



# SLIIT

*Discover Your Future*

## IT2060/IE2061

### Operating Systems and System Administration

#### Lecture 01

#### Introduction to Operating System

.....

# Introduction

- Learn the major components of the operating systems
- Practice with utilities of Unix system administration.
- Students will also apply the knowledge they learn in the lectures, tutorial and labs to complete the unit's programming assignment

# Method of Delivery

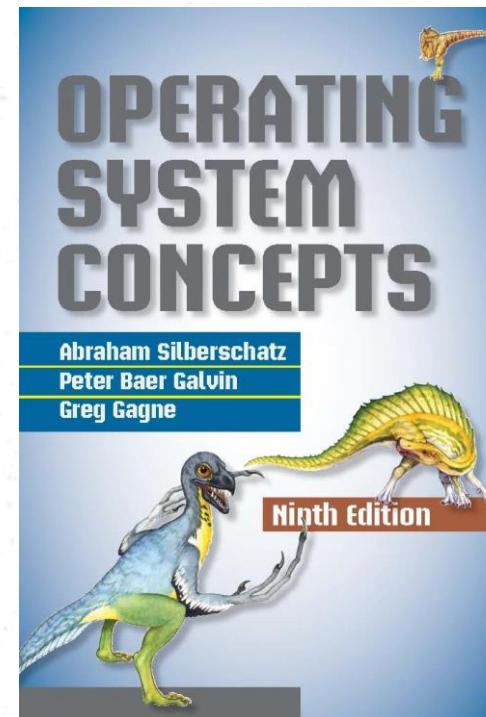
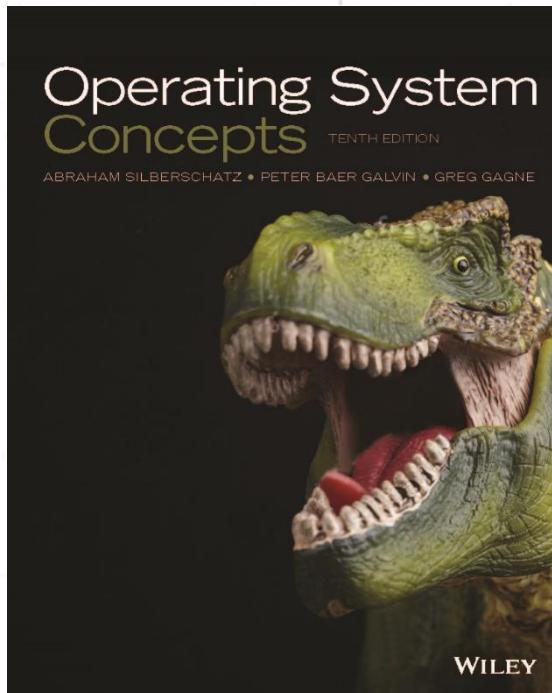
- 2 Hours Lecture
- 1 Hour Tutorial
- 2 Hours Practical
- Attendance will be marked.

# Assessment Criteria

- Mid Term Test (Online) 20% Lessons 1 to 5
- Assignment (Online) 20% Based on Practical Sessions (C Language)
- Final Examination (Written) 60% Lessons 6 to 12

# References

A. Silberschatz, P.B. Galvin, G. Gagne, Operating System Concepts, 10th Edition, John Wiley & Sons, 2018



# Linux Programming

Beginning Linux Programming

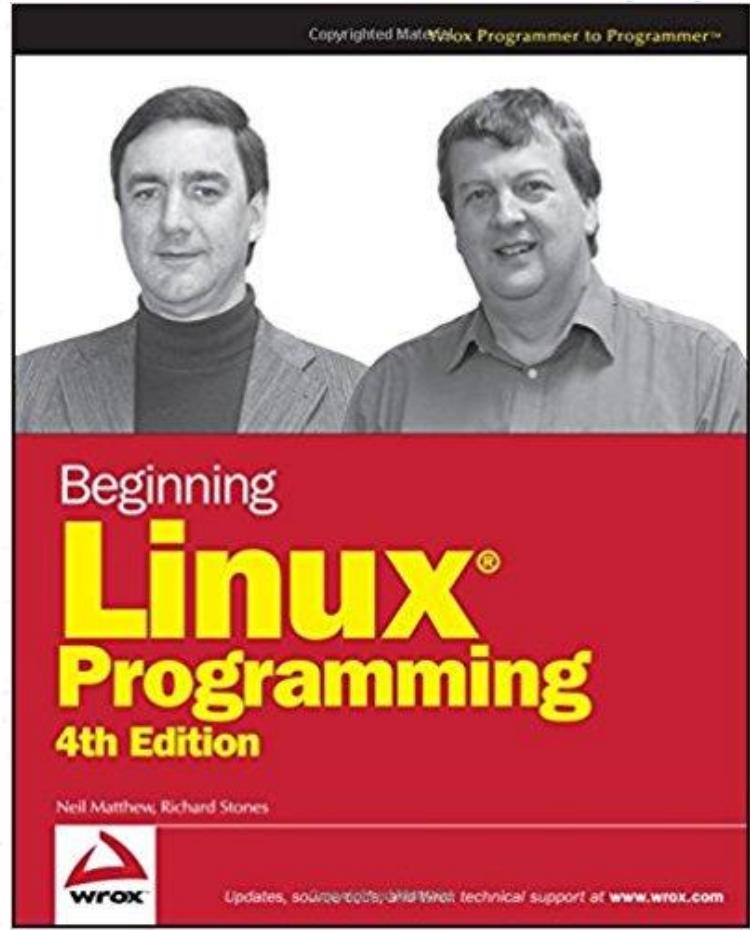
4th Edition

by Neil Matthew Richard Stones

PThreads Primer

A Guide to Multithreaded Programming

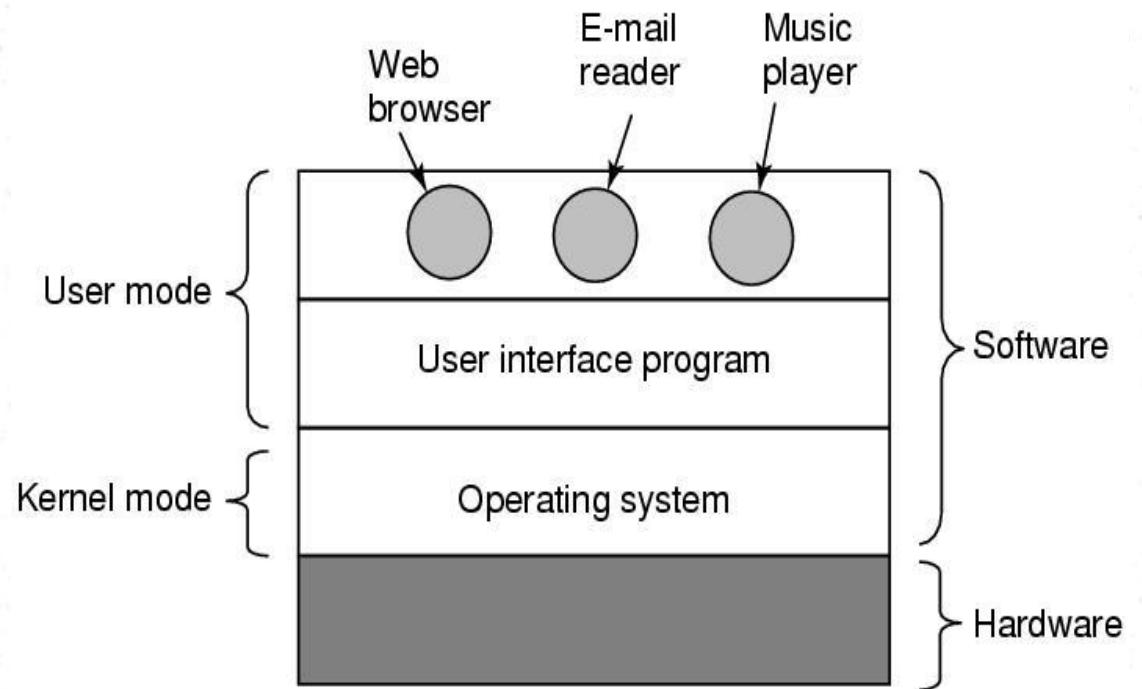
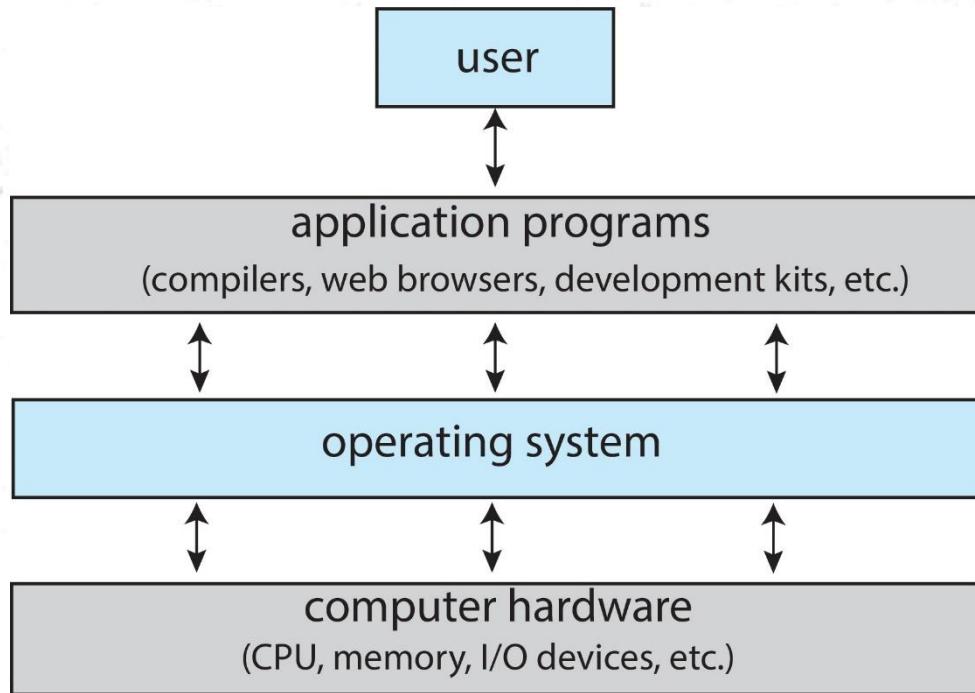
Bil Lewis Daniel J. Berg



# Computer System Structure

- Computer system can be divided into four components:
  - Hardware – provides basic computing resources
    - CPU, memory, I/O devices
  - Operating system
    - Controls and coordinates use of hardware among various applications and users
  - Application programs – define the ways in which the system resources are used to solve the computing problems of the users
    - Word processors, compilers, web browsers, database systems, video games
  - Users
    - People, machines, other computers

# Abstract View of Components of Computer

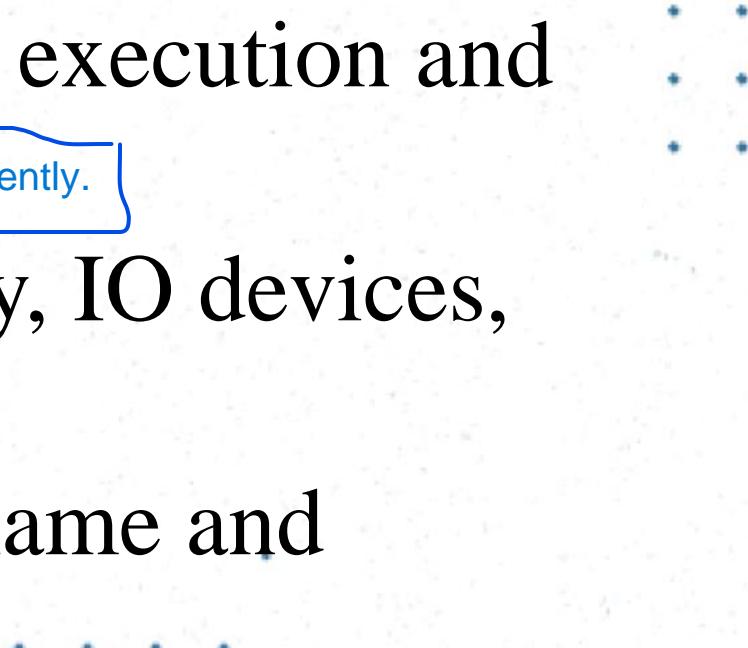


# What is an Operating System?

- An Operating System is a program that acts as an intermediary/interface between a user of a computer and the computer hardware.
- OS goals:
  - Control/execute user/application programs.
  - Make the computer system convenient to use.
  - Ease the solving of user problems.
  - Use the computer hardware in an efficient manner.

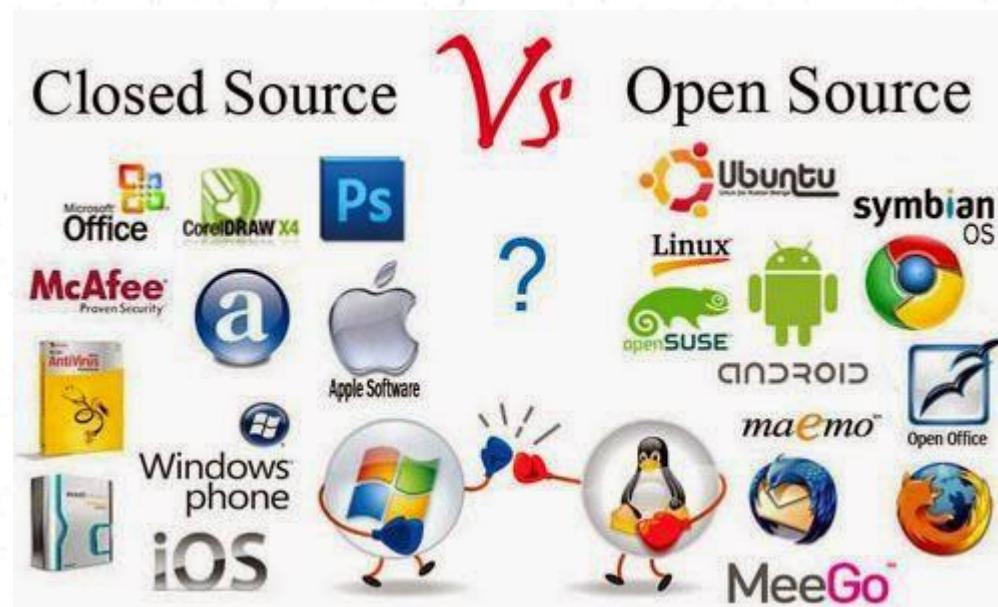
# Purposes of OS:

- Provide the environment for program execution and development
  - Run the application very efficiently and conveniently.
- Manage the resources (CPU, memory, IO devices, hard disk, files etc..)
- Provide the access controlling (username and password)



# Types of OS

Open source OS	Proprietary OS
Freely downloadable, code can be opened	Buy the OS and code is locked
Unix, Linux, Minix, Fedora, Ubuntu	MS-Windows , Mac OS ,DOS



# User Interface of the Operating System

Graphical User Interface (GUI)	Command Line Interface (CLI)
Very user-friendly and easy to learn the OS	Not much user-friendly and user has to remember the commands
Slow and need more resources	Fast and need less resources
Main components are icons, menu, windows and pointer	Main component is command interpreter


```
C:\>dir
Volume in drive C is MS-DOS 6
Volume Serial Number is 3057-0442
Directory of C:\

DOS          <DIR>    23/02/04  0:34
COMMAND   COM      54,645 31/05/94  6:22
WINA20   386       9,349 31/05/94  6:22
CONFIG   SYS      144 23/02/04  0:42
AUTOEXEC BAT      188 23/02/04  0:42
5 file(s)        64,326 bytes
2,134,048,768 bytes free
C:\>
```

Doesn't have windows, icons, cursors .etc.

# OS Components

- Process Management
- Main-memory management
- Secondary-storage management
- File Management
- I/O System Management
- Protection System
- Networking (Distributed Systems)
- Command- interpreter System



•  
•  
•  
•

# OS Services

## For helping users:

- Provide user interface (UI)
  - Command line interface – using text commands
  - Batch interface – commands and their directives are put in a file
  - Graphical user interface (GUI) – window system with pointing devices
- Provide environment for program execution.
  - OS must load program and run it.
  - Program must end normally or abnormally.
- Provide some means to do I/O
  - user programs cannot execute I/O operations directly.
- Provide mechanism to do file-system manipulation.
  - Capability to read, write, create, and delete files, directory trees etc.
- Provide mechanism for process communication.
  - Exchange information between processes executing on the same computer or on different systems through a network.
  - Implemented via shared memory or message passing.
- Detect errors and take appropriate actions to ensure correct and consistent computing.
  - Detect errors in CPU and memory hardware, in I/O devices, or in user programs.

USB, HardDisk, DVD, CD

Can detect some errors also. It will be managed by the OS

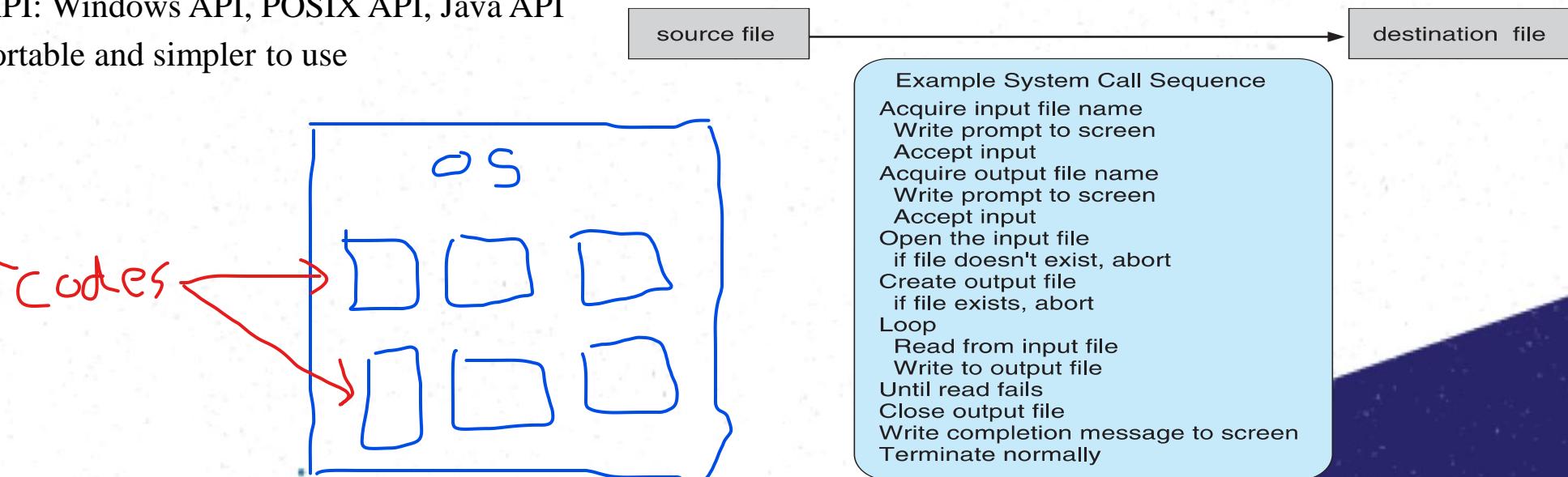
# OS Services

## For efficient operation:

- Resource allocation
  - Allocating resources to multiple users or multiple jobs running at the same time (CPU scheduling, etc.).
- Accounting
  - Keep track of and record which users use how much and what kinds of computer resources for account billing or for accumulating usage statistics .
- Protection and security
  - Ensuring that all access to system resources is controlled (access permissions, etc.).

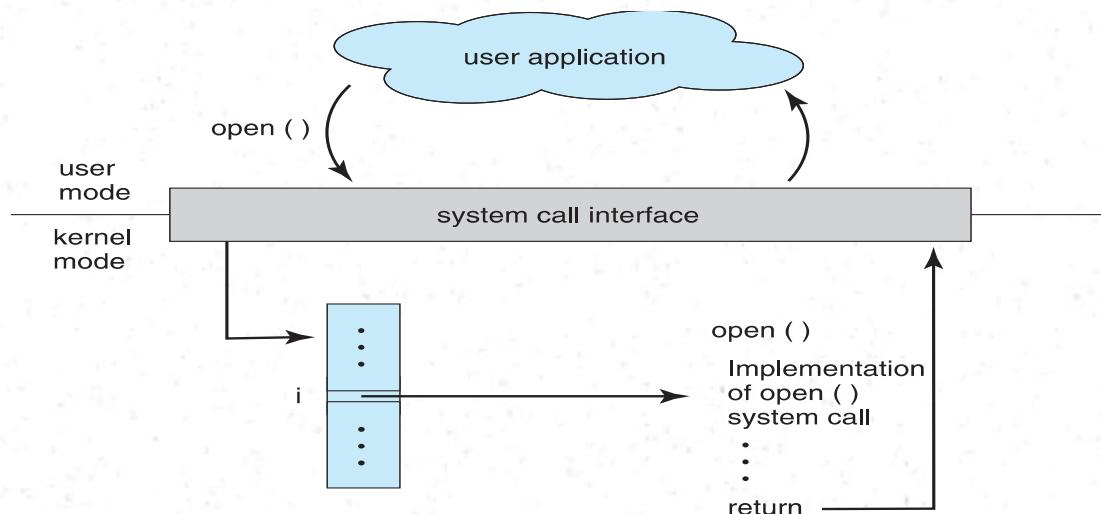
# System Calls

- A system call is a user interface to the OS services
  - Available in assembly-language instructions or in high level languages for systems programming (e.g., C)
    - E.g., fork () is a system call to ask OS to create a new process
  - Application Program Interface (API): a set of functions available to an application programmer.
    - An API typically invokes the actual system calls for the application programmers.
    - Examples of API: Windows API, POSIX API, Java API
    - API is more portable and simpler to use



# System Calls (contd. )

- System call result/termination.
  - Normal termination (exit or return).
  - Terminate current program and return to the command interpreter.
  - Abnormal termination (trap) — program error.



## 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
-----------------	------------------	------------

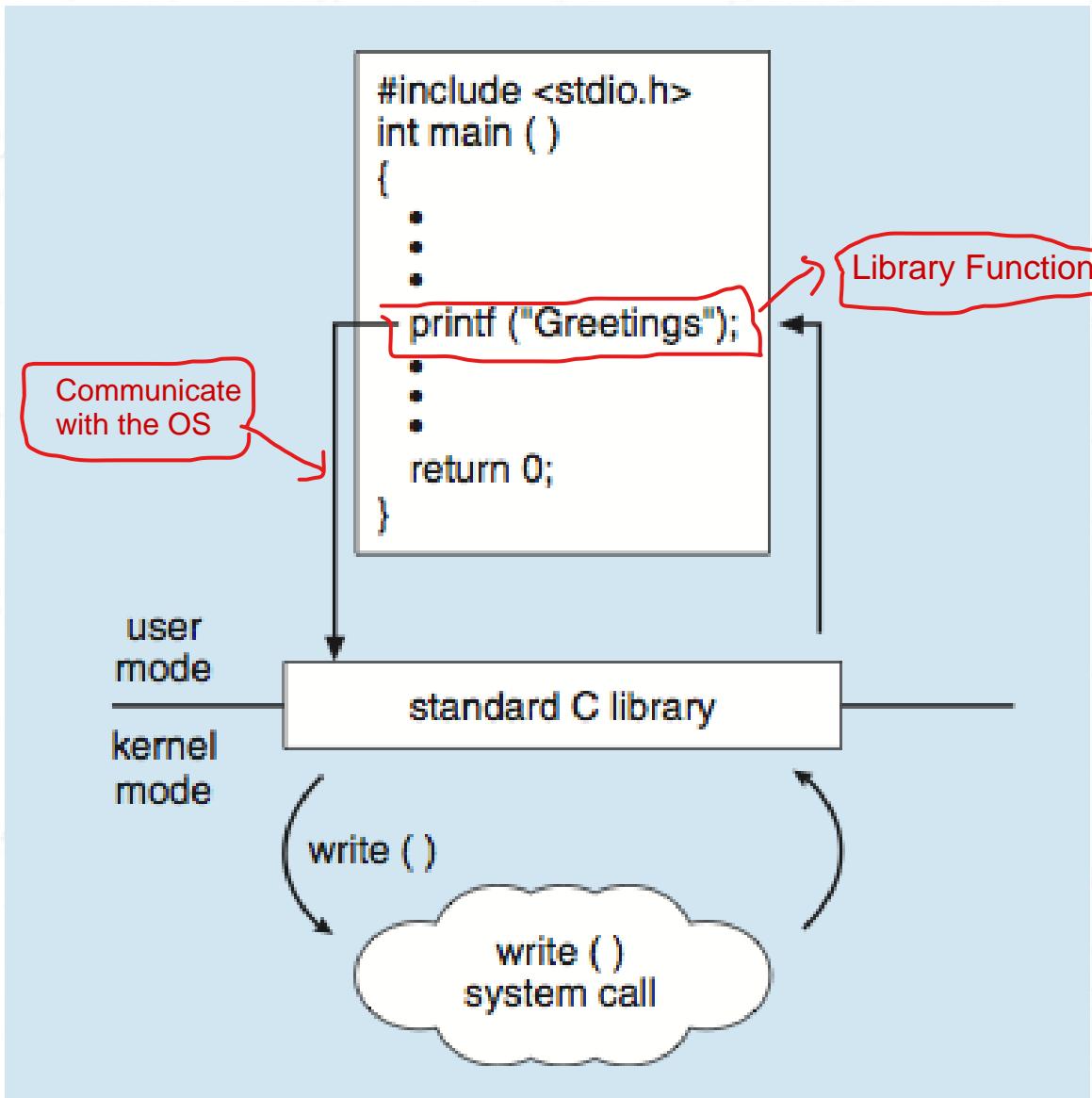
A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `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`.

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() <b>write()</b> → 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() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

*system  
call*



# System Calls (cont. )

## Types of system calls

### 1) Process control

- End, abort (running program); Load, execute; Create, terminate; Signal event, etc.
- Examples: fork(), exit(), wait(), etc.

### 2) File management

- Create, delete; Open, close; Read, write, reposition; Get and set file attributes, etc.
- Examples: open(), read(), write(), close(), etc.

### 3) Device management (memory, tape drives etc.)

- Request device; Release device; Read, write, reposition, etc.
- Examples: ioctl(), read(), write(), etc.

### 4) Information maintenance

- Get or set time or date; Get or set process attributes, etc.
- Examples: getpid(), alarm(), sleep(), etc.

### 5) Communications

- Create, delete communication connection; Send and receive messages, etc.
- Examples: pipe(), shmget(), mmap(), etc.

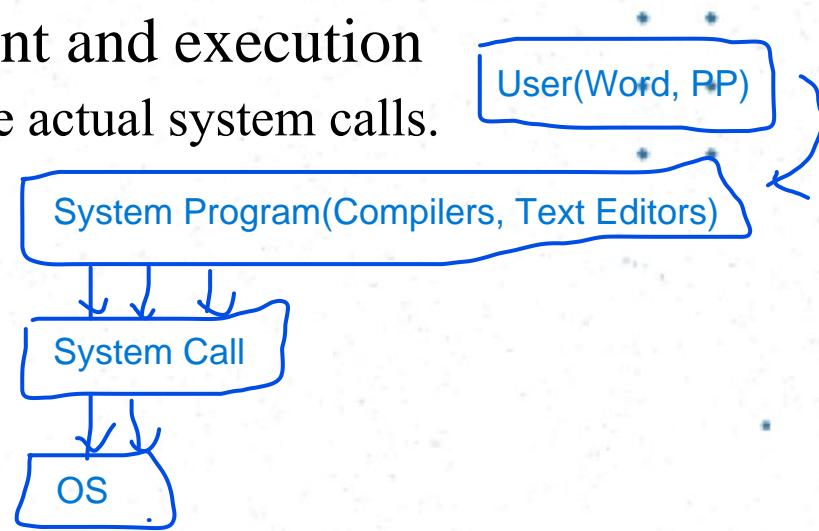
### 6) Protection

- Resource access control
- Examples: chmod(), umask(), chown(), etc.

# System programs or utilities

Users are not going to communicate with the System Call

- Provide a convenient environment for program development and execution
  - Most users' view of the OS is defined by system programs, not the actual system calls.
  - Some of them are simple user interface to system calls.
- System programs can be divided into:
  - File manipulation: Create, delete, copy, rename, print, dump, list.
  - Status information: Date, time, memory, disk space, number of users.
  - File modification: Text editors to create and modify content of files.
  - Programming-language support: Compilers, interpreters, assemblers.
  - Program loading and execution: Absolute, Relocatable, Overlay loaders.
  - Communication: Programs that provide the mechanism for creating virtual connections among processes, users, and different computer systems.



# Operating System Design Goals

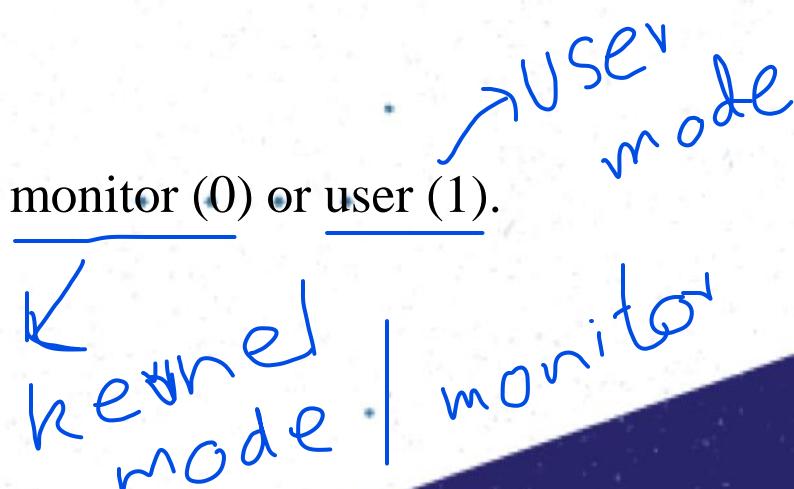
- Design of the OS is affected by the hardware choice, and system type (batch, timeshared, etc.)
  - User goals – OS should be convenient to use, easy to learn, reliable, safe, and fast.
  - System goals – OS should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
- Separation of *policy* and *mechanism* is a very important principle
  - It allows maximum flexibility if policy decisions are to be changed later. → Policies are changing time to time, place to place, machine to machine
  - Mechanisms determine *how* to do something
  - Policies decide *what* will be done. → If the policies and mechanism are not separated whenever the policy changed we have to change the mechanism
- Policies are likely to change from place to place and time to time, and therefore a general mechanism would be more desirable.
- Example: A mechanism for ensuring CPU protection is the timer construct; and the decision on how long the timer is set for a particular user is a policy decision.

# OS Operation

- In multiprogramming, OS must ensure that an incorrect (or malicious) program cannot cause other programs to execute incorrectly.
- Many programming errors are detected by the hardware, and the errors are normally handled by the OS.

## Dual-mode operation

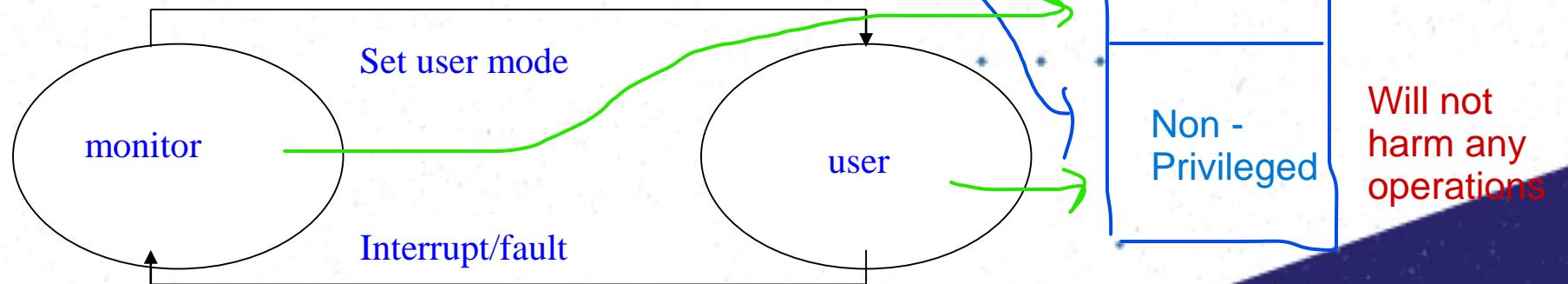
- Hardware provides at least two modes of operations:
  - User mode – in which user programs run.
  - Monitor mode, also called Supervisor, System, or privileged mode.
- Mode bit is provided by the hardware to indicate the current mode: monitor (0) or user (1).



# OS Operation (cont.)

## Dual-mode operation (cont.)

- Some machine instructions that may cause harm are designated (by hardware) as privileged instructions.
  - E.g., the MSB of the machine code of the instruction is a bit ‘1’
- A privileged instruction can be executed only in monitor mode.
- At system boot-time the hardware starts in monitor mode → OS is loaded.
- OS starts user processes in user mode; a user process could never gain control of the computer in monitor mode.
- When an interrupt or fault occurs hardware switches to monitor mode.



# OS Operation (cont.)

## I/O

- All I/O instructions are privileged instructions.
- How does the user program perform I/O ? Use system call.
  - Usually takes the form of a trap to a specific location in the interrupt vector.
  - Control passes through the interrupt vector to a service routine in the OS, and the mode bit is set to a monitor mode.
  - The monitor verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call.

# OS Operation (cont.)

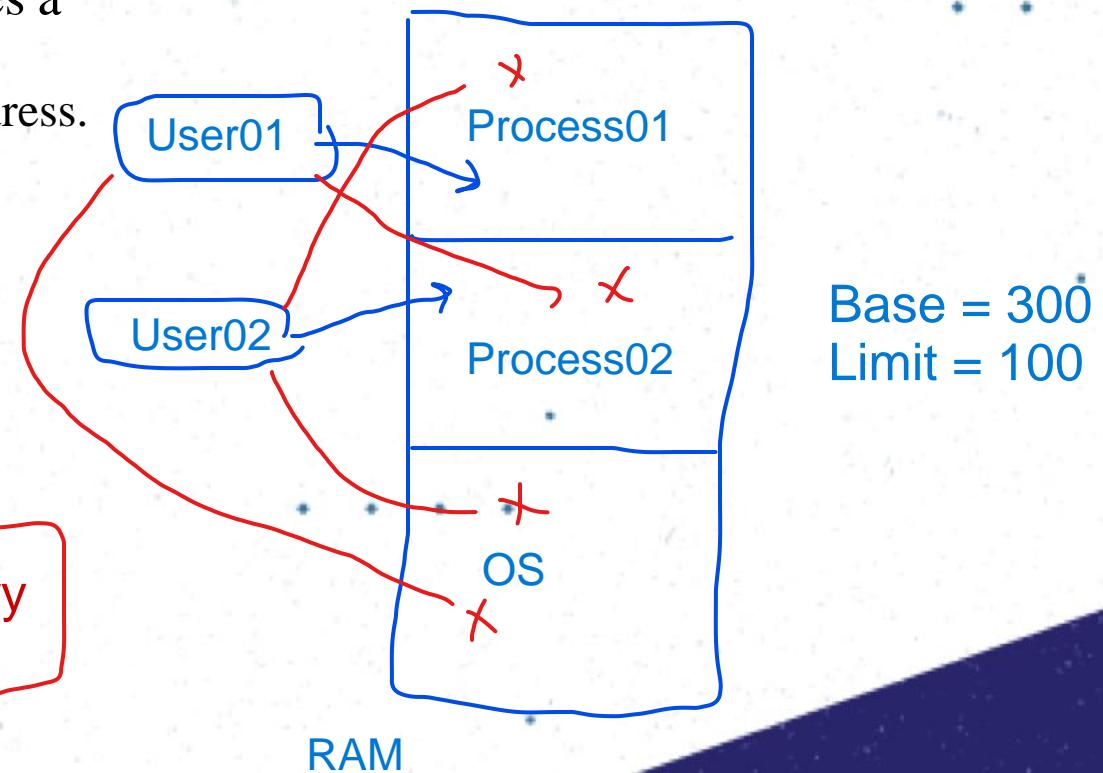
## Memory

System must provide memory protection at least for the interrupt vector and the Interrupt Service Routine.

- Use two registers that determine the range of legal addresses a program may access:
  - Base register – holds the smallest legal physical memory address.
  - Limit register – contains the size of the range.
- Memory outside the defined range is protected.
- The load instructions for the base and limit registers are privileged instructions.
- OS has unrestricted access to monitor and user memory.

{ CPU - Generates the Address  
(Base  $\leq$  Address  $<$  Base + Limit)  
(300  $\leq$  Address  $<$  300 + 100)}

How the memory  
is protected



# Memory Protection

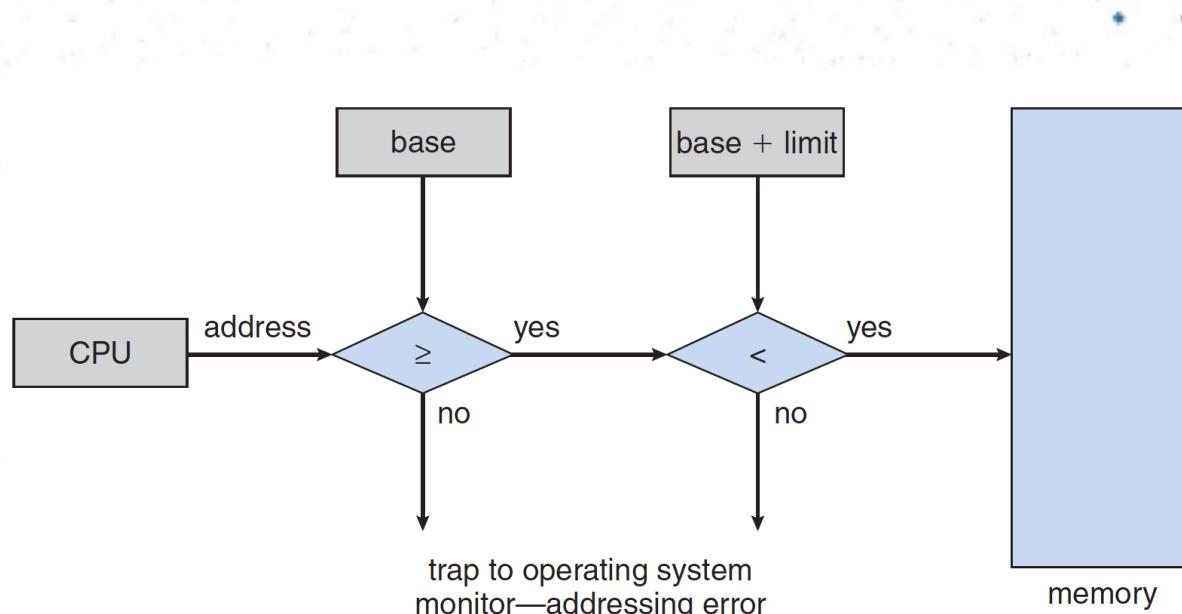
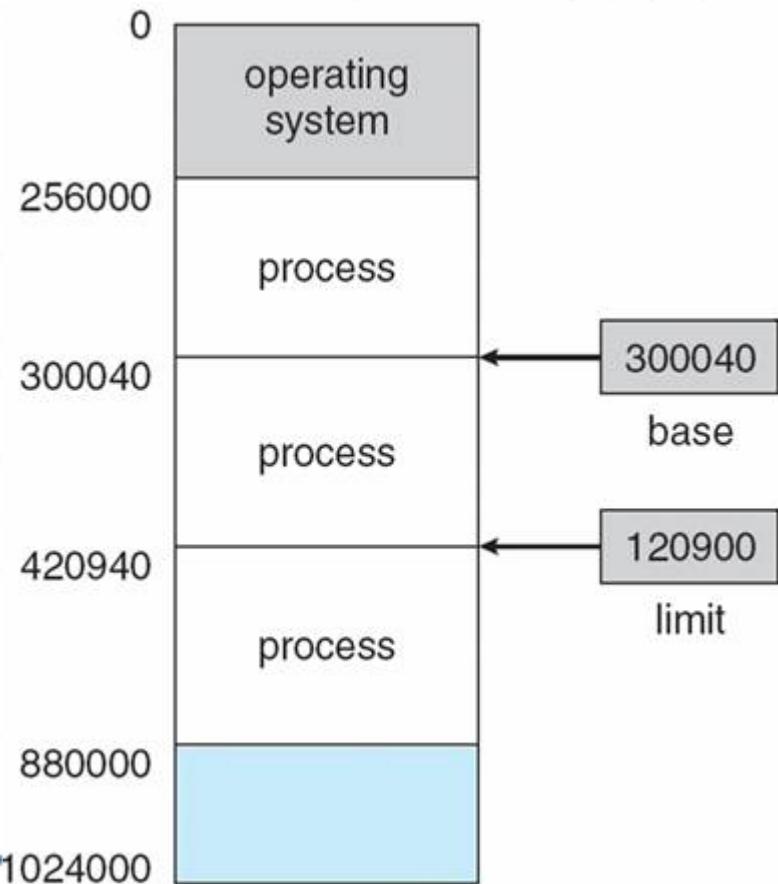
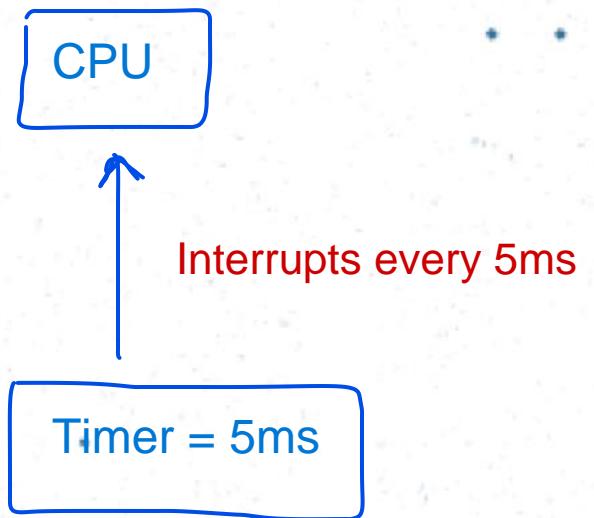


Figure 7.2 Hardware address protection with base and limit registers.

# OS Operation (cont.)

## CPU

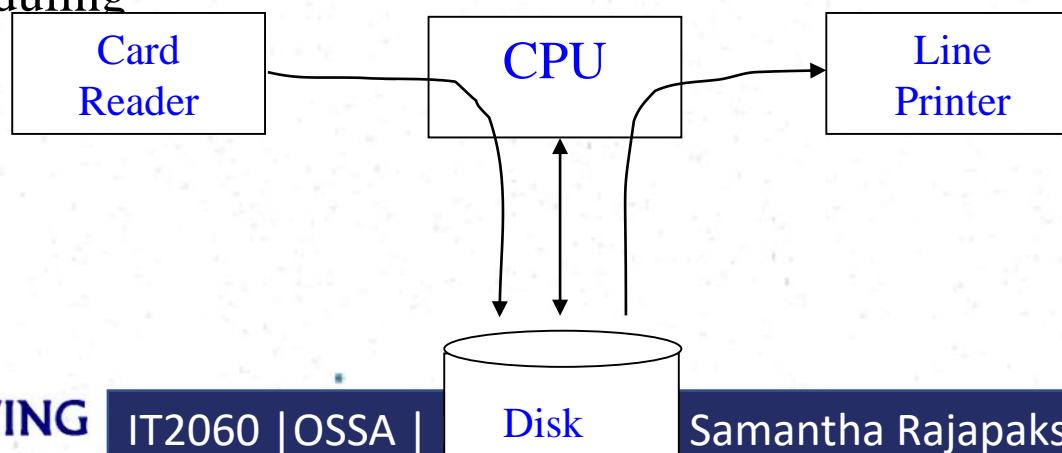
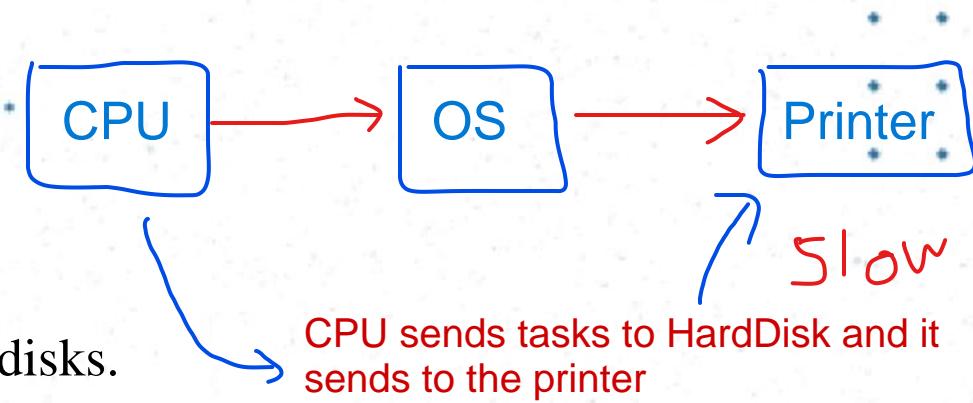
- System must prevent one user program using CPU all the time
  - Because of getting stuck in an infinite loop, or non-fair users
- Use *timer* interrupt after specified period to ensure OS maintains control
  - Timer is decremented every clock tick.
  - When timer reaches value 0, an interrupt occurs.
- Timer is commonly used to implement time sharing.
- Timer is also used to compute the current time.
- Loading timer value is a privileged instruction. Why?



# OS development (cont.)

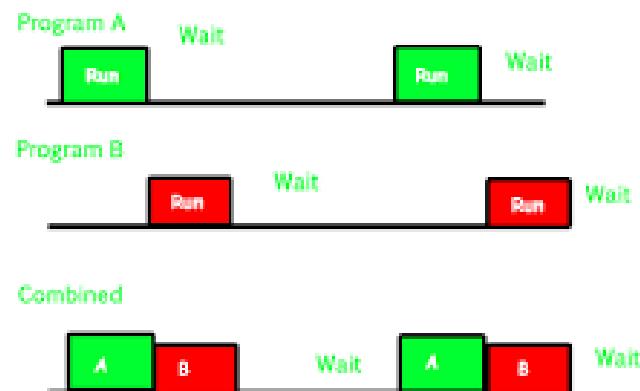
## Simultaneous Peripheral Operation OnLine (SPOOL):

- Goal: To keep I/O and CPU busy all the time.
- Use a faster disk as huge buffer → disk was a new technology
- Inputs from cards are read into disks.
- CPU reads input from disks and outputs are written by CPU to disks.
- The I/O of a job is overlapped with its own computation or with another job.
- Need a Job Pool – data structure that allows OS to select job that runs next in order to increase CPU utilization → job scheduling

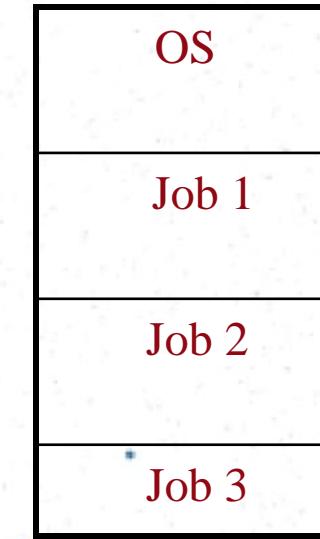


# Multiprogramming

- Goal: to increase the CPU utilization
- Several jobs are kept in main memory at the same time, and the CPU is multiplexed among them
- It needs these OS features:
  - ❖ Job scheduling → OS chooses jobs in the job pool and put them into memory.
  - ❖ Memory management → OS allocates the memory for each job.
  - ❖ CPU scheduling → OS chooses one among the jobs in memory (called processes) that is ready to run.
  - ❖ I/O allocation.



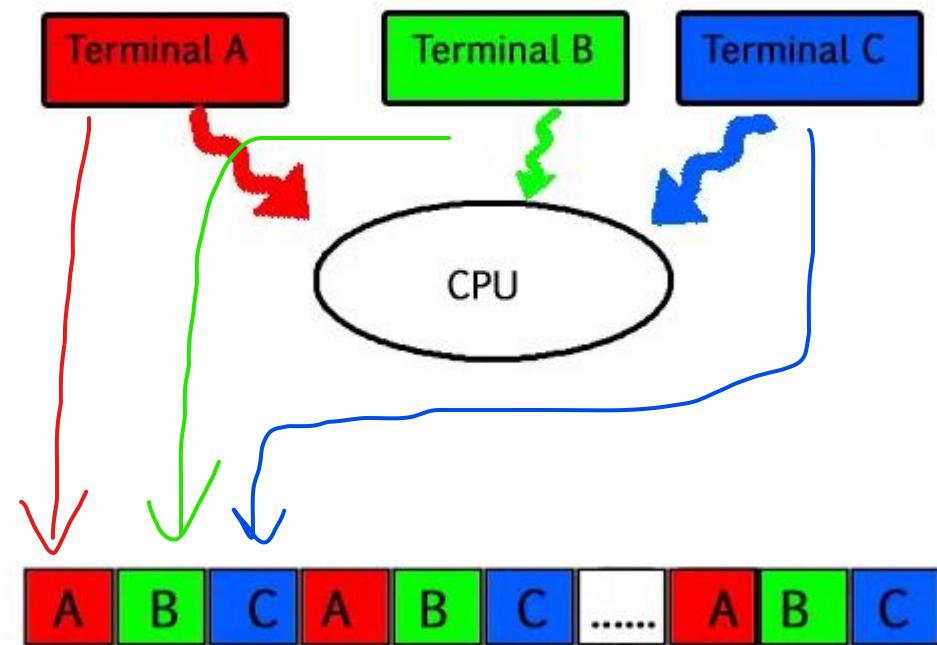
**Memory partition**



Keeping more  
than one  
memory

# Time-Sharing/Multitasking Systems

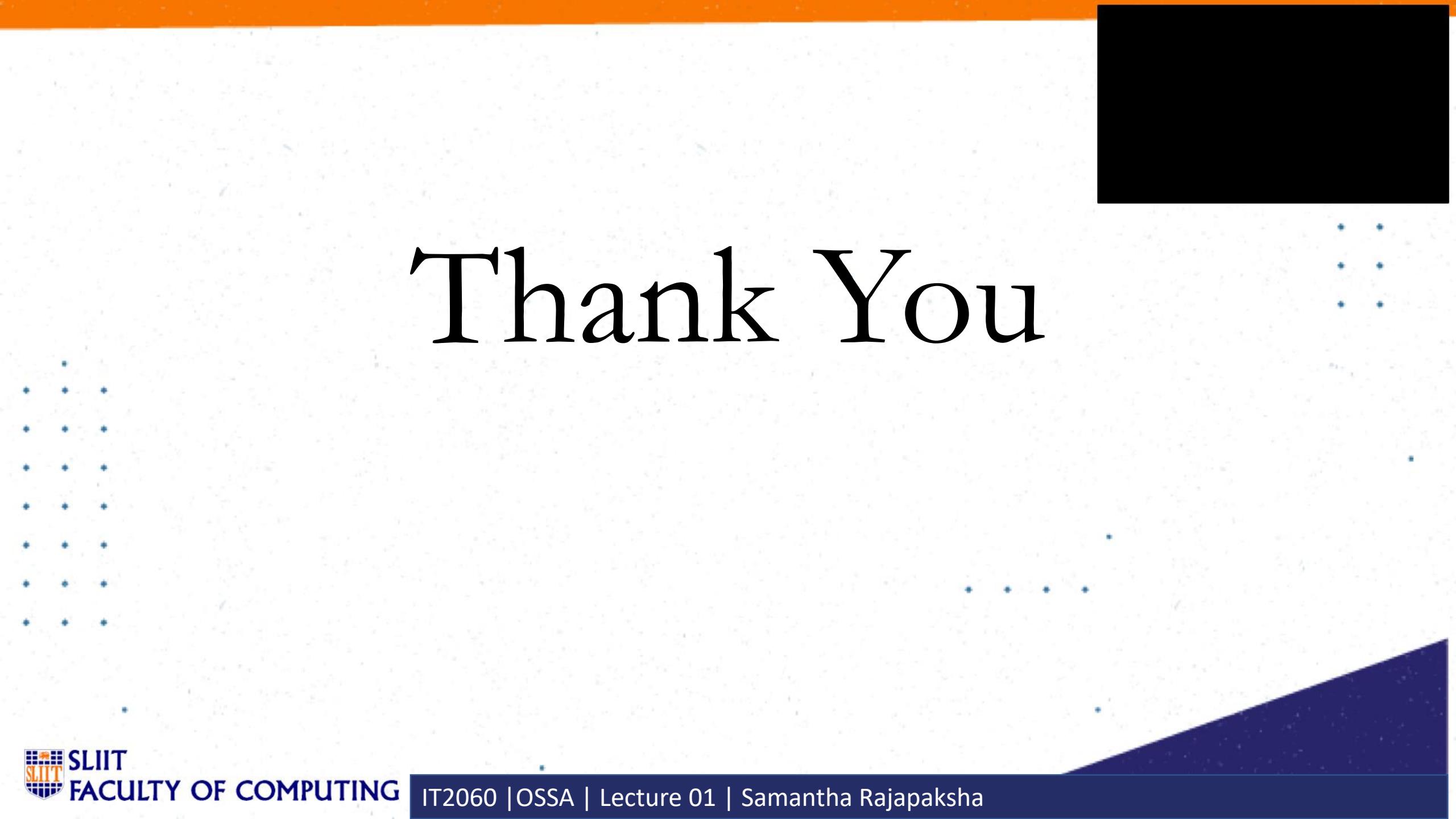
- **Goal:** to provide interactive use of computer system at reasonable cost (one computer – several users/jobs).
- It is a variant of multiprogramming → but user input is from on-line terminal.
- CPU is multiplexed among jobs in memory frequently ( $\leq 1$  sec?) so that users can interact with their running program.
- Today, most systems provide both batch processing and time sharing.



Running each program for a period of time  
(Not running the program continuously)

# Real-time Systems

- Well-defined fixed-time constraint – the system is functional if it returns the correct result within the time constraint.
- Hard real-time system. **If there is a time delay damage is high - Air Line**
  - ❖ Guarantees that critical tasks complete on time.
  - ❖ Often used as a control device in a dedicated application
    - ❖ controlling scientific experiments, medical imaging systems, industrial control systems, and some display systems
    - ❖ Secondary storage is limited or absent
      - ❖ data is stored in short-term memory, or in ROM.
- Soft real-time system **If there is a time delay damage is less - ATM Machine**
  - ❖ A critical-time task gets priority over others until it completes.
  - ❖ Limited utility in industrial control robotics.
  - ❖ Useful in applications (multimedia) requiring advanced OS features.



# Thank You



# SLIIT

*Discover Your Future*

# IT2060/IE2061

## Operating Systems and System Administration

### Lecture 02

### Introduction to Operating System

### U. U. Samantha Rajapaksha

M.Sc.in IT, B.Sc.(Engineering) University of Moratuwa

Senior Lecturer SLIIT

[Samantha.r@slit.lk](mailto:Samantha.r@slit.lk)

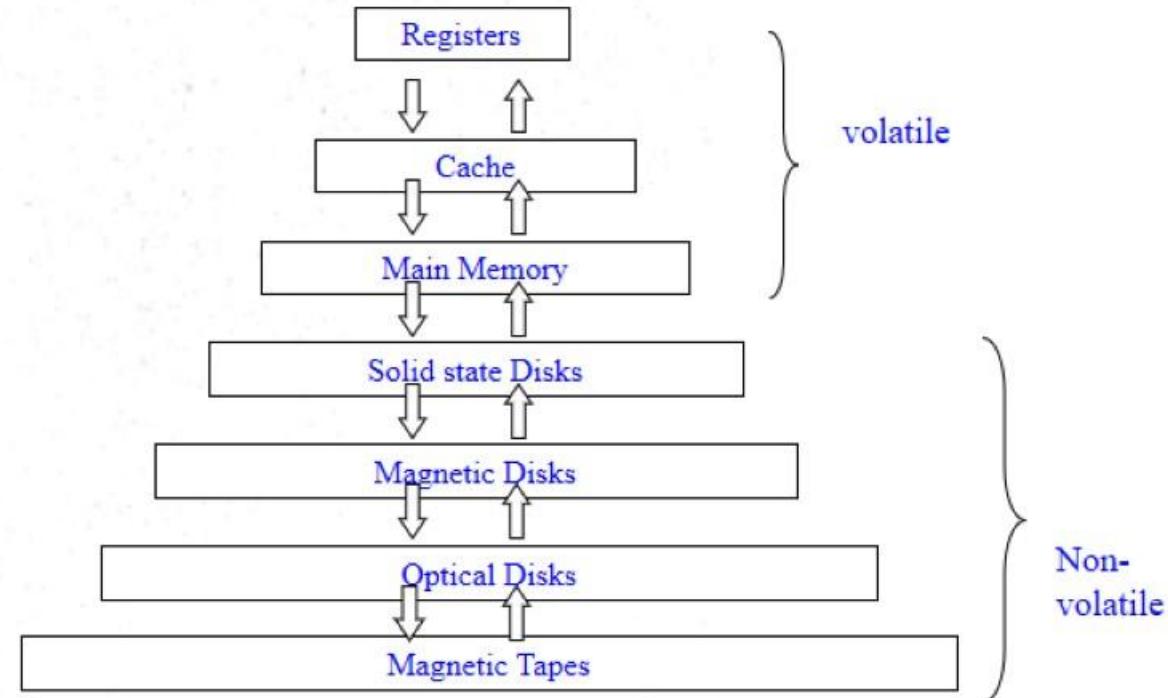
# Content

- Storage Structure
- Booting Up process
- Multiprocessor System
- Interrupts handling
- Operating System Structures

# Storage Structure

Storage systems are organized in hierarchy in terms of:

- Speed
- Cost
- Volatility
- Size/capacity



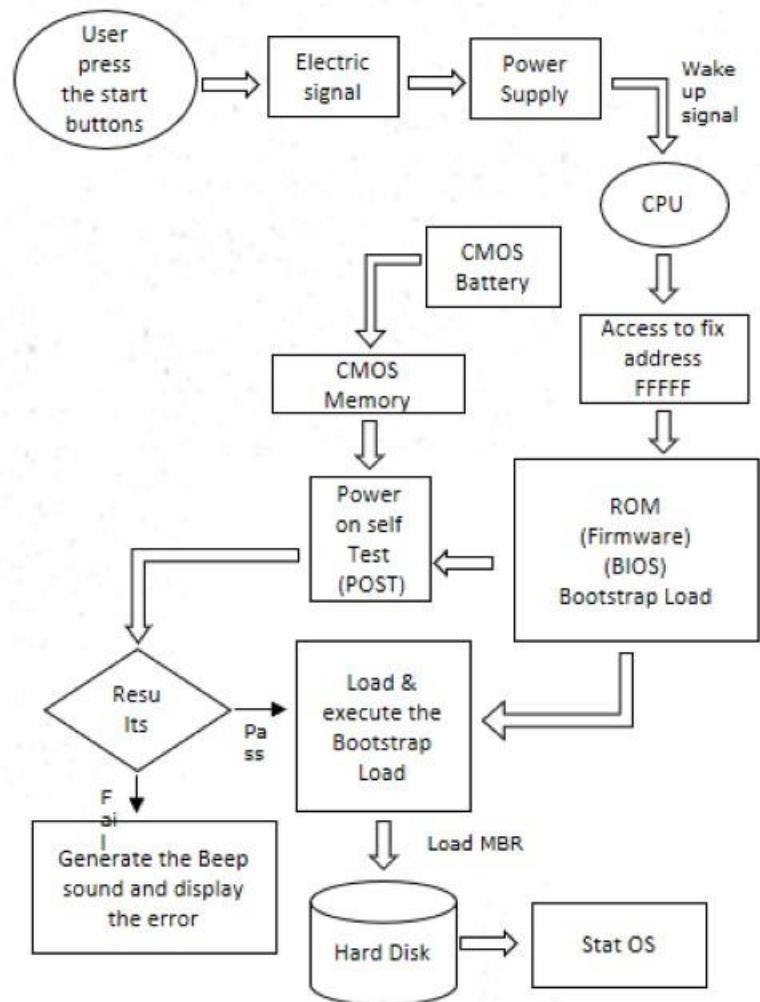
# Performance of various levels of storage

Figure 1.11 (Textbook)

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

# Computer Startup

- **Booting Up the Computer:** Booting is a process or set of operations that loads and hence starts the operating system, starting from the point when user switches on the power button.
- **bootstrap program** is loaded at power-up or reboot
  - Typically stored in ROM or EPROM, generally known as **firmware**
  - Initializes all aspects of system
  - Loads operating system kernel and starts execution



# Computer-System Architecture

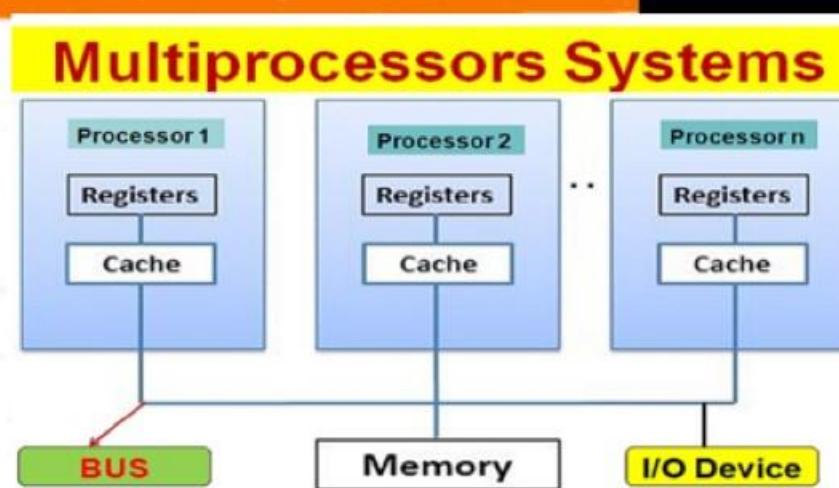
- A single-processor system contains only one CPU to execute general purpose instructions
  - However, it also contains special purpose processors
    - E.g., disk, keyboard, DMA, graphic controllers
    - These specialized processors do not run user processes
    - OS may be able to manage the processors, e.g., task disk controllers to use given scheduling algorithms.
- A multiprocessor system contains two or more processors working together
  - They may share computer bus, system clock, memory, I/O devices
  - Also called parallel system or multicore system

7

# Multiprocessor (cont.)

- Three main advantages of multiprocessor system

- **Increased throughput:** get more work done in less time
  - The speed up in  $n$  processor system is NOT  $n$  due to overhead
- **Economy of scale:** I/O devices, memory storage, and power supplies can be shared
- **Increased reliability:** failure of one processor does not make the whole system down
  - **Graceful degradation:** the system can perform operations proportional to the level of operations of the surviving parts of the system
  - **Fault-tolerant:** the system can continue its function in the event of component failures
    - Require failure detection, diagnoses, and even correction
    - May use multiple hardware and software duplicates to execute the same tasks in parallel, and take as output the result from the majority of the duplicates



# Multiprocessor (cont.)

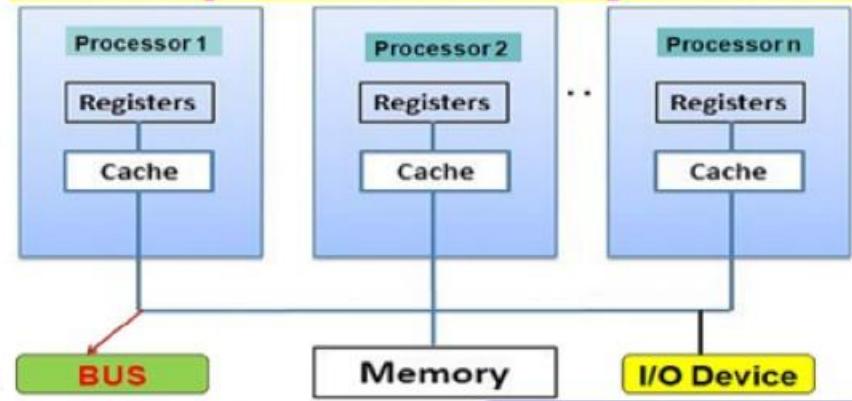
- Two types of multiprocessor system:

- Asymmetric multiprocessing
- Symmetric multiprocessing (SMP)

- Asymmetric multiprocessing:

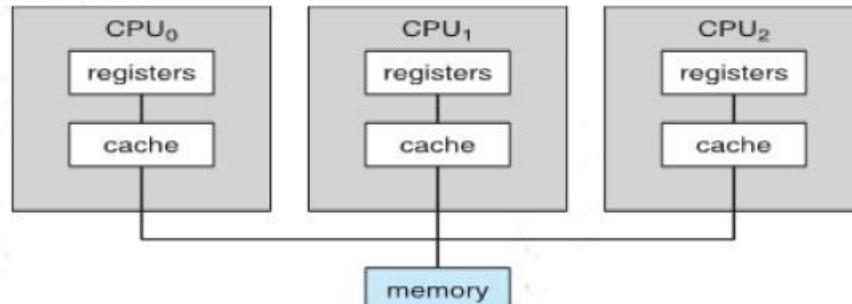
- Use a **master** processor to schedule and allocate work to (**slave**) processors.
- Each slave processor waits for instruction from the master or has predefined task
- More common in extremely large systems
- . . .
- . . .
- . . .
- . . .

## Multiprocessors Systems



# Multiprocessor Systems (cont. )

- SMP – not the master-slave model; a more common system
  - Each processor runs an identical copy of the OS.
  - Each processor has its own registers and cache
    - + However, they share the same memory
  - Many processes can run at once without significant performance deterioration
    - + Need load balancing to improve performance
- Symmetric and Asymmetric may be the result of either hardware or software.

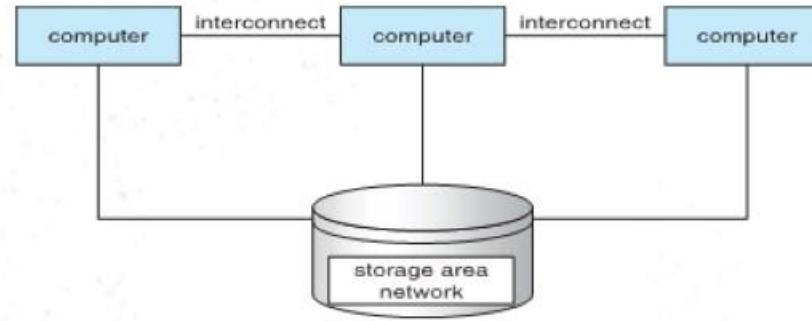


10

# Clustered Systems

- A clustered system consists of multiple CPUs, like multiprocessors

- However they are individual systems or nodes
- Each node can be a single processor or a multicore
- They share storage and communicate via LAN
- It offers high-availability service



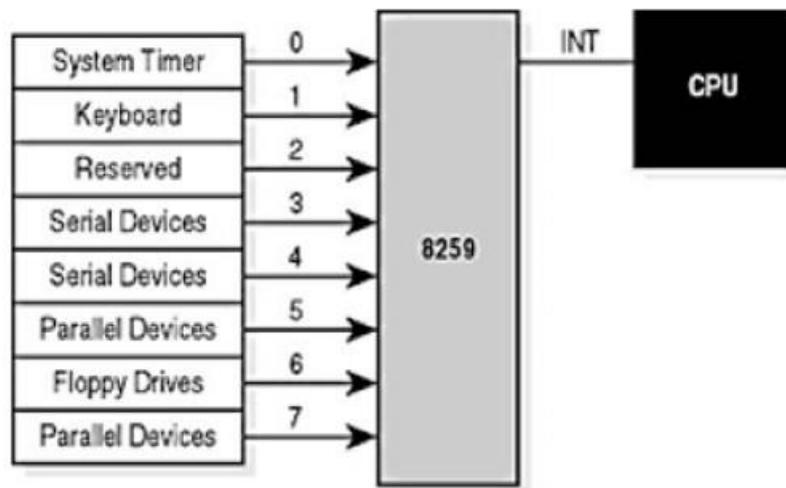
11

# Interrupts

- **Interrupts** are signals sent to the CPU by external devices, normally I/O devices. They tell the CPU to stop its current activities and execute the appropriate part of the operating system.

There are three types of interrupts:

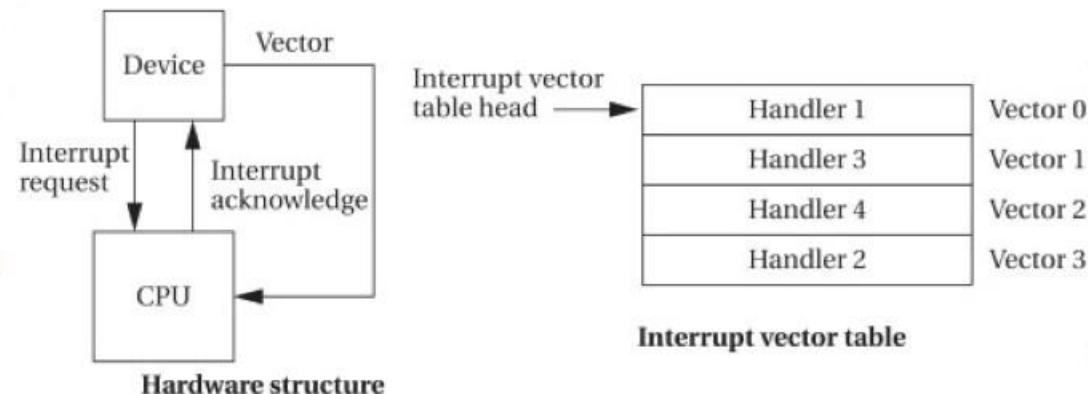
- **Hardware Interrupts** are generated by hardware devices to signal that they need some attention from the OS. They may have just received some data (e.g., keystrokes on the keyboard or data on the ethernet card); or they have just completed a task which the operating system previously requested, such as transferring data between the hard drive and memory.
- **Software Interrupts** are generated by programs when they want to request a system call to be performed by the operating system.
- **Traps** are generated by the CPU itself to indicate that some error or condition occurred for which assistance from the operating system is needed.



# Interrupt Handling

## Common functions of interrupts

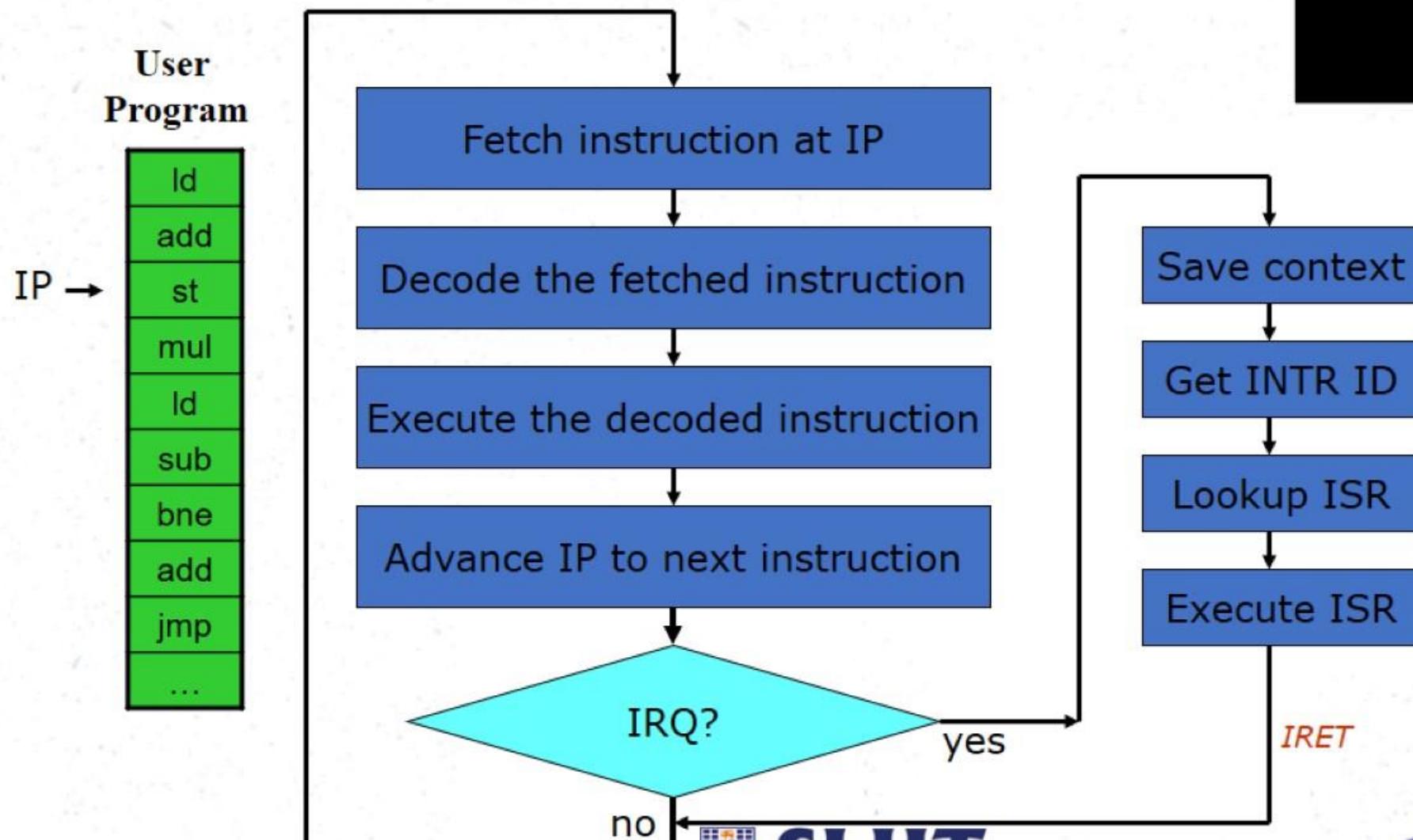
- An interrupt is an event that suspends the execution of one program and begins the execution of another program.
- For each type of interrupt, separate segments of code in the OS must determine what action should be taken
- This code is called **INTERRUPT SERVICE ROUTINE**.
- Associated with each I/O device there is a location near the bottom of memory called **INTERRUPT VECTOR**
- This contains the address of the interrupt service routine for the various devices.



# Interrupt Handling

- When an interrupt (or trap) , hardware transfers control to OS
- The OS preserves the state of the CPU by storing registers and the Program Counter.
- Separate segments of code (Interrupt Service Routine) determine what action should be taken for each type of interrupt.

# CPU's 'fetch-execute' cycle



# Intel Pentium interrupt vector table

Figure 13.4 (textbook)

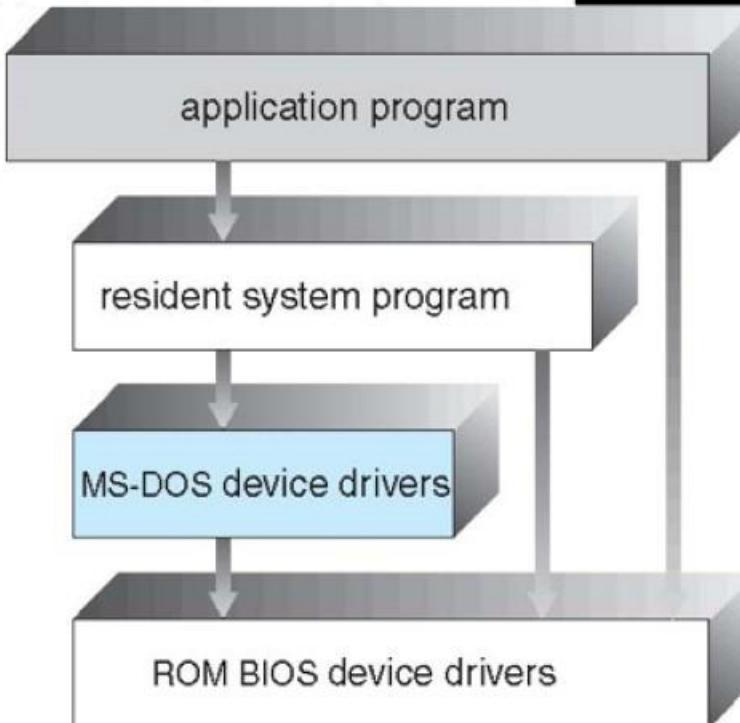
vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
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

# Operating System Structure

- General-purpose OS is very large program
- Various ways to structure ones
  - Simple structure – MS-DOS
  - More complex -- UNIX
  - Layered – an abstraction
  - Microkernel -Mach
- . . .
- . . .
- . . .
- . . .
- . . .
- . . .
- . . .

# Simple Structure -- MS-DOS

- MS-DOS – written to provide the most functionality in the least space
- Not divided into modules
- Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated



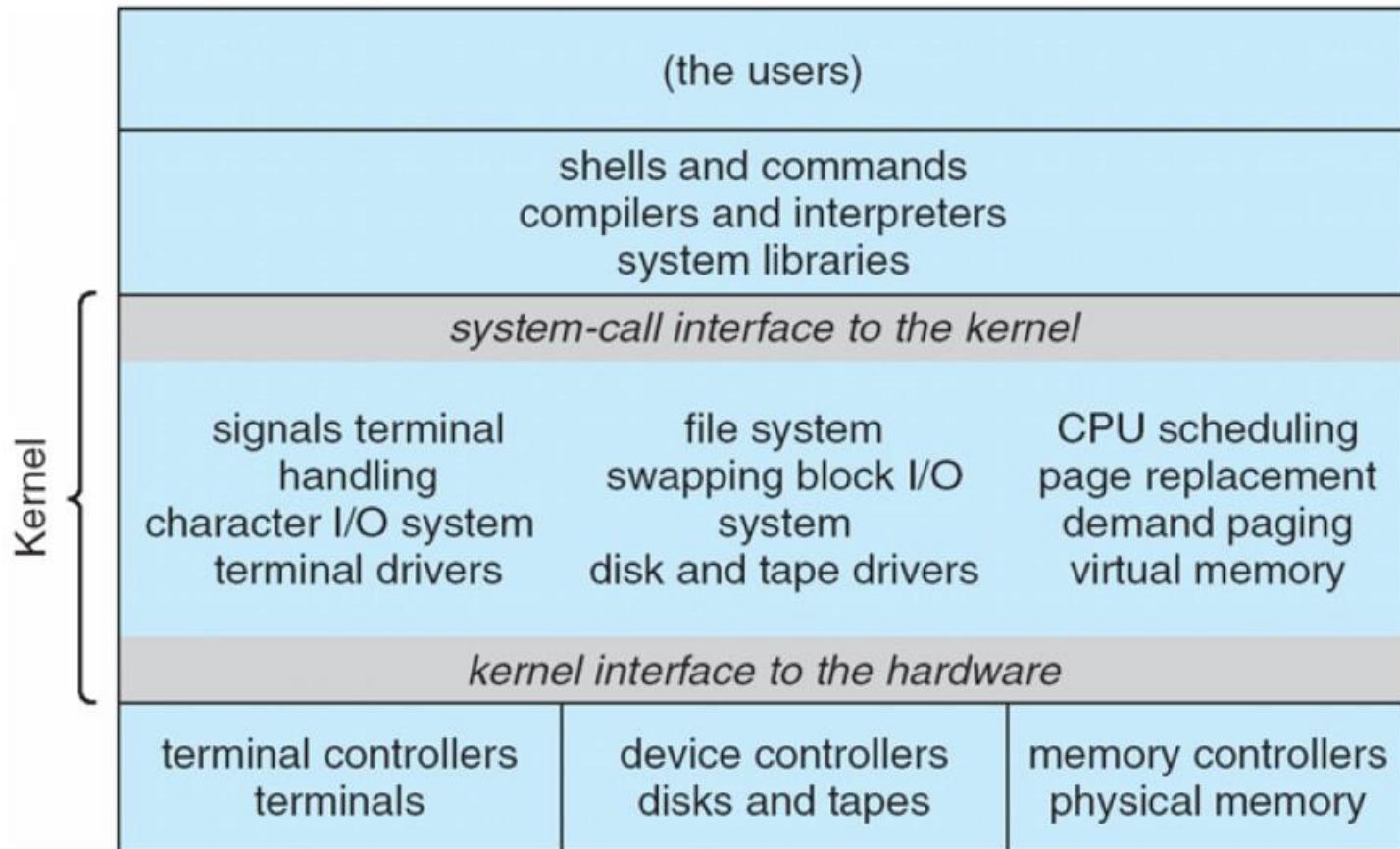
# Non Simple Structure -- UNIX

UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts

- Systems programs
- The kernel
  - Consists of everything below the system-call interface and above the physical hardware
  - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

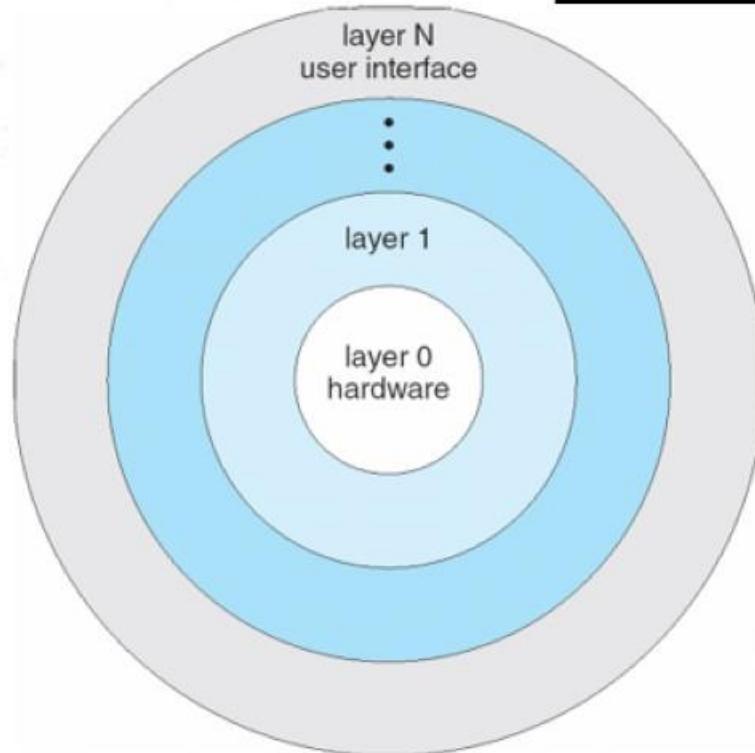
# Traditional UNIX System Structure

Beyond simple but not fully layered



# Layered Approach

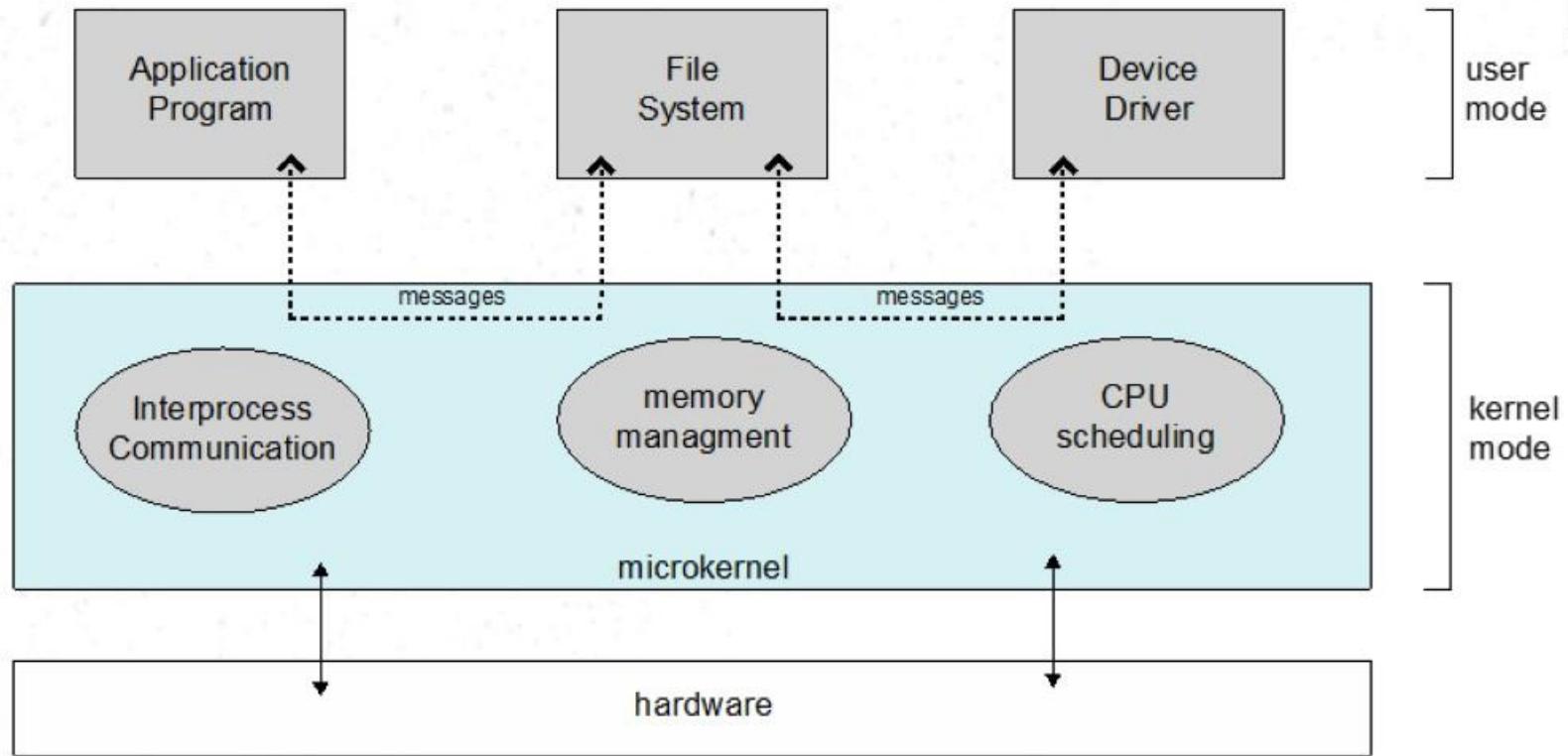
- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.  
• . . .  
• . . .  
• . . .  
• . . .  
• . . .  
• . . .  
• . . .
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers



# Microkernel System Structure

- Moves as much from the kernel into user space
- **Mach** example of **microkernel**
  - Mac OS X kernel (**Darwin**) partly based on Mach
- Communication takes place between user modules using **message passing**
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

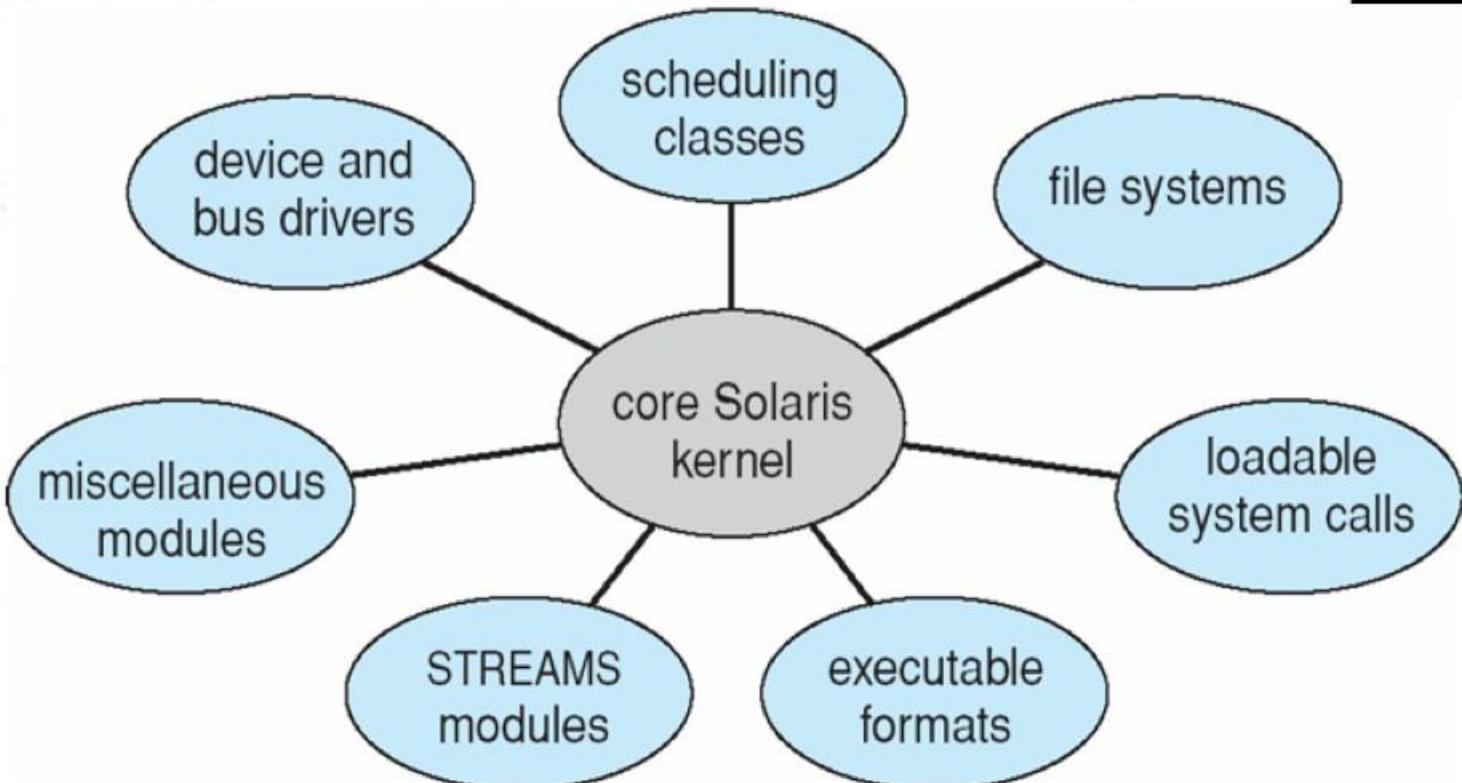
# Microkernel System Structure



# Modules

- Many modern operating systems implement **loadable kernel modules**
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
  - Overall, similar to layers but with more flexible
    - Linux, Solaris, etc

# Solaris Modular Approach



Thank you very much



# SLIIT

*Discover Your Future*

# IT2060/IE2061

## Operating Systems and System Administration

### Lecture 03

### Introduction to Processes

**U. U. Samantha Rajapaksha**

M.Sc.in IT, B.Sc.(Engineering) University of Moratuwa

Senior Lecturer SLIIT

[Samantha.r@slit.lk](mailto:Samantha.r@slit.lk)

# Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Inter process Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

# Process Concept

- **Process** – a program in execution.
- Program is a *passive* entity stored on disk (**executable file**), process is *active*.
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes when multiple users executing the same program.

# CPU

PC=4

Memory Address	Content
0	INS 1(Executed)
1	INS 2(Executed)
2	INS 3(Executed)
3	INS 4(Executed)
4	INS 5
5	

# RAM

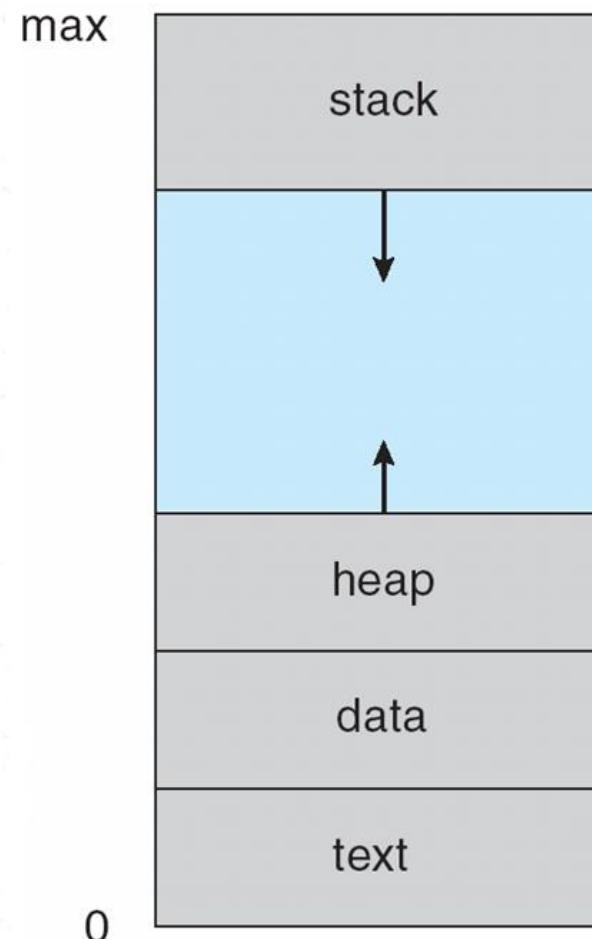
# Process Concept (Cont.)

- Process execution is sequential.
- A process has a **Program Counter**.
  - It's a register entry which specifies the next instruction to execute.
- A process needs resources (CPU time, memory, files and I/O devices) to complete the execution successfully.

# Process Concept (Cont.)

- A process consists of multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

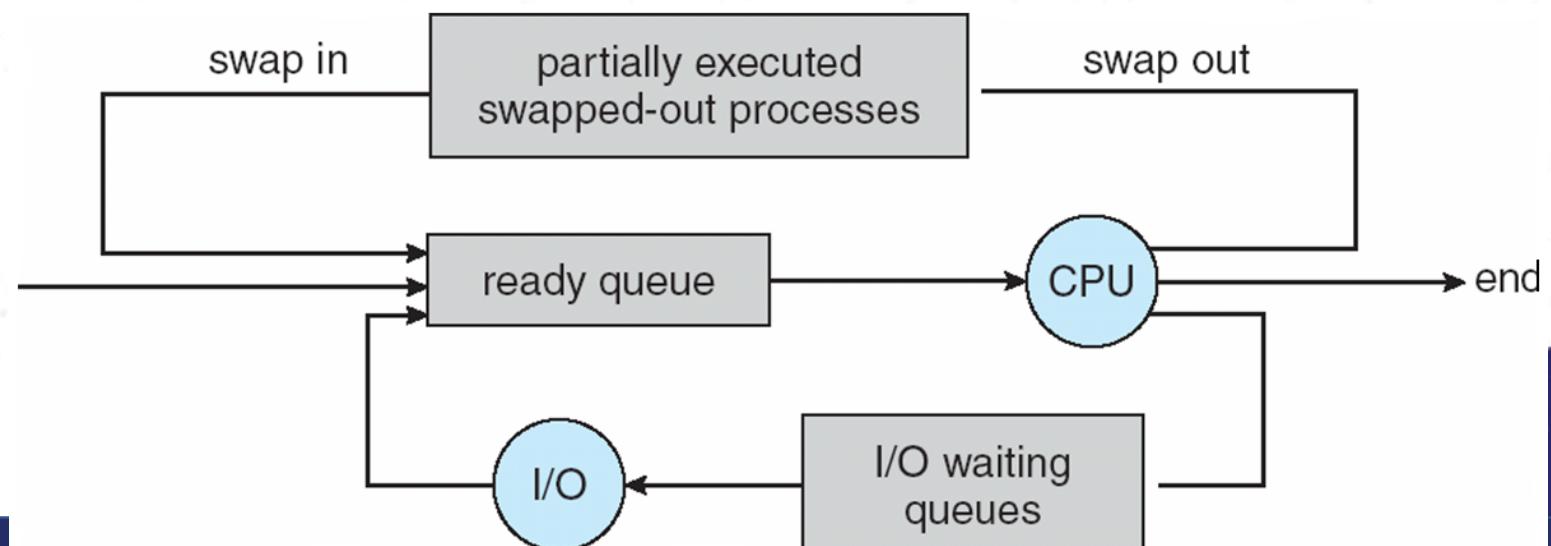
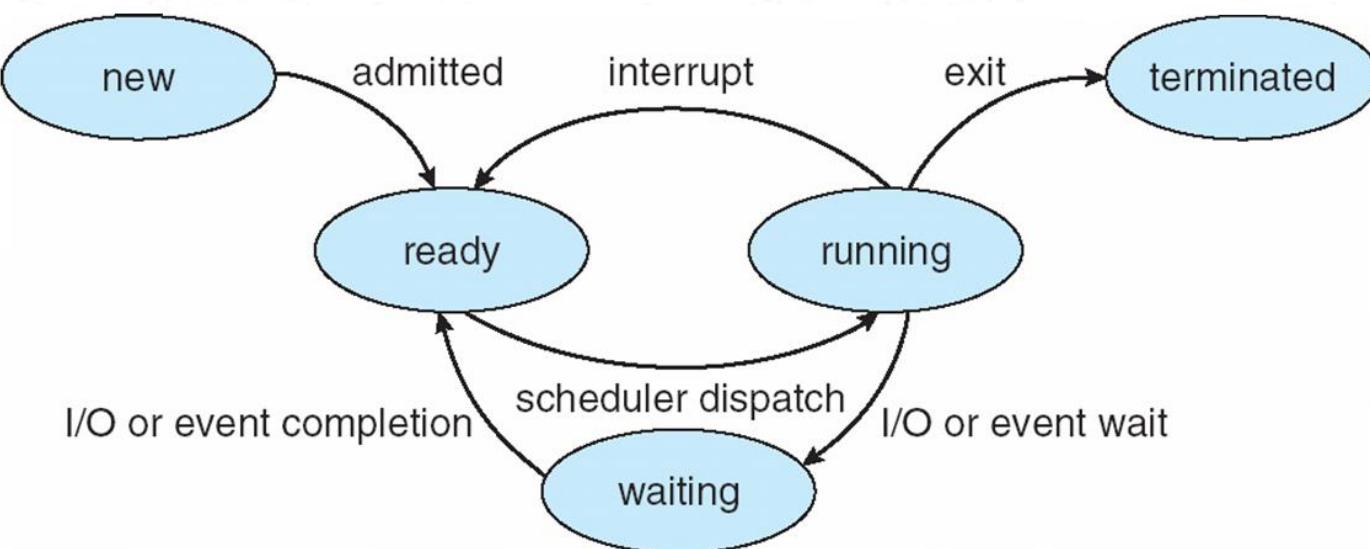
# Process in Memory

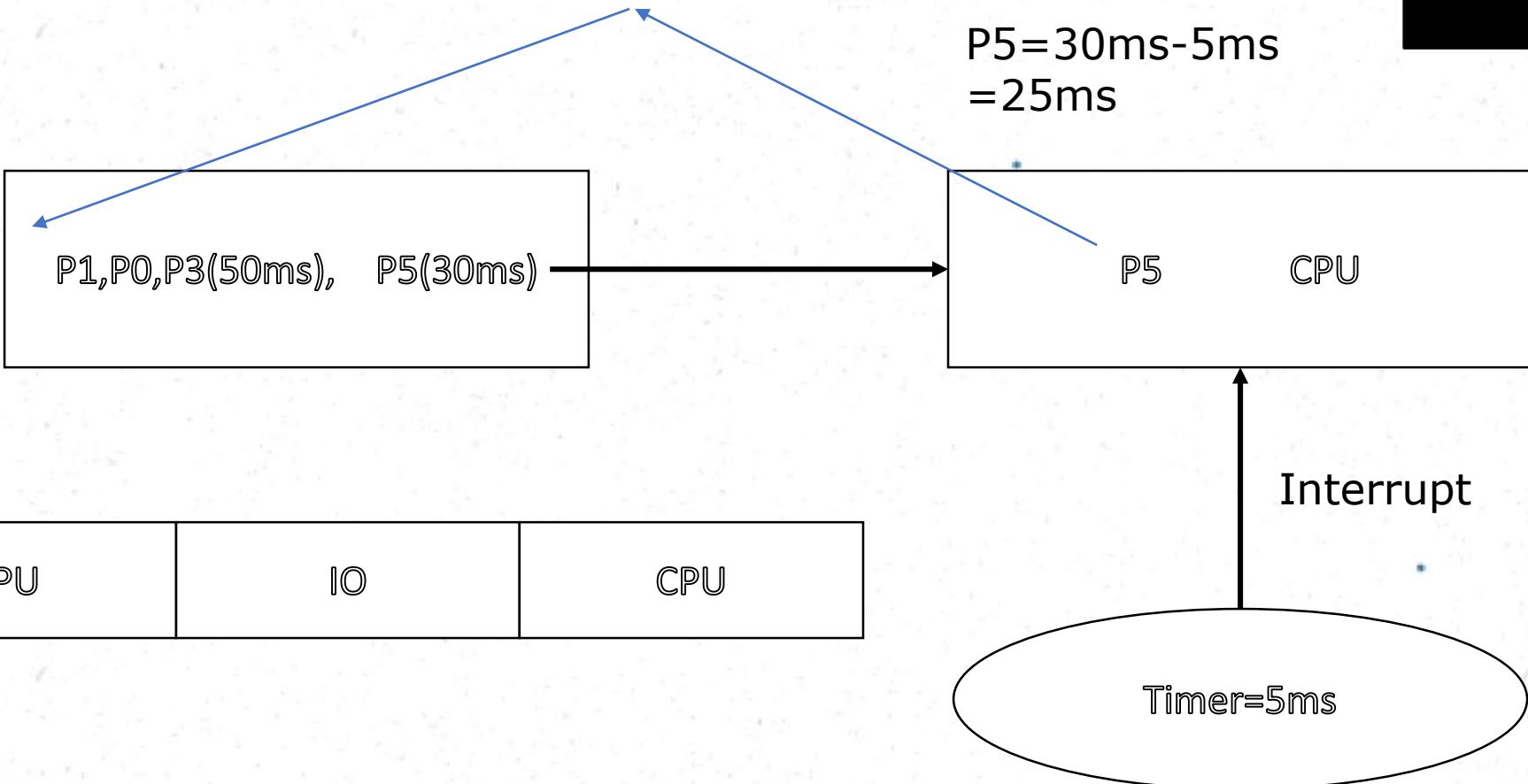
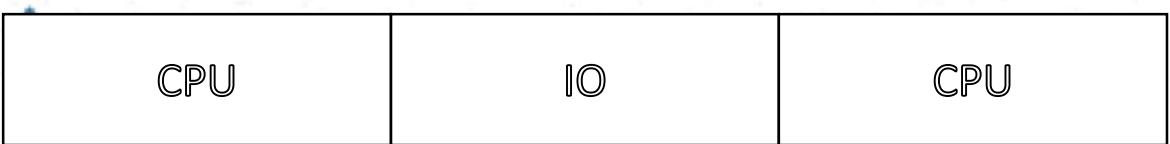


# Process State

- As a process executes, it changes **state**
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

# Diagram of Process State

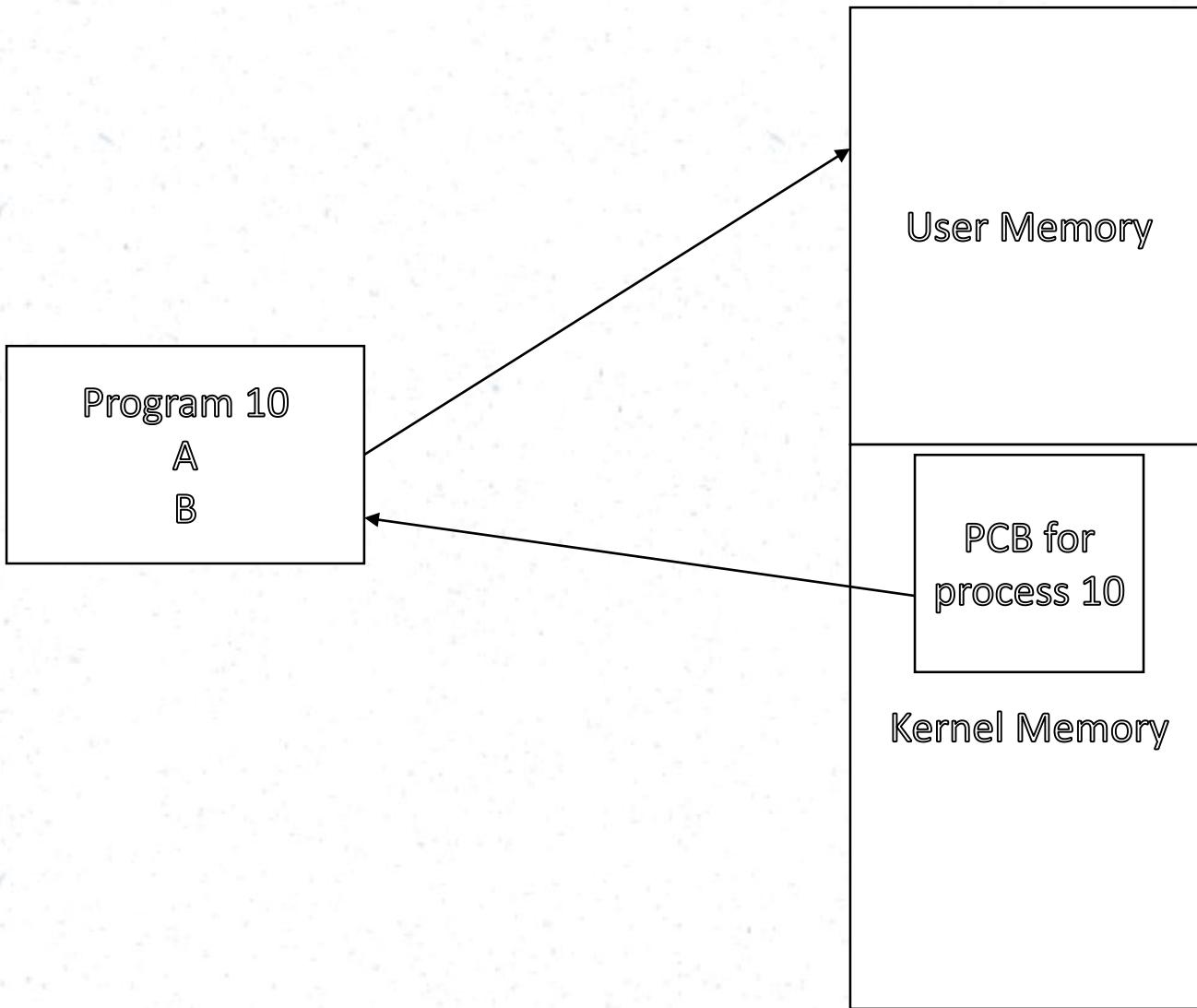




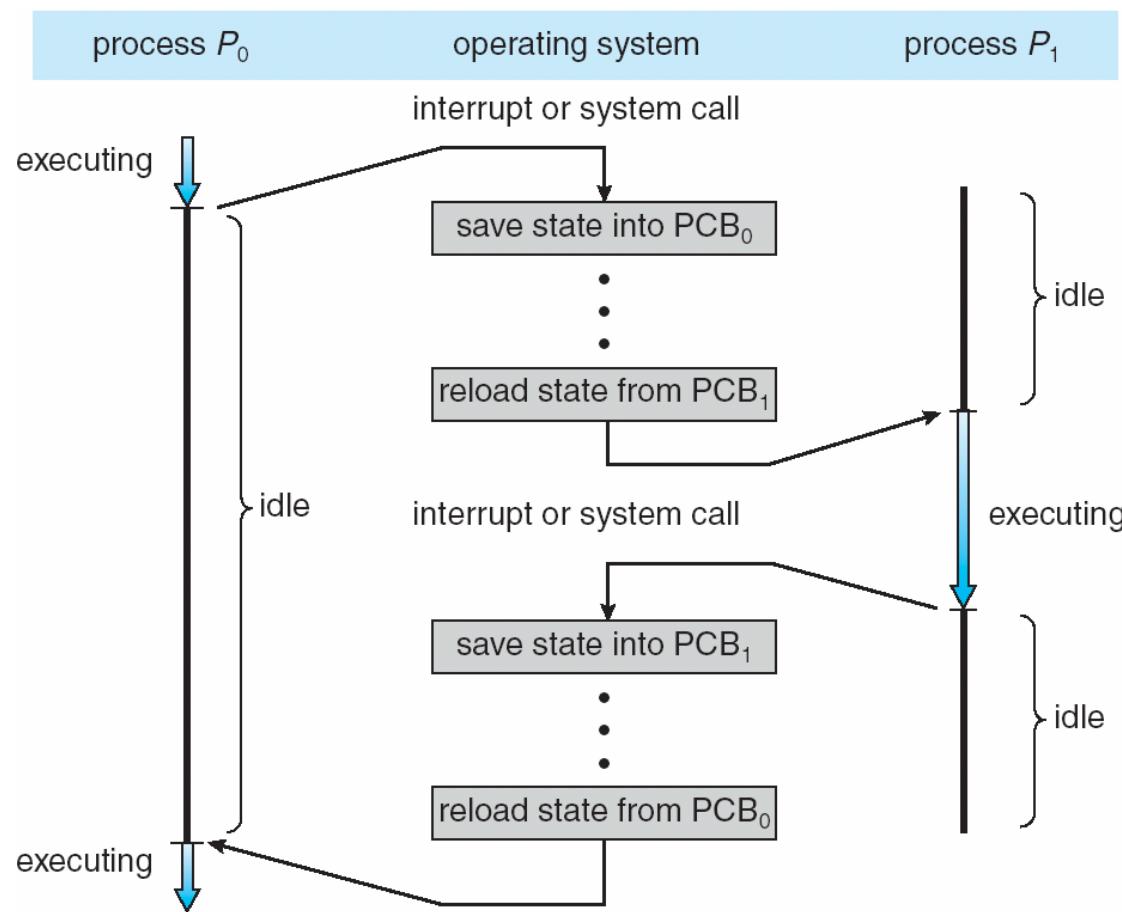
# Process Control Block (PCB)

- Information of each process is in PCB  
(also called **task control block**)
- Each PCB contains:
  - Process state – running, waiting, etc
  - Process number (process ID)
  - Program counter – location of instruction to next execute
  - CPU registers – contents of all process-centric registers
  - CPU scheduling information- priorities, scheduling queue pointers
  - Memory-management information – memory allocated to the process
  - Accounting information – CPU used, clock time elapsed since start, time limits
  - I/O status information – I/O devices allocated to process, list of open files





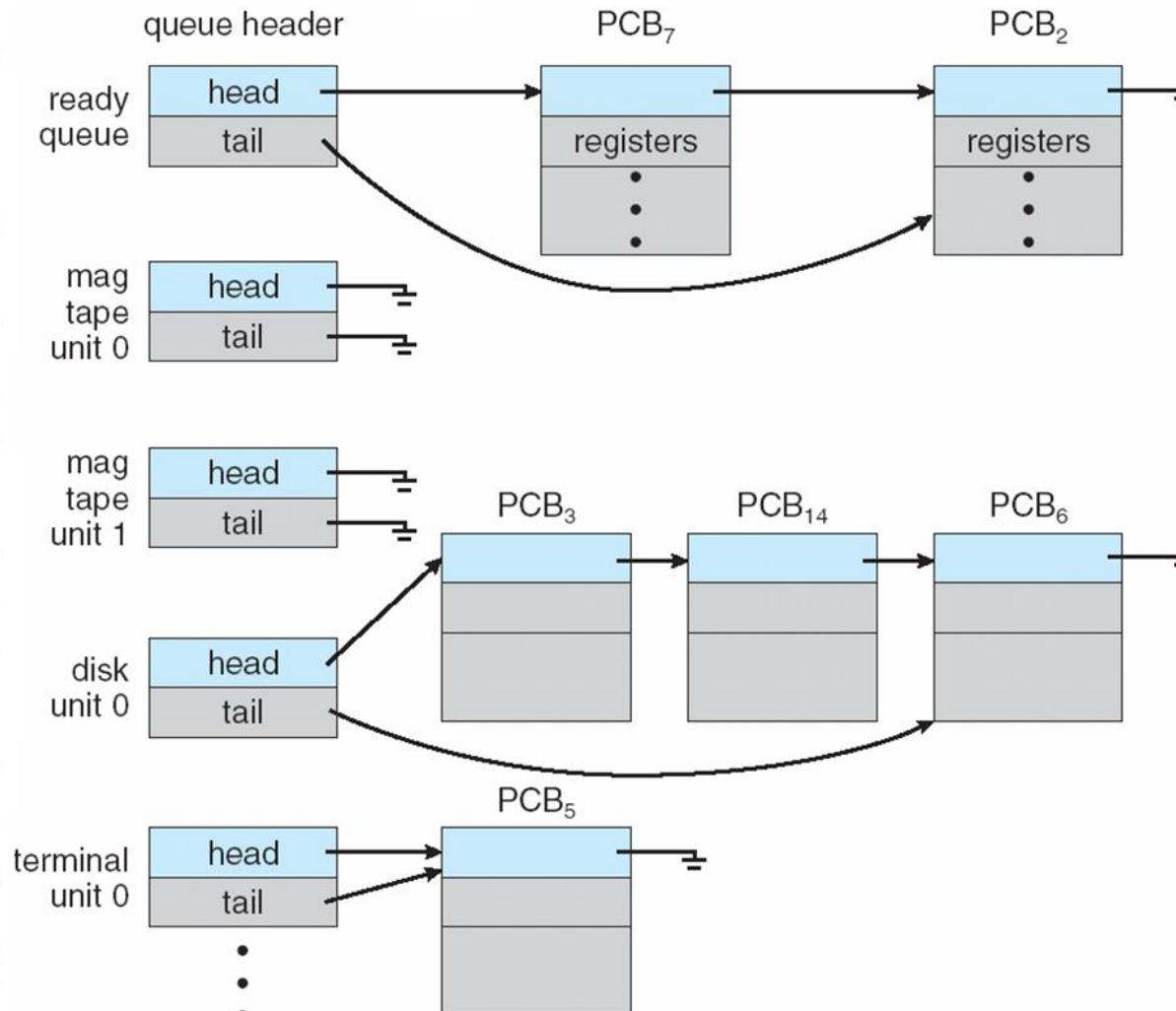
# CPU Switch From Process to Process



# Process Scheduling

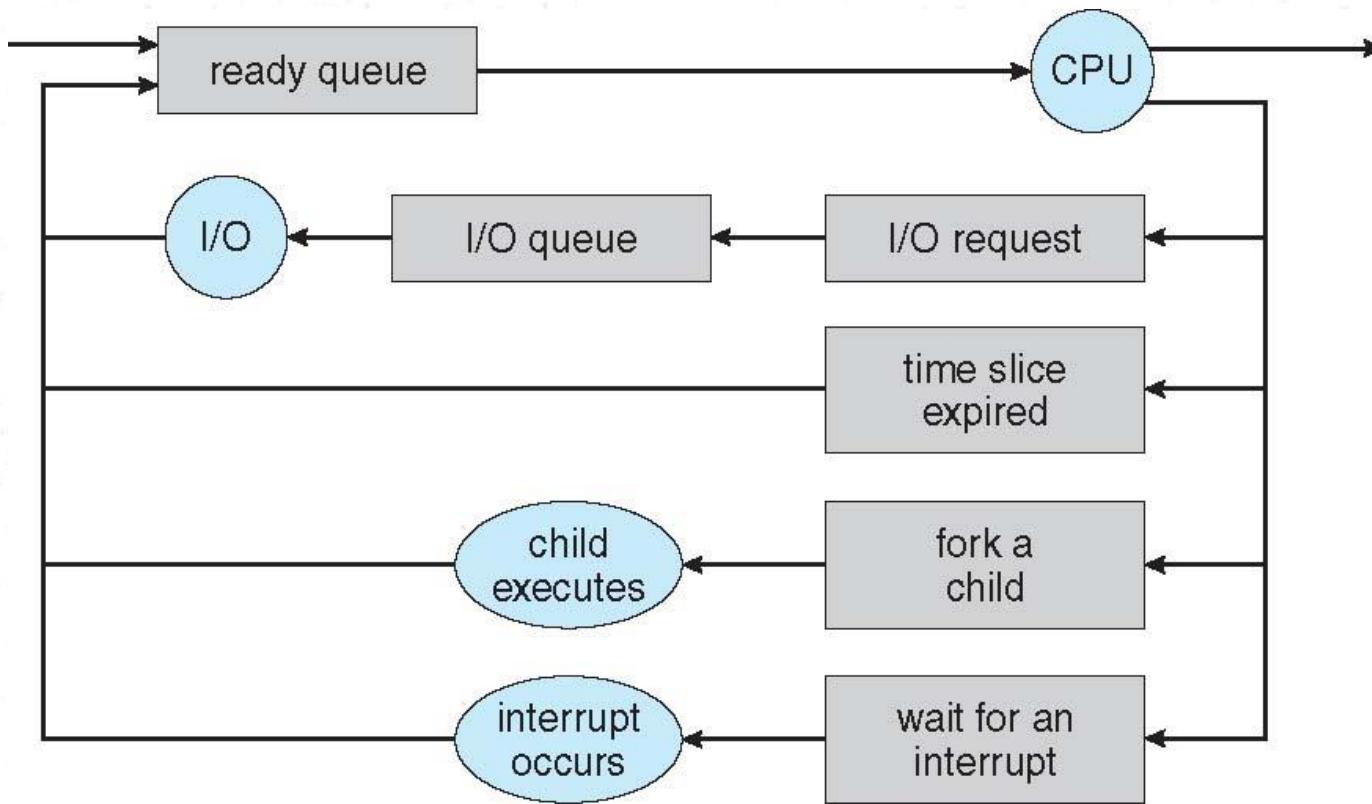
- **Process scheduler** selects among available processes for next execution on CPU
  - Scheduler in multiprogramming environment maximizes CPU use.
  - In time sharing, it quickly switches processes onto CPU
  
- There are several **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

# Ready Queue And Various I/O Device Queues



# Representation of Process Scheduling

- Queueing diagram represents queues, resources, flows

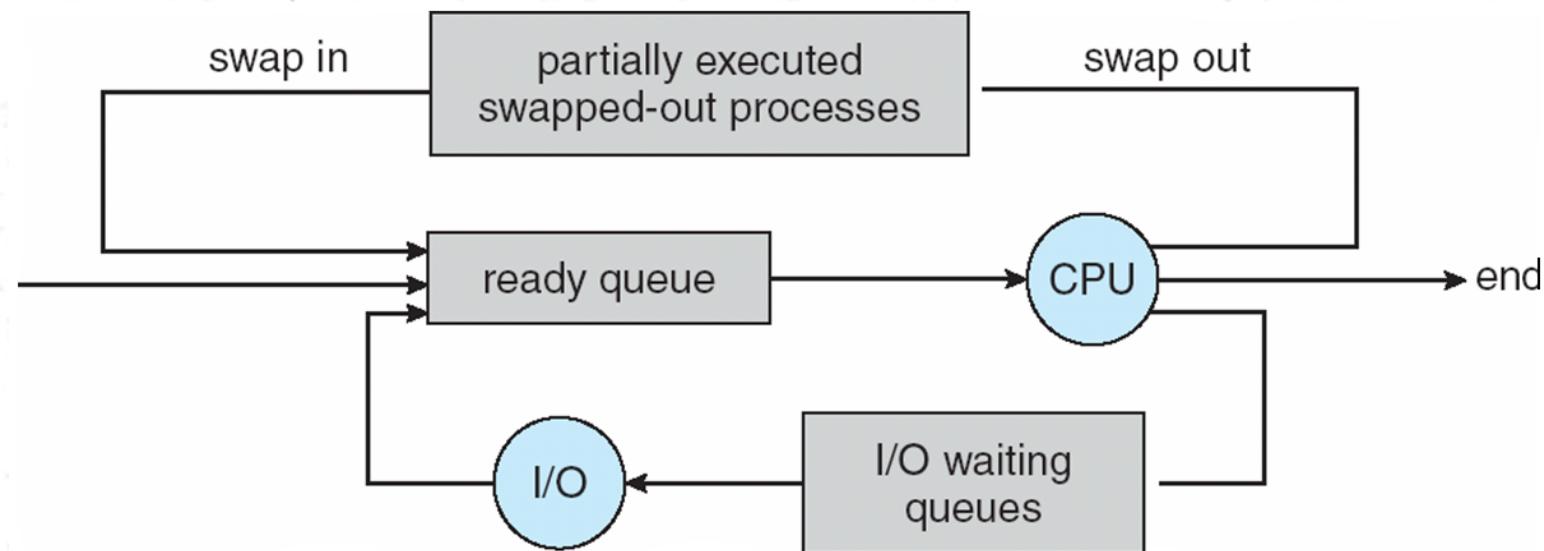


# Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds)  $\Rightarrow$  (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes)  $\Rightarrow$  (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***

# Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



# Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB → the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once

# Operations on Processes

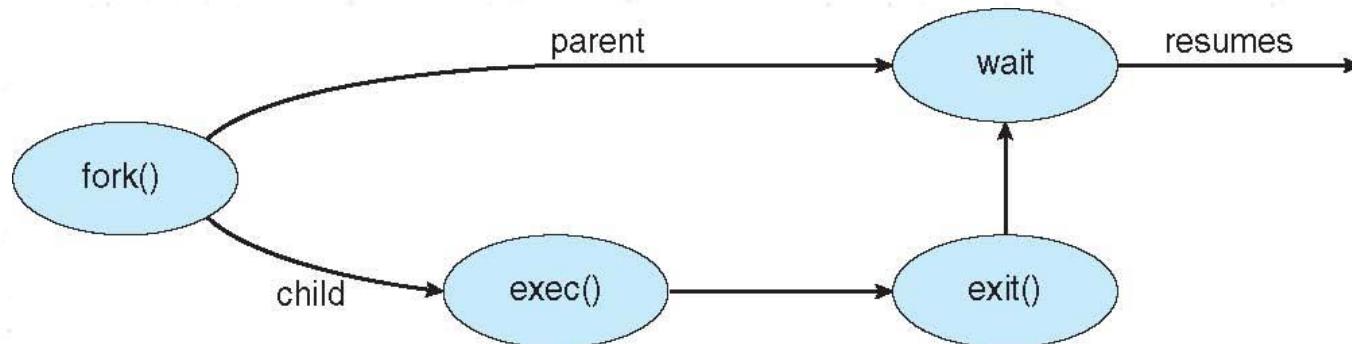
- System must provide mechanisms for:
    - process creation,
    - process termination,

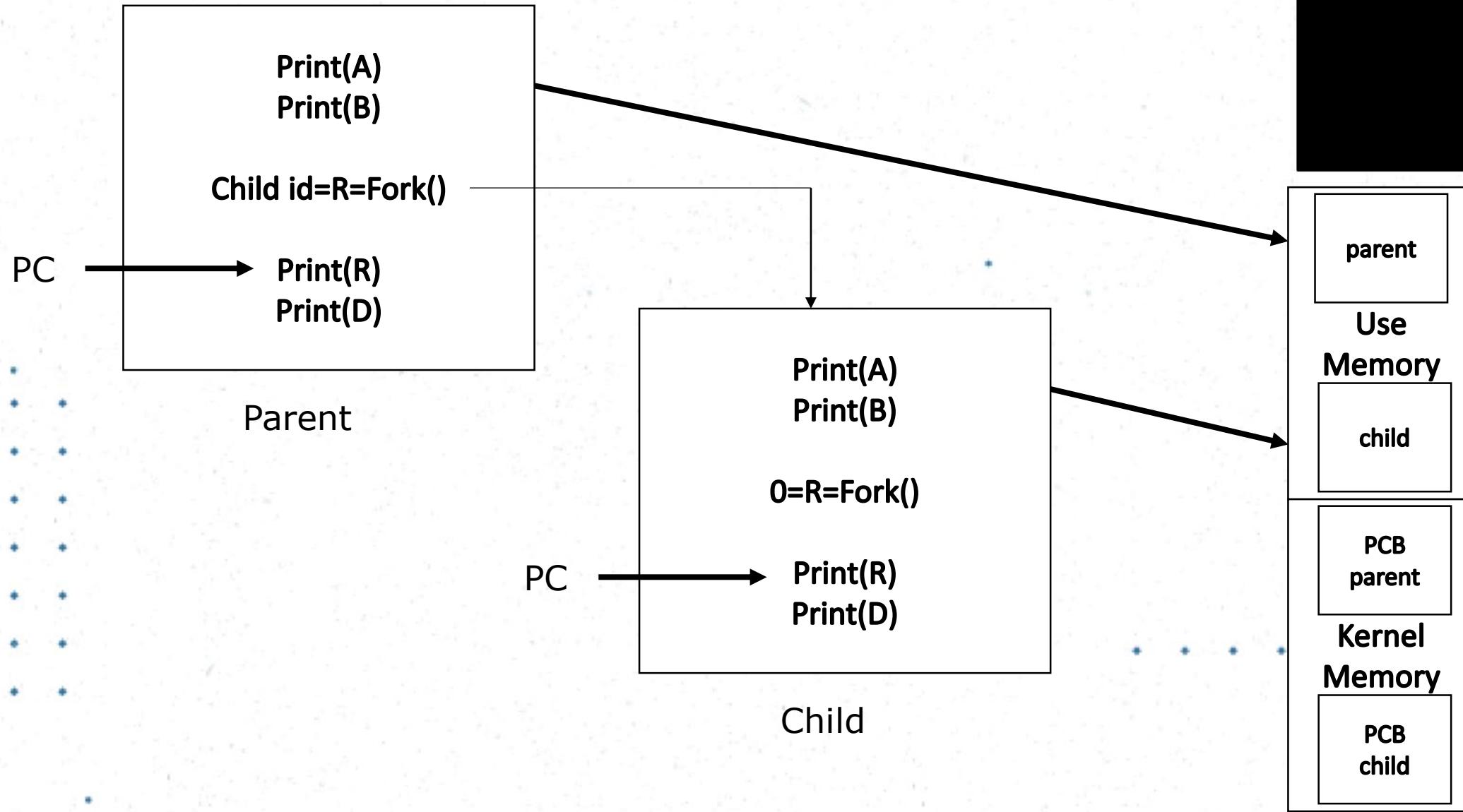
# Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (pid)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

# Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork()** system call creates new process
  - **exec()** system call used after a **fork()** to replace the process' memory space with a new program
- • •



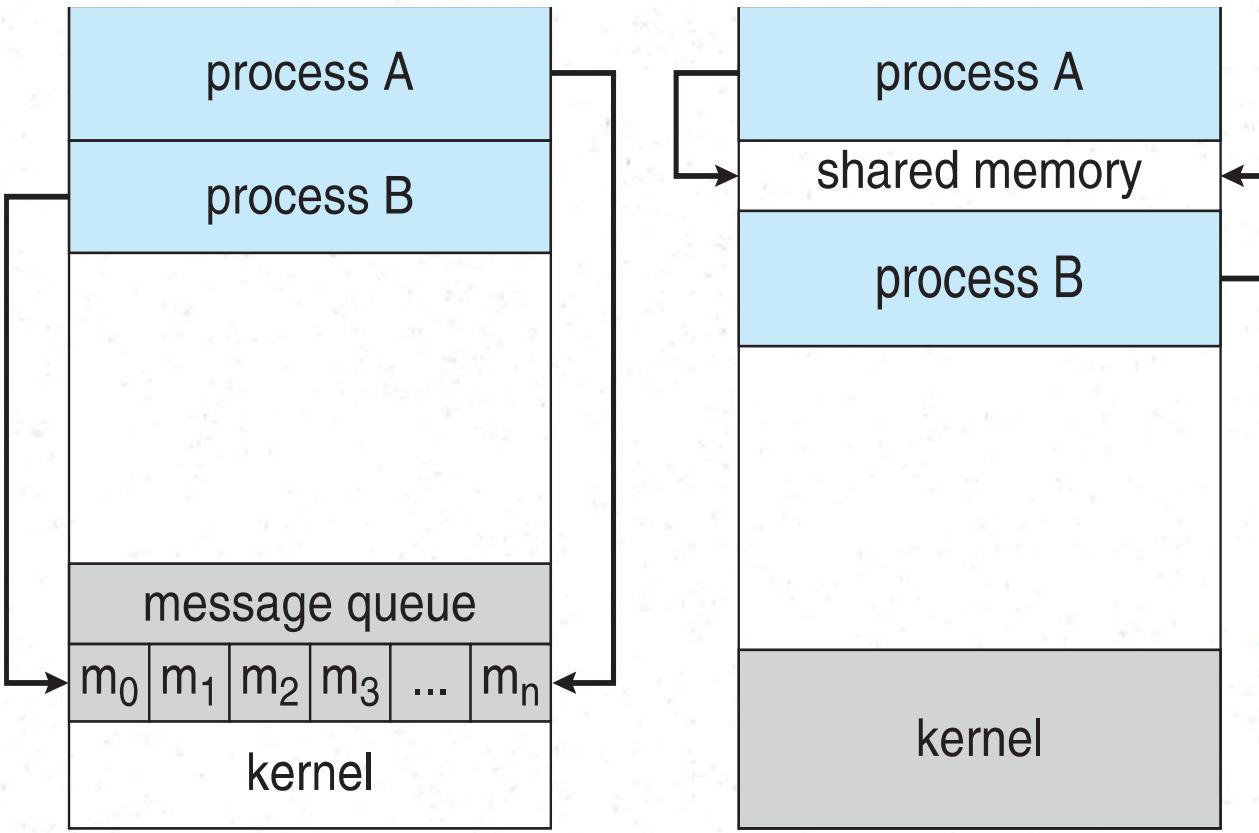


# Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
  - **Shared memory**
  - **Message passing**

# Communications Models

(a) Message passing. (b) shared memory.



(a)



(b)

# IPC through shared memory

Inter Process Communication through shared memory is a concept where two or more process can access the common memory. And communication is done via this shared memory where changes made by one process can be viewed by another process.

The problem with pipes, fifo and message queue – is that for two process to exchange information. The information has to go through the kernel

# System call in SM

ftok(): is use to generate a unique key.

shmget(): int shmget(key\_t, size\_t, int shmflg); upon successful completion, shmget() returns an identifier for the shared memory segment.

shmat(): Before you can use a shared memory segment, you have to attach yourself

to it using shmat(). void \*shmat(int shmid, void \*shmaddr, int shmflg);  
shmid is shared memory id. shmaddr specifies specific address to use but we should set it to zero and OS will automatically choose the address.

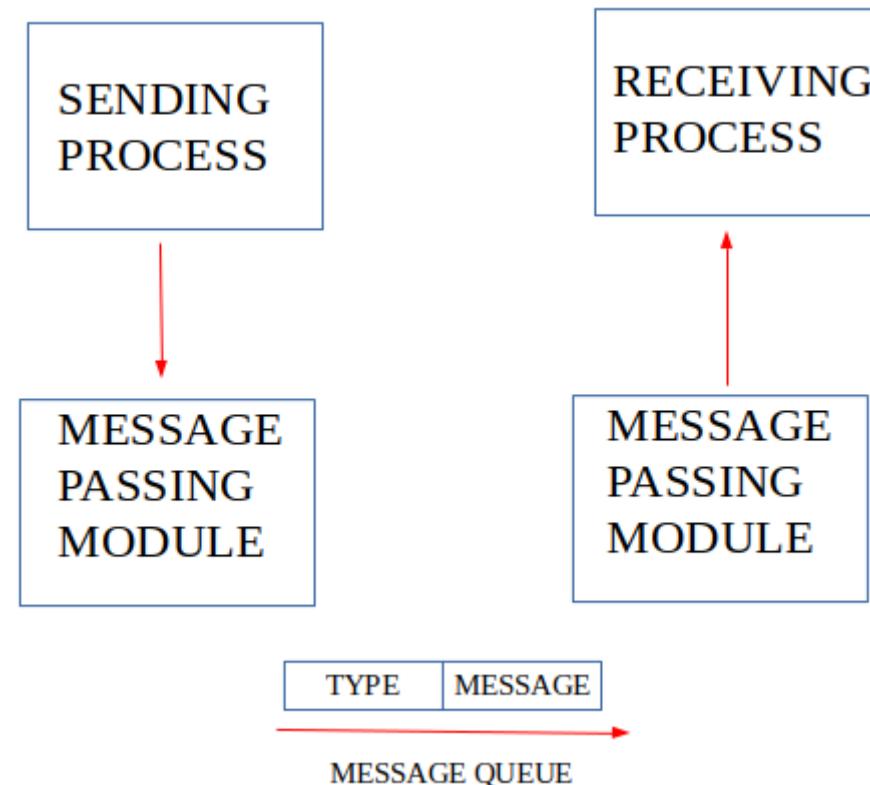
shmdt(): When you're done with the shared memory segment, your program should detach itself from it using shmdt(). int shmdt(void \*shmaddr);

shmctl(): when you detach from shared memory, it is not destroyed. So, to destroy

shmctl() is used. shmctl(int shmid, IPC\_RMID, NULL);

# IPC using Message Queues

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened by msgget().



# System calls in MQ

ftok(): is use to generate a unique key.

msgget(): either returns the message queue identifier for a newly created message queue or returns the identifiers for a queue which exists with the same key value.

msgsnd(): Data is placed on to a message queue by calling msgsnd().

msgrcv(): messages are retrieved from a queue.

msgctl(): It performs various operations on a queue. Generally it is use to destroy message queue.

# Pipes

Oldest (and perhaps simplest) form of UNIX IPC

Half duplex.

The oldest mechanism for IPC in Unix is pipes.

Here's the prototype:

```
#include <unistd.h>  
  
int pipe(int filedes[2]);
```

Pipe takes an array of two ints (two file descriptors) and, if the kernel succeeds in creating the pipe, it puts the file descriptor for the reading end of the pipe in the 0th entry

e.g. filedes[0], and it puts the file descriptor for the write end of the pipe in the 1st entry,

e.g. filedes[1]. Pipe returns 0 if successful and -1 otherwise.

# Example

Let's do the example we just did with FIFOs with pipes.

Here it is:

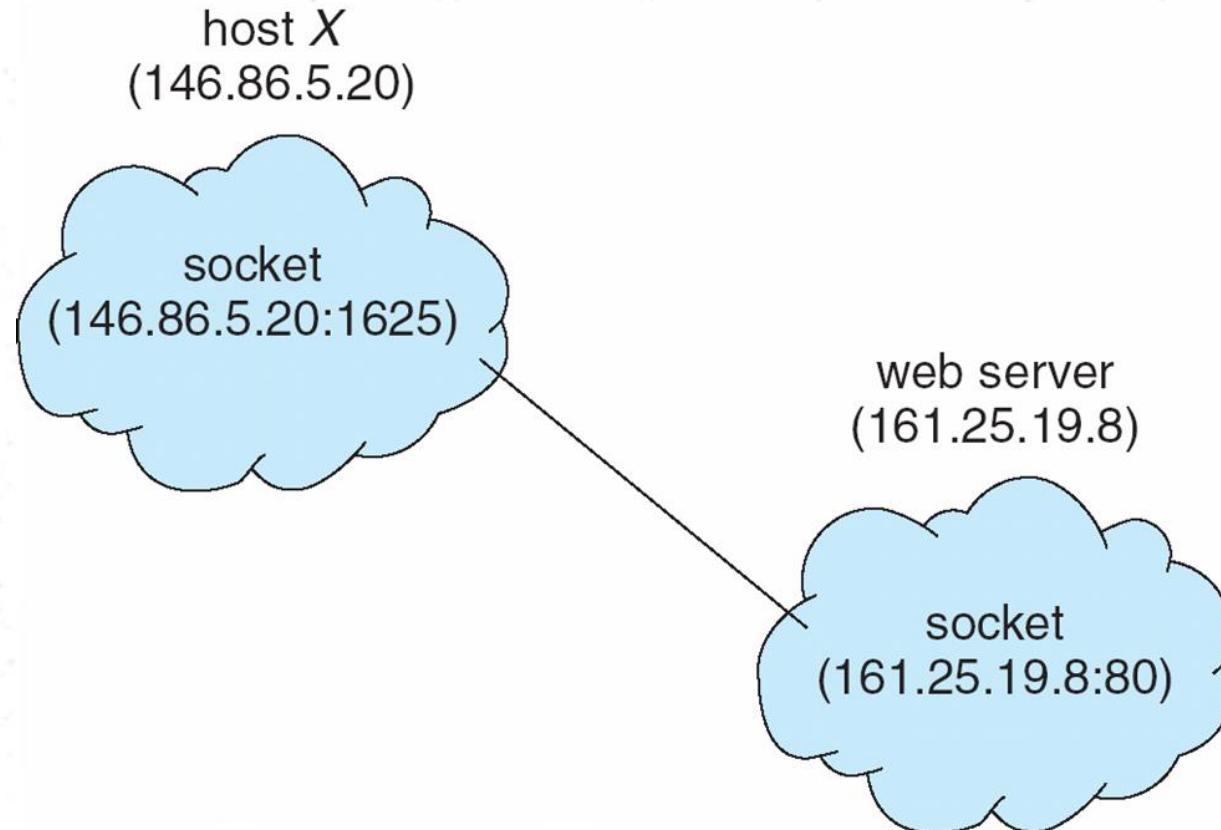
```
#include <unistd.h>

int main()
{
    int pfd[2], fv;
    pipe(pfd);
    fv = fork();
    if (fv)
    {
        close(pfd[0]);
        dup2(pfd[1], STDOUT_FILENO);
        execlp("cat", "cat", NULL);
    }
    else
    {
        close(pfd[1]);
        dup2(pfd[0], STDIN_FILENO);
        execlp("tr", "tr", " ", "x", NULL);
    }
    return 0;
}
```

# Sockets

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are **well known**, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running

# Socket Communication





**U.U.Samantha Rajapaksha**  
BSc. Eng. (Moratuwa), MSc in IT  
Senior Lecturer  
Sri Lanka Institute of Information Technology  
New Kandy Road,  
Malabe, Sri Lanka  
Tel:0112-301904  
email: [samantha.r@sliit.lk](mailto:samantha.r@sliit.lk)  
Web: [www.sliit.lk](http://www.sliit.lk)



# SLIIT

*Discover Your Future*

# IT2060/IE2061

## Operating Systems and System Administration

### Lecture 04

### Introduction to Threads

**U. U. Samantha Rajapaksha**

M.Sc.in IT, B.Sc.(Engineering) University of Moratuwa

Senior Lecturer SLIIT

[Samantha.r@slit.lk](mailto:Samantha.r@slit.lk)

# Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks within the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

# Threads

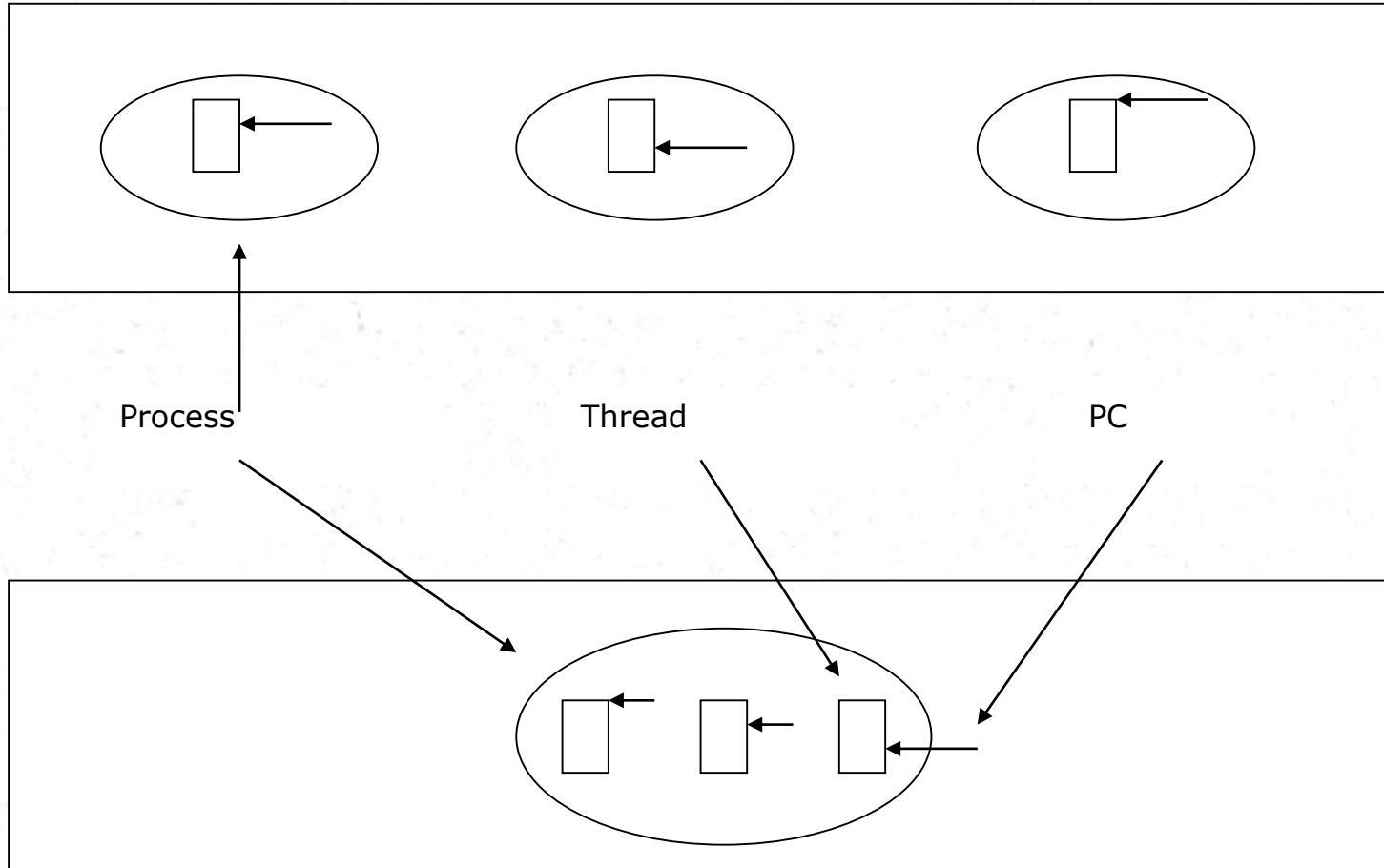
- A thread of control is an independent sequence of execution of program code in a process.
- A thread (or lightweight process) is a basic unit of CPU utilization.
  - A traditional process (or heavyweight process) is equal to a task with one thread.
- A traditional process has a single thread that has sole possession of the process's memory and other resources → context switch becomes performance bottleneck
  - Threads are used to avoid the bottleneck.
  - Threads share all the process's memory, and other resources.
- Threads within a process:
  - are generally invisible from outside the process.
  - are scheduled and executed independently in the same way as different single-threaded processes.
- On a multiprocessor, different threads may execute on different processors
  - On a uni-processor, threads may interleave their execution arbitrarily.

# Threads (cont.)

- Threads operate, in many respects, in the same manner as processes:
  - Threads can be in one of several states: ready, blocked, running, or terminated, etc.
  - Threads share CPU; only one thread at a time is running.
  - A thread within a process executes sequentially, and each thread has its own PC and Stack.
  - Thread can create child threads, can block waiting for system calls to complete
    - if one thread is blocked, another can run.
- One major difference with process: threads are not independent of one another
  - all threads can access every address in the task → a thread can read or write any other thread's stack.
  - There is no protection between threads (within a process);
    - however this should not be necessary since processes may originate from different users and may be hostile to one another while threads (within a process) should be designed (by same programmer) to assist one another.

# Threads (cont.)

Three processes with one thread each

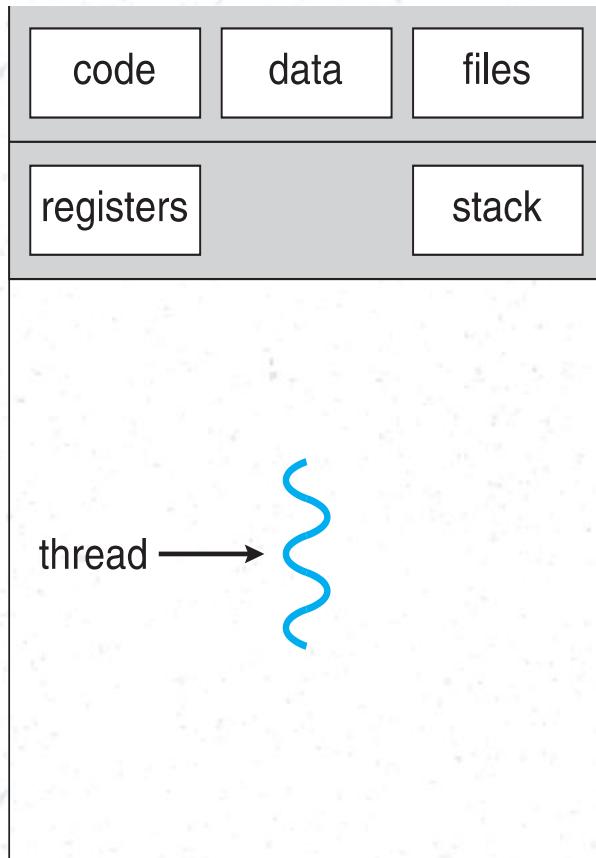


One process with three threads

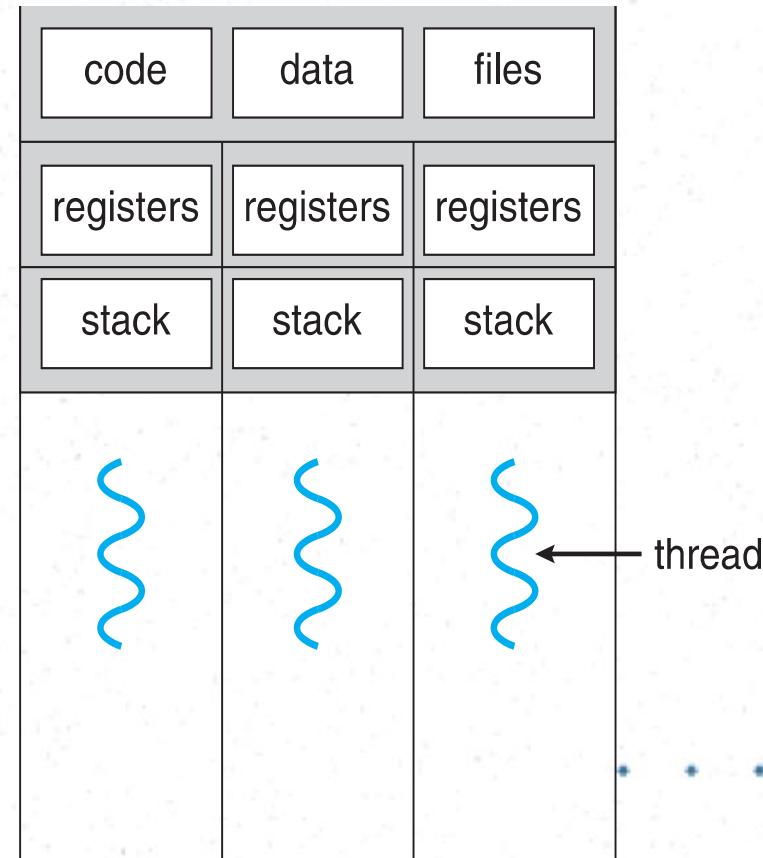
# Benefits

- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multiprocessor architectures

# Single and Multithreaded Processes



single-threaded process



multithreaded process

# User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

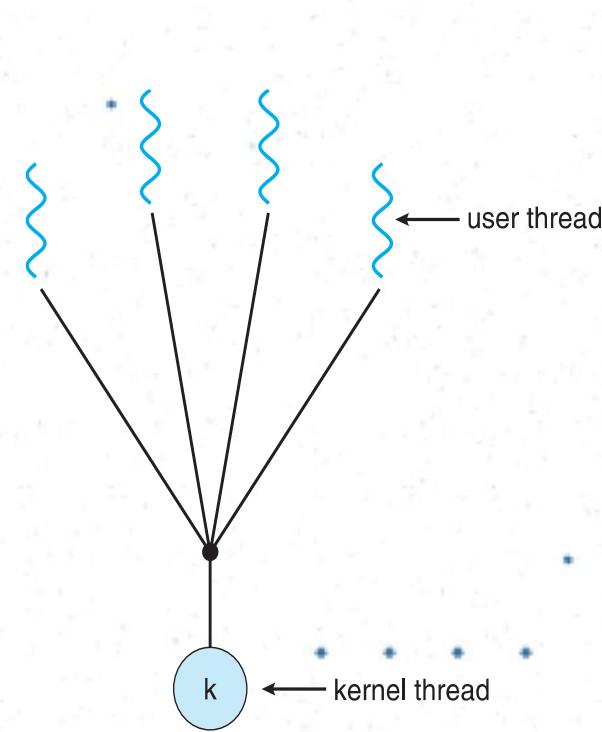
# Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many



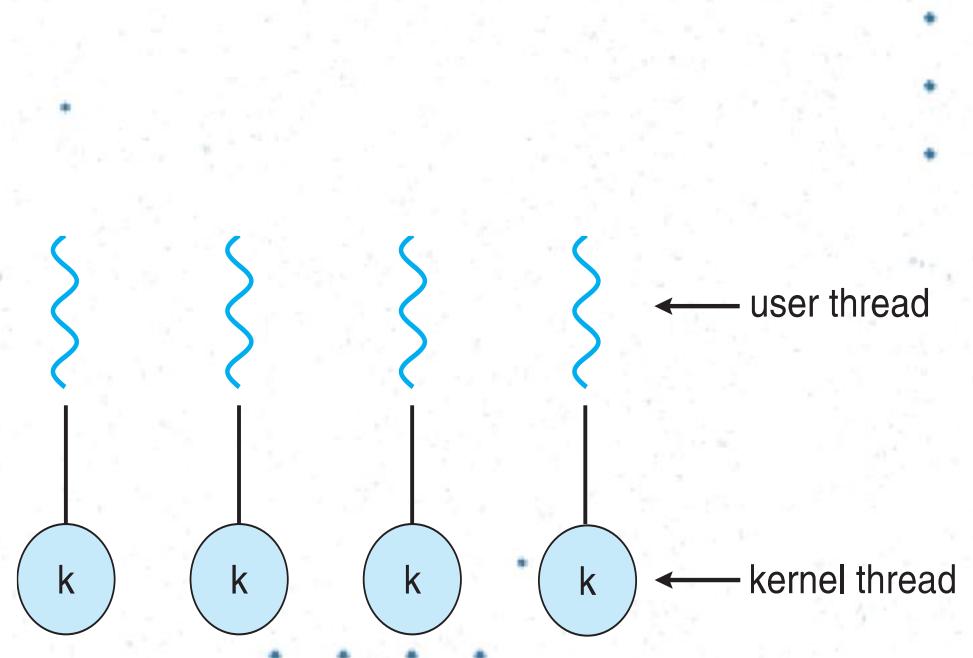
# Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



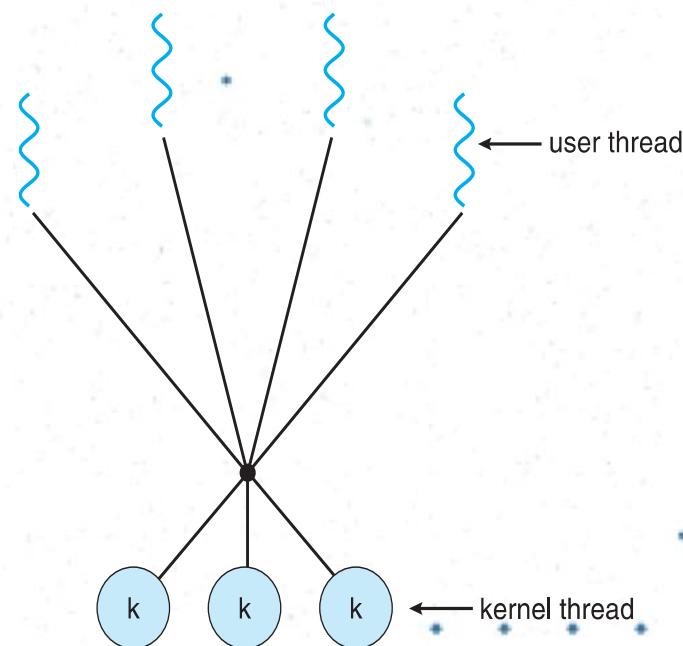
# One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



# Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



# Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

# Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

# Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - i.e. Tasks could be scheduled to run periodically

# Signal Handling

- n **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- n A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
    1. default
    2. user-defined
- n Every signal has **default handler** that kernel runs when handling signal
  - | **User-defined signal handler** can override default
  - | For single-threaded, signal delivered to process

# Signal Handling (Cont.)

- n Where should a signal be delivered for multi-threaded?
  - | Deliver the signal to the thread to which the signal applies
  - | Deliver the signal to every thread in the process
  - | Deliver the signal to certain threads in the process
  - | Assign a specific thread to receive all signals for the process

# Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

# Thread Example

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4

void *hello (void *arg) { /* thread main */
    printf("Hello Thread\n");
    return 0;
}

int main (void) {
    int i;
    pthread_t tid[NUM_THREADS];
    for (i = 0; i < NUM_THREADS; i++) { /* create/fork threads */
        pthread_create(&tid[i], NULL, hello, NULL);
    }
    for (i = 0; i < NUM_THREADS; i++) { /* wait/join threads */
        pthread_join(tid[i], NULL);
    }
    return 0;
}
```

```

#include<pthread.h> #include <stdio.h>

/* Producer/consumer program illustrating conditional variables */ /* Size of shared buffer */ #define BUF_SIZE 3

int buffer[BUF_SIZE];                                     /* shared buffer */

int add=0;                                                 /* place to add next element */

int rem=0;                                                 /* place to remove next element */

int num=0;                                                 /* number elements in buffer */

pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;           /* mutex lock for buffer */

pthread_cond_t c_cons=PTHREAD_COND_INITIALIZER; /* consumer waits on this cond var */

pthread_cond_t c_prod=PTHREAD_COND_INITIALIZER; /* producer waits on this cond var */

void *producer(void *param);

void *consumer(void *param);

main (int argc, char *argv[])
{
    pthread_t tid1, tid2;                                /* thread identifiers */

    int i; /* create the threads; may be any number, in general */

    if (pthread_create(&tid1,NULL,producer,NULL) != 0) {
        fprintf (stderr, "Unable to create producer thread\n"); exit (1);
    }

    if (pthread_create(&tid2,NULL,consumer,NULL) != 0) {
        fprintf (stderr, "Unable to create consumer thread\n"); exit (1);
    }

    /* wait for created thread to exit */

    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    printf ("Parent quiting\n");
}

```

```
/* Produce value(s) */
void *producer(void *param)
{
    int i;
    for (i=1; i<=20; i++) {
        /* Insert into buffer */
        pthread_mutex_lock (&m);
        if (num > BUF_SIZE) exit(1);      /* overflow */
        while (num == BUF_SIZE)           /* block if buffer is full */
            pthread_cond_wait (&c_prod, &m);
        /* if executing here, buffer not full so add element */
        buffer[add] = i;
        add = (add+1) % BUF_SIZE;
        num++;
        pthread_mutex_unlock (&m);
        pthread_cond_signal (&c_cons);
        printf ("producer: inserted %d\n", i); fflush (stdout);
    }
    printf ("producer quiting\n");
}
```

```
/* Consume value(s); Note the consumer never terminates */

void *consumer(void *param)

{
    int i;
    while (1) {
        pthread_mutex_lock (&m);
        if (num < 0) exit(1); /* underflow */
        while (num == 0)          /* block if buffer empty */
            pthread_cond_wait (&c_cons, &m);
        /* if executing here, buffer not empty so remove element */
        i = buffer[rem];
        rem = (rem+1) % BUF_SIZE;
        num--;
        pthread_mutex_unlock (&m);
        pthread_cond_signal (&c_prod);
        printf ("Consume value %d\n", i); fflush(stdout);
    }
}
```

# Thank you



# SLIIT

*Discover Your Future*

# IT2060/IE2061

## Operating Systems and System Administration

### Lecture 05

### Introduction to CPU Scheduling

**U. U. Samantha Rajapaksha**

M.Sc.in IT, B.Sc.(Engineering) University of Moratuwa

Senior Lecturer SLIIT

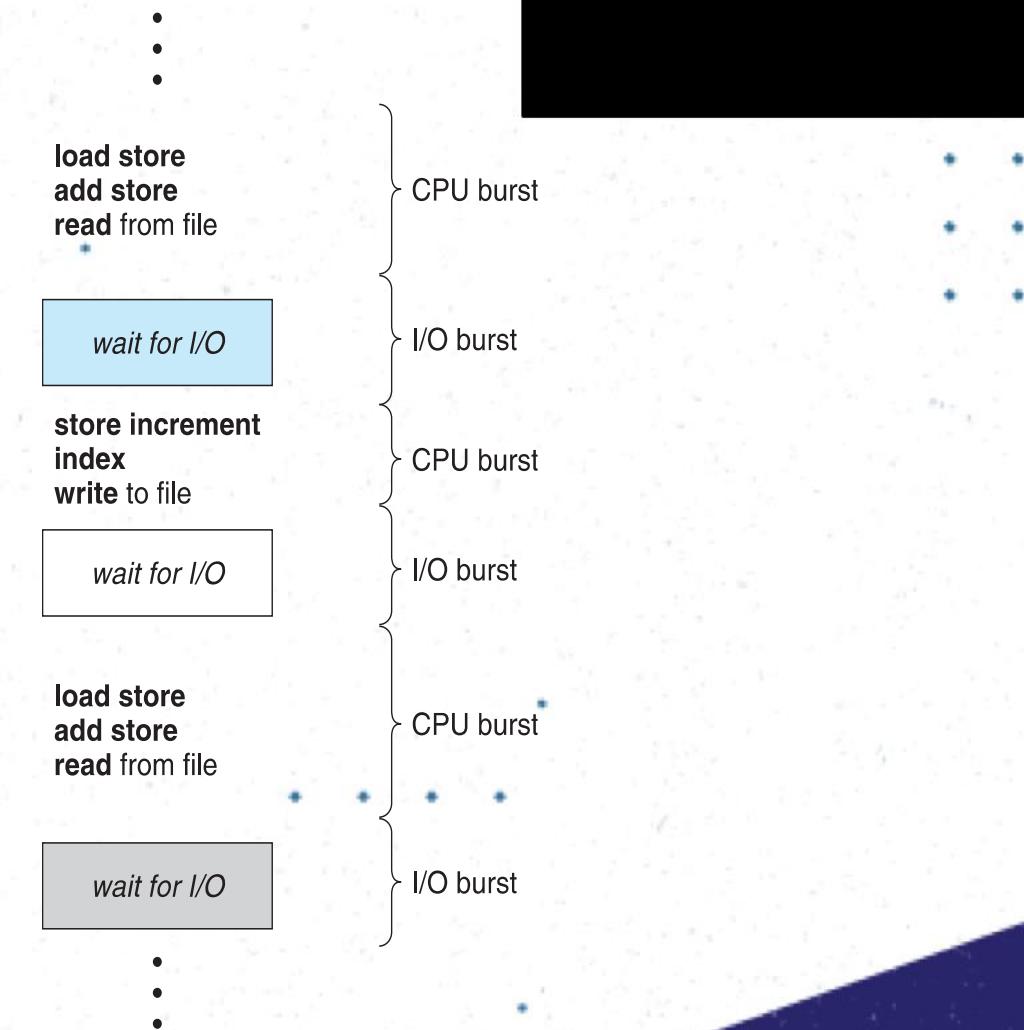
[Samantha.r@slit.lk](mailto:Samantha.r@slit.lk)

# Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Algorithm Evaluation

# Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
  - CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
  - **CPU burst** followed by **I/O burst**
  - CPU burst distribution is of main concern



# CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**

# Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

# Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

# Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

## First- Come, First-Served (FCFS) Scheduling

<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

- Suppose that the processes arrive in the order:  $P_1, P_2, P_3$
- The Gantt Chart for the schedule is:



- Waiting time for  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$
- Average waiting time:  $(0 + 24 + 27)/3 = 17$

# FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2, P_3, P_1$$

- The Gantt chart for the schedule is:



- Waiting time for  $P_1 = 6$ ;  $P_2 = 0$ ;  $P_3 = 3$
- Average waiting time:  $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

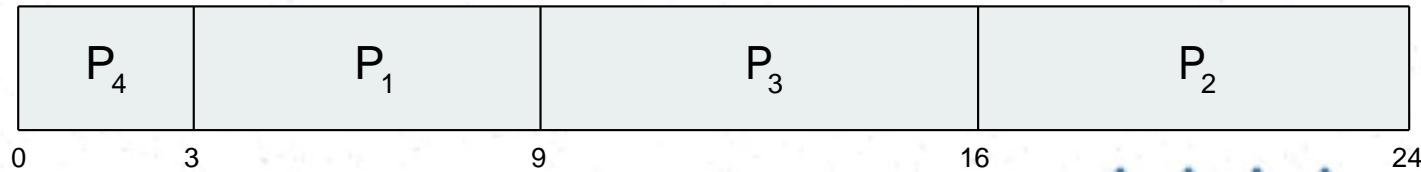
# Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user

# Example of SJF

<u>Process</u>	<u>Burst Time</u>
$P_1$	6
$P_2$	8
$P_3$	7
$P_4$	3

- SJF scheduling chart



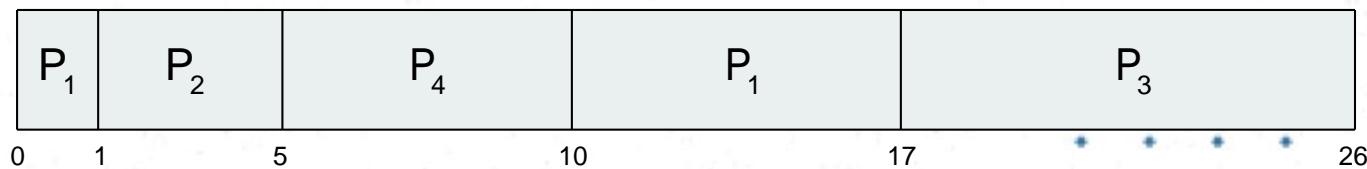
- Average waiting time =  $(3 + 16 + 9 + 0) / 4 = 7$

## Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
$P_1$	0	8
$P_2$	1	4
$P_3$	2	9
$P_4$	3	5

- Preemptive SJF Gantt Chart*



- Average waiting time =  $[(10-1)+(1-1)+(17-2)+5-3]/4 = 26/4 = 6.5$  msec

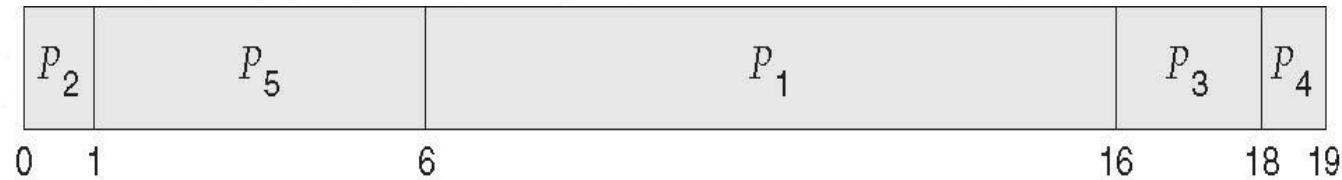
# Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer  $\equiv$  highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem  $\equiv$  **Starvation** – low priority processes may never execute
- Solution  $\equiv$  **Aging** – as time progresses increase the priority of the process

# Example of Priority Scheduling

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	10	3
$P_2$	1	1
$P_3$	2	4
$P_4$	1	5
$P_5$	5	2

- Priority scheduling Gantt Chart



- Average waiting time = 8.2 msec

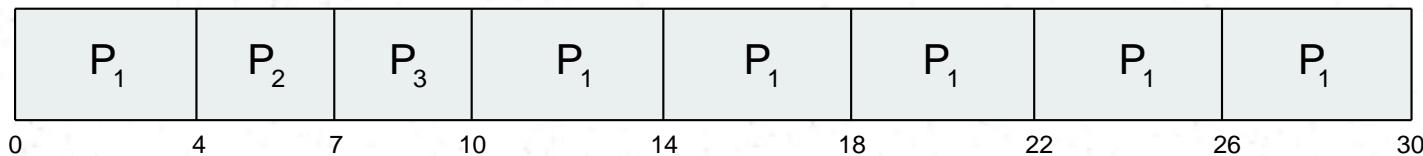
# Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum**  $q$ ), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are  $n$  processes in the ready queue and the time quantum is  $q$ , then each process gets  $1/n$  of the CPU time in chunks of at most  $q$  time units at once. No process waits more than  $(n-1)q$  time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - $q$  large  $\Rightarrow$  FIFO
  - $q$  small  $\Rightarrow q$  must be large with respect to context switch, otherwise overhead is too high

# Example of RR with Time Quantum = 4

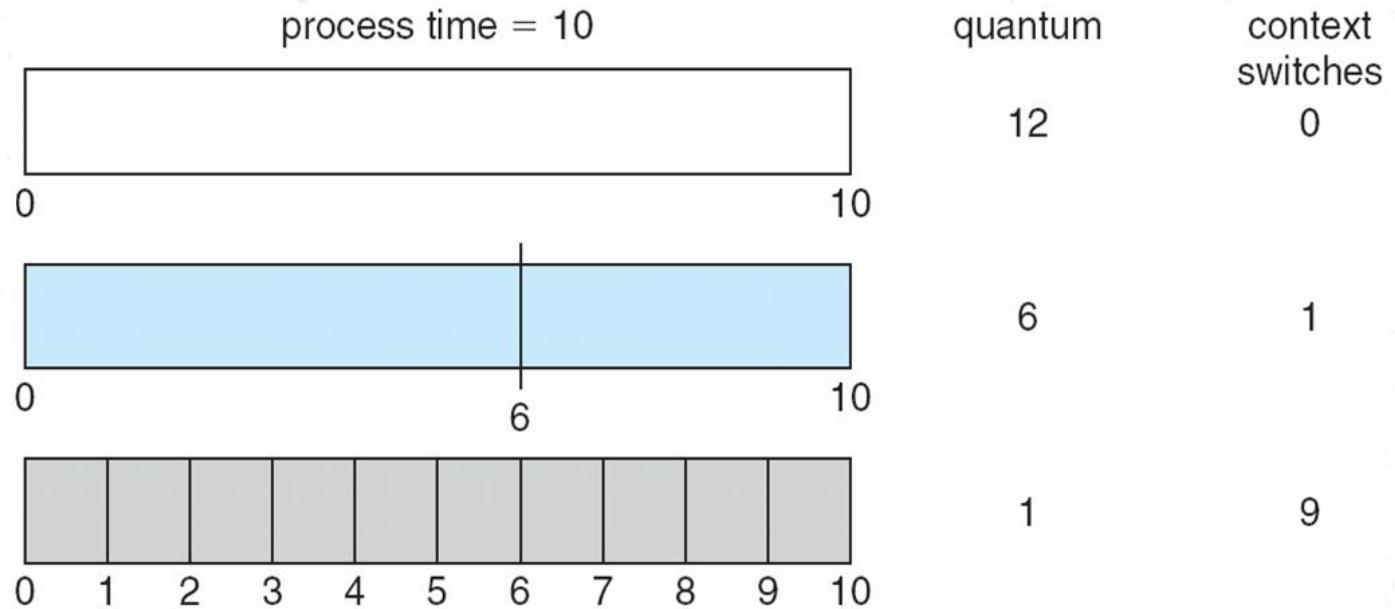
<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better **response**
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

# Time Quantum and Context Switch Time



# Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS

# Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

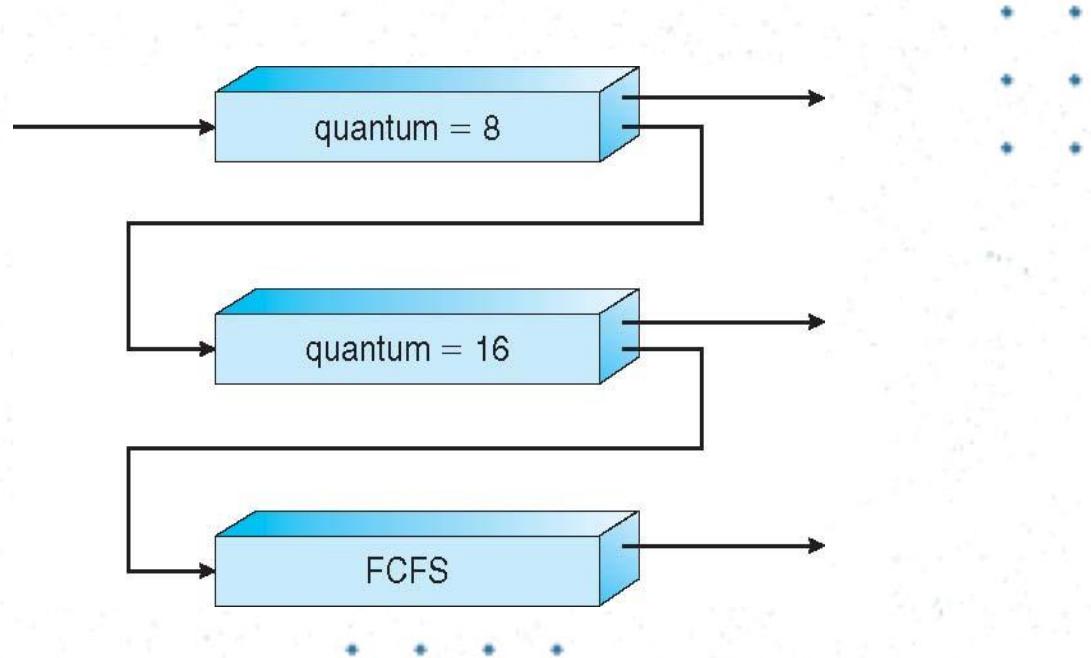
# Example of Multilevel Feedback Queue

- Three queues:

- $Q_0$  – RR with time quantum 8 milliseconds
- $Q_1$  – RR time quantum 16 milliseconds
- $Q_2$  – FCFS

- Scheduling

- A new job enters queue  $Q_0$  which is served FCFS
  - When it gains CPU, job receives 8 milliseconds
  - If it does not finish in 8 milliseconds, job is moved to queue  $Q_1$
- At  $Q_1$  job is again served FCFS and receives 16 additional milliseconds
  - If it still does not complete, it is preempted and moved to queue  $Q_2$



# Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

Process	Burst Time
$P_1$	10
$P_2$	29
$P_3$	3
$P_4$	7
$P_5$	12

# Deterministic Evaluation

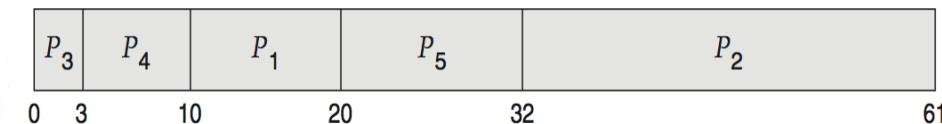
For each algorithm, calculate minimum average waiting time

Simple and fast, but requires exact numbers for input, applies only to those inputs

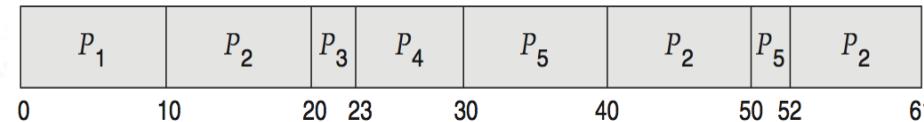
- FCS is 28ms:



- Non-preemptive SJF is 13ms:



- RR is 23ms:



# Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc

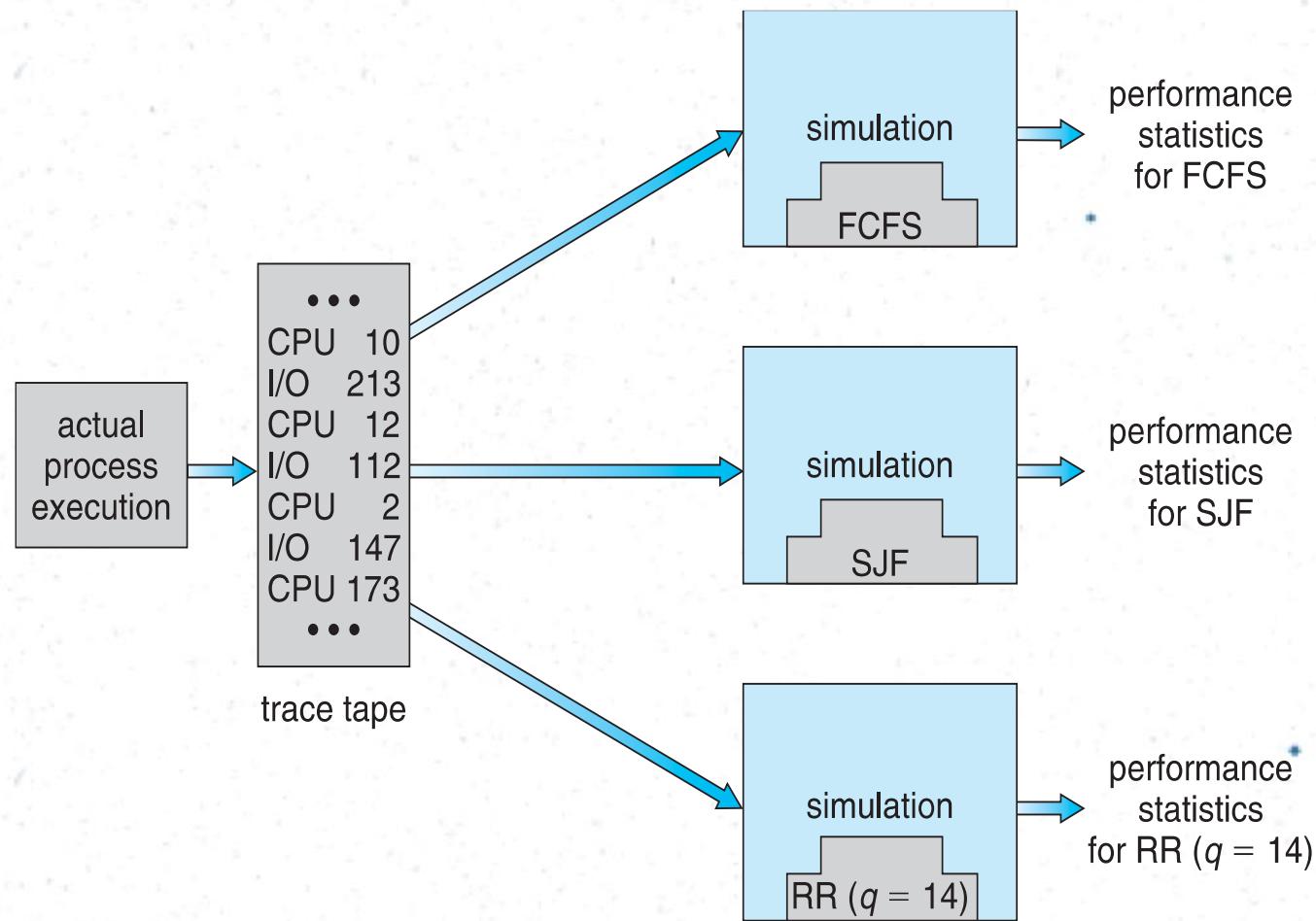
# Little's Formula

- $n$  = average queue length
- $W$  = average waiting time in queue
- $\lambda$  = average arrival rate into queue
- Little's law – in steady state, processes leaving queue must equal processes arriving, thus:  
$$n = \lambda \times W$$
- Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds

# Simulations

- Queueing models limited
- **Simulations** more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically or empirically
    - Trace tapes record sequences of real events in real systems

# Evaluation of CPU Schedulers by Simulation



# Implementation

- Even simulations have limited accuracy
  - Just implement new scheduler and test in real systems
    - High cost, high risk
    - Environments vary
  - Most flexible schedulers can be modified per-site or per-system
  - Or APIs to modify priorities
  - But again environments vary



# SLIIT

*Discover Your Future*

# IT2060/IE2061

## Operating Systems and System Administration

### Lecture 06

### Introduction to Deadlock

**U. U. Samantha Rajapaksha**

M.Sc.in IT, B.Sc.(Engineering) University of Moratuwa

Senior Lecturer SLIIT

[Samantha.r@slit.lk](mailto:Samantha.r@slit.lk)

# DEADLOCKS

- Several processes may compete for a finite number of resources, and some of them may wait for the resources forever because the resources are held by other waiting processes → deadlock.
- A set of processes is in a deadlock state if every process in the set is waiting for an event that can be caused only by another process in the set.
- • •

• •  
• •  
• •

## Example

- System has two tape drives.
- P1 and P2 each hold one tape drive and each needs another one.

## Example

- Semaphores A and B, initialized to 1.

<u>P0</u>	<u>P1</u>
<i>wait (A)</i>	<i>wait (B)</i>
<i>wait (B)</i>	<i>wait (A)</i>

• • •

# System Model

- Resources are partitioned into several types, each consists of some number of identical *instances*.
  - **Identical**: allocation of *any* instance of the type will satisfy process's request.
  - Resources may be physical resources (printers, tape drives, CPU cycles), or logical resources (files, semaphores, and monitors).
  - A **pre-emptible** resource is one that can be taken away from a process with no ill effect to the process; e.g., memory.
  - A **non-preemptible** resource is one that cannot be taken away from its user since it will make the user fails; e.g., printers
    - In general, potential deadlocks involve this resource type.
- Each process uses a resource as follows:
  - **Request** the resource; a process must wait if the resource is being used by another process.
  - **Use** the resource; e.g., the process can print on the printer.
  - **Release** the resource.

## Necessary conditions for deadlock

Four conditions must hold for a deadlock to occur (Coffman et al.):

1. **Mutual exclusion condition.** Only one process at a time can use the resource.
  - or each resource is either currently assigned to exactly one process or is available.
2. **Hold and wait condition.** A process holding at least one resource is waiting to acquire additional resources held by other processes.
3. **No pre-emption condition.** A resource can be released only voluntarily by the process holding it after that process has completed its task.
4. **Circular wait condition.** There exists a set  $\{P_0, P_1, \dots, P_n\}$  of waiting processes such that  $P_0$  is waiting for a resource that is held by  $P_1$ ,  $P_1$  is waiting for a resource that is held by  $P_2$ , ...,  $P_{n-1}$  is waiting for a resource that is held by  $P_n$ , and  $P_n$  is waiting for a resource that is held by  $P_0$ .

**Note:** the four conditions are not completely independent, e.g., the circular-wait condition implies the hold-and-wait condition.

# Deadlock Modelling

- Deadlocks can be described more precisely in terms of a directed graph  $G(V, E)$ 
  - called *System resource-allocation graph*
- $V$  is partitioned into two types:
  - Set of processes in the system:  $P=\{P_1, P_2, \dots, P_n\}$ .
  - Set of all resource types in the system:  $R=\{R_1, R_2, \dots, R_n\}$
- *Request edge* – directed edge  $P_i \rightarrow R_j$ 
  - process  $P_i$  requests an instance of resource  $R_j$
- *Assignment edge* – directed edge  $R_j \rightarrow P_i$ 
  - an instance of resource  $R_j$  has been allocated to process  $P_i$

# Model Symbols

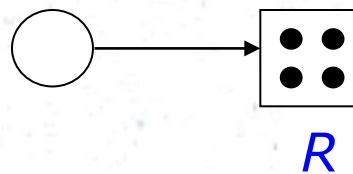
- Process:



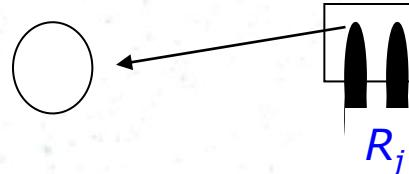
Resource type with 4 instances:



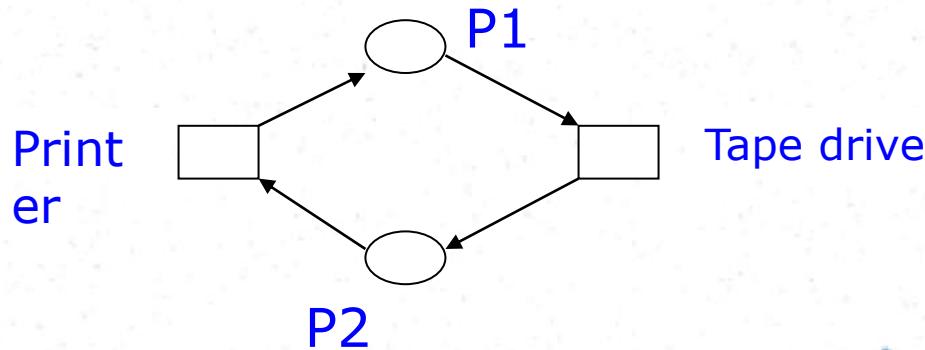
- $P_i$  requests  $R_j$ :



- $P_i$  uses  $R_j$ :

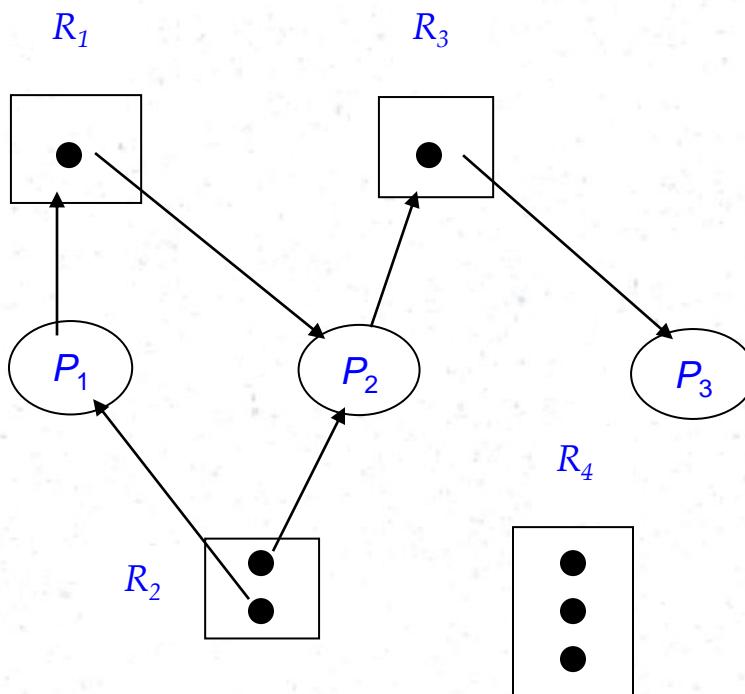


## Example: Deadlock



- If the graph contains **no cycles**, no process in the system is deadlocked.
- If the graph contains **a cycle**, deadlock *may* exist.
  - If each resource type has **one instance**, **cycle means deadlock**.
  - If each resource type has **several instances**, cycle is necessary but **not sufficient condition for deadlock**.

## Example: resource allocation graph (with no cycles)



### The sets $P$ , $R$ , and $E$ :

$$P = \{P_1, P_2, P_3\}$$

$$R = \{R_1, R_2, R_3, R_4\}$$

$$E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, \\ R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$$

### Resource instances:

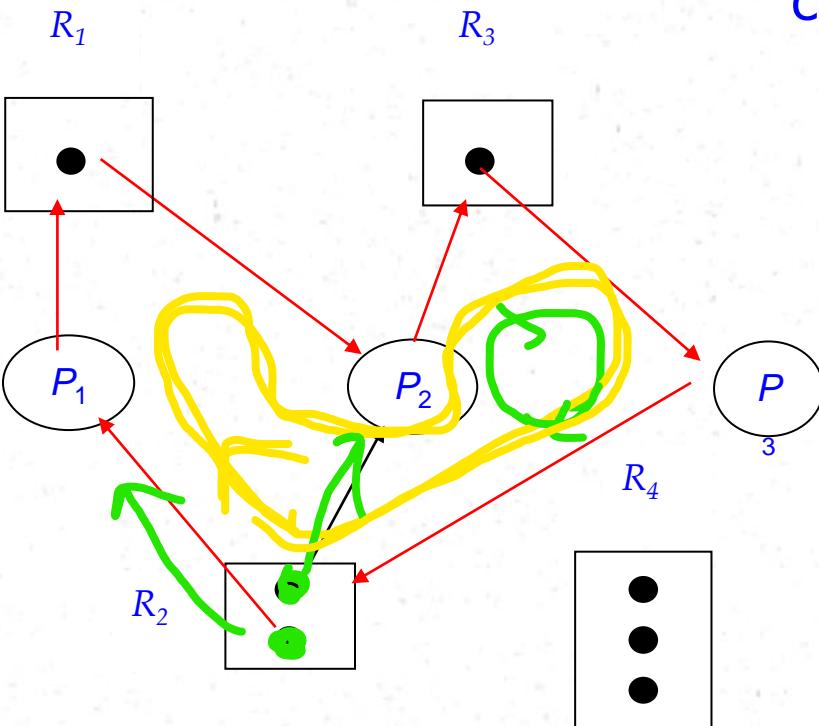
- One instance of resource type  $R_1$
- Two instances of resource type  $R_2$
- One instance of resource type  $R_3$
- Three instances of resource type  $R_4$

### Process states:

- $P_1$  is holding an instance of  $R_2$ , and waiting for an instance of  $R_1$
- $P_2$  is holding an instance of  $R_1$  and  $R_2$ , and is waiting for an instance of  $R_3$
- $P_3$  is holding an instance of  $R_3$

# Example

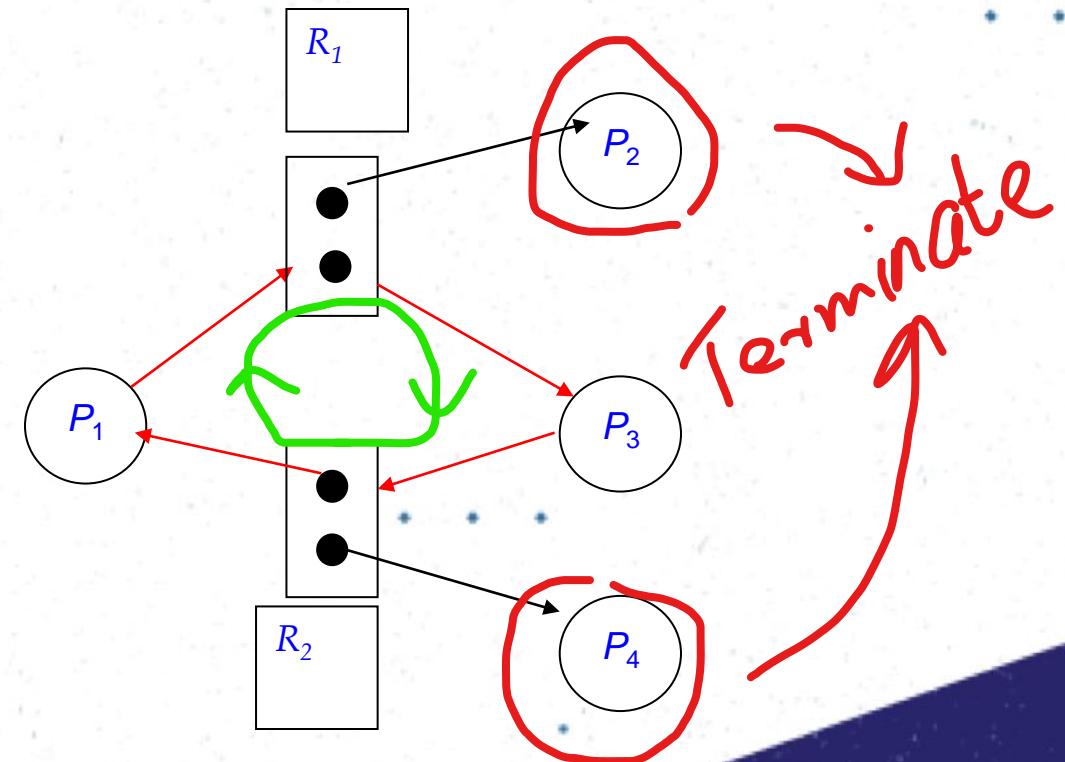
## A cycle and deadlock



Deadlock

## A cycle but no deadlock

$P_4$  can release  $R_2$  which gets allocated to  $P_3$ ; breaking the cycle



No-Deadlock

## Three Methods for handling deadlock

- Use a protocol to ensure that the system will *never* reaches deadlock
  - Using *deadlock prevention* and/or *deadlock avoidance* techniques
- Allow the system to enter a deadlock state and then recover
  - needs *deadlock detection* and *deadlock recovery* algorithms
- Ignore the problem and pretend that deadlocks never occur in the system
  - used by most OS's, including UNIX
  - Also called the **ostrich** algorithm!

# Deadlock prevention

- Restrain the ways resource requests can be made
  - Use a set of methods to ensure that **any one** of the four deadlock conditions cannot hold

## (1) Deny mutual exclusion

- Not required for sharable resources (e.g., read-only files, cannot be in deadlock)
- Must hold for non-sharable resources (a printer cannot be simultaneously shared by several processes)
- **In general**, it is not possible to prevent deadlock by denying mutual-exclusion condition since some resources are non-sharable

## (2) Deny hold and wait

- Must guarantee that whenever a process requests a resource, it does not hold any other resources

### Options:

- Each process is granted all resources before it starts
- Allows a process to request resources only when it has none
  - If a process needs more resources, release all resources before requesting new ones

### Problem:

- Resource utilisation is low
- Possible starvation.
  - A process that needs popular resources may have to wait indefinitely

## Deadlock prevention (cont.)

### (3) Prevent no pre-emption (i.e., allow pre-emption)

- When a process holding some resources requests other resource that cannot be immediately allocated, it must release all resources currently being held
- The pre-empted resources are added to the process's list of requested resources
- The process is restarted when it regains its old resources and obtains the new one it is requesting

#### Problem:

- Can be applied easily to resources whose state can be saved easily (e.g., memory), but not so easily for others (e.g., printer)

# Deadlock prevention (cont.)

#### (4) Deny circular wait

- All resource types are ordered, e.g.,
    - $F(\text{card reader}) = 1$                            $F(\text{disk drive}) = 5$
    - $F(\text{tape drive}) = 7$                            $F(\text{printer}) = 12$
  - Each process must request increasing order of resources
  - Protocol:
    - Each process requests resources in increasing order
    - Initially a process can request for any  $R_i$
    - After that, it can request  $R_j$  only if  $F(R_j) > F(R_i)$
  - **Problem:** It may be impossible to find a resource ordering that satisfies everyone

## Deadlock avoidance

- The system must have some additional *a priori* information about which resources a process will request and use during its lifetime
  - With the additional information, the system can decide for each request whether or not the process should wait
  - The simplest and most useful model requires that each process declare the *maximum* number of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- A resource-allocation *state* is defined by:
  - The number of available and allocated resources, and
  - The maximum demands of the processes

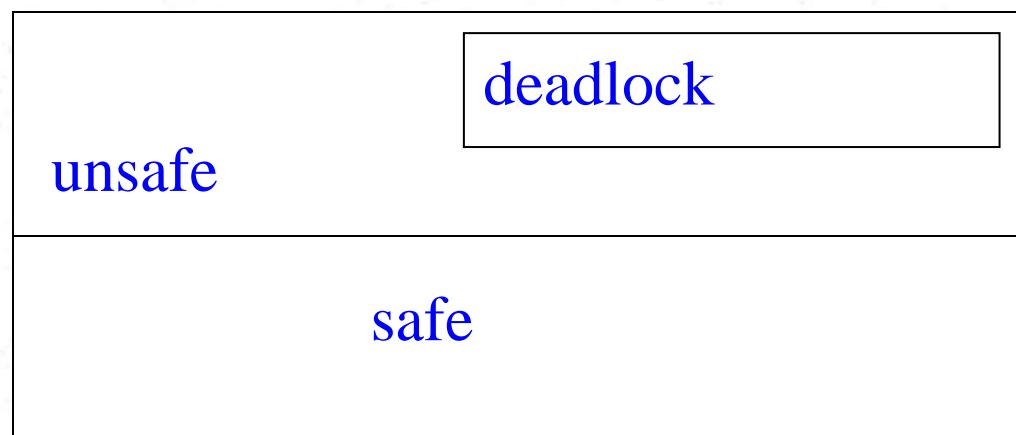
## Safe State

- When a process requests an available resource, the system checks if its allocation keeps the system in. safe state
- The system is in *safe state* if there exists a safe sequence of all processes
- A sequence  $\langle P_1, P_2, \dots, P_n \rangle$  is *safe* if, for each  $P_i$ , the resources requested by  $P_i$  can be allocated from the currently available resources + resources held by all  $P_j$ , with  $j < i$ 
  - If  $P_i$ 's resource needs are not immediately available,  $P_i$  waits until all  $P_j$  have finished
  - When all  $P_j$  are finished,  $P_i$  obtains the needed resources, executes, returns the allocated resources, and terminates
  - When  $P_i$  terminates,  $P_{i+1}$  can obtain its needed resources, and so on

## Safe State (cont.)

### Basic facts

- If a system is in safe state → no deadlocks
- If a system is in unsafe state → possibility of deadlock
- Avoidance ensures that the system never enters an unsafe state
- A process requesting for a currently available resource may have to wait
  - Thus, resource allocation is lower than without deadlock avoidance algorithm



## Example

Consider a system with 12 resources of the same type, and 3 processes with the following resource needs and allocation

Single Instance

	<u>Maximum needs</u>	<u>Allocation</u>	<u>Current need</u> (නොකළ)	
$P_0$	10	5	5	5
$P_1$	4	2	2	$3 \rightarrow 1$ unused resource
$P_2$	9	2	7	10
		Allocate = <del>2/9</del> → 3		(12)

- At time  $t_0$ , **available resource = 3**, and the system is in safe state
  - There is a safe sequence  $\langle P_1, P_0, P_2 \rangle$
- What if at  $t_1$  one more resource is allocated to process  $P_2$ ?
  - The system is in unsafe state
    - Deadlock can occur

$\langle P_1, P_0, P_2 \rangle$

# Resource-Allocation Graph Algorithm

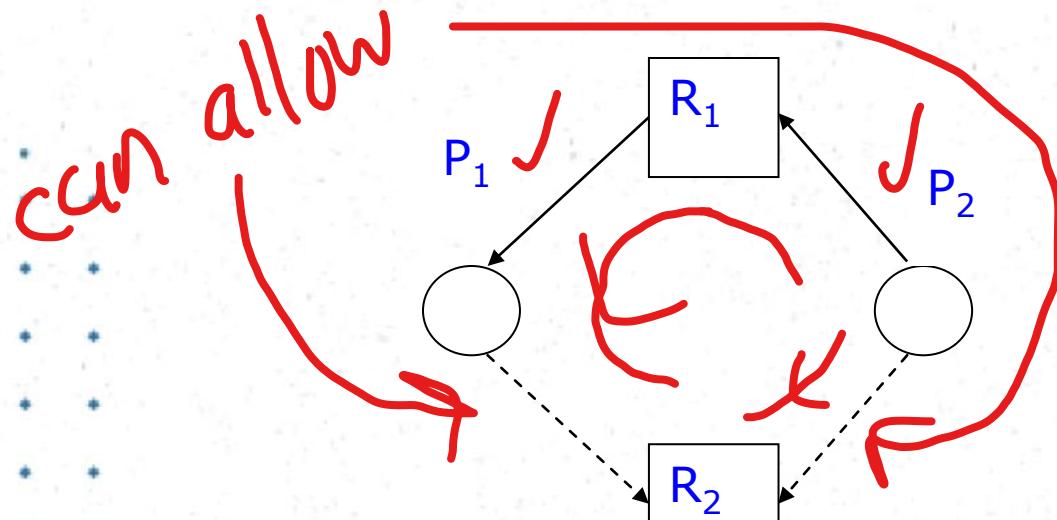
- Claim edge  $P_i \rightarrow R_j$  indicates that process  $P_i$  may request resource  $R_j$ 
  - represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- When a resource is released by a process, assignment edge converts to a claim edge
- Resources must be claimed a priori in the system
- Need a cycle detection algorithm  $\rightarrow O(n^2)$
- This algorithm can not be used for system comprising resource types with multiple instances

↑ vertile  
↓ resour  
complexity  
↑ sys ↓

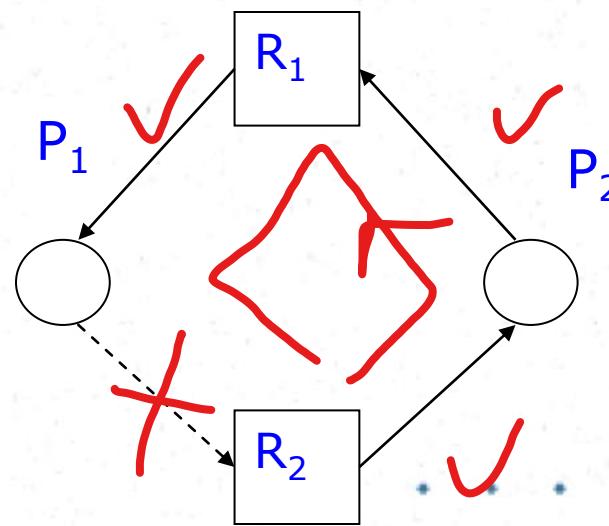
## Example

Suppose  $P_2$  requests  $R_2$

Although  $R_2$  is currently free, allocating it to  $P_2$  may lead to unsafe state (a cycle in right figure)



NO cycle  
NO deadlock



# Banker's Algorithm

- The algorithm for a system comprising resource types with **multiple instances**
- Similar to a bank: never allocates its available cash if it can no longer satisfy the needs of all customers
- Each process must *a priori* claim maximum number of instances of each resource type that it may need
- When a process requests a resource:
  - It may have to wait (if resource allocation may lead to unsafe state) until some other process releases enough resources
- When a process gets all its resources:
  - It must return them in a finite amount of time

# Banker's Algorithm (cont.)

## Algorithm

Let  $n$  = number of processes, and  $m$  = number of resource types

### **Data structures:**

- *Available*: Vector of length  $m$ 
  - $available[j] = k$ ; means  $k$  instances of resource type  $R_j$  are available
- *Max*:  $n \times m$  matrix
  - $Max[i, j] = k$ ; means process  $P_i$  may request at most  $k$  instances of resource type  $R_j$ .
- *Allocation*:  $n \times m$  matrix
  - $Allocation[i, j] = k$ ; means process  $P_i$  is currently allocated  $k$  instances of resource type  $R_j$
- *Need*:  $n \times m$  matrix
  - $Need[i, j] = k$ ; means process  $P_i$  may need  $k$  more instances of resource type  $R_j$  to complete its task.
  - $Need[i, j] = max[i, j] - allocation [i, j]$

# Implementation of the safety algorithm

// Time complexity =  $O(mn^2)$

1. Let  $work$  and  $finish$  be vectors of length  $m$  and  $n$ , respectively

initialise:

$work = available$

$finish[i] = false$       for  $i = 1, 2, \dots, n$

// Find an unfinished process  $i$ ; it still needs resources

2. Find a value of  $i$  such that both:

- $finish[i] = false$ , and

- $need_i \leq work$

- If no such  $i$  exists, go to step 4

// process  $i$  pretends to finish, so it releases its resources i.e.,  $allocation_i$

3.  $work = work + allocation_i$

$finish[i] = true$

go to step 2

4. If  $finish[i] = true$  for all  $i$ , the system is in safe state.

## Resource-request algorithm for process $P_i$

$Request_i$  = request vector for process  $P_i$

If  $Request_i[j] = k$ , then process  $P_i$  wants  $k$  instances of resource type  $R_j$

1. If  $request_i \leq need_i$ , go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If  $request_i \leq available$ , go to step 3. Otherwise  $P_i$  must wait, since resources are not available.
3. The system pretends to allocate requested resources to  $P_i$  by modifying the state as follows:

$$available = available - request_i$$

$$allocation_i = allocation_i + request_i$$

$$need_i = need_i - request_i$$

- If resulting state is safe, resources are allocated to  $P_i$
- else  $P_i$  must wait, and the old resource-allocation state is restored.

## Example of Banker's algorithm

Multiple  
Instance

- 5 processes  $P_0$  through  $P_4$ ; 3 resource types A (10 instances), B (5 instances), and C (7 instances)
- Snapshot at time  $T_0$ :

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	7	5	3	3	3	2	7	4	3
$P_1$	2	0	0	3	2	2				1	2	2
$P_2$	3	0	2	9	0	2				6	0	0
$P_3$	2	1	1	2	2	2				0	1	1
$P_4$	0	0	2	4	3	3				4	3	1

7 2 5

- The content of matrix *Need* is defined to be *Max – Allocation*
- The system is in a safe state since the sequence  $\langle P_1, P_3, P_4, P_2, P_0 \rangle$  satisfies the safety criteria

## Example ( $P_1$ requests (1,0,2)):

- Check that  $request \leq need$  (that is,  $(1, 0, 2) \leq (1, 2, 2)$ )  $\rightarrow$  true
- Check that  $request \leq available$  (that is,  $(1, 0, 2) \leq (3, 3, 2)$ )  $\rightarrow$  true

**Before  
Adjustment**

	Allocation		
	A	B	C
$P_0$	0	1	0
$P_1$	2	0	0
$P_2$	3	0	2
$P_3$	2	1	1
$P_4$	0	0	2

**After  
Adjustment**

	Alloc. .			Need			Avail.		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	7	4	3	2	3	0
$P_1$	3	0	2	0	2	0			
$P_2$	3	0	2	6	0	0			
$P_3$	2	1	1	0	1	1			
$P_4$	0	0	2	4	3	1	.	.	.

- \*  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$  or  $\langle P_1, P_4, P_3, P_0, P_2 \rangle$  satisfies safety requirement
- \* Can request for (3, 3, 0) by  $P_4$  be granted? (0, 2, 0) by  $P_0$ ?

## Deadlock detection

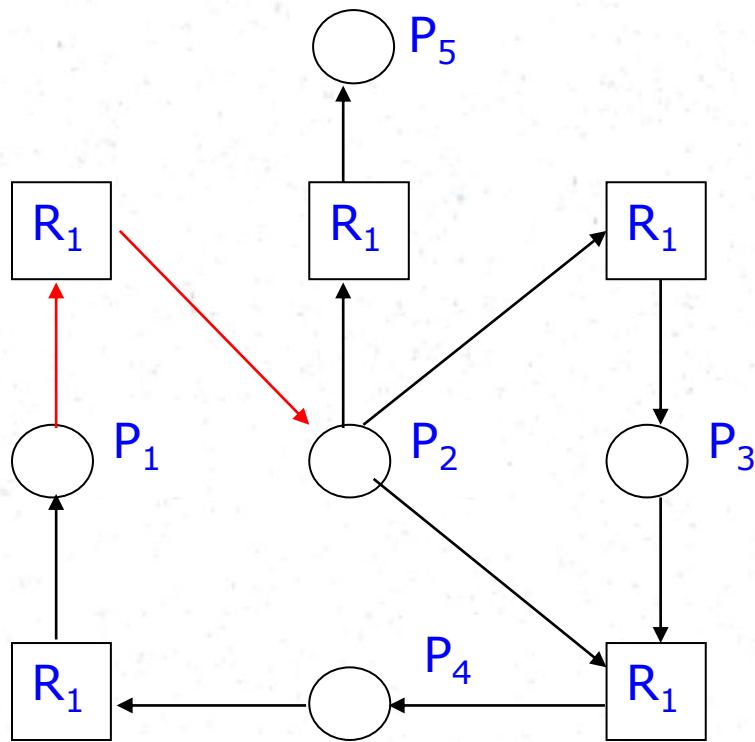
- If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may occur
  - Need a *deadlock detection* algorithm that examines the state of the system to determine whether a deadlock has occurred
  - Need a *recovery* algorithm to recover from deadlock

## Deadlock detection for single instance of each resource type

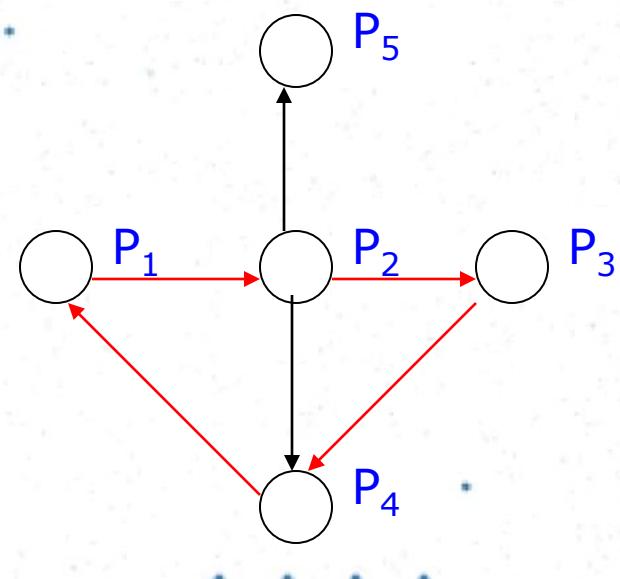
- Maintain a *wait-for graph*
    - Nodes are processes
    - $P_i \rightarrow P_j$  if  $P_i$  is waiting for  $P_j$
  - Periodically invoke an algorithm that searches for a cycle in the graph
    - An algorithm to detect a cycle in a graph requires  $O(n^2)$  operations,
      - $n$  is the number of vertices in the graph

# Example

Resource Allocation Graph



Wait for  
Graph



### 1) Terminate processes

- Kill (abort) all deadlocked processes
- Kill one process at a time until deadlock cycle eliminated
- In which order should we choose process to abort?
  - The process with lowest priority
  - How long the process has computed, and how much longer to completion
  - Resources the process has used
  - Resources the process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?

**Problem:** what if the process is in the middle of updating a file?

Aborting the process may lead to incorrect file

# Deadlock recovery

## 2) Pre-empt a resource from a process.

- How to select a victim (process) to minimize cost?
- Roll back the process to some safe state and restart from there
  - How do we find a safe state?
    - Easiest way: destroy the process and restart
    - Use checkpoints during execution
  - Starvation – same process may always be picked as victim
    - How do we ensure no starvation?
    - Include number of rollbacks in cost factor



# SLIIT

*Discover Your Future*

## IT2060/IE2061

# Operating Systems and System Administration

### Lecture 09

## Mass Storage Scheduling

### U. U. Samantha Rajapaksha

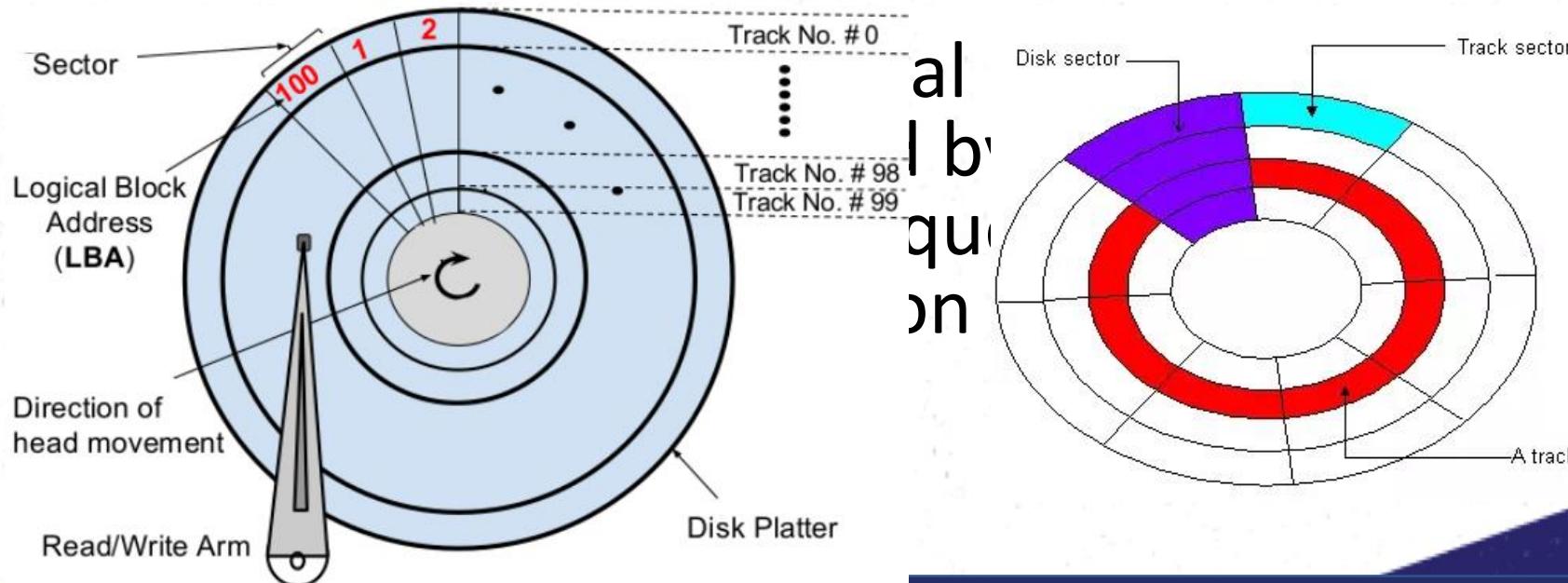
M.Sc.in IT, B.Sc.(Engineering) University of Moratuwa

Senior Lecturer SLIIT

[Samantha.r@slit.lk](mailto:Samantha.r@slit.lk)

# Disk Scheduling

- The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth
- Minimize seek time
- Seek time  $\approx$  seek distance



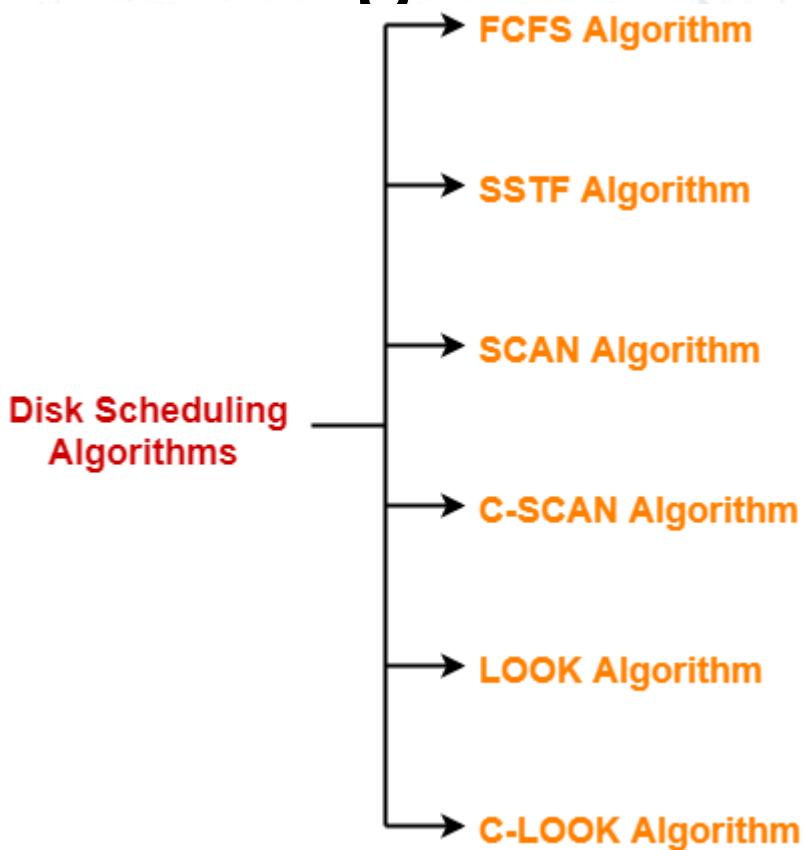
# Disk Scheduling (Cont.)

- Note that drive controllers have small buffers and can manage a queue of I/O requests (of varying “depth”)
- Several algorithms exist to schedule the servicing of disk I/O requests
- The analysis is true for one or many platters
- We illustrate scheduling algorithms with a request queue (0-199)

98, 183, 37, 122, 14, 124, 65, 67

Head pointer 53

# Disk Scheduling

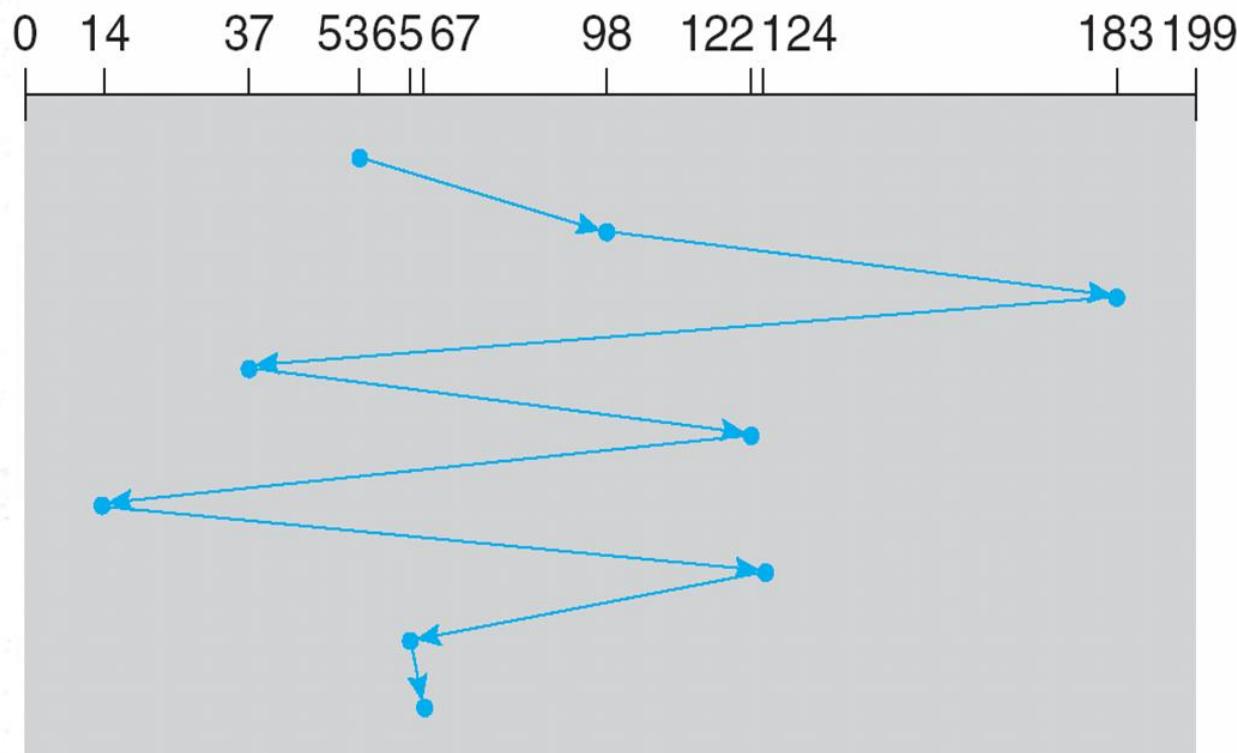


# FCFS

Illustration shows total head movement of 640 cylinders

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



# SSTF

- Shortest Seek Time First selects the request with the minimum seek time from the current head position

- SST

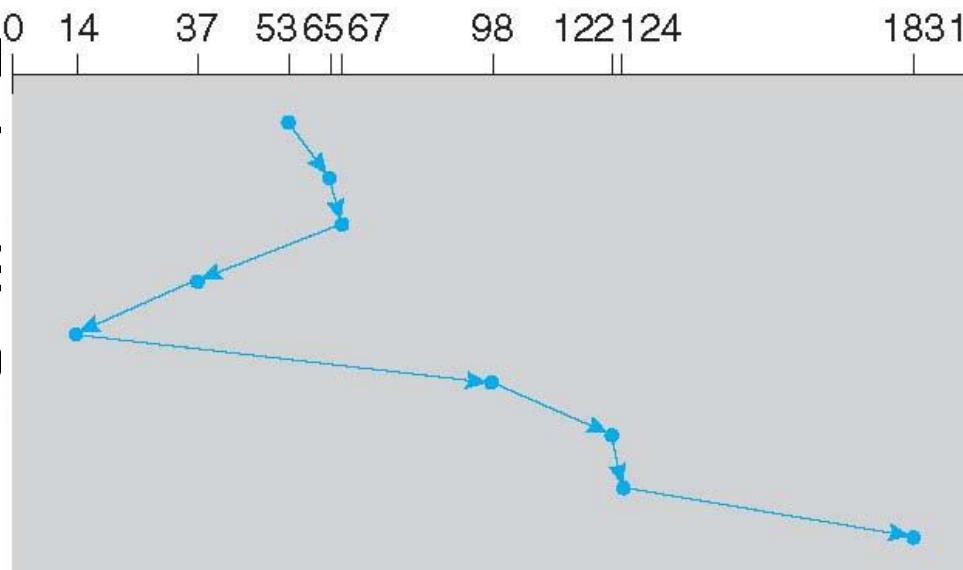
sch

son

- Illus

mov

queue = 98, 183, 37, 122, 14, 124, 65, 67  
head starts at 53



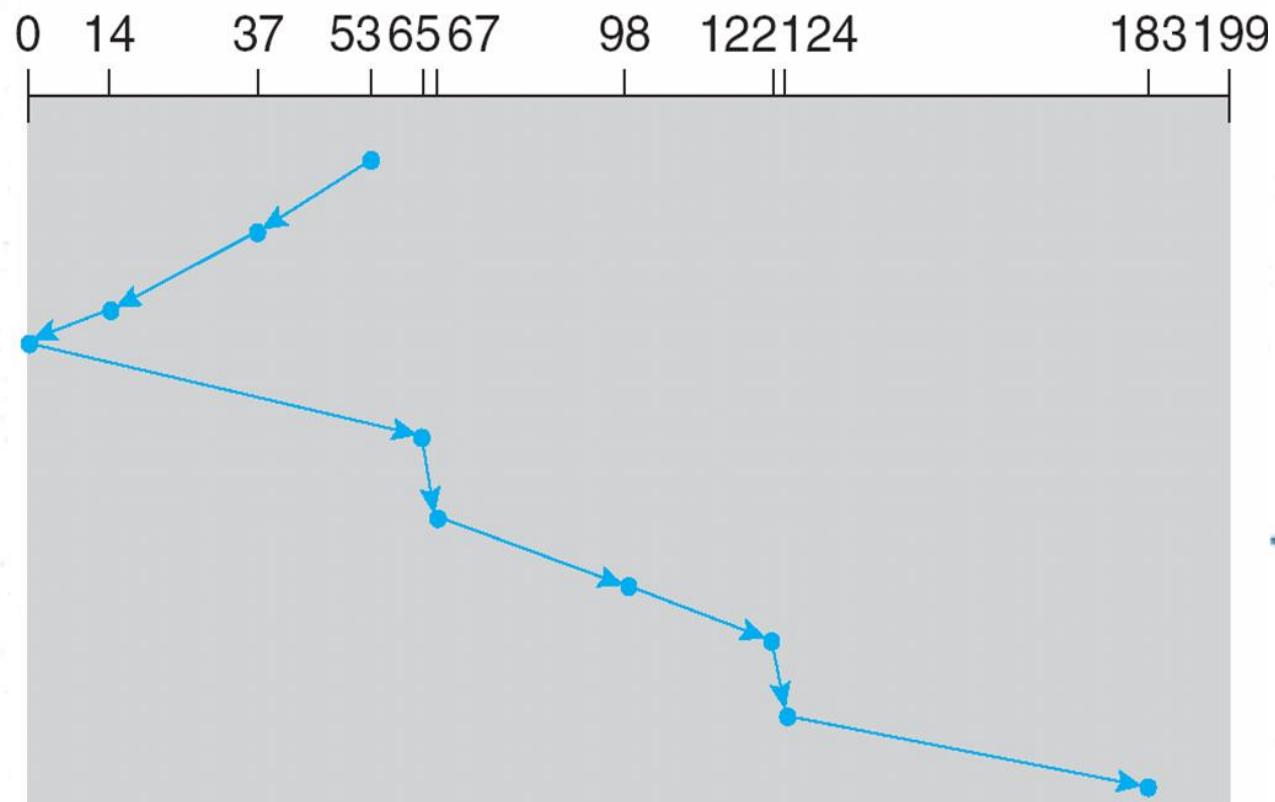
# SCAN

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- **SCAN algorithm** Sometimes called the **elevator algorithm**
- Illustration shows total head movement of 236 cylinders
- But note that if requests are uniformly dense, largest density at other end of disk and those wait the longest

# SCAN (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



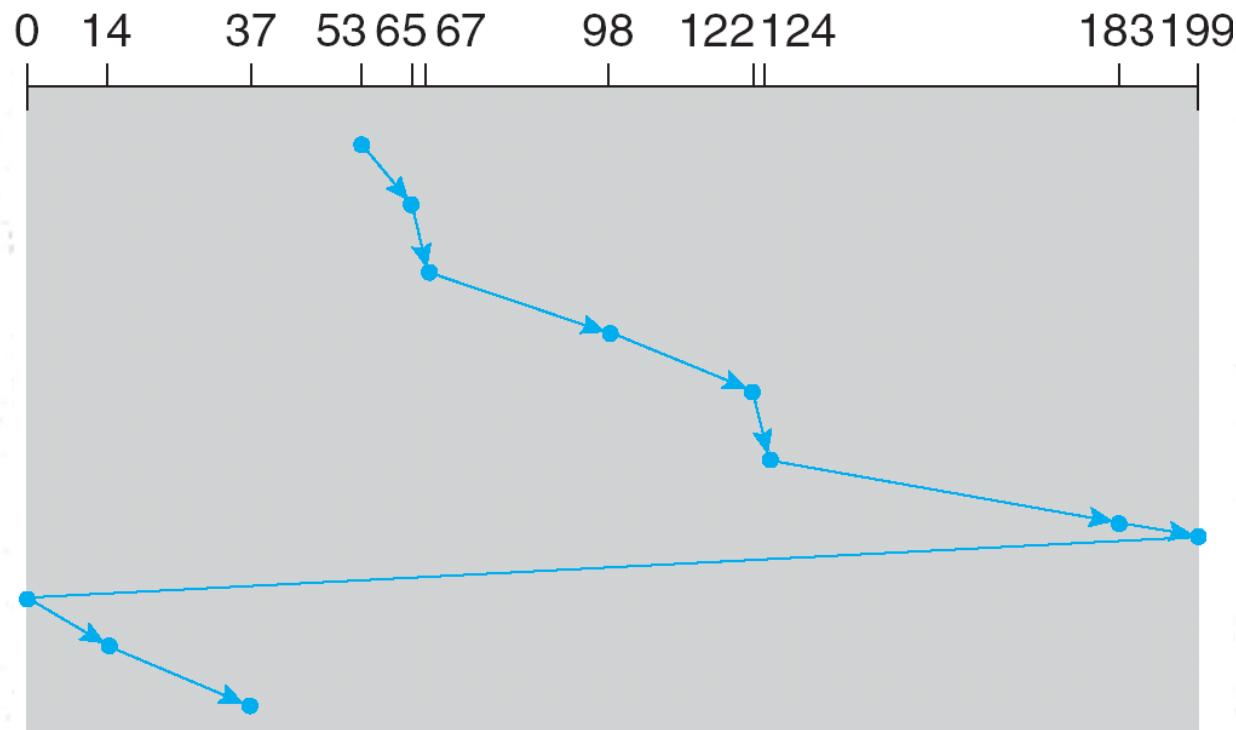
# C-SCAN

- Provides a more uniform wait time than SCAN
- The head moves from one end of the disk to the other, servicing requests as it goes
  - When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one
- Total number of cylinders?

# C-SCAN (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

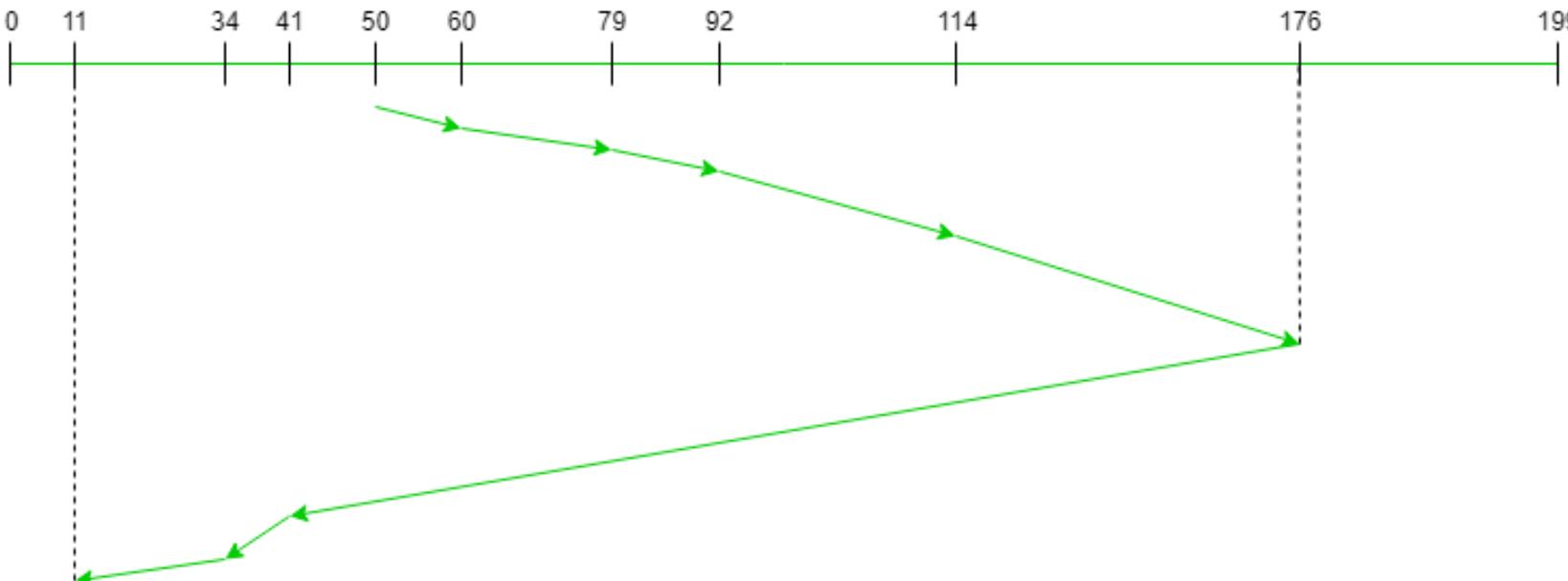


# LOOK Scheduling

Request sequence = {176, 79, 34, 60, 92, 11, 41, 114}

Initial head position = 50 Direction = right (We are moving from left to right)

The LOOK algorithm services request similarly as SCAN algorithm meanwhile it also "looks" ahead as if there are more tracks that are needed to be serviced in the same direction. If there are no pending requests in the moving direction the head reverses the direction and start servicing requests in the opposite direction.

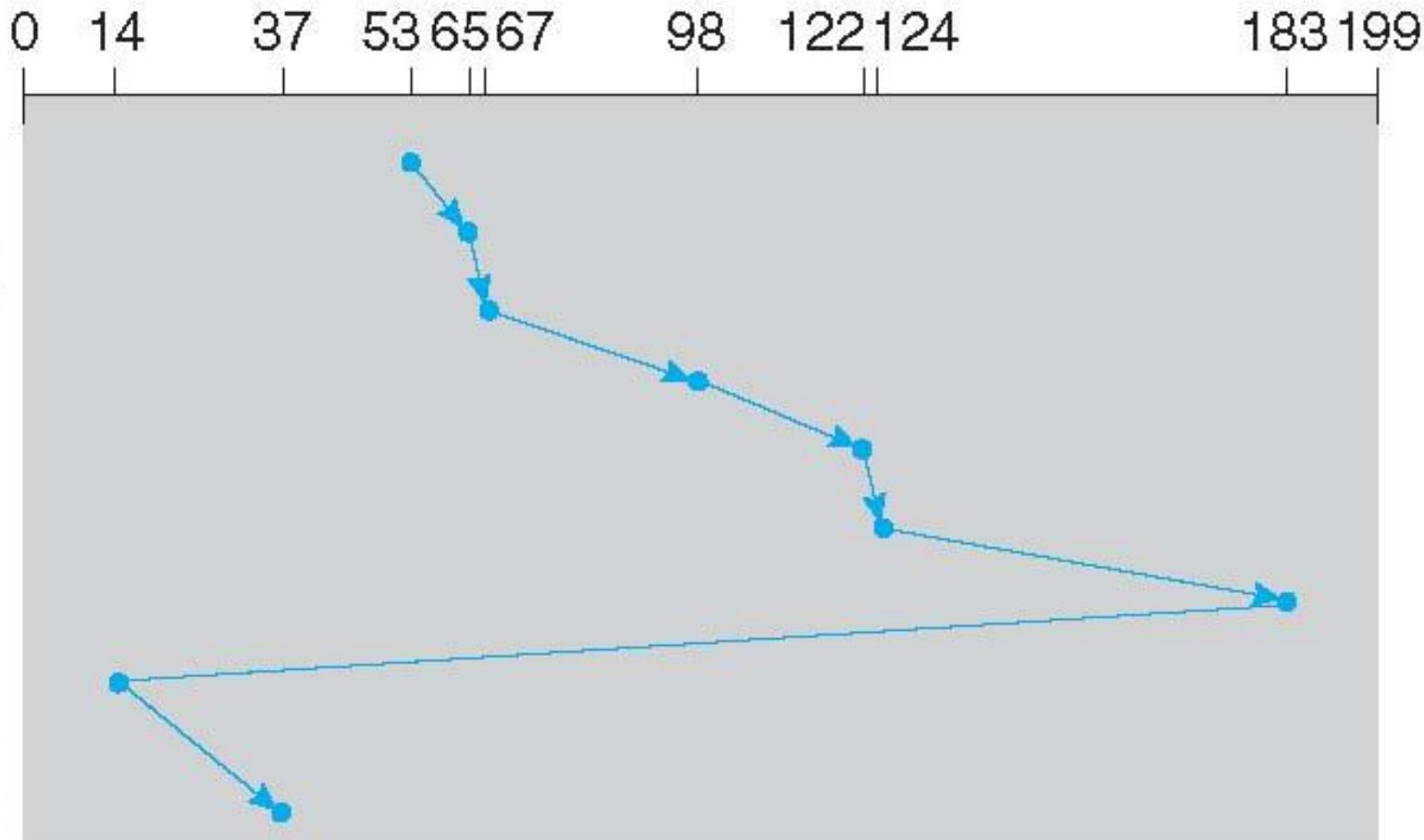


# C-LOOK

- LOOK a version of SCAN, C-LOOK a version of C-SCAN
- Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk
- Total number of cylinders?

# C-LOOK (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67  
head starts at 53





# SLIIT

*Discover Your Future*

# IT2060/IE2061

## Operating Systems and System Administration

### Lecture 07

### Process Syncronization

### U. U. Samantha Rajapaksha

M.Sc.in IT, B.Sc.(Engineering) University of Moratuwa

Senior Lecturer SLIIT

[Samantha.r@slit.lk](mailto:Samantha.r@slit.lk)

# Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Semaphores
- Classic Problems of Synchronization
- Monitors

# Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:  
Suppose that we wanted to provide a solution to the consumer-producer problem that fills *all* the buffers. We can do so by having an integer **counter** that keeps track of the number of full buffers. Initially, **counter** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

# Producer

```
while (true) {
    /* produce an item in next_produced */

    while (counter == BUFFER_SIZE) ;
        /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

# Consumer

```
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
    /* consume the item in next consumed */  
}
```

# Race Condition

- **counter++**

could be implemented as

```
register1 = counter  
register1 = register1 + 1  
counter = register1
```

⋮  
⋮  
⋮  
⋮  
⋮

- **counter--**

could be implemented as

```
register2 = counter  
register2 = register2 - 1  
counter = register2
```

⋮  
⋮  
⋮  
⋮  
⋮

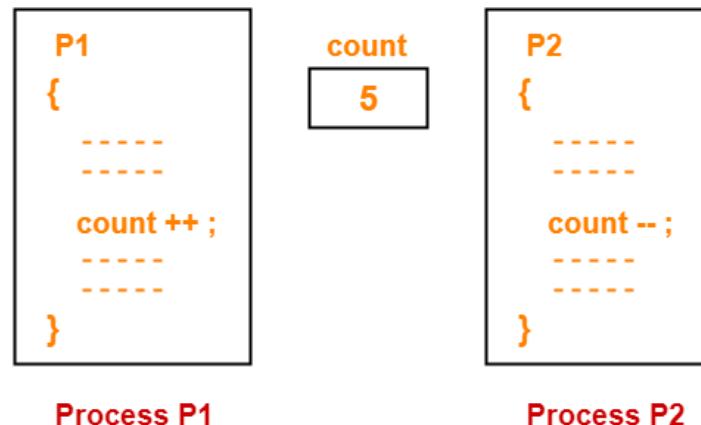
- Consider this execution interleaving with “count = 5” initially:

```
S0: producer execute register1 = counter  
S1: producer execute register1 = register1 + 1  
S2: consumer execute register2 = counter  
S3: consumer execute register2 = register2 - 1  
S4: producer execute counter = register1  
S5: consumer execute counter = register2
```

{register1 = 5}  
{register1 = 6}  
{register2 = 5}  
{register2 = 4}  
{counter = 6 }  
{counter= 4}

# Race condition

- *Race condition* is a situation where several processes access and manipulate the same data concurrently.
  - The outcome of the execution depends on particular order in which the access takes place.
- In order to prevent race condition on *counter*, we need to ensure that only one process at a time can be manipulating *counter*
  - We need some form of *process synchronization*.



# Critical Section Problem

- Consider system of  $n$  processes  $\{p_0, p_1, \dots, p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

# Critical Section

- General structure

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```

• • •  
• • •  
• • •  
• • •  
• • •  
• • •  
• • •  
• • •

# Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

## Solution (cont.)

### Simplest solution:

Each process *disables* all interrupts just *after entering* its critical section and *re-enables* them just *before leaving* it.

- Not wise, because enabling/disabling interrupt is a privileged instruction.

⋮  
⋮  
⋮  
⋮  
⋮

### Solution in kernel mode:

- **Preemptive kernel:** a process can be preempted while running in the kernel mode
  - Otherwise, the kernel is **nonpreemptive kernel:** allows the process to run until it exits kernel mode, blocks, or voluntarily yields CPU.
- Nonpreemptive kernel is free from race conditions on kernel data structure
  - However preemptive kernel is more responsive and suitable for real time system.

# Solution for two processes, $P_0$ and $P_1$

Software-based

## ALGORITHM 1

```
var turn: (0 .. 1); // Initially turn = 0; turn = i means  $P_i$  can enter its CS
```

Process  $P_i$

repeat

    while  $turn \neq i$  do no-op;

        Critical Section

$turn = j$ ;

        Remainder Section

    until false;

- Satisfies mutual exclusion, but not progress requirement.

- If  $turn = 0$ ,  $P_1$  cannot enter its CS even though  $P_0$  is in its RS.

- Taking turn is not good when one process is slower than other.

- *Busy waiting*: continuously testing a variable waiting for some value to appear
    - not good since it wastes CPU time

Process  $P_j$

repeat

    while  $turn \neq j$  do no-op;

        Critical Section

$turn = i$ ;

        Remainder Section

    until false;

# Solution for two processes (cont. )

Software-based

## ALGORITHM 2

// Initially  $flag[0] = flag[1] = false$ ;  $flag[i] = true$  means  $P_i$  wants to enter its CS  
**var**  $flag$ : array [0 .. 1] of boolean;

Process  $P_i$

**repeat**

$flag[i] = true$ ;

**while**  $flag[j]$  **do no-op**;

Critical Section;

$flag[i] = false$ ;

Remainder Section;

**until**  $false$ ;

Process  $P_j$

**repeat**

$flag[j] = true$ ;

**while**  $flag[i]$  **do no-op**;

Critical Section;

$flag[j] = false$ ;

Remainder Section;

**until**  $false$ ;

.....

- Satisfy mutual exclusion, but violates the progress requirement:

$T_0$ :  $P_0$  sets  $flag[0] = true$ .

$T_1$ :  $P_1$  sets  $flag[1] = true$ .

→  $P_0$  and  $P_1$  are looping in their respective **while**.

# Solution for two processes (cont. )

Software-based

// Peterson's solution: Combine shared variables of Algorithms 1 and 2

Process  $P_i$

**repeat**

$flag[i] = true;$

$turn = j;$

**while** ( $flag[j]$  and  $turn = j$ ) **do no-op;**  
 $op;$

critical section

$flag[i] = false;$

remainder section

**until**  $false;$

- Solves the critical-section problem for two processes.

**It meets all the three requirements**

- Proof: need to show that:

- Mutual exclusion is preserved.
- The progress requirement is satisfied.
- The bounded waiting time requirement is met.

- For detailed proof, read the textbook.

Process  $P_j$

**repeat**

$flag[j] = true;$

$turn = i;$

**while** ( $flag[i]$  and  $turn = i$ ) **do no-**

critical section

$flag[j] = false;$

remainder section

**until**  $false;$

# Bakery Algorithm

- The solution to the critical section problem for  $n$  processes by Leslie Lamport
- Before entering its critical section, each process receives a number.
  - The holder of the smallest number enters the critical section.
  - If processes  $P_i$  and  $P_j$  receive the same number, if  $i < j$ , then  $P_i$  is served first; else  $P_j$  is served first.
- Notation (**ticket#**, process **id#**)
  - $(a, b) < (c, d)$  if  $a < c$  or if  $a = c$  and  $b < d$ .
  - $\max(a_0, \dots, a_{n-1})$  is a number  $k$  such that  $k \geq a_i$  for  $i = 0, \dots, n-1$ .
- shared data
  - **var choosing: array[0..n-1] of boolean;**
  - **number: array [0 .. n-1] of integer;**
- data structures are initialised to *false* and 0, respectively

# Bakery Algorithm (contd.)

Process  $P_i$

**repeat**

*choosing[i] = true;*

*number[i] = max (number[0], number[1], ..., number[n-1]) +1;*

*choosing[i] = false;*

**for  $j = 0$  to  $n-1$  do**

**begin**

**while  $choosing[j]$  do no-op;**

**while  $number[j] \neq 0$  and  $(number[j], j) < (number[i], i)$  do no-op;**

**end;**

critical section

*number[i] = 0;*

remainder section

**until  $false$**

# Synchronization Hardware

- There is no guarantee that the software-based solution will work correctly in all computer architectures.
- The simple solution to critical section problem: disable interrupt while a shared variable is being modified.
  - This solution is not feasible in multiprocessor. Why?
- Use special hardware instructions such as *Test-and-Set* and *Swap*.
  - *Test-and-set* or *Swap* is an atomic instruction: it can not be interrupted until it completes its execution

```
// Test and set the content of a  
// word atomically  
function Test-and-Set (var  
    boolean: target)  
begin  
    Test-and-Set = target;  
    target = true;  
end;
```

```
// Swapping instruction is  
// done atomically  
procedure Swap (var  
    boolean: a, b)  
var boolean: temp;  
begin  
    temp = a;  
    a = b;  
    b = temp;  
end;
```

# How to use them?

## Mutual Exclusion with *Test-and-Set*

*var boolean: lock; lock is a shared variable, initially set to false.*

**Repeat // Process  $P_i$**   
    **while Test-and-Set (lock) do no-op;**  
        Critical Section  
        *lock = false;*  
        Remainder Section  
**until false;**

**Repeat // Process  $P_j$**   
    **while Test-and-Set (lock) do no-op;**  
        Critical Section  
        *lock = false;*  
        Remainder Section  
**until false;**

**Mutual Exclusion with *Swap***  
**Repeat // Process  $P_i$**   
    *key = true;*  
    **repeat**  
        *Swap (lock, key);*  
    **until key = false;**  
        Critical section  
        *lock = false;*  
        Remainder section  
**until false;**

**Repeat // Process  $P_j$**   
    *key = true;*  
    **repeat**  
        *Swap (lock, key);*  
    **until key = false;**  
        Critical section  
        *lock = false;*  
        Remainder section  
**until false;**

- Both do not satisfy the bounded waiting requirement.

# Correct solution with Test-and-set

Shared data: **var** *waiting*: array[0..*n*-1] **of boolean**; *lock*: boolean; //All initialized to *false*

## Process $P_i$

```
var j: 0..n-1; key: boolean;  
repeat  
    waiting[i] = true;  
    key = true;  
    while waiting [i] and key do      // enter CS if either waiting[i] or key is false  
        key = Test-and-Set (lock);      // key is false if lock is false
```

```
    waiting[i] = false;  
    Critical Section
```

```
    j = i+1 mod n;  
    while (j ≠ i) and not waiting[j] do // check if any  $P_j$  is waiting for CS  
        j = j+1 mod n
```

```
    if j = i then  
        lock = false;          // no other process is waiting for CS
```

```
    else  
        waiting[j] = false;    //  $P_j$  is waiting, let it enter CS next
```

\* \* \* \* \*

*Remainder Section*

```
until false;
```

Proof: Read textbook.

# Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.

- Semaphore **S** – integer variable

- Can only be accessed via two indivisible (atomic) operations

- **wait()** and **signal()**

- Originally called **P()** and **V()**

- Definition of the **wait() operation**

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

- Definition of the **signal() operation**

```
signal(S) {  
    S++;  
}
```

# Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1

- Same as a **mutex lock**

- Can solve various synchronization problems

- Consider  $P_1$  and  $P_2$  that require  $S_1$  to happen before  $S_2$

Create a semaphore “**synch**” initialized to 0

**P1:**

```
s1;  
signal(synch);
```

**P2:**

```
wait(synch);  
s2;
```

- Can implement a counting semaphore  $S$  as a binary semaphore

# Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the `wait` and `signal` code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
  - Note that applications may spend lots of time in critical sections and therefore this is not a good solution

# Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - block** – place the process invoking the operation on the appropriate waiting queue
  - wakeup** – remove one of processes in the waiting queue and place it in the ready queue

```
• typedef struct{  
    int value;  
    struct process *list;  
} semaphore;
```

$x \leq 0$ ;  
**wait(x){**

$0 - 1 = -1$ ; //block

}

$x < 0$ ;  
**signal(x){**

$-1 + 1 = 0$ ; //wakeup

}

## Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {  
    S->value--;  
  
    if (S->value < 0) {  
        add this process to S->list;  
        block();  
    }  
}  
* * *  
* * *  
  
signal(semaphore *S) {  
    S->value++;  
  
    if (S->value <= 0) {  
        remove a process P from S->list;  
        wakeup(P);  
    }  
}  
* * *
```

# Classical Problems of Synchronization

- Classical problems used to test new synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

# Bounded-Buffer Problem

- $n$  buffers, each can hold one item
  - Semaphore `mutex` initialized to the value 1
  - Semaphore `full` initialized to the value 0
  - Semaphore `empty` initialized to the value  $n$
- .....

# Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {  
    ...  
    /* produce an item in next_produced */  
    ...  
    wait(empty);  
    wait(mutex);  
    ...  
    /* add next produced to the buffer */  
    ...  
    signal(mutex);  
    signal(full);  
} while (true);
```

# Bounded Buffer Problem (Cont)

- ② The structure of the consumer process

```
Do {  
    wait(full);  
    wait(mutex);  
    ...  
    /* remove an item from buffer to next_consumed */  
    ...  
    signal(mutex);  
    signal(empty);  
    ...  
    /* consume the item in next consumed */  
    ...  
} while (true);
```

# Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
  - Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore `rw_mutex` initialized to 1
  - Semaphore `mutex` initialized to 1
  - Integer `read_count` initialized to 0

# Readers-Writers Problem (Cont.)

- The structure of a writer process

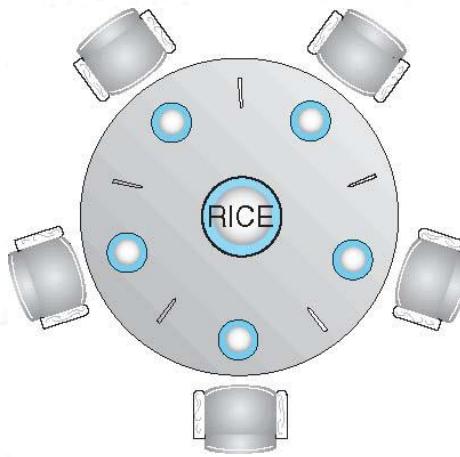
```
do {  
    wait(rw_mutex);  
  
    ...  
    /* writing is performed */  
  
    ...  
  
    signal(rw_mutex);  
} while (true);
```

# Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {  
    wait(mutex);  
    read_count++;  
    if (read_count == 1)  
        wait(rw_mutex);  
    signal(mutex);  
  
    /* ... reading is performed */  
  
    ...  
  
    wait(mutex);  
    read_count--;  
    if (read_count == 0)  
        signal(rw_mutex);  
    signal(mutex);  
} while (true);  
....
```

# Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore **chopstick [5]** initialized to 1

# Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {  
    wait (chopstick[i] );  
    wait (chopStick[ (i + 1) % 5] );  
  
        // eat  
  
    signal (chopstick[i] );  
    signal (chopstick[ (i + 1) % 5] );  
  
        // think  
  
} while (TRUE);
```

- What is the problem with this algorithm?

## Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section).
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

# Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) .... wait (mutex)
  - wait (mutex) ... wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.

# Monitors

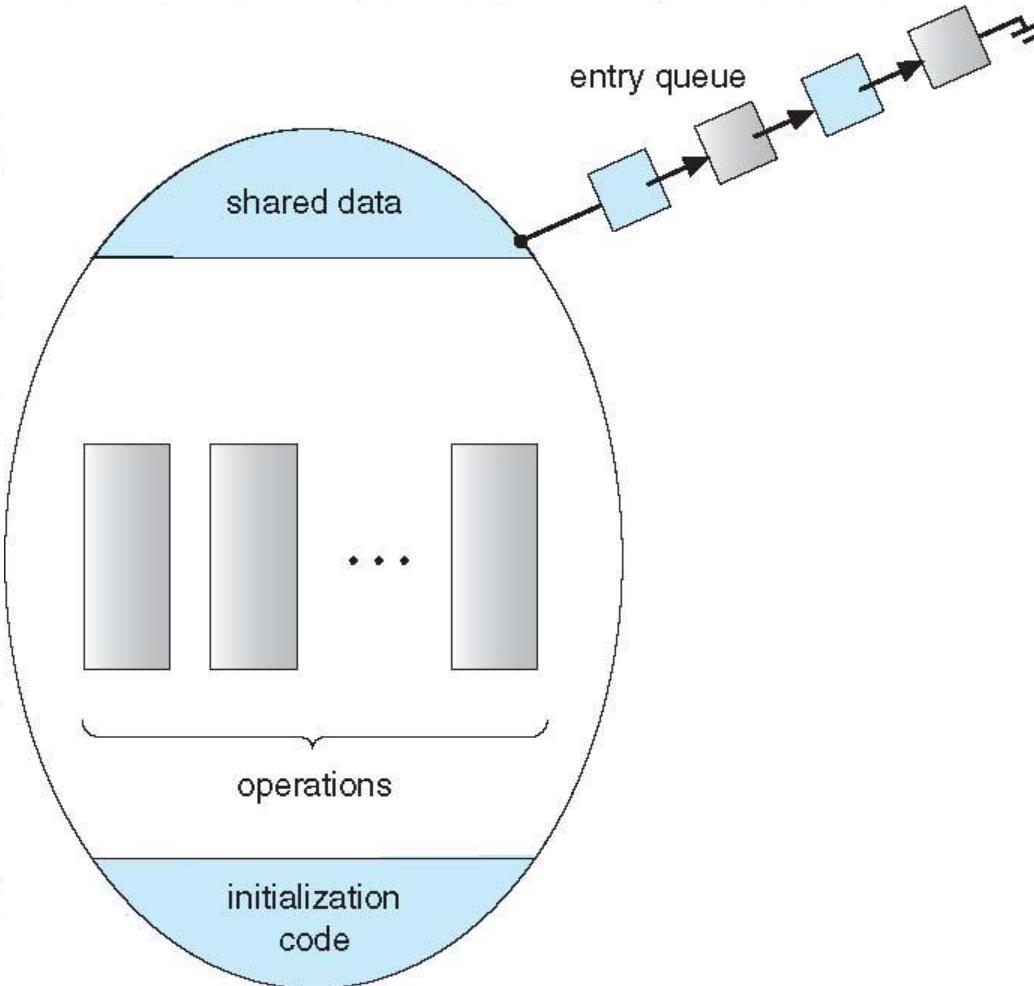
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { .... }

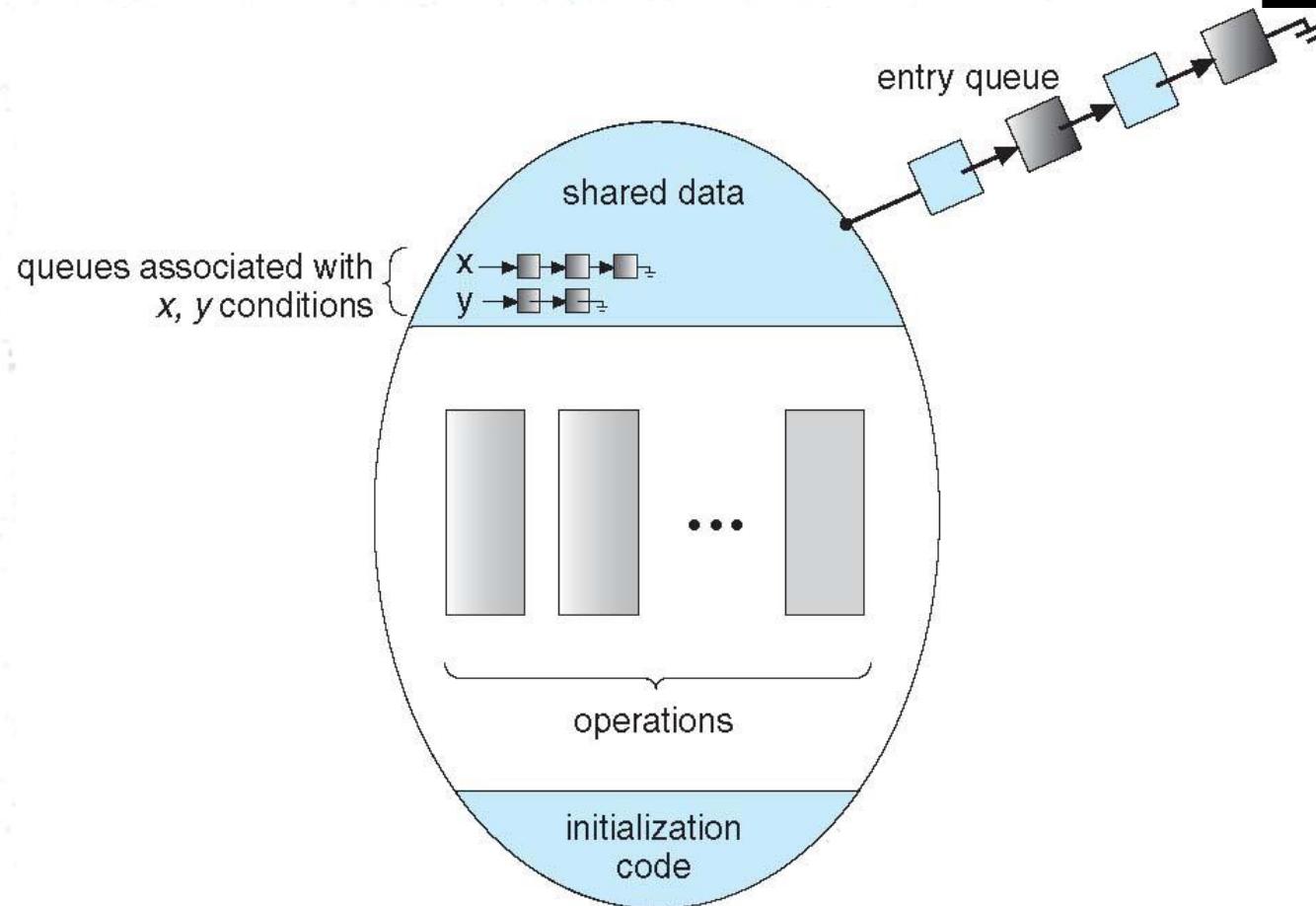
    procedure Pn (...) {.....}

    Initialization code (...) { ... }
}
```

# Schematic view of a Monitor



# Monitor with Condition Variable



# End of Chapter 5

**U.U.Samantha Rajapaksha**

Senior Lecturer  
Coordinator-M.Sc. in IT  
Faculty of Computing  
B.Sc.(Engineering) Moratuwa, M.Sc. IT (SLIIT)  
Email: samantha.r@sliit.lk  
Tel:112301904 Ext:4116  
Web:[www.sliit.lk](http://www.sliit.lk)



# SLIIT

*Discover Your Future*

## IT2060/IE2061

# Operating Systems and System Administration

## Lecture 08

### Memory Management

**U. U. Samantha Rajapaksha**

M.Sc.in IT, B.Sc.(Engineering) University of Moratuwa

Senior Lecturer SLIIT

[Samantha.r@slit.lk](mailto:Samantha.r@slit.lk)

# Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Paging

# Main Memory – Chapter 8

- Memory is a large array of words or bytes, each with its own address
    - It is a repository of quickly accessible data shared by CPU and I/O devices.
    - Memory and registers are the only storage that CPU can directly access
  - Main memory is a volatile storage device
    - It loses its contents in the case of system failure.
  - A program must be mapped to absolute addresses and loaded into memory.
  - Selection of a memory management scheme for a specific system depends on many factors, especially on the hardware design of the system.
  - The OS is responsible for the following activities:
    - ❖ Keep track of which parts of memory are being used and by whom.
    - ❖ Decide which processes to load next when memory space becomes available.
    - ❖ Allocate and deallocate memory.



**For n bits address, the memory capacity =  $2^n$  bytes**

Binary Address	Hex	Memory Bytes
0000 0000 0000 0000	0000	[ ]
0000 0000 0000 0001	0001	[ ]
0000 0000 0000 0010	0002	[ ]
0000 0000 0000 0011	0003	[ ]
0000 0000 0000 0100	0004	[ ]
0000 0000 0000 0101	0005	[ ]
...		
0000 0000 0100 1001	0049	[ ]
0000 0000 0100 1010	004A	[ ]
0000 0000 0100 1011	004B	[ ]
...		
1111 1111 1111 1111	FFFF	[ ]

**Figure 1.2:** Memory and Addresses

# Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
- Protection of memory required to ensure correct operation

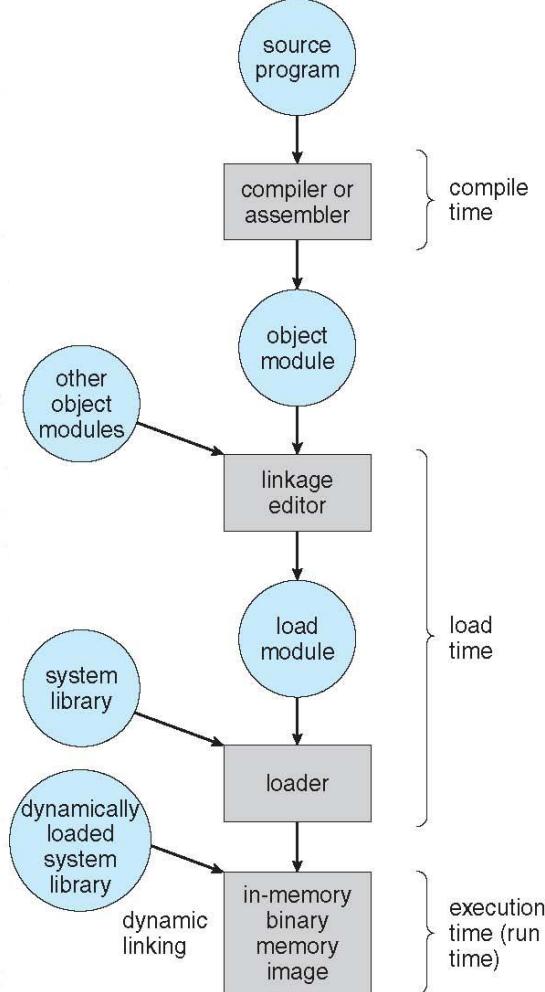
# Address Binding

- Programs on disk, ready to be brought into memory to execute from an **input queue**
    - Without support, must be loaded into address 0000
  - Inconvenient to have first user process physical address always at 0000
    - How can it not be?
  - Further, addresses represented in different ways at different stages of a program's life
    - Source code addresses usually symbolic
    - Compiled code addresses **bind** to relocatable addresses
      - i.e. "14 bytes from beginning of this module"
    - Linker or loader will bind relocatable addresses to absolute addresses
      - i.e. 74014
    - Each binding maps one address space to another

# Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time:** If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time:** Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)

# Multistep Processing of a User Program



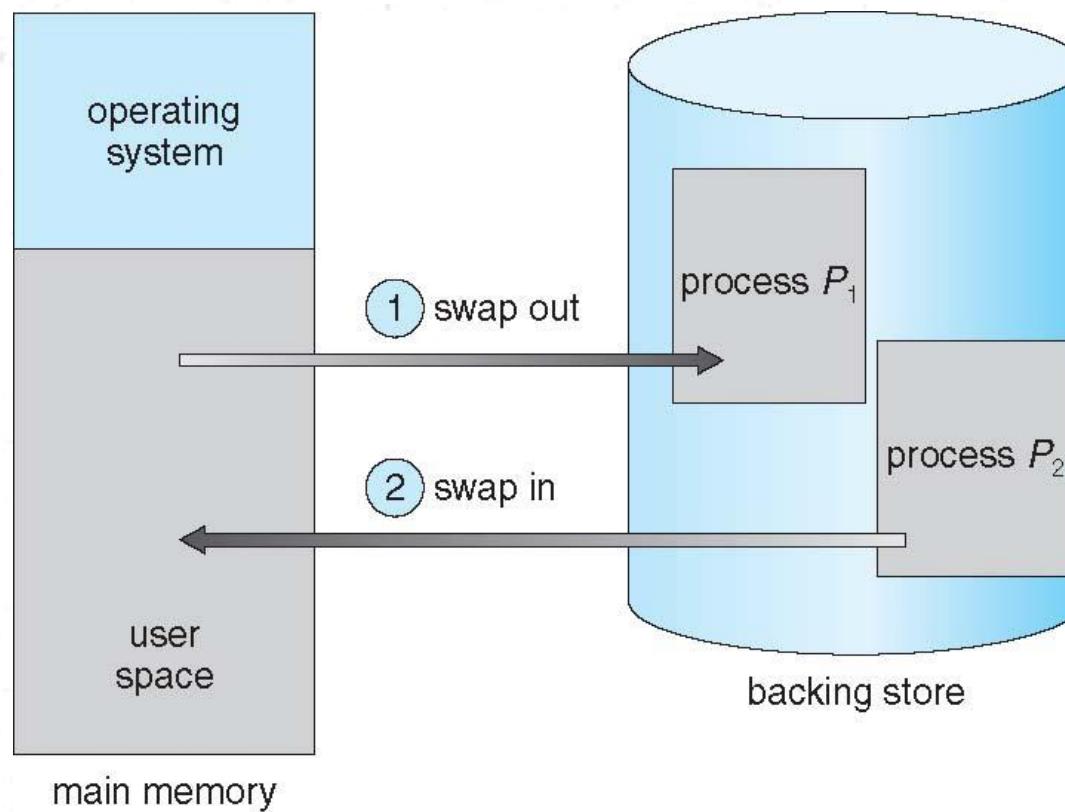
# Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

# Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

# Schematic View of Swapping



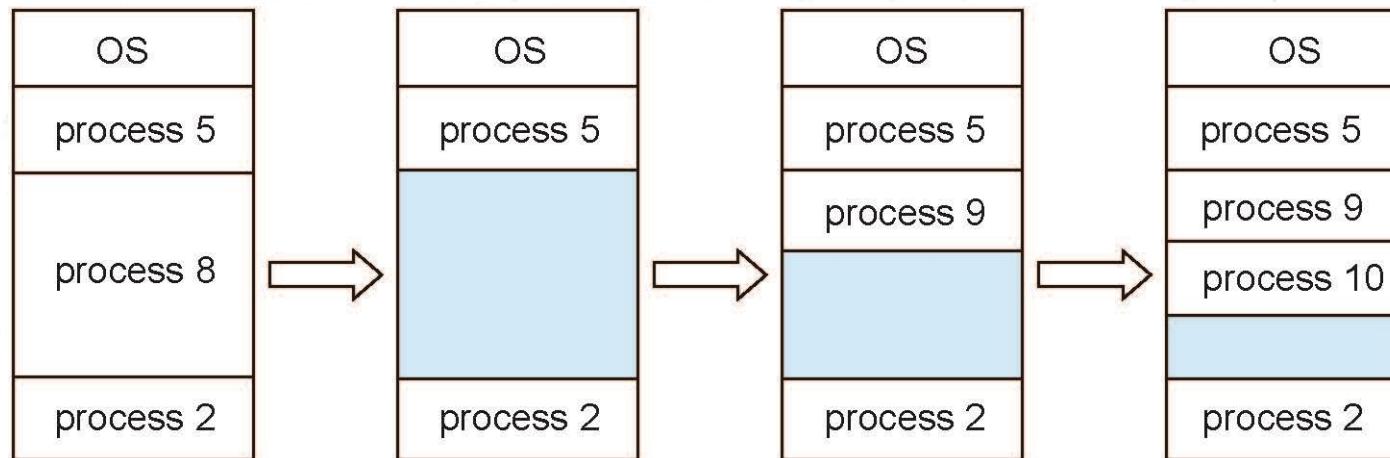
# Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory

# Multiple-partition allocation

- Multiple-partition allocation

- Degree of multiprogramming limited by number of partitions
- **Variable-partition** sizes for efficiency (sized to a given process' needs)
- **Hole** – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about:
  - a) allocated partitions
  - b) free partitions (hole)



# Dynamic Storage-Allocation Problem

How to satisfy a request of size  $n$  from a list of free holes?

- **First-fit:** Allocate the *first* hole that is big enough
- **Best-fit:** Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit:** Allocate the *largest* hole; must also search entire list
  - Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

# Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given  $N$  blocks allocated,  $0.5 N$  blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

# Fragmentation (Cont.)

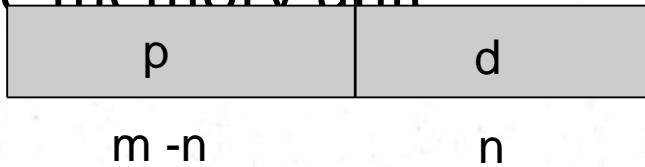
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems

# Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size  $N$  pages, need to find  $N$  free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation

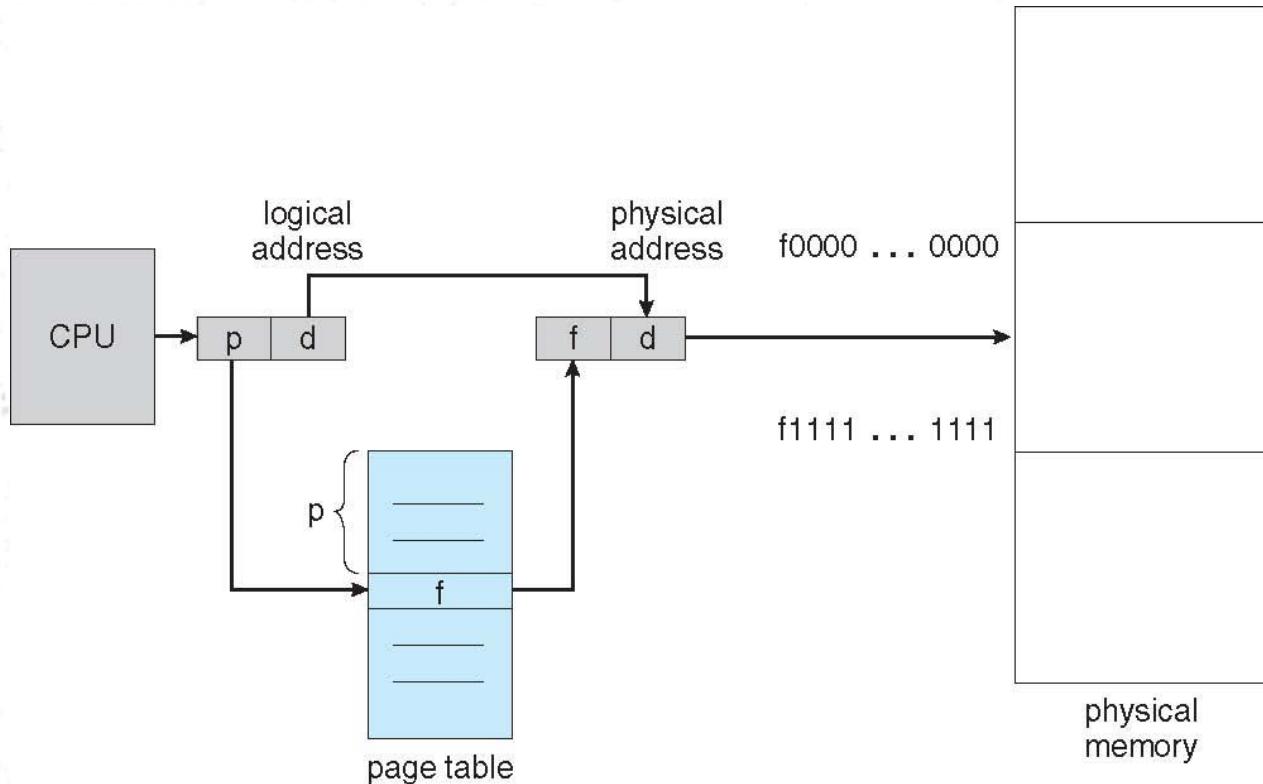
# Address Translation Scheme

- Address generated by CPU is divided into
  - **Page number ( $p$ )** – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset ( $d$ )** – combined with base address to define the **physical memory address** that is sent to the **Memory Unit**

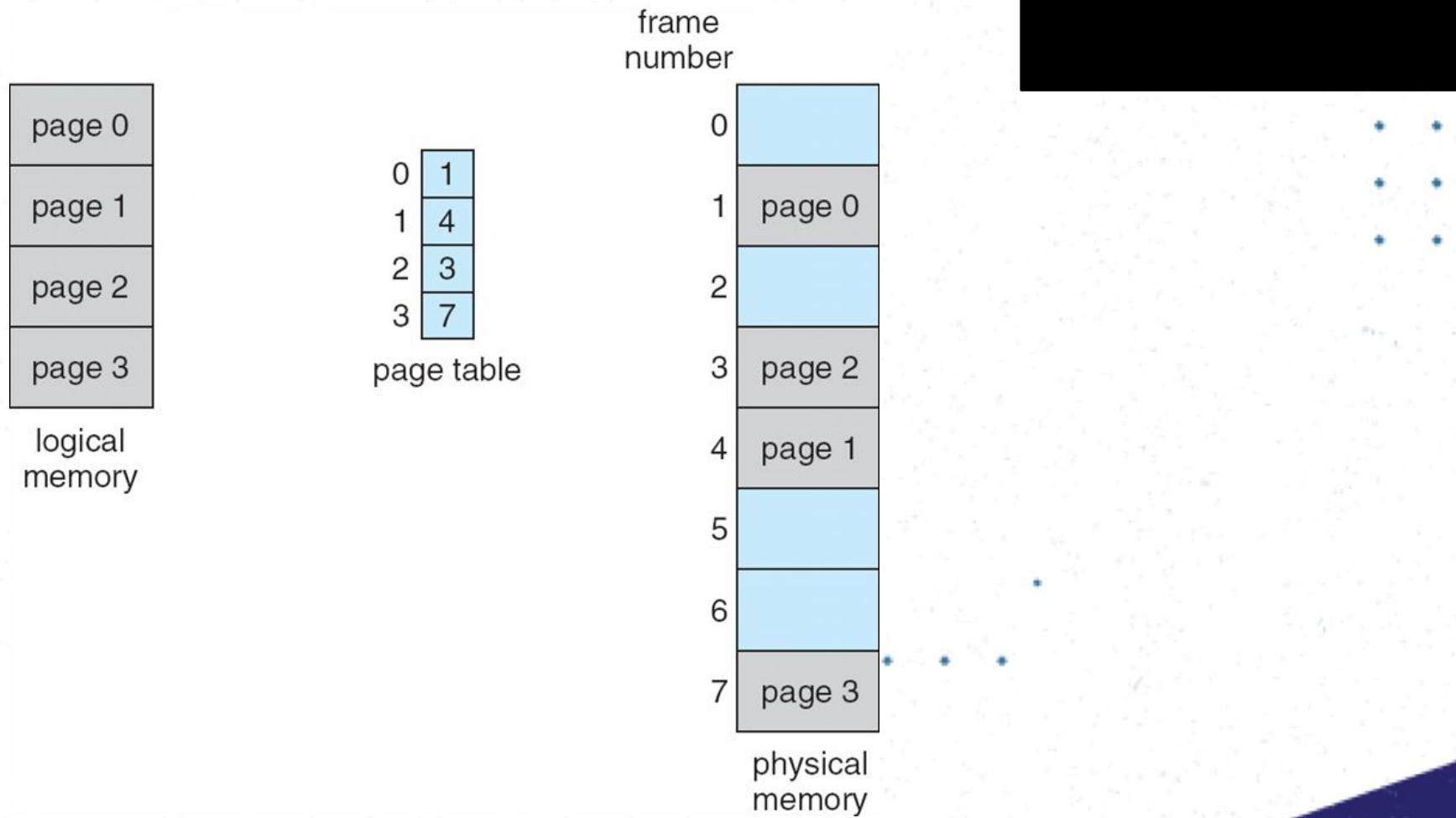


- For given logical address space  $2^m$  and page size  $2^n$

# Paging Hardware



# Paging Model of Logical and Physical Memory



# Paging Example

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

logical memory

0	5
1	6
2	1
3	2

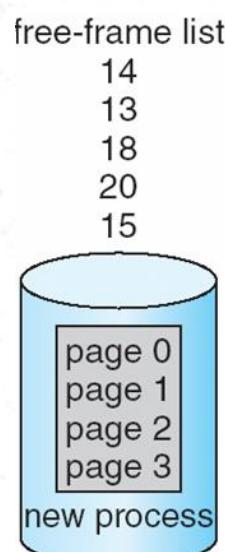
page table

0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

physical memory

$n=2$  and  $m=4$  32-byte memory and 4-byte pages

# Free Frames

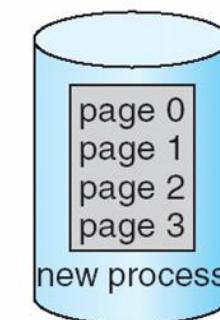


(a)

Before allocation

free-frame list

15
----



(b)

After allocation

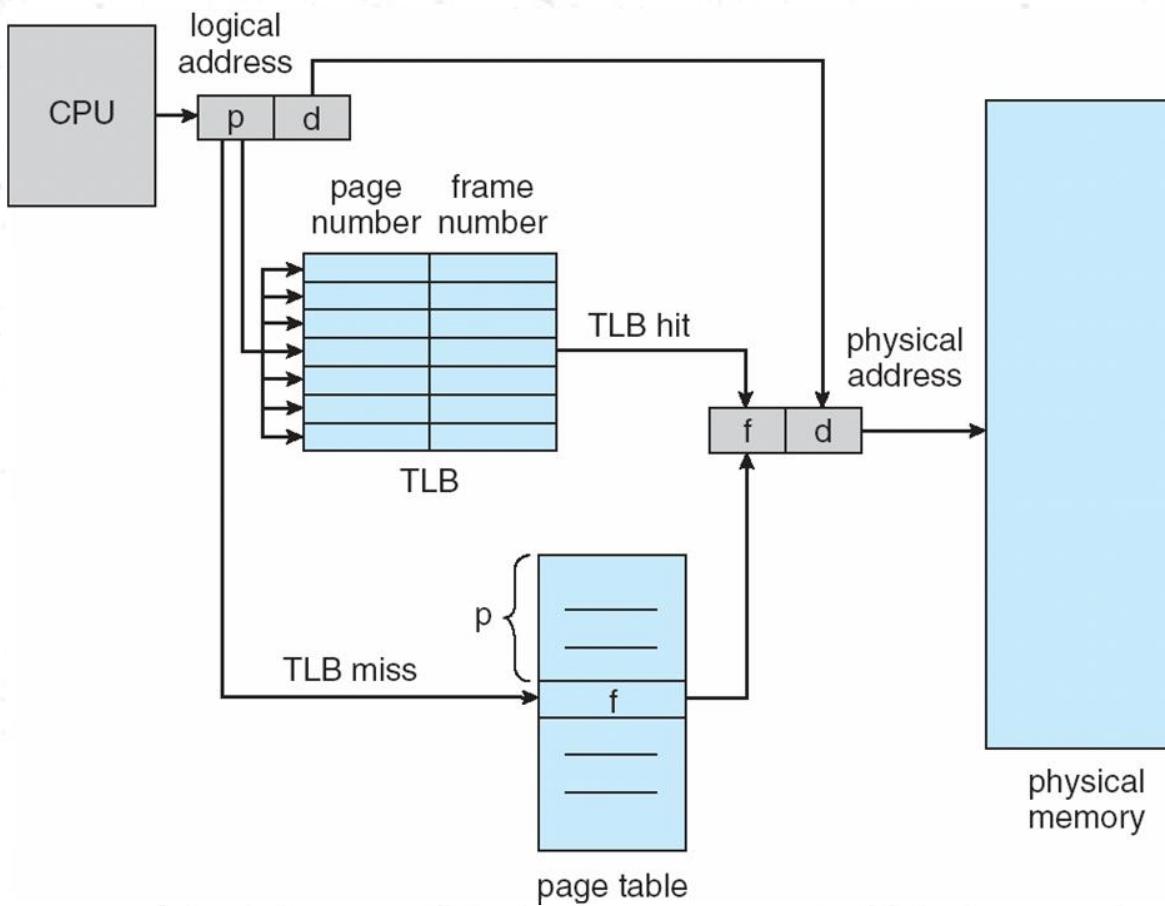
# Associative Memory

- Associative memory – parallel search

Page #	Frame #

- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory

# Paging Hardware With TLB



# Effective Access Time

- Associative Lookup =  $\varepsilon$  time unit
  - Can be < 10% of memory access time
- Hit ratio =  $\alpha$ 
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Consider  $\alpha = 80\%$ ,  $\varepsilon = 20\text{ns}$  for TLB search,  $100\text{ns}$  for memory access
- **Effective Access Time (EAT)**

$$\begin{aligned}\text{EAT} &= (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha) \\ &= 2 + \varepsilon - \alpha\end{aligned}$$

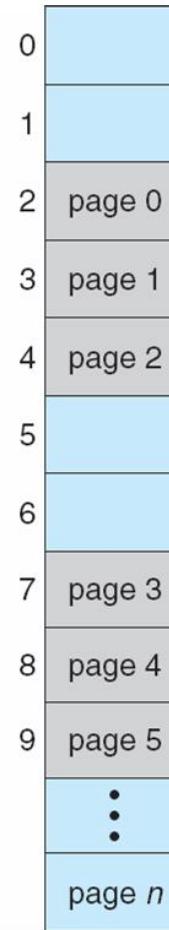
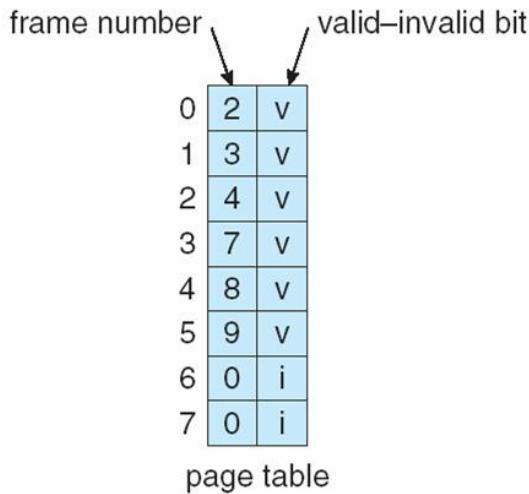
- Consider  $\alpha = 80\%$ ,  $\varepsilon = 20\text{ns}$  for TLB search,  $100\text{ns}$  for memory access
  - $\text{EAT} = 0.80 \times 100 + 0.20 \times 200 = 120\text{ns}$
- Consider more realistic hit ratio  $\rightarrow \alpha = 99\%$ ,  $\varepsilon = 20\text{ns}$  for TLB search,  $100\text{ns}$  for memory access
  - $\text{EAT} = 0.99 \times 100 + 0.01 \times 200 = 101\text{ns}$

# Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
  - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
  - “invalid” indicates that the page is not in the process’ logical address space
  - Or use **page-table length register (PTLR)**
- Any violations result in a trap to the kernel

# Valid (v) or Invalid (i) Bit In A Page Table

00000	page 0
	page 1
	page 2
	page 3
	page 4
10,468	page 5
12,287	



# Page Table Structure

- Hierarchical Paging.
- Hashed Page Tables.
- Inverted Page Tables.

## Hierarchical Paging

- Most modern computer systems support a very large logical address, e.g.,  $2^{32}$  to  $2^{64} \rightarrow$  page table becomes very large.

**Example:** For a system with 32 bit logical address, and page size = 4KB (12 bits offset)

- page table = 1 Million entries ( $32-12 = 20$  bits)  $\rightarrow$  4 MB of physical address spaces for page table;  $2^{20} * 4$  bytes/entry
- One solution is to divide the page table into smaller pieces
  - For a two-level paging scheme, the page table itself is also paged.

## Two-level paging example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits.
  - a page offset consisting of 12 bits.
- Since the page table is paged, the page number is further divided into:
  - a 10 bit page number.
  - a 10 bit page offset.
- Thus, a logical address is as follows:

Page number	Page offset
$p_1$	$p_2$

10

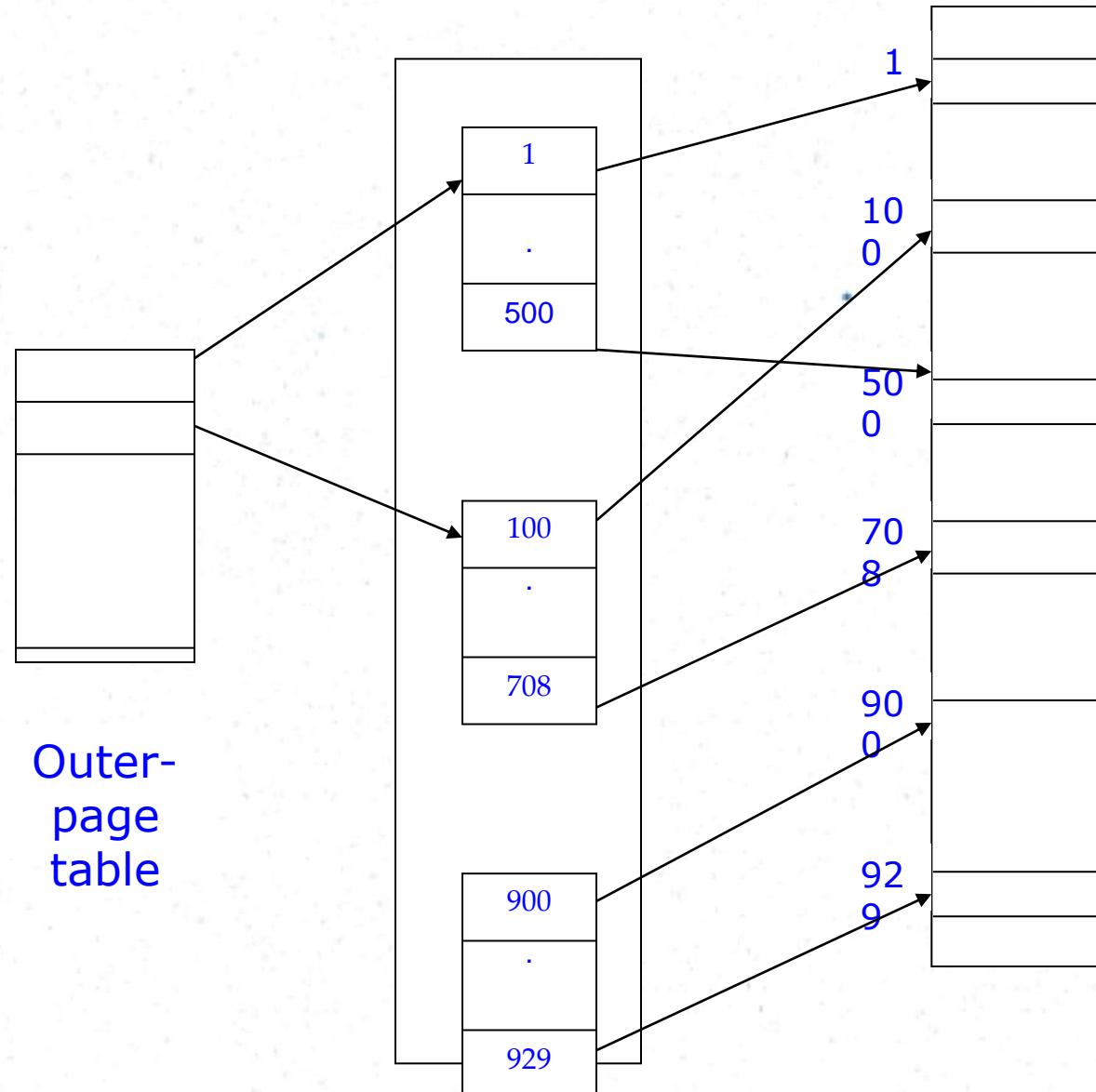
10

12

.....

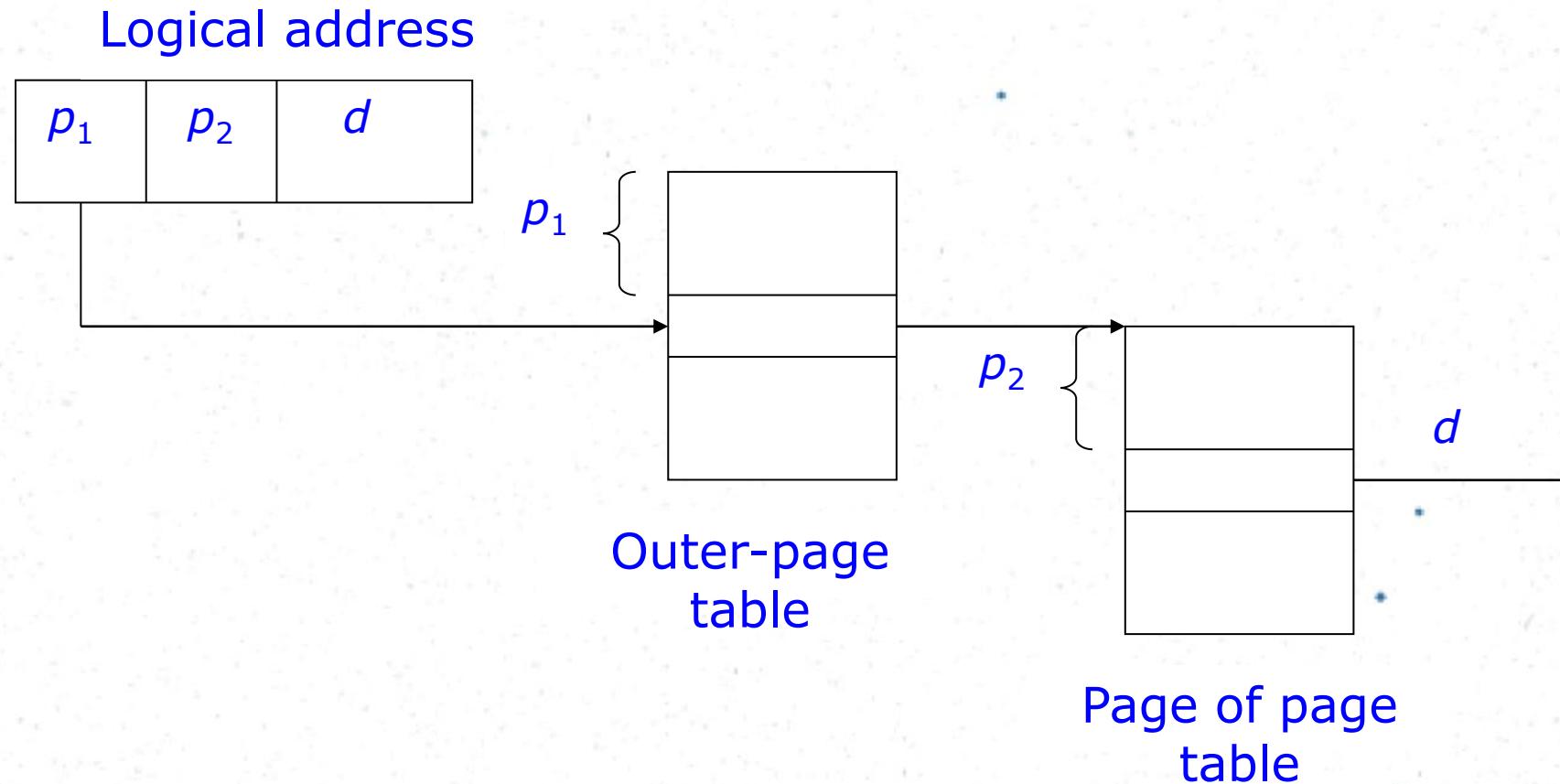
- $p_1$  is an index into the *outer page table*, and  $p_2$  is the displacement within the page of the *inner page table*
  - Translation starts from the outer page table

# Two-level paging example



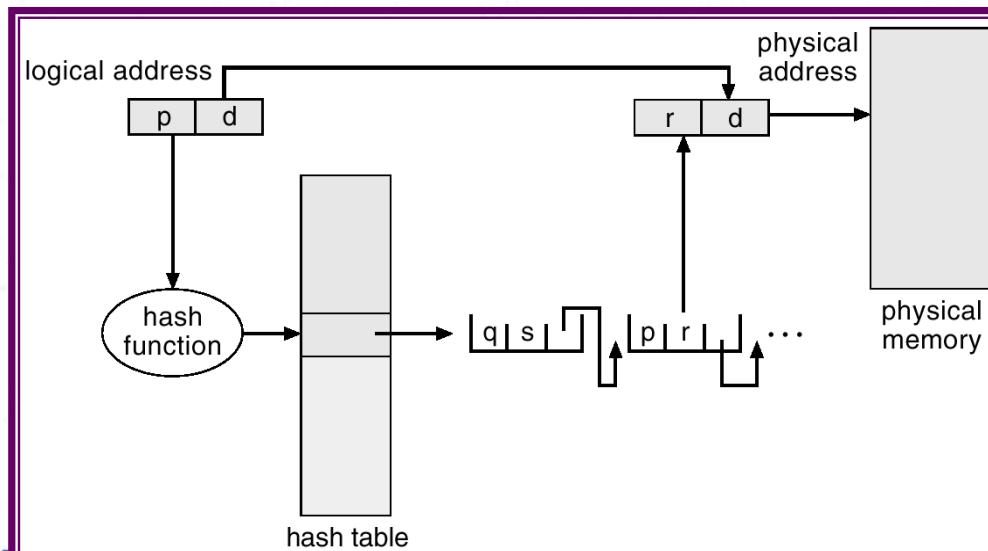
# Address-translation scheme

## Two-level 32-bit paging architecture



# Hashed Page Tables

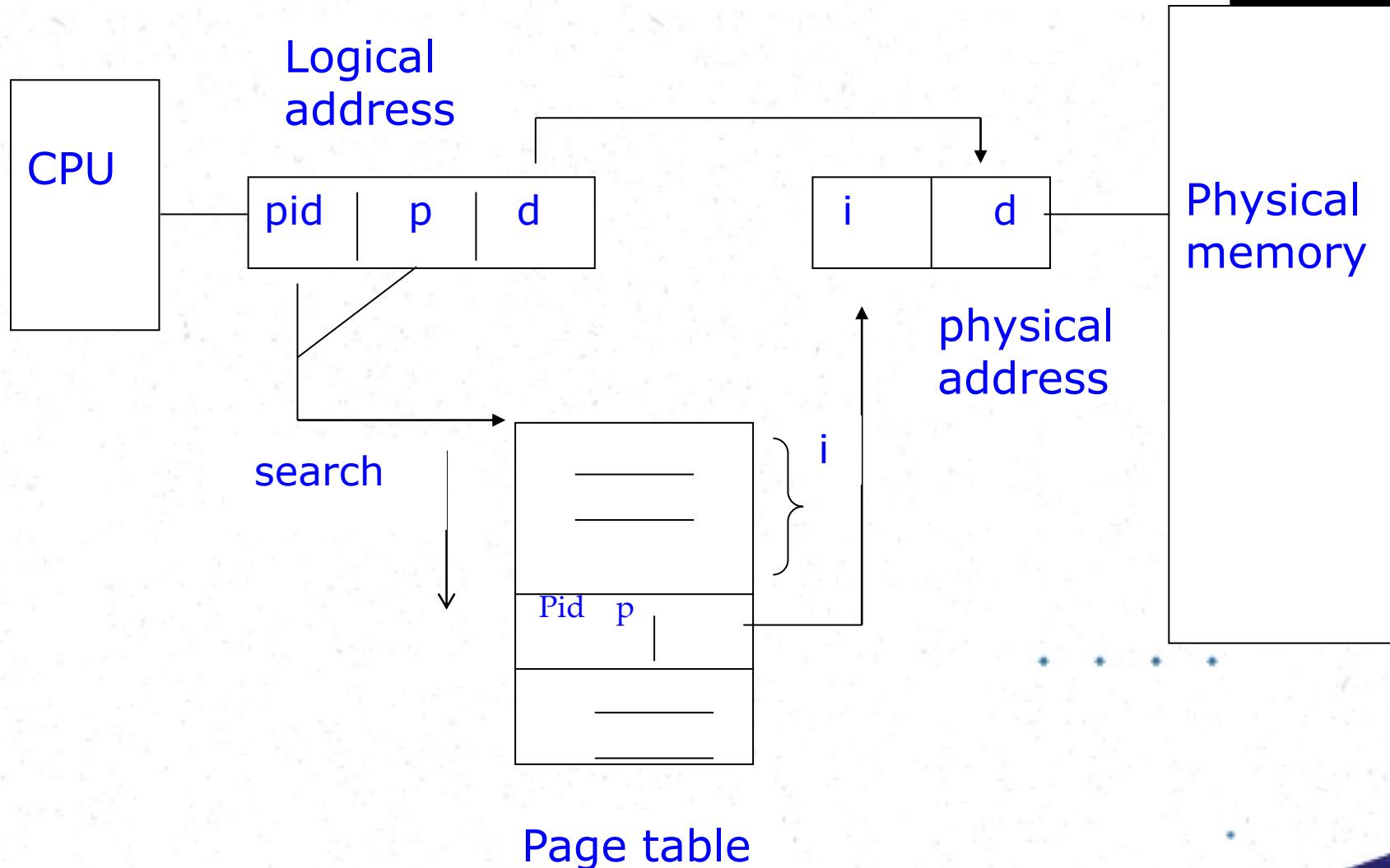
- Is commonly used when address spaces > 32 bits.
- The virtual page number is hashed into a page table.
  - This page table contains a chain of elements hashing to the same location.
  - Each element contains: virtual page number, the value of the mapped frame, pointer to the next element
- Virtual page number is compared in this chain searching for a match.
  - If a match is found, the corresponding physical frame is extracted.
- For address spaces > 64 bits, use Clustered page tables.
  - Similar to hash page table except each entry in page table refers to several pages (e.g., 16).



## Inverted Page Table

- In the paging system, each process has a page table associated with it.
  - Drawback: each page table may consist of millions of entries → page tables consume large amount of physical memory.
- To solve this problem, use Inverted Page Table.
  - It uses ONLY one table that has one entry for each real page of memory.
  - Each entry consists of the virtual address of the page stored in that real memory location, with information about the process (e.g., PID) that owns that page.
  - This scheme *decreases* memory needed to store each page table, but *increases* time needed to search the table when a page reference occurs.
    - It may use hash table to limit the search to one – or at most a few page table entry.

# Inverted page table architecture



## Shared pages

- Other advantage of paging is the possibility of *sharing* common code (reentrant code also called pure code).
- Reentrant code is non-self-modifying code → its code never change during execution.
- Shared code.
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
- Private code and data.
  - Each process keeps a separate copy of the code and data.
  - The pages for the private code and data can appear anywhere in the logical address space.

# Shared pages example

Process

p <sub>1</sub>	ed 1
	ed 2
	ed 3
	data 1
⋮	⋮
⋮	⋮
⋮	⋮
⋮	⋮
⋮	⋮
⋮	⋮
⋮	⋮
⋮	⋮
⋮	⋮
⋮	⋮
⋮	⋮

Page table  
for p<sub>1</sub>

3
4
6
1

Process

p <sub>2</sub>	ed 1
	ed 2
	ed 3
	data 3
⋮	⋮
⋮	⋮
⋮	⋮
⋮	⋮
⋮	⋮
⋮	⋮
⋮	⋮
⋮	⋮
⋮	⋮
⋮	⋮

Page table  
for p<sub>3</sub>

3
4
6
2

Process

p <sub>2</sub>	ed 1
	ed 2
	ed 3
	data 2
⋮	⋮
⋮	⋮
⋮	⋮
⋮	⋮
⋮	⋮
⋮	⋮
⋮	⋮
⋮	⋮
⋮	⋮

Page table  
for p<sub>2</sub>

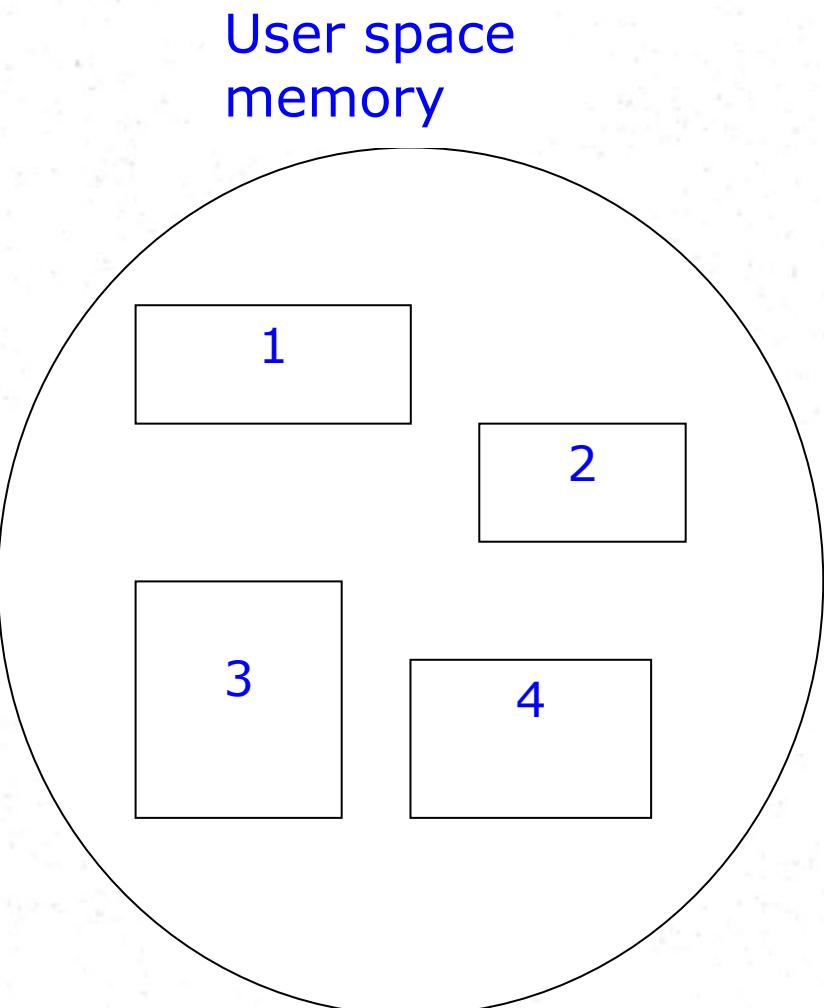
3
4
6
7

0 data 1  
1 data 3  
2 ed 1  
3 ed 2  
4  
5 ed 3  
6 data 2  
7  
8  
9  
10

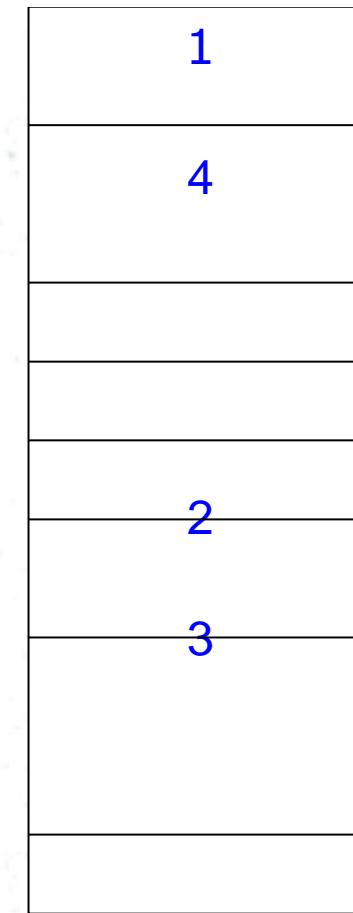
# Segmentation

- Memory management with paging makes separation between user view of memory and the actual physical memory.
- Segmentation is memory-management scheme that supports user view of memory.
- From user's view, a program is a collection of segments, and a segment is a logical unit such as:
  - Main program.
  - Procedure / Function.
  - Local variables, global variables.
  - Common block.
  - Stack.
  - Symbol table, arrays.
- Each segment has a name and length; the addresses specify both the segment name and an offset.
- Program specifies each address by two quantities: a segment name, and an offset (within the segment).

# Logical view of segmentation



Physical  
memory



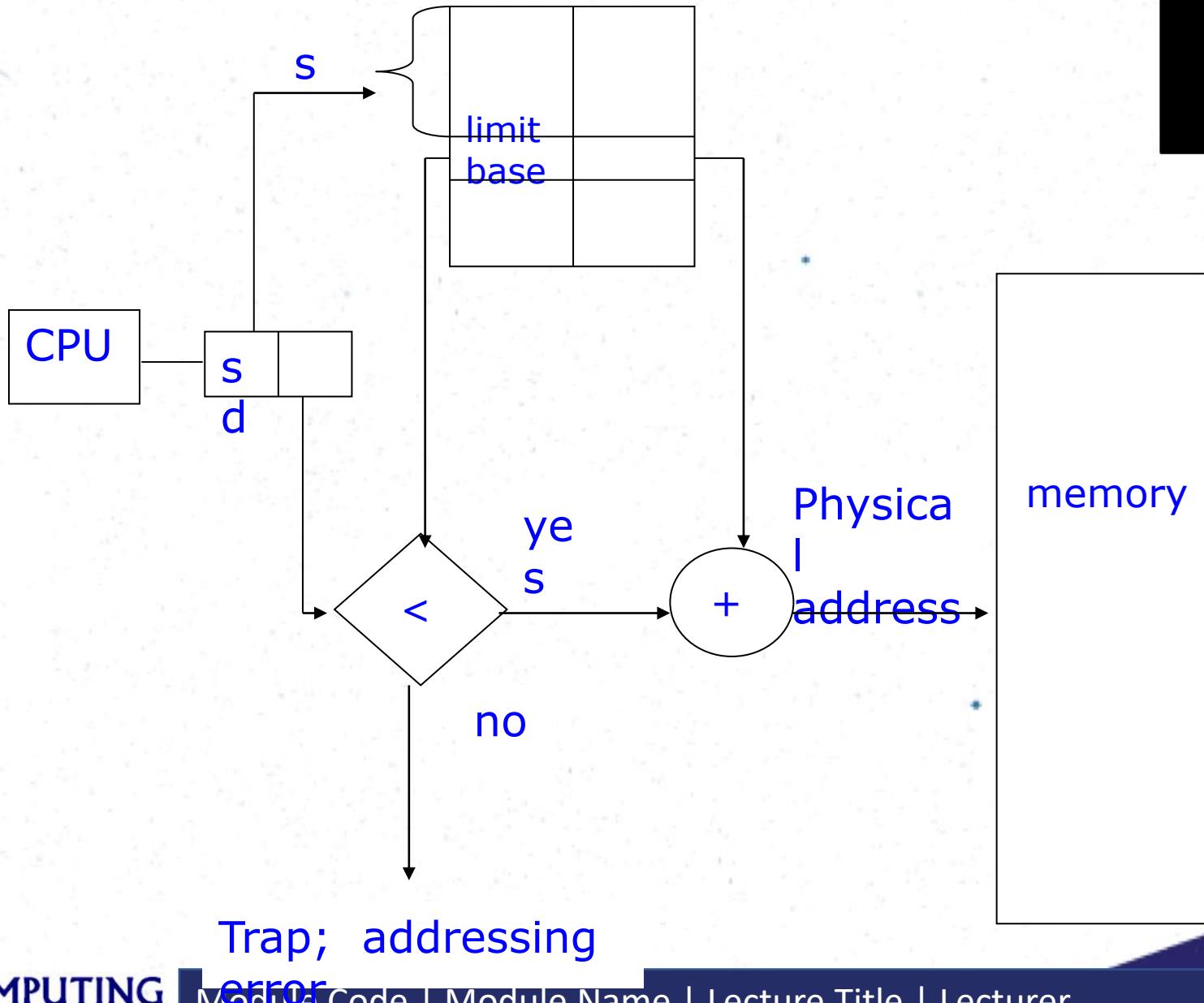
# Segmentation architecture

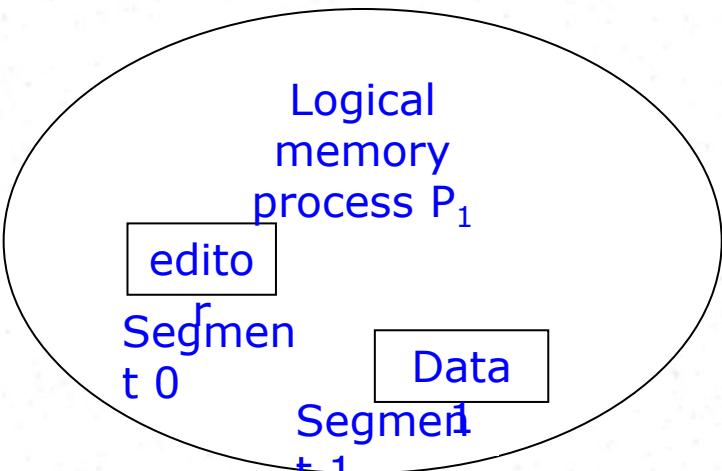
- Logical address consists of a two tuple: <segment-number, offset>.
- *Segment table* – maps two-dimensional user-defined addresses into one-dimensional physical addresses.
  - Each table entry has:
    - *Base* – contains the starting physical address where the segments reside in memory.
    - *Limit* – specifies the length of the segment.
  - *Segment-table base register* (STBR) points to the segment table's location in memory.
  - *Segment-table length register* (STLR) indicates the number of segments used by a program;
    - segment number  $s$  is legal if  $s < \text{STLR}$ .

## Segmentation architecture (cont.)

- Sharing.
  - Shared segments.
  - Same segment number.
- Allocation
  - First fit/best fit.
  - External fragmentation.
- Protection
  - With each entry in segment table associate:
    - Validation bit = 0 → illegal segment.
    - Read/write/execute privileges.
  - Protection bits associated with segments; code sharing occurs at segment level.
  - Since segments vary in length, memory allocation is a dynamic storage-allocation problem.

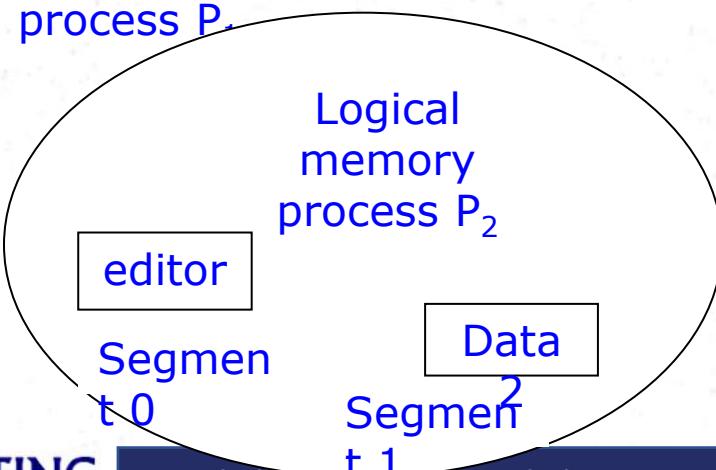
# Segmentation Hardware



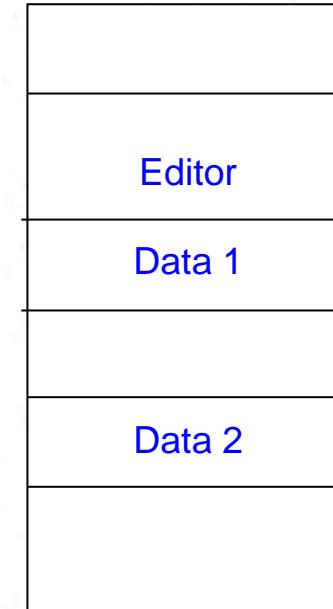


	limit	base
0	25286	43062
1	4425	68348

Segment table  
process  $P_1$



4306  
2  
6834  
8  
7277  
3  
9000  
3  
9855  
3

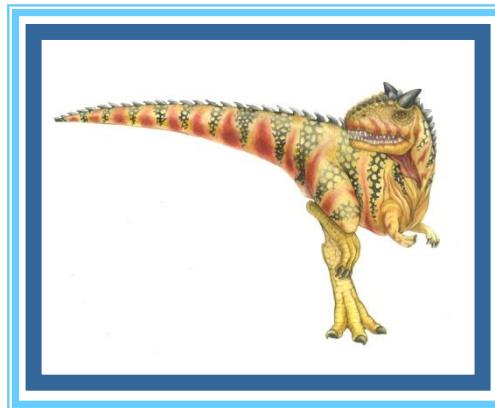


	limit	base
0	25286	43062
1	8850	90003

Segment table  
process  $P_2$

# Chapter 11:

# File-System Interface





# File Concept

---

- Contiguous logical address space
- Types:
  - Data
    - ▶ numeric
    - ▶ character
    - ▶ binary
  - Program
- Contents defined by file's creator
  - Many types
    - ▶ Consider **text file, source file, executable file**





# File Attributes

---

- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk
- Many variations, including extended file attributes such as file checksum
- Information kept in the directory structure

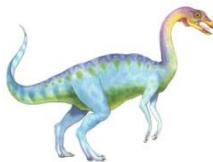




## File concept

- A file is a named collection of related info stored in secondary storage.
- User's perspective: a file is the smallest unit of logical secondary storage
  - data is written to secondary storage in the form of files.
- OS maps files (logical-storage units) to (non-volatile) physical devices.
- Many different types of information may be stored in a file:
  - Data files can be: Numeric, alphabetic, alphanumeric, or binary.
  - Program files can:
    - ▶ Source: a sequence of routines, functions, declarations, and executable statements.
    - ▶ Object: a sequence of bytes organized into blocks understandable by the system's linker.
    - ▶ executable: a series of code sections that the loader can bring into memory and execute.

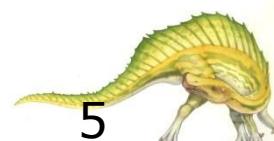




## File concept (cont.)

---

- **File system** consists of:
  - A collection of files – each storing related data.
  - A directory structure – organizes and provides information about all the files in the system.
  - Partitions – used to separate physically or logically large collections of directories.
  
- **File Attributes:** Information about files are kept in the directory structure, which is maintained on the disk, and typically consist of:
  - **Name** – A file is referred to by its name – only information kept in human-readable form.
  - **Type** – needed for systems that support different types.
  - **Location** – pointer to file location on device.
  - **Size** – current file size – in bytes, words, or blocks.
  - **Protection** – controls who can do reading, writing, executing.
  - **Time, date, and user identification** – data for protection, security, and usage monitoring.





# File operations

- The OS provides (through system calls) the following basic file operations:
  - Creating a file.
  - Writing a file.
  - Reading a file.
  - Reposition within file – file seek.
  - Deleting a file.
  - Truncating a file.
- Other common operations:
  - Appending a file.
  - Renaming a file.
  - Copying file.
- Most operations involve searching the directory for entry to the named file.
  - To avoid constant searching, OS keeps **open file table** that contains information about all open files.





## File operations (cont.)

- Most system requires a file to be opened explicitly
  - **open( $F_i$ )** – Search the directory on disk for entry  $F_i$ , copy directory entry (typically a pointer) into the **open-file-table**.
  - **close( $F_i$ )** – Remove directory entry for  $F_i$  from open-file-table.
- In general, an open file needs the following information:
  - File pointer – unique for each process operating on the file.
  - File open count – tracks the number of processes calling **open/close** to certain file.
    - ▶ If count = 0, entry is removed from open-file-table.
  - Disk location of the file: keep in memory, to avoid reading it from disk for each operation.
- Multiuser OS, such as UNIX, keeps two levels of open-file-table:
  - **Per-process table:** keeps information of all open files for each process
    - ▶ e.g., file pointer for each file for use in the read and write calls, and pointers to the system-wide table.
  - **System-wide open-file-table:** contains information which is process-independent
    - ▶ e.g., file location on disk, access dates, file size





# File types

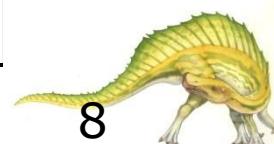
## ❑ OS that recognizes and support file types avoids mistake

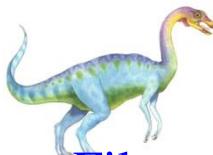
- ❑ e.g., printing a binary-object of a program,

## ❑ Implement file type as part of file name

- ❑ File name is split into – a *name* and an *extension*.

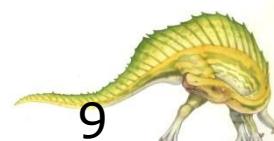
File type	Usual extension	Function
Executable	exe, com, bin, or none	ready-to-run machine language program
Object	obj, o	compiled, machine language, not linked
Source code	c, cc, java, perl, asm	source code in various languages
Batch	bat, sh	commands to the command interpreter
markup	txt, html, xml	textual data, documents
Word processor	docx, rtf, etc	various word-processor formats
Library	lib, a, dll	libraries of routines
Print or view	gif, pdf, jpg	ASCII or binary file for printing/viewing
Archive	zip, tar	Related files grouped into one file, sometimes compressed
multimedia	Mpeg, mp3	Binary file containing audio or A/V information

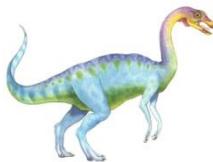




# File Structure

- Files must conform to a required structure that is understood by the OS.
- Some of the file structures are:
  - Free forms, such as text files – sequence of characters organized into lines/pages.
  - Simple record structure.
    - ▶ Lines.
    - ▶ Fixed length.
    - ▶ Variable length.
  - Complex structures.
    - ▶ Formatted document.
    - ▶ Relocatable load file.
- OS must support at least one structure – i.e., for executable files – so that the system is able to load and run programs.





# File Structure (cont.)

## Who decides?

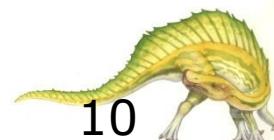
- Operating system: the more file structures, the more complex the OS to support them.
- Program/its creator: the problem if the used structure is not supported by OS.

## Example: UNIX

- A file is a sequence of 8-bit byte; no interpretation from OS for these bits.
  - The scheme provides maximum flexibility, but little support
    - ▶ Each application program must include its own code to interpret an input file into the appropriate structure.

## Internal file structure

- All disk I/O is performed in units of one block (physical record) and all blocks are the same size (e.g., 512 bytes).
- Logical records may vary in size.
  - Solution: packing a number of logical records into physical blocks.
    - ▶ This may create internal fragmentation.





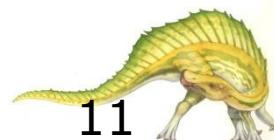
# Access Methods

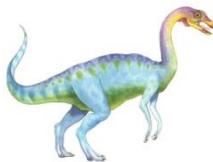
---

Information in a file can be accessed using **sequential access** and **direct access**.

## Sequential access

- Information in the file is processed in order, one record after the other.
  - Based on the *tape* model.
  - Editors and compilers use this model.
- Common operations:
  - *read*: reads *next* portion of the file, and advances file pointer.
  - *write*: appends to the end of file, advances file pointer.
  - *reset*: file pointer to the beginning of file.

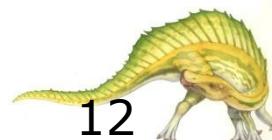




## Access Methods (cont.)

### Direct access (relative access).

- A file is made up of fixed-length logical records that allow programs to read and write records in no particular order.
- Based on disk model → allow random access to any file block, i.e., reads block 14, 32, and then writes to block 7.
- User provides only *relative block number*: an index relative to the beginning of file.
  - Databases are often of this type.
- Common file operations include:
  - *read n* → read block number *n*.
  - *write n* → write block number *n*.
  - OR: *position to n* followed by *read next*.
  - OR: *position to n* followed by *write next*.



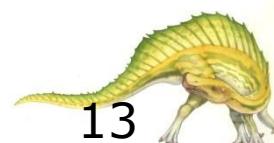
12



# Directory structure

---

- To manage large number of files, we need to organize them
  - The file system is broken into **partitions/volumes**.
    - ▶ One disk may contain several partitions.
    - ▶ One partition may also consist of several disks.
  - Each partition has device directory or volume table of contents (called *directory*).
- Advantage of organizing directory (logically):
  - Efficiency – locate a file quickly.
  - Naming – convenient to users;
    - ▶ Two users can have same name for different files.
    - ▶ The same file can have several different names.
  - Grouping – logical grouping of files by properties, (e.g., all C programs, all games, ... ).





## Directory structure (cont.)

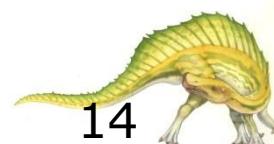
- A directory can be viewed as a symbol table that translates file names into their directory entries.
  - or a collection of nodes containing information about all files.
- Both the directory structure and the files reside on disk.

### Operations performed on a directory are:

- Search/create/delete/rename a file.
- List a directory and traverse the file system.

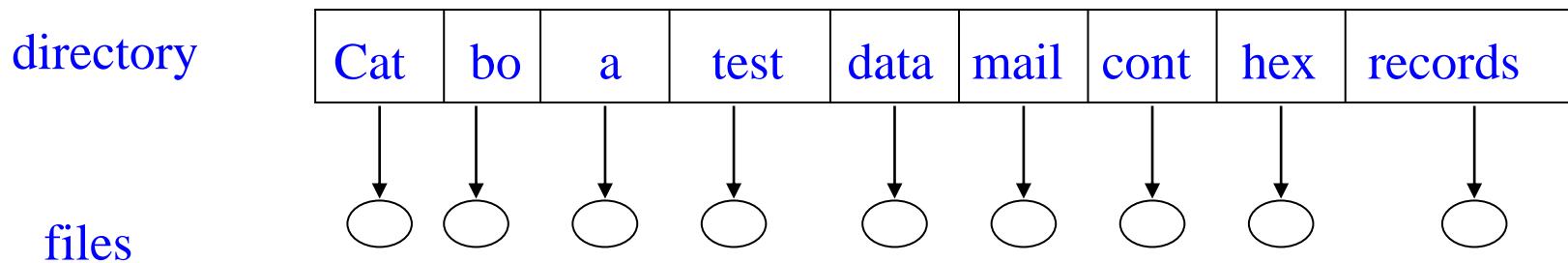
### Information in a device directory

- Name
- Type
- Address
- Current length
- Maximum length
- Date last accessed
- Date last updated
- Owner ID
- Protection information





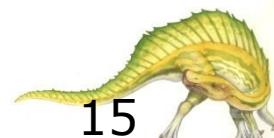
# Single-level directory



- A single directory for all users.
- The simplest directory structure.

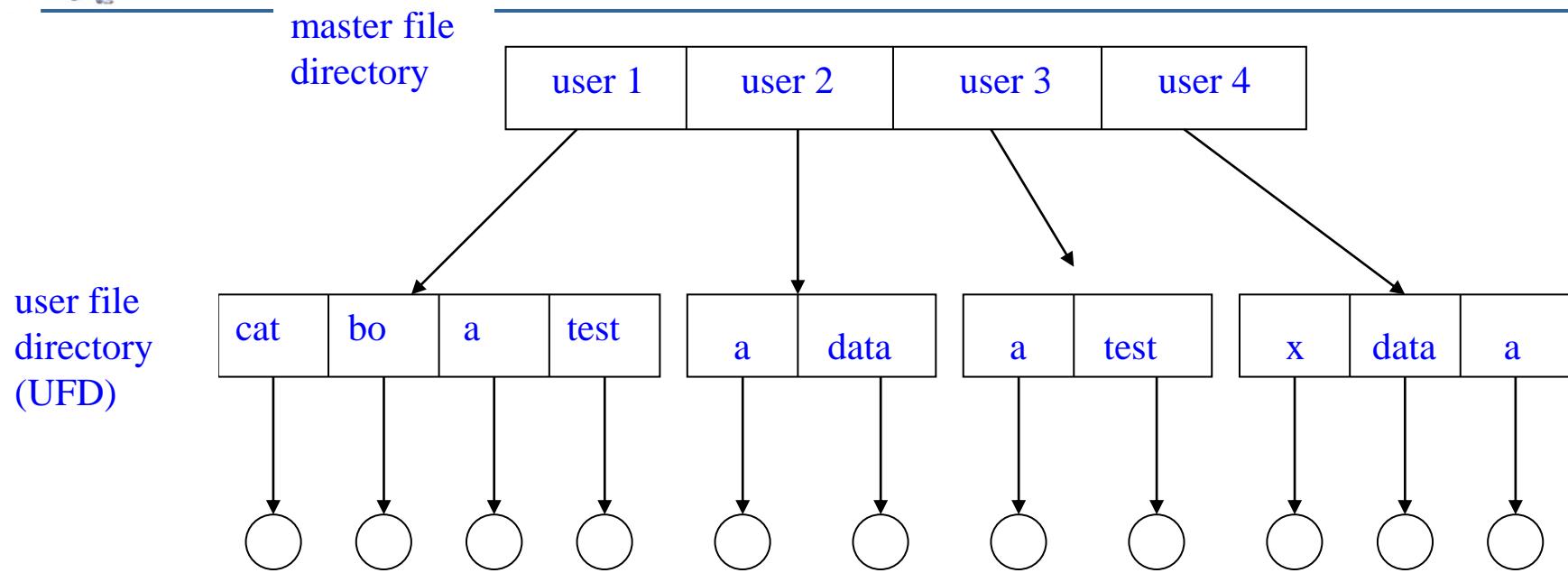
## Problems:

- Naming problem: two users may have same file names.
- For hundreds/thousands of files, it is difficult to remember all file names.

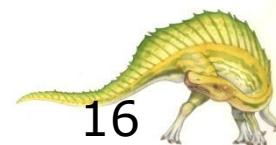




## Two-level directory - two-level tree



- Create a separate directory for each user.
- Each user needs to search only his/her own user file directory (UFD) to find file.
- OS searches only UFD for creating/deleting/accessing certain file.
- A user name and a file name define a path name
  - every file has a path name.





## Two-level directory- two-level tree (cont.)

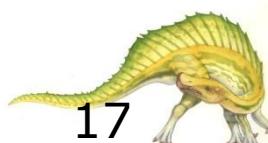
---

### Advantage:

- Solve the name collision problem
  - can have the same file name for different user.
- Efficient searching.

### Disadvantage:

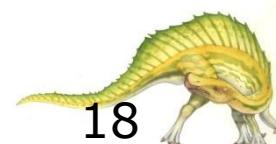
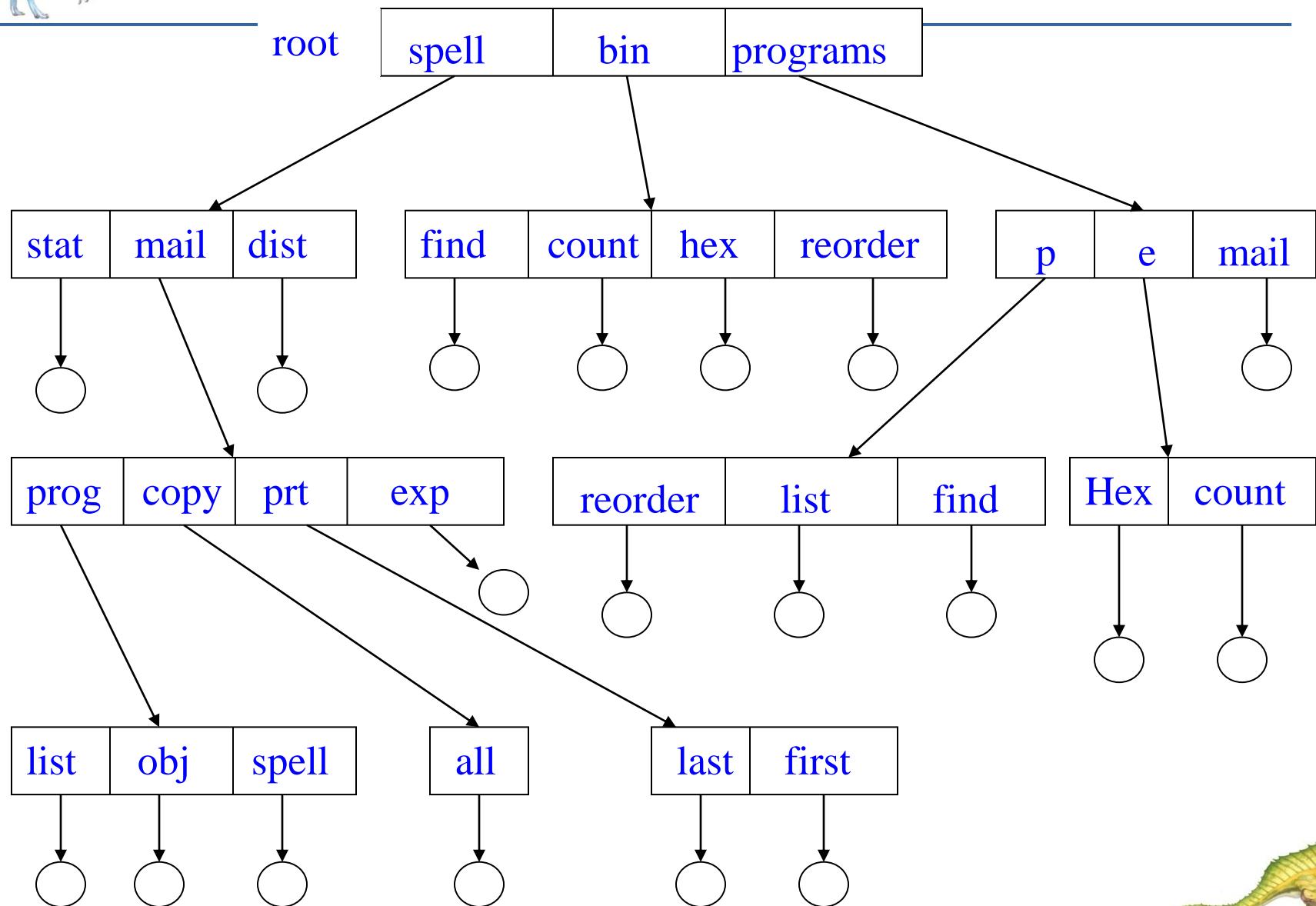
- Isolate one user from another
  - users cooperating on same task cannot access one another's files.
- How do users access system files?



17



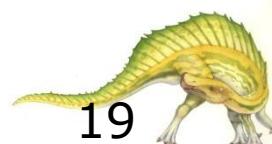
# Tree-structure directories





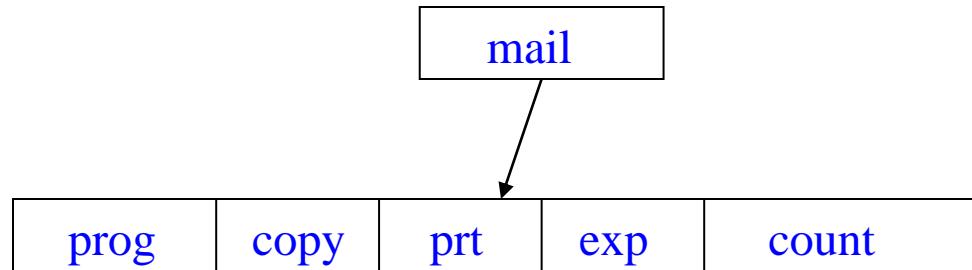
## Tree-structure directories (cont.)

- A directory contains a set of files or **subdirectories**.
  - One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1).
  - The most common directory structure.
- Two types of path name:
  - *absolute* (begins at the root), or
  - *relative* (begins from the current directory) path name.
- What is the policy to delete a non-empty directory?
  - Do not delete a non-empty directory.
  - Allow deleting the directory including its contents (UNIX)
    - ▶ Using **rm -rf**





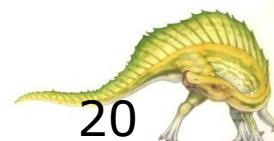
## Example (UNIX)



- Current directory is /spell/mail
  - **mkdir** count: to create a new directory
  - deleting ‘mail’ → deleting the entire subtree rooted by ‘mail’.

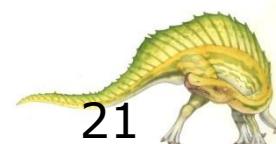
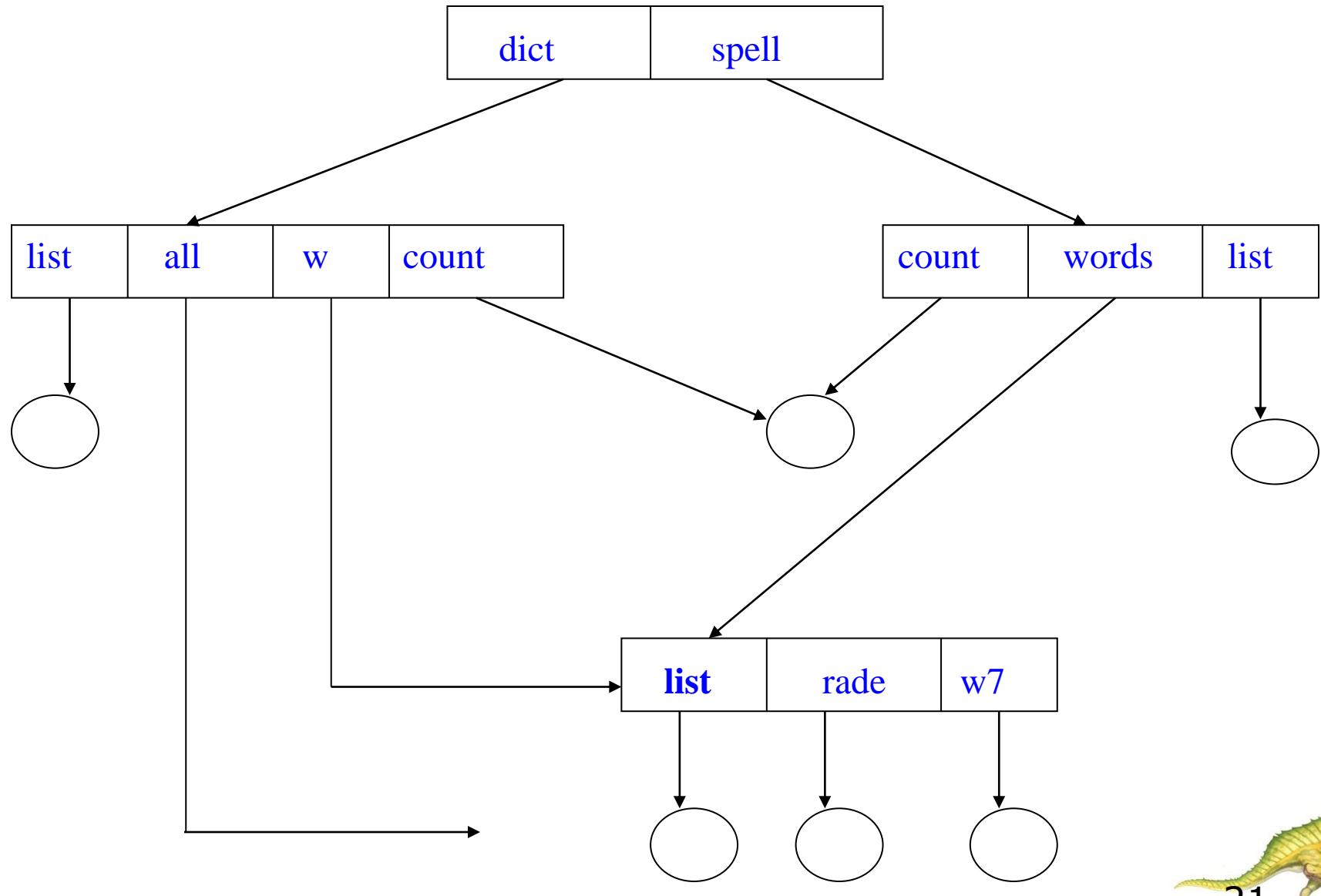
### Advantages of tree-structure:

- Efficient searching.
- Grouping capability – users can access files of other users.





# Acyclic-graph directories



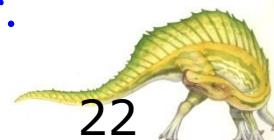


# Acyclic-graph directories

- An acyclic graph allows directories to have **shared subdirectories** and files.
  - The **same file or subdirectory** may be in two different directories.
  - **Tree structure prohibits** the sharing of files or directories.
  - An acyclic graph – a graph with no cycles – is a **generalization of a tree-structured directory**.
- Shared files or directories can be implemented:
  - Unix creates a new directory entry called *link* – a pointer to another file or subdirectory;  
OS ignores these links when traversing directory trees to preserve the acyclic structure of the system.
  - Duplicate all information about them in both sharing directories – problem on how to keep the two consistent if the file is modified.

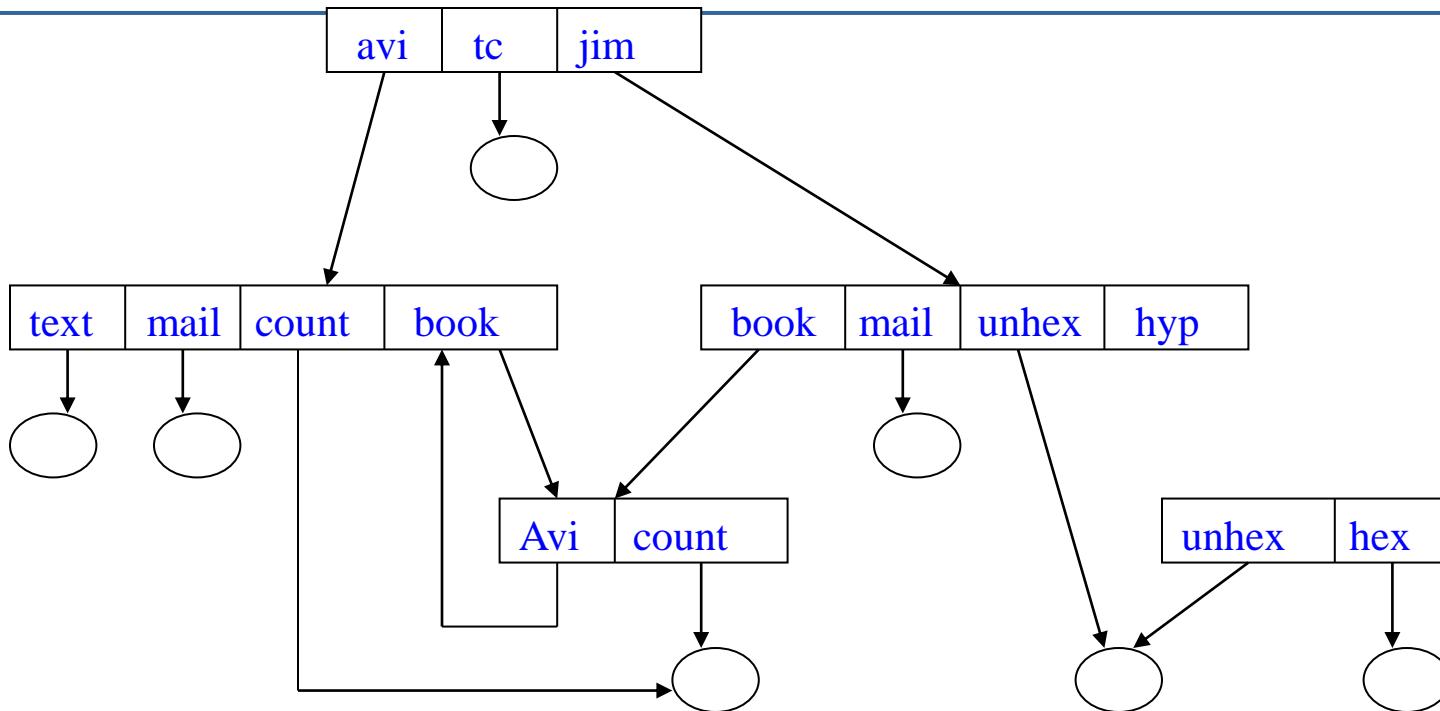
## Problems in Acyclic graph:

- A file may have several absolute path names.
  - Distinct file names may refer to the same file.
- Problem in deletion, e.g., if *dict* deletes *list* → dangling pointer.
  - Solutions: entry-hold-count solution





# General Graph Directory



How do we guarantee no cycles?

- Allow only links to files (not subdirectories).
- Garbage collection → time consuming.
- Every time a new link is added use a cycle detection algorithm to determine if it is ok.





# Protection

- File owner/creator should be able to control:
  - Type of accesses to the file and by whom.
- Access types:
  - read.
  - write.
  - execute.
  - append.
  - delete.
  - list.



24



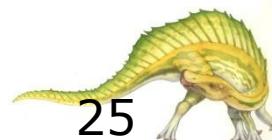
# Access Lists and Groups

- Access right depends on user identity
  - Access control list (ACL) gives user names and access rights for each user
- Three classes of users:
  - Owner.
  - Group.
  - Public.
- Type of accesses: read, write, execute.
- Can create a group (unique name), say *G*, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.

**chmod** 761 *game*

- Attach a group to a file.

**chgrp** *G* *game*





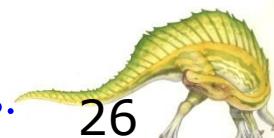
# File System Implementation

## Design problems

- How the file system should appear to users.
- How to design data structure and algorithms to map logical addresses to physical addresses.

## File System Structure

- File structure
  - Logical storage unit.
  - Collection of related information.
- File system resides on secondary storage (disks).
  - I/O transfers between memory and disk are performed in units of blocks.
  - A block is stored in one or more sectors.
  - Sector size varies from 32 bytes to 4096 bytes; usually 512 bytes.





# File system is organized into layers

Application programs



Logical file system



File-organization module



Basic file system



I/O control



devices

- **I/O control:** consists of device drivers and interrupt handlers to transfer information between memory and disk system.
- **Basic file system:** uses generic commands to the device driver to read/write physical blocks on the disk.
- **File organization module:** translates logical block addresses into physical block addresses.
- **Logical file system:** uses the directory structure to provide the file organization module with information given a symbolic file name.
- **Application programs:** creates a file, calls the logical file system.

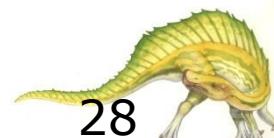




# Open file table

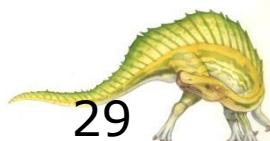
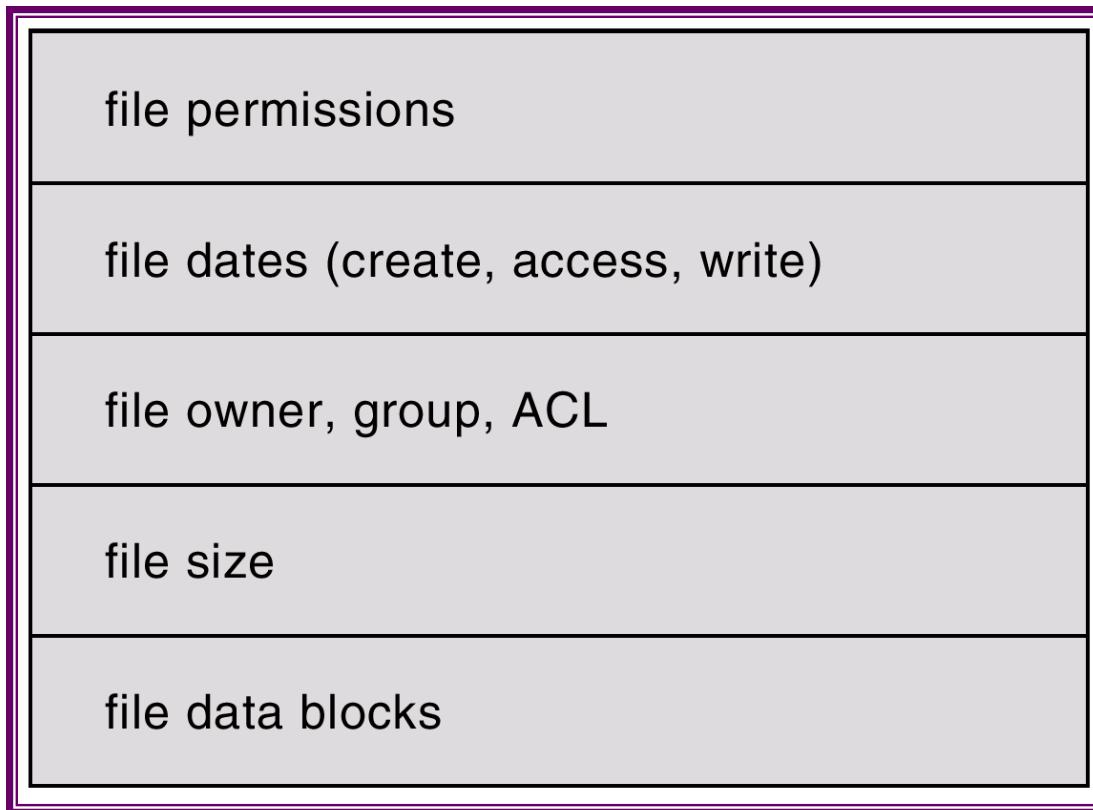
index	File name	permissions	Access dates	Pointer to disk block
0	test.c	rw rw rw	...	→
1	mail.txt	rw		→
2				
...	...	...	...	...
$n$				

- Before a file can be accessed, it must be opened.
- The directory structure is searched for the desired file entry.
  - Parts of the directory structure are usually cached in memory (open file table) to speed up directory operations.
- *Open* will return an *index*
  - called *file descriptor* (UNIX), *file handle* (Windows), or *file control block* (FCB).
  - Use the index for further reference to the file
- FCB contains information about the file.
  - E.g., ownership, permissions, location of the file contents, etc.





# A Typical File Control Block (FCB)

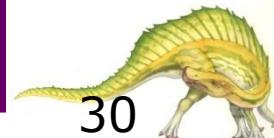
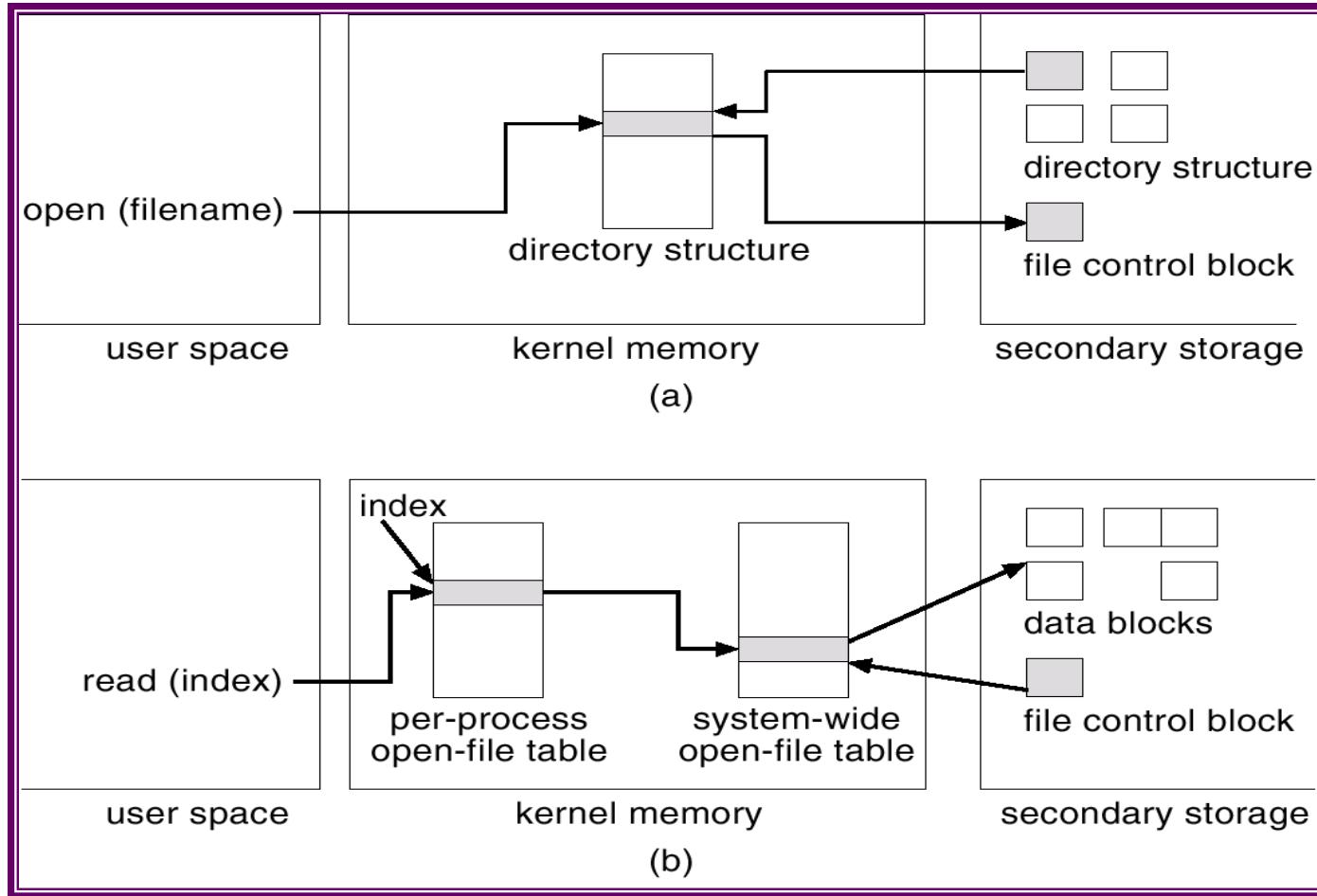


29



# In-Memory File System Structures

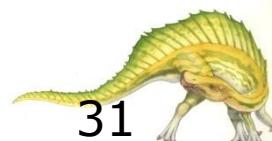
- This figure shows the file system structures provided by OS.
  - Figure (a) refers to opening a file; Figure (b) refers to reading a file.



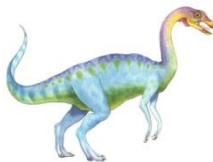


# Directory Implementation

- Use *Linear list* of file names with pointers to the data blocks.
  - Require a linear search to find a particular entry.
  - Simple to program.
  - Disadvantage: Time-consuming to execute.
  
- Use *Hash table* – linear list with hash data structure.
  - Decrease directory search time.
  - *Collision* – situation where two file names hash to the same location.
  - Difficulties:
    - ▶ Fixed size of the hash table, and the dependence of the hash function on the size of the hash table.

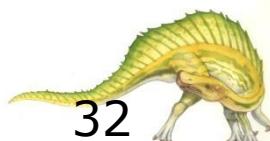


31



# Allocation Methods

- How to allocate disk space (blocks) to files so that the space can be utilized effectively and files can be accessed quickly?
  
- Three major methods:
  - Contiguous.
  - Linked.
  - Indexed.

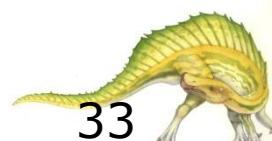


32



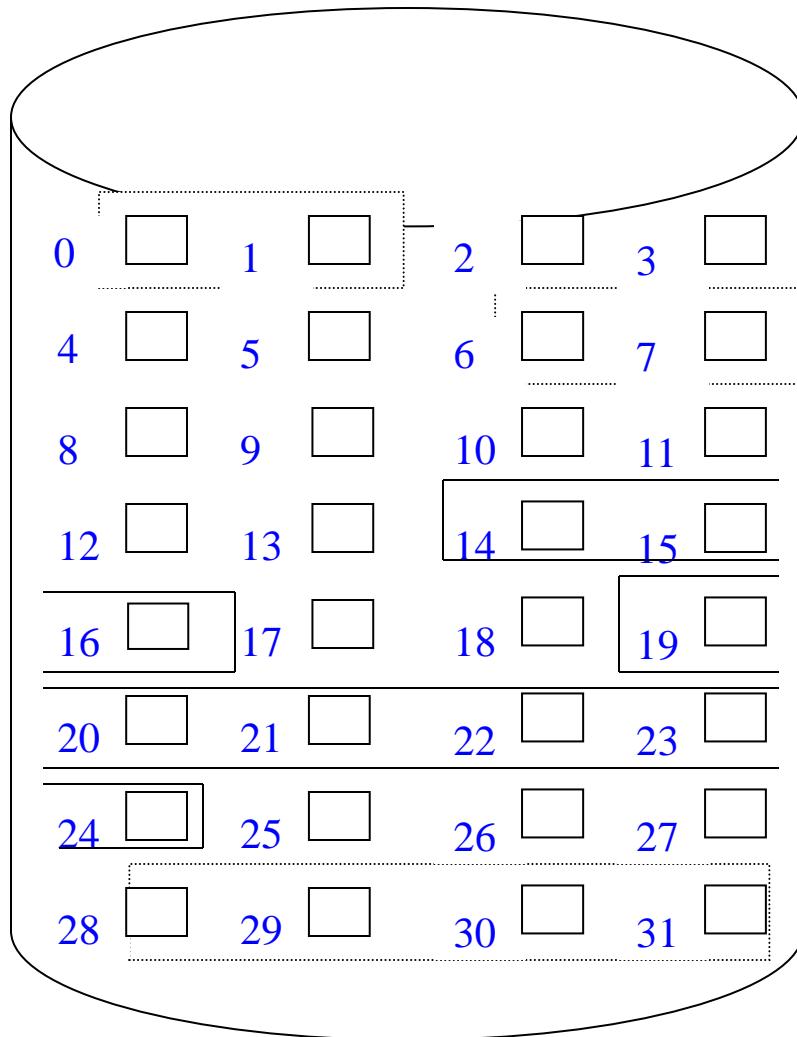
# Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk.
  - Simple – only starting location (block #) and length (number of blocks) are required and kept in directory entry.
  - Support sequential and direct accesses.
- Disadvantages:
  - How to find the *best* space for a new file (dynamic storage-allocation problem).
    - How to satisfy a request of size  $n$  from a list of free blocks.
    - Use first-fit, best-fit, worst-fit algorithm → create external fragmentation.
    - Need disk compaction → expensive.
  - Files can not grow.
    - How much space is needed by a file?
- Mapping from logical address (LA) to physical:
  - $Q = \text{LA DIV } 512, R = \text{LA MOD } 512.$ 
    - Block to be accessed is the  $Q^{\text{th}}$  block; the first is block 0
    - Offset address:  $R$
    - E.g.,  $\text{LA} = 1000$  is in block 1 (the second block), offset 488



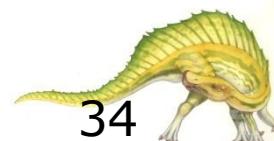


# Contiguous Allocation (cont.)



directory

<u>file</u>	<u>start</u>	<u>length</u>
A	0	2
B	14	3
C	19	6
D	28	4
E	6	2

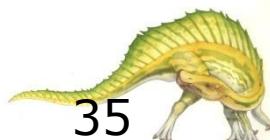




## Extent-based systems

---

- Extent-based systems allocate disk blocks in **extents**.
  - a modified contiguous allocation scheme.
- An **extent** is a contiguous block of disks.
  - Extents are allocated for file allocation.
  - A file consists of one or more extents.

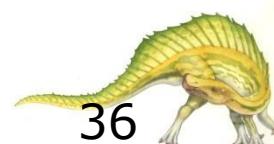


35



## Link Allocation

- Each file is a linked list of disk blocks.
  - Allocate as needed, link together; e.g., file starts at block 9.
  - Blocks may be scattered anywhere on the disk.
  - Solve external fragmentation and size declaration problem of contiguous allocation.
- Each block contains a pointer to next block.
  - If block size = 512 bytes, and a pointer requires 4 bytes, users can use only 508 bytes.
- No external fragmentation, and files can grow as long as there are free blocks.
- Mapping:  $Q = LA \text{ DIV } 508$ ,  $R = LA \text{ MOD } 508$ .
  - Block to be accessed is the  $Q^{\text{th}}$  block in the list.
  - Offset address:  $R$

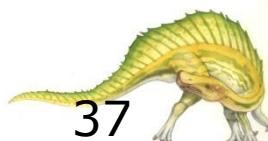




## Link Allocation (cont.)

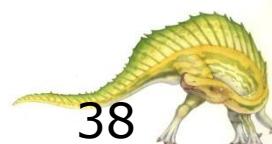
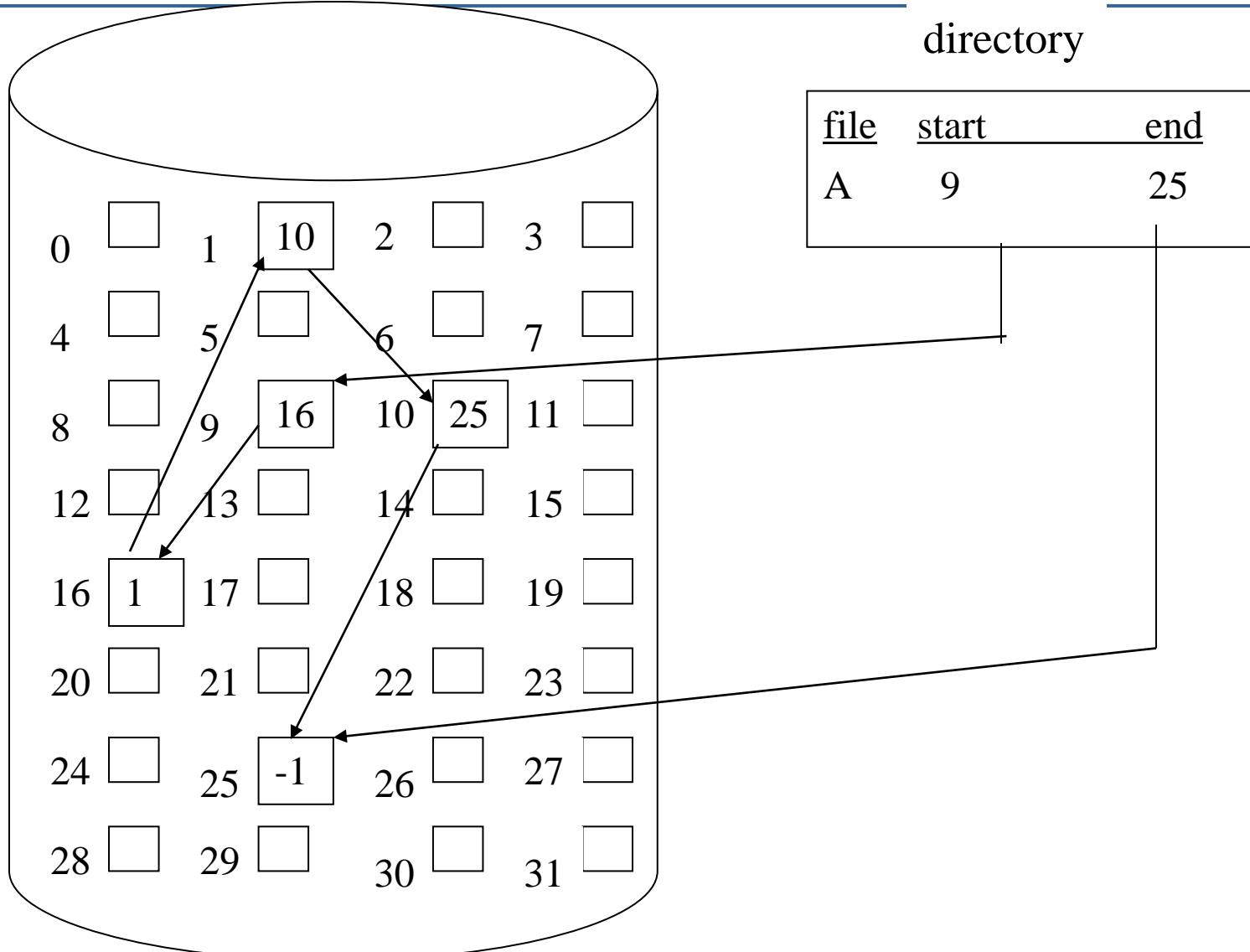
---

- Disadvantages:
  - Efficient only for sequential accesses.
    - ▶ For direct access it has to search from the beginning → not efficient.
  - Reliability issue: if a pointer is lost/damaged.
  - 4 of 512 bytes used for pointers.
    - ▶ Solution: use clusters of blocks; allocate clusters rather than blocks.
    - ▶ Problem: internal fragmentation.
- *File-allocation table* (FAT) - disk-space allocation used by MS-DOS and OS/2 is a variant of linked allocation.





# Link Allocation (cont.)





# File-Allocation Table

directory entry

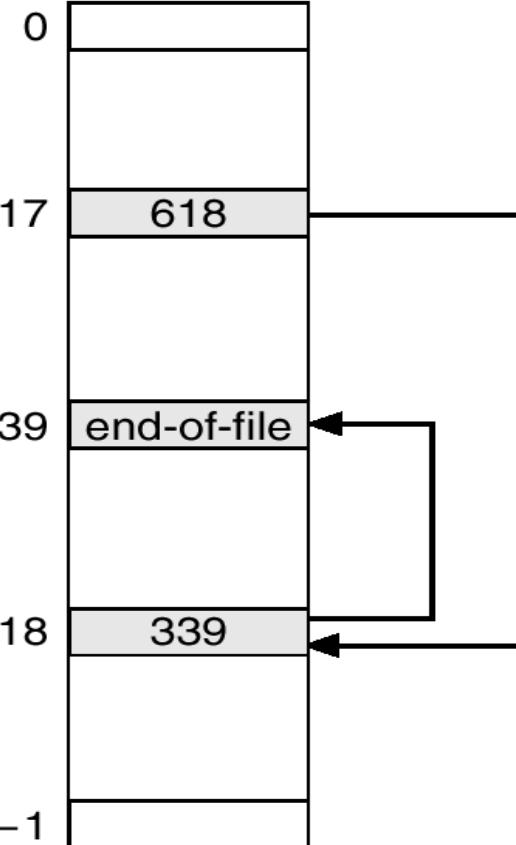


name

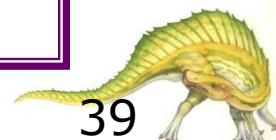
start block

no. of disk blocks

-1



FAT

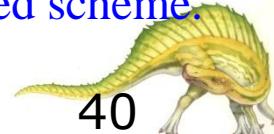


39



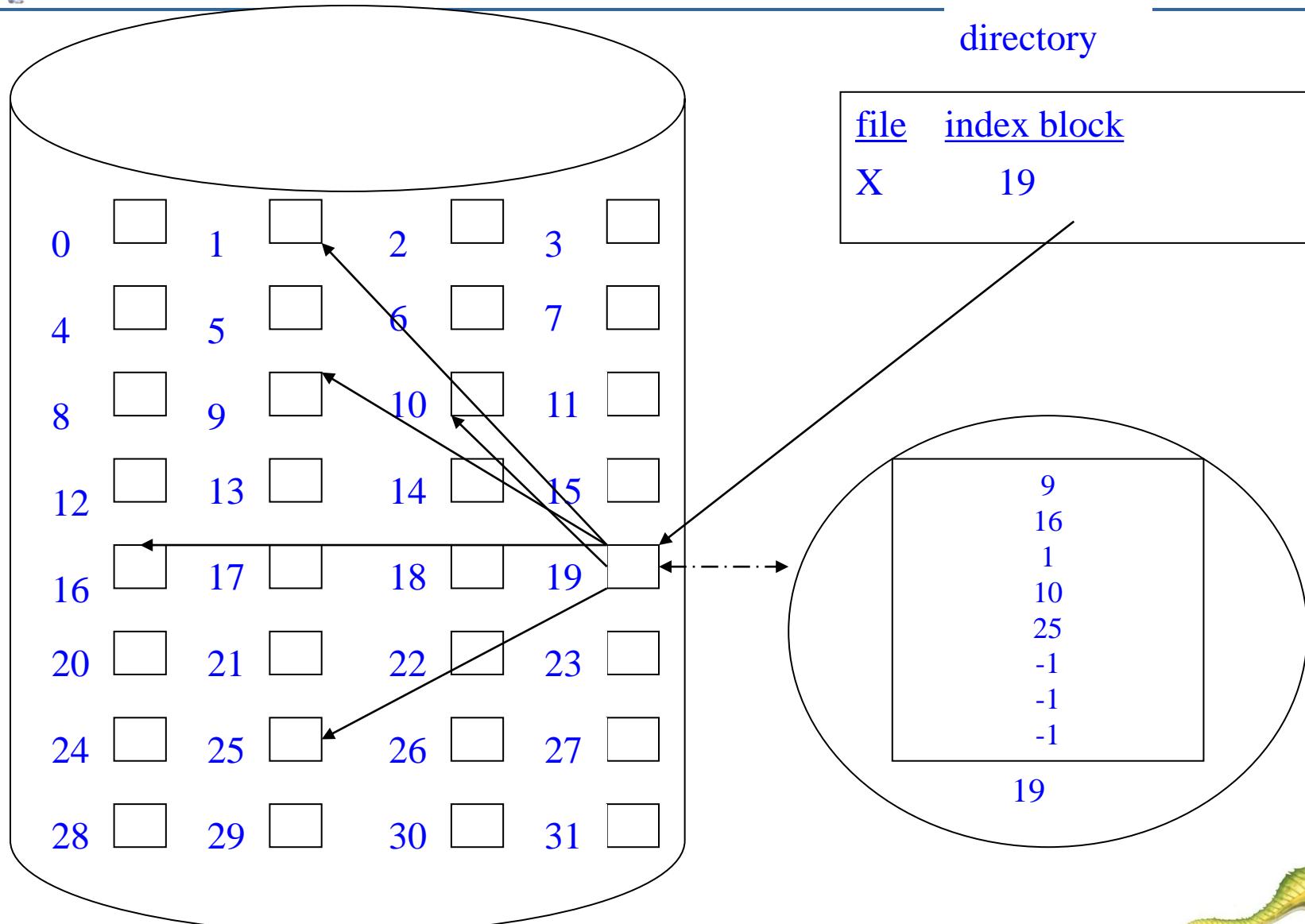
## Indexed Allocation

- Each file has its own index block – an array of disk block addresses.
  - It stores all pointers together in the *index block*.
  - $i$ -th entry in the index block points to the  $i$ -th block of the file.
  - The directory contains the address of the index block.
  - An index block is typically one disk block.
  - Support direct accesses.
- It supports dynamic access without external fragmentation, but have overhead of the index block.
  - Must determine how large an index block is.
    - Too small, it cannot support large files
    - Too large, it will waste space.
    - For a large file, use a linked scheme, a multilevel index, or a combined scheme.





# Indexed Allocation (cont.)





# Indexed Allocation Mapping

- Mapping from logical to physical in a file of **maximum size** of 64K bytes and block size of 512 bytes.

- Assume each pointer is 4 bytes → each index block contains 128 pointers

Q = displacement into index table

R = displacement into block

- Mapping from logical to physical in a file of **unbounded** length (block size of 512 bytes, 4 byte pointers).

- Linked scheme** – Link blocks of index table,
  - Each link block has 127 pointers, and thus can address up to  $127 * 512$  words.
  - e.g., 2 linked blocks: **max file size:**  $2 * 127 * 512$

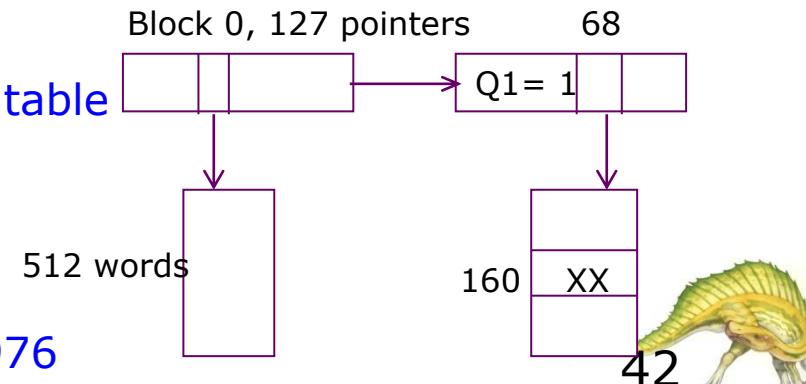
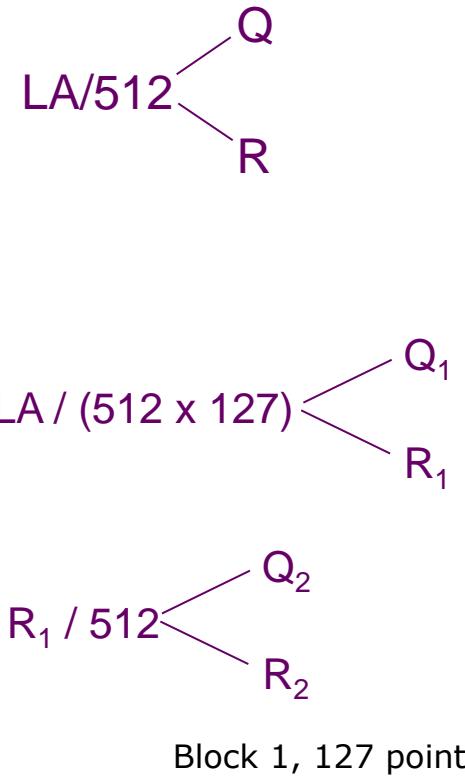
Q<sub>1</sub> = block of index table

Q<sub>2</sub> = displacement into block of index table

R<sub>2</sub> = displacement into block of file

- Example: LA = 100000

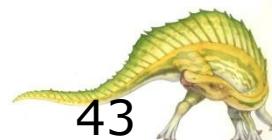
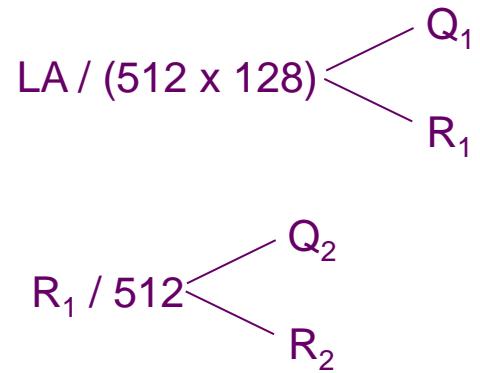
- Q<sub>1</sub> =  $100000 \text{ DIV } (512 * 127) = 1$
- R<sub>1</sub> =  $100000 \text{ MOD } (512 * 127) = 34976$
- Q<sub>2</sub> =  $34976 \text{ DIV } 512 = 68$





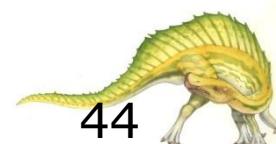
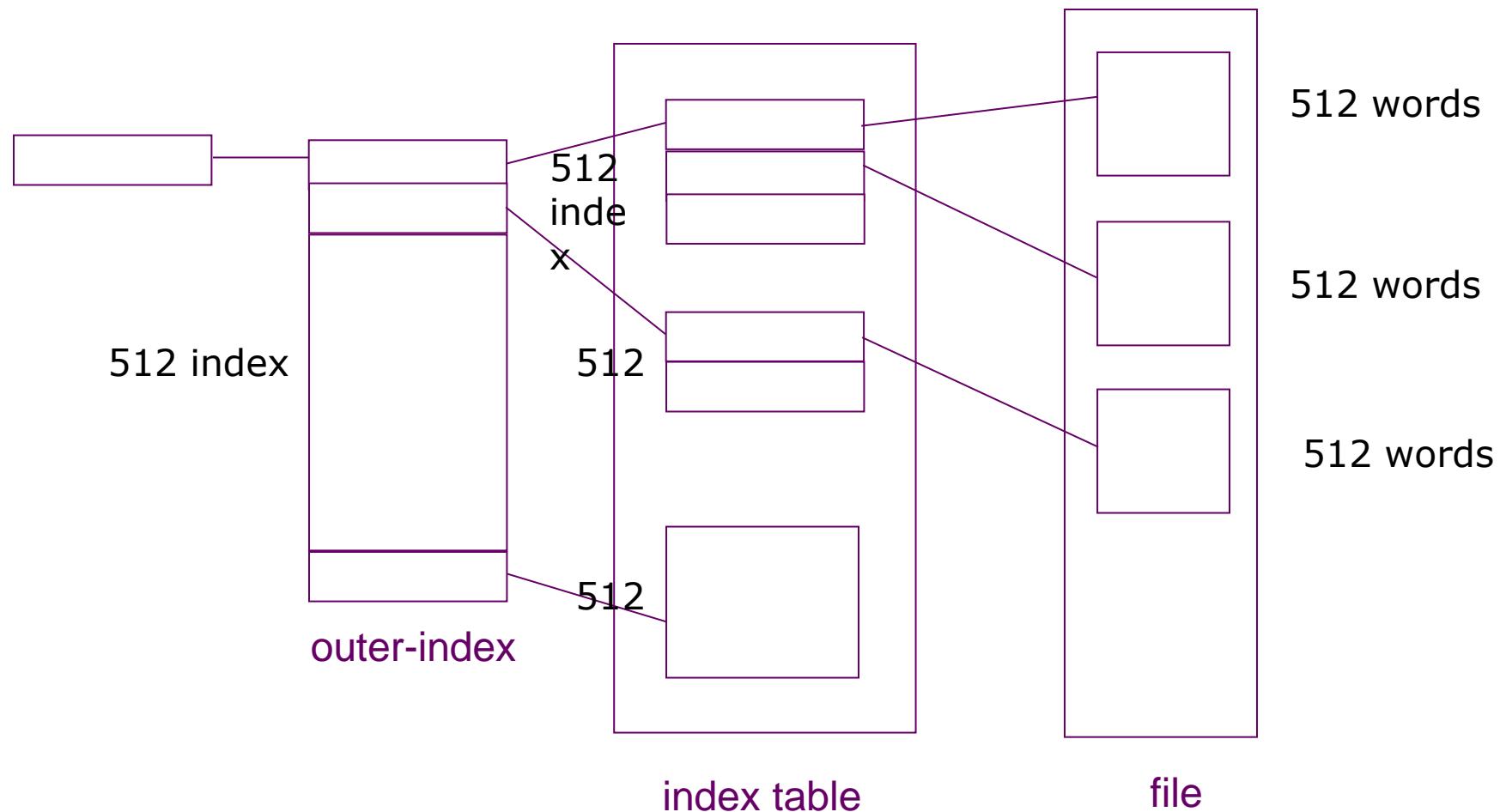
# Indexed Allocation Mapping (cont.)

- \* Mapping from logical to physical in a file of **unbounded length** (block size of 512 bytes, 128 4-byte pointers).
  - **Two-level index (maximum file size is  $128^2 * 512$  bytes)**
    - Q1 = displacement into outer-index
    - Q2 = displacement into block of index table
    - R2 = displacement into block of file
  
- \* Assume a system with block size = 4096 bytes = 4KB and 4 byte pointers
  - There are  $4096/4 = 1024$  pointers per index block
  - For two levels, there are  $1024^2$  pointers each of which can point to one data block of size 4KB
    - Thus, such configuration supports up to  $1024^2 * 4KB = 4GB$  of file.



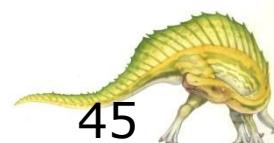
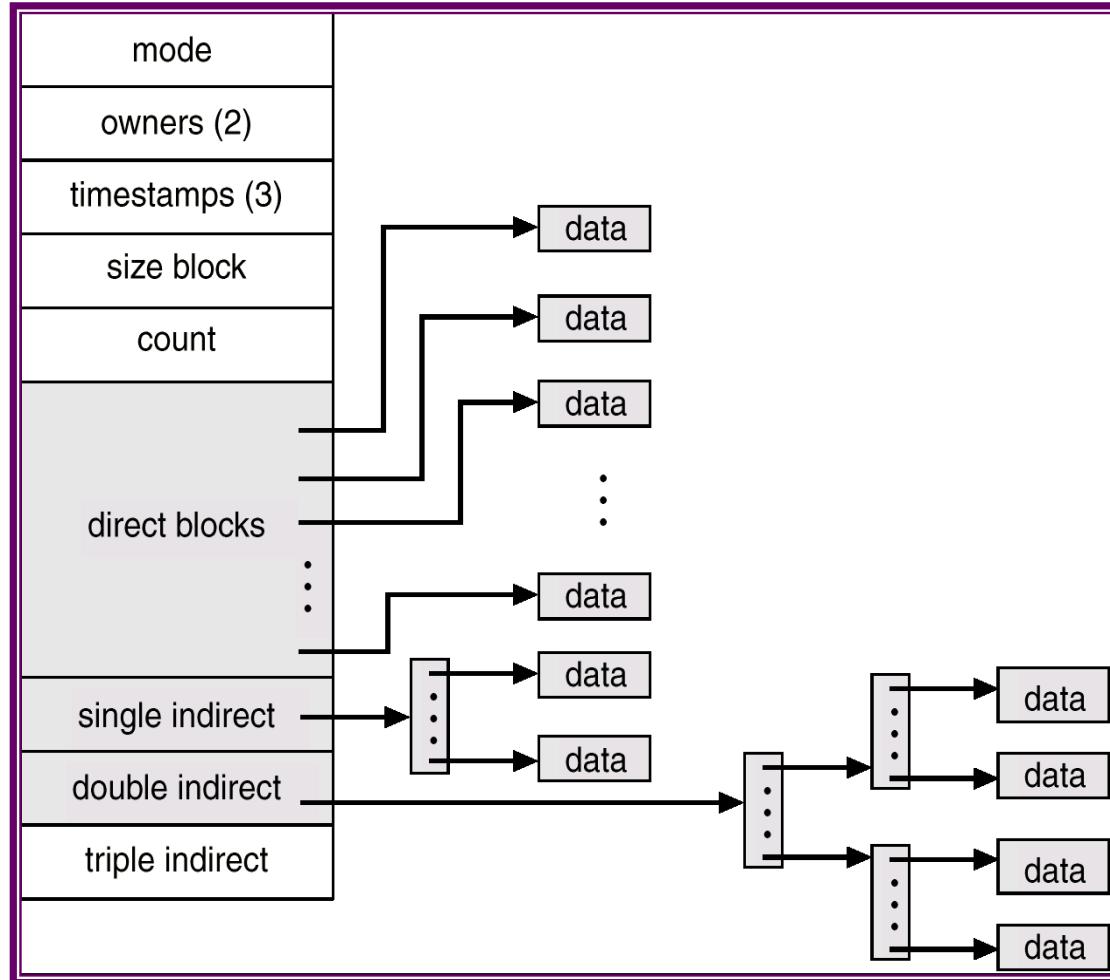


## Two level index





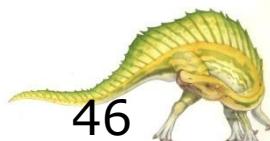
# Combined Scheme: UNIX (4K bytes per block)



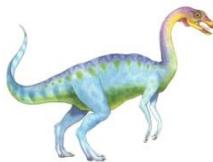


# Free-Space Management

- OS maintains a free space list to keep track of free disk space.
  - The space is limited → need to reuse the space from deleted files for new files, if possible.
- Several implementations:
  - Bit vector/ bit map.
  - Linked list.
  - Grouping.
  - Counting.



46



## Bit vector (n blocks)

- Each block is represented by 1 bit: free (1), allocated(0)

### Example:

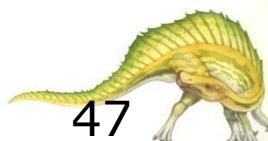
- Free blocks: 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27. The free space bit map: 0011110011111100011000000111

### Advantage:

- Simple and efficient to find the first free block, or  $n$  consecutive free blocks on the disk.
- Many computer supply bit-manipulation instruction for this purpose.

### Disadvantage:

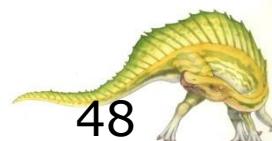
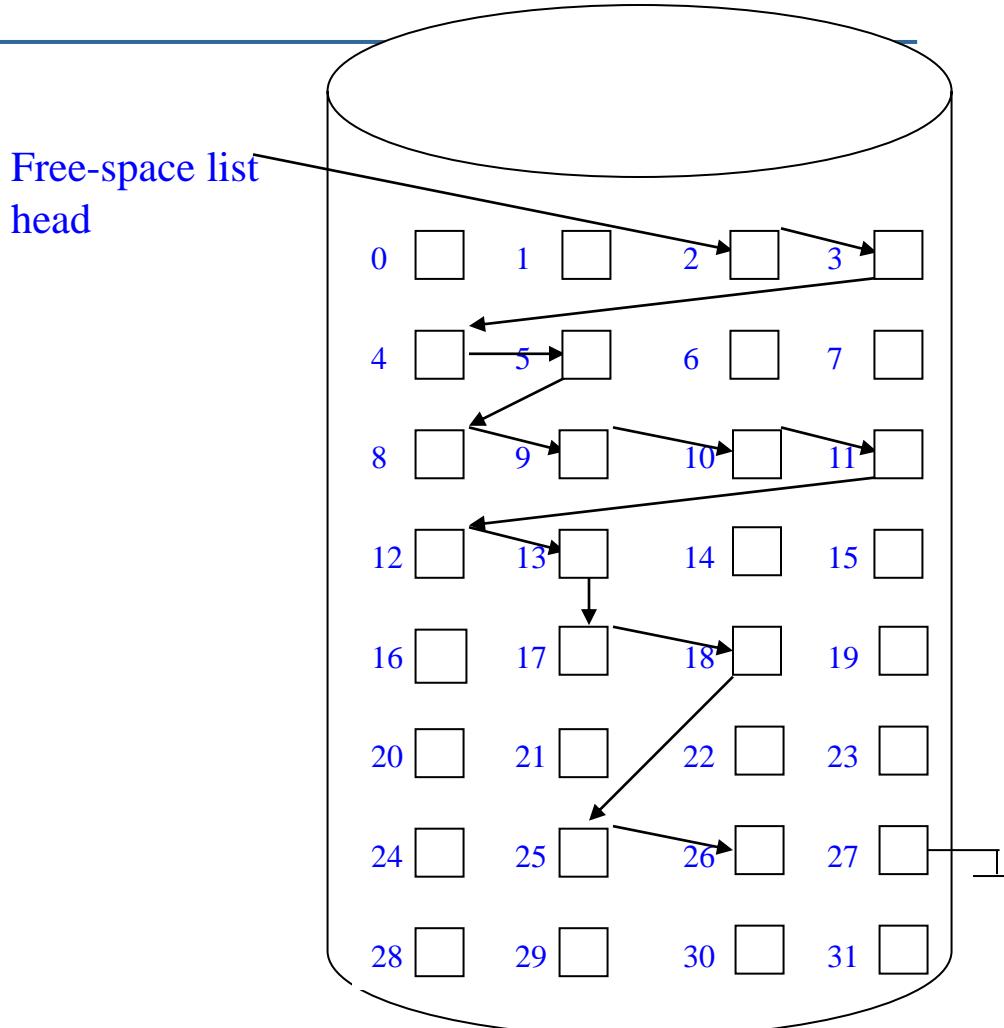
- Bit map requires extra space.





## Link List (free list)

- Link together all the free blocks, and keep a pointer to the first free block in a special location on the disk, and caching it in memory.
- The first block contains a pointer to next free disk block, and so on.
- Not efficient – to traverse the list, we must read each block – requires substantial I/O time.
- Cannot get contiguous space easily.





# Grouping & Counting

## Grouping

- A modified free list approach.
- Store the addresses of  $n$  free blocks in the first free block.
- The first  $n-1$  of these blocks are actually free, and the last block contains the addresses of another  $n$  free blocks, etc.

## Counting

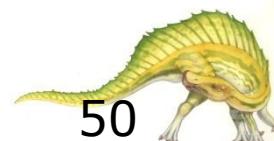
- Generally several contiguous blocks may be allocated or freed simultaneously.
- Keep the address of the first free block and the number  $n$  of free contiguous blocks that follow the first block.
- Each entry in the free space list consists of a disk address and a count.





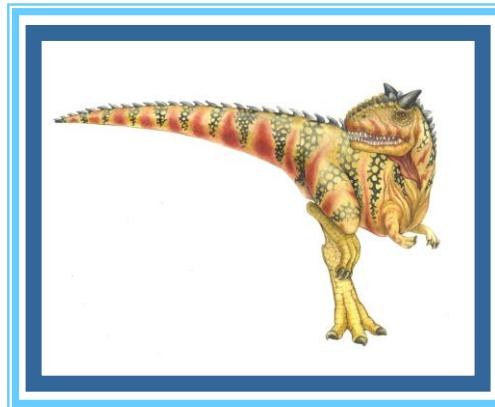
# Efficiency and Performance

- Disks tend to be the major bottleneck in system performance – they are the slowest main computer component.
- Efficiency depends on:
  - Disk allocation and directory algorithms.
  - Types of data kept in file's directory entry.
- Needs techniques to improve the efficiency and performance of disks.
- Performance
  - *Disk cache* – separate section of main memory for frequently used block.
  - *Free-behind* and *read-ahead* – techniques to optimize sequential access.



50

# Chapter 9: Virtual Memory





# Background

---

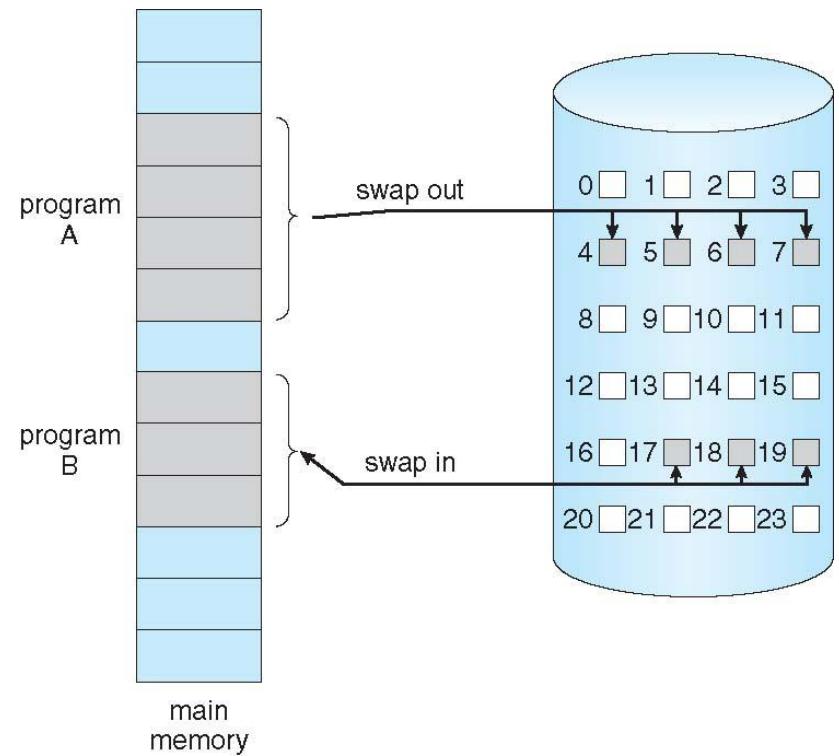
- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes

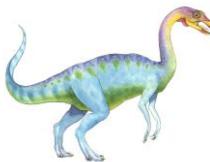




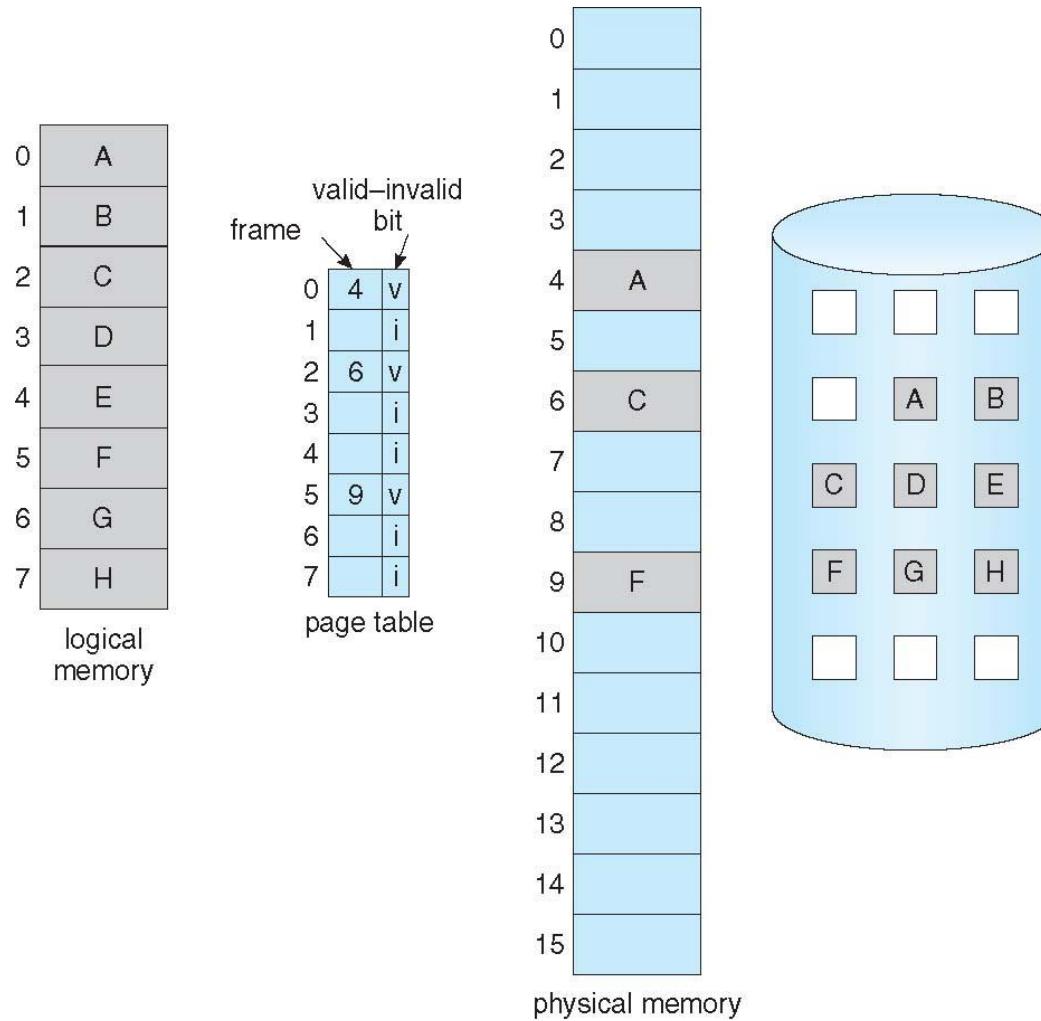
# Demand Paging

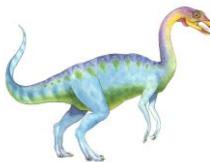
- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**





# Page Table When Some Pages Are Not in Main Memory





# Page Fault

---

- If there is a reference to a page, first reference to that page will trap to operating system:

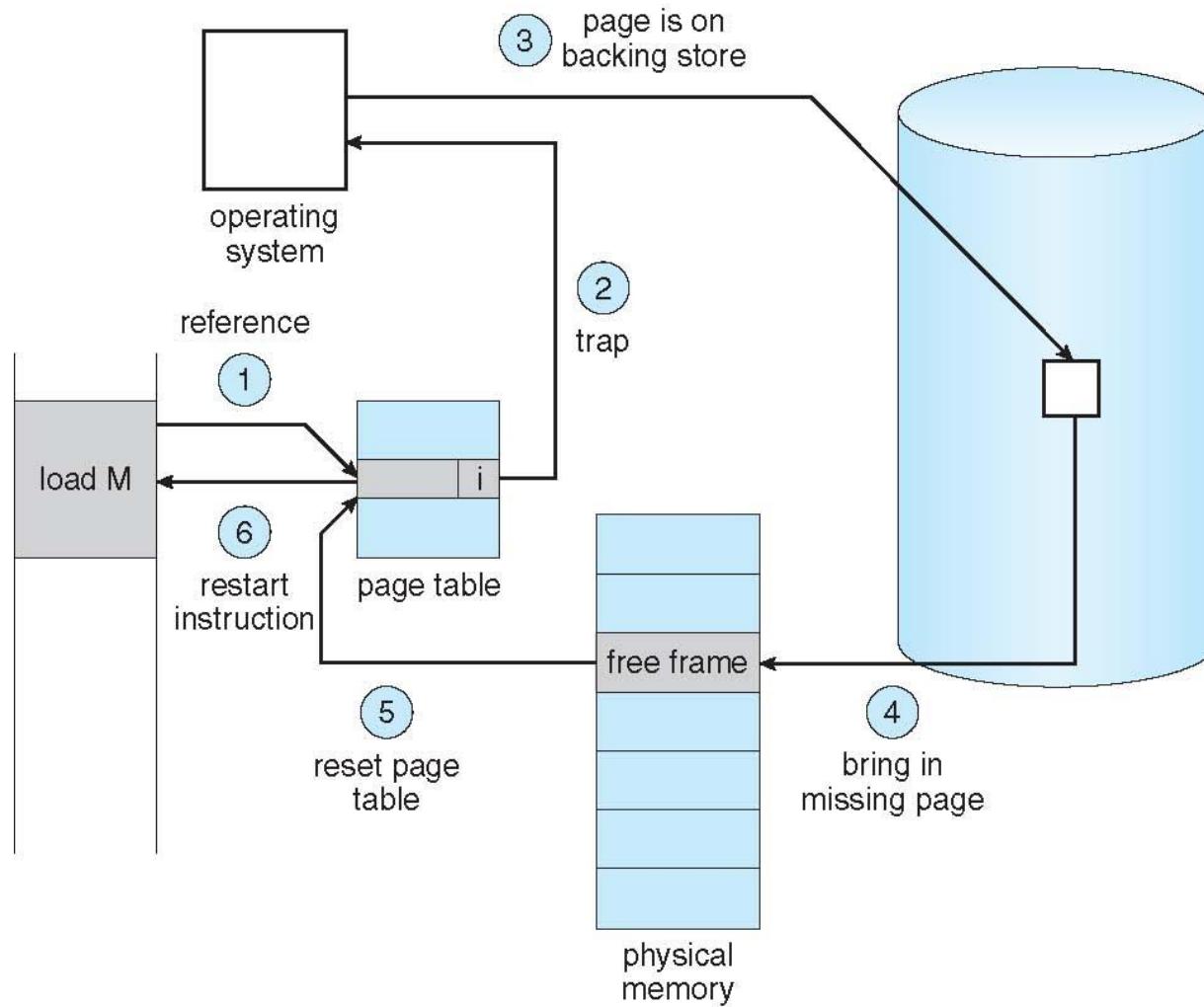
## page fault

1. Operating system looks at another table to decide:
  - Invalid reference  $\Rightarrow$  abort
  - Just not in memory
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory  
Set validation bit = **V**
5. Restart the instruction that caused the page fault





# Steps in Handling a Page Fault



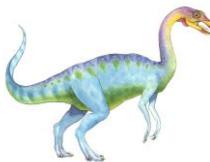


# Performance of Demand Paging

## □ Stages in Demand Paging (worse case)

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
  1. Wait in a queue for this device until the read request is serviced
  2. Wait for the device seek and/or latency time
  3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction





# Basic Page Replacement

---

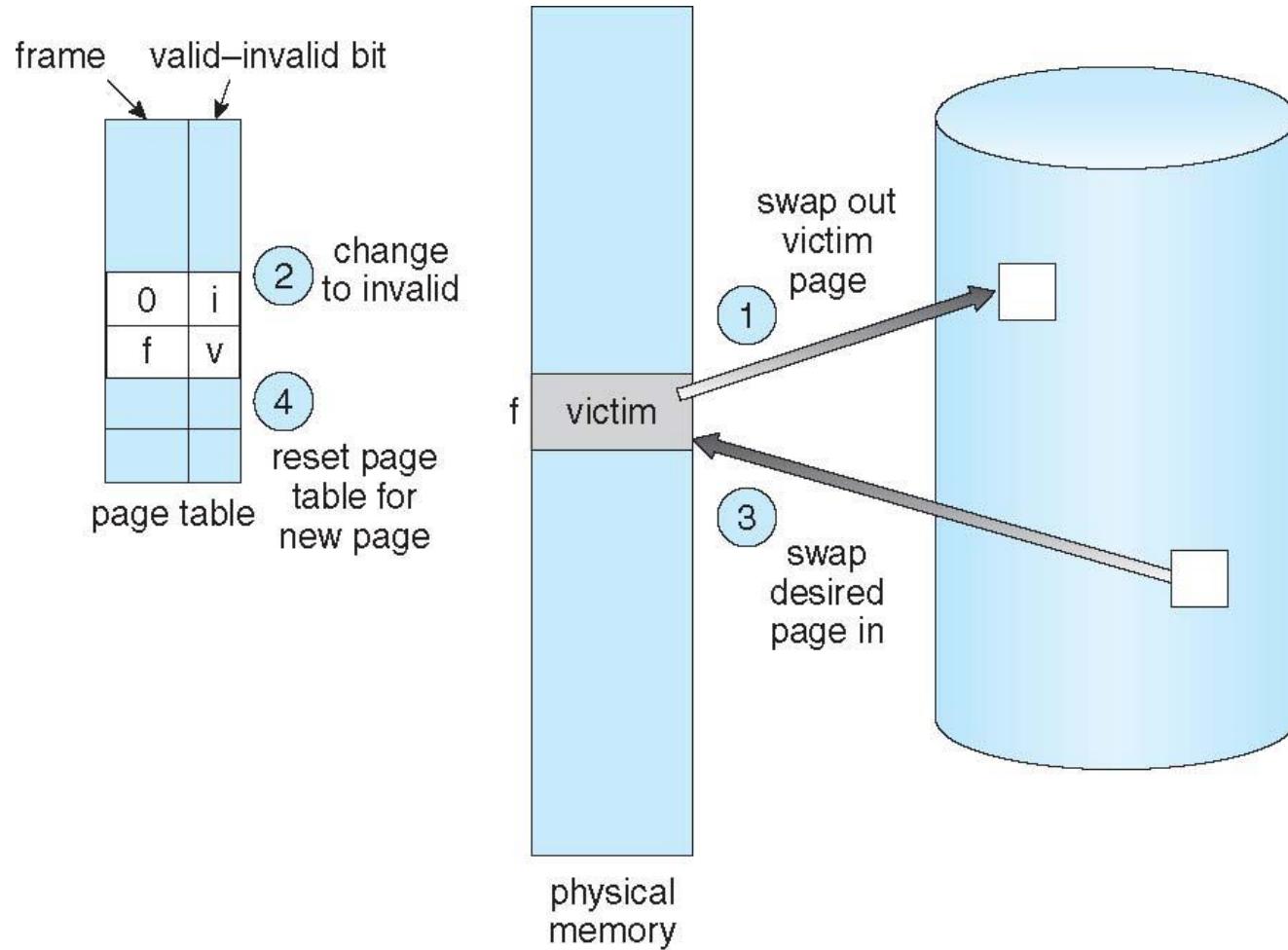
1. Find the location of the desired page on disk
2. Find a free frame:
  - If there is a free frame, use it
  - If there is no free frame, use a page replacement algorithm to select a **victim frame**
    - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

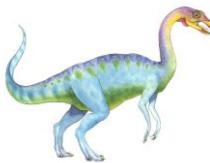
Note now potentially 2 page transfers for page fault – increasing EAT





# Page Replacement





# Page-replacement algorithms

- There are many page replacement algorithms
  - We want an algorithm that results in the lowest *page-fault rate*.
- Evaluate algorithm by running it on a particular string of memory references (reference string) and compute the number of page faults on that string.
- Reference string can be generated
  - by random number generator (artificial) or
  - by tracing a given system and recording the address of each memory reference.

## Example:

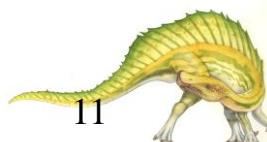
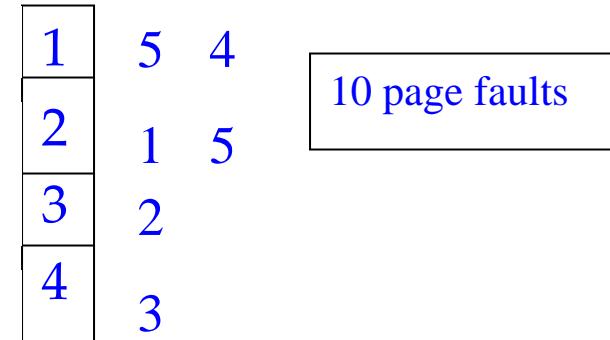
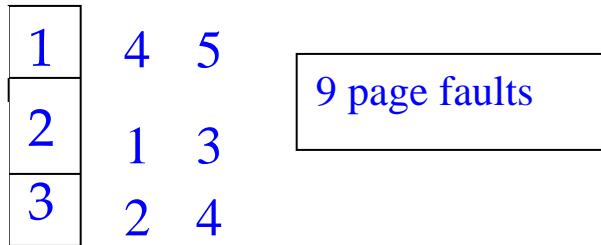
- Consider the following memory address sequence:  
  
0100, 0432, 0101, 0612, **0102**, **0103**, **0104**, **0101**, 0611, **0102**, **0103**,  
**0104**, **0101**, 0610, **0102**, **0103**, **0104**, **0101**, 0609, **0102**, **0105**.
- If system's page size = 100 bytes, we have the following page reference string: 1, 4, 1, 6, **1**, 6, **1**, 6, **1**, 6, **1**
- To determine the number of page faults for a particular reference string and page-replacement algorithm, we need to know the total number of frames available.

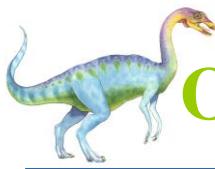




# First-in-first-out (FIFO) algorithm

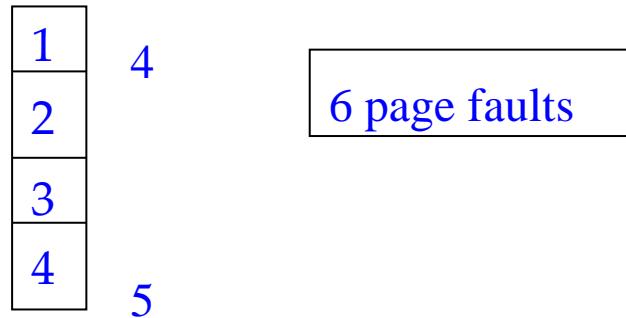
- The simplest page-replacement algorithm.
  - Use FIFO queue: replace the page at the head of the queue, and insert a new page at the tail.
- FIFO replacement suffers from Belady's anomaly
  - More frames result in more page faults – an anomaly!
- Examples: reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 or 4 frames (3 or 4 pages can be in memory at a time per process)



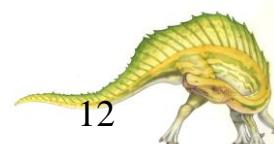


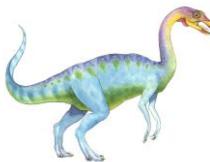
# Optimal page replacement algorithm (OPT)

- OPT replaces page that will not be used for the longest period of time
- For a fixed number of frames, OPT has the lowest page fault rate of all algorithms
  - It also never suffers from Belady's anomaly.
- 4 frames example: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



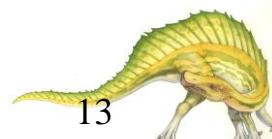
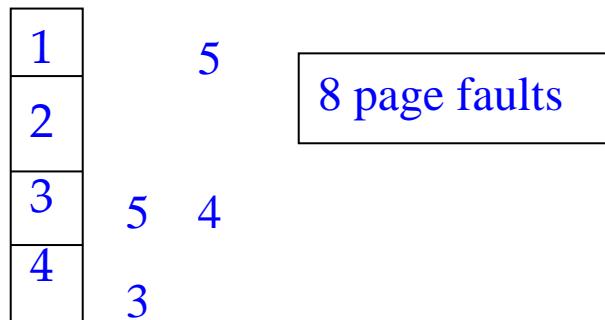
- \* **Problem:** It is difficult to implement
  - ☞ It requires future knowledge of the reference string!
- \* This algorithm is used mainly to measure how well a new algorithm performs





# Least recently used (LRU) algorithm

- LRU replaces the page that has not been used for the longest period of time.
  - Associate each page with its last use
  - It has good performance, and is often used
  - It does not suffer from Belady's anomaly
- A class of algorithms that do no suffer from Belady's anomaly are called *stack algorithms*.
  - In a stack algorithm, the set of pages in memory for  $n$  frames is always a subset of the set of pages that would be in memory with  $n+1$  frames.
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5





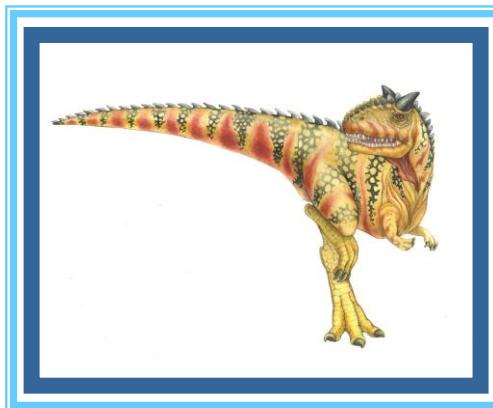
# Page-Buffering Algorithms

---

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected



# Chapter 14: Protection and Security



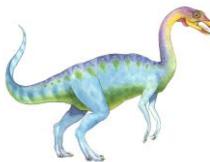


# Goals of Protection

---

- In one protection model, computer consists of a collection of objects, hardware or software
- Each object has a unique name and can be accessed through a well-defined set of operations
- Protection problem - ensure that each object is accessed correctly and only by those processes that are allowed to do so





# Principles of Protection

- Guiding principle – **principle of least privilege**
  - Programs, users and systems should be given just enough **privileges** to perform their tasks
  - Limits damage if entity has a bug, gets abused
  - Can be static (during life of system, during life of process)
  - Or dynamic (changed by process as needed) – **domain switching, privilege escalation**
  - “Need to know” a similar concept regarding access to data





# Access Matrix

- View protection as a matrix (**access matrix**)
- Rows represent domains
- Columns represent objects
- **Access**( $i$ ,  $j$ ) is the set of operations that a process executing in Domain $_i$  can invoke on Object $_j$

object domain	$F_1$	$F_2$	$F_3$	printer
$D_1$	read		read	
$D_2$				print
$D_3$		read	execute	
$D_4$	read write		read write	





# The Security Problem

- System **secure** if resources used and accessed as intended under all circumstances
  - Unachievable
- **Intruders (crackers)** attempt to breach security
- **Threat** is potential security violation
- **Attack** is attempt to breach security
- Attack can be accidental or malicious
- Easier to protect against accidental than malicious misuse





# Security Measure Levels

---

- Impossible to have absolute security, but make cost to perpetrator sufficiently high to deter most intruders
- Security must occur at four levels to be effective:
  - **Physical**
    - ▶ Data centers, servers, connected terminals
  - **Human**
    - ▶ Avoid **social engineering, phishing, dumpster diving**
  - **Operating System**
    - ▶ Protection mechanisms, debugging
  - **Network**
    - ▶ Intercepted communications, interruption, DOS
- Security is as weak as the weakest link in the chain
- But can too much security be a problem?





# Program Threats

---

- Many variations, many names
- **Trojan Horse**
  - Code segment that misuses its environment
  - Exploits mechanisms for allowing programs written by users to be executed by other users
  - **Spyware, pop-up browser windows, covert channels**
  - Up to 80% of spam delivered by spyware-infected systems
- **Trap Door**
  - Specific user identifier or password that circumvents normal security procedures
  - Could be included in a compiler
  - How to detect them?





# Program Threats (Cont.)

---

- **Logic Bomb**

- Program that initiates a security incident under certain circumstances

- **Stack and Buffer Overflow**

- Exploits a bug in a program (overflow either the stack or memory buffers)
  - Failure to check bounds on inputs, arguments
  - Write past arguments on the stack into the return address on stack
  - When routine returns from call, returns to hacked address
    - ▶ Pointed to code loaded onto stack that executes malicious code
  - Unauthorized user or privilege escalation

