

# **Chapter One**

# **Introduction to Operating Systems**

**Operating Systems**

# Objective

At the end of this session students will be able to:

- Understand the basic functions, concepts and principles of operating systems.
- Computer system structure
- Identify the different OS types: Mainframe (Servers), Desktop Systems, Real-time systems, Multiprocessor Systems (Shared memory multiprocessor, Clusters and Distributed Systems), Handheld Systems and others
- Understand the operating system services and concepts
- Understand system calls and their types
- Explain about operating system structures and operations
- Identify the different computing environments
- Understand about OS generations

# Introduction

- Without its software, a computer is basically a useless lump of metal.
- With its software, a computer can store, process, and retrieve information; play music and videos; send e-mail, search the Internet; and engage in many other valuable activities to earn its keep.
- Computer software can be divided roughly into two kinds:
  1. **System software**:- which manages the operation of the computer itself.
  2. **Application programs**:-which performs the actual work the user wants.
- The most important system program is the **operating system (OS)**, whose job is to control all the computer's resources and provide a base upon which the application programs can be written.
- The OS controls and coordinates the use of the hardware among the various system programs and application programs for a various users.

# What is an Operating System?

- A program that acts as an intermediary between a user of a computer and the computer hardware.
- Is a program that controls the execution of application programs.
- It controls and coordinates use of hardware among various applications and users.
- It simply provides an environment with which other programs can do useful work.
- OS performs basic tasks such as recognizing input from the keyboard, keeping track of files and directories on the disk, sending output to the display screen and controlling peripheral devices.
- Operating system goals:
  - ❖ Execute user programs and make solving user problems easier.
  - ❖ Make the computer system convenient to use.
  - ❖ Use the computer hardware in an efficient manner.

## Contd.

- It's a **resource allocator**
  - ❖ Decides between conflicting requests for efficient and fair resource use
  - ❖ Manages resources such as:
    - ✚ **Time management**:- CPU and disk transfer scheduling
    - ✚ **Space management**:- main and secondary storage allocation
    - ✚ **Synchronization and deadlock handling**:- IPC, critical section, coordination
    - ✚ **Accounting and status information**:- resource usage tracking
- It's a **control program**
  - ❖ Controls the execution of programs to prevent errors and improper use of the computer
- **User Environment**:- OS layer transforms bare hardware machine into higher level abstractions
  - ✚ **Execution environment**:- process management, file manipulation, interrupt handling, I/O operations, language.
  - ✚ **Error detection and handling**
  - ✚ **Protection and security**
  - ✚ **Fault tolerance and failure recovery**

## Contd.

The OS must support the following tasks:

1. Providing the facility to create and modify program and data using an editor.
  2. **Access to the compiler** for translating the user program from high-level language to machine language.
  3. Providing a **loader** program to move the compiled program code to the computer's memory for execution.
  4. Providing routines that handle the details of I/O programming.
- Operating system is the first layer of software loaded in to the computer working memory.
- It is a program that acts as an interface between the user, the computer software and the hardware resources.
- ❖ It provides a software platform on top of which other program can run
- “The one program running at all times on the computer” is the **kernel**. Everything else is either a system program (ships with the operating system) or an application program.

# Operating System Functions

- I. **Managing Hardware** :- OS interacts with hardware using drivers or BIOS.
- II. **Providing a user interface**: the OS provides a GUI(graphical user interface) so that users will be able to interact with the system easily.
- III. **Managing files**: it OS manages and controls the files and folders created.
- IV. **Running & managing applications**: The OS installs and runs all other PC software
  - ❖ Applications rely on the OS for support operations
  - ❖ Applications are typically tailored to a single OS
- V. **Security Management**
- VI. **Coordination of communication on the network**

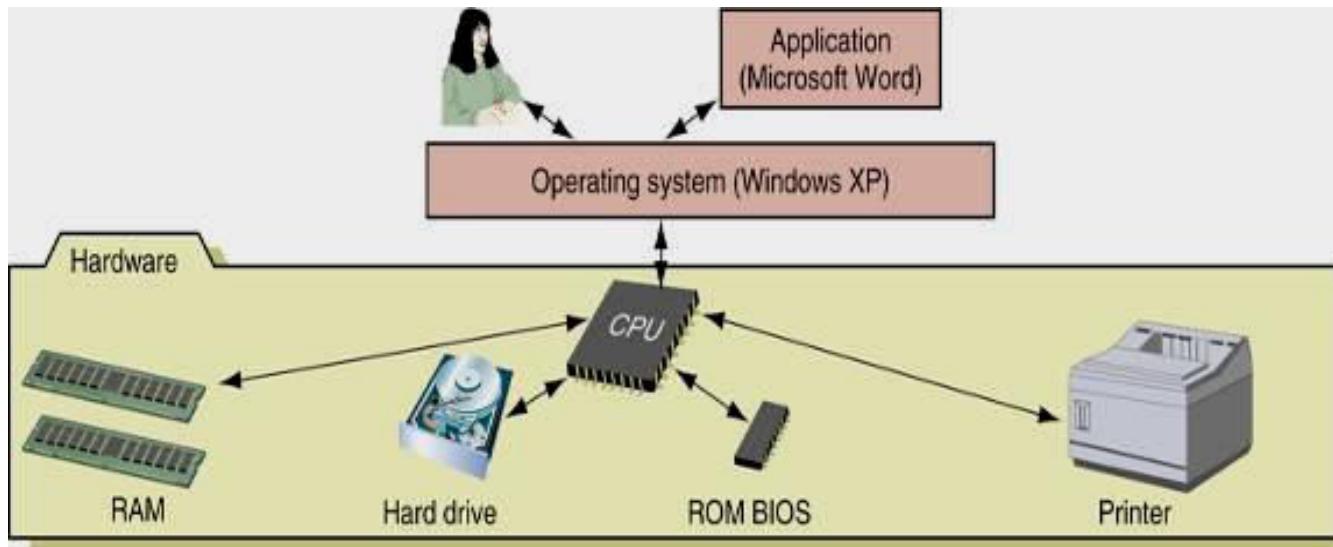
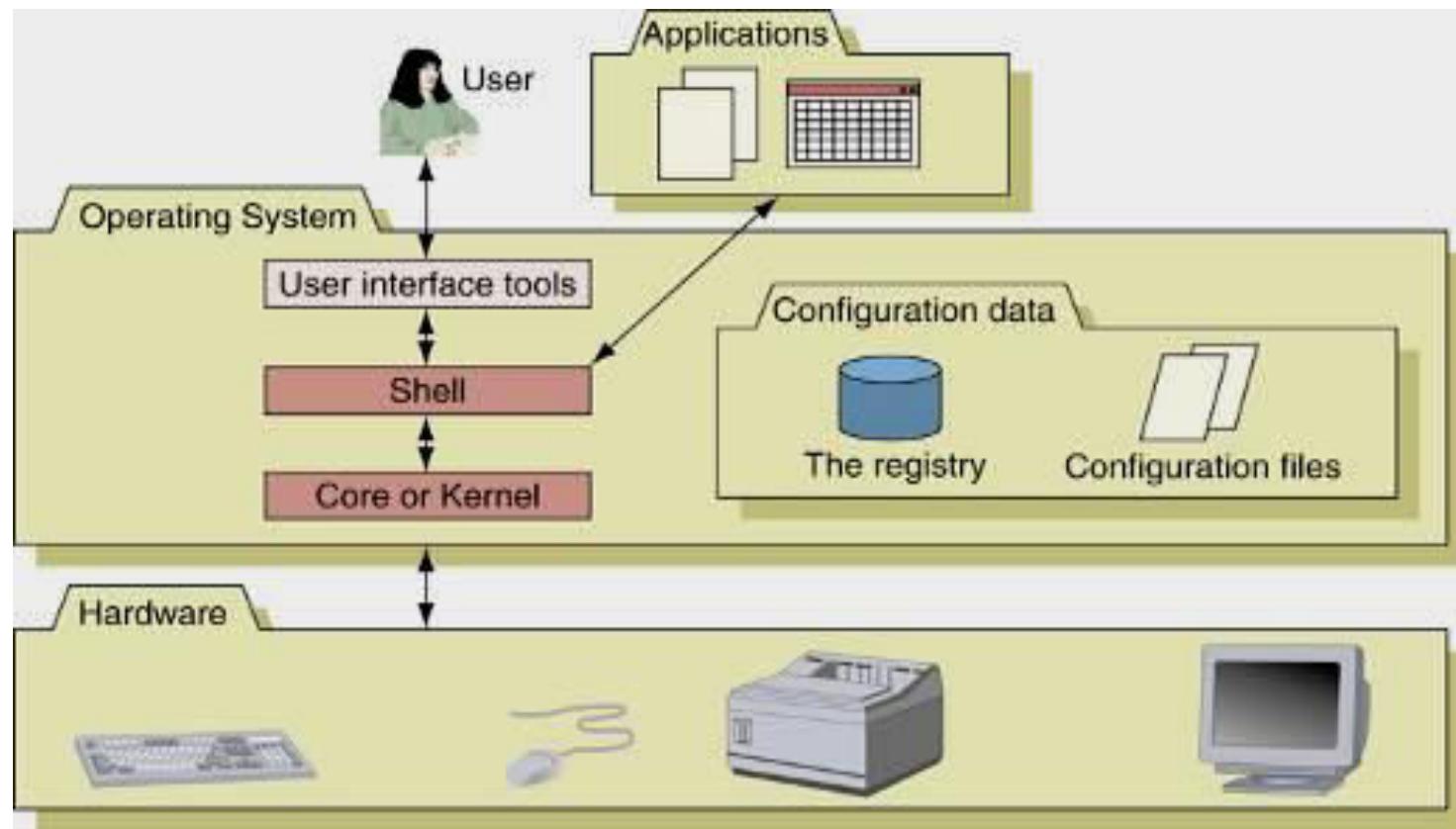


Figure 1-1 Users and applications depend on the OS to relate to all components

# OS Components

- All OSs have similar core components
  1. Shell
  2. Kernel (core)
- The shell exposes functions to users and applications
  - ❖ Piece of software that provides an interface for users using either command-line or graphical interface
  - ❖ Acts as an interface between the user and the kernel.
  - ❖ Normally reside on disks and are loaded into memory when needed.
- The kernel (nucleus) interacts with hardware devices
  - ❖ Allocates time & memory to programs and handles the file store & communication in response to system calls
  - ❖ Reside in memory all the time and is the hub of the OS

## Contd.



**Figure 1-2** Inside an operating system, different components perform various functions

## Common tasks performed by OS

1. Maintain a list of authorized users.
2. Maintain a list of all resources in the system.
3. Maintain current status of all users currently using the system (They are called active users of the system).
4. Maintain current status of all programs being executed by active users and ensure that they receive adequate attention of the CPU.
5. Maintain current status of all resources in the system and allocate resources to programs when requested.
6. Handle the requests made by users and their programs.

# Computer System Components

- **Hardware**:- provides basic computing resources (CPU, memory, I/O devices).
- **Operating system**:- controls and coordinates the use of the hardware among the various application programs for the various users.
- **Applications programs**:- define the ways in which the system resources are used to solve the computing problems of the users (compilers, database systems, video games, business programs).
- **Users** (people, machines, other computers).

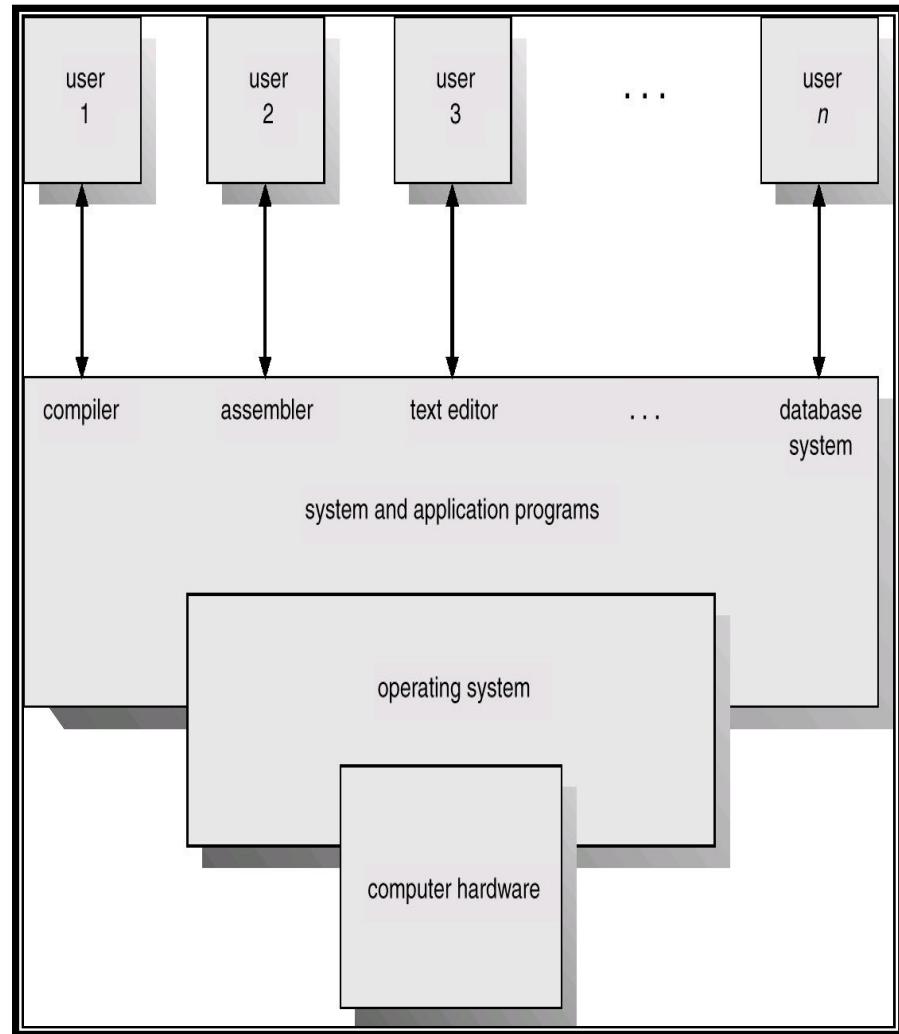


Figure 1-3 Abstract View of System Components

# Resource Allocation & Related Functions of OS

- The resource allocation function performs binding of one or more resources with a requesting program.
- It also deallocates resources from a program and allocates them to other programs.
- There are two popular strategies for resource allocation:

**A. Partitioning of resource:-** the OS decides a priori what resources should be allocated to user programs.

- ❖ This approach is called **static allocation** because the allocation is made before the execution of the program begins.
- ❖ It is **simple** to implement but it suffers from a **lack of flexibility**.
- ❖ In this approach, there is a **resource wastage** because allocation is made on the basis of perceived need of a program rather than its actual needs.

## Contd.

**B. Pool-Based Approach:-** the OS maintains a common pool of resources and allocates from this pool whenever a program requests a resources.

- ❖ This approach is called **dynamic allocation** because the allocation takes place during **execution of a program**.
- ❖ Dynamic allocation can lead to better utilization of resources because resources are not wasted.
- ❖ In the partition resource allocation approach, the OS considers the number of resources and the number of programs in the system and decides how many resources of each kind would be allocated to a program.
- ❖ But in case of pool-based approach, OS consults the resource table (a table which contains the name and address of a resource unit ad its present status, i.e. whether it is free or allocated to some program) when a program makes a request for a resource, allocates the resource if it is free.

## Contd.

- ▶ Resources can be shared by a set of programs in two ways:
  - Sequential sharing:- a resource is allocated for exclusive use by a program.

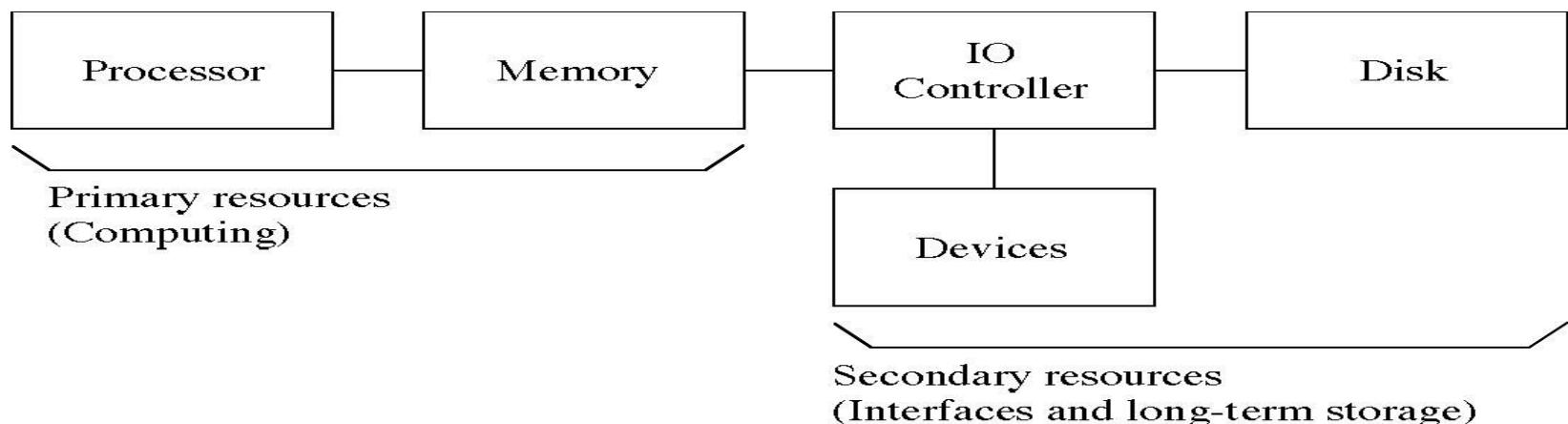
When the resource is deallocated, it is marked as *free* in the resource table so that it will be allocated to another program.
  - Concurrent sharing:- two or more programs can concurrently use the same resource.
    - Data file like bus time tables are concurrently shared resources.
    - But other resources are shared sequentially.

# Resource Preemption.

- ❖ The OS can deallocate a resource when the program to which it is allocated either **terminates** or makes an **explicit request for deallocation**.
- ❖ Alternatively, it can deallocate a resource by **force**. This actions is called **resource preemption**.
  - ✚ System resources may be preempted by an OS to enforce fairness in their use by programs, or to realize certain system-level goals.
  - ✚ A preempted program can not execute unless the preempted resource unit, or some resource unit of the same resource class, is allocated to it.
- ❖ The CPU can be shared only in sequential manner where as the memory can be shared in both sequential as well as pool-based manner.

# Hardware Resources

- ❖ **Processor:** execute instructions
- ❖ **Memory:** store programs and data
- ❖ **Input/output (I/O)controllers:** transfer to and from devices
- ❖ **Disk devices:** long-term storage
- ❖ **Other devices:** conversion between internal and external data representations



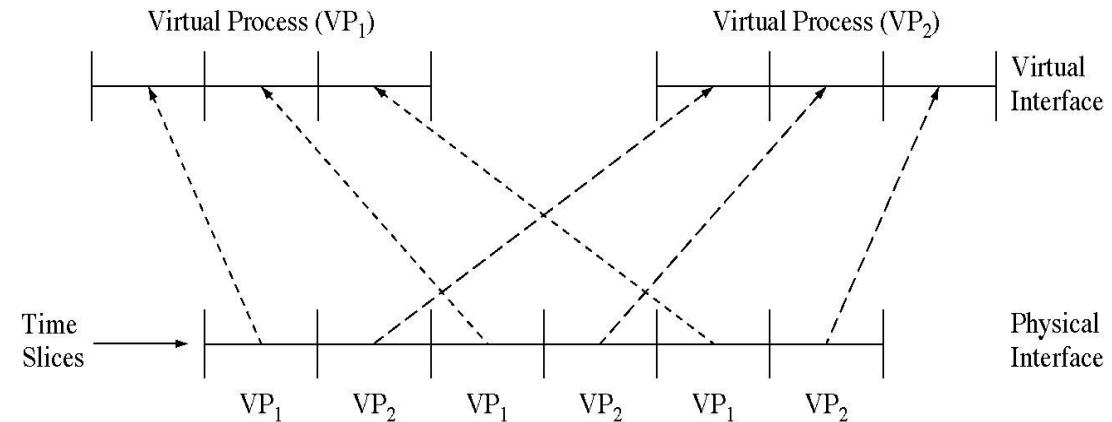
# Resource Management Functions

- ❖ Transforming physical resources to logical resources
  - Making the resources easier to use
- ❖ Multiplexing one physical resource to several logical resources
  - Creating multiple, logical copies of resources
- ❖ Scheduling physical and logical resources
  - Deciding who gets to use the resources

# Types of multiplexing

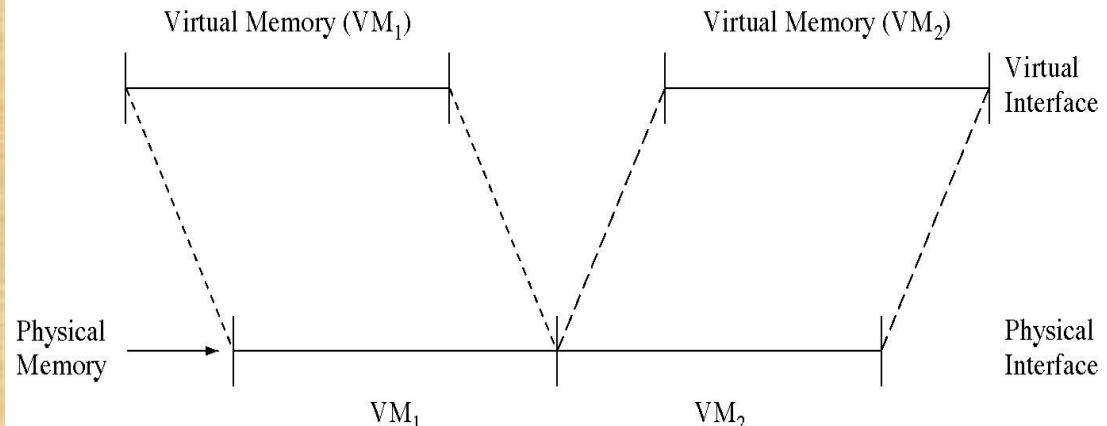
## 1. Time multiplexing

- time-sharing
- scheduling a serially-reusable resource among several users

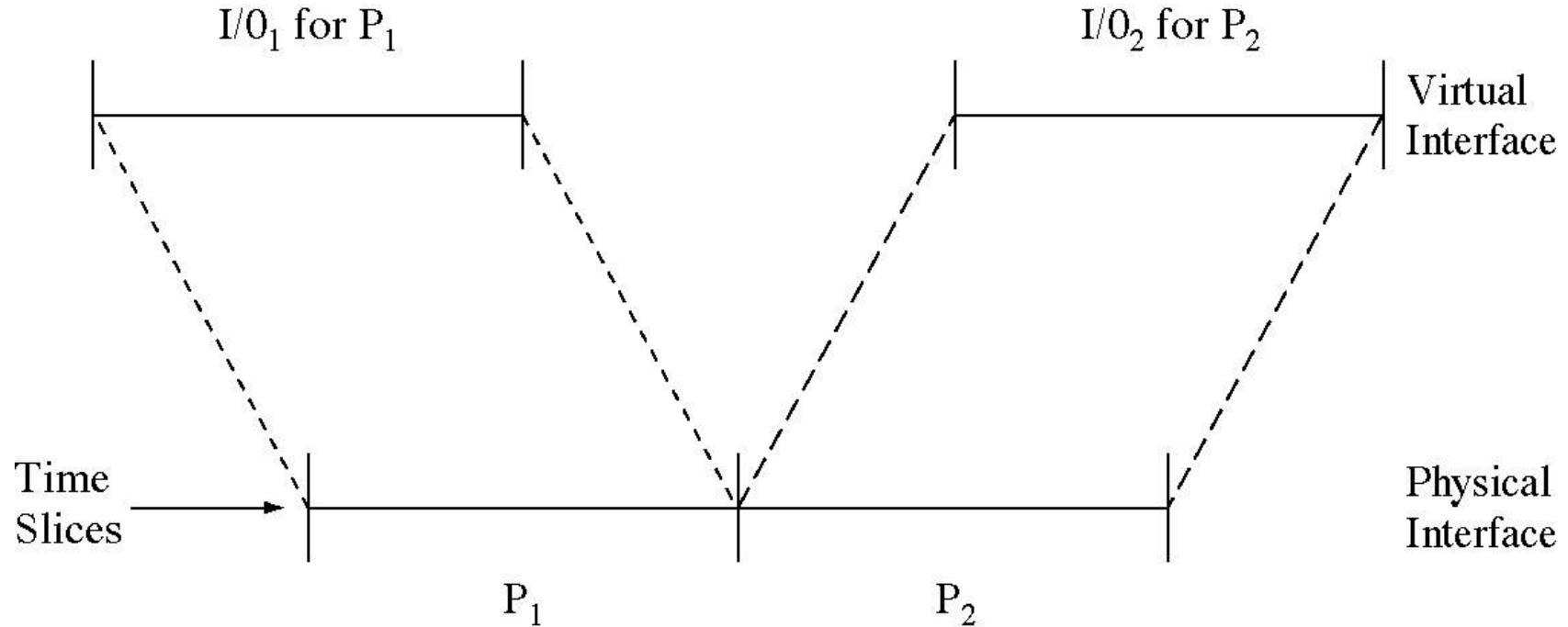


## 2. Space multiplexing

- space-sharing
- dividing a multiple-use resource up among several users



# Time-multiplexing I/O devices



## Examples of where you will find OSs

1. Servers (Mainframe Computers):- specialized for processing large volumes of requests for a given service.
  - Generally has a large number of necessary resources.
  - web server, file server, print server, time-sharing systems for interactive use, database management systems.
  - Reduce setup time by batching similar jobs
  - Automatic job sequencing – automatically transfers control from one job to another. First rudimentary operating system.
- ▶ Resident monitor
  - ▶ initial control in monitor
  - ▶ control transfers to job
  - ▶ when job completes control transfers back to monitor

## Contd.

2. **Desktops:-** specialized for single-user (or limited number of users).
  - ✚ Generally has a GUI that is integral to system software
  - ✚ designed to minimize response time to user requests
  - ✚ User convenience and responsiveness.
  - ✚ Can adopt technology developed for larger operating system' often individuals have sole use of computer and do not need advanced CPU utilization of protection features.
  - ✚ May run several different types of operating systems (Windows, MacOS, UNIX, Linux).

## Contd.

3. **Embedded Systems**:- specialized for managing a limited number of resources such as power, memory, I/O speed.
  - ✓ PDAs, cell phones, smart cards, automobiles.
  - ▶ Issues:
    - ▶ Limited memory
    - ▶ Slow processors
    - ▶ Small display screens.
4. **Clustered (Message-passing multicomputer)Systems**:- are interconnected by a high-speed interconnection fabric. Each processor has its own local memory and they communicate using the interconnect.
  - ▶ May share secondary storage.
  - ▶ Loosely coupled.

## Contd.

- ▶ Clustering can be:
  - ▶ Asymmetric clustering: one server runs the application while other servers standby.
  - ▶ Symmetric clustering: all N hosts are running the application.
- 5. **Real-time Systems:-** Specialized systems that must meet stringent performance and behavior requirements.
  - ✚ The goal is to bound worst-case behavior
  - ✚ Medical devices, avionic systems, multi-media players, routers.
  - ✚ Were originally used to control autonomous systems such as **satellites**, **robots** and **communication systems**.
  - ✚ RTOS is one that must react to inputs and responds to them quickly.

## Contd.

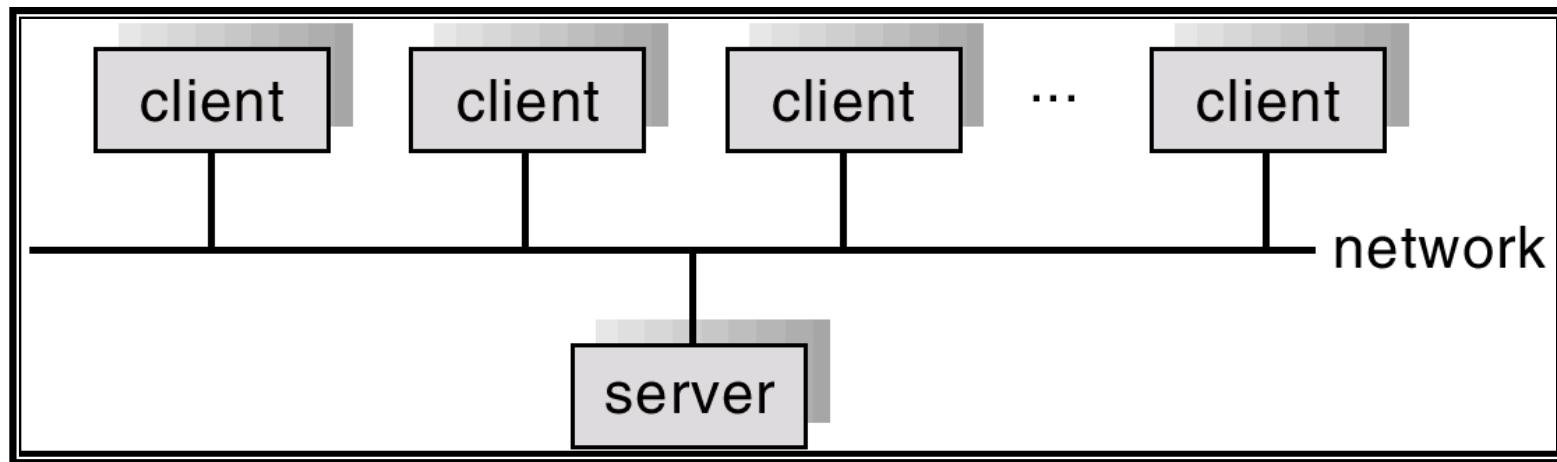
- ⊕ Often used as a control device in a dedicated application such as controlling scientific experiments, medical imaging systems, industrial control systems, and some display systems.
  - ⊕ Well-defined fixed-time constraints.
  - ⊕ Real-Time systems may be either hard or soft real-time.
1. **Hard real-time:-** Secondary storage limited or absent, data stored in short term memory, or read-only memory (ROM)
    - Conflicts with time-sharing systems, not supported by general-purpose operating systems.
  2. **Soft real-time:-** Limited utility in industrial control of robotics
    - Useful in applications (multimedia, virtual reality) requiring advanced operating-system features.

## 6. Distributed Systems

- Distribute the computation among several physical processors.
- Distributed OS runs on and controls the resources of multiple machines.
- Distributed OS owns the whole network and makes it look like a virtual uniprocessor or may be virtual multiprocessor.
- *Loosely coupled system* – each processor has its own local memory, processors communicate with one another through various communications lines, such as high-speed buses or telephone lines.
- Advantages of distributed systems.
  - Resources Sharing
  - Computation speed up – load sharing
  - Reliability
  - Communications

## Contd.

- Requires networking infrastructure.
- Local area networks (LAN) or Wide area networks (WAN)
- Each node is separate, standalone computer and the message delay times over a WAN can reach 100's of msecs.
- May be either client-server or peer-to-peer systems.



General Structure of Client-Server

## 7. Parallel Systems

- Multiprocessor systems with more than one CPU in close communication.
- *Tightly coupled system* – processors share memory and a clock; communication usually takes place through the shared memory.
- Advantages of parallel system:
  - Increased *throughput*
  - Economical
  - Increased reliability
    - ▶ graceful degradation
    - ▶ fail-soft systems

## Contd.

### A. *Symmetric multiprocessing (SMP)*

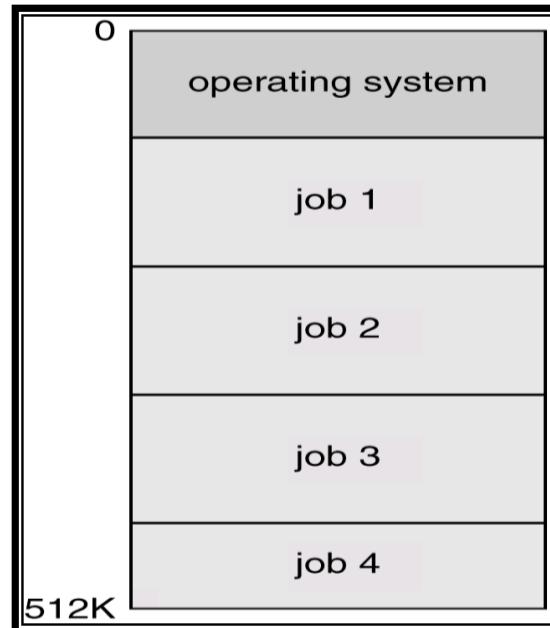
- Each processor runs an identical copy of the operating system.
- Many processes can run at once without performance deterioration.
- Most modern operating systems support SMP

### B. *Asymmetric multiprocessing*

- Each processor is assigned a specific task; master processor schedules and allocates work to slave processors.
- More common in extremely large systems

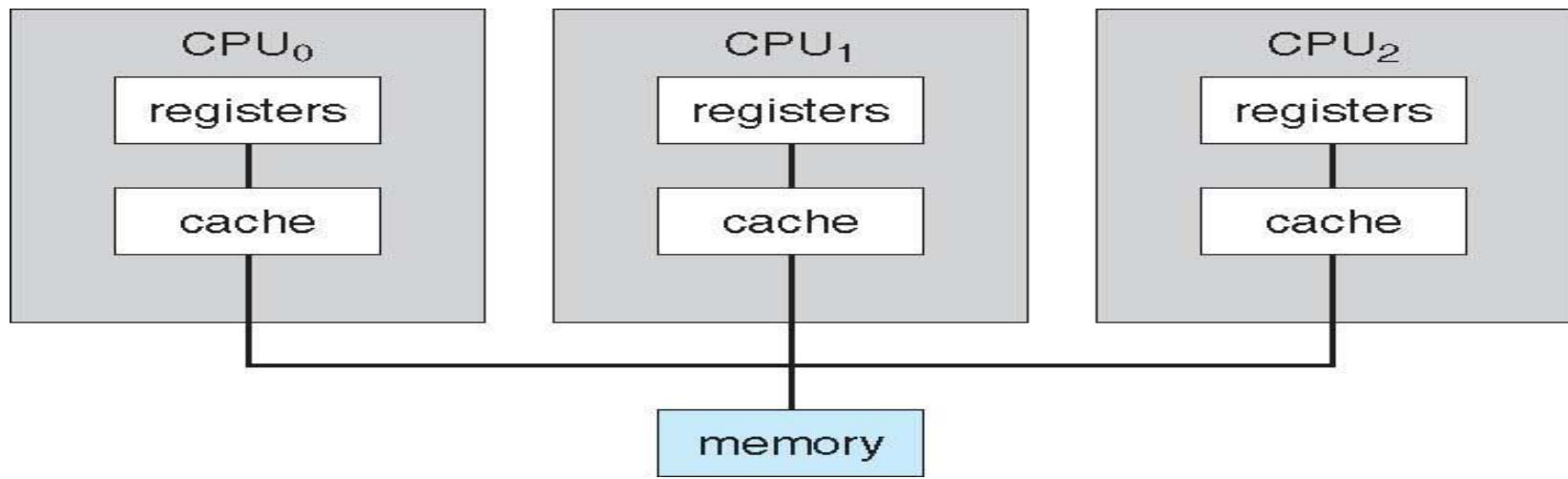
## 8. Multiprogrammed Batch System

- Several jobs are kept in main memory at the same time, and the CPU is multiplexed among them.
- It assumes a single processor that is being shared
- It increases CPU utilization by organizing jobs so that the CPU always has one to execute.



# OS Features Needed for Multiprogramming

- I/O routine supplied by the system.
- Memory management – the system must allocate the memory to several jobs.
- CPU scheduling – the system must choose among several jobs ready to run.
- Allocation of devices.



## 9. Time-Sharing Systems–Interactive Computing

- Time-sharing systems supports interactive users.
- It is also called **multitasking** and it is a logical extension of multiprogramming.
- The CPU is multiplexed among several jobs that are kept in memory and on disk (the CPU is allocated to a job only if the job is in memory).
- Time-sharing systems uses CPU **scheduling** and **multiprogramming** to provide an economical interactive system of two or more users.
- Here each user is given a **time-slice** for executing his job in round robin fashion. Job continues until the time-slice ends.
- On-line communication between the user and the system is provided; when the operating system finishes the execution of one command, it seeks the next “control statement” from the user’s keyboard.
- On-line system must be available for users to access data and code.

# Operating-System Operations

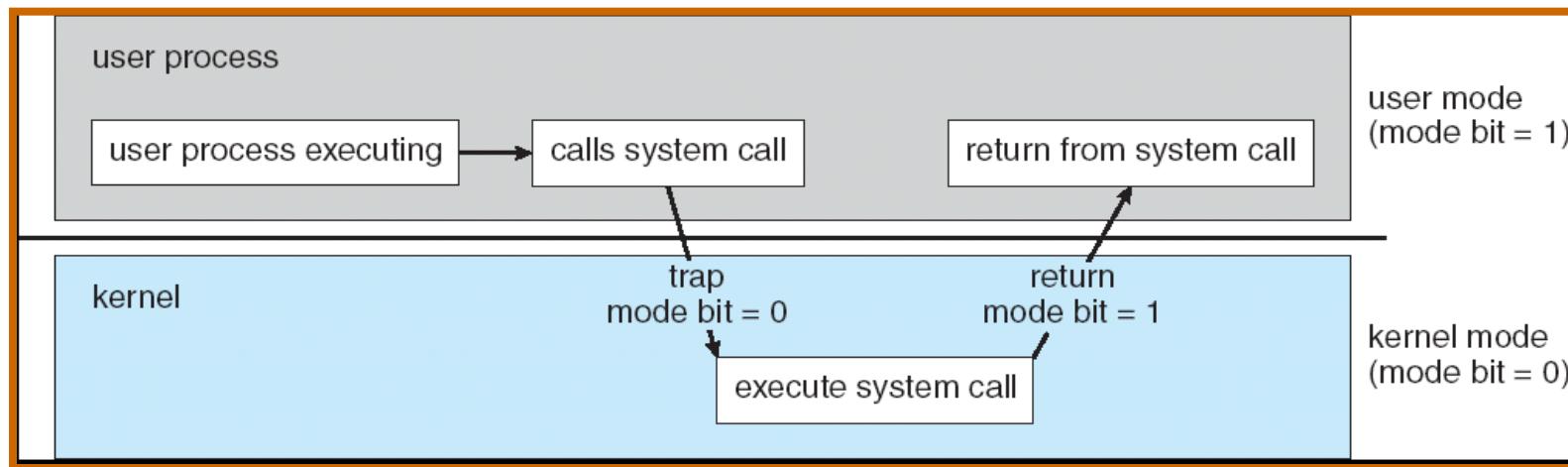
- ❖ Modern OS are Interrupt-driven by hardware.
  - ✿ If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly waiting for something to happen.
  - ✿ Events are almost always signaled by the occurrence of an **interrupt** or a **trap**
- ❖ A **trap** (or an **Exception**) is a software-generated interrupt caused either by an error(for division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed.
  - ✿ Other process problems include infinite loop, processes modifying each other or the operating system.
- ❖ **The interrupt-driven nature of an operating system defines that system's general structure**
- ❖ A properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other program to execute incorrectly

# Dual-Mode Operation

- ❖ In order to ensure the proper execution of the operating system, we must be able to distinguish between the **execution of operating-system code** and **user-defined code**.
- ❖ The approach taken by most computer systems is to provide **hardware support** that allows us to differentiate among various modes of execution.
- ❖ Two separate modes of operations are there namely:
  1. **User Mode**
  2. **Kernel (supervisor or system or Privileged)mode**
- ❖ A bit, called the **mode bit** is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1).
- ❖ With the mode bit it is possible to distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user.

## Transition from User to Kernel Mode

- When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), it must transition from user mode to kernel mode to fulfill the request.



- At system boot time, the hardware starts in **kernel mode**.
- The OS is then loaded and starts user applications in user mode.
- Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode.
- Thus, whenever the operating system gains control of the computer, it is in kernel mode.
- The system always switches to user mode before passing control to a user program.

## Contd.

- ❖ The dual mode of operation provides us with the means for protecting the OS from errant users-and errant users from one another by designing some of the machine instructions as **privileged instructions**.
  - The hardware allowed privileged instructions to be executed only in kernel mode.
  - If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.
- ❖ The instruction to switch to kernel mode, I/O control timer management and interrupt management are examples of privileged instructions.
- ❖ We must ensure that the OS maintains a control over the CPU. i.e. we can not allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the OS.
- ❖ To accomplish this goal we must use a **timer** that can be set to interrupt the computer after specific period.

### How it works?

- The OS sets the counter
- Every time the clock ticks, the counter is decremented
- When the counter reaches zero generate an interrupt
- Set up before scheduling process to regain control or terminate program that exceeds allotted time.

# Process Management

- ❖ A process is a program in execution. It is a unit of work within the system.
- ❖ Program is a *passive entity*, process is an *active entity*.
- ❖ Process needs resources to accomplish its task
  - ✚ CPU, memory, I/O, files
  - ✚ Initialization data
- ❖ Process termination requires reclaim of any reusable resources
- ❖ Single-threaded process has one **program counter** specifying location of next instruction to execute
  - ✚ Process executes instructions sequentially, one at a time, until completion
- ❖ Multi-threaded process has one program counter per thread
- ❖ Typically system has many processes, some user, some operating system running concurrently on one or more CPUs.
  - ✚ Concurrency by multiplexing the CPUs among the processes / threads

# Process Management Activities

- The operating system is responsible for the following activities in connection with process management:
  - ❖ Creating and deleting both user and system processes
  - ❖ Suspending and resuming processes
  - ❖ Providing mechanisms for process synchronization
  - ❖ Providing mechanisms for process communication
  - ❖ Providing mechanisms for deadlock handling

# Memory Management

- ❖ All data in memory before and after processing
- ❖ All instructions need to be in memory in order to execute
- ❖ Memory management determines what is in memory when
  - ❖ **Optimizing CPU utilization and computer response to users**
- ❖ Memory management activities
  - ❖ Keeping track of which parts of memory are currently being used and by whom
  - ❖ Deciding which processes (or parts thereof) and data to move into and out of memory.
  - ❖ Allocating and deallocating memory space as needed

# Storage Management

- OS provides uniform, logical view of information storage
  - ✚ Abstracts physical properties to logical storage unit - file
  - ✚ Each medium is controlled by device (i.e., disk drive, tape drive)
  - ✚ Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
- File-System management
  - ✚ Files usually organized into directories
  - ✚ Access control on most systems to determine who can access what
- OS activities include
  - ✚ Creating and deleting files and directories
  - ✚ Primitives to manipulate files and directories
  - ✚ Mapping files onto secondary storage
  - ✚ Backup files onto stable (non-volatile) storage media

# Mass-Storage Management

- Usually disks used to store data that does not fit in main memory or data that must be kept for a “long” period of time.
- Proper management is of central importance
- Entire speed of computer operation hinges on disk subsystem and its algorithms
- **OS activities**
  - ✚ Free-space management
  - ✚ Storage allocation
  - ✚ Disk scheduling
- Some storage need not be fast
  - ✚ Tertiary storage includes optical storage, magnetic tape
  - ✚ Still must be managed
- Varies between WORM (write-once, read-many-times) and RW (read-write)

# I/O Subsystems

- One purpose of OS is to hide peculiarities of hardware devices from the user
- I/O subsystem responsible for
  - ✚ Memory management of I/O including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping of output of one job with input of other jobs)
  - ✚ General device-driver interface
  - ✚ Drivers for specific hardware devices

# Protection and Security

- **Protection** – any mechanism for controlling access of processes or users to resources defined by the OS.
- **Security** – defence of the system against internal and external attacks
  - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first distinguish among users, to determine who can do what
  - User identities (**user IDs**, security IDs) include name and associated number, one per user
  - User ID then associated with all files, processes of that user to determine access control
  - Group identifier (**group ID**) allows set of users to be defined and controls managed, then also associated with each process, file
  - **Privilege escalation** allows user to change to effective ID with more rights

# Operating System Services

1. **User Interface:-** Almost all operating systems have a **User Interface**.
  - This interface can take several forms:- Command Line Interface (CLI). Batch Interface, in which commands and directives to control those commands are entered into files, and those files are executed, and Graphical User Interface (GUI)
2. **Program execution:-** OS loads a program into memory and executes the it.
3. **I/O Operations:-** since user programs cannot execute I/O operations directly, the operating system must provide some means to perform I/O.
4. **File-system manipulation:-** program needs to read, write, create, and delete files.
  - + The OS gives the permission to the program for operations on file.
5. **Communications-** exchange of information between processes executing either on the same computer or on different systems tied together by a network. Implemented via *shared memory* or *message passing*.
6. **Error detection:-** ensure correct computing by detecting errors in the CPU and memory hardware, in I/O devices, or in user programs.

# System Calls

- System calls provide the interface between a **running program(Process)** and the **operating system kernel**.
  - A system call instruction is an instruction that generates an interrupt that causes the OS to gain control of the processor.
  - A system call is used whenever a program needs to access a restricted source. Ex. A file on hard disk, any hardware device.
  - Most operations interacting with the system require permissions not available to a user level process, or any form of communication with other processes requires the use of system calls.
  - Generally available as assembly-language instructions.
  - Languages defined to replace assembly language for systems programming allow system calls to be made directly (e.g., C, C++).
- **Three general methods** are used to pass parameters between a running program and the operating system.
  1. Pass parameters in *registers*.
  2. Store the parameters in a table in memory, and the table address is passed as a parameter in a register.
  3. *Push* (store) the parameters onto the *stack* by the program, and *pop* off the stack by operating system

## Contd

- ❖ A System Call is the main way a user program interacts with the Operating System.
- ❖ A **system call** is how a program requests a service from an OS's **kernel**. This may include hardware related services (e.g. accessing the hard disk), creating and executing new processes, and communicating with integral kernel services (like scheduling). System calls provide an essential interface between a process and the operating system.

# System Call Types

- A system call is made using the system call machine language instruction.
- System calls can be roughly grouped into five major categories:

## 1. Process Management

- Load
- Execute
- Create process
- Terminate process
- Get/set process attributes
- Wait for time, wait for event, wait for signal
- Allocate, free Memory

## 2. Inter-Process Communication

- I. create, delete communication connection
- II. send, receive messages
- III. transfer status information
- IV. attach or detach remote devices

## 3. File management

- a. create file, delete file
- b. open, close
- c. read, write, reposition
- d. get/set file attributes

## 5. I/O Device Management

- i. request device, release device
- ii. read, write, reposition
- iii. get/set device attributes
- iv. logically attach or detach devices

## 6. Information Maintenance

- a. get/set time or date
- b. get/set system data
- c. get/set process, file, or device attributes

# Operating System Structure

- Modern operating systems are complex and consists of different components.
- These components of modern systems are interconnected and melded to the kernel.
- five different OS structures have been tried so far.
- The **five** designs are:
  - a. monolithic systems
  - b. layered systems
  - c. virtual machines and
  - d. Microkernel's
  - e. Modules

# Chapter Two

## Process and Thread Management

**Part Two**

**Threads and Multithreading**

**Operating Systems**  
**(SEng 2043)**

# Objective

At the end of this session students will be able to:

- Understand the basic concepts, principles and notions of a thread, a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems.
- Understand the differences and similarities of threads and processes
- Identify the basic thread libraries and threading issues.
- To examine issues related to multithreaded programming and multithreading Models
- Threading Issues
- Operating System Examples

# Threads

- ☞ Inter-process communication is simple and easy when used occasionally
  - ⊕ If there are many processes sharing many resources, then the mechanism becomes cumbersome- **difficult to handle.**
  - ⊕ Threads are created to make this kind of resource sharing simple & efficient
- ☞ Each process has the following two characteristics:
  1. **Unit of resource ownership:-** When a new process is created , an address space containing program text and data , as well as other resources(files, child processes, pending alarms, signal handlers, accounting information) is allocated.
    - ⊕ The unit of resource ownership is usually referred to as a **Task** or a **Process** .
  2. **Unit of dispatching:-** is a **thread of execution**, usually shortened to just thread.
    - ⊕ The thread has a **program counter**, that keeps track of which instruction to execute next.
    - ⊕ It has **registers**, which hold its current working variables.

## Contd.

- ❖ It has a **stack**, which contains the execution history, with one frame for each procedure called but not yet returned from.
- ❖ The unit of dispatching is usually referred to a **Thread** or a **Light-Weight Process (LWP)**.
- ❖ A thread is a basic unit of CPU utilization that consists of:

- ❖ Thread id
- ❖ Execution State
- ❖ Program counter
- ❖ Register set
- ❖ Stack

Address space, global variables, Child Processes, Pending alarms, Signals and signal Handlers, protected access to Open files, I/O and other resources and Accounting Information

- ❖ Threads belonging to the same process share:

- ❖ its code
- ❖ its data section
- ❖ other OS resources such as **open files** and

# Process Vs. Thread

- ☞ A **thread of execution** is the smallest unit of processing that can be scheduled by an OS.
- ☞ The implementation of threads and processes differs from one OS to another, but in most cases, **a thread is contained inside a process**.
- ☞ Multiple threads can exist within the same process and share resources such as **memory**, while different processes do not share these resources.
- ☞ Like process states, threads also have states:
  - ✚ **New, Ready, Running, Waiting and Terminated**
- ☞ Like processes, the OS will **switch** between threads (even though they belong to a single process) for CPU usage.
- ☞ Like process creation, thread creation is supported by **APIs**
- ☞ Creating threads is **inexpensive** (cheaper) compared to processes
  - ✚ They do not need **new address space, global data, program code or operating system resources**
  - ✚ **Context switching** is **faster** as the only things to save/restore are program counters, registers and stacks

## Contd.

### Similarities

- ☞ Both **share CPU** and only one thread/process is active (running) at a time.
- ☞ Like processes, threads within a process execute **sequentially**.
- ☞ Like processes, thread can **create children**.
- ☞ Like process, if one thread is blocked, another thread can run.

### Difference

- ☞ Unlike processes, threads are **not independent** of one another.
- ☞ Unlike processes, all threads can access every address in the task.
- ☞ Unlike processes, threads are designed to assist one other.

Contd.

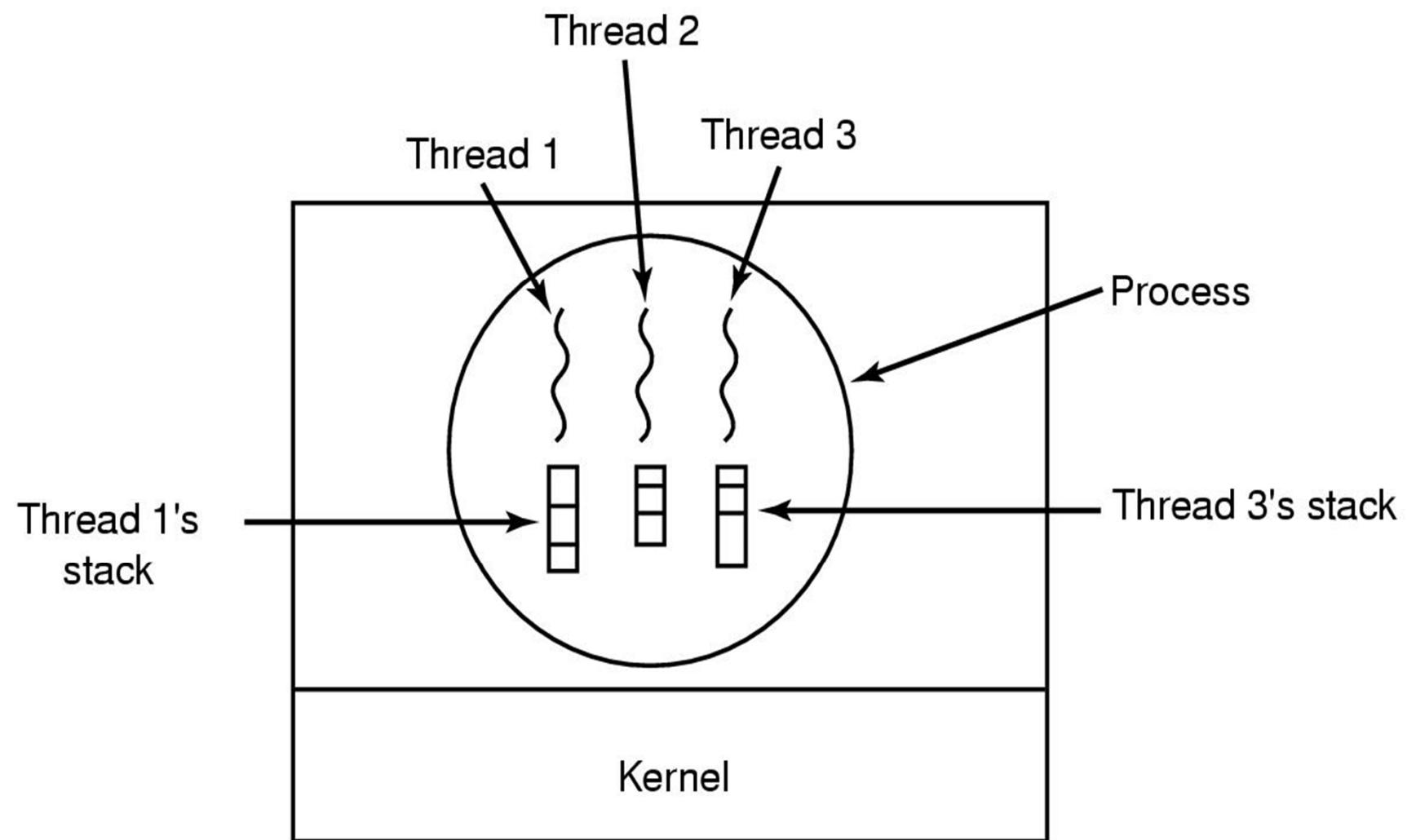


Fig. 2.2.1 Each thread has its own Stack

# Thread Libraries

- ☞ Thread libraries provide programmers an API to create and manage threads
- ☞ They can be implemented in either user space or kernel space

## User space libraries

- ✚ All code and data structure of the library are in user space
- ✚ Invoking a function in the library results in a local function call in user space and not a system call

## Kernel space libraries

- ✚ Code and data structure of the library is in kernel space and is directly supported by OS.
- ✚ Invoking an API for the library results in a system call to the kernel

- ☞ There are **three basic libraries** used today:

1. POSIX pthreads
2. WIN32 threads
3. Java threads

Contd.



## 1. POSIX pthreads

- ⊕ They may be provided as either a user or kernel library, as an extension to the POSIX standard
- ⊕ Systems like Solaris, Linux and Mac Oss implement pthreads specifications

## 2. WIN32 threads

- ⊕ These are provided as a kernel-level library on Windows systems.

## 3. Java threads

- ⊕ Since Java generally runs on a Java Virtual Machine (JVM), the implementation of threads is based upon whatever OS and hardware the JVM is running on i.e. either Pthreads or Win32 threads depending on the system.
- ⊕ On Windows systems, Java threads are typically implemented using the Win32 API whereas UNIX and Linux systems often use Pthreads.

## Examples of Threads

### In a word processor,

- ⊕ A **background** thread may check spelling and grammar, while a **foreground** thread processes user input (keystrokes), while yet a third thread loads images from the hard drive, and a fourth does periodic automatic backups of the file being edited

### In a spreadsheet program,

- ⊕ one thread could display menus and read user input, while another thread executes user commands and updates the spreadsheet

### In a web server,

- ⊕ Multiple threads allow for multiple requests to be satisfied simultaneously, without having to service requests sequentially or to fork off separate processes for every incoming request

## Thread usage example: Web Server

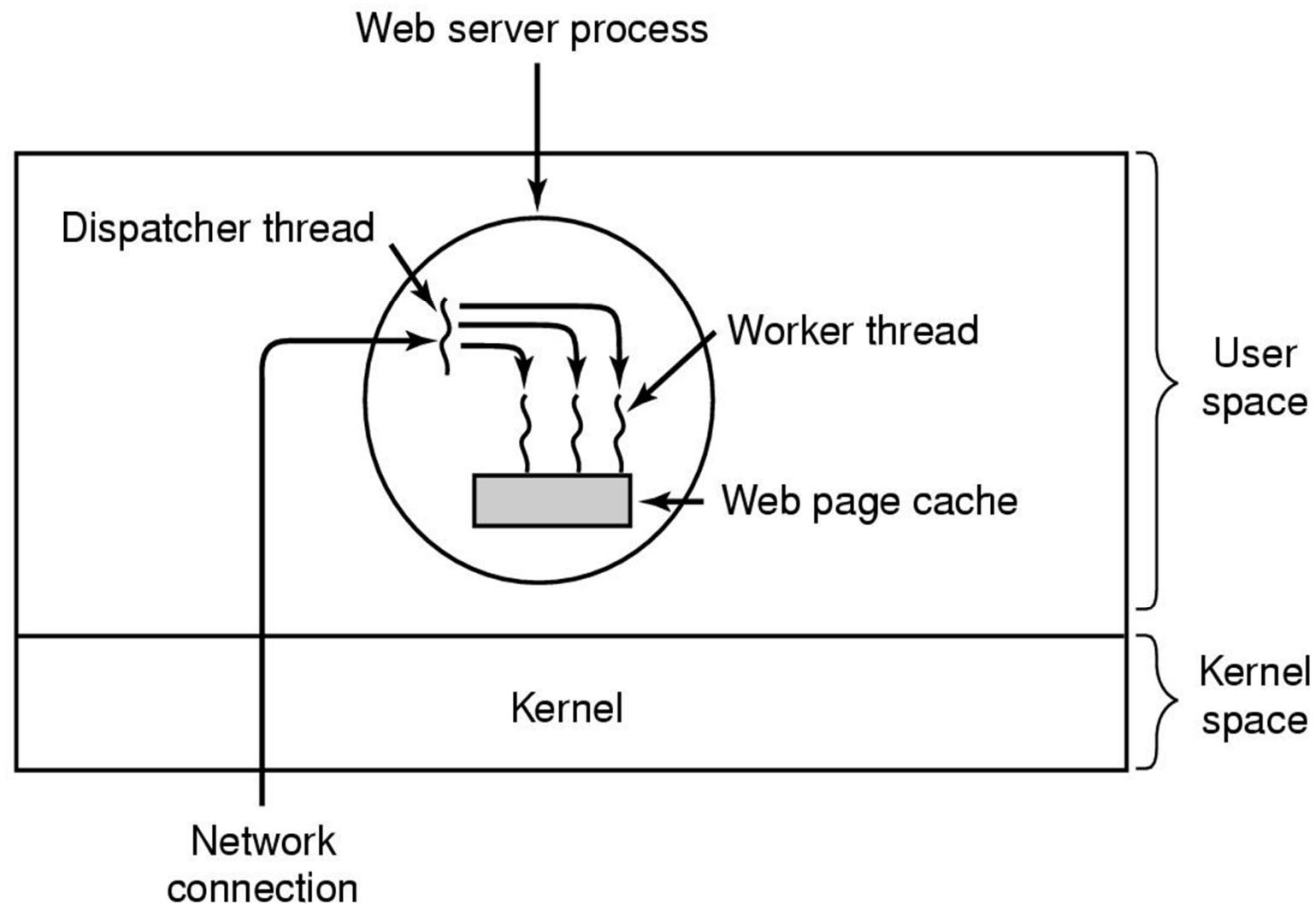


Fig. 2.2.2a Web server process and its threads

## Contd.

```
//Dispatcher Thread  
  
while(true)  
{  
    get_next_request(&buf);  
    handoff_work(&buf);  
}
```

```
//Worker Thread  
  
while(true)  
{  
    wait_for_work(&buf);  
    look_for_page_in_cache(&buf, &page);  
    if(page_not_in_cache(&page))  
        read_page_from_disk(&buf, &page);  
    return_page(&page);  
}
```

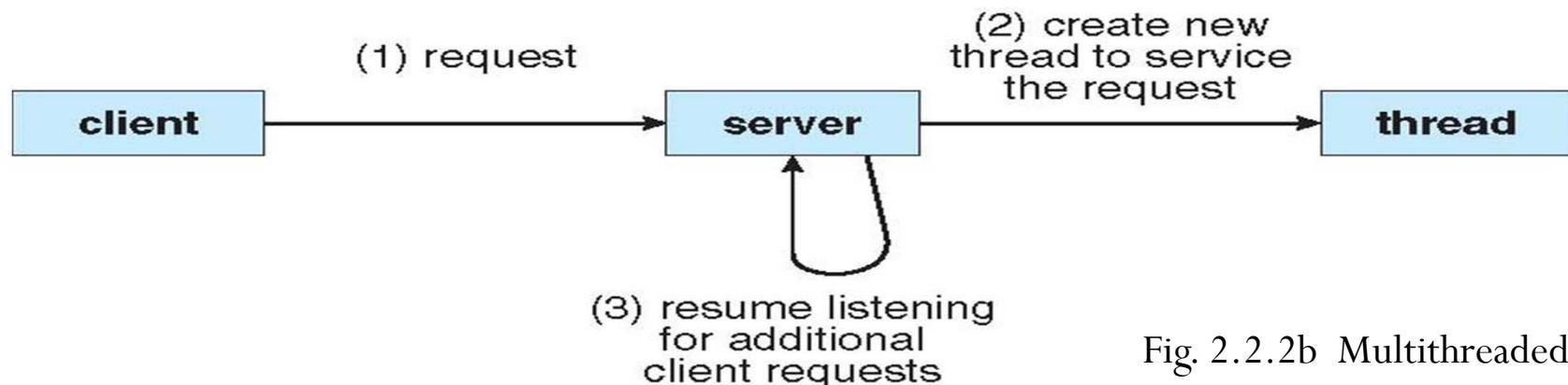


Fig. 2.2.2b Multithreaded Server Architecture

## Multithreading

- ☞ Multithreading refers to the ability on an operating system to support multiple threads of execution within a single process.
- ☞ A traditional (heavy weight) process has a single thread of control
  - + There's one program counter and a set of instructions carried out at a time
- ☞ If a process has multiple thread of control, it can perform more than one task at a time
  - + Each threads have their own program counter, stacks and registers
  - + But they share common code, data and some operating system data structures like files

## Multitasking Vs. Multithreading

- ☞ **Multitasking** is the ability of an OS to execute more than one program simultaneously.
    - ✿ Though we say so but in reality **no two programs** on a single processor machine can be executed at the same time.
  - ☞ The CPU switches from one program to the next so quickly that appears as if all of the programs are executing at the same time.
  - ☞ **Multithreading** is the ability of an OS to execute the different parts of the program, called **threads, simultaneously**.
  - ☞ The program has to be designed well so that the different threads do not interfere with each other.
  - ☞ Individual programs are all isolated from each other in terms of their memory and data, but individual threads are not as they all share the same memory and data variables.
- ✿ **Hence, implementing multitasking is relatively easier in an operating system than implementing multithreading.**

## Contd.

- ☞ Traditionally there is a single thread of execution per process.
  - ✚ **Example: MSDOS** supports a single user process and single thread.
  - ✚ **Older UNIX** supports multiple user processes but only support one thread per process.

## Multithreading

- ✚ **Java run time environment is an example of one process with multiple threads.**
- ✚ **Examples of OS supporting multiple processes, with each process supporting multiple threads: Windows 2000, Solaris, Linux, Mac, and OS/2**

Contd.

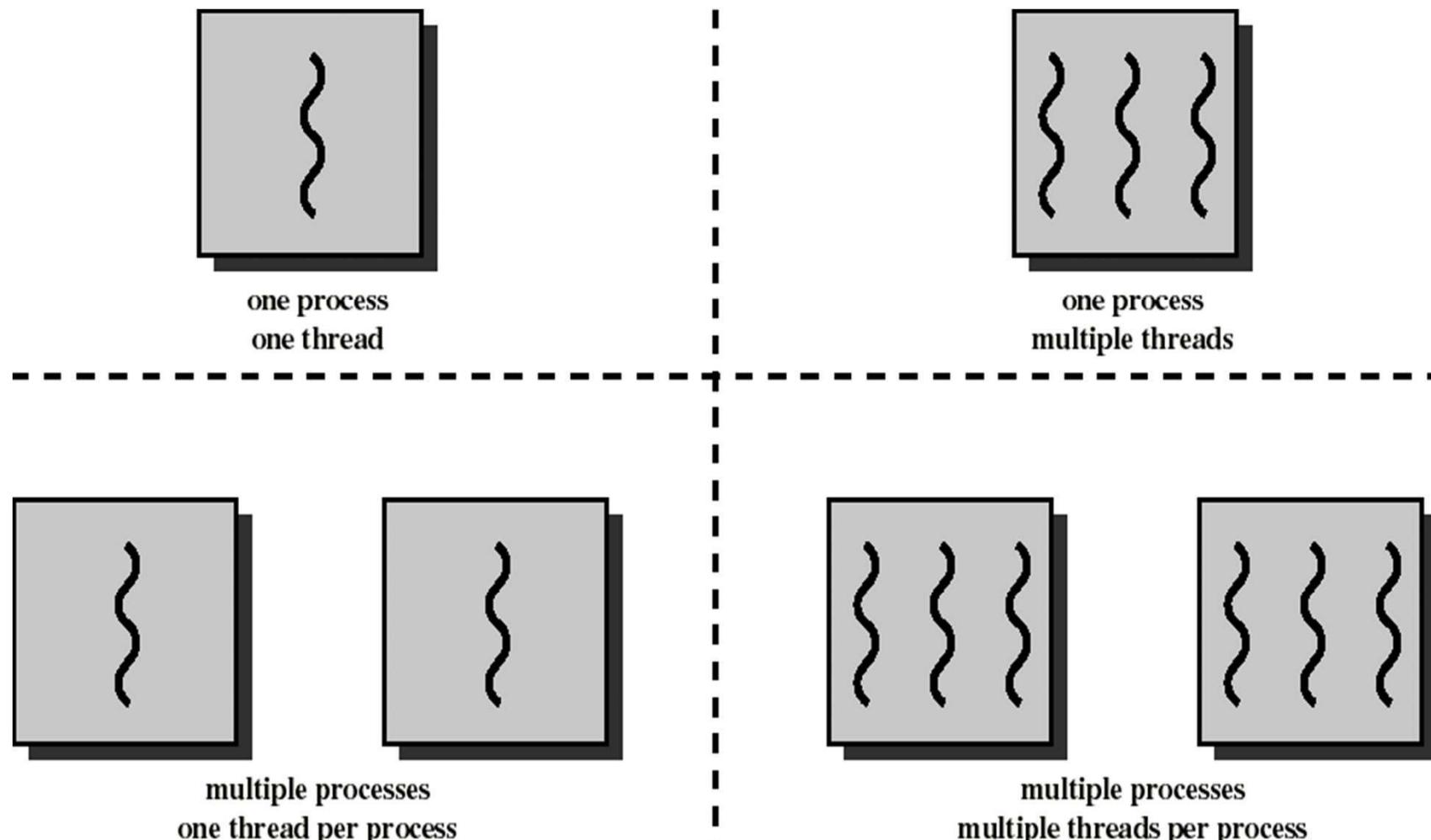


Fig. 2.2.3 Combinations of Threads and Processes

## Single and Multithreaded Processes

- ☞ In a single threaded process model, the representation of a process includes its **PCB**, **user address space**, as well as **user** and **kernel stacks**
- ☞ When a process is running, the contents of these registers are controlled by that process, and the contents of these registers are saved when the process is not running.
- ☞ In a multithreaded environment
  - ✚ There is a single PCB and address space,
  - ✚ However, there are separate stacks for each thread as well as separate control blocks for each thread containing register values, priority, and other thread related state information.

## Contd.

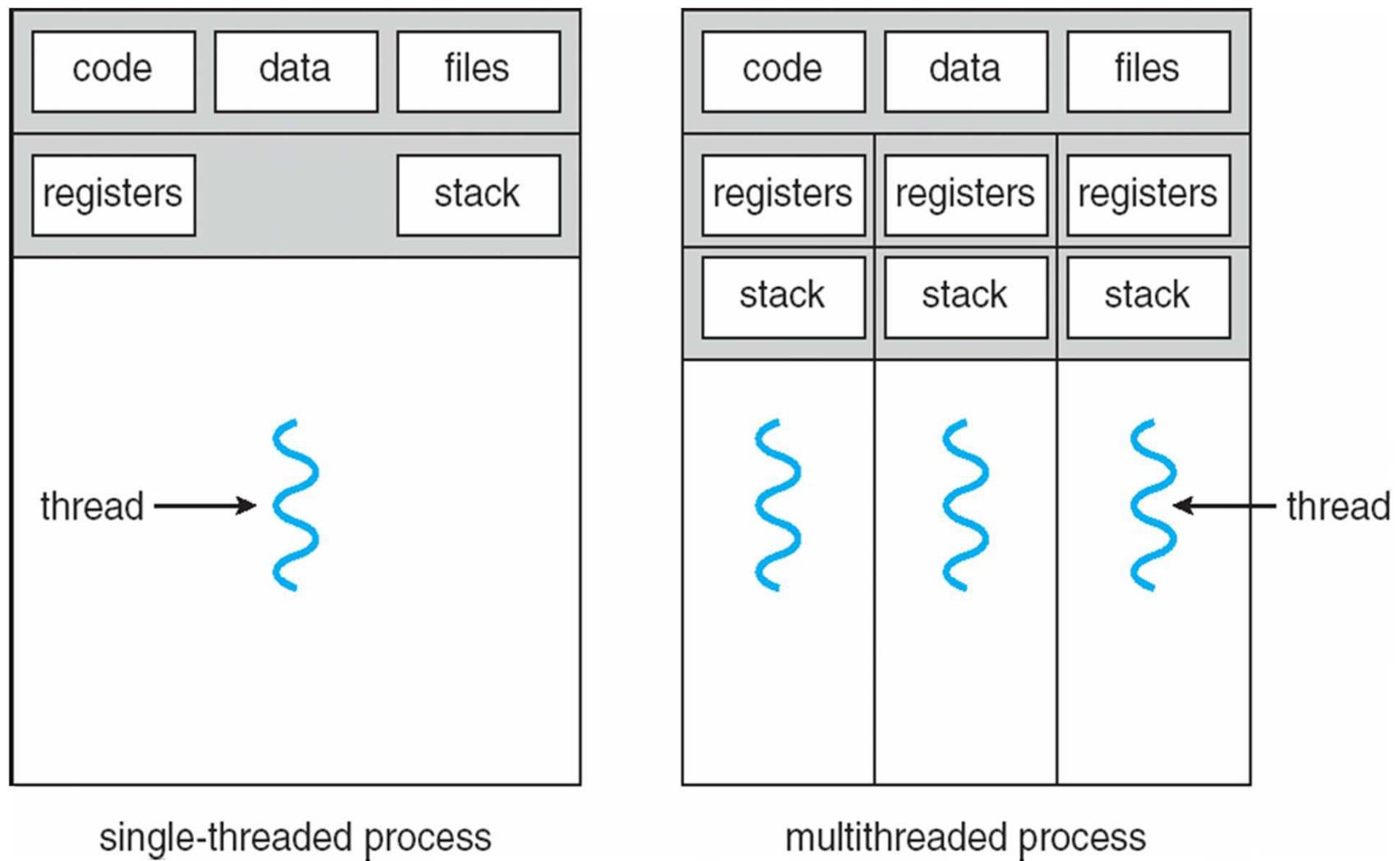


Fig. 2.2.4 Single and Multithreaded processes

# Benefits of Multithreading

## 1. Responsiveness

- ⊕ A program is allowed to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
- ⊕ One thread can give response while other threads are blocked or slowed down doing computations

## 2. Resource Sharing

- ⊕ Threads share common **code**, **data** and **resources of the process** to which they belong.
- ⊕ This allows multiple tasks to be performed within the same **address space**

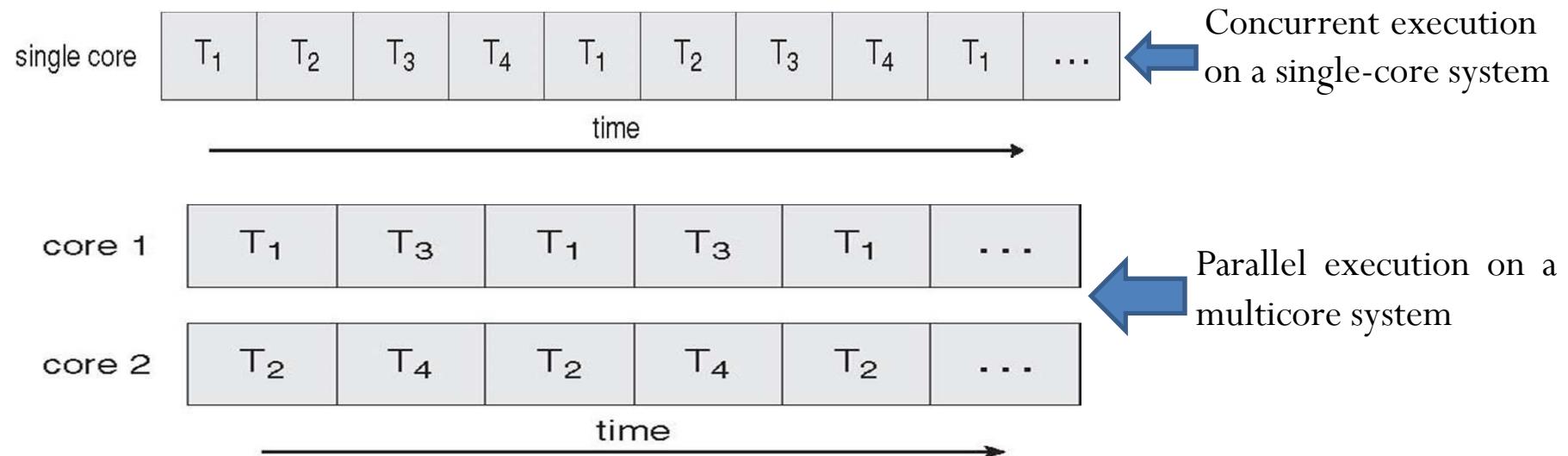
## 3. Economy

- ⊕ Creating and allocating memory and resources to processes is **expensive**, while creating threads is cheaper as they share the resources of the process to which they belong.
- ⊕ **Hence, it's more economical to create and context-switch threads.**

**Contd.**

#### 4. Scalability/utilization of multi-processor architectures

- ✚ The benefits of multithreading is increased in a multiprocessor architecture, where threads may be executed in parallel on different processors.
- ✚ A single-threaded process can run on one CPU, no matter how many are available.
- ✚ Multithreading on a multi CPU machine increases **parallelism**.



# Challenges in programming for multicore systems

☞ There are five challenges in programming for multicore systems:

1. **Dividing Activities:-** involves examining applications to find areas that can be divided into **separate, concurrent** tasks and thus can run in parallel on individual cores
2. **Balancing:-** While identifying tasks that can run in parallel, programmers must also ensure that the **tasks perform equal work of equal value.**
  - ⊕ In some instances, a certain task may not contribute as much value to the overall process as other tasks; using a separate execution core to run that task may not be worth the cost.
3. **Data Splitting:-** Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.

## Contd.

**4. Data Dependency:-** The data accessed by the tasks must be examined for **dependencies** between two or more tasks.

- ⊕ In instances where one task depends on data from another, programmers must ensure that the **execution** of the tasks is **synchronized** to accommodate the data dependency.

**5. Testing and Debugging:-** When a program is running in parallel on multiple cores, there are many different execution paths.

- ⊕ Testing and debugging such concurrent programs is inherently **more difficult** than testing and debugging single-threaded applications.

# Multithreading Models

☞ There are **three types** of multithreading models in modern operating systems:

**1. Kernel Level threads:-** are supported by the OS kernel itself.

- ⊕ All modern OS support kernel threads
- ⊕ Need user/kernel mode switch to change threads

**2. User Level threads:-** are threads application programmers put in their programs.

- ⊕ They are managed without the kernel support
- ⊕ Has problems with blocking system calls
- ⊕ Cannot support multiprocessing

**2. Hybrid Level threads:-** the combination of both Kernel and User level threads.

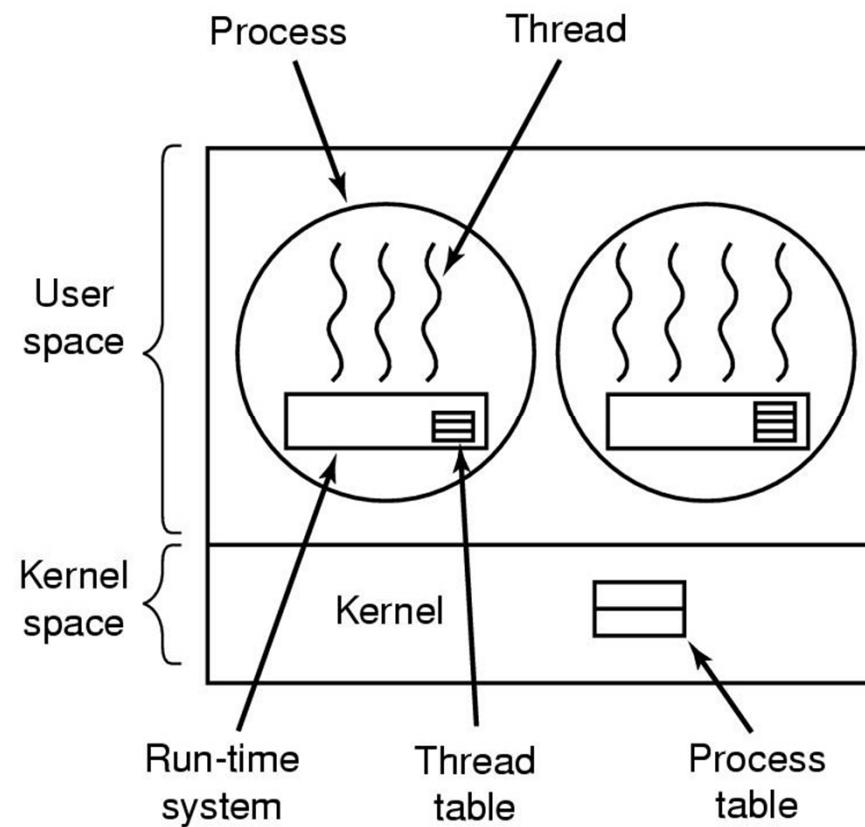
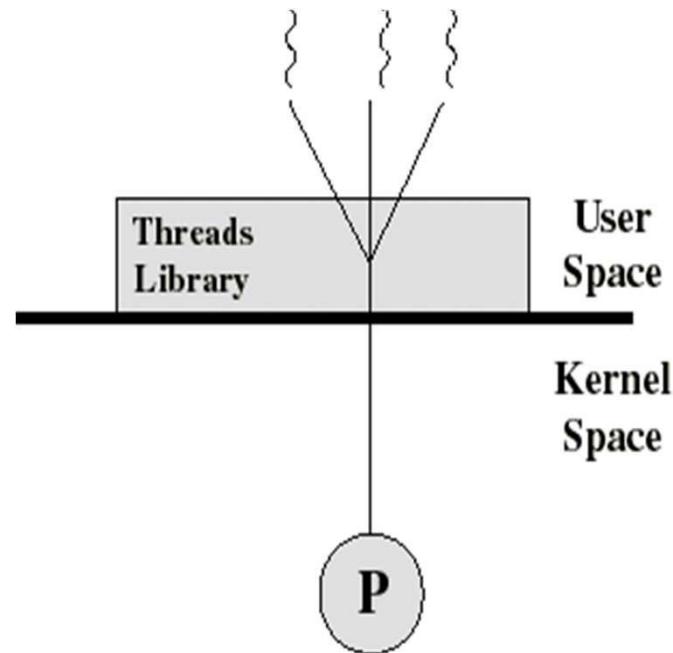
☞ There must be a relationship between the kernel threads and the user threads.

☞ There are three common ways to establish this relationship:

- A. **Many-to-One**
- B. **One-to-One**
- C. **Many-to-Many**

## 1. User level Threads (ULTs)

- ☞ The kernel is not aware of the existence of threads.
- ☞ All thread management is done by the application by using a thread library.
- ☞ Thread switching does not require kernel mode privileges (no mode switch).
- ☞ Scheduling is application specific.



## ULT Idea

- ☞ Thread management done by user-level threads library.
- ☞ Threads library contains code for:
  - ✚ creating and destroying threads.
  - ✚ passing messages and data between threads.
  - ✚ scheduling thread execution.
  - ✚ saving and restoring thread contexts.

## Kernel activity for ULTs

- ☞ The kernel is not aware of thread activity but it is still managing process activity.
- ☞ When a thread makes a system call, the whole task will be blocked.
  - ✚ But for the thread library that thread is still in the running state.
  - ✚ So thread states are independent of process states.

## Advantages and inconveniences of ULT

### Advantages

- ☞ Thread switching does not involve the kernel: no mode switching.
- ☞ Scheduling can be application specific: choose the best algorithm.
- ☞ ULTs can run on any OS.
  - + Only needs a thread library.

### Inconveniences

- ☞ Most system calls are blocking and the kernel blocks processes.
  - + So all threads within the process will be blocked.
- ☞ The kernel can only assign processes to processors.
  - + Two threads within the same process cannot run simultaneously on two processors.

## 2. Kernel Level Threads (KLTs)

- ❖ All thread management is done by kernel and kernel maintains context information for the process and the thread.
- ❖ No thread library but an API to the kernel thread facility.
- ❖ Switching between threads requires the kernel. i.e. Threads are supported by kernel.
- ❖ Scheduling on a thread basis

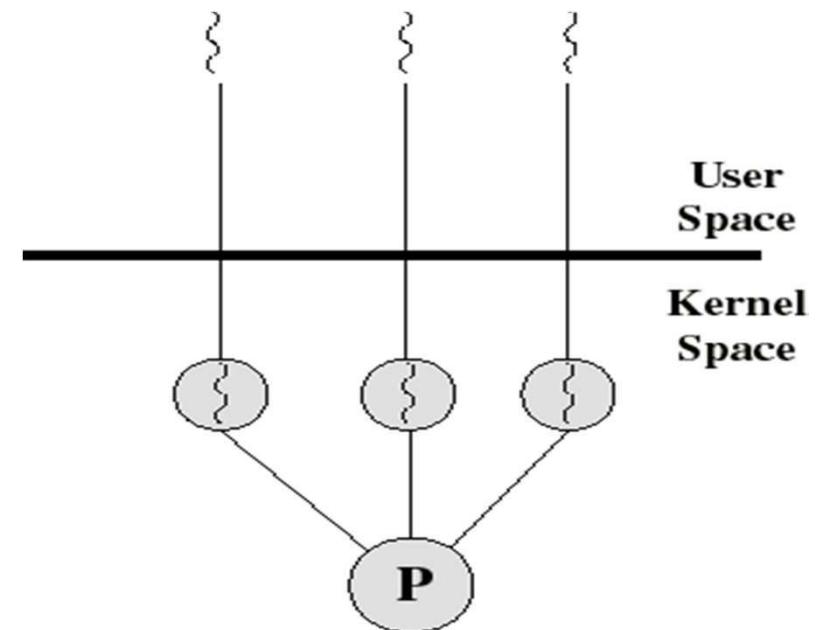
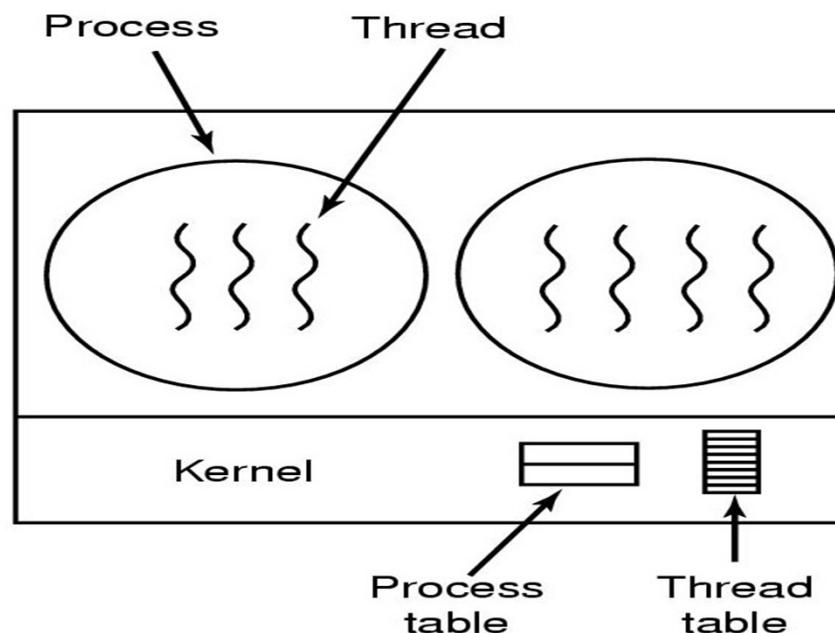


Fig. 2.2.7 KLT and its implementation

## Advantages and inconveniences of KLT

### Advantages

- ☞ The kernel can simultaneously schedule many threads of the same process on many processors.
- ☞ blocking is done on a thread level.
- ☞ kernel routines can be multithreaded.

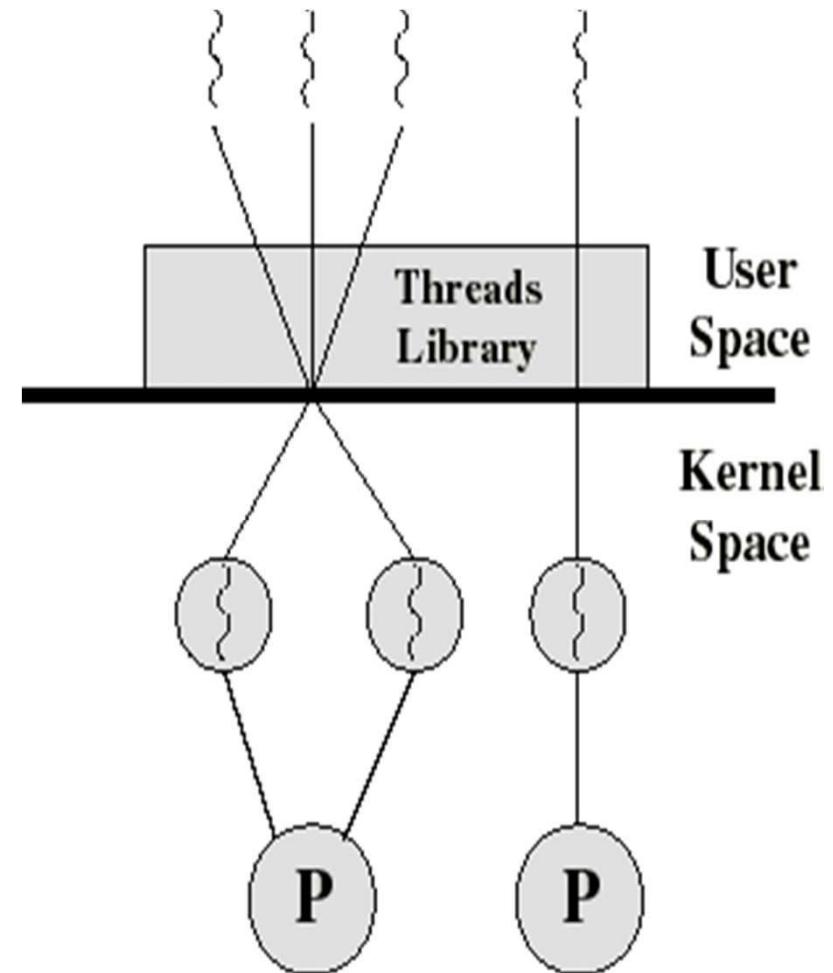
### Inconveniences

- ☞ thread switching within the same process involves the kernel.
- ☞ We have 2 mode switches per thread switch.
  - ☞ This results in a significant slowdown.

Examples of KLT OSs: Windows NT/2000/XP, Linux, Solaris, Tru64 UNIX, Mac OS X

### 3. Hybrid ULT/KLT Approaches

- ☞ Thread creation is done in the user space.
- ☞ Bulk of scheduling and synchronization of threads done in the user space.
- ☞ The programmer may adjust the number of KLTs.
- ☞ May combine the best of both approaches.



Example is Solaris prior to version 9.

Fig. 2.2.8 Hybrid ULT/KLT Approach

Contd.

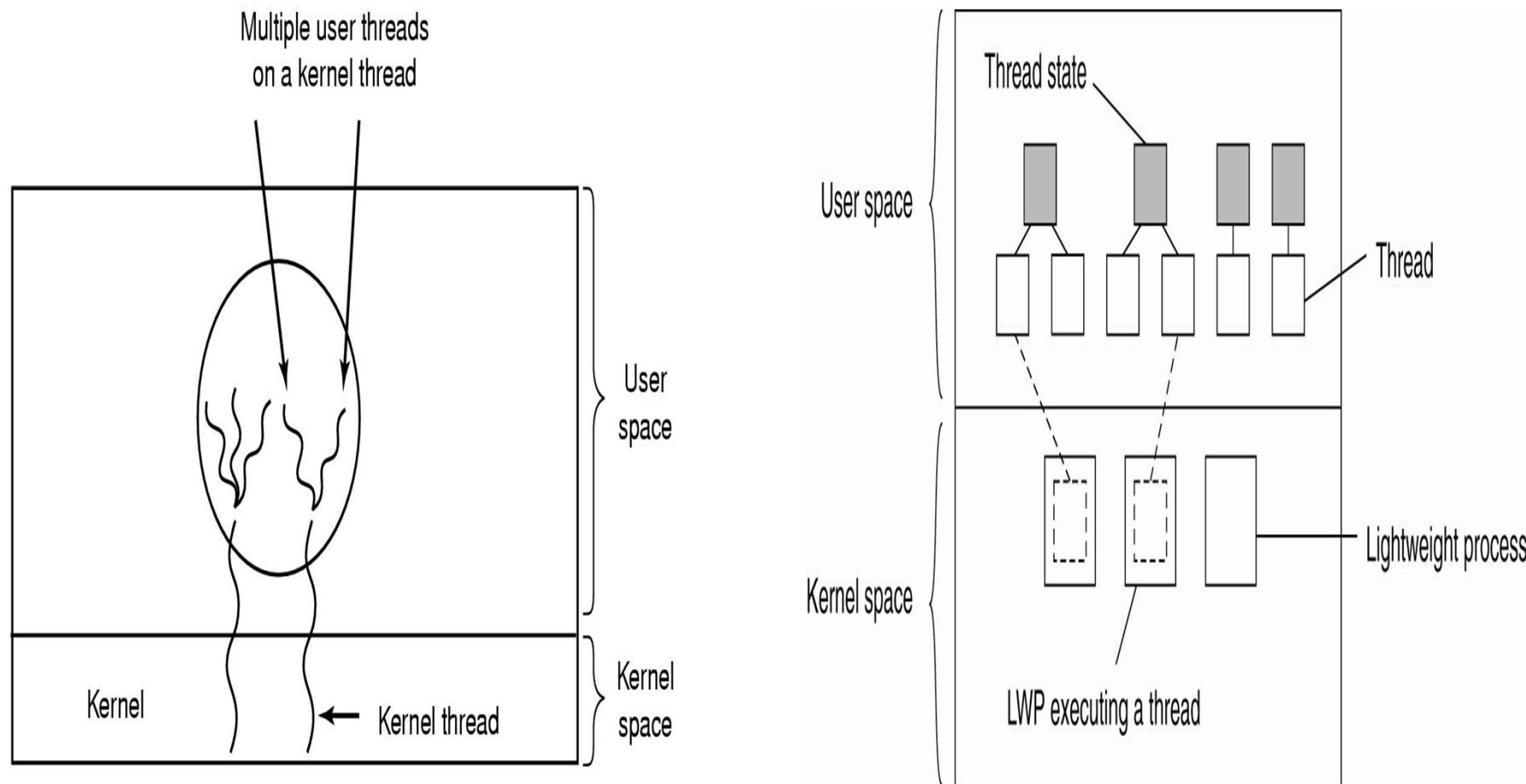


Fig. 2.2.9 Multiplexing user-level threads onto kernel-level threads.

## Multithreading Models: Many-to-One Model

- ☞ It maps many user level threads in to one kernel thread
- ☞ Thread management is done by thread library in user space.
  - + Hence, it is **efficient** because there is no mode switch but if a thread makes blocking system call, it blocks
- ☞ Only one thread can access the kernel at a time, so multiple threads can not run on multiprocessor systems
- ☞ Used by ULT Libraries on systems that do not support kernel threads (KLT).

**Examples:- Solaris Green Threads, GNU Portable Threads**

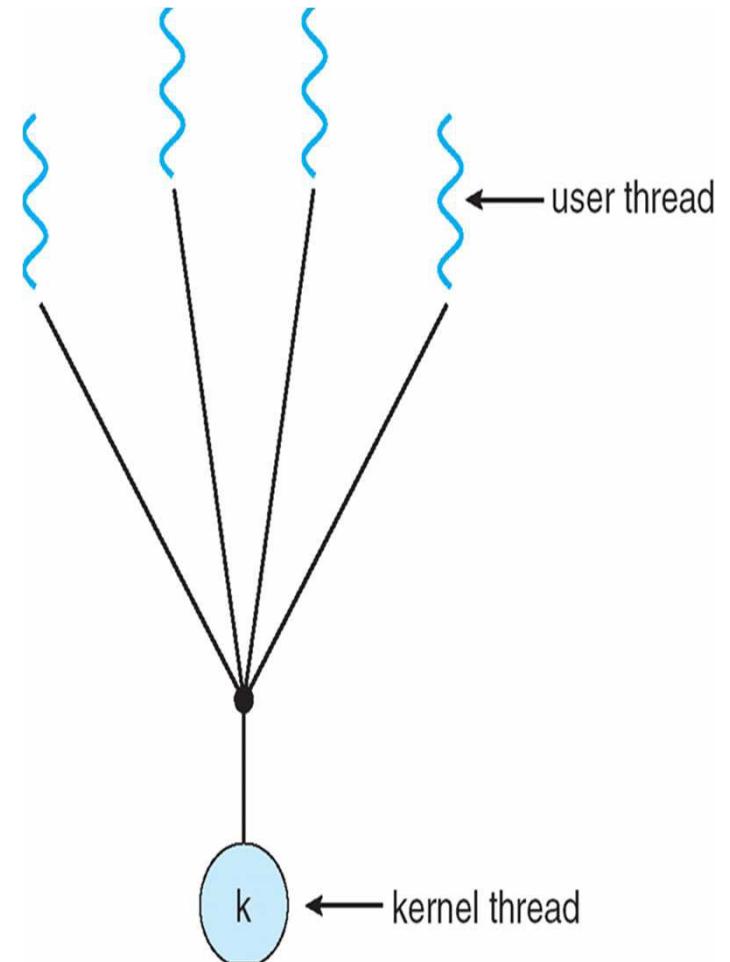


Fig. 2.2.10 Many-to-One Model

## Multithreading Models: One -to-One Model

- ☞ Each user-level thread maps to kernel thread.
- ☞ **A separate kernel thread is created to handle each user-level thread**
- ☞ It provides more concurrency and solves the problems of blocking system calls
  - ▣ Allows another thread to run when a thread is blocked.
  - ▣ Allows multiple threads to run in parallel on multiprocessors.
- ☞ Managing the one-to-one model involves **more overhead** and slows-down system
- ☞ **Drawback:-** creating user thread requires creating the corresponding kernel thread.
- ☞ Most implementations of this thread puts restrictions on the number of threads created.

**Examples:-** Windows NT/XP /2000, Linux, Solaris 9 and later

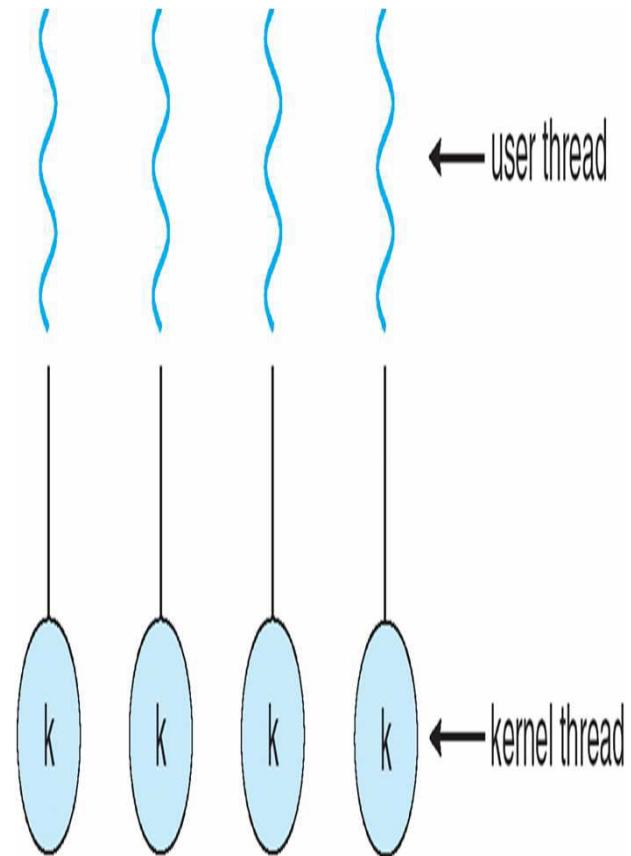


Fig. 2.2.11 One-to-One Model

## Multithreading Models: Many-to-Many Model

- ☞ Allows the mapping of many user level threads in to many(smaller or equal number ) kernel threads
- ☞ Allows the OS to create sufficient number of kernel threads(specific to either a particular application or a particular machine) .i.e.it is **most flexible**
- ☞ **It combines the best features of one-to-one and many-to-one model**
- ☞ Users have no restrictions on the numbers of threads created
- ☞ Blocking kernel system calls do not block the entire process.

**Examples:-**Solaris prior to version 9, Windows NT/2000 with the ThreadFiber package

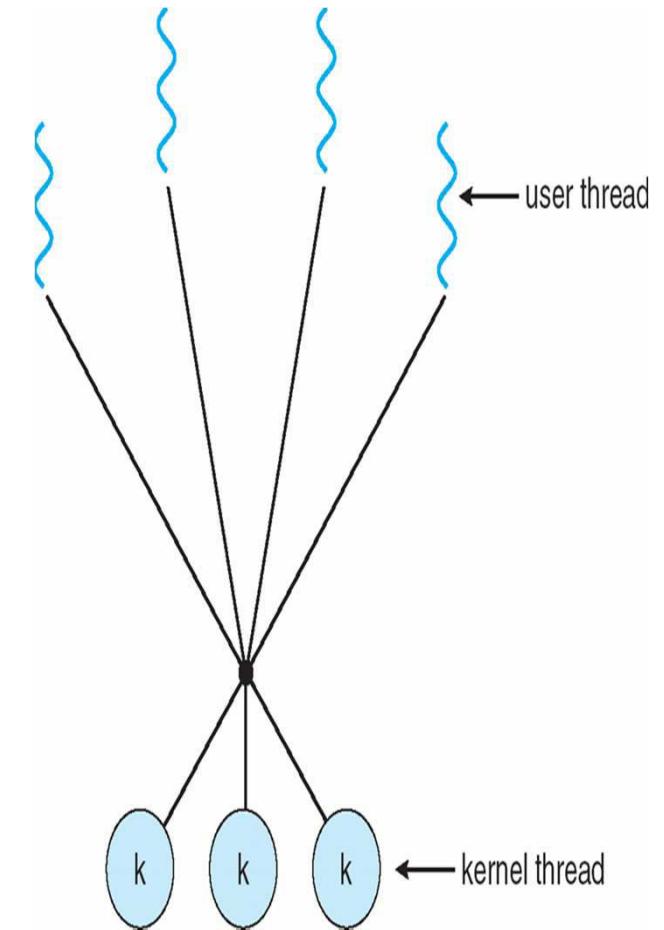


Fig. 2.2.12 Many-to-Many Model

## Multithreading Models: Many-to-Many Model

- ☞ Processes can be split across multiple processors
- ☞ Individual processes may be allocated variable numbers of kernel threads, depending on the number of CPUs present and other factors
- ☞ One popular variation of the many-to-many model is the **two-tier model**, which allows **either many-to-many or one-to-one operation.**
- ☞ Is Similar to Many-to-Many model, except that it allows a user thread to be **bound to kernel thread**

**Examples:-**Solaris 8 and earlier, IRIX, HP-UX,

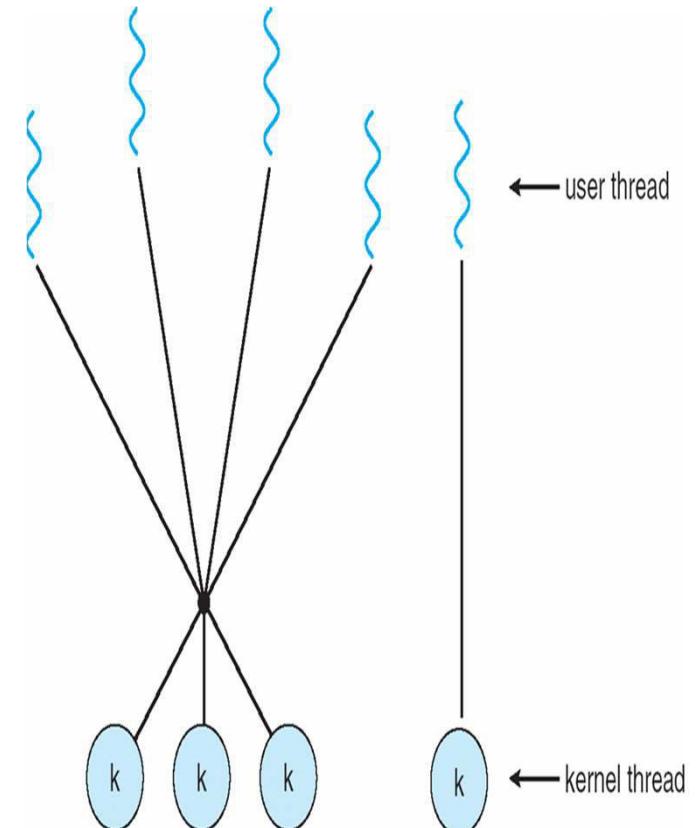


Fig. 2.2.12 Two-tier Many-to-Many Model

# Threading Issues

☞ Some of the issues to be considered for multithreaded programs:

- ✚ Semantics of fork() and exec() system calls.
- ✚ Thread cancellation
- ✚ Signal handling
- ✚ Thread pools
- ✚ Thread-specific data

## 1. Semantics of fork() and exec() system calls

☞ In multithreaded program, the semantics of the fork and exec systems calls change.

☞ **If one thread calls fork, there are two options:**

- ✚ New process can duplicate all the threads or new process is a process with single thread
- ✚ Some systems have chosen two versions of fork
- ✚ The **exec()** system call loads the selected thread to memory and executes the thread.

## Contd.

2. **Thread Cancellation:-** is the task of terminating thread before its completion.

Example: if multiple threads are searching a database, if one gets the result others should be cancelled

☞ A thread that is to be canceled is often referred to as the **target thread**.

☞ Cancellation of a target thread may occur in two different scenarios:

### A. Asynchronous cancellation

⊕ One thread immediately terminates the target thread

### B. Deferred cancellation

⊕ The target thread can periodically checks if it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

## Contd.

The **difficulty** with cancellation occurs in situations **where resources have been allocated to a canceled thread or where a thread is canceled while in the midst of updating data it is sharing with other threads.**

- This becomes especially troublesome with **asynchronous cancellation**.
  - ✚ Often, the OS will reclaim system resources from a canceled thread but will not reclaim all resources.
  - ✚ Therefore, canceling a thread asynchronously may not free a necessary system-wide resource.
- But in case of **deferred cancellation**, one thread indicates that a target thread is to be canceled, but cancellation occurs only after the target thread has checked a flag to determine whether or not it should be canceled.
  - ✚ The thread can perform this check at a point which it can be canceled safely.
  - ✚ Pthreads refers to such points as **cancellation points**

## Contd.

**3. Signal Handling:-** Signals are used in UNIX systems to **notify** a process that a particular event has occurred

- ☞ A signal may be received either synchronously or asynchronously, depending on the source of and the reason for the event being signaled.
- ☞ All signals, whether **synchronous** (illegal memory access or division by zero) or **asynchronous** (signal is generated by an event external to a running process), are handled using signal handler following the below pattern:
  - a. **Signal is generated by particular event**
  - b. **Signal is delivered to a process**
  - c. **Signal is handled**
- ☞ A signal may be handled by one of two possible handlers:
  1. **A default signal handler**
  2. **A user-defined signal handler**

## Contd.

- Every signal has a **default signal handler** that is run by the kernel when handling that signal.
- This default action can be overridden by a **user-define signal handler** that is called to handle the signal.
- Signals are handled in different ways.
  - Some signals (such as changing the size of a window) are simply ignored; others (such as an illegal memory access) are handled by terminating the program.
  - Handling signals in **single-threaded** programs is **straightforward**: signals are always delivered to a process.
  - However, delivering signals is more complicated in multithreaded programs, where a process may have several threads.
  - Where, then, should a signal be delivered?

### Signal Handling Options:

- 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

## Contd.

**4. Thread Pools:-** Creating new threads every time one is needed and then deleting it when it is done can be inefficient, and can also lead to a very large (unlimited) number of threads being created.

- ☞ Two potential problems: **the amount of time required to create the thread prior to servicing the request** and **not setting a bound on a no. of concurrently running threads**.
- ☞ An alternative solution is to create a number of threads when the process first starts, and put those threads into a thread pool.
  - ✚ Threads are allocated from the pool as needed, and returned to the pool when no longer needed.
  - ✚ When no threads are available in the pool, the process may have to wait until one becomes available.
  - ✚ The (maximum) number of threads available in a thread pool may be determined by adjustable parameters, possibly dynamically in response to changing system loads.
  - ✚ Win32 provides thread pools through the "**PoolFunction**" function.
  - ✚ Java also provides support for thread pools.

## Contd.

**5. Thread Specific data:-** Most data is shared among threads, and this is one of the major benefits of using threads in the first place.

- However sometimes threads need thread-specific data also in some circumstances; Such data is called thread-specific data.
- For example, in a transaction-processing system, we might service each transaction in a separate thread.
- Furthermore, each transaction might be assigned a unique identifier.
- To associate each thread with its unique identifier, we could use thread-specific data.
- Most major thread libraries ( pThreads, Win32, Java ) provide support for thread-specific data.

## Contd.

**6. Scheduler Activation:-** Both Many-to-many and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application

- ❖ Scheduler activations provide **upcalls**, a communication mechanism from the kernel to the user thread library
- ❖ Upcalls are handled by the thread library with upcall handler
- ❖ This communication allows an application to maintain the correct number of kernel threads

## Reading Assignment

# Chapter Two

## Process and Thread Management

### Part Three

#### Process Scheduling

Operating Systems  
(SEng 2043)

# Objective

At the end of this session students will be able to:

- Understand the basic concepts of CPU scheduling, which is the basis for multiprogrammed operating systems.
- Describe various CPU-scheduling algorithms and scheduling criteria.
- Discuss about the basics of Multiple-Processor Scheduling
- Discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system.

# Introduction

- ☞ In a single processor system, only one process can run at a time
  - ✚ Other processes must wait until the CPU is free and can be rescheduled
- ☞ The purpose of multiprogramming is to have some process running at all times
  - ✚ **So as to maximize CPU utilization**
- ☞ **Multiprogramming:-**A number of programs can be in memory at the same time. **Allows overlap of CPU and I/O.**
  - ✚ Several processes are kept in memory at one time.
  - ✚ When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process
  - ✚ **Jobs(batch):-** are programs that run without user interaction.
  - ✚ **User(time share):-** are programs that may have user interaction.
  - ✚ **Process:-** is the common name for both (Jobs or user) in running.

# CPU Scheduling

- ☞ Like all resources CPU, one of the primary resource, is scheduled before use.
  - ❖ CPU scheduling is the basis of multiprogrammed operating systems
  - ❖ By switching the CPU among processes, the operating systems can be made more productive
- ☞ **Scheduling** refers to the way processes are assigned to run on the available CPUs - since there are typically many more processes running than there are available CPUs.
- ☞ This assignment is carried out by software known as a **scheduler** and **dispatcher**.
- ☞ Not surprisingly, in different environments different scheduling algorithms are needed.

## Contd.

- ☞ This situation arises because different application areas (and different kinds of operating systems) have different goals.
- ☞ Three environments worth distinguishing are
  - ✚ **Batch.**
  - ✚ **Interactive.**
  - ✚ **Real time.**
- ☞ In **batch systems**, there are no users impatiently waiting at their terminals for a quick response.
  - ✚ This approach reduces process switches and thus improves performance.
- ☞ In an environment with **interactive users**, preemption is essential to keep one process from hogging the CPU and denying service to the others.

## Contd.

- Even if no process intentionally run forever, due to a program bug, one process might shut out all the others indefinitely.
- Preemption is needed to prevent this behavior.
- ☞ In systems with **real-time** constraints, **preemption** is, oddly enough, **sometimes not needed** because the processes know that they may not run for long periods of time and usually do their work and block quickly.
- The difference with interactive systems is that real-time systems **run only programs that are intended to further the application at hand**.
- Interactive systems are general purpose and may **run arbitrary programs that are not cooperative or even malicious**.

## CPU-I/O Burst Cycle

☞ The success of CPU scheduling depends on an observed property of processes: **process execution consists of a cycle of CPU execution and I/O wait.**

- ✚ Processes alternate between these two states. Process execution begins with a CPU burst which is followed by an I/O burst, then is followed by another CPU burst, then another I/O burst, and so on.
- ✚ Eventually, the final CPU burst ends with a system request to terminate execution.

☞ **CPU-I/O burst cycle:-** characterizes process execution, which alternates, between CPU and I/O activity.

- ✚ CPU times are generally much **shorter** than I/O times.
- ✚ **I/O-bound** program typically has many short CPU bursts.
- ✚ **CPU-bound** program might have a few long CPU bursts.

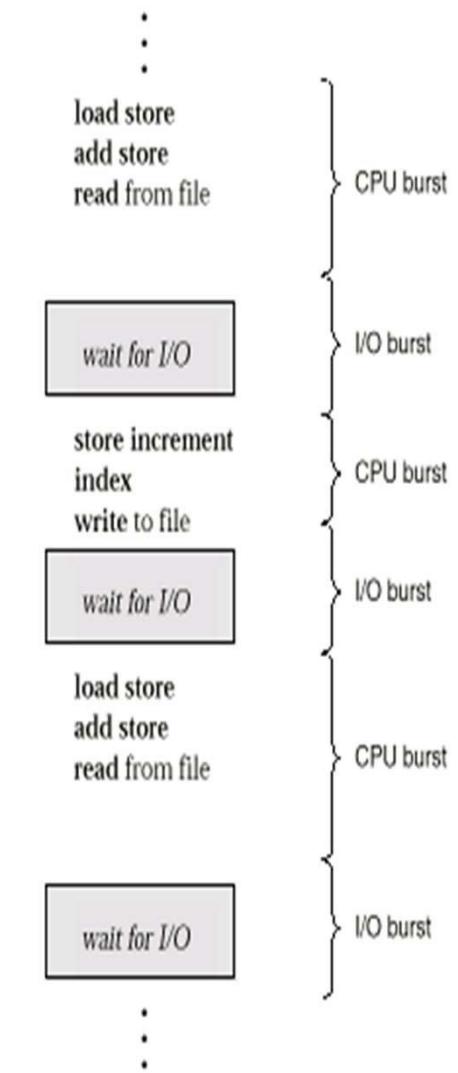


Fig. 2.3.1 CPU-I/O burst Cycle

## CPU Scheduler

- ☞ Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.
  - The selection done by the **short-term** scheduler (or **CPU scheduler**).
  - The scheduler selects a process from the processes in memory that are ready to be executes and allocates the CPU to that process.
- ☞ Note that the ready queue is not necessarily a first-in, first-out (**FIFO**) queue.
  - As we shall see when we consider the various scheduling algorithms, a ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list.
  - Conceptually, however all the processes in the ready queue are lined up waiting for a chance to run on the CPU.
  - The records in the queues are generally process **PCBs** of the processes.

## Preemptive Vs. Non- Preemptive Scheduling

- ☞ CPU Scheduling decisions may take place when a process:
  1. switches from **running** to **waiting** state( for example, as the result of **an I/O request** or an invocation of wait for the termination of one of the child processes)
  2. switches from **running** to **ready** state(for example when an **interrupt occurs**)
  3. switches from **waiting** to **ready**(for example at **competition of I/O**)
  4. Terminates/exits
- ☞ When scheduling occurs in either 1<sup>st</sup> or 4<sup>th</sup> way, then the scheduling scheme is called **non-preemptive** or **cooperative**, all other scheduling scheme is termed as **preemptive**(eg. scheduling 2 and 3)
- ☞ In **non-preemptive scheduling**, once the CPU is allocated to a process, the process keeps using the CPU until it either finishes its execution or it enters in to a waiting state
  - ✚ It is used on certain/ some hardware platforms since it does not require special hardware needed by preemptive scheduling
  - ✚ This scheduling method was used by **Microsoft Windows 3.x.**

## Contd.

☞ In **preemptive scheduling**, an interrupt causes currently running process to give up the CPU and be replaced by another process (the situations 2nd and 3rd )

### 1. The design of the operating system kernel is affected

- ✚ During the processing of a system call, the kernel may be busy with an activity on behalf of a process. Such activities may involve changing important kernel data (for instance, I/O queues).
- ✚ What happens if the process is preempted in the middle of these changes and the kernel (or the device driver) needs to read or modify the same structure?

**Solution:-** by waiting either for a system call to complete or for an I/O block to take place before doing a context switch (Ex. Most Versions of UNIX)

### 2. it incur cost associated with access to shared data

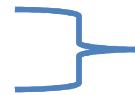
- ✚ Consider the case of two processes that share data. While one is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state. In such situations, we need new mechanisms to coordinate access to shared data

## Dispatcher

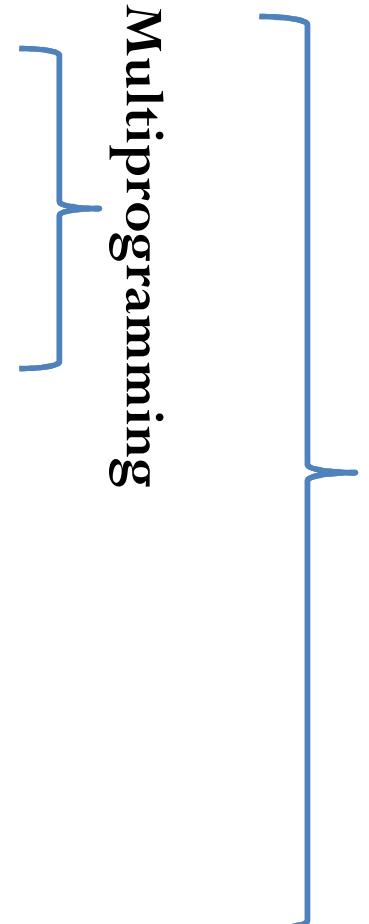
- ☞ The **dispatcher** is the module that gives control of the CPU to the process selected by the **short-term** scheduler.
- ☞ This function involves the following:
  - ✚ switching context
  - ✚ switching to user mode
  - ✚ jumping to the proper location in the user program to restart that program
- ☞ The dispatcher should be as **fast** as possible, since it is invoked during every process switch.
- ☞ The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

## When do Batch, Multiprogramming & Multitasking System do Scheduling?

1. When a process terminates
2. When a process switches from running to waiting state
  - \* E.g., when it does an I/O request
3. When a process switches from running to ready state
  - \* E.g., when a timer interrupt occurs
4. When a process switches from waiting state to ready state
  - \* E.g., completion of I/O



**Batch**



**Multiprogramming**

**Multi-tasking or time-sharing**

## Scheduling Criteria

- ☞ Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another.
- ☞ In choosing which algorithm to use in a particular situation, the following scheduling criteria should be considered:

**1. CPU Utilization:-** The fraction of time a device is in use. ( ratio of in-use time / total observation time )

- ⊕ Conceptually, CPU utilization can range from 0 to 100 percent.
- ⊕ In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).

**2. Throughput:-** The number of processes completed in a period of time. (jobs / second )

- ⊕ For long processes, this rate may be **one process per hour**; for short transactions, it may be **ten processes per second**.

## Contd.

**3. Turnaround time:-** The interval between the submission of a process to its execution

- + It is the sum of the periods spent waiting to get the memory, waiting in the ready queue, executing on the CPU and doing I/O

**4. Waiting time:-** The sum of the periods spent waiting in the ready queue

- + The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue

**5. Service time:-** The time required by a device to handle a request.

**6. Response time:-** Amount of time it takes from the submission of a request till the first response is produced.

- + is the time it takes to start responding, not the time it takes to output the response.
- + The turnaround time is generally limited by the speed of the output device.

## Scheduling Optimization Criteria

- ❖ Maximum CPU utilization
- ❖ Maximum throughput time
- ❖ Minimum turnaround time
- ❖ Minimum waiting time
- ❖ Minimum response time

### Note

- It is desirable to **maximize CPU utilization** and **throughput** and to **minimize turnaround time, waiting time, and response time**.
- In most cases, we optimize the **average measure**.
- However, under some circumstances, it is desirable to optimize the minimum or maximum values rather than the average.
- For example, to guarantee that all users get good service in **interactive systems**, we may want to **minimize the maximum response time**.

## Scheduling Algorithms

- ↳ Scheduling deals with the problem of deciding which of the outstanding requests is to be allocated resources.
- ↳ Scheduling algorithms are used for distributing resources among parties which simultaneously and asynchronously request them, in OS (to share CPU time among both threads and processes )
- ↳ The main purposes of scheduling algorithms are to minimize resource starvation and to ensure fairness amongst the parties utilizing the resources.
- ↳ There are many different scheduling algorithms:
  1. **First-Come, First Served(FCFS)**
  2. **Shortest Job First(SJF)**
  3. **Priority Based Scheduling**
  4. **Round Robin Scheduling**
  5. **Multi-Level Queues**
  6. **Multi-Level Feedback Queues**

## 1. First-Come, First Served(FCFS)

- ☞ Is the simplest **non-preemptive** scheduling algorithm where **the process that requests the CPU first is allocated the CPU first**
- ☞ The implementation of this algorithm is handled by FIFO queue
  - ✚ **Arriving jobs** are inserted in to the **tail(rear)** of the ready queue and the **process to be executed next** is removed from the **front (head)** of the ready queue
- ☞ Relative importance of jobs is measured by arrival time
  - ✚ The **average waiting time** is quite **too long**
  - ✚ **Throughput** can be low, since long processes can hog the CPU
  - ✚ **Turnaround time, waiting time and response time can be high**
- ☞ It is **easy to understand and equally easy to program.**
- ☞ A long CPU-bound process may hog the CPU and may force shorter processes to wait for a prolonged period.

## Contd.

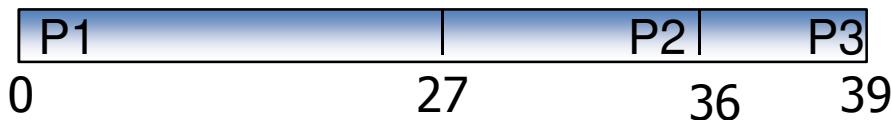
- ☞ This may lead to a long queue of ready jobs in the ready queue (convoy effect)
- ☞ The convoy effect results in a lower CPU and device utilization
- ☞ Favors CPU-bound processes
  - ✚ A CPU-bound process monopolizes the processor
  - ✚ I/O-bound processes have to wait until completion of CPU-bound process
  - ✚ I/O-bound processes may have to wait even after their I/Os are completed (poor device utilization)
  - ✚ Better I/O device utilization could be achieved if I/O bound processes had higher priority

## Contd.

**Example 1:** Consider the following processes that arrive at time zero, with the length of the CPU burst given in milliseconds

Process	Burst Time
P1	27
P2	9
P3	3

- ❖ If the processes arrive in the order of P1, P2 and P3 and are served in the FCFS order, then the waiting time for each of the processes will be as follows:



- Waiting time for **P1** is **0 ms**, meaning it starts immediately
- Waiting time for **P2** is **27 ms**, before starting
- Waiting time for **P3** is **36 ms**
- Q? What if the order of the processes was P2, P3, P1? What will be the average waiting time? Check [avg. waiting time= 7 ms] what do you notice from this?

➤ Average Completion Time =

$$(27+36+39)/3=34 \text{ ms}$$

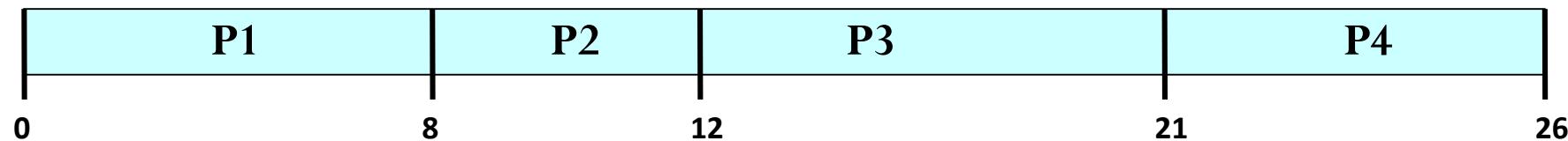
Contd.

Example 2

Process	Arrival time	Service time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

$$\begin{aligned}\text{Average wait} &= ((0) + (8-1) + (12-2) + (21-3)) / 4 \\ &= 35/4 = 8.75\end{aligned}$$

Waiting time for  $P_1 = 0$ ;  $P_2 = 8-1$ ;  $P_3 = 12-2$ ;  $P_4 = 21-3$



## 2. Shortest Job First(SJF)

- ☞ Selects the job with the shortest (expected) processing time first.
  - ✿ With this strategy the scheduler arranges processes with the **least estimated** processing time remaining to be next in the queue.
  - ✿ This requires advance knowledge or estimations about the time required for a process to complete
- ☞ When the CPU is available, it is assigned to the process that has the **smallest next CPU burst**.
  - ✿ If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.
- ☞ Note that a more appropriate term for this scheduling method would be the **shortest-next-CPU-burst** algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.

## Contd.

❖ The real difficulty with the SJF algorithm is knowing the length of the next CPU request.

☞ Two schemes:

1. **Non-preemptive** – once CPU is given to a process, it cannot be preempted in the current CPU burst
2. **Preemptive** – if a new process arrives with CPU burst length less than the remaining time of current process, preempt (**shortest remaining time-first**)

☞ One major difficulty with SJF is the **need to know or estimate the processing time of each job**

☞ This scheme is known as the **Shortest-Remaining-Time-First (SRTF)**

☞ **SJF is optimal:-** gives minimum average waiting time for a given set of processes

☞ Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process.

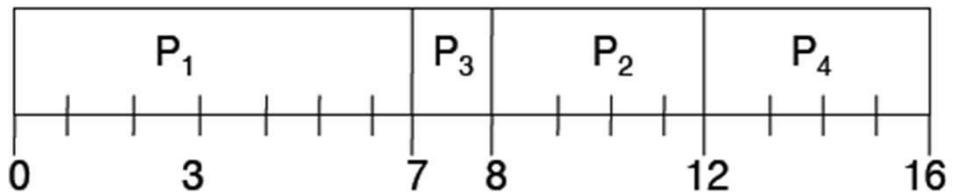
→ Consequently, the average waiting time decreases.

☞ **Starvation is possible** especially in a busy system with many small processes being run.

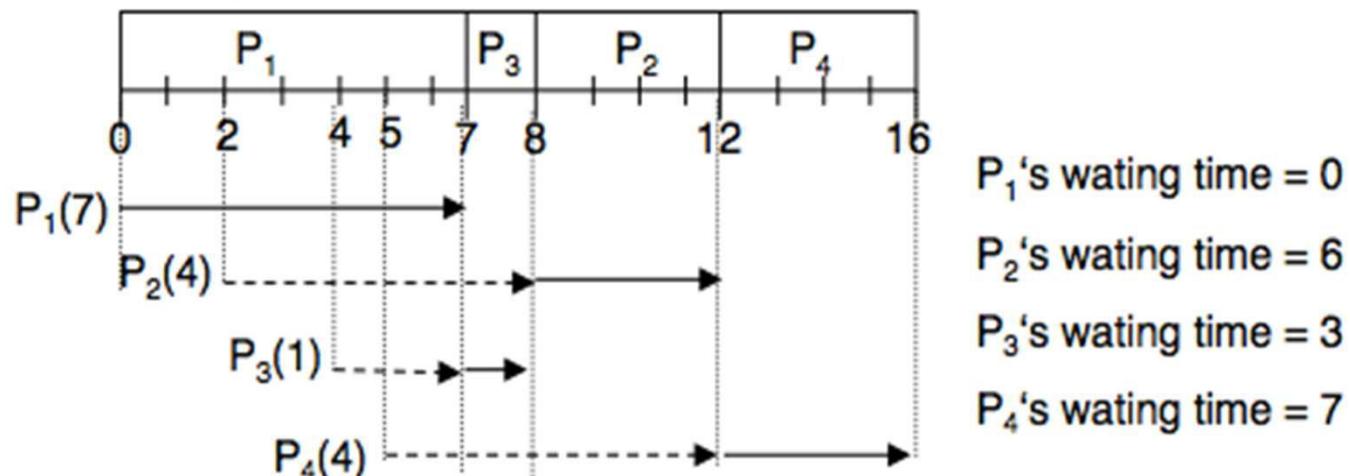
## Example

Process	Arrival time	Service time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

### a. Non Preemptive



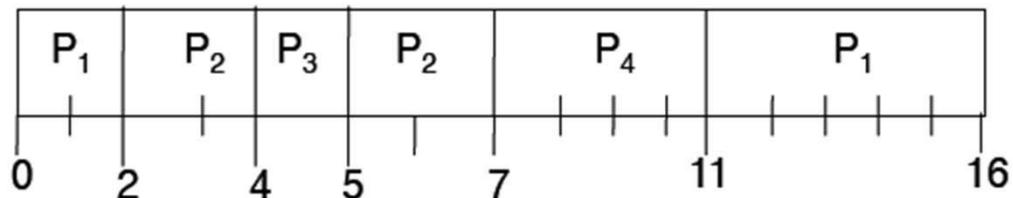
$$\begin{aligned}\text{Average waiting time} &= (0 + (8 - 2) + (7 - 4) + (12 - 5)) / 4 \\ &= 4\end{aligned}$$



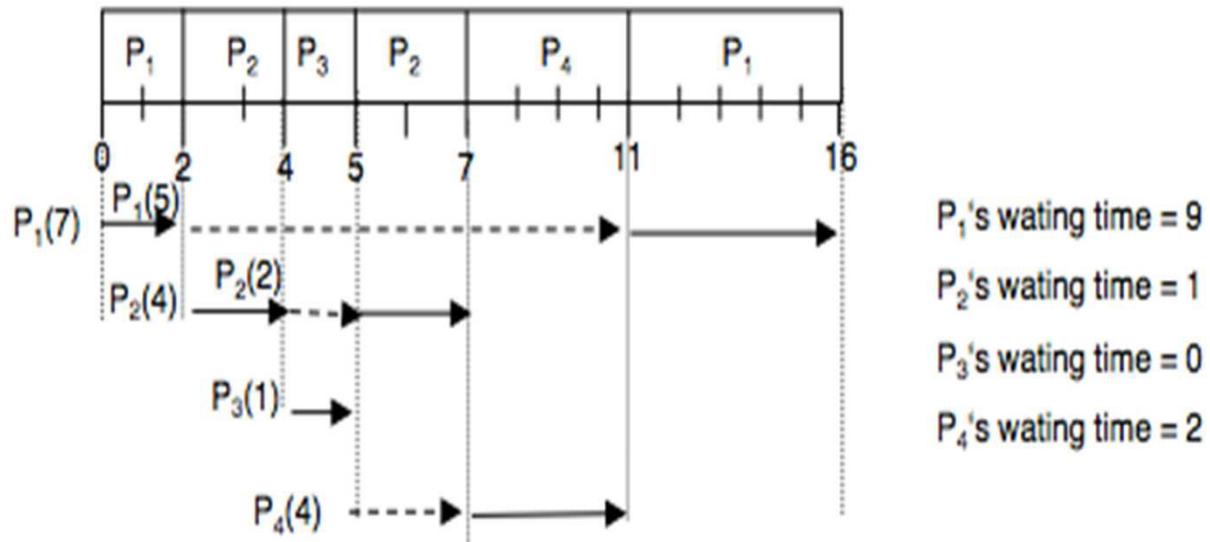
Contd.

Process	Arrival time	Service time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

b. Preemptive



$$\text{Average waiting time} = ((11-2) + (5-4) + (4-4) + (7-5))/4 \\ = 3$$



## Shortest Remaining Time First(SRTF)

- ☞ With this algorithm, the scheduler always chooses the process whose remaining run time is the shortest.
  - ✚ Is a **preemptive** version of shortest job first.
  - ✚ Here the run time has to be known in advance.
  - ✚ When a new job arrives, its total time is compared to the current process' remaining time.
  - ✚ If the new job needs less time to finish than the current process, the current process is suspended and the new job started.

## Contd.

### Starvation

- ✚ SRTF can lead to starvation if many small jobs!
- ✚ Large jobs never get to run
- ✚ Somehow need to predict future

### How can we do this?

- ✚ Some systems **ask the user** (But users may cheat) or **predict the next CPU burst length**
  - ✚ When you submit a job, you have to say how long it will take
  - ✚ To stop cheating, system kills job if it takes too long
  - ✚ But even non-malicious users have trouble predicting runtime of their jobs
- ⚡ Bottom line, can't really tell how long job will take. However, can use SRTF as a yardstick for measuring other policies, since it is optimal.

## Predicting the Future (the next CPU burst Time)

- ☞ Requires future knowledge
- ☞ But, may estimate or predict

### How to estimate or predict the future?

- ☞ Use history to predict duration of next CPU burst
  - + E.g., base on duration of last CPU burst and a number summarizing duration of prior CPU bursts

$$\tau_{n+1} = \alpha * t_n + (1 - \alpha) * \tau_n$$

$$\tau_{n+1} = \alpha * t_n + (1 - \alpha) * \tau_{n-1} + \dots + (1 - \alpha)^j * \tau_{n-j} + \dots + (1 - \alpha)^{n+1} * \tau_{n_0}$$

Where:

- +  $t_n$  is the actual duration of  $n^{\text{th}}$  CPU burst value for the process,
- +  $\alpha (0 \leq \alpha \leq 1)$  is a constant indicating how much to base estimate on last CPU burst, and
- +  $\tau_n$  is the estimate of CPU burst duration for time n

## Example

Given :

- ✚  $\alpha = 0.5, \tau_0 = 10$  (last estimate)
- ✚ Current (measured) CPU burst,  $t = 6$
- ✚ Will just give some reasonable guess for first  $\tau$

What is estimate of next CPU burst?

$$\tau_{n+1} = \alpha * t_n + (1 - \alpha) * \tau_n$$

$$\tau_1 = 0.5 * 6 + 0.5 * 10 = 8$$

### 3. Priority Based Scheduling

- ☞ The SJF algorithm is a special case of the general priority scheduling algorithm.
- ☞ A priority number (integer) is associated with each process
- ☞ The CPU is allocated to the process with the **highest priority** (**smallest integer = highest priority**).i.e. Schedule highest priority first.
  - All processes within same priority are handled through **FCFS**
- ☞ Priority may be determined by **user** or by **some default mechanism**
  - The system may determine the priority based on **memory requirements, time limits, or other resource usage**
- ☞ Priority scheduling can be either preemptive or non-preemptive
  - A **preemptive** approach will preempt the CPU if the priority of the newly-arrived process is higher than the priority of the currently running process
  - A **non-preemptive** approach will simply put the new process (with the highest priority) at the head of the ready queue

## Contd.

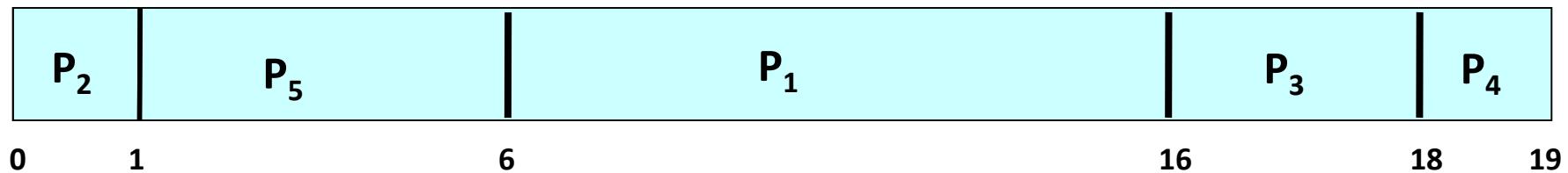
- ☞ SJF is a **priority scheduling** algorithm where priority (p) is the **inverse** of the **predicted next CPU burst time**.
  - ✚ The larger the CPU burst, the lower the priority, and vice versa.
- ☞ The main **problem** with priority scheduling is **indefinite blocking** or **starvation**, that is, low priority processes may never execute
- ☞ A **solution** is **aging**; as time progresses, the priority of a process that waits in the ready queue for long time is increased.
  - ✚ Delicate balance between giving favorable response for interactive jobs, but not starving batch jobs.
- ☞ **Note that:** There is no general agreement on whether smaller numbers represent highest priority or not.
  - ✚ Some systems use low numbers to represent low priority; others use low numbers for high priority. [but here, we assume that low numbers represent high priority].

## Example

Consider the following processes that arrive at time zero, with the length of the CPU burst and priorities given.

Process	Burst Time	Priority
P <sub>1</sub>	10	3
P <sub>2</sub>	1	1
P <sub>3</sub>	2	4
P <sub>4</sub>	1	5
P <sub>5</sub>	5	2

Using Priority scheduling, we would schedule these processes according to the following Gant chart:



$$\text{Average waiting time} = (6+0+16+18+1)/5 = 8.2$$

## Contd.

☞ Priorities can be defined either **internally** or **externally**.

1. **Internally defined priorities** use some measurable quantity or quantities to compute the priority of a process, such as time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities.
2. **External priorities** are set by criteria outside the OS, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political factors.

☞ Priority scheduling can be either **preemptive** or **non-preemptive**. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.

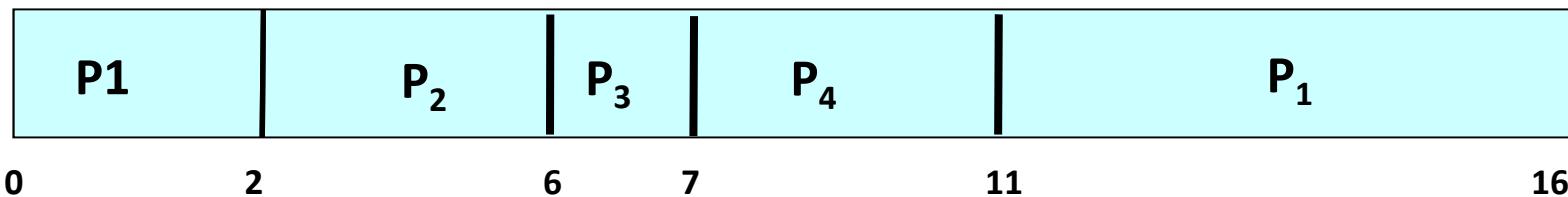
- + A **preemptive priority** scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
- + A **non-preemptive priority** scheduling algorithm will simply put the new process at the head of the ready queue.

**Contd.**

**Example:** Consider the following processes that arrive at different arrival time, with the length of the CPU burst and priorities given.

Process	Arrival time	Priority	CPU burst Time
P1	0	5	7
P2	2	1	4
P3	4	2	1
P4	5	3	4

Using preemptive Priority scheduling, we would schedule these processes according to the following Gant chart:



## 4. Round Robin (RR) Scheduling

- ☞ The **round-robin (RR)** scheduling algorithm is designed especially for **timesharing systems**.
- ☞ It is **similar** to **FCFS** scheduling, but **preemption** is added to enable the system to switch between processes.
- ☞ A small unit of time, called a **time quantum** or **time slice (from 10 to 100 milliseconds)**, is defined.
- ☞ 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
- ☞ The name of the algorithm comes from the round-robin principle known from other fields, where each person takes an equal share of something in turn.

## Contd.

- ☞ To implement RR scheduling, the ready queue is treated as a **circular FIFO queue**.
  - ✚ New processes are added to the tail of the ready queue.
- ☞ The CPU scheduler picks the **first process** from the ready queue, sets a timer to interrupt after **1 time quantum**, and dispatches the process.
- ☞ One of two things will then happen.
  1. The process may have a CPU burst of **less than 1 time quantum**.
    - ✚ In this case, the process itself will release the **CPU voluntarily**.
    - ✚ The scheduler will then proceed to the next process in the ready queue.
  2. Otherwise, if the CPU burst of the currently running process is **longer than 1 time quantum**, the timer will go off and will cause an **interrupt** to the operating system.
    - ✚ A context switch will be executed, and the process will be put at the tail of the ready queue.
- ✚ The CPU scheduler will then select the next process in the ready queue.

## Contd.

- The average waiting time under the RR policy is often long
- In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process).
  - ✚ If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue.
  - ✚ The RR scheduling algorithm is thus **preemptive**.
- The performance of the RR algorithm depends heavily on the size of the **time quantum (q)**.
  - If **q** is extremely **large**, the RR policy is the same as the FCFS policy.
  - If **q** is extremely small (say, 1 millisecond), the RR approach is called **processor sharing** and (in theory) creates the appearance that each of **n** processes has its own processor running at  $1/n$  the speed of the real processor.
    - ✚ **q** must be large with respect to **context switch**, otherwise overhead is **too high**.

## Contd.

Turnaround time also depends on the size of the time quantum.

- ☞ If the context-switch time is approximately 10 percent of the time quantum, then about 10 percent of the CPU time will be spent in context switching.
  - ❖ In practice, most modern systems have time quanta ranging from 10 to 100 milliseconds.
  - ❖ The time required for a context switch is typically less than 10 microseconds; thus, the context-switch time is a small fraction of the time quantum.

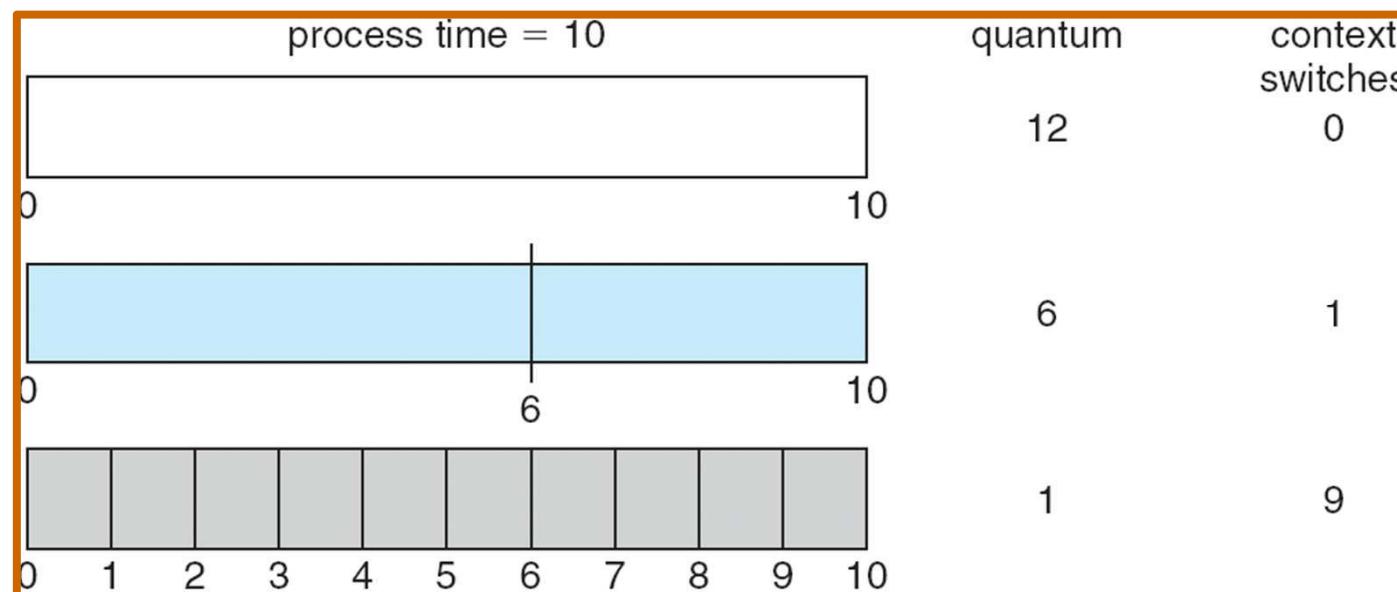


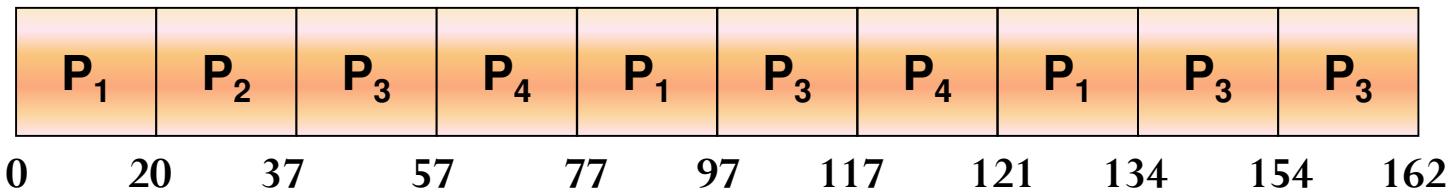
Fig. 2.3.2 How a smaller time quantum increases context switches.

Contd.

Example 1: RR with time quantum = 20

Process	Burst time
P1	53
P2	17
P3	68
P4	24

Typically, higher average **turnaround** than SJF, but better **response**

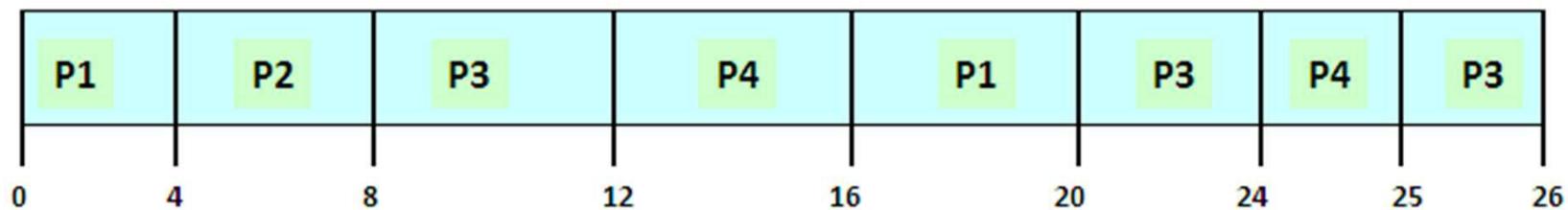


Waiting Time	Completion Time
P1: $(77-20)+(121-77) = 120$	P1: 134
P2: $(20-0) = 20$	P2: 37
P3: $(37-0)+(97-57)+(134-117) = 94$	P3: 162
P4: $(57-0)+(117-77) = 97$	P4: 121
<b>Average Waiting Time:</b> $(120+20+94+97)/4 = 82.75$	
<b>Average Completion Time:</b> $(134+37+162+121)/4 = 113.5$	

Contd.

Example 2: RR with time **quantum =4**, no priority-based preemption

Process	Arrival time	Burst time
P1	0	8
P2	1	4
P3	2	9
P4	3	5



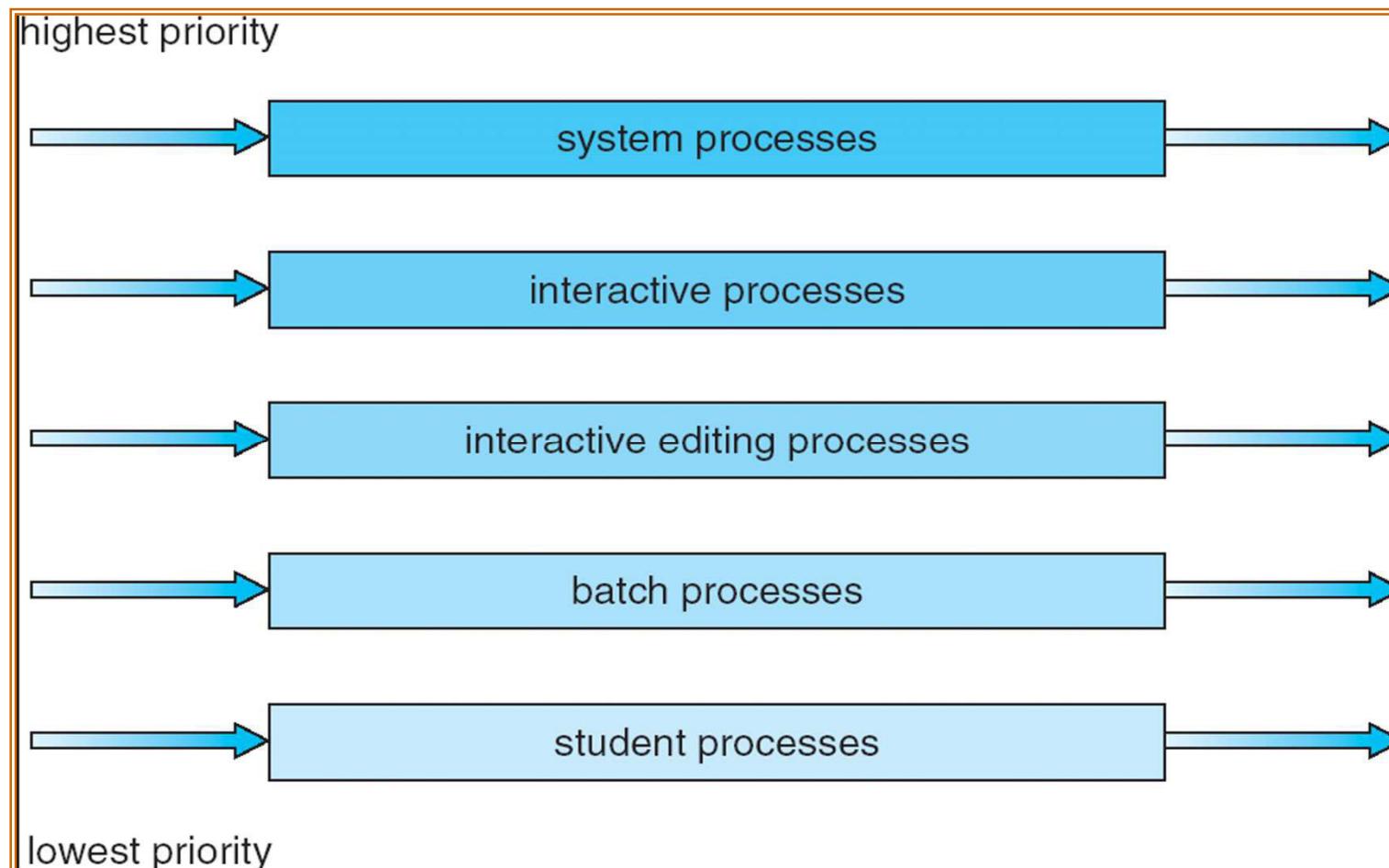
$$\text{Average wait} = ((20-0) + (8-1) + (26-2) + (25-3)) / 4 = 74 / 4 = 18.5$$

## 5. Multilevel Queue Scheduling

- ☞ Ready queue is partitioned into several separate queues:
  - ✿ foreground (interactive):- may have priority (externally defined) over background processes.
  - ✿ background (batch)
- ☞ The processes are permanently assigned to one queue, generally based on some property of the process, such as **memory size**, **process priority**, or **process type**
- ☞ Each queue has its own scheduling algorithm
  - ✿ foreground – RR
  - ✿ background – FCFS
- ☞ For Example, the foreground queue may have absolute priority over the background queue. **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

## Contd.

- For example, could separate system processes, interactive, interactive editing batch, favored, unfavored processes.



## 6. Multilevel Feedback Queue Scheduling

- ☞ When the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system.
  - ✿ If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature.
  - ✿ This setup has the advantage of **low scheduling overhead**, but it is **inflexible**.
- ☞ The multilevel feedback queue scheduling algorithm allows a process to move between queues.
- ☞ The idea is to separate processes according to the **characteristics of their CPU bursts**.
  - ✿ If a process uses too much CPU time, it will be moved to a **lower-priority** queue.
- ☞ This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
  - ✿ In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue.
- ✿ This form of **aging prevents starvation**.

## Contd.

- ☞ Multilevel-feedback-queue scheduler is defined by the following parameters:
  - ✚ number of queues
  - ✚ scheduling algorithms for each queue
  - ✚ method to determine when to upgrade a process
  - ✚ method to determine when to demote a process
  - ✚ method used to determine which queue a process will enter when that process needs service

## Example

☞ Three queues:

- Q0:-RR with time quantum 8 milliseconds
- Q1:-RR time quantum 16 milliseconds
- Q2:- FCFS

☞ Scheduling

- A new job enters queue Q0: when it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q1
- At Q1 job: it receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q2

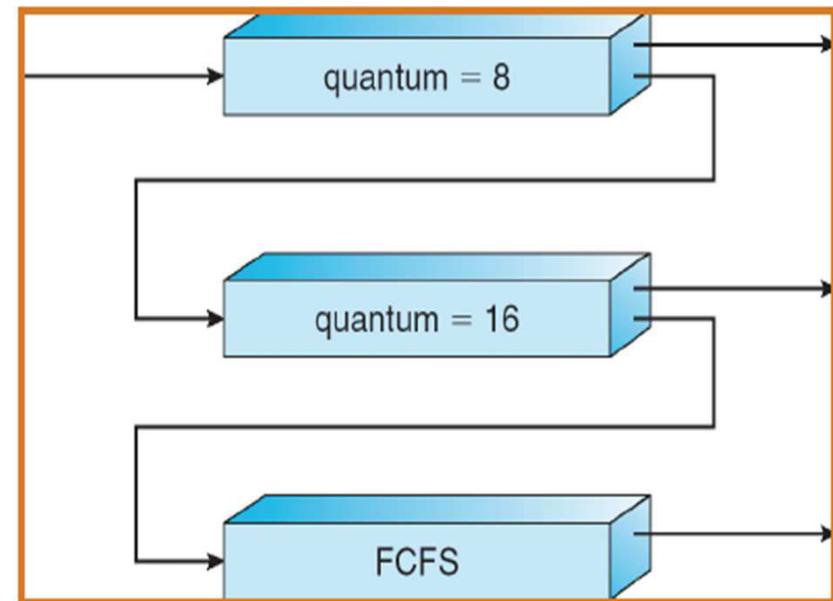


Fig. 2.3.4 Multilevel Feedback queue scheduling.

## 7. Multiple processor Scheduling

- ☞ If multiple CPUs are available, load sharing among them becomes possible; the scheduling problem becomes more complex
- ☞ We concentrate in this discussion on systems in which the processors are identical (homogeneous) in terms of their functionality
  - ✿ We can use any available processor to run any process in the queue
- ☞ Two approaches: **Asymmetric** processing and **symmetric** processing

### 1. Asymmetric Multiprocessing (ASMP)

- ❖ One processor handles all scheduling decisions, I/O processing, and other system activities
- ❖ The other processors execute only user code
- ❖ Because only one processor accesses the system data structures, the need for data sharing is reduced

## Contd.

### 2. Symmetric Multiprocessing (SMP)

- ❖ Each processor schedules itself
- ❖ All processes may be in a common ready queue or each processor may have its own ready queue
- ❖ Either way, each processor examines the ready queue and selects a process to execute
- ❖ Efficient use of the CPUs requires **load balancing** to keep the workload evenly distributed
  - ✿ In a **Push migration approach**, a specific task regularly checks the processor loads and redistributes the waiting processes as needed
  - ✿ In a **Pull migration approach**, an idle processor pulls a waiting job from the queue of a busy processor
- ❖ Virtually all modern operating systems support SMP, including **Windows XP, Solaris, Linux, and Mac OS X**

## Symmetric Multithreading(SMT)

- ❖ SMP systems allow several threads to run concurrently by providing multiple physical processors
- ❖ An alternative approach is to provide **multiple logical** rather than **physical processors**. Such a strategy is known as **symmetric multithreading (SMT)** or **hyperthreading technology**
- ❖ The idea behind SMT is to create multiple logical processors on the same physical processor
  - This presents a view of several logical processors to the operating system, even on a system with a single physical processor
  - Each logical processor has its own architecture state, which includes general-purpose and machine-state registers
  - Each logical processor is responsible for its own interrupt handling
  - However, each logical processor shares the resources of its physical processor, such as cache memory and buses.

## Contd.

- ❖ SMT is a feature provided in the hardware, not the software
  - ❖ The hardware must provide the representation of the architecture state for each logical processor, as well as interrupt handling

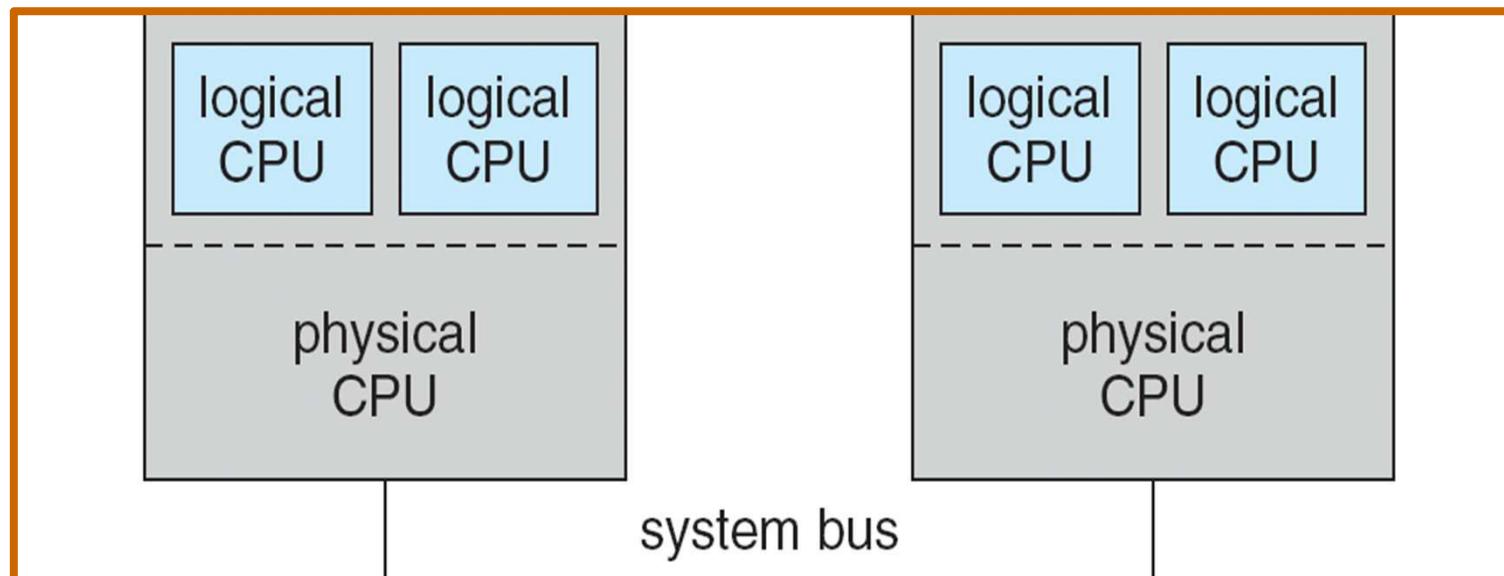


Fig. 2.3.5 A typical Symmetric Multithreading Architecture

# Scheduling Algorithms

## 1. Real-time systems

- ☞ Hard real-time systems:- required to complete a critical task within a guaranteed amount of time
- ☞ Soft real-time computing:- requires that critical processes receive priority over less fortunate ones

## 2. Multiple Processor Scheduling

- ☞ Different rules for homogeneous or heterogeneous processors
- ☞ Load sharing in the distribution of work, such that all processors have an equal amount to do.
- ☞ Each processor can schedule from a common ready queue (equal machines) OR can use a master slave arrangement

Contd.

### 3. Thread Scheduling

- ☞ Local Scheduling – How the user threads library decides which thread to put onto an available LWP (Light Weight Process)--- **process contention scope**
- ☞ Global Scheduling – How the kernel decides which kernel thread to run next

## **Operating System Examples**

- 1. Solaris scheduling**
- 2. Windows XP scheduling**
- 3. Linux scheduling**

## Solaris Scheduling

- ❖ Solaris uses priority-based thread scheduling
- ❖ It has defined four classes of scheduling (in order of priority)
  1. Real time
  2. System (kernel use only)
  3. Time sharing (the default class)
  4. Interactive
- ❖ Within each class are different priorities and scheduling algorithms

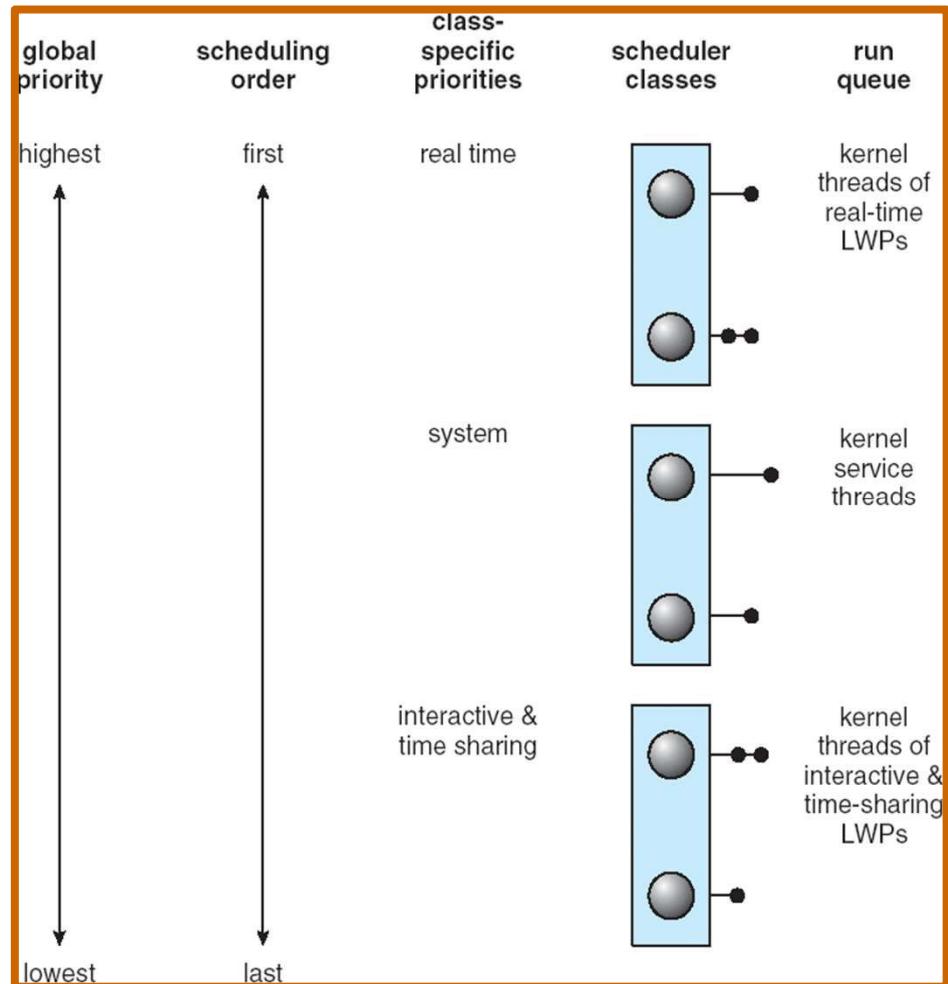


Fig. 2.3.6 Solaris Scheduling

## Contd.

- ❖ The default scheduling class for a process is time sharing
  - ❖ The scheduling policy for time sharing dynamically alters priorities and assigns time slices of different lengths using a multi-level feedback queue
- ❖ By default, there is an inverse relationship between priorities and time slices
  - ❖ The higher the priority, the lower the time slice (and vice versa)
  - ❖ Interactive processes typically have a higher priority
  - ❖ CPU-bound processes have a lower priority
  - ❖ This scheduling policy gives good response time for interactive processes and good throughput for CPU-bound processes
- ❖ The interactive class uses the same scheduling policy as the time-sharing class, but it gives windowing applications a higher priority for better performance

## Solaris Dispatch Table

- ❖ In the figure below, the priority column, a higher number indicates a higher priority

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

**Table 2.3.1** The dispatch table for scheduling interactive and time-sharing threads

## Windows XP Scheduling

- ❖ Windows XP schedules threads using a **priority-based, preemptive scheduling** algorithm
  - ❖ The scheduler ensures that the highest priority thread will always run
- ❖ The portion of the Windows kernel that handles scheduling is called the dispatcher
- ❖ A thread selected to run by the dispatcher will run until it is preempted by a higher-priority thread, until it terminates, until its time quantum ends, or until it calls a blocking system call such as I/O
- ❖ If a higher-priority real-time thread becomes ready while a lower-priority thread is running, lower-priority thread will be preempted
- ❖ This preemption gives a real-time thread preferential access to the CPU when the thread needs such access
- ❖ The dispatcher uses a **32-level priority** scheme to determine the order of thread execution

## Contd.

- ❖ Priorities are divided into two classes
  1. The variable class contains threads having priorities 1 to 15
  2. The real-time class contains threads with priorities ranging from 16 to 31
- ❖ There is also a thread running at priority 0 that is used for memory management
- ❖ The dispatcher uses a queue for each scheduling priority and traverses the set of queues from highest to lowest until it finds a thread that is ready to run
- ❖ If no ready thread is found, the dispatcher will execute a special thread called the idle thread
- ❖ There is a relationship between the numeric priorities of the Windows XP kernel and the Win32 API

## Contd.

- ❖ The Windows Win32 API identifies six priority classes to which a process can belong as shown below
  - 1. Real-time priority class
  - 2. High priority class
  - 3. Above normal priority class
  - 4. Normal priority class
  - 5. Below normal priority class
  - 6. Low priority class
- ❖ Priorities in all classes except the read-time priority class are variable
- ❖ This means that the priority of a thread in one of these classes can change

## Contd.

- ❖ Within each of the priority classes is a relative priority as shown below
- ❖ The priority of each thread is based on the priority class it belongs to and its relative priority within that class

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

## Contd.

- ☞ The initial priority of a thread is typically the base priority of the process that the thread belongs to
- ☞ When a thread's time quantum runs out, that thread is interrupted
- ☞ If the thread is in the variable-priority class, its priority is lowered
  - ▣ However, the priority is never lowered below the base priority
- ☞ Lowering the thread's priority tends to limit the CPU consumption of compute-bound threads
- ☞ When a variable-priority thread is released from a wait operation, the dispatcher boosts the priority
  - ▣ The amount of boost depends on what the thread was waiting for
  - ▣ A thread that was waiting for keyboard I/O would get a large increase
  - ▣ A thread that was waiting for a disk operation would get a moderate increase
- ☞ This strategy tends to give good response time to interactive threads that are using the mouse and windows

## Contd.

- ☞ It also enables I/O-bound threads to keep the I/O devices busy while permitting compute-bound threads to use spare CPU cycles in the background
- ☞ This strategy is used by several time-sharing operating systems, including UNIX
- ☞ In addition, the window with which the user is currently interacting receives a priority boost to enhance its response time
- ☞ When a user is running an interactive program, the system needs to provide especially good performance for that process
- ☞ Therefore, Windows XP has a special scheduling rule for processes in the normal priority class
- ☞ Windows XP distinguishes between the foreground process that is currently selected on the screen and the background processes that are not currently selected
- ☞ When a process moves in the foreground, Windows XP increases the scheduling quantum by some factor – typically by 3
- ☞ This increase gives the foreground process three times longer to run before a time-sharing preemption occurs

## Linux Scheduling

- ❖ Linux does not distinguish between processes and threads; thus, we use the term **task** when discussing the Linux scheduler
- ❖ The Linux scheduler is a **preemptive, priority-based algorithm** with two separate priority ranges
  1. A real-time range from 0 to 99
  2. A nice value ranging from 100 to 140
- ❖ These two ranges map into a global priority scheme whereby numerically lower values indicate higher priorities
- ❖ Unlike Solaris and Windows, Linux assigns higher-priority tasks longer time quanta and lower-priority tasks shorter time quanta
- ❖ The relationship between priorities and time-slice length is shown on the next slide

## Contd.

- ❖ A **runnable task** is considered eligible for execution on the CPU as long as it has time remaining in its time slice
- ❖ When a task has exhausted its time slice, it is considered expired and is not eligible for execution again until all other tasks have also exhausted their time quanta
- ❖ The kernel maintains a list of all runnable tasks in a run queue data structure
- ❖ Because of its support for SMP, each processor maintains its own run queue and schedules itself independently

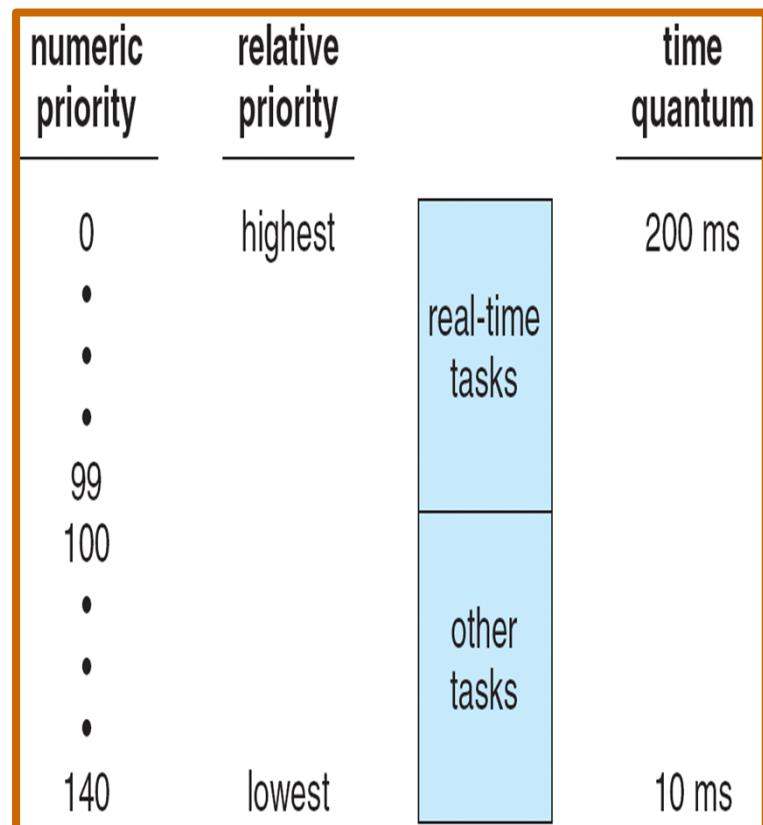


Fig. 2.3.7 The relationship between priorities and time-slice length

## Contd.

- ❖ Each **runqueue** contains two priority arrays
  - ❖ The **active array** contains all tasks with time remaining in their time slices
  - ❖ The **expired array** contains all expired tasks
- ❖ Each of these priority arrays contains a list of tasks indexed according to priority
- ❖ The scheduler selects the task with the highest priority from the active array for execution on the CPU
- ❖ When all tasks have exhausted their time slices (that is, the active array is empty), then the two priority arrays exchange roles

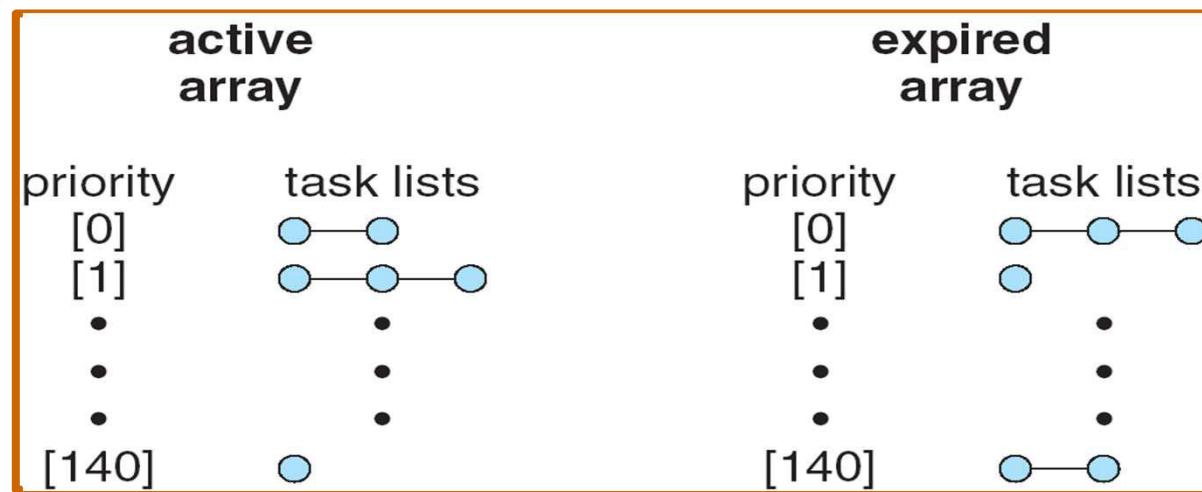


Fig. 2.3.8 Active and Expired Arrays

## Contd.

- ❖ Linux implements real-time POSIX scheduling
  - Real-time tasks are assigned static priorities
- ❖ All other tasks have dynamic priorities that are based on their nice values plus or minus the value 5
  - The interactivity of a task determines whether the value 5 will be added to or subtracted from the nice value
  - A task's interactivity is determined by how long it has been sleeping while waiting for I/O
  - Tasks that are more interactive typically have longer sleep times and are adjusted by -5 as the scheduler favors interactive tasks
  - Such adjustments result in higher priorities for these tasks
  - Conversely, tasks with shorter sleep times are often more CPU-bound and thus will have priorities lowered
- ❖ The recalculation of a task's dynamic priority occurs when the task has exhausted its time quantum and is moved to the expired array
  - ❖ Thus, when the two arrays switch roles, all tasks in the new active array have been assigned new priorities and corresponding time slices

# Algorithm Evaluation Summary

## Which algorithm is the best?

- ☞ The answer depends on many factors:
  - the system workload (extremely variable)
  - hardware support for the dispatcher
  - relative importance of performance criteria (response time, CPU utilization, throughput...)
  - The evaluation method used (each has its limitations...)
- ☞ Which one works the best is application dependent
  - General purpose OS will use priority based, round robin, preemptive
  - Real Time OS will use priority, no preemption

# Chapter Two

## Process and Thread Management

### Part Four

#### Process Synchronization

Operating Systems  
(SEng 2043)

# Objective

At the end of this session students will be able to:

- Understand the basic concepts of Process Synchronization
- Understand the Critical-Section Problem , whose solutions can be used to ensure the consistency of shared data.
- Familiar with Critical-Section problem solutions: Software, Hardware, Operating System and Programming Language solutions.
- Semaphores
- Classic Problems of Synchronization
- Monitors
- To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity.

# Introduction to Process Synchronization

Concurrent Processes can be

## 1. Independent processes

☞ cannot affect or be affected by the execution of another process.

## 2. Cooperating processes

☞ is one that can affect or be affected by other processes executing in the system.

☞ Cooperating processes can either directly **share a logical address space** (that is, both code and data) or be allowed to share data only through **files** or **messages**.

☞ Concurrent access to shared data may lead to **data inconsistency**.

☞ Concurrent execution requires **process communication and process synchronization**

## Contd.

- ☞ If there is no controlled access to shared data, some processes will obtain an inconsistent view of the shared data
  - Consider two processes P1 and P2, accessing shared data. While P1 is updating data, it is preempted (because of timeout, for example) so that P2 can run. Then P2 try to read the data, which are partly modified, which will result in **data inconsistency**
- ☞ In such cases, the outcome of the action performed by concurrent processes will then depend on the order in which their execution is interleaved
- ☞ Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

## Synchronization in Producer-Consumer Problem

- Producer process produces information that is consumed by a consumer process.
- The previous solution considered the use of a **bounded buffer**
  - Producer produce items and puts it into the buffer
  - Consumer consumes items from the buffer
- Producer and Consumer must **synchronize**.
- 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 **count** that keeps track of the number of full buffers.
  - Initially, **count** is set to 0.
  - It is **incremented** by the producer after it produces a new buffer
  - It is **decremented** by the consumer after it consumes a buffer

# Producer-Consumer Problem

## Producer

```
while (true)
{
    /* produce an item and put in
    nextProduced */
    while (count == BUFFER_SIZE)
        ; // do nothing
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```

## Consumer

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

- Although both producer and consumer routines are correct when executed separately, they may not properly work when executed concurrently.
- The statements **count++;** and **count--;** must be performed *atomically*.
- Atomic/Indivisible operation means an operation that completes its entirety without interruption.

## Contd.

☞ To illustrate this, let's assume the current value of the count variable is 5 and that the producer and consumer processes execute the statements **counter++** and **counter--** concurrently.

☞ Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6!

❖ The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately.

☞ **count++** could be implemented as (in a typical machine language):

**register1 = count**

**register1 = register1 + 1**

**count = register1**

☞ **count--** could be implemented as (in a typical machine language):

**register2 = count**

**register2 = register2 - 1**

**count = register2**

## Contd.

- ☞ The concurrent execution of **count++** and **count--** is equivalent with the execution of the statements of each operations where arbitrary interleaving of the statements occur.
- ☞ A simple arbitrary interleaving of the operations can be as follows:

**S0:** producer execute **register1 = count** {**register1 = 5**}

**S1:** producer execute **register1 = register1 + 1** {**register1 = 6**}

**S2:** consumer execute **register2 = count** {**register2 = 5**}

**S3:** consumer execute **register2 = register2 - 1** {**register2 = 4**}

**S4:** producer execute **count = register1** {**count = 6** }

**S5:** consumer execute **count = register2** {**count = 4**}

- ☞ At **S5**, we have arrived at the value of counter=4, if we reversed **S4** and **S5**, we would have arrived at count=6
- ☞ We would arrive at the incorrect value of the count variable because we allowed both processes to manipulate the variable concurrently.

## Race Condition

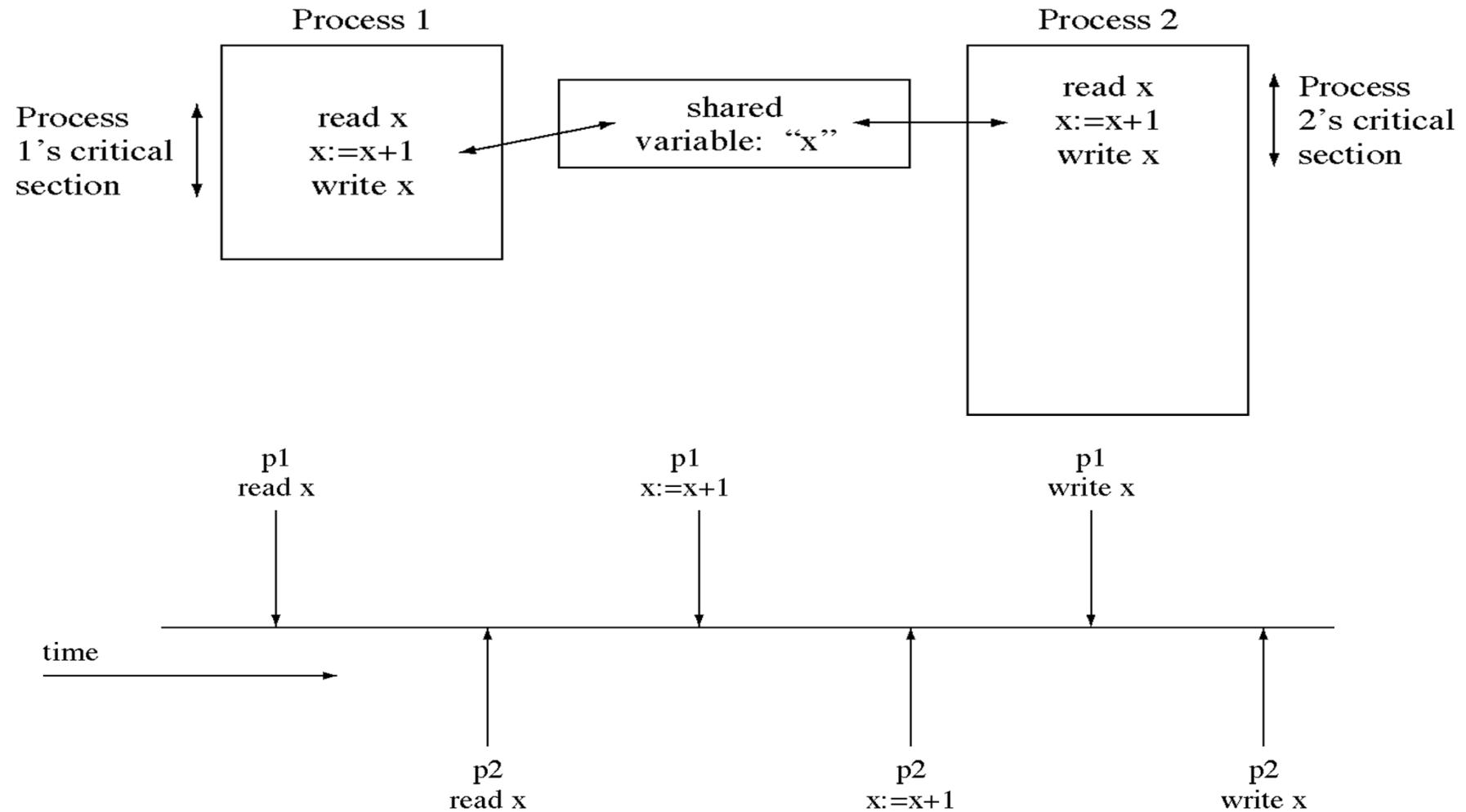
☞ **Race condition** is a situation where several processes access and manipulate the shared data concurrently and the outcome of the execution depends on the particular order in which the access takes place.

➤ The final value of the shared data depends upon which process finishes last

☞ The key to preventing trouble here and in many other situations involving shared memory, shared files, and shared everything else is to find some way to prohibit more than one process from reading and writing the shared data at the same time

☞ To prevent race conditions, concurrent processes must coordinate or be **synchronized.**

## Example: Race condition updating a variable



## The Critical-Section Problem

- n processes competing to use some shared data
- Each process has a code segment, called **Critical-Section (CS)**, in which the process may be changing common variables, updating a table, writing a file, and so on.
- A critical section is a piece of code in which a process or thread accesses a common resource
  - So each process must first request permission to enter its critical section. The section of code implementing this request is called the **Entry Section (ES)**
  - The critical section (CS) might be followed by a **Leave/Exit Section (LS)**
  - The remaining code is the **Remainder Section (RS)**

Contd.

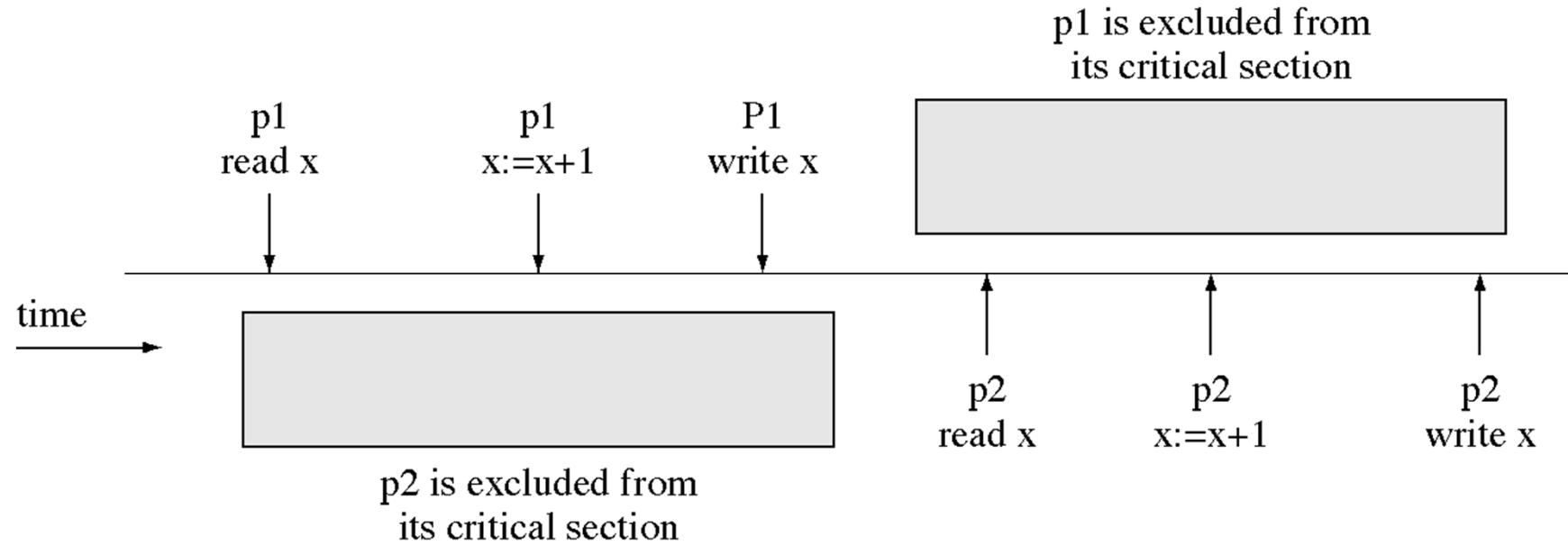
```
do {  
    Entry section  
    critical section  
    Exit section  
    remainder section  
} while (TRUE);
```

Fig. General structure of a typical process

**Problem:-** ensure that when one process is executing in its CS, no other process is allowed to execute in its CS

- ☞ When a process executes code that manipulates shared data (or resource), we say that the process is in it's Critical Section (for that shared data)
- ☞ The execution of critical sections must be **mutually exclusive**: at any time, only one process is allowed to execute in its critical section (**even with multiple processors**)

## Critical section to prevent a race condition



- Multiprogramming allows logical parallelism, uses devices efficiently but we lose correctness when there is a race condition.
- So we forbid logical parallelism inside critical section so we lose some parallelism but we regain correctness.

## Solution to Critical-Section Problem

A solution to a critical section problem must satisfy the following three requirements:

**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 only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection can not be postponed indefinitely.

**3. Bounded Waiting:-** A **bound** or a **limit** 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.



Assume that each process executes at a nonzero speed

## Solution to CS Problem – Mutual Exclusion

**Mutual Exclusion** – If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections.

### Requirements for Mutual Exclusion

- ❖ No assumption may be made about speeds or the number of CPUs.
- ❖ No process running outside its critical region may block other processes.
- ❖ No process should have to wait forever to enter its critical region.
- ❖ A process must not be delayed access to a critical section when there is no other process using it
- ❖ If a process somehow halts/waits in its critical section, it must not interfere with other processes.
- ❖ A process remains inside its critical section for a finite time only

**Contd.**

## **Advantages**

1. Applicable to any number of processes on either a single processor or multiple processors sharing main memory
2. It is simple and therefore easy to verify
3. It can be used to support multiple critical sections

## **Disadvantages**

1. Busy-waiting consumes processor time
  2. Starvation is possible when a process leaves a critical section and more than one process is waiting.
  3. Deadlock
- If a low priority process has the critical region and a higher priority process needs, the higher priority process will obtain the processor to wait for the critical region.

## Types of solutions to CS problem

**1. Software solutions:-** algorithms who's correctness does not rely on any other assumptions.

**Example:-** Lock variable, Peterson's algorithm, Semaphores, Monitors, Dekker's algorithm, Lamport's bakery algorithm, The black-white bakery algorithm and Szymanski's Algorithm

**2. Hardware solutions:-** rely on some special machine instructions.

- ❖ An indivisible test-and-set of a flag could be used in a tight loop to wait until the other processor clears the flag. When the code leaves the critical section, it clears the flag.

**Example:-** Lock Mechanism, Test-and-set, Atomic Swap, Exclusive access to memory location, Interrupts that can be turned off

**3. Operating System solutions:-** provide some functions and data structures to the programmer through system/library calls.

**4. Programming Language solutions:-** Linguistic constructs provided as part of a language.

## 1. Lock Variable

It is a software solution which uses a single, shared (lock)variable, initialized to 0.

- ☞ When a process wants to enter its critical region, it first tests the lock.
- ☞ If the lock is 0, the process sets it to 1 and enters the critical region.
- ☞ If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.
- ☞ Unfortunately, this idea contains exactly the same fatal flaw that we saw in the spooler directory. Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time.

## Contd.

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
}  
while (TRUE);
```

- Suppose that one process reads the lock and sees that it is 0.
- Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1.
- When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time.

## 2. Peterson's Solution

- ❖ It is a concurrent programming algorithm for mutual exclusion that allows two processes to share a single-use resource without conflict, using **only shared memory for communication.**
- ❖ While Peterson's original formulation worked with only two processes, the algorithm can be generalized for more than two processes.
- ❖ The algorithm uses two variables, **flag** (Boolean type) and **turn** (integer type).
  - ❖ A **flag[n]** value of true indicates that the process n wants to enter the critical section.
  - ❖ The variable **turn** holds the ID of the process whose turn it is. i.e. **flag[i]=true** implies that process  $P_i$  is ready to enter the critical section.
  - ❖ Entrance to the critical section is granted for process  $P_0$  if  $P_1$  does not want to enter its critical section or if  $P_1$  has given priority to  $P_0$  by setting turn to 0.

## Contd.

```
flag[i] = false;  
flag[j] = false;  
turn;
```

### Algorithm for Process P<sub>i</sub>

```
while (true)  
{  
  
    flag[i] = true;  
    turn = j;  
  
    while ( flag[j]==true && turn == j);  
  
    CRITICAL SECTION  
  
    flag[i] = FALSE;  
  
    REMAINDER SECTION  
}
```

### Algorithm for Process P<sub>j</sub>

```
while (true)  
{  
  
    flag[j] = true;  
    turn = i;  
  
    while ( flag[i]==true && turn == i);  
  
    CRITICAL SECTION  
  
    flag[j] = FALSE;  
  
    REMAINDER SECTION  
}
```

## Contd.

### ❖ Mutual Exclusion is preserved:

- ❖ Let us assume  $P_i$  execute in its critical section.
- ❖  $P_i$  can only execute in its Critical Section only if
  - either  $\text{flag}[j] = \text{false}$  or  $\text{turn} = i$

❖ If  $P_1$  and  $P_2$  could not have successfully executed their **while statement** because the value of turn can either be 1 or 2 but cannot be both.

❖ Hence, only one process can execute the **while statement** successfully.

❖ Say  $P_1$  executes it successfully then  $P_2$  will spin on the while loop as long as the  $P_1$  executes its Critical Section.

**Thus, mutual exclusion is preserved.**

❖ In addition to this the **progress requirement is satisfied** and the **bounded-waiting requirement is met**.

### 3. Semaphores

- ☞ Semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: **wait ()** and **signal ()**.
  - ✚ It is used for signaling
- ☞ Semaphore is a variable that has an integer value that may be initialized to a nonnegative number.
  - ✚ Wait operation decrements the semaphore value
  - ✚ Signal operation increments semaphore value
  - ✚ Wait and signal operations cannot be interrupted.
    - That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

```
wait (S) {  
    while (S <= 0)  
        ; // no-op  
    S--;  
}  
signal (S) {  
    S++;  
}
```

## Semaphores as General Synchronization Tool

Operating systems often distinguish between counting and binary semaphores

### 1. Counting semaphore:- integer value can range over an unrestricted domain

- ☞ Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
- ☞ Each process that wishes to use a resource performs a **wait()** operation on the semaphore (thereby decrementing the count).
- ☞ When a process releases a resource, it performs a **signal()** operation.
- ☞ When the **count** for the semaphore goes to **0**, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

### 2. Binary semaphore:- integer value can range only between 0 and 1; can be simpler to implement

- ☞ Also known as **mutex locks** as they are locks that provide **mutual exclusion**
- ☞ Can implement a counting semaphore **S** as a binary semaphore
- ☞ We can use binary semaphores to deal with the critical-section problem for multiple processes.

## Semaphore Implementation

- ❖ Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- ❖ Thus, 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 of the critical section 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 S;  
// initialized to 1  
wait (S);  
Critical Section  
signal (S);
```

## 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:

1. **Block**:- place the process invoking the operation on the appropriate waiting queue.
2. **Wakeup**:- remove one of the processes in the waiting queue and place it in the ready queue.

#### Implementation of wait:

```
wait (S){  
    value--;  
    if (value < 0) {  
        add this process to waiting  
        queue  
        block(); }  
    }  
-----
```

#### Implementation of signal:

```
Signal (S){  
    value++;  
    if (value <= 0) {  
        remove a process P from the  
        waiting queue  
        wakeup(P); }  
    }
```

## Semaphore Problems

- ❖ Semaphores provide a powerful tool for enforcing mutual exclusion and coordinate processes
  - But wait (`S`) and signal (`S`) are scattered among several processes. Hence, difficult to understand their effects
- ❖ Usage must be correct in all the processes (correct order, correct variables, no omissions)
  - One bad (or malicious) process can fail the entire collection of processes

## Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

(Reading Assignment)

## Bounded-Buffer Problem

- ❖ The buffer pool contains **N** buffers, each can hold one item
- ❖ Semaphore **mutex** initialized to the value 1
  - allows mutual exclusion for accesses to the buffer pool
- ❖ Semaphore **full** initialized to the value 0
  - counts the number of full buffers
- ❖ Semaphore **empty** initialized to the value **N**
  - counts the number of empty

### Producer Process:

```
while (true) {
    // produce an item
    wait (empty);
    wait (mutex);
    // add the item to the buffer
    signal (mutex);
    signal (full);
}
```

### Consumer Process:

```
while (true) {
    wait (full);
    wait (mutex);
    // remove an item from buffer
    signal (mutex);
    signal (empty);
    // consume the removed item
}
```

## 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**:- to allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.

- Shared Data
- Data set
- Semaphore **mutex** initialized to 1
- Semaphore **wrt** initialized to 1
- Integer **readcount** initialized to 0

**Writer Process:**

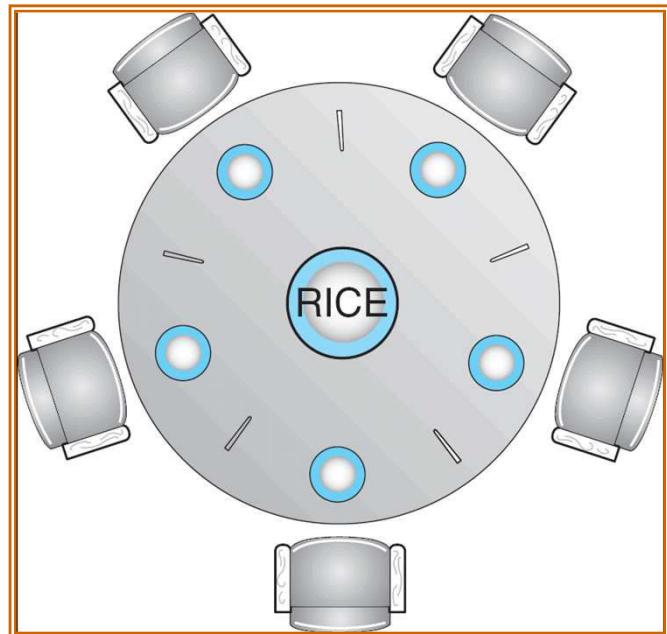
```
while (true) {  
    wait (wrt);  
    // writing is performed  
    signal (wrt);  
}
```

**Reader Process:**

```
while (true) {  
    wait (mutex);  
    readcount ++;  
    if (readcount == 1)  
        wait (wrt);  
        signal (mutex)  
    // reading is performed  
    wait (mutex);  
    readcount --;  
    if (readcount == 0)  
        signal (wrt);  
        signal (mutex);  
}
```

# Dining-Philosophers Problem

- Shared data
- Bowl of rice (**data set**)
- Semaphore **chopstick [5]** initialized to 1



Problems with  
Semaphores

30

The structure of Philosopher i:

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

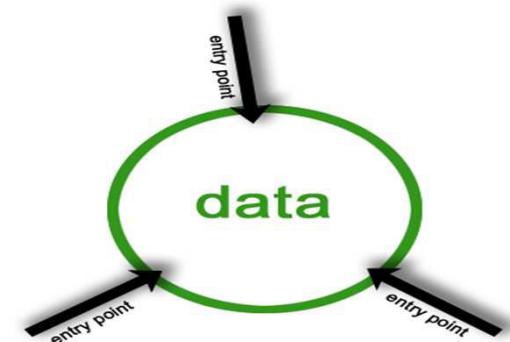
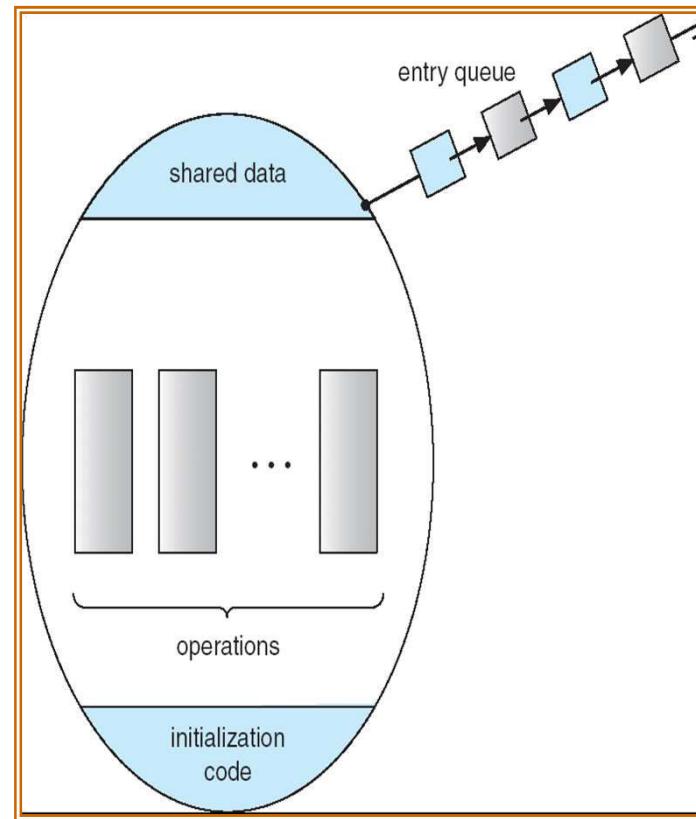
- Incorrect use of semaphore operations:
- signal (mutex) .... wait (mutex)
- wait (mutex) ... wait (mutex)
- Omitting of wait (mutex) or signal (mutex) (or both)

## 4. Monitors

- ☞ A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- ☞ A monitor is a software module consisting of sets of **procedures** and **local data** variables
- ☞ The main characteristics of monitors are:
  1. Local variables of a monitor are only accessible by the monitor's procedures (no external procedure can access variables of the monitor)
  2. Only one process may be active within the monitor at a time
  3. A process enters the monitor by invoking one of its procedures
  4. By only allowing one process to execute at a time, the monitor provides a mutual exclusion facility
    - Thus, a shared data structure can be included as part of a monitor and will be protected
    - If the data in the monitor represents a shared resource, then the monitor provides a mutual exclusion facility on the

Contd.

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



## Monitors : Condition Variables

- ☞ A monitor supports synchronization by the use of **condition variables** that are contained within the monitor and accessible only within the monitor
- ☞ Condition variables are a special data type in monitors,  
**Condition x, y;**
- ☞ The only operations that can be invoked on condition variables are **wait()** and **signal()**
- ☞ The operation **x.wait ()** means a process that invokes the operation is suspended until another process invokes the **x.signal()** operation
- ☞ **x.signal ()**:- resumes one of the suspended processes.
- ☞ If there are no suspended operations, then the operation will have no effect
- ☞ The wait() and signal() operations in monitor and semaphore are different:
  - ☞ If a process in a monitor signals and no task is waiting on the condition variable, the signal is lost.
  - ☞ However, in semaphores, the signal operation will always have effect on the state/value of the semaphore variable.

## Contd.

- If process P invokes `x.signal ()`, with Q in `x.wait ()` state, what should happen next?

**If Q is resumed, then P must wait**

Options include

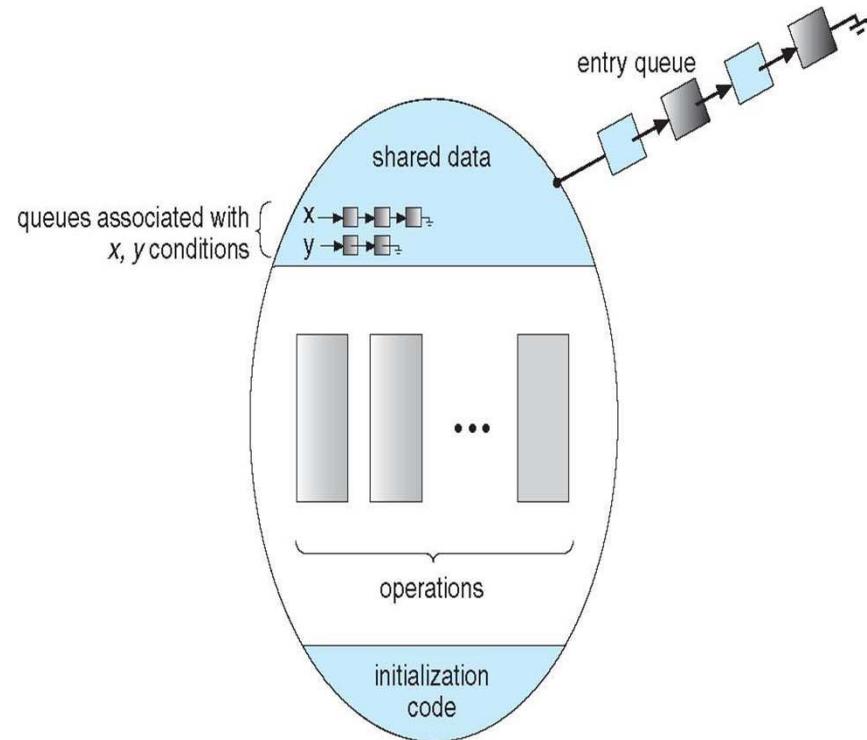
**Signal and wait** – P waits until Q leaves monitor or waits for another condition

**Signal and continue** – Q waits until P leaves the monitor or waits for another condition

- Both have pros and cons – language implementer can decide

- For example Monitors implemented in concurrent Pascal compromise

- **P executing signal immediately leaves the monitor, Q is resumed**



**Fig.** Monitor with Condition Variables

# Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
    enum {THINKING, HUNGRY, EATING} state [5];
    condition self [5];

    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self [i].wait;
    }

    void putdown (int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    void test (int i) {
        if ( (state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) ) {
            state[i] = EATING;
            self[i].signal ();
        }
    }

    initialization_code() {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}
```

## Reading Assignments

- Hardware based solutions
  - **Swap**
  - **Test and Set**
  - **Lock**
- What is the difference between signal () and wait () operations in Semaphores and monitors?

# Chapter Two

## Process and Thread Management

### Part Five

#### Process Synchronization

Operating Systems  
(SEng 2043)

# Objective

At the end of this session students will be able to understand:

- ✚ System Model
- ✚ Deadlock Characterization
- ✚ Methods for handling Deadlock
  - Deadlock Prevention
  - Deadlock Avoidance
  - Deadlock Detection
  - Recovery from Deadlock

## System Model

- ☞ A system contains a finite number of resource types ( $R_1, R_2, \dots, R_m$ ) to be distributed among competing processes
- ☞ The resource types are partitioned into several types (e.g. files, I/O devices, CPU cycles, memory), each having a number of identical instances.
- ☞ If a process requests an instance of a resource type, the allocation of **any instance** of the type will satisfy the request.
- ☞ If it will not, then the instances are not identical, and the resource type classes have not been defined properly.
- ☞ A process must request a resource before using it and must release it after making use of it.

## Contd.

- ☞ A process may request as many resources as it requires to carry out its designated task.
  - The number of resources requested may not exceed the total number of resources available in the system.
- ☞ In other words, a process cannot request three printers if the system has only two.

### 1. Request

- A process requests for an instance of a resource type.
  - If the resource is **free**, the request will be **granted**.
  - Otherwise the process should **wait** until it acquires the resource

### 2. Use

- The process uses the resource for its operations

### 3. Release

- The process releases the resource

## Deadlock

- ☞ For each use of a kernel-managed resource by a process or thread, the operating system checks to make sure that the process has requested and has been allocated the resource.
- ☞ A **system table** records whether each resource is free or allocated.
  - For each resource that is allocated, the table also records the process to which it is allocated.
  - If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.
- ☞ A set of processes are in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.
- ☞ Deadlock can be defined as a **permanent blocking of processes** that either compete for system resources or communicate with each other
  - The set of blocked processes each hold a resource and wait to acquire a resource held by another process in the set.
  - All deadlocks involve **conflicting needs for resources** by two or more processes

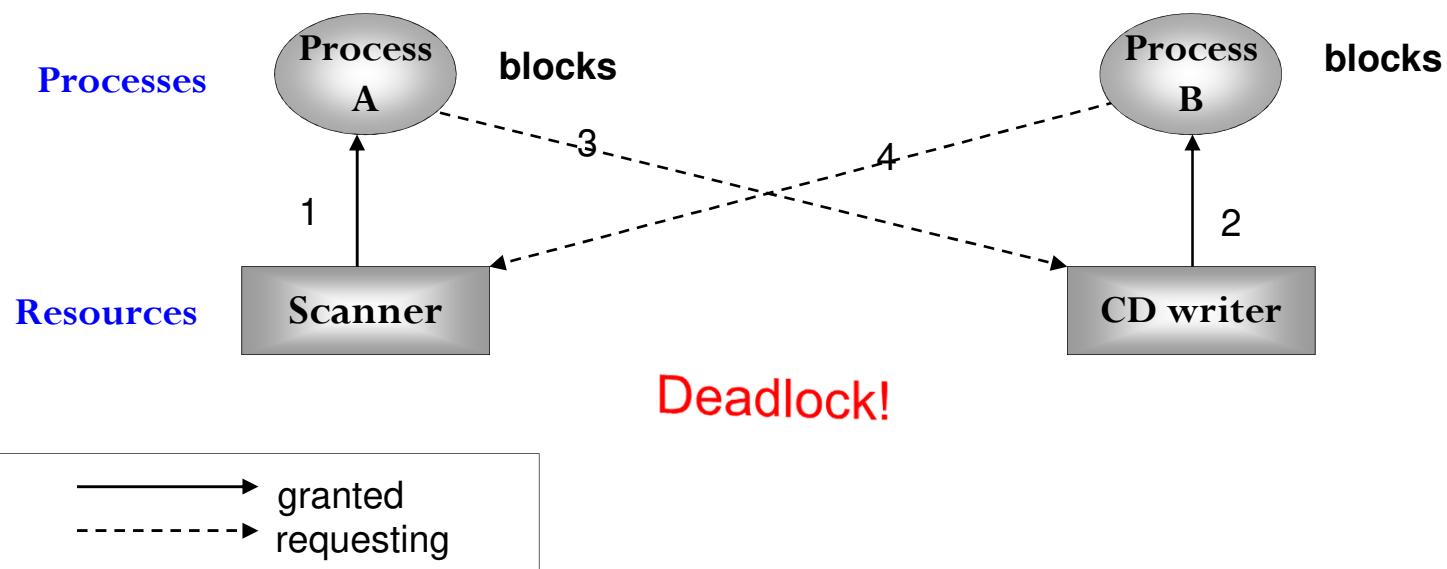
Contd.

### Example 1

- Suppose a system has 2 disk drives
- If P1 is holding disk 2 and P2 is holding disk 1 and if P1 requests for disk 1 and P2 requests for disk 2, then deadlock occurs

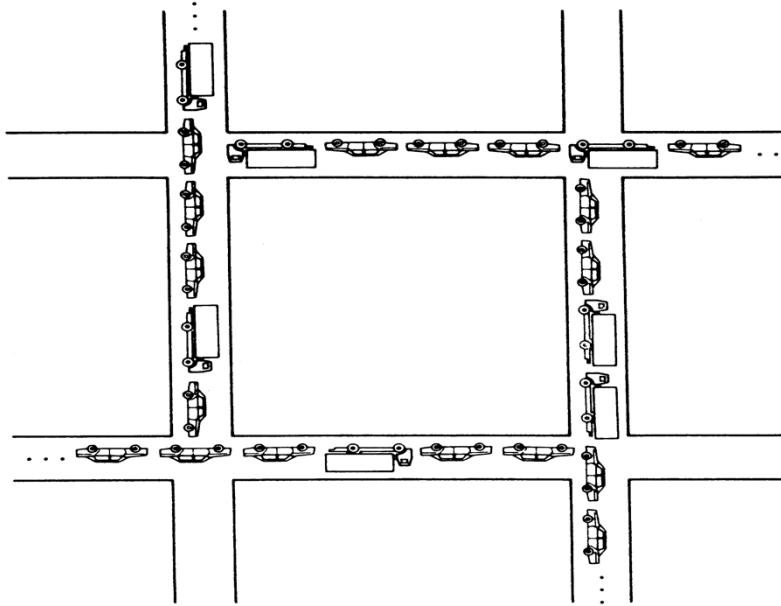
### Example 2

- A is programmed to request Scanner first and then CD Writer
- B is programmed to request CD Writer first and then Scanner



Contd.

**Example 3: Traffic Jam**



**Example 4: Dining Philosophers**



## Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously in a system:

1. **Mutual exclusion**:- only one process at a time can use a resource (**non-shareable**).
  - No process can access a resource unit that has been allocated to another process
2. **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes.
3. **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. **Circular wait**: 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  $P_2, \dots,$
  - $P_{n-1}$  is waiting for a resource that is held by  $P_n$ , and
  - $P_n$  is waiting for a resource that is held by  $P_0$

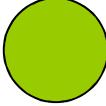
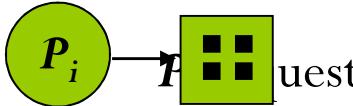
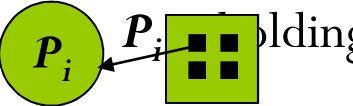
## Resource Allocation Graph (RAG)

- ☞ Deadlock can better be described by using a directed graph called **system resource allocation graph**
- ☞ The graph consists of a set of **vertices V** and a set of **edges E**
- ☞ V is partitioned into two types:
  - $P = \{P_1, P_2, \dots, P_n\}$ , the set consisting of all the **processes** in the system.
  - $R = \{R_1, R_2, \dots, R_m\}$ , the set consisting of all **resource types** in the system.
- ☞ The edge E can be of two types:
  - **Request edge:-** directed edge  $P_i \rightarrow R_j$       Process  $P_i$  has requested  $R_j$

If a Resource Allocation Graph contains a **cycle**, then a **deadlock** may exists.  
 $P_i$ .

## Contd.

Diagrammatically, processes and resources in RAG are represented as follow:

- 
-  Process
-  Requests for
-  Holding

### Note that

- Resources having more than one instance are represented as a dot within the rectangle.
- A request edge points to only the rectangle  $R_j$ , whereas an assignment edge must also designate one of the dots in the rectangle.

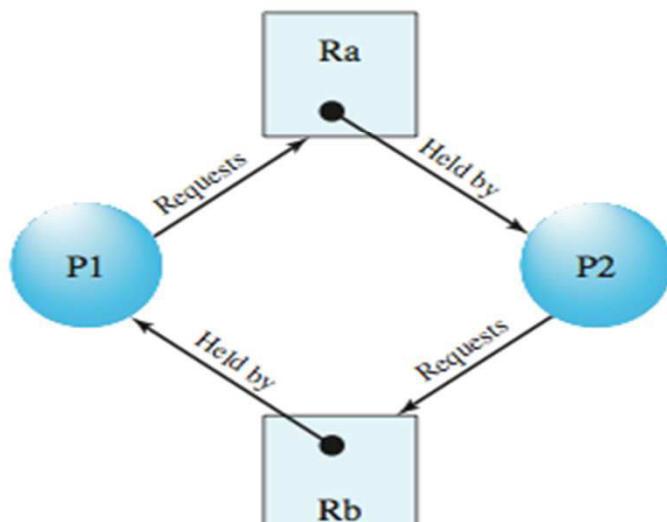
## Example of Resource Allocation Graph



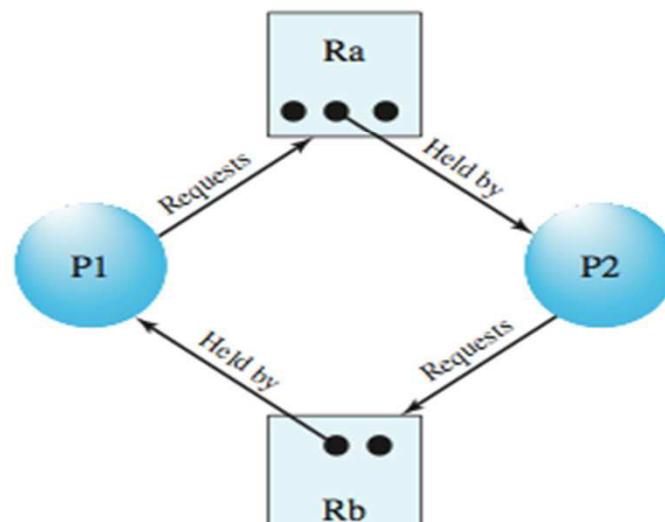
(a) Resource is requested



(b) Resource is held



(c) Circular wait



(d) No deadlock

Fig.2.5.1 Resource Allocation Graph Examples

## Graph With A Cycle But No Deadlock

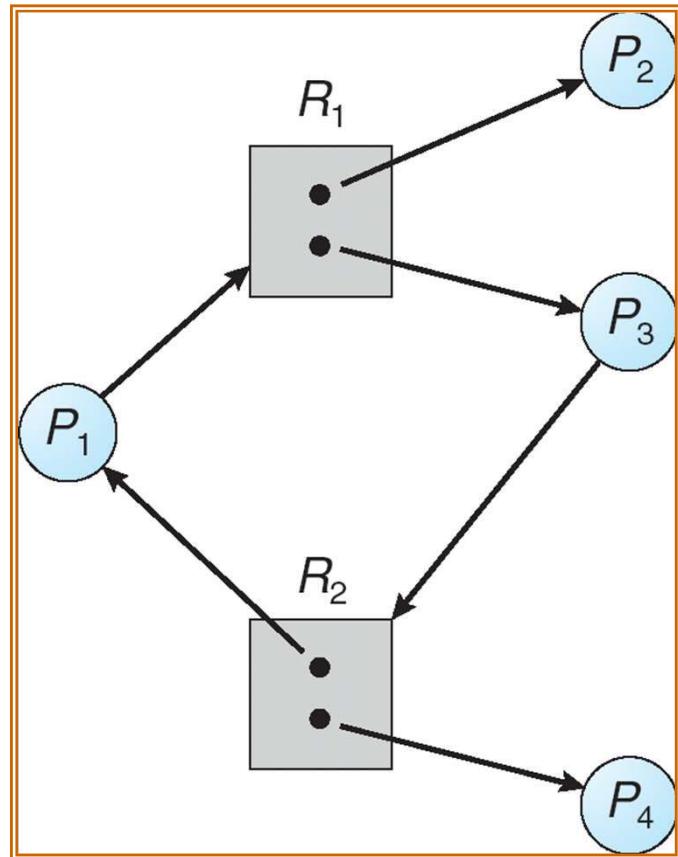


Fig.2.5.2 RAG graph with a cycle  
but no deadlock

### Basic Facts

- If graph contains no cycles then there's **no deadlock**
- If however a graph contains a cycle then there are two possible situations:
  - If there is only one instance per resource type, then deadlock can occur
  - If there are several instances per resource type, there's a **possibility of deadlock**

## Example of Resource Allocation Graph

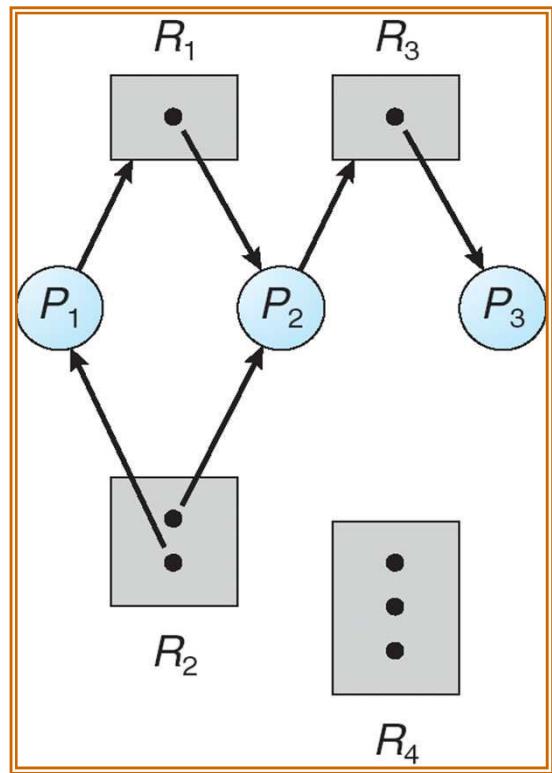


Fig.2.5.3 RAG graph



The RAG shown here tells us about the following situation in a system:

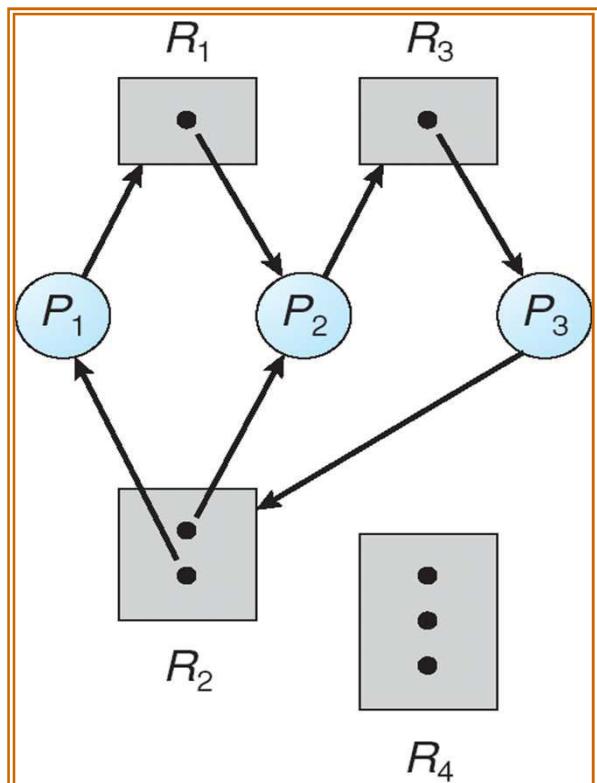
- $P = \{P_1, P_2, P_3\}$
- $R = \{R_1, R_2, R_3, R_4\}$
- $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_1, P_3 \rightarrow R_1, P_2 \rightarrow R_2, P_1 \rightarrow R_3, P_3 \rightarrow R_3\}$

**The process states**

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

Contd.

### Resource Allocation Graph With a Deadlock



There are two cycles in this graph



$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_1$



$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

Fig.2.5.4 RAG graph with a deadlock ➤

Processes  $P_1$ ,  $P_2$  and  $P_3$  are **deadlocked**:

- $P_1$  is waiting for  $P_2$  to release  $R_1$
- $P_2$  is waiting for  $R_3$  held by  $P_3$  and

## Methods for handling Deadlocks

Deadlock problems can be handled in one of the following **three** ways:

1. Using a protocol that prevents or avoids deadlock by ensuring that a system will never enter a deadlock state, **deadlock prevention** and **deadlock avoidance scheme** are used.
2. Allow the system to enter a deadlock state, detect it and then recover.  
**(Deadlock Recovery)**
3. Ignore the problem and pretend that deadlocks never occur in the system;
  - Used by most operating systems, including **UNIX** and **Windows**.
  - It is then up to the application developer to write programs that handle deadlocks.

## Deadlock Prevention

- By ensuring at least one of the necessary conditions for deadlock will not hold, deadlock can be prevented.
  - This is mainly done by restraining how requests for resources can be made
- Deadlock prevention methods fall into **two classes**:
  1. An **indirect method** of deadlock prevention prevents the occurrence of one of the three necessary conditions listed previously (items 1 through 3).  
i.e. **Mutual exclusion, Hold and wait and No preemption**
  2. A **direct method** of deadlock prevention prevents the occurrence of a circular wait (item 4, **Circular wait**)

## Contd.

1. **Mutual Exclusion:-** This is not required for sharable resources; however to prevent a system from deadlock, **the mutual exclusion condition must hold for non-sharable resources**
2. **Hold and Wait:-** in order to prevent the occurrence of this condition in a system, we must guarantee that **whenever a process requests a resource, it does not hold any other resources.**

**Two protocols are used to implement this:**

- A. Require a process to request and be allocated all its resources before it begins execution or
- B. Allow a process to request resources only when the process has none

**Both protocols have two main disadvantages:**

- Since resources may be allocated but not used for a long period, **resource utilization will be low**
- A process that needs several popular resources has to wait indefinitely because one of the resources it needs is allocated to another process. **Starvation is possible.**

## Contd.

### 3. No Preemption:- If a process holding certain resources is denied further request, that process must release its original resources allocated to it

- ☞ If a process requests a resource allocated to another process waiting for some additional resources, and the requested resource is not being used, then the resource will be preempted from the waiting process and allocated to the requesting process
- ☞ Preempted resources are added to the list of resources for which the process is waiting
- ☞ **Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting**
- ☞ This approach is practical to resources whose state can easily be saved and retrieved easily (e.g. CPU registers )

### 4. Circular Wait:- A linear ordering of all resource types is defined and each process requests resources in an increasing order of **enumeration**

- ☞ So, if a process initially is allocated instances of resource type **R**, then it can subsequently request instances of resource types following **R** in the ordering.

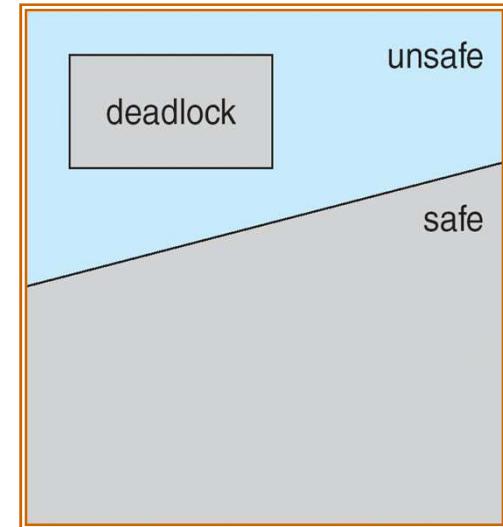
## Deadlock Avoidance

- ☞ Deadlock avoidance scheme requires each process to declare the **maximum number of resources** of each type that it may need in advance.
- ☞ Having this full information about the sequence of requests and release of resources, we can know whether or not the system is entering **unsafe state**.
- ☞ The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a **circular-wait** condition
- ☞ Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes
- ☞ A **state is safe** if the system can allocate resources to each process in some order (safe sequence) avoiding a deadlock.
  - A deadlock state is an unsafe state.

# Safe, Unsafe and Deadlock States

## Basic Facts

- ☞ If a system is in a **safe state**, then there are **no deadlocks**
- ☞ If a system is in **unsafe state**, then there is a **possibility of deadlock**
- ☞ **Deadlock avoidance** method ensures that a system will never enter an unsafe state



**Example:** Consider a system with 12 tape drives. Assume there are three processes : P1, P2, P3.

Assume we know the **maximum** number of tape drives that each process may request:P1: 10, P2: 4, P3: 9

Suppose at time  $t_{now}$ , 9 tape drives are **allocated** as follows : P1: 5, P2 : 2, P3 : 2

**So, we have three more tape drives which are free**

- This system is in a safe state because we have a safe allocation sequence of processes: <P2, P1, P3>, then P2 can get two more tape drives and it finishes its job, and returns four tape drives to the system
- Then the system will have 5 free tape drives.
  - Allocate all of them to P1, it gets 10 tape drives and finishes its job.
  - P1 then returns all 10 drives to the system.
- Then P3 can get 7 more tape drives and it does its job

## Deadlock Avoidance Algorithms

- ☞ Based on the concept of safe state, we can define algorithms that ensures the system will never deadlock
- ☞ **Idea:-** Ensure that the system will always remain in a safe state.
  - Initially, the system is in a safe state.
  - Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait.
  - The request is granted only if the allocation leaves the system in a safe state.
- ☞ If there is a **single instance of a resource type**,
  - Use a **Resource-Allocation Graph**
- ☞ If there are **multiple instances of a resource type**,
  - Use the **Dijkstra's banker's algorithm**.

## Deadlock Avoidance Algorithms using RAG

- ☞ A new type of edge (**Claim edge**), in addition to the request and assignment edge is introduced
- ☞ Claim edge  $P_i \rightarrow R_j$  indicates that process  $P_i$  may request resource  $R_j$  at some point in the future. The edge resembles a request edge but is represented by a dashed line in the graph
  - **Claim edge** is converted to **request edge** when a process requests a resource
  - **Request edge** is converted to an **assignment edge** when the resource is allocated to the process
  - When a resource is released by a process, assignment edge reconverts to a claim edge
  - If no cycle exists in the allocation, then system is in safe state otherwise the system is in unsafe state
- ☞ Resources must be claimed a priori in the system
  - i.e, before a process starts executing, all its claim edge must show up in the allocation graph.

## Contd.

- Suppose that process  $P_i$  requests a resource  $R_j$ , we allocate it if changing the request edge  $P_i \rightarrow R_j$  to an assignment edge  $R_j \rightarrow P_i$  will not form a cycle in the allocation graph
- The request can be granted only if converting the request edge to an assignment edge does not lead to the formation of a cycle in the res

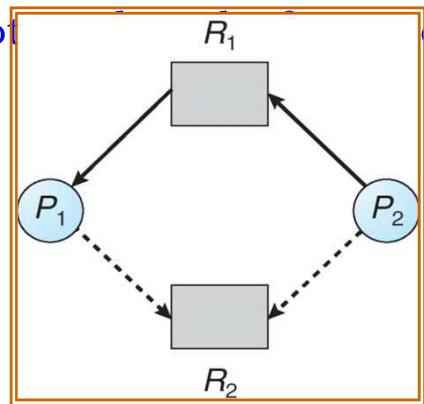


Fig.2.5.5 a RAG for deadlock avoidance.

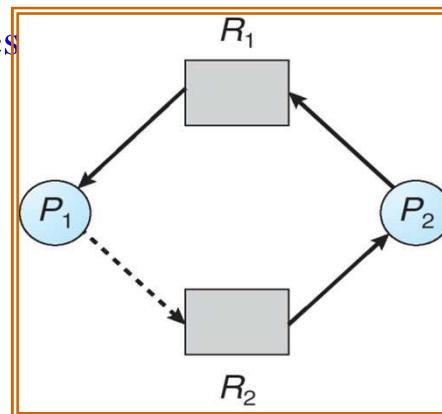


Fig.2.5.5 b .An unsafe state in a RAG.

- If we suppose  $P_2$  requests  $R_2$  /  $P_1$  requests  $R_2$
- Although  $R_2$  is currently free, we cannot allocate it to  $P_2$ , since this action will create a cycle in the graph Fig.2.5.5 b .
  - A cycle indicates that the system is in an **unsafe state**.
  - If  $P_1$  requests  $R_2$ , and  $P_2$  requests  $R_1$ , then a deadlock will occur.

## Deadlock Avoidance Algorithms : Banker's Algorithm

- ☞ This algorithm is used when there are **multiple instances of resources**
- ☞ When a process enters a system, it must declare the **maximum number** of each instance of resource types it may need
  - The number however may not exceed the total number of resource types in the system
- ☞ When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state.
  - If it will, the resources are allocated; otherwise, the process must wait 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 is **less efficient** than the resource-allocation graph scheme.

## Contd.

The following data structures are used in the algorithm:

Let  $n$  = number of processes and

$m$  = number of resources types

**1. Available:-** Vector of length  $m$ . It indicates the number of available resources of each type

- If **available** [ $j$ ] =  $k$ , there are  $k$  instances of resource type  $R_j$  available

**2. Max:-**  $n \times m$  matrix that defines the maximum demand of each process from each resource type

- If **Max** [ $i, j$ ] =  $k$ , then process  $P_i$  may request at most  $k$  instances of resource type  $R_j$ .

**3. Allocation:-**  $n \times m$  matrix which defines the **number of resources of each type currently allocated to each process**

- If **Allocation** [ $i, j$ ] =  $k$  then  $P_i$  is currently allocated  $k$  instances of  $R_j$

## Contd.

4. **Need:-**  $n \times m$  matrix that indicates the **remaining need** of each process, of each resource type

- If  $\text{Need } [i, j] = k$ , then  $P_i$  may need  $k$  more instances of  $R_j$  to complete its task

$$\text{Need } [i, j] = \text{Max } [i, j] - \text{Allocation } [i, j]$$

5. **Request:**  $n \times m$  matrix which indicates the **pending requests** of each process, of each resource type.

☞ These data structures **vary over time in both size and value**.

Define the  $\leq$  relation between two vectors  $X$  and  $Y$ , of **equal size  $= n$**  as :

$$X \leq Y \Leftrightarrow X[i] \leq Y[i], i=1,2,\dots,n$$

$$X \neq Y \Leftrightarrow X[i] > Y[i] \text{ for some } i$$

- For example, if  $X = (1, 7, 3, 2)$  and  $Y = (0, 3, 2, 1)$ , then  $Y \leq X$ .

In addition,  $Y < X$  if  $Y \leq X$  and  $Y \neq X$ .

## Contd.

The algorithm is as follows:

1. Process  $P_i$  makes requests for resources.
  - o Let **Request (i)** be the corresponding request vector.
  - o So, if  $P_i$  wants  $k$  instances of resource type  $R_j$ , then **Request (i) [j] = k**
2. If **Request (i) !≤ Need (i)**, there is an **error**, Since the process has exceeded its maximum claim
3. Otherwise, if **Request (i) !≤ Available**, then  $P_i$  must wait
4. Otherwise, Modify the data structures as follows:
  - o **Available = Available – Request (i)**
  - o **Allocation (i) = Allocation (i) + Request (i)**
  - o **Need (i) = Need (i) – Request (i)**
5. Check whether the resulting **state is safe**. (Use the safety algorithm presented in the next slide)
6. If the **state is safe, do the allocation**.
  - o Otherwise,  $P_i$  must wait for Request(i)

## Safety Algorithm

- It is used to identify whether or not a system is in a safe state.
- The algorithm can be described as follow:

1. Let **Work** and **Finish** be vectors of length **m** and **n**, respectively.

Initialize:

**Work = Available**

**Finish [i] = false for i = 0, 1, ..., n- 1.**

2. Find an index **i** such that both:

**Finish [i] = false**

**Need<sub>i</sub> < Work**

If no such **i** exists, go to **step 4.**

3. **Work = Work + Allocation<sub>i</sub>**

**Finish [i] = true**

**go to step 2**

4. If **Finish [i]==true** for all **i**, then the system is in a **safe state**

## Resource-Request Algorithm

- ☞ This algorithm determines if a request can be safely granted. The algorithm is described below:
- ☞ Let  $\text{Request}_i$  be the request vector for process  $P_i$ . If  $\text{Request}_i[j] = k$  then process  $P_i$  wants  $k$  instances of resource type  $R_j$
- ☞ When a request for resources is made by process  $P_i$  the following actions are taken:
  1. If  $\text{Request}_i \leq \text{Need}_i$ , go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
  2. If  $\text{Request}_i \leq \text{Available}$ , go to step 3. Otherwise  $P_i$  must wait, since resources are not available.
  3. Pretend to allocate requested resources to  $P_i$  by modifying the state as follows:
    - $\text{Available} = \text{Available} - \text{Request}_i$
    - $\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i$
    - $\text{Need}_i = \text{Need}_i - \text{Request}_i$
    - o If safe  $\Rightarrow$  the resources are allocated to  $P_i$
    - o If unsafe  $\Rightarrow$   $P_i$  must wait, and the old resource-allocation state is restored.

## Examples of Banker's algorithm

**Example 1:-** Assume a system has

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

↙ Snapshot at time  $T_0$ :

<u>Process</u>	<u>Allocation</u>			<u>Max</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	7	5	3	3	3	2
$P_1$	2	0	0	3	2	2			
$P_2$	3	0	2	9	0	2			
$P_3$	2	1	1	2	2	2			
$P_4$	0	0	2	4	3	3			

- The system is in a **safe state** since the sequence  
 $\langle P_1, P_3, P_4, P_2, P_0 \rangle$  satisfies safety criteria
- What about  $\langle P_1, P_3, P_0, P_2, P_4 \rangle$  ?

The content of the matrix *Need* is defined to be  $Max - Allocation$

<u>Process</u>	<u>Need</u>
	A B C
$P_0$	7 4 3
$P_1$	1 2 2
$P_2$	6 0 0
$P_3$	0 1 1
$P_4$	4 3 1

## Contd.

**Example 1:-**  $P_1$  requests  $(1, 0, 2)$ ;

- Check that Request Available (that is,  $(1,0,2) \leq (3,3,2)$ )  $\Rightarrow$

<u>Process</u>	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	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	

- Executing safety algorithm shows that sequence  $< P_1, P_3, P_4, P_0, P_2 >$  satisfies safety requirement.
- Can request for  $(3,3,0)$  by  $P_4$  be granted? **No**  $\rightarrow$  resources are not available
- Can request for  $(0,2,0)$  by  $P_0$  be granted? **No**  $\rightarrow$  even if the resources are available, it leads to unsafe state

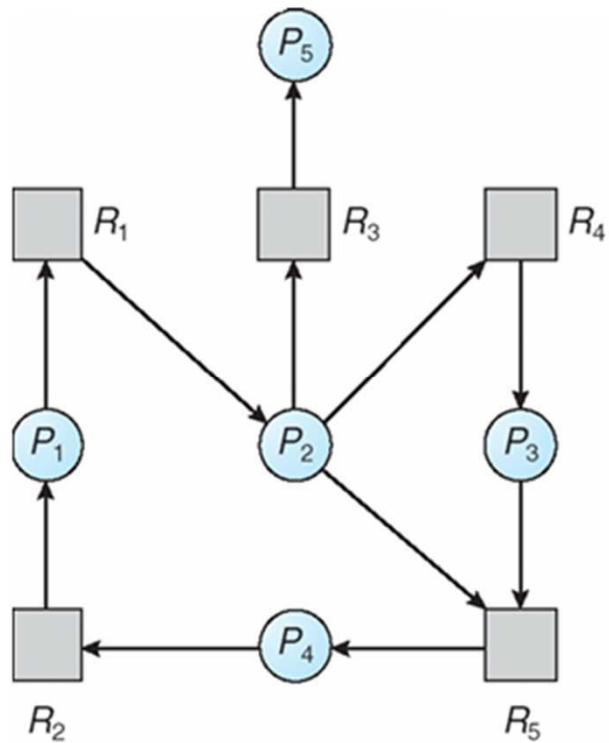
## Deadlock Detection

- ☞ If a system does not employ either a deadlock prevention or avoidance algorithm, then a deadlock situation may occur.
- ☞ In this environment, the system may provide:
  - A **deadlock detection** algorithm that examines the state of the system if there is an occurrence of deadlock
  - An algorithm to recover from the deadlock (**deadlock Recovery scheme**)

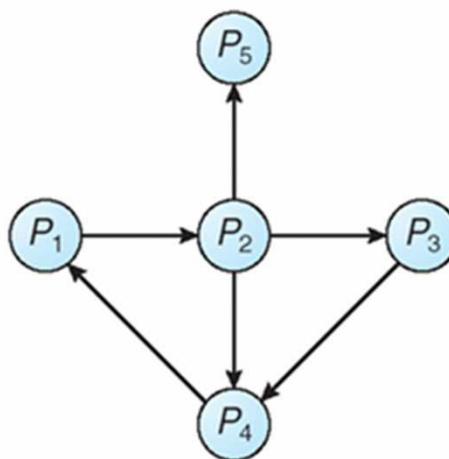
## Deadlock Detection: Single Instance of Each Resource Type

- ☞ If there are single instances of each resources in a system, then an algorithm that uses a type of resource allocation graph called **wait-for graph** will be used
- ☞ **The wait-for graph is obtained from the resource allocation graph by removing the resource nodes and collapsing the corresponding edges**
- ☞ If a process  $P_i$  points to  $P_j$  in a **wait-for graph**, it indicates that  $P_i$  is waiting for  $P_j$  to release a resource  $P_i$  needs
- ☞ To detect deadlocks, the system needs to periodically invoke an algorithm that searches for a cycle in the graph.
  - **If there is a cycle, there exists a deadlock.**
- ☞ An algorithm to detect a cycle in a graph requires an order of  $n^2$  operations, where  $n$  is the number of vertices in the graph

Contd.



(a) Resource-allocation graph



(b) Corresponding wait-for graph.

## Deadlock Detection: Several Instance of Each Resource Type

- ☞ When there are **multiple instances** of a resource type in a resource allocation system, **the wait-for graph is not applicable.**
- ☞ Hence, a deadlock detection algorithm is used
- ☞ The algorithm uses **several data structures** similar to the ones in banker's algorithm

**1. Available:-** A vector of length  $m$  indicates the number of available resources of each type

**2. Allocation:-** An  $n \times m$  matrix defines the number of resources of each type currently allocated to each process

**3. Request:-** An  $n \times m$  matrix indicates the current request of each process.

- If  $\text{Request}[i,j] = k$ , then process  $P_i$  is requesting  $k$  more instances of resource type  $R_j$

## Deadlock Detection Algorithm

1. Let *Work* and *Finish* be vectors of length  $m$  and  $n$ , respectively. Initialize:

- a.  $\text{Work} = \text{Available}$
- b. For  $i = 1, 2, \dots, n-1$ , if  $\text{Allocation}_i \neq \emptyset$ , then  
 $\text{Finish}[i] = \text{false}$ ; otherwise,  $\text{Finish}[i] = \text{true}$

2. Find an index  $i$  such that both:

- (a)  $\text{Finish}[i] == \text{false}$
- (b)  $\text{Request}_i \leq \text{Work}$

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

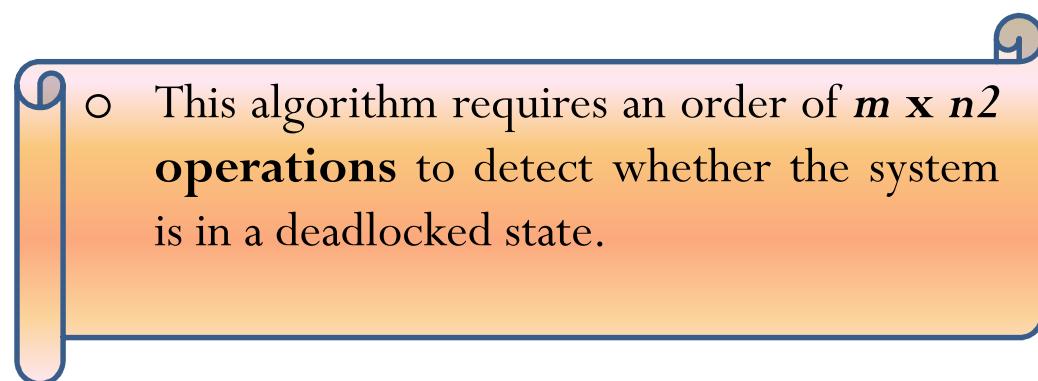
3.  $\text{Work} = \text{Work} + \text{Allocation}_i$

$\text{Finish}[i] = \text{true}$

go to step 2

4. If  $\text{Finish}[i] == \text{false}$ , for some  $i, 1 \leq i \leq n$ , then the system is in **deadlock state**.

Moreover, if  $\text{Finish}[i] == \text{false}$ , then  $P_i$  is deadlocked



## Contd.

**Example 1:-** Assume that there are **Five** processes  $P_0$  through  $P_4$ ; and **three** resource types A (7 instances), B (2 instances), and C (6 instances)

- Snapshot at time  $T_0$ :

<u>Process</u>	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	0	0	0	0	0	0
$P_1$	2	0	0	2	0	2			
$P_2$	3	0	3	0	0	0			
$P_3$	2	1	1	1	0	0			
$P_4$	0	0	2	0	0	2			

Sequence  $\langle P_0, P_2, P_3, P_1, P_4 \rangle$  will result in  $Finish[i] = \text{true}$  for all  $I$

## Contd.

**Example 2:-** Assume that  $P_2$  requests additional instances of resource C.

- The Request matrix is modified as follows:

<u>Process</u>			
<u>Request</u>			
	A	B	C
$P_0$	0	0	0
$P_1$	2	0	2
$P_2$	0	0	1
$P_3$	1	0	0
$P_4$	0	0	2

State of system?

- Can reclaim resources held by process  $P_0$ , but insufficient resources to fulfill other processes' requests.
- **Deadlock exists**, consisting of processes  $P_1$ ,  $P_2$ ,  $P_3$ , and  $P_4$

## Deadlock Detection Algorithm Usage

- ☞ When, and how often, to invoke the detection algorithm depends on:
  1. How often a deadlock is likely to occur?
  2. How many processes will be affected (need to be rolled back)?
    - If deadlock occurs frequently, then the algorithm is invoked frequently,
      - Resources allocated to deadlocked processes will be **idle until the deadlock can be broken**
      - **The number of processes in the deadlock may increase**
    - If the algorithm is invoked for every resource request not granted, it will incur a computation time **overhead** on the system
- ☞ If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock

## Recovery From Deadlock

- ☞ Once a deadlock has been detected, recovery strategy is needed.
- ☞ There are two possible recovery approaches:
  - **Process termination** and
  - **Resource preemption**

### Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated

**Question:** In which order should we choose a process to abort?

**Answer:** Choose the process with

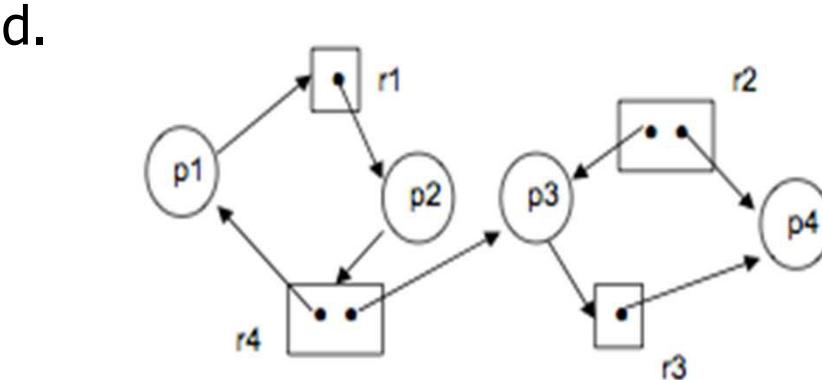
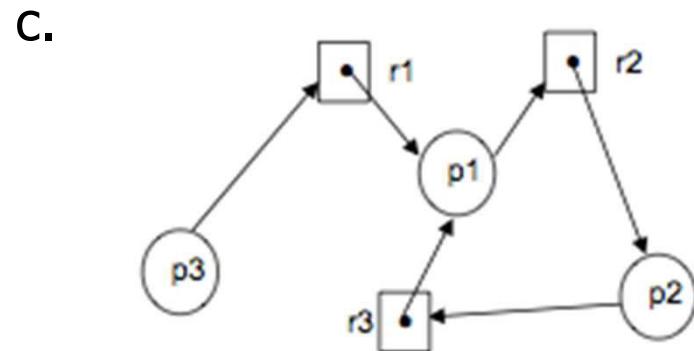
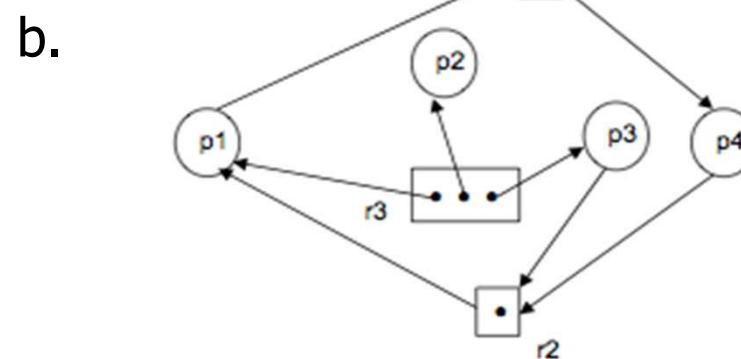
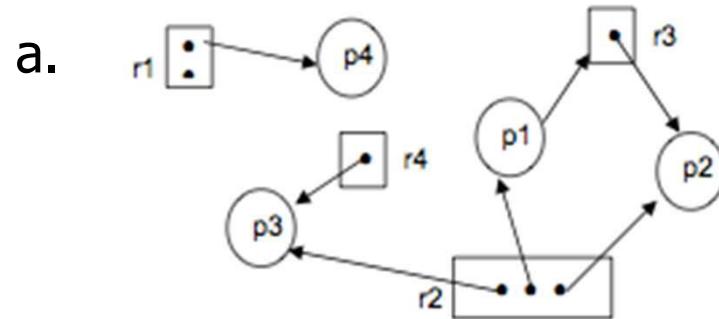
- Least amount of processor time consumed so far
- Least amount of output produced so far
- Most estimated time remaining
- Least total resources allocated so far
- Lowest priority

## Process Preemption

- ☞ In this recovery strategy, we successively preempt resources and allocate them to another process until the deadlock is broken
- ☞ While implementing this strategy, there are **three** issues to be considered
  - 1. Selecting a victim:-** which resources and process should be selected to minimize cost just like in process termination.
    - The cost factors may include parameters like the number of resources a deadlocked process is holding, number of resources it used so far
  - 2. Rollback:-** if a resource is preempted from a process, then it can not continue its normal execution
    - The process must be rolled back to some safe state and started
  - 3. Starvation:-** same process may always be picked as victim several times. As a result, starvation may occur.
    - The best solution to this problem is to only allow a process to be picked as a victim for a limited finite number of times.
    - This can be done by including the number of rollback in the cost factor

## Exercises

1. For each of the resource allocation graphs below, determine whether there is a deadlock or not. Give explanations for each of your answers.



## Contd.

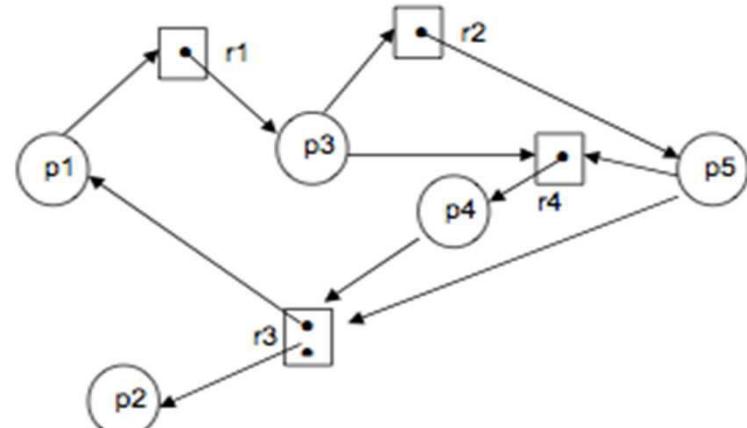
2. There are four processes in a system and they are going to share nine tape drives.

Their current and maximum number of allocation numbers are as follows :

Process	Allocation	Maximum
P1	3	6
P2	1	2
P3	4	9
P4	0	2

- Is the system in a safe state? Why or why not?
- Is the system deadlocked? Why or why not?

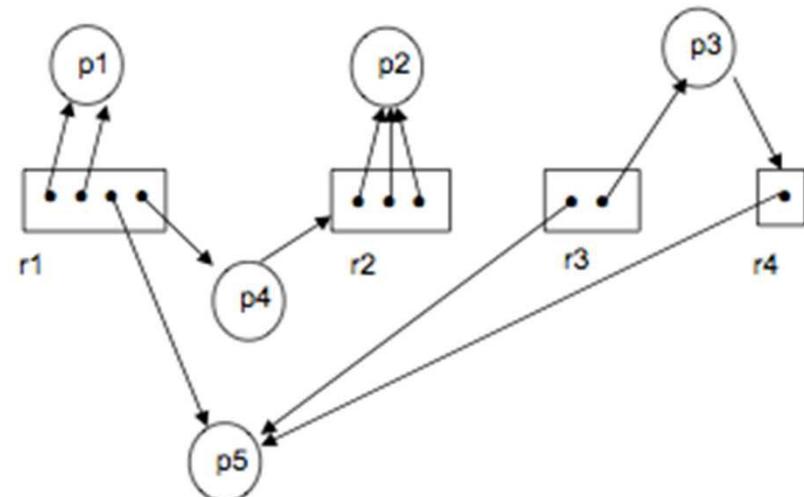
3. Explain if the system in the given resource allocation graph here is deadlocked. If not, give an execution order of the processes which successfully terminates



## Contd.

4. Given the resource allocation graph below:

- Apply the deadlock detection algorithm and either indicate why the system is deadlocked, or specify a safe allocation sequence
- If the process P2 also request 2 instances of resource r1, does the system enter a deadlock? Why?



5. Consider the following snapshot of a system:

Process	Allocation	Max	Available
	A B C D	A B C D	A B C D
P1	0 0 1 2	0 0 1 2	1 5 2 0
P2	1 0 0 0	1 7 5 0	
P3	1 3 5 4	2 3 5 6	
P4	0 6 3 2	0 6 5 2	
P5	0 0 1 4	0 6 5 6	

- Use the banker's algorithm to answer the following questions:
  - What is the content of the matrix Need?
  - Is the system in a safe state?
  - If a request from process P1 arrives for (0,4,2,0), can the request be granted immediately?

# Chapter Three

## Memory Management

**Part One**

**Main Memory**

**Operating Systems**  
**(SEng 2043)**

# Objective

At the end of this session students will be able to:

- Understand the basics of Memory management and its Requirements
- Understand the detailed description of various ways of organizing memory hardware.
- Discuss about the various memory-management techniques:
  - Swapping
  - Continuous Memory Allocation
  - Paging and
    - Structure of the Page Table
  - Segmentation.

# Introduction

- ☞ Program must be brought (from disk) into memory and placed within a process for it to be run.
- ☞ **Input Queue** contains list of processes on the disk that are waiting to be brought into memory for execution.
- ☞ **Main memory** and **registers** are only storage CPU can access directly.
  - Register access in one CPU clock (or less).
  - Main memory can take many cycles.
- ☞ Memory management is the task carried out by the OS (called **memory manager**) and hardware to accommodate multiple processes in main memory.
- ☞ Memory needs to be allocated efficiently to pack as many processes into memory as possible.

**Cache Memory** sits between main memory and CPU registers.

Protection of memory required to ensure correct operation

## Problem

- How to manage relative speed of accessing physical memory
- How to Ensure correct operation to protect the operating system from being accessed by user process and user processes from one another.

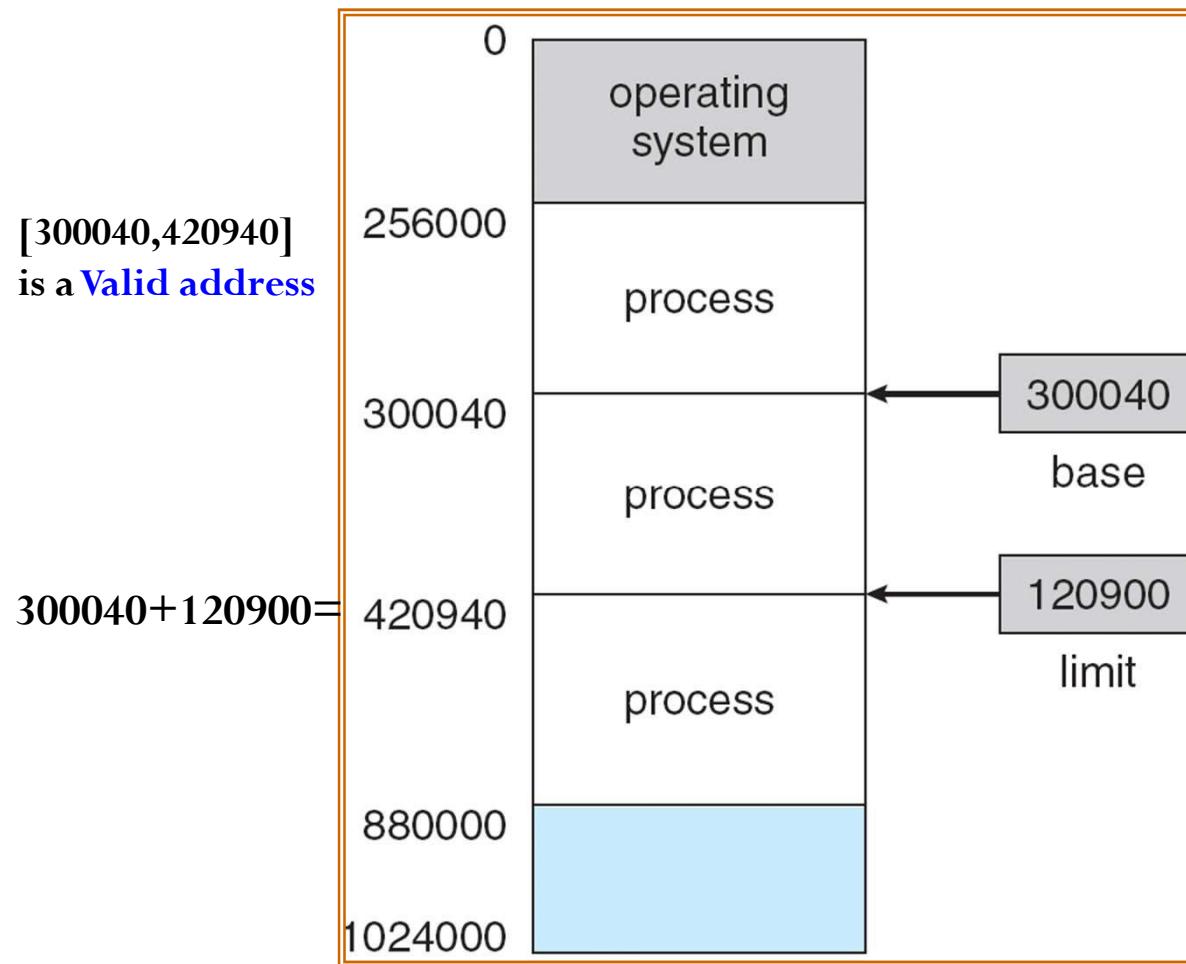
## One possible solution for the above problem

- ☞ Make sure that each process has a **separate address space**.
  - Determine the legal addresses that the process can access legally
  - Ensure that a process can access only these legal addresses.
- ☞ This protection can be done using **two registers**
  - 1. **Base registers**: - holds the smallest legally accepted physical address.
  - 2. **Limit registers**:-specifies the range.
    - Use a HW to compare every address generated in user space with registers value
- ☞ A trap is generated for any attempt by a user process to access beyond the limit.
- ☞ Base and limit registers are loaded only by the OS, using special privileged instructions(**what's its advantage?**)

## Base and Limit Registers

OS have unrestricted access to both OS & users' memory

A pair of **base** and **limit registers** define the logical address space

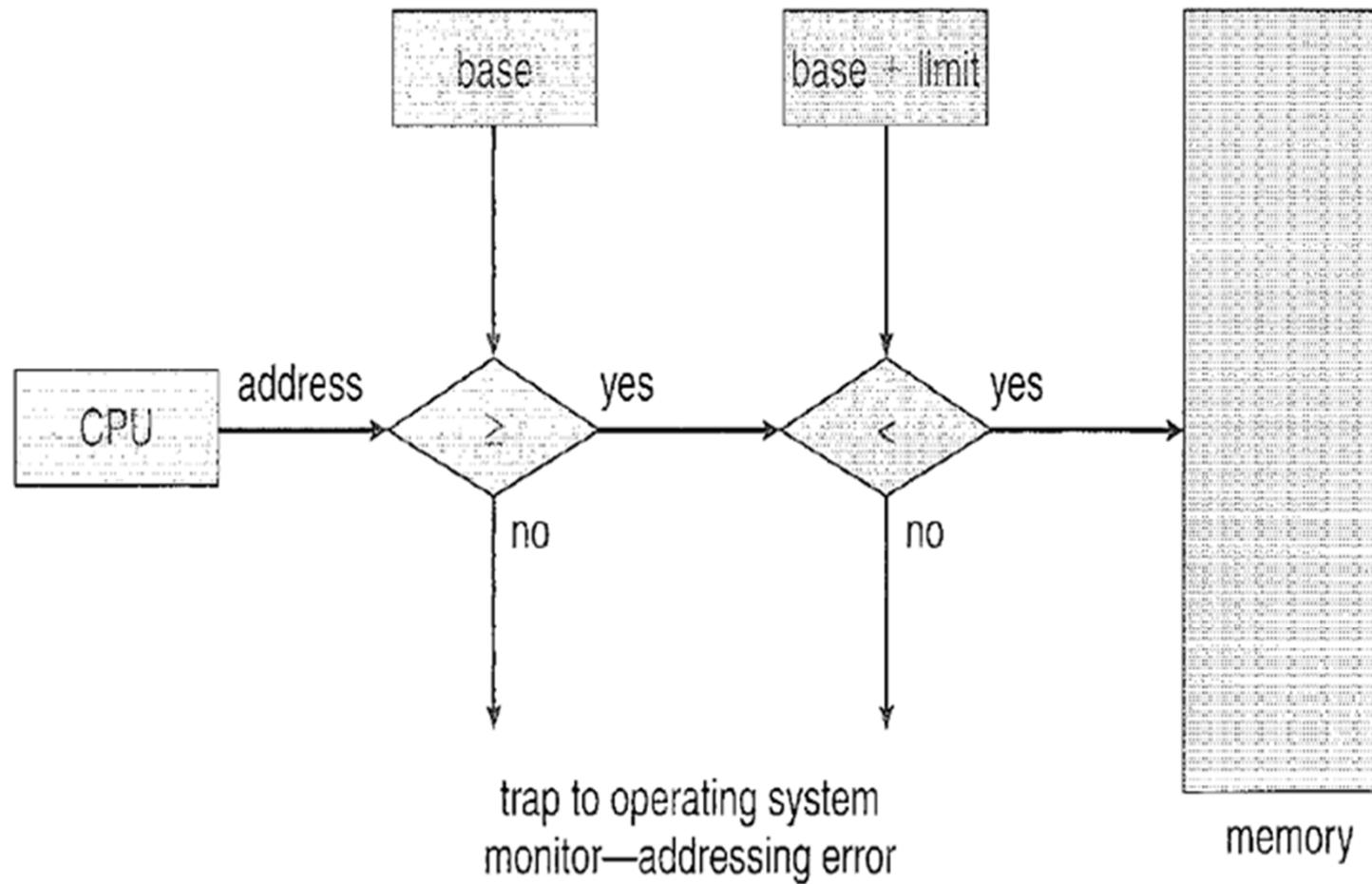


- Protection of memory space is accomplished by having the CPU hardware compare *every* address generated in user mode with the registers.
- Any attempt by a program executing in user mode to access OS memory or other users' memory results in a **trap** to the OS, which treats the attempt as a **fatal error**

**Fig 3.1** A base and a limit register define a logical address space.

Contd.

Can you explain what this figure depicts?



**Fig 3.2** Hardware address protection with base and limit registers

# Address Binding

What is the purpose of Binding?

- ☞ Usually a program resides in a **disk** in the form of **executable binary file**.
- ☞ It is brought to memory for execution (it may be moved between disk and memory in the meantime).
- ☞ When a process is executed it accesses **instructions and data from memory**.
  - When execution is completed the memory space will be freely available.
- ☞ A user program may be placed at any part of the memory.
- ☞ A user program passes through a number of steps before being executed.
- ☞ Addresses may be represented in different ways during these steps.
  - **Symbolic addresses**:- addresses in source program(E.g. **count**)  Compiler
  - **Re-locatable addresses**:- (E.g. 14 bytes from the beginning of this module)  Linkage editor or loader
  - **Absolute addresses**:- (E.g. 74014)

## Binding of instructions and data to memory

☞ Address binding of instructions and data to memory addresses can happen at **three different stages**.

1. **Compile time:-** If memory location is known a prior, absolute code can be generated;
  - Must **recompile** code if starting location changes.
  - The MS-DOS command format programs are bound at compile time.
2. **Load time:-** If it is not known at compile time where the process will reside in memory, then the compiler must generate a re-locatable code.
  - Final binding is delayed **until load time**.
  - If the starting address changes, only **reloading** the user code is needed to incorporate this changed value.
3. **Execution time:-** Binding delayed until runtime 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).

## Multi-step Processing of a User Program

- Memory Manager keeps track of which parts of memory are in use and which are not, to allocate memory to processes when they need it and deallocate it when they are done, and to manage swapping between main memory and disk when main memory is too small to hold all the processes
- This multi-step processing of the program invokes the appropriate utility and generates the required module at each step.

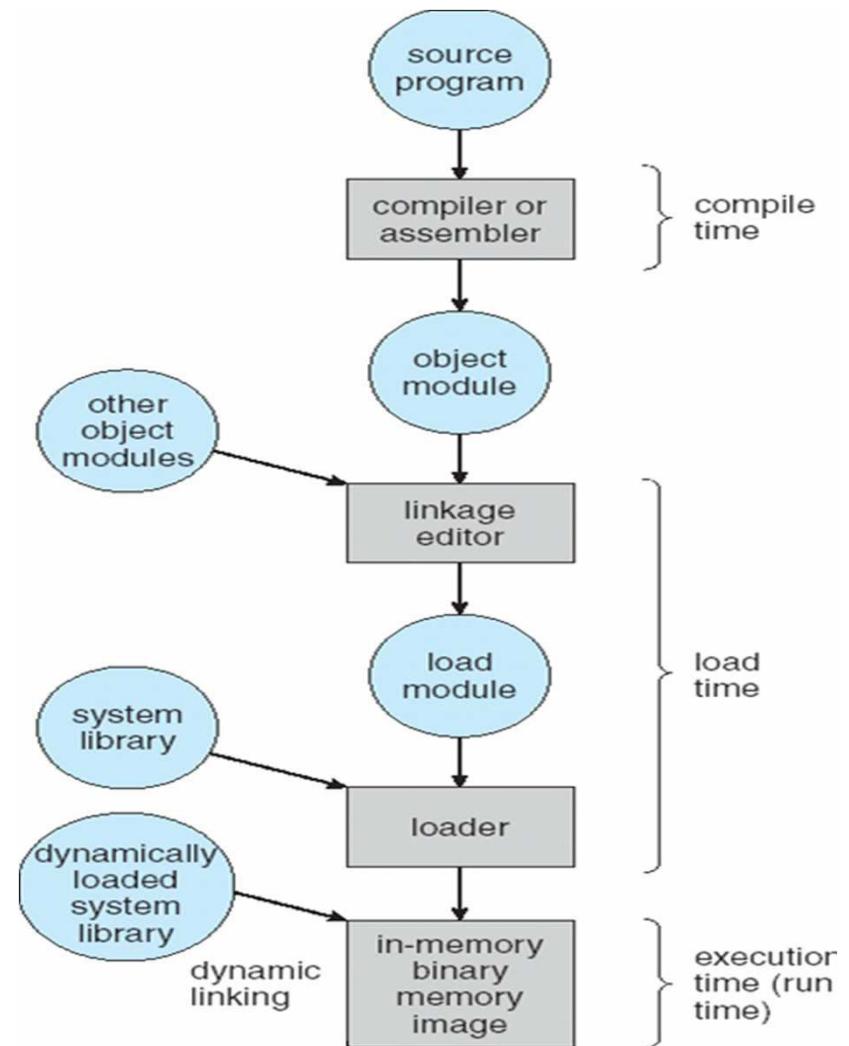


Fig 3.3 Multistep processing of a user program

# 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**.

**1. Logical Address:** or virtual address - generated by CPU

- Set of logical addresses generated by a program is called **logical address space**.

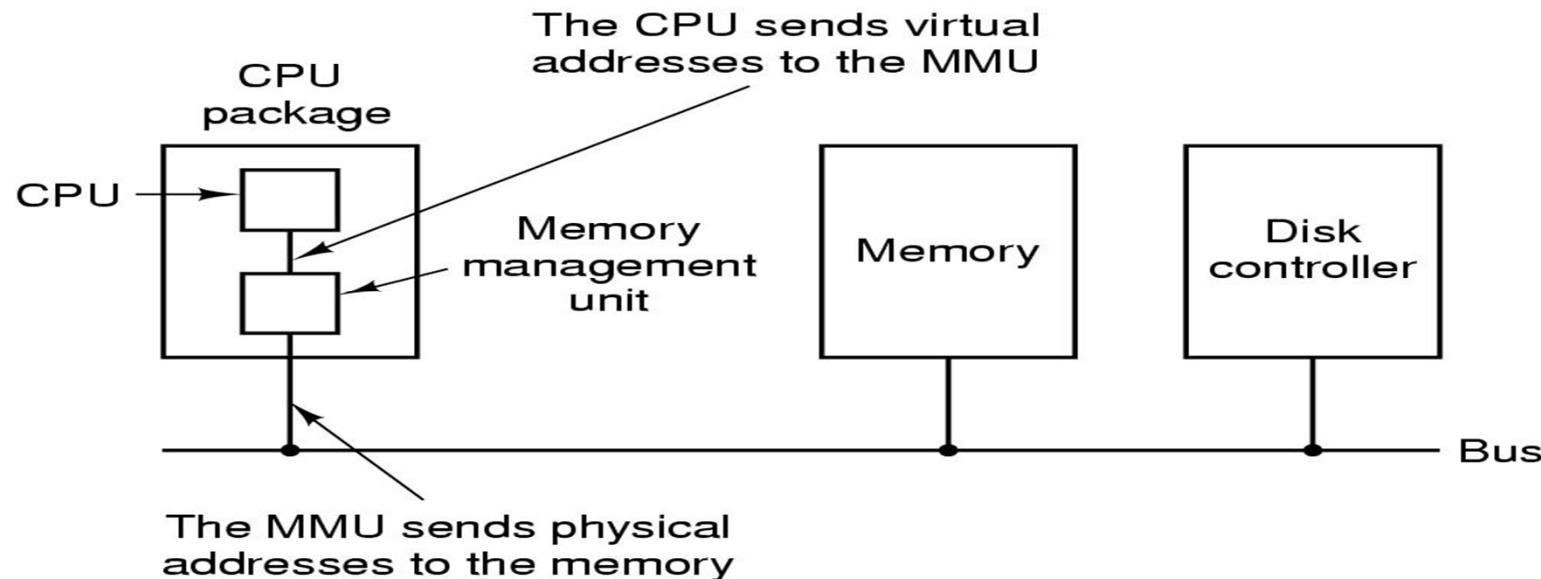
**2. Physical Address: address seen by memory unit.**

- Set of physical addresses corresponds to logical space is called **physical address space**

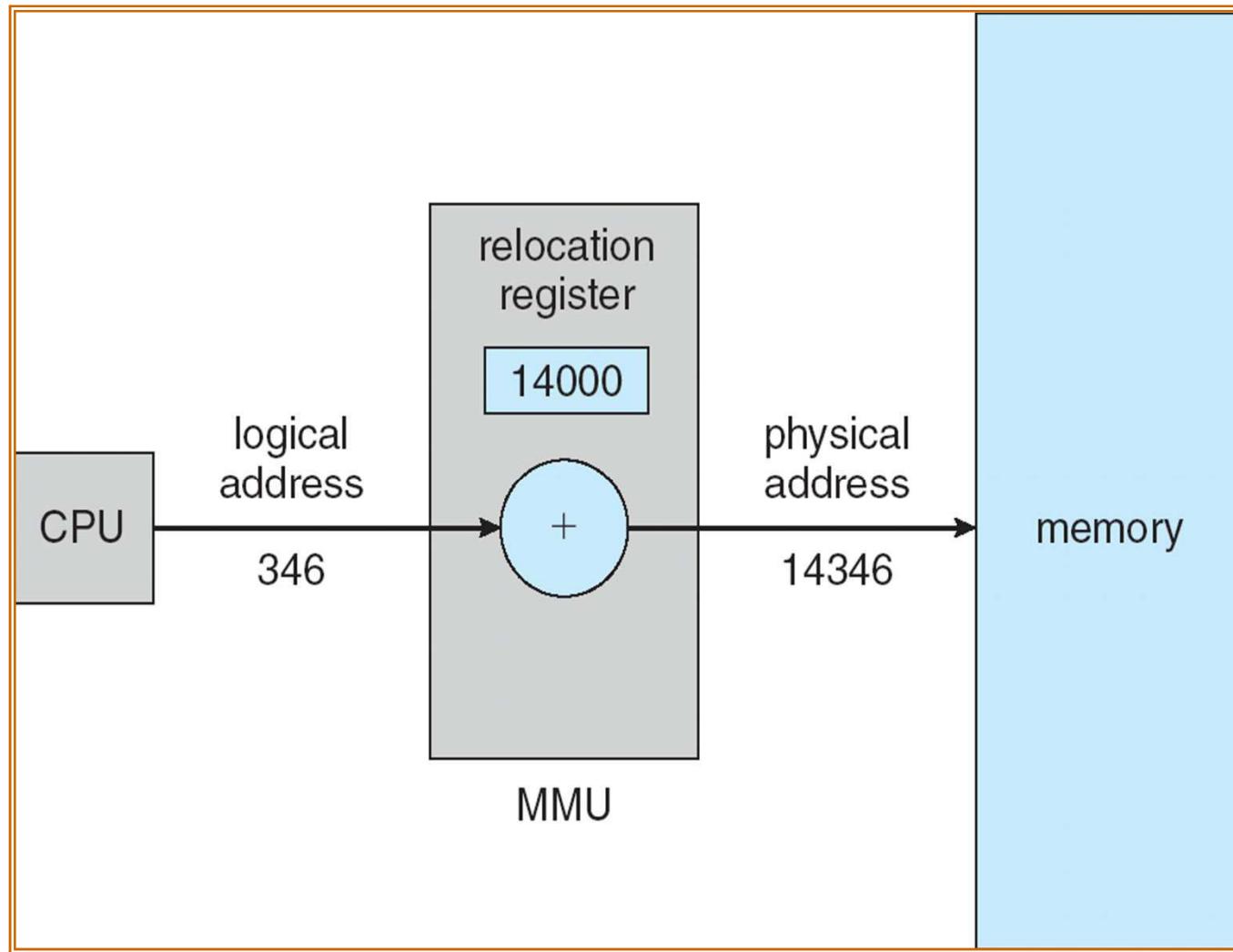
☞ **Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.**

# Memory Management Unit (MMU)

- Is a hardware device that maps **logical/virtual address** to **physical address** during the run-time.
- In MMU scheme, the value in the **base(relocation)** register is added to every virtual address generated by a user process at the time it is sent to memory.
- The user program deals with logical addresses; it never sees the real physical address.**



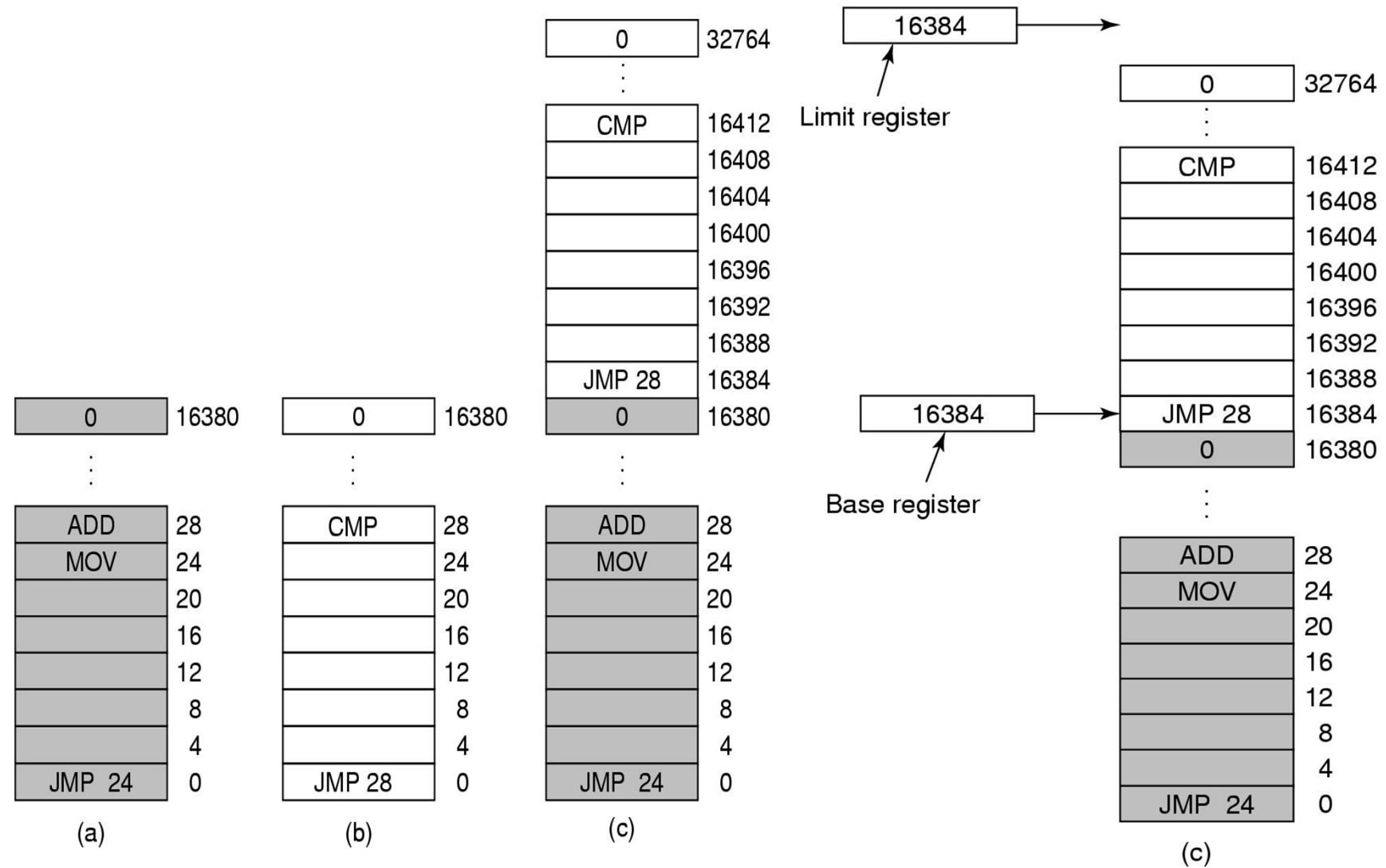
## Dynamic relocation using a relocation register



Can you explain what this figure depicts?

**Fig. 3.5** Dynamic relocation using a relocation register

## Relocation Example



## Dynamic Loading

☞ In the discussion made so far, we assume that a program and its associated resources must be loaded to memory before execution(not efficient in resource utilization).

- The size of a process is limited to the size of physical memory.

☞ **But in Dynamic loading routine is not loaded until it is called.**

- ❖ **Better memory-space utilization;** unused routine is never loaded.
- ❖ All routines are kept on **disk in a re-locatable load format.**
- ❖ The main program is loaded into memory and is executed.
  - When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been **loaded**.
  - If it has not, the **re-locatable linking loader** is called to load the desired routine into memory and to update the program's address tables to reflect this change.
  - Then control is passed to the newly loaded routine.

## Advantages of Dynamic Loading

1. An unused routine is never loaded. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines.
  - Useful when large amounts of code are needed to handle infrequently occurring cases, such as **error routines**.
  - In this case, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.
2. It **does not require special support from the OS**; implemented through program design.
  - It is the responsibility of the users to design their programs to take advantage of such a method.
  - OS may help the programmer, however, by **providing library routines** to implement dynamic loading.

## Dynamic Linking and Shared Libraries

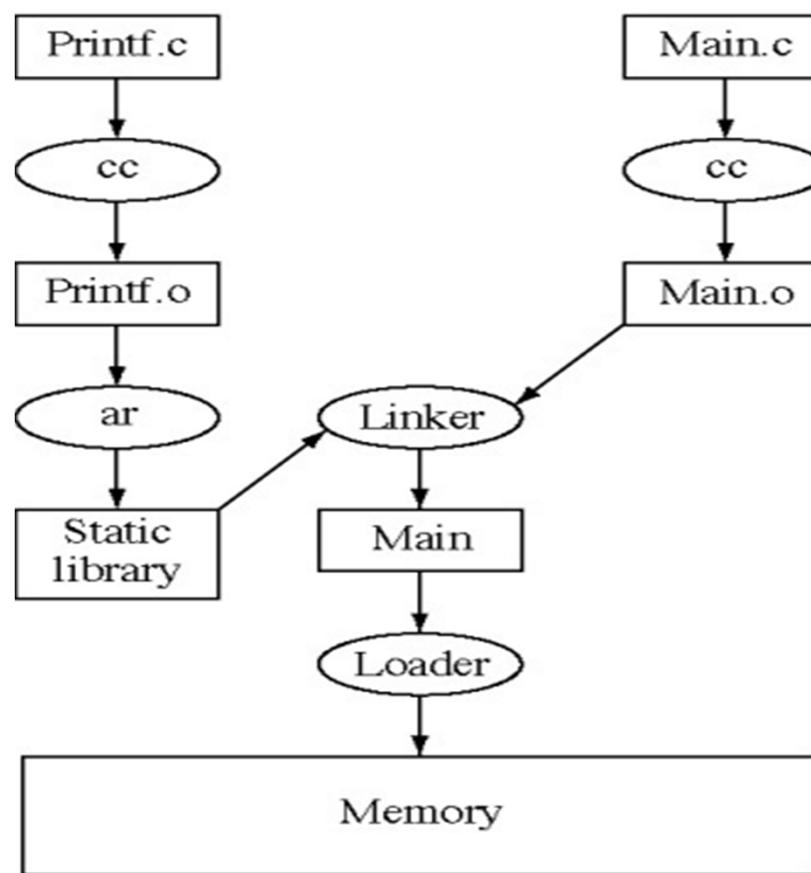
- ☞ Some Operating Systems support only **static linking**, in which system language libraries are treated like any other object module and are combined by the loader into the binary program image.
- ☞ Dynamic linking, in contrast, is similar to dynamic loading; here, though, linking, rather than loading, is postponed until execution time.
- ☞ With dynamic linking a small piece of code, **stub**, used to locate the appropriate memory-resident library routine.
  - Stub indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present.
  - When the Stub is executed it checks to see whether the needed routine is already in memory.
    - If it is not, the program loads the routine into memory replaces.
  - Either way, the stub replaces itself with the address of the routine and executes the routine.

**Contd.**

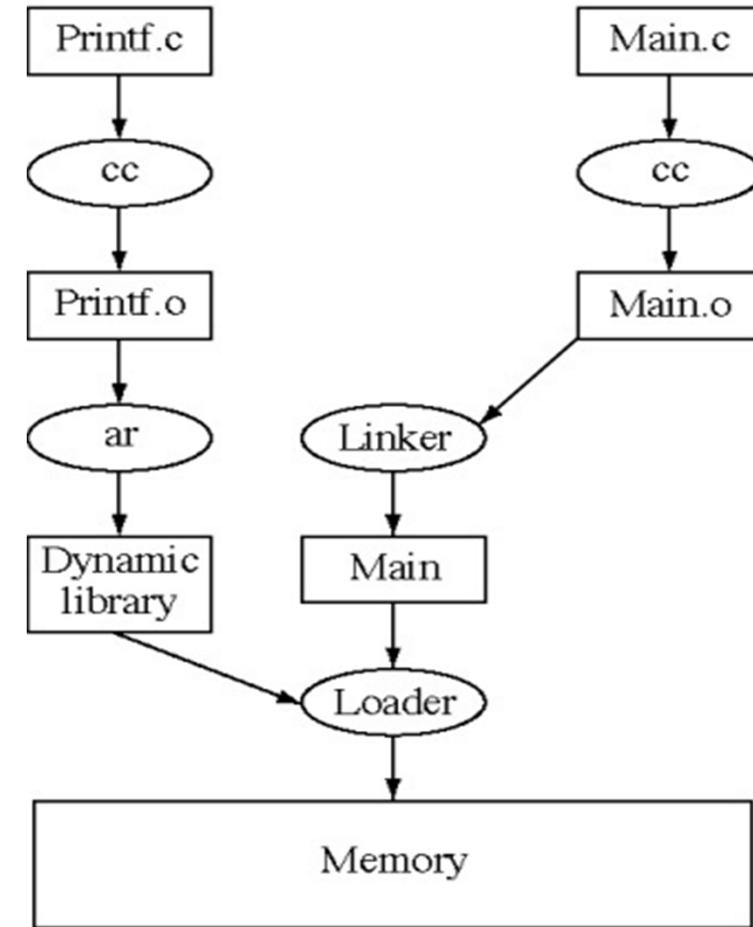
**Unlike dynamic loading, dynamic linking generally requires help from the OS.**

- Thus, the next time that particular code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking.
- Under this scheme, all processes that use a language library execute only one copy of the library code.
- ☞ A library may be replaced by a new version, and all programs that reference the library will automatically use the new version.
  - Without dynamic linking, all such programs would need to be relinked to gain access to the new library.
- ☞ More than one version of a library may be loaded into memory, and each program uses its version information to decide which copy of the library to use.
  - Versions with minor changes retain the same version number, whereas versions with major changes increment the number.
- ☞ Thus, only programs that are compiled with the new library version are affected by any incompatible changes incorporated in it.
- ☞ Other programs linked before the new library was installed will continue using the older library. This system is also known as **shared Libraries**.

## Static vs. Load-time Dynamic Linking



(a) Static dynamic  
linking



(b) Load-time dynamic linking

## Overlays

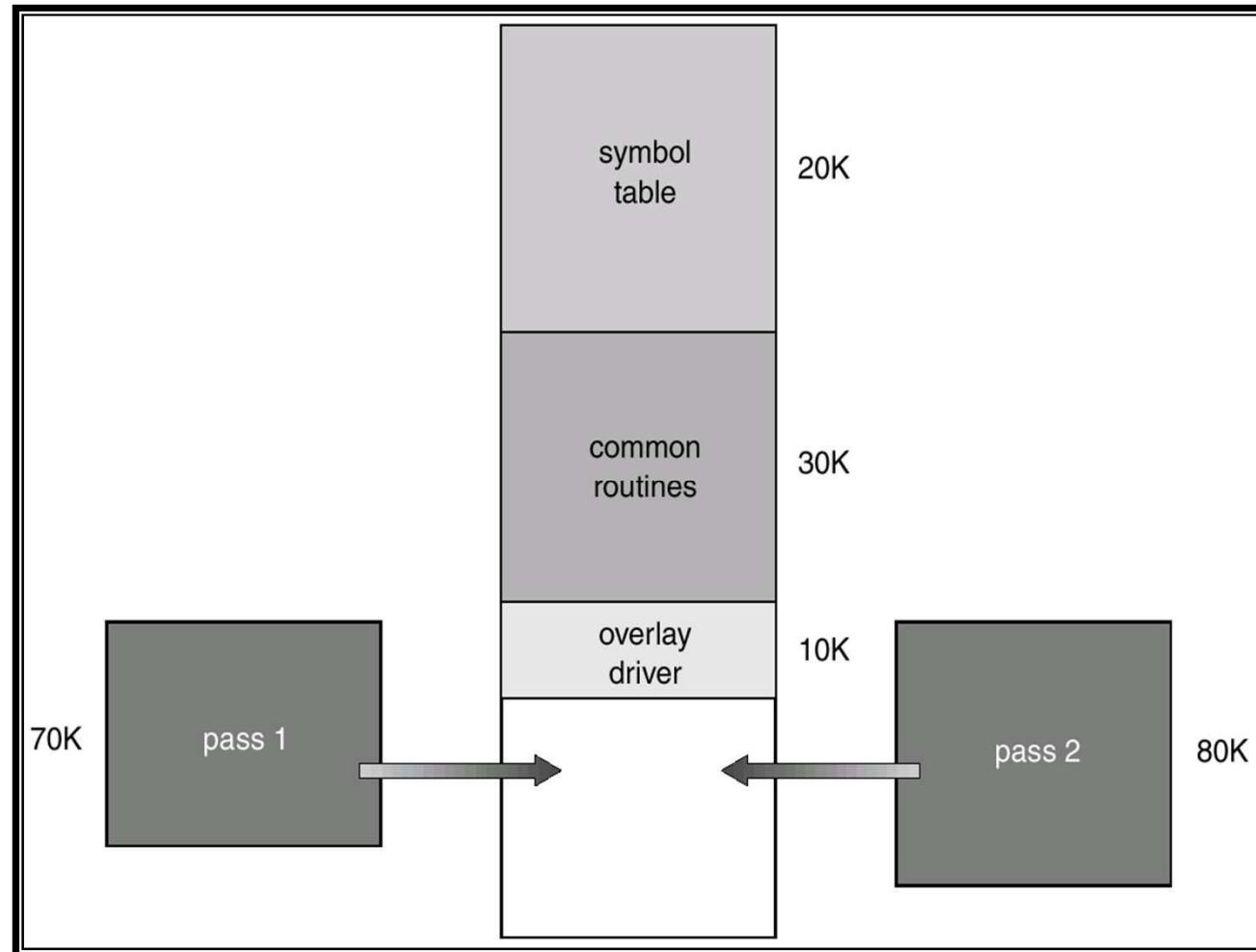
**Q:-** What to do when program size is larger than the amount of memory/partition (that exists or can be) allocated to it?

**Answer:-** there are two basic solutions in real memory management: **Overlays** and **Dynamic Link Libraries (DLLs)**

### Overlays

- Keep in memory only the **overlay** (those instructions and data that are) needed at any given phase/time.
- Overlays can be used only for programs that fit this model, i.e., multi-pass programs like compilers.
- Overlays are designed/implemented by programmer. **Needs an overlay driver.**
- No special support needed from operating system, but program design of overlays structure is **complex**.

## Overlays for a Two-Pass Assembler



**Fig. 3.8** Overlays for a two-pass assembler

## Memory Management Requirements

- ☞ If only a few processes can be kept in main memory, then much of the time all processes will be waiting for I/O and the CPU will be idle.
- ☞ Hence, memory needs to be allocated efficiently in order to pack as many processes into memory as possible.
- ☞ Need additional support for:
  - **Relocation**
  - **Protection**
  - **Sharing**
  - **Logical Organization**
  - **Physical Organization**

## Contd.

### 1. Relocation

- ☞ Programmer cannot know where the program will be placed in memory when it is executed.
- ☞ A process may be (often) relocated in main memory due to swapping/compaction:
  - ✚ Swapping enables the OS to have a larger pool of ready-to-execute processes.
  - ✚ Compaction enables the OS to have a larger contiguous memory to place programs in.

### 2. Protection

- ☞ Processes should not be able to reference memory locations in another process without permission.
- ☞ Impossible to check addresses in programs at compile/load-time since the program could be relocated.
- ☞ Address references must be checked at execution-time by hardware.

## Contd.

### 3. Sharing

- ☞ must allow several processes to access a common portion of main memory without compromising protection:
- ☞ Better to allow each process to access the same copy of the program rather than have their own separate copy.
- ☞ Cooperating processes may need to share access to the same data structure.

### 4. Logical Organization

- ☞ Users write programs in modules with different characteristics:
  - instruction modules are execute-only.
  - data modules are either read-only or read/write.
  - some modules are private and others are public.
- ☞ To effectively deal with user programs, the OS and hardware should support a basic form of a module to provide the required protection and sharing.

## Contd.

### 5. Physical Organization

- ☞ External memory is the long term store for programs and data while main memory holds programs and data currently in use.
- ☞ **Moving information between these two levels of the memory hierarchy is a major concern of memory management:**
  - It is highly inefficient to leave this responsibility to the application programmer.

## Swapping

- ☞ A process can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution.
  - **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(might be?).
- ☞ In swapping
  - context time must be taken into consideration.
  - Swapping a pending process for an I/O needs care.

## Contd.

☞ Modified versions of swapping are found on many systems, i.e. **UNIX** and **Microsoft Windows**.

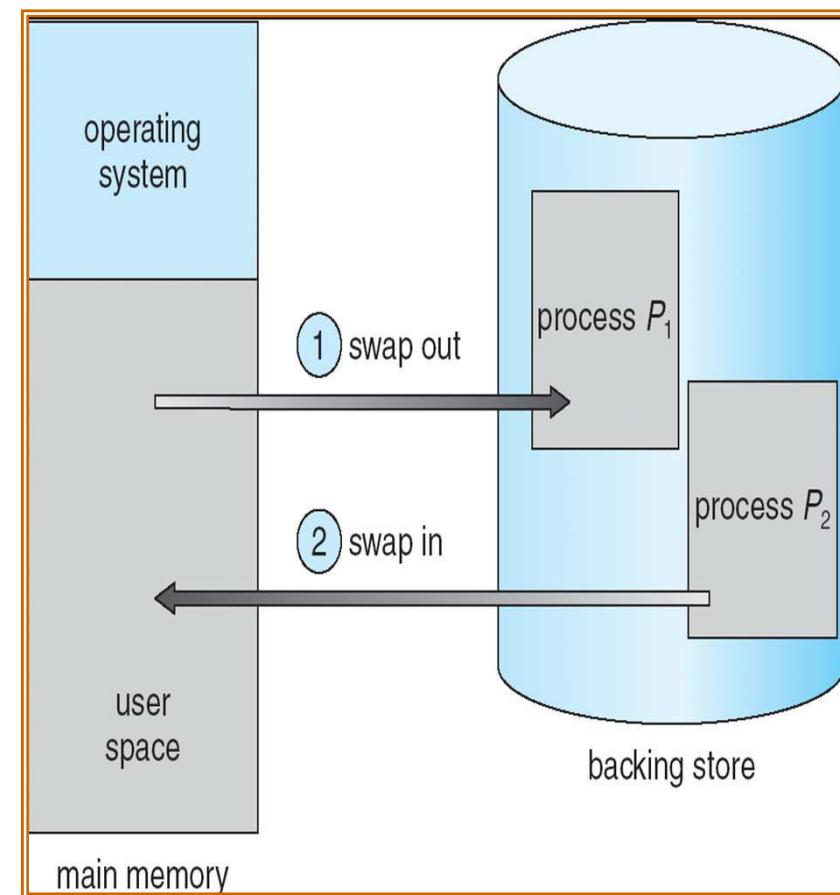
- System maintains a ready queue of **ready-to-run processes** which have memory images on disk.
- **Give possible solution**

### Swapping is done when

- Quantum expires
- Priority issues arises

### Conditions for swapping

- Process to be executed must be
- not in memory
- No sufficient free memory location



# Contiguous Allocation

- ☛ Main memory usually divided into two partitions
  1. Resident Operating System, usually held in low memory with interrupt vector.
  2. User processes then held in high memory.

## Single partition allocation

- Relocation register scheme used to protect user processes from each other, and from changing OS code and data.
- Relocation register contains value of **smallest physical address**;
  - limit register contains range of logical addresses -each logical address must be less than the limit register.
- When CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values. Every address generated by CPU is compared against these values.
- Ensures memory protection of OS and other processes from being modified by the running process.

## Hardware Support for Relocation and Limit Registers

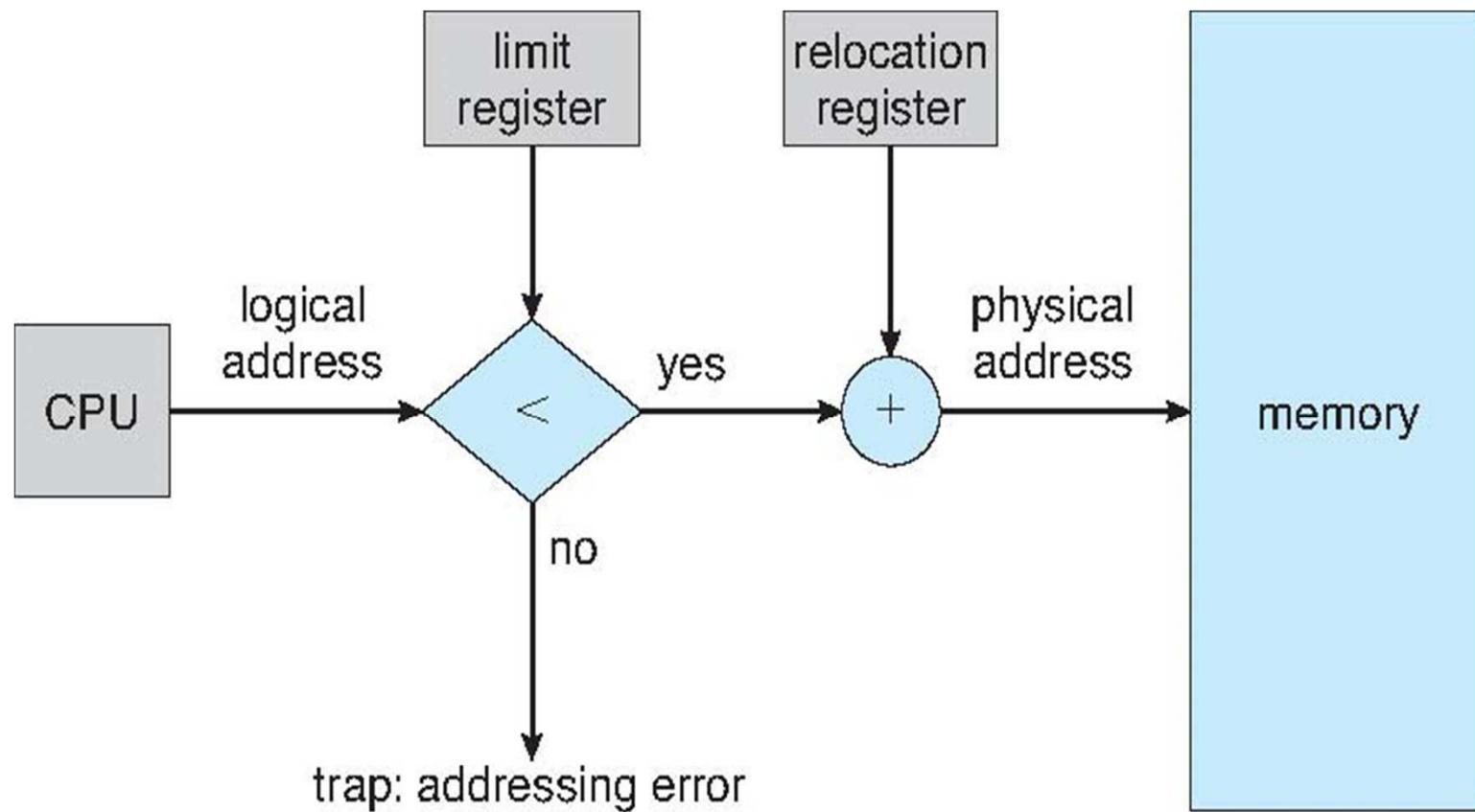


Fig. 3.9 Hardware Support for Relocation and Limit Registers

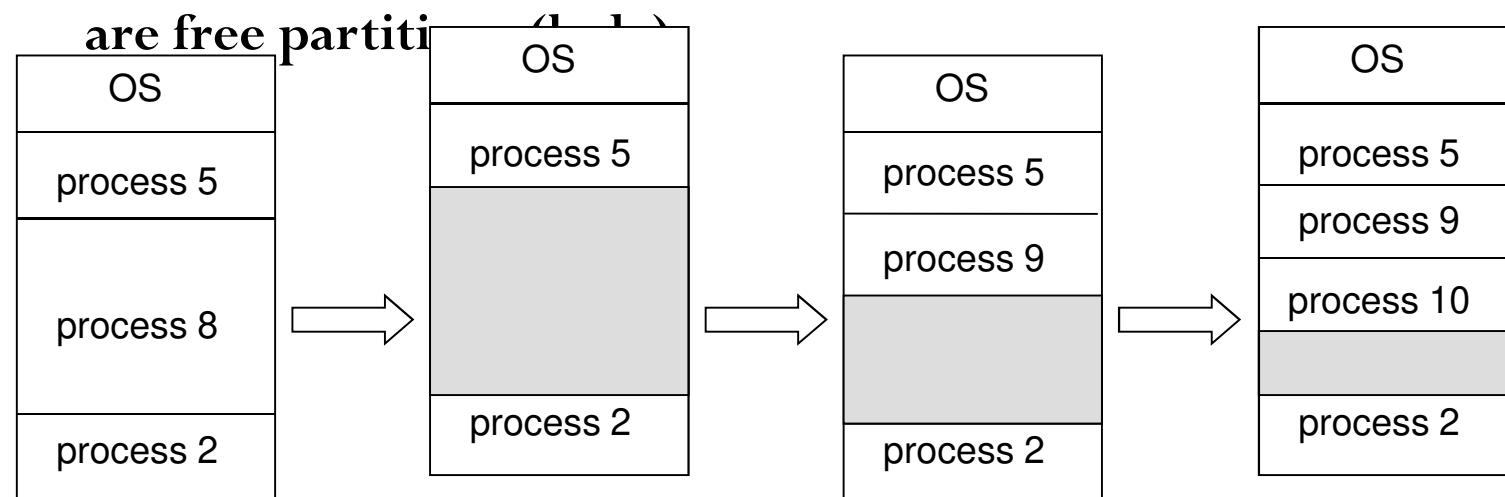
**Contd.**

## Multiple Partition Allocation

- **Hole:-** block of available memory; holes of various sizes are scattered throughout memory.

- When a process arrives, it is allocated memory from a hole large enough to accommodate it.

- Operating system maintains information about which partitions are **allocated partitions** and **which are free partitions**



**Fig. 3.10** Contiguous Allocation example

## Fixed Partitioning

- Partition main memory into a set of non-overlapping memory regions called partitions.
- Fixed partitions can be of equal or unequal sizes.
- Leftover space in partition, after program assignment, is called internal fragmentation.

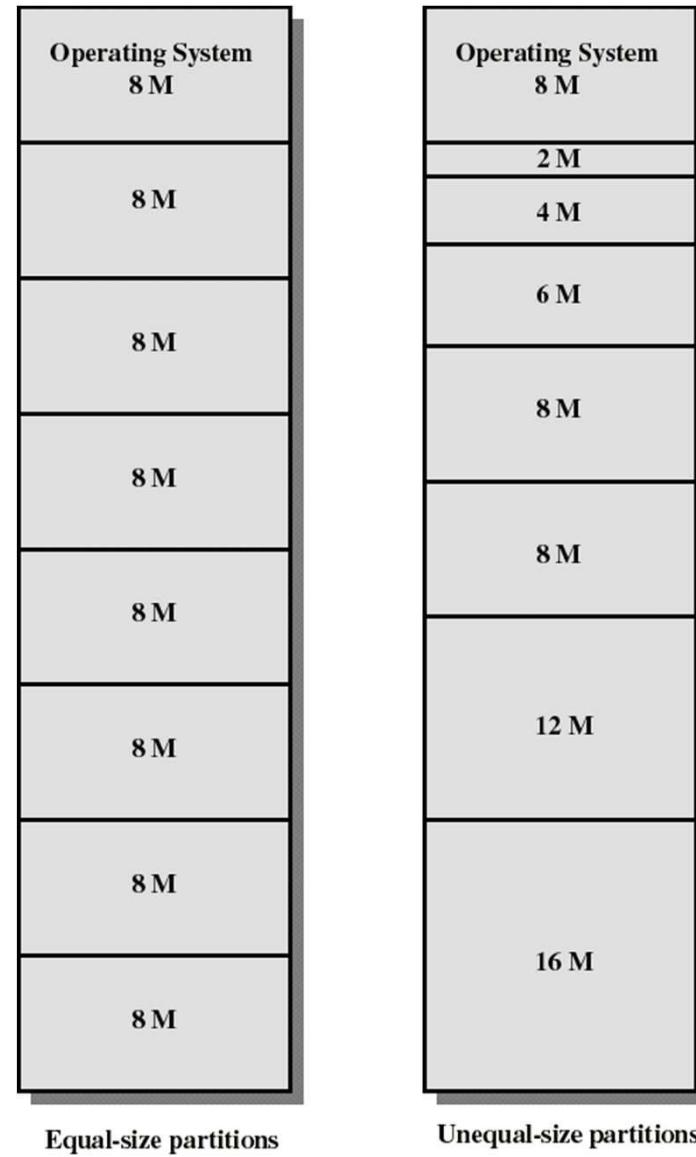


Fig. 3.11 Equal-size Vs. unequal-size Partitions

# Placement Algorithm with Partitions

## A. Equal-size partitions:

- If there is an available partition, a process can be loaded into that partition:
- Because all partitions are of equal size, it does not matter which partition is used.
- If all partitions are occupied by blocked processes, choose one process to swap out to make room for the new process.

## B. Unequal-size partitions using multiple queues

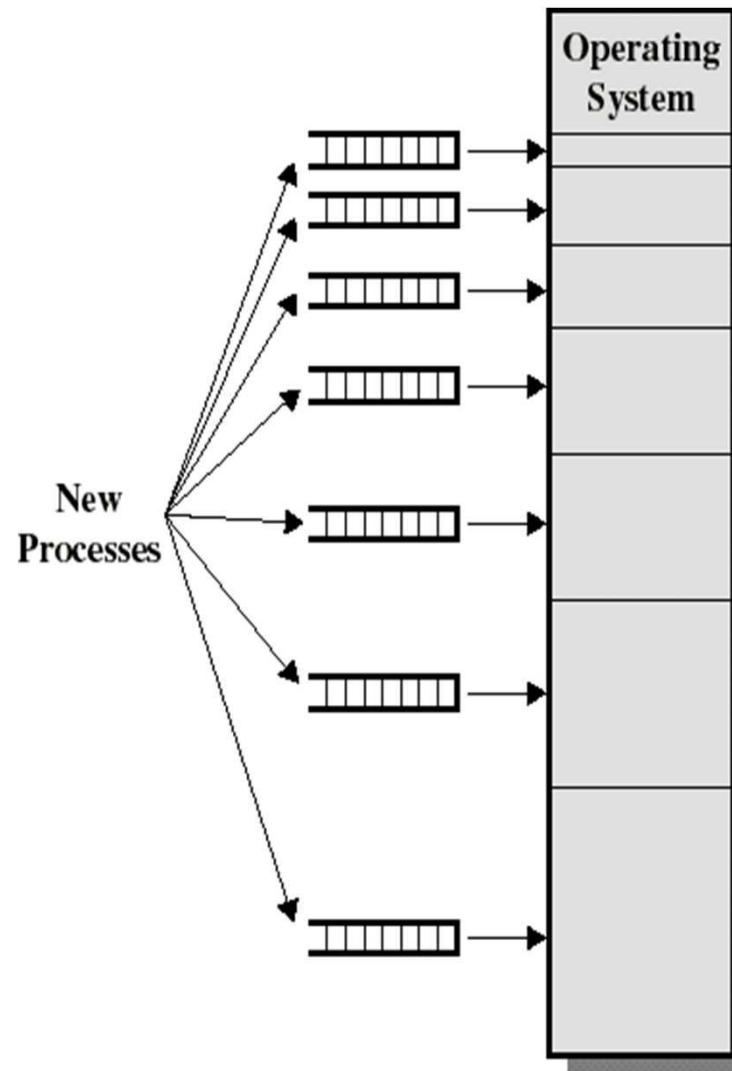
- assign each process to the smallest partition within which it will fit.
- a queue exists for each partition size.
- Tries to minimize internal fragmentation.

## Contd.

**Problem:-** some queues might be empty while some might be loaded.

**Solutions:-** Have at least one small partition around.

- Such a partition will allow small jobs to run without having to allocate a large partition for them.
- Have a rule stating that a job that is eligible to run may not be skipped over more than k times.



**Fig. 3.12** Unequal-size Partitions using Multiple queues

Contd.

### C. Unequal-size partitions using single queue:

- when it's time to load a process into memory, the smallest available partition that will hold the process is selected.
- increases the level of multiprogramming at the expense of **internal fragmentation**.

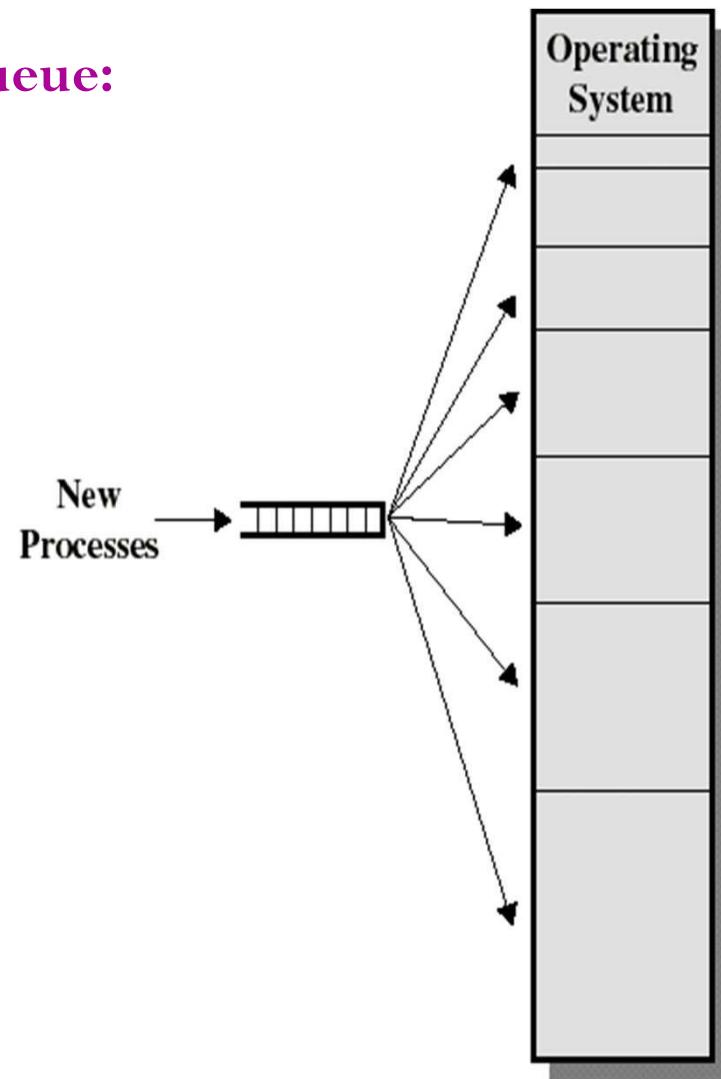


Fig. 3.12 Unequal-size Partitions using Single queue

## Dynamics of Fixed Partitioning

- ☞ Any process whose size is less than or equal to a partition size can be loaded into the partition.
  - If all partitions are occupied, the OS can **swap** a process out of a partition.
- ☞ If a program is too **large** to fit in a partition, then the programmer must design the program with **overlays**.
- ☞ Main memory use is **inefficient**. Any program, no matter how small, occupies an entire partition. This can cause **internal fragmentation**.
  - Unequal-size partitions lessens these problems but they still remain ...
  - Equal-size partitions was used in early IBM's OS/MFT (Multiprogramming with a Fixed number of Tasks).

## Dynamic (Variable) Partitioning

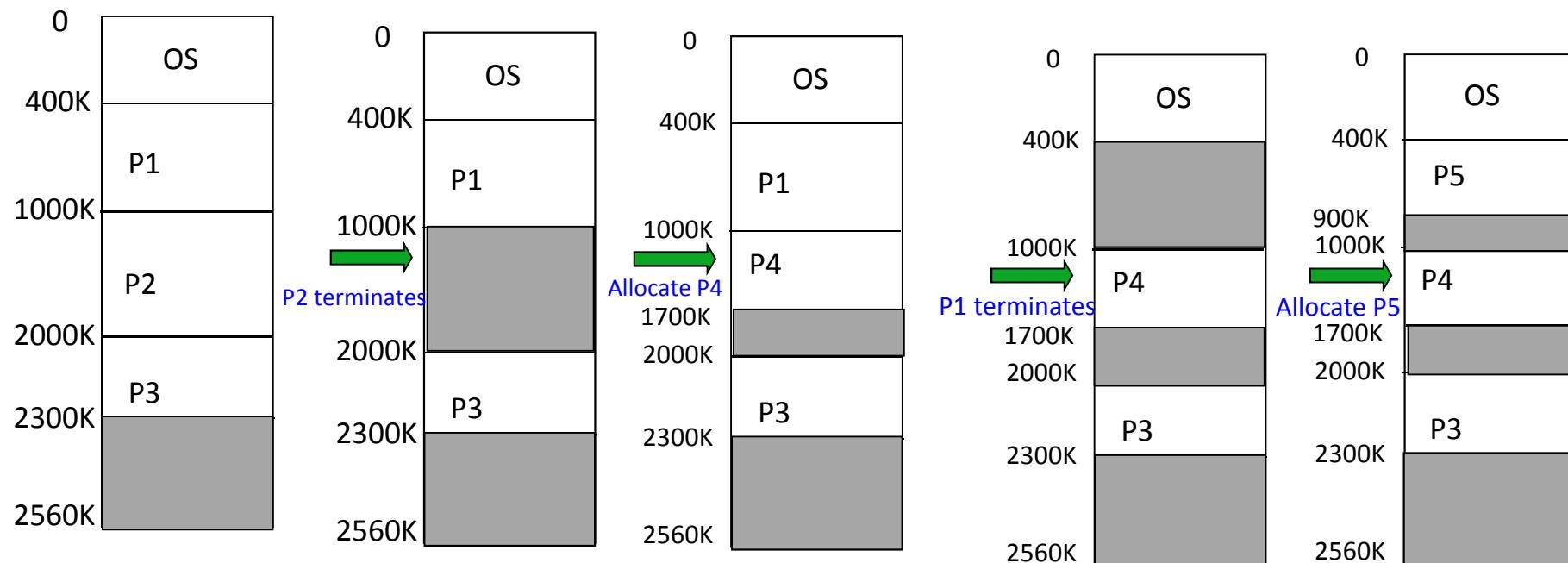
- ☞ In the **Variable partitioning** scheme, the operating system keeps a **table** indicating which parts of memory are available and which are occupied.
- ☞ Initially, all memory is available for user processes and is considered one large block of available memory a **hole**.
  - Memory contains a set of holes of various sizes scattered throughout memory.
- ☞ When a process arrives, it is allocated memory from a hole large enough to accommodate it.
- ☞ Eventually get holes in the memory.
  - This is called **External Fragmentation**
- ☞ Must use compaction to shift processes so they are contiguous and all free memory is in one block

## Contd.

**Example 1:-** 2560K of memory available and a resident OS of 400K. Allocate memory to processes P1...P4 following FCFS.

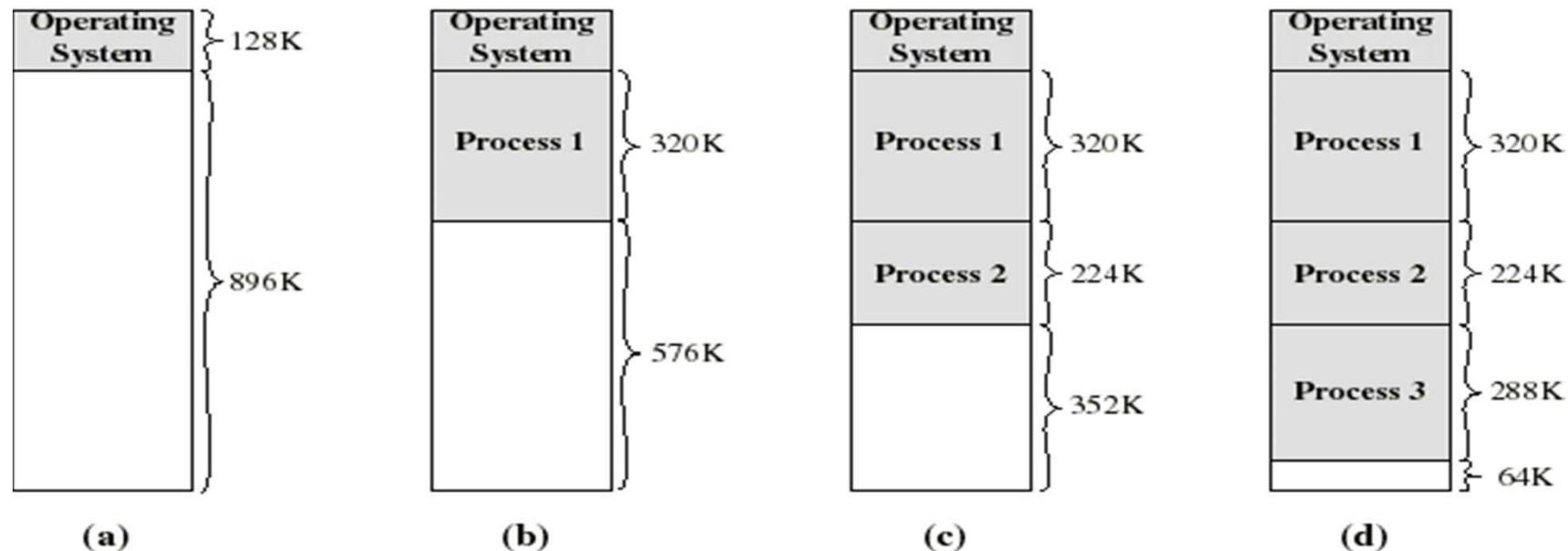
- Shaded regions are holes
- Initially P1, P2, P3 create first memory map.

Process	Memory	time
P1	600K	10
P2	1000K	5
P3	300K	20
P4	700K	8
P5	500K	15



**Fig. 3.11** Dynamic Partitioning example

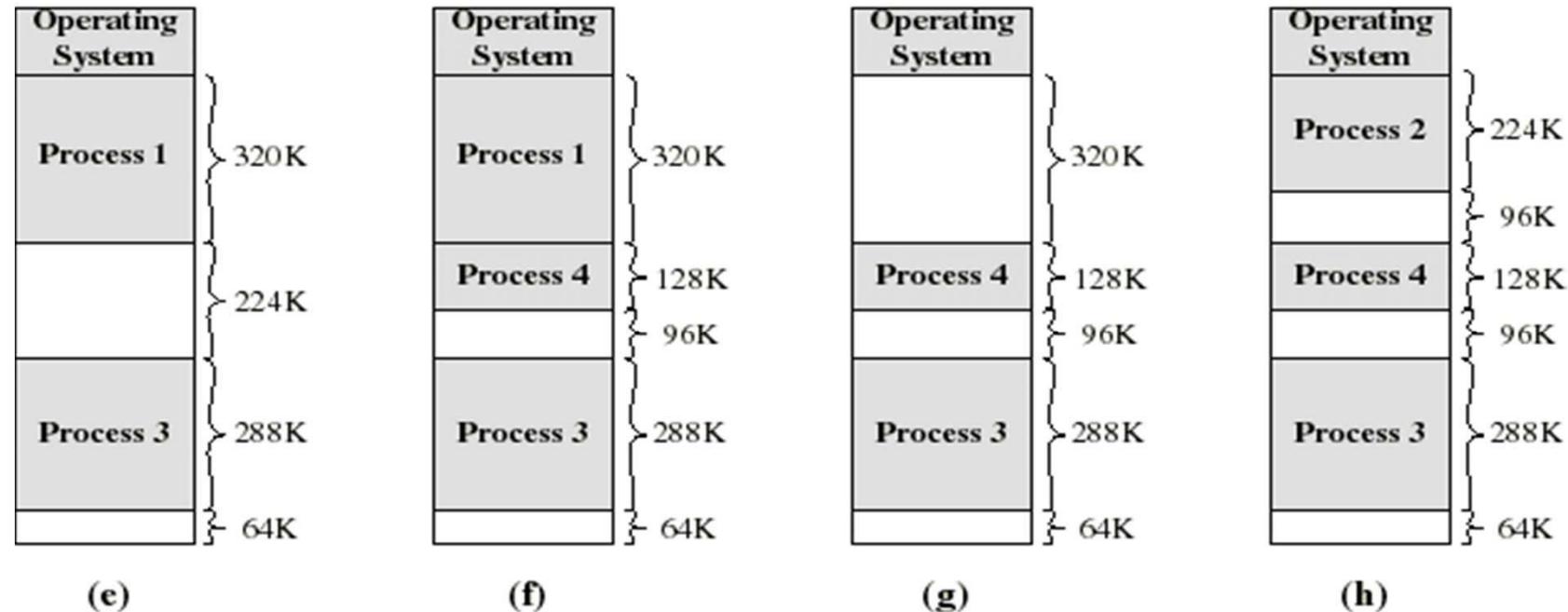
## Contd.



**Fig. 3.12 a.** Dynamic Partitioning example 2

- A hole of 64K is left after loading 3 processes: not enough room for another process.
- Eventually each process is blocked. The OS swaps out process 2 to bring in process 4 (128K).

## Contd.



**Fig. 3.12 b.** Dynamic Partitioning example 2

- Another hole of 96K is created.
- Eventually each process is blocked.
- The OS swaps out process 1 to bring in again process 2 and another hole of 96K is created ...

## Internal/External Fragmentation

- ☞ There are two types of fragmentation:
  1. **Internal Fragmentation:-** allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
    - This is when memory is handed out in some fixed way and requesting program doesn't use it all.
  2. **External Fragmentation:-** total memory space exists to satisfy a size  $n$  request, but that memory is not contiguous.
    - This occurs when there are many small pieces of free memory.
    - **50 % rule:-** Given  $N$  allocated blocks 0.5 blocks will be lost due to fragmentation.

## Reducing External Fragmentation

- ☞ Reduce external fragmentation by doing compaction:
  1. Shuffle memory contents to place all free memory together in one large block (or possibly a few large ones).
  2. Compaction is possible only if **relocation is dynamic**, and is done at **execution time**.
- ☞ Total memory space may exist to satisfy a request but it is not contiguous.
  - Compaction reduces external fragmentation by shuffling memory contents to place all free memory together in one large block.

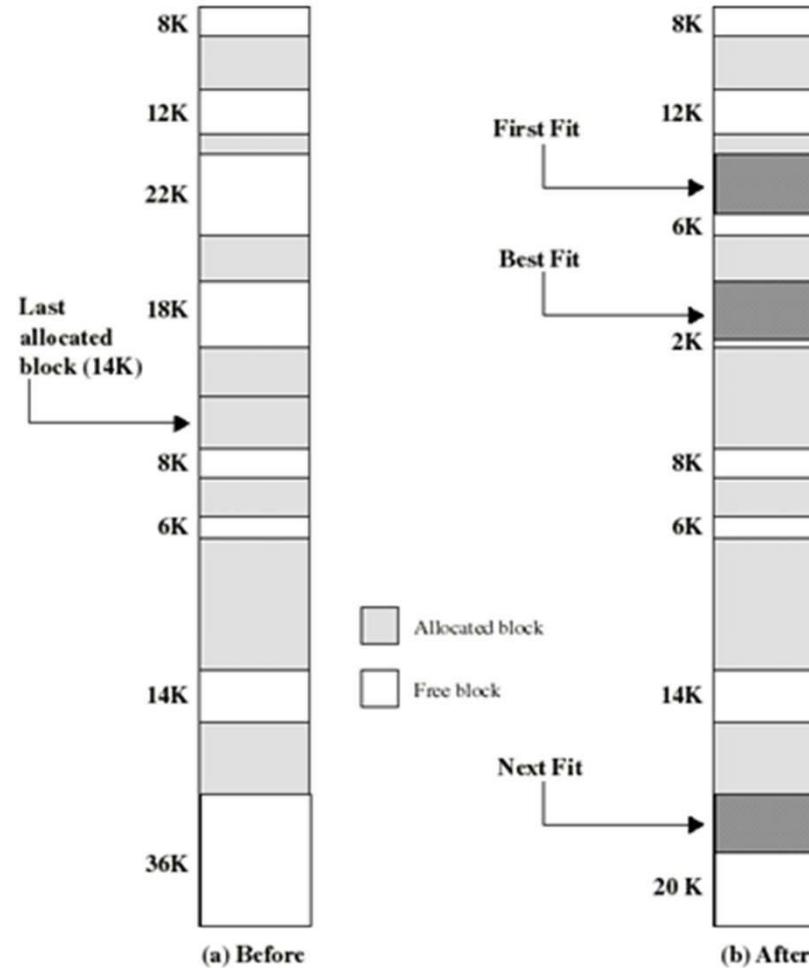
## Dynamic Storage Allocation Problem

- Operating system must decide which free block to allocate to a process.
- How to satisfy a request of size n from a list of free holes? **use algorithms**
  1. **First-fit:-** is the fastest allocation technique that allocates the first hole that is big enough
  2. **Next-fit:-** Same logic as first-fit but starts search always from the last allocated hole (need to keep a pointer to this) in a wraparound fashion
  3. **Best-fit:-** Allocate the smallest hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
    - the smallest amount of fragmentation is left
    - Worst performer overall
  4. **Worst-fit:-** Allocate the largest hole; must also search entire list.
    - Produces the largest leftover hole.

**❑ First-fit and best-fit are better than worst-fit in terms of speed and storage utilization.**

## Placement Algorithm

- Used to decide which free block to allocate to a process.
- Goal: to reduce usage of compaction procedure (Its time consuming).
- Example algorithms:
  - First-fit
  - Next-fit



**Fig. 3.13** Memory Configuration Before and After allocation of 16Kb Block

# Solution To Fragmentation

☞ There are three possible solutions for fragmentation:

- 
- 
- 

**Compaction**

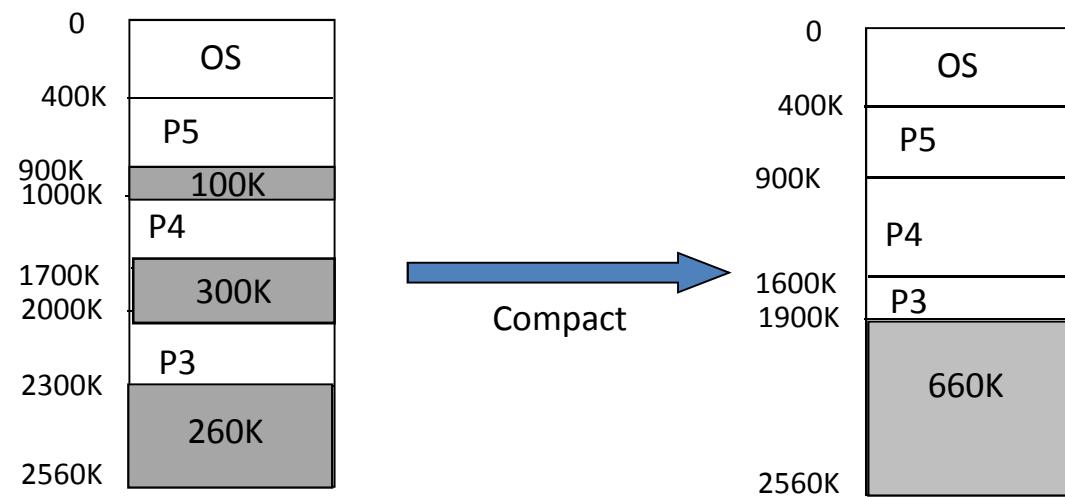
**Paging**

**Segmentation**

I. **Compaction:-** Reduces 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.
- Compaction depends on cost.

Fig. 3.14 Compaction Example



## II. Paging

**Basic Idea:** logical address space of a process can be made **noncontiguous**; process is allocated physical memory wherever it is available.

- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames

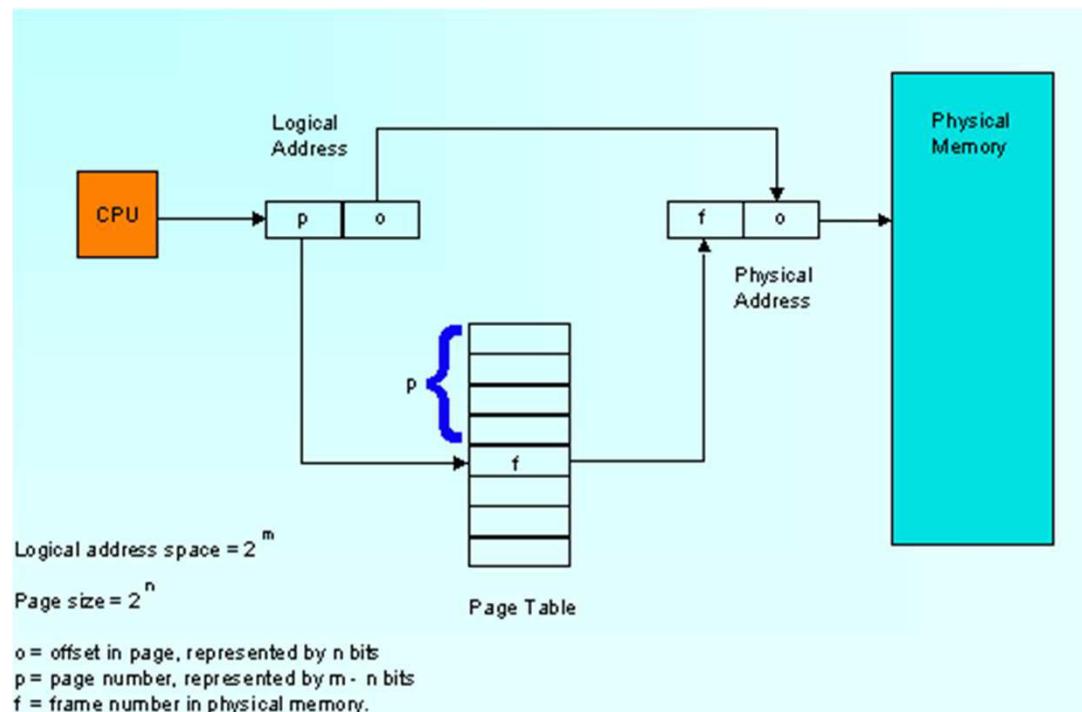
☞ The process pages can thus be assigned to any available frames in main memory; a process does not need to occupy a contiguous portion of physical memory.

☞ 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
- **Note:: Internal Fragmentation possible!!**

# Basic Paging Architecture

- ☞ A **page table** translates logical address to physical Address.
- ☞ No external fragmentation but small Internal fragmentation is possible (for page at end of program).



**Fig. 3.15** Basic Paging Architecture

## Address Translation Scheme

- ☞ Address generated by CPU is divided into:
  1. **Page number (p)**:- used as an index into a *page table* which contains base address of each page in physical memory.
  2. **Page offset (d)**:- combined with base address to define the physical memory address that is sent to the memory unit.
    - Page number is an index to the page table.
    - The page table contains base address of each page in physical memory.
    - The base address is combined with the page offset to define the physical address that is sent to the memory unit.
- ☞ The size of logical address space is  $2^m$  and page size is  $2^n$  address units.
  - Higher  $m-n$  bits designate the page number
  - $n$  lower order bits indicate the page offset.

page number	page offset
$p$	$d$
$m-n$	$n$

## Example of Paging

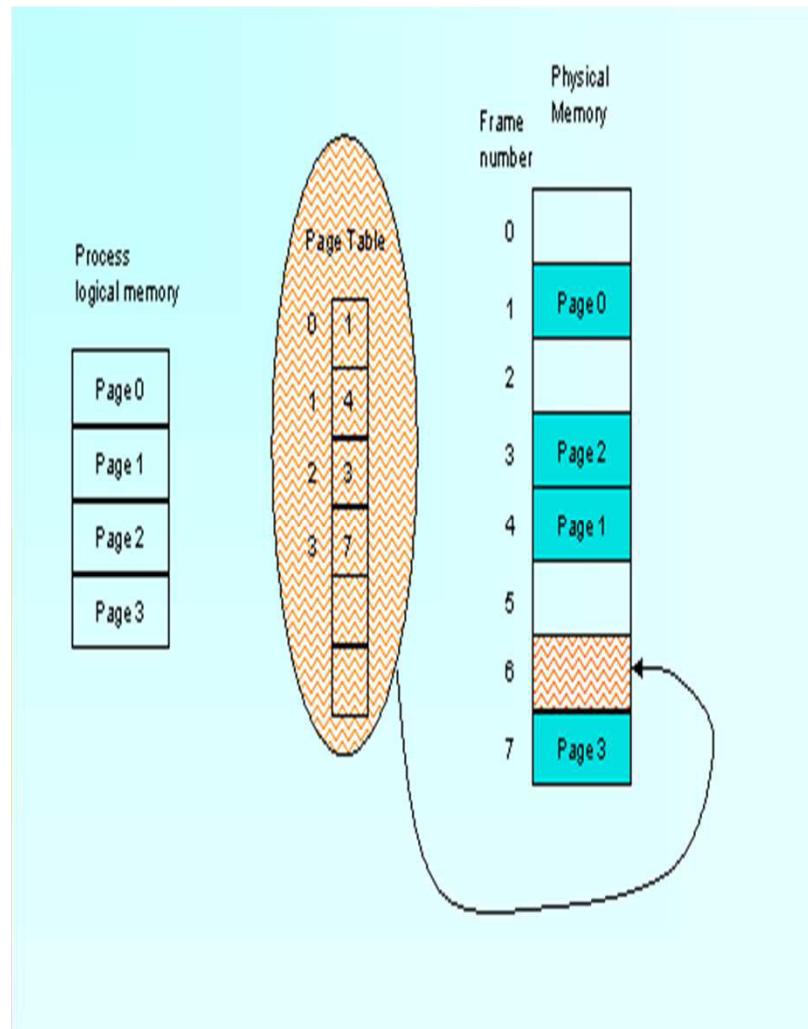


Fig. 3.16 Paging Example

47

- Each process has its own page table.
- Page table is kept in memory.
- **Page table base register** points to the page table.
- **Page table length register** indicates the size of the page table.
- How many physical memory access are required for each logical memory access?
- Answer: **two** (one for the page table and one for the data/instruction).
- **Can we do better? Yes how? Using TLB**

# How to implement Page Table?

## 1. Keep Page Table in main memory:

- **Page-table base register (PTBR)** points to the page table.
- **Page-table length register (PTLR)** indicates size of the page table.
- However, in this scheme, every data/instruction access requires two memory accesses, one for the page table and one for the data/instruction.

## 2. Keep Page Table in hardware (in MMU):-

however, page table can be large, too expensive.

## 3. Use a special fast-lookup hardware cache called **Associative Memory (Registers)** or **Translation Look-aside Buffer (TLB)**

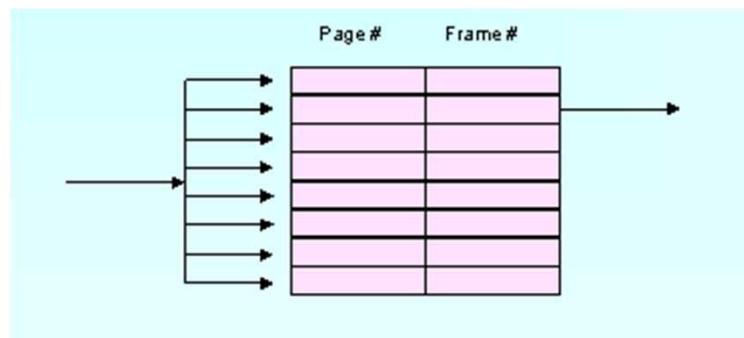
- If p is in associative register, get frame # out.
- Otherwise get frame # from page table in memory.

## Translation Look-aside Buffers(TLB)

- Also called **an associative memory – parallel search**
- A set of associative registers is built of especially high-speed memory.
- Each register consists **of two parts: key and value**
- When associative registers are presented with an item, it is compared with all keys simultaneously.
- If corresponding field is found, corresponding field is output.
- Associative registers contain only few of page table entries.
  - When a logical address is generated it is presented to a set of associative registers.
  - If yes, the page is returned.
  - Otherwise memory reference to page table mode. Then that page # is added to associative registers.

## Contd.

- Address translation ( $A'$ ,  $A''$ )
  - If  $A'$  is in associative register, get frame # out.
  - Otherwise get frame # from page table in memory
- It may take 10 % longer than normal time.
- % of time the page # is found in the associative registers is called hit ratio.



## Paging Hardware With TLB

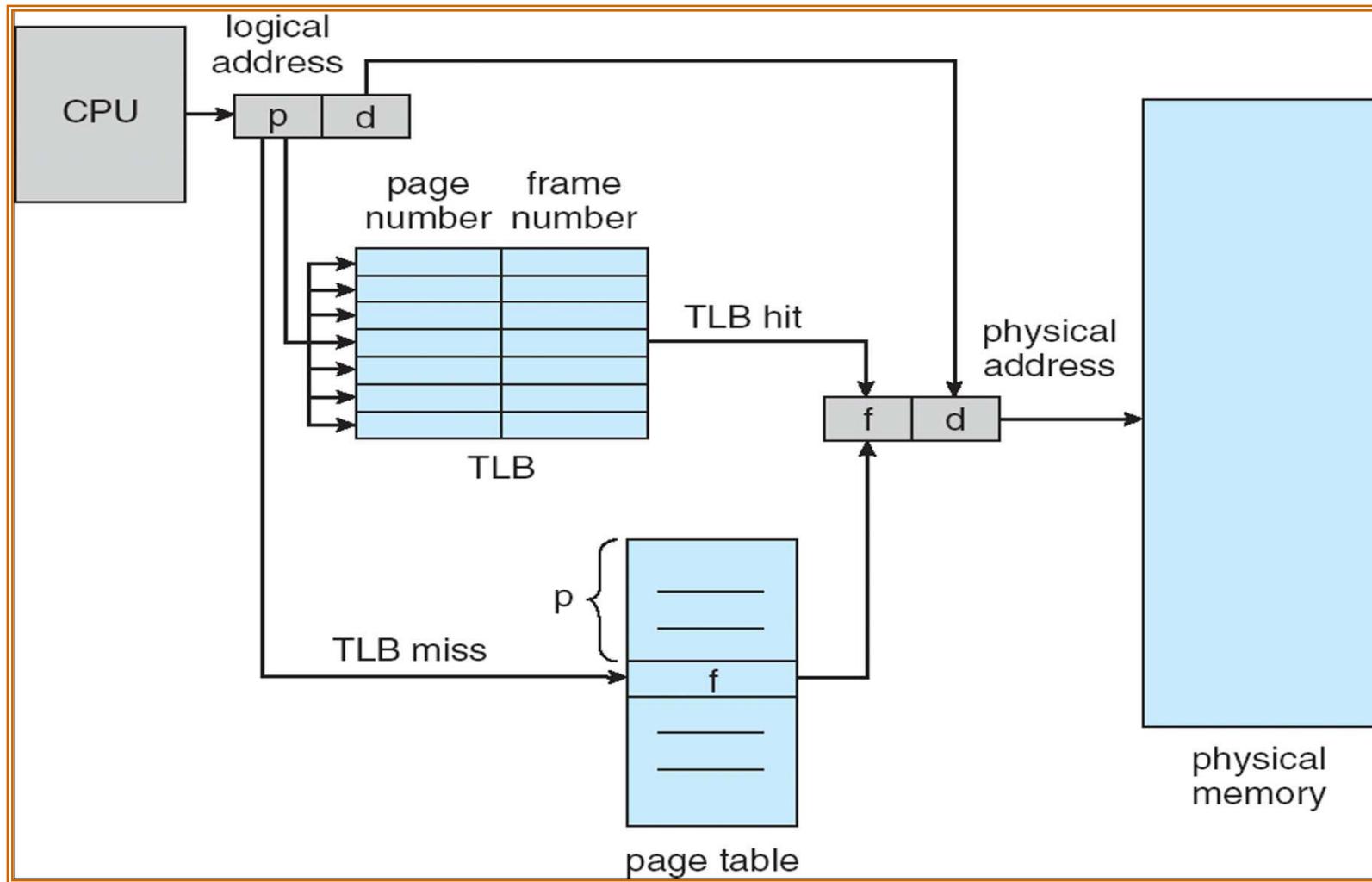


Fig. 3.17 Paging Hardware with TLB

## Effective Access Time

- ☞ **Associative Lookup =  $\epsilon$**  time unit, Assume memory cycle time is 1 microsecond
- ☞ **Hit ratio:-**percentage of times that a page number is found in the associative registers; ratio related to number of associative registers. **Hit ratio =  $\alpha$**

$$\text{Effective Access Time (EAT)} = (1 + \epsilon) \alpha + (2 + \epsilon)(1 - \alpha) = 2 + \epsilon - \alpha$$

**Example:** If it takes 20 nsec to search the associative registers and **100 nsec to access** memory and hit ratio is 80 %, then, it takes 20 nanosecond to search the TLB. memory access takes 120nanosecond .->  $120+100=220$ (total memory access time)

**Effective access time=hit ratio\*Associate memory access time +miss ratio\* memory access time.**

$$0.80 * 120 + 0.20 * 220 = 140 \text{ nsec.}$$

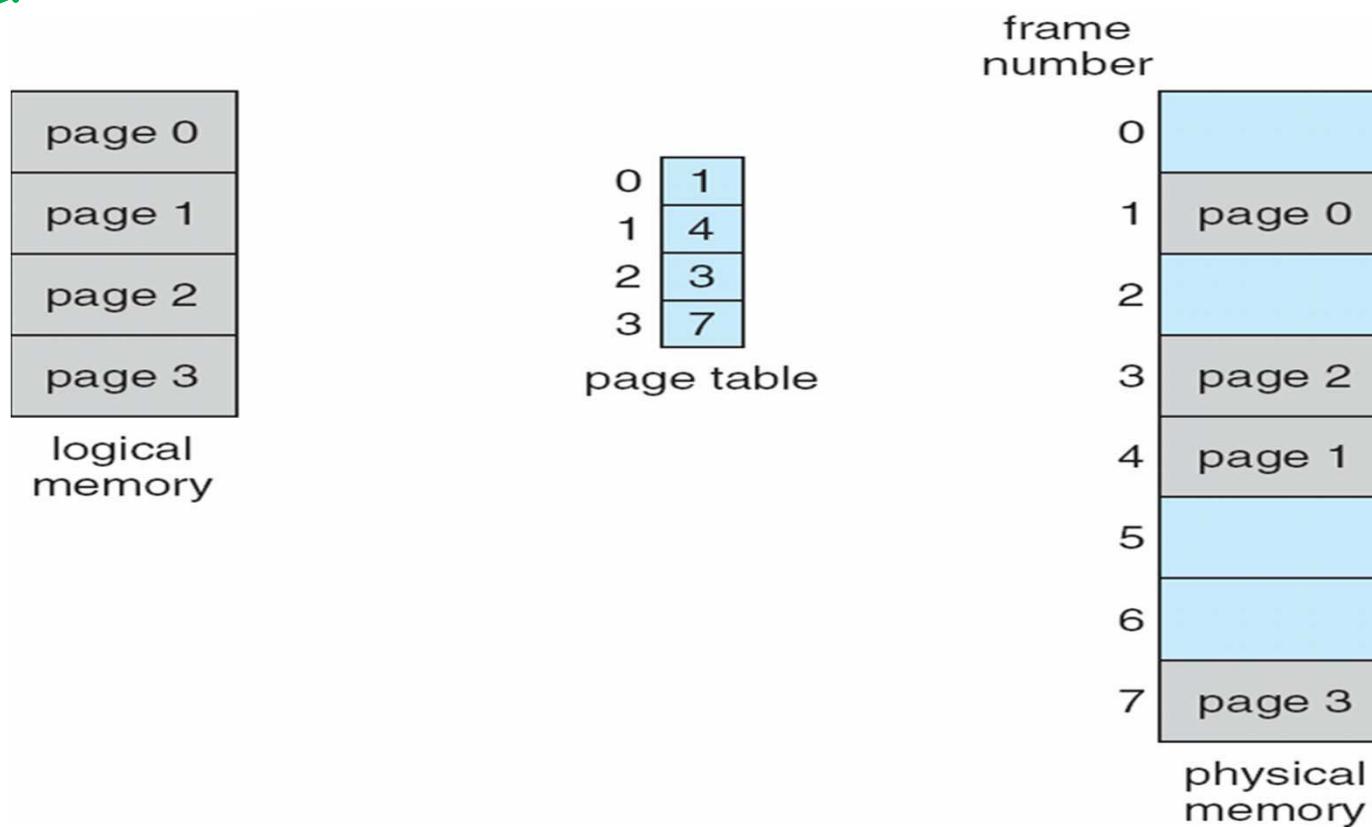
**40 % slowdown.**

- For 98-percent hit ratio, we have

$$\text{Effective access time} = 0.98 * 120 + 0.02 * 220$$

$$= 122 \text{ nanoseconds} = 22 \% \text{ slowdown.}$$

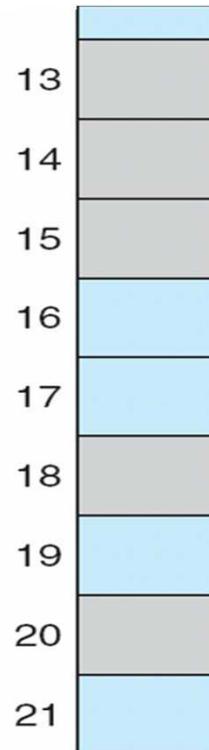
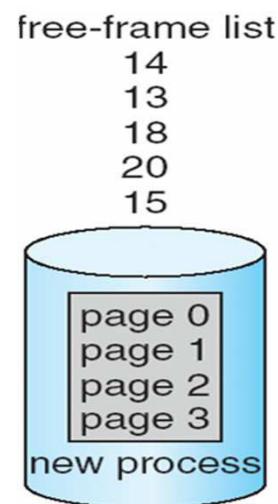
## Contd.



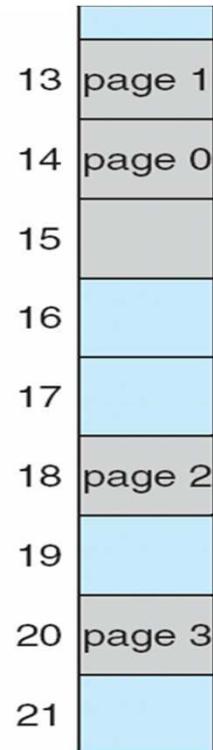
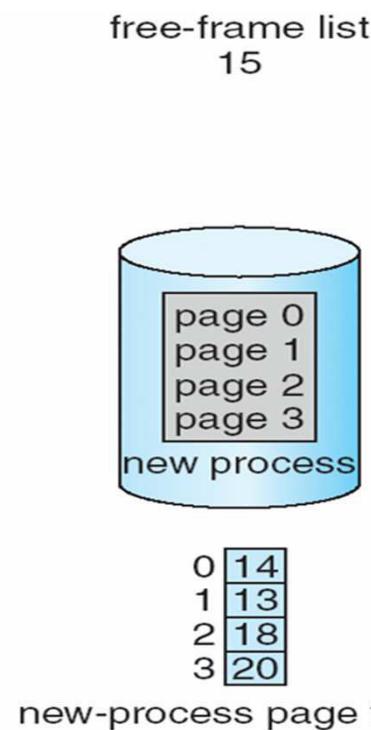
- **Page size= 4 bytes; Physical memory=32 bytes (8 pages)**
  - Logical address 0 maps 1\*4+0=4
  - Logical address 3 maps to= 1\*4+3=7
  - Logical address 4 maps to =4\*4+0=16
  - Logical address 13 maps to= 7\*4+1=29.

## Free Frames

- ❖ When a process arrives the size in pages is examined
- ❖ Each page of process needs one frame.
- ❖ If n frames are available these are allocated, and page table is updated with frame number.



(a) Before allocation

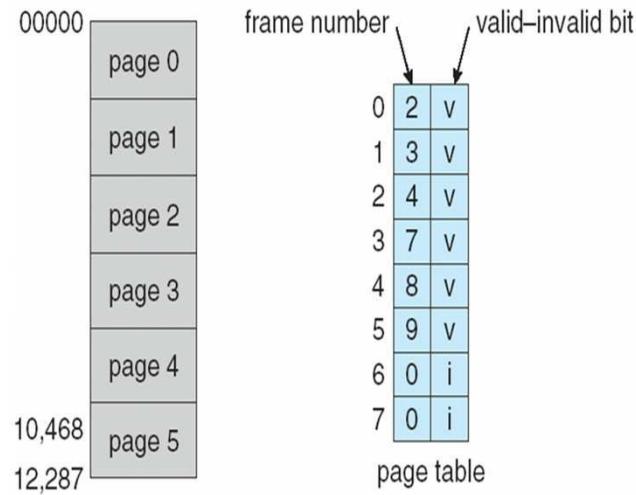


(b) After allocation

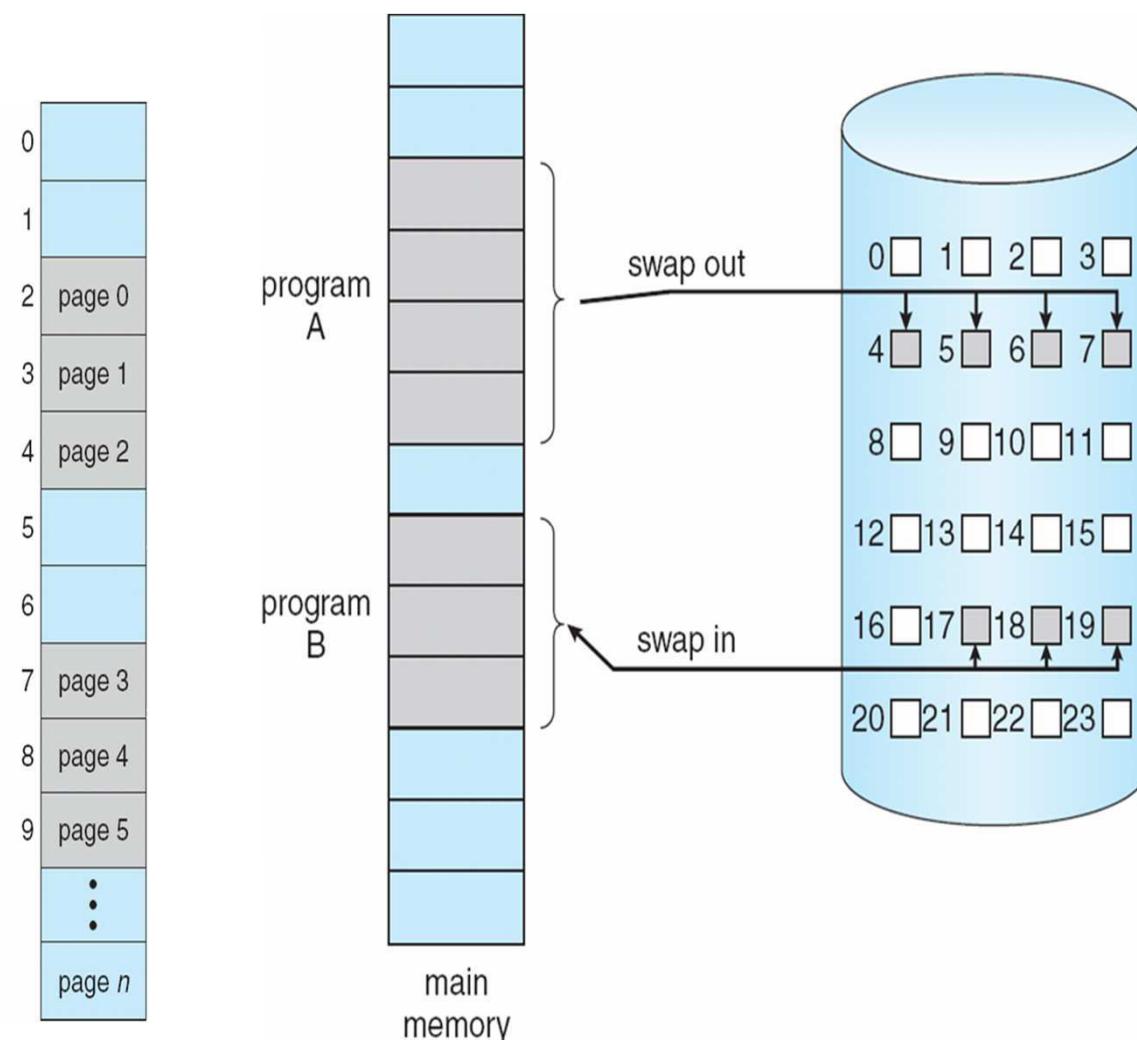
## Memory Protection

- ☞ Is a way to control memory access rights on a computer, and is a part of most modern operating systems.
- ☞ The main purpose of memory protection is to *prevent a process from accessing memory that has not been allocated to it.*
- ☞ This prevents a bug within a process from affecting other processes, or the operating system itself.
- ☞ Memory protection implemented by associating a **protection bit** with each frame.
- ☞ **Valid-invalid bit** attached to each entry in the page table:
  1. “**valid**” indicates that the associated page is in the process’ logical address space, and is thus a legal page.
  2. “**invalid**” indicates that the page is not in the process’ logical address space.

## Contd.



(a) Valid(v) or invalid(i) bit in a page table

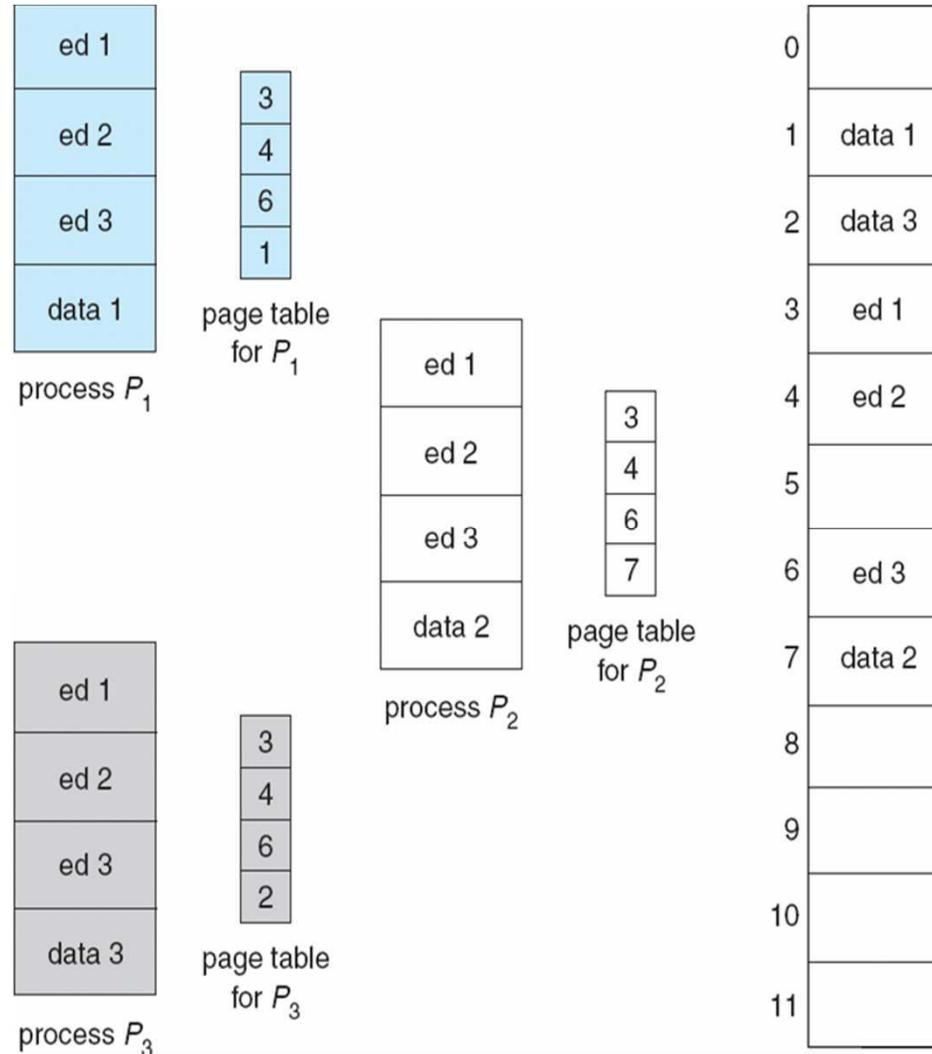


(b) Transfer of a Paged Memory to Contiguous Disk Space

# Shared Pages

## Shared code

- One copy of **read-only code shared among processes (i.e., text editors, compilers, window systems)**.
- Shared code **must appear in same location in the logical address space of all processes**



## 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

## Page Table Types

- ❖ Essentially, a page table must store the **virtual address**, the **physical address**, and possibly some address space information like a bit that is set when the page is modified, the protection code (read/write/execute permissions).
- ❖ There are several different types of page tables, that are best suited for different requirements.

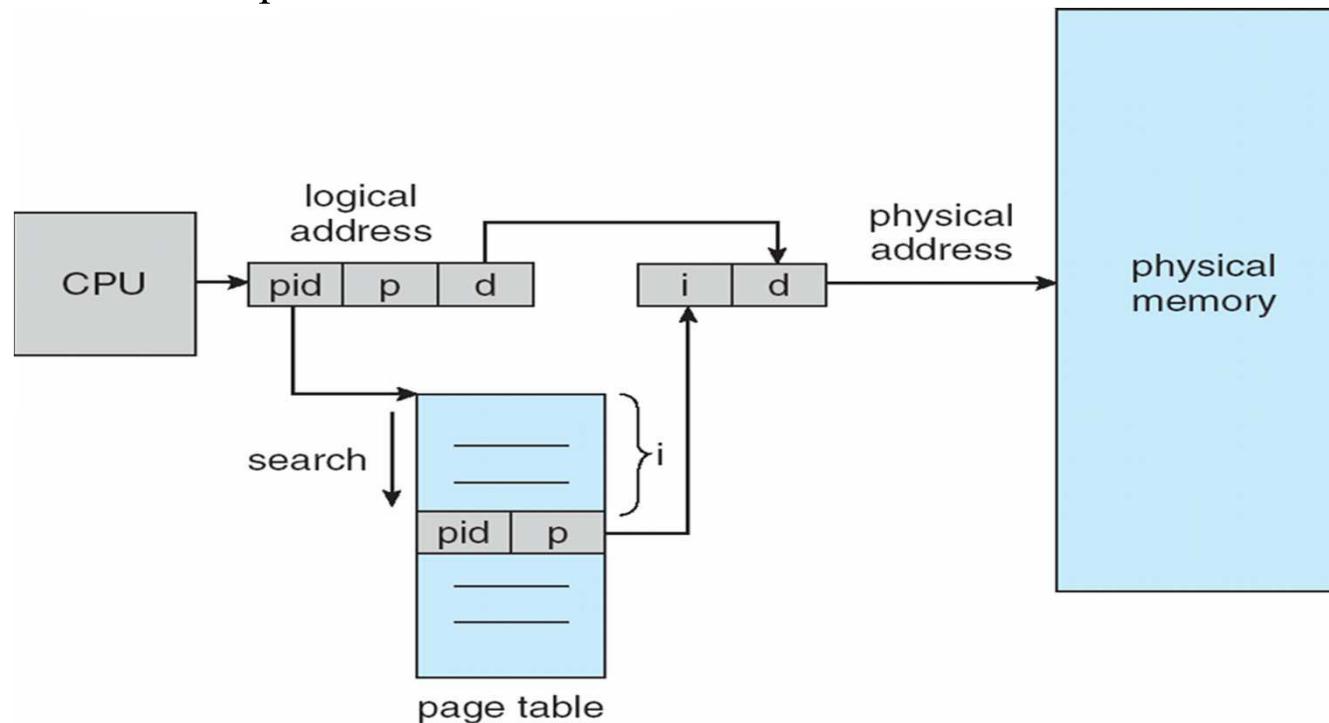
- A. **Inverted Page Tables**
- B. **Multilevel Page Tables**
- C. **Virtualized Page Table**
- D. **Nested Page Tables**

## A. Inverted Page Tables

- Traditional page tables require one entry per virtual page, since they are indexed by virtual page number.
- As virtual address spaces grow so much bigger than physical addresses, it becomes possible that we can save space and time by inverting the page table, **mapping physical to virtual**, instead of the other way around.
- IPTE **decreases** memory needed to store each page table, but increases time **needed to search the table** when a page reference occurs.
- Although inverted page tables save vast amounts of space, they have a serious downside: **virtual-to-physical translation becomes much harder** and **time consuming**.

## Contd.

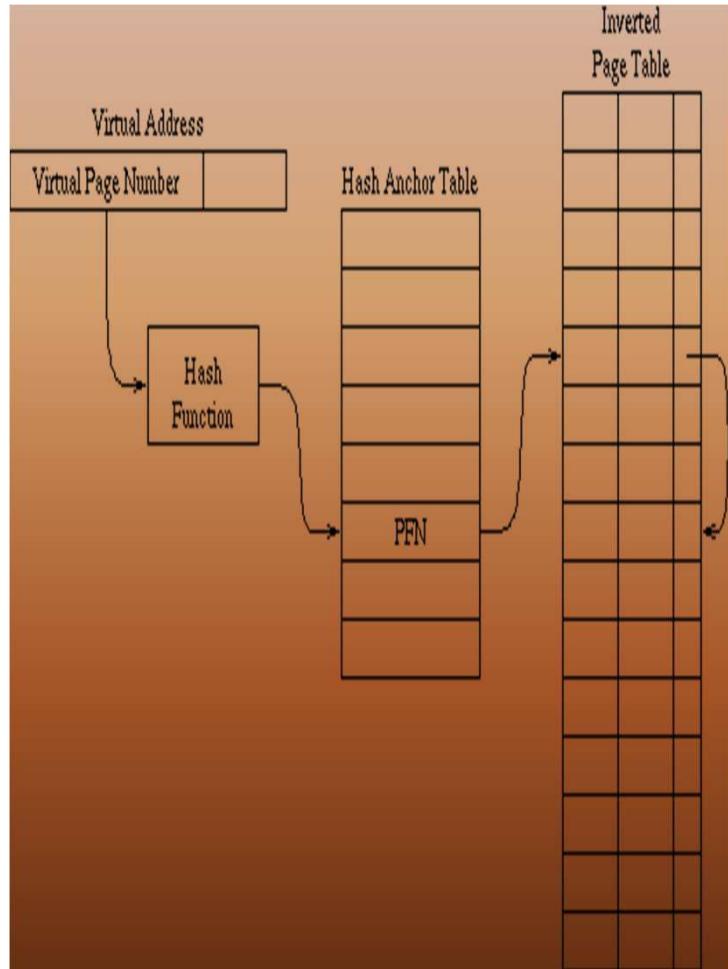
- Then, each inverted Page table entry (IPTE) has to contain:
  - The process ID of the owning process.
  - The virtual page number.
  - A pointer to the next IPTE in the hash chain.
  - The normal protection, valid, modified bits referenced.



## Translation steps

1. Hash the process ID(PID) and virtual page number (VPN) to get an index into the HAT (hash anchor table).
2. Look up a Physical Frame Number in the HAT.
3. Look at the inverted page table entry, to see if it is the right PID and VPN. If it is, you're done.
4. If the PID or VPN does not match, follow the pointer to the next link in the hash chain. Again, if you get a match then you're done; if you don't, then you continue.

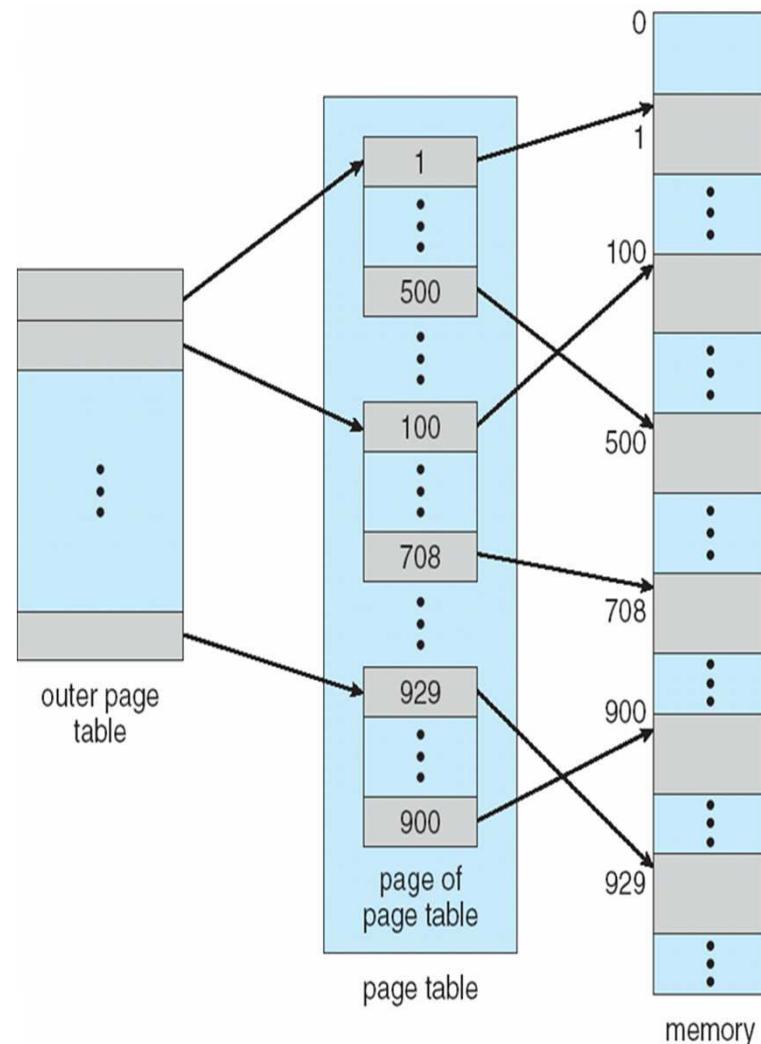
- Eventually, you will either get a match or you will find a pointer that is marked invalid. If you get a match, then you've got the translation; if you get the invalid pointer, then you have a miss.



## B. Multilevel Page Tables

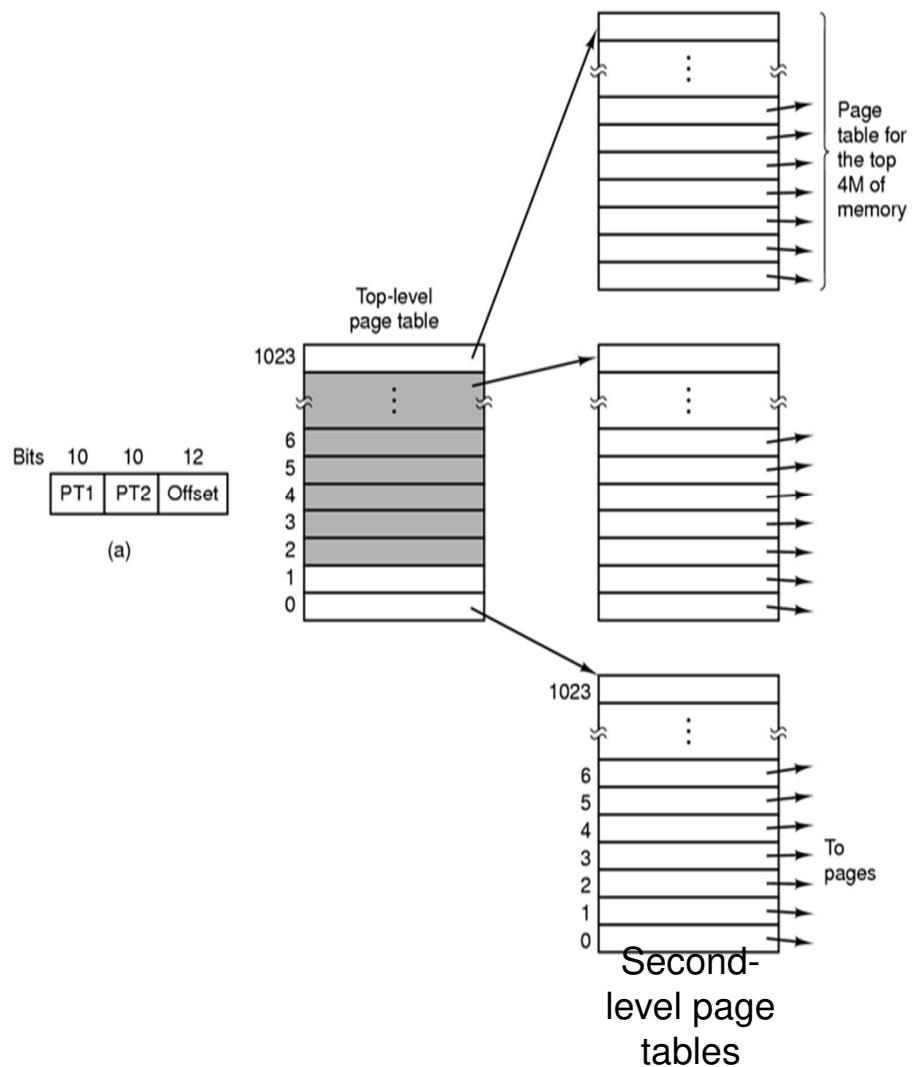
- ☞ Also known as **hierarchical page tables**.
- ☞ **Is a means of using page tables for large address spaces.**
- ☞ Instead of having a table which keeps a listing of mappings for all frames in physical memory, in case of IPT, we could create a page table structure that contains mappings for virtual pages.
  - Break up the logical address space into multiple page tables.
  - Each of these smaller page tables are linked together by a **master page table**, effectively creating a tree data structure.
- ☞ **The secret to the multilevel page table method is to avoid keeping all the page tables in memory all the time.**
- ☞ A **virtual address** in this schema could be split into **three**, the **index in the root page table**, the **index in the sub-page table**, and the **offset** in that page.

Contd.



Two-Level Page-Table Scheme

## Two-Level Page-Table Example

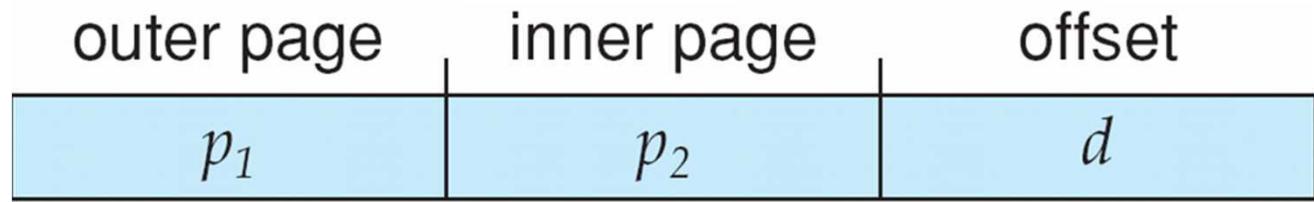


## Two-Level Paging Example

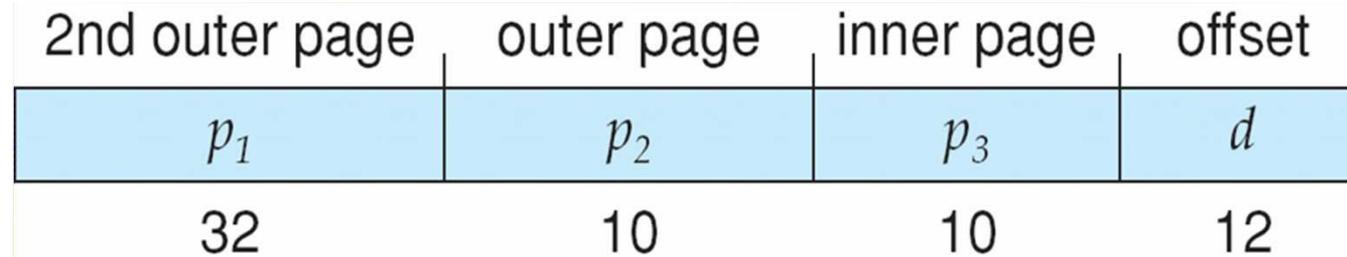
- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- | page number |       | page offset |
|-------------|-------|-------------|
| $P_i$       | $P_2$ | $d$         |
| 12          | 10    | 10          |

Thus, a logical address is as follows:

## Three-Level Paging Scheme (64-bit)



42                    10                    12



32                    10                    10                    12

Examples: 32-bit SPARC (three-level), 32-bit Motorola 68030 (four-level)

## C. Virtualized Vs. Nested Page Tables

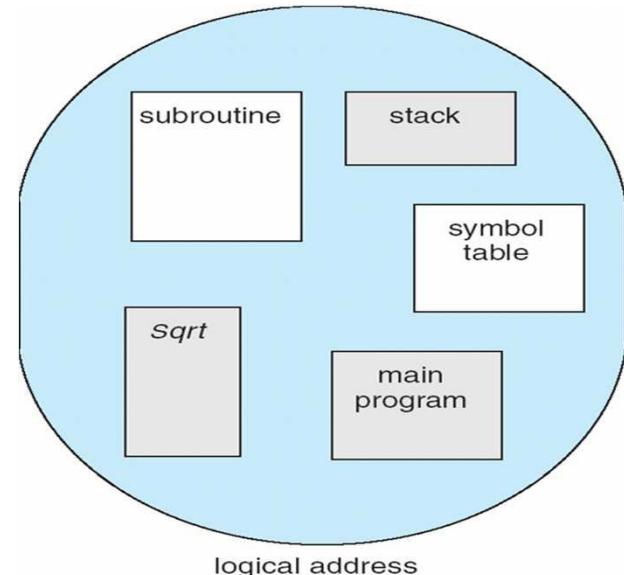
- ❖ It was mentioned that creating a page table structure that contained mappings for every virtual page in the virtual address space could end up being wasteful.
- ❖ But, we can get around the excessive space concerns by putting the page table in **virtual memory**, and letting the virtual memory system manage the memory for the page table.
- ❖ **However, part of this linear page table structure must always stay resident in physical memory**, in order to prevent against circular page faults, that look for a key part of the page table that is not present in the page table, which is not present in the page table, etc.
- ❖ Nested page tables can be implemented to increase the performance of hardware virtualization.

### III. Segmentation

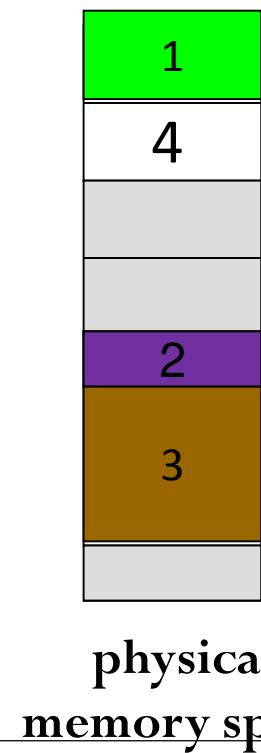
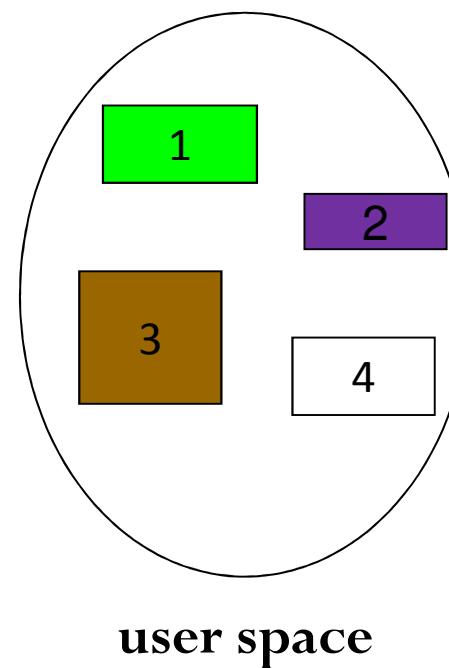
- ☞ The virtual memory discussed so far is **one-dimensional** because the virtual addresses go from 0 to some maximum address, one address after another.
- ☞ But it is possible to provide the machine with many completely **independent address spaces**, called **segments**; using segmentation.
  - Each segment consists of a linear sequence of addresses, from 0 to some maximum.
- ☞ The length of each segment may be anything from 0 to the maximum allowed.
  - **Different segments may, and usually do, have different lengths.**
- ☞ Because each segment constitutes a separate address space, different segments can grow or shrink independently, without affecting each other.

## Contd.

- ☞ Segmentation supports a user's view of a program.
- ☞ A program is a collection of segments, logical units, such as: **main program, subprogram, class, procedure, function, object, method, local variables, global variables, common block, stack, symbol table, arrays.**



User's view of a program



## Dynamics of Simple Segmentation

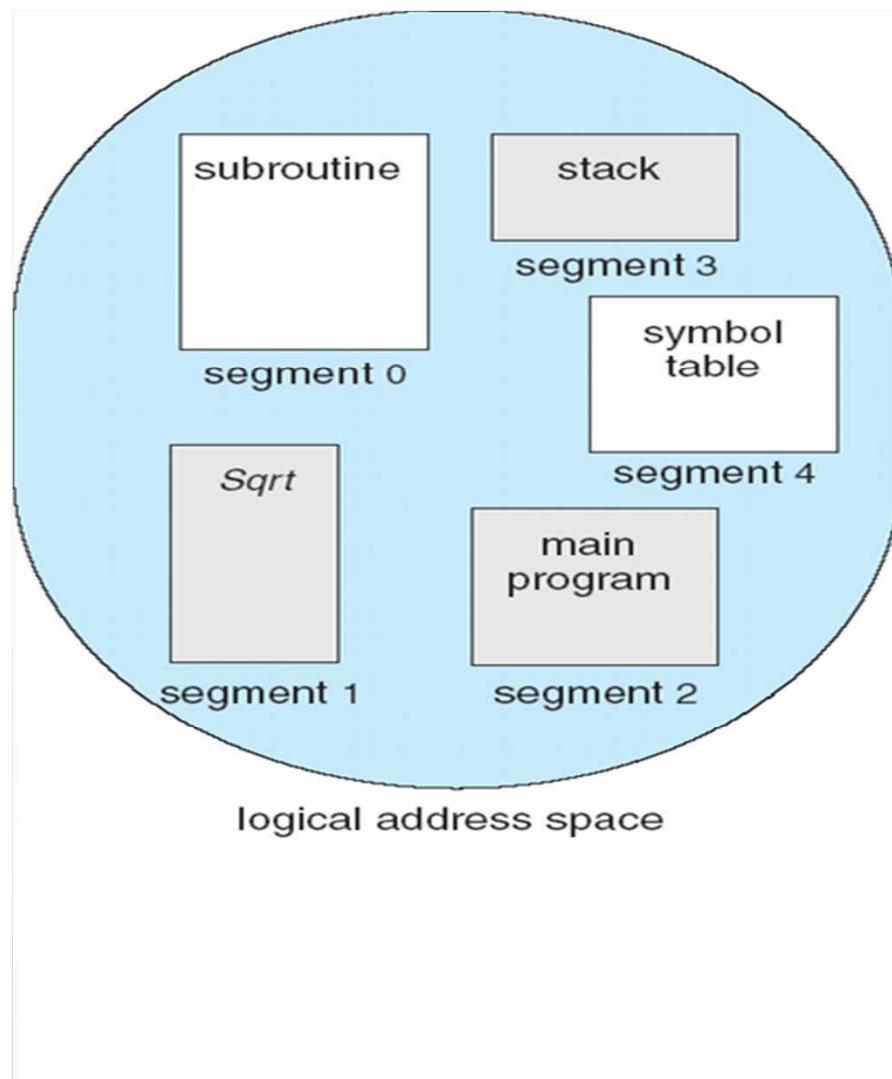
- ☞ Each program is subdivided into blocks of non-equal size called segments.
- ☞ When a process gets loaded into main memory, its different segments can be located anywhere.
- ☞ Each segment is fully packed with instructions/data;
  - **No internal fragmentation.**
  - There is **external fragmentation**; it is reduced when using small segments.
- ☞ In contrast with paging, **segmentation is visible to the programmer**
  - provided as a convenience to organize logically programs (example: data in one segment, code in another segment).
  - must be aware of segment size limit.
- ☞ The OS maintains a segment table for each process.
- ☞ Each entry contains:
  - **the starting physical addresses of that segment.**
  - **the length of that segment (for protection).**

## 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
    1. **Base:-** contains the starting physical address where the segments reside in memory.
    2. **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;

Contd.



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table

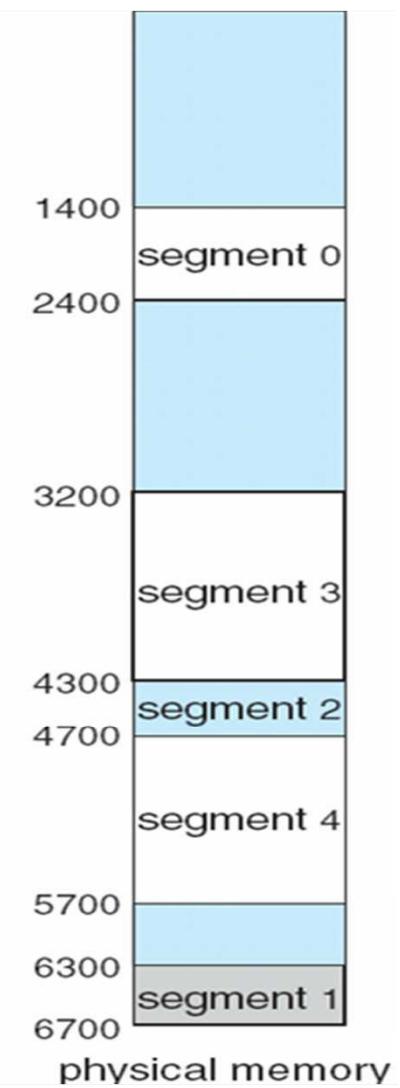


Fig. Segmentation Example

# Protection in Segmentation

- ☞ Protection – with each entry in segment table associate:
  - validation bit = 0  $\Rightarrow$  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**.

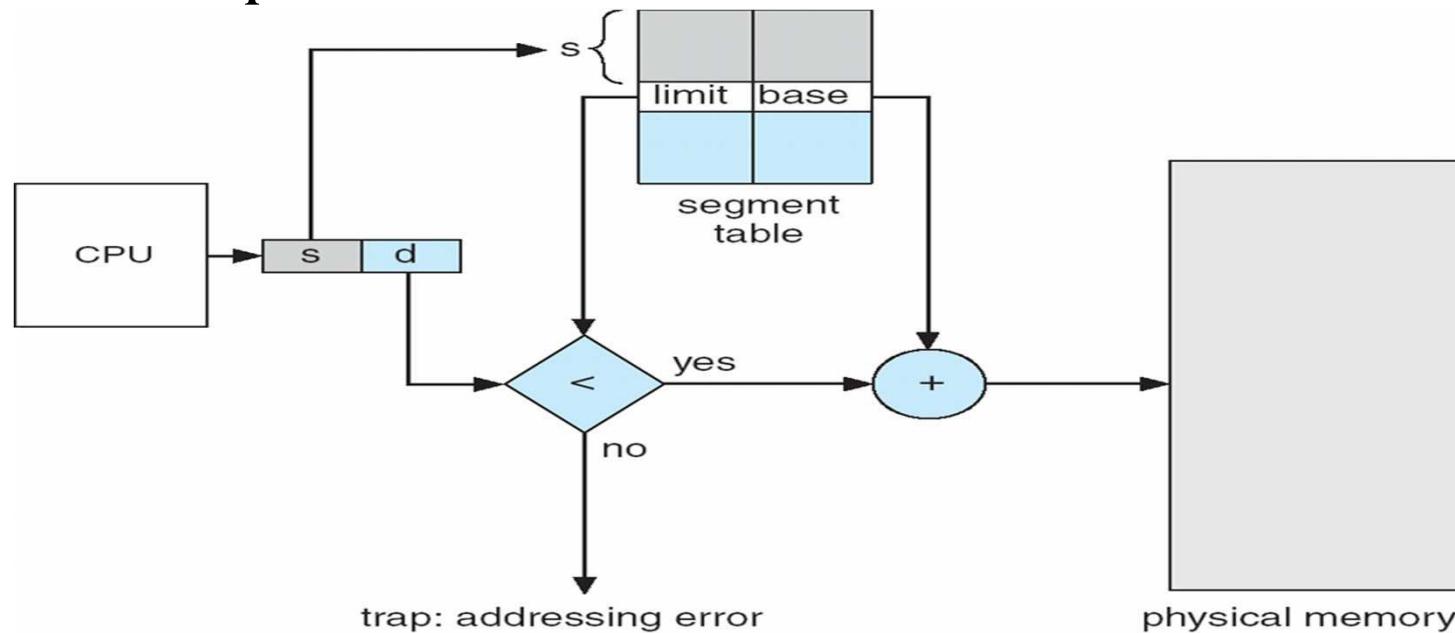


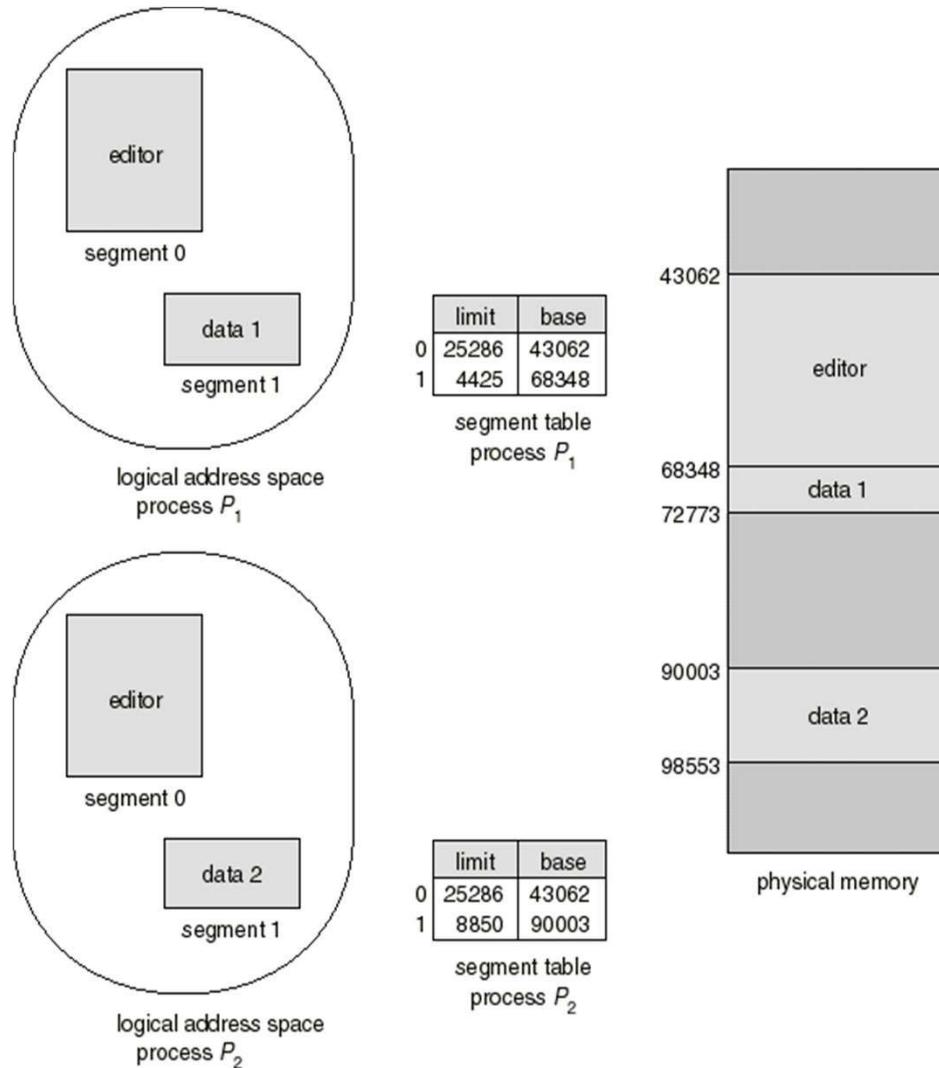
Fig. Segmentation Hardware

## Sharing in Segmentation Systems

- Segments are shared when entries in the segment tables of two different processes point to the same physical locations.

Example: the same code of a text editor can be shared by many users:

- Only one copy is kept in main memory.
- But each user would still need to have its own private data segment.



**Fig.** Sharing Segments

## Comparison Between Paging and Segmentation

- ☞ Segmentation is visible to the programmer whereas paging is transparent.
- ☞ Segmentation, naturally supports protection/sharing.
- ☞ Segmentation can be viewed as commodity offered to the programmer to logically organize a program into segments while using different kinds of protection (example: execute-only for code but read-write for data).
- ☞ Segments are variable-size; Pages are fixed-size.
- ☞ Segmentation requires more complicated hardware for address translation than paging.
- ☞ Segmentation suffers from external fragmentation. Paging only yields a small internal fragmentation.
  - Maybe combine Segmentation and Paging?

## Contd.

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection