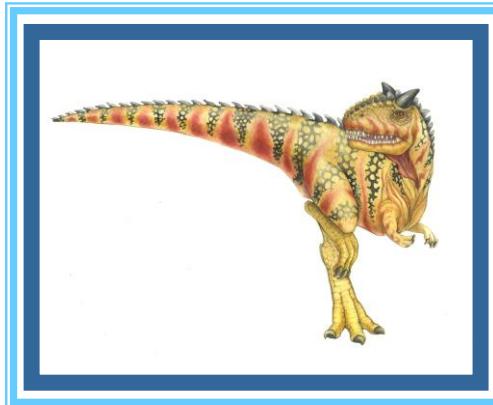


Chapter 1: Introduction

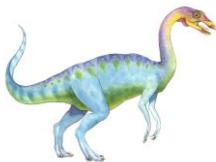




Chapter 1: Introduction

- What Operating Systems Do
- Operating-System Structure
- Operating-System Operations
- Process Management
- Memory Management
- Storage Management

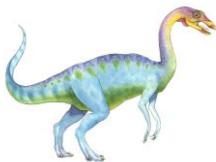




Objectives

- To describe the basic organization of computer systems
- To provide a grand tour of the major components of operating systems
- To give an overview of the many types of computing environments





What is an Operating System?

- An operating system is a program that manages a computer's hardware.
- A program that acts as an intermediary between a user of a computer and the computer hardware
- some operating systems are designed to be convenient, others to be efficient, and others to be some combination of the two.
- 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





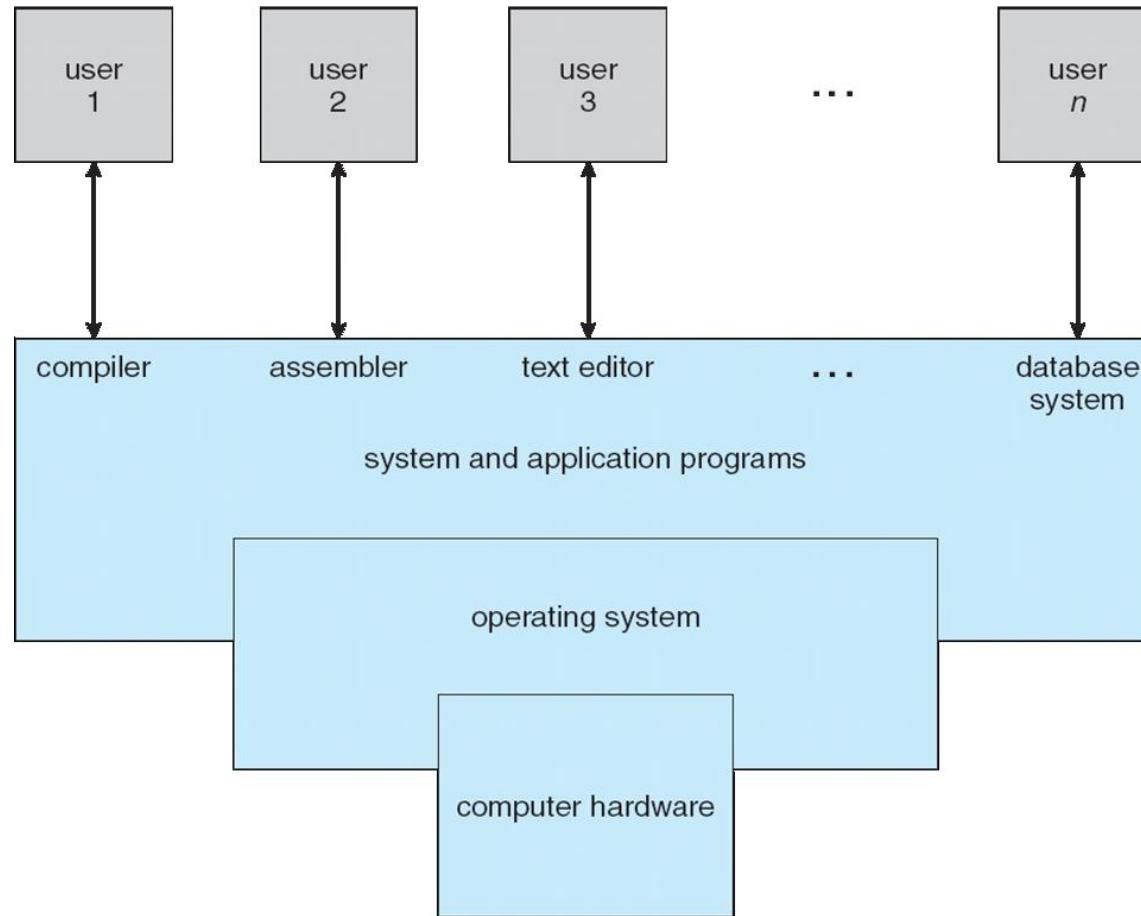
Computer System Structure

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



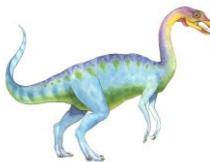


Four Components of a Computer System



Abstract view of the components of a computer system





What Operating Systems Do

- Depends on the **point of view –User view & System view**

User View:

- **PC** users want convenience, **ease of use** and **good performance**
 - Don't care about **resource utilization, dedicated to single user experience**
- But shared computer such as **mainframe** or **minicomputer** must keep all users happy
- Users of dedicate systems such as **workstations** have dedicated resources but frequently use shared resources from **servers**
- Handheld computers are resource poor, optimized for usability and battery life
- Some computers have little or no user interface, such as embedded computers in devices and automobiles.





Operating System Definition

System View: OS is the program involved with hardware.

Thus, we can view OS as

- OS is a **resource allocator**
 - Manages all resources (CPU, memory, disk etc)
 - Decides between conflicting requests for efficient and fair resource use
- OS is a **control program**
 - Controls execution of programs to prevent errors and improper use of the computer

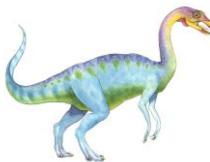




Operating System Definition (Cont.)

- No universally accepted definition
- Simple viewpoint : “Everything a vendor ships when you order is an operating system” is a good approximation
- More common definition:
“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.





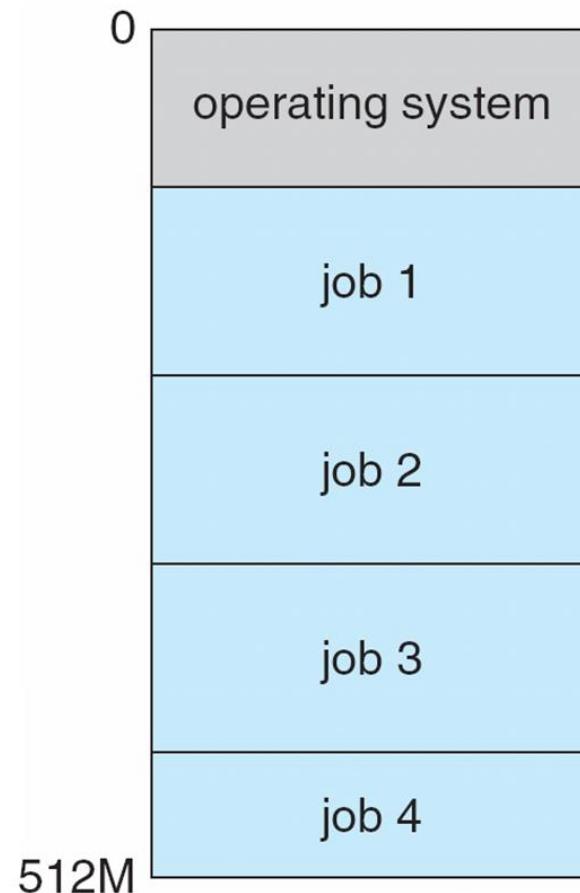
Operating System Structure

- **Multiprogramming (Batch system)** needed for efficiency
 - Single user cannot keep CPU and I/O devices busy at all times
 - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
 - A subset of total jobs in system is kept in memory
 - One job selected and run via **job scheduling**
 - When it has to wait (for I/O for example), OS switches to another job
- **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
 - **Response time** should be < 1 second
 - Each user has at least one program executing in memory \Rightarrow **process**
 - If several jobs ready to run at the same time \Rightarrow **CPU scheduling**
 - If processes don't fit in memory, **swapping** moves them in and out to run
 - **Virtual memory** allows execution of processes not completely in memory





Memory Layout for Multiprogrammed System





Operating-System Operations

- **Interrupt driven** (hardware and software)
 - Hardware interrupt by one of the devices
 - Software interrupt (**exception** or **trap**):
 - ▶ Software error (e.g., division by zero)
 - ▶ Request for operating system service
 - ▶ Other process problems include infinite loop, processes modifying each other or the operating system





Operating-System Operations (cont.)

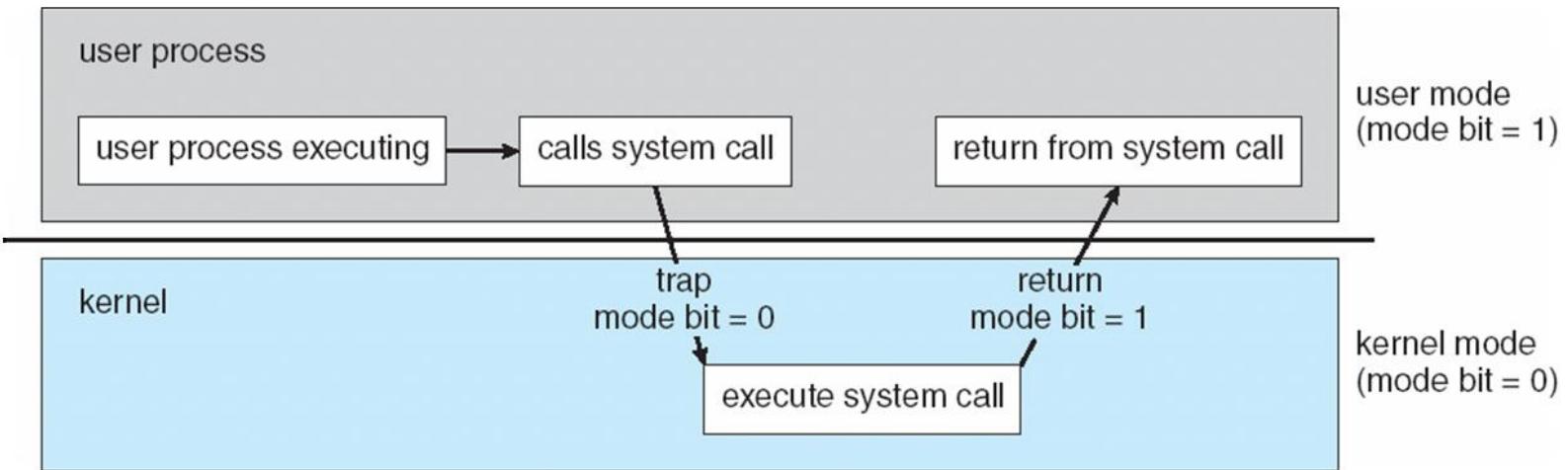
- Dual-mode operation allows OS to protect itself and other system components
 - User mode and kernel mode
 - Mode bit provided by hardware
 - ▶ Provides ability to distinguish when system is running user code or kernel code
 - ▶ Some instructions designated as privileged, only executable in kernel mode
 - ▶ System call changes mode to kernel, return from call resets it to user
- Increasingly CPUs support multi-mode operations
 - i.e. virtual machine manager (VMM) mode for guest VMs

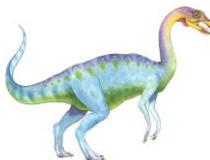




Transition from User to Kernel Mode

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

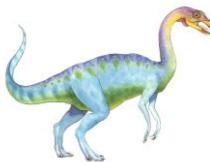




Process Management

- 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

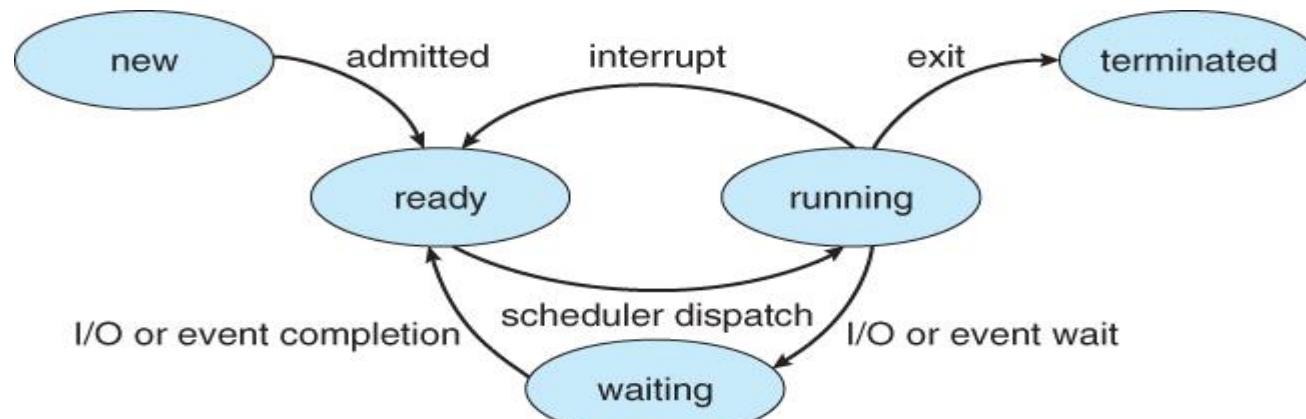




Process Management Activities

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

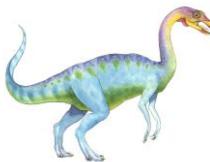




Memory Management

- To execute a program all (or part) of the instructions must be in memory
- All (or part) of the data that is needed by the program must be in memory.
- Memory management determines what is in memory and 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 – by OS or applications





Performance of Various Levels of Storage

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

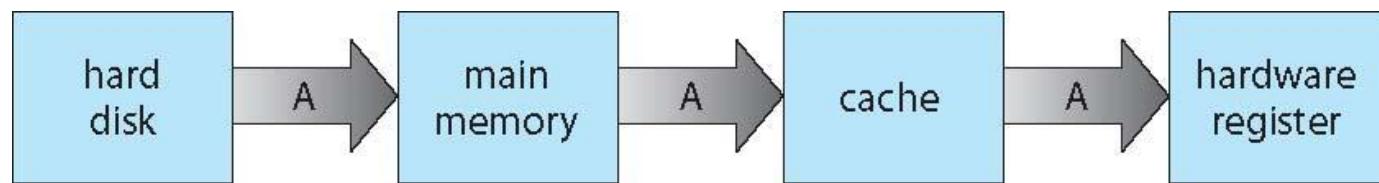
Movement between levels of storage hierarchy can be explicit or implicit





Migration of data “A” from Disk to Register

- Multitasking environments must be careful to use most recent value, no matter where it is stored in the storage hierarchy



- Multiprocessor environment must provide **cache coherency** in hardware such that all CPUs have the most recent value in their cache



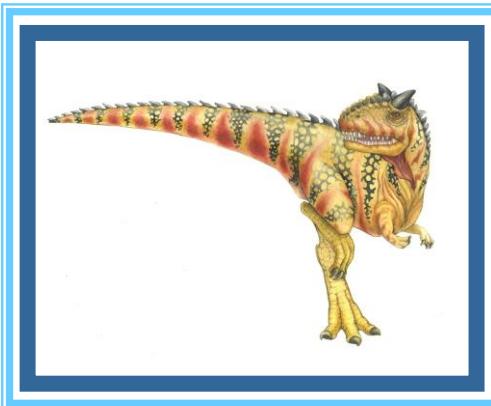


I/O Subsystem

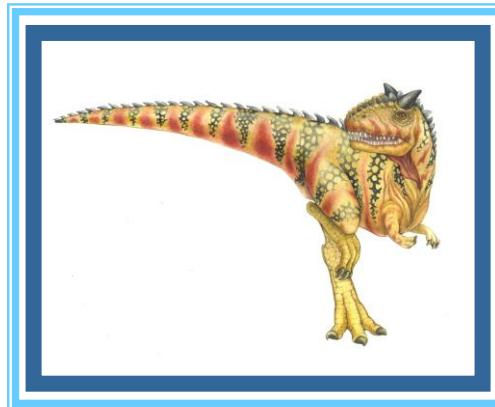
- One purpose of OS is to hide peculiarities of hardware devices from the user
- I/O subsystem responsible for
 - Memory management of I/O including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping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



End of Chapter 1



Chapter 2: Operating-System Structures





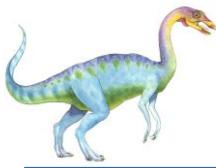
Chapter 2: Operating-System Structures

In this chapter, we explore all three aspects of operating systems,

- 1) Services that the system provides
- 2) An interface that makes it available to users and programmers
- 3) Its components and their interconnections

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Structure
- System Boot





Objectives

- To describe the services an operating system provides to users, processes, and other systems

- To discuss the various ways of structuring an operating system





Operating System Services

There are 9 Operating System Services

- 1. User Interface**
- 2. Program execution**
- 3. I/O Operations**
- 4. File System Manipulation**
- 5. Communications**
- 6. Error Detection**
- 7. Resources Allocation**
- 8. Accounting**
- 9. Protection and Security**





Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users
- One set of operating-system services provides functions that are helpful to the user:
 - **User interface** - Almost all operating systems have a user interface ([UI](#)).
 - ▶ Varies between [Command-Line \(CLI\)](#), [Graphics User Interface \(GUI\)](#), [Batch](#)
 - **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
 - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device





Operating System Services (Cont.)

- One set of operating-system services provides functions that are helpful to the user (Cont.):
 - **File-system manipulation** - The file system is of particular interest. Programs need to read and write **files** and **directories**, create and delete them, search them, list file information, permission management.
 - **Communications** – Processes may exchange information, on the same computer or between computers over a network
 - ▶ Communications may be via **shared memory** or through **message passing** (packets moved by the OS)
 - **Error detection** – OS needs to be constantly aware of possible errors
 - ▶ May occur in the CPU and memory hardware, in I/O devices, in user program
 - ▶ For each type of error, OS should take the appropriate action to ensure correct and consistent computing
 - ▶ Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system





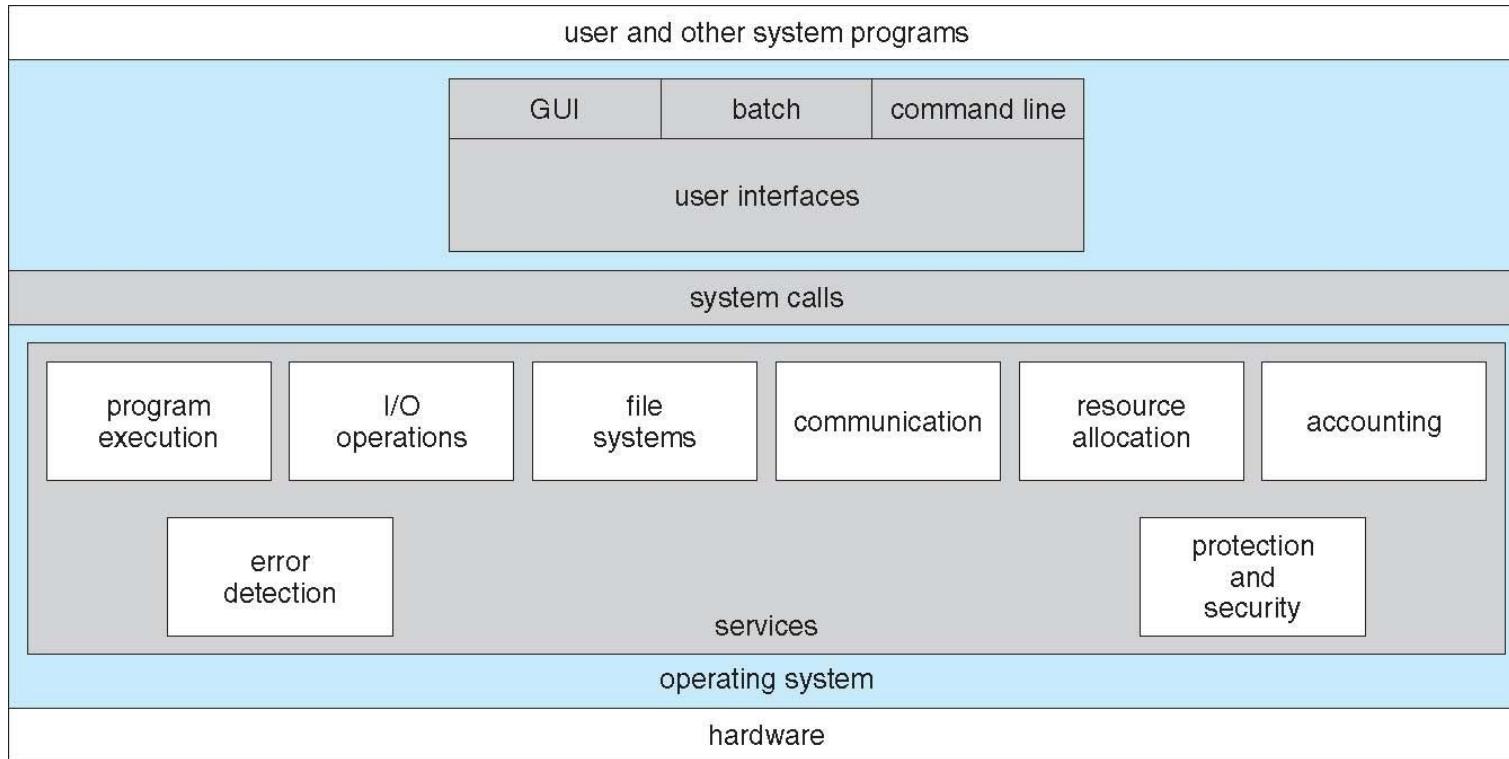
Operating System Services (Cont.)

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
 - **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
 - ▶ Many types of resources - **CPU cycles, main memory, file storage, I/O devices.**
 - **Accounting** - To keep track of which users use how much and what kinds of computer resources
 - **Protection and security** - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
 - ▶ **Protection** involves ensuring that all access to system resources is controlled
 - ▶ **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts





A View of Operating System Services





User Operating System Interface - CLI

CLI or **command interpreter** allows direct command entry

- OS provides **command-line interface**, or command interpreter that allows users to directly enter commands to be performed by the operating system.
- Sometimes implemented in kernel, sometimes by systems program
- Sometimes multiple flavors implemented – **shells**
- Primarily fetches a command from user and executes it
- Sometimes commands are built-in, sometimes just names of programs(implements most commands through system programs)
 - ▶ If the latter, adding new features doesn't require shell modification





User Operating System Interface - GUI

- User-friendly **desktop** interface
 - Usually mouse, keyboard, and monitor
 - **Icons** represent files, programs, actions, etc
 - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
- Many systems now include both CLI and GUI interfaces
 - Microsoft Windows is GUI with CLI “command” shell
 - Apple Mac OS X is “Aqua” GUI interface with UNIX kernel underneath and shells available
 - Unix and Linux have CLI with optional GUI interfaces



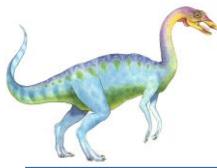


Touchscreen Interfaces

Touchscreen devices require new interfaces

- Mouse not possible or not desired
- Actions and selection based on gestures
- Virtual keyboard for text entry
- Voice commands.





The Mac OS X GUI



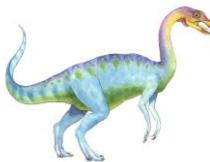


System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

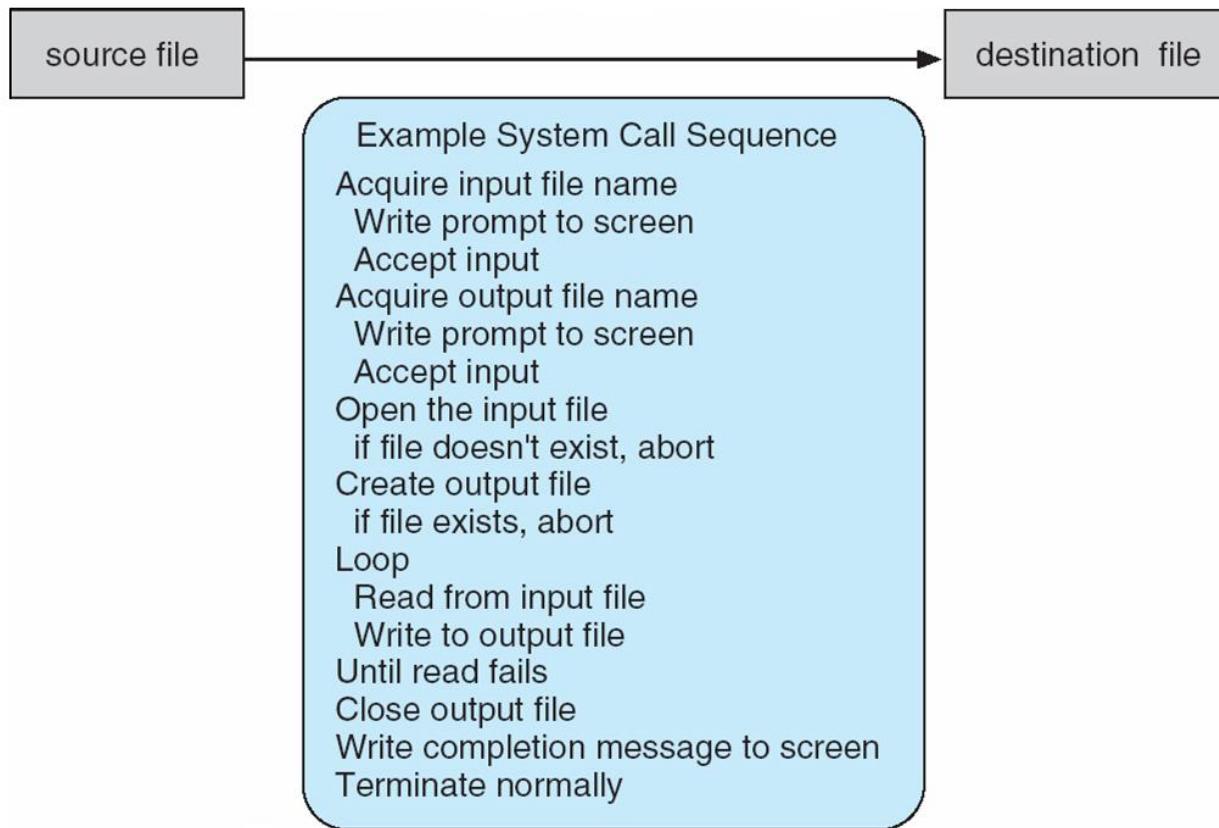
Note that the system-call names used throughout this text are generic





Example of System Calls

- System call sequence to copy the contents of one file to another file





Example of Standard API

EXAMPLE OF STANDARD API

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

```
man read
```

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

```
#include <unistd.h>

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

return value function name parameters

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

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

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





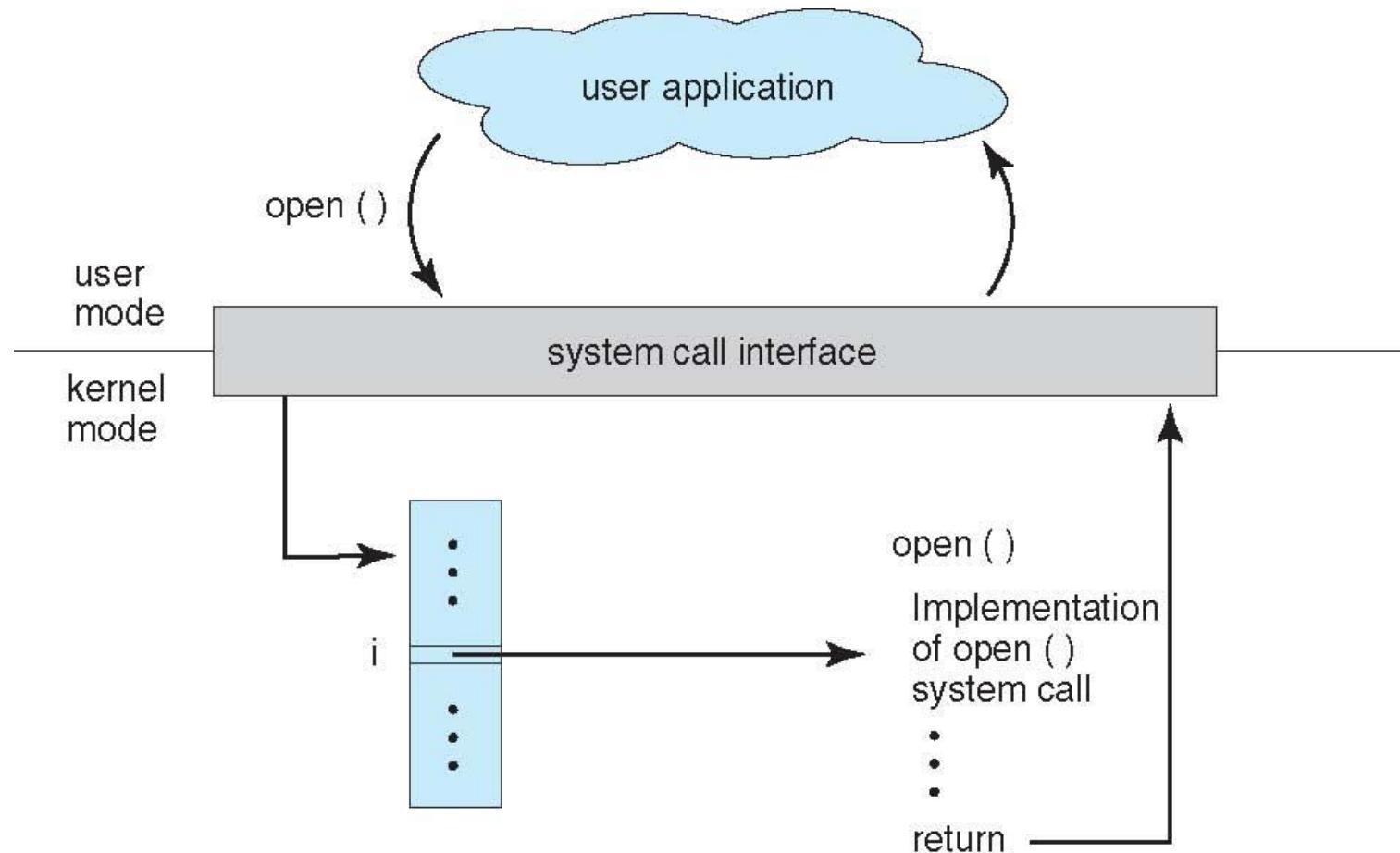
System Call Implementation

- Typically, a number associated with each system call
 - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
 - Just needs to obey API and understand what OS will do as a result call
 - Most details of OS interface hidden from programmer by API
 - ▶ Managed by run-time support library (set of functions built into libraries included with compiler)





API – System Call – OS Relationship





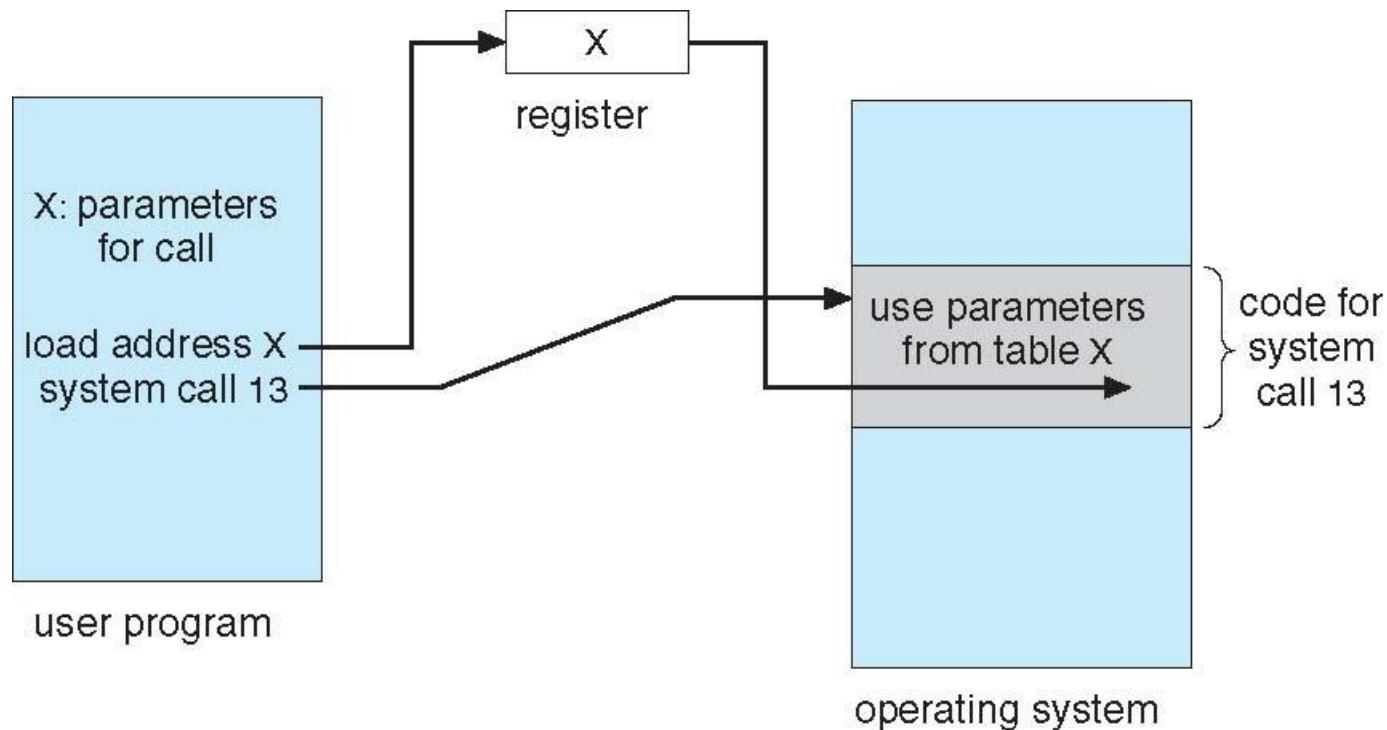
System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
 - Exact type and amount of information vary according to OS and call
- Three general methods used to **pass parameters** to the OS
 - Simplest: pass the parameters in **registers**
 - ▶ In some cases, may be more parameters than registers
 - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
 - ▶ This approach taken by Linux and Solaris
 - Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system
 - Block and stack methods do not limit the number or length of parameters being passed





Parameter Passing via Table



Passing of parameters as a table.



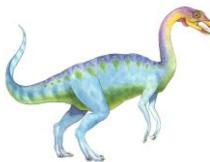


Types of System Calls

System calls can be grouped roughly into six major categories:

- ❖ **Process control**
- ❖ **File manipulation**
- ❖ **Device manipulation**
- ❖ **Information maintenance**
- ❖ **Communications**
- ❖ **Protection**



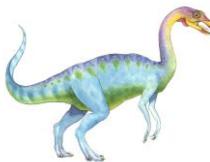


Types of System Calls

■ Process control

- create process, terminate process
- end, abort
- load, execute
- get process attributes, set process attributes
- wait for time
- wait event, signal event
- allocate and free memory
- Dump memory if error -corrupted system files, damaged HDD,
Corrupted RAM, Compatibility of h/w and s/w
- **Debugger** for determining **bugs, single step** execution
- **Locks** for managing access to shared data between processes
- Under either normal or abnormal circumstances, the operating system must transfer control to the invoking command interpreter. The command interpreter then reads the next command.





Types of System Calls

- File management
 - create file, delete file
 - open, close file
 - read, write, reposition
 - get and set file attributes
- Device management
 - request device, release device
 - read, write, reposition
 - get device attributes, set device attributes
 - logically attach or detach devices

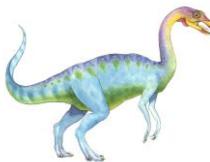




Types of System Calls (Cont.)

- Information maintenance
 - get time or date, set time or date
 - get system data, set system data
 - get and set process, file, or device attributes
- Communications: **message passing model**, **Shared-memory model**
 - create, delete communication connection
 - send, receive messages if **message passing model** to **host name** or **process name**
 - ▶ useful for exchanging **smaller amounts** of data
 - **Shared-memory model** create and gain access to memory regions
 - allows **maximum** speed
 - Must ensure processes are not writing to the same location simultaneously
 - attach and detach remote devices





Types of System Calls (Cont.)

- Protection
 - Control access to resources
 - Get and set permissions
 - Allow and deny user access
- Typically, system calls providing protection include **set permission()**, **get permission()**,





Examples of Windows and Unix System Calls

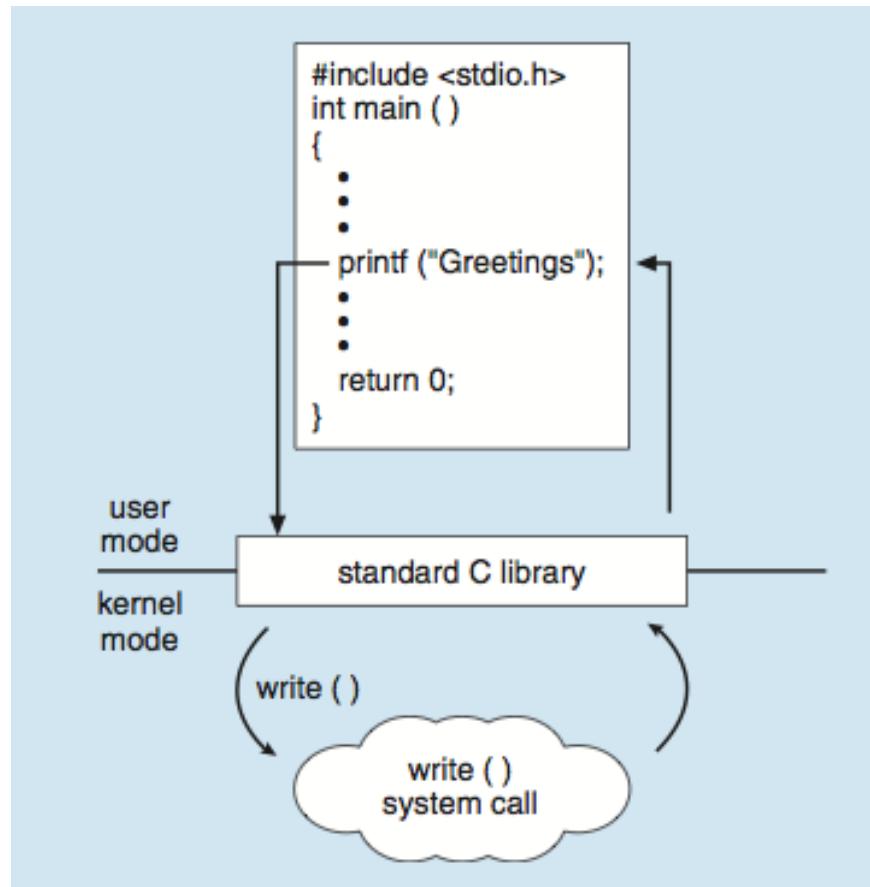
	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information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

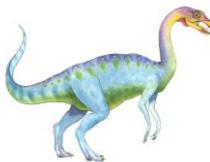




Standard C Library Example

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

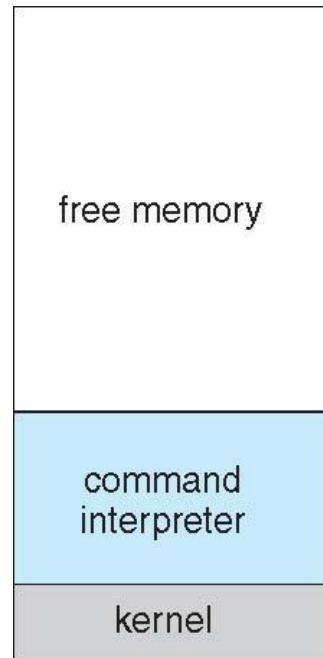




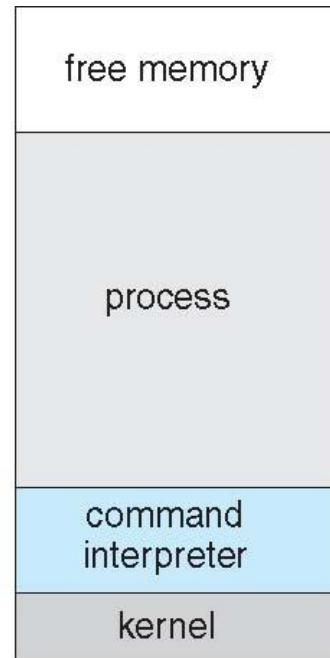
Variations in process control

Example: MS-DOS

- Single-tasking System
- Shell invoked when system booted
- Simple method to run program
 - Running a program **replaces the currently running program.**
 - The currently running program is overwritten,
- Single memory space
- Loads program into memory, overwriting all but the kernel
- Program exit -> shell reloaded



(a)
At system startup



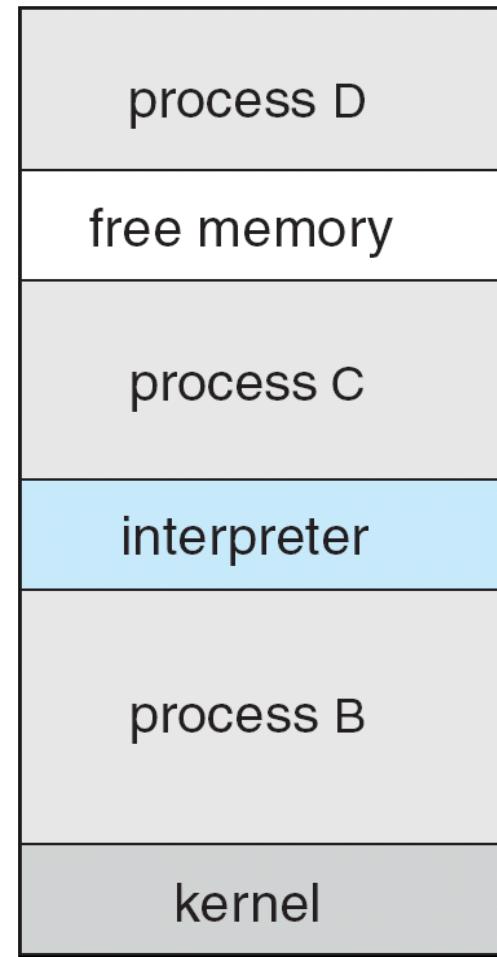
(b)
running a program



Example: FreeBSD

FreeBSD (**Free Berkeley Software Distribution**), FreeBSD is a **free and open-source Unix-like operating system**

- Unix variant
- Multitasking
- User login -> invoke user's choice of shell
- Shell executes fork() system call to create process
 - Executes exec() to load program into process
 - Shell waits for process to terminate or continues with user commands
- Process exits with:
 - code = 0 – no error
 - code > 0 – error code





System Programs

- System programs provide a convenient environment for program development and execution.

They can be divided into:

- File manipulation
 - Status information
 - ▶ File modification
 - Programming language support
 - Program loading and execution
 - Communications
 - Background services
 - Application programs
- Most users' view of the operating system is defined by system **programs**, not the actual **system calls**





System Programs

- Provide a convenient environment for program development and execution
 - Some of them are simply user interfaces to system calls; others are considerably more complex
- **File management** - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
 - Some ask the system for info - date, time, amount of available memory, disk space, number of users
 - Others provide detailed performance, logging, and debugging information
 - Typically, these programs format and print the output to the terminal or other output devices
 - Some systems implement a **registry** - used to store and retrieve configuration information





System Programs (Cont.)

■ File modification

- Text editors to create and modify files
- Special commands to search contents of files or perform transformations of the text

■ Programming-language support - Compilers, assemblers, debuggers and interpreters sometimes provided

■ Program loading and execution- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language

■ Communications - Provide the mechanism for creating virtual connections among processes, users, and computer systems

- Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another





System Programs (Cont.)

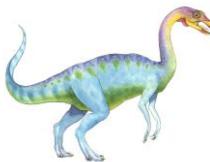
■ Background Services

- Launch at boot time
 - ▶ Some for system startup, then terminate
 - ▶ Some from system boot to shutdown
- Provide facilities like disk checking, process scheduling, error logging, printing
- Run in user context not kernel context
- Known as **services, subsystems, daemons**

■ Application programs

- Run by users
- Not typically considered part of OS
- Launched by command line, mouse click, finger poke





Operating System Structure

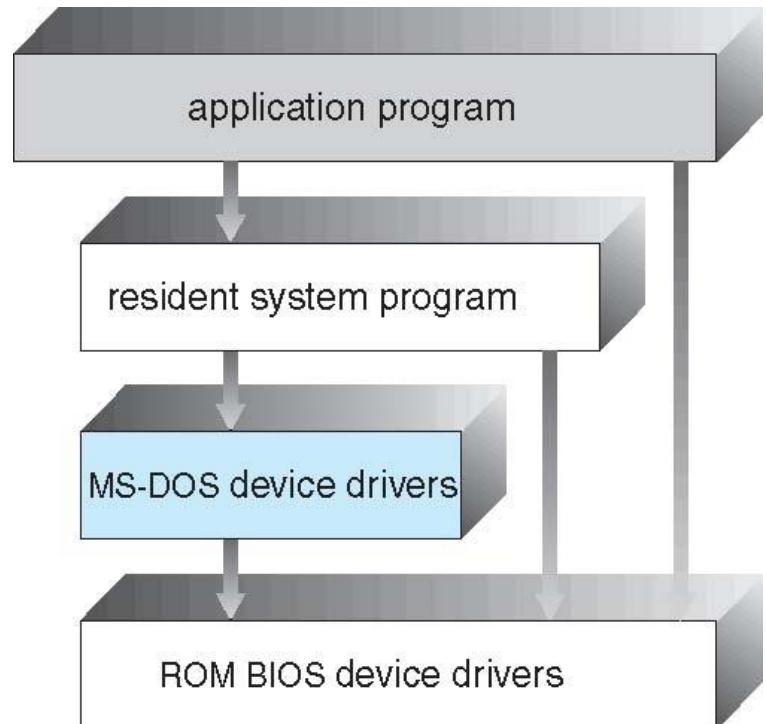
- General-purpose OS is very large program
- Various ways to structure ones
 - Simple structure – MS-DOS
 - More complex -- UNIX
 - Layered – an abstraction
 - Microkernel -Mach

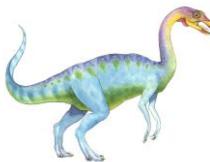




Operating System Structure

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





Non Simple Structure -- UNIX

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

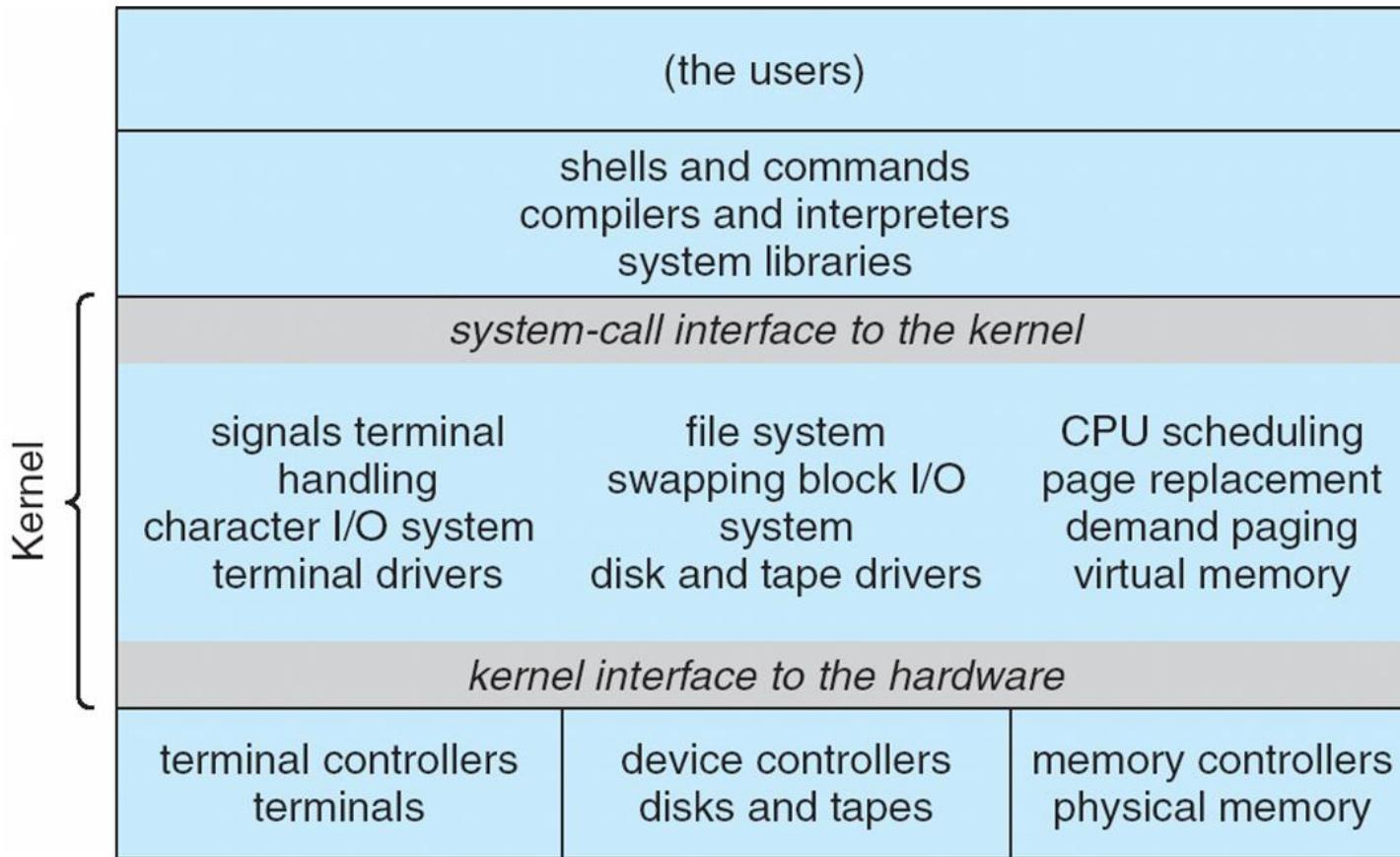
- Systems programs
- The kernel
 - ▶ Consists of everything below the system-call interface and above the physical hardware
 - ▶ Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level





Traditional UNIX System Structure

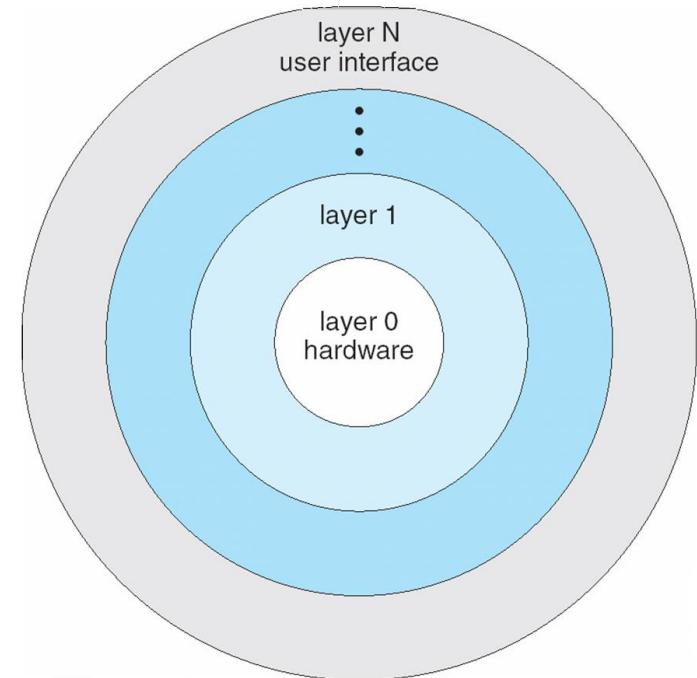
Beyond simple but not fully layered: **monolithic structure, difficult** to implement and maintain

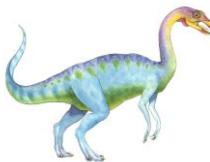




Layered Approach

- The operating system is divided into several layers (levels), each built on top of lower layers.
- The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With **modularity**, layers are selected such that each uses functions (operations) and services of only **lower-level layers**.
- Easy to implement and debug.
- The major difficulty with the layered approach involves appropriately defining the various layers.
- **Ex:** the **device driver** for the backing store (disk space used by virtual-memory algorithms) must be at a lower level than the **memory-management routines** because memory management requires the ability to use the backing store.

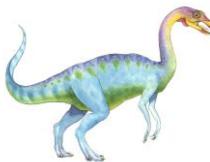




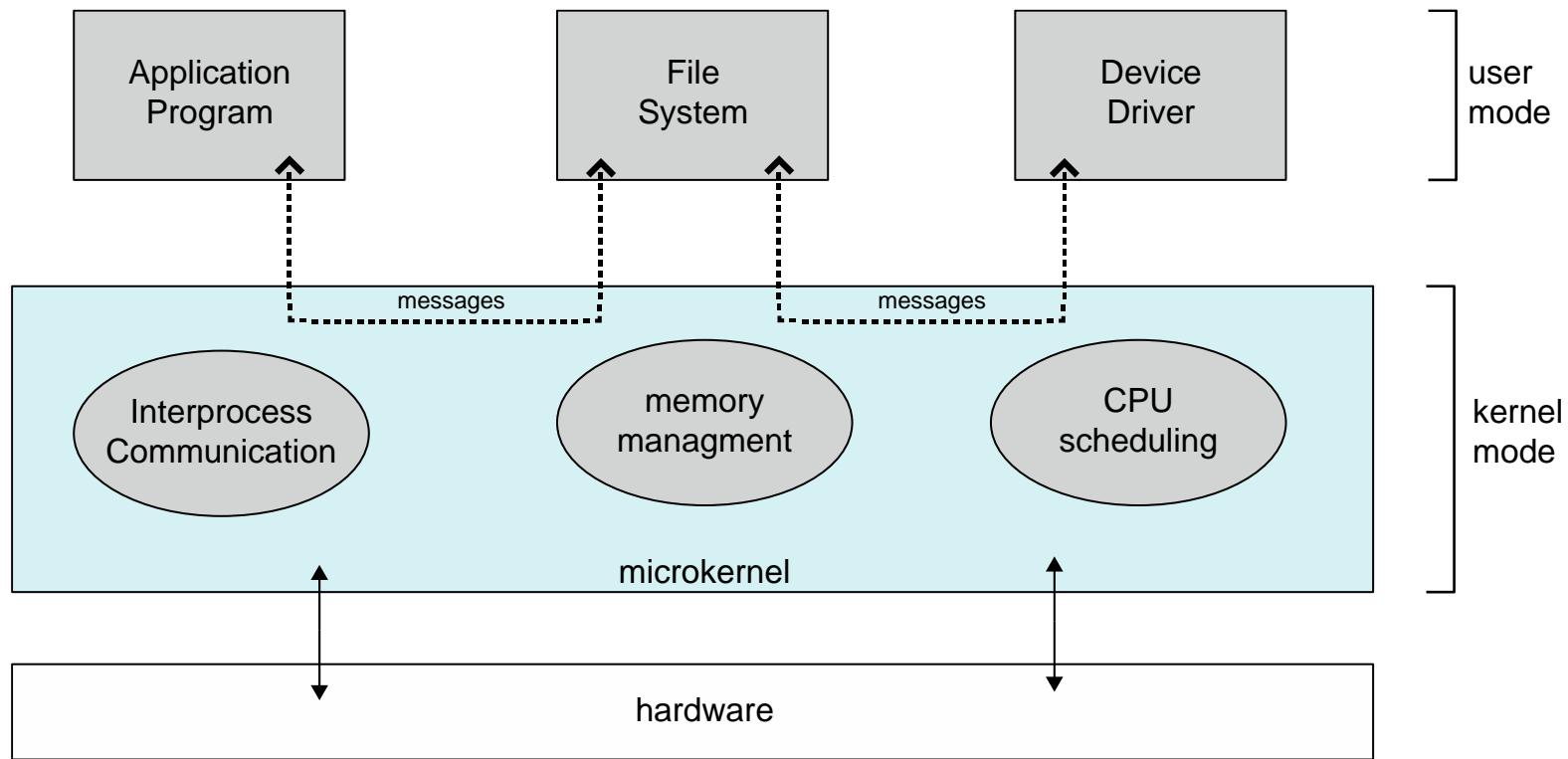
Microkernel System Structure

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





Microkernel System Structure



Architecture of a typical microkernel.





Monolithic vs Microkernel

MICROKERNEL KERNEL

A kernel type that provides mechanisms such as low-level address space management, thread management and interprocess communication to implement an operating system

OS services and kernel are separated

Slow

Failure in one component will not affect the other components

Easier to add new functionalities

Smaller in size

MONOLITHIC KERNEL

A type of kernel in operating systems where the entire operating system works in the kernel space

Kernel contains the OS services

Fast

Failure in one component will affect the entire system

Difficult to add new functionalities

Larger in size



System Boot

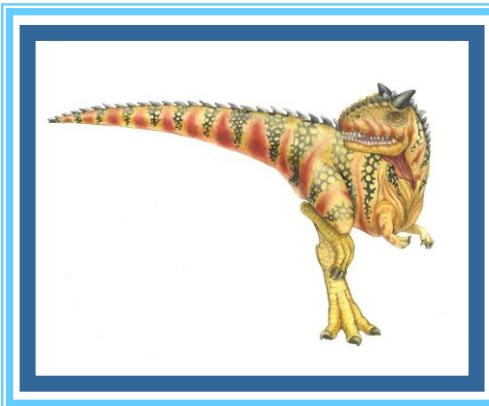
- When power is initialized on system, execution starts at a fixed memory location.
- The operating system must be made available to hardware so hardware can start it
- The procedure of starting a computer by loading the **kernel** is known as **booting** the system
- Small piece of code – **bootstrap loader**, stored in **ROM** or **EEPROM** locates the kernel, loads it into memory, and starts it
- Common bootstrap loader, **GRUB**, allows selection of kernel from multiple disks, versions, kernel options

Sometimes **it's a two-step** process where simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the **kernel**.

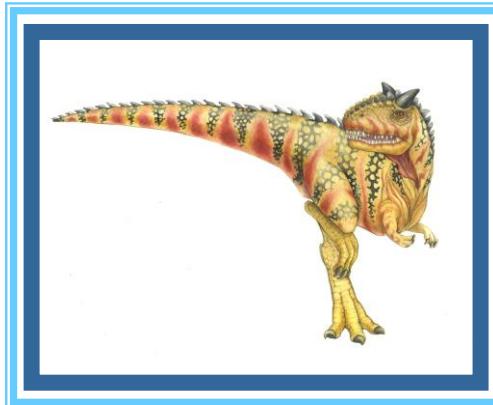
- Kernel loads and system is then **running**



End of Chapter 2



Chapter 3: Processes

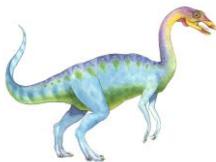




Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication





Objectives

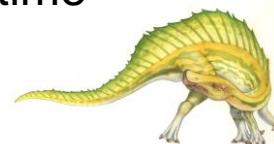
- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explore interprocess communication using shared memory and message passing

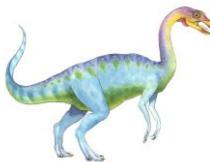




Process Concept

- An operating system executes a variety of programs:
 - Batch system – **jobs**
 - Time-shared systems – **user programs** or **tasks**
- Textbook uses the terms **job** and **process** almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
- Multiple parts
 - The program code, also called **text section**
 - Current activity including **program counter**, processor registers
 - **Stack** containing temporary data
 - ▶ Function parameters, return addresses, local variables
 - **Data section** containing global variables
 - **Heap** containing memory dynamically allocated during run time

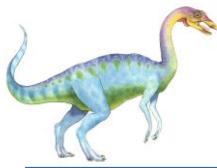




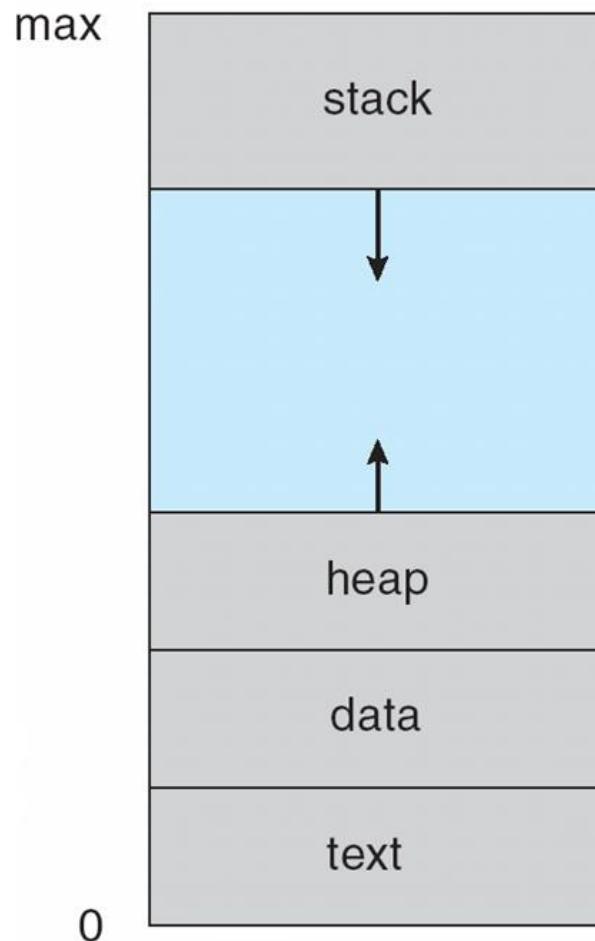
Process Concept (Cont.)

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





Process in Memory





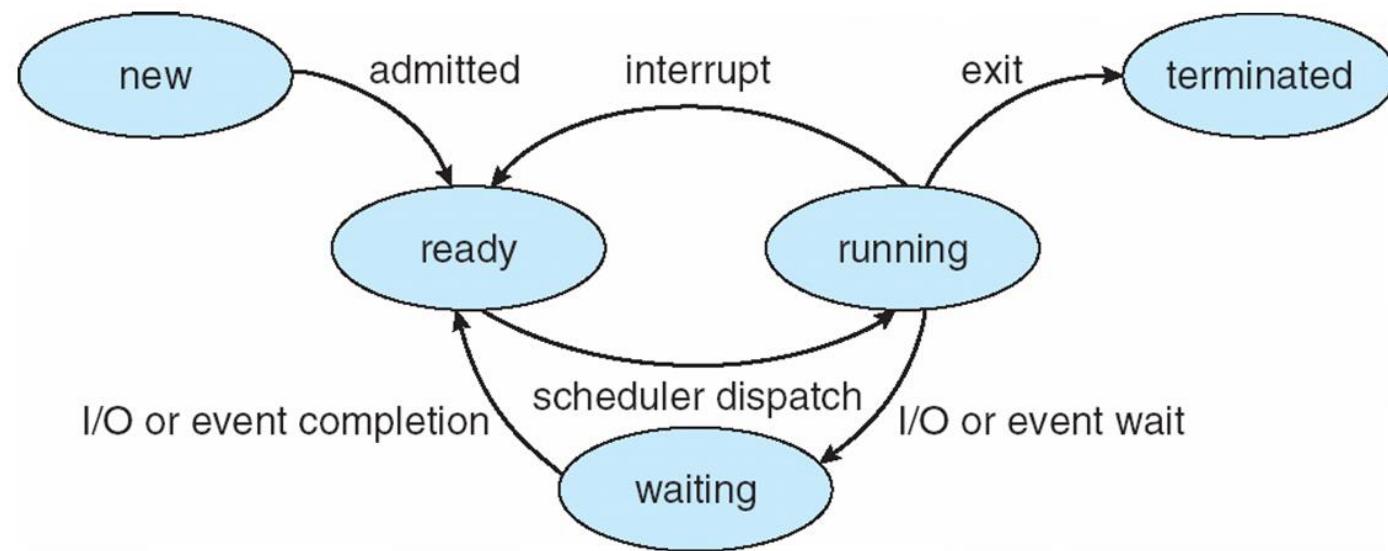
Process State

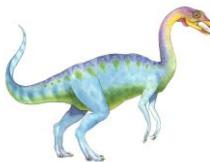
- As a process executes, it changes **state**
 - **new**: The process is being created
 - **running**: Instructions are being executed
 - **waiting**: The process is waiting for some event to occur
 - **ready**: The process is waiting to be assigned to a processor
 - **terminated**: The process has finished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





Diagram of Process State





Process Control Block (PCB)

PCB contains all information associated with each process

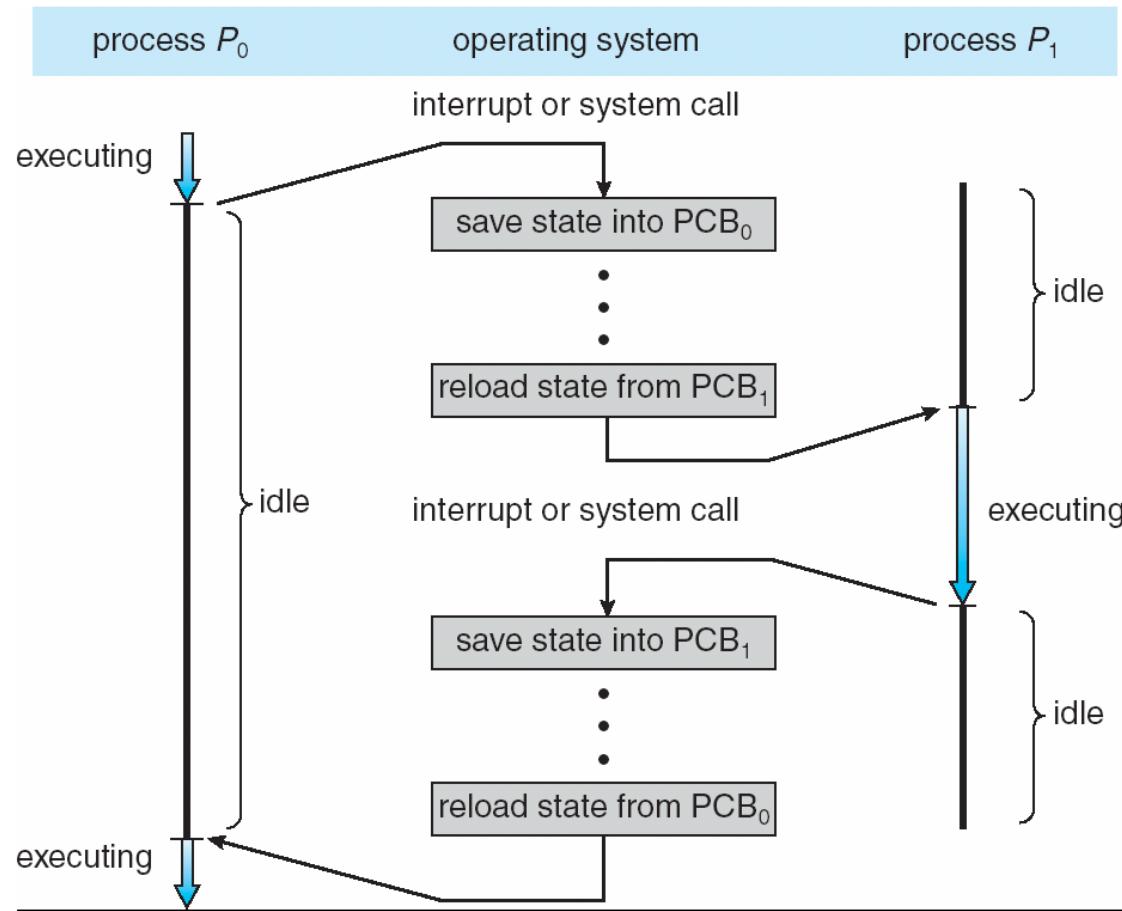
(also called **task control block**)

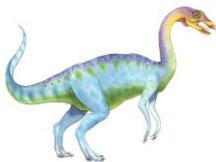
- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files





CPU Switch From Process to Process





Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
 - Multiple locations can execute at once
 - ▶ Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB





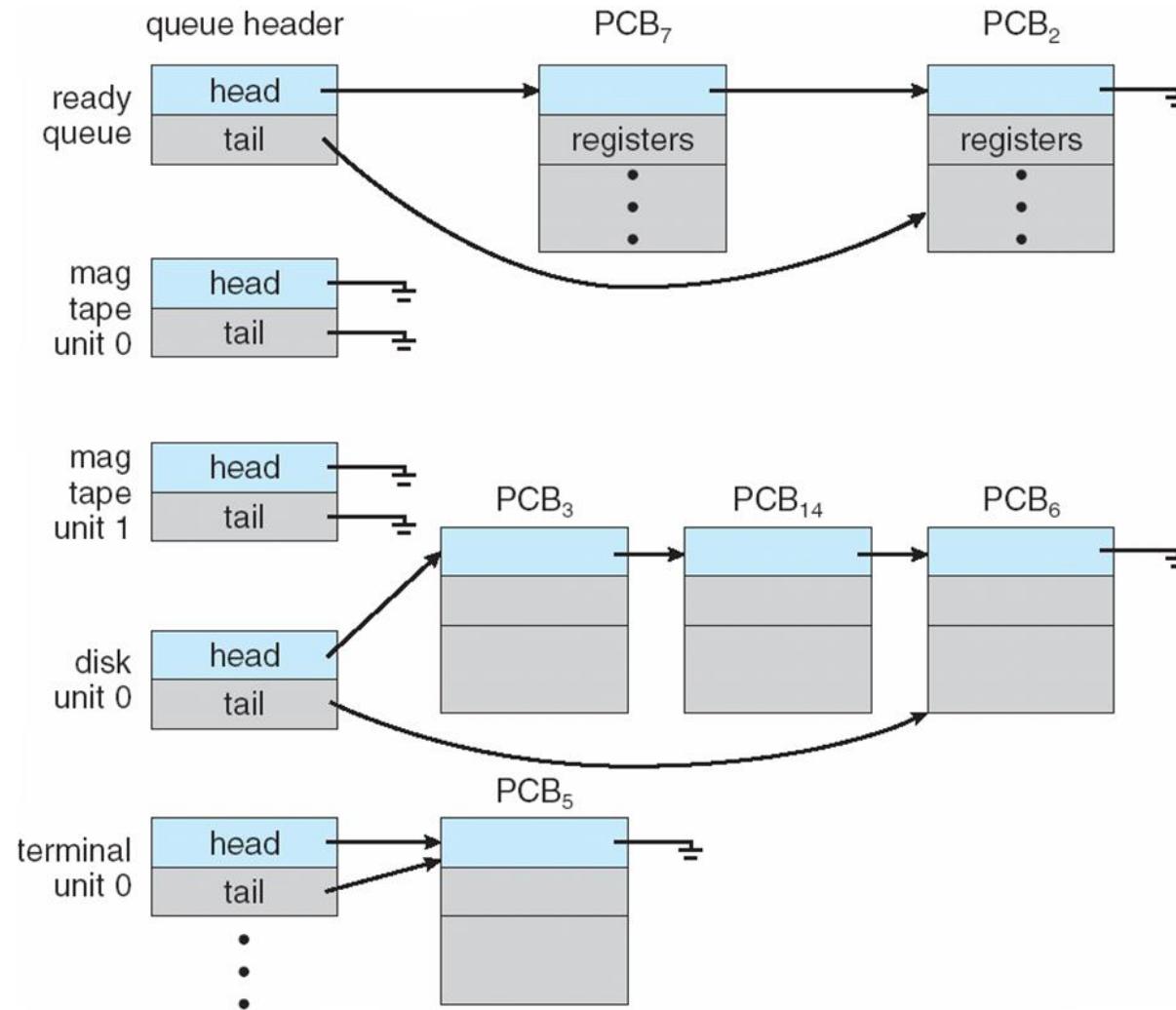
Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
 - **Job queue** – set of all processes in the system
 - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
 - **Device queues** – set of processes waiting for an I/O device
 - Processes migrate among the various queues





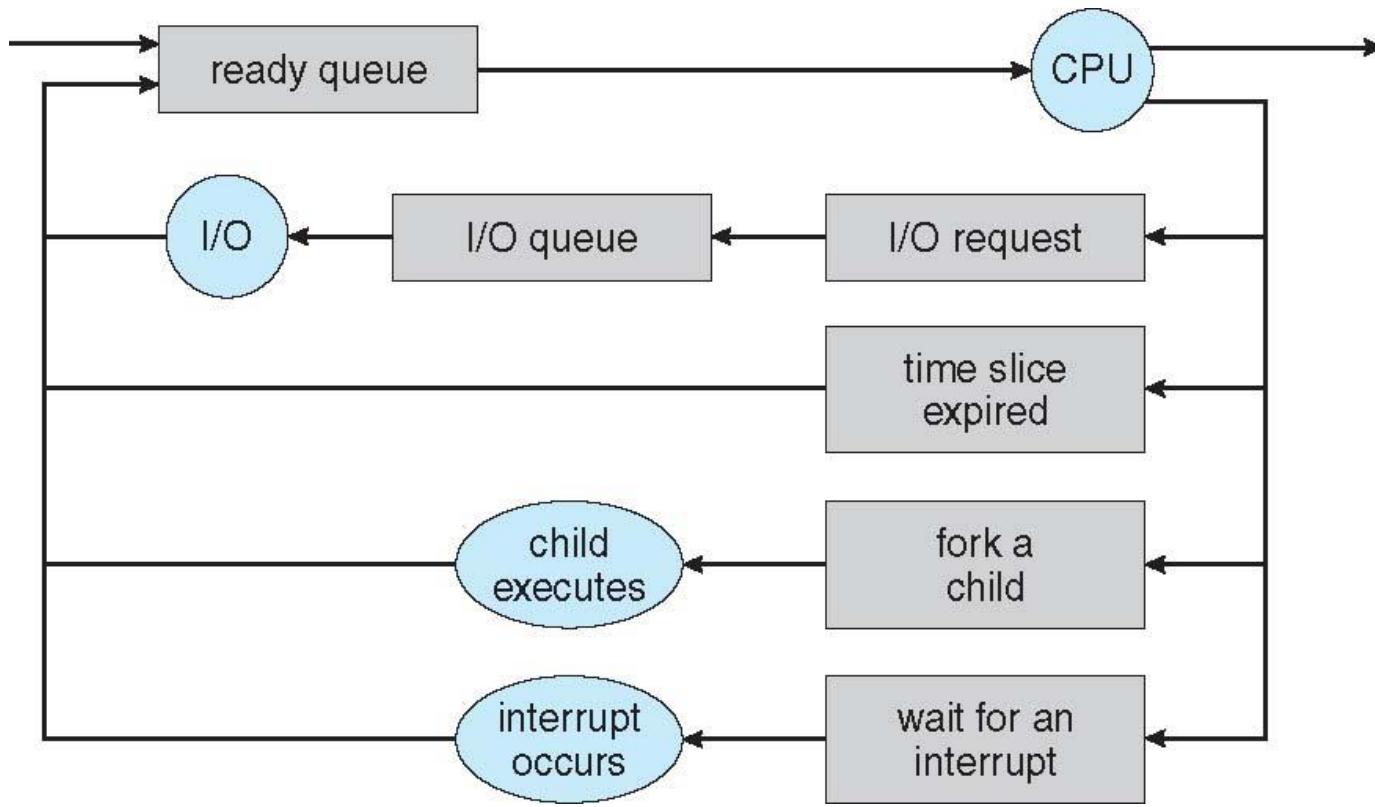
Ready Queue And Various I/O Device Queues





Representation of Process Scheduling

- **Queueing diagram** represents process scheduling, queues, resources, flows
- Rectangle represent queues, circle represents resources,





Schedulers

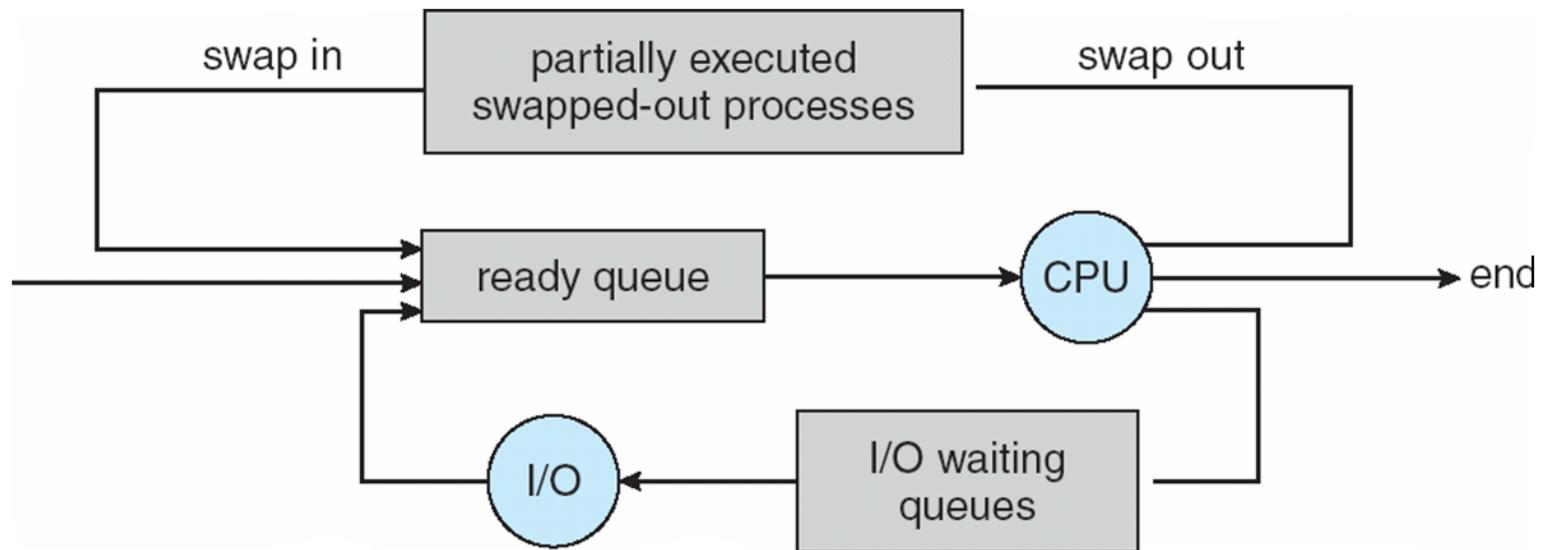
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
 - Sometimes the only scheduler in a system
 - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which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





Addition of Medium Term Scheduling

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





Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- user interface limits iOS provides for a
 - Single **foreground** process- controlled via user interface
 - Multiple **background** processes– in memory, running, but not on the display, and with limits
 - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
 - Background process uses a **service** to perform tasks
 - Service can keep running even if background process is suspended
 - Service has no user interface, small memory use





Context Switch

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





Operations on Processes

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

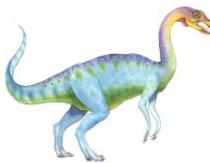




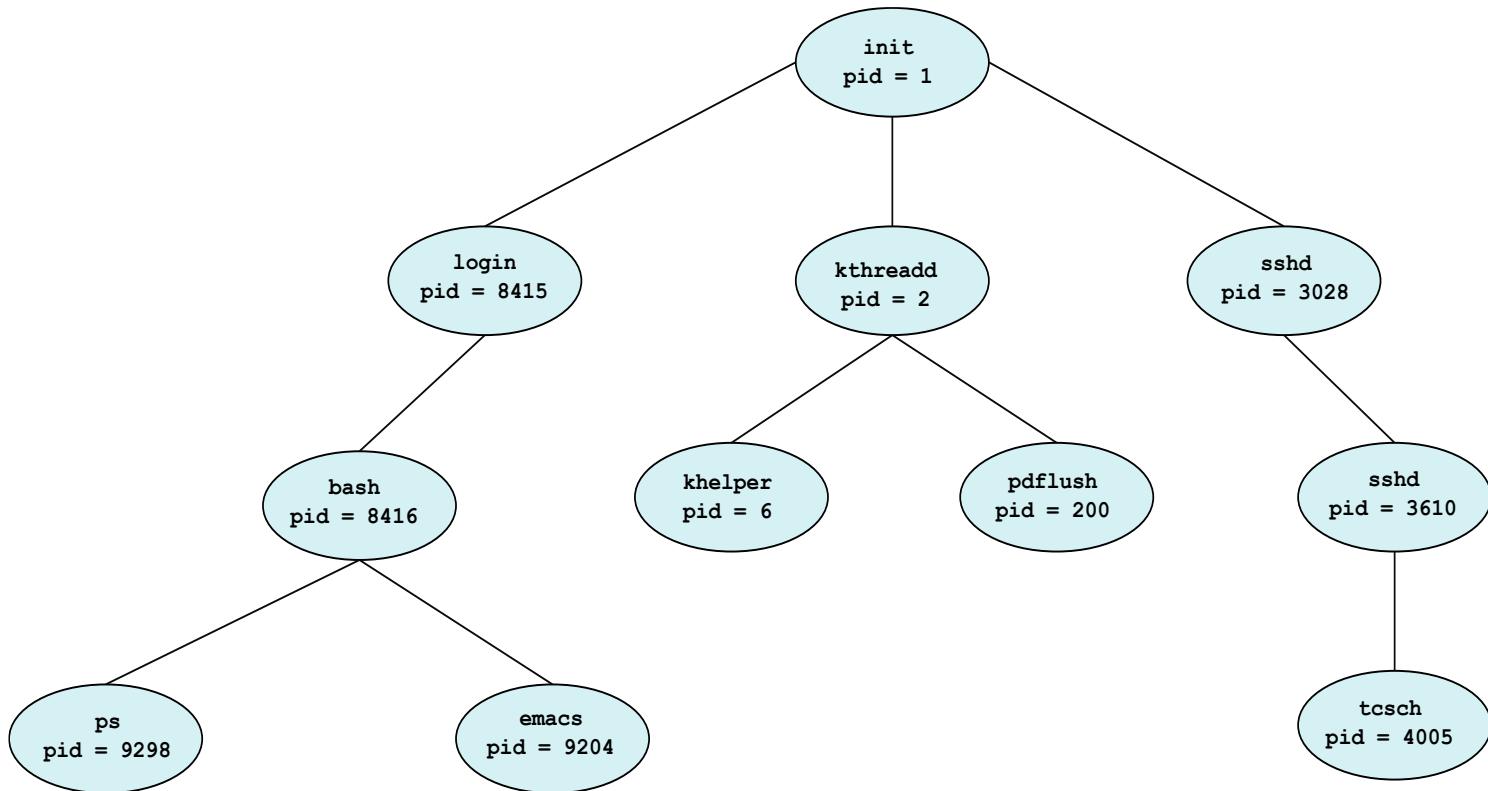
Process Creation

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





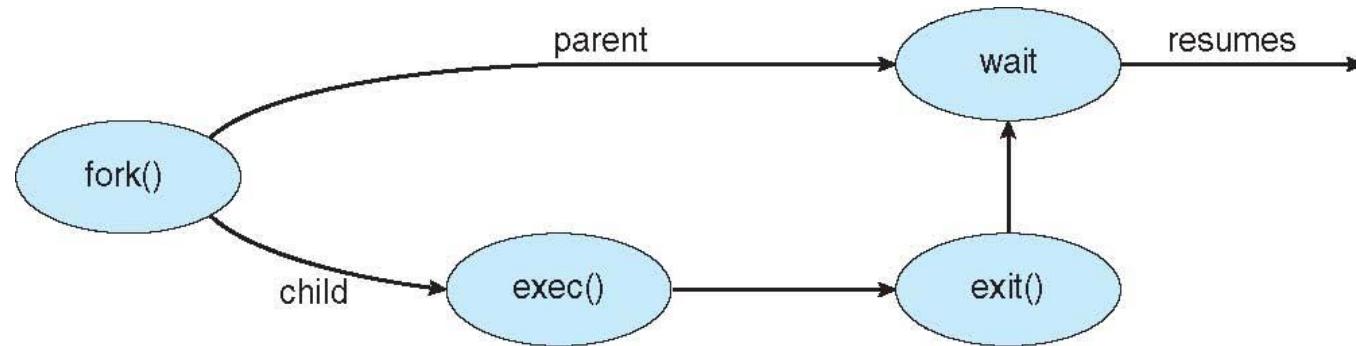
A Tree of Processes in Linux





Process Creation (Cont.)

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





Process Termination

- Process executes last statement and then asks the operating system to delete it using the `exit()` system call.
 - Returns status data from child to parent (via `wait()`)
 - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the `abort()` system call. Some reasons for doing so:
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required
 - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates





Process Termination

- Some operating systems do not allow child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
 - **cascading termination.** All children, grandchildren, etc. are terminated.
 - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```
- If no parent waiting (did not invoke `wait()`) process is a **zombie**
- If parent terminated without invoking `wait`, process is an **orphan**





Interprocess Communication

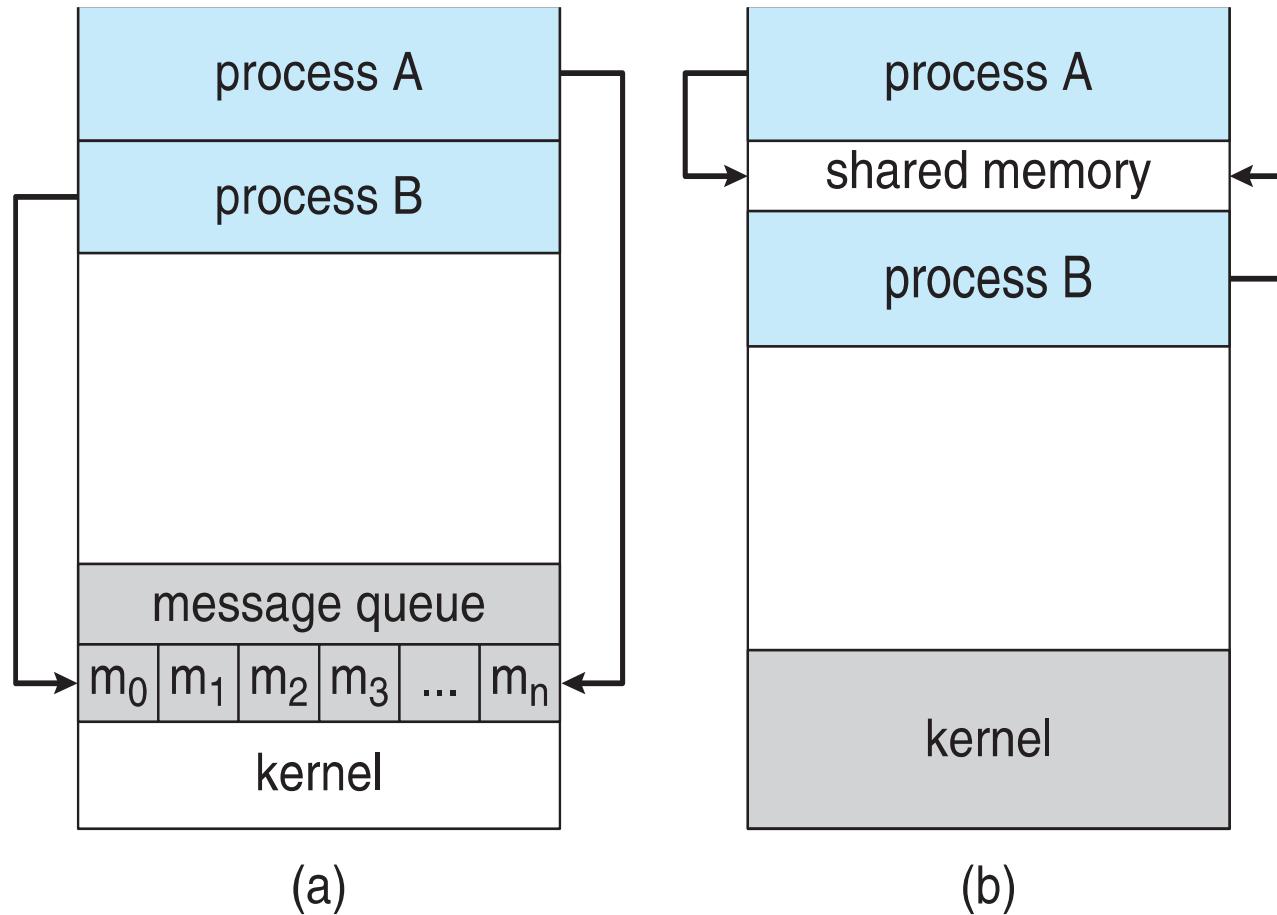
- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons and advantages for cooperating processes:
 - Information sharing
 - Computation speedup
 - Modularity
 - Convenience
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
 - **Shared memory**
 - **Message passing**

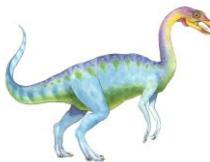




Communications Models

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





Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
 - **unbounded-buffer** places no practical limit on the size of the buffer
 - **bounded-buffer** assumes that there is a fixed buffer size

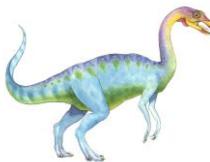




Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the user processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.





Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
 - `send(message)`
 - `receive(message)`
- The *message size* is either fixed or variable
- PROS and CONS of both





Message Passing (Cont.)

- If processes P and Q wish to communicate, they need to:
 - Establish a **communication link** between them
 - Exchange messages via send/receive
- Implementation issues:
 - How are links established?
 - Can a link be associated with more than two processes?
 - How many links can there be between every pair of communicating processes?
 - What is the capacity of a link?
 - Is the size of a message that the link can accommodate fixed or variable?
 - Is a link unidirectional or bi-directional?





Message Passing (Cont.)

- Implementation of communication link
 - Physical:
 - ▶ Shared memory
 - ▶ Hardware bus
 - ▶ Network
 - Logical:
 - ▶ Direct or indirect
 - ▶ Synchronous or asynchronous
 - ▶ Automatic or explicit buffering





Direct Communication

- Processes must name each other explicitly:
 - **send** (P , message) – send a message to process P
 - **receive**(Q , message) – receive a message from process Q
- Properties of communication link
 - Links are established automatically
 - A link is associated with exactly one pair of communicating processes
 - Between each pair there exists exactly one link
 - The link may be unidirectional, but is usually bi-directional





Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
 - Each mailbox has a unique id
 - Processes can communicate only if they share a mailbox
- Properties of communication link
 - Link established only if processes share a common mailbox
 - A link may be associated with many processes
 - Each pair of processes may share several communication links
 - Link may be unidirectional or bi-directional





Indirect Communication

- Operations

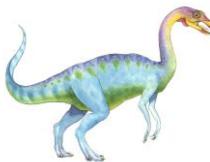
- create a new mailbox (port)
 - send and receive messages through mailbox
 - destroy a mailbox

- Primitives are defined as:

`send(A, message)` – send a message to mailbox A

`receive(A, message)` – receive a message from mailbox A

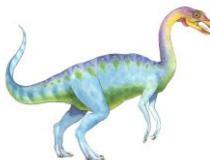




Indirect Communication

- Mailbox sharing
 - P_1 , P_2 , and P_3 share mailbox A
 - P_1 , sends; P_2 and P_3 receive
 - Who gets the message?
- Solutions
 - Allow a link to be associated with at most two processes
 - Allow only one process at a time to execute a receive operation
 - Allow the system to select arbitrarily the receiver.
Sender is notified who the receiver was.





Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
 - **Blocking send** -- the sender is blocked until the message is received
 - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
 - **Non-blocking send** -- the sender sends the message and continue
 - **Non-blocking receive** -- the receiver receives:
 - A valid message, or
 - Null message
- Different combinations possible
 - If both send and receive are blocking, we have a **rendezvous**





Synchronization (Cont.)

- Producer-consumer becomes trivial

```
message next_produced;
while (true) {
    /* produce an item in next produced */
    send(next_produced);
}

message next_consumed;
while (true) {
    receive(next_consumed);

    /* consume the item in next consumed */
}
```



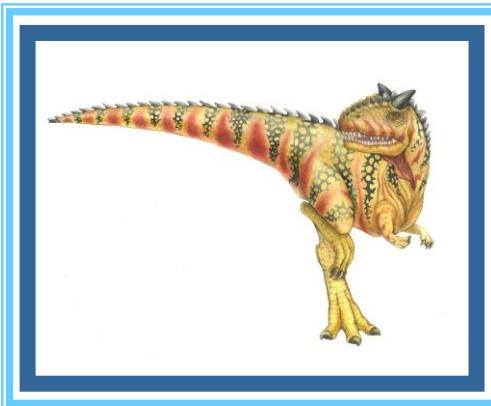


Buffering

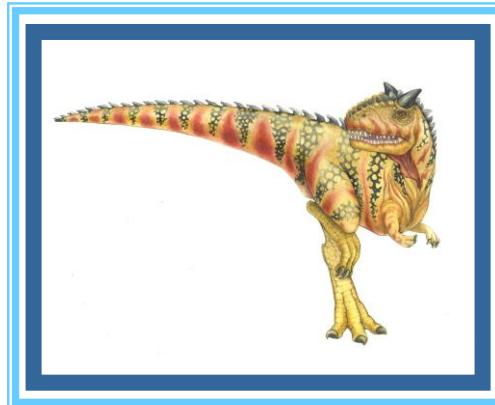
- Queue of messages attached to the link.
- Queues are implemented in one of three ways
 1. Zero capacity – no messages are queued on a link.
Sender must wait for receiver
 2. Bounded capacity – finite length of n messages
Sender must wait if link full
 3. Unbounded capacity – infinite length
Sender never waits



End of Chapter 3



Chapter 4: Threads

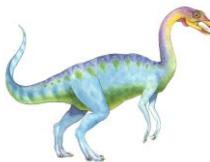




Chapter 4: Threads

- Overview
- Multithreading Models
- Thread Libraries
- Threading Issues





Objectives

- To introduce the notion of a thread—a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Windows, and Java thread libraries
- To examine issues related to multithreaded programming





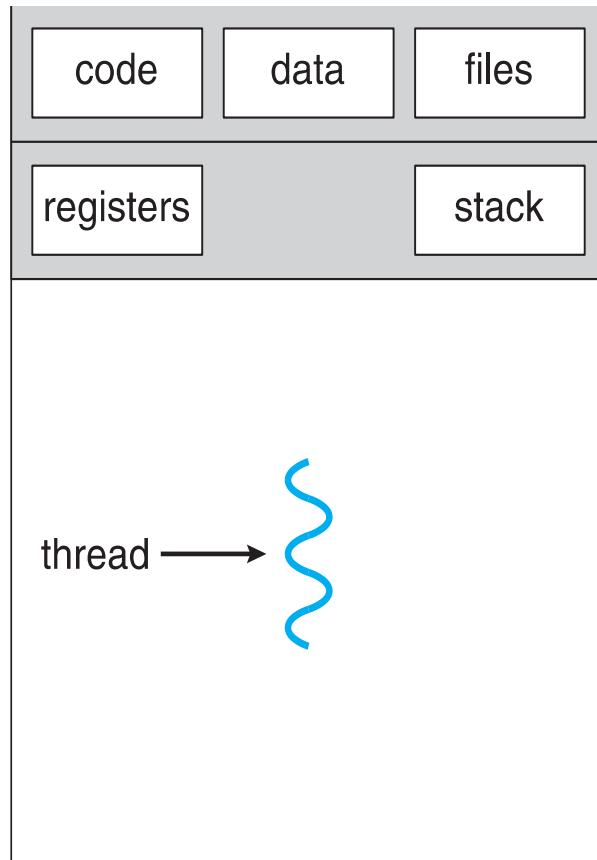
Motivation

- Most modern applications are **multithreaded**
- Threads run within application
- Multiple tasks within the application can be implemented by separate threads
 - Update display
 - Fetch data
 - Spell checking
 - Answer a network request
- Process creation is **heavy-weight** while thread creation is **light-weight**
- Can simplify code, increase efficiency
- Kernels are generally multithreaded with each thread managing specific task (managing device, memory, interrupt etc)

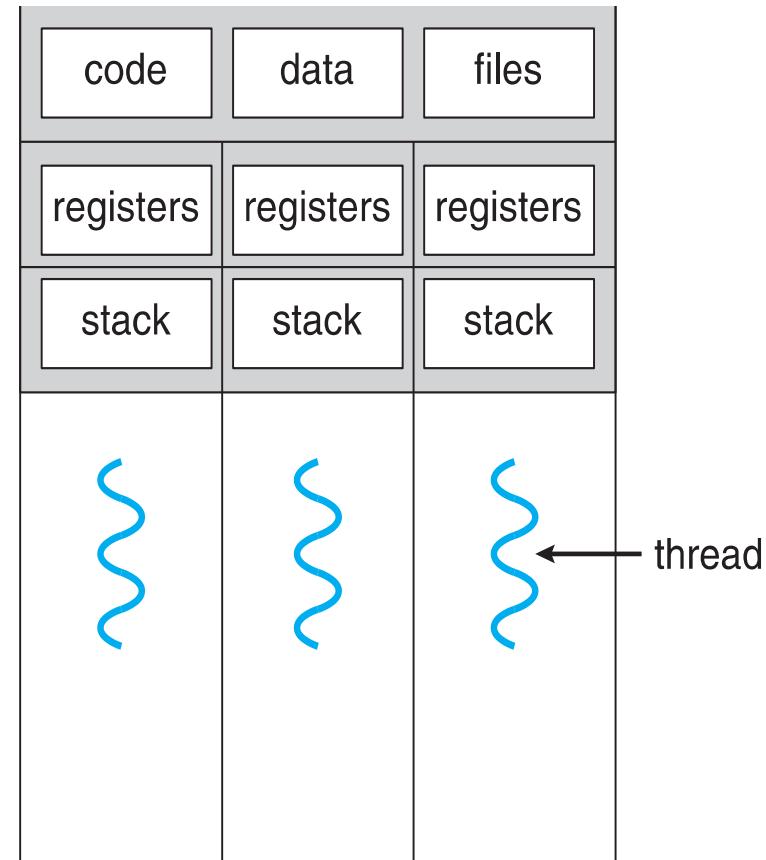




Single and Multithreaded Processes



single-threaded process

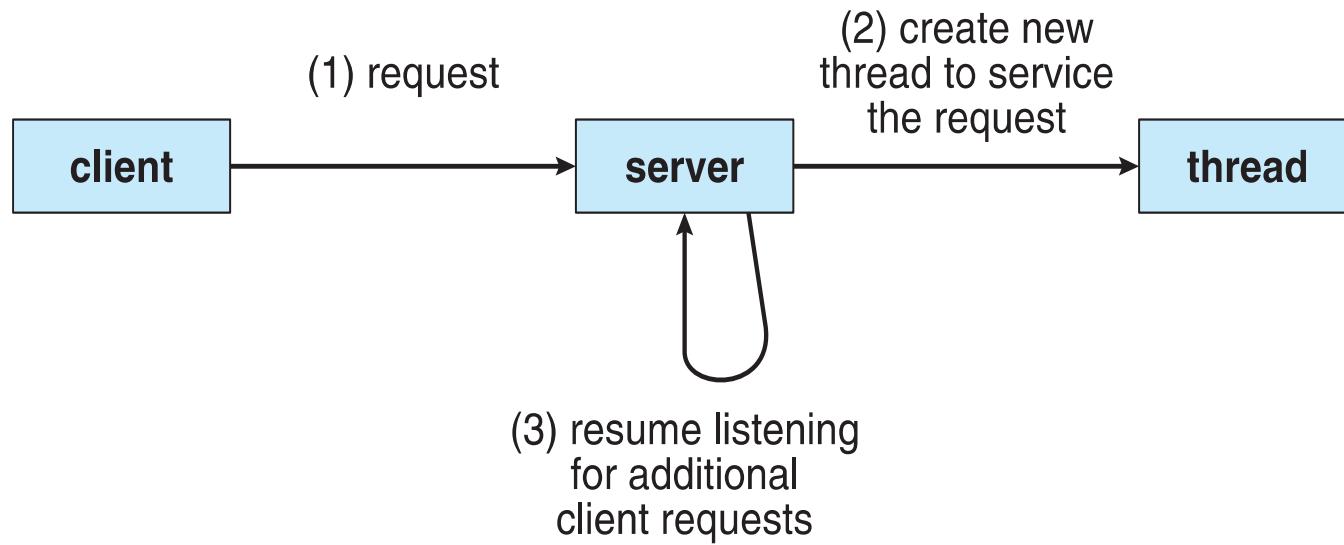


multithreaded process





Multithreaded Server Architecture





Benefits: Multithreaded

- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for multi-threaded interactive applications. Ex: user interfaces
- **Resource Sharing** – threads share resources of the process it belongs to by default, easier than shared memory or message passing.
- **Economy** – Allocating resources for process creation is costly. Thread creation and switching is more economical.
- **Scalability** – Multithreaded process can take advantage of multiprocessor architectures, where threads can run in parallel on different processing cores.





User Threads and Kernel Threads

Threads can be supported at two levels

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





Multithreading Models

Relation(mapping) between user and kernel threads:

- Many-to-One
- One-to-One
- Many-to-Many
- Use of these models:
 - User level thread cannot achieve true parallelism on their own
 - Kernel level thread support parallelism and allows threads to run simultaneously on multiple CPU cores.

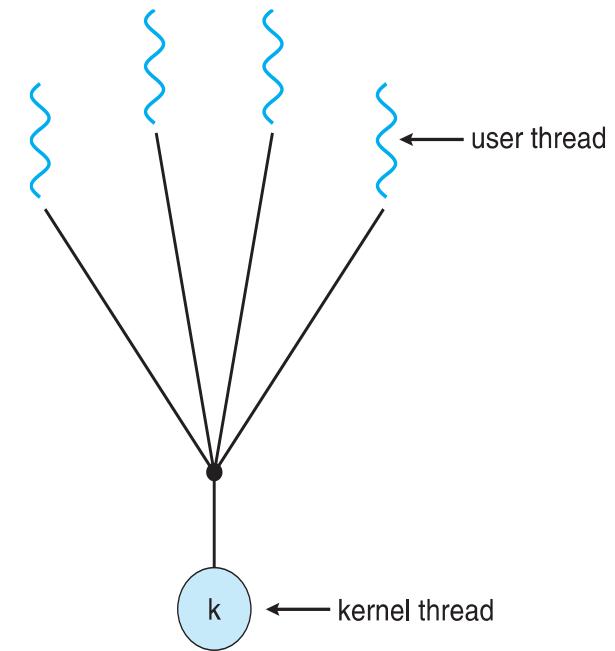
Many modern systems use threading models

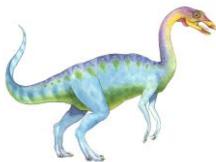




Many-to-One

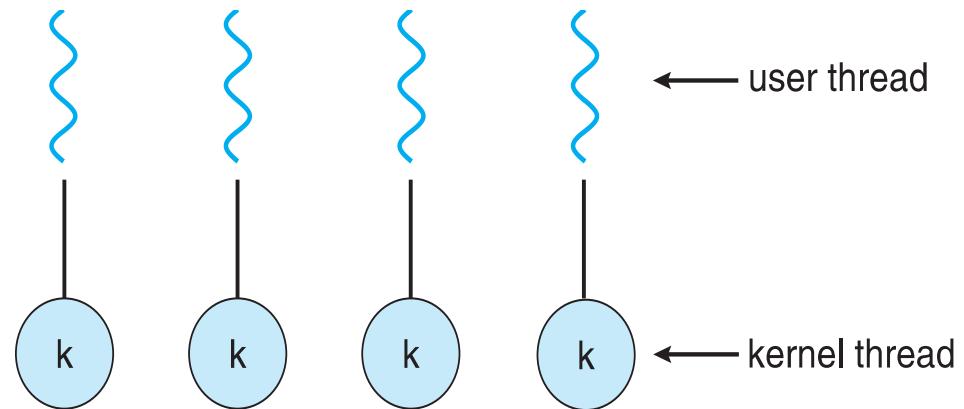
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
 - **Solaris Green Threads**
 - **GNU Portable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





One-to-One

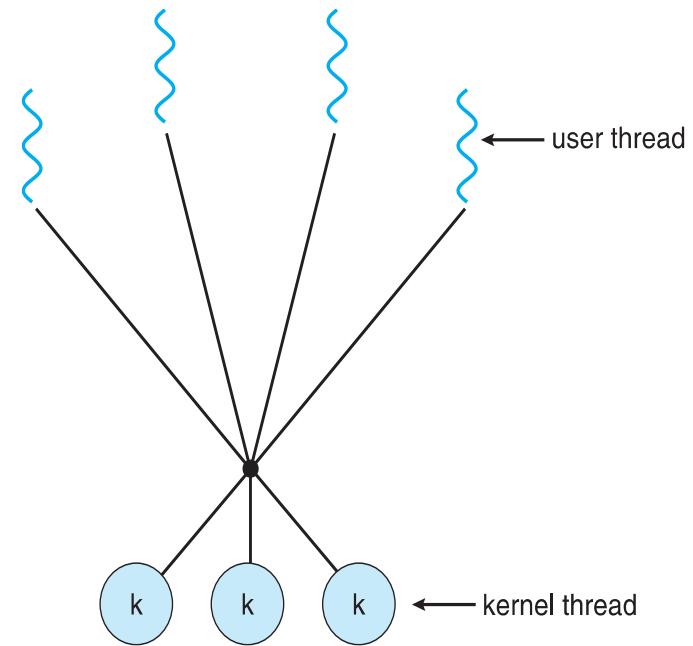
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
 - Windows
 - Linux
 - Solaris 9 and later





Many-to-Many Model

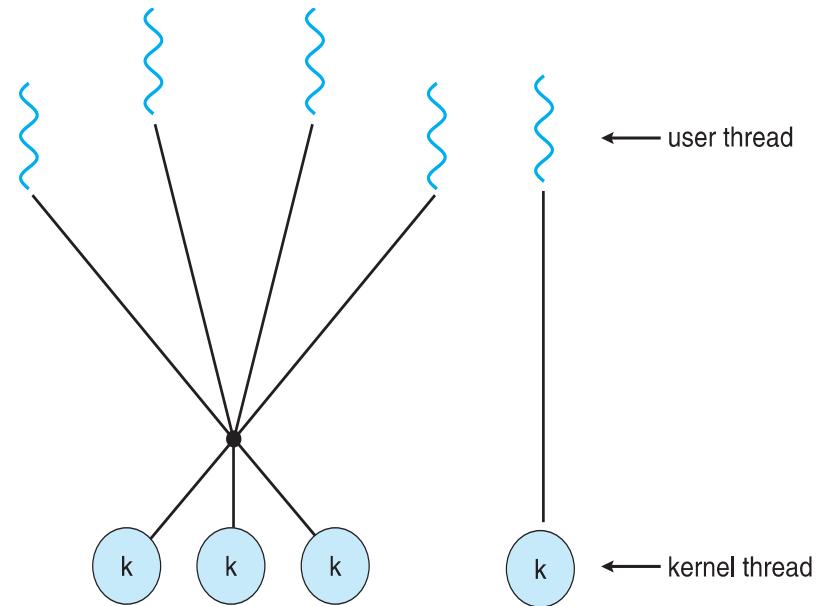
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package

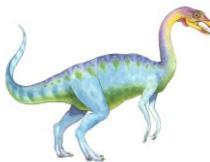




Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
 - IRIX
 - HP-UX
 - Tru64 UNIX
 - Solaris 8 and earlier





Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
 - Library entirely in user space
 - Kernel-level library supported by the OS
- Three main thread libraries are in use today: POSIX Pthreads, Windows, and Java.
- POSIX (Portable Operating System Interface) is a **set of standard operating system interfaces based on the Unix operating system**.
- The Windows thread library is a kernel-level library available on Windows systems.
- The Java thread API allows threads to be created and managed directly in Java programs.





PThreads

- May be provided either as user-level or kernel-level
- It's a POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- The Portable Operating System Interface is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.
- *This is a **Specification**, not **implementation**. OS can implement their way*
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in **UNIX operating systems** (Solaris, Linux, Mac OS X)
- **Windows** does not support this





Pthreads Example

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
        return -1;
    }
}
```





Pthreads Example (Cont.)

```
/* get the default attributes */
pthread_attr_init(&attr);
/* create the thread */
pthread_create(&tid,&attr,runner,argv[1]);
/* wait for the thread to exit */
pthread_join(tid,NULL);

printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```





Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

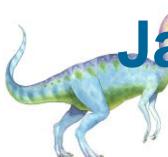




Java program for the summation of a non-negative integers

```
public class Summation implements Runnable {  
    private int upper;  
    private int sum;  
    public Summation(int upper) {  
        this.upper = upper;  
    }  
    public int getSum() {  
        return sum;  
    }  
    public void run() {  
        sum = 0;  
        for (int i = 0; i <= upper; i++) {  
            sum += i;  
        }  
    }  
}
```





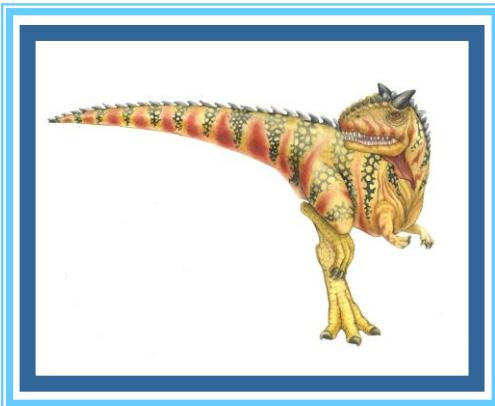
Java program for the summation of a non-negative integers

```
public static void main(String[] args) {  
    if (args.length > 0)  
    {  
        int upper = Integer.parseInt(args[0]);  
        if (upper < 0) {  
            System.err.println(args[0] + " must  
be >= 0.");  
        } else {  
            Summation task = new  
            Summation(upper);  
Thread thrd = new Thread(task);  
thrd.start();  
    }  
}
```

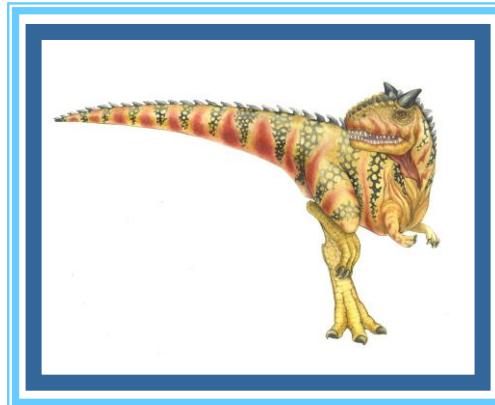
```
try {  
    thrd.join();  
    System.out.println("The sum of " +  
upper + " is " + task.getSum());  
}  
catch (InterruptedException ie) {  
    System.err.println("Thread  
interrupted.");  
}  
}  
else {  
    System.err.println("Usage:  
Summation <integer value>");  
}  
}  
}
```

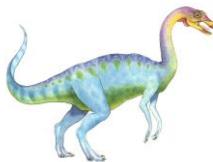


End of Chapter 4



Chapter 5: Process Scheduling

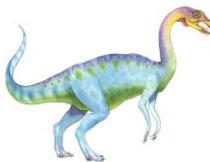




Chapter 6: Process Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling





Objectives

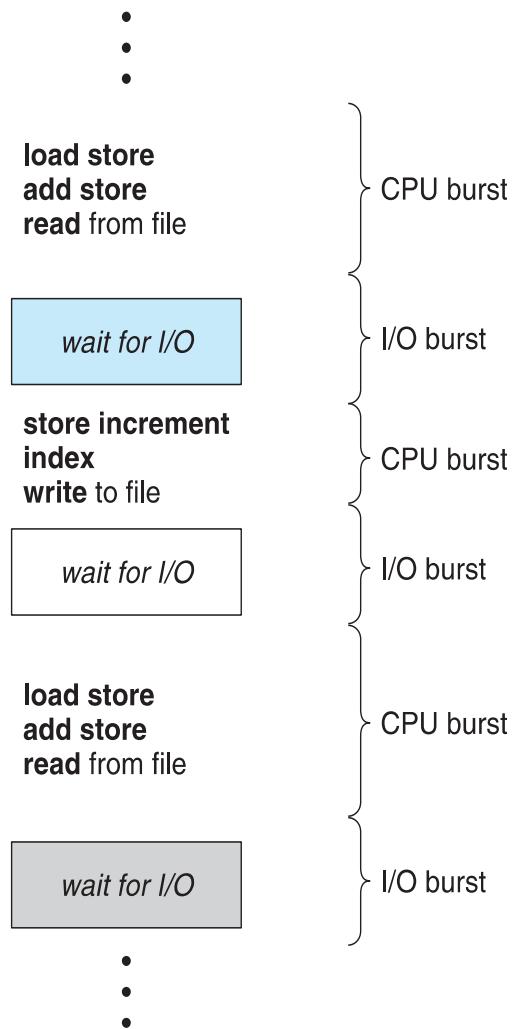
- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To understand the scheduling algorithms of operating systems





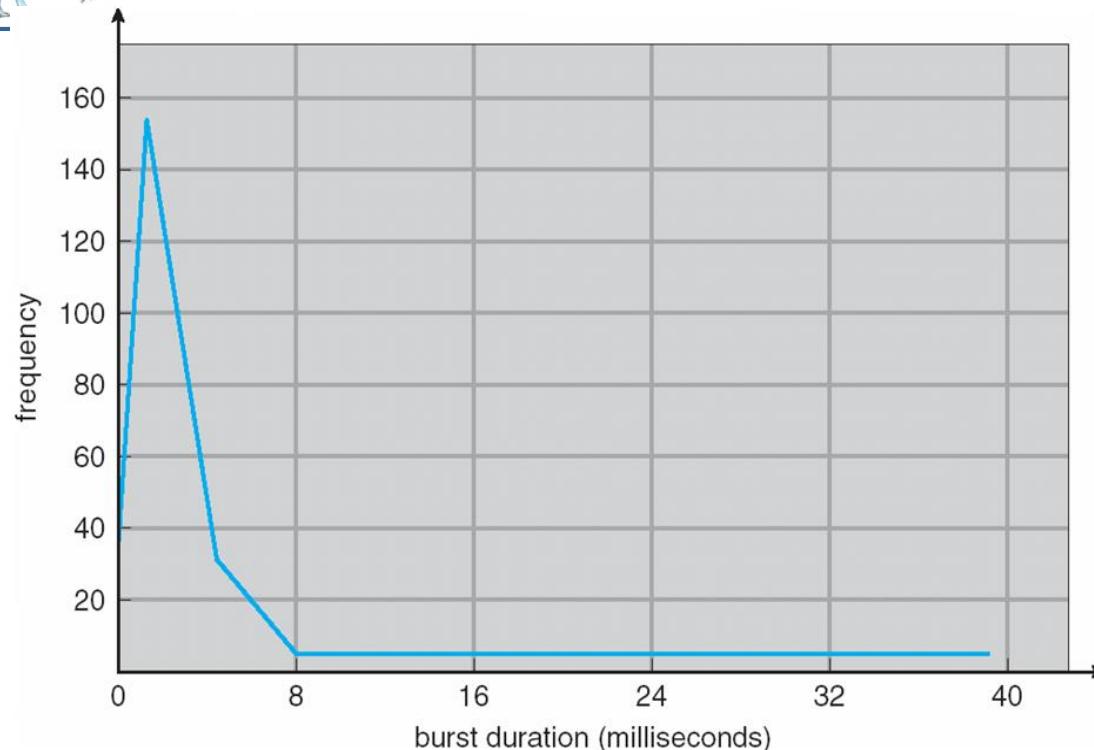
Basic Concepts

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





Histogram of CPU-burst Times



- The curve is generally characterized as **exponential or hyperexponential**, with a large number of short CPU bursts and a small number of long CPU bursts
- An I/O-bound program typically has many short CPU bursts.
- A CPU-bound program might have a few long CPU bursts.
algorithm.
- This distribution is important for selecting CPU-scheduling alg

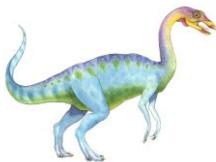




CPU Scheduler

- **Short-term scheduler** selects from among the processes in **ready queue**, and allocates the **CPU** to one of them
 - Ready Queue may be ordered in various ways(FIFO, priority etc)
- CPU scheduling decisions may take place when a process:
 1. Switches from running to waiting state
 2. Switches from running to ready state
 3. Switches from waiting to ready
 4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive or cooperative**
- All other scheduling is **preemptive**
 - Consider access to shared data (may result in inconsistent data)
 - Consider preemption while in kernel mode
 - Consider interrupts occurring during crucial OS activities

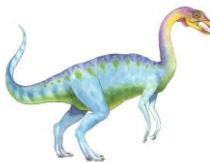




Dispatcher

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





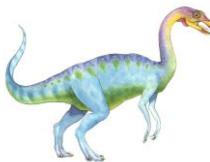
Scheduling Criteria

Criteria for comparing CPU-scheduling algorithm

- **CPU utilization** – Must keep the CPU as busy as possible(40% to 90% utilization)
- **Throughput** – # of processes that complete their execution per time unit. For long process it may be 1 process/hour and for short process it may be 10 process/sec
- **Turnaround time** – The interval from the time of submission of a process to the time of completion. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

It is desirable to **maximize CPU utilization and throughput** and to **minimize turnaround time, waiting time, and response time**.

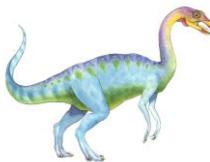




Scheduling Algorithm Optimization Criteria

- Max CPU utilization
 - Max throughput
 - Min turnaround time
 - Min waiting time
 - Min response time
-
- $TAT = CT - AT$
 - $WT = TAT - BT$
-
- **Completion Time (CT), . Arrival Time (AT),**
 - **Waiting Time (WT) , Burst Time (BT):**

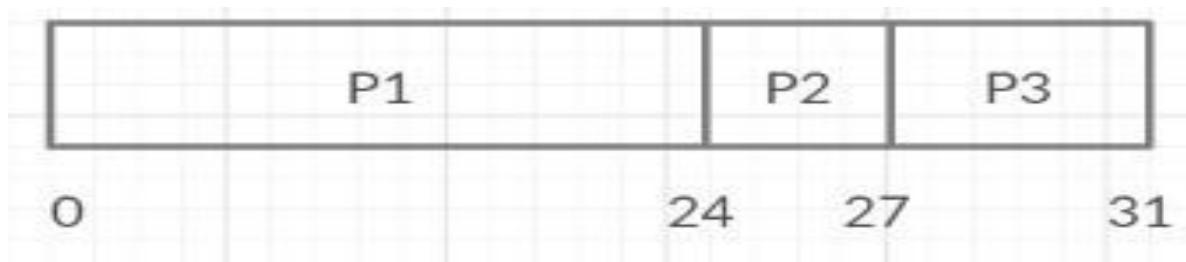




Example:

Process	Burst Time (in sec.)
P1	24
P2	3
P3	4

- Gantt Chart:



- Avg. TAT = $(24 + 27 + 31) / 3 = 27.33$ sec
- Avg. WT = $(0 + 24 + 27) / 3 = 17.0$ sec





First- Come, First-Served (FCFS) Scheduling

- Simplest CPU-scheduling algorithm
- With this scheme, the process that requests the CPU first is allocated the CPU first.
- The implementation of the FCFS policy is easily managed with a FIFO queue.
- On the negative side, the average waiting time under the FCFS policy is often quite long.
- Consider the example with set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:





First-Come, First-Served (FCFS) Scheduling

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

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



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





FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2, P_3, P_1$$

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$, $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- **Convoy effect** - short process wait for one long process to get the CPU
- This result in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.
 - Consider one long CPU-bound and many I/O-bound processes





Shortest-Job-First (SJF) Scheduling

- Associates with each process the length of its next CPU burst
 - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
 - The difficulty is knowing the length of the next CPU request
 - SJF is used frequently **in long term scheduler**, with user specifying the process time during submission.
 - Difficult to implement at **short-term scheduler** level.
 - For short-term scheduler next CPU burst can be calculated or predicted using the current CPU burst.

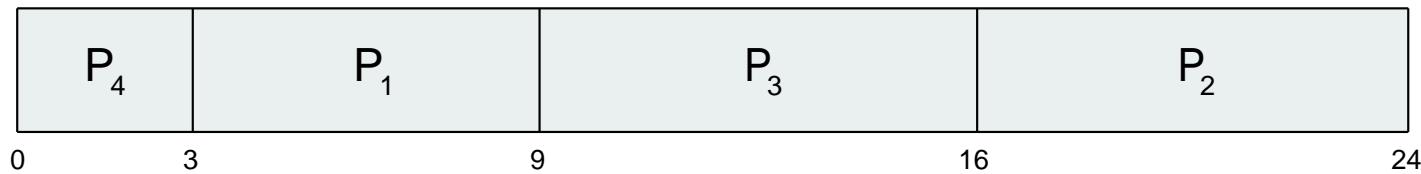




Example of SJF

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

- SJF scheduling chart



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





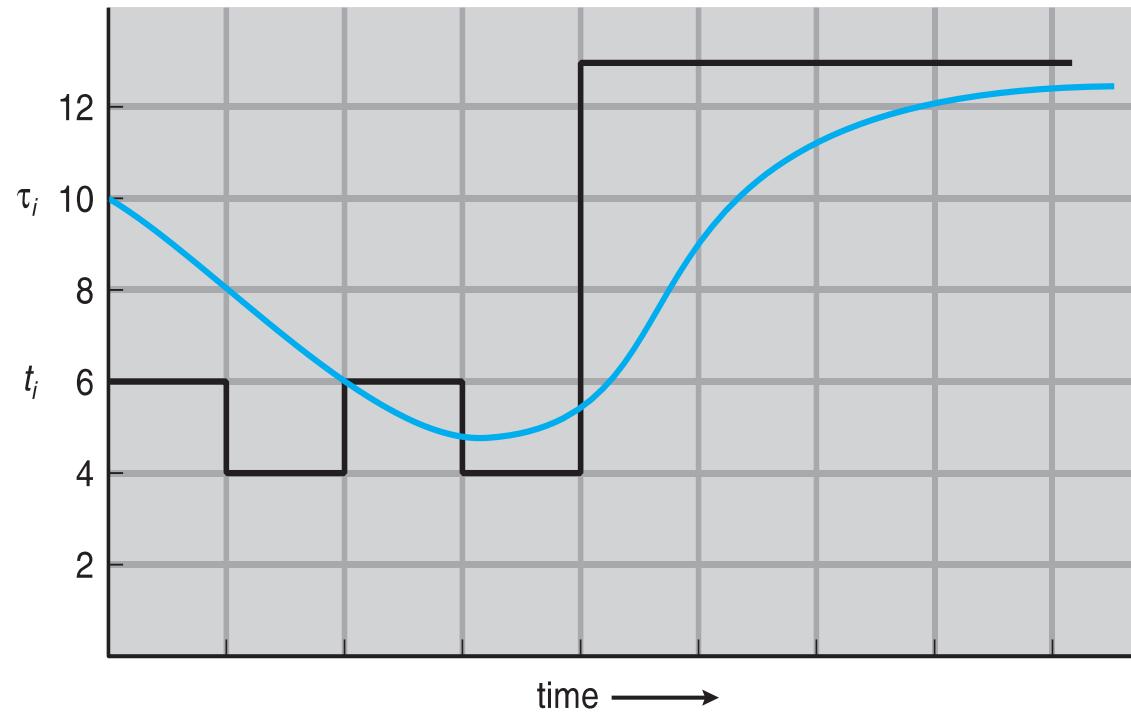
Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
 - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using **exponential averaging**
 1. t_n = actual length of n^{th} CPU burst
 2. τ_{n+1} = predicted value for the next CPU burst
 3. $\alpha, 0 \leq \alpha \leq 1$
 4. Define: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$.
 - α controls the relative weight of recent and past history in our prediction
 - Commonly, α set to $\frac{1}{2}$
 - If $\alpha = 1$, then $\tau_{n+1} = t_n$, and only the most recent CPU burst matters.
 - Preemptive version called **shortest-remaining-time-first**





Prediction of the Length of the Next CPU Burst



CPU burst (t_i)	6	4	6	4	13	13	13	...	
"guess" (τ_i)	10	8	6	6	5	9	11	12	...





Examples of Exponential Averaging

- $\alpha = 0$
 - $\tau_{n+1} = \tau_n$
 - Recent history does not count
- $\alpha = 1$
 - $\tau_{n+1} = \alpha t_n$
 - Only the actual last CPU burst counts
- If we expand the formula, we get:
$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \dots + (1 - \alpha)^j \alpha t_{n-j} + \dots + (1 - \alpha)^{n+1} \tau_0$$
- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor



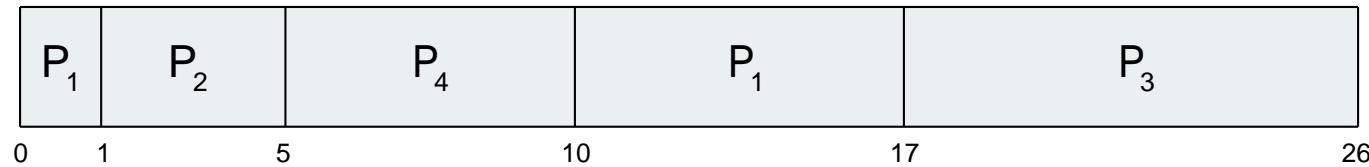


Example of Shortest-remaining-time-first

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

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

- Preemptive SJF Gantt Chart**



- Average waiting time = $[(10-1)+(1-1)+(17-2)+5-3]/4 = 26/4 = 6.5$ msec
- What is the average waiting time of NP-SJF ?





Priority Scheduling

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

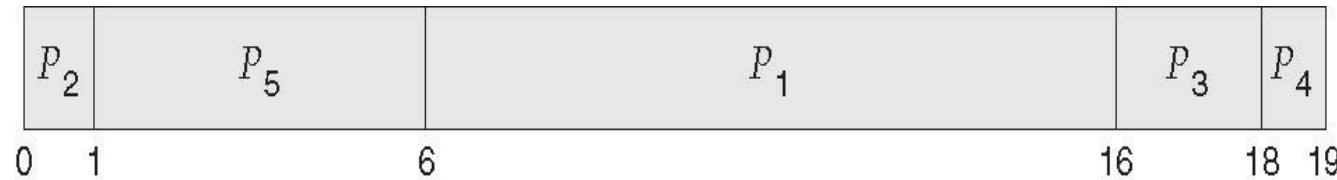




Example of Priority Scheduling

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

- Priority scheduling Gantt Chart



- Average waiting time = 8.2 msec





Round Robin (RR)

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

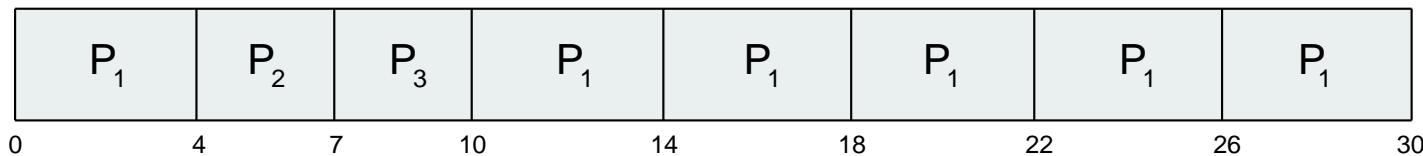




Example of RR with Time Quantum = 4

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

- The Gantt chart is:

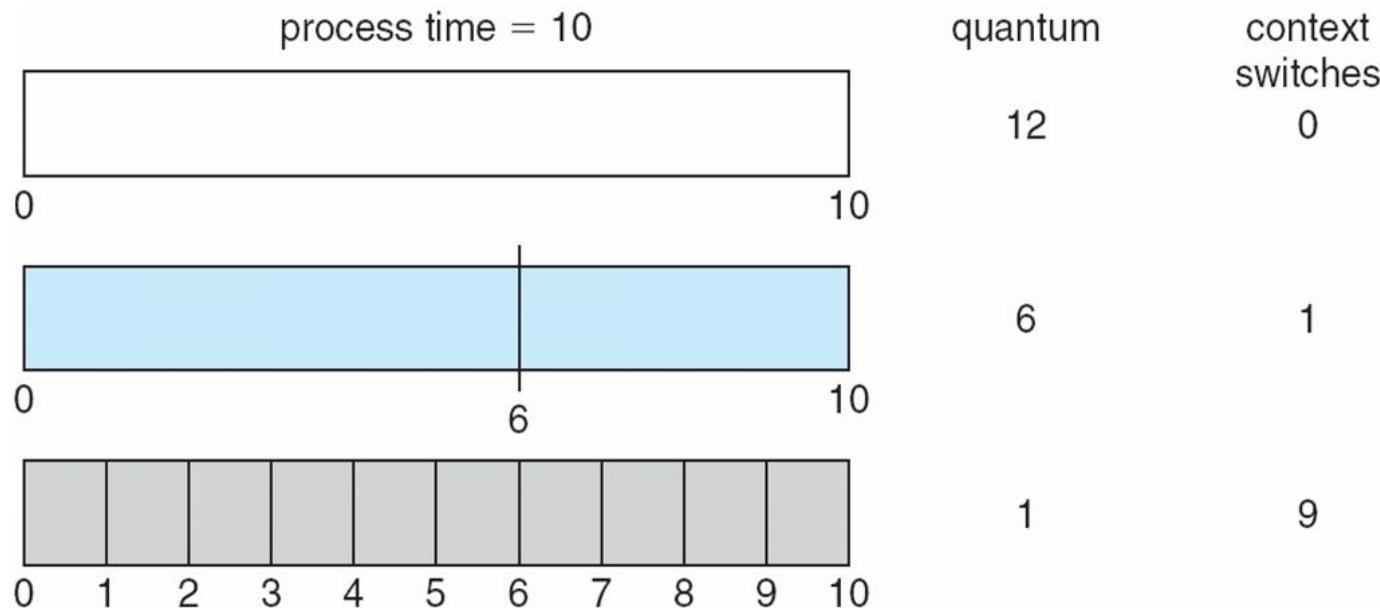


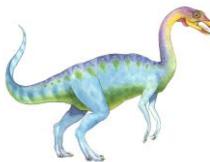
- Process waits for, $P1=(10-4)=6$, $P2=4$, $P3=7$, average waiting time is $17/3=5.66$ ms
- Typically, higher average turnaround than SJF, but better **response**
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec



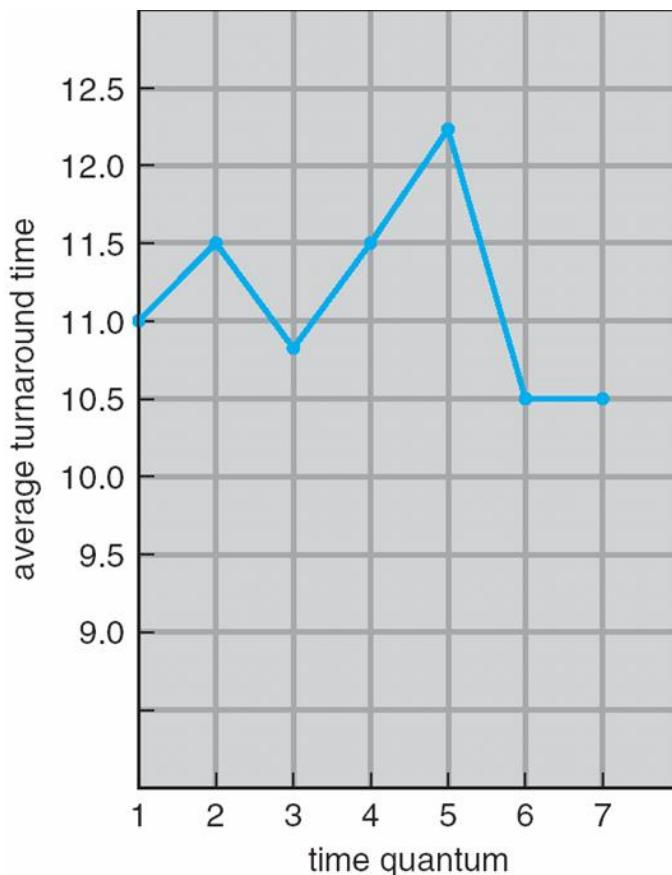


Time Quantum and Context Switch Time





Turnaround Time Varies With The Time Quantum

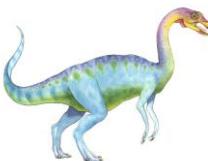


process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts
should be shorter than q

- Turn around time also depends upon the time quantum
- But, average turn around time does not necessarily improve with increase in time quantum, as shown in fig.
- Generally, average TAT can improve if most processes finish their next CPU burst in a single quantum





Multilevel Queue

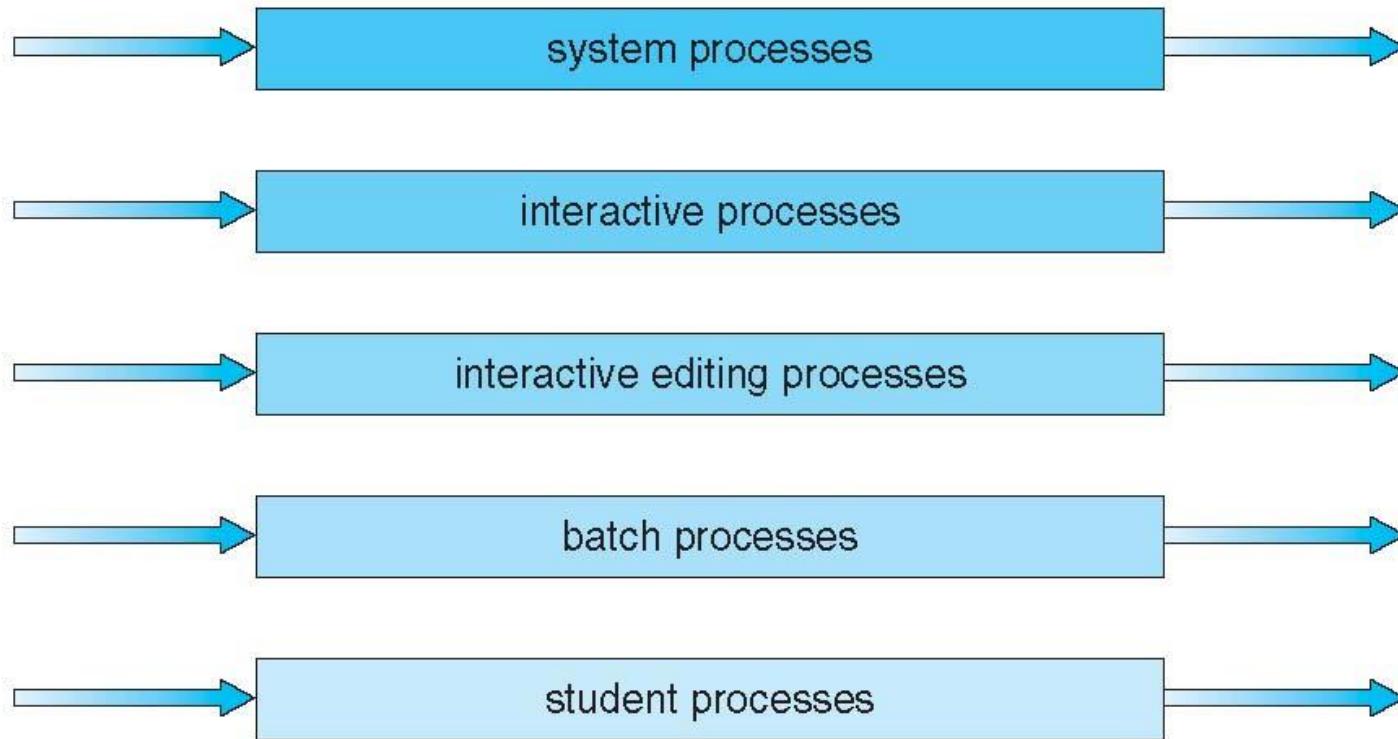
- **Ready queue** is partitioned into separate queues, eg:
 - **foreground** (interactive)
 - **background** (batch)
- Process permanently assigned a queue based upon properties like **memory size, process type, priority etc.**
- Each queue has its own scheduling algorithm:
 - foreground – RR
 - background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





Multilevel Queue Scheduling

highest priority



lowest priority





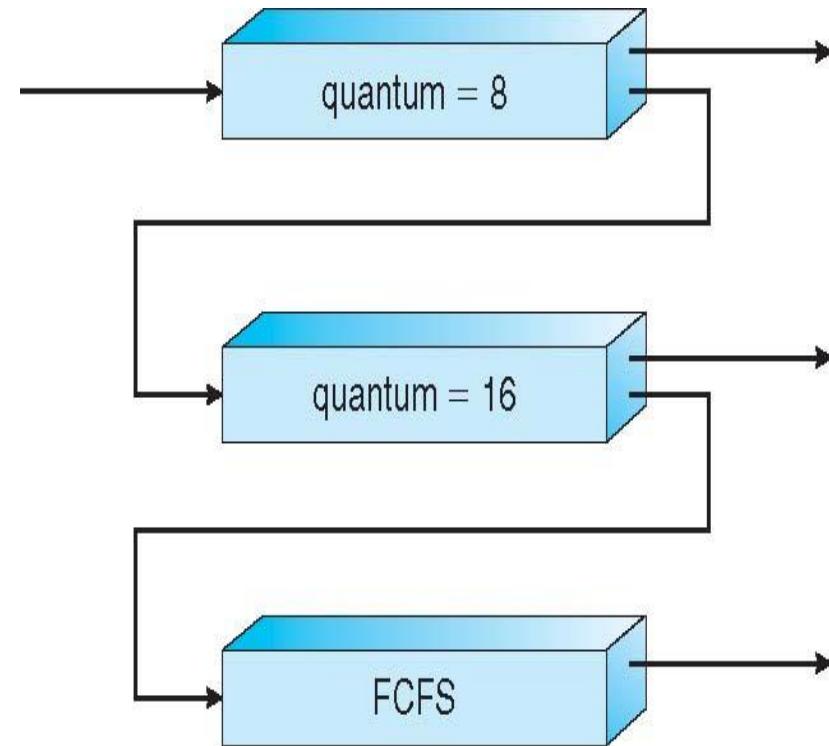
Example of Multilevel Feedback Queue

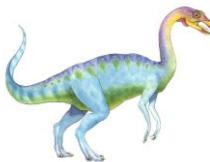
■ Three queues:

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

■ Scheduling

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

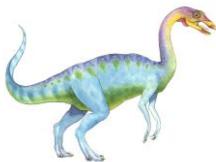




Multilevel Feedback Queue

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



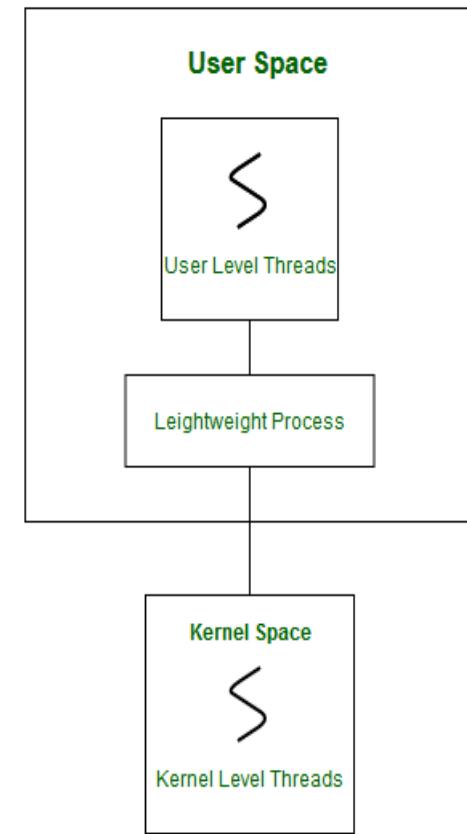


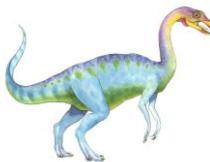
Thread Scheduling

- Scheduling of threads involves two boundary scheduling.
- Scheduling of **user-level threads (ULT)** to kernel-level **threads (KLT)** via lightweight process (LWP)
- Scheduling of **kernel-level threads** by the system scheduler to perform different unique OS functions.

Lightweight Process (LWP)

- Light-weight process are threads in the user space that acts as an interface for the ULT to access the physical CPU resources.
- Thread library schedules which thread of a process to run on which LWP and how long.
- The number of LWPs created by the thread library depends on the type of application.





Thread Scheduling

- **Contention Scope:** The word contention here refers to the competition among the User level threads to access the kernel resources. Thus, this control defines the extent to which contention takes place. It is defined by the application developer using the thread library.
- Contention Scope is classified as-
- **.Process Contention Scope (PCS) :**
The contention takes place among threads **within a same process**. The thread library schedules the high-prioritized PCS thread to access the resources via available LWPs (priority as specified by the application developer during thread creation).
- **System Contention Scope (SCS) :**
The contention takes place among **all threads in the system**. In this case, every SCS thread is associated to each LWP by the thread library and are scheduled by the system scheduler to access the kernel resources.





Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
- The POSIX Pthread library provides function ***Pthread_attr_setscope*** to define the type of contention scope for a thread during its creation.

`int Pthread_attr_setscope(pthread_attr_t *attr, int scope)`

- The first parameter denotes to **which thread within the process** the scope is defined.
- The second parameter defines **the scope of contention** for the thread pointed. It takes two values.
 - `PTHREAD_SCOPE_PROCESS` schedules threads using PCS scheduling
 - `PTHREAD_SCOPE_SYSTEM` schedules threads using SCS scheduling





Exercises

- 1 Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	2	2
P_2	1	1
P_3	8	4
P_4	4	2
P_5	5	3

The processes are assumed to have arrived in the order P_1, P_2, P_3, P_4, P_5 , all at time 0.

- Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, non-preemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).
- What is the turnaround time of each process for each of the scheduling algorithms in part a?
- What is the waiting time of each process for each of these scheduling algorithms?
- Which of the algorithms results in the minimum average waiting time (over all processes)?





Exercise 3

Consider the following set of processes, arriving at the given times and having the following CPU burst time and priorities (Smaller number is having higher priority):

Draw a Gantt chart and calculate average waiting time and turnaround time of each process using SJF, Priority and Round robin (quantum 3 ms). Assume pre-emptive scheduling policy for SJF and Priority scheduling.

Process	Arrival Time(ms)	Burst Time (ms)	Priority
A	0	8	3
B	3	4	1
C	5	7	4
D	8	3	2





Exercises

The following processes are being scheduled using a preemptive, round-robin scheduling algorithm. Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. In addition to the processes listed below, the system also has an *idle task* (which consumes no CPU resources and is identified as P_{idle}). This task has priority 0 and is scheduled whenever the system has no other available processes to run. The length of a time quantum is 10 units. If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.

Thread	Priority	Burst	Arrival
P_1	40	20	0
P_2	30	25	25
P_3	30	25	30
P_4	35	15	60
P_5	5	10	100
P_6	10	10	105



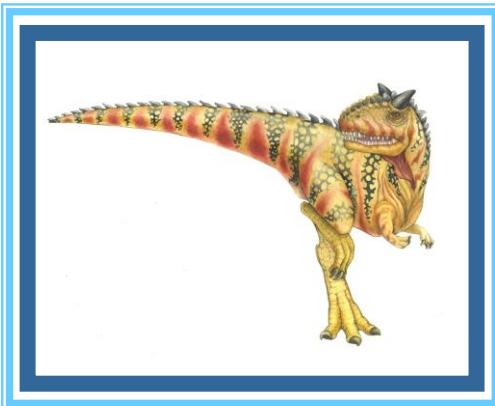


Exercises

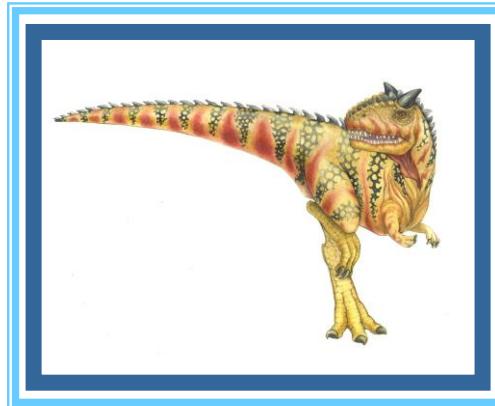
- a. Show the scheduling order of the processes using a Gantt chart.
- b. What is the turnaround time for each process?
- c. What is the waiting time for each process?
- d. What is the CPU utilization rate?

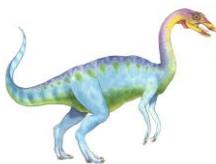


End of Chapter 5



Chapter 6: Process Synchronization

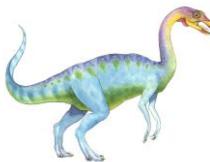




Module 6: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores





Objectives

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity

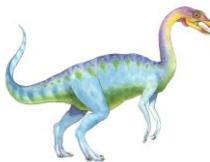




Process Synchronization

- Process Synchronization is a task in a multi-process system to ensure that the access of shared resources is done in a controlled and predictable manner.
- It aims to resolve the problem of **race conditions** and other synchronization issues in a **concurrent system**.
- The main objective of process synchronization is to ensure that multiple processes access shared resources without interfering with each other and to prevent the possibility of **inconsistent data** due to concurrent access.





Background

Consider the example of Producer-consumer 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 and is decremented by the consumer after it consumes a buffer.

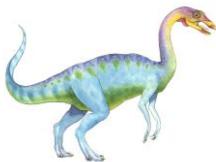




Producer

```
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  
}  
}
```





Race Condition

- `count++` could be implemented as in machine language

```
register1 = count //register 1 is local CPU registers  
register1 = register1 + 1  
count = register1
```

- `count--` could be implemented as

```
register2 = count  
register2 = register2 - 1  
count = register2
```

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

```
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}
```





Race Condition

- A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.
- To guard against the race condition we need to ensure that only one process at a time can be manipulating the variable counter.
- To make such a guarantee, we require that the processes be **synchronized** using **process synchronization** techniques





Critical Section Problem

Critical section problem is a means of designing a way for **cooperative processes** to access **shared resources** without creating data inconsistencies.

A critical section is a **code segment** that can be accessed by only one process at a time.

Consider system of n processes $\{p_0, p_1, \dots, p_{n-1}\}$

- Each process has **critical section**
- Here, process may be changing common variables, updating table, writing file, etc
 - When one process in critical section, no other may be in its critical section
- Each process must ask permission to enter critical section in **entry section**, that follow **critical section** with **exit section**, then **remainder section**





Critical Section

- General structure of process P_i

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

entry section: Each process must request permission to enter its critical section. Entry section is the section of code implementing this request.



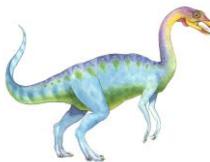


Solution to Critical-Section Problem

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

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





Peterson's Solution

Is a classic software-based solution to the critical-section problem

- Is restricted to **two process solution**
- The two processes share two variables:
 - **int turn;**
 - **Boolean flag[2]**
- The variable **turn** indicates whose turn it is to enter the critical section.
- The **flag** array is used to indicate if a process is ready to enter the critical section.
- **flag[i] = true** implies that process Pi is ready!





Algorithm for Process P_i

```
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j); //wait until j exit  
        critical section  
    flag[i] = false;//indicate Pi is out of critical section  
        remainder section  
} while (true);
```

Peterson's Solution preserves all three conditions:

- Mutual Exclusion is assured as only one process can access the critical section at any time.
- Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.
- Bounded Waiting is preserved as every process gets a fair chance.





Synchronization Hardware

- Many systems provide **hardware support** for implementing the **critical section code**.
- All solutions are based on idea of **locking**
 - Protecting critical regions via locks
- In Uniprocessors – **interrupts could be disabled** from occurring during a shared variable was being modified.
- Currently running code would execute without preemption
 - Generally **inefficient on multiprocessor** systems
 - ▶ Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
 - ▶ **Atomic** = non-interruptible
 - Either test memory word and set value
 - Or swap contents of two memory words





Solution to Critical-section Problem Using Locks

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

- The abstract of main concepts behind these hardware instructions for critical section problem are discussed using
- **test and set()** and **compare and swap()** instructions.





TestAndSet Instruction

■ Definition:

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

Must be executed atomically





Solution using TestAndSet

- Shared Boolean variable lock, initialized to **false**
- Solution:

```
do {  
    while ( TestAndSet (&lock ) )  
        ; // do nothing  
        // critical section  
    lock = FALSE;  
        // remainder section  
} while (TRUE);
```





TestAndSet Instruction

- **Test and Set:** The shared variable **lock** is initialized to **false**.
- TestAndSet(lock) algorithm working –
- It always returns whatever value is sent to it and sets lock to **true**.
- The first process will enter the critical section at once as TestAndSet(lock) will return **false** and it'll break out of the while loop.
- The other processes cannot enter now as lock is set to **true** and so the while loop continues to be true.
- **Mutual exclusion is ensured.**

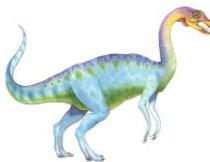




TestAndSet Instruction

- Once the first process gets out of the critical section, lock is changed to **false**
- So, now the other processes can enter one by one.
Progress is also ensured.
- However, after the first process, any process can go in. There is no queue maintained, so any new process that finds the lock to be false again can enter.
- **So bounded waiting is not ensured.**





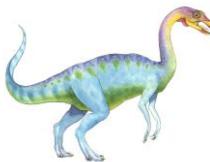
compare_and_swap Instruction

Definition:

```
int compare_and_swap(int *value, int expected, int  
new_value) {  
    int temp = *value;  
    if (*value == expected)  
        *value = new_value;  
    return temp;  
}
```

- Returns the original value of passed parameter “**value**”
- **value** is set to new value only if the expression ($*\text{value} == \text{expected}$) is true.





Solution using compare_and_swap

- Shared integer “lock” initialized to **0 (false)**
- Solution:

```
do {  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
    /* critical section */  
    lock = 0;  
    /* remainder section */  
} while (true);
```



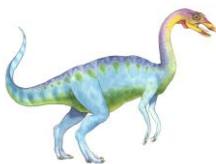


compare_and_swap Instruction

Mutual exclusion can be provided as follows:

- A global variable (**lock**) is declared and is **initialized to 0**.
- The first process that invokes compare and swap() will set **lock to 1**.
- It will then enter its critical section, because the original value of lock (i.e 0) was equal to the expected value of 0.
- Subsequent calls to compare and swap() will not succeed, because lock now is not equal to the expected value of 0 (**changed to 1 by previous entered process**).
- When a process exits its critical section, **it sets lock back to 0**, which allows another process to enter its critical section.



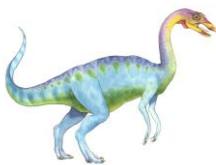


Bounded-waiting Mutual Exclusion with TestandSet()

The common data structures are

- **boolean waiting[n];** //for each process which checks whether or not a process has been waiting.
- **boolean lock;** //These data structures are initialized to **false**
- A ready queue is maintained with respect to the process in the critical section.
- All the processes coming in next are added to the ready queue with respect to their process number,
- Once the i^{th} process gets out of the critical section, it does not turn lock to false, Instead, it checks if there is any process waiting in the queue.
- If there is **no process** waiting then the **lock value is changed to false** and any process which comes next can enter the critical section.
- If there is, **then that process' waiting value is turned to false**, so that the **first while loop** becomes false and it can enter the critical section.
- This ensures **bounded waiting**.
- Thus, the problem of process synchronization can be solved through this algorithm.





Bounded-waiting Mutual Exclusion with TestandSet()

```
do {  waiting[i] = TRUE;  
      key = TRUE;  
      while (waiting[i] && key) // loop will continue until key becomes false  
      key = TestAndSet(&lock); // TestAnsSet will set the value of key  
      waiting[i] = FALSE;  
          // critical section  
      j = (i + 1) % n;  
      while ((j != i) && !waiting[j]) // check any process is waiting in queue  
          j = (j + 1) % n;  
      if (j == i)  
          lock = FALSE;  
      else  
          waiting[j] = FALSE; // Indicates that the process j is selected  
          // remainder section  
} while (TRUE);
```





Mutex Locks

- Previous hardware solutions are complicated and generally inaccessible to application programmers
- OS designers build **software tools** to solve critical section problem. Simplest is **mutex** lock
- Protect a critical section by first **acquire()** a lock then **release()** the lock
 - Uses boolean variable “**available**” indicating if lock is available or not
- Calls to **acquire()** and **release()** must be atomic
 - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
 - This lock therefore called a **spinlock**

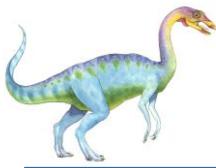




acquire() and release()

- `acquire() {`
 `while (!available)`
 `; /* busy wait */`
 `available = false;`
 `}`
- `release() {`
 `available = true;`
 `}`
- `do {`
 `acquire lock`
 `critical section`
 `release lock`
 `remainder section`
`}` `while (true);`

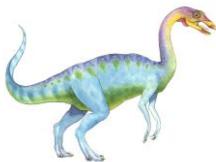




Semaphore

- Semaphore is a synchronization tool similar to mutex but more sophisticated that does not require busy waiting. It can be used to manage access to a **pool of resources** rather than just one
- Semaphore S – **integer variable**
- Semaphore is accessed through two standard atomic operations **wait()** and **signal()**
 - Originally called **P()** and **V()**
- Less complicated
 - **wait (S) {**
 while S <= 0
 ; // no-op
 S--;
 }
 - **signal (S) {**
 S++;
}





Semaphore

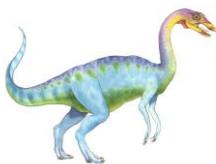
■ Wait (P) Operation:

- If the semaphore's count is greater than zero, decrement the count and allow the thread to proceed.
- If the count is zero, the thread is blocked until the count becomes greater than zero.

■ Signal (V) Operation:

- Increment the semaphore's count.
- If there are threads blocked on a wait operation, one of them is unblocked.



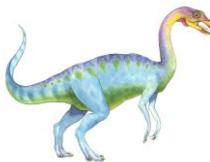


Semaphore as General Synchronization Tool

- **Counting** semaphore – integer value can range over an unrestricted domain
- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement
 - Also known as **mutex locks**
- Can implement a counting semaphore **S** as a binary semaphore
- Provides mutual exclusion

```
Semaphore mutex; // initialized to 1  
do {  
    wait (mutex);  
    // Critical Section  
    signal (mutex);  
    // remainder section  
} while (TRUE);
```

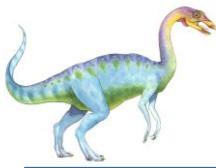




Semaphore Implementation

- 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 is short
 - ▶ Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

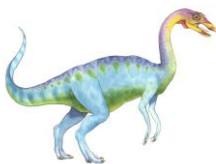




Semaphore Implementation with no Busy waiting

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





Semaphore Implementation with no Busy waiting (Cont.)

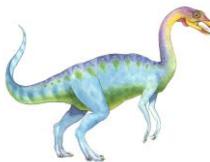
- Implementation of wait:

```
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        add this process to S->list;  
        block();  
    }  
}
```

- Implementation of signal:

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





Deadlock and Starvation

- Implementation of a semaphore with a waiting queue may result in a deadlock state, if one or two process are waiting for an event that can be caused by only one of the waiting process. Such a state is called **deadlock**.
- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let **S** and **Q** be two semaphores initialized to 1

P_0

wait (S);
wait (Q);

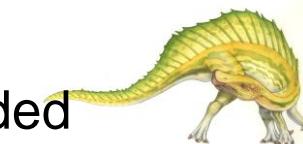
P_1

wait (Q);
wait (S); .

.
. .
signal (S);
signal (Q);

.
. .
signal (Q);
signal (S);

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended



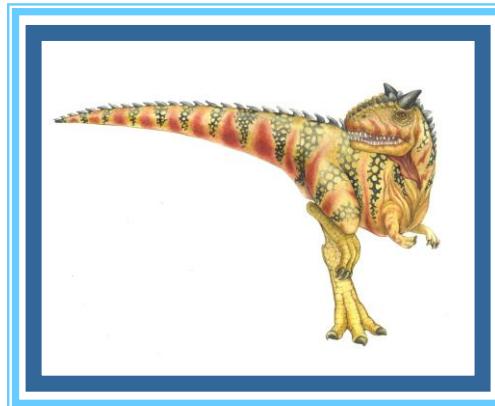


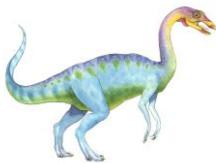
Priority Inversion

- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
- As an example, assume we have **three processes: *L*, *M*, and *H***, whose priorities follow the order $L < M < H$.
- Assume **process *H*** requires **resource *R***, which is currently being accessed by **process *L***.
- Suppose that **process *M*** becomes runnable, thereby preempting process *L*. Indirectly, a process with a lower priority: **process *M***, has affected how long process *H* must wait for *L*(**Priority Inversion**)
- **priority-inheritance protocol** is used to solve this problem:
- According to this protocol, all processes that are accessing resources needed by a higher-priority process **inherit the higher priority** until they are finished with the resources. When they are finished, their priorities revert to their original values.
- In this example **process *L*** will inherit the priority of **process *H***, thereby preventing **process *M*** from preempting its execution.
- Thus, resource *R* would now be available, process *H* not *M*.



Chapter 7: Deadlocks





Chapter 7: Deadlocks

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance





Chapter Objectives

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system

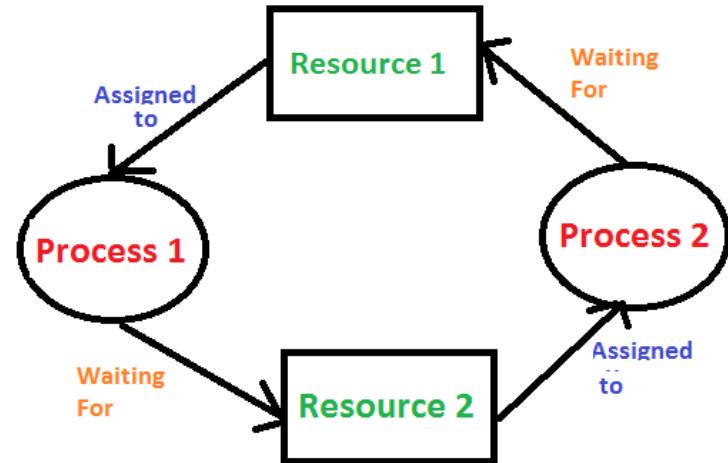




System Model

A **deadlock** is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

- Consider a system consists of resources
- Resource types R_1, R_2, \dots, R_m
CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
 - **request**
 - **use**
 - **Release**



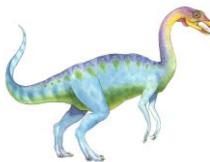


Deadlock Characterization

Deadlock can arise if **four conditions** hold simultaneously
(Necessary Conditions)

- **Mutual exclusion:** Two or more resources are **non-shareable**. only one process at a time can use a resource
- **Hold and wait:** a process holding **at least one resource** is **waiting** to acquire additional resources held by other processes
- **No preemption:** a resource can be released only **voluntarily** by the process holding it, after that process has completed its task
- **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 , ..., 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 .





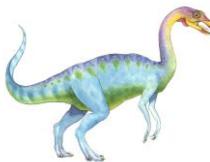
Resource-Allocation Graph

A **resource allocation graphs** shows which resource is held by which process and which process is waiting for a resource of a specific kind.

A set of vertices V and a set of edges E .

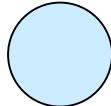
- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$



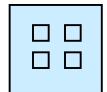


Resource-Allocation Graph (Cont.)

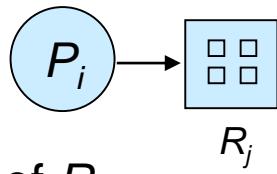
- Process



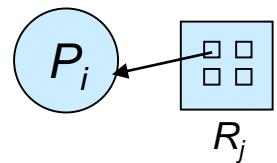
- Resource Type with 4 instances



- P_i requests instance of R_j

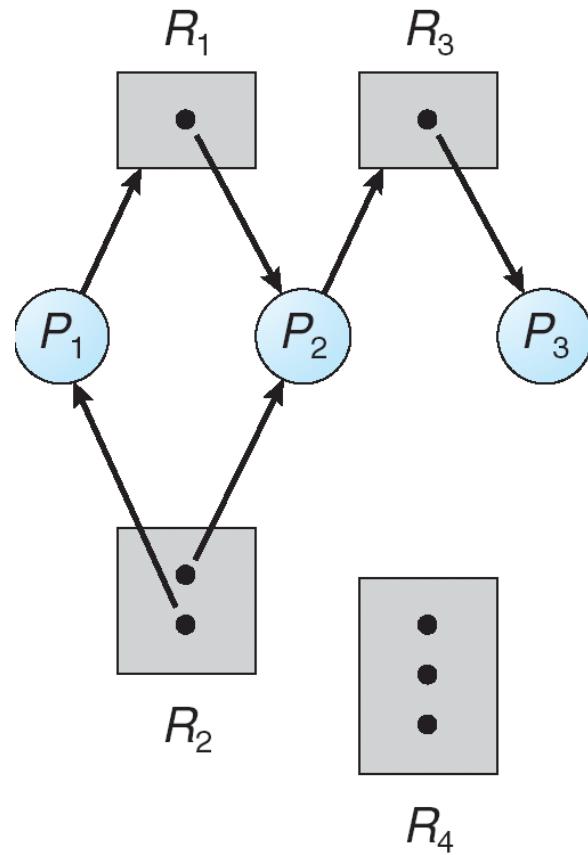


- P_i is holding an instance of R_j



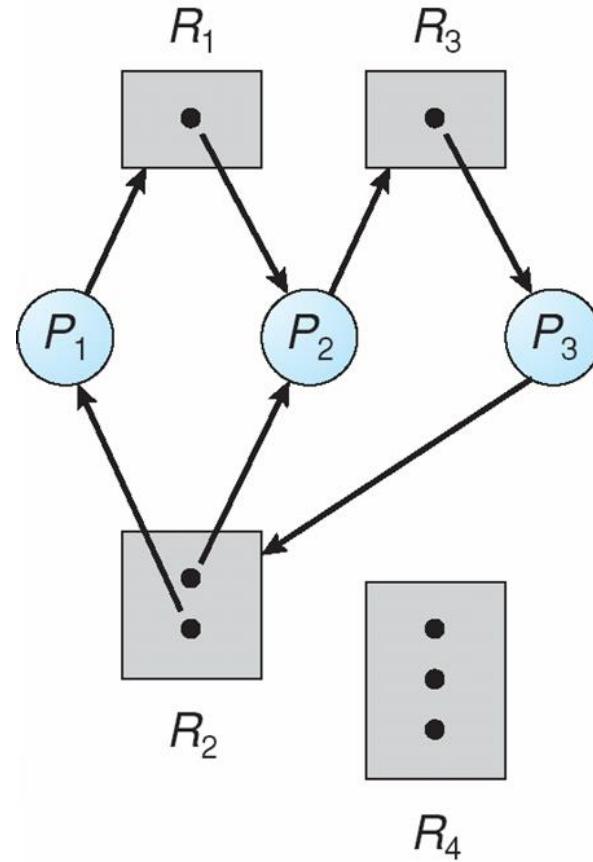


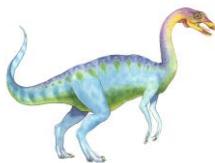
Example of a Resource Allocation Graph



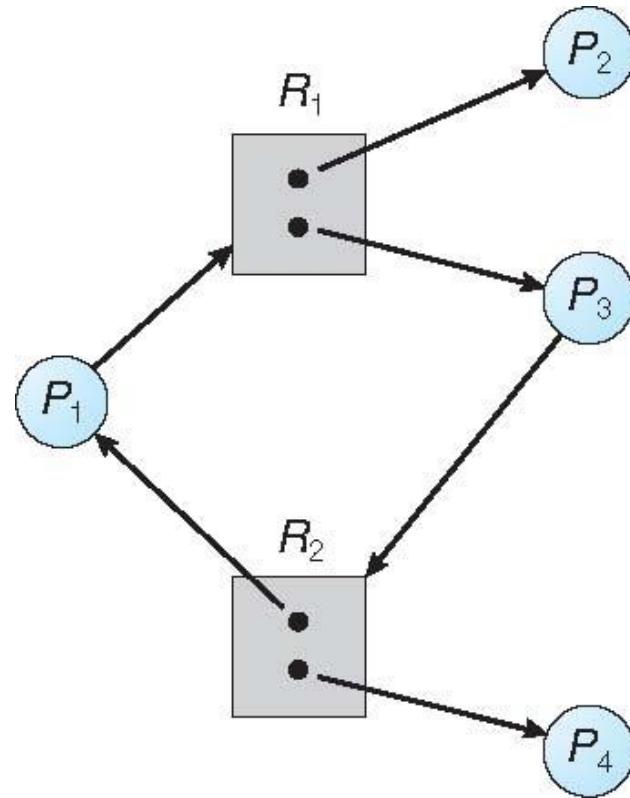


Resource Allocation Graph With A Deadlock





Graph With A Cycle But No Deadlock





Basic Facts

- If graph contains no cycles \Rightarrow no deadlock
- If graph contains a cycle \Rightarrow
 - if only one instance per resource type, then deadlock
 - if several instances per resource type, possibility of deadlock





Methods for Handling Deadlocks

Deal with the deadlock problem in one of **three ways**:

1) Deadlock prevention or avoidance:

Prevention: Ensure that the system will **never** enter a deadlock state. provides a set of methods to ensure that at least one of the **necessary conditions cannot hold**. These methods prevent deadlocks by constraining how requests for resources can be made.

Avoidance: requires that the operating system be given **additional information** in advance concerning which resources a process will request and use during its lifetime.

2) Deadlock detection and recovery: We can allow the system to enter a deadlocked state, **detect it, and recover**.

3) Deadlock ignorance: We can ignore the problem altogether and pretend that deadlocks never occur in the system. **If a deadlock is very rare, then let it happen and reboot the system.**





Deadlock Prevention

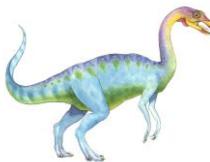
Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a process **requests a resource, it does not hold any other resources**
 - Require process **to request and be allocated all its resources before it begins execution**
 - Allow process to **request resources** only when the process has **none allocated** to it.
 - Process must release all resources before requesting any new resource

Disadvantages

- **Low resource utilization:** allotted resources may not be used for long time
- **Starvation:** a process may need to wait for long time for a resource





Deadlock Prevention (Cont.)

- **No Preemption** : Resources allocated are not preempted(to solve this use the following protocol)
 - If a process that is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
 - 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
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration to **eliminate circular wait**.

Example: Each resource will be assigned a **numerical number**. A process can request the resources to increase/decrease order of numbering. If the **P1** process is allocated **R5** resources, now next time if **P1** asks for **R4**, **R3** lesser than **R5** such a request will **not be granted**, only a request for resources more than **R5** will be granted.





Deadlock Example

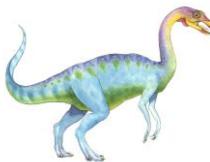
```
/* thread one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex); // F(first_mutex)=1
    pthread_mutex_lock(&second_mutex); // F(second_mutex)=5
    /** * Do some work */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    pthread_exit(0);
}

/* thread two runs in this function */
void *do_work_two(void *param)
{ pthread_mutex_lock(&second_mutex);

    pthread_mutex_lock(&first_mutex);
    /** * Do some work */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);
    pthread_exit(0);
}
```

Only defining order will not help. Developers must follow the ordering to prevent deadlock





Deadlock Avoidance

Requires that the system has some additional *a priori* information available

- Simplest and most useful model requires that each process declare the **maximum number** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a **circular-wait condition**
- Resource-allocation *state* is defined by the number of **available and allocated resources**, and the maximum demands of the processes





Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a **safe state**
- System is in **safe state** if there exists a sequence $\langle P_1, P_2, \dots, P_n \rangle$ of all processes in the systems such that for each P_i , the resources that P_i can still request can be satisfied by currently **available resources + resources held by all the P_j , with $j < i$**
- That is:
 - If P_i resource needs are **not immediately available**, then P_i can wait until all **P_j have finished**
 - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
 - When **P_i terminates**, P_{i+1} can obtain its needed resources, and so on

If no such sequence exists then the system safe is said **unsafe**.

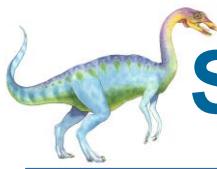




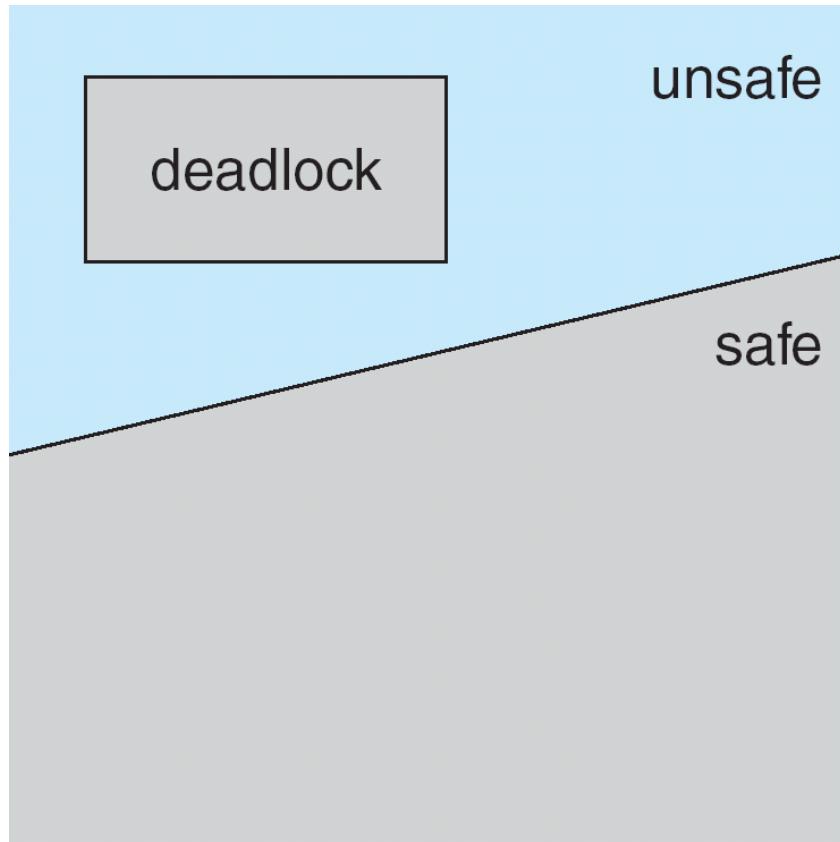
Basic Facts

- If a system is in safe state \Rightarrow no deadlocks
- If a system is in unsafe state \Rightarrow possibility of deadlock
- A safe state is not a deadlocked state.
- Avoidance \Rightarrow ensure that a system will never enter an unsafe state.





Safe, Unsafe, Deadlock State spaces





Avoidance Algorithms

- Single instance of a resource type
 - Use a resource-allocation graph algorithm

- Multiple instances of a resource type
 - Use the banker's algorithm





Resource-Allocation Graph Scheme

- If we have a resource-allocation system with only **one instance of each resource type**, we can use a **variant of the resource-allocation graph** for deadlock avoidance.
- Here, in addition to the **request** and **assignment** edges we introduce a new type of edge, called a **claim edge**
- A **Claim edge** $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j in future
- Claim edge is represented by a **dashed line**





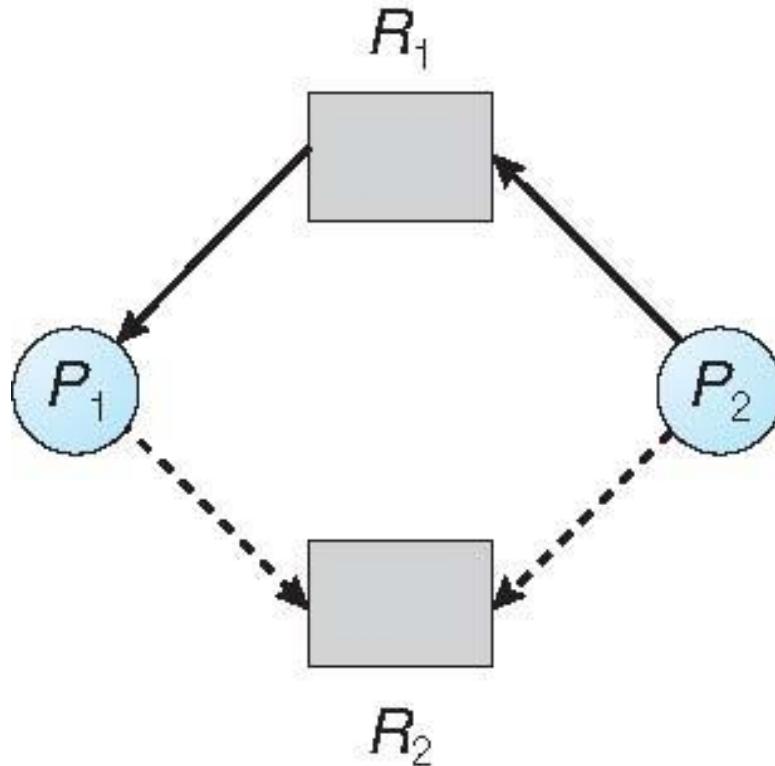
Resource-Allocation Graph Scheme

- When a process **requests** a resource, **claim edge** converts to **request edge**
- When the resource is **allocated** to the process **request edge** converted to an **assignment edge**
- When a resource is **released** by a process, **assignment edge** reconverts to a **claim edge**
- Resources must be claimed *a priori* in the system, thus before process P_i starts executing, all its **claim edges** must already appear in the resource-allocation graph



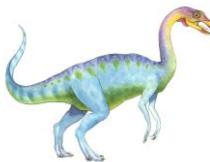


Resource-Allocation Graph



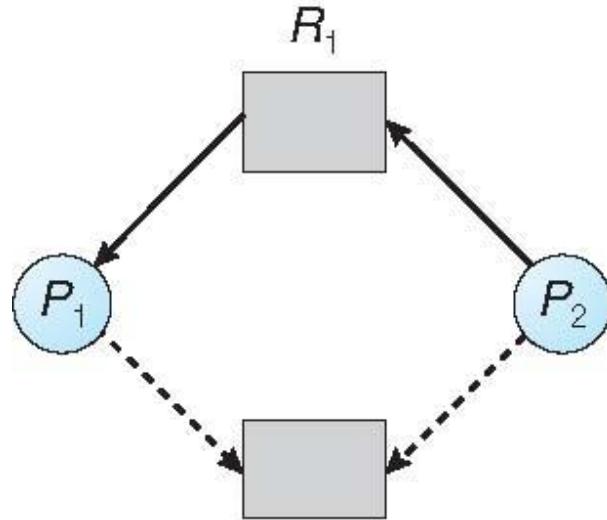
Resource-allocation graph for deadlock avoidance



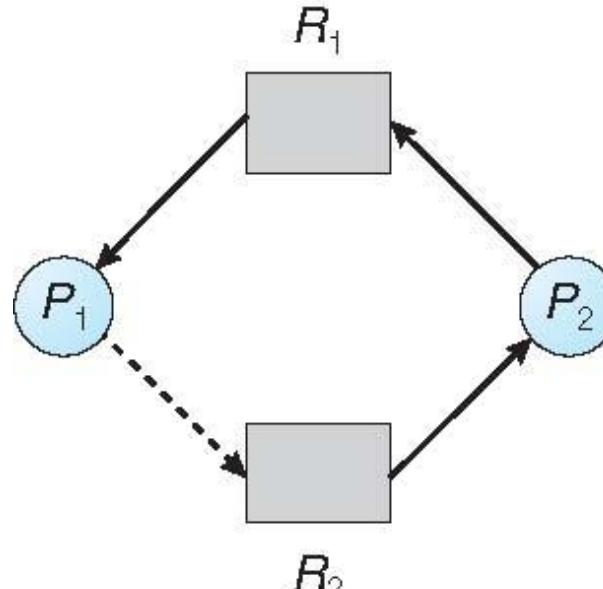


Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the **request edge** to an **assignment edge** does not result in the formation of a **cycle** in the resource allocation graph
- If a cycle is found, in that case, process P_i will have to wait for its requests to be satisfied.
- Consider the example shown, that illustrate the formation of cycle



Resource-allocation graph
for deadlock avoidance



An unsafe state in a
resource-allocation graph





Banker's Algorithm

- The resource-allocation-graph algorithm is not applicable to a resource allocation system with **multiple instances** of each resource type
- Each process must declare the maximum number of instances of each resource type that it may need
- 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





Data Structures for the Banker's Algorithm

Let $n = \text{number of processes}$, and $m = \text{number of resources types}$.

Following **data structures** are maintained to implement the banker's algorithm.

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

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





Safety Algorithm

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

Work = Available

Finish [i] = false for $i = 0, 1, \dots, n-1$

2. Find an i such that both:
 - (a) **Finish [i] = false**
 - (b) **Need_i ≤ Work**If no such i exists, go to step 4
3. **Work = Work + Allocation_i,**
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 for Process P_i

Next, describe the algorithm for determining whether requests can be safely granted.

Let, Request_i = request vector for process P_i . If $\text{Request}_i[j] = k$ then process P_i wants k instances of resource type R_j

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:

$$\text{Available} = \text{Available} - \text{Request}_i;$$

$$\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i;$$

$$\text{Need}_i = \text{Need}_i - \text{Request}_i;$$

- If safe \Rightarrow the resources are allocated to P_i
- If unsafe $\Rightarrow P_i$ must wait, and the old resource-allocation state is restored





Example of Banker's Algorithm

- 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>Allocation</u>			<u>Max</u>	<u>Available</u>	
	A	B	C	A	B	C
P_0	0	1	0	7	5	3
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





Example (Cont.)

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

	<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

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





Applying the Safety algorithm on the given system,

$m=3, n=5$	Step 1 of Safety Algo								
Work = Available									
Work = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>3</td><td>3</td><td>2</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	3	3	2	0	1	2	3	4	
3	3	2							
0	1	2	3	4					

For $i = 0$	Step 2
Need ₀ = 7, 4, 3	7, 4, 3 3, 3, 2
Finish [0] is false and Need ₀ > Work	
So P ₀ must wait	But Need \leq Work

For $i = 1$	Step 2
Need ₁ = 1, 2, 2	1, 2, 2 3, 3, 2
Finish [1] is false and Need ₁ < Work	
So P ₁ must be kept in safe sequence	

3, 3, 2	2, 0, 0	Step 3											
Work = Available													
Work = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>A</td><td>B</td><td>C</td></tr><tr><td>5</td><td>3</td><td>2</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	A	B	C	5	3	2	0	1	2	3	4		
A	B	C											
5	3	2											
0	1	2	3	4									

For $i = 2$	Step 2
Need ₂ = 6, 0, 0	6, 0, 0 5, 3, 2
Finish [2] is false and Need ₂ > Work	
So P ₂ must wait	

For $i = 3$	Step 2
Need ₃ = 0, 1, 1	0, 1, 1 5, 3, 2
Finish [3] = false and Need ₃ < Work	

So P₃ must be kept in safe sequence

5, 3, 2	2, 1, 1	Step 3											
Work = Work + Allocation ₃													
Work = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>A</td><td>B</td><td>C</td></tr><tr><td>7</td><td>4</td><td>3</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	A	B	C	7	4	3	0	1	2	3	4		
A	B	C											
7	4	3											
0	1	2	3	4									

For $i = 4$	Step 2
Need ₄ = 4, 3, 1	4, 3, 1 7, 4, 3
Finish [4] = false and Need ₄ < Work	

So P₄ must be kept in safe sequence

7, 4, 3	0, 0, 2	Step 3											
Work = Work + Allocation ₄													
Work = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>A</td><td>B</td><td>C</td></tr><tr><td>7</td><td>4</td><td>5</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	A	B	C	7	4	5	0	1	2	3	4		
A	B	C											
7	4	5											
0	1	2	3	4									

For $i = 0$	Step 2
Need ₀ = 7, 4, 3	7, 4, 3 7, 4, 5
Finish [0] is false and Need ₀ < Work	

So P₀ must be kept in safe sequence

7, 4, 5	0, 1, 0	Step 3											
Work = Work + Allocation ₀													
Work = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>A</td><td>B</td><td>C</td></tr><tr><td>7</td><td>5</td><td>5</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	A	B	C	7	5	5	0	1	2	3	4		
A	B	C											
7	5	5											
0	1	2	3	4									

6, 0, 0	7, 5, 5	Step 2
Finish [2] is false and Need ₂ < Work		
So P ₂ must be kept in safe sequence		

7, 5, 5	3, 0, 2	Step 3											
Work = Work + Allocation ₂													
Work = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>A</td><td>B</td><td>C</td></tr><tr><td>10</td><td>5</td><td>7</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	A	B	C	10	5	7	0	1	2	3	4		
A	B	C											
10	5	7											
0	1	2	3	4									

Finish [i] = true for $0 \leq i \leq n$ Step 4
Hence the system is in Safe state

The safe sequence is P₁, P₃, P₄, P₀, P₂



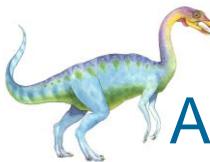
Example: P_1 Request (1,0,2)

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

	<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?
- Can request for (0,2,0) by P_0 be granted?





Applying the Safety algorithm on the given system,

What will happen if process P_1 requests one additional instance of resource type A and two instances of resource type C?

A B C
Request₁ = 1, 0, 2

To decide whether the request is granted we use Resource Request algorithm

Step 1
1, 0, 2 1, 2, 2
Request₁ < Need₁

Step 2
1, 0, 2 3, 3, 2
Request₁ < Available

Step 3

Available = Available – Request₁
Allocation₁ = Allocation₁ + Request₁
Need₁ = Need₁ - Request₁

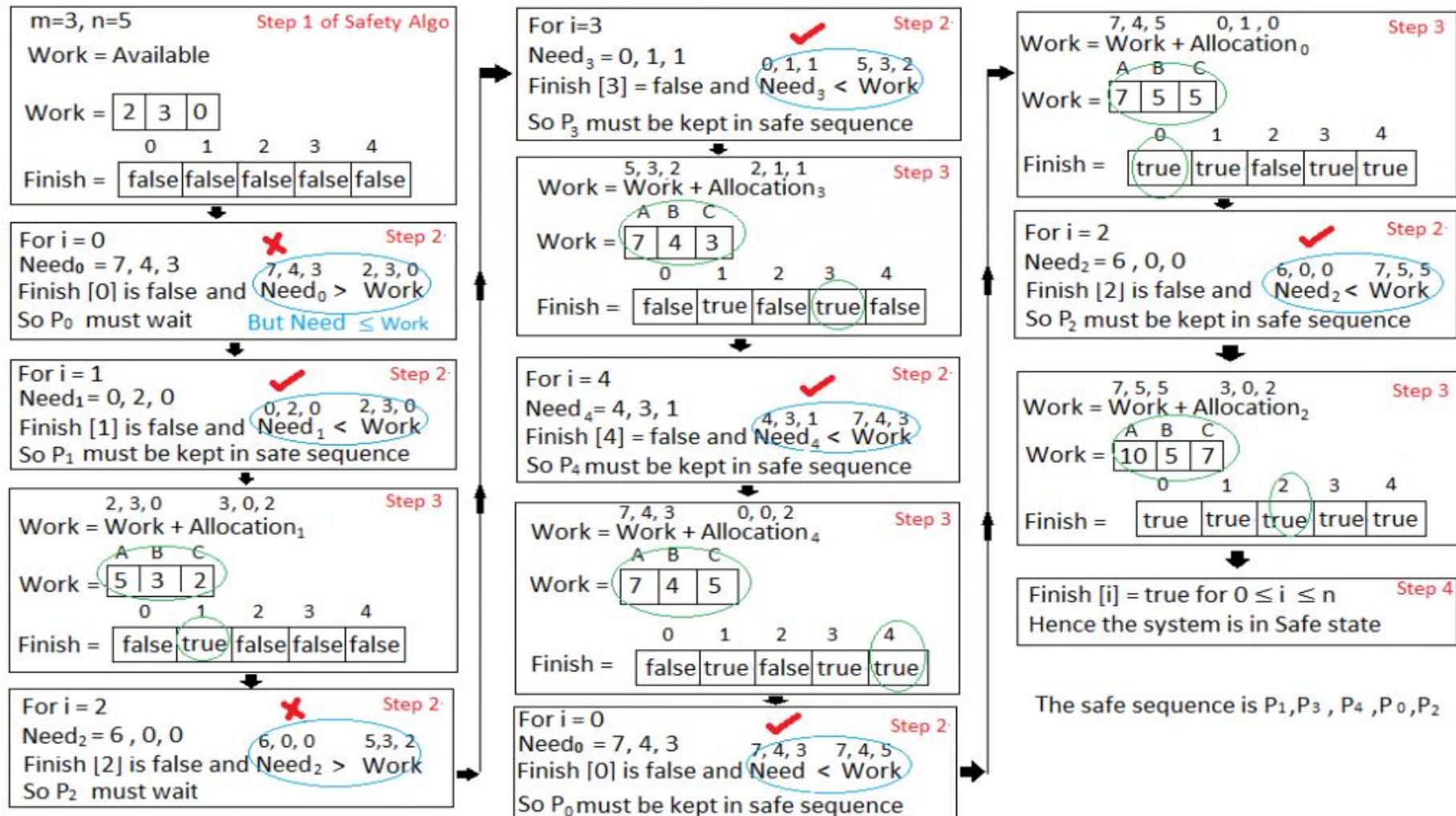
Process	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P ₀	0	1	0	7	4	3	2 3 0		
P ₁	3	0	2	0	2	0			
P ₂	3	0	2	6	0	0			
P ₃	2	1	1	0	1	1			
P ₄	0	0	2	4	3	1			

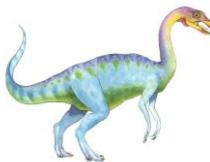




Applying the Safety algorithm on the given system,

We must determine whether this new system state is safe. To do so, we again execute Safety algorithm on the above data structures.





Example (Cont.)

- P_2 requests an additional instance of type **C**

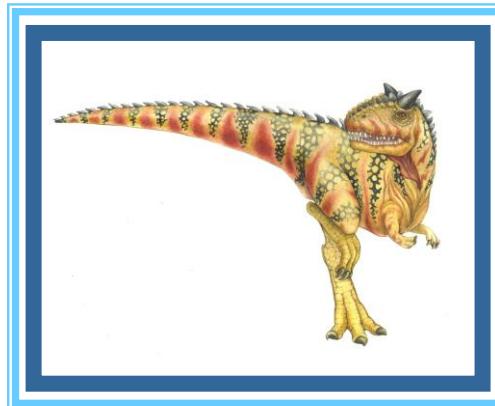
Request

	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

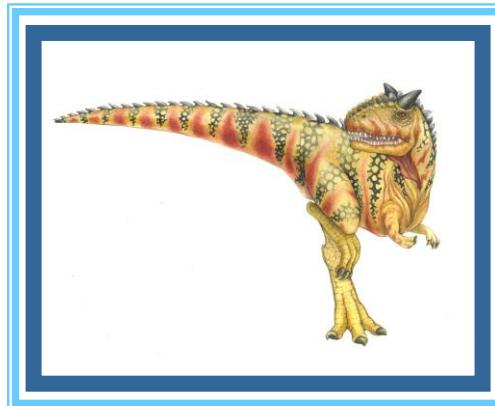
- State of system?
 - Can reclaim resources held by process P_0 , but insufficient resources to fulfill other processes; requests
 - Deadlock exists, consisting of processes P_1 , P_2 , P_3 , and P_4



End of Chapter 7



Chapter 8: Main Memory





Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table





Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging





Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage that **CPU can access directly**
- Register access in **one CPU clock** (or less)
- Main memory can take **many cycles**, causing a **stall**
- **Thus, Cache** sits between main memory and CPU registers to improve access speed.
- Not only speed but protection of memory is required to ensure correct operation,
- For proper system operation we must protect the **operating system** from access by **user processes** and also **processes must be protected from each other**
- This protection must be provided by the hardware





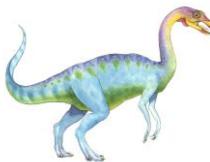
Background

One way of implementing protection is:

- Each process has a **separate memory space**.
- Separate per-process memory space protects the processes from each other and **multiple processes can be loaded in memory for concurrent execution**
- Provide **two registers**, called **base** and **a limit** to determine the range of **legal addresses** that the process may access
- The **base register** holds the smallest legal physical memory address; the **limit register** specifies the size of the range.

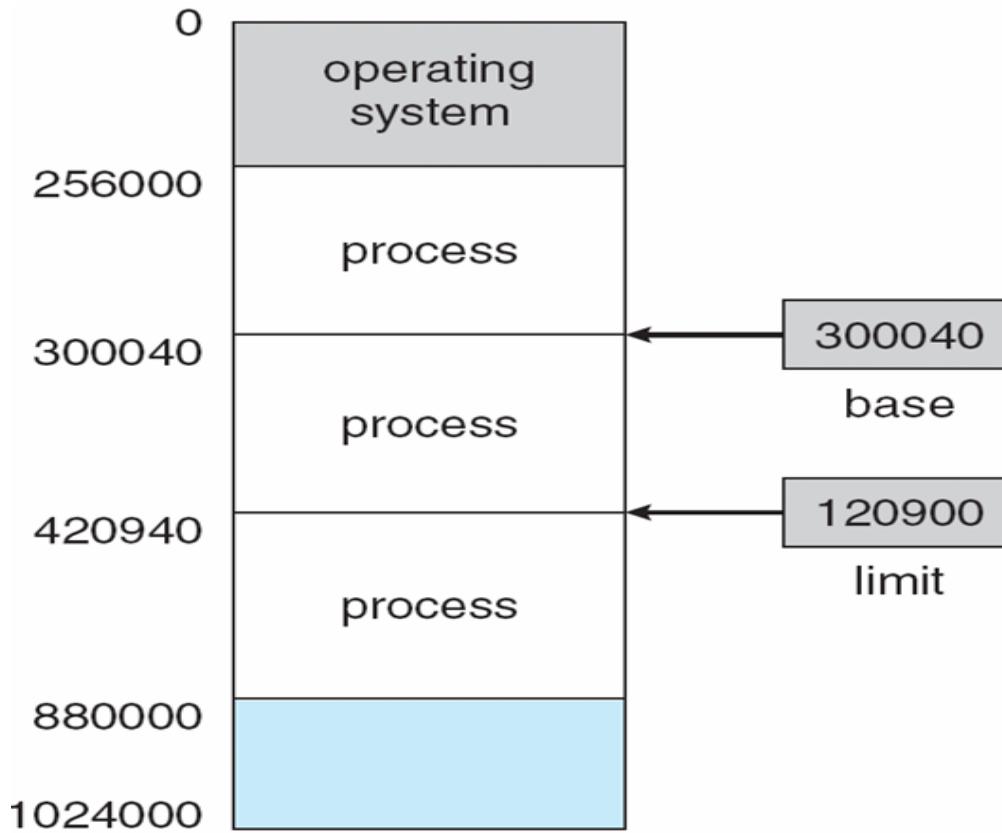
- **For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).**

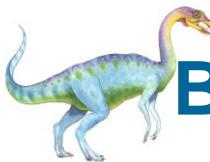




Background: Base and Limit Registers

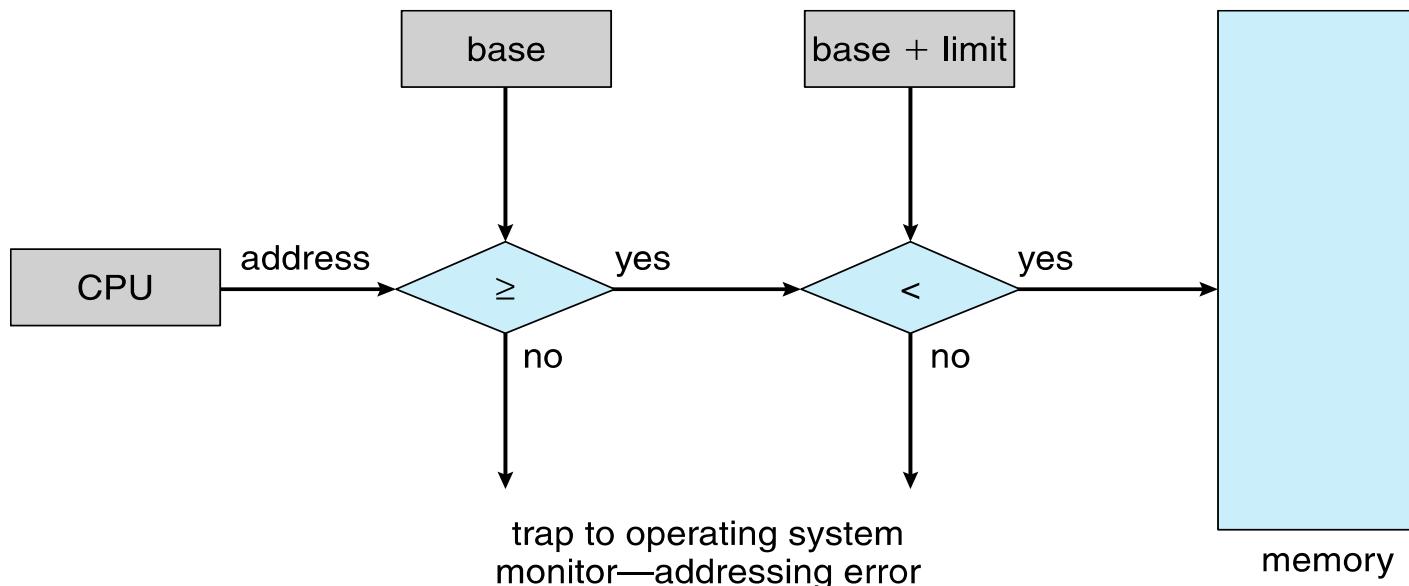
- A pair of **base** and **limit registers** define the logical address space





Background: Hardware Address Protection

- CPU must check every memory access generated in **user mode** to be sure it is between **base and limit** for that user.
- The **base and limit** registers can be loaded only by the **operating system**, which uses a special privileged instruction.
- Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system,





Address Binding

- Programs stored in secondary memory must be brought into main memory for execution.
- Most systems allow a user process to reside in any part of the physical memory
- Programs on disk, ready to be brought into memory to execute from an **input queue**
- User programs go through **many steps before execution**
- The first step done by OS in terms of memory management is **address binding**
- **Binding an address to data or instruction of a process is called address binding**





Address Binding

- CPU knows only the **actual address** of secondary memory
- Main memory address must be calculated for the process to execute.
- Thus it is needed to bind this main memory address with the instruction of the program for execution using address binding.
- Address binding can be done at three point of time
 - 1) Compile time
 - 2) Load time
 - 3) Run time





Binding of Instructions and Data to Memory

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





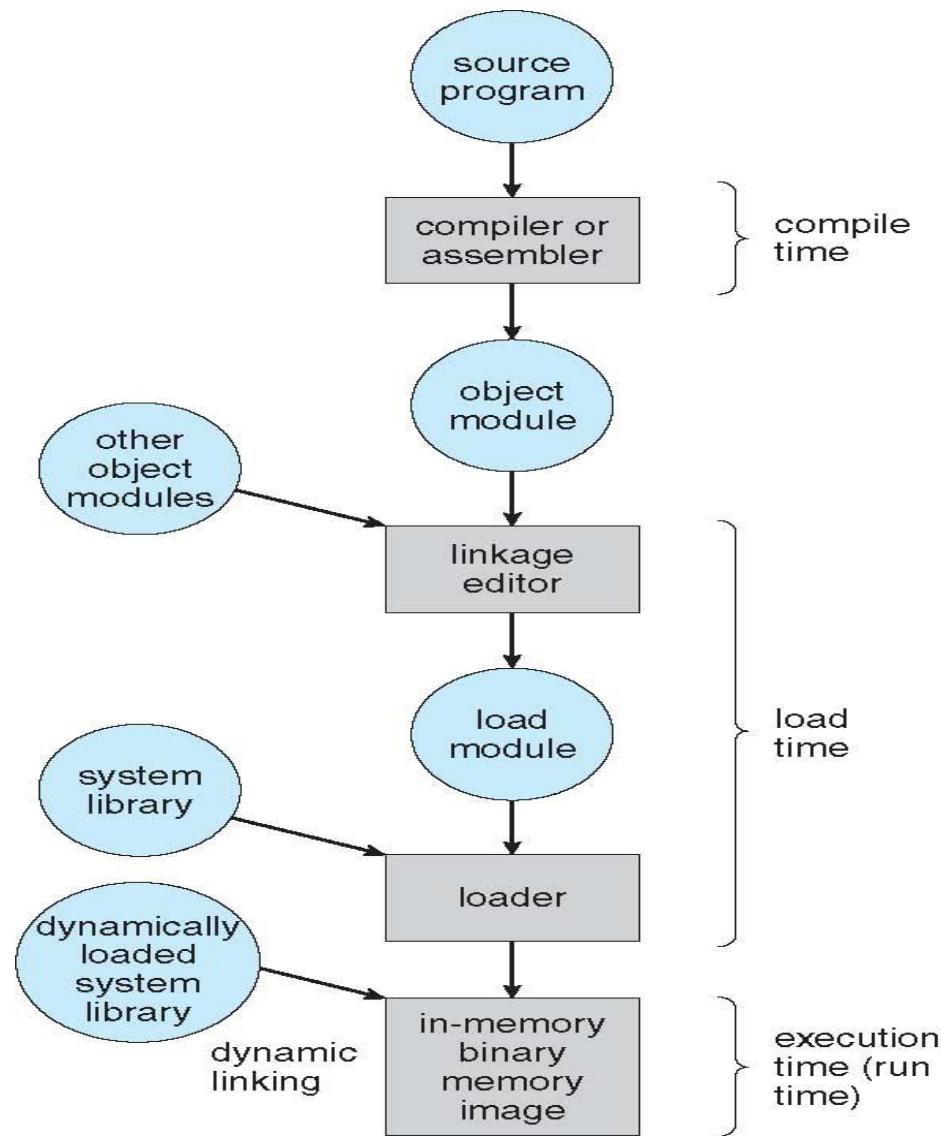
Address Binding

- Further, addresses can be represented in different ways at different stages of a program's life
 - Source code addresses are usually **symbolic(variables)**
 - Compiler **binds symbolic address to relocatable addresses**
 - ▶ i.e. “14 bytes from beginning of this module”
 - **Linker or loader will bind relocatable addresses to absolute addresses**
 - ▶ i.e. 74014
 - Each binding maps one address space to another





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

The process of **address binding** requires **2 types of address**:

- **Logical address** – generated by the CPU; also referred to as **virtual address**
- **Physical address** – address seen by the memory unit

- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes;
- Logical and physical addresses differ in execution-time address-binding scheme

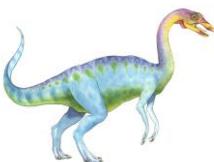




Memory-Management Unit (MMU)

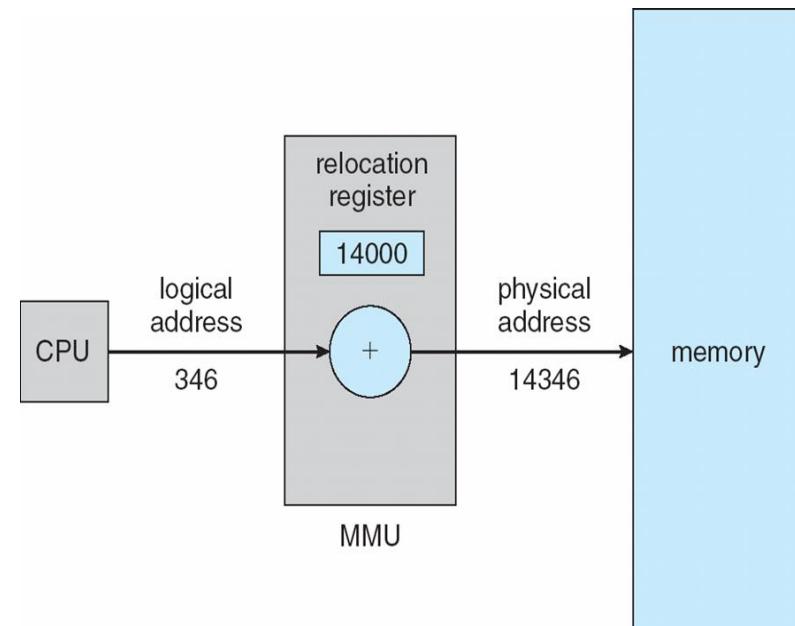
- The translation from **logical addresses** to **physical addresses** is managed by the operating system's **memory management system**.
- The value in the **relocation register(base register)** is added to every address generated by a user process at the time it is sent to memory
- The user program deals with ***logical addresses***; it never sees the ***real* physical addresses**





Dynamic Loading

- Routine is not loaded until it is called
- Better **memory-space utilization**; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
 - Implemented through program design
 - OS can help by providing libraries to implement dynamic loading



Dynamic relocation using a relocation register





Dynamic Linking

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- Dynamic linking –linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
 - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Consider applicability to patching system libraries
 - Versioning may be needed





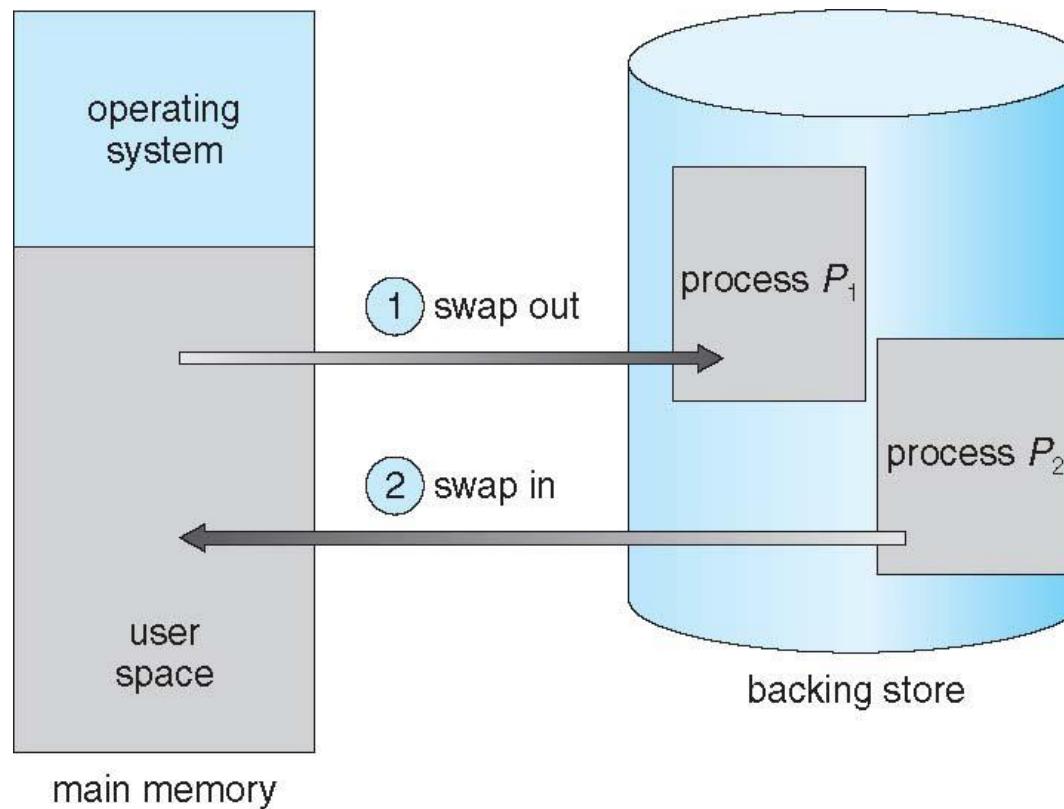
Swapping

- A process can be **swapped** temporarily out of memory to a backing store (disk), and then brought back into memory for continued execution
 - Total physical memory space of processes can exceed real physical memory
 - Swapping increases **degree of multiprogramming**
- **Dispatcher** is used to select a new process whenever CPU decides to execute a new process.
- If next processes is not in memory, dispatcher swap out a process in memory and swap in desired process.
- **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
- System maintains a **ready queue** of **ready-to-run** processes which have memory images on disk





Schematic View of Swapping





Context Switch Time including Swapping

- Major part of swap time is **transfer time**; total transfer time is directly proportional to the amount of memory swapped
- **Context switch time** can then be very high in swapping system
- Consider a 100MB process swapping to hard disk with transfer rate of 50MB/sec
 - Swap to or from main memory takes $100\text{ MB}/50\text{ MB/sec} = 2000\text{ ms (2 Sec)}$ of swap time
 - Plus swap in of same sized process
 - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if we reduce size of memory swapped – by knowing how much memory really **being used** rather than it **might be using**
 - System calls to inform OS of memory use can be used via `request_memory()` and `release_memory()`





Context Switch Time and Swapping (Cont.)

- Other constraints on swapping are
 - Process must be **completely idle** for swapping
 - Process with pending I/O can't swap – can't swap out as I/O would occur to wrong process
 - Execute I/O always to kernel (OS) space, then to I/O device
 - ▶ Then transfer between OS buffer and process memory occurs only when process is swapped in.
 - ▶ Known as **double buffering**, adds overhead
- Standard swapping not used in modern operating systems
 - But modified version common
 - ▶ Swap only when free memory extremely low





Swapping (Cont.)

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
 - Plus consider pending I/O to / from process memory space
- **Modified versions** of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
 - Swapping normally disabled
 - But started if more than threshold amount of memory allocated
 - Disabled again once memory demand reduced below threshold
 - Swapping only portion of process and not entire process to decrease swap time.





Swapping on Mobile Systems

- Not typically supported due to following constraints
 - **Flash memory** is used
 - ▶ Small amount of space
 - ▶ Limited number of write supported by flash memory
 - ▶ Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
 - iOS **asks** apps to voluntarily relinquish allocated memory
 - ▶ Read-only data thrown out and reloaded from flash if needed
 - ▶ Failure to free can result in termination
 - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
 - Both OSes support paging as discussed next





Contiguous Allocation

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





Contiguous Allocation (Cont.)

Memory Protection

- **Relocation registers** used to protect user processes from each other, and from changing operating-system code and data
 - **Base register** contains value of smallest physical address
 - **Limit register** contains range of logical addresses – each logical address must be less than the limit register
 - **MMU** maps logical address *dynamically*

Hardware Support for Relocation and Limit Register

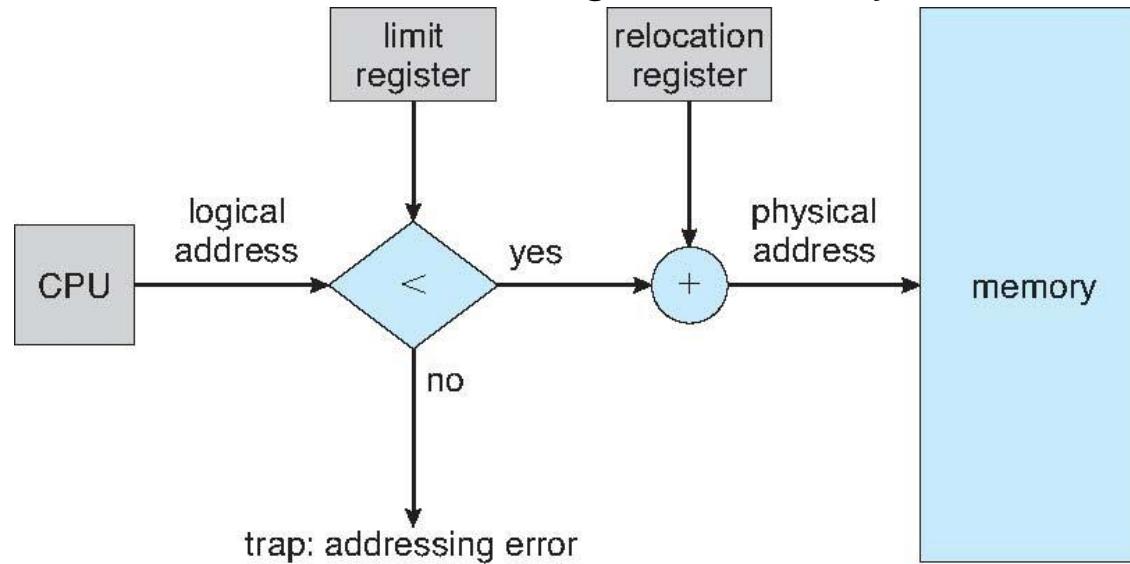




Contiguous Allocation (Cont.)

Memory Protection

- When the CPU scheduler selects a process for execution, the **dispatcher loads the relocation and limit registers** with the correct values
- Every address generated by a CPU is checked against these registers,
- Thus, protect both the operating system and the other users' programs and data from being modified by this running process





Memory allocation

■ Multiple-partition allocation

- One of the simplest method for allocating memory is to divide memory into several **fixed-sized partitions**.
- Each partition may contain exactly one process.
- Degree of **multiprogramming** limited by **number of partitions**
- When a partition is free, a process is selected from the input queue and is loaded into the free partition.
- When the process terminates, the partition becomes available for another process.
- This method is no longer used.

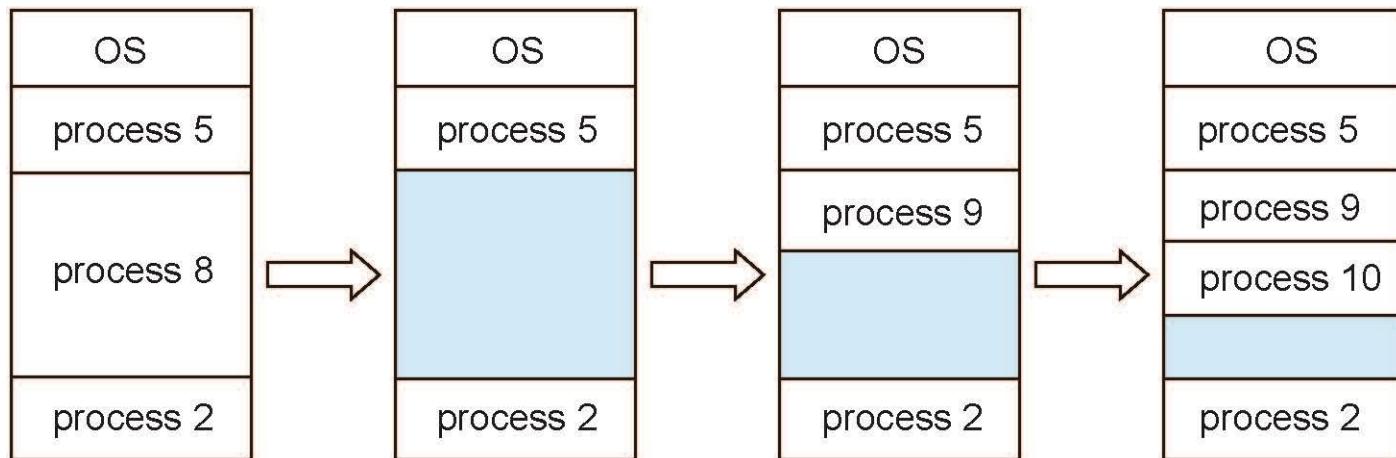


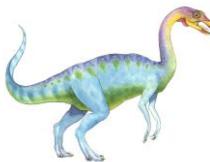


Memory allocation

■ Variable-partition allocation

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





Dynamic Storage-Allocation Problem

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

- **First-fit:** Allocate the ***first*** hole that is **big enough**. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- **Best-fit:** Allocate the ***smallest*** hole that is **big enough**; must search entire list, unless ordered by size
 - Produces the smallest leftover hole
- **Worst-fit:** Allocate the ***largest*** hole; must also search entire list
 - Produces the largest leftover hole

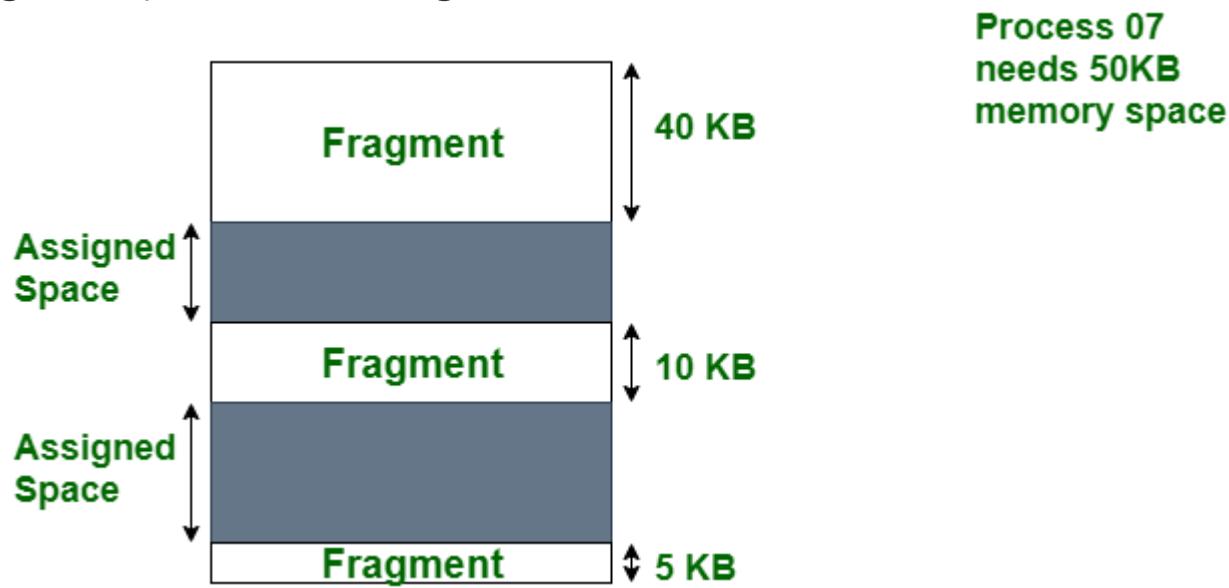
First-fit and best-fit better than worst-fit in terms of speed and storage utilization





Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous.
- Whether you apply a first-fit or best-fit memory allocation strategy it'll cause external fragmentation
- In the diagram, we can see that, there is enough space (55 KB) to run a process-07 (required 50 KB) but the memory (fragment) is not contiguous.





Fragmentation (Cont.)

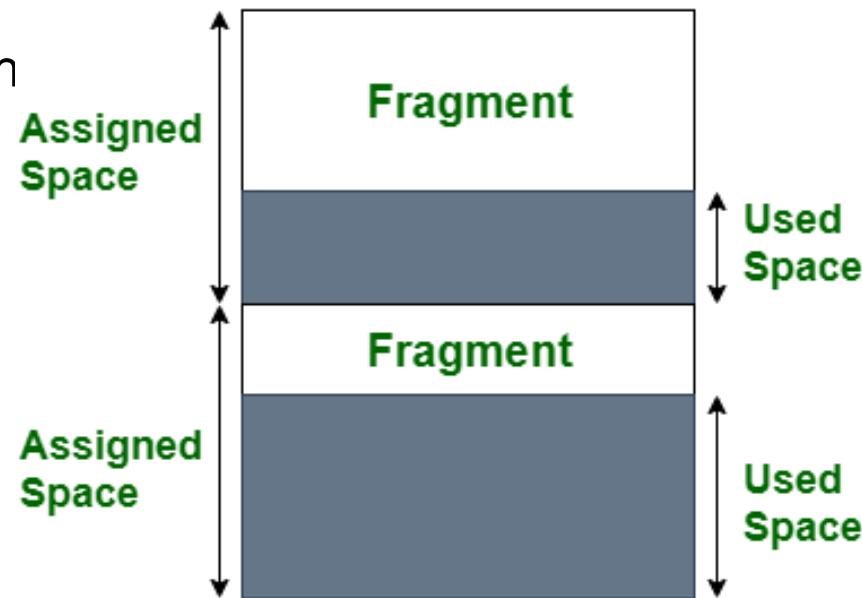
- Reduce external fragmentation by **compaction**
 - Shuffle memory contents to place all free memory together in one large block
 - Compaction is possible *only* if relocation is dynamic, and is done at execution time
- Another possible solution to the external-fragmentation problem is
- To permit the logical address space of the processes to be noncontiguous,
- thus allowing a process to be allocated physical memory wherever such memory is available.
- ✓ **Segmentation and Paging** is used





Fragmentation

- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory;
- Fragmentation because the difference between memory allocated and required space or memory is called **internal fragmentation**.
- The size difference is memory internal to a partition, but not being used
- Internal fragmentation happens when mounted-sized blocks.





Segmentation

- Is a memory-management scheme that supports **user view of memory**
- A **logical address space** is a collection of segments
- Each segment has **a name and a length**.
- The logical addresses specify both the **segment name** and the **offset** within the segment
- A program is a collection of segments : is a logical unit such as:

main program

procedure

function

method

object

local variables, global variables

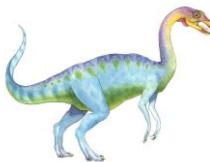
common block

stack

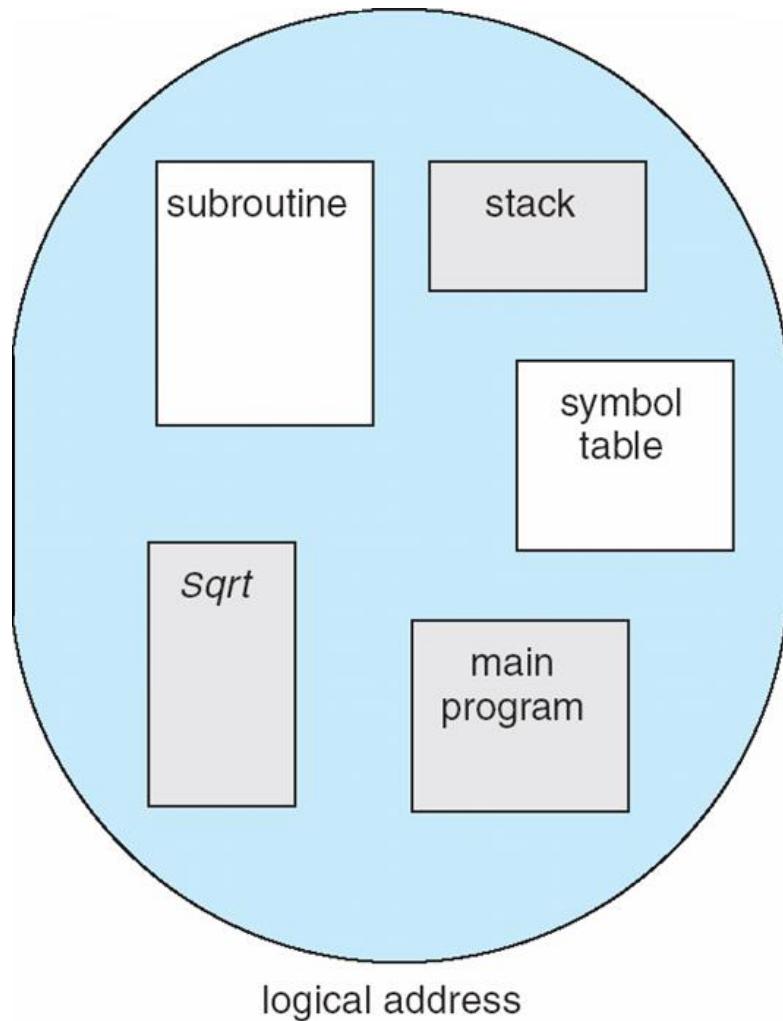
symbol table

arrays



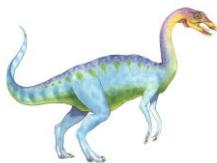


User's View of a Program

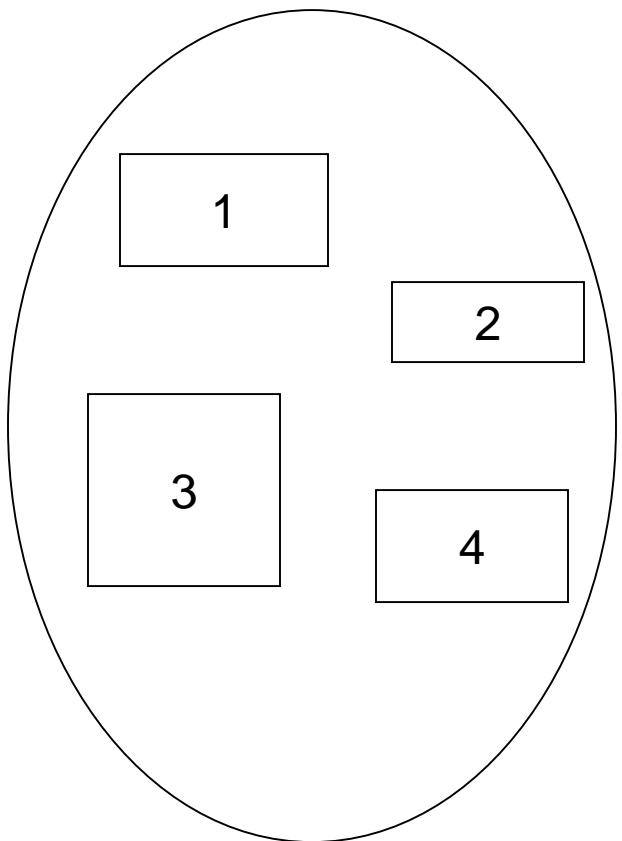


- Normally, when a program is compiled, the compiler automatically constructs segments reflecting the input program.
- Libraries that are linked in during compile time might be assigned separate segments.
- The loader would take all these segments and assign them segment numbers.

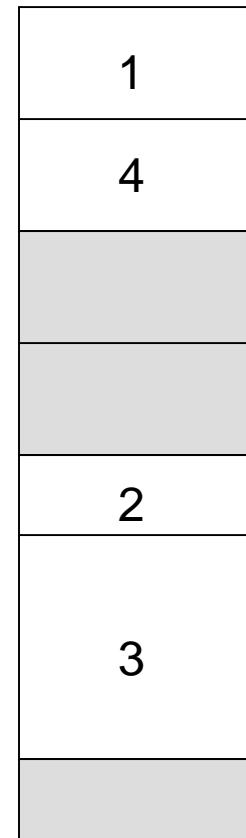




Logical View of Segmentation



user space



physical memory space





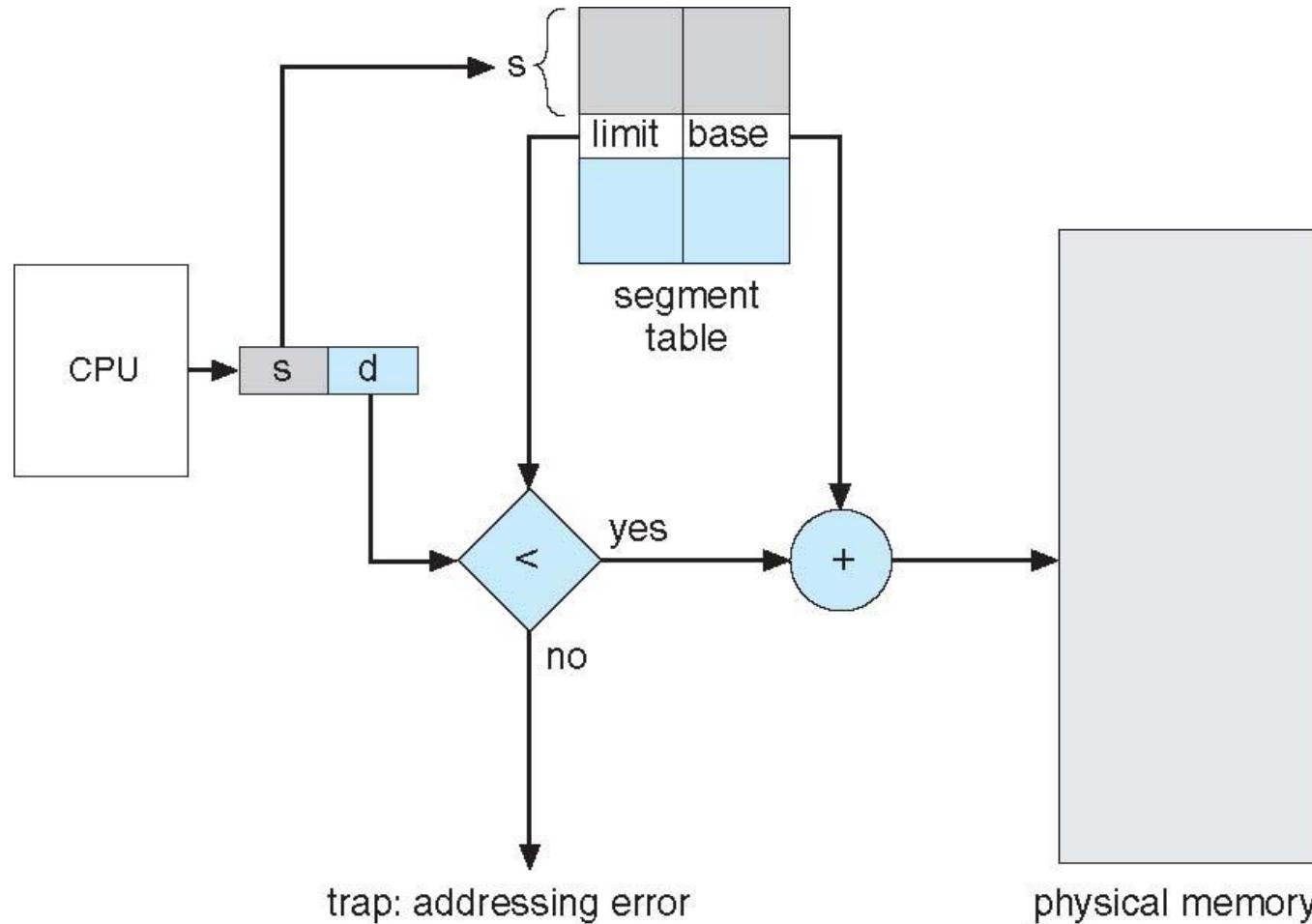
Segmentation Architecture

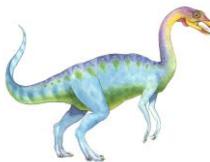
- Logical address consists of a **two tuple**:
 $\langle \text{segment-number}, \text{offset} \rangle,$
- **Segment table** – maps two-dimensional physical addresses to one-dimension physical address; each table entry has:
 - **base** – contains the starting physical address where the segments reside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



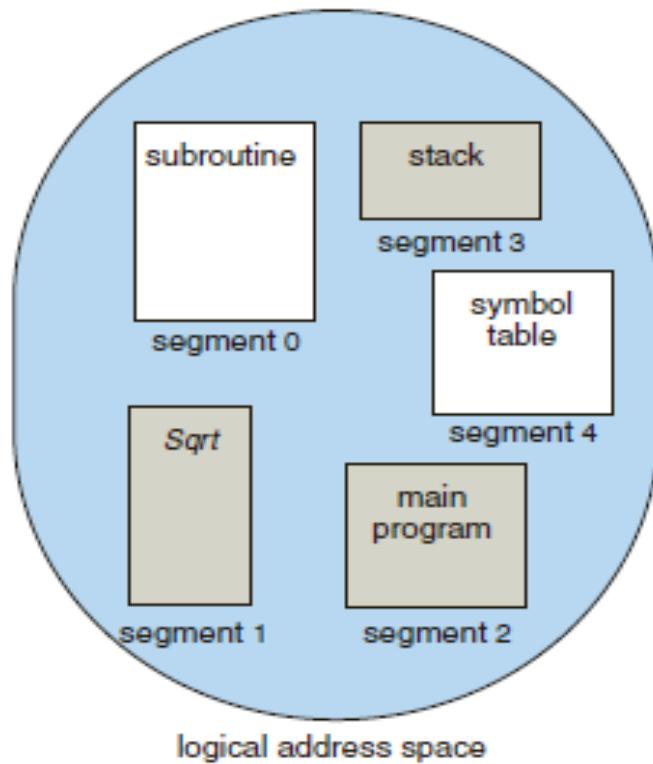


Segmentation Hardware



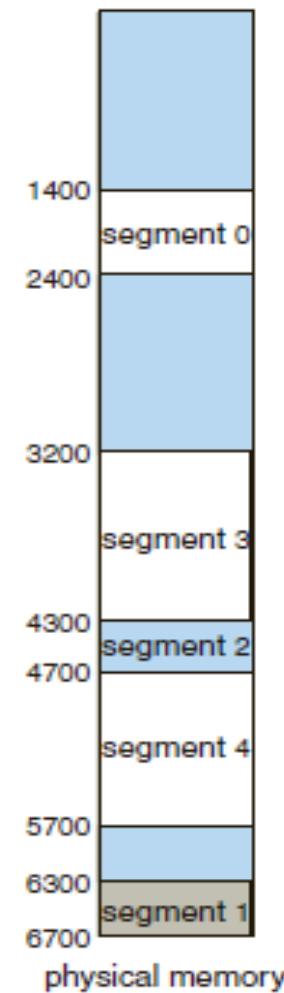


Segmentation Example



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

segment table





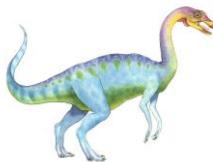
Paging

- Paging is another **memory management** scheme that eliminates the need for a contiguous allocation of physical memory.
- Avoids external fragmentation

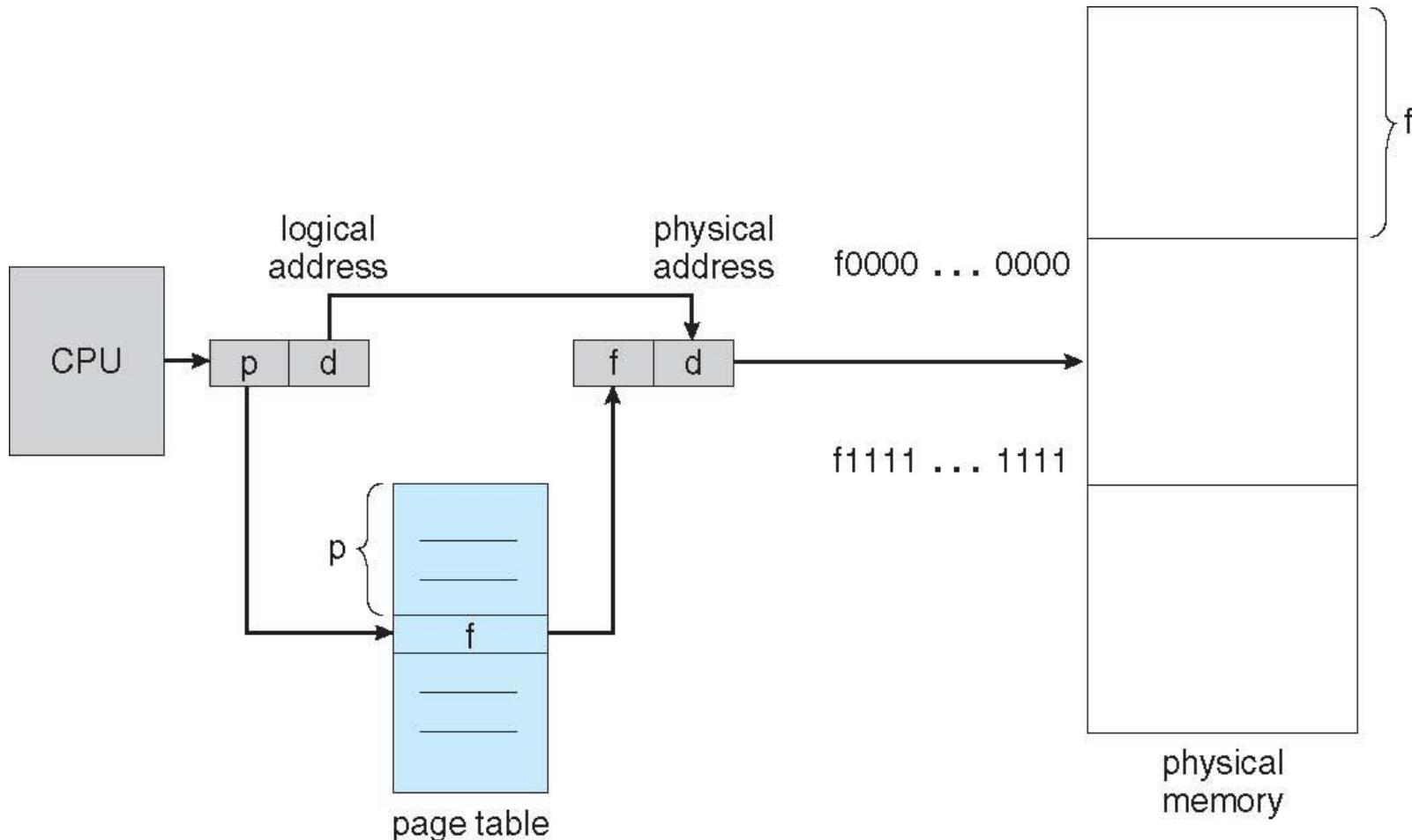
Basic Method

- Divide physical memory into **fixed-sized blocks** called **frames**
 - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**, **size is defined by hardware**
- Keep track of all free frames
- When a process is to be executed, its pages are loaded into any available memory frames
- The backing store is divided into **fixed-sized blocks** that are the same size as the **memory frames** or clusters of multiple frames





Paging Hardware





Address Translation Scheme

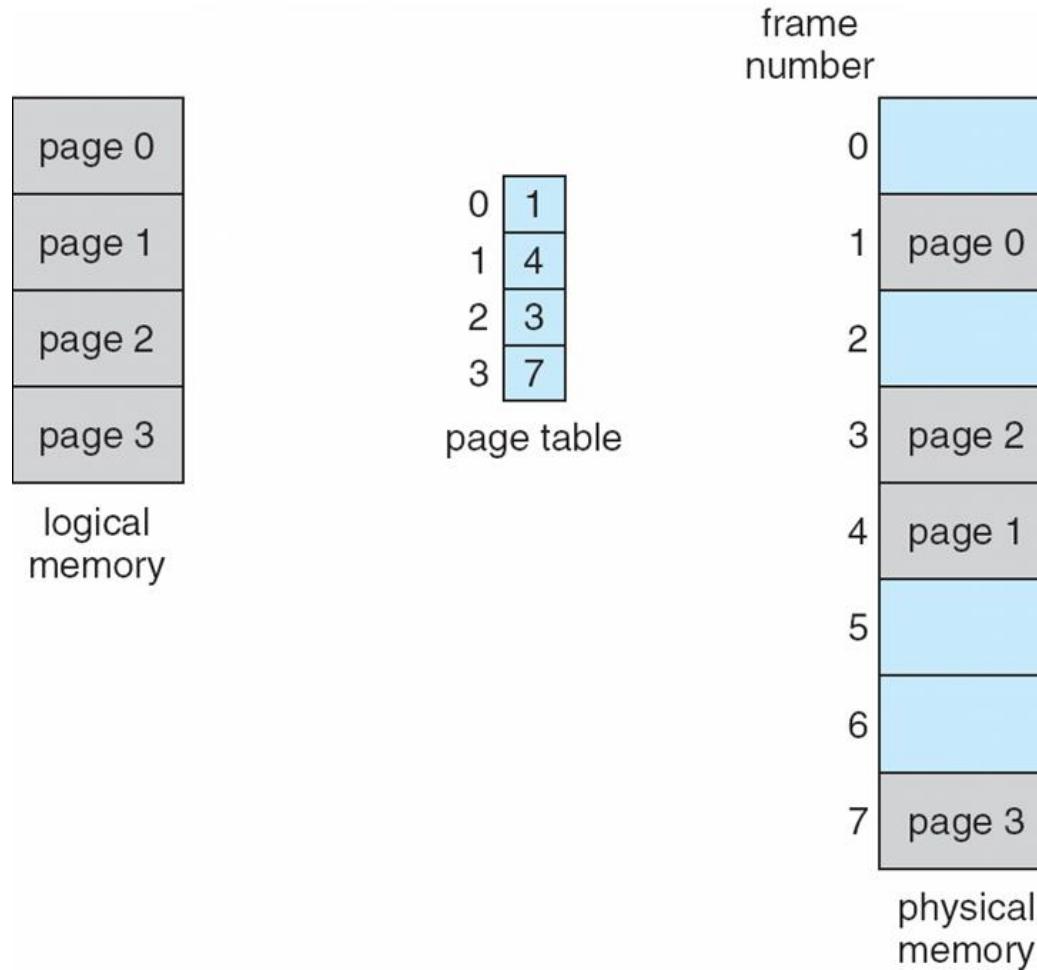
- Address generated by CPU is divided into 2 parts:
 - **Page number (p)** – used as an index into a **page table**. The **page table** contains base address of each page in physical memory
 - **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit
- For given logical address space 2^m and page size 2^n ,
- The high-order **$m-n$ bits** of a logical address designate the **page Number(p)**, and the **n low-order bits** designate the **page offset(d)**.
- Thus, the logical address is as follows:

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





Paging Model of Logical and Physical Memory





Paging Example

Consider the example with logical address, $n=2$ and $m=4$. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages),

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

logical memory

0	5
1	6
2	1
3	2

page table

0	
4	
8	
12	
16	
20	
24	
28	

physical memory



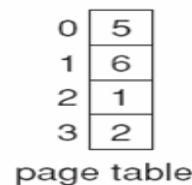


Paging Example

- Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5.
- Thus, logical address 0 maps to physical address 20 [= (5 × 4) +0]

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

logical memory



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

physical memory





Paging (Cont.)

- Paging may have some **internal fragmentation**. If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full, leading to internal fragmentation.
- Consider
 - Page size = 2,048 bytes, Process size = 72,766 bytes
 - The process needs 35 pages + 1,086 bytes
 - Internal fragmentation of $2,048 - 1,086 = 962$ bytes
- In the worst case, a process would need n pages plus 1 byte, resulting in internal fragmentation of almost an entire frame.
 - On average fragmentation = $1 / 2$ frame size
 - So small frame sizes desirable?
 - But overhead is involved in each page-table entry
 - Page sizes growing over time
 - ▶ Solaris supports two page sizes – 8 KB and 4 MB

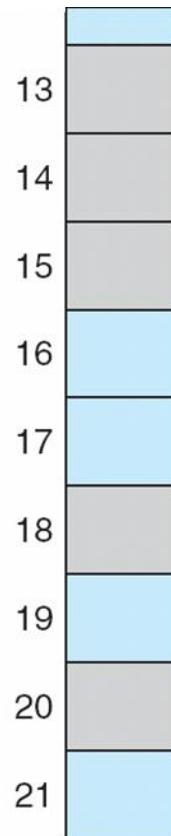
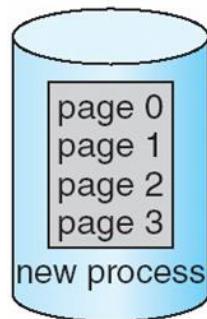




Free Frames

free-frame list

14
13
18
20
15

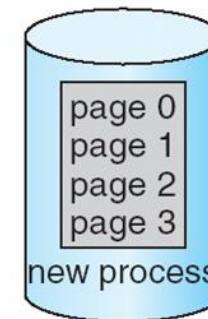


(a)

Before allocation

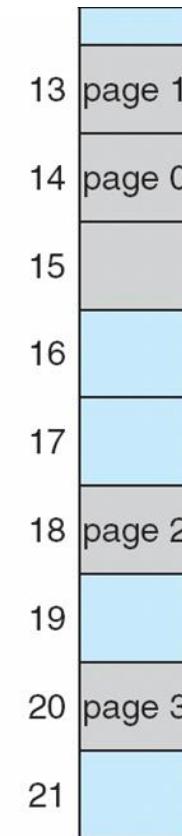
free-frame list

15



0	14
1	13
2	18
3	20

new-process page table



(b)

After allocation





Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PTLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
 - One for the page table and one for the data / instruction
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**





Implementation of Page Table (Cont.)

- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process
 - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
 - Replacement policies must be considered
 - Some entries can be **wired down** for permanent fast access





Associative Memory

- Associative memory – parallel search

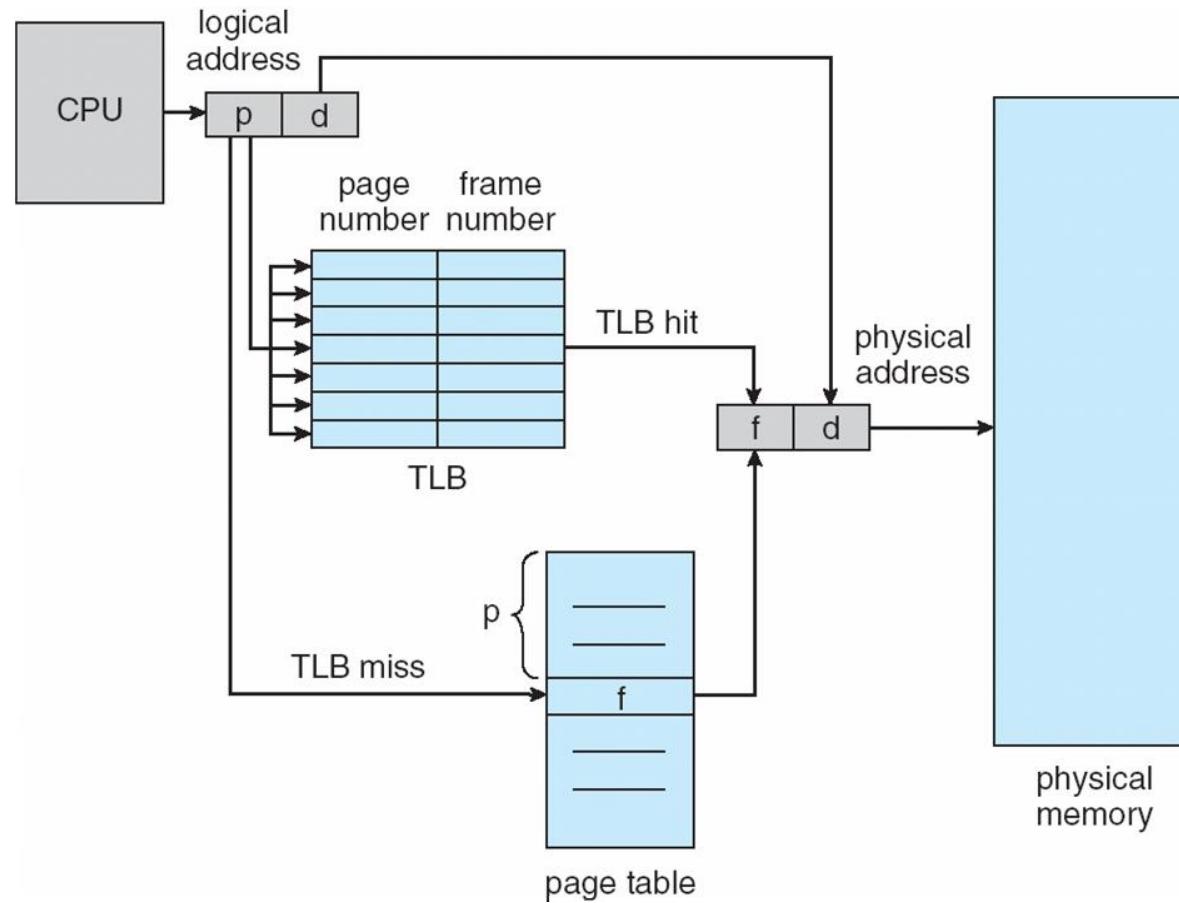
Page #	Frame #

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





Paging Hardware With TLB





Effective Access Time

- Associative Lookup = ε time unit
 - Can be < 10% of memory access time
- Hit ratio = α
 - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Consider $\alpha = 80\%$, $\varepsilon = 20\text{ns}$ for TLB search, 100ns for memory access
- **Effective Access Time (EAT)**
$$\text{EAT} = 0.80 \times 120 + 0.20 \times 220 = 140\text{ns}$$
- Consider more realistic hit ratio -> $\alpha = 99\%$, $\varepsilon = 20\text{ns}$ for TLB search, 100ns for memory access
 - $\text{EAT} = 0.99 \times 120 + 0.01 \times 220 = 121\text{ns}$

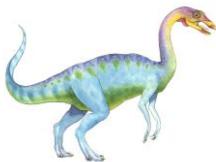




Memory Protection

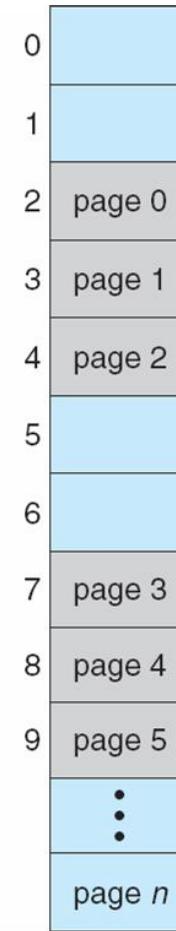
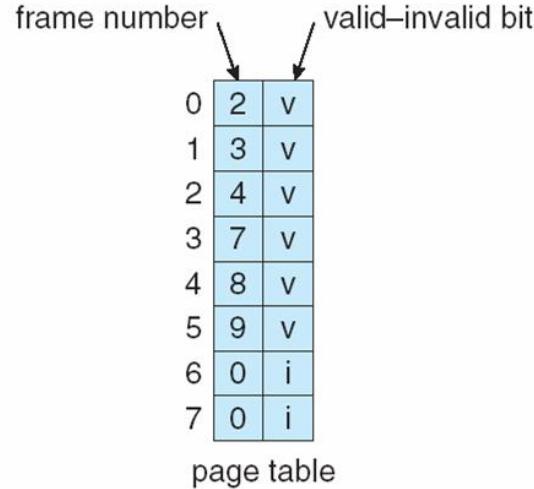
- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
 - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
 - “valid” indicates that the associated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





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

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





Shared Pages

Another advantage of paging is possibility of **sharing**

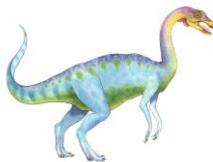
■ Shared code

- One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
- Similar to multiple threads sharing the same process space
- Also useful for interprocess communication if sharing of read-write pages is allowed

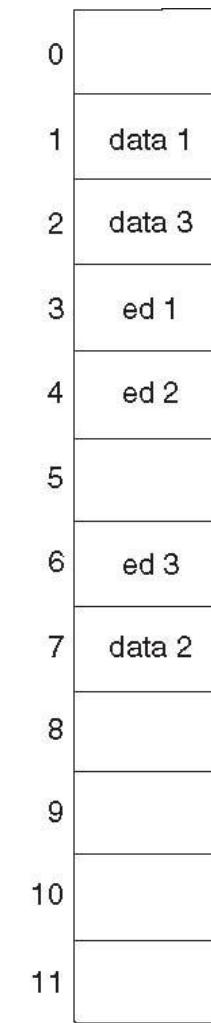
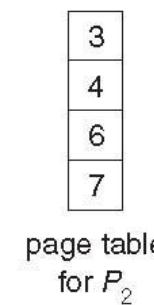
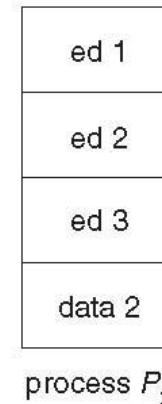
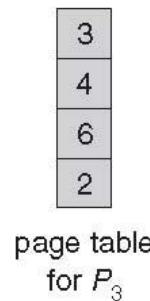
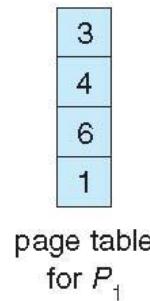
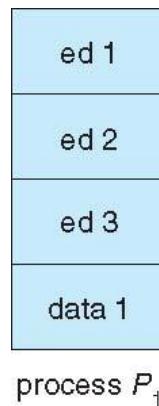
■ Private code and data

- Each process keeps a separate copy of the code and data
- The pages for the private code and data can appear anywhere in the logical address space





Shared Pages Example





Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
 - Consider a 32-bit logical address space as on modern computers
 - Page size of 4 KB (2^{12})
 - Page table would have 1 million entries ($2^{32} / 2^{12}$)
 - If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
 - ▶ That amount of memory used to cost a lot
 - ▶ Don't want to allocate that contiguously in main memory
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables





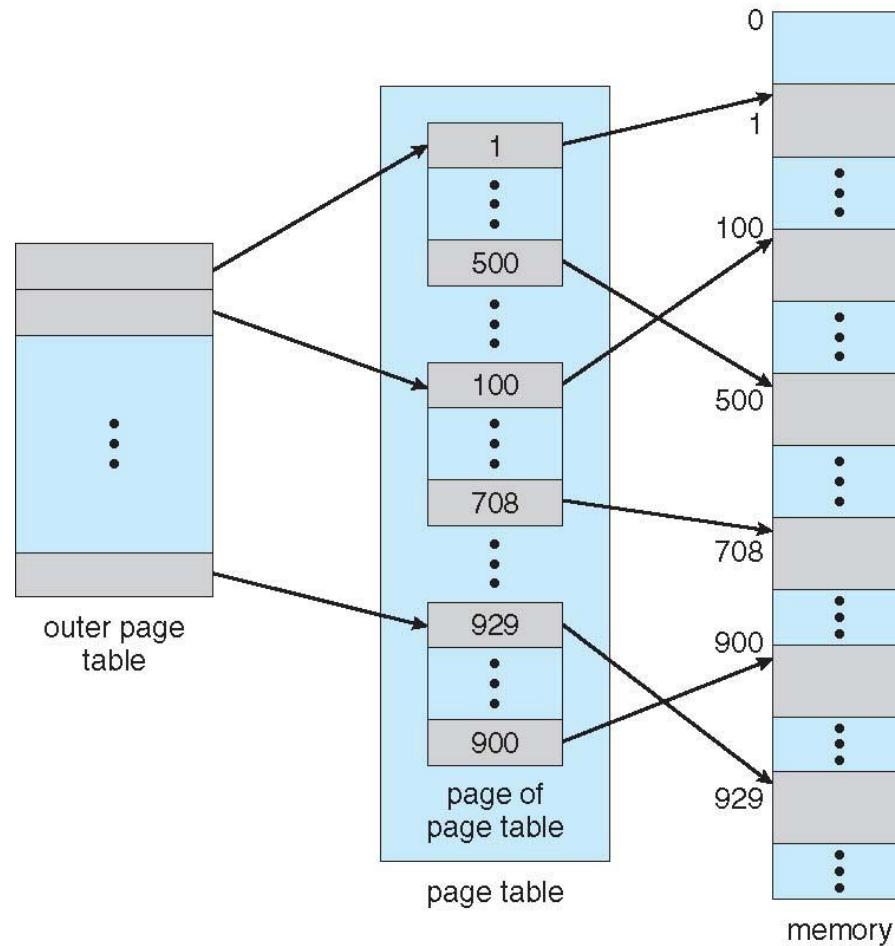
Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table





Two-Level Page-Table Scheme





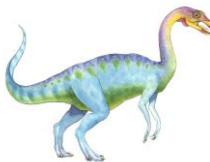
Two-Level Paging Example

- 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
- Thus, a logical address is as follows:

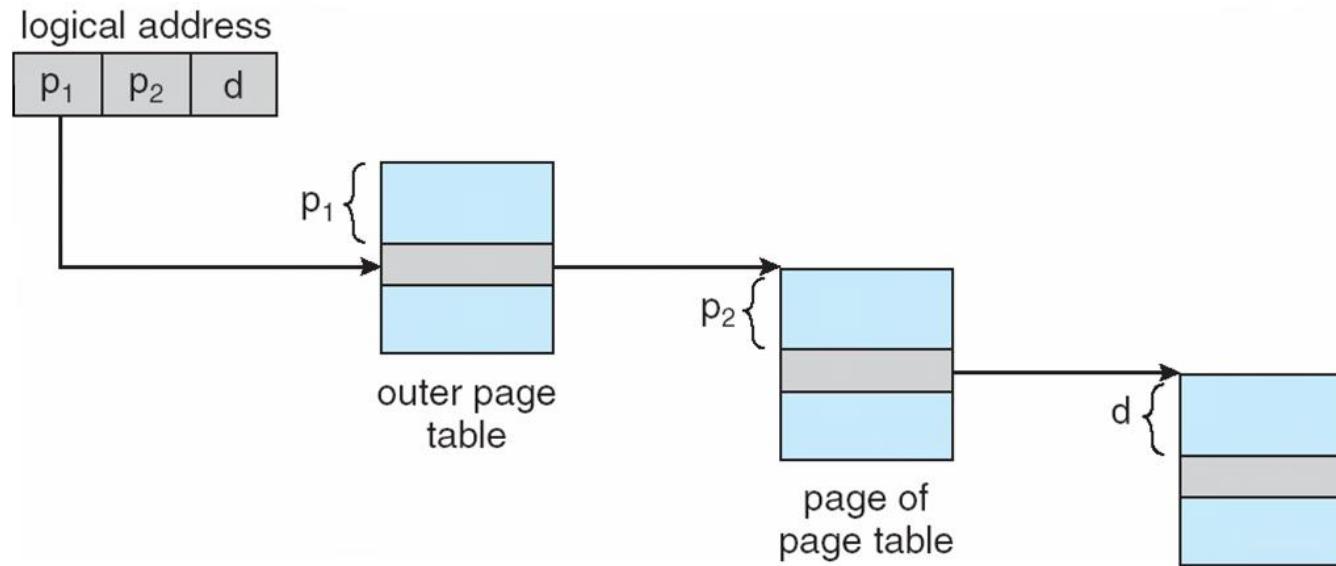
page number	page offset
p_1	p_2
12	10

- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**





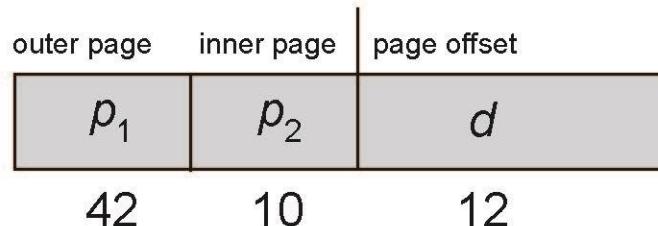
Address-Translation Scheme





64-bit Logical Address Space

- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
 - Then page table has 2^{52} entries
 - If two level scheme, inner page tables could be 2^{10} 4-byte entries
 - Address would look like

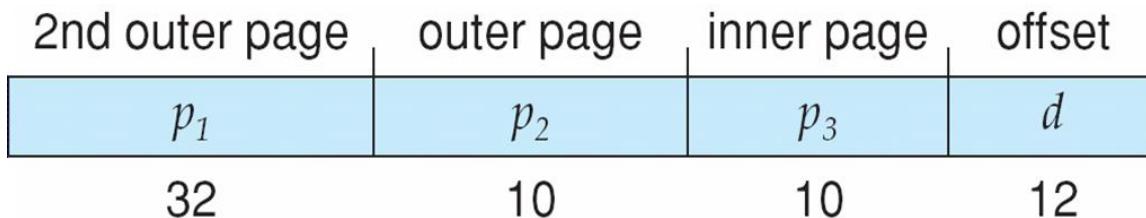
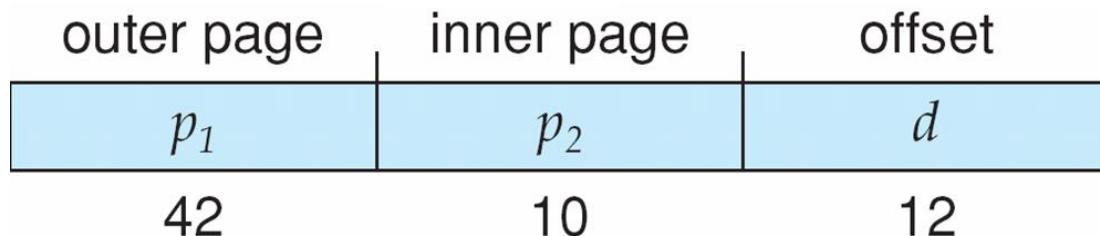


- Outer page table has 2^{42} entries
- One solution is to add a 2nd outer page table
- But in the following example the 2nd outer page table is still 2^{34} bytes in size
 - ▶ And possibly 4 memory access to get to one physical memory location





Three-level Paging Scheme

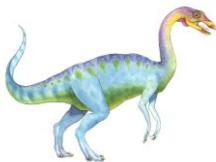




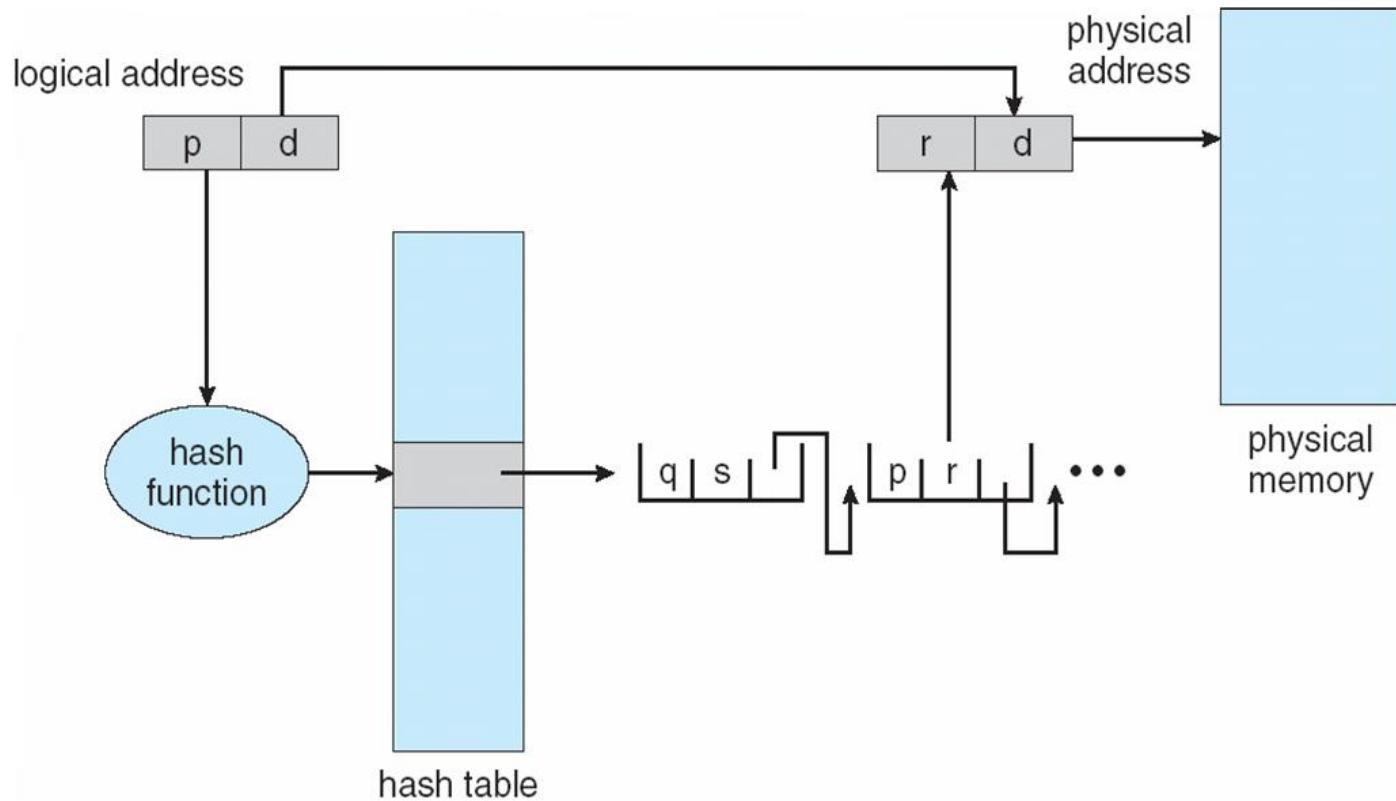
Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
 - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
 - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
 - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
 - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)





Hashed Page Table





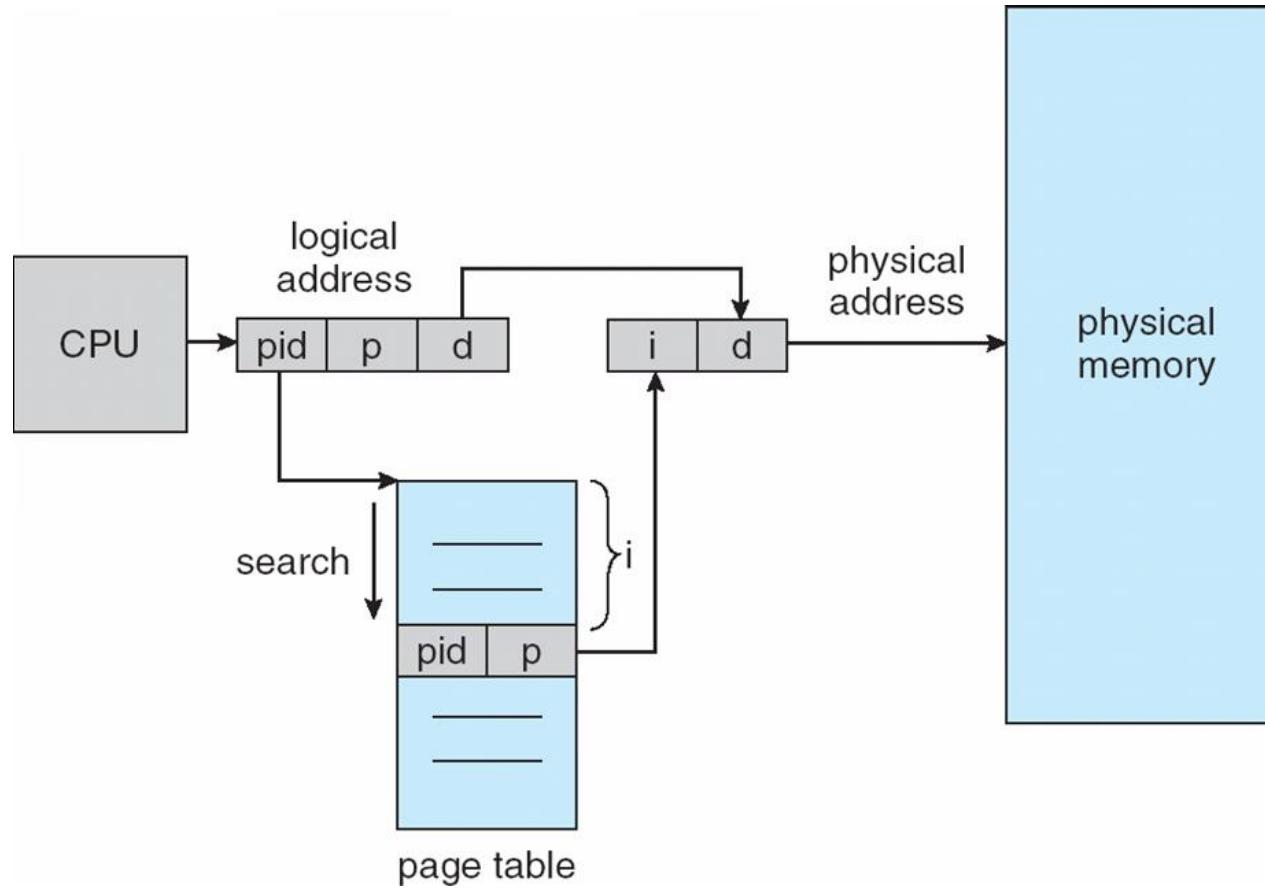
Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
 - TLB can accelerate access
- But how to implement shared memory?
 - One mapping of a virtual address to the shared physical address





Inverted Page Table Architecture



End of Chapter 8

