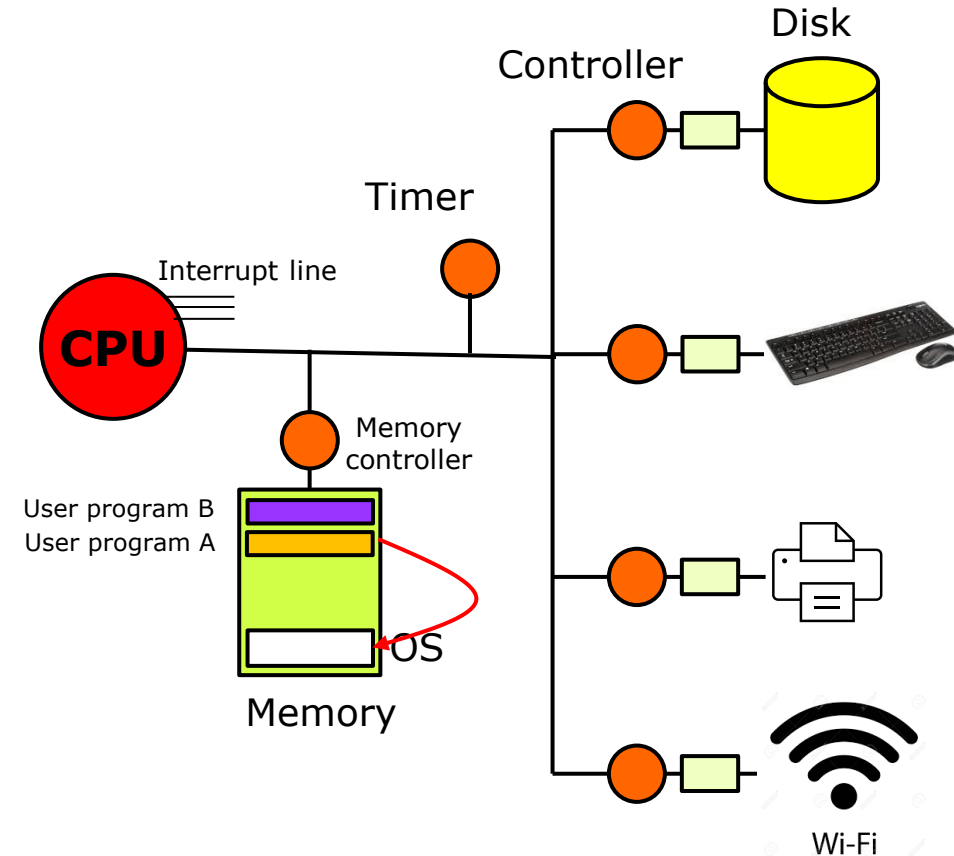
---

- Privileged instructions only executable in kernel mode (not in user mode)
  - **Privileged instructions:** Instructions that only OS can execute
    - ▶ e.g., request I/O hardware, file transfer etc.
- What happens if a user program tries to execute a privileged instruction?
  - Illegal operation! Software interrupt (trap) occurs.
  - Hardware traps to the operating system – terminate program abnormally and give error message
- But what if a user program wants to use the hardware? (e.g., send I/O request to read file)
  - Ask politely to OS – **System Calls**



# System Calls

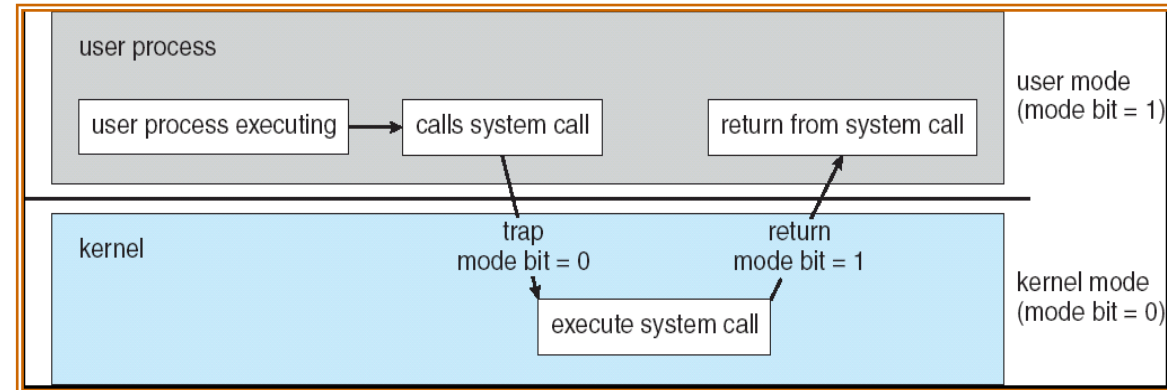
- When user program
  - reads file from disk
  - uses printer
  - needs to use Internet (Wi-Fi)
  - ...
- Need System Calls!
  - Since the above are all “privileged instructions”



# System Calls (Ch. 1.4.2 & 2.3)

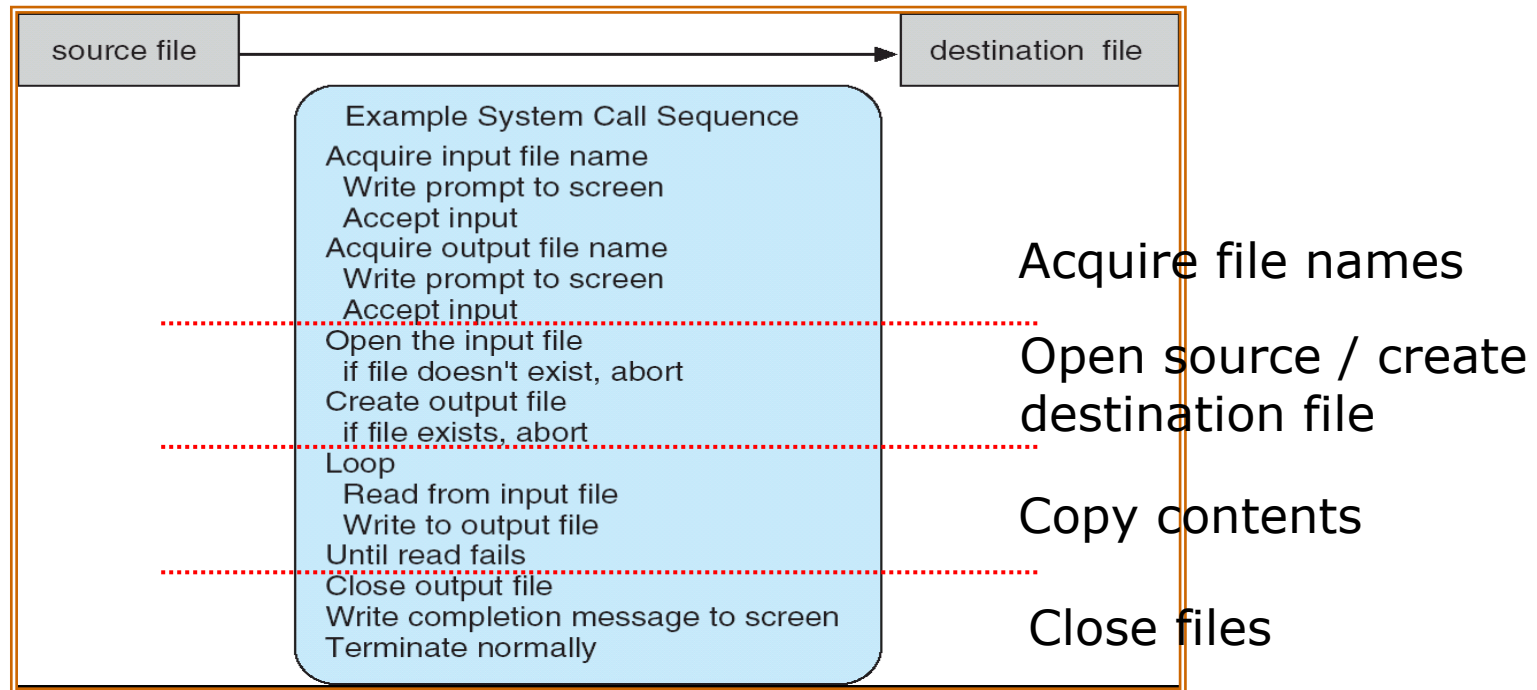
## ■ System Calls

- A request from user program to the operating system to perform some task
  - **Software interrupt**
- Provide **interface** for the **user program** and the services provided by the **OS**



# Example of System Calls: File Copy

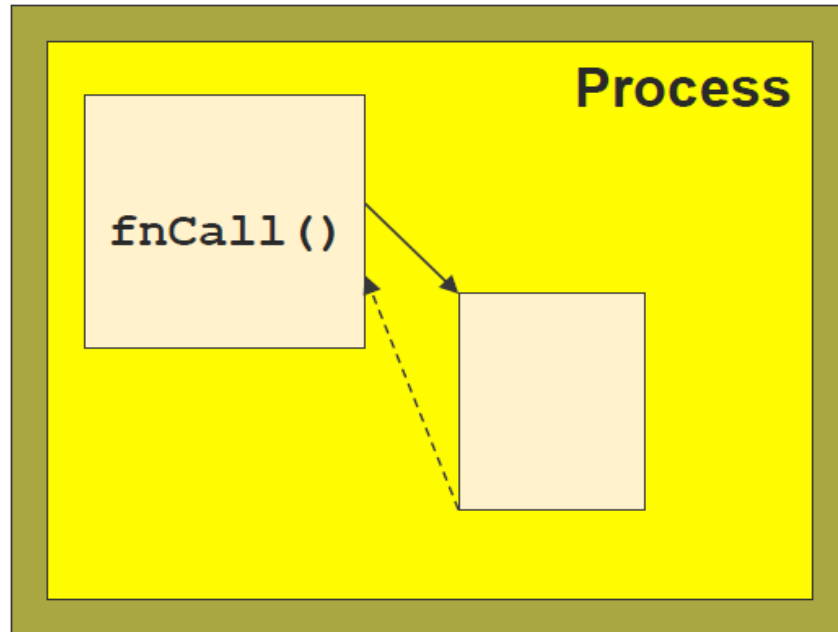
- Requires sequence of multiple system calls to **make file copy**



- Even simple programs make heavy use of the operating systems!
- Thousands of systems calls per second generally happen in an OS
- Read more details in pp. 62-63 (Ch. 2.3)

# Function Call vs. System Call

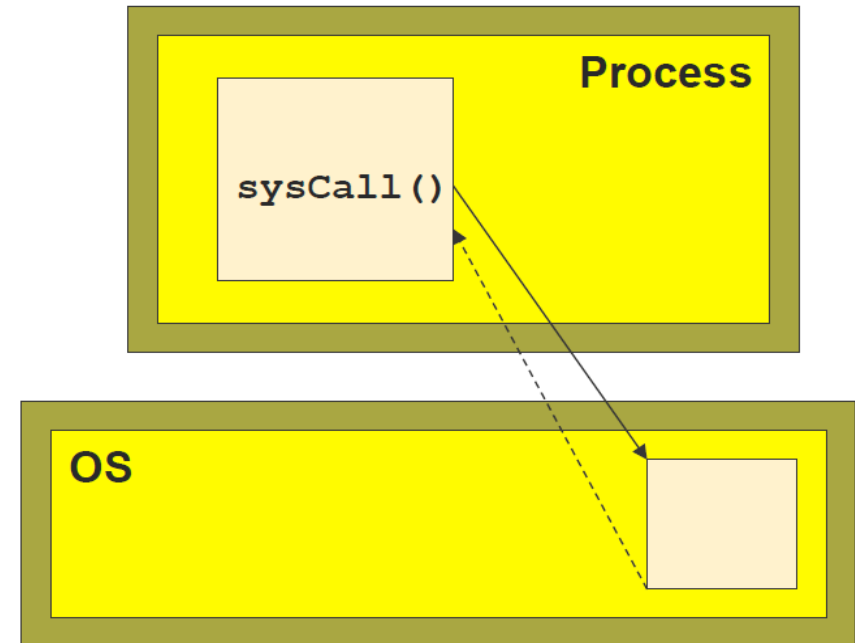
Function call



Caller and callee are in the same Process

- Same user
- Same "domain of trust"

System call



- OS is trusted; user is not.
- OS has super-privileges; user does not
- Must take measures to prevent abuse

# Timer (Ch. 1.5.2)

---



- OS needs maintain control over CPU
  - User program can go into infinite loop or not return system resources
  
- **Timer**
  - Timer is set to **interrupt** the computer after some time period
  - Keep a counter that is decremented by the physical clock
    - ▶ Operating system sets the counter (privileged instruction)
    - ▶ When counter zero, generate an interrupt
  - Set up before scheduling process to regain control that exceeds allotted time

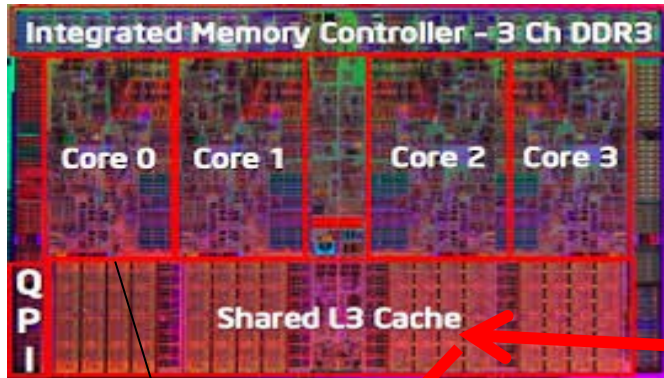
# Chapter 2: System Structures

---

- Basic Operations
  - Multitasking (Ch. 1.4)
  - Interrupt (Ch. 1.2)
  - Protection
    - ▶ Dual-mode operation (Ch. 1.4.2)
    - ▶ Types of System Calls
  - Cache Management (Ch. 1.5.5)
- Operating System Structure & Data Structures

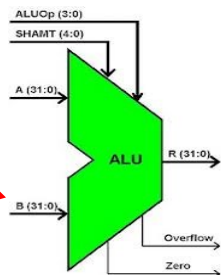
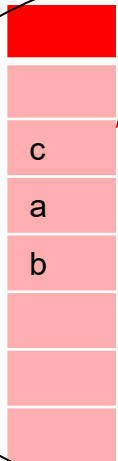
# Basics: From Disk to Memory to CPU

## CPU

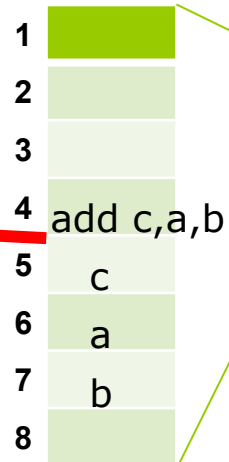


## Cache

## Register



## RAM



## Disk

[C code]  
`c=a+b;`



# Caching

---

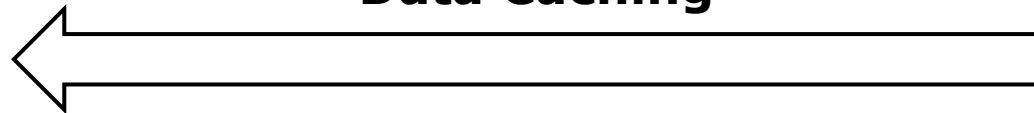
- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information **in use** copied from slower to faster storage temporarily
  - Example: (previous page) variable 'c' is copied from RAM to cache memory
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache smaller than storage being cached
  - Cache management important design problem
  - Cache size and replacement policy



# Characteristics of Various Types of Storage

Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	25,000-50,000	5,000,000
Bandwidth (MB/sec)	20,000-100,000	5,000-10,000	1,000-5,000	500	20-150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

## Data Caching



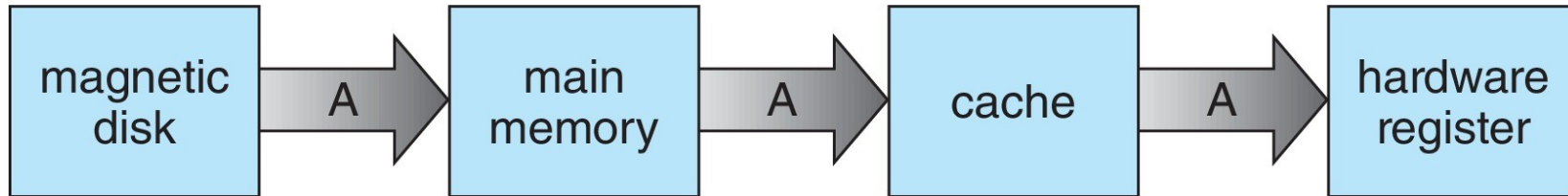
# Caching

---

- Cache smaller than storage being cached
  - **Cache management** important design problem
  - **Cache size** and **replacement policy**
  - But this is a hardware issue: OS? or Computer Architecture?
- Hardware (Computer Architecture) issues in caching
  - Main memory (RAM) → Cache memory
  - Cache memory → CPU (Register)
- Software (OS) issues in caching
  - Disk → Main memory (RAM)

# Migration of data “A” from Disk to Register

- Load integer A (in file B: disk), and increment by 1:



- A++: Increment is done on hardware (CPU) register
  - A differs in the various storage systems
  - Becomes same only after new value of A is copied from register all the way to disk
  - **Cache coherence** (균 일) problem becomes complex in multitasking environment (discussed later on Ch. 9-10)

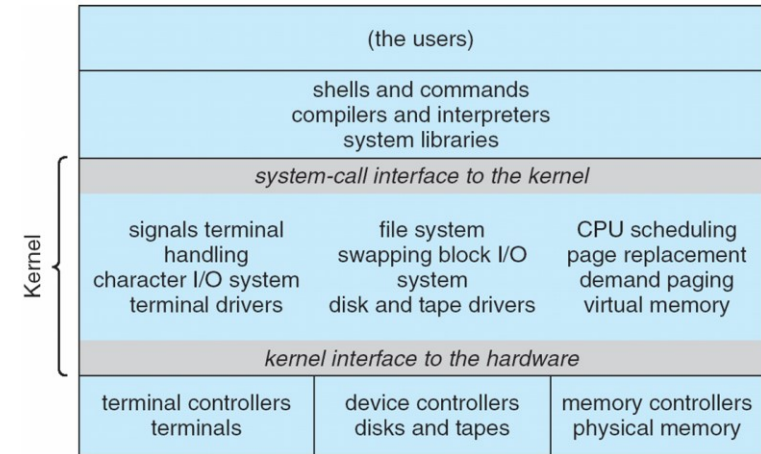
# Chapter 2: System Structures

---

- Basic Operations
  - Multitasking (Ch. 1.4)
  - Interrupt (Ch. 1.2)
  - Protection
    - ▶ Dual-mode operation (Ch. 1.5)
    - ▶ Types of System Calls
- Operating System Structure & Data Structures

# Monolithic structure

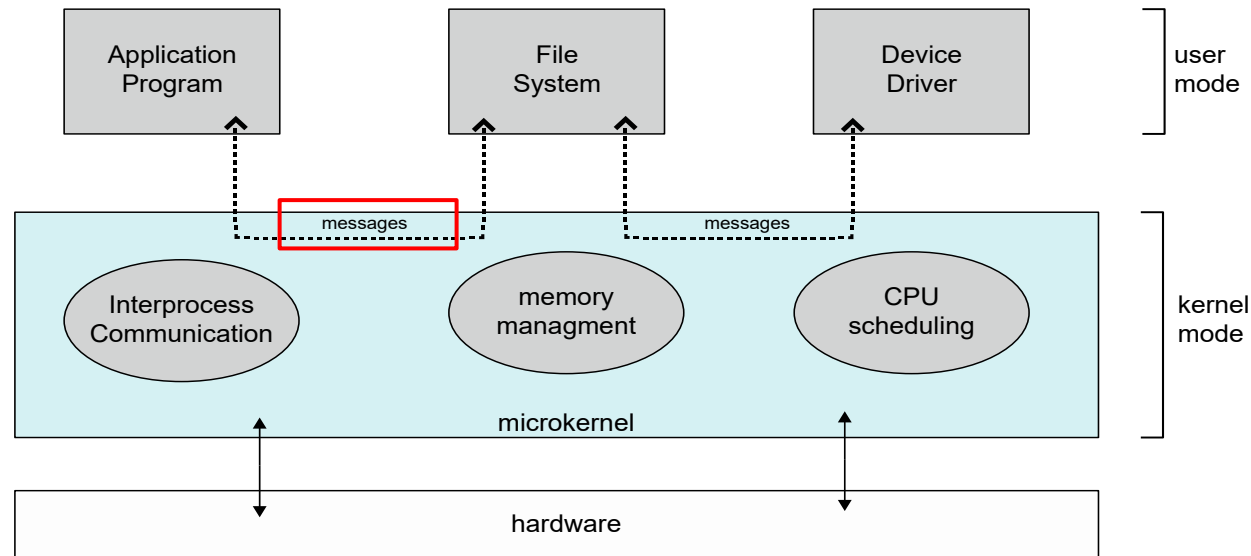
- Place all functionality of the kernel into a single, static binary file that runs in single address space
- Common technique for designing operating systems
  - UNIX, Linux, Windows still use this structure in part.
- Advantages: Performance is good –system call/kernel function is fast
- e.g., UNIX
  - Generally used for (Multi-user) Servers.
  - Source code written in C. (C language was invented to develop UNIX)
  - Base for various OSs (e.g., Linux, MAC OS).



# Microkernel System Structure

## ■ Motivation

- Monolithic made the kernel too complex – as UNIX expanded.
- Move non-essential components of the kernel into user space (system and user-level programs)
- client program need to communicate with various services through **message passing** – increased system-function overhead



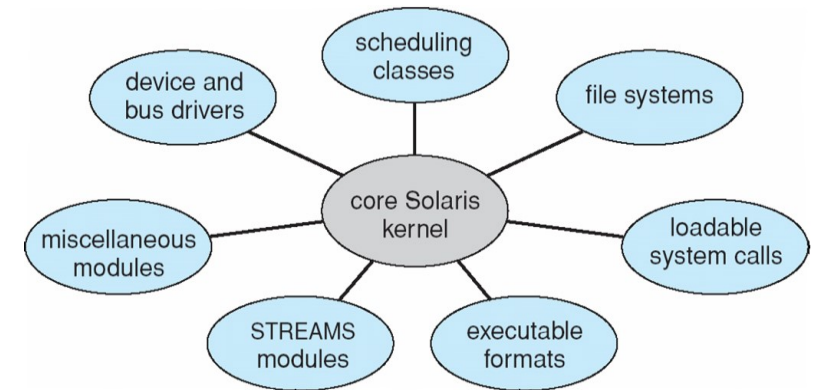
# Microkernel System Structure

---

- Results
  - smaller kernel!
    - ▶ microkernel provides only minimal process and memory management, communication facility
    - ▶ extending kernel is easier – all new services are added to user space and do not need modification at kernel
- **Darwin:** Mac OS kernel, iOS
  - partly based on Mach microkernel
- **Windows NT 4.0** was microkernel based but very slow... why?

# Loadable Kernel Modules (LKM)

- Most modern operating systems implement **loadable kernel modules**
  - Closer to monolithic – most functions in kernel
  - However, each function is loadable as needed within kernel, as the kernel is running
  - e.g., Linux device drivers and file systems

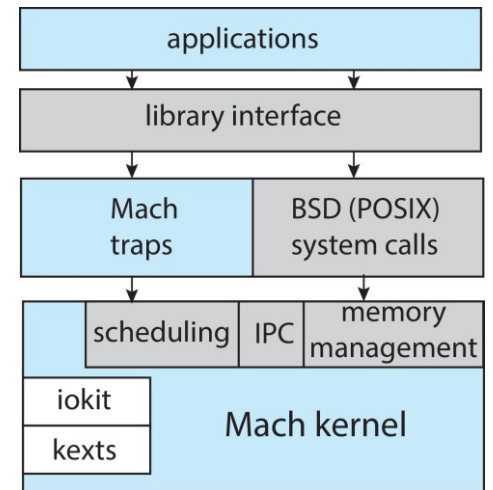


- Advantages
  - Adding new features does not require recompilation of entire kernel – as in monolithic kernel
  - User program can talk directly to LKMs via system call, no need for message passing – as in microkernel



# Hybrid Systems

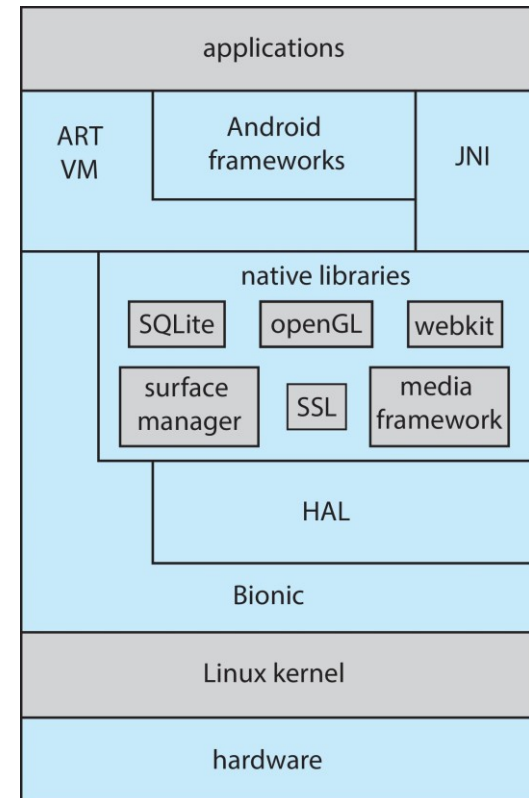
- Most modern operating systems actually not one pure model
- Hybrid combines multiple approaches to address performance, security, usability needs
- Apple macOS and iOS
  - Darwin kernel: Mach microkernel + BSD UNIX kernel



# Hybrid Systems: Android

## ■ Android

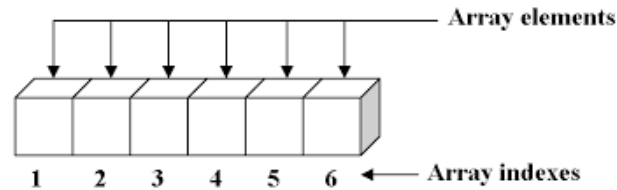
- Developed by Open Handset Alliance (mostly Google)
- Based on **Linux kernel** but modified
  - ▶ Provides process, memory, device-driver management
  - ▶ Adds power management
- Apps developed in **Java** using Android API
  - ▶ Java applications are executed on the Android RunTime (ART)



# Kernel Data Structures: Lists (Ch. 1.10)

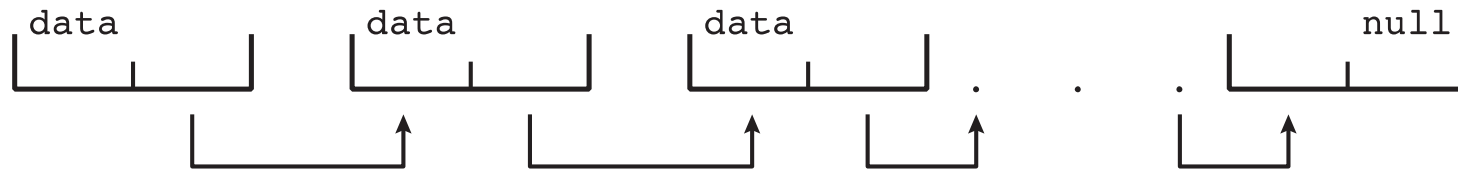
- Used by kernel algorithms

- Array*

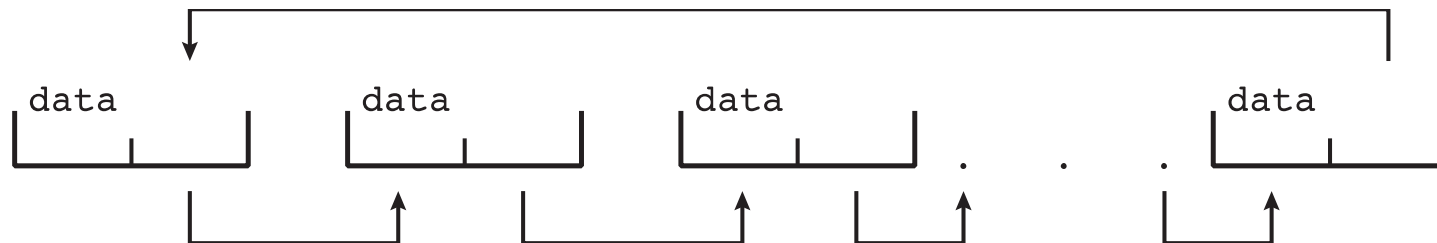


One-dimensional array with six elements

- Singly linked list*



- Circular linked list*

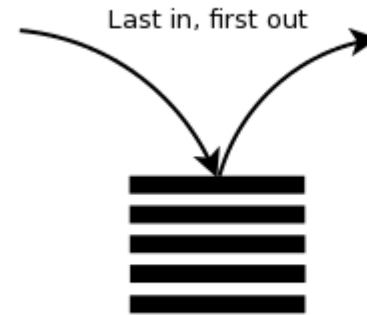


# Kernel Data Structures: Stack/Queue

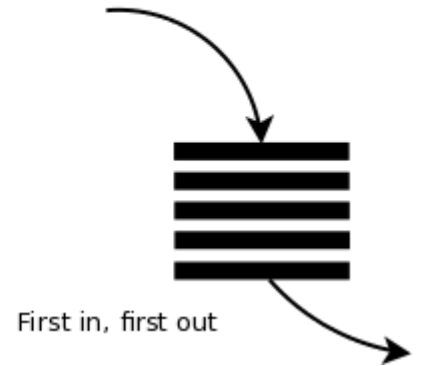
## ■ Stack: LIFO

- Push / Pop
- Invoking **function calls**
  - ▶ Parameters, local variables, and return address are **pushed** on to stack when a function is called
  - ▶ Returning from the function call **pops** those items off the stack

Stack:

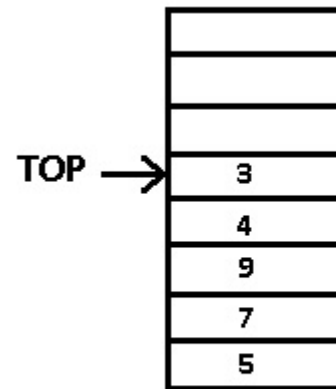


Queue:

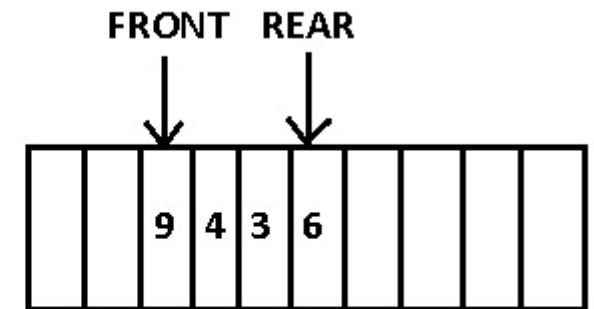


## ■ Queue: FIFO

- Printer jobs
- CPU scheduling



STACK



QUEUE

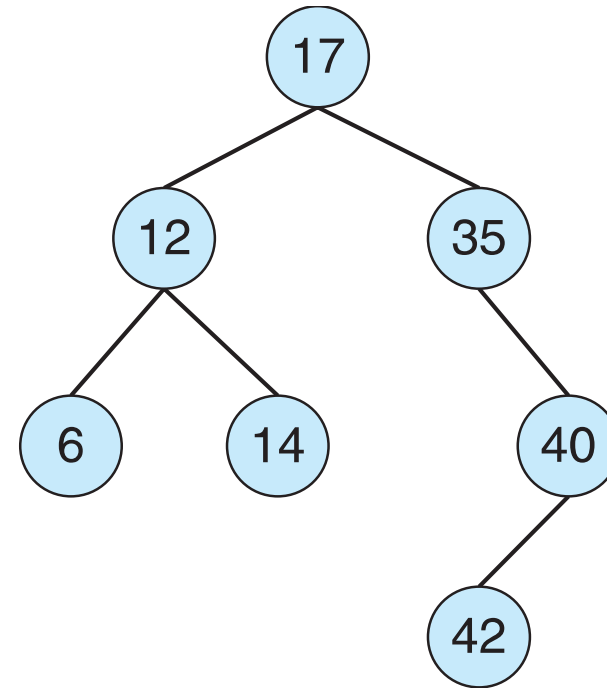
# Kernel Data Structures: Trees

- **Binary search tree**

left  $\leq$  right

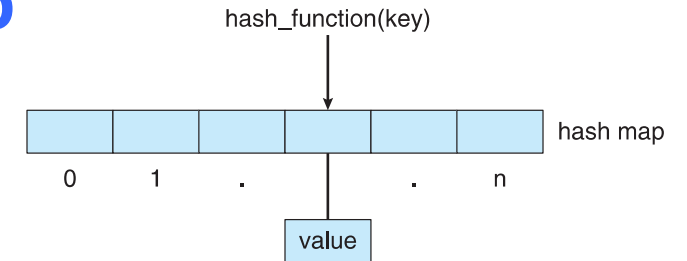
- Search performance is  $O(n)$
- **Balanced binary search tree** is  $O(\log n)$

- Used for Linux CPU scheduling



# Kernel Data Structures: Hash

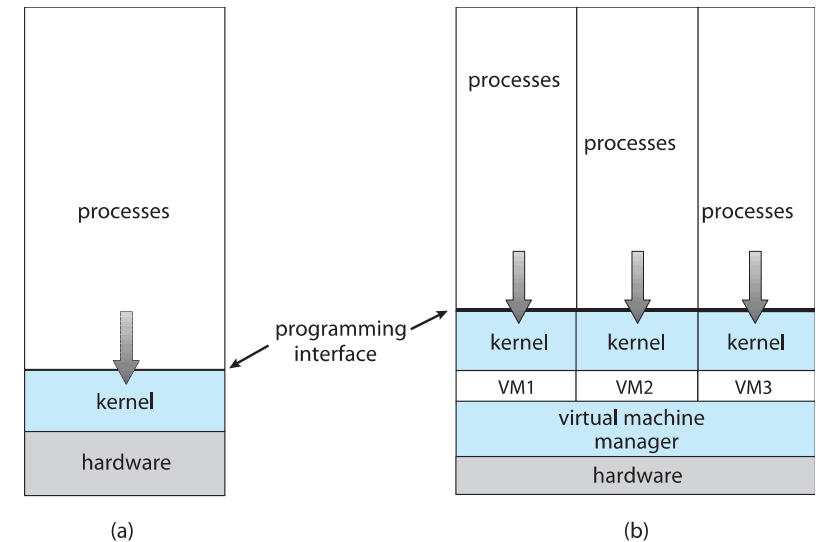
- **Hash function** can create a **hash map**
  - Used in Inverted page table



- Linux (open source) data structures defined in ***include*** files `<linux/list.h>`, `<linux/kfifo.h>`, `<linux/rbtree.h>`

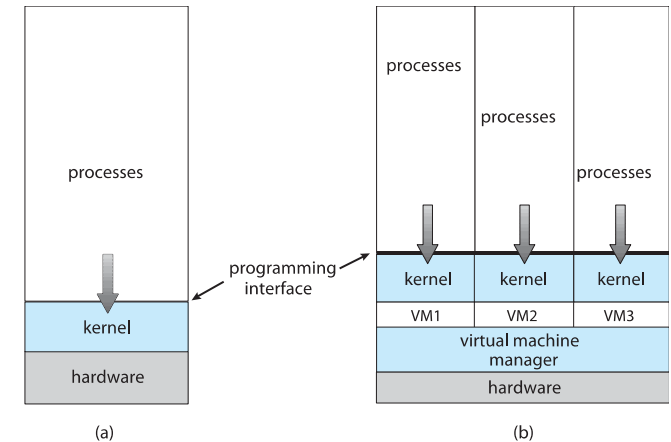
# Virtualization (Ch. 1.7)

- Allows operating systems to run applications within other OSes
  - Vast and growing industry
- **Virtualization** –running **guest** OSes also natively compiled
  - Consider VMware running Windows guests, each running applications, all on native Windows **host** OS
  - **VMM** (virtual machine Manager) provides virtualization services

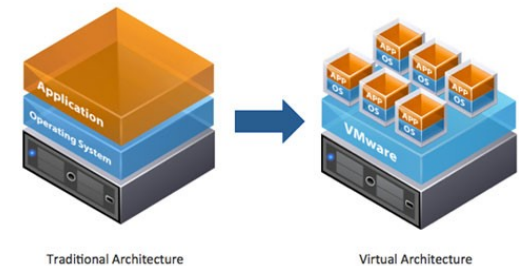


# Virtualization (cont.)

- Use cases involve laptops and desktops running multiple OSES for exploration or compatibility
  - Apple laptop running Mac OS X host, Windows as a guest
  - Developing apps for multiple OSES without having multiple systems
- Executing and managing compute environments within data centers
  - VMM can run natively, in which case they are also the host



Virtualization Defined  
For those more visually inclined...

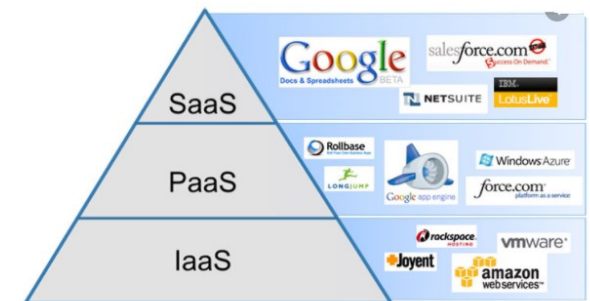


<https://analytica.com/datacenter-capacity-planning-and-modeling-virtualization/>



# Computing Environments – Cloud Computing (Ch.1.10.5)

- Delivers computing, storage, even apps as a service across a network
- Logical extension of virtualization because it uses virtualization as the base for its functionality.
  - Amazon **EC2** has thousands of servers, millions of virtual machines, petabytes of storage available across the Internet, pay based on usage
- Many types
  - **Public cloud** – available via Internet to anyone willing to pay
  - **Private cloud** – run by a company for the company's own use
  - **Hybrid cloud** – includes both public and private cloud components
  - Software as a Service (**SaaS**) – one or more applications available via the Internet (i.e., word processor)
  - Platform as a Service (**PaaS**) – software stack ready for application use via the Internet (i.e., a database server)
  - Infrastructure as a Service (**IaaS**) – servers or storage available over Internet (i.e., storage available for backup use)



<https://cloudcomputinggate.com/saas-paas-iaas-examples/>

# Computing Environments – Cloud Computing

- Cloud computing environments composed of traditional OSeS, plus VMMs, plus cloud management tools
  - Internet connectivity requires security like firewalls
  - Load balancers spread traffic across multiple applications

