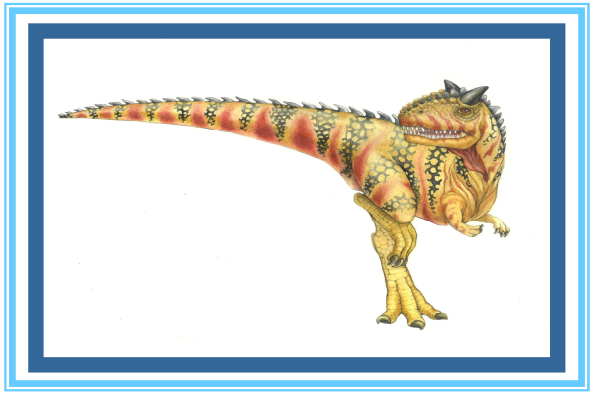
Chapter 2: System Structures

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.

Objectives

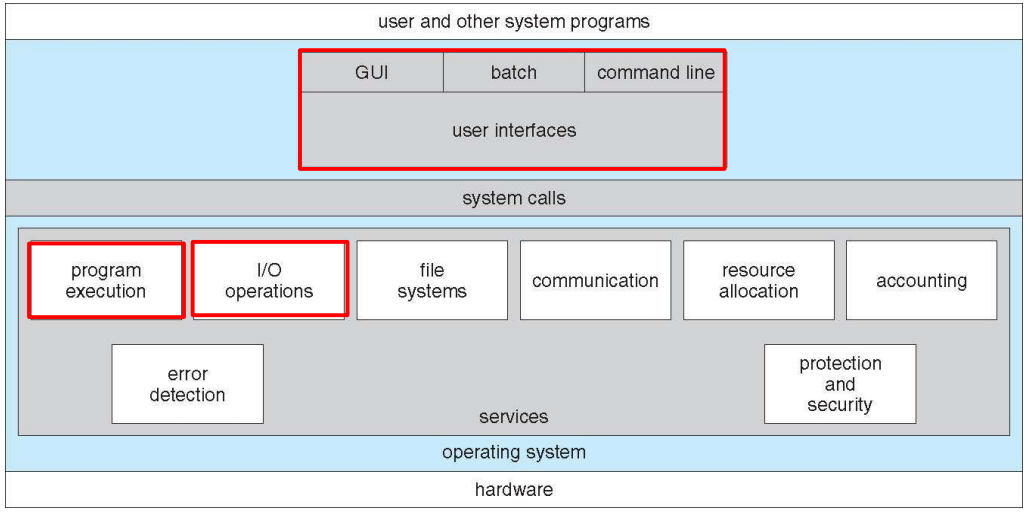
- To describe the services an operating system provides to users
- To discuss the concepts of multitasking, interrupt, protection and system calls
- To discuss the various ways of structuring an operating system

Chapter 2: System Structures

- Operating System Services
- Basic Operations
  - Interrupt
  - Multitasking
  - Protection
    - Dual-mode operation
    - Types of System Calls
  - Timer
- Operating System Structure
- Kernel Data Structures

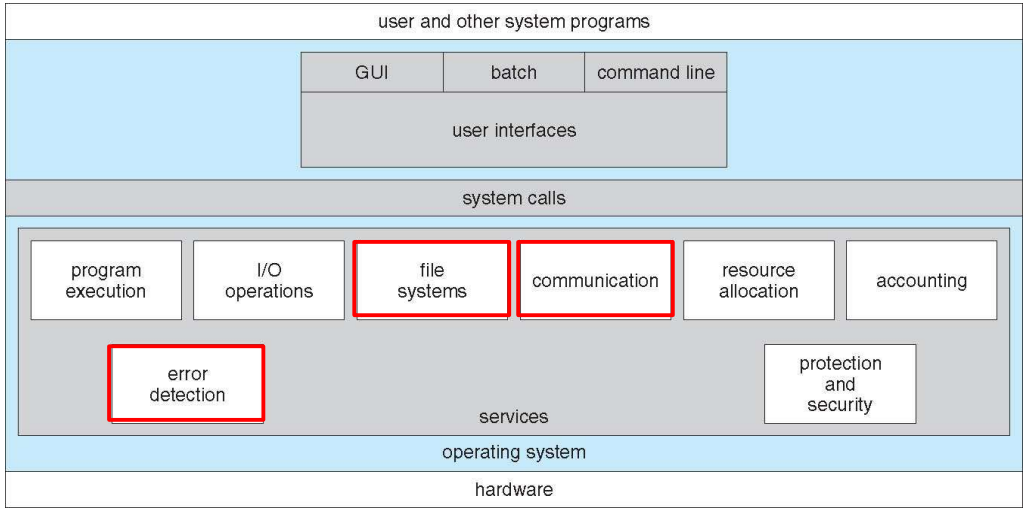
OS Services to User

- Operating systems provide **services** to programs and users
- One set of operating-system services provides functions that are **helpful to the user**:
  - **User interface (UI)**
    - Almost all operating systems have a user interface (**UI**): **Command-Line (CLI)**, **Graphics User Interface (GUI)**
  - **User program execution**
    - The system must be able to **load** a program into memory and to **run** that program (=process), **end** execution
  - **I/O operations**
    - A running program may require **I/O**, which may involve a file or an I/O device



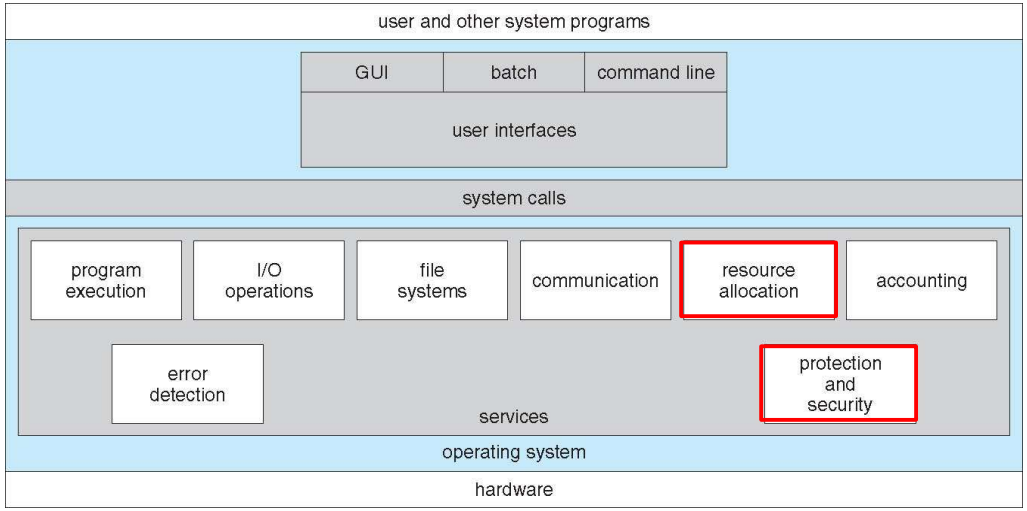
OS Services to User

- **File-system manipulation**
  - Programs need to read and write **files** and directories,…
- **Communications**
  - Processes may exchange information, on the same computer or between computers over a **network**
  - Communications may be via shared memory or through message passing (packets moved between processes by the OS)
- **Error detection**
  - OS needs to be constantly aware of possible **errors**
  - May occur in the CPU and memory hardware, in I/O devices, in user program
  - For each type of error, OS should take the appropriate action to ensure correct and consistent computing



OS Services for resource management

- Another set of OS functions exists for ensuring the **efficient operation** of the system itself via **resource sharing**
  - **Resource allocation**
    - **Resources:** CPU cycles, main memory, file storage, others (such as I/O devices)
    - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
  - **Protection and security**
    - **Protection**
      - OS resources is protected
      - concurrent processes should not interfere with each other
    - **Security**
      - from outsiders:
        requires user authentication, extends to defending external I/O devices from invalid access attempts



Chapter 2: System Structures

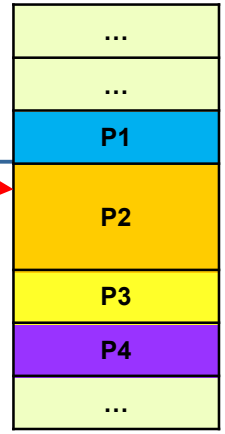
- Operating System Services and Interface
- Basic Operations
  - Interrupt
  - Multitasking
  - Protection
    - Dual-mode operation
    - Types of System Calls
  - Timer
- Operating System Structure
- Kernel Data Structures

Primitive Operating Systems

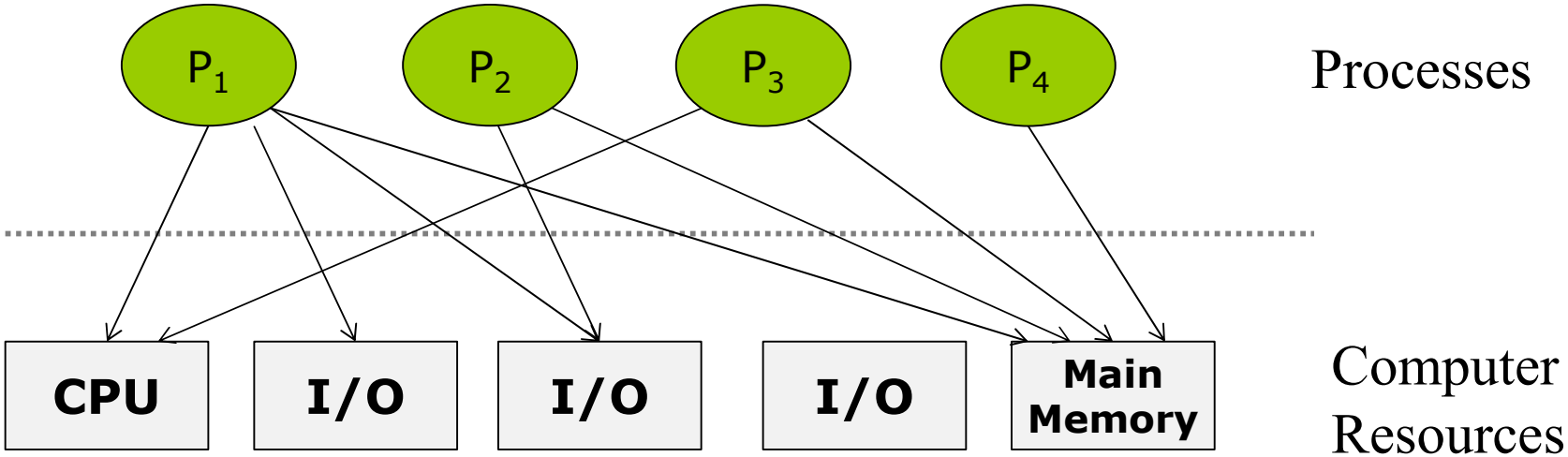
- Just a library of standard services

- Simplifying assumptions
  - System runs one program at a time
  - No bad users or programs (often bad assumption)
- Problem: Poor utilization
  - . . . of hardware (e.g., CPU idle while waiting for disk)
  - . . . of human user (must wait for each program to finish)

Multitasking



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



- Managing and controlling several processes simultaneously are challenging

Challenging Issues

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

What does an OS do?

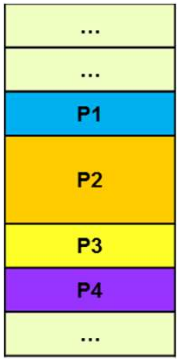
- At any given time, each resource (e.g. CPU) can serve only one process
  - OS → Process Management
- Many processes can be executed while memory is limited
  - OS → Memory Management
- Using I/O device takes a long time, and the rest of the machine is idle and therefore underutilized
  - OS → I/O Management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

Process Management

- A ***process*** *:* a program in execution
  - A process needs certain resources, including CPU time, memory, files, and I/O devices, to accomplish its task
- OS Functionality
  - Create/run/terminate processes
  - Who goes first? Process (thread) scheduling
  - Mechanisms for resource sharing and synchronization

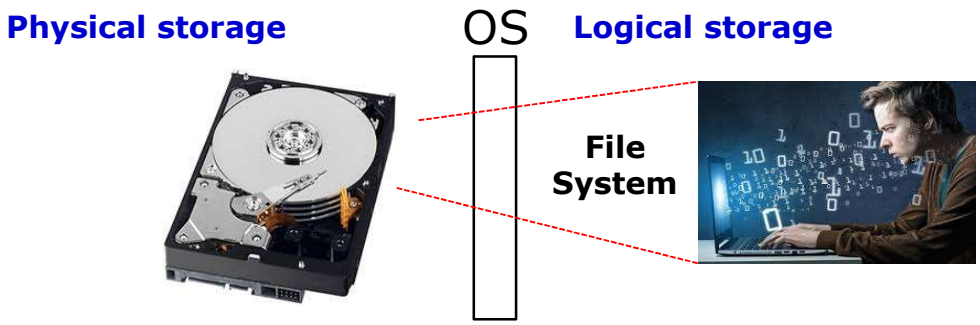
Memory Management

- Main memory (RAM)
  - A large array of bytes, each byte with its own address
  - CPU reads both **instructions** and **data** from the main memory
    – it must first be loaded to the main memory
- **Multitasking**
  - Multiple processes (= tasks, jobs) **in the main memory**
  - Improves utilization of CPU and response to its users
- **(Virtual) Memory management**
  - 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

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

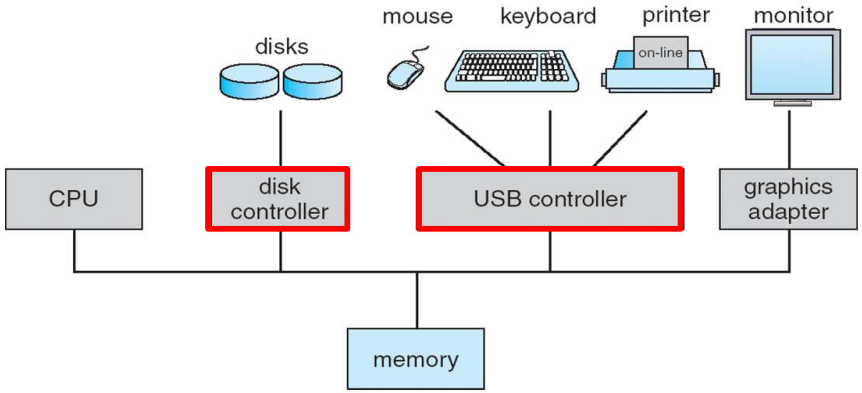


- OS Functionality
  - *File-system management*
  - *Mass-storage (disk) management*

I/O Subsystem

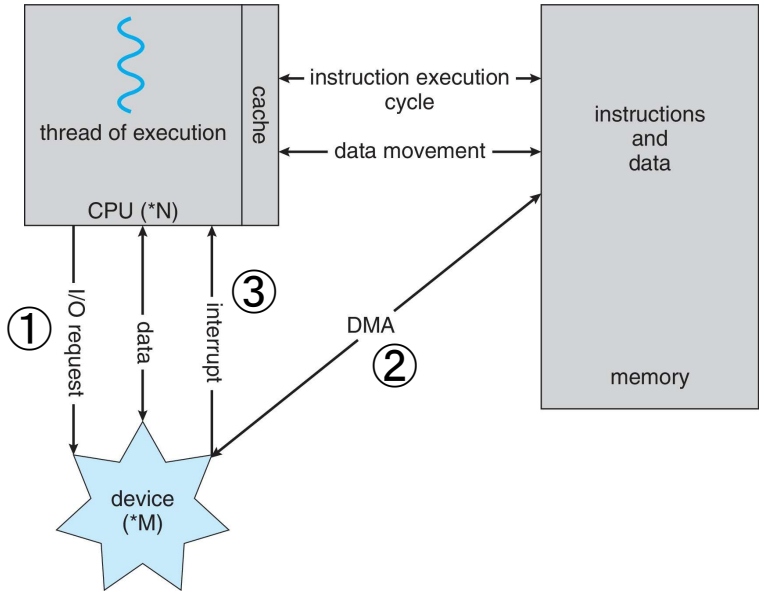
- **Device controller** (in I/O Device)
  - In charge of one or more specific type of device (e.g., disk, USB, network controller).
  - Maintain some local buffer storage
  - Move the data between the peripheral devices and its **local buffer** storage

- **Device driver** (in OS)
  - OS has a device driver for each device controller
  - Understands the controller and provide OS with an interface to device.



I/O Subsystem contd.

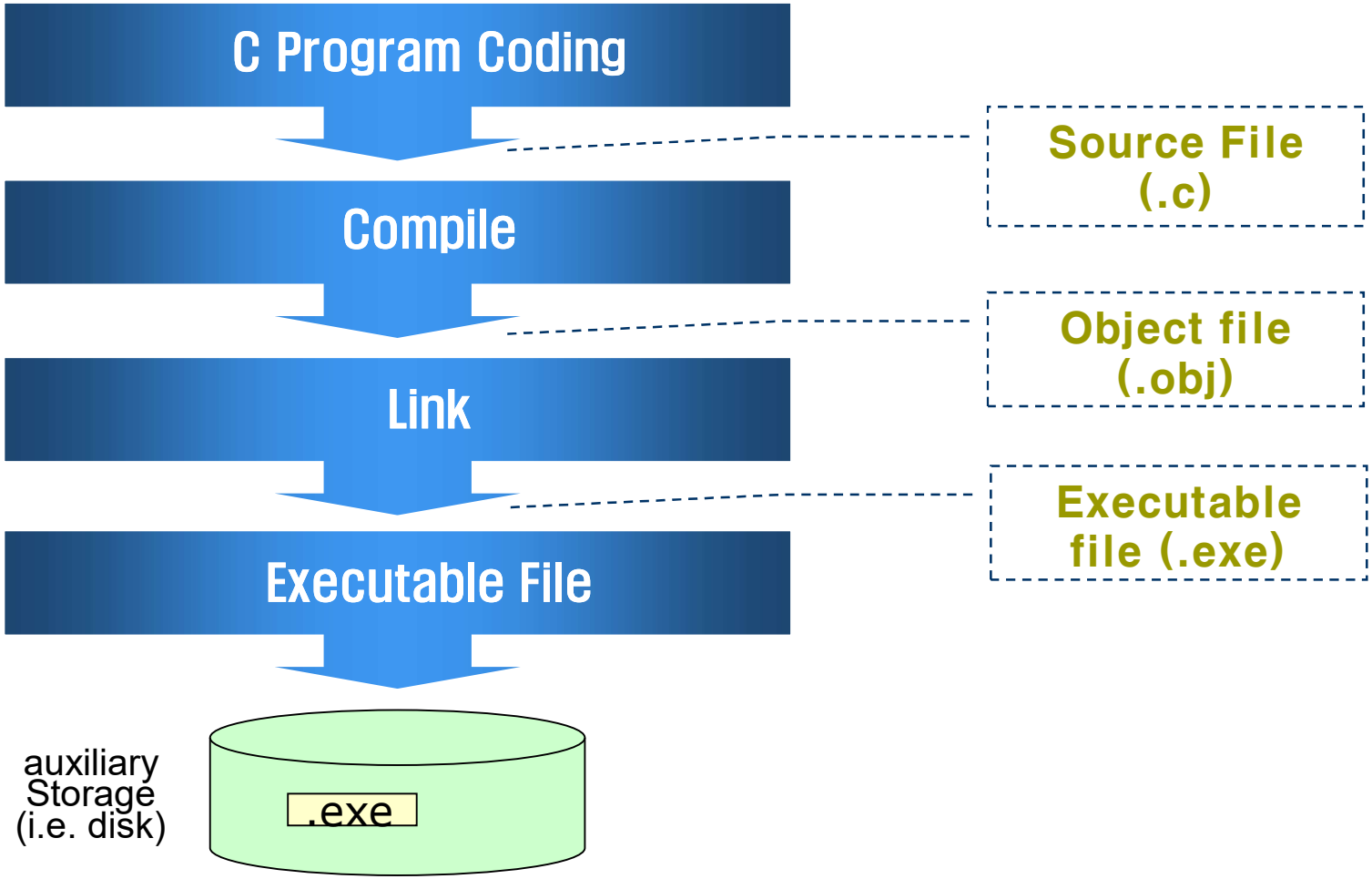
- I/O Operation
  - ① **I/O request** from application (e.g., read file from disk): device driver delivers command to device controller
    - controller determines which action to take
    - e.g., “read block#369 from hard disk”
  - ② **device controller** takes action
    - e.g., read block from disk,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



* DMA: direct memory access

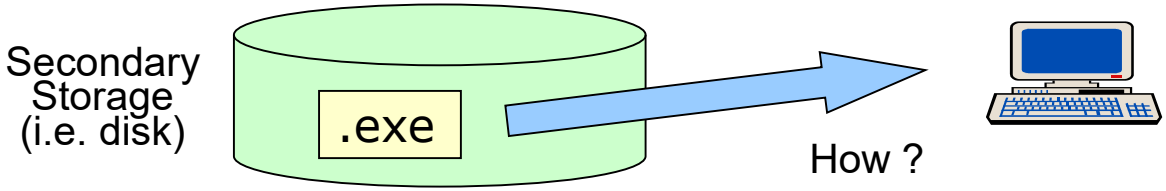
Example: Execution of a C-Program

- C programing step



Execution of a C-Program

- C-program execution sequence



(1) Input Command (name of program)

(2) Interpretation of the command

(3) Seek the file & read data from the disk

(4) Load the data into main memory

(5) Process creation & execution

(6) Print output on a monitor (I/O device)

What OSs Do for the execution?

(1) Input Command (name of program)

(2) Interpretation of the command

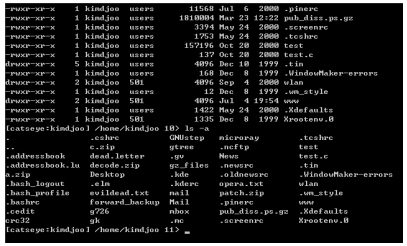
(3) Seek the file & read data from the disk

(4) Load the data into main memory

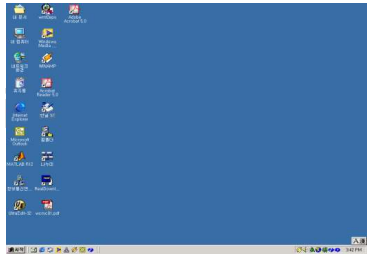
(5) Process creation & execution

(6) Print output on a monitor (I/O device)

- OS provide User Interface
  - Command-Line Interface vs. Graphic User Interface



(e.g., type “a.exe”) (e.g., double click



What OSs Do for the execution?

(1) Input Command (name of program)

(2) Interpretation of the command

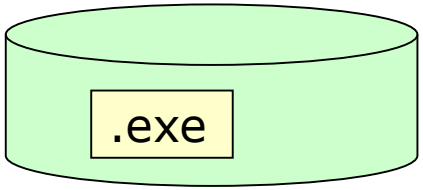
(3) Seek the file & read data from the disk

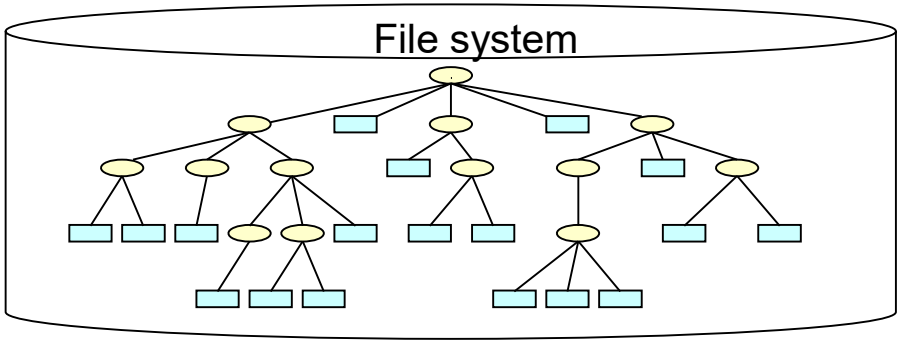
(4) Load the data into main memory

(5) Process creation & execution

(6) Print output on a monitor (I/O device)

- OS : File System Management, Disk Scheduling
  - Management of user and system files, disk scheduling, seek time optimization





What OSs Do for the execution?

(1) Input Command (name of program)

(2) Interpretation of the command

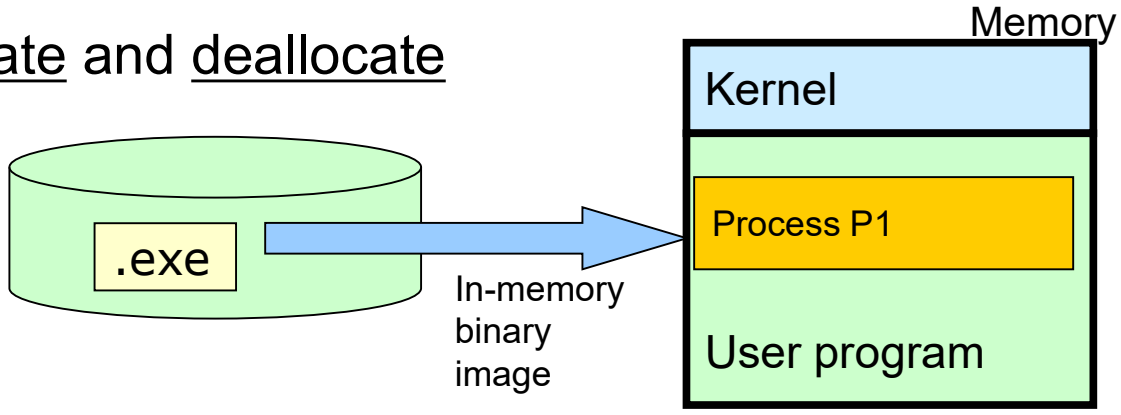
(3) Seek the file & read data from the disk

(4) Load the data into main memory

(5) Process creation & execution

(6) Print output on a monitor (I/O device)

- OS : Memory Management
  - allocate and deallocate



What OSs Do for the execution?

(1) Input Command (name of program)

(2) Interpretation of the command

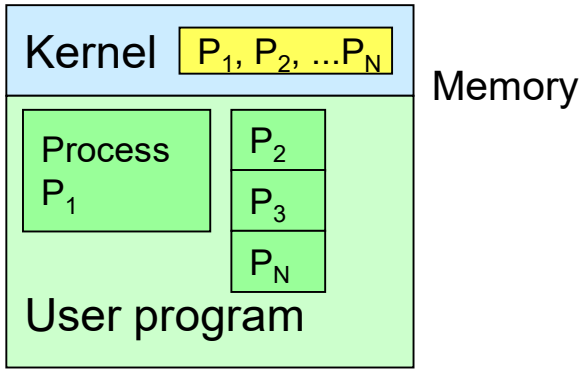
(3) Seek the file & read data from the disk

(4) Load the data into main memory

(5) Process creation & execution

(6) Print output on a monitor (I/O device)

- OS : Process Management
  - Process creation, management, scheduling



What OSs Do for the execution?

(1) Input Command (name of program)

(2) Interpretation of the command

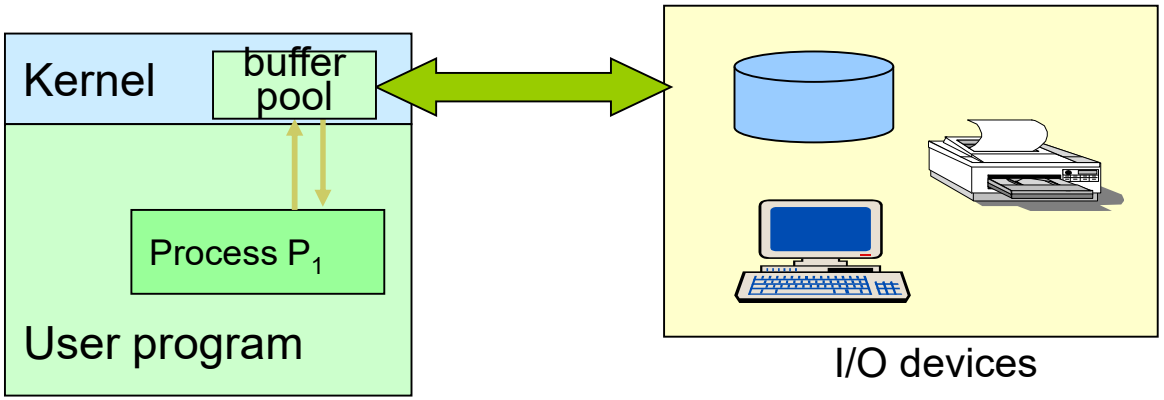
(3) Seek the file & read data from the disk

(4) Load the data into main memory

(5) Process creation & execution

(6) Print output on a monitor (I/O device)

- OS : I/O Management

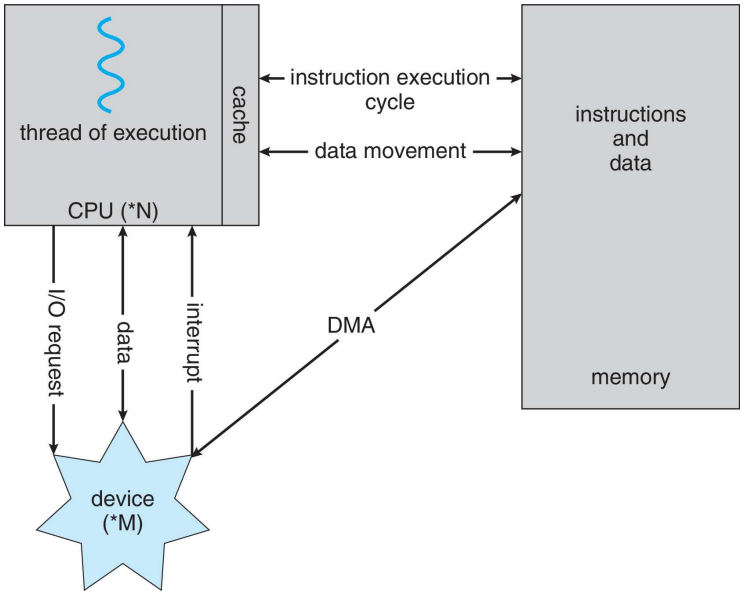


Chapter 2: System Structures

- Operating System Services and Interface
- Basic Operations
  - Interrupt
  - Multitasking
  - Protection
    - Dual-mode operation
    - Types of System Calls
  - Timer
- Operating System Structure
- Kernel Data Structures

Interrupt

- I/O generally takes much longer than CPU cycles
  - CPU and I/O devices can run in parallel (i.e., together)
  - CPU can work on other things while waiting for I/O – better CPU utilization!
- 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!**
- An operating system is **interrupt driven**
  - **Interrupt** is a key part of OS-hardware interaction

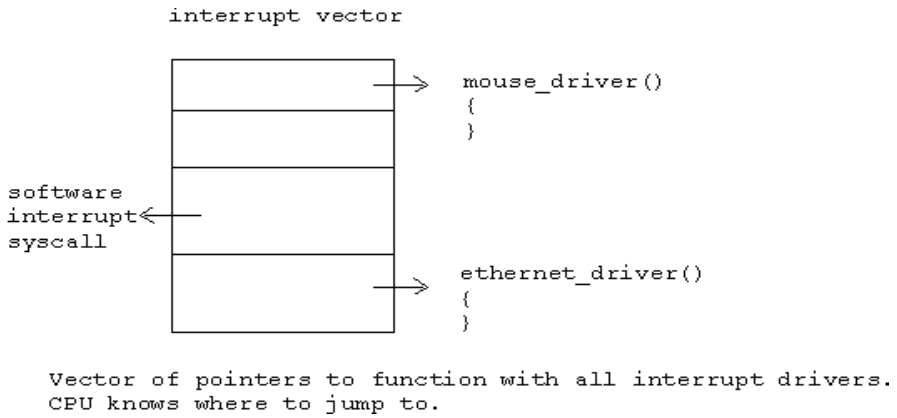


Interrupt

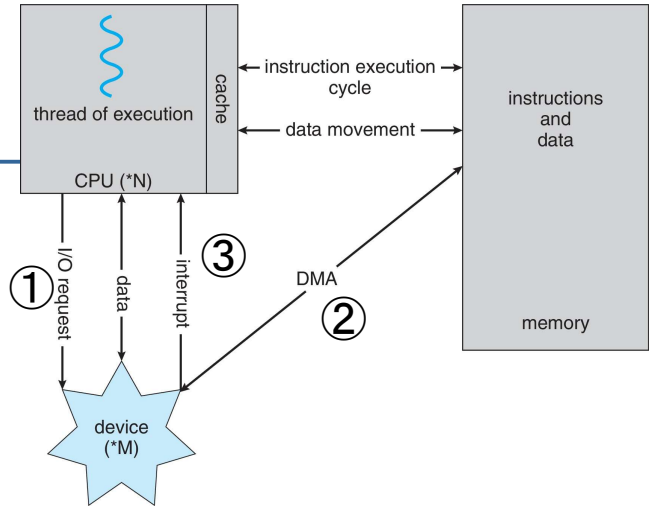
- **Hardware and Software Interrupt**
  - The occurrence of an event (e.g., error, completion of a job, ..) is signaled by an interrupt from H/W or S/W
  - **Hardware Interrupt**: produced by **hardware I/O devices**
    - mouse movement, keyboard, Ethernet packet, disk/CD/Tape driver, timer interrupts (used for scheduling multiple processes), sound (microphone)
  - **Software Interrupt** (= **traps**)
    - **system calls** (by user programs)
    - **exceptions**: divide by zero exception
- An interrupt is an event that requires CPU attention. If an interrupt occurs:
  - The CPU will **stop** anything it is doing and it will jump to an ***interrupt handler***.

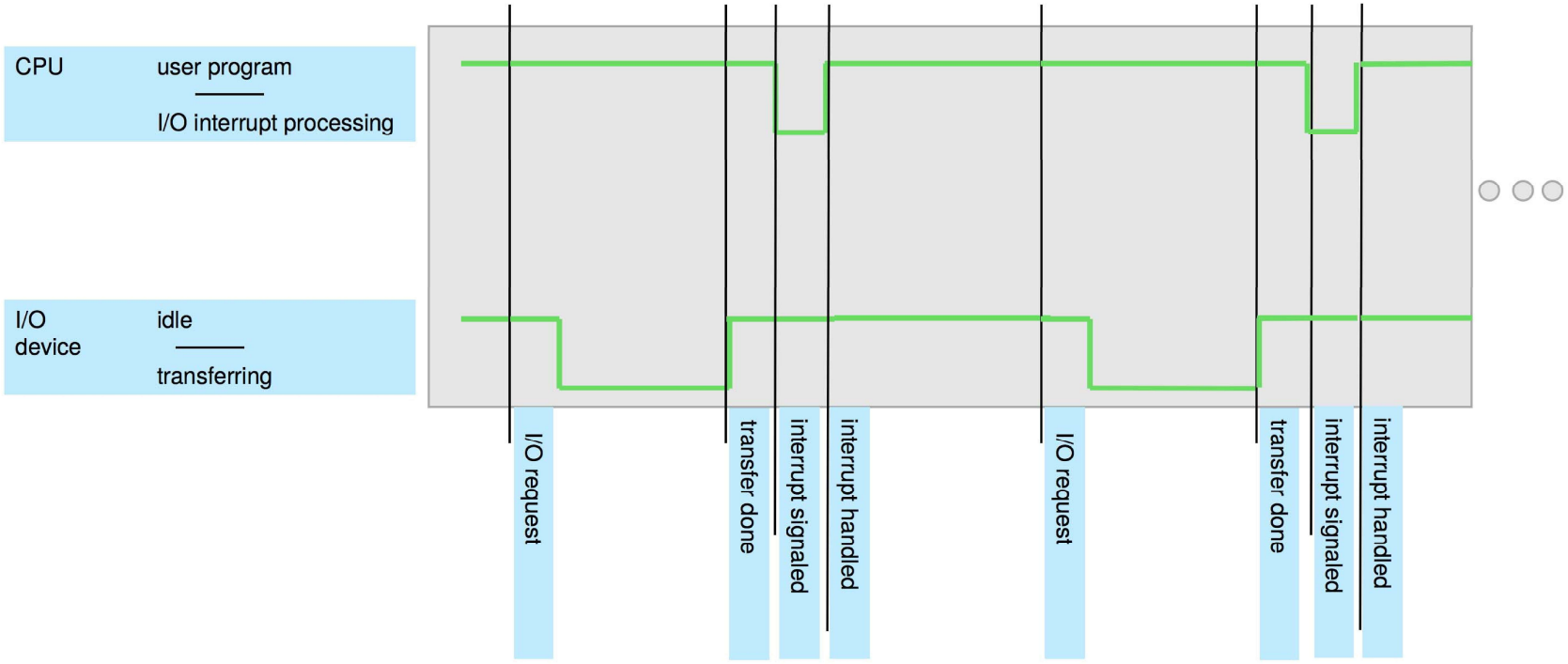
Interrupt Handling

- When the CPU is interrupted, the CPU
  - **stops** what it is doing
  - transfer execution to a **fixed memory location** (called **interrupt vector**)
    - The interrupt vector has the addresses of the **service routine** for the interrupt
    - Usually stored in low memory (first few hundred)
  - 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)



Hardware Interrupt Timeline



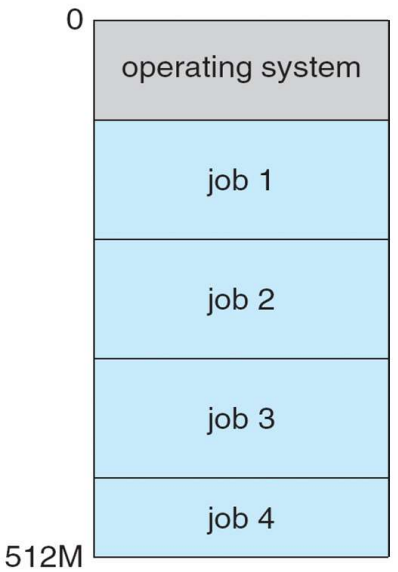


Chapter 2: System Structures

- Operating System Services and Interface
- Basic Operations
  - Interrupt
  - Multitasking
  - Protection
    - Dual-mode operation
    - Types of System Calls
  - Timer
- Operating System Structure
- Kernel Data Structures

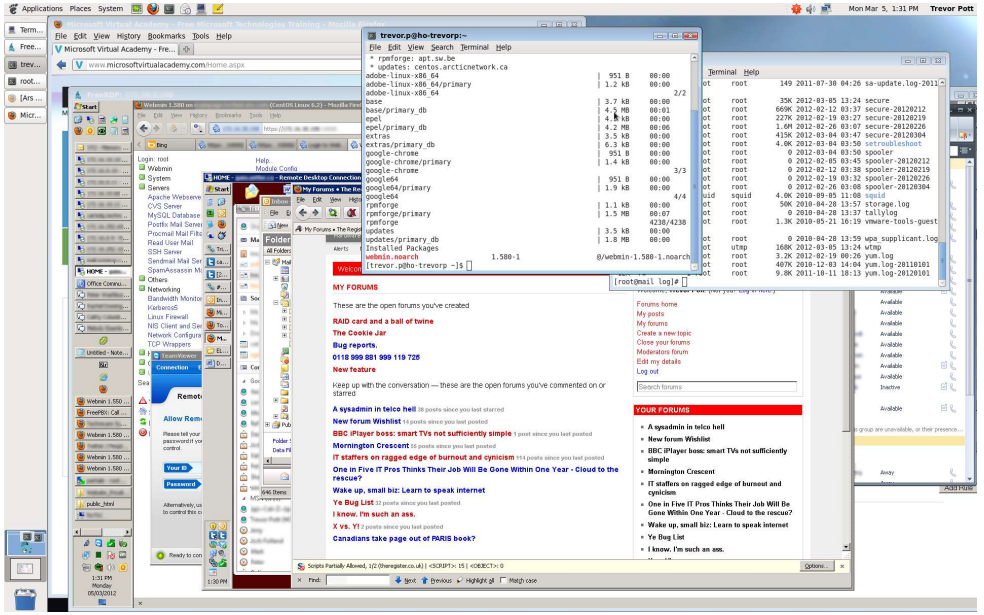
Multitasking

- **Multitasking** (=Multiprogramming, Multiprocessing) needed for efficiency
  - Users frequently have multiple programs running
  - Single program cannot keep CPU and I/O devices busy at all times
- Adavantage1: **CPU utilization**
  - Keep several jobs in memory simultaneously
  - One job selected and run via **scheduling**
  - When it has to wait (e.g., for I/O operation), OS switches to another job
  - As long as at least one job needs to execute, the CPU never stays idle
  - What happens if it is not switched?
    - Hint: Waste CPU cycles (why?)



Multitasking

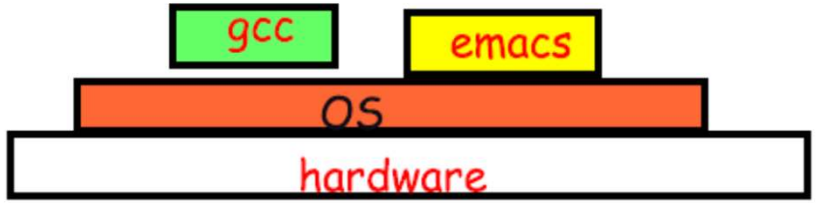
- Advantage 2: **Timesharing**
  - A logical extension in which CPU switches jobs so **frequently** that the user have an **illusion of multitasking**
  - Allows many users to share the computer simultaneously
  - If several jobs ready to run at the same time, which process do we run first? ⇨ **CPU Job scheduling**



Chapter 2: System Structures

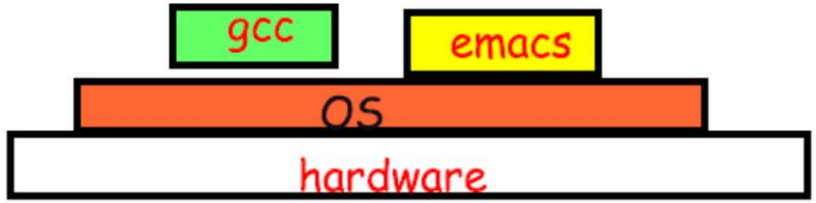
- Operating System Services and Interface
- Basic Operations
  - Interrupt
  - Multitasking
  - Protection
    - Dual-mode operation
    - Types of System Calls
  - Timer
- Operating System Structure
- Kernel Data Structures

What can happen when Multitasking?



- OS and Multi-users share hardware and software resources
  - What if process (=active program) P1 (bug or virus, …) illegally accesses process P2's information? (memory contents)
  - What if process P3 tries to manipulate OS's operation?

What can happen when Multitasking?



- OS and Multi-users share hardware and software resources
  - What if process (=active program) P1 (bug or virus, …) illegally accesses process P2's information? (memory contents)
    - **Memory protection**
  - What if process P3 tries to manipulate OS's operation?
    - **Dual-mode operation**

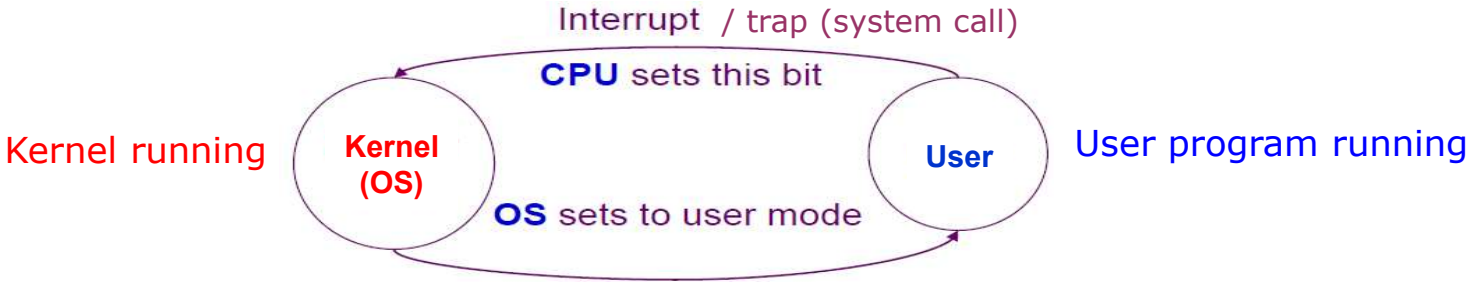
Dual-Mode Operation

- OS **Privileged instructions**
  - Some instructions that may cause harm to others
  - e.g., I/O hardware, file transfer etc.
  - Only OS can run privileged instructions!

- **Dual-mode operation** allows OS to protect itself and other system components
  - **Kernel mode** vs. **User mode**
  - **Mode bit** provided by **hardware support** (e.g., CPU)
    - Provides ability to distinguish when system is running user code or kernel code
    - Privileged instructions only executable in kernel mode (not in user mode)

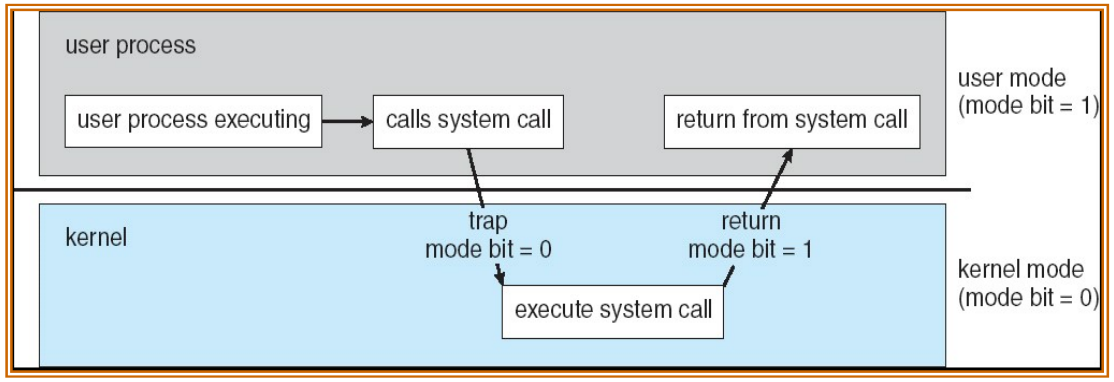
Dual-Mode Operation

- ***Mode bit*** added to computer hardware to indicate the current mode: **kernel (0)** or **user (1)**.
  - When an interrupt/trap occurs, hardware switches to kernel mode (0).



- **System call** changes mode to kernel, return from call resets it to user

  If user wants to use the privileged instructions it invokes **system calls**

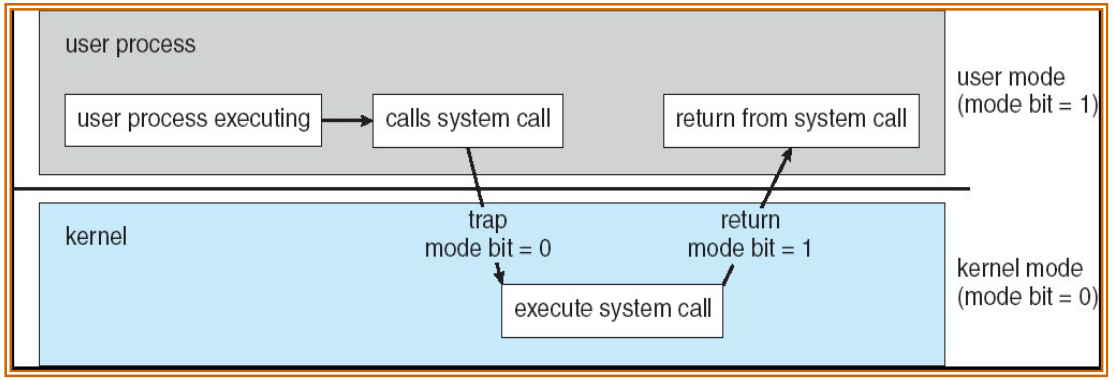


Dual-Mode Operation

- What happens if there is no dual-mode? (single mode)
  - User program has access to all resources (e.g. MS DOS)
- What happens if a user program fails to abide by the dual mode?
  - e.g., attempt to execute illegal instruction by user program
  - e.g., access memory beyond user program's memory address space
  - Hardware *traps* to the operating system
    – terminate program abnormally and give error message
- Life cycle of instruction
  - OS running (kernel mode)
  - User programs running (user mode)
- But what if a user program wants to use the hardware? (e.g., print something...)
  - Ask politely to OS – **System Calls**

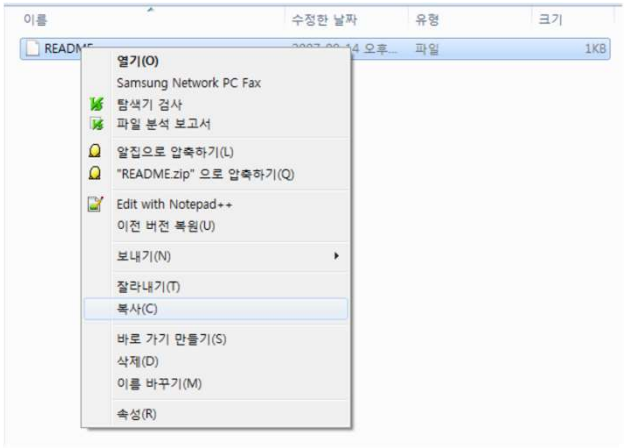
System Calls

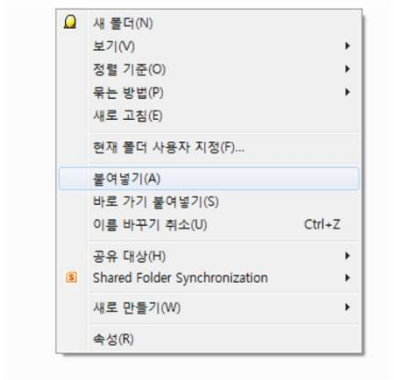
- **System Calls**
  - A request from user program to the operating system to perform some task
  - The task cannot be done by the user program (requires privileged instructions)
- Provide **interface** for the **user program** and the services provided by the **OS**



Example: Very Simple File Copy & Paste

- Windows



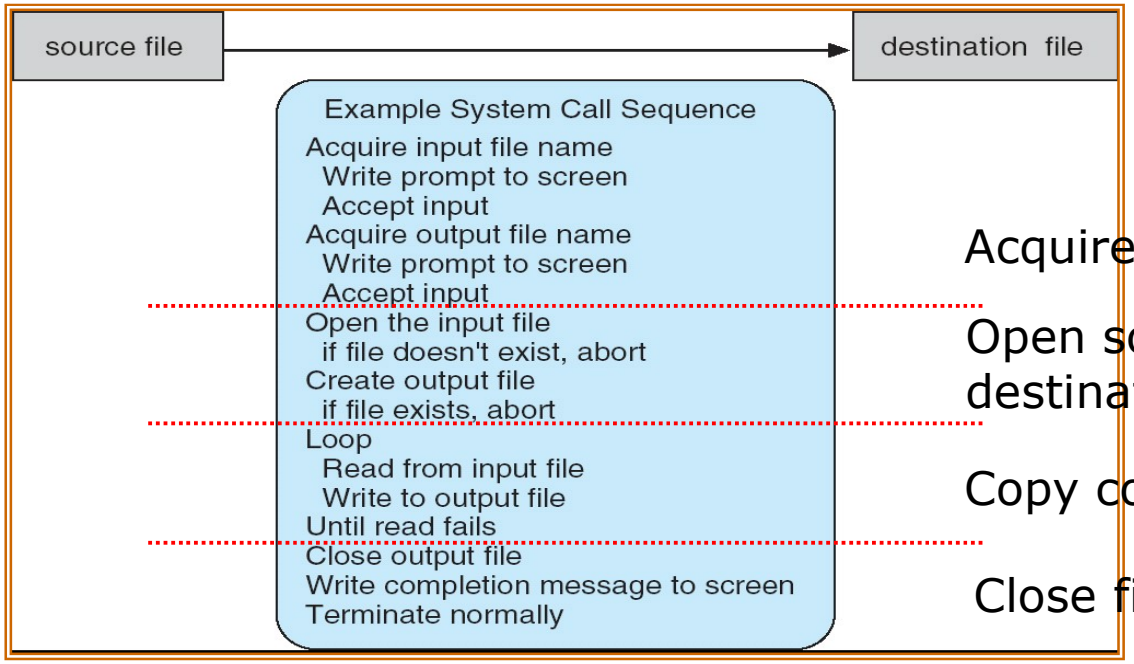


- Linux

```
joonyoo2@ubuntu:~$ ls
Desktop    Downloads         Music     Public     test1.txt
Documents  examples.desktop  Pictures  Templates  Videos
joonyoo2@ubuntu:~$ cp test1.txt test2.txt
joonyoo2@ubuntu:~$ ls
Desktop    Downloads         Music     Public     test1.txt  Videos
Documents  examples.desktop  Pictures  Templates  test2.txt
joonyoo2@ubuntu:~$
```

Example of System Calls: File Copy

- Requires sequence of multiple system calls to make file copy

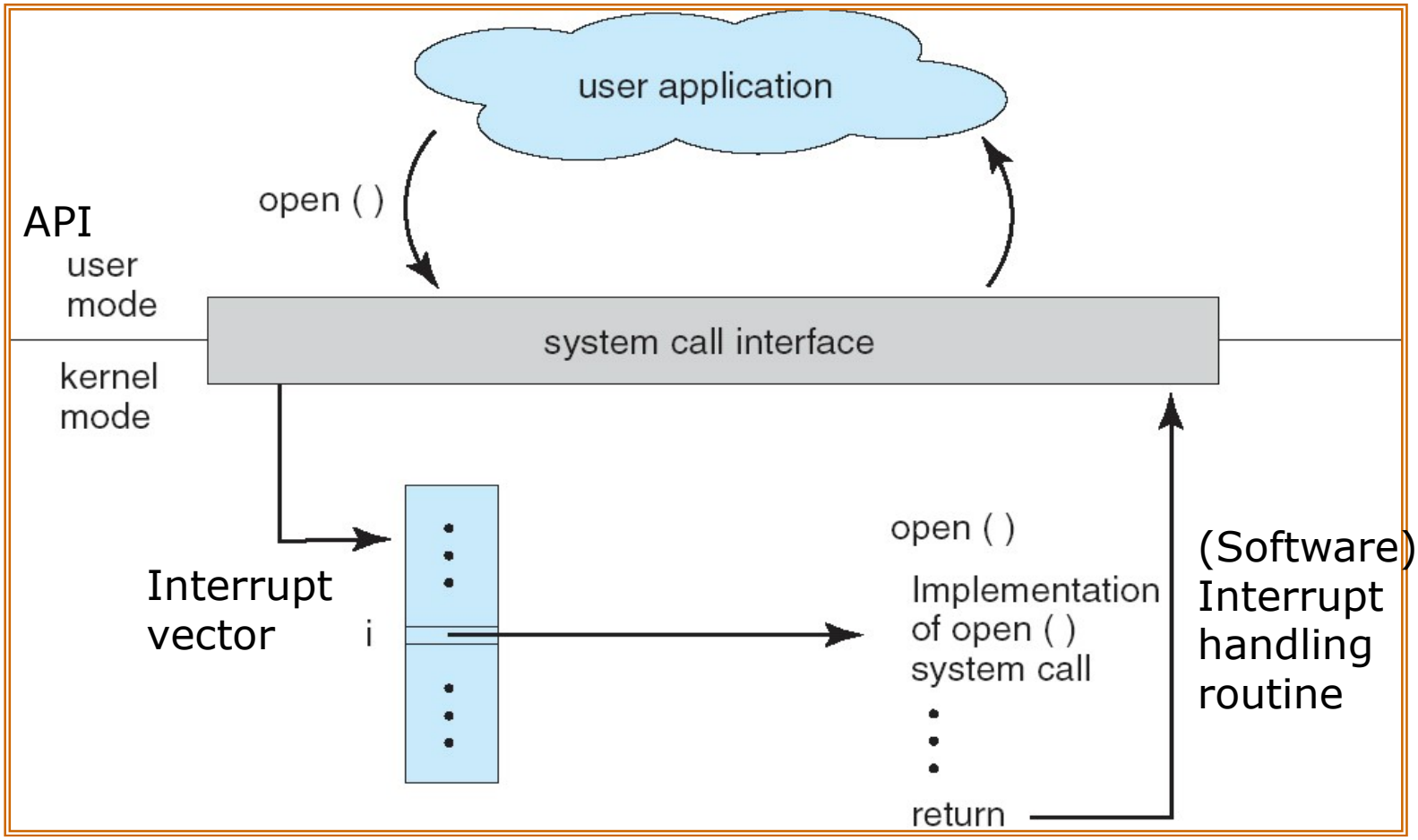


- Even simple programs make heavy use of the operating systems!
  - Thousands of systems calls per second
- Read more details (Ch. 2.3)

Application Programming Interface (API)

- The **set of functions** that are available to the programmer to conduct **systems calls** to the OS kernel
- Three most common APIs
  - **Windows API** for Windows, **POSIX API** for POSIX-based systems (e.g., UNIX, Linux, and Mac OS X), **Java API** for the Java virtual machine (JVM)
- Application developers design programs according to the **API**
  - Just needs to obey API rules (e.g., parameters) and understand what OS will execute
  - Most details of OS system call implementation hidden from programmer by API

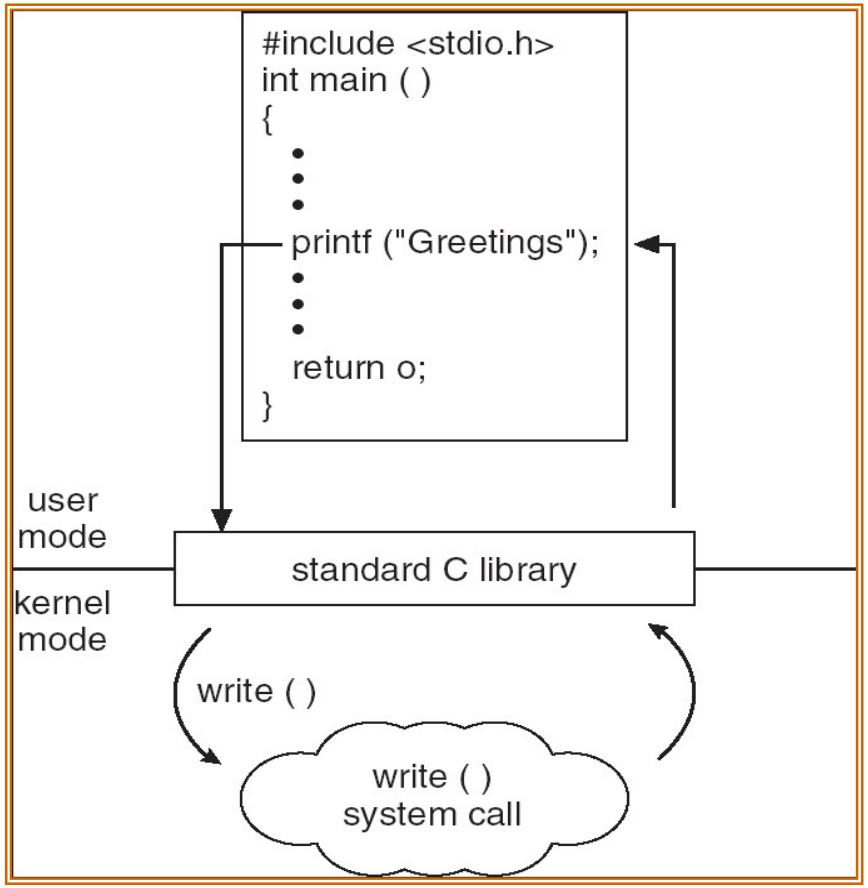
API – System Call – OS Relationship



Standard C Library Example

- C program invoking printf() API call, which calls write() system call

For example, there's a library of standardized C system calls referred to as the **Standard C Library.**
This library includes the *printf() ("print formatted data")* instruction which is illustrated follows…



Why use API?

- Why would a programmer prefer using API rather than invoking actual system calls?
  - printf() API vs. write() system call
  - Ease of use
    - Actual system calls are more detailed and difficult compared to API
    - Programmer doesn't need to know the details of how system call is implemented
  - Program portability
    - Program using one API can run on any system that support the same API
    - System calls are OS dependent!

Example of Standard API

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t    read(int fd, void *buf, size_t count)
```

return value — function name — parameters

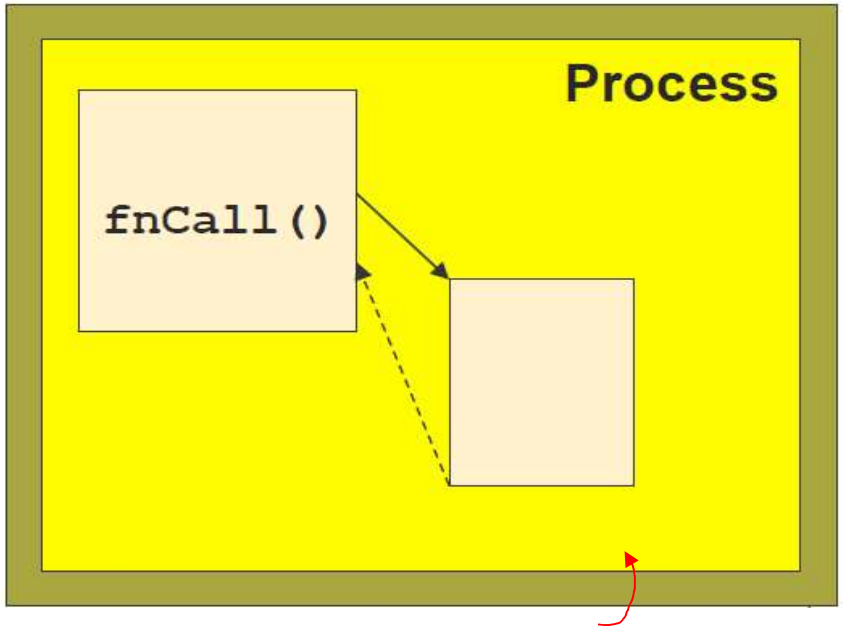
A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

Function Call vs. System Call

Function call

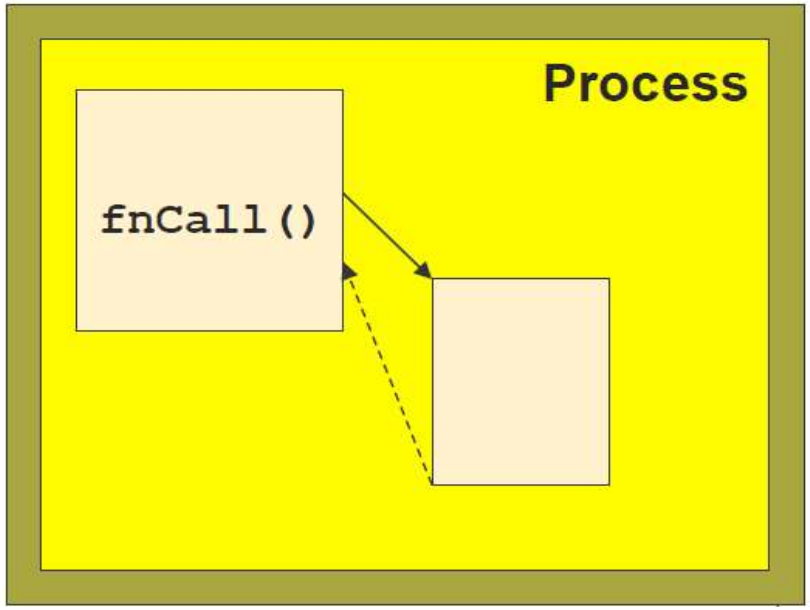


Caller and callee are in the same Process

- Same user
- Same “domain of trust”

Function Call vs. System Call

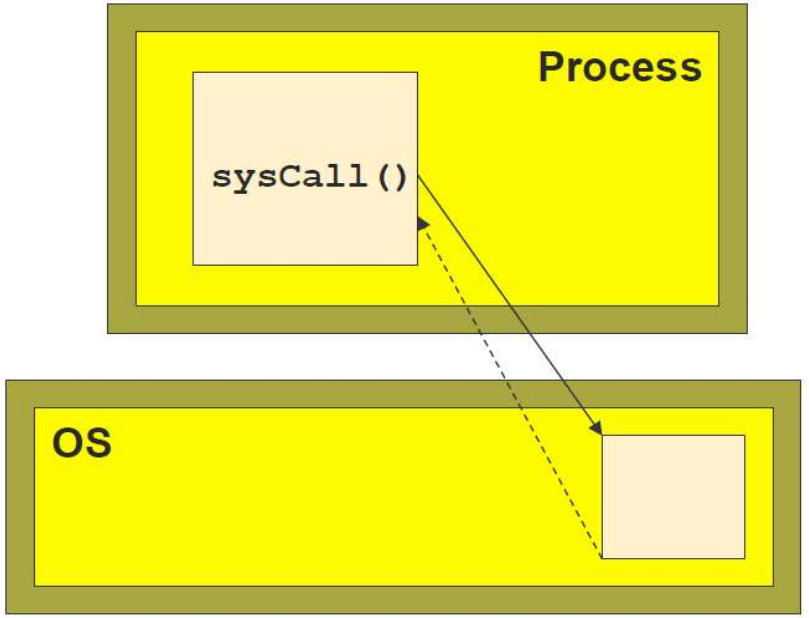
Function call



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

Various System Calls

- Process control
- File management
- Device management
- System Information maintenance
- Communications

- Note: Why do these operations need system call?

Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Chapter 2: System Structures

- Operating System Services and Interface
- Basic Operations
  - Interrupt
  - Multitasking
  - Protection
    - Dual-mode operation
    - Types of System Calls
  - Timer
- Operating System Structure
- Kernel Data Structures

Timer

- OS needs maintain control over CPU
  - User program can go into infinite loop or not return system resources

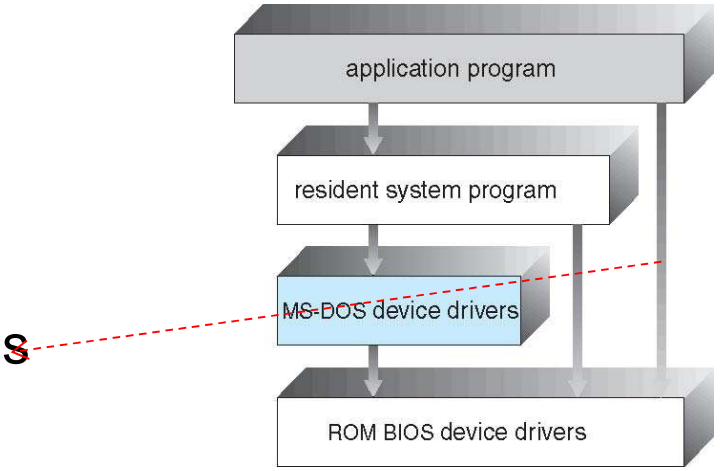
- **Timer**
  - Timer is set to **interrupt** the computer after some time period
  - Keep a counter that is decremented by the physical clock
    - Operating system sets the counter (privileged instruction)
    - When counter zero, generate an interrupt
  - Set up before scheduling process to regain control or terminate program that exceeds allotted time

Chapter 2: System Structures

- Operating System Services and Interface
- Basic Operations
  - Interrupt
  - Multitasking
  - Protection
    - Dual-mode operation
    - Types of System Calls
  - Timer
- Operating System Structure
- Kernel Data Structures

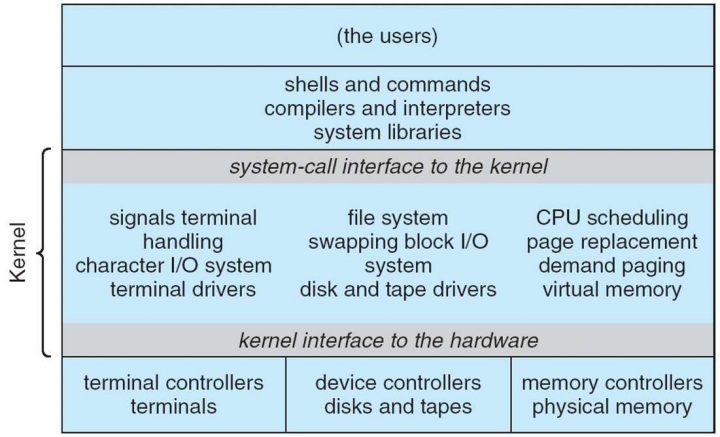
Simple Structure

- E.g., MS-DOS
  - written to provide the most functionality in the least space
  - application programs can directly access hardware I/O
    - Problems?



Monolithic structure

- E.g., UNIX
  - Systems programs
  - The kernel has all the functions in ***one level***
    - too large and complex – difficult to implement and maintain
  - Advantages
    - Performance is good – very little overhead in system call

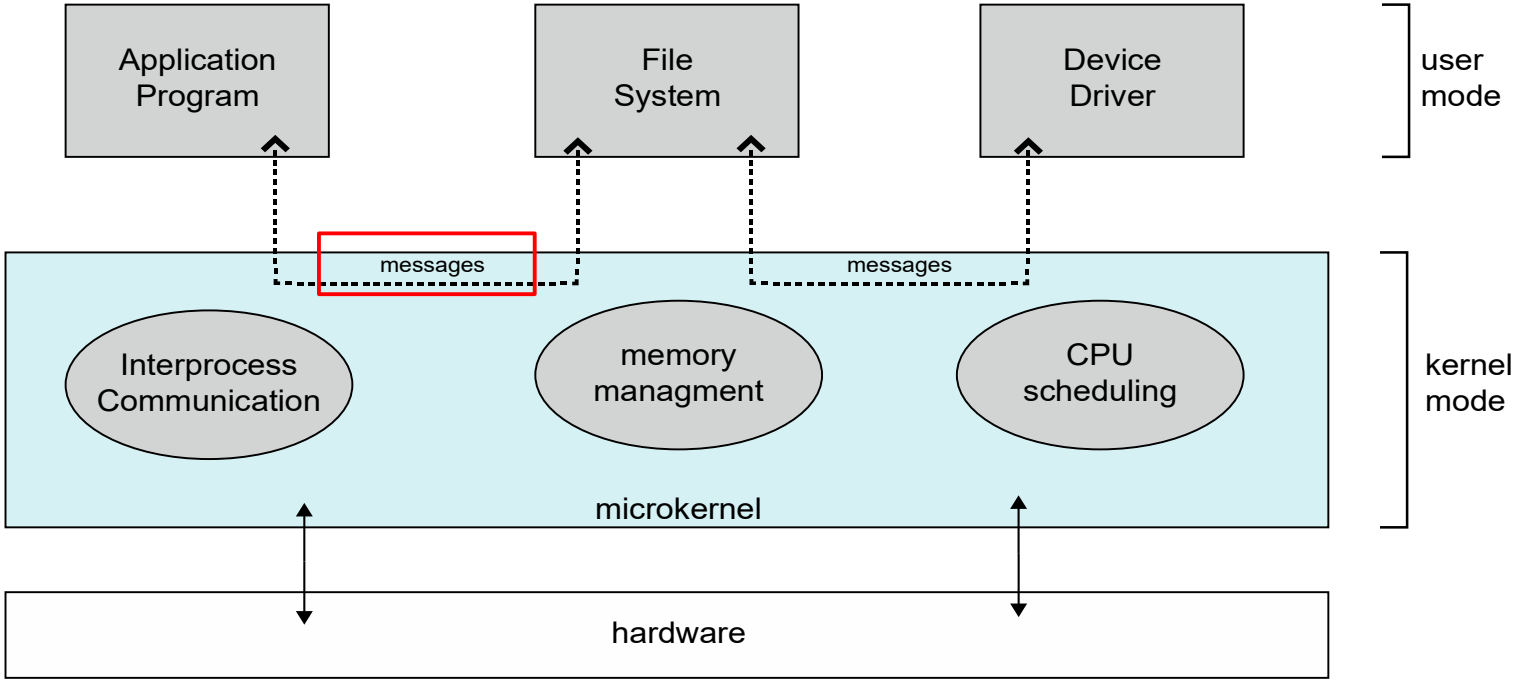


Microkernel System Structure

- Motivation
  - Single-level approach made the kernel too complex – UNIX
  - Move non-essential components of the kernel into user space (system and user-level programs)
  - **Mach:** example of **microkernel** (e.g., Mac OS X kernel (Darwin), iOS partly based on Mach, Windows NT 4.0)

- Results
  - smaller kernel!
    - microkernel provides only minimal process and memory management, communication facility
    - extending/modifying/porting kernel is easier

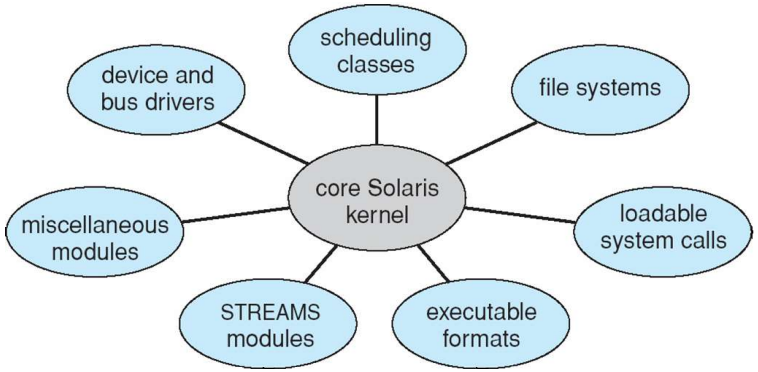
Microkernel System Structure



- Disadvantages?
  - client program need to communicate with various services through **message passing** – increased system-function overhead

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., SUN Solaris OS



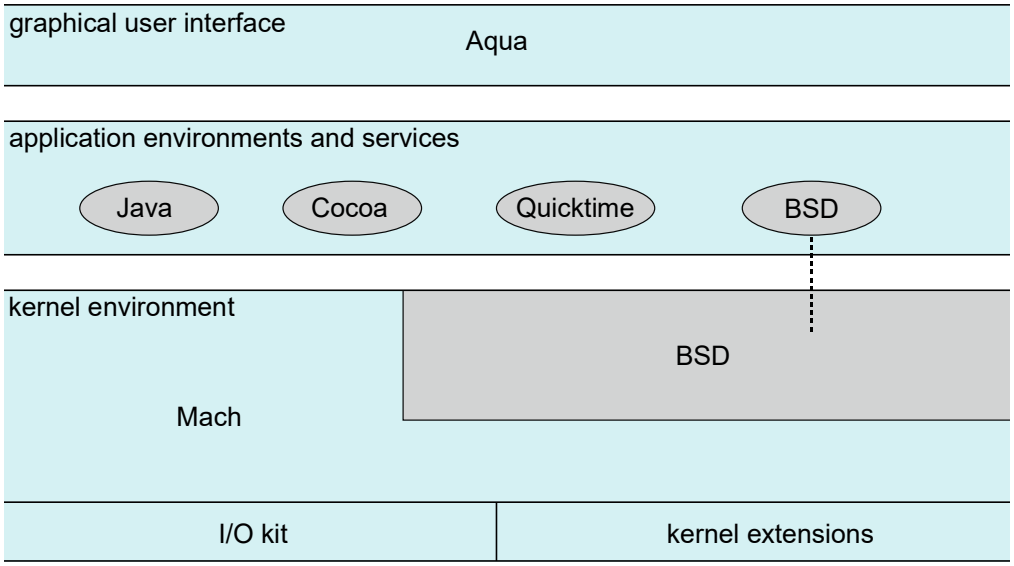
- Advantages
  - Adding new features does not require recompilation of entire kernel – as in monolithic kernel
  - Each module can talk directly to core kernel, no need for message passing – as in microkernel

Hybrid Systems

- Most modern operating systems actually not one pure model
- Hybrid combines multiple approaches to address performance, security, usability needs
  - **Linux** and **Solaris** kernels in single address space, so monolithic (performance), plus modular for dynamic loading of functionality
  - **Windows** mostly monolithic (performance), plus microkernel for different subsystem

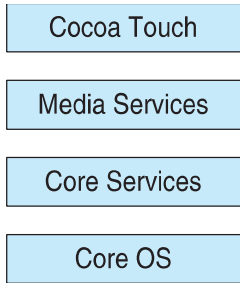
Mac OS X Structure

- Apple Mac OS X hybrid, layered, **Aqua** UI plus **Cocoa** programming environment (APIs for Objective-C)
- Below is kernel consisting of **Mach microkernel** and **BSD Unix kernel**, plus I/O kit and dynamically loadable modules (called **kernel extensions**)

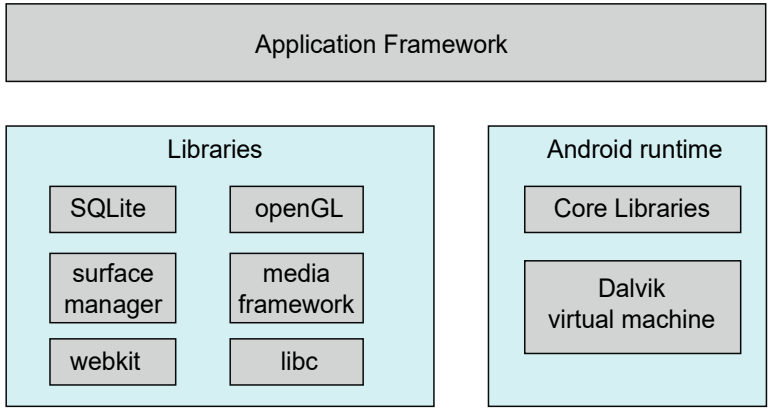


Mobile OS

- Apple mobile **iOS** for ***iPhone***, ***iPad***
  - Structured on Mac OS X, added functionality for mobile
  - Core operating system, based on Mac OS X kernel



- **Android**
  - Developed by Open Handset Alliance (mostly Google)
  - Based on **Linux kernel** but modified
    - Provides process, memory, device-driver management
    - Adds power management
  - Runtime environment includes core set of libraries and Dalvik virtual machine
    - Apps developed in **Java** plus Android API
    - Java class files compiled to Java bytecode then translated to executable than runs in Dalvik VM

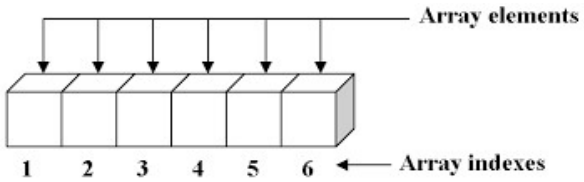


Chapter 2: System Structures

- Operating System Services and Interface
- Basic Operations
  - Interrupt
  - Multitasking
  - Protection
    - Dual-mode operation
    - Types of System Calls
  - Timer
- Operating System Structure
- Kernel Data Structures

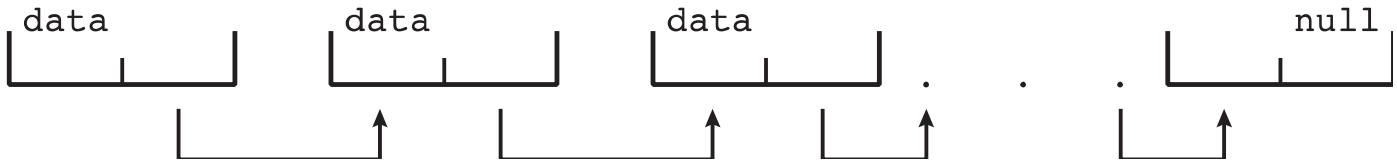
Kernel Data Structures: Lists

- Used by kernel algorithms
- *Array*

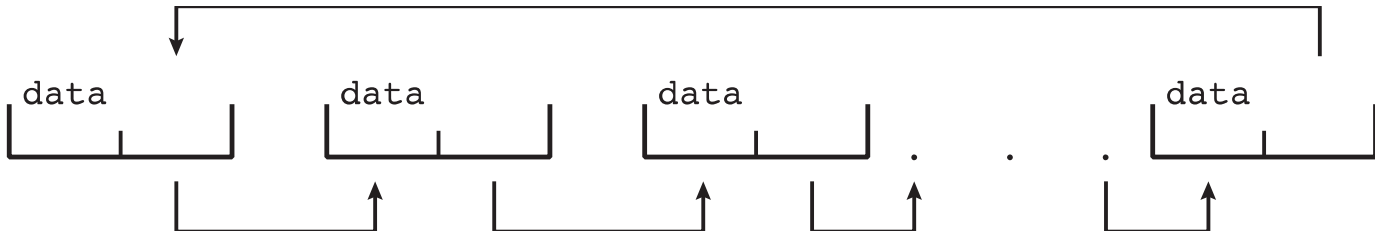


One-dimensional array with six elements

- *Singly linked list*

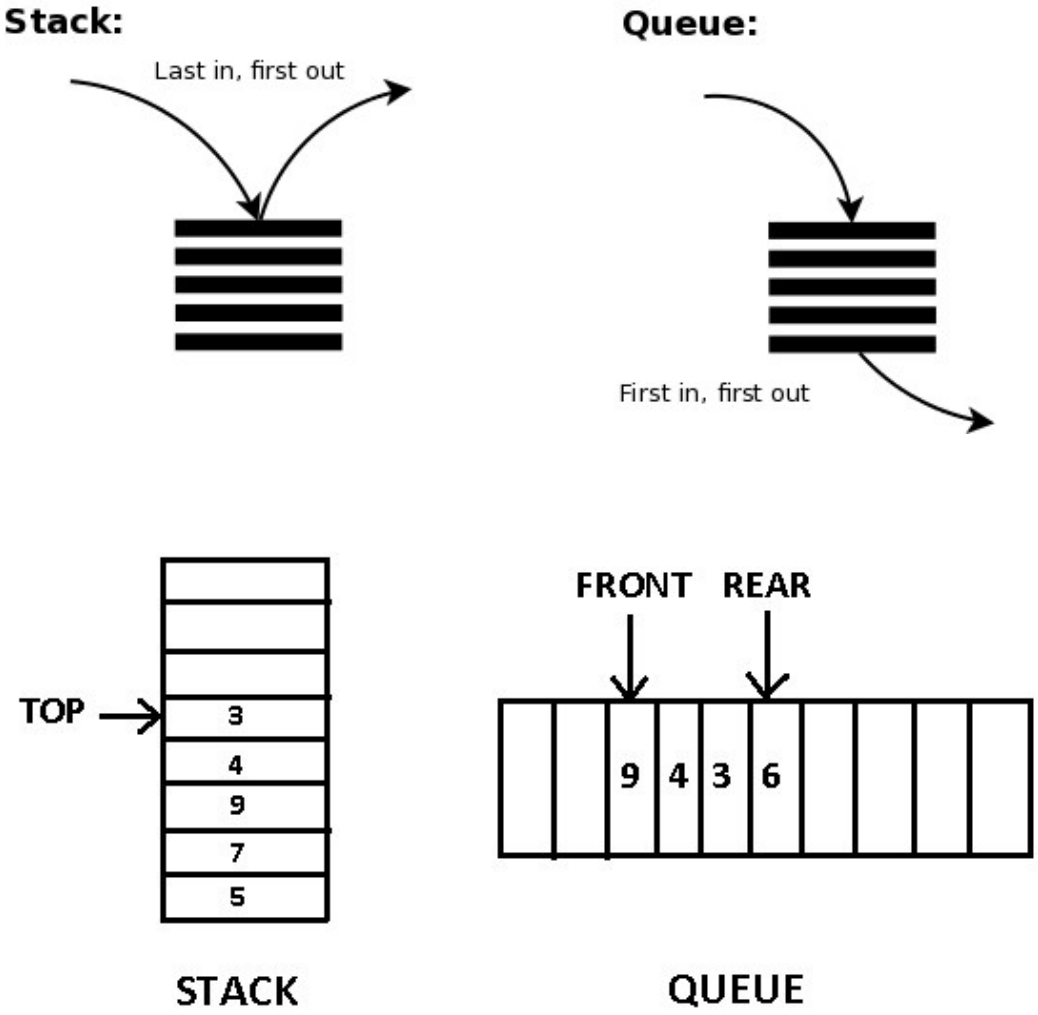


- *Circular linked list*



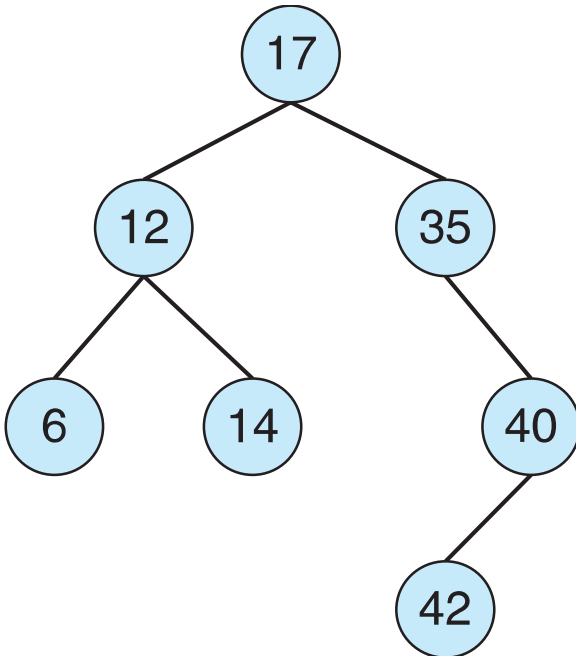
Kernel Data Structures: Stack/Queue

- 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
- Queue: FIFO
  - Printer jobs
  - CPU scheduling



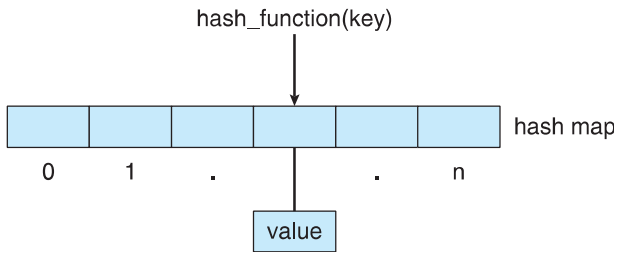
Kernel Data Structures: Trees

- **Binary search tree**
  left <= 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>`